Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-Order Processor Verification

Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zhu

April 20, 1998

CMU-CS-98-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted for FMCAD'98

This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.

Keywords: out-of-order execution, automatic verification, temporal logic, symbolic model checking, theorem proving

Abstract

We present a new approach to the verification of hardware systems with data dependencies using temporal logic symbolic model checking. As a benchmark we take Tomasulo's algorithm [HP96] for out-of-order instruction scheduling. Our approach is similar to the idea of uninterpreted function symbols [BD94]. We use symbolic values and instructions instead of concrete ones. This allows us to show the correctness of the machine independently of the actual instruction set architecture and the implementation of the functional units. Instead of using first order terms as in [BD94], we represent symbolic values with a new compact encoding. In addition, we apply some other reduction techniques to the model. This significantly reduces the state space and allows the use of highly efficient symbolic model checkers like SMV instead of special decision procedures. The correctness of the method has been proven formally with the PVS theorem prover.

1 Introduction

Modern microprocessors are becoming extremely complicated, involving superscalar pipelines and *out-of-order execution* (OOO). This complexity increases the demand for fast but reliable validation methods to ensure timely delivery of the product with as few errors as possible. Formal verification is the most precise technique that can guarantee the correctness of the design. However, the growing complexity of microprocessors makes formal verification increasingly difficult because data and control flow are tightly coupled. A formal model has to capture all of the data dependencies. As a result, the state space may become enormous. Straightforward model checking techniques [CE81, CES86] can not handle this complexity because of the state explosion problem. Theorem proving [DP97, SH98] alone usually involves significant manual effort. Moreover, the proofs are too tedious to be easily manageable. Symbolic execution using uninterpreted function symbols [BD94] is based on extensive term rewriting and simple proof-theoretic reasoning, and thus, can be easily automated. However, when the circuit becomes too large, each cycle in the symbolic execution produces large formulas. In addition, the number of cycles required to complete the verification grows as well.

Usually the terms that appear during symbolic execution are not arbitrary. Often, they share subterms and have similar structure. We introduce a special representation for such terms that reduces the number of copies of identical subterms. This representation is based on a data structure called the *reference file*. Terms that share common subexpressions simply have references to the same entries in the reference file. This greatly reduces the memory requirements and also simplifies the problem of checking equivalence between terms — we simply compare the references. We propose to use symbolic model checking techniques [BCM92, McM93] to perform the actual symbolic execution of the circuit. The correctness of the method has been proven in the PVS theorem prover [SOR93].

Recently there has been done a lot of work on the verification of superscalar microprocessors both with and without OOO execution. Burch and Dill [BD94] use the notion of *uninterpreted functions* to represent data and instructions symbolically. Interpreting these symbols results in a particular run of the concrete processor being verified. All of the formulas with uninterpreted function symbols that can be proven remain true under arbitrary interpretations. Thus, their approach can prove the correctness of a device regardless of the concrete implementation. This approach, however, requires special decision procedures for uninterpreted function symbols and does not use previously existing techniques like BDDs [Bry86]. Skakkebæk et al. [SJD98] propose an incremental flushing technique to verify an OOO design. Their approach is also based on uninterpreted function symbols and uses the SVC tool as a decision procedure. Our approach does not require any special decision procedures. We only need a powerful symbolic model checker.

Sajid et al. [SGZ+98] have extended the decision procedures for uninterpreted function symbols to use BDDs. However, their work shares the disadvantage of [BD94] that their decision procedure can not be easily combined with symbolic model checking. Moreover, they do not have a notion of fairness nor can their method verify more involved temporal properties. They have also not investigated how their techniques can be applied to the verification of OOO.

Hojati and Brayton [HB95] have developed a formal description technique for integer combinational/sequential (ICS) systems. Their technique is even more general than uninterpreted functions ([BD94]). For a restricted class of models they show, by applying the notion of data independence [Wol86], that 0-1 instantiation can reduce the size of the model to a finite number of states. This allows the usage of efficient symbolic model checkers. However, OOO is inherently data dependent and thus these abstraction techniques can not be applied. [HB95] also investigates how approximate reachability analysis can be performed for general ICS models. They give an algorithm that makes use of BDDs, but it is only used for reachability analysis and no limit on the number of terms occurring in the verification can be given. Our approach does not require a new algorithm, and the number of terms that need to be considered can be calculated a priori. Therefore, our model can be represented very efficiently in previously existing symbolic model checkers.

McMillan in [McM98] has used model checking to verify a variant of Tomasulo's algorithm (see Section 2). However, his model has only one functional unit that computes only one concrete operation (integer addition). The main idea in his paper is to use compositional reasoning to reduce the complexity of the verification. This part is orthogonal to our approach and can be combined with our representation of uninterpreted functions. He also exploits symmetry to reduce the number of reservation stations, registers and their width. However, because the functional unit is modeled explicitly, his approach is unsuitable for verifying examples with a large number of instruction types or complex functional units (e.g. for operations like integer multiplication). He also does not consider liveness, which is also the case for [DP97]. Our representation makes it possible to abstract away the concrete instructions and functional units. Moreover, we have developed a number of new transformations in addition to the classical symmetry reduction [CJ95] that significantly reduce the state space. In particular, these reductions enable us to verify liveness easily.

The approaches of Burch & Dill [BD94] and McMillan [McM98] are both valuable for verifying out-of-order execution algorithms. However, they both have major limitations. The decision procedures of Burch & Dill can not be used directly with symbolic model checking. McMillan's work, on the other hand, does use symbolic model checking techniques. But he does not use uninterpreted function symbols, and therefore, is unable to handle arbitrary instructions. Our approach does not suffer from these disadvantages. It includes the benefits of both together with a powerful new technique for symbolically representing the contents of registers. We believe that in order to verify realistic superscalar out-of-order designs automatically all of these techniques will be necessary.

Our paper is organized as follows: In Section 2 an overview of Tomasulo's algorithm is given. Section 3 presents the basic abstraction technique. Tomasulo's algorithm with reference file is presented in Section 4. Section 5 explains the overall strategy that we use for verifying the OOO algorithm. The next section describes other abstraction techniques that allow us to speed up the verification. Section 7 shows some experimental results. The paper concludes in Section 8 with a discussion of some directions for future research.

2 Out-of-Order Execution

As benchmarks for our method we have verified several configurations of our implementation of Tomasulo's algorithm [HP96]. This algorithm is the basic technique that is used to implement OOO in modern microprocessors. It also has been used in previous work on the verification of OOO [DP97, McM98]. We will explain how OOO works and how Tomasulo's algorithm implements it. At the end of the section we briefly describe our model.

