Using Positive Equality to Prove Liveness for Pipelined Microprocessors

Miroslav N. Velev

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

Abstract—The paper presents an indirect method to automatically prove liveness for pipelined microprocessors. This is done by first proving safety—correctness for one step, starting from an arbitrary initial state that is possibly restricted by invariant constraints. By induction, the implementation will be correct for any number of steps; we need to prove that for some fixed number of steps, *n*, the implementation will fetch at least one instruction that will be completed. This was proved efficiently by using the property of Positive Equality. Modeling restrictions made the method applicable to designs with exceptions and branch prediction. The indirect method and the modeling restrictions resulted in 4 orders of magnitude speedup, enabling the automatic liveness proof for dual-issue superscalar and VLIW designs.

I. Introduction

Previous work on microprocessor formal verification has almost exclusively addressed the proof of *safety*—that if a processor does something during a step, it will do it correctly—as also observed in [2], while ignoring the proof of *liveness—that a processor will complete a new instruction after a finite number of steps*. Several authors used theorem proving to check liveness [11][12][13][15][18][20][24][26], but invested extensive manual work. This paper is the first to automatically prove liveness for pipelined processors, including models with exceptions and branch prediction.

In the current paper, the implementation and specification are described in the high-level hardware description language AbsHDL [32], based on the logic of Equality with Uninterpreted Functions and Memories (EUFM) [6]. In EUFM, word-level values are abstracted with terms (see Section IV) whose only relevant property is that of equality with other terms. Restrictions on the style for describing high-level processors [27][28] reduced the number of terms that appear in both positive and negated equality comparisons—and are so called *g-terms* (for general terms)—and increased the number of terms that appear only in positive polarity—and are so called *p-terms* (for positive terms). The property of Positive Equality [27][28] allowed us to treat syntactically different p-terms as not equal when evaluating the validity of an EUFM formula, thus achieving significant simplifications and orders of magnitude speedup. (See [4] for a correctness proof.)

The formal verification is done with an automatic tool flow, consisting of: 1) the term-level symbolic simulator TLSim [32], used to symbolically simulate the implementation and specification, and produce an EUFM correctness formula; 2) the decision procedure EVC [32] that exploits Positive Equality and other optimizations to translate the EUFM correctness formula to an equivalent Boolean formula, which has to be a tautology in order for the implementation to be correct; and 3) an efficient SAT-checker. This tool flow was used at Motorola [14] to formally verify a model of the M•CORE processor, and detected bugs. The tool flow was also used in an advanced computer architecture course [34][36], where students designed and formally verified pipelined processors.

II. Definition of Safety and Liveness

The formal verification is done by correspondence checking—comparison of a pipelined implementation against a non-pipelined specification. The abstraction function, *Abs*, maps an implementation state to an equivalent specification state, and is computed by *flushing* [6]—feeding the implementation pipeline with bubbles (combinations of control signals that do not modify architectural state) until all partially executed instructions are completed. The safety property (see Fig. 1) is expressed as a formula in the logic of EUFM, and checks that one step of the implementation corresponds to between 0 and *k* steps of the specification, where *k* is the issue width of the implementation. F_{Impl} is the transition function of the implementation, and F_{Spec} is the transition function of the specification. We will refer to the sequence of first applying *Abs* and then F_{Spec} as the *specification side* of the diagram in Fig. 1, and to the sequence of first applying F_{Impl} and then *Abs* as the *implementation side.*

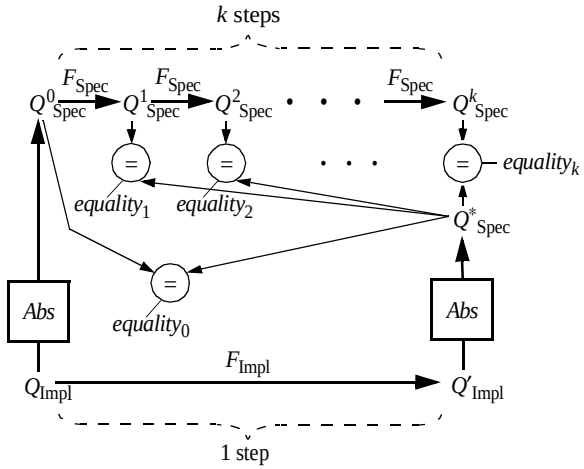

Fig. 1. The safety correctness property for an implementation processor with issue width *k*: one step of the implementation should correspond to between 0 and *k* steps of the specification, when the implementation starts from an arbitrary initial state Q_{Impl} that is possibly restricted by a set of invariant constraints.

