Compositional Reasoning Based on WEB-Refinement for the Automatic Verification of Pipelined Machines

ABSTRACT

We show how to use compositional reasoning to automatically prove that pipelined machine models satisfy the same safety and liveness properties as their corresponding instruction set architecture models. By applying compositional reasoning to deep pipelines we demonstrate both that we obtain exponential savings in verification times over previous monolithic approaches and that we can, quite easily, verify machines that state-of-the-art tools cannot currently handle. We discuss how compositional reasoning can be added to the design cycle and give an example where we verify a complex pipelined machine with branch prediction, instruction and data caches, and a write buffer. Compositional reasoning allowed us to reduce the verification time by more than a factor of 10 over current state-of-the-art monolithic approaches, and, perhaps more importantly, the counterexamples generated are much simpler, as bugs are isolated to a particular step in the composition.

1. INTRODUCTION

We present a compositional reasoning technique that can be used to automatically reason about pipelined machines that are too complex to handle with current state-of-the-art methods and tools. Our technique allows one to verify that a pipelined machine is correct in stages, by starting with the instruction set architecture, showing that it is refined by a simple pipelined machine, which is refined by a more complex machine, and so on until the final pipelined machine is verified. Each stage of the proof 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. As WEB-refinement is a compositional notion, 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 WEB-refinement proof to a statement expressible in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU), which is a decidable logic [2]. We use the tool UCLID [7] to transform the CLU formula into a CNF (Conjunctive Normal Form) formula, which we then check with the SAT solver Siege [16].

The goal of much of the current work in verification is to design methods, tools, and techniques that extend the range of automatic methods. While there has been much success, we are still very far away from being able to automatically verify industrial designs. Compositional reasoning is one of the main methods for dealing with complex systems and the idea is simply to decompose the verification problem into manageable pieces that are then composed together to produce the final result. How might we apply this idea to pipelined machine verification? For example, suppose that MA is a complex pipelined machine model and ISA is the corresponding instruction set architecture and suppose that proving that MA refines ISA is too difficult to do in one step. An obvious idea is to define an intermediate machine MA' and then show that MA' refines ISA and that MA refines MA'. 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 MA' refines ISA, how does one prove that MA refines MA'? If we use flushing, we have to flush both machines, but then MA' just becomes a complex variant of ISA and the proof is even more complex than proving that MA refines ISA directly. It is not immediately clear how to take advantage of the fact that we already established that MA' refines ISA. Our main contribution is to show how to do this in an automatic way for both safety and liveness (the Burch and Dill approach only provides safety [8]) that also leads to drastic improvements in verification times.

Our compositional method for pipelined machine verification can be used to significantly extend the types of machines that can be verified with state-of-the-art tools such as UCLID and Siege. We show that increasing the number of cycles required to fetch an instruction leads to an exponential growth in verification times, when using a refinement map based on flushing. Thus, while a a 6-stage pipelined machine can be verified in about 10 seconds, by the time we get to a 10-stage pipelined machine, the verification problem is too complicated for Siege to solve. However, by using compositional reasoning the verification time for the complete proof is less than a second longer than the verification time for the 6-stage machine. It is worth noting that other deep pipelines will suffer from the exponential increase in verification times and our method can be used to deal with the problem.

Having established that exponential savings are possible, we turn our attention to another example, using a different refinement map (based on the commitment approach), to show the applicability of our method. We consider a complex machine that includes a data cache, an instruction cache, and a write buffer, and we show how to verify the machine in a compositional manner, by treating each of the caches and the write buffer in separate refinement steps. As a result, we reduce the running time by more than a factor of 10.

Can we really obtain the benefits of composition without paying a price? Actually, we often have to provide invariants. But, for our examples they are simple. 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 and did not make the verification task more difficult. If one uses a hierarchical, compositional approach to design, then the invariants should be known, as they allow different engineers to implement different parts of the system independently. Therefore, composition can fit nicely into the design cycle, which is also compositional. This is why we claim that our method is automatic, because in the context of compositional design, the invariants allowing for the separation of concerns are known.

Compositional verification has several important advantages over monolithic verification that are perhaps even more important than the increased performance. 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.

In this way, the development and debugging processes are simplified. By focusing on a new feature at a time, the engineer is assured that all previous features are correct. Debugging is with respect to the current stage in the refinement process, not with respect to the ISA, thus, as we later show, the running times are faster, the counterexamples are shorter and clearer, and design understanding is enhanced.