To achieve greater throughput of instructions, superscalar microprocessors use several functional units that can operate in parallel. However, if two instructions depend on each other (e.g. the later needs the result of the first instruction) one of them has to wait until the other has finished. In this case one functional unit is idle. But if a different instruction, potentially following the other two in the instruction sequence, does not depend on their results, then it can be executed on the free functional

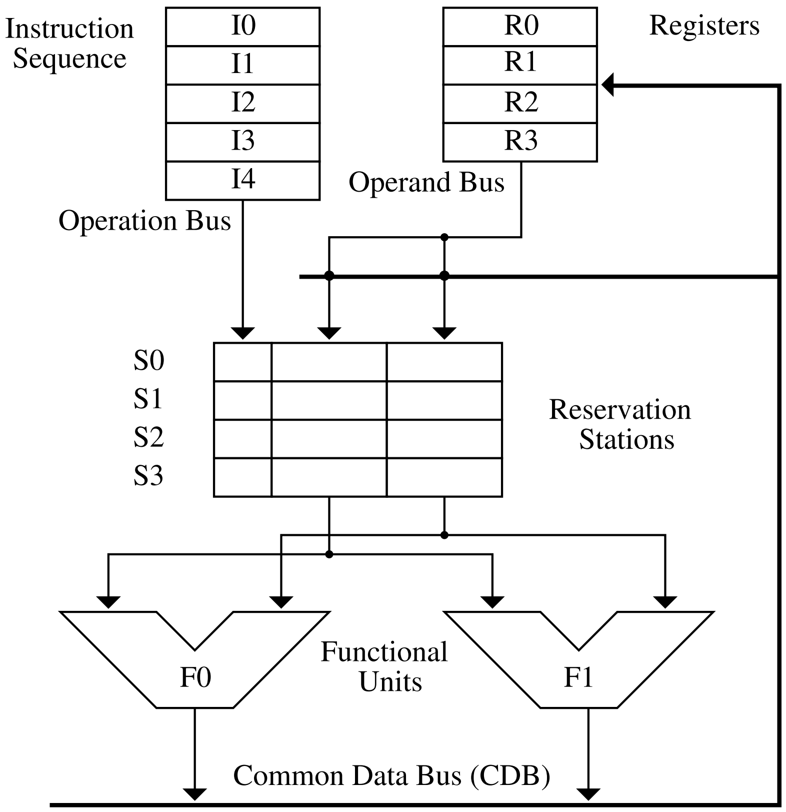

Figure 1: A model of an implementation of Tomasulo's Algorithm.

I0	R0 := R0 $*$ R1
I1	R0 := R0 $*$ R1
I2	R1 := R1 + R1

Figure 2: An instruction sequence that allows OOO.

unit. This is the main idea behind OOO.

For instance, in Figure 2 instruction I1 depends on the result computed by I0, and I2 does not have to wait for I0 or I1. This allows I2 to be executed in parallel to I0. Since a multiplication can take much longer than an addition, the execution of I2 could have finished before I0 has finished and I1 has started. In this case R1 is updated with the result of I2 before I1 is started. In this example the execution of I1 would read the wrong value from register R1. This is a data hazard (write after read) and has to be handled properly. Tomasulo's algorithm is designed to avoid such problems.

The main idea behind Tomasulo's algorithm is to *dispatch* instructions from the instruction sequence into a pool of *reservation stations* (Figure 1). From a reservation station it schedules an instruction for execution on an unoccupied functional unit as soon as all operands are available. To avoid data hazards two additional mechanisms are needed. First in addition to the operator, the operands are also stored in the reservation station. This avoids write after read hazards that are explained in the previous paragraph. If the value of a register is not yet computed, the register contains a *tag* that points to the reservation station that will produce this value. Second, since the value of an operand may not be available while dispatching the instruction into the reservation station, the tag of the appropriate register is used instead of the actual value. Read after write hazards are now handled by updating the operands in the reservation stations together with the registers when results are written back.

In the actual hardware implementations of Tomasulo's algorithm tags and values are usually stored separately, and a special flag controls whether the value or the tag should be used. In our model tags and values share the same memory for efficiency. Figure 1 is a high level floorplan of our model. It is similar to the OOO unit described in [HP96]. It consists of a set of registers (the *register file*), a pool of reservation stations, and several functional units. The main difference is that reservation stations are not associated with specific functional units. In our paper we build a pool of reservation stations instead. The Pentium Pro$^{\text{TM}}$ [Gwe95] microprocessor also has a pool of reservation stations. A similar model was used in [McM98].

3 Basic Abstraction Techniques

Symbolic model checking techniques [McM93] have proven to be of great value for the verification of reactive systems. They have made it possible to verify a large number of practical examples, often with enormous state spaces. The techniques are automatic and can generate counterexamples when a system fails to satisfy its specification. A variety of powerful model checkers are now available, and they are becoming widely used in industry. Because of the power of symbolic model checking techniques we decided to investigate whether they could be used in combination with uninterpreted function symbols to verify OOO designs.

The advantage of model checking over theorem proving [DP97, SH98] is that it is much more automatic. But even with the use of symbolic methods like BDDs there is a limit on the size of the models that can be handled. We will show that the direct application of symbolic model checking to the verification of OOO designs is infeasible. To reduce the size of the model enough for symbolic techniques to be applicable, powerful abstraction techniques are needed. In this section we consider three different encodings of a microprocessor. The first two turn out to be impractical, since they do not provide enough reduction of the state space. We introduce the third representation that makes the use of model checking feasible and allows us to verify non-trivial OOO designs.

3.1 Unabstracted Representation

An OOO unit of a microprocessor can be described as a finite state machine. However, the number of states will be very large. Assume that the microprocessor has r registers of width w, s reservation stations, and f functional units. Each reservation station has to be able to save the contents of two registers (one for each operand of the instruction). Each register contains w bits. This leads to a lower bound of $w \cdot (r + 2 \cdot s)$ on the number of bits n needed for encoding a state. For modern microprocessors we can assume $w \geq 32$, $r \geq 16$, $f \geq 6$, $s \geq 2 \cdot f = 12$ and derive $n \geq 32 \cdot (16 + 2 \cdot 12) = 960$. This number of state bits is clearly out of the scope of all currently available model checkers.

3.2 The Brute Force Approach

The most obvious abstraction is to use only a small number of bits for each register. This abstraction technique is described in [Wol86, HB95] and minimizes the size of registers (and reservation stations) containing data. While, in general, this is a conservative abstraction, it assumes *data independence*.