The safety property is a proof by induction, since the initial implementation state, Q_{Impl}, is completely arbitrary. If the implementation is correct for all transitions that can be made for one step from an arbitrary initial state, then the implementation will be correct for one step from the next implementation state, Q'_{Impl}, since that state will be a special case of an arbitrary state as used for the initial state, and so on for any number of steps. For some processors, e.g., where the control logic is optimized by using unreachable states as don't-care conditions, we might have to impose a set of *invariant constraints* for the initial state

in order to exclude unreachable states. Then, we need to prove that those constraints will be satisfied in the implementation state after one step, Q'_{Impl}, so that the correctness will hold by induction for that state, and so on for all subsequent states. The reader is referred to [1][2] for a discussion of correctness criteria.

To illustrate the safety property in Fig. 1, let the implementation and specification have three architectural state elements—Program Counter (PC), Register File, and Data Memory. Let PC^i_{Spec}, $RegFile^i_{\mathrm{Spec}}$, and $DMem^i_{\mathrm{Spec}}$ be the state of the PC, Register File, and Data Memory, respectively, in specification state Q^i_{Spec} (i = 0, ..., k) along the specification side of the diagram. Let PC^*_{Spec}, $RegFile^*_{\mathrm{Spec}}$, and $DMem^*_{\mathrm{Spec}}$ be the state of the PC, Register File, and Data Memory in specification state Q^*_{Spec}, reached after the implementation side of the diagram. Then, each disjunct $equality_i$ (i = 0, ..., k) is defined as:

$$equality_i \leftarrow pc_i \wedge rf_i \wedge dm_i,$$

where

$$pc_i \leftarrow (PC^i_{\mathrm{Spec}} = PC^*_{\mathrm{Spec}}),$$
$$rf_i \leftarrow (RegFile^i_{\mathrm{Spec}} = RegFile^*_{\mathrm{Spec}}),$$
$$dm_i \leftarrow (DMem^i_{\mathrm{Spec}} = DMem^*_{\mathrm{Spec}}).$$

That is, $equality_i$ is conjunction of pair-wise equality comparisons for all architectural state elements, thus ensuring that they are updated in synchrony by the same number of instructions. In processors with more architectural state elements, an equality comparison is conjuncted similarly for each additional state element. Hence, for this implementation processor, the safety property is:

$$pc_0 \wedge rf_0 \wedge dm_0 \vee pc_1 \wedge rf_1 \wedge dm_1 \vee \ldots \vee pc_k \wedge rf_k \wedge dm_k = true.$$

We can prove liveness by a modified version of the safety correctness criterion—by symbolically simulating the implementation for a finite number of steps, n, and proving that:

$$equality_1 \vee equality_2 \vee \ldots \vee equality_{n \times k} = true \qquad (1)$$

where k is the issue width of the implementation. The formula proves that n steps of the implementation match between 1 and $n \times k$ steps of the specification, when the implementation starts from an arbitrary initial state that may be restricted by invariant constraints. Note that (1) *guarantees that the implementation has made at least one step*, while the safety correctness criterion allows the implementation to stay in its initial state when formula $equality_0$ (checking whether the implementation matches the initial state of the specification) is *true*. The correctness formula is generated automatically in the same way as the formula for safety, except that the implementation and the specification are symbolically simulated for many steps, and formula $equality_0$ is not included. As in the formula for safety, every formula $equality_i$ is the conjunction of equations, each comparing corresponding states of the same architectural state element. That is, formula (1) consists of top-level positive equations that are conjuncted and disjuncted but not negated, allowing us to exploit Positive Equality when proving liveness. The minimum number of steps, n, to symbolically simulate the implementation, can be determined experimentally, by trial and error, or can be provided by the user, based on knowledge about the stalling and squashing behavior of the implementation (see Section V).

The contribution of this paper is a method to indirectly prove liveness (1) by first proving safety, thus inductively the implementation correctness for n steps, and then using Positive Equality to prove that $equality_0$ will be *false* after n steps. For the latter proof, every PC transition is made unique for each instruction; the logic comparing actual and predicted branch targets to find mispredictions is abstracted; and the specification is enriched with the abstract mechanism for correcting branch mispredictions.

III. Related Work

Safety and liveness were first defined by Lamport [16]. Most of the previous research on formal verification of processors has addressed only safety, as also observed in [2]. The most popular theorem-proving approach for proving microprocessor liveness is to prove that for each pipeline stage that can get stalled, if the stalling condition is *true* then the instruction initially in that stage will stay there, and if the stalling condition is *false* then the instruction will advance to the next stage. It is additionally proved that if the stalling condition is *true*, then it will eventually become *false*, given the implementation of the control logic and fairness assumptions about arbiters. Liveness was proved in this way by Srivas and Miller [26], Hosabettu et al. [11], Jacobi and Kröning [12], Müller and Paul [20], Kröning and Paul [13], and Lahiri et al. [15]. Sawada [24] similarly proved that if an implementation is fed with bubbles, it will eventually get flushed. However, note that a buggy processor, where the architectural state elements are always disabled, may pass the check that stall signals will eventually become *false*, and that the pipeline will eventually get flushed, as well as satisfy the safety correctness criterion (where formula $equality_0$ will be *true*), but will fail the liveness check done here. Using a different theorem-proving approach, Manolios [18] also accounted for liveness by proving that a given state can be reached from a flushed state after an appropriate number of steps. McMillan [19] used circular compositional reasoning to check the liveness of a reduced model of an out-of-order processor with ALU and move instructions. His method requires the manual definition of lemmas and case-splitting expressions; the manual reduction of the proof to one that involves two reservation stations and one register; and the manual introduction of fairness assumptions for the abstracted arbiter. The approaches in the above nine papers will require significant manual work to apply to the models that are automatically checked for both safety and liveness in the current paper. Aagaard et al. [1] formulated a liveness condition, but did not present results.

