Efficient Formal Verification of Pipelined Processors with Instruction Queues

Miroslav N. Velev
http://www.ece.cmu.edu/~mvelev
mvelev@ece.cmu.edu

Abstract
Presented is a method for formal verification of pipelined processors with long instruction queues. The execution engine and the fetch engine (where the instruction queue is) are formally verified separately, after abstracting the other engine with a nondeterministic FSM derived from the high-level specification of that engine. Without the presented method, the monolithic formal verification of 9-stage, 9-wide VLIW processors—implementing many realistic and speculative features inspired by the Intel Itanium—scaled for models with 5 instruction-queue entries, but ran out of memory if the instruction queue was longer. The presented method resulted in 2 orders of magnitude speedup for the processor with 5 instruction-queue entries, and enabled scaling for designs with 64 instruction-queue entries.

Categories and Subject Descriptors
B.6.3 [Logic Design]: Design Aids—*Verification*; F.3.1 [Specifying and Verifying and Reasoning about Programs]: Mechanical Verification.

General Terms
Algorithms, Design, Verification.

Keywords
Decomposition, Logic of Equality, Positive Equality, SAT.

1. Introduction
To improve the performance of pipelined processors by bridging the gap between the speed of memory and CPU [12], designers are using mechanisms such as instruction queues in order to decouple the fetch engine from the execution engine, and allow the fetch engine to continue supplying instructions when the execution engine is unable to accept them. The IBM PowerPC 750 [16] has a 6-entry instruction queue, the Intel Itanium [17][30] an 8-entry instruction queue, and the PicoJava processor [23] an instruction queue that can hold up to 12 instructions. The goal of this work is to develop an approach for formal verification of complex pipelined processors with long instruction queues.

The logic of Equality with Uninterpreted Functions and Memories (EUFM) [5]—see Sect. 2—allows us to abstract functional units and memories, while completely modeling the control of a processor. Restrictions [31][32] on the modeling style for defining high-level processors in EUFM resulted in correctness formulas where most of the terms (abstracted word-level values) appear only in positive equations (equality comparisons). Such term variables can be treated as distinct constants [3], thus significantly simplifying an EUFM correctness formulas, and resulting in orders of magnitude speedup of the formal verification; we call this property *Positive Equality*. The restrictions, together with techniques to model multicycle functional units, exceptions, and branch prediction [33], allowed our tool flow [41] to be used at Motorola [19] to formally verify a model of the M•CORE processor, and detect bugs. Our tool flow consists of: 1) the term-level symbolic simulator TLSim, used to symbolically simulate the implementation and specification processors, and produce an EUFM correctness formula; 2) the decision procedure EVC that exploits Positive Equality and other optimizations to translate the EUFM formula to an equivalent Boolean formula; and 3) an efficient SAT-solver. The tool flow was also used in an advanced computer architecture course [37][42].

Newly developed efficient SAT-solvers [8][25][28] significantly sped up the solving of Boolean formulas from formal verification of processors, but would not scale for such formulas if not for the property of Positive Equality (see [36]) that results in at least 5 orders of magnitude speedup when formally verifying complex dual-issue superscalar designs. Efficient translation to CNF [38][39][40] led to another 2 orders of magnitude speedup.

GLSVLSI'04, April 26-28, 2004, Boston, Massachusetts, USA

2. Background
The formal verification is done by *correspondence checking*—comparison of a pipelined implementation against a non-pipelined specification, using flushing [5][6] to automatically compute an *abstraction function* that maps an implementation state to an equivalent specification state. The safety property (see [41]) 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. (See [1] for a discussion of correctness criteria.)

The syntax of EUFM [5] 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 on a list of argument terms, a term variable, or an *ITE* (for "if-then-else") operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, *term1*, *term2*) will evaluate to *term1* when *formula* = *true*, and to *term2* when *formula* = *false*. The syntax for terms can be extended to model memories by means of the functions *read* and *write* [5][35]. Formulas are used 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. UFs and UPs are used to abstract the functional units by replacing them with "black boxes" that satisfy only the property of *functional consistency*, i.e., equal input values produce equal output values. Then, we will prove a more general problem—that the processor is correct for any functionally consistent implementation of its functional units—but this problem is easier to prove.

