A Complete Compositional Reasoning Framework for the Efficient Verification of Pipelined Machines

Panagiotis Manolios
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30318
Email: manolios@cc.gatech.edu

Sudarshan K. Srinivasan
School of Electrical & Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30318
Email: darshan@ece.gatech.edu

***Abstract*— We present a compositional reasoning framework based on refinement for verifying that pipelined machines satisfy the same safety and liveness properties as their instruction set architectures. Our framework consists of a set of convenient, easily-applicable, and complete compositional proof rules. We show that our framework greatly extends the applicability of decision procedures by verifying a complex, deeply pipelined machine that state-of-the-art tools cannot currently handle. We discuss how our framework can be added to the design cycle and highlight what arguably is the most important benefit of our approach over current methods, that the counterexamples generated are much simpler, as bugs are isolated to a particular step in the composition proof.**

I. Introduction

We present a complete compositional framework based on Well-founded Equivalence Bisimulation (WEB) refinement that can be used to reason about pipelined machines that are too complex to handle with current tools and methods. State-of-the-art tools based exclusively on decision procedures (such as UCLID) fail as processor models exceed the tool's complexity threshold. As a result, the goal of much of the current work in mechanical verification is to design tools and techniques that extend the range of automatic methods. While there has been much success, we are still far away from being able to use such methods to verify industrial designs. The compositional framework we introduce takes us a step closer, as it allows us to substantially extend this complexity threshold.

Using our framework we can verify in under 20 seconds a complex pipelined machine that UCLID cannot directly handle. This machine is quite complicated and to make its definition a manageable process, we defined a series of machines starting with the base processor model `M6`, a 6 stage pipelined machine, which we extended first with a pipelined fetch stage, then with an instruction queue holding up to 3 instructions, then with a direct mapped instruction cache, then a direct mapped data cache, and, finally, a write buffer, to obtain `M10IDW`. Unfortunately, proving that `M10IDW` refines `ISA`, the instruction set architecture, is beyond the capabilities of UCLID. Our compositional framework allows us to verify the machine the same way we defined it, one feature at a time, which leads to a manageable process. Each stage of the proof essentially entails establishing a WEB-refinement proof, which means that, relative to a refinement map and up to stuttering, the two machines have exactly the same infinite behaviors.

We introduce compositional proof rules that guarantee that this sequence of refinement proofs implies that the final pipelined machine has the same behaviors as the instruction set architecture. In terms of temporal logic, we have that the machines satisfy exactly the same $\mathrm{CTL}^* \setminus \mathsf{X}$ properties expressible at the instruction set architecture level. We automate the process by reducing the proof obligations to statements expressible in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU), which is a decidable logic [6]. We use the tool UCLID [14] to transform the CLU formula into a CNF (Conjunctive Normal Form) formula, which we then check with the SAT solver Siege [25].

A major advantage, perhaps even more important than the increased performance, of our compositional framework over monolithic approaches is that counterexamples are shorter and clearer, which greatly simplifies debugging. Suppose that modifications are made to the design and in the process a bug is introduced. Compositional verification allows us to focus in on where the bug first appears and the counterexample generated is with respect to a specific refinement stage, *i.e.*, the counterexample is at exactly the right level of abstraction required to easily understand and correct the problem. For example, if the bug does not involve the cache, then neither does the counterexample, whereas in a monolithic approach, there is no way to know if the cache was involved; thus, as the verification engineer is trying to understand the counterexample, she is forced to manually rule out the possibility that the cache contributed to the error. By using our compositional approach, the engineer can bridge the abstraction gap on her own terms and at a rate that makes sense given available tools and the development process.

The paper is organized as follows. We end this section by briefly reviewing related work. In Section II, we review the theory of refinement upon which our correctness proofs depend. In Section III, we describe the modeling and monolithic verification of the processor models, and in Section IV, we describe the compositional techniques we developed for

This research was funded in part by NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

pipelined machine verification. Everything required to reproduce our results, *e.g.*, machine models, correctness statements, CNF formulas, etc., is available upon request. Conclusions and an outline of future work appear in Section V.

Related Work

Pipelined machine verification is an active area of research. One popular approach involves the use of theorem provers, which have the advantage that the underlying logics are very powerful and expressive, but are also undecidable. Examples of this line of research include the work by Sawada and Hunt, who use an intermediate abstraction called MAETT to verify some safety and liveness properties of complex pipelined machines [26], [27]. Another example of a theorem proving approach is the work by Hosabettu et al., who use the notion of completion functions [11], and the work of Arons [2].

Our main concern, however, is with highly automatic methods. An early and influential paper in this area is due to Burch and Dill, who showed how to automatically compute refinement maps using flushing [7] and gave a decision procedure for the logic of uninterpreted functions with equality and Boolean connectives. The idea with flushing is that a pipelined machine state is related to an instruction set architecture state by completing partially completed instructions without fetching any new instructions. Another refinement map that can be automatically computed is based on the commitment approach [15], [18], where a pipelined machine state is related to an instruction set architecture state by invalidating all the partially executed instructions in the pipeline and rolling back the programmer-visible components so that they correspond with the last committed instruction. There has been recent work on commitment [19], [24], [1], and on the use of refinement maps that partly flush and partly commit [21]. There has also been related work on assume-guarantee reasoning by Henzinger et al. [10].

More directly related to this paper is the work on decision procedures for the CLU logic [6], which is based on previous work on exploiting positive equality [5]. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors [14], and is one of the most powerful, freely available tools for deciding formulas arising from processor verification [9] and which we use to verify the models presented in this paper.

