A Refinement-Based Compositional Reasoning Framework for Pipelined Machine Verification

Panagiotis Manolios, *Member, IEEE*, and Sudarshan K. Srinivasan, *Member, IEEE*

***Abstract*—We present a refinement-based compositional framework for showing that pipelined machines satisfy the same safety and liveness properties as their non-pipelined specifications. Our framework consists of a set of convenient, easily applicable, and complete compositional proof rules. We show how to apply our compositional framework in the context of microprocessor verification to verify both abstract, term-level models and executable, bit-level models. Our framework enables us to verify machine models that are significantly more complex than the kinds of models that can be verified using current state-of-the-art automated decision procedures. For example, using our framework, we can verify a 32-bit, 10-stage, executable pipelined machine model. In addition, our compositional framework offers drastic improvements in the context of design debugging over monolithic approaches, in part because bugs are isolated to particular steps in the compositional proof and because the counter examples generated are much smaller.**

***Index Terms*—Compositional reasoning, pipelined machine verification, refinement.**

I. Introduction

PIPELINING is a key optimization technology that is used extensively in hardware systems such as microprocessors, multicore systems, and cache coherence protocols. For example, the Intel Pentium 4 microprocessor uses hyper-pipelined technology and has pipelines with as many as 31 stages [8]. Even though pipelining has been around for decades, there are currently no efficient and scalable techniques that can check that pipelines work correctly.

The focus of the verification of pipelined machine models—models that describe the pipelined behavior of hardware designs—has been to show that pipelined machines implement or refine their non-pipelined specifications. Typically, the verification times required for pipelines increases exponentially with increase in the number of stages or the complexity of the model. Therefore, the verification of the pipelines of many industrial designs is beyond the complexity threshold of automatic verification tools. 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.

We present a compositional reasoning framework based on well-founded equivalence bisimulation (WEB) refinement for pipelined machine verification that provides a high degree of scalability and takes us a step closer in handling industrial designs. The framework allows us to decompose correctness proofs for pipelined machines into manageable pieces and can therefore be used to substantially extend the complexity threshold of automatic tools. We show how to apply our framework in the context of both term-level and bit-level pipelined machine verification. The work presented in this paper extends a previous conference version [20] by showing how to apply our compositional reasoning framework to the verification of bit-level pipelined machine models and also by including a detailed description of the techniques developed.

We show that using our framework, we can verify a 32-bit, 10-stage pipelined machine model defined at the bit-level. Such a proof was previously not possible using automatic verification tools and would have required a heroic amount of theorem proving effort to complete. We also show that using our framework we can obtain exponential savings in verification time when checking the correctness of a complex term-level pipelined machine. Our compositional framework takes advantage of the way we defined the models. For example, the term-level 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 six-stage pipelined machine, which we extended first with a pipelined fetch stage, then with an instruction queue holding up to three instructions, then with a direct mapped instruction cache, then a direct mapped data cache, and, finally, a write buffer, to obtain `M10IDW`. 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. Our overall proof strategy is highly-automated as the proof obligations required by our compositional framework can be automatically handled using satisfiability testing (SAT)-based decision procedures. For the term-level verification, we use the UCLID decision procedure [4], [15]. For verification at the bit-level, we use the bit-level analysis tool (BAT) [26], [27]. The Siege [33] and

Manuscript received September 5, 2006. This work was supported in part by the National Science Foundation (NSF) under Grant CCF-0429924, Grant IIS-0417413, and Grant CCF-0438871, and by ND EPSCoR under NSF Grant EPS-0447679.

P. Manolios is with the College of Computer and Information Science, Northeastern University, Boston, MA 02115 USA (e-mail: pete@ccs.neu.edu).

S. K. Srinivasan is with the Department of Electrical and Computer Engineering, North Dakota State University, Fargo, ND 58105 USA (e-mail: sudarshan.srinivasan@ndsu.edu).

Digital Object Identifier 10.1109/TVLSI.2008.918120

the RSat [32] SAT solvers are used to check the conjunctive normal form (CNF) problems generated by UCLID and BAT, respectively.

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.

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 straightforward, 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.

This paper is organized as follows. In Section II, we review the theory of refinement upon which our correctness proofs depend. In Section III, we describe the compositional proof rules developed for pipelined machine verification. In Sections IV and V, we describe the application of our compositional reasoning framework to the verification of complex pipelined machine models at the term-level and the bit-level, respectively. Everything required to reproduce our results, e.g., machine models, correctness statements, CNF formulas, etc., is available upon request. We give related work in Section VI. Conclusions and an outline of future work appear in Section VII.

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 [17] and [18]. 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 micro-architecture 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 [2], where by $fp(\sigma, s)$ we mean that σ is a fullpath (infinite path) starting at s. The definition also depends on the notion of matching which we only informally describe here. We are given a relation B on a set S. We say that an infinite sequence σ (of elements from S) matches an infinite sequence δ (of elements from S), written match(B, σ, δ), if the sequences can be partitioned into non-empty, finite segments such that elements in related segments are related by B.

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

$$\begin{aligned}
&(\mathrm{Stb1}) \quad L.s = L.w \\
&(\mathrm{Stb2}) \quad \langle \forall \sigma : fp(\sigma, s) : \langle \exists \delta : fp(\delta, w) : \text{match}(B, \sigma, \delta)\rangle\rangle \\
&(\mathrm{Stb3}) \quad \langle \forall \sigma : fp(\sigma, w) : \langle \exists \sigma : fp(\sigma, s) : \text{match}(B, \sigma, \delta)\rangle\rangle
\end{aligned}$$

Browne, Clarke, and Grumberg have shown that states that are stuttering bisimilar satisfy the same next-time-free temporal logic formulas [2].

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

We note that stuttering bisimulation differs from weak bisimulation [30] 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 an 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.

In the sequel, we refer to any relation than can serve the role of B, above, as a *witness* to the refinement $\mathcal{M} \approx_r \mathcal{M}'$.