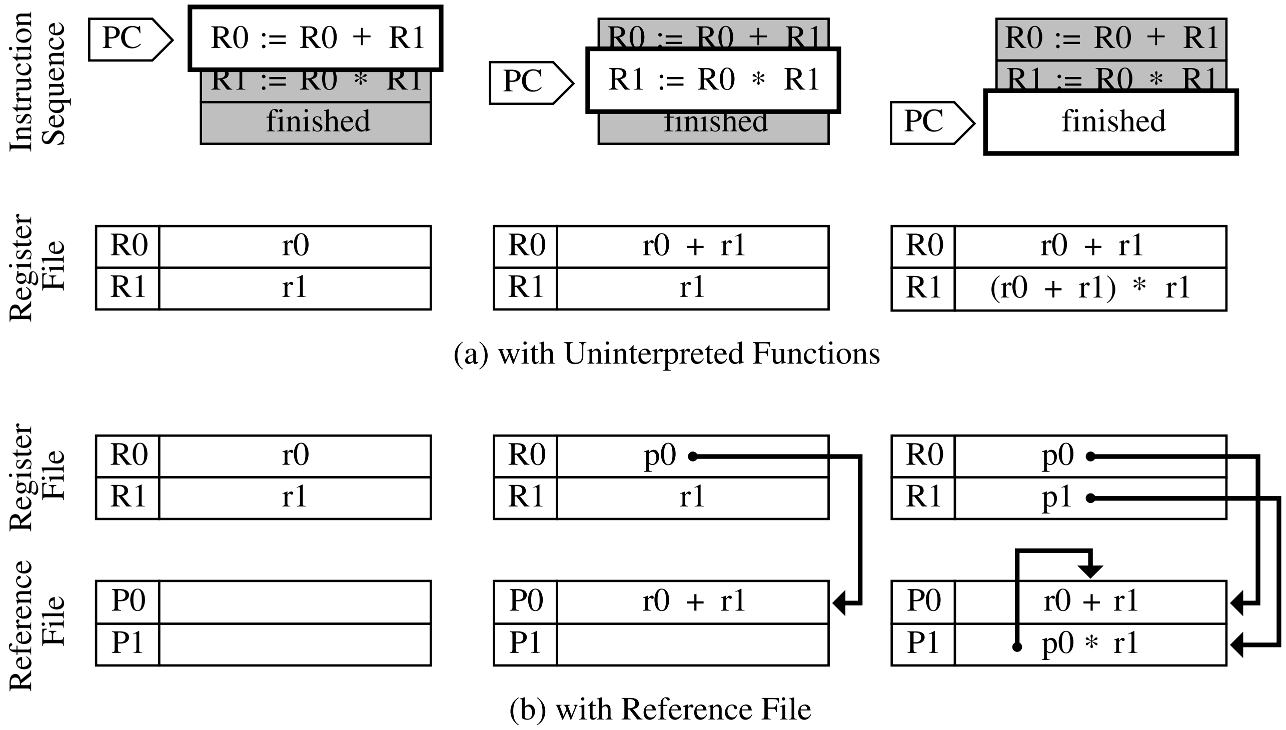

Figure 3: Execution trace of a sequential machine.

Since the functional units operate on the data and the verification task is to compare the results of these operations, we can not make this assumption.

Burch & Dill ([BD94]) use *uninterpreted functions* to overcome this problem. In their approach the operations become uninterpreted function symbols and the data is represented by *terms* over these symbols. For their model of a microprocessor the verification task amounts to checking the equivalence of two deeply nested terms. A special decision procedure for quantifier-free first order logic with equality [BD94] is needed for this purpose. The original [BD94] paper only considered a single scalar pipeline. This work was extended to a superscalar in-order microprocessor in [Bur96]. However, his project required a non-trivial amount of human interaction and manual modification of the model. The decision procedure of Burch & Dill can not make use of highly effective symbolic model checking techniques and is unable to handle OOO properly, as noted in [SJD98].

We start with the same basic idea as [BD94]. The model of the microprocessor does not compute concrete values. It only manipulates symbolic terms made of constants and uninterpreted function symbols. This is explained in Figure 3(a) where a symbolic execution trace of a sequential microprocessor is shown. The processor has two registers and the instruction sequence consists of two instructions. The registers R0 and R1 contain the initial symbolic values r0 and r1 respectively. The first instruction adds these two values and stores the symbolic result ‘r0 + r1’ into register R0. Note that ‘+’ is treated as an uninterpreted function with no actual meaning. In particular, we can not assume commutativity or associativity of these functions. The second instruction computes the product of the result and the initial value r1 of register R1. The final symbolic result ‘(r0 + r1) * r1’ is generated and stored in register R1.

In general, we associate a unique constant with the initial value of each register. Since the number

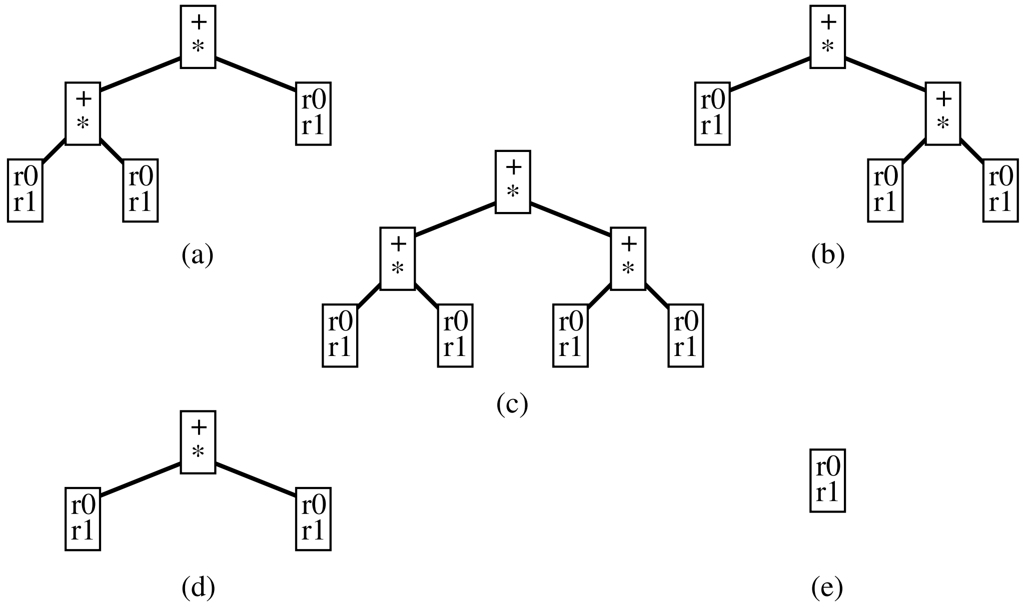

Figure 4: All possible terms for 2 registers and 2 instructions. Every node in these graphs consists of two possible labellings. So each graph represents $2^{\#nodes}$ possible terms.

of registers is finite, the number of constants is also finite. Thus, if we execute a finite number of instructions, the number of terms that occur during this symbolic execution will also be finite. Consequently, the processor model will be finite and can, in principle, be represented in a finite state model checker. A *direct encoding* of all possible terms would achieve our goal of combining uninterpreted function symbols with symbolic model checking. However, we will show that the number of bits needed to represent these terms grows exponentially with the number of instructions i. This makes this approach infeasible.

First, consider the example in Figure 3(a). There are two constants and two uninterpreted function symbols of arity two ('+' and '*'). To count the number of possible terms consider the graphs (a) – (e) in Figure 4. From (e) we get two terms of height 1 and 8 terms of height 2 from (d). The graphs (a) and (b) represent all $64 = 32 + 32$ terms of height 3 where one of the root children has a height of 1. The remaining graph (c) includes all the terms with the maximal number of nodes. These make up additional 128 terms. All together we have 202 different terms, and it takes 8 bits to encode one term.