Previous work on decomposing refinement proofs includes the work of Arons and Pnueli [3], who use the PVS theorem prover to verify a machine with speculative instruction execution. They use an inductive proof to show that machines which differ only in the size of the retirement buffer are related; however, due to the complexity of the refinement maps involved, they conclude that a direct approach is far simpler than the inductive one. Jones et al. [12] verify an out-of-order execution unit using incremental flushing. Their approach relates the implementation to an intermediate machine, where the scheduling logic is abstracted, which is then related to the `ISA`. In comparison, we can deal with any refinement map, we have a general theory with a complete rule for relating any number of intermediate machines, and we guarantee that all safety *and* liveness properties are preserved.

The notion of correctness for pipelined machines that we use was first proposed in [15], and is based on WEB-refinement [16]. The first proofs of correctness for pipelined machines based on WEB-refinement were carried out using the ACL2 theorem proving system [13]. The advantage of using a theory of refinement over using the Burch and Dill notion of correctness —even when augmented with a "liveness" criterion— is that the Burch and Dill approach cannot detect deadlock [15], whereas it follows directly from the WEB-refinement approach that deadlock (or any other liveness problem) will be detected. In [18], it is shown how to automatically verify safety *and liveness* properties of pipelined machines using WEB-refinement. Our results extend this work by showing how to use WEB-refinement to automatically prove safety and liveness in a compositional fashion.

Why hasn't something like this been done before? Well, consider carrying out this proof using the standard Burch and Dill notion of correctness. The problem is that, while it is clear how to prove that a pipelined machine refines an instruction set architecture, how does one prove that one pipelined machine refines another? If we use flushing, we have to flush both machines, but then it would be easier to just verify against the instruction set architecture directly. Our main contribution is to show how to do this using state-of-the-art tools for both safety and liveness (the Burch and Dill approach only provides safety [15]), and with the use of any refinement map, not just flushing.

Can we really obtain the benefits of composition without paying a price? Actually, we often have to provide invariants. But, invariants are needed to verify complex designs anyway. For example, to verify a write-through cache, we need the invariant that the valid cache entries are consistent with memory. The invariants we used were straight-forward, requiring a few hours of thought; in contrast, defining the refinement maps can easily take days. If one uses a hierarchical, refinement-based approach to design, then the invariants should be known, as they allow for the separation of concerns that enables different engineers to implement different parts of the system independently. Therefore, composition can fit nicely into the design cycle, which is also compositional. Finally, there seems to be industrial interest in taking a fresh look at refinement-based methodologies.[1]

II. Preliminaries on Refinement

In this section, we review the required background on the theory of refinement used in this paper; for a full account see [16], [17]. Pipelined machine verification is an instance of the refinement problem: given an abstract specification, S, and a concrete specification, I, show that I refines (implements) S. In the context of pipelined machine verification, the idea is to show that `MA`, a machine modeled at the microarchitecture

[1]For example, Greg Spirakis, a vice-president at Intel gave talks at FMCAD 2004 and DATE 2004 describing new work at Intel on higher-level and refinement-based design.

level, a low level description that includes the pipeline, refines ISA, a machine modeled at the instruction set architecture level. A refinement proof is relative to a *refinement map*, r, a function from MA states to ISA states. The refinement map, r, shows us how to view an MA state as an ISA state, *e.g.*, the refinement map has to hide the MA components (such as the pipeline) that do not appear in the ISA.

The ISA and MA machines are arbitrary transition systems (TS). A TS, $\mathcal{M}$, is a triple $\langle S, \dashrightarrow, L\rangle$, consisting of a set of states, S, a left-total transition relation, $\dashrightarrow \subseteq S^2$, and a labeling function L whose domain is S and where $L.s$ (we sometimes use an infix dot to denote function application) corresponds to what is "visible" at state s.

Our notion of refinement is based on the following definition of stuttering bisimulation [4], where by $fp(\sigma, s)$ we mean that σ is a fullpath (infinite path) starting at s, and by $match(B, \sigma, \delta)$ we mean that the fullpaths σ and δ are equivalent sequences up to finite stuttering (repetition of states).

Definition 1: $B \subseteq S \times S$ is a stuttering bisimulation (STB) on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ iff B is an equivalence relation and for all s, w such that sBw:

(Stb1) $\quad L.s = L.w$

(Stb2) $\quad \langle \forall \sigma : fp(\sigma, s) : \langle \exists \delta : fp(\delta, w) : match(B, \sigma, \delta)\rangle\rangle$

Browne, Clarke, and Grumberg have shown that states that are stuttering bisimilar satisfy the same next-time-free temporal logic formulas [4].

Lemma 1: Let B be an STB on $\mathcal{M}$ and let sBw. For any CTL* \ X formula f, $\mathcal{M}, w \models f$ iff $\mathcal{M}, s \models f$.

We note that stuttering bisimulation differs from weak bisimulation [22] in that weak bisimulation allows infinite stuttering. Stuttering is a common phenomenon when comparing systems at different levels of abstraction, *e.g.*, if the pipeline is empty, MA will require several steps to complete an instruction, whereas ISA completes an instruction during every step. Distinguishing between infinite and finite stuttering is important, because (among other things) we want to distinguish deadlock from stutter.

When we say that MA refines ISA, we mean that in the disjoint union ($\uplus$) of the two systems, there is an STB that relates every pair of states w, s such that w is an MA state and $r(w) = s$.