A major shortcoming of the previous formulation of refinement is that it requires reasoning about infinite paths, something that is difficult to automate [31]. In [17], 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 [19], 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

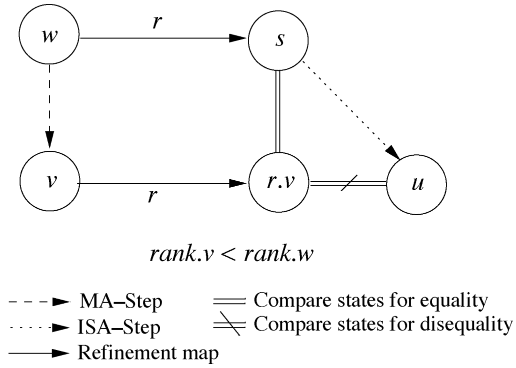

Fig. 1. Pictorial representation of Theorem 1, showing a strengthening of the WEB-refinement theorem that facilitates the automation of reasoning about the correctness of pipelined machines.

is the following theorem, where $rank$ is a function that maps states of MA into the natural numbers.

Theorem 1: $\texttt{MA} \approx_r \texttt{ISA}$ if

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

In the previous formula, 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. The previous formula is also shown pictorially in Fig. 1.

Note that the notion of WEB refinement is independent of the refinement map used. In this paper, we use the standard flushing refinement map [5], 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.

It is worth pointing out that the choice of refinement map can have a big impact on the verification effort [12], [21], [22]. Nevertheless, we will not explore this further here, as 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 previous theorem, we can derive several other composition results; see the following for example.

Theorem 3: (Composition)

$$\frac{\begin{array}{c}\texttt{MA} \approx_r \cdots \approx_q \texttt{ISA} \\ \texttt{ISA} \| P \vdash \varphi\end{array}}{\texttt{MA} \| P \vdash \varphi}.$$

The previous theorem states that to prove $\texttt{MA} \parallel P \vdash \varphi$ (that MA, the pipelined machine, executing program P satisfies property φ, a property over the ISA visible state), it suffices to prove $\texttt{MA} \approx \texttt{ISA}$ and $\texttt{ISA} \parallel P \vdash \varphi$: 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 previous rule exactly matches the compositional proof rules in [6]. 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.

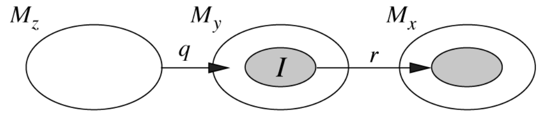

Fig. 2. Invariant mismatch. When trying to apply Theorem 2 to prove that M_z refines M_x compositionally, by first proving that M_z refines M_y, say with refinement map q, and then that $M_y|_I$ refines M_x, say with refinement map r, we run into the following problem. We need an invariant on M_z whose image under q is I; defining such an invariant can be quite difficult.

III. Composition Rules

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 s.

A model in any standard hardware description or specification language (e.g., BAT) 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 specification to values of the right type. The transition relation $\dashrightarrow$ is similarly defined over S. An *inductive invariant*, I, is a subset of S that is closed under the transition relation ($s \in I$ implies that the image of s under the transition relation, $\dashrightarrow (\{s\})$, is a subset of 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 TSs, 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, $v \in I$ and, consequently, $w \in B^{S'}(I) = I'$.

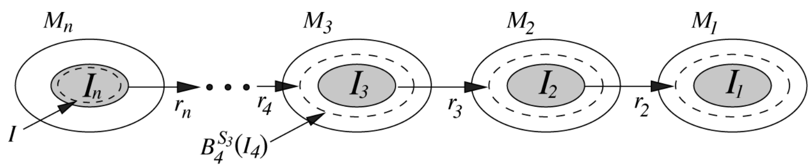

Fig. 3. This figure depicts the local composition rule. The shaded subset of $\mathcal{M}_k$ for $1 \leq k \leq n$ corresponds to inductive invariant I_k of TS $\mathcal{M}_k$. Suppose for all $k \in [2, \ldots, n]$, $(\mathcal{M}_k)|_{I_k} \approx_{r_k} \mathcal{M}_{k-1}$ with witness B_k and $I_{k-1} \subseteq B_k^{S_{k-1}}(I_k)$. Then, there exists an inductive invariant $I \subseteq I_n$ and refinement map $R = (r_n; r_{n-1}; \cdots; r_2)|_I$ such that $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$.

Therefore, I' is an inductive invariant of $\mathcal{M}'$. Also, $\mathcal{M}|_I \approx_{r|_I} \mathcal{M}'|_{I'}$ with witness $B' = B \cap (I \cup I')^2$. First, note that B' is an equivalence relation. Now, consider any a, b such that $aB'b$. The fullpaths from a and b can be matched by B', since they only include states in $I \cup I'$ and I, I' are inductive invariants. 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 us 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: 1) we have an inductive invariant, I, of My, giving rise to My$|_I$ and 2) 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 invariant mismatch problem arises as shown in Fig. 2: 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 see in Section IV.) 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 $k \in [1 \ldots n]$, I_k is an inductive invariant of TS $\mathcal{M}_k = \langle S_k, \dashrightarrow_k, L_k \rangle$. Suppose also that for all $k \in [2, \ldots, n]$, $(\mathcal{M}_k)|_{I_k} \approx_{r_k} \mathcal{M}_{k-1}$ 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, $u \in I$. ∎

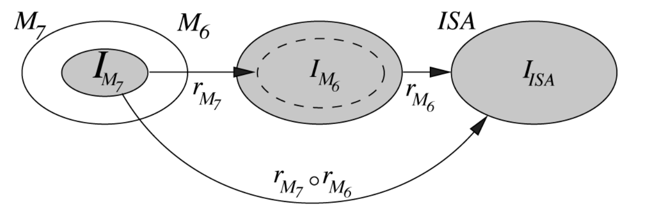

Fig. 4. Incompleteness of the local composition rule. This figure depicts the situation we find ourselves in when proving that M7 refines ISA. The use of Theorem 4 (corresponding to taking the top path through the figure) requires that $r_{M7}(I_{M7}) \supseteq I_{M6}$. Unfortunately, is not true, as I_{M6} is the set of all M6 states. However, I_{M7} really does refine ISA with refinement map $r_{M7} \circ r_{M6}$ (corresponding to taking the bottom path through the figure). Thus, the local proof rule is incomplete.

