Microprocessor Hazard Analysis via Formal Verification of Parameterized Systems*

Lukáš Charvát, Aleš Smrčka, and Tomáš Vojnar

Brno University of Technology, FIT, IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, Czech Republic
{icharvat,smrcka,vojnar}@fit.vutbr.cz

Abstract. The current stress on having a rapid development cycle for microprocessors featuring pipeline-based execution leads to a high demand of automated techniques supporting the design, including a support for its verification. We present an automated technique exploiting static analysis of data paths and formal verification of parameterized systems in order to discover flaws caused by improperly handled data hazards. In particular, as a complement of our previous work on read-after-write hazards, we focus on write-after-write and write-after-read hazards in microprocessors with a single pipeline.

1 Introduction

Implementation of pipeline-based execution of instructions in purpose-specific microprocessors is an error prone task, which implies a need of proper verification of the resulting designs. Therefore, our long-term goal is to develop a set of verification techniques with formal roots, each of them specialised in checking absence of a certain kind of errors in pipeline-based execution of such microprocessors. The main idea is that, this way, a high degree of automation and scalability can be achieved since only parts of a design related to a specific error are to be investigated. In our previous works [4, 5], we proposed, with the above goal in mind, fully automated approaches for checking correctness of the implementation of instructions when executing in isolation and for verifying absence of read-after-write (RAW) hazards. In this paper, we extend our approach to handle also *write-after-write (WAW)* and *write-after-read (WAR)* hazards in microprocessors with a single pipeline. We have implemented our approach and present encouraging results from its experimental evaluations.

Related Work Showing absence of data hazards is a native part of checking conformance between an RTL design and a formally encoded ISA description. The perhaps most cited approach to such checking is the so-called flushing technique [3], which has been extended, e.g., in [9, 13, 7], to handle rather complicated designs with multi-cycle execution units, exceptions, and branch prediction. The main challenge of these works is to overcome the semantic gap between the different levels of a processor description. Dealing with this issue typically requires a significant user intervention in the form of providing various additional assertions about the design or transforming it to a purpose-specific description language.

In [8], the so-called self-consistency check that compares possible executions of each instruction in two scenarios is introduced. The comparison is made wrt. a property

* This work was supported by the Czech Science Foundation under the project 14-11384S, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal BUT project FIT-S-14-2486.

given by the user, e.g., a property concerning data hazards which deals with (i) executions of an instruction enclosed by any (random) instructions within the pipeline and (ii) executions of the same instruction surrounded by `NOP` instructions only. If the self-consistency check succeeds, conformance of the RTL and ISA descriptions of a processor can be established by separately showing conformance of the RTL/ISA descriptions of each individual instruction. The main drawback of the approach is that it requires the enclosing instructions from the first run not to violate a so-called consistent state of the microprocessor, which has to be manually defined by the user.

In [1], a formal model based on a notion of stages, parcels (instructions), and hazards has been introduced. Once the user defines predicates needed for describing the pipeline, the design can be automatically formally proven correct under a correctness criterion given in the work. Another, a bit similar approach has been proposed in [10]. The approach introduces an abstract formal model whose components are to be linked by the user with the concrete cycle-accurate implementation through a number of mappings. Afterwards, the validity of several properties based on the established mappings and together implying correctness of the pipeline behaviour is checked. Again, both of the above methods require a significant manual user intervention.

Compared with the above approaches, we do not aim at full conformance checking between RTL and ISA implementations. Instead, we address one specific property—namely, absence of problems caused by data hazards. On the other hand, our approach is almost fully automated—the only step required from the user is to identify the architectural resources (such as registers and memory ports) and the program counter.

2 Preliminaries

Our approach expects a processor to be described in the form of a so-called *processor structure graph* (PSG) which can be represented by a tuple $G = (V, E, s, t)$. Here, V is a finite set that is the union $V_s \cup V_f$ of a set V_s of *storages* and a set V_f of *Boolean circuits*, $V_s \cap V_f = \emptyset$. The set V_s further consists of a set V_a of *architectural* and a set V_p of *pipeline* storages, $V_a \cap V_p = \emptyset$. To simplify the explanation, we will not deal with micro-architectural registers, memories and their ports in this paper, however as we show in [5], designs including these entities can be easily verified by the proposed approach as well. Without a loss of generality, we can also expect all storages of the set V_s to have a unit write and zero read delay since longer access times can be modelled by introducing sequentially connected registers emulating the required delay. The set V_f of Boolean circuits is the union $V_{mx} \cup V_g$ of a set V_{mx} of circuits implementing *multiplexers* and a set V_g of the remaining (generic) circuits, $V_{mx} \cap V_g = \emptyset$.