We will refer to a transformation applied to both the implementation and the specification as a *conservative approximation* if it omits some properties, making the new models more general than the original ones. However, if the more general model of the implementation is verified against the more general model of the specification, then so would be the detailed implementation against the detailed specification, whose additional properties were not necessary for the verification. A conservative approximation may result in a false negative verification result, if the omitted properties are required for the verification, but would not lead to a false positive verification result.

3. VLIW Architecture to be Formally Verified

The goal of this paper is to formally verify variants of the 9-stage, 9-wide VLIW processor in Fig. 1—an extension of the model from [34] with extra pipeline stages and an instruction queue—for many entries in the queue. The Fetch Engine contains the Program Counter (PC), and a read-only Instruction Memory that is accessed over 2 clock cycles—in the two Instruction Fetch stages. The Instruction Memory maps the instruction address, provided by the PC, to a VLIW packet of 9 instructions that are already matched with one of 9 execution pipelines, each containing a specialized functional unit: 4 pipelines with integer units (Int FU), 2 with floating-point units (FP FU), and 3 with branch-address units (BA FU). Instructions in a packet do not have data dependencies between each other, as guaranteed by the compiler, and any number of instructions in a packet can be valid. Data values are stored in 4 register files—Integer (Int), Floating-Point (FP), Branch-Address (BA), and Predicate (Pred). A location in the Predicate Register File contains a single bit of data. Every instruction is predicated with a qualifying predicate register, such that the result of the instruction is written to architectural state only if the instruction's qualifying predicate register has a value of 1. A Branch Predictor supplies the Fetch Engine with one prediction on every clock cycle.

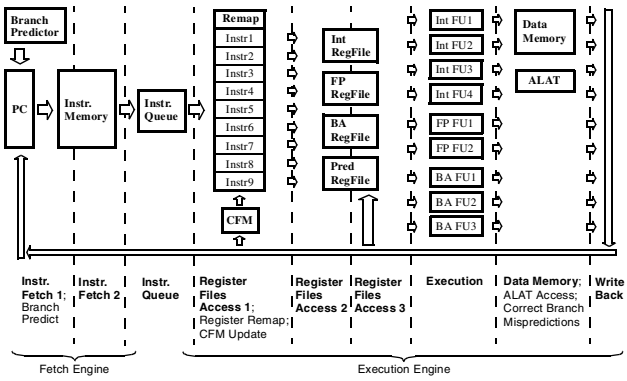

Fig. 1. Block diagram of the VLIW architecture that is formally verified.

A Current Frame Marker register (CFM) [17][30] is used to remap the register identifiers for accessing the register files. The CFM can be modified by every instruction in a packet. Two of the Integer Functional Units can generate addresses for accessing the Data Memory that stores both integer and floating-point values. An Advanced Load Address Table (ALAT) [17][30] is used as hardware support for advanced and speculative loads, which provide mechanisms for compile-time speculation. Both the CFM and the ALAT are architectural state elements. The CFM is updated speculatively in the first Register-Files-Access stage.

4. Results from Monolithic Formal Verification

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 physical memory, and running Red Hat Linux 9.0. The term-level symbolic simulator TLSim [41] was used to generate the EUFM correctness formulas for safety of the models. The decision procedure EVC [41] was applied with the e_{ij} encoding [7] of g-equations, where a new Boolean variable is introduced for each unique low-level g-equation between term variables; the property of transitivity of equality was enforced as described in [4]. Translation of the Boolean correctness formulas to CNF was done as described in [38][39][40]. The SAT-solvers siege_v4 [28] and BerkMin621 [9] were used for the experiments. The abstraction function was computed with controlled flushing [6] for all of the designs.