In general, any term occurring in the symbolic execution will have a height no larger than $i + 1$. The height of a term is defined as the height of its syntax tree or, equivalently, the maximal number of nested function applications plus one. As a rough lower bound on the number of all possible terms we use a lower bound on the number of terms with maximal height and maximal number of subterms (e.g. Figure 4(c)). There are at least as many terms as there are combinations of constants at the leaves. Because the tree has 2^i leaves and each leaf is one of r constants, a lower bound will be r^{2^i}.

Since this is doubly exponential, the number of bits needed in a binary encoding of that domain would grow exponentially with the number of instructions. As an example consider the case where $r = 4$, $s = 4$ and $i = 5$ (see Section 7). In a binary encoding we have to use at least $\log_2 4^{2^5} = 2^6 = 64$ bits for each register and other locations where a data value can be stored. With four registers and four reservation stations the number of state bits would be at least $64 \cdot (4 + 2 \cdot 4) = 768$. Note that 64 bits is often as big as the width of a register in a concrete model.

3.3 The Reference File

While the brute force approach of direct encoding of all possible terms is not feasible, it is important to note that in one execution trace not all possible terms can occur. Moreover, the same terms or subterms are *referenced* at different locations. For instance, in the final state of the execution trace in Figure 3(a) the subterm 'r0 + r1' occurs both in register R0 and R1. In this model it has to be stored twice and can not be shared.

A similar problem occurs in the implementation of logic and functional programming languages like Prolog and Lisp [War83, Jon87, Kog91]. They use a *heap* to store newly generated terms. Registers in the abstract machine for these languages (e.g. WAM [War83]) only contain constant values or pointers to the heap. This prevents unnecessary copying and allows sharing of common subterms.

We use a special data structure, called *reference file*, similar to a heap. This is a much more compact encoding of the terms occurring during an execution. Each entry of the reference file contains an application of an uninterpreted function symbol. Each operand of the function application is either an initial value of a register or a pointer to another entry of the reference file. Unlike the heap, the size of the reference file is finite and is equal to the number of instructions i.

As an example, Figure 3(b) shows the execution of the same instruction sequence as in Figure 3(a). Now all terms are represented with a reference file. After the first instruction the entry P0 of the reference file stores the corresponding function symbol ('+') together with its operands. In our case the operands are the constants 'r0' and 'r1'. Instead of the whole term ('r0 + r1') only the pointer ('p0') is written into the destination register R0. This result is further used by the second instruction. In the entry P1 of the reference file allocated for this instruction the first operand is again stored as a pointer ('p0') without being expanded. Finally, the pointer 'p1' is written to the destination register R1 of the second instruction.

Compared to the first execution trace in Figure 3(a), the difference is that the registers do not contain terms anymore. Symbolic constants ('r0' and 'r1') or pointers to the reference file ('p0' and 'p1') are stored in the registers. When a functional unit finishes a computation, a new entry is allocated in the reference file for a newly generated term, and the registers are updated with pointers to it. Note that the terms occurring in the first run can easily be restored from the reference file by expanding the pointers.

Now, we can calculate an upper bound on the number of bits for the representation of the reference file. Each entry has to store a function symbol and two operands, where an operand is a constant or a pointer to the reference file. There are $i+r$ values for each operand and i function symbols. This requires $2 \cdot \log_2(i+r)$ bits to encode the two operands and $\log_2 i$ bits for a function symbol yielding the total of $2 \cdot \log_2(i+r) + \log_2 i$ bits per entry. The entire reference file consists of i entries, and thus, can be encoded with $O(i \cdot \log_2(i+r))$ bits.

We also need to encode the contents of the register file and the reservation stations. There, in addition to data values, we also need to store tags (see Section 2). This requires $\log_2(i+r+s)$ bits, since we have s different tag values. Each reservation station contains a busy bit, two operands and an opcode. This takes $s \cdot (1 + 2 \cdot \log_2(i+r+s) + \log_2 i)$ bits for all s reservation stations. We also need $r \cdot \log_2(i+r+s)$ bits for r registers. Putting it all together, our model of Tomasulo's algorithm can be encoded with $O((i+r+s) \cdot \log_2(i+r+s))$ state bits.

For an example with four registers ($r = 4$), four reservation stations ($s = 4$), and five instructions ($i = 5$) the exact number of bits is 16 for the register file, 48 bits for the reservation stations, and 55 bits for the reference file, giving the total of 119 bits.

In fact, the number of different instructions that have to be considered is $i = s + 1$, since only s instructions can be stored in the reservation stations at any time, and only one new instruction needs to be dispatched (see Section 5). With i instructions at most $3 \cdot i$ registers can be involved (2 operands and one destination register for each instruction). Thus, if the actual processor has more than $3 \cdot i$ registers, the unused registers can be eliminated. Hence, the only free parameter is the number of reservation stations s, and the upper bound simplifies to $O(s \cdot \log_2 s)$.