While our main contribution is to show how to use composition to automatically prove safety and liveness properties for pipelined machines, as far as we know, we are also the first to model data and instruction caches and write buffers, and the first to automatically prove safety and liveness for these features. In addition, we are the first to show how to relate two pipelined machines.

The paper is organized as follows. In Section 2, we briefly review related work. In Section 3, we provide an overview of WEB-refinement, the compositional theory of refinement upon which our correctness proofs depend. In Section 4, we examine the deep pipeline example, and in Section 5, we consider a pipelined machine with branch prediction, instruction and data caches, and a write buffer. 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 6.

2. 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 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 [17, 19, 18]. Another example of a theorem proving approach is the work by Hosabettu et al., who use the notion of completion functions [4].

Our main concern, however, is with automatic methods. An early and influential paper in this area is due to Burch and Dill, who showed how to automatically compute the refinement map using flushing [3]. The idea is that a pipelined machine is related to an instruction set architecture state by feeding the pipeline with enough bubbles to complete all the partially executed instructions. The use of flushing is now widespread, although there has been recent work on another, dual approach to flushing called the commitment approach [8, 11], where a pipelined machine state is related to an instruction set architecture state by invalidating all the partially executed instructions in the pipeline and reseting the PC so that it points to the oldest invalidated instruction. Different types of automatic methods have been used, *e.g.,* McMillan uses model-checking and symmetry reductions [12]; Patankar et al. use Symbolic Trajectory Evaluation (STE) to verify a processor that is a hybrid between ARM7 and StrongARM [15]; and SVC is used to check the correct flow of instructions in a pipelined DLX model [13].

More directly related to this paper is the work on decision procedures for boolean logic with equality and uninterpreted function symbols [1]. The results in [1] were further extended in [2], where a decision procedure for the CLU logic is given. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors [7] and which we use to verify the models presented in this paper. We use the UCLID tool and models that are similar to the models in [11], which in turn are similar to the models in [20].

The notion of correctness for pipelined machines that we use was first proposed in [8], and is based on WEB-refinement [9]. The first proofs of correctness for pipelined machines based on WEB-refinement were carried out using the ACL2 theorem proving system [5, 6]. The advantage of using a theory of refinement over using the Burch and Dill notion of correctness, even if augmented a "liveness" criterion, is that deadlock may avoid detection with the Burch and Dill approach [8], whereas it follows directly from the WEB-refinement approach that deadlock (or any other liveness problem) is ruled out. In [11], it is shown how to automatically verify safety *and liveness* properties of pipelined machines using WEB-refinement. The proofs are carried out using UCLID and Siege, and it is shown that Siege outperforms Chaff [14], which is why we use Siege in this paper. Our results extend this work by showing how to use WEB-refinement to automatically prove safety and liveness in a compositional fashion.

3. A COMPOSITIONAL THEORY OF REFINEMENT

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 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. What exactly do we mean when we say MA refines ISA? We mean that the two systems are *stuttering bisimilar*: for every pair of states w, s such that w is an MA state and $r(w) = s$, we have that for every infinite path σ starting at s, there is a "matching" infinite path δ starting at w, and conversely. That σ and δ "match" implies that applying r to the states in δ results in a sequence that is equivalent to σ up to finite stuttering (repetition of states). 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. Of course, reasoning about infinite paths is difficult to automate, and in [9], WEB-refinement, an equivalent formulation is given that requires only local reasoning. We will define WEB-refinement and will show how to use it to reason compositionally and automatically about pipelined machines.

The definitions are given in terms of general transition systems (TS). A TS, $\mathcal{M}$, is a triple $\langle S, \dashrightarrow, L\rangle$, consisting of a set of states, S, a transition relation, $\dashrightarrow$, and a labeling function L whose domain S and where $L(s)$ corresponds to what is visible at state s. We start by defining the notion of WEB on a single transition system.

DEFINITION 1. *$B \subseteq S \times S$ is a WEB on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ iff:*

(1) B is an equivalence relation on S; and

(2) $\langle \forall s, w \in S :: sBw \Rightarrow L(s) = L(w)\rangle$; and

(3) There exist functions $erankl : S \times S \to \mathbb{N}, erankt : S \to W$, such that $\langle W, \lessdot\rangle$ is well-founded, and

$\langle \forall s, u, w \in S :: sBw \;\wedge\; s \dashrightarrow u \;\Rightarrow$

