A Computationally Efficient Method Based on Commitment Refinement Maps for Verifying Pipelined Machines.*

Panagiotis Manolios
College of Computing
Georgia Tech, Atlanta, GA 30318

Sudarshan K. Srinivasan
School of Electrical & Computer Engineering
Georgia Tech, Atlanta, GA 30318

Abstract

We introduce a new method of automating the verification of term-level pipelined machine models that is based on commitment refinement maps. Our method is much simpler to implement than current alternatives. More importantly, as our extensive experiments show, our method leads to more than a 30-fold improvement in verification times over the standard approaches to pipeline machine verification, which use refinement maps based on flushing and commitment. In addition, we can verify machines that are too complex to directly verify using flushing-based refinement maps.

1. Introduction

In this paper, we describe a new, automatic method for defining commitment refinement maps that provide over a 30-fold improvement in verification times over both the previous method for defining commitment-based refinement maps and the standard method for defining flushing-based refinement maps. We define 42 processor models of varying complexity and, with extensive profiling, show that proving the invariant accounts for almost all of the verification time required when using the standard commitment-based refinement maps. Based on this observation, we introduce a new invariant that can be used for commitment-based refinement proofs. Not only does the new invariant lead to an average speedup factor of about 30, but it is also *much* simpler to define, leading to a decrease both in the human effort required to verify pipelined machines and in the code size.

We automatically and efficiently verify the pipelined machines described in this paper by showing that they have exactly the same infinite executions as the machines defined by the corresponding instruction set architectures, up to stuttering. This is accomplished by constructing a WEB-refinement proof, which implies that the pipelined machine satisfies exactly the same CTL$^* \setminus \mathsf{X}$ properties satisfied by the instruction set architecture [13]. Thus, we verify both safety and liveness properties of the pipelined machine models we study. Automation is attained by expressing the WEB-refinement proof obligation in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU), which is a decidable logic [5]. We use the tool UCLID [11] to transform the CLU formula into a CNF (Conjunctive Normal Form) formula, which we then check with the SAT solver Siege [24]. The pipelined machine models that we verify are described at the *term-level*, where the data path is abstractly represented using integers and many of the combinational circuit blocks are abstractly represented using uninterpreted functions. The pipeline control logic, however, is described in detail.

The refinement proofs depend on a critical parameter: the refinement map, a function that relates pipelined machine states to instruction set architecture states. Refinement maps tend to be complex functions that in essence step the pipelined machine multiple times, in order to force it into a state where all the pipeline latches are invalid, so that an instruction set architecture state can be obtained by projecting out the programmer visible components. For this reason, the refinement maps used can have a drastic impact on verification times. The refinement maps we consider are based on commitment [12, 13] and flushing [6]. The idea with commitment is that partially completed instructions are invalidated and the programmer visible components are rolled back to correspond with the last committed instruction. Flushing is a kind of dual of commitment, where partially completed instructions are made to complete without fetching any new instructions. Using refinement maps based on commitment requires the use of invariants, and the main contribution of this paper is the introduction of a new method for defining a suitable invariant that leads to drastic improvements in verification times and is much easier to implement, debug, and maintain.

The paper is organized as follows. In Section 2, we pro-

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

vide an overview of refinement based on WEBs, the theory used for our correctness proofs. In Section 3, we describe the pipelined machine models and verification benchmarks that we use for the experiments. The models and benchmarks are available from the authors. In Section 4, we briefly describe the flushing and commitment refinement maps. In Section 5, we introduce the Greatest Fixpoint (GFP) invariant and show results obtained by applying the commitment refinement map using the GFP invariant on our 42 pipelined machine models. In Section 6, we describe how to use the recently introduced idea of intermediate refinement maps to obtain even faster verification times. Finally, we present related work in Section 7, and conclude in Section 8.

2. 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 [13, 14]. 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`.

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

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

(Stb1) $L.s = L.w$
(Stb2) $\langle \forall \sigma :: fp(\sigma, s) \Rightarrow \langle \exists \delta :: fp(\delta, w) \wedge match(B, \sigma, \delta)\rangle\rangle$

Browne, Clarke, and Grumberg have shown that states that are stuttering bisimilar satisfy the same next-time-free temporal logic formulas [3].