The proof rule embodied in Theorem 4 is shown pictorially in Fig. 3. It is completely local—every proof obligation involves at most two TSs— 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 previous proof rule. (This situation is shown in Fig. 4 and arises, for example, when proving that M7 refines ISA, as explained in Section IV-C.) In such cases, the following complete proof rule should be used.

Theorem 5: Suppose that for all $k \in [1 \ldots n]$, I_k is an inductive invariant of TS $\mathcal{M}_k = \langle S_k, \dashrightarrow_k, L_k \rangle$. Suppose also that for all $k \in [2, \ldots, n]$, $(\mathcal{M}_k)|_{I_k} \approx_{r_k} \mathcal{M}_{k-1}$ 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 $I' = B_2^{S_1}(\cdots B_{n-1}^{S_{n-2}}(B_n^{S_{n-1}}(I_n)) \cdots) \supseteq I_1$, 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 $I = B_n^{I_n}(\cdots B_3^{I_3}(B_2^{I_2}(I')) \cdots)$. For the proof from right to left, we show that I, I' are inductive invariants, that $I' \supseteq I_1$, $I \subseteq I_n$, and $(\mathcal{M}_n)|_I \approx_R \mathcal{M}_1|_{I_1}$. 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 $(\mathcal{M}_n)|_I \approx_R \mathcal{M}_1|_{I_1}$. The rest of the proof is similar to the proof of Theorem 4. ∎

We call the proof rule embodied in Theorem 5 the *global rule*. It is depicted in Fig. 5. Notice that the global rule gives us much

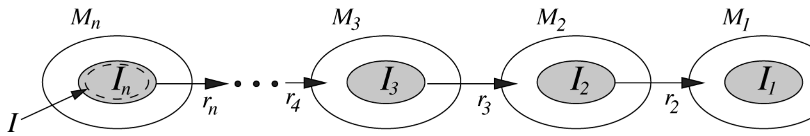

Fig. 5. This figure depicts the global composition rule. The shaded subset of $\mathcal{M}_k$ for $1 \leq k \leq n$ corresponds to inductive invariant I_k of TS $\mathcal{M}_k$. Suppose that for all $k \in [2, \ldots, n]$, $(\mathcal{M}_k)|_{I_k} \approx_{r_k} \mathcal{M}_{k-1}$ with witness B_k. Then, there exists an inductive invariant $I \subseteq I_n$ such that $(\mathcal{M}_n)|_I \approx_R (\mathcal{M}_1)|_{I_1}$ and $\langle \forall s \in I_1 :: \langle \exists u \in I :: sBu \rangle\rangle$ iff $B_2^{S_1}(\cdots B_{n-1}^{S_{n-2}}(B_n^{S_{n-1}}(I_n))\cdots) \supseteq I_1$, where $R = (r_n; r_{n-1}; \cdots; r_2)|_I$.

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 equivalence class of every $s \in I_i$, under B_i, has exactly one element from S_{i-1}, then the global rule 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.

IV. Term-Level Compositional Reasoning

In this section, we show how to apply refinement-based compositional reasoning to the verification of pipelined machine models defined at the term-level. In Sections IV-A and IV-B, we describe the models and show how they are verified monolithically, respectively. In Section IV-C, we describe how we use the compositional rules developed in Section III to efficiently verify complex pipelined machine models.

A. Pipelined Machine Model

We define a complex pipelined machine and describe how to model it at the term-level using the UCLID system. 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 six-stage pipelined machine with the following stages: instruction fetch (IF), instruction decode (ID), execute (EX), data memory access (M1 and M2), and write back (WB). M6 has the following instruction types: branches, loads, stores, and arithmetic logic unit (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 three 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 Fig. 6.

The term-level pipelined machine models are defined using UCLID, which allows one to write formulas in the logic of counter arithmetic with restricted Lambda expressions and uninterpreted functions (CLU). The CLU logic consists of Booleans and terms, whose values are integers. Counter arithmetic includes the successor and predecessor functions, whose arguments are terms. Terms can be compared using $=$ or $<$. The logic also contains uninterpreted functions (UFs) and uninterpreted predicates (UPs), which correspond to arbitrary functions from integers to integers, and integers to Booleans,

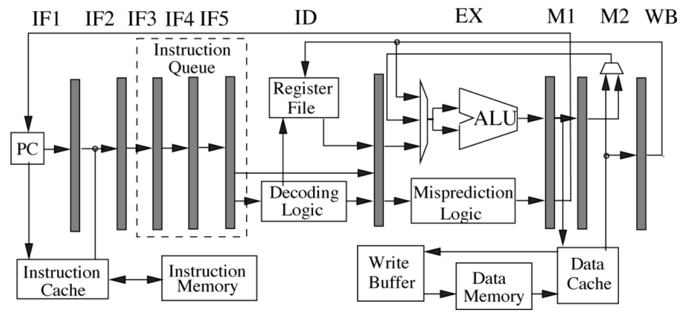

Fig. 6. M10IDW is a processor model with 10 pipeline stages, an instruction queue, an instruction and data cache, and a write buffer.

respectively. Lambda expressions only take integer inputs and are restricted so that it is not possible to describe any form of recursion using the CLU logic.

The term-level pipelined machines modeled using UCLID use numerous abstractions. The data path is abstracted using terms. Combinational circuit blocks such as the ALU and the instruction decoder functions are abstracted using UFs and UPs. Memories are modeled as functions that map addresses to data values using restricted lambda expressions. Since the instruction memory is never updated, it can be modeled using a UF.

Since the modeling of pipelined machines using UCLID is well documented [4], [19], we only describe some of the features we have modeled in UCLID. The instruction cache is modeled using three memory elements ICache − Valid, ICache − Tag, and ICache − Block that take the index as input and return a Boolean value indicating if the entry in the instruction cache is valid, the tag, and the data block, respectively. Three UFs *GetIndex*, *GetTag*, and *GetBlockOffset* take the program counter as input, and return the index, tag, and block offset, respectively. Another UF *SelectWord* is used to extract the instruction from the data block. When an instruction cache is used, the instruction memory is modeled as a UF that takes two arguments, an index and a tag, and returns a block of data. This way of modeling the instruction memory allows us to match the contents of the instruction memory and the instruction cache.

