Appears in the proceedings of *Design, Automation and Test in Europe (DATE '02)*, March 2002, pp. 28–35.

Using Rewriting Rules and Positive Equality to Formally Verify Wide-Issue Out-Of-Order Microprocessors with a Reorder Buffer[1]

Miroslav N. Velev
mvelev@ece.cmu.edu
http://www.ece.cmu.edu/~mvelev
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract

Rewriting rules and Positive Equality [4] are combined in an automatic way in order to formally verify out-of-order processors that have a Reorder Buffer, and can issue/retire multiple instructions per clock cycle. Only register-register instructions are implemented, and can be executed out-of-order, as soon as their data operands can be either read from the Register File, or forwarded as results of instructions ahead in program order in the Reorder Buffer. The verification is based on the Burch and Dill correctness criterion [6]. Rewriting rules are used to prove the correct execution of instructions that are initially in the Reorder Buffer, and to remove them from the correctness formula. Positive Equality is then employed to prove the correct execution of newly fetched instructions. The rewriting rules resulted in up to 5 orders of magnitude speedup, compared to using Positive Equality alone. That made it possible to formally verify processors with up to 1,500 instructions in the Reorder Buffer, and issue/retire widths of up to 128 instructions per clock cycle.

1 Introduction

The property of Positive Equality [4] allowed the automatic verification of complex pipelined microprocessors with exceptions, multicycle functional units, and branch prediction [29][30]. By restricting the style for describing high-level processors [28], yet without losing expressive power, we can get a correctness formula where most of the word-level values appear only in positive (not negated) equalities. These word-level values can be treated as distinct constants, due to the structure of the formula, thus allowing us to dramatically prune the solution space. By encoding the remaining word-level equalities (appearing in both positive and negative polarity) with a new Boolean variable, an e_{ij} variable [8], we can translate the correctness formula into an equivalent Boolean formula. Recent advances in SAT-checkers [22][34] have allowed us to efficiently solve such formulas [32].

The formal verification is done by correspondence checking—comparison of an implementation processor against a non-pipelined specification (the Instruction Set Architecture), based on the Burch and Dill commutative diagram [6]. The correctness criterion is expressed as a formula in the logic of Equality with Uninterpreted Functions and Memories (EUFM) [6] (see Sect. 2) and states that all user-visible state elements in the implementation should be updated in sync by either 0, or 1, or up to k instructions after each clock cycle, where k is the issue width of the design. An abstraction function is used to map an implementation state to an equal specification state. Burch and Dill [6] used flushing—feeding the implementation with bubbles until all partially executed instructions complete—in order to compute an abstraction function. The commutative diagram has an implementation side—one step of the implementation, followed by application of the abstraction function; and a specification side—application of the abstraction function on the initial implementation state, followed by up to k steps of the specification. The two sides should produce equal user-visible states.

Positive Equality has not yet been exploited when formally verifying out-of-order processors with a Reorder Buffer. Jhala and McMillan [15] have conjectured that the reason for this is the complexity of the abstraction function that depends on the entire machine state.

Rewriting rules have been previously used for verification of simple single-issue pipelined processors by Levitt and Olukotun [19][20]. They had to define a large set of such rules—one for each pipeline stage with a given configuration of features. On the other hand, all instructions in an abstract out-of-order processor with a Reorder Buffer are executed by logic with identical structure, and can be processed by the same set of few rewriting rules.

Rewriting rules and Positive Equality have already been combined when formally verifying pipelined processors with in-order execution, exceptions, multicycle functional units, and branch prediction [31]. In that work, rewriting rules were used in order to automatically separate the effects of the forwarding and stalling logic and to abstract the forwarding logic that does not interact with stalling conditions. That dramatically reduced the number of equalities appearing in both positive and negated polarity, i.e., helped to more fully exploit Positive Equality, and resulted in an order of magnitude speedup.

All previous work on formal verification of out-of-order processors [1][2][3][11][13][14][15][16][17][18][21][23][24] has examined designs that can issue and retire only a single instruction per clock cycle. In contrast, the processors verified in this paper can issue and retire up to 128 instructions per clock cycle. Also, they have up to 1,500 instructions in the Reorder Buffer. Scaling to larger configurations was limited by the available physical memory of 4 GB. However, the above numbers are more than 10 times greater than those in current state-of-the-art processors. For example, the Intel® Pentium® 4 [12] and the AMD Athlon™ [7] have, respectively, 126 and 72 Reorder Buffer entries, and can issue/retire up to 3 instructions[2] per cycle; the Alpha 21364 [9] can have up to 80 instructions in flight, and can issue/retire up to 4 instructions per cycle.

The contribution of this paper is a method for correctness proof of an out-of-order processor that can issue and retire multiple instructions per cycle. The method has a higher degree of automation than previous techniques. Manually defined rewriting rules are applied mechanically to prove that each instruction initially in the processor will produce equal updates along both sides of the diagram, so that those equal updates can be removed from the formula. Then, Positive Equality is exploited automatically to prove that the instructions fetched during the one cycle of regular operation of the implementation are executed correctly. Positive Equality reduces the manual effort compared to previous work, where the entire proofs are constructed by the verification engineer. Furthermore, the presented method does not require the definition of an induction hypothesis—also done manually in previous research—as all entries in the Reorder Buffer are fully instantiated.

1. This research was supported by the SRC under contract 01-DC-684

2. Micro-ops, derived from decoding ×86 (IA-32) instructions

2 Background

The correctness EUFM formula is generated automatically, using a term-level symbolic simulator `TLSim` [33]. That formula is then translated to an equivalent Boolean formula by another automatic tool, `EVC` [33], which exploits the properties of Positive Equality [4], the e_{ij} encoding [8], conservative transformations, and rewriting rules. The resulting Boolean formula should be a tautology in order for the implementation processor to be correct. We can check this by negating the formula and using a SAT-checker [22][34] to prove that the negation is unsatisfiable.

The syntax of EUFM [6] includes terms and formulas. Terms are used in order to abstract word-level values—data operands, register identifiers, memory addresses, as well as the entire states of memory arrays. A term can be an Uninterpreted Function (UF) applied on a list of argument terms; a term variable; or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, *term*1, *term*2) will evaluate to *term*1 when *formula* = **true** and to *term*2 when *formula* = **false**. Formulas are used in order to model the control path of a microprocessor, as well as to express the correctness condition. A formula can be an Uninterpreted Predicate (UP) applied on a list of argument terms; a propositional variable; an *ITE* operator selecting between two argument formulas based on a controlling formula; or an equation (equality comparison) of two terms. Formulas can be negated and connected by Boolean connectives. We will refer to both terms and formulas as expressions.