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

We note that stuttering bisimulation differs from weak bisimulation [18] in that weak bisimulation allows infinite stuttering. Stuttering is a common phenomenon when comparing systems at different levels of abstraction, *e.g.*, if the pipeline is empty, `MA` will require several steps to complete an instruction, whereas `ISA` completes an instruction during every step. Distinguishing between infinite and finite stuttering is important, because (among other things) we want to distinguish deadlock from stutter.

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

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

STB refinement is a generally applicable notion. However, since it is based on bisimulation, it is often too strong a notion and in this case refinement based on stuttering *simulation* should be used (see [13, 14]). The reader may be surprised that STB refinement theorems can be proved in the context of pipelined machine verification; after all, features such as branch prediction can lead to non-deterministic pipelined machines, whereas the `ISA` is deterministic. While this is true, the pipelined machine is related to the `ISA` via a refinement map that hides the pipeline; when viewed in this way, the nondeterminism is masked and we can prove that the two systems are stuttering bisimilar (with respect to the `ISA` visible components).

A major shortcoming of the above formulation of refinement is that it requires reasoning about infinite paths, something that is difficult to automate [21]. In [13], 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 [15], it is shown how to automate the refinement proofs in the context of pipelined machine verification. The idea is to strengthen, thereby simplifying, the refinement proof obligation; the result is the following CLU-expressible formula, where *rank* is a function that maps states of `MA` into the natural numbers.

Theorem 1

$$\begin{array}{lll} \langle \forall w \in \mathtt{MA} :: & s = r(w) \ \wedge \ u = \mathtt{ISA\text{-}step}(s) \ \wedge \\ & v = \mathtt{MA\text{-}step}(w) \ \wedge \ u \neq r(v) \\ \quad \Rightarrow & s = r(v) \ \wedge \ rank(v) < rank(w)\rangle \end{array}$$

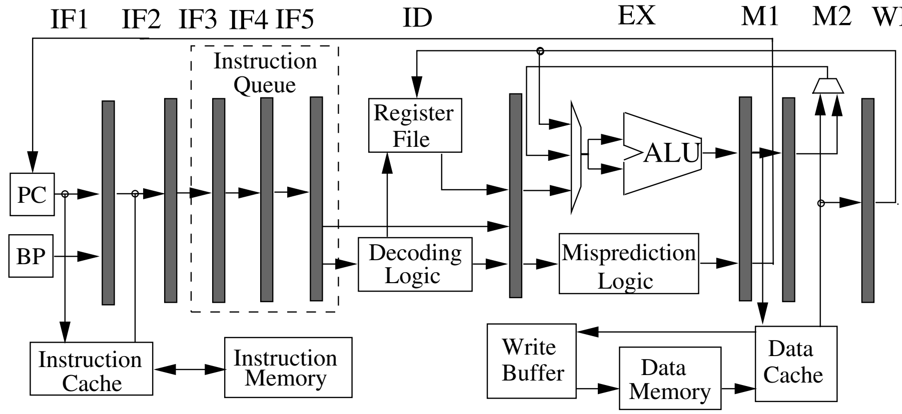

Figure 1. High-level organization of a 10-stage pipelined machine model including all the features we model, including a pipelined fetch stage, a 3-stage instruction queue, branch prediction, instruction and data caches, and a write buffer.

In the formula above s and u are `ISA` states, and w and v are `MA` states; `ISA-step` is a function corresponding to stepping the `ISA` machine once and `MA-step` is a function corresponding to stepping the `MA` machine once. It may help to think of the first conjunct of the consequent ($s = r.v$) as the safety component of the proof and the second conjunct $rank.v < rank.w$ as the liveness component. Also note that for the machines described in this paper the definitions of the step functions can be quite complex, *e.g.*, for some of our examples, `MA-step` takes thousands of lines to define using UCLID.

Note that the notion of WEB refinement is independent of the refinement map used. This is what allows us to compare verification times when using different refinement maps, *e.g.*, when comparing the flushing refinement map [6] with the commitment refinement map [12].

3. Pipelined Machine Models and Benchmarks

For our experiments, we have created 42 pipelined machine models of varying complexity. We start with a base processor model and extend it with features such as a pipelined fetch stage, a 3-stage instruction queue, two different ways of abstracting branch predictors, an instruction cache, a data cache, and a write buffer. The base processor model is a 6 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). We implemented ALU instructions, register-register and register-immediate addressing modes, loads, stores, and branch instructions. We assign names to the pipelined machine models that are consistent with the names in the "Processor" column of Table 1. The model names start with a number indicating the number of stages followed optionally by the letters "I", "D", "W", "B" and "N" indicating the presence of an instruction cache, data cache, write buffer, branch prediction abstraction scheme 1, and branch prediction abstraction scheme 2, respectively. By applying different refinement maps to the pipelined machine models, we get in all 210 benchmarks (5 verification problems for each pipelined machine model).