Next, E denotes a finite set of *transfer edges*. Then, mappings $s, t : E \to V \times \mathbb{T}$ assign to each edge its source (resp., target) vertex where $\mathbb{T} = \{\mathtt{d}, \mathtt{q}, \mathtt{en}, \mathtt{st}, \mathtt{cl}, \mathtt{sel}\} \cup \{\mathtt{a}_i, \mathtt{c}_i \mid i \in \mathbb{N}\}$ is a set of *connection types*. It is required that a PSG contains no cycle formed only by vertices representing Boolean circuits. The `d`, `q`, and `en` connection types represent commonly used input, output, and enable connections of flip-flop registers with their usual semantics. Pipeline registers do also have `st` (stall) and `cl` (clear) connections. In case of stalling, each stalled register keeps its current value to the next cycle. Clearing a register sets its value to zero. The $\mathtt{a}_i$ connection types represent arguments of functional vertices $v_g \in V_g$. Further, `sel` and $\mathtt{c}_i$ are connection types related to multiplexers only. The value transferred through the `sel` connection selects which of its $\mathtt{c}_i$ inputs is propagated to the `q` output of a multiplexer. Since each vertex $v \in V$ can have at most one inbound edge for a single connection type, one can use a notation $v.\mathtt{c}$ to uniquely describe an edge $e \in E$ that satisfies $t(e) = (v, \mathtt{c})$.

In this paper, we will work with an annotated version of a PSG. The annotation can be given via a *stage* mapping $\varphi\colon V_s \to \mathbb{S}$, $\mathbb{S} = \{0, ..., n\}$, $n \in \mathbb{N}$, assigning storages to pipeline stages. The annotation can be given manually or techniques such as data-flow analysis [5] can be used to obtain one. From a stage mapping φ, we can easily get the *write stage* φ_{wr} (*read stage* φ_{rd}, respectively) mapping, $\varphi_{wr}, \varphi_{rd}\colon V_s \to 2^{\mathbb{S}}$, describing which stages directly influence (use) the content of the given storage.

Our approach further uses the common notion of a parameterized system operating on a linear topology where processes (i.e., executed instructions) may perform local transitions or universally/existentially guarded global transitions [6, 2]. A parameterized system is a pair $P = (Q, \Delta)$ where Q is a finite set of states of a process and Δ is a set of transition rules over Q. A transition rule is of the form $\mathbb{Q} j \circ i\colon G \models q \to q'$ where $\mathbb{Q} \in \{\forall, \exists\}$, $\circ \in \{<, >, =\}$, $G \subseteq Q$, and $q, q' \in Q$. A parameterized system induces a transition system whose configurations are finite words over Q. A configuration $q_1...q_i...q_n$, $1 \leq i \leq n$, changes to $q_1...q_i'...q_n$ when the ith process goes from its state q_i to q_i' using some of the transition rules. The rule can be applied only if its guard is satisfied. For example, the meaning of the guard $\exists j < i\colon G$ is "there should be at least one process j to the left of i (in the linear topology) so that the jth process is in a state that belongs to the set G".

We will work with the reachability problem given by a parameterized system P, a regular set $I \subseteq Q^+$ of initial configurations, and a regular set $Bad \subseteq Q^+$ of bad configurations. In particular, we assume Bad to be given as the upward closure of a finite set $B \subseteq Q^+$ of minimal bad configurations, this is, $Bad = \{c \in Q^+ \mid \exists b \in B\colon b \sqsubseteq c\}$ where $\sqsubseteq$ is the usual sub-word relation (i.e., $u \sqsubseteq s_1...s_n \Leftrightarrow u = s_{i_1}...s_{i_k}$ for some $1 \leq i_1 \leq ... \leq i_k \leq n$, $0 \leq k \leq n$). Now, let $R \subseteq Q^*$ denote the set of all reachable configurations. We say that the system P is safe wrt. I and Bad iff no bad configuration is reachable, i.e., $R \cap Bad = \emptyset$.

3 Description of The Proposed Data Hazard Verification Method

We assume the processor under verification to be represented using a PSG, which can be easily obtained from a description of the processor on the register transfer level (RTL) written in common hardware description languages, such as VHDL or Verilog.