Henzinger et al. [10] also enriched the specification, but had to do that even to prove safety for a 3-stage pipeline with ALU and move instructions.

Biere et al. [3] enriched a model with a witnessing mechanism that records whether a property has been satisfied, thus allowing them to model check liveness of a communication protocol as safety. Pnueli et al. [21] proved the liveness of mutual-exclusion algorithms by deriving an abstraction, and enriching it with conditions that allowed the efficient liveness check in a way that implies the liveness of the original model.

IV. EUFM and Positive Equality

The syntax of EUFM [6] includes *terms* and *formulas*. Terms are used to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memory arrays. A term can be an Uninterpreted Function (UF) applied to 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, term1, term2*) will evaluate to *term1* if *formula = true*, and to *term2* if *formula = false*. The syntax for terms can be extended to model memories by means of the functions *read* and *write* [6][31]. Formulas are used to model the control path of a microprocessor, and to express the correctness condition. A formula can be an Uninterpreted Predicate (UP) applied to a list of argument terms, a Boolean 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 combined by Boolean connectives. We will refer to both terms and formulas as *expressions*. UFs and UPs are used to abstract the implementation details of functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*—that the same combinations of values to the inputs of the UF (UP) produce the same output value.

The efficiency from exploiting Positive Equality is due to the observation that the truth (validity) 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 (a p-equation) evaluates to *false*; and that of a g-term variable with a syntactically distinct g-term variable (a g-equation) could evaluate to either *true* or *false*, and can be encoded with Boolean variables, e.g., as done in [8][22][35].

To model branch prediction, the mechanism for correcting mispredictions is fully implemented, while the branch predictor is abstracted with a *generator of arbitrary values* [29] that makes non-deterministic predictions on every clock cycle, thus allowing us to prove correctness of the processor for any actual implementation of the branch predictor. To model exceptions raised by a functional unit [29], we add an uninterpreted predicate that has the same inputs as the functional unit, and output that indicates whether an exception is raised; additional architectural state elements keep the exception status.

V. Processor Benchmarks and Their Liveness

The benchmarks include 1×DLX-C, a single-issue pipelined DLX [9]; 1×DLX-C-BP-EX, an extension with both branch prediction and exceptions; 2×DLX-CA, a dual-issue superscalar version with one complete pipeline that can execute all instruction types, and a second pipeline that can execute only ALU instructions; 2×DLX-CC, a dual-issue model with two complete pipelines; and versions with both branch prediction and exceptions, 2×DLX-CA-BP-EX and 2×DLX-CC-BP-EX. The reader is referred to [29] for detailed descriptions of these 6 models. The most complex benchmarks are variants of a VLIW architecture [30][33]: 9VLIW implements predicated execution, register remapping, and advanced loads [25]; and 9VLIW-BP-EX is an extension with both branch prediction and exceptions. All models have 5 pipeline stages: Fetch, Decode, Execute, Memory, and Write-Back.

To illustrate the choice of number of steps, *n*, for the liveness proof of 1×DLX-C, note that the longest delay before this model fetches a new instruction that is guaranteed to be completed is 5 cycles. This will happen if the Decode stage contains a branch that will be taken, but has a data dependency on a load in Execute. The branch will be stalled for one cycle, due to the load-interlock [9]; then it will take a cycle to go through Decode; another to go through Execute, where the branch target and direction will be computed; a fourth cycle to go through Memory, updating the PC with the branch target, and squashing all subsequent instructions; and a fifth cycle to fetch a new instruction that is guaranteed to be completed, since the pipeline will be empty by then. Hence, a correct 1×DLX-C has to be symbolically simulated for 5 steps to fetch a new instruction that is guaranteed to be completed. In the case of 1×DLX-C-BP-EX, where exceptions are resolved in the Write-Back stage by squashing all subsequent instructions and updating the PC with an exception-handler address, it takes an extra cycle for an excepting instruction to get to Write-Back, so that this model has to be simulated for 6 steps to fetch a new instruction that will be completed.