UFs and UPs are used to abstract away the implementation details of functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*. Namely, that equal values of the inputs to the UF (UP) produce equal output values. Then, it no longer matters whether the original functional unit is an adder or a multiplier, etc., as long as the same UF (or UP) is used to replace it in both the implementation and the specification. Note that in this way we will prove a more general problem—that the processor is correct for any implementation of its functional units. However, that more general problem is easier to prove.

One scheme for imposing the property of functional consistency of UFs and UPs is by using nested *ITE*s. The first application of some UF, $f(a_1, b_1)$, is replaced by a new term variable c_1. A second application, $f(a_2, b_2)$, will be replaced by $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new term variable, and so on. UPs are eliminated similarly by using new Boolean variables, instead of new term variables.

The EUFM syntax for terms can be extended to model memories (including Register Files) by means of the special uninterpreted functions *read* and *write* [6][28][31]. Function *read* takes two terms, serving as memory state and address, respectively, and returns a term for the data at that address. Function *write* takes three terms—memory state, address, and data—and returns a term for the new memory state after the update. The two functions satisfy the forwarding property of the memory semantics—a *read* returns the data written by the last *write*, if their addresses are equal, or the data from the previous memory state otherwise. The initial state of a memory is abstracted with a term variable.

Positive Equality allows the identification of two types of terms in the structure of an EUFM formula—those which appear in only positive equations and are called *p-terms* (for positive terms), and those which appear in both positive and negative equations and are called *g-terms* (for general terms). A negative equation is one which appears under an odd number of negations or as part of the controlling formula for an *ITE* operator. The efficiency from exploiting Positive Equality is due to the observation that the truth of an EUFM formula under a maximally diverse interpretation of the p-terms implies the truth of the formula under any interpretation. A maximally diverse interpretation is one where the equality comparison of a term variable with itself evaluates to **true**, that of a p-term variable with a syntactically distinct term variable evaluates to **false**, and that of a g-term variable with a syntactically distinct g-term variable could be either **true** or **false** and is encoded with a new e_{ij} Boolean variable.

3 Microarchitecture to be Formally Verified

The out-of-order implementation processor to be formally verified is shown in Fig. 1. The design can execute only register-register instructions. Up to k of them are fetched in program order by the Fetch Engine on every clock cycle, where k is the issue width of the design. The newly fetched instructions are placed at the end of the Reorder Buffer (ROB), a FIFO structure that maintains the program order of the instructions. The actual number of fetched instructions is determined by the Scheduler, based on the available ROB entries and structural resources (decoding units, buses, etc.), the state of the processor, and the implemented scheduling algorithm. The Scheduler communicates that number by signals $fetch_i$, for $1 \le i \le k$. The Fetch Engine also computes the next value of the Program Counter (PC), based on the values of signals $fetch_i$. Every instruction has five fields: a *Valid* bit, indicating whether the instruction will update the Register File; an *Opcode*; a destination register *Dest*; and two source registers, *Src1* and *Src2*. The Fetch Engine gets the instructions from a read-only Instruction Memory.

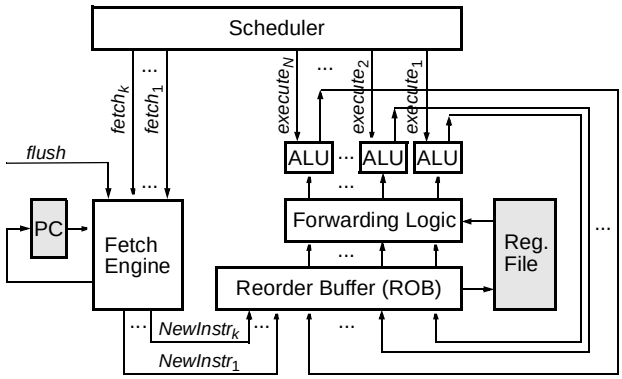

Figure 1: Block diagram of the implementation processor.

Every entry in the ROB has the same fields as an instruction, as well as a *ValidResult* bit, indicating whether the result of the instruction has been computed, in which case it is stored in field *Result*. Instructions are executed out of program order, as soon as their operands can be either read from the Register File or forwarded from the *Result* fields of preceding entries in the ROB, i.e., from instructions ahead in program order. A similar out-of-order design is described by Hennessy and Patterson [10] (pages 309–310). Another out-of-order processor where the ROB contains all information necessary for instruction execution is presented by Sohi [26].

Up to l instructions are retired in program order on every clock cycle, i.e., they are removed from the beginning of the ROB. An instruction among the first l in the ROB is retired if either its *Valid* bit is **false** (the instruction will not modify the Register File), or its *ValidResult* bit is **true** (its result has been computed) and all instructions ahead are retired in the current cycle. This condition is illustrated by formula (1) for the second instruction in an ROB. If the *Valid* bit of an instruction that is being retired is **true**, then its result is written, in program order, to the instruction's destination register in the Register File. In-order retirement will allow us to incorporate precise exceptions, as discussed by Smith and Pleszkun [25]: an exception will be processed only when its instruction enters the retire width, and no older instruction has raised an exception. That is how precise exceptions are implemented in the Intel® Pentium® 4 [12].

To simplify the presentation in the rest of the paper, we will assume that the issue width k and the retire width l are equal. However, the presented method does not depend on this.

The user-visible state consists of the PC and Register File. The specification processor executes one instruction per clock cycle by fetching the instruction from the same read-only Instruction Memory, incrementing the PC, computing the ALU result, and writing it to the destination register in the Register File under the condition that the instruction's *Valid* bit is **true**.

4 Abstracting the Out-Of-Order Core

The scheduling logic that controls how many instructions to fetch in program order (up to the issue width k) during the single step of regular operation of the implementation processor is abstracted with k new Boolean variables, $NDFetch_i$, $1 \le i \le k$. Signal $fetch_i$ that determines whether to fetch instruction i, for $1 \le i \le k$, is formed as the conjunction of variables $NDFetch_j$, where $1 \le j \le i$. Signals $fetch_i$ are non-deterministic (can be either **true** or **false**) and satisfy the property that if $fetch_i$ is **false**, then all $fetch_j$ after it ($i < j \le k$) are **false**. Hence, up to k instructions will be fetched in program order.