The basic features of the pipelined machines are modeled in a style similar to [15], which in turn are similar to [28]. The models are described at the term-level. Word-level values are abstracted using terms or integers and much of the combinational circuit blocks that are common between the pipelined machine and its ISA are abstracted using Uninterpreted Functions (functions that only satisfy the property of functional consistency). We use restricted lambda expressions to model memories. The caches and write buffer are modeled as described below.

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 the index as input and return 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* take the program counter as input, and 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 us to relate the contents of the instruction memory with the instruction cache contents. We re-

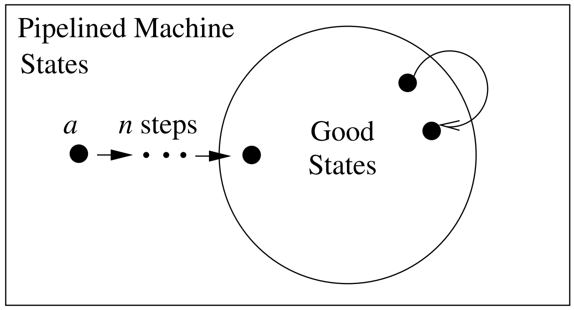

Figure 2. The Greatest Fixpoint Invariant (GFI) characterizes the set of states that can be reached in *n* steps from some pipelined machine state.

quire an invariant about the instruction cache that valid entries in the cache are consistent with 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, w, then it holds for v, where v is obtained by stepping w once.

The data cache is direct mapped and the way we model it is similar to the way we model the instruction cache. Writes are write-through and update both the data cache and memory. Also, an invariant stating that all the valid entries in the data cache are consistent with the data memory is required.

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. We require an inductive invariant for the write buffer that states that the combined state of the write buffer and the data memory is consistent with a data memory that is updated directly, without using a write buffer.

4. Flushing and Commitment

Burch and Dill in [6] showed how to automatically use flushing to define refinement maps. The idea with flushing is that partially executed instructions in the pipeline latches are made to complete and update the programmer visible components without fetching any new instructions. The programmer visible components are the program counter, the register file, and the instruction and data memories. After the pipelined machine is flushed, the pipeline latches are all invalid, and an ISA state is obtained by projecting out the programmer visible components. We also need a rank function, which we define as the number of steps required to fetch and complete a new instruction.

The commitment refinement map was first introduced in [12] and automated in [15]. Commitment can be thought of as the dual of flushing, as partially executed instructions in the pipeline are invalidated instead of being completed, and the programmer visible components are rolled back to correspond to the last committed instruction. We store some history information to simplify the implementation of the refinement map. The rank function is defined as the length, in latches, from the end of the pipeline to a valid latch.

For the commitment approach, we require an inductive invariant on the pipelined machine states. One such invariant is the “Good MA” invariant: it states that a pipelined machine state is good if it can be reached within a bounded, machine-dependent number of steps, say n, from a committed state. A committed state is a state in which all the pipeline latches are invalid. All states that are within n steps from a committed state clearly satisfy the invariant, so our proof obligation reduces to showing that the successor of any state that is n steps away from a committed state satisfies the invariant. As shown in Table 1, it turns out that the invariant proof is computationally expensive, accounting for more than 98% of the verification time required by the commitment approach.

5. Greatest Fixpoint Invariant

We introduce a new inductive invariant that can be used with commitment-based refinement maps; we call it the Greatest Fixpoint (GFP) invariant. In this section, we define the GFP invariant and compare, based on proof times and the ease of implementation, the commitment approach that uses the GFP invariant, with the flushing approach and the commitment approach that uses the “Good MA” invariant.

The definition of the the Greatest Fixpoint (GFP) invariant is straightforward.