The dual-issue superscalar models without exceptions will have to be simulated for 7 cycles to fetch a new instruction that is guaranteed to be completed. This will happen if the older instruction in Decode has a data dependency on a load in Execute, so that both instructions in Decode will be stalled during the first cycle (the superscalar models have in-order issue); if the older instruction in Decode is a load that provides data for the younger instruction in that stage, then the younger instruction will be shifted to the first slot in Decode during the second cycle, and then stalled in that slot during the third cycle; and if that younger instruction is a branch that is taken/mispredicted and not squashed, it will take another 4 cycles (as in the single-issue models) to fetch a new instruction that is guaranteed to be completed. Since a dual-issue processor can fetch and complete up to 2 instructions per cycle, it could complete up to 14 instructions over 7 steps, and so has to be compared for a match with between 1 and 14 specification steps. In the case of 2×DLX-CA-BP-EX and 2×DLX-CC-BP-EX, exceptions are again resolved in the Write-Back stage that takes an extra cycle to reach, so that these models have to be symbolically simulated for 8 cycles to fetch a new instruction that is guaranteed to be completed, and can match between 1 and 16 specification steps.

VI. Proving Liveness Indirectly

To avoid the validity checking of the monolithic liveness correctness formula (1), which can get very big for complex processors, we can prove liveness indirectly:

THEOREM 1. *If after n implementation steps, $equality_0$ = false under a maximally diverse interpretation of the p-terms, and the safety property is valid, then the liveness property is valid under any interpretation.*

Proof: If the safety property is valid, by induction over *n* implementation steps, it follows that:

$$equality_0 \vee equality_1 \vee \ldots \vee equality_{n \times k} = true \qquad (2)$$

under any interpretation, including a maximally diverse interpretation of the p-terms. Then, if after *n* implementation steps $equality_0 = false$ under a maximally diverse interpretation of the p-terms, it follows that:

$$equality_1 \vee \ldots \vee equality_{n \times k} = true \qquad (3)$$

under a maximally diverse interpretation of the p-terms. Since formula (3) is the same as the liveness condition (1), then from the property of Positive Equality, it follows that the liveness condition (1) will be valid under any interpretation. □

Note that under an interpretation that is not a maximally diverse interpretation of the p-terms, the condition $equality_0$ may become *true*, e.g., in the presence of software loops, or if multiple instructions raise the same exception (in the benchmarks, a raised exception did not suppress later ones) and so update the PC with the same exception-handler address. However, the liveness condition (1) will be still valid, since it can only get "bigger" under an interpretation that is not a maximally diverse interpretation of the p-terms (and that is how it was indirectly proved valid under any interpretation).

Since $equality_0$ is conjunction of the pair-wise equality comparisons for all architectural state elements, it suffices to prove that any of those equality comparisons is *false* under a maximally diverse interpretation of the p-terms. In particular, we can prove that $pc_0 = false$, where pc_0 is the equality comparison between the PC state after the implementation side of the diagram (see Fig. 1), and the PC that is part of the initial specification state. Note that choosing the Register File or the Data Memory instead would not

work, since they are not updated by each instruction, and so there can be infinitely long instruction sequences that do not modify these architectural state elements, making it impossible to prove that a given (finite) number of steps will always result in at least one update. For the benchmarks from Section V, the PC is the only state element that is updated by every instruction.

Note that proving *forward progress—that the PC is updated at least once after n implementation steps*, i.e., proving $pc_0 = false$ under a maximally diverse interpretation of the p-terms—is done without the specification. However, the specification is used to prove safety, thus inductive correctness for any number of steps.

VII. Modeling Restrictions for Processors with Exceptions and Branch Prediction

In order to apply Theorem 1 to processors with exceptions and branch prediction, we impose two modeling restrictions. The goal is: 1) to make each PC transition unique for every instruction by introducing *witness functions*—UFs that take as arguments the instruction opcode and the original term for one of the next PC values, and produce a unique new term used to update the PC in the modified design, thus witnessing each PC transition for each instruction; and 2) to have the PC updated with p-terms only. Such restrictions allow us to use Positive Equality to efficiently prove forward progress.

Unique PC transitions. In models without branch prediction, the symbolic values used to update the PC are: *SequentialPC*, pointing to the sequential instruction and obtained after incrementing the PC, where the incrementing is abstracted with a UF; *Target*, the target address for a jump or a branch, as computed by a functional unit abstracted with a UF; exception-handler addresses, each modeled with a term variable [29]; and the Exception PC (EPC) that contains the address of the last excepting instruction, and is used by return-from-exception instructions to restart the execution from the last excepting instruction. With this modeling, the *SequentialPC* and the *Target* are different symbols for each instruction, since they are produced by UFs that depend on, respectively, the PC and the opcode that are p-terms. However, the exception-handler addresses remain the same symbols, and the EPC can keep its value during the execution of many instructions (if no exceptions are raised), so that the PC could be updated with these terms by many instructions, making it impossible to prove forward progress by proving $pc_0 = false$, as there is no way to distinguish between updates with the same term by different instructions. Our solution is to make the exception-handler addresses and the EPC unique for each instruction by introducing witness functions—UFs taking the instruction opcode and an original term for a next PC value, and producing a new term used to update the PC. Since the opcodes are p-terms that are unique for each symbolic instruction, they will make the outputs of those UFs also unique for each instruction, thus allowing us to mark each PC transition for each instruction. If an EUFM formula is valid with such UFs, the formula will be valid for any functionally consistent implementation of the UFs, including the original one where one of the inputs is directly connected to the output. These abstractions have to be applied manually to both the implementation and the specification. In earlier work [27], the opcode was also used as extra input to UFs and UPs, resulting in *functional non-consistency.*