The data cache is direct mapped and is similarly modeled. Writes to the data memory are write-through and update the data cache. Similar to the instruction memory, when a data cache is used, the data memory is modeled using a lambda expression that takes two arguments, an index and a tag, and returns a block of data. This way of modeling the data memory allows us to match the contents of the data memory and the data cache.

The write buffer is implemented as a queue and has four entries. Each entry has a data part, an address part, and a valid bit. Store instructions do not update the data memory directly, but write to the tail of the write buffer queue. The head of the write buffer queue is read and used to update the data memory. Reads from the data memory have to take into account the valid entries in the write buffer, as the write buffer has the most recent data values. Among the write buffer entries, priority is given to the entries closer to the tail.

B. Monolithic Verification

In this section, we describe the monolithic verification of the pipelined machine models described in Section IV-A. For this, we use UCLID, since it includes a decision procedure for the CLU logic. Overall, we find that the verification times increase exponentially with increase in the complexity of the pipelined machine models. Also, the most complex model `M10IDW` cannot be directly verified using UCLID.

For the monolithic verification, we use the flushing refinement map. As stated earlier, refinement maps are functions from pipelined machine states to ISA states and are defined using the programmer visible components of a pipelined machine state, which are the pipelined machine's state elements that are also present in the ISA state. For the machines we consider, the programmer visible components include the program counter, the register file, the instruction memory, and the data memory. Burch and Dill [5] showed how to automatically define the flushing refinement map, which is constructed by completing all the partially executed instructions in the pipeline without fetching any new instructions and projecting out the programmer visible components in the resulting state.

For the verification of the pipelined machines with an instruction cache, we need an invariant stating that valid instruction cache entries should be consistent with those in the instruction memory

$$\begin{aligned} &\mathtt{ICache-Valid}(I) \quad \wedge \quad \mathtt{ICache-Tag}(I) = T \\ \Longrightarrow \quad &\mathtt{ICache-Block}(I) = \mathtt{IMemory}(I, T). \end{aligned}$$

In the previous formula, I is an arbitrary index value and T is an arbitrary tag value. The invariant states that if the entry corresponding to index I in the instruction cache is valid and the tag in the cache is equal to T, then the data block in the cache should be equal to the data block from the instruction memory. We also prove that the instruction cache invariant is inductive, i.e., we prove that if the invariant holds for an arbitrary pipelined machine state, it also holds for any successor state.

An inductive invariant similar to the instruction cache is required for the data cache, stating that all the valid entries in the data cache are consistent with the data memory. We also require an inductive invariant for the write buffer establishing that if we update the data memory with all the valid entries in the write buffer, then we obtain the memory we would have obtained if a write buffer were not used. That is, if D is the data memory and U is the memory state obtained after updating all the valid write buffer entries to D, then $U = R$, where R is a memory that is similar to D except that store instructions directly update R (instead of going through the write buffer).

In Table I we show various verification statistics when checking that the processor models defined before refine the instruction set architecture using flushing as the refinement map. For all experimental results presented in this section, we used the default settings of the UCLID decision procedure (version 1.0) coupled with the siege SAT solver [33] (variant 4), using an Intel Pentium III Mobile CPU, 1.2-GHz processor with an L2 cache size of 512 kB.

TABLE I
VERIFICATION TIMES AND CNF STATISTICS FOR THE VARIOUS TERM-LEVEL PIPELINE MACHINE MODELS

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

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 [19]; hence, liveness is not the culprit. Second, as can be seen from Table I, 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 as follows.

The flushing refinement map is constructed using the flush operation, which is a step of the pipelined machine that does not fetch any new instruction. The flush operation is almost as complex as stepping the pipelined machine. Therefore, as the complexity of the machine increases, the flush operation also becomes more complex. Also, the number of flush operations required to construct the flushing refinement map increases as the length of the pipeline increases. Therefore, the verification conditions generated when using the flushing refinement map tend to be quite complex, leading to an exponential increase in the verification times. Other refinement maps also suffer, to varying degrees, from the problem of exponentially increasing verification times in the complexity of the pipelined machines being verified [22].

Would it not 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 we not use the already established result that `M6` refines `ISA` to simplify the proof? In Section IV-C, we show how to do this.

C. Compositional Verification

All of the theory required to verify `M10IDW` in a compositional manner has been developed in Section III. An overview of the compositional verification of `M10IDW` is shown in Fig. 7. Our proof scripts are available upon request and the few invariants required took us less than a day to define. In addition, the rank

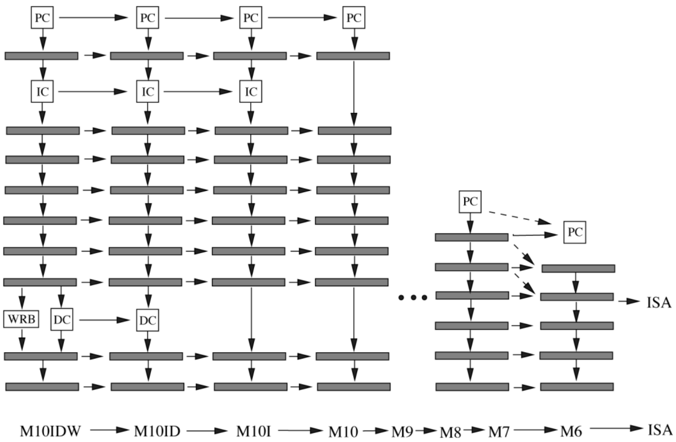

Fig. 7. Refinement maps for the compositional verification of M10IDW.

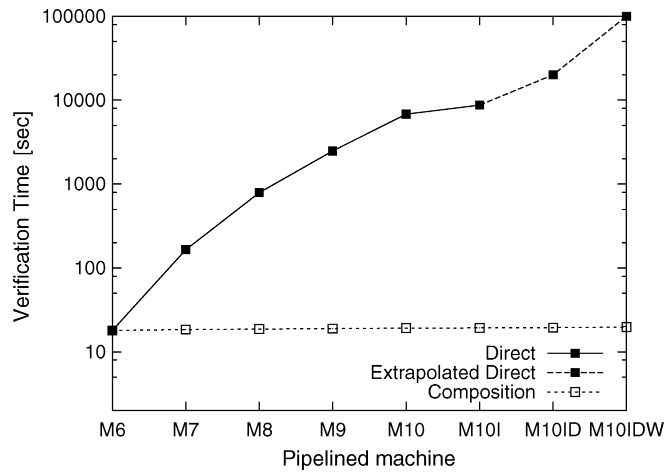

Fig. 8. Comparison of direct and compositional approaches for term-level reasoning.