Processor Model	Flushing		Commitment (Good MA)				Commitment (GFP)	
	CNF Vars	Ref. Proof Time (sec)	Inv. Proof Time (sec)	Ref. Proof Time (sec)	Total CNF Vars	Total Time (sec)	CNF Vars	Ref. Proof Time (sec)
6	28,256	20	23	1	12,334	24	9,498	3
6I	48,917	22	172	4	36,498	176	16,955	5
6ID	114,124	202	414	20	75,405	434	29,089	10
6IDW	159,620	458	438	35	80,434	473	34,107	16
7	53,165	168	24	1	13,296	25	17,528	13
7IDW	263,022	1,012	723	95	105,313	818	55,182	40
8	95,092	630	47	1	14,100	48	27,107	43
8IDW	393,719	2,995	1,054	156	131,364	1,210	96,710	147
9	144,045	1,394	24	1	15,214	25	39,346	100
9IDW	526,651	Fail	1,754	98	161,759	1,852	125,585	265
10	198,375	3,841	30	1	17,121	31	55,763	164
10I	293,862	4,752	1,325	8	82,795	1,333	91,416	384
10ID	580,355	Fail	2,685	126	195,562	2,811	159,638	540
10IDW	690,598	Fail	2,885	88	197,258	2,973	174,122	536
6B	37,002	14	89	1	21,850	90	13,495	4
6BI	63,824	23	1,423	11	51,114	1,434	23,891	8
6BID	137,935	283	2,968	167	100,406	3,135	40,371	17
6BIDW	191,101	439	2,851	229	105,639	3,080	45,567	25
7B	70,985	216	232	1	26,058	233	25,676	22
7BIDW	311,425	1,627	5,537	579	144,441	6,116	76,820	70
8B	121,645	733	701	1	31,914	702	40,559	150
8BIDW	424,604	5,376	30,438	1,043	177,741	31,481	122,550	304
9B	183,371	1,737	686	2	36,757	688	59,110	674
9BIDW	628,179	Fail	58,029	876	230,500	58,905	173,479	1,145
10B	256,272	4,563	1,555	2	43,517	1,557	81,569	1,756
10BI	371,249	4,706	73,710	299	126,785	74,009	136,545	1,780
10BID	695,833	Fail	160,523	926	276,289	161,449	221,420	4,431
10BIDW	824,633	Fail	233,928	1,193	278,137	235,121	237,485	6,039
6N	37,452	19	101	1	37,147	102	12,631	4
6NI	63,563	23	878	8	95,821	886	23,229	8
6NID	137,885	282	3,599	51	161,995	3,650	40,132	18
6NIDW	190,399	428	3,472	267	163,763	3,739	45,259	23
7N	70,667	188	240	1	27,500	241	23,936	14
7NIDW	310,434	1,679	11,103	307	162,225	11,410	75,496	73
8N	121,499	499	794	1	53,697	795	39,165	140
8NIDW	424,124	5,968	34,423	433	259,031	34,856	12,1170	270
9N	185,149	2,027	970	2	62,536	972	54,631	447
9NIDW	626,884	Fail	75,453	417	350,587	75,870	170,918	899
10N	255,780	4,910	2,136	2	73,163	2,138	75,676	1,938
10NI	368,888	4,544	51,514	493	224,692	52,007	131,642	2,101
10NID	698,555	Fail	225,636	4,455	414,530	230,091	217,725	4,229
10NIDW	824,210	Fail	286,285	3,479	416,378	289,764	233,852	6,155

Table 1. Verification statistics for the flushing approach, the commitment approach using the "Good MA" invariant, and the commitment approach using the Greatest Fixpoint invariant for various pipelined machines.

Definition 3 gfp.w *iff* $\langle \exists a \in S :: a \dashrightarrow_n w \rangle$

In the above definition, S is the set of all pipelined machine states, $\dashrightarrow$ is the transition relation, and $\dashrightarrow_n$ is the n-fold composition of $\dashrightarrow$ (*i.e.*, it relates u to v if v can be reached in n steps from u). The definition states that a pipelined machine state w is in the invariant if it can be reached from some state in n steps. Reasonable values of n depend on the pipelined machine in question and should be selected to correspond to the minimum number of steps required to replace all the partially executed instructions in the pipeline with instructions fetched from the instruction memory. For the pipelined machine models that we consider, n is the number of steps required to flush the machine.

The reason we call this invariant the greatest fixpoint invariant is that we have the following lemma.

Lemma 2
$\langle \forall k \in \mathbb{N} :: \langle \exists a \in S :: a \dashrightarrow_{k+1} w \rangle \Rightarrow \langle \exists a \in S :: a \dashrightarrow_k w \rangle$

Therefore, for the sequence of sets $S_0, \ldots, S_n$, where $S_i = \{w \in S :: \langle \exists a \in S :: a \dashrightarrow_i w \rangle\}$, we have $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_n$.