PC updates with p-terms only. In the case of models with branch prediction, the mechanism for correcting mispredictions compares the actual and predicted targets for equality, and the output is used in dual polarity [29]—positive polarity when updating the PC, and negated polarity when squashing subsequent instructions—which makes the actual and predicted targets g-terms. Then, in formula pc_0, i.e., $(PC^0_{Spec} = PC^*_{Spec})$, the term after the implementation side, PC^*_{Spec}, could equal the term after the specification side, PC^0_{Spec}, when software loops are executed, making it impossible to prove $pc_0 = false$ under a maximally diverse interpretation of the p-terms. Our solution is to abstract the equality comparison between the actual and predicted targets with a UP that also takes the opcode as input. Then, the actual and predicted targets become p-terms. Normally, a specification processor does not require branch prediction, since the actual branch outcome is known by the end of a step, just in time to fetch the correct instruction in the next step. However, the abstracted branch-target equality in the implementation requires that the specification be enriched with that abstraction, and be extended with a branch prediction mechanism in order to produce identical PC updates. Otherwise, the equations comparing PC states in the EUFM correctness formulas, will not be able to correlate the predicted targets from the implementation and the actual targets from the specification. In order to enrich the specification with branch prediction, and synchronize the predictions with those made by the implementation for the same instructions, the predicted direction is produced by a UP and the predicted target by a UF, both having the opcode as input. These UP and UF are introduced in both the implementation and the specification. Since the opcodes are p-terms, each opcode will be mapped to a new Boolean variable for the predicted direction, and a new term variable for the predicted target, so that the predictions will still be arbitrary. For example, let the original circuit for the next PC value in the specification be:

$$\begin{aligned} &use\_target && \leftarrow is\_Jump \vee is\_Branch \wedge taken\_branch \\ &nextPC && \leftarrow ITE(use\_target, Target, SequentialPC) \end{aligned}$$

where control bits *is_Jump* and *is_Branch* indicate a jump and a branch; *taken_branch* is the actual direction of a branch; *Target* is the actual target; and *SequentialPC* is the sequential value of the PC. Then, the enriched specification will have the following circuit for the next PC value:

$$\begin{aligned} &use\_target && \leftarrow is\_Jump \vee is\_Branch \wedge taken\_branch \\ &nextPC\_old && \leftarrow ITE(use\_target, Target, SequentialPC) \\ &PredictedTarget && \leftarrow GetPredictedTarget(Op) \\ &predict\_taken && \leftarrow GetPredictedDirection(Op) \\ &equal\_targets && \leftarrow EqualTargets(Op, PredictedTarget, Target) \\ &use\_pred\_target && \leftarrow equal\_targets \wedge (is\_Jump \\ & && \quad \vee is\_Branch \wedge taken\_branch \wedge predict\_taken) \\ &nextPC && \leftarrow ITE(use\_pred\_target, \\ & && \quad PredictedTarget, nextPC\_old) \end{aligned}$$

where *GetPredictedTarget*() is a UF that maps the opcode to a predicted target; *GetPredictedDirection*() is a UP that maps the opcode to a predicted direction; *EqualTargets*() is the UP that abstracts the equality comparison between the actual and predicted branch targets. Hence, if the prediction is correct—based on UP *EqualTargets*()—the next PC value of the specification will be the predicted target, *PredictedTarget*; otherwise, the original next value, *nextPC_old*. It can be proved that if *EqualTargets*(*Op*, *PredictedTarget*, *Target*) is replaced with the original equation (*PredictedTarget* = *Target*), then *nextPC* = *nextPC_old*, i.e., the behavior will be the same as in the original specification. This equivalence proof requires a positive equality comparison, and can be done with the decision procedure EVC [32]. The resulting Boolean correctness formula had 5 variables, its negation had 16 CNF variables and 40 clauses, and was proved unsatisfiable in 0.001 seconds by the SAT-checker Siege [23]. As a result of abstracting the branch-target equation with a UP, the branch targets become p-terms, and a Boolean variable is used to encode only each unique pair of actual and predicted targets for each opcode.