The ROB is abstracted with $N + k$ latches. The first N of them hold information about instructions that are initially in the ROB. The logic for preventing data hazards by stalling the execution of an instruction until its data operands can be either read from the Register File or forwarded from preceding entries in the ROB is fully implemented. So is the logic for retiring the first k instructions in program order. The additional k latches will accept any newly fetched instructions. The *Valid* bits for the newly fetched instructions will be formed as the conjunction of the original *Valid* signals coming from the Instruction Memory and the corresponding signal $fetch_i$. Hence, the above abstraction of an ROB will simulate all possible behaviors of adding/removing instructions in any actual implementation with size N and issue/retire width of k during one cycle of regular operation. This suffices for checking the safety property of an out-of-order processor with an ROB—if the processor does something in the single cycle of regular operation, it will do it correctly. The focus of this paper is on how to efficiently process the complex EUFM correctness formulas, generated in checking this safety property. The synthesis of a fully operational ROB will be addressed in future work.

The execution of the instructions in the ROB is abstracted with identical computation slices. During the single cycle of regular operation of the implementation processor, instruction i that is ready for execution is completed non-deterministically, as determined by a new Boolean variable, $NDExecute_i$, that is used to abstract signal $execute_i$ in Fig. 1, for $1 \le i \le N$. An instruction is ready for execution if its *Valid* bit is **true**, its *ValidResult* bit is **false** (the result is not computed yet), and the instruction's data operands can be either read from the Register File or forwarded from the *Result* fields of preceding entries in the ROB. If an instruction is completed, its ALU result is stored in field *Result* of the instruction's ROB entry, and the *ValidResult* bit in that entry is set to **true**. Note that instructions can execute out-of-order, as long as their data dependencies can be satisfied.

The above computation slices are similar to the way that Hosabettu, *et al.* [13][14] abstract the execution of instructions. Since the completion of every ready instruction is non-deterministic, the abstract out-of-order processor will simulate the behavior of any actual implementation that will be able to execute only a subset of the ready instructions, due to structural hazards, e.g., a limited number of functional units or data buses. Also, note that the non-deterministic completion of ready instructions abstracts the behavior of multicycle functional units, each of which could finish a computation during the single cycle of regular operation of the implementation processor.

When the abstraction function is applied to the state of the implementation processor, signal *flush* is set to **true** (it is **false** during regular operation), and the computation slices are activated one by one in program order, according to the ordering of the ROB entries. For an activated slice, if the *ValidResult* bit is **true**, the data in the *Result* field is written to the instruction's destination register, *Dest*, in the Register File. Otherwise, if the *ValidResult* bit is **false**, the data operands are read directly from the Register File, the result is computed instantaneously (all ALUs are abstracted with the same UF) and is written to register *Dest* in the Register File. In both cases, the *write*s to the Register File occur only if the instruction's *Valid* bit is **true**.

The above way to complete the execution of a partially-executed instruction is called *completion function* [13][14]. The logic computing an instruction's completion function is included in the abstract computation slice for that instruction and is enabled only when signal *flush* is **true**. This extra logic will be optimized away when an actual implementation is synthesized from the abstract out-of-order processor. Note that all computation slices have similar structure—they differ only in the logic for preventing data hazards, as that logic has to consider all instructions ahead in program order in the ROB.

5 Structure of the Correctness Formulas

The structure of the correctness formulas will be illustrated for an implementation processor with 3 ROB entries, and issue/retire width of 2. Let *PC* and *RegFile* be term variables that abstract the initial state of the PC and Register File, respectively. In this design, the PC is not changed when the abstraction function is applied to the initial state of the implementation processor; the supported register-register instructions modify the PC during fetching only, but not when already in the ROB. Thus, the initial state of the PC to be used by the specification processor is:

$$PC_{Spec,\,0} \leftarrow PC$$

After one step of the specification, the state of the PC will be:

$$PC_{Spec,\,1} \leftarrow \text{NextPC}(PC)$$

where NextPC() is an UF abstracting the incrementer of the PC. And, after another step, the state of the PC will be:

$$PC_{Spec,\,2} \leftarrow \text{NextPC}(PC_{Spec,\,1})$$

On the implementation side of the diagram, the new state of the PC will depend on signals $fetch_1$ and $fetch_2$:

$$\begin{aligned} PC_{Impl} \leftarrow\ & ITE(fetch_2, \text{NextPC}(\text{NextPC}(PC)),\\ & ITE(fetch_1, \text{NextPC}(PC), PC)) \end{aligned}$$

where

$$\begin{aligned} fetch_1 &\leftarrow NDFetch_1\\ fetch_2 &\leftarrow NDFetch_1 \wedge NDFetch_2 \end{aligned}$$

such that $NDFetch_1$ and $NDFetch_2$ are new Boolean variables, as discussed in Sect. 4. A new Boolean variable is modeled as the output of an UP with no arguments.

An *update* is a conditional *write* operation of the form *ITE*(*context*, *write*(*prevMemState*, *addr*, *data*), *prevMemState*), where *context* is a formula for the condition under which the *write* occurs, *prevMemState* is an expression for the previous memory state before the *write*, and *addr* and *data* are expressions for the address and data of the *write*. The structure of the expressions for the Register File state after both sides of the commutative diagram is shown in Fig. 2.a. There, the arrows point to the previous memory state. Some arrows have a label—the name of the expression after the previous update. The updates are represented as triples of the form ⟨*context*, *address*, *data*⟩. The same notation for updates was previously used when abstracting memory arrays at the bit level [27].