4 Tomasulo's Algorithm with Reference File

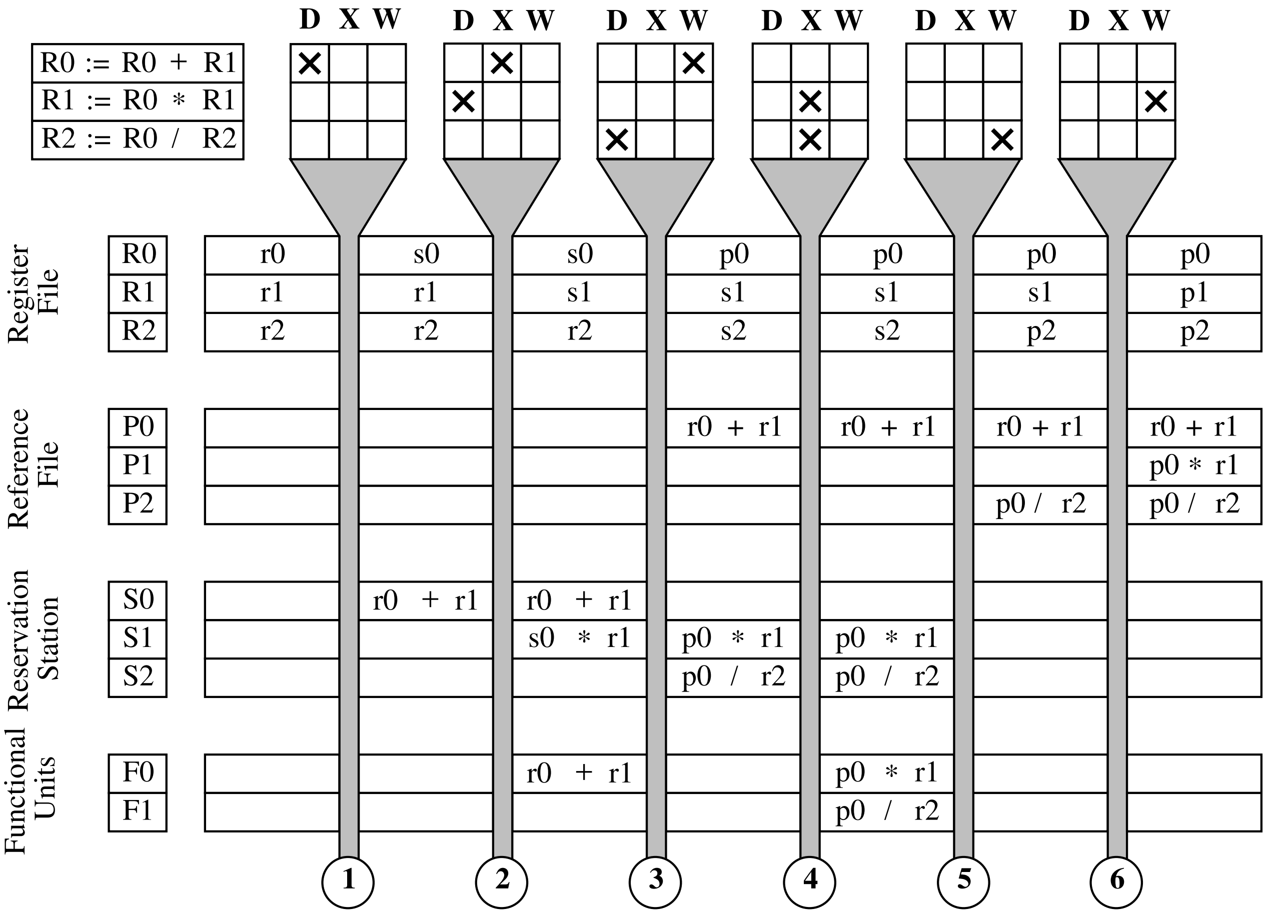

Figure 5: Execution trace of an out-of-order machine using a Reference File (**D** = Dispatch, **X** = Execute, **W** = Write-Back).

Now we can explain how our model of Tomasulo's algorithm actually works (Section 2) when combined with the reference file (Section 3). The configuration shown in Figure 5 consists of three registers, three reservation stations, and two functional units. Since we only consider three instructions in this example, the number of entries in the reference file is also three (P0, P1, and P2).

The instruction sequence is the same as in Figure 3(a) and (b) with one additional instruction. The first instruction places its result into the register R0. This register is then read by the two following instructions, which do not depend on each other. This allows the second and third instructions to be executed in parallel. The result of the first instruction will be written into the entry P0 of the reference file. Similarly, the results of the second and third instructions are written into P1 and P2 respectively. This establishes a one-to-one mapping between the instructions and the entries of the reference file.

When a reservation station becomes available, a new instruction can be dispatched into it. Dispatching means copying the value of the operands and the opcode into this free reservation station. In addition, the tag of the destination register is updated to point to this reservation station. For instance, when the first instruction ‘R0 := R0 + R1’ in Figure 5 is dispatched, the value of the operands (‘r0’ and ‘r1’) together with the opcode (‘+’) are copied into S0. At the same time R0 is updated with ‘s0’, a pointer to S0. At the next step the second instruction is dispatched. This is handled similarly to the first step. However, note that a dependency between S1 and S0 is generated by copying ‘s0’ into the first operand field of S1.

Up to this point the reference file was not needed, since no actual computation took place. Now, in step three, the result of the execution of the first instruction is placed on the CDB by the functional unit F0 and written back into register R0. In our abstract model the result of the computation is a new term. It consists of the uninterpreted function symbol ‘+’ applied to the symbolic values ‘r0’ and ‘r1’. The term is represented by a pointer (‘p0’) to the reference file entry P0 after P0 is updated appropriately. Thus, ‘p0’ is written back into register R0. At the same time the tag ‘s0’ in S1 is updated with ‘p0’.

Beside writing back, the third step involves dispatching the third instruction. This instruction also needs the result of the first instruction in its first operand. When copying the value of the first operand from register R0 to the reservation station, the new value ‘p0’ on the CDB has to be used instead of the old value ‘s0’ in the register. Not forwarding the value from the CDB seems to be a frequent error while designing an OOO unit based on Tomasulo’s algorithm. We made this mistake in a preliminary version of our model and during the verification the model checker produced a counter example. A similar error is also reported in [McM98].

In the fourth step the second and third instructions are executed in parallel. After that, in the fifth step, the execution of both instructions is finished. However, only one functional unit at a time can use the CDB to write back. Our model nondeterministically chooses the second functional unit F1 to be the bus master. This illustrates that the reference file can be written out-of-order. Thus, the third entry P2 of the reference file is updated with the result of the third instruction while the second entry P1 remains empty. In the sixth step P1 is updated with the result of the second instruction. Note, that the result of an instruction is placed in the reference file in the same order as it occurs in the instruction sequence. In particular, the reference file acts like a reorder buffer that never retires instructions. Since this order is independent of the order of the actual execution, the final contents of the reference file will also be independent of the execution order.

5 Overall Verification Approach

The verification of an OOO design consists of proving partial and total correctness. The proof of the total correctness is accomplished by showing a liveness property, and in our model can be easily model checked (see Section 7). The partial correctness (safety) is usually stated in terms of sequential execution (Figure 6). That is, executing an arbitrary sequences of instructions in both the OOO and the sequential machine produces the same result in the register file (for simplicity, we do not consider memory in our example here). A standard approach to deal with instruction sequences is to use