THEOREM 2. *If an EUFM formula is valid when the equation between actual and predicted branch targets is abstracted with a UP in the mechanism for correcting branch mispredictions in the implementation, and the specification is enriched with branch prediction, then the EUFM formula will be valid for any functionally consistent implementation of that UP, including the original equation that turns the modified implementation and specification into their original versions.*

VIII. RESULTS

Table I summarizes the results from formally verifying safety of the benchmarks, after applying the modeling restrictions. The e_{ij} Boolean variables [8] encode g-equations, with some of those variables added in order to enforce the property of transitivity of equality. This property was enforced in EVC [32] by triangulating the e_{ij}-comparison graph [5]—where each edge represents an equality comparison between a pair of g-term variables—adding extra e_{ij} variables in a greedy manner, and imposing transitivity constraints for each triangle. Transitivity of equality constraints were included in all CNF formulas generated by EVC, although only needed for processors having branch prediction, and designed without modeling restrictions. The reader is referred to [33] for the translation to CNF format. All benchmarks were formally verified by computing the abstraction function with controlled flushing [7], where the user provides a flushing schedule that eliminates the triggering of stalling conditions, thus simplifying the correctness formula. The reported time is the sum of the TLSim, EVC, and SAT-checking times. SAT-checking was done with Siege [23]—one of the winners in the SAT'03 competition [17]. The experiments were conducted on a Dell OptiPlex GX260 having a 3.06-GHz Intel Pentium 4 processor with a 512-KB on-chip L2-cache, 2 GB of memory, and running Red Hat Linux 9.

TABLE I
PROVING SAFETY, WITH THE MODELING RESTRICTIONS

Processor	Impl Steps	Matching Spec Steps	Boolean Variables		CNF		Time [sec]
			e_{ij}	Total	Vars	Clauses	
1×DLX-C	1	0–1	30	67	363	1,825	0.08
1×DLX-C-BP-EX	1	0–1	36	84	425	2,225	0.08
2×DLX-CA	1	0–2	179	241	1,475	13,558	0.33
2×DLX-CA-BP-EX	1	0–2	183	310	2,350	20,875	0.49
2×DLX-CC	1	0–2	183	257	1,823	17,200	0.63
2×DLX-CC-BP-EX	1	0–2	185	335	2,963	27,575	1.12
9VLIW	1	0–1	2,450	2,688	11,722	158,913	16
9VLIW-BP-EX	1	0–1	2,746	3,094	13,565	203,166	25

As Table I shows, proving safety of the benchmarks, designed with the modeling restrictions, took between 0.08 and 25 seconds. The Boolean correctness formulas had between 67 and 3,094 Boolean variables; between 363 and 13,565 CNF variables; and between 1,825 and 203,166 CNF clauses. Without modeling restrictions, the safety proofs for these benchmarks required similar verification times.

Results from the monolithic proof of liveness, based on criterion (1)—without the modeling restrictions—are presented in Table II. The verification time was between 4 and 2,605 seconds for the single-issue benchmarks, but more than 24 hours for the dual-issue and VLIW benchmarks. The Boolean correctness formulas had between 439 and 26,505 Boolean variables; between 6,129 and 316,869 CNF variables; and between 65,114 and 7,744,554 CNF clauses.

TABLE II
MONOLITHIC PROOF OF LIVENESS, WITHOUT MODELING RESTRICTIONS

Processor	Impl Steps	Matching Spec Steps	Boolean Variables		CNF		Time [sec]
			e_{ij}	Total	Vars	Clauses	
1×DLX-C	5	1–5	346	439	6,129	65,114	4
1×DLX-C-BP-EX	6	1–6	734	919	24,104	339,857	2,605
2×DLX-CA	7	1–14	2,948	3,170	115,047	2,148,916	>24h
2×DLX-CA-BP-EX	8	1–16	4,408	4,832	421,067	8,752,989	>24h
2×DLX-CC	7	1–14	2,950	3,190	135,281	2,552,910	>24h
2×DLX-CC-BP-EX	8	1–16	4,931	5,389	600,699	11,589,467	>24h
9VLIW	5	1–5	24,801	25,391	209,622	6,064,778	>24h
9VLIW-BP-EX	5	1–5	25,669	26,505	316,869	7,744,554	>24h

Table III summarizes the benefit of modeling restrictions when proving liveness monolithically. For the most complex single-issue benchmark, 1×DLX-C-BP-EX, the Boolean variables were reduced from 919 to 674; the CNF variables from 24,104 to 14,628; the CNF clauses from 339,857 to 161,477 (more than 2×); and the SAT-checking time from 2,605 seconds to 32 seconds (81×). However, the modeling restrictions did not help with the dual-issue and VLIW designs—the Boolean correctness formulas were smaller (up to 2.5×), but the verification still did not complete in 24 hours. Compared to the safety proof for these benchmarks (see Table I), the monolithic liveness proof resulted in an order of magnitude increase in Boolean variables, and up to two orders of magnitude increase in CNF variables and clauses, as both the implementation and the specification were symbolically simulated 5–8 times longer, thus generating more complex formulas.