Definition 2: (STB Refinement) Let $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$, and $r : S \to S'$. We say that $\mathcal{M}$ is a STB refinement of $\mathcal{M}'$ with respect to refinement map r, written $\mathcal{M} \approx_r \mathcal{M}'$, if there exists a relation, B, such that $\langle \forall s \in S :: sBr.s\rangle$ and B is an STB on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L}\rangle$, where $\mathcal{L}.s = L'.s$ for s an S' state and $\mathcal{L}.s = L'(r.s)$ otherwise.

A major shortcoming of the above formulation of refinement is that it requires reasoning about infinite paths, something that is difficult to automate [23]. In [16], WEB-refinement, an equivalent formulation is given that requires only local reasoning, involving only MA states, the ISA states they map to under the refinement map, and their successor states. In [18], it is shown how to automate the refinement proofs in the context of pipelined machine verification. The idea is to strengthen, thereby simplifying, the refinement proof obligation; the result is the CLU-expressible formula, where $rank$ is a function that maps states of MA into the natural numbers.

Theorem 1: MA $\approx_r$ ISA if:

$$\begin{aligned}\langle \forall w \in \texttt{MA} \;::\; & s = r.w \;\wedge\; u = \texttt{ISA-step}(s) \;\wedge \\ & v = \texttt{MA-step}(w) \;\wedge\; u \neq r.v \\ \Longrightarrow \quad & s = r.v \;\wedge\; rank.v < rank.w\rangle\end{aligned}$$

In the formula above s and u are ISA states, and w and v are MA states; ISA-step is a function corresponding to stepping the ISA machine once and MA-step is a function corresponding to stepping the MA machine once. It may help to think of the first conjunct of the consequent ($s = r.v$) as the safety component of the proof and the second conjunct $rank.v < rank.w$ as the liveness component.

Note that the notion of WEB refinement is independent of the refinement map used. In this paper, we use the standard flushing refinement map [7], where MA states are mapped to ISA states by executing all partially completed instructions without fetching any new instructions, and then projecting out the ISA visible components. The emphasis of this paper is on exploiting the compositionality of WEB refinement.

Theorem 2: (Composition) If $\mathcal{M} \approx_r \mathcal{M}'$ and $\mathcal{M}' \approx_q \mathcal{M}''$ then $\mathcal{M} \approx_{r;q} \mathcal{M}''$.

Above, $r; q$ denotes composition, *i.e.*, $(r; q)(s) = q(r.s)$.

From the above theorem we can derive several other composition results; for example:

Theorem 3: (Composition)

$$\frac{\texttt{MA} \approx_r \cdots \approx_q \texttt{ISA} \qquad \texttt{ISA} \parallel P \vdash \varphi}{\texttt{MA} \parallel P \vdash \varphi}$$

The above theorem states that to prove MA $\parallel P \vdash \varphi$ (that MA, the pipelined machine, executing program P satisfies property φ, a property over the ISA visible state), it suffices to prove MA $\approx$ ISA and ISA $\parallel P \vdash \varphi$: that MA refines ISA (which can be done using a sequence of refinement proofs) and that ISA, executing P, satisfies φ. In this form, the above rule exactly matches the compositional proof rules in [8]. What makes such a rule useful is that it can lead to drastically faster verification times, as we show in this paper. It will turn out that the verification times depend much more on the semantic difference between models than on their complexity, *e.g.*, verifying that a complex pipelined machine, MA, refines a similar complex pipelined machine can take a fraction of a second, even though current tools may not be able to verify that MA refines (the much simpler) instruction set architecture.

III. Processor Modeling and Monolithic Verification

In this section, we define a complex pipelined machine and describe how to model and verify it using UCLID. The machine is quite complicated and to make its definition a manageable process, we defined a series of machines starting with the base processor model M6, a 6 stage pipelined machine with the following stages: instruction fetch (IF), instruction

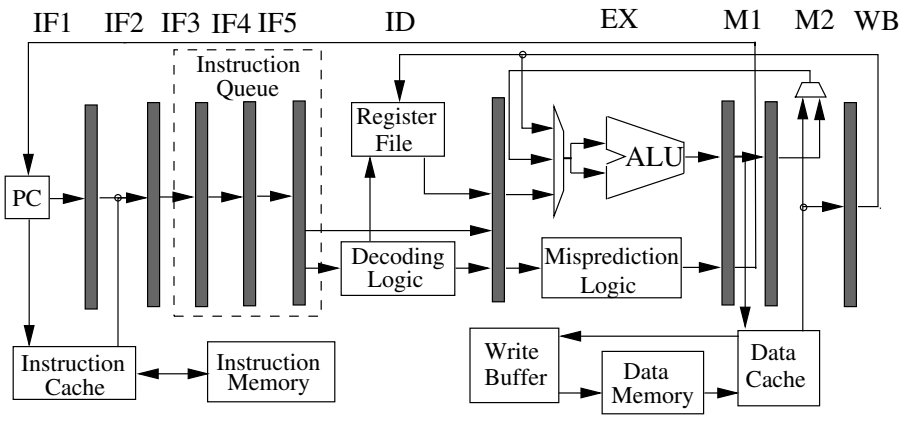

Fig. 1. M10IDW is a processor model with 10 pipeline stages, an instruction queue, an instruction and data cache, and a write buffer.

decode (ID), execute (EX), data memory access (M1 and M2), and write back (WB). M6 has the following instruction types: branches, loads, stores, and ALU instructions. The addressing modes include register-register and register-immediate. M6 also has a simple branch prediction scheme that always predicts that the branch is taken. Once M6 was designed and verified, we extended it with a pipelined fetch stage to obtain M7; then we added an instruction queue holding up to 3 instructions, giving rise to machines M8, M9, and M10. Finally we added a direct mapped instruction cache, a direct mapped data cache, and a write buffer, giving rise to machines M10I, M10ID, and M10IDW. The final machine, M10IDW is shown in Figure 1.