The first benchmark, 9×VLIW-8stages, did not have an instruction queue. The monolithic formal verification of this design took a total of 496 seconds—see Table 1. This processor was then extended with instruction queues of between 1 and 6 entries—the models are designated with suffixes -IQ1, ..., -IQ6, respectively. The resulting designs had 9 pipeline stages. The monolithic formal verification of the models with 1 to 5 instruction-queue entries took, respectively, 3,163 seconds, 6,152 seconds, 10,927 seconds, 18,415 seconds, and 29,328 seconds (8 hours and 9 minutes). Both SAT-solvers ran out of memory on the CNF from the model with a 6-entry instruction queue.

Table 1. Statistics from monolithic formal verification of safety.

Processor	CNF		Formal Verification Time [sec]			
	Vars	Clauses	TLSim	EVC	SAT	Total
9×VLIW-8stages	43,329	1,033,915	0.18	7.9	488[a]	496
9×VLIW-9stages-IQ1	55,676	1,513,618	0.23	11.5	3,151	3,163
9×VLIW-9stages-IQ2	73,883	2,418,165	0.24	14.9	6,137	6,152
9×VLIW-9stages-IQ3	95,272	3,628,526	0.29	26.5	10,900	10,927
9×VLIW-9stages-IQ4	143,677	6,383,040	0.36	50.4	18,364	18,415
9×VLIW-9stages-IQ5	148,693	7,245,211	0.37	55.7	29,272	29,328
9×VLIW-9stages-IQ6	181,045	9,782,015	0.43	73.2	> mem.	——

a. The SAT time for the first benchmark is from the SAT-solver siege_v4 [28], which was faster than BerkMin621 [9] on that CNF formula; the SAT times for the other benchmarks are from the SAT-solver BerkMin621, which was faster than siege_v4 on those CNFs.

The instruction queues were implemented as shift registers, like in the PowerPC 750 [16]: the oldest instruction is always in the first entry, and other instructions are placed sequentially in subsequent entries. Such instruction queues with 2 or more entries should satisfy the invariant constraint that if an entry is empty then so are all subsequent entries. Checking the invariance of these constraints for the processors having between 2 and 5 instruction queue entries, took between 0.28 and 0.37 seconds, using the SAT-solver siege_v4 and including the time for term-level symbolic simulation with TLSim and for translation of the EUFM correctness formula to CNF by EVC.

5. Using Decomposition and Abstraction to Scale the Formal Verification

This paper advocates the separate formal verification of the Fetch and Execution Engines, after abstracting the other engine with a conservative approximation derived from the high-level specification of that engine. In these conservative approximations, non-determinism is used to abstract any actual implementation of the control logic. To produce non-deterministic values, we can use an FSM, where the initial state is an arbitrary term, the next state is produced by a UF that depends on the present state only, and a UP maps the present state to a new Boolean value. Since the initial state is arbitrary, then so is the first prediction produced through the UP, and so is the next state (produced through the UF) that will be used to determine the next prediction, and so on [33]. The high-level specifications of the Fetch and Execution Engines are provided by the formal verification engineer.

In the case of the Fetch Engine, the high-level specification is the same as the specification used in the monolithic formal verification of the implementation processor, and consists of a latch that contains the current PC value, a read-only Instruction Memory, a UF that maps the PC to the sequential PC value, and logic to update the PC with the target address of a taken branch computed by the Execution Engine. This high-level specification is enriched with a generator of arbitrary values, used to control whether to fetch a new instruction in each clock cycle of regular

symbolic simulation. If the non-deterministic output of this generator, signal ND_Complete, is *false*, then the PC is not incremented, and signal Valid that is supplied to the Execution Engine becomes *false*. In processors with branch prediction, the FSM abstraction of the Fetch Engine is further enriched with a branch predictor, which is itself abstracted with a generator of arbitrary values, producing an arbitrary term for the predicted target, and an arbitrary formula for the predicted direction. Signal Flush, when asserted to *true*, controls the flushing of the implementation; signal Flush is *false* during regular symbolic simulation.