TABLE III
MONOLITHIC PROOF OF LIVENESS, WITH MODELING RESTRICTIONS

Processor	Impl Steps	Matching Spec Steps	Boolean Variables		CNF		Time [sec]
			e_{ij}	Total	Vars	Clauses	
1×DLX-C[a]	5	1–5	346	439	6,129	65,114	4
1×DLX-C-BP-EX	6	1–6	467	674	14,628	161,477	32
2×DLX-CA[a]	7	1–14	2,948	3,170	115,047	2,148,916	>24h
2×DLX-CA-BP-EX	8	1–16	3,666	4,136	223,495	3,921,442	>24h
2×DLX-CC[a]	7	1–14	2,950	3,190	135,281	2,552,910	>24h
2×DLX-CC-BP-EX	8	1–16	3,666	4,180	258,650	4,520,133	>24h
9VLIW[a]	5	1–5	24,801	25,391	209,622	6,064,778	>24h
9VLIW-BP-EX	5	1–5	24,637	25,497	243,258	6,387,669	>24h

a. The numbers for 1×DLX-C, 2×DLX-CA, 2×DLX-CC, and 9VLIW are the same as in Table II, since these benchmarks do not have branch prediction or exceptions, and were not affected by the modeling restrictions

Table IV presents results from proving $pc_0 = false$ under a maximally diverse interpretation of the p-terms. Then, from Theorem 1, if the safety property is valid (see Table I), the liveness property follows indirectly. As Table IV shows, the indirect method enabled the liveness check for all benchmarks. While the monolithic proof of liveness (see Table III) did not complete in 24 hours for the dual-issue superscalar and VLIW processors, the indirect method takes between 3.9 and 52 seconds for these models. That is, the speedup is at least *4 orders of magnitude* for the dual-issue superscalar benchmarks, and at least *3 orders of magnitude* for the VLIW benchmarks. Note that we can prove safety and $pc_0 = false$ in parallel, if 2 CPUs are available, and so achieve additional speedup that might be helpful for more complex designs.

TABLE IV
INDIRECT PROOF OF LIVENESS, BY PROVING PC_0 = *FALSE* UNDER A MAXIMALLY DIVERSE INTERPRETATION OF THE P-TERMS, WITH THE MODELING RESTRICTIONS

Processor	Impl Steps	Boolean Variables		CNF		Time [sec]		
		e_{ij}	Total	Vars	Clauses	pc_0 = *false*	Safety[a]	Total
1×DLX-C	5	162	216	2,531	24,243	0.17	0.08	0.25
1×DLX-C-BP-EX	6	217	342	5,829	66,713	0.43	0.08	0.51
2×DLX-CA	7	1,166	1,287	59,865	657,139	3.6	0.33	3.9
2×DLX-CA-BP-EX	8	1,456	1,714	101,979	1,137,124	7.7	0.49	8.2
2×DLX-CC	7	1,166	1,305	66,533	785,391	3.45	0.63	4.1
2×DLX-CC-BP-EX	8	1,456	1,758	115,111	1,349,156	10.1	1.12	11.2
9VLIW	5	4,534	4,873	59,323	1,041,919	12	16	28
9VLIW-BP-EX	5	9,158	9,695	131,514	2,618,909	27	25	52

a. The verification times for proving safety are from Table I

The author spent two hours to apply the modeling restrictions to the 4 benchmarks with branch prediction and exceptions. However, together with the indirect method, the restrictions enabled the liveness proof for dual-issue and VLIW designs, previously requiring more than 24 hours each.

IX. CONCLUSIONS

To indirectly prove liveness for pipelined processors, and avoid validity checking of the monolithic criterion, several modeling restrictions were imposed in order to introduce witness functions for each PC transition of each instruction; the equality comparator between actual and predicted branch targets, used in the circuitry for correcting branch mispredictions, was abstracted with an uninterpreted predicate, thus turning the targets into p-terms; and the specification was enriched with a branch prediction mechanism, based on the same abstractions as in the implementation. These techniques resulted in PC state that consists of only p-terms, allowing the use of Positive Equality to efficiently prove that the PC has been modified after a finite number of implementation steps. When this result is combined with validity of the safety condition, we get an indirect proof of liveness that resulted in 4 orders of magnitude speedup, enabling the liveness proof for dual-issue superscalar and VLIW benchmarks.

REFERENCES

[1] M.D. Aagaard, N.A. Day, and M. Lou, "Relating multi-step and single-step microprocessor correctness statements," *Formal Methods in Computer-Aided Design (FMCAD '02)*, M.D. Aagaard, and J.W. O'Leary, *eds.*, LNCS 2517, Springer-Verlag, November 2002, pp. 123–141.

[2] M.D. Aagaard, B. Cook, N.A. Day, and R.B. Jones, "A framework for superscalar microprocessor correctness statements," Software Tools for Technology Transfer (STTT), Vol. 4, No. 3 (May 2003), pp. 298–312.