The pipelined machine models in this paper are defined using UCLID and are therefore term-level models, which implies that they are quite abstract. For example: the datapath is abstracted away, the ALU is modeled as an uninterpreted function, and only a small number of instructions are considered. The details of how this is done for M10IDW are discussed in detail elsewhere [19], [20]. Here we merely point out that the caches are write-through and that inductive invariants — essentially stating that all the valid entries in the caches are consistent with memory— are required. The invariants took only a few minutes to define. The write buffer is implemented as a queue of length four and memory reads first check if the data to be read is in the write buffer. A simple invariant stating that the write buffer is consistent is needed.

Unfortunately, M10IDW is too complex to directly verify with UCLID. In Table I we show various verification statistics when checking that the processor models defined above refine the instruction set architecture using flushing as the refinement map. For all experimental results presented in this paper, we used the default settings of the UCLID decision procedure (version 1.0) coupled with the siege SAT solver [25] (variant 4), using an Intel Pentium III Mobile CPU, 1.2 GHz processor with an L2 cache size of 512KB.

To understand the experimental results, first note that as was pointed out in Section II, our refinement proofs imply that the pipelined machines satisfy *all* the $\mathrm{CTL}^* \setminus \mathsf{X}$-expressible safety *and* liveness properties satisfied by the ISA machine. Manolios and Srinivasan have shown that the running time for checking safety and liveness using WEB refinement increases by about 5% over the running time for only checking safety [18]; hence, liveness is not the culprit. Second, as can be seen from the table, the verification cost increases exponentially as new features or pipeline stages are added, leading eventually to machines that are too complex to directly verify with UCLID and Siege. The reason for the exponential increase in verification times is mostly due to the fact that the number of symbolic simulation steps required by the flushing approach depends on the length of the pipeline and that the size of the SAT instance generated by UCLID depends on the complexity gap between the pipelined machine and its instruction set architecture.

Wouldn't it be great if we could use the same approach to verifying M10IDW that we used to design it? Recall that since M10IDW was too complicated to design directly we defined a sequence of intermediate machines instead. This allowed us to add features one at a time, making the design a manageable process. Why not verify M10IDW in the same way? For example, when proving M7 refines ISA, why can't we use the already established result that M6 refines ISA to simplify the proof? In the next section, we show how to do this.

IV. Compositional Verification

In this section we develop techniques that allow us to prove that M10IDW refines ISA in a compositional manner, by proving that M10IDW refines M10ID, which refines M10I, ..., which refines M6, which refines ISA. We present a sound and complete method for proving such theorems, where most of the reasoning is local, *i.e.*, restricted to pairs of machines. By applying our techniques, we transform the problem of verifying that M10IDW refines ISA from one that UCLID cannot handle to one that takes less than 20 seconds.

A UCLID specification gives rise to a transition system $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$, where s is a state ($s \in S$) iff it maps the variables appearing in the UCLID specification to values of the right type. The transition relation $\dashrightarrow$ is similarly

Refinement Proof	CNF		Verification Times (sec)		
	Vars	Clauses	UCLID	Siege	Total
M6	28,256	83,725	8	10	18
M7	53,165	158,182	15	150	165
M8	95,092	283,465	25	766	791
M9	144,045	429,973	41	2,436	2,477
M10	198,375	592,660	55	6,762	6,817
M10I	293,862	876,820	92	8,641	8,733
M10ID	580,355	1,730,704	244	FAILED	NA
M10IDW	690,598	2,060,557	297	FAILED	NA

TABLE I

VERIFICATION TIMES AND CNF STATISTICS FOR THE VARIOUS PIPELINE MACHINE MODELS.

defined over S. An *inductive invariant*, I, is a subset of S that is closed under the transition relation ($s \in I$ implies $\dashrightarrow(\{s\}) \subseteq I$). Put another way, I is an inductive invariant if $\mathcal{M}' = \langle I, \dashrightarrow|_I, L|_I\rangle$, which we sometimes denote $\mathcal{M}|_I$, is a transition system (*i.e.*, the restriction of $\dashrightarrow$ to I is a subset of I^2). It is sometimes useful to identify a subset of S, $Init(\mathcal{M})$, as "initial." If B is a relation, we define $B^X(Y)$ to be $B(Y) \cap X$. We start with two basic observations.

Lemma 2: If $\mathcal{M} = \langle S, \dashrightarrow, L\rangle, \mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$ are TS's, and $\mathcal{M} \approx_r \mathcal{M}'$ with witness B, then: (a) if I is an inductive invariant of $\mathcal{M}$, then $I' = B^{S'}(I)$ is an inductive invariant of $\mathcal{M}'$ and $\mathcal{M}|_I \approx_{r|_I} \mathcal{M}'|_{I'}$, and (b) if I' is an inductive invariant of $\mathcal{M}'$, then $I = B^S(I')$ is an inductive invariant of $\mathcal{M}$ and $\mathcal{M}|_I \approx_{r|_I} \mathcal{M}'|_{I'}$.