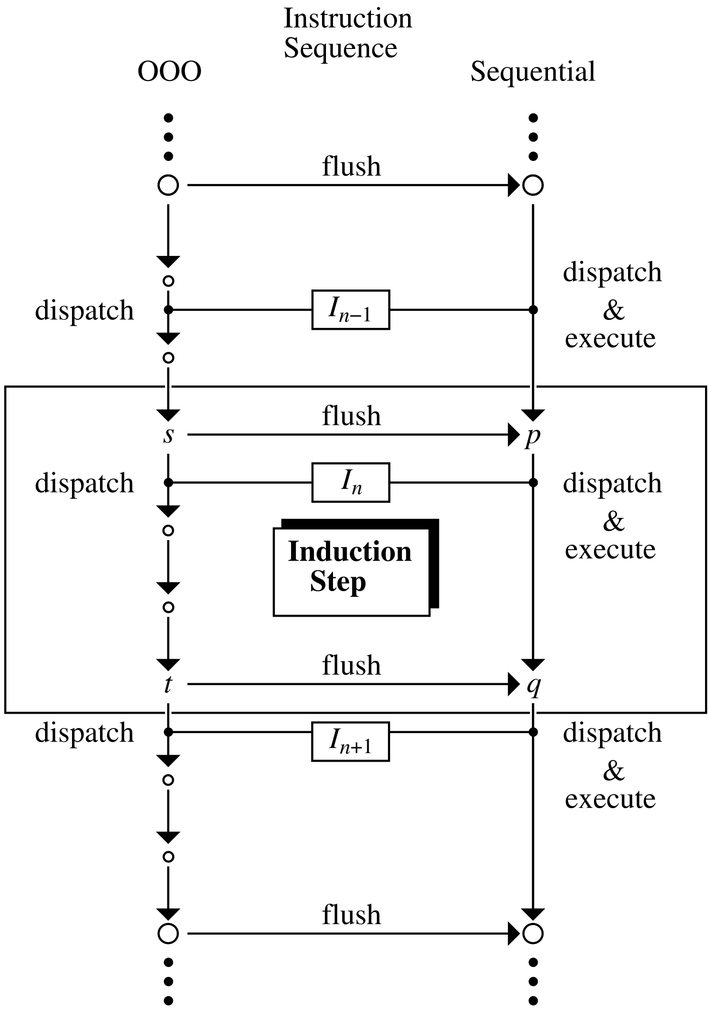

Figure 6: Induction Principle

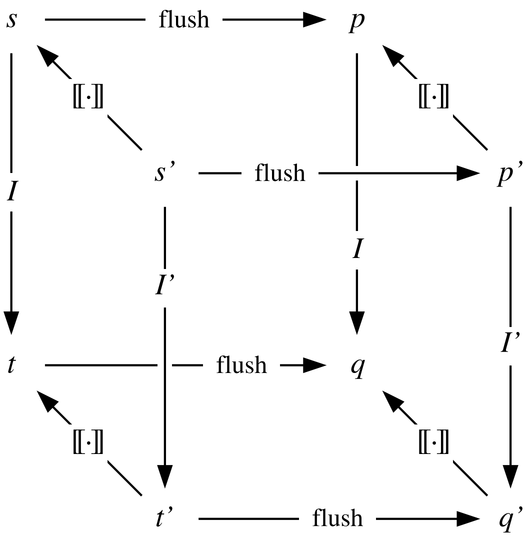

Figure 7: The commutative diagrams of the concrete and abstract machines and their relationship.

induction on the length of the sequence [BD94]. Below we use s and t to denote states of the OOO machine, and p and q for the sequential machine.

In the induction step we have to compare states of the OOO and the sequential machines. We say that two states s and p are equivalent if *flushing* the OOO machine from s produces the same contents of the register file as in the state p. Flushing means executing all of the pending instructions in the reservation stations without dispatching new ones. Given two equivalent states s and p, the induction step consists of dispatching an arbitrary instruction I in the OOO machine and executing the same instruction in the sequential machine. Then we need to check that the resulting states t and q are again equivalent. This property is often described as a *commutative diagram* [BD94] (the rear side of the cube in Figure 7). There are two paths from s to q in Figure 7. The path where dispatching of I is followed by flushing is called the *OOO path*. The other path where flushing is followed by executing I is called the *sequential path*. Then the the diagram commutes if and only if the two paths lead to the same state.

More formally, let $\mathsf{Exec}(p,I)$ be a function that executes an instruction I sequentially in a state p. Similarly, $\mathsf{Disp}(s,I)$ dispatches the instruction I in the state s yielding a new OOO state. And $\mathsf{Flush}(s)$ flushes the OOO machine from the state s and returns the equivalent sequential state. Note that both Disp and Flush may span over several steps of the execution. Moreover, Flush may never terminate if the model has an error and there is a circular dependency among reservation stations. In other words, it may be a partial function. We verify that this function is always total, which implies the total correctness (liveness). We state the partial correctness of the processor as the following theorem:

Theorem 1. *(Partial Correctness) For any finite sequence of instructions $I_1,\ldots,I_n$ and two sequences of states $s_0,s_1,\ldots,s_n$ and $p_0,p_1,\ldots,p_n$ of the OOO and sequential machines respectively, such that*

$$\forall i<n.\;\; s_{i+1}=\mathsf{Disp}(s_i,I_i) \quad \textit{and} \quad p_{i+1}=\mathsf{Exec}(p_i,I_i),$$

if $p_0=\mathsf{Flush}(s_0)$, then $p_n=\mathsf{Flush}(s_n)$.

The proof is by induction over the length of the instruction sequence n. The base case of the induction ($n=0$) is trivial. The inductive step relies on the following lemma:

Lemma 2. *(Commutative Diagram)* $\quad \forall s.\forall I.\;\mathsf{Exec}(\mathsf{Flush}(s),I)=\mathsf{Flush}(\mathsf{Disp}(s,I)).$

If we prove Lemma 2, then the proof of Theorem 1 is easy. However, proving the lemma is hard and is the main bottleneck of the verification of OOO designs. In part, our method of proving this lemma is similar to the one of Burch & Dill [BD94]. As in their approach, we use an initial abstraction that replaces the concrete instructions and data values by uninterpreted constants and function symbols. But then we apply symbolic model checking techniques, including the reference file, as opposed to special decision procedures. In the rest of this section we describe the entire method more formally, and give an idea of how it can be encoded and justified formally in a theorem prover.

The first observation is that the number of instructions j stored in the processor at any given time is always finite and usually not very large. Thus, we only have to deal with at most $j+1$ instructions, including the new instruction being dispatched. Since Tomasulo's algorithm is independent of the actual instructions, we can use $j+1$ different instruction symbols instead. Registers initially contain distinct symbolic constants. Symbolic instructions are then used as function symbols to construct new terms over these constants. The correspondence between such a symbolic representation and the concrete microprocessor is given by the *semantic function* $[\![\cdot]\!]$. Proving the commutativity of the abstract diagram (the front side of the cube in Figure 7) and providing an appropriate semantic function $[\![\cdot]\!]$ is sufficient for proving the commutativity of the concrete diagram. This, in turn, completes the proof of the main correctness theorem for arbitrary sequences of instructions. Formally, we need to prove the following two lemmas. We denote abstract (symbolic) values with primed letters (s', I', etc.), and concrete values by corresponding unprimed letters.

Lemma 3. *(Abstract Commutative Diagram)* $\forall s'.\forall I'.\ \mathsf{Exec}(\mathsf{Flush}(s'),I') = \mathsf{Flush}(\mathsf{Disp}(s',I'))$

Lemma 4. *(Abstraction) Let $[\![\cdot]\!]$ be a mapping of symbols from the abstract domain to the actual functions and constants in the implementation domain, such that*

$$\forall I'.\forall s'.\ \mathsf{Disp}([\![s']\!],[\![I']\!]) = [\![\mathsf{Disp}(s',I')]\!] \quad \textit{and} \quad \forall I'.\forall p'.\ \mathsf{Exec}([\![p']\!],[\![I']\!]) = [\![\mathsf{Exec}(p',I')]\!].$$

Then, if

$$\forall s'.\forall I'.\ \mathsf{Exec}(\mathsf{Flush}(s'),I') = \mathsf{Flush}(\mathsf{Disp}(s',I'))$$

in the abstract domain, then

$$\forall s'.\forall I'.\ \mathsf{Exec}(\mathsf{Flush}([\![s']\!]),[\![I']\!]) = \mathsf{Flush}(\mathsf{Disp}([\![s']\!],[\![I']\!])).$$

Figure 7 gives a graphical view of the approach: we prove the commutativity of the front (abstract) side of the cube (Lemma 3) using model checking and provide a semantic function $[\![\cdot]\!]$ that makes the left and right sides commute (conditions in Lemma 4). This implies the commutativity of the rear side, and that is exactly the inductive step for the concrete processor (Lemma 2).