Expressions $SpecData_1$, $SpecData_2$, and $SpecData_3$ in Fig. 2.a are computed by the abstraction function along the spec-

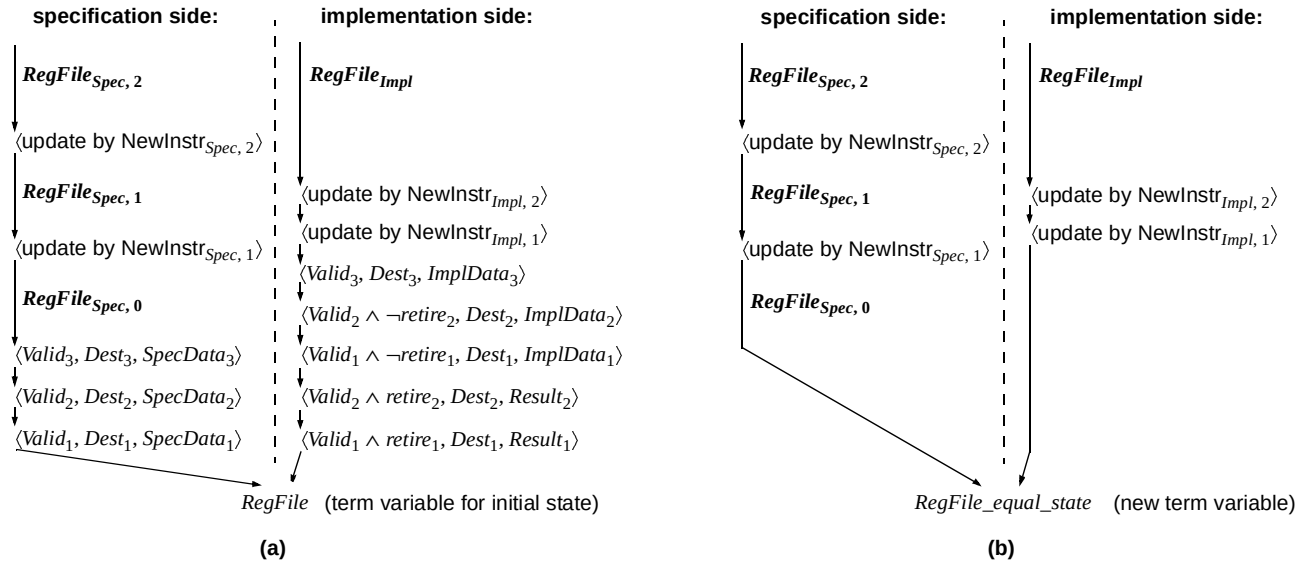

Figure 2: Structure of the expressions for the Register File state in an out-of-order processor with 3 Reorder Buffer entries and issue/retire width of 2: a) before, and b) after rewriting rules are automatically applied to prove that the instructions initially in the Reorder Buffer will produce equal updates along both sides of the diagram, allowing the removal of the equal update sequences. The updates are represented as triples of the form ⟨*context, address, data*⟩. The arrows point to the previous Register File state.

ification side of the diagram, based on the initial implementation state. Expressions $ImplData_1$, $ImplData_2$, and $ImplData_3$ are computed by the abstraction function along the implementation side of the diagram, based on the implementation state after one step of regular operation of the implementation processor.

Signals $retire_1$ and $retire_2$ (see Fig. 2.a) are the conditions to retire, in program order, the first and second instruction, respectively, in the retire width. These signals are defined in the implementation processor as:

$$\begin{aligned} retire_1 &\leftarrow \neg Valid_1 \vee ValidResult_1 \\ retire_2 &\leftarrow \neg Valid_2 \vee ValidResult_2 \wedge retire_1 \end{aligned} \quad (1)$$

The condition that the PC be updated by 0 instructions during the single step of the implementation processor is:

$$equal_{PC,\,0} \leftarrow (PC_{Impl} = PC_{Spec,\,0})$$

Similarly, the condition that the PC be updated by 1 instruction:

$$equal_{PC,\,1} \leftarrow (PC_{Impl} = PC_{Spec,\,1})$$

and, by 2 instructions:

$$equal_{PC,\,2} \leftarrow (PC_{Impl} = PC_{Spec,\,2})$$

We can form the same conditions for the Register File:

$$\begin{aligned} equal_{RegFile,\,0} &\leftarrow (RegFile_{Impl} = RegFile_{Spec,\,0}) \\ equal_{RegFile,\,1} &\leftarrow (RegFile_{Impl} = RegFile_{Spec,\,1}) \\ equal_{RegFile,\,2} &\leftarrow (RegFile_{Impl} = RegFile_{Spec,\,2}) \end{aligned}$$

Then, the EUFM correctness formula is the condition that the PC and Register File (the user-visible state elements) be updated in sync by 0, or 1, or 2 (the issue width) instructions:

$$\begin{aligned} correctness \leftarrow\ & equal_{PC,\,0} \wedge equal_{RegFile,\,0} \\ & \vee\ equal_{PC,\,1} \wedge equal_{RegFile,\,1} \\ & \vee\ equal_{PC,\,2} \wedge equal_{RegFile,\,2} \end{aligned}$$

6 Rewriting Rules Used

The goal of applying rewriting rules is to prove that the instructions initially in the ROB will produce equal sequences of updates along both sides of the commutative diagram. Then, these equal sequences of updates can be replaced with the same new term variable in the EUFM correctness formula, which will now depend only on the updates done by the newly fetched instructions and can be evaluated by exploiting Positive Equality.

First, we notice that the instructions within the retire width of the ROB (i.e., the first *k* instructions initially there) appear twice in the sequence of updates along the implementation side of the correctness diagram. Once, when they are retired during the single cycle of regular operation of the implementation—that is done under the condition that their result is ready and the instructions ahead are retired—and a second time when they are completed by the abstraction function. Also, the updates by instructions initially in the ROB have as address a term variable that represents the initial value of the instruction's destination register. (These term variables are introduced automatically by the term-level symbolic simulator `TLSim` [33] that generates the EUFM correctness formula.) Hence, two updates are done by the same instruction if their addresses are the same term variable.

We start by rearranging the sequence of updates along the implementation side of the diagram in order to bring the second update of the same symbolic (term variable) address down to the first update of that address. An update can be moved before another one if their contexts (conditions under which the updates are made) cannot be **true** simultaneously. For example, the second update to address $Dest_1$ in Fig. 2.a can be moved before the first update to address $Dest_2$, because their contexts, $Valid_1 \wedge \neg retire_1$ and $Valid_2 \wedge retire_2$, respectively, do not overlap. Indeed, from (1), it follows that the context of the latter update is:

$$\begin{aligned} & Valid_2 \wedge retire_2 = \\ & Valid_2 \wedge (\neg Valid_2 \vee ValidResult_2 \wedge retire_1) = \\ & Valid_2 \wedge ValidResult_2 \wedge retire_1 \end{aligned}$$

which cannot be **true** when the context of the former update is **true**, since both expressions are conjunctions that have input $retire_1$ in opposite polarities. Note that this form of the two contexts is a consequence of the in-order retirement.