Proof: For the proof of (a), let $s \in I'$ and let $s \dashrightarrow' w$; we show that $w \in I'$. By the definition of I', there is some $u \in I$ such that uBs. Since s and u are stuttering bisimilar, w can be matched by a state reachable from u, say by v, but since I is an invariant, $v \in I$, therefore I' is an inductive invariant of $\mathcal{M}'$, and $\mathcal{M}|_I \approx_{r_I} \mathcal{M}'|_{I'}$ with witness $B \cap (I \cup I')^2$. The proof of (b) is similar. □

We will make use of the following corollary of Lemma 2, since it applies to all of our examples in this paper.

Corollary 1: If in Lemma 2 the equivalence class, under B, of every $s \in I$ has exactly one element from S', we can replace $B^{S'}(I)$ by $r(I)$ and $B^S(I')$ by $r^{-1}(I')$.

Let's consider applying what we have so far to show that M10IDW refines ISA. Since we consider all states in ISA to be initial, this means that our refinement map has to be surjective. Recall that we are after a compositional proof, so we will prove a sequence of theorems. Let us say that one of these theorems shows that My refines Mx, which implies that: (a) we have an inductive invariant, I, of My, giving rise to $\mathtt{My}|_I$, and (b) $\mathtt{My}|_I$ refines Mx, say with refinement map r.

To prove that Mz refines Mx, we only need to prove that Mz refines My, say with refinement map q, as we can then appeal to the composition theorem and the theorem that My refines Mx. When one tries to do this in practice, the following problem arises: we need an invariant on Mz whose image under q is I, but defining such an invariant can be quite difficult, requiring much trial and error. (For example, this arises when proving that M9 refines M7, as we will shortly see.) As we show with the following proof rule, it is in fact enough if the image of the invariant under q is a superset of I.

In the sequel, if Z is a set, then $Z^=$ and $Z^{\equiv}$ denote the identity relation on Z and the reflexive, symmetric, transitive closure on Z, respectively.

Theorem 4: Suppose that for all $k \in [1..n]$, I_k is an inductive invariant of TS $\mathcal{M}_k = \langle S_k, \dashrightarrow_k, L_k\rangle$. Suppose also that for all $k \in [2..n]$, $(\mathcal{M}_k)|_{I_k} \approx_{r_k} \mathcal{M}_{k-1}$ with witness B_k and $I_{k-1} \subseteq I'_{k-1}$, where $I'_{k-1} = B_k^{S_{k-1}}(I_k)$. Then, there exists an inductive invariant $I \subseteq I_n$ such that $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$ with witness B and $\langle \forall s \in I_1 :: \langle \exists u \in I :: sBu\rangle\rangle$, where $R = (r_n; r_{n-1}; \cdots; r_2)|_I$ and $B = (I^=; B_n; B_{n-1}; \cdots; B_2; I_1^=)^{\equiv}$.

Proof: The proof is by induction on n, where the base case ($n = 2$) follows from Lemma 2. For the induction step, we have by the induction hypothesis, I', an inductive invariant of $\mathcal{M}_{n-1}$, such that $I' \subseteq I_{n-1}$, $(\mathcal{M}_{n-1})|_{I'} \approx_{R'} (\mathcal{M}_1)|_{I_1}$, and $\langle \forall s \in I_1 :: \langle \exists u \in I' :: sB'u\rangle\rangle$, where $R' = (r_{n-1}; r_{n-2}; \cdots; r_2)|_{I_{n-1}}$ and $B' = (I^=; B_{n-1}; B_{n-2}; \cdots; B_2; I_1^=)^{\equiv}$. Now, letting $I = B_n^{S_n}(I')$, we see that $I \subseteq I_n$ and I is an inductive invariant, such that $(\mathcal{M}_n)|_I \approx_{r_n|_I} (\mathcal{M}_{n-1})|_{I'}$ (by Lemma 2). By Theorem 2, $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$. Finally, let $s \in I_1$; by the induction hypothesis, there is a $w \in I'$ such that $sB'w$. Now, let $u \in B_n^{I_n}(\{w\})$, which is non-empty, but since $w \in I'$ and I is an inductive invariant, $u \in I$. □

The proof rule embodied in Theorem 4 is completely local —every proof obligation involves at most two TS's— and should be used where applicable. Unfortunately, it is *incomplete*: it is possible that there is an inductive invariant $I \subseteq I_n$ such that $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$, but we cannot prove it with the above proof rule. (This situation arises when proving that M7 refines ISA, as explained later.) In such cases, the following complete proof rule should be used.

Theorem 5: Suppose that for all $k \in [1..n]$, I_k is an inductive invariant of TS $\mathcal{M}_k = \langle S_k, \dashrightarrow_k, L_k\rangle$. Suppose also that for all $k \in [2..n]$, $(\mathcal{M}_k)|_{I_k} \approx_{r_k} \mathcal{M}_{k-1}$ with witness B_k. Then, there exists an inductive invariant $I \subseteq I_n$ such that $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$ with witness B and $\langle \forall s \in I_1 :: \langle \exists u \in I :: sBu\rangle\rangle$ iff $B_2^{S_1}(\cdots B_{n-1}^{S_{n-2}}(B_n^{S_{n-1}}(I_n))\cdots) \subseteq I_1$, where $R = (r_n; r_{n-1}; \cdots; r_2)|_I$ and $B = (I^=; B_n; B_{n-1}; \cdots; B_2; I_1^=)^{\equiv}$.