We use the PVS theorem prover to prove Theorem 1 and Lemmas 2 and 4, providing Lemma 3 as an assumption. This is a relatively easy part of the verification. Lemma 2 immediately follows from Lemma 3 and Lemma 4. The proofs of Theorem 1 and Lemma 4 consist only of manipulating the next state functions and relations without expanding them, and thus, can be done only once for an arbitrary configuration of an OOO design. The constraints on the semantic function in Lemma 4 require some information of the actual implementation details and have to be checked for each concrete model. However, the proof of it only needs to establish that the transition relations of both concrete and abstract machines are structurally the same and differ only in the data types (bit vectors versus function and constant symbols). This lemma can be easily automated, or even avoided by creating a preprocessor that compiles the implementation model into the abstract domain.

The last Lemma 3 is proven automatically using our model checking technique. It is encoded in CTL as an AG formula of the form

$$\mathrm{AG}(\mathrm{OOO.finished} \wedge \mathrm{SEQ.finished} \rightarrow \mathrm{OOO.regfile} = \mathrm{SEQ.regfile} \wedge \mathrm{OOO.reffile} = \mathrm{SEQ.reffile}).$$

The lemma isolates all the complexity of the verification from other problems like enormous of even infinite state space. Its validity is equivalent to the correctness of the whole design. That is, a counterexample for the lemma will be a counterexample in the concrete processor, and if the lemma is proven correct, the processor is also correct.

6 Other Optimizations

With the basic abstraction techniques presented in previous sections we are able to verify several interesting configurations of OOO designs (Figure 9). However, to verify designs that are closer to real microprocessors we need to apply other optimizations as well.

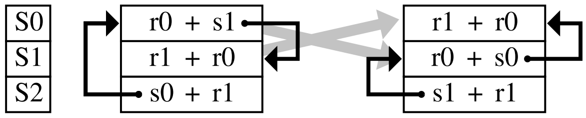

Figure 8: Reducing the number of states by permuting reservation stations.

(Classical) Symmetry Reduction for Functional Units [CJ95]. In our model the result of only one functional unit can be written back at each clock cycle. We make all the functional units completely general such that they can execute any type of instructions. Hence, by a symmetry argument we can assume that only one functional unit writes back on the Common Data Bus. This leaves all the other functional units idle, and they can be eliminated.

Execution of One Machine at a Time. To prove that the diagram in Figure 7 commutes, we need to traverse both the sequential and the OOO paths (see page 11) and compare the results. In a model checker it amounts to running two independent machines from one and the same initial state. The first machine dispatches an instruction and flushes the OOO state. The second machine first flushes the original state and then executes the same instruction as a sequential machine. It turns out that starting one machine only after the other one has finished greatly reduces the number of states and the complexity of the verification. Experimental results suggest that the best order is to run the sequential path first, and then the OOO path. In big examples this can reduce the state space by an order of magnitude and significantly shorten the time of the verification.

Partial Order on Reservation Stations. The next two reductions are very important and will be described in greater details in the full version of the paper. Here we only sketch them briefly. First, we notice that the OOO model is invariant to the order of reservation stations. That is, applying an arbitrary permutation to the reservation stations preserves the transition relation (Figure 8). Note that all the tags pointing to the permuted reservation stations have to be adjusted accordingly.

We partition the set of reachable states into equivalence classes, where two states are equivalent if one can be obtained from the other by permuting its reservation stations. Then we pick a few representative states out of each class. Instead of proving the commutative diagram (Lemma 3) for arbitrary initial states, we prove it only for this set of representative states.

A representative state is obtained by permuting reservation stations together with tags in such a way that the tags always point to reservation stations with smaller indices. For example, in Figure 8 we swap the first two reservation stations. This makes all the tags point upwards in the picture.

In general, for any reachable state we can always find such a permutation that gives us a representative state. Briefly, the formal argument is the following. We define a directed *dependency graph* G_d whose vertices are reservation stations. There is an edge from S_i to S_j if S_i has a tag pointing to S_j. For any reachable state of the OOO machine this graph has to be acyclic. This is an additional invariant of the model that we have to verify.

The dependency graph G_d induces a *partial order* on reservation stations $\leq_d$ such that $\mathrm{S}_j \leq_d \mathrm{S}_i$ if and only if there is an edge from S_i to S_j in G_d. It is known that any partial order can be extended to a total (linear) order. Thus, for any reachable state of the OOO machine there has to be a linear order $\leq_d^{\mathrm{lin}}$ on the reservation stations that respects the data dependencies.

Since reservation stations have unique indices that are used as tags, these indices induce a natural linear order $\leq_i$ on the reservation stations: $\mathrm{S}_0 \leq_i \mathrm{S}_1 \leq_i \ldots$. Now we find a permutation π of reservation stations that makes the two linear orderings identical: $\pi(\leq_d^{\mathrm{lin}}) = \leq_i$. By construction, after the permutation the tags in each reservation station are strictly smaller than its index.

In addition, we can also require that all busy reservation stations have smaller indices than all free ones. This reduces the state space further.

Dynamic Permutation of Reservation Station. This is an extension of the previous reduction technique. Essentially, we construct a *quotient transition relation* under the permutations of reservation stations. As in the previous paragraph, we restrict the entire set of reachable states to a set of representative states, and define a new transition relation in the following way. There is a transition from x to y in the quotient model if and only if there is a transition from some state u to some v in the original model, where u and v belong to the same equivalence classes as x and y, respectively.

In our implementation the quotient model applies an appropriate permutation to the reservation stations after every transition. This maintains the property that all busy RSs are followed by all free ones, as well as the tags are always pointing towards RSs with smaller indices. When a new instruction is dispatched it is placed in the first free RS that in our case directly follows all busy ones. This keeps $\leq_d^{\text{lin}}$ and $\leq_i$ identical all the time. All together, this dramatically reduces the state space while leaving the model fully general.

AG Splitting and the Cone of Influence Reduction. Our specification of (partial) correctness consists of comparing many state variables of the OOO and the sequential machines. Instead of writing such a specification in one formula (see page 12), we split it into multiple formulas each comparing only a few state variables. This transformation is called *AG splitting*. Although ultimately we have to check all of the state variables, we can do it in multiple runs of the model checker. Each split formula depends on a small number of variables and another technique called the *cone of influence reduction* may eliminate some of the state variables from the model. In particular, notice that in our model we only write to the reference file and never read it. Thus, no state variable depends on any of the entries in the reference file. If we split our specification to compare one entry of the reference file at a time, then all the other entries can be removed from the model. A similar technique was used in [McM98].

***Early Evaluation* of AG Specifications** (e.g. [BBDE$^+$97]) was invaluable in debugging the model. This is a technique of evaluating AG specifications at every iteration of the breadth first search for the set of reachable states. If the search reaches a state that violates the specification, then a counterexample can be generated immediately without computing the entire set of reachable states. In addition, an incorrect design may lead to significantly larger BDDs that might prevent us from building the whole set of reachable states. In this case with early evaluation we still may be able to find an error.