The GFP invariant is depicted in Figure 2. If an arbitrary pipelined machine state is stepped for the number of steps required to flush the pipeline, then all the partially executed instructions in the pipeline are made to complete and update the programmer visible components, and the pipeline latches are filled with new instructions. For the machines we consider, the flow of an instruction in the pipeline depends only on older instructions in the pipeline. Therefore, all the instructions in pipeline latches of the original arbitrary state are guaranteed to complete and be replaced by new instructions from the instruction memory. The new partially executed instructions in the pipeline latches of the resulting state will be consistent, thereby avoiding the problems inherent in the commitment approach. That GFP is an invariant follows from the following lemma.

Lemma 3 $\langle \forall w, v \in S :: (\text{gfp}.w \wedge w \dashrightarrow v) \Rightarrow \text{gfp}.v \rangle$

The lemma is true by definition. Recall that checking the standard invariant used for the commitment approach is where most of the verification time is spent, but by the above lemma, no such check is required for the GFP invariant approach. As we show in the next section, this leads to drastically faster verification times.

The commitment refinement map using the GFP invariant is defined as follows. A pipelined machine state satisfying the GFP invariant is committed by invalidating the partially executed instructions in the pipeline and rolling back the programmer visible components (program counter, instruction and data memory, and register file) so that they correspond with the last committed instruction. The programmer visible components are then projected out, resulting in an ISA state. The rank function is the same as the one used for the "Good MA" commitment approach (*i.e.*, it is

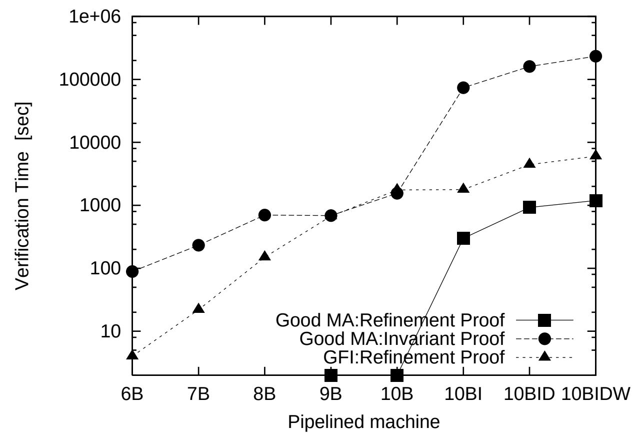

Figure 3. The invariant and refinement proof times for the "Good MA" commitment approach and the refinement proof times for the GFP commitment approach for pipelined machine models with increasing complexity.

the length, in latches, from the end of the pipeline to a valid latch).

5.1. Results and Analysis

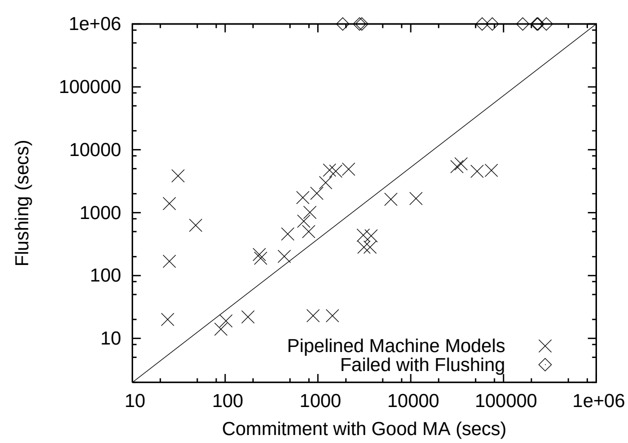

Figure 4. A comparison of the verification times required for our benchmark problems between commitment using the "Good MA" invariant and flushing.

We used flushing, the commitment approach with the "Good MA" invariant, and the commitment approach with the GFP invariant to verify the 42 pipelined machine mod-

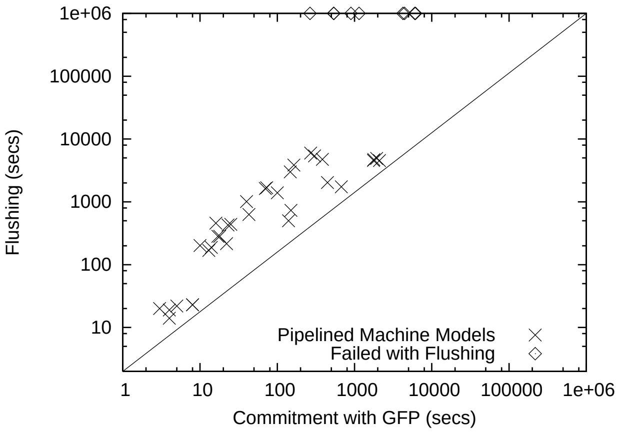

Figure 5. A comparison of verification times required for our benchmark problems between commitment using the GFP invariant and flushing.