Proof: Let $I = B_2^{S_1}(\cdots B_{n-1}^{I_{n-2}}(B_n^{I_{n-1}}(I_n))\cdots), I' = B_n^{I_n}(\cdots B_3^{I_3}(B_2^{I_2}(I))\cdots)$. For the proof from left to right, we show that I, I' are inductive invariants, that $I \supseteq I_1, I' \subseteq I_n$, and $(\mathcal{M}_n)|_{I'} \approx_R \mathcal{M}_1|_I$. For the induction step, we can use the conclusion of the induction hypothesis because $B_2^{S_1}(\cdots B_{n-1}^{S_{n-2}}(I_{n-1})\cdots) \supseteq I_1$ (since $B_{n-1}^{S_{n-2}}(B_n^{S_{n-1}}(I_n)) \subseteq B_{n-1}^{S_{n-2}}(I_{n-1})$). We now have that $B_n^{S_{n-1}}(I_n)$ and I_{n-1} are inductive invariants, thus so is $B_n^{I_{n-1}}(I_n)$, which is not empty as $B_{n-1}^{S_{n-2}}(B_n^{I_{n-1}}(I_n)) = B_{n-1}^{S_{n-2}}(B_n^{S_{n-1}}(I_n))$. Using the induction hypothesis, we get that I, I' are inductive invariants, that $I \supseteq I_1, I' \subseteq I_n$, and $(\mathcal{M}_n)|_{I'} \approx_R \mathcal{M}_1|_I$. The rest of the proof is similar to the proof of Theorem 4. □

Notice that Theorem 5 gives us much more flexibility than Theorem 4, because the relationship between I'_k and I_k can be arbitrary. Also, if (as is the case in our applications) the

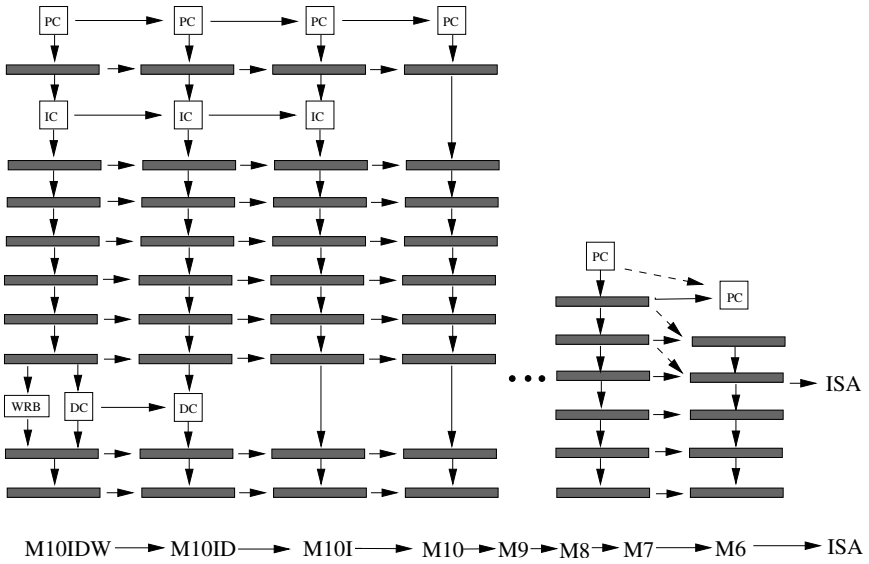

Fig. 2. Refinement maps for the compositional verification of M10IDW.

equivalence class of every $s \in I_i$, under B_i, has exactly one element from S_{i-1}, then the global condition amounts to showing that any state $s \in I_1$ can be reached by starting in some state in I_n and applying the following sequence of refinement maps: $r_n, r_{n-1}, \ldots, r_2$. For pipelined machines, this turns out to be easy to show because applying this sequence of refinement maps to pipelined machines whose non-ISA components are invalid amounts to projecting out the ISA-visible components; thus, every state in ISA is reachable.

We have now developed all of the theory required to verify M10IDW. An overview of the process is shown in Figure 2. Our proof scripts are available upon request and the few invariants required took us less than a day to define. In addition, the rank functions required are much easier to define than in the monolithic case, and there is a simple recipe for doing this described elsewhere [18]. The verification times and related statistics are given in Table II. The names in the "Refinement Proof" column indicate which refinement proof the row corresponds to. The models are expressed in the UCLID language, and are translated to CNF formulas using the UCLID tool.

Figure 3 depicts the verification times required for both the direct and the composition methods for each of the processor models. As can be seen from Figure 3, if we compare the verification times required by the direct method versus our compositional method, then we see that the verification cost increases exponentially (the y-axis uses a logarithmic scale) for the direct approach for each new feature/pipeline stage, whereas, for the compositional approach, the verification cost is almost a constant. The data reported for the compositional proofs includes the total time required, including the time required for the proof of invariants, and everything else required by our proof rule. Notice that the SAT solver Siege failed to produce a result when applying the direct approach to M10ID, whereas with the compositional approach, the proof of M10IDW required less than 20 seconds.

We now explain the refinement proofs shown in Figure 2 in more detail. First, we discuss how to deal with deep pipelines. Second, we show how to handle caches and write buffers. Finally, we discuss counterexamples.

A. Deep Pipelines

The first five refinement proofs in Table II, which together show that MA10 refines ISA, are described next. We use $I_{\mathtt{M}}$ to denote the invariant on machine M and $r_{\mathtt{M}}$ to denote the refinement map from machine M. (The range is uniquely determined by Table II.) Recall that $I_{\mathtt{ISA}}$ is the set of all ISA states. The proof of M6−ISA is a straightforward direct proof using flushing as the refinement map, thus $I_{\mathtt{M6}}$ is the set of all M6 states.