TABLE II
VERIFICATION TIMES AND CNF STATISTICS FOR THE TERM-LEVEL COMPOSITIONAL VERIFICATION PROBLEMS

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

functions required are much easier to define than in the monolithic case, and there is a simple recipe for doing this described elsewhere [19]. 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.

Fig. 8 depicts the verification times required for both the direct and the composition methods for each of the processor models. As can be seen from Fig. 8, 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 s.

We now explain the refinement proofs shown in Fig. 7 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.

1) 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 Fig. 7). 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 six-stage machine is IF1_ID (see Fig. 6).

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 two cases. In both cases, the pipeline latches EX_M1, M1_M2, M2_WB, the register file, the instruction memory, and the data 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. Otherwise, case 2 occurs 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. In case 2, the rank is given a value 0 because the situation in which IF1_IF2, IF2_ID, and ID_EX in M7 are valid and EX_M1, M1_M2, and M2_WB are invalid is the result of a stuttering step by M7. Note that for states satisfying case 2 that are not the result of a stuttering step, we could have assigned any value for their rank.

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 the global rule embodied in 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 local proof rule embodied in Theorem 4. For example, in the 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 before.

2) 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 local proof rule. 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 the 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. Note that we required less than a man-week of expert user effort to define the refinement maps, rank functions, invariants, and witness functions required for all the compositional proofs.

3) 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 4429 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. Bit-Level Compositional Reasoning

We now describe how to apply our compositional reasoning framework to the verification of bit-level models. For this purpose, we define a complex 10-stage 32-bit pipelined machine model (we call M10B) and describe the verification of this model using both monolithic and compositional approaches.

A. Pipelined Machine Models

In this section, we describe the 10-stage, 32-bit pipelined machine (M10B), which is modeled using the specification language of the BAT [25], [27], [28]. We give a brief overview of the BAT specification language, before we describe the model.

The BAT specification language is a strongly typed lisp-based language that can be a target for synthesizable subsets of VHDL or Verilog. The language is quite expressive and easy to use. Bit-vector concatenation, arithmetic, logical, and relational operators are supported. User defined functions are supported. Functions can be defined to return multiple values, a feature that we use to define functions that correspond to the operational semantics of pipelined and ISA machines. Memories are treated as first-class objects and can be compared directly for equality and passed as arguments to functions.

We now describe the model itself. The high-level pipeline structure and the pipeline stages of M10B are similar to that of M10 described in Section IV-A. Whereas M10 is not executable and is defined at the term-level using numerous abstractions, M10B is an executable model defined at the bit-level. As stated earlier, the word size of the data path is 32 bits. M10B has an instruction memory and a data memory, both of which have 2^{32} words each. The register file has 16 registers. M10B implements various ALU, load, store, and branch instructions and has a simple branch prediction scheme that always predicts that a branch instruction is taken.

As with term-level modeling, to make the definition of M10B a manageable process, we define a series of machines starting with a base processor model M4B, a four-stage 32-bit pipelined machine that has an instruction fetch (IF), instruction decode (ID), execute (EX), and a memory access stage (M1). Write

TABLE III
VERIFICATION TIMES AND CNF STATISTICS FOR THE VARIOUS BIT-LEVEL PIPELINED MACHINE VERIFICATION PROBLEMS

Refinement Proof	CNF		Verification Times (sec)		
	Vars	Clauses	BAT	RSat	Total
M4B-ISAB	11,163	48,712	16.07	1,174.18	1,190.25
M5B-ISAB	12,128	53,836	22.59	1,309.32	1,331.91
M6B-ISAB	12,056	52,911	16.47	3,495.95	3,512.42
M7B-ISAB	23,271	117,553	43.69	TO	NA
M8B-ISAB	33,534	174,141	66.94	TO	NA
M9B-ISAB	45,551	240,488	102.56	TO	NA
M10B-ISAB	59,258	316,432	168.33	TO	NA

back is performed in the memory access stage M1. M4B was extended with a write-back stage (WB) to obtain M5B; then the memory access stage was pipelined to get a two-cycle memory access stage (M1 and M2) resulting in M6B; next the instruction fetch stage was pipelined to get a two-cycle fetch stage (IF1 and IF2). Finally, we added an instruction queue holding up to three instructions, giving rise to machines M8B, M9B, and M10B, where M10B is the final machine. All these bit-level machines implement the same instruction set architecture, ISAB. Note that the high-level pipeline structure and the pipeline stages of the bit-level models M6B, M7B, M8B, M9B, and M10B are similar to the term-level models M6, M7, M8, M9, and M10, respectively.

B. Monolithic Verification

We verified the bit-level pipelined machines M4B, ..., M10B, monolithically by showing that they refine ISAB. We used flushing refinement maps for the correctness proofs. The methods for computing flushing refinement maps and corresponding rank functions for the bit-level pipelined machines are similar to those for the term-level machines and are described in more detail in Section IV-B.

The verification conditions for each of the machines is discharged by the BAT. This is achieved by describing the correctness statement for a machine as a BAT specification. BAT transforms the input specification to a SAT problem in CNF format using a state-of-the-art memory abstraction technique [25] and a novel and efficient method for generating SAT problems from a high-level circuit representation [28]. We use version 2.01 of the RSat SAT solver [32] to check the CNF problems generated by BAT.

Table III shows the verification times and the CNF statistics for the monolithic verification of the pipelined machines M4B, ..., M10B, using BAT. All the BAT experiments described in this paper were performed using a 2.40-GHz Intel(R) Pentium(R) 4 processor with an L2 cache size of 512 kB. The "BAT" column in Table III gives only the time for converting the bit-level verification problem to a CNF problem. The CNF problem that BAT generates is then checked using version 2.01 of the RSat SAT solver. The running times for RSat are shown in the "RSat" column of Table III. The total verification time is the sum of the BAT and RSat times. A "TO" entry indicates that the RSat SAT solver timed out. We set a timeout limit of 5000 s. An "NA" entry in the "Total" column of the table indicates that RSat timed out, and therefore, we do not have the total verification time for that problem.