7 Experimental Results

We conducted a set of experiments for several configurations of our model of Tomasulo's algorithm using an enhanced version of the SMV model checker ([McM93]). To make the description of our model more general the model was described in the M4 macro language. The M4 script can generate SMV code for an arbitrary configuration of the OOO unit. The parameters to the script are the number of functional units, reservation stations, registers and instruction symbols. Another set of parameters controls which entry of the reference file to keep, what amount of nondeterminism is allowed, and whether dynamic reordering of reservation stations is used.

In all configurations the model only contains one Common Data Bus (CDB). This restricts the number of instructions that can be completed in one clock cycle to one. Therefore, the symmetry reduction described in Section 6 reduces the number of functional units to one. That is, proving the

Regs	RSs	FUs	Instr.	Time (sec.)	States	BDD Nodes $\cdot 10^6$	Iter.
3	2	1	3	94	$5.4 \cdot 10^{14}$	0.7	7
3	3	1	4	240	$3.2 \cdot 10^{15}$	1.78	9
3	4	1	5	1480	$2.0 \cdot 10^{26}$	6.59	11
4	4	1	5	3362	$1.5 \cdot 10^{28}$	15.9	11
5	4	1	5	3941	$8.0 \cdot 10^{30}$	14.3	11
6	4	1	5	8427	$5.1 \cdot 10^{31}$	21.3	11
3	3	2	4	1116	$3.5 \cdot 10^{17}$	7.27	10
3	3	3	4	3380	$4.8 \cdot 10^{19}$	23.1	10
3	4	2	5	> 41505	$> 1 \cdot 10^{28}$	> 28.4	> 1

Figure 9: Experimental results for different OOO machine configurations. The sequential path was run before the OOO one.

correctness of the system with only one functional unit implies the correctness of the same system with an arbitrary number of functional units. Nevertheless, we included several functional units in some of the configurations of our model to study how the complexity of the verification grows with the number of functional units. The table in Figure 9 summarizes the results for a number of different configurations of the machine.

The actual implementation consists of an OOO machine and a sequential machine that start from one and the same state. In this state the dependencies between the reservation stations respect the order of their indices as described in Section 6. The sequential machine flushes all of the instructions from reservation stations and then executes a new instruction sequentially. This implements the sequential path of the commutative diagram (see page 11). The OOO machine dispatches the same new instruction as soon as one of the reservation stations becomes available, and then flushes the state. After both machines have finished their computation, the final values of the register and reference files are compared.

This finishes the main part of the partial correctness proof. In addition, to make sure that the application of the symmetry reductions from Section 6 is correct, we have to show that a certain invariant holds while executing both machines. The invariant states that the dependency graph for the reservation stations is acyclic. It is very important to check this invariant in order to prove the system correct. For example, one of our models had a subtle error that did not show up in the verification of the main correctness formula. The error happened when dispatching a new instruction in the OOO implementation occurred at the same cycle as writing back to a register, whose result was used by the instruction. The tag of that register had not been yet updated, and thus, was copied to the reservation station. At the next cycle, however, the register already had a new value, but the operands in the reservation station still maintained an old tag pointing to an empty RS. This tag is never replaced later, since the write back has already occurred, and this reservation station will never be ready to execute (see Section 4 for a more detailed description of this kind of error). This makes the safety property on page 12 vacuously true. We were able to catch this error only by checking that all tags point to busy reservation stations with smaller indices.

8 Conclusion

We have presented a set of new model checking techniques that combine symbolic model checking with the idea of uninterpreted function symbols. This allows us to prove more complex designs than with standard symbolic model checking or decision procedures for uninterpreted functions alone. Compared to straightforward symbolic model checking we are able to handle arbitrary complex functional units since we can abstract from their actual implementation. On the other hand, we can use the broad spectrum of powerful symbolic model checking techniques, whereas decision procedures for uninterpreted functions can not be easily combined with symbolic model checking.

The key improvement is a new compact encoding of data terms that requires only $O(s \cdot \log_2 s)$ bits for an OOO design with s reservation stations as opposed to an exponential number of bits for a straightforward encoding.

In addition to that, we have developed a number of different transformations that dramatically reduce the size of the set of reachable states and increase the scale of the designs that we are able to handle. Most of the transformations can be formulated as a symmetry reduction [CJ95] and we want to investigate to what extent these reductions can be automated.

The technique includes a theorem proving part using PVS where we check that we did not overlook any important detail. First, we prove a general Theorem 1 together with Lemma 4. Both have to be proven only once, and can be applied directly to any configuration of an OOO design, for which Lemma 3 holds. This Lemma is actually proven by the SMV model checker.

We applied our technique to a set of non-trivial configurations of an OOO design based on Tomasulo's algorithm. We plan to extend it to handle additional hardware structures. In particular, we have already tried some preliminary experiments with memory operations (load and store instructions). In these experiments we assumed that the execution of load and store instructions is always in-order. A more realistic model would enable some degree of OOO execution. In this scenario an address might be the result of a previously executed instruction and must therefore be represented by a symbolic term. Since two addresses represented as terms might actually be the same even if the terms differ, a more involved analysis is required. We are also going to implement the reorder buffer and support for precise exception handling. Finally, it is very important to combine our method with compositional approaches like those of McMillan's [McM98].

References

[BBDE+97] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. Rulebase: Model checking at IBM. In *Computer Aided Verification (CAV'97)*, number 1254 in Lecture Notes in Computer Science, pages 480–483. Springer-Verlag, 1997.

[BCM92] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*. Springer-Verlag, 1994.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Bur96] J. R. Burch. Techniques for verifying superscalar microprocessors. In *33rd Design Automation Conference (DAC'96)*, pages 552–557, 1996.

[CAV98] *Computer Aided Verification (CAV'98)*, 1998. To appear.

[CE81] E. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CJ95] E. M. Clarke and S. Jha. Symmetry and induction in model checking. Number 1000 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In D. Probst, editor, *CHARME'97*. Chapman & Hall, 1997. To appear.

[Gwe95] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995.

[HB95] R. Hojati and R. K. Brayton. Automatic datapath abstraction of hardware systems. In *Computer Aided Verification (CAV'95)*. Springer-Verlag, 1995.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publishers, 1996.

[Jon87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Kog91] Peter M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill, 1991.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[McM98] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In CAV'98 [CAV98]. To appear.

[SGZ+98] K. Sajid, A. Goel, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In CAV'98 [CAV98]. To appear.

[SH98] J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In CAV'98 [CAV98]. To appear.

[SJD98] J. U. Skakkebæk, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In CAV'98 [CAV98]. To appear.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, 1993. Also appears in Tutorial Notes, *Formal Methods Europe'93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[War83] D. H. D. Warren. An abstract prolog instruction set. Tech. Note 309, SRI International, 1983.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th annual ACM Symposium on Principles of Programming Languages (POPL'86)*, pages 184–193. ACM, 1986.