Our approach consists of the following steps: (i) a static detection of instructions that can potentially cause a data hazard, (ii) generation of a parameterized system modelling mutual interaction among the instructions, and (iii) an analysis of the constructed parameterized system identifying whether some unhandled data hazard may occur.

Example 1. Fig. 1 shows a PSG describing a part of a simple microprocessor with an accumulator architecture with two architectural registers: X (a memory index register) and A (an accumulator). For the sake of brevity, the PSG exhibits only the parts of the microprocessor that are used during execution of arithmetic and instructions with an auto-increment. Moreover, it also omits control connections (`en`, `st`, and `cl`) of pipeline

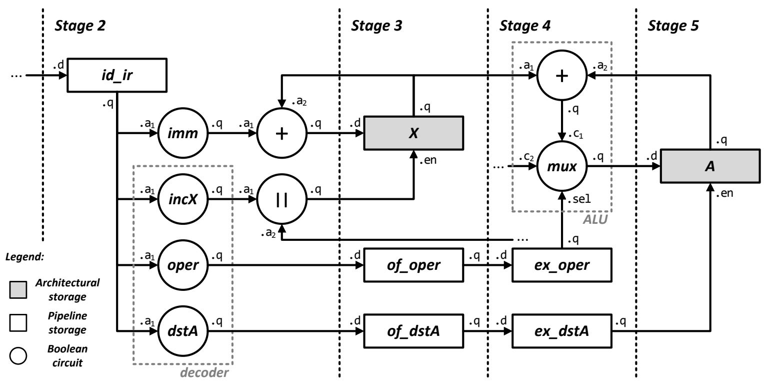

Fig. 1: A processor structure graph of a part of a CPU with an accumulator architecture.

registers. In the CPU, an instruction fetched from the memory is stored into the storage id_ir representing the instruction register. The opcode part is sent to the decoder to determine the type of the ALU operation to be performed and to select its destination by activation of the appropriate enable (`en`) connection of the X or A register. An early auto-increment of register X can be performed in stage 3. Such a feature allows the CPU to execute sequences of instructions working with juxtaposed data in the memory without a penalty (brought, e.g., by unnecessary stalls of the pipeline) which would be present if the update of X was done in a later stage. $\triangleleft$

3.1 Static Detection of Data Hazards

A static hazard analysis examines the PSG and its annotation in the form of the pipeline stage mappings φ, φ_{wr}, and φ_{rd} to identify a finite set of so-called *hazard cases*, each of them describing one potential source of a data hazard. In order to construct the hazard cases, we will use a notion of an *influence path*.

We define an *influence path* as a path $\langle v_1, e_1, ..., v_k \rangle$ in a PSG where the value read from an architectural storage $v_1 \in V_a$ can influence a value stored to an architectural storage $v_a \in V_a$ by writing to a *target* storage $v_k \in V_s$. Each influence path must fulfill the following set of properties: (i) The target storage v_k must either be (a) an architectural register, i.e., the case when $v_k = v_a$, or (b) a pipeline register s.t. $t(e_{k-1}) = (v_k, \texttt{cl})$. Indeed, clearing of the pipeline register v_k will surely influence all programmer visible storages that belong to stages $s \geq \varphi(v_k)$. Next, (ii) the influence path must not traverse through stall connections of pipeline registers. Such paths cannot influence the value of any programmer visible register. Their only impact can be stalling a stage which does not influence a proper execution of instructions if one assumes correctness of in-order execution of instructions (that can be automatically checked by the method described in [11]). Finally, (iii) there must exist an *execution plan* $\tau \colon V \to \mathbb{S}$ which assigns elements of the path to stages from which they are accessed by an instruction that performs a computation over the given influence path.

The access stage of each element that is given by the execution plan has to conform to φ_{rd} and φ_{wr}, i.e., (a) $v_i \in V_s \Rightarrow \tau(v_i) + 1 \in \varphi_{rd}(v_i)$ for all $1 \leq i < k$ and (b) $v_j \in V_s \Rightarrow \tau(v_j) \in \varphi_{wr}(v_j)$ for all $1 < j \leq k$. Moreover, the stages of the execution plan must form a non-decreasing sequence, i.e., (c) $\tau(v_{i-1}) \leq \tau(v_i)$ which increases at each path element with a write delay, i.e., (d) $\tau(v_i) = \tau(v_{i-1}) + 1$ if $v_i \in V_s$. Otherwise, in the case that any of the rules (a–d) fails, there could not be any instruction capable of a data transfer along the influence path.