As can be seen from Table III, the verification times increases as more features are added. Also, for machines M7B, M8B, M9B, and M10B, BAT generates CNF that current SAT solvers cannot handle.

C. Compositional Verification

We now describe the compositional verification of the bit-level pipelined machine models. We start with the base model M4B, and show that it refines ISAB. Next, we show that M5B refines M4B, M6B refines M5B, ..., and M10B refines M9B. This sequence of refinement proofs is performed using the compositional rules described in Section III and guarantees that M10B refines ISAB.

The compositional proofs involve defining refinement maps, rank functions, witness functions, and invariants. The proof obligations for the compositional proofs are then automatically discharged using BAT. The bit-level compositional proofs M7B-M6B, M8B-M7B, M9B-M8B, and M10B-M9B are similar to the term-level compositional proofs M7-M6, M8-M7, M9-M8, and M10-M9, respectively.

Under a refinement map that flushes the last stage of M5B, machine models M5B and M4B do not stutter with respect to one another; therefore, we can prove that these machines are bisimilar. For the compositional proof M5B-M4B, the refinement map is defined from M5B to M4B and maps the program counter, the instruction memory, the data memory, and the pipeline latches IF_ID, ID_EX, EX_M in M5B directly to the program counter, the instruction memory, the data memory, and the pipeline latches IF_ID, ID_EX, EX_M in M4B. If a valid instruction is present in the M_WB latch in M5B, it is made to update the register file and the refinement map projects the resulting updated register file onto the register file in M4B. The compositional proof M5B-M4B also does not require any invariants, and therefore no other proof obligation is required to establish that M5B refines M4B.

For the compositional proof M6B-M5B, the refinement map, which is similar to the refinement map for the proof M5B-M4B, is defined by mapping all the states in M6B directly onto the corresponding states in M5B, except for the pipeline latch M2_WB and the register file. The refinement map updates the register file in M6B with the instruction in M2_WB, and the resulting updated register file is projected onto the register file in M5B. We require two invariant properties on M6B for the M6B-M5B compositional proof. The first invariant states that the first argument stored in the ID_EX pipeline latch is equal to the value in the register file corresponding to the first source address stored in the ID_EX latch. The second invariant is a similar invariant property for the second argument stored in the ID_EX latch. Note that the $I_{\mathtt{M5B}}$ states is the set of all M5B states. Therefore, we cannot use the local compositional rule defined in Theorem 4 as $r_{\mathtt{M6B}}(I_{\mathtt{M6B}}) \supseteq I_{\mathtt{M5B}}$ is not true. Therefore, we use Theorem 5, which is the global compositional rule. The proof M6B-M5B using Theorem 5 is similar to the proof of M7-M6.

Table IV gives the verification times and the CNF statistics for the bit-level compositional proofs. The organization of Table IV is similar to the organization of Table III. Also, the experimental setup used for the monolithic approach described in Section V-B is also used here. As can be seen from Table IV, the verification times for the compositional proofs are almost negligible when compared to the M4B-ISAB proof. Fig. 9 compares the bit-level monolithic and compositional approaches. Fig. 9 shows the increase in the verification times for both

TABLE IV
VERIFICATION TIMES AND CNF STATISTICS FOR THE BIT-LEVEL COMPOSITIONAL VERIFICATION PROBLEMS

Refinement Proof	CNF		Verification Times (sec)		
	Vars	Clauses	BAT	RSat	Total
M4B-ISAB	11,163	48,712	16.07	1,174.18	1,190.25
M5B-M4B	1,573	5,459	1.55	2.28	3.83
M6B-M5B	2,905	9,655	3.23	7.69	10.92
M7B-M6B	9,332	37,835	11.91	16.49	28.40
M8B-M7B	3,969	13,670	5.06	2.27	7.33
M9B-M8B	3,615	12,078	4.29	2.01	6.30
M10B-M9B	3,432	10,950	3.94	1.15	5.09

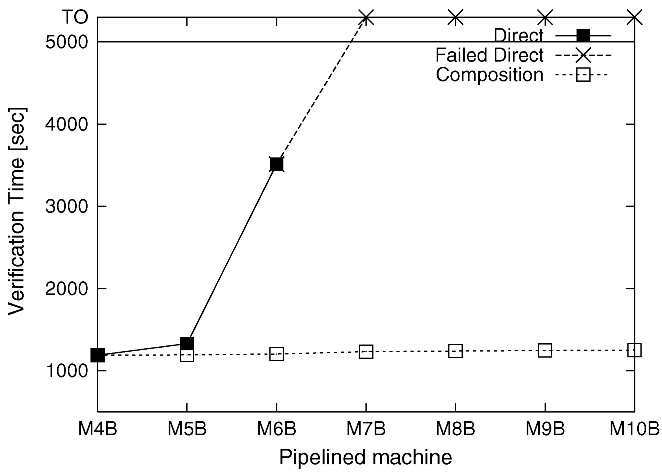

Fig. 9. Comparison of direct and compositional approaches for bit-level reasoning.

approaches as more features are added to the base pipelined machine model M4B. As can be seen from Fig. 9, the verification times for the monolithic approach increases as more features are added and the monolithic approach very quickly reaches its complexity threshold and cannot handle models M7B, M8B, M9B, and M10B, whereas the verification times for the compositional approach remains almost a constant and the compositional approach can handle the most complex pipelined machine model M10B in under 1500 s. As with the term-level verification, we required less than a man-week of expert user effort to define the refinement maps, rank functions, invariants, and witness functions required for the compositional proof.

VI. RELATED WORK

Pipelined machine verification is an active area of research. Previous work in this area can be classified into approaches that are based on the use of deductive reasoning engines (otherwise known as theorem provers) and approaches that use decision procedures.