We can similarly rearrange the updates when the processor can retire k instructions per clock cycle, for $k > 2$. Then, the context of the first update by instruction i, for $1 \le i \le k$, can be rewritten to the form:

$$Valid_i \wedge ValidResult_i \wedge retire_1 \wedge ... \wedge retire_{i-1}$$

while the second update by some instruction j that is ahead of i in program order (i.e., $1 \le j < i$) will be $Valid_j \wedge \neg retire_j$, so that the two contexts are conjunctions having input $retire_j$ in opposite polarities. This form of the contexts can be checked automatically, by examining the structure of their expressions. Hence, we can move the second update next to the first one for the same destination address of an instruction in the retire width.

Next, we notice that the contexts of the two updates to address $Dest_i$ of instruction i in the retire width ($1 \le i \le k$) are $Valid_i \wedge retire_i$ and $Valid_i \wedge \neg retire_i$, respectively. Therefore, $Dest_i$ will be updated if $Valid_i \wedge retire_i \vee Valid_i \wedge \neg retire_i = Valid_i$ is **true**, which is exactly the context for the update to $Dest_i$ along the specification side of the diagram. The above form of the contexts can be checked mechanically.

For instructions i that are initially in the ROB, but not within the retire width ($k < i \le N$)—they have only one update to their destination register along the implementation side of the diagram—the updates along both sides of the diagram are done under context $Valid_i$. We can automatically check that their contexts are the same Boolean variable.

Finally, we have to prove that the data values written to the same symbolic (term variable) address along both sides of the diagram are equal. This is done by case-splitting on the *ValidResult* bit for each instruction. The initial state of that bit is a Boolean variable, $ValidResult_i$, that can be identified automatically, by analyzing the structure of the data expression written to the instruction's destination address along the specification side of the diagram. There, $ValidResult_i$ controls an *ITE* expression and, if **true**, selects a term variable, $Result_i$, that represents the initial state of the *Result* field for that instruction. Else, when $ValidResult_i$ is **false**, the *ITE* will select an UF application that abstracts the ALU and computes the result, using data read from the previous Register File state. Hence, the two cases are:

1.) If $ValidResult_i$ is **true**, then the data values written along both sides of the diagram should evaluate to $Result_i$. Checking this can be done by automatically inspecting the structure of the *ITE* expressions for the data value written along the implementation side of the diagram.

2.) If $ValidResult_i$ is **false**, then the data value along the specification side evaluates to an UF application (abstracting the ALU) whose data operands are read from the previous Register File state. We need to prove the same for the data values along the implementation side. Two subcases need to be considered:

2.1) The computation is executed during the single cycle of regular operation of the implementation processor. The condition for that can be detected as a formula controlling an *ITE* expression that is part of the data expression for the update along the implementation side. That formula is the conjunction of $Valid_i$, $\neg ValidResult_i$ (these two Boolean variables can be identified from the corresponding update along the specification side of the diagram), and an expression for the condition that the data dependencies of the instruction can be satisfied from the initial state of the implementation processor—by being either forwarded from the *Result* fields of preceding instructions in the ROB, or read from the Register File. We will call that expression *dependencies_OK*, and can use it to prove that, when it is **true**, the *read* operations done from the specification side for the corresponding update there will evaluate to the same value as along the implementation side. Namely, that they will both evaluate to the same term variable that abstracts the initial state of some *Result* field in the ROB, or to the same *read* operation from the initial Register File state.

2.2) The computation is executed when applying the abstraction function. Then, along both sides of the diagram, the data operands are read from the Register File state after the previous update. However, note that if the contexts of two adjacent updates do not overlap, i.e., we can swap the two without affecting the final Register File state, then we can also move the *read*s, done from the (original) previous Register File state for each of the updates, to be performed from the new previous Register File state for the corresponding update after the swap. We can do that because the data values produced by those *read*s will be used in the expression for the update only when its context is **true**. Since that context does not overlap with the context of the other update in the swap, then the other update will not affect the *read*s.

In the case of an instruction within the retire width, we can do a further transformation after the second update along the implementation side is moved next to the first update. Since the contexts of both updates do not overlap—one is $Valid_i \wedge retire_i$, while the other is $Valid_i \wedge \neg retire_i$, for $1 \le i \le k$—we can move the second update's *read*s to be performed from the Register File state before the first update. For the instruction that is first in program order in the ROB, i.e., has $Dest_1$ as destination register, the state before the first update is the initial Register File state—a term variable. After the updates to $Dest_1$ are proved equal along both sides of the diagram, those updates are removed from the expressions for the Register File, since the states resulting after the updates will be equal. We can replace the resulting equal Register File states with a new (temporary) term variable.

The updates to $Dest_2$, the destination of the second instruction in program order in the ROB, are then processed similarly, and so on, until the list of updates in state $RegFile_{Spec,\,0}$ (Fig. 2.a) becomes empty. Then, state $RegFile_{Spec,\,0}$ is replaced by a new term variable, *RegFile_equal_state*, also used as the initial Register File state in the simplified expression along the implementation side of the diagram—see Fig. 2.b.

The resulting simpler expressions for the Register File state, not containing updates by instructions initially in the ROB, are processed as part of the correctness formula (Sect. 5) by exploiting Positive Equality.

7 Results

The experiments were performed on a 336-MHz Sun4 with 4 GB of physical memory. Both user-visible state elements—PC and Register File—were considered when generating the EUFM correctness formula. The abstraction function was computed by completion functions [13][14]—see Sect. 4. There was no need to impose constraints for the initial state of the implementation processor, since its logic for resolving data hazards is fully instantiated and does not depend on the previous behavior of the design. Data operands are either read from the Register File, or forwarded from the initial state of the ROB.

Table 1 shows the CPU times required by the term-level symbolic simulator `TLSim` [33] to produce the EUFM correctness formulas for implementations with different ROB sizes and issue/retire widths. The same specification was used in all cases. The implementation processors were generated by a C program, taking as parameters the size of the ROB and the issue width. It was assumed that the issue and retire widths are equal (the presented method does not depend on this) and do not exceed the ROB size. The descriptions of processors with 1,500 ROB entries required more than 900 MB of disk space each. In Tables 1–4, a dash means that the configuration is impossible, since the issue/retire width cannot exceed the ROB size.