An incorrectly handled data hazard is manifested upon the first write of improper data into some programmer visible storage of the design. Therefore, it suffices to further deal with the minimal influence path which is an influence path where $v_i \notin V_a$ and $t(e_{i-1}) \notin V_p \times \{\texttt{cl}\}$ for all $1 < i < k$. A standard breadth-first search algorithm with rules (i–iii) and the minimality checked on-the-fly can be used to obtain the minimal influence paths in the given PSG.

Table 1: The access stage mappings for architectural registers.

Register	Stage	Write stages	Read stages
	φ	φ_{wr}	φ_{rd}
X	3	$\{2, 4\}$	$\{2, 4\}$
A	5	$\{4\}$	$\{4\}$

A *WAR hazard case* is a tuple $(v_a, s_w, s_r, v_t, s_t, \pi)$ consisting of (i) an architectural storage $v_a \in V_a$, (ii) its write stage $s_w \in \varphi_{wr}(v_w)$, and (iv) read stage $s_r \in \varphi_{rd}(v_a)$ such that $s_w < s_r$ in order that the storage is written before it is read to evoke a WAR hazard, (v) a target storage v_t where the potentially incorrect value read from v_a is stored, (vi) a stage $s_t \in \varphi_{wr}(v_t)$, $s_r \leq s_t$, in which the incorrect value is stored, and (vii) a minimal influence path π describing how data are propagated from v_a to v_t between the stages s_r and s_t. Similarly, a *WAW hazard case* (v_a, s_{w_1}, s_{w_2}) consists of an architectural storage $v_a \in V_a$ and its two different write stages $s_{w_1}, s_{w_2} \in \varphi_{wr}(v_w)$, $s_{w_2} < s_{w_1}$ so the WAW hazard may occur. There is no need to include any influence path in this case since an error in WAW hazard case handling would be demonstrated instantly by writing an incorrect value to the storage v_a. Note that, since the definitions of a hazard cases speak about storages, their access stages, and the path along which the problematic data is transferred, it is not related to a single instruction only but to an entire class of instructions.

Example 2. Consider the PSG from Fig. 1 and the mappings shown in Table 1. One can see that there is a potential WAR hazard on register X because, for example, it can be written in stage 2 ($\varphi_{wr}(X) = \{2, 4\}$) and read in stage 4 ($\varphi_{rd}(X) = \{2, 4\}$). By the definition, to form a WAR hazard, there must also exist an influence path π in the PSG leading from X to some target storage. For instance, we can assume the register A (written in stage 4) as a target with $\pi = \langle X, +.\mathtt{a}_1, +, mux.\mathtt{c}_1, mux, A.\mathtt{d}, A\rangle$. This observation gives us a WAR hazard case $hc = (X, 2, 4, A, 4, \pi)$. A similar reasoning can applied to derive WAW hazard cases as well. ◁

3.2 Construction of Parameterized Systems Modelling the Potential Hazards

As we have shown in [5], the behaviour of the instructions given by constraints of a hazard case can be modelled using a parameterized system $P = (Q, \Delta)$ which maps n instructions in the pipeline to n processes in a linear array. Initially, they are in a state saying that their execution has not started. Then, they proceed through individual stages of the pipeline during which they may interact with each other by means of the pipeline flow logic, e.g., an earlier instruction may force a later instruction to be stalled or cleared. Finally, the instructions end up in a state denoting that they left the pipeline. The structure of the generated parameterized system depends on the type of the hazard case. The system P models interactions among three classes of processes (and hence 3 types of instructions) for both WAR and WAW hazard case. For a WAR hazard case $(v_a, s_w, s_r, v_t, s_t, \pi)$, a w-class of processes is used to model every instruction that writes to the storage v_a in the stage s_w. An rw-class models instructions that read from the storage v_a in the stage s_r, perform a data computation that involves the data path π, and write to the storage v_t in the stage s_t. Finally, any-class instructions are used as pipeline fillers representing *any* other instructions. For a WAW hazard case (v_a, s_{w_1}, s_{w_2}), we use processes of the w_1-class (w_2-class, respectively) which model instructions writing to the storage v_a in the stage s_{w_1} (s_{w_2}). The purpose of the any-class instructions remains the same as in the previous case.