Approaches based on decision procedures are highly automated, but are only applicable to term-level models. An early and influential paper in this area is due to Burch and Dill [5], who showed how to compute refinement maps based on flushing. More directly related to this paper is the work on decision procedures for the CLU logic [4], which is based on previous work on exploiting positive equality [3]. The decision procedure is implemented in UCLID, which has been used to verify various processor models [15].

In previous work, we have also shown how to automatically verify pipelined machines using refinement maps based on commitment [16], [19], which can be thought of as the dual of flushing as partially executed instructions are invalidated and the programmer visible components are rolled back to correspond with the last committed instruction. Since refinement maps have a drastic impact on verification times, there have been several approaches that use optimized refinement maps, which lead to drastic improvement in verification times [12], [21], [22]. The main drawback of these monolithic approaches is that the verification times increases almost exponentially with increase in the complexity of the models being verified. Therefore, these approaches are not scalable. In comparison, we have shown that our compositional approach provides a high degree of scalability.

There have also been approaches based on the use of automatic verification engines that aim to decompose correctness proofs into smaller manageable pieces. Jones *et al.* [10], [11] 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. They also state that the amount of effort required to deductively justify the proof decompositions offsets the advantages obtained using the decomposition. In our approach, the individual refinement proofs can be chained together as WEB-refinement is a compositional notion.

Jhala and McMillan [9] describe an approach for showing that processor models behave like their instruction set architecture models using compositional model checking. The proof methodology is to decompose the correctness criterion into a number of temporal properties that can then be automatically verified using a model checker. The decomposition is performed using the SMV proof assistant [29]. They apply this approach to verify an abstract microprocessor model that has many features such as branch prediction, speculative execution, and out-of-order execution. In their models, combinational circuit blocks such as the ALU are abstracted using UFs. They do not quantitatively describe the amount of expert user effort required for the proofs, but state that their approach is "considerably more laborious" than model checking finite state machines. In contrast, we required only about one week of expert user effort for the compositional refinement proof. Also, our approach accounts for liveness, whereas, they do not check any liveness properties.

Another popular approach for pipelined machine verification is based on the use of theorem provers. Theorem provers typically have underlying logics that 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 [34], [35]. Another example of a theorem proving approach is the work by Hosabettu *et al.*, who use the notion of completion functions [7], and the work of Arons [1]. While such approaches are applicable to bit-level designs, they usually require a prohibitive amount of effort on the part of the expert user.

In previous work, we have also shown how to verify bit-level designs using a combination of deductive reasoning and decision procedures. We integrated the UCLID decision procedure with the ACL2 theorem proving system. We then used the resulting combined system in a compositional refinement framework to verify a bit-level pipelined machine [23], [24]. The high-level idea of the approach is to use ACL2 to reduce the bit-level verification problem to a term-level problem. UCLID is then used to reason about the pipeline at the term-level.

A recent trend is the development of efficient decision procedures to reason about designs at the bit-level, an example of which is BAT [26], [27]. The BAT system has in fact been used to prove correctness of a five-stage 32-bit pipelined machine in less than 2 min of running time [25]. In this paper, we show how to use BAT in a compositional reasoning framework to verify a 10-stage 32-bit pipelined machine. The correctness proof required only a week of expert user effort.

The notion of correctness for pipelined machines that we use was first proposed in [16] and is based on WEB-refinement [17]. The first proofs of correctness for pipelined machines based on WEB-refinement were carried out using the ACL2 theorem proving system [13], [14]. 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 [16], whereas it follows directly from the WEB-refinement approach that deadlock (or any other liveness problem) will be detected. There is also work that shows how to automatically verify safety *and liveness* properties of pipelined machines using WEB-refinement [19]. Our results extend this work by showing how to use WEB-refinement to automatically prove safety and liveness in a compositional fashion.

Why has something like this not 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 develop a complete compositional theory of refinement that enables us to do this for both safety and liveness (the Burch and Dill approach only provides safety [16]), and with the use of any refinement map, not just flushing.

VII. Conclusion and Future Work

We have presented a refinement-based compositional reasoning framework for proving that pipelined machines satisfy the same safety and liveness properties as their instruction set architectures. We have shown how to apply our framework to verify complex pipelined machines defined at both the term-level and the bit-level. Using our framework, we verified a 32-bit, 10-stage, complex pipelined machine model. Such a proof is not possible using state-of-the-art decision procedures, and would have required an extraordinary amount of expert user effort with current theorem proving technology. In contrast, we required less than one man-week of effort for the bit-level proof, which was used to define the refinement maps, rank functions, invariants, and witness functions. 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. For future work, we plan to extend our compositional results to refinement based on stuttering simulation and to apply compositional reasoning to more complex designs.

References

[1] T. Arons, "Verification of an advanced MIPS-type out-of-order execution algorithm," in *Proc. Comput.-Aided Verification (CAV)*, 2004, vol. 3114, pp. 414–426.

[2] M. Browne, E. M. Clarke, and O. Grumberg, "Characterizing finite Kripke structures in propositional temporal logic," *Theoretical Comput. Sci.*, vol. 59, pp. 115–131, 1988.

[3] R. E. Bryant, S. German, and M. N. Velev, "Exploiting positive equality in a logic of equality with uninterpreted functions," in *Proc. Comput.-Aided Verification (CAV)*, 1999, vol. 1633, pp. 470–482.

[4] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Proc. Comput.-Aided Verification (CAV)*, 2002, vol. 2404, pp. 78–92.

[5] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Proc. Comput.-Aided Verification (CAV)*, 1994, vol. 818, pp. 68–80.

[6] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Boston, MA: MIT Press, 1999.

[7] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Proof of correctness of a processor with reorder buffer using the completion functions approach," in *Proc. Comput.-Aided Verification (CAV)*, 1999, vol. 1633, pp. 686–698.

[8] Intel, Santa Clara, CA, "Intel pentium 4 processor—Product overview," 2005 [Online]. Available: http://www.intel.com/design/pentium4/prodbref/

[9] R. Jhala and K. L. McMillan, "Microarchitecture verification by compositional model checking," in *Proc. Int. Conf. Comput.-Aided Verification (CAV)*, 2001, vol. 2102, pp. 396–410.