Our first refinement proof involving two pipelined machines relates M7 to M6 using refinement map $r_{\mathtt{M7}}$ (see Figure 2). We now describe $r_{\mathtt{M7}}$ and merely note that the refinement maps for the other proofs are similar. We name pipeline latches based on the pipeline stage names surrounding them, *e.g.*, the pipeline latch between IF1 and ID in the 6 stage machine is IF1_ID.

The only essential difference between M7 and M6 is that when a branch mispredict occurs, the number of cycles required for M7 to recover is four, while M6 only needs three cycles. To deal with this stuttering, we define three invariants on M7; essentially, they state that a branch mispredict results in four consecutive bubbles in the pipeline. The invariants are 1) if IF1_IF2 is invalid, then IF2_ID, ID_EX, and EX_M1 are invalid; 2) if IF1_IF2 is valid and IF2_ID is invalid, then ID_EX, EX_M1, and M1_M2 are invalid; and 3) if both IF1_IF2 and IF2_ID are valid, and ID_EX and EX_M1 are invalid, then M1_M2 and M2_WB are invalid.

The definition of the refinement map $r_{\mathtt{M7}}$ consists of three cases. In all the cases, the pipeline latches EX_M1, M1_M2, M2_WB, the register file, the instruction memory, and the data

Refinement Proof	CNF		Verification Times (sec)		
	Vars	Clauses	UCLID	Siege	Total
M6–ISA	28,256	83,725	8.00	10.00	18.00
M7–M6	1,116	3,124	0.39	0.06	0.45
M8–M7	479	1,291	0.24	0.01	0.25
M9–M8	380	1,045	0.21	0.01	0.22
M10–M9	433	1,201	0.29	0.01	0.30
M10I–M10	213	562	0.08	0.01	0.09
M10ID–M10I	469	1,210	0.15	0.01	0.16
M10IDW–M10ID	837	2,149	0.23	0.03	0.26

TABLE II
VERIFICATION TIMES AND CNF STATISTICS FOR THE COMPOSITIONAL VERIFICATION PROBLEMS.

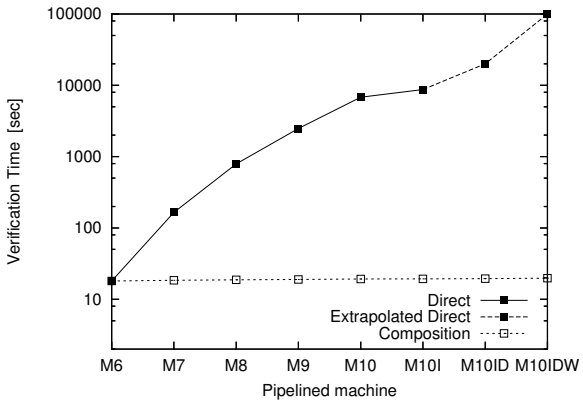

Fig. 3. Comparison of direct and compositional approaches.

memory in M7 get mapped to their counterparts in M6. Case 1 occurs if in M7, IF1_IF2 is invalid, IF1_IF2 is valid and IF2_ID is invalid, or IF1_IF2 and IF2_ID are valid and ID_EX and EX_M1 are invalid. In this case, the program counter, IF1_IF2, and IF2_ID in M7 get mapped to the program counter, IF1_ID, and ID_EX in M6, and the rank is 1. Case 2 occurs when IF1_IF2, IF2_ID, and ID_EX in M7 are valid and EX_M1, M1_M2, and M2_WB are invalid. This is the result of a stuttering step by M7. The rank is 0 and we map the program counter associated with the instruction in IF1_IF2 of M7 to the program counter in M6, while IF2_ID and ID_EX in M7 are mapped to IF1_ID and ID_EX in M6. Otherwise, the mapping of states is the same as in case 2, except that the rank is 0.

To prove compositionally that M7 refines ISA requires the use of Theorem 5. To see why, note that the use of Theorem 4 requires that $r_{\mathtt{M7}}(I_{\mathtt{M7}}) \supseteq I_{\mathtt{M6}}$, which is not true, as I_6 is the set of all M6 states. However, $I_{\mathtt{M7}}$ satisfies the property that $r_{\mathtt{M6}}(r_{\mathtt{M7}}(I_{\mathtt{M7}})) \supseteq I_{\mathtt{ISA}}$, and therefore we can use Theorem 5. To prove this using UCLID, we define a witness function, f, that given an ISA state returns the M7 state with the same programmer visible components, but all of whose pipeline latches are invalid. It is now enough to show that for every state s in $I_{\mathtt{ISA}}$, we have that $r_{\mathtt{M6}}(r_{\mathtt{M7}}(f(s))) = s$ and that $f(s) \in I_{\mathtt{M7}}$.

For the rest of the deep pipeline proofs, it turns out that we can use the simpler Theorem 4. For example, in case of the M8–M7 proof, we have to show that $r_{\mathtt{M8}}(I_{\mathtt{M8}}) \supseteq I_{\mathtt{M7}}$, which we do by defining a suitable witness function that maps states in $I_{\mathtt{M7}}$ to $I_{\mathtt{M8}}$ and then proceed as above.

B. Instruction Caches, Data Caches, and Write Buffers

We now show how to verify the instruction cache, the data cache, and the write buffer. This corresponds to the last three refinement proofs in Table II. For all of these proofs, we use the proof rule given in Theorem 4. Since we have seen how to apply the theorem in the previous section, here we only describe the refinement map, invariants, and witness function for each of the proofs.