In order to efficiently simulate implementation processors with many ROB entries, it was necessary to optimize `TLSim`. Its event-driven symbolic simulation engine was modified to sym-

bolically evaluate only the cone of influence of latches or memories whose state is updated in the current time step. This dramatically reduced the number of events that have to be processed, since only one computation slice of the processor is active at a time when applying the abstraction function.

Reorder Buffer Size	Issue/Retire Width							
	1	2	4	8	16	32	64	128
1	0.02	—	—	—	—	—	—	—
2	0.04	0.04	—	—	—	—	—	—
4	0.05	0.05	0.05	—	—	—	—	—
8	0.09	0.09	0.10	0.11	—	—	—	—
16	0.20	0.20	0.20	0.24	0.28	—	—	—
32	0.62	0.66	0.66	0.68	0.73	0.85	—	—
64	2.39	2.41	2.42	2.49	2.50	2.63	3.11	—
128	11	11	11	11	11	11	12	14
256	62	62	62	62	62	62	63	64
512	361	364	366	367	369	369	369	375
1,024	2,391	2,392	2,394	2,399	2,413	2,421	2,426	2,446
1,250	4,118	4,122	4,127	4,129	4,132	4139	4,147	4,163
1,500	6,821	6,829	6,837	6,845	6,862	6,867	6,872	6,897

Table 1: CPU time in seconds for symbolically simulating the out-of-order implementation and the specification, when generating the EUFM correctness formula.

7.1 Using Only Positive Equality

The EUFM correctness formulas were first translated to equivalent Boolean formulas by exploiting only Positive Equality, but no rewriting rules. The translation was done with the tool EVC [33]. Designs with less than 16 ROB entries required up to 3 seconds for the translations; designs with 16 ROB entries required up to 21 seconds. Then, the SAT-checker Chaff [22][34] was used to evaluate the Boolean formulas for being unsatisfiable—the case for correct designs—and the scaling is shown in Table 2. Statistics for the CNF formulas for correctness of processors with 8 ROB entries are presented in Table 3. Primary inputs are the variables in the Boolean correctness formula before it is translated to CNF format—see [32] for details about that translation. The e_{ij} variables encode equality comparisons of register identifiers. The category "Other Primary Inputs" of the correctness formula includes variables encoding control bits in the initial state of the ROB—signals *Valid* and *ValidResult* for each entry—as well as the *Valid* bits of newly fetched instructions. Additionally, that category includes Boolean variables abstracting the control signals that determine whether to complete the execution of each instruction whose data operands are ready, and whether to fetch each new instruction, up to the issue width. Row "CPU Time" in Table 3 is the same as the row for Reorder Buffer Size 8 in Table 2.

As Table 2 shows, doubling the Reorder Buffer from 4 to 8 entries increases the CPU time for SAT-checking the CNF formulas by 3 orders of magnitude. If the Reorder Buffer is further doubled to 16 entries, the verification runs out of memory (i.e., requires more than 4 GB) after more than 18,000 seconds, for all issue/retire widths. Hence, the method does not scale for models with 16 or more ROB entries, if only Positive Equality is used.

Reorder Buffer Size	Issue/Retire Width							
	1	2	4	8	16	32	64	128
1	0.03	—	—	—	—	—	—	—
2	0.03	0.05	—	—	—	—	—	—
4	0.55	3	5	—	—	—	—	—
8	3,222	3,034	12,819	**38,705**	—	—	—	—
16	>18,000 (Out of Memory: >4 GB)					—	—	—

Table 2: CPU time in seconds for checking the unsatisfiability of the CNF formula, i.e., the correctness of the implementation processor, when only Positive Equality was used. The SAT-checker Chaff [22][34] was employed.

Reorder Buffer Size = 8	Issue/Retire Width							
	1	2	4	8	16	32	64	128
e_{ij} Primary Inputs	108	142	213	428	—	—	—	—
Other Primary Inputs	26	28	32	40	—	—	—	—
Total Primary Inputs	134	170	245	468	—	—	—	—
CNF Variables	822	1,041	1,533	2,794	—	—	—	—
CNF Clauses	5,685	7,804	13,430	33,704	—	—	—	—
CPU Time [s]	3,222	3,034	12,819	**38,705**	—	—	—	—

Table 3: Statistics for the CNF formulas for correctness of models with 8 Reorder Buffer entries, when only Positive Equality was used. The SAT-checker Chaff [22][34] was employed, and its CPU time is reported.

7.2 Using Both Rewriting Rules and Positive Equality

Then, the EUFM correctness formulas were translated to equivalent Boolean formulas by exploiting both rewriting rules and Positive Equality. Table 4 shows the CPU times for that translation. A large portion of the CPU times was spent reading the EUFM correctness formulas from disk and building the necessary data structures. For example, in the case of the processor with 1,500 ROB entries and issue/retire width of 128, that took 3,347 seconds out of the total 5,485 seconds. Since the correct execution of instructions initially in the ROB is proved by rewriting rules, the Boolean correctness formulas depend only on the newly fetched instructions.

Statistics for the CNF representations of the Boolean correctness formulas are shown in Table 5. Those formulas do not contain e_{ij} variables, since the newly fetched instructions are executed strictly in program order—either when flushing the implementation processor along the implementation side of the commutative diagram by using completion functions, or when exercising the non-pipelined specification along the specification side of the diagram. That makes it possible to abstract the special uninterpreted functions *read* and *write*, used for modeling memories and satisfying the forwarding property of the memory semantics, with completely general uninterpreted functions, which do not satisfy that property [31]. That abstraction is done automatically in the tool EVC. By not considering the forwarding

property, we employ a more conservative memory model. The category "Other Primary Inputs" in Table 5 includes Boolean variables that encode the *Valid* bits of newly fetched instructions, or abstract the control signals that determine whether to fetch each new instruction. The formulas do not depend on Boolean variables that represent the initial state of control bits in the ROB, or determine whether to complete the execution of ready instructions, as those variables are eliminated by the rewriting rules.