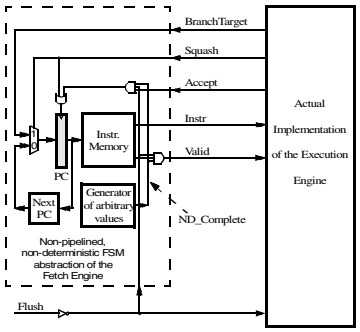

Fig. 2. Abstraction of the Fetch Engine when formally verifying the Execution Engine. The Fetch Engine is abstracted with its high-level specification, after that specification is enriched with a generator of arbitrary values that determines whether to complete the fetching of a new instruction in each cycle. During regular symbolic simulation, when signal Flush is *false*, signal Valid will have a non-deterministic value, formed as the conjunction of the valid bit from the Instruction Memory and a non-deterministic signal produced by the generator of arbitrary values.

Since signal ND_Complete is non-deterministic, then so is signal Valid that is supplied to the actual implementation of the Execution Engine. Then, during the one implementation step needed to prove safety, the actual implementation of the Execution Engine will be supplied with a new symbolic instruction non-deterministically. This behavior of the abstraction of the Fetch Engine covers any actual implementation of this engine.

In the case of the Execution Engine, the high-level specification is an FSM where the entire state of the Execution Engine is abstracted with a single term, EX_State; the next state is computed with UF NextState that has as arguments the present FSM state and the new symbolic instruction supplied by the Fetch Engine; the Target address of branch instructions is computed by a UF, and the taken outcome of the branch (i.e., signal Squash) by a UP, both of which also depend on the present FSM state and the new symbolic instruction supplied by the Fetch Engine. The decision whether to accept a new symbolic instruction from the Fetch Engine is made non-deterministically, as determined by a generator of arbitrary values. A branch is taken only if the new symbolic instruction supplied by the Fetch Engine is valid (signal Valid is *true*) and is accepted. For processors with branch prediction, the FSM abstraction of the Execution Engine is further enriched with a mechanism to detect branch mispredictions [33].

Since signal ND_Accept_bar is non-deterministic, then so is signal Accept that is supplied to the actual implementation of the Fetch Engine. Only valid and accepted instructions update the state of the FSM abstracting the Execution Engine. Hence, this FSM records the sequence of symbolic instructions supplied by the Fetch Engine. In the specification used for formal verification of the Fetch Engine, the Execution Engine is replaced by a deterministic version of the same FSM, i.e., one that always accepts a new instruction. If the Fetch Engine is formally verified with this FSM abstraction of the Execution Engine, then the Fetch Engine will be correct for any actual implementation of this FSM, including the actual Execution Engine where the present state is contained in many memories and latches, and the next state is computed by many UFs, UPs, and actual control logic.

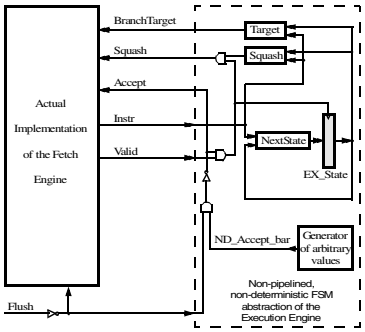

Fig. 3. Abstraction of the Execution Engine when formally verifying the Fetch Engine. The Execution Engine is abstracted with its high-level specification—where the entire state is abstracted with a single term, EX_State, that is updated only by valid instructions that are accepted—after that specification is enriched with a generator of arbitrary values that determines whether to accept a new instruction in each cycle.

6. Results from Using Decomposition and Abstraction for the Formal Verification

Table 2 summarizes the results from decomposition of the processor into Fetch and Execution Engines, with each of them formally verified after the other is abstracted as discussed in Sect. 5.

Table 2. Statistics from formal verification of safety by using decomposition of the processor into Fetch and Execution Engines, with each of them formally verified after the other engine is abstracted.