The state components of M10I and M10 are identical except for the instruction cache. Thus, the refinement map just ignores the instruction cache and is the identity mapping for all other state components. Since two machines do not stutter with respect to one another, we can in fact prove a bisimulation. This means that the WEB-refinement proof can be reduced further, as no rank function is needed. The only invariant required is that the valid instruction cache entries are consistent with the instruction memory.

The data cache is direct mapped and is similar to the instruction cache. The proof of M10ID–M10I is similar to the proof of M10I–M10. The refinement map ignores the data cache and retains all the other state components, including the instruction cache. Also, an invariant similar to the one used for the instruction cache is required stating that all valid entries in the data cache are consistent with the data memory.

M10IDW differs from M10ID only in that it contains a write buffer. These two machines do not stutter with respect to each other; thus, we can prove a bisimulation result, as before. The refinement map is obtained by first updating the data memory with the valid entries in the write buffer, and projecting out the remaining state elements (including the instruction and data cache states). We prove the invariant that the combined state of the write buffer and the data memory is consistent with the state of the data memory of a machine that does not have a write buffer.

Finally, the witness function from $I_{\mathtt{M10}}$ to $I_{\mathtt{M10I}}$ just adds an instruction cache, all of whose elements are invalid to an $I_{\mathtt{M10}}$ state. The witness functions for the other proofs are similarly defined.

C. Counterexamples

The most tedious and time-intensive part of the verification effort is often debugging and understanding counterexamples. Since the compositional approach reduces the verification problem into simpler subproblems, the debugging process is much simplified. This is because one can isolate the cause of failure simply by noting which stage of the composition proof fails. This is impossible to do when verifying the complex processor in a monolithic fashion and it is difficult to overstate the importance of this aspect of our work, as the differences in the complexity of the error traces can be quite drastic.

As a concrete example of how compositional verification simplifies the debugging task, we note that when we tried

to verify a buggy variant of the instruction cache —there was a bug because when determining whether a cache hit has occurred, the design did not check the validity of the cache block— we found that the counter example generated by UCLID for the direct approach was 4,429 lines long while the counter example generated from the composition step was 390 lines long. Obviously, the shorter counterexample was much simpler to understand and, consequently, fixing the bug was much easier. All the bugs we encountered were similarly much easier to check in the compositional framework and this aspect of compositional verification may well be more important than the improvement we obtained in verification times.

V. Conclusions and Future Work

We presented a complete compositional framework based on refinement for proving that pipelined machine models satisfy the same safety and liveness properties as their corresponding instruction set architecture models. This allowed us to obtain exponential savings in verification times over previous monolithic approaches, and, in fact, we were able to easily verify models that state-of-the-art tools cannot directly handle. We also showed how compositional reasoning based on refinement can be integrated into the design cycle and how this leads to faster verification times, shorter and clearer counterexamples, and enhanced design understanding by verification engineers. All of our models are available upon request and for future work we plan to extend our compositional results to refinement based on stuttering simulation and to apply compositional reasoning to more complex processors.

References

[1] M. D. Aagaard, N. A. Day, and R. B. Jones. Synchronization-at-retirement for pipeline verification. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 3312 of *LNCS*, pages 113–127. Springer-Verlag, November 2004.

[2] T. Arons. Verification of an advanced mips-type out-of-order execution algorithm. In *Computer-Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 414–426. Springer-Verlag, 2004.

[3] T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *Lecture Notes in Computer Science*, pages 487–502. Springer-Verlag, March 2000.

[4] M. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59, 1988.

[5] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 470–482. Springer-Verlag, 1999.

[6] R. E. Bryant, S. K. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. Larsen, editors, *Computer-Aided Verification–CAV 2002*, volume 2404 of *LNCS*, pages 78–92. Springer-Verlag, 2002.

[7] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[9] L. de Moura and H. Ruess. An experimental evaluation of ground decision procedures. In *Computer Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 162–174, July 2004.

[10] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Assume-guarantee refinement between different time scales. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 208–221. Springer-Verlag, 1999.

[11] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

[12] R. Jones, J. Skakkebæk, and D. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, November 1998.

[13] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[14] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, 2002.

[15] P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design–FMCAD 2000*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.

[16] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001. See URL `http://-www.cc.gatech.edu/~manolios/publications.html`.

[17] P. Manolios. A compositional theory of refinement for branching time. In D. Geist and E. Tronci, editors, *12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*, volume 2860 of *LNCS*, pages 304–318. Springer-Verlag, 2003.

[18] P. Manolios and S. Srinivasan. Automatic verification of safety and liveness for XScale-like processor models using WEB-refinements. In *Design Automation and Test in Europe, DATE'04*, 2004.

[19] P. Manolios and S. Srinivasan. A computationally efficient method based on commitment refinement maps for verifying pipelined machines models. In *ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 189–198, 2005.

[20] P. Manolios and S. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2005. to appear.

[21] P. Manolios and S. Srinivasan. Refinement maps for efficient verification of processor models. In *Design Automation and Test in Europe, DATE'05*, pages 1304–1309, 2005.

[22] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1990.

[23] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 284–296, 1997.

[24] S. Ray and W. A. Hunt, Jr. Deductive verification of pipelined machines using first-order quantification. In *Computer-Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 31–43. Springer-Verlag, 2004.

[25] L. Ryan. Siege homepage. See URL `http://www.cs.sfu.ca/~loryan/personal`.

[26] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, Dec. 1999. See URL `http://-www.cs.utexas.edu/users/sawada/dissertation/`.

[27] J. Sawada. Verification of a simple pipelined machine model. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 137–150. Kluwer Academic Publishers, June 2000.