(a) $\langle \exists v :: w \dashrightarrow v \;\wedge\; uBv\rangle \;\vee$

(b) $(uBw \;\wedge\; erankt(u) \lessdot erankt(s)) \;\vee$

(c) $\langle \exists v :: w \dashrightarrow v \;\wedge\; sBv \;\wedge$
$erankl(v,u) < erankl(w,u)\rangle\rangle$

If states s and w are in the same equivalence class, then they have the same infinite behaviors, up to stuttering. The first two conditions are straightforward. The third WEB condition states that given states s and w in the same class, such that s can step to u, u is either matched by a step from w, or u and w are in the same class and *erankt* decreases (which guarantees that w is eventually forced to take a step), or some successor v of w is in the same class as s and *erankl* decreases (to guarantee that u is eventually matched). The point of the above definition is that it allows us to avoid reasoning about infinite sequences, since to prove the WEB conditions reasoning about single steps of $\dashrightarrow$ suffices. We now show how to use the definition of WEB to define the notion of WEB-refinement.

DEFINITION 2. *(WEB 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 WEB 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 a WEB 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.*

To apply the above definition in the context of pipelined machine verification, $\mathcal{M}'$ corresponds to the ISA and $\mathcal{M}$ corresponds to MA. It turns out that if MA is a refinement of ISA, then the two machines satisfy the same formulas expressible in the temporal logic $\text{CTL}^* \setminus \mathsf{X}$, over the state components visible at the instruction set architecture level. $\text{CTL}^* \setminus \mathsf{X}$ is a very expressive temporal logic, allowing one to express both safety and liveness properties.

The above notion is *compositional*, that is we can prove the following theorem, where $r;q$ denotes composition, *i.e.*, $(r;q)(s) = q(r(s))$.

THEOREM 1. *(Composition)*
If $\mathcal{M} \approx_r \mathcal{M}'$ and $\mathcal{M}' \approx_q \mathcal{M}''$ then $\mathcal{M} \approx_{r;q} \mathcal{M}''$.

To use WEB-refinement for automatic verification of pipelined machines we strengthen the WEB-refinement proof obligation such that we obtain a CLU-expressible statement that holds for the examples we consider. The details appear elsewhere [11]; here we review the essential elements. The equivalence classes of B consist of one ISA state and all the MA states that map to the ISA state under r, thus, condition 2 of the WEB definition clearly holds. By using oracle variables, we can make ISA and MA deterministic [10], and after some symbolic manipulation, we can strengthen condition 3 of the WEB definition to the following "core theorem", where *rank* is a function that maps states of MA into the natural numbers.

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

In the formula above `ISA-step` is the function which steps the ISA machine once and `MA-step` is the function which steps the MA machine once. The proof obligation relating s and v is the safety component, and the proof obligation that $rank(v) < rank(w)$ is the liveness component.

4. DEEP PIPELINE EXAMPLE

We prove that a pipeline model "MA2" with 7 stages including 2-cycle instruction fetch (F1, F2), instruction decode (ID), execute (EX), 2-cycle memory access (M1, M2), and write back (WB) termed refines MA1, pipeline model with 6 stages including F1, ID, EX, M1, M2, and WB. Both machines implement the following abstract instruction types. ALU instructions, loads, stores, and branch instructions, with register-register and register-immediate addressing modes. The pipeline organization is inspired by the Intel XScale processor. Notice that the difference between MA1 and MA2 is that MA2 has a 2-cycle fetch stage and a total of 6 pipeline latches, and MA1 has a 1-cycle fetch stage with a total of 5 pipeline latches. Relating machines with different number of pipeline latches allows us to verify deep pipelined machines in stages, by exploiting the compositional property of WEB-refinements. We demonstrate the verification of a 10 stage deep pipelined machine with a 5-cycle fetch stage in 5 steps, where the steps involve relating the 10 stage pipeline to a 9 stage pipeline, to an 8 stage pipeline, . . . , to a 6 stage pipeline, to the ISA, using flushing as a refinement map. We now describe how we define the refinement map between the various machines.

Bubbles are introduced in the pipeline due to stalls and mispredicted branch instructions in both MA1 and MA2. For MA2, stall

Table 1: Verification times and statistics for the direct and compositional verification of the deep pipelined example.