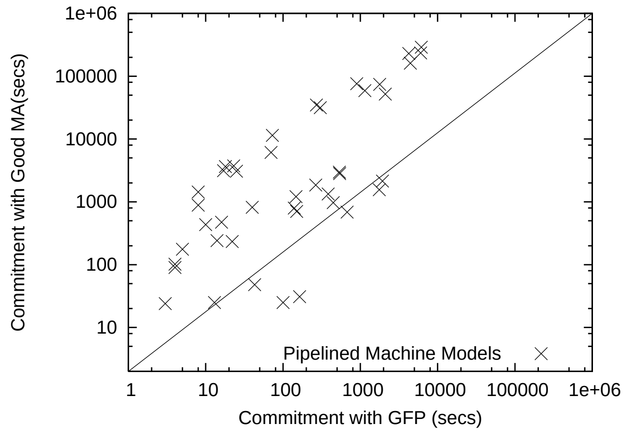

Figure 6. A comparison of verification times required for our benchmark problems between commitment using the GFP invariant and commitment using the "Good MA" invariant.

els described in Section 3. For all experimental results presented in this paper, we used the UCLID decision procedure (version 1.0) coupled with the Siege SAT solver [24] (variant 4), using a 3.06 GHz Intel Xeon, with an L2 cache size of 512 KB. The results should be interpreted taking into consideration the following two factors. First, the Siege SAT solver uses a random number generator and large variations in the running times are possible, *e.g.*, in previous work we noticed that the standard deviation of the Siege running times can be significant [15]. Second, the machines we used for the experiments are part of a public cluster. While we tried to use idle machines, the running times we obtained could have been slightly influenced by other jobs running on the machines.

Table 1 shows the verification times and related statistics for the various pipelined machine models. The names in the "Processor" column start with a number indicating the number of stages followed optionally by the letters "I", "D", "W", "B", and "N" indicating the presence of an instruction cache, data cache, write buffer, branch prediction abstraction scheme 1, and branch prediction abstraction scheme 2, respectively. Branch prediction abstraction schemes 1 and 2 refer to two different ways of abstracting branch predictors. For all the three approaches we report the time taken by both UCLID and Siege to complete the refinement proof. For the commitment approach based on the "Good MA" invariant, we also report the time taken by both UCLID and Siege for the invariant proof and the total time for both the refinement proof and the invariant proof. A "Fail" entry indicates that Siege failed on the problem (by immediately reporting that the problem is too complex and quiting).

Figure 3 shows the running times for the refinement proof and the invariant proof for the "Good MA" approach and the refinement proof for the GFP approach, as the complexity of the pipelined machine models increases. An interesting observation is that more than 98% of the total proof time for the "Good MA" approach is spent in proving the invariant. This motivates the use of the Greatest Fixpoint (GFP) invariant, which is computationally less expensive.

Figures 4, 5, and 6 are scatter plots with log scales for both axes and compare the use of commitment ("Good MA") vs. flushing, commitment (GFP) vs. flushing, and commitment (GFP) vs. commitment ("Good MA"), respectively. The comparison is based on running times for verifying 42 pipelined machine benchmarks. From Figure 4, it can be seen that flushing and commitment based on "Good MA" have similar performance characteristics on the models that flushing completes. But, flushing fails to produce a result on 9 of the more complex benchmarks. Commitment ("Good MA") scales better than flushing, but the verification times for the more complex benchmarks reach over 250,000 seconds. Figure 5 shows that the commitment based on GFP does better than flushing on all the 42 benchmarks.