[3] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," Electronic Notes in Theoretical Computer Science 66, 2002.

[4] R.E. Bryant, S. German, and M.N. Velev, "Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic," ACM Transactions on Computational Logic (TOCL), Vol. 2, No. 1 (January 2001), pp. 93–134.

[5] R.E. Bryant, and M.N. Velev, "Boolean satisfiability with transitivity constraints," ACM Transactions on Computational Logic (TOCL), Vol. 3, No. 4 (October 2002), pp. 604–627.

[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] J.R. Burch, "Techniques for verifying superscalar microprocessors," *33rd Design Automation Conference (DAC '96)*, June 1996.

[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] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann, San Francisco, 2002.

[10] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Decomposing refinement proofs using assume-guarantee reasoning," *International Conference on Computer-Aided Design (ICCAD '00)*, November 2000, pp. 245–252.

[11] 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)*, LNCS 1633, Springer-Verlag, 1999.

[12] C. Jacobi, and D. Kröning, "Proving the correctness of a complete microprocessor," *30. Jahrestagung der Gesellshaft für Informatik*, Springer-Verlag, 2000.

[13] D. Kröning, and W.J. Paul, "Automated pipeline design," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 810–815.

[14] S. Lahiri, C. Pixley, and K. Albin, "Experience with term level modeling and verification of the M•CORE™ microprocessor core," *International Workshop on High Level Design, Validation and Test (HLDVT '01)*, November 2001.

[15] S.K. Lahiri, S.A. Seshia, and R.E. Bryant, "Modeling and verification of out-of-order microprocessors in UCLID," *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, Springer-Verlag, November 2002.

[16] L. Lamport, "Proving the correctness of multiprocess programs," IEEE Transactions on Software Engineering," Vol. 3, No. 2 (March 1977), pp. 125–143.

[17] D. Le Berre, and L. Simon, "Results from the SAT'03 solver competition," *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, 2003. http://www.lri.fr/~simon/contest03/results/

[18] P. Manolios, "Mechanical verification of reactive systems," Ph.D. Thesis, Department of Computer Sciences, University of Texas at Austin, 2001.

[19] K.L. McMillan, "Circular compositional reasoning about liveness," Technical Report, Cadence Berkeley Labs, 1999.

[20] S.M. Müller, and W.J. Paul, *Computer Architecture: Complexity and Correctness*, Springer-Verlag, 2000.

[21] A. Pnueli, J. Xu, and L. Zuck, "Liveness with (0, 1, infinity)-counter abstraction," *Computer-Aided Verification, (CAV '02)*, E. Brinksma, and K.G. Larsen, *eds.*, LNCS 2404, Springer-Verlag, July 2002, pp. 107–122.

[22] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel, "The small model property: how small can it be?", Journal of Information and Computation, Vol. 178, No. 1 (October 2002), pp. 279–293.

[23] L. Ryan, Siege SAT Solver v.3. http://www.cs.sfu.ca/~loryan/personal/

[24] J. Sawada, "Verification of a simple pipelined machine model," in *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[25] H. Sharangpani, and K. Arora, "Itanium processor microarchitecture," IEEE Micro, Vol. 20, No. 5 (September–October 2000), pp. 24–43.

[26] M.K. Srivas, and S.P. Miller, "Formal verification of an avionics microprocessor," Technical Report CSL-95-4, SRI International, 1995.

[27] M.N. Velev, and R.E. Bryant, "Exploiting positive equality and partial non-consistency in the formal verification of pipelined microprocessors," *36th Design Automation Conference (DAC '99)*, June 1999, pp. 397–401.

[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," *Correct Hardware Design and Verification Methods (CHARME '99)*, 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," *37th Design Automation Conference (DAC '00)*, June 2000, pp. 112–117.

[30] M.N. Velev, "Formal verification of VLIW microprocessors with speculative execution," *Computer-Aided Verification (CAV '00)*, E.A. Emerson, and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 296–311.

[31] M.N. Velev, "Automatic abstraction of memories in the formal verification of superscalar microprocessors," *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, "EVC: a validity checker for the logic of equality with uninterpreted functions and memories, exploiting positive equality and conservative transformations," *Computer-Aided Verification (CAV '01)*, LNCS 2102, Springer-Verlag, July 2001, pp. 235–240.

[33] M.N. Velev, and R.E. Bryant, "Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors," Journal of Symbolic Computation (JSC), Vol. 35, No. 2 (February 2003), pp. 73–106.

[34] M.N. Velev, "Integrating formal verification into an advanced computer architecture course," *ASEE Annual Conference & Exposition*, June 2003.

[35] M.N. Velev, "Automatic abstraction of equations in a logic of equality," *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '03)*, M.C. Mayer, and F. Pirri, *eds.*, LNAI 2796, Springer-Verlag, September 2003, pp. 196–213.

[36] M.N. Velev, "Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs," *International Test Conference (ITC '03)*, October 2003, pp. 138–147.