The set Q of states of a parameterized system P is then given by pairs (k, s) where k gives a class of an instruction and s gives the stage in which the instruction is currently executed. We will use the notation q^k_s to denote a state $(k, s) \in Q$. For a pipeline of length m, the sequence $q^k_1, ..., q^k_m$ records each step of a k-class instruction in the pipeline. Transition rules Δ of a system P are then constructed by reasoning over constraints given in the from of bit-vector logic formulae. These formulae describe behaviour in each state of the execution of a k-class instruction. The required reasoning is done automatically by utilizing an SMT solver (for additional technical details regarding the construction of Δ, please see [5]). Such a system is then checked whether there exists some sequence of instructions that could reach hazardous conditions. In parameterized systems, hazard conditions can in particular be expressed by the regular set Bad of bad configurations. The most crucial part for the construction of the Bad set is determination of the so-called *commit* and *hazardous states*, which is discussed below.

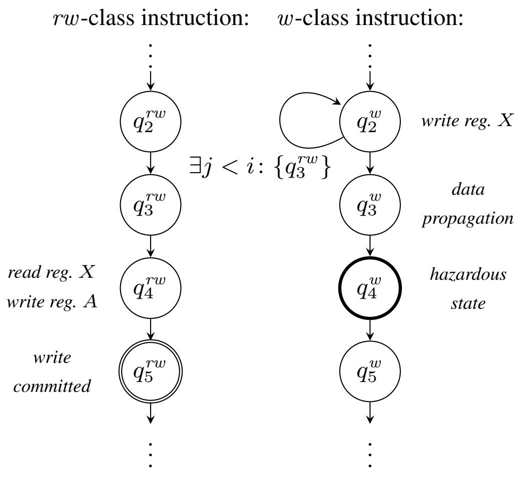

Fig. 2: A part of the control automata of processes representing rw/w-class instructions involved in the hazard case hc from Example 2.

Given a WAR hazard case $(v_a, s_w, s_r, v_t, s_t, \pi)$, $s_w < s_r \leq s_t$, one can infer that the data supposed to be written to v_a are computed in the stage s_w, and the computed value is committed to v_a in the next cycle, thus in the stage $s_w + 1$. To ensure that the value read in stage the s_r is correct, no write to v_a can occur for $h = s_r - (s_w + 1)$ cycles which is the difference between reading and commitment of the value from/to v_a. Otherwise, an rw-class instruction would necessarily read and compute with incorrect data that were written too early (in stage s_w) by a later w-class instruction. The WAR hazard is exhibited only after commitment of the incorrectly fetched data from the register v_a in the stage s_r to the register v_t which happens in the stage $s_t + 1$. Such a data propagation lasts $p = (s_t + 1) - s_r$ cycles. Note that, if the rw-class instruction is canceled during the propagation period of p, there is no further write to v_t caused by the instruction. Thus, for a w-class instruction, we denote the states $\{q^w_{s_w+p+i} \mid 1 \leq i \leq h\}$ as hazardous. A configuration of the parameterized system P is then considered as bad if it includes an occurrence of a commit state $q^{rw}_{s_t+1}$ of an rw-class instruction followed by a hazard state.

An analogical reasoning can be performed also for a WAW hazard case $(v_a,\, s_{w_1},\, s_{w_2})$, $s_{w_2} < s_{w_1}$. Here, no write to v_a can occur for $h = s_{w_2} - s_{w_1}$ cycles. Otherwise, the execution of an earlier w_1-class instruction would overwrite the value storage v_a that was already set by a later w_2-class instruction. Therefore, we tag the states $\{q^{w_2}_{s_{w_2}+i} \mid 1 \leq i \leq h\}$ as hazardous. Finally, we include a configuration into Bad if it contains a commit state $q^{w_1}_{s_{w_1}+1}$ of a w_1-class instruction followed by a hazard state.

Example 3. Consider the hazard case hc described in Example 2 and the inferred processes shown in Fig. 2. The execution of the rw-class instructions reading X and writing to A is passing through the sequence of states $q^{rw}_0, q^{rw}_1, q^{rw}_2, q^{rw}_3, q^{rw}_4, q^{rw}_5, q^{rw}_6$. Here, X is read and A written in the state q^{rw}_4. Because the value of A is committed in the state q^{rw}_5 (in stage 5) and X is read in the state q^{rw}_4 (in stage 4) the length of the data propagation p is $5 - 4 = 1$. The execution of a w-class instruction writing to the X register is described by a process going through the sequence of states $q^w_0, q^w_1, q^w_2, q^w_3, q^w_4, q^w_5, q^w_6$ where X is written in the state q^w_2 and q_0, q_6 denote initial, resp. final, state. Because a w-class instruction commits the value to X in stage 3, the distance h (between reading and commitment from/to X) is $4-3 = 1$. Thus, the set of minimal bad configurations is $\{q^{rw}_5 q^w_4\}$. A chosen parametric verification method can then be used to check whether a bad configuration, e.g., $q^{any}_6 q^{rw}_5 q^w_4 q^{any}_3 q^{any}_2 q^{any}_1 q^{any}_0$, is reachable. $\triangleleft$