From Figure 6, it can be seen that commitment based on the GFP invariant does better than commitment based on the "Good MA" invariant for most of the benchmarks. We note that the time required for the refinement proofs of the two commitment approaches differs. From Figure 3 and Table 1, we see that the time for the refinement proofs for the GFP approach is much higher. The difference can be explained by noting that once the invariants are proved, the sets of states satisfying the invariants are defined as the set of states reachable from an initial state after some number of

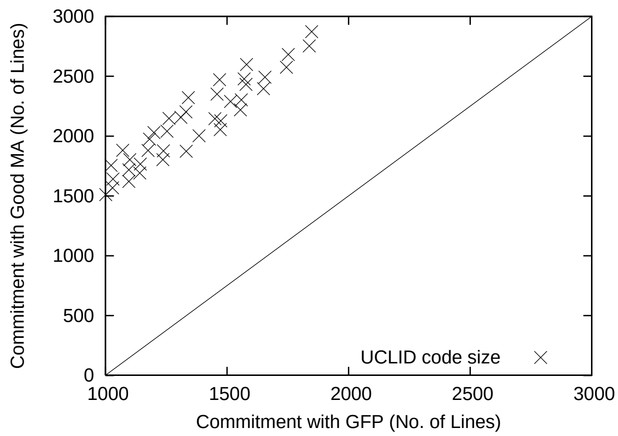

Figure 7. A comparison of the size of the UCLID specifications required for our benchmark problems between commitment using the GFP invariant and commitment using the "Good MA" invariant.

steps. The maximum number of such steps required for the two approaches depends on the number of steps required for a newly fetched instruction to reach the end of the pipeline. For the "Good MA" approach, the initial state is a committed state (this is an advantage of using the "Good MA" invariant), meaning that all the pipeline latches are initially invalid. Therefore, the flow of the first newly fetched instruction is uninhibited (*e.g.*, it cannot stall) and depends only on the length of the pipeline. For example, for the 10 stage pipeline models, starting from a committed state, 9 steps of the pipelined machine are required for a newly fetched instruction to reach the end of the pipeline. In contrast, when using the GFP approach, the initial state is an arbitrary state and the flow of the first newly fetched instruction in the pipeline depends on the older instructions in the pipeline. For example, if the first newly fetched instruction is data dependent on older instructions in the pipeline, it will stall. For the 10 stage pipeline, starting from an arbitrary state, a maximum of 14 steps of the pipelined machine is required for a newly fetched instruction to reach the end of the pipeline. As a rule of thumb, verification times increase exponentially with the number of symbolic simulation steps required, therefore, the refinement proof times for the GFP approach is much higher than for the "Good MA" approach. Of course, if we look at the total time required for verification, the GFP approach is the clear winner because it does not require us to prove an invariant.

One further important observation is worth making and that is that the GFP approach is much easier to implement than the "Good MA" approach, because (in contrast to the "Good MA" approach) we do not require an extra invariant proof. Such proofs require that we symbolically simulate a pipelined machine state in two different ways and check that that results are equal. Figure 7 is a scatter plot that compares the size of the UCLID specifications for the two commitment based refinement maps for each of the 42 pipelined machine models. The UCLID specifications consist of the machine model and the refinement map. As can be seen from the figure, the UCLID specifications that use the "Good MA" approach are much larger than those that use the GFP approach. Further, once the machine models are defined, implementing the commitment refinement map based on the GFP approach required about a couple of hours while the implementation of the "Good MA" approach took more than twice as long.

6. Using GFP with Intermediate Refinement Maps

Intermediate refinement maps, an approach that combines flushing and commitment and leads to drastic reductions in verification times, was proposed in [16]. Intermediate refinement maps are obtained by selecting a pipeline stage, which we will call the reference point, committing all pipeline latches before the reference point, and flushing all latches after the reference point. The rank function for these refinement maps returns a pair of natural numbers $rank_c$ and $rank_f$, which are exactly the same rank functions for commitment and flushing. The less-than ordering on the rank is the component-wise order. An invariant is required for the part of the pipeline being committed, but is not required for the part of the pipeline being flushed.

For any given pipelined machine, many intermediate refinement maps can be defined by selecting different stages in the pipeline as the reference point. The fastest verification times are obtained when selecting a reference point that is close to the middle of the pipeline. We implemented the intermediate refinement map IR4 that commits the first 4 pipeline latches and flushes all other latches, for the most complex processor model with branch prediction scheme 1 (10BIDW) using both the old and the new characterization of commitment. For the new characterization of commitment, we used the GFP invariant to characterize the set of reachable states by stepping the pipelined machine for i steps, where i is the number of steps required to replace the instructions in the pipeline latches being committed with new instructions from memory. The rank function for the intermediate refinement map defined using the GFP invariant is the same as the rank function for the intermediate refinement map defined using the "Good MA" invariant.