Processor	CNF Vars	CNF Clauses	Verification Time [sec]	
			Siege	Total
F6	40,083	119,083	13.5	20.6
F7	53,441	159,010	128.4	138.2
F8	95,456	284,557	582.6	599.6
F9	143,954	429,700	2139.2	2165.7
F10	566,784	1,696,156	FAILED	NA
6-7	855	2,341	0.01	0.22
7-8	442	1,180	0.01	0.17
8-9	337	916	0.01	0.15
9-10	3351	9754	0.01	0.78

introduces a bubble in the 3rd latch. Branches are resolved in the memory stage and mispredicted branches introduce bubbles in the first 4 latches. Based on the above observations, we implement 3 invariants, and also check that they are inductive. The invariants are 1) if latch 1 is invalid, then latches 2, 3, and 4 are invalid; 2) if latch 1 is valid and latch 2 is invalid, then latches 3, 4, and 5 are invalid; 3) if both latches 1 and 2 are valid, and latches 3 and 4 are invalid, then latches 5 and 6 are invalid. We required invariant 1 to prove invariant 2 and both invariants 1 and 2 to prove invariant 3. These invariants reduce the reachable MA2 states, allowing us to simplify the refinement map.

MA1 stutters with respect to MA2 when a branch mispredict occurs, as the number of cycles required for MA1 and MA2 to recover from a branch mispredict are 3 and 4, respectively. Therefore, the proof obligation requires both safety and liveness components and is identical to the "core theorem". The refinement map from MA2 to MA1 is defined considering 3 cases. In all the cases, the last 3 latches, the register file, and the data memory in MA2 map to the last 3 latches, the register file, and the data memory in MA1. Case 1 occurs if in MA2, the first latch is invalid, or the first latch is valid and the second latch is invalid, or the first 2 latches are valid and latches 3 and 4 are invalid. For case 1, the program counter and the first 2 latches in MA2 map to the program counter and the first 2 latches in MA1. The rank function is assigned a value 1. Since latch 3 in MA2 is invalid in the first case, we ignore it. Case 2 occurs when latches 1 through 3 in MA2 are valid and latches 4 through 6 are invalid. In this case, MA2 has stuttered. So we assign a rank value 0, and we project the history of the program counter in latch 1 of MA2 to the program counter in MA1. Latches 2 and 3 in MA2 map to latches 1 and 2 in MA1, and we throw away latch 1 in MA2. Case 3 occurs when both case 1 and case 2 do not hold. In case 3, the mapping of states is the same as in case 2, but MA2 is assigned a rank value of 1. The other refinement maps are defined in a similar fashion.

Table 4 presents the results for this example. Prefix "F" refers to applying flushing as a refinement map using the direct technique. "6", "7", "8", "9", and "10" refer to the 6 stage, 7 stage, 8 stage, 9 stage, and 10 stage MA machines, respectively. The models are expressed in the CLU logic, which are then translated to CNF formulas using UCLID. The Siege SAT solver is used to check the CNF formulas. The experiments were conducted on an Intel XEON 2.20GHz processor with an L1 cache size of 512KB.

Figure 4 depicts the verification time required for both the direct and the composition methods for each of the intermediate models

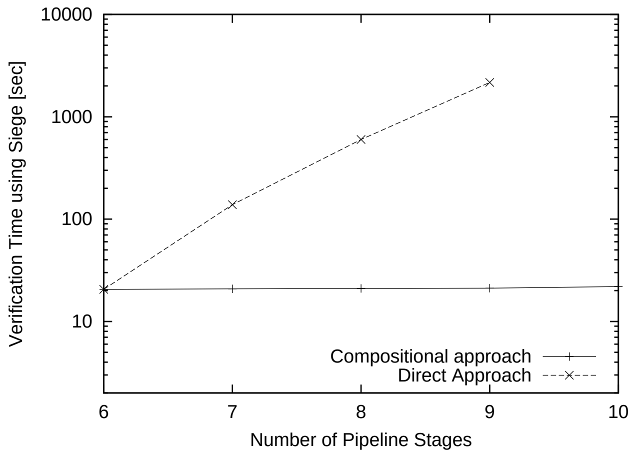

Figure 1: Comparison of direct and compositional approach for the deep pipelined example.

and for the final 10 stage pipeline. As can be seen from Figure 4, the verification cost is exponential (the y-axis is a logarithmic scale) for the direct method for each new pipeline stage, whereas, for the composition technique, the verification cost is almost a constant. Notice that the state-of-the- art SAT solver Siege failed to produce a result when applying the direct method to the 10 stage pipelined machine, and the composition technique required just 22 seconds for this example. Siege has been shown to have a speedup of a 20 over the Chaff SAT solver [11].