Reorder Buffer Size	Issue/Retire Width							
	1	**2**	**4**	**8**	**16**	**32**	**64**	**128**
1	0.08	—	—	—	—	—	—	—
2	0.11	0.11	—	—	—	—	—	—
4	0.11	0.12	0.15	—	—	—	—	—
8	0.13	0.14	0.16	**0.31**	—	—	—	—
16	0.24	0.26	0.27	0.41	1.12	—	—	—
32	0.65	0.65	0.69	0.82	1.55	7.10	—	—
64	2.40	2.40	2.40	2.52	3.23	8.72	64	—
128	10	10	10	10	11	16	71	745
256	47	47	47	48	48	54	110	794
512	266	267	269	270	271	279	338	1,065
1,024	1,760	1,760	1,760	1,761	1,762	1,778	1,830	2,716
1,250	2,598	2,598	2,599	2,600	2,601	2,603	2,672	3,709
1,500	4,465	4,467	4,472	4,473	4,478	4,482	4,592	5,485

Table 4: CPU time in seconds for translating the EUFM correctness formula to an equivalent Boolean formula, when both rewriting rules and Positive Equality were used.

Any Reorder Buffer Size	Issue/Retire Width							
	1	**2**	**4**	**8**	**16**	**32**	**64**	**128**
e_{ij} **Primary Inputs**	0	0	0	0	0	0	0	0
Other Primary Inputs	2	4	8	16	32	64	128	256
Total Primary Inputs	2	4	8	16	32	64	128	256
CNF Variables	10	21	51	135	399	1,311	4,671	17,535
CNF Clauses	24	65	250	1,204	7,016	46,800	339,360	~2.6M
CPU Time [s]	0.02	0.02	0.02	**0.04**	0.18	1.35	14	171

Table 5: Statistics for the CNF formulas for correctness of models with any Reorder Buffer size, when both rewriting rules and Positive Equality were used. The results do not depend on the size of the Reorder Buffer, since the instructions initially there were processed by rewriting rules. The SAT-checker `Chaff` [22][34] was employed, and its CPU time is reported.

The total time to formally verify an out-of-order processor, when using rewriting rules, includes the time for symbolic simulation to generate the EUFM correctness formula (Table 1), the time for EUFM to CNF translation (Table 4), and the time to prove that the CNF formula is unsatisfiable (Table 5). For a processor with 1,500 ROB entries and issue/retire width of 128, the total time was 12,553 seconds, or 3:29 hours. The CNF formula for that design had almost 2.6 million clauses (Table 5), and was proved to be unsatisfiable in 171 seconds by the SAT-checker `Chaff`.

When no rewriting rules were used, the processor with 8 ROB entries and issue/retire width of 8 was formally verified after 38,708 seconds—3 seconds to translate the EUFM correctness formula to a CNF formula, and 38,705 seconds to prove that the CNF formula was unsatisfiable (Table 2). Rewriting rules reduced the total time to 0.35 seconds—0.31 seconds for the EUFM to CNF translation (Table 4), and 0.04 for SAT-checking (Table 5). In both cases, the CPU time to generate the EUFM correctness formula was 0.31 seconds (Table 1). Hence, rewriting rules result in 5 orders of magnitude speedup—reducing the CPU time from 38,708 seconds to 0.35 seconds—when solving the correctness formula for that processor.

Rewriting rules were also used when verifying a buggy variant of the processor with 128 ROB entries and issue/retire width of 4. The bug was in the forwarding logic for one of the data operands of the 72nd instruction in the ROB. The rewriting rules took 9 seconds to identify the 72nd computation slice as not conforming to the expected expression structure. (The correct design was verified in 10 seconds, when rewriting rules were employed —see Table 4.) In contrast, when only Positive Equality was used, `EVC` [33] ran out of memory after 6,100 seconds during the EUFM to CNF translation. Although it remains to be proved that the bug identified by the rewriting rules is not a false negative (that would be addressed in future work), the speedup in detecting the (potential) buggy computation slice is dramatic. Also, when the correct version of the processor was verified by using only Positive Equality, the EUFM to CNF translation similarly ran out of memory after more that 6,000 seconds. Hence, without rewriting rules, it would have been impossible to formally verify most of the benchmarks.

The manual definition of the rewriting rules took 3 days.

8 Comparison with Related Work

All previous methods for formal verification of out-of-order processors [1][2][3][11][13][14][15][16][17][18][21][23][24] are based entirely on theorem-proving, and require extensive manual work. For example, Hosabettu, *et al.* [13] report 19 person days for verification of an out-of-order processor with a Reorder Buffer and register-register instructions. In contrast, the method presented in this paper required 3 days of manual work for the definition of the rewriting rules, and is otherwise automatic. Also, the user does not have to define an induction hypothesis, as the entries in the Reorder Buffer are fully instantiated. So is the logic for resolving data hazards—in the verified processors, that is done by stalling the execution of instructions until their data operands can be either read from the Register File or forwarded from preceding entries in the Reorder Buffer. Hence, the abstract processors verified in the current paper are closer to actual implementations, compared to the abstract models in previous work. For example, McMillan and Jhala [15][21], who consider only 1 or 2 instructions in the Reorder Buffer, will need to prove the correctness of the complete logic for resolving data hazards, when the properties they proved are refined to an actual implementation.

Additionally, all previous research has examined models that can issue and retire only a single instruction per clock cycle. In contrast, the processors verified in this paper can issue and retire up to 128 instructions per clock cycle. Verifying such designs is possible due to: 1) the use of rewriting rules for processing the

instructions within the retire width, and 2) the use of Positive Equality for proving the correct execution of newly fetched instructions within the issue width.

9 Conclusions and Future Work

Rewriting rules and Positive Equality were combined to formally verify abstract out-of-order processors that implement register-register instructions, have a Reorder Buffer with up to 1,500 instructions, and can issue/retire up to 128 instructions per clock cycle. Scaling to larger configurations was limited by the available 4 GB of physical memory. However, the above numbers are more than 10 times greater than those in current state-of-the-art processors [7][9][12]. The identical structure of the computation slices, abstracting the execution of instructions initially in the Reorder Buffer, makes it easy to define rewriting rules to prove that each instruction initially in the processor produces equal updates along both sides of the commutative diagram. Then, the equal updates are removed, and the simplified correctness formula, which depends only on the updates done by the newly fetched instructions, is processed by exploiting Positive Equality. Rewriting rules resulted in up to 5 orders of magnitude speedup, compared to using Positive Equality alone. Indeed, without rewriting rules, it would have been impossible to formally verify most of the benchmarks.