[10] R. Jones, J. Skakkebæk, and D. Dill, "Reducing manual abstraction in formal verification of out-of-order execution," in *Formal Methods in Computer-Aided Design (FMCAD)*, G. Gopalakrishnan and P. Windley, Eds. New York: Springer-Verlag, 1998.

[11] R. B. Jones, J. U. Skakkebæk, and D. L. Dill, "Formal verification of out-of-order execution with incremental flushing," *Formal Methods Syst. Des.*, vol. 20, no. 2, pp. 139–158, Mar. 2002.

[12] R. Kane, P. Manolios, and S. K. Srinivasan, "Monolithic verification of deep pipelines with collapsed flushing," in *Proc. Des., Autom. Test Eur. (DATE)*, 2006, pp. 1234–1239.

[13] M. Kaufmann, P. Manolios, and J S. Moore, Eds., *Computer-Aided Reasoning: ACL2 Case Studies*. Norwell, MA: Kluwer, Jun. 2000.

[14] M. Kaufmann, P. Manolios, and J S. Moore, *Computer-Aided Reasoning: An Approach*. Norwell, MA: Kluwer, Jul. 2000.

[15] S. Lahiri, S. Seshia, and R. Bryant, "Modeling and verification of out-of-order microprocessors using UCLID," in *Proc. Formal Methods Comput.-Aided Des. (FMCAD)*, 2002, vol. 2517, pp. 142–159.

[16] P. Manolios, "Correctness of pipelined machines," in *Proc. Formal Methods Comput.-Aided Des. (FMCAD)*, 2000, vol. 1954, pp. 161–178.

[17] P. Manolios, "Mechanical verification of reactive systems" Ph.D. dissertation, Dept. Comput. Sci., Univ. Texas at Austin, Austin, TX, Aug. 2001 [Online]. Available: http://www.cc.gatech.edu/~manolios/publications.html

[18] P. Manolios, "A compositional theory of refinement for branching time," in *Proc. 12th IFIP WG 10.5 Adv. Res. Work. Conf. (CHARME)*, 2003, vol. 2860, pp. 304–318.

[19] P. Manolios and S. K. Srinivasan, "Automatic verification of safety and liveness for xscale-like processor models using WEB refinements," in *Proc. Des., Autom. Test Eur. Conf. Expo. (DATE)*, 2004, pp. 168–175.

[20] P. Manolios and S. K. Srinivasan, "A complete compositional reasoning framework for the efficient verification of pipelined machines," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 2005, pp. 863–870.

[21] P. Manolios and S. K. Srinivasan, "A computationally efficient method based on commitment refinement maps for verifying pipelined machines," in *Proc. Formal Methods Models Co-Des. (MEMOCODE)*, 2005, pp. 188–197.

[22] P. Manolios and S. K. Srinivasan, "Refinement maps for efficient verification of processor models," in *Proc. Des., Autom. Test Europe (DATE)*, 2005, pp. 1304–1309.
[23] P. Manolios and S. K. Srinivasan, "Verification of executable pipelined machines with bit-level interfaces," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 2005, pp. 855–862.
[24] P. Manolios and S. K. Srinivasan, "A framework for verifying bit-level pipelined machines based on automated deduction and decision procedures," *J. Autom. Reason.*, vol. 37, no. 1–2, pp. 93–116, 2006.
[25] P. Manolios, S. K. Srinivasan, and D. Vroon, "Automatic memory reductions for RTL model verification," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 2006, pp. 786–793.
[26] P. Manolios, S. K. Srinivasan, and D. Vroon, "BAT: The bit-level analysis tool," 2006 [Online]. Available: http://www.cc.gatech.edu/~manolios/bat/
[27] P. Manolios, S. K. Srinivasan, and D. Vroon, "BAT: The bit-level analysis tool," in *Proc. Int. Conf. Comput.-Aided Verification (CAV)*, 2007, pp. 303–306.
[28] P. Manolios and D. Vroon, "Efficient circuit to CNF conversion," in *Proc. Int. Conf. Theory Appl. Satisfiability Test.*, 2007, pp. 4–9.
[29] K. L. McMillan, "A methodology for hardware verification using compositional model checking," *Sci. Comput. Program*, vol. 37, no. 1–3, pp. 279–309, 2000.
[30] R. Milner, *Communication and Concurrency*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
[31] K. S. Namjoshi, "A simple characterization of stuttering bisimulation," in *Proc. 17th Conf. Foundations Softw. Technol. Theoretical Comput. Sci.*, 1997, vol. 1346, pp. 284–296.
[32] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Proc. 10th Int. Conf. Theory Appl. Satisfiability Test. (SAT)*, 2007, pp. 294–299.
[33] L. Ryan, "Siege homepage," 2008 [Online]. Available: http://www.cs.sfu.ca/~loryan/personal
[34] J. Sawada, "Formal verification of an advanced pipelined machine" Ph.D. dissertation, Dept. Comput. Sci., Univ. Texas at Austin, Austin, TX, Dec. 1999 [Online]. Available: http://www.cs.utexas.edu/users/sawada/dissertation/
[35] J. Sawada, "Verification of a simple pipelined machine model," in *Computer-Aided Reasoning: ACL2 Case Studies*. Norwell, MA: Kluwer, Jun. 2000, pp. 137–150.

Panagiotis Manolios (M'07) received the B.S. and M.A. degrees in computer science from Brooklyn College, Brooklyn, NY, in 1991 and 1992, and the Ph.D. degree in computer science from The University of Texas at Austin, Austin, in 2001.

He is currently an Associate Professor with Northeastern University, Boston, MA. In 2001, he joined the College of Computing, Georgia Institute of Technology, Atlanta, where he became an Adjunct Assistant Professor with the School of Electrical and Computer Engineering, in 2003. His primary research interests include mechanized formal verification and validation. His other areas of interest include design automation, programming languages, distributed computing, logic, software engineering, algorithms, computer architecture, aerospace, and pedagogy.

Sudarshan K. Srinivasan (M'07) received the M.S. and Ph.D. degrees in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, in 2003 and 2007, respectively, and the B.E. degree in electrical and electronics engineering from the University of Madras, Chennai, India, in 2001.

He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, North Dakota State University, Fargo. His primary research interests include formal verification. His other research interests include computer architecture and electronic design automation.