The experiments were conducted using the same experimental set up (tools and machines) described in Section 5.1. We found that the verification time for 10BIDW using IR4 defined with the commitment approach based on "Good

MA" invariant and the GFP invariant to be 3,500 seconds and 550 seconds, respectively. Note that with using only the commitment approach ("Good MA"), the verification time for 10BIDW is 235,121 seconds. Thus, the GFP invariant can be fruitfully combined with intermediate refinement maps to get verification times that are about 6 times faster than the previous approach, for the most complex processor model (10BIDW).

7. 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 [25, 27, 26]. Another example of a theorem proving approach is the work by Hosabettu et al., who use the notion of completion functions [8].

Our main concern, however, is with highly automatic methods. An early and influential paper in this area is due to Burch and Dill, who showed how to automatically compute refinement maps using flushing [6] and gave a decision procedure for the logic of uninterpreted functions with equality and Boolean connectives. The idea with flushing is that a pipelined machine state is related to an instruction set architecture state by completing partially completed instructions without fetching any new instructions. Another refinement map that can be automatically computed is based on the commitment approach [12, 15], where a pipelined machine state is related to an instruction set architecture state by invalidating all the partially executed instructions in the pipeline and rolling back the programmer-visible components so that they correspond with the last committed instruction. There has been recent work on commitment [23, 2], and on the use of refinement maps that partly flush and partly commit [16].

Different types of automatic methods have been used, *e.g.*, McMillan uses model-checking and symmetry reductions [17]; Patankar et al. use Symbolic Trajectory Evaluation (STE) to verify a processor that is a hybrid between ARM7 and StrongARM [22]; and SVC is used to check the correct flow of instructions in a pipelined DLX model [19]. Aagaard et al., in [1] describe a survey of various pipelined machine correctness statements. There has also been related work based on assume-guarantee reasoning by Henzinger et al. [7].

More directly related to this paper is the work on decision procedures for boolean logic with equality and uninterpreted function symbols [4]. The results in [4] were further extended in [5], 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 [11] and which we use to verify the models presented in this paper.

The notion of correctness for pipelined machines that we use was first proposed in [12], and is based on WEB-refinement [13]. The first proofs of correctness for pipelined machines based on WEB-refinement were carried out using the ACL2 theorem proving system [9, 10]. The advantage of using a theory of refinement over using the Burch and Dill notion of correctness, even if augmented with a "liveness" criterion is that, deadlock may avoid detection with the Burch and Dill approach [12], whereas it follows directly from the WEB-refinement approach that deadlock (or any other liveness problem) is ruled out. In [15], 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 tends to outperforms Chaff [20] for such problems, 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 using a new characterization of the commitment refinement map.

8. Conclusion

We have introduced a new method for automatically verifying pipelined machines using commitment-based refinement maps. Our method is based on a greatest fixed-point characterization of the commitment invariant. We defined 210 benchmark verification problems and 42 processor models, which we used to compare our method with previous approaches. Our results clearly show that our new method is easier to define and automate, and gives rise to more than a 35-fold reduction in verification times over the standard approach to verifying commitment-based refinement maps. We noticed a similar improvement over flushing, although the standard flushing approach was not able to complete the verification of 9 of the 42 flushing benchmarks. We also showed that further improvements in verification times are possible by using the recently introduced notion of intermediate refinement maps. The verification engines we used are the UCLID decision procedure and the Siege SAT solver. For future work, we plan to apply our method of defining commitment maps to a wider class of pipelined machines, and we will continue to explore methods for reducing the verification times of refinement based approaches to pipelined machine verification.

References

[1] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for microprocessor correctness statements. In

Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), volume 2144 of *Lecture Notes in Computer Science*, pages 433–448. Springer, 2001.

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] 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.

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

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

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

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

[16] P. Manolios and S. Srinivasan. Refinement maps for efficient verification of processor models. In *Design Automation and Test in Europe, DATE'05*, 2005. To appear.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

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

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

[25] 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/`.

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

[27] 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.

[28] S. K. Srinivasan and M. N. Velev. Formal verification of an Intel XScale processor model with scoreboarding, specialized execution pipelines, and imprecise data-memory exceptions. In *Formal Methods and Models for Codesign (MEMOCODE'03)*, pages 65–74, 2003.