Model	CNF		Formal Verification Time [sec]			
	Vars	Clauses	TLSim	EVC	SAT	Total
Execution-Engine	28,109	511,139	0.15	4.54	210	215
Fetch-Engine-IQ1	81	304	0.01	0.01	0.01	0.03
Fetch-Engine-IQ2	177	743	0.01	0.03	0.01	0.05
Fetch-Engine-IQ3	281	1,285	0.02	0.03	0.01	0.06
Fetch-Engine-IQ4	397	1,985	0.02	0.03	0.01	0.06
Fetch-Engine-IQ5	539	2,946	0.02	0.05	0.02	0.09
Fetch-Engine-IQ6	702	4,171	0.03	0.07	0.02	0.12
Fetch-Engine-IQ8	1,091	7,545	0.04	0.10	0.04	0.18
Fetch-Engine-IQ16	3,487	38,201	0.11	0.45	0.22	0.78
Fetch-Engine-IQ32	11,404	267,510	0.35	3.64	2.13	6.12
Fetch-Engine-IQ64	46,087	1,656,185	1.30	34.80	28.33	64.43
Fetch-Engine-IQ128	——	——	5.22	>mem.	——	——

The Execution Engine was formally verified in a total of 215 seconds. Models of the Fetch Engine with 1 to 64 entries in the instruction queue were formally verified after 0.03 to 64.43 seconds, respectively. (The decision procedure EVC ran out of memory for a variant of the Fetch Engine with 128 entries in the instruction queue.) Hence, the formal verification of a processor with a 5-entry instruction queue takes 215 seconds—the sum of the formal verification times for the Execution Engine and the Fetch Engine with a 5-entry instruction queue. That is, the presented method results in 2 orders of magnitude speedup for the processor with a 5-entry instruction queue, compared to its monolithic formal verification (see Table 1). Furthermore, the speedup is increasing for processors with longer instruction queues. Most importantly, the presented method made possible the formal verification of VLIW models with 64 entries in the instruction queues.

7. Related Work

In order to formally verify processors with long pipelines, researchers have proved the correctness of pipelining transformations [2][14][15][18][26], but invested months of manual work or reported that their tools did not scale for complex designs. The reverse approach is to use unpipelining transformations [10][11][20][21][22]. However, any set of such transformations would require modifications to work on designs defined in a different coding style, and significant extensions to scale for wide-issue processors.

McMillan [24] used circular compositional reasoning to formally verify an out-of-order processor. The design was decomposed into modules, and each of them was formally verified separately, after its inputs were restricted appropriately by the specifications of the other modules. Henzinger et al. [13] used a similar approach, called assume-guarantee reasoning, but had to invest extensive manual effort to formally verify even a 3-stage pipeline with ALU and move instructions. Sawada and Hunt [29] used theorem proving to formally verify an out-of-order model with a 4-entry instruction queue, but had to define over 4,000 lemmas.

8. Conclusions

Presented was a method for efficient formal verification of pipelined processors with long instruction queues. The execution engine and the fetch engine (where the instruction queue is) were formally verified separately, after abstracting the other engine with a non-deterministic FSM derived from the high-level specification of that engine. Without the presented method, the monolithic formal verification of 9-stage, 9-wide VLIW processors with many realistic and speculative features scaled for models with 5 instruction-queue entries, but ran out of memory for designs with longer instruction queues. The presented method resulted in 2 orders of magnitude speedup when formally verifying the processor with 5 instruction-queue entries, and enabled scaling for designs with 64 instruction-queue entries.

References

[1] M.D. Aagaard, B. Cook, N.A. Day, R.B. Jones, "A Framework for Superscalar Microprocessor Correctness Statements," Software Tools for Technology Transfer (STTT), Vol. 4, No. 3 (May 2003).

[2] E. Börger, and S. Mazzanti, "A Practical Method for Rigourously Controllable Hardware Design," *Int'l Conference of Z Users (ZUM '97)*, LNCS 1212, Springer-Verlag, April 1997, pp. 151–187.

[3] 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.

[4] 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.

[5] J.R. Burch, and D.L. Dill, "Automated Verification of Pipelined Microprocessor Control," *Computer-Aided Verification*, June 1994.