5. VERIFICATION OF INSTRUCTION CACHE, DATA CACHE AND WRITE BUFFER

We use the compositional property of WEB-refinement to verify a complex processor model in stages. We compare the compositional approach to verifying the processor model using the commitment approach. The model has 7 stages and features such as instruction cache, data cache, and a write buffer. We prove the correctness of the 7 stage model, and then the instruction cache, data cache and write buffer in stages. In the following discussion, we use ISA for the specification, "7S" for 7 stage model, "IC" for the model with instruction cache, "DC" for the model with instruction and data cache, and "WRB" for the model with instruction cache, data cache and write buffer.

We model a direct mapped instruction and data cache. The instruction cache is modeled using three memory elements `ICache-Valid`, `ICache-Tag`, and `ICache-Block` that take an index as input and returns a predicate indicating if the entry in the instruction cache is valid, the tag, and the data block, respectively. Three uninterpreted functions *GetIndex*, *GetTag* and *GetBlockOffset*, that take the program counter as input, are used to obtain the index, tag, and the block offset, respectively. Another uninterpreted function *SelectWord* is used to extract the instruction from the data block. The instruction memory is modeled as a lambda expression that takes 2 arguments, an index and a tag, and returns a block of data. This way of modeling the instruction memory allows the us to match up the contents of the instruction memory and the instruction cache.

The state components of IC and 7S are identical except that IC has an instruction cache. Therefore 7S and IC do not stutter with respect to one another. Let 7S be MA1 and IC be MA2. The WEB-

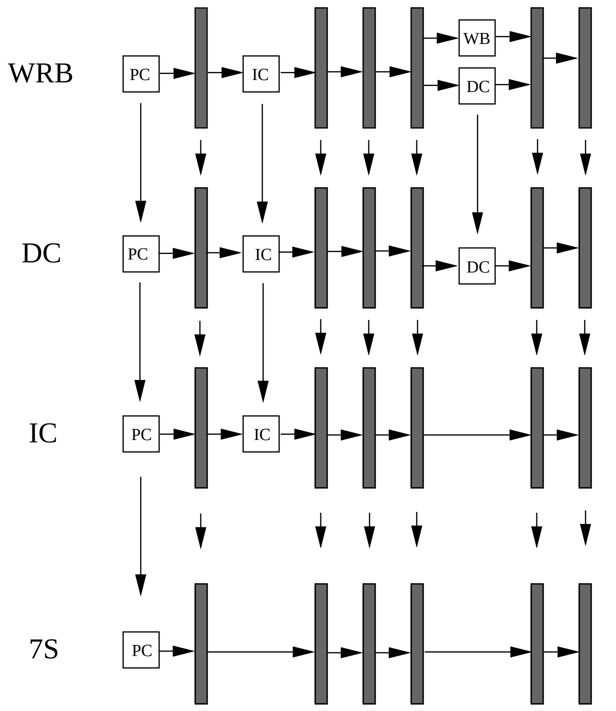

Figure 2: Refinement map projecting WRB on DC, DC on IC, IC on 7S.

refinement proof reduces to:

$$s = r(w) \ \wedge \ u = \texttt{MA1-step}(s) \ \wedge \ v = \texttt{MA2-step}(w) \implies u = r(v)$$

where, s and u are MA1 states, w and v are MA2 states, r is the refinement map from MA2 to MA1, and `MA1-step` and `MA2-step` are the functions corresponding to stepping MA1 and MA2, respectively. The above theorem is applicable to any MA1 and MA2 that do not stutter. All the state components, including the program counter, data memory, register file and pipeline latches, in IC are identical to the state components in 7S, except that IC has an instruction cache. Therefore, a refinement map from IC to 7S can be obtained by dropping the instruction cache state and retaining all other state components. We use the following invariant.

$$\texttt{ICache-Valid}(I) \ \wedge \ \texttt{ICache-Tag}(I) = T \implies \texttt{ICache-Block}(I) = \texttt{IMemory}(I,T)$$

In the above formula, I is an arbitrary index value and T an arbitrary tag value. The invariant states that if I and T are the index and tag for a particular memory address, 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. The idea is that valid instruction cache entries should be consistent with those in 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 IC state w, then it holds for v, where v is obtained by stepping w once.

The data cache is direct mapped and modeled similar to the instruction cache. Writes to the data memory are implemented as write-through, and update the data cache. The proof that DC (the implementation with an instruction and data cache) refines IC is identical to the proof that IC refines 7S. The refinement map from DC to IC is defined by ignoring the data cache state and retaining all other state elements in DC, including the instruction cache. An invariant similar to the instruction cache is required for the data cache that all the valid entries in the data cache are consistent with the data memory.