Future work will focus on verifying out-of-order processors based on register renaming, as well as processors that support more instruction types, and implement exceptions and branch prediction. Also, it will be useful to give the user flexibility in defining rewriting rules. Another direction will be to automate the synthesis of an implementation with a specific structural configuration of functional units, data buses, etc., starting from a formally verified abstract model. Finally, the presented method has potential for exploiting parallelism—the rewriting rules can be applied simultaneously to each of the instructions initially in the Reorder Buffer.

References

[1] T. Arons, and A. Pnueli, "Verifying Tomasulo's Algorithm by Refinement " *12th International Conference on VLSI Design (VLSI '99)*, June 1999, pp. 306–309.

[2] T.Arons, and A.Pnueli, "A Comparison of Two Verification Methods for Speculative Instruction Execution," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, S. Graf, and M. Schwartzbach, *eds.*, LNCS 1785, Springer-Verlag, March–April 2000, pp. 487–502.

[3] S. Berezin, E.M. Clarke, A. Biere, and Y. Zhu, "Verification of Out-Of-Order Processor Designs Using Model Checking and a Light-Weight Completion Function," Journal on Formal Methods in System Design (FMSD), 2001.

[4] R.E. Bryant, S. German, and M.N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic,"[3] ACM Transactions on Computational Logic (TOCL), Vol. 2, No. 1 (January 2001).

[5] R.E. Bryant, and M.N. Velev, "Boolean Satisfiability with Transitivity Constraints,"[3] ACM Transactions on Computational Logic (TOCL), *accepted to appear*, 2002.

[6] J.R. Burch, and D.L. Dill, "Automated Verification of Pipelined Microprocessor Control," *Computer-Aided Verification* (*CAV '94)*, D.L. Dill, *ed.*, LNCS 818, Springer-Verlag, June 1994, pp. 68–80.

[7] K. Diefendorff, "Athlon Outruns Pentium III," Microprocessor Report, Vol. 13, No. 11 (August 23, 1999), pp. 1, 6–11.

[8] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 244–255.

[9] L. Gwennap, "Alpha 21364 to Ease Memory Bottleneck," Microprocessor Report, Vol. 12, No. 14 (October 26, 1998), pp. 12–15.

[10] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[11] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "You Assume, We Guarantee: Methodology and Case Studies," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 440–451.

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium® 4 Processor,"[4] Intel Technology Journal, 1st Quarter, 2001.

[13] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Proof of Correctness of a Processor with Reorder Buffer Using the Completion Functions Approach," *Computer-Aided Verification (CAV '99)*, N. Halbwachs, and D. Peled, *eds.*, LNCS 1633, Springer-Verlag, July 1999, pp. 45–59.

[14] R. Hosabettu, "Systematic Verification of Pipelined Microprocessors," Ph.D. Thesis, Department of Computer Science, University of Utah, August 2000.

[15] R. Jhala, and K.L. McMillan, "Microarchitecture Verification by Compositional Model Checking," *Computer-Aided Verification (CAV '01)*, G. Berry, H. Comon, and A. Finkel, *eds.*, LNCS 2102, Springer-Verlag, July 2001, pp. 396–410.

[16] R.B. Jones, J.U. Skakkebæk, and D.L. Dill. "Formal Verification of Out-of-Order Execution with Incremental Flushing," Journal on Formal Methods in System Design (FMSD), 2001.

[17] R.B. Jones, *Formal Verification with Symbolic Simulation*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.

[18] D. Kroening, S.M. Mueller, and W.J. Paul, "A Rigorous Correctness Proof of a Tomasulo Scheduler Supporting Precise Interrupts," *Systemics, Cybernetics and Informatics (SCI '99), and Information Systems, Analysis and Synthesis (ISAS '99)*, August 1999.

[19] J. Levitt, and K. Olukotun, "Verifying Correct Pipeline Implementation for Microprocessors," *International Conference on Computer-Aided Design (ICCAD '97)*, November 1997, pp. 162–169.

[20] J.R. Levitt. "Formal Verification Techniques for Digital Systems," Ph.D. Thesis, Department of Electrical Engineering, Stanford University, December 1998.

[21] K.L. McMillan, "A Methodology for Hardware Verification Using Compositional Model Checking," Science of Computer Programming, Vol. 37, No. 1–3 (May 2000), pp. 279–309.

[22] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 530–535.

[23] J. Sawada, "Formal Verification of an Advanced Pipelined Machine," Ph.D. thesis, Department of Computer Science, University of Texas at Austin, December 1999.

[24] J. Sawada, and W.A. Hunt, Jr., "Verification of FM9801: Out-of-Order Processor with Speculative Execution and Exceptions That May Execute Self-Modifying Code," Journal on Formal Methods in System Design (FMSD), 2001.

[25] J.E. Smith, and A.R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Transactions on Computers, Vol. 37, No. 5 (May 1988), pp. 562–573.

[26] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," IEEE Transactions on Computers, Vol. 39, No. 3 (March 1990), pp. 349–359.

[27] M.N. Velev, and R.E. Bryant, "Incorporating Timing Constraints in the Efficient Memory Model for Symbolic Ternary Simulation,"[3] *International Conference on Computer Design (ICCD '98)*, October 1998, pp. 400–406.

[28] M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic,"[3] *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre, and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September 1999, pp. 37–53.

[29] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction,"[3] *37th Design Automation Conference (DAC '00)*, June 2000, pp. 112–117.

[30] M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution,"[3] *Computer-Aided Verification (CAV '00)*, E.A. Emerson, and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 86–98.

[31] M.N. Velev, "Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors,"[3] *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, T. Margaria, and W. Yi, *eds.*, LNCS 2031, Springer-Verlag, April 2001, pp. 252–267.

[32] M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors,"[3] *38th Design Automation Conference (DAC '01)*, June 2001, pp. 226–231.

[33] M.N. Velev, and R.E. Bryant, "EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations,"[3] *Computer-Aided Verification (CAV '01)*, G. Berry, H. Comon, and A. Finkel, *eds.*, LNCS 2102, Springer-Verlag, July 2001, pp. 235–240.

[34] L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," *International Conference on Computer-Aided Design (ICCAD '01)*, November 2001.

3. http://www.ece.cmu.edu/~mvelev

4. http://developer.intel.com/technology/itj/q12001/articles/art_2.htm