[6] J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *33rd Design Automation Conference (DAC '96)*, June 1996.

[7] 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)*, LNCS 1427, June 1998.

[8] E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver," *Design, Automation, and Test in Europe*, March 2002.

[9] E. Goldberg, and Y. Novikov, SAT-Solver BerkMin621, June 2003.

[10] N.A. Harman, "Verifying a Simple Pipelined Microprocessor Using Maude," *Workshop on Recent Trends in Algebraic Development Techniques (WADT '01)*, LNCS 2267, Springer-Verlag, April 2001.

[11] J.D. Hauke, and J.P. Hayes, "Microprocessor Design Verification Using Reverse Engineering," *High Level Design Validation and Test Workshop (HLDVT '99)*, 1999.

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

[13] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Decomposing Refinement Proofs Using Assume-Guarantee Reasoning," *International Conference on Computer-Aided Design (ICCAD '00)*, 2000.

[14] H. Hinrichsen, H. Eveking, and G. Ritter, "Formal Synthesis for Pipeline Design," *Discrete Mathematics and Theoretical Computer Science (DMTCS '99)*, Springer-Verlag, 1999.

[15] J.K. Huggins, and D. Van Campenhout, "Specification and Verification of Pipelining in the ARM2 RISC Microprocessor," ACM Transactions on Design Automation of Electronic Systems, Vol. 3, No. 4 (October 1998), pp. 563–580.

[16] IBM Corporation, *PowerPC 740™/PowerPC 750™: RISC Microprocessor User's Manual*, 1999.

[17] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999.

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

[19] S. Lahiri, C. Pixley, and K. Albin, "Experience with Term Level Modeling and Verification of the M•CORE™ Microprocessor Core," *High Level Design, Validation and Test (HLDVT '01)*, 2001.

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

[21] M.N. Lis, "Superscalar Processors via Automatic Microarchitecture Transformations," M.S. Thesis, EECS, M.I.T., June 2000.

[22] J. Matthews, and J. Launchbury, "Elementary Microarchitecture Algebra," *Computer-Aided Verification (CAV '99)*, July 1999.

[23] H. McGhan, and M. O'Connor, "PicoJava: A Dyrect Execution Engine foe Java Bytecodes," IEEE Computer, October 1998.

[24] 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.

[25] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *DAC '01*, June 2001.

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

[27] 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).

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

[29] 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), Vol. 20, No. 2 (March 2002).

[30] H. Sharangpani, and K. Arora, "Itanium Processor Microarchitecture," IEEE Micro, September–October 2000, pp. 24–43.

[31] M.N. Velev, and R.E. Bryant, "Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors," *Design Automation Conference (DAC '99)*, June 1999.

[32] 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*, LNCS 1703, Springer-Verlag, 1999.

[33] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction," *Design Automation Conference*, June 2000.

[34] M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution," *Computer-Aided Verification*, July 2000.

[35] M.N. Velev, "Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors," *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2031, 2001.

[36] 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.

[37] 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.

[38] M.N. Velev, "Using Automatic Case Splits and Efficient CNF Translation to Guide a SAT-Solver When Formally Verifying Out-of-Order Processors," *Artificial Intelligence and Mathematics (AI&MATH '04)*, January 2004, pp. 242–254.

[39] M.N. Velev, "Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors," *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, January 2004.

[40] M.N. Velev, "Exploiting Signal Unobservability for Efficient Translation to CNF in Formal Verification of Microprocessors," *Design, Automation and Test in Europe (DATE '04)*, February 2004.

[41] M.N. Velev, and R.E. Bryant, "TLSim and EVC: A Term-Level Symbolic Simulator and an Efficient Decision Procedure for the Logic of Equality with Uninterpreted Functions and Memories," International Journal of Embedded Systems (IJES), 2004.

[42] M.N. Velev, "Integrating Formal Verification into an Advanced Computer Architecture Course," IEEE Transactions on Education, 2004.