Table 2: Verification times and statistics for direct and compositional verification of the instruction cache, data cache, and write buffer.

Processor	CNF Vars	CNF Clauses	Verification Time [sec]	
			Siege	Total
C7S	12,495	36,925	29	32.3
CIC	41,486	122,617	98.9	106.4
CDC	70,090	206,701	208.1	230.2
CWRB	101,065	298,780	324.5	360.5
CM-IC	96	229	0.01	0.06
CM-DC	154	379	0.01	0.08
CM-WRB	247	613	0.01	0.17

The write buffer is implemented as a queue and has 4 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. WRB (the implementation with instruction cache, data cache, and write buffer), is identical in structure to DC, other than the write buffer. WRB and DC do not stutter with respect to each other and so the proof obligation that WRB refines DC reduces to the proof obligation for the example with the instruction cache. The refinement map is defined by updating the data memory with valid entries in the write buffer, ignoring the write buffer states and retaining all other state elements including the instruction and data cache states. We require an inductive invariant for the write buffer that the combined state of the write buffer and the data memory is consistent with the state of a data memory that was updated directly by the implementation, without going through a write buffer. A combined data memory and write buffer state can be obtained by updating the data memory with all the valid entries in the write buffer. If D is the data memory in WRB, R, is a memory that is similar to D except that store instructions directly update R instead of moving through the write buffer, U, is the memory state obtained after writing all the valid write buffer entries to D, then the invariant states that $R = U$.

Table 5 presents the results for the composition example with the instruction cache, data cache, and write buffer. Prefix “C” refers to the use of commitment approach as a refinement map, “CM-IC”, “CM-DC”, and “CM-WRB” refer to the composition steps between IC and 7S, DC and IC, and WRB and DC. The experimental setup and tool flow are identical with the previous example. It can be seen from Table 5 that when comparing the verification times, the compositional approach has a speedup of about 11 over the direct approach for WRB.

Probably the most difficult part of the verification effort is understanding bugs. Since the compositional approach reduces the verification problem into simpler subproblems, the debugging process is much simplified. In the example of the instruction cache, one can determine that a bug is not due to the instruction cache simply by noting that another stage of the composition fails. Similarly, if the bug occurs in the composition step involving the cache, then the bug is due to the instruction cache. This is impossible to do when verifying the complex processor in a monolithic fashion. As a concrete example of how compositional verification makes debugging simpler, we note that when trying to verify a buggy variant of the instruction cache (while determining whether a cache hit has occurred, the design did not take into account if the cache block is valid), we found that the counter example generated by UCLID for the direct approach is 4429 lines in size while the counter example generated from the composition step is 390 lines. Obviously the shorter counterexample was much simpler to understand and, consequently, fixing the bug was much easier. This aspect of compositional verification may well be more important than the impovement we get in verification times.

6. CONCLUSIONS AND FUTURE WORK

Our main contribution was to show how to use compositional reasoning to automatically prove that pipelined machine models satisfy the same safety and liveness properties as their corresponding instruction set architecture models. This allowed us to reason about deep pipelines in an efficient way, obtaining exponential savings in verification times over previous monolithic approaches, and, in fact, we can easily verify models that state-of-the-art tools cannot currently handle. As a further example of our compositional method, we showed how to verify a complex pipelined machine with branch prediction, instruction and data caches, and a write buffer, in stages, thereby, reducing the verification time by more than a factor of 10 over current state-of-the-art monolithic approaches. We also showed how to integrate compositional reasoning into the design cycle and how this leads to faster verification times, shorter and clearer counterexamples, and enhanced design understanding by verification engineers. For future work, we plan to apply compositional reasoning to more complex processors.

7. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] Omitted for blind review.

[6] M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://www.cs.utexas.edu/users/moore/-acl2`.

[7] 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.

[8] Omitted for blind review.

[9] Omitted for blind review.

[10] Omitted for blind review.

[11] Omitted for blind review.

[12] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 110–121. Springer-Verlag, 1998.

[13] P. Mishra and N. Dutt. Modeling and verification of pipelined embedded processors in the presence of hazards and exceptions. In *IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems (DIPES'02)*, 2002.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[15] V. A. Patankar, A. Jain, and R. E. Bryant. Formal verification of an ARM processor. In *Twelfth International Conference On VLSI Design*, pages 282–287, 1999.

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

[17] 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/`.

[18] Omitted for blind review.

[19] J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.

[20] Omitted for blind review.