4 Experimental Evaluation

We have implemented the above described method in a prototype tool called *Hades* and tested it on three kinds of processors: *TinyCPU* is a small 8-bit processor that we mainly use for testing of new verification methods. *CompAcc* is an 8-bit processor based on an accumulator architecture. Finally, *DLX5AI* is a 5-staged 32-bit processor able to execute a subset of the instruction set (without floating point instructions) of the DLX architecture which differs from commonly known implementation [12] by having an auto-increment logic. Some of the processors were in multiple variants that differ from each other, e.g., in the way how data hazards are avoided, yielding seven test cases in total.

Table 2: Verification times.

Processor / features		Static Analysis [s]	Parametric model verification [s]	Total time [s]	Hazard Cases [#]
TinyCPU	S	1	24	25	14
	SF	1	25	26	14
	B	1	38	39	24
CompAcc	SF	2	70	72	31
	BF	2	61	63	33
DLX5AI	S	5	418	423	69
	B	384	420	804	69

S stalling logic B bypassing logic F flag reg.

We conducted a series of experiments on a PC with Intel Core i7-3770K @3.50GHz and 16 GB RAM with results presented in Table 2. The columns give the verified processor, its variant, the time needed for the static analysis, and the time spent by verification of the parameterized systems that are created based on each hazard case, and the overall verification time. The last column represents the number of hazard cases that had to be verified during the model verification phase. Note that each hazard case represents a separate task so the part of model verification can be run in parallel. As can be seen, the results look promising in that the verification times are in minutes for all types of the presented microprocessors. The longer time of static analysis encountered for DLX is mainly due to the larger number of paths that have to be considered (by the BFS algorithm) during the computation of the sets of hazard cases.

5 Conclusion

We have presented an approach that harnesses methods for formal verification of parameterized systems in order to discover incorrectly handled data hazards in the RTL implementation of pipeline-based execution. The approach was developed with the aim to be highly automated, not requiring any additional efforts from the developers (apart from specifying the architectural registers). We have implemented the approach and successfully tested it on several non-trivial microprocessors.

In the future, we plan to further extend the approach presented in the paper by techniques suitable for verification of other processor features, such as control hazards. This is motivated by our general idea of trying to split processor verification into several simpler, more specialised tasks.

References

1. M. D. Aagaard. A hazards-based correctness statement for pipelined circuits. In *Proc. of CHARME'03*, volume 2860 of *LNCS*, pages 66–80. Springer, 2003.
2. P. A. Abdulla, F. Haziza., and L. Holik. All for the price of few. In *Proc. of VMCAI'13*, volume 7737 of *LNCS*, pages 476–495. Springer, 2013.
3. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of CAV'94*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.
4. L. Charvat, A. Smrcka, and T. Vojnar. Automatic formal correspondence checking of ISA and RTL microprocessor description. In *Proc. of MTV'12*, pages 6–12. IEEE, 2012.
5. L. Charvat, A. Smrcka, and T. Vojnar. Using formal verification of parameterized systems in RAW hazard analysis in microprocessors. In *Proc. of MTV'14*, pages 83–89. IEEE, 2014.
6. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 126–141. Springer, 2006.
7. K. Hao, S. Ray, and F. Xie. Equivalence checking for function pipelining in behavioral synthesis. In *Proc. of DATE'14*, pages 1–6. IEEE, 2014.
8. R. B. Jones, C. H. Seger, and D. L. Dill. Self-consistency checking. In *Proc. of FMCAD'96*, volume 1166 of *LNCS*, pages 159–171. Springer, 1996.
9. A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to RTL equivalence checking. In *Proc. of DATE'09*, pages 196–201. IEEE, 2009.
10. U. Kuhne, S. Beyer, J. Bormann, and J. Barstow. Automated formal verification of processors based on architectural models. In *Proc. of FMCAD'10*, pages 129–136. IEEE, 2010.
11. P. Mishra, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *Proc. of DATE'02*, pages 36–43. IEEE, 2002.
12. D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann, Boston, fourth edition, 2012.
13. M. N. Velev and P. Gao. Automatic formal verification of multithreaded pipelined microprocessors. In *Proc. of ICCAD'11*, pages 679–686. IEEE, 2011.