International Journal of Parallel Programming, Vol. 34, No. 1, February 2006 (© 2006)
DOI: 10.1007/s10766-005-0004-8

Embedded Software Verification Using Symbolic Execution and Uninterpreted Functions

David Currie,[1] Xiushan Feng,[2] Masahiro Fujita,[3] Alan J. Hu,[2,6] Mark Kwan,[4] and Sreeranga Rajan[5]

Symbolic simulation and uninterpreted functions have long been staple techniques for formal hardware verification. In recent years, we have adapted these techniques for the automatic, formal verification of low-level embedded software—specifically, checking the equivalence of different versions of assembly language programs. Our approach, though limited in scalability, has proven particularly promising for the intricate code optimizations and complex architectures typical of high-performance embedded software, such as for DSPs and VLIW processors. Indeed, one of our key findings was how easy it was to create or retarget our verification tools to different, even very complex, machines. The resulting tools automatically verified or found previously unknown bugs in several small sequences of industrial and published example code. This paper provides an introduction to these techniques and a review of our results.

KEY WORDS: Formal verification; embedded software; DSP; VLIW.

[1]Synopsys, Inc., Marlbaro, MA, USA.
[2]Department of Computer Science, University of British Columbia, 2366 Main Mall, Vancouver, BC V6T 1Z4, Canada. E-mail: ajh@cs.ubc.ca
[3]University of Tokyo, Tokyo, Japan.
[4]Stanford University, Stanford, CA, USA.
[5]Fujitsu Laboratories of America, Sunnyvale, CA, USA.
[6]To whom correspondence should be addressed.

1. INTRODUCTION

Symbolic simulation and uninterpreted functions have long been staple techniques for formal hardware verification. In recent years, we have adapted these techniques for the automatic, formal verification of low-level embedded software. This paper provides an introduction to these techniques and a review of our results.

Formal verification means *proving* that *properties* hold about *models* of systems. "Proving" is in the sense of mathematical proof—a far higher level of assurance than can be obtained using conventional techniques. "Properties" and "models" refer to fundamental limitations of any (formal or informal) validation technique—some notion of desired behavior must be defined, and the analysis must be done at some level of modeling, e.g., source code, object code, processor microarchitecture, gates, transistors, etc. Historically, formal verification was too difficult and labor-intensive to be used beyond pedagogical examples and a small number of the most safety-critical systems.

The past 15 years, however, have witnessed a revolution in formal verification, primarily for hardware verification, but beginning to affect software verification as well. Currently, all major microprocessor manufacturers have significant formal verification teams, and all major VLSI CAD vendors as well as several start-ups sell formal verification tools. The most commercially successful formal verification technique—checking the equivalence of two similar blocks of combinational logic—has largely supplanted the formerly standard and extremely time-consuming process of RTL-to-gate simulation.

This resurgence of formal verification has many causes, ranging from the staggering verification complexity of modern computer systems to the new, highly automated verification algorithms. A key factor, though, was a paradigm shift: seeing formal verification as just another part of the overall verification strategy and justifying formal verification based on finding more bugs with less effort. This paradigm shift emphasized automatic or easy-to-use techniques, even if they lacked mathematical power. And the new paradigm targeted specific verification problems that were challenging enough (for existing methods) to be practically important, yet small or easy enough to be amenable to the new, automatic formal verification techniques.

Like hardware, embedded software appears to be an excellent application domain to benefit from the new formal verification approaches. Fundamentally, embedded software shares with hardware—and differs from desktop and enterprise software—the frequent need for extreme optimization. The software must hit hard performance, power consumption,

and code-size targets. Code that is slightly too big might necessitate moving to a larger, more expensive device, or code that is slightly too slow might result in unacceptable, non-real-time performance. Therefore, very aggressive optimization is the norm, including possible manual tuning of synthesis(hardware)/compiler(software) output. Compounding the problem, the underlying embedded processor is often designed with similar optimization goals—maximum performance at lowest cost or power, with minimal consideration to the ease of writing or understanding code. Embedded processors (including DSPs) often are highly non-orthogonal, have many specialized instructions, and perform many operations in parallel, with the resulting artefacts (exposed pipelines, long branch delays, VLIW, etc.). All of these features enable very highly optimized, high-performance code, but they also greatly complicate code generation and optimization. Finally, the embedded market is less tolerant of defective software than some other software markets, because patching embedded software in the field can be too difficult, too expensive, or unacceptable to customers. All of these factors point toward very demanding verification requirements. Even a limited tool, if sufficiently automatic and easy-to-use, and if it helps find bugs faster, would be enormously valuable.

In this paper, we review our research towards developing such tools. We have explicitly chosen to mimic the most successful formal hardware verification technique—checking the combinational equivalence of two similar blocks of hardware—and define our problem as checking whether two similar segments of assembly code compute the same results. We envision the primary target of our approach to be the automated verification of highly aggressive, intricate optimizations of small, critical segments of code. To make the approach usable on practical examples, it is necessarily approximate. For example, our approach abstracts away details of bit-precision and datapath, so bit-level optimizations or major algorithmic changes might be erroneously declared inequivalent. In almost all cases, the approximation is safe—equivalent code might be flagged as inequivalent, but not vice-versa—but there are rare situations, which we will note, where our approach might make errors in the other direction as well. Nevertheless, our prototype tools running on real software samples have proven very promising, for example, finding bugs in software previously believed to be equivalent, or confirming that difficult-to-understand optimizations were done correctly. Furthermore, we have found it easy to retarget our approach for different processor architectures, including very complex ones, suggesting that the approach is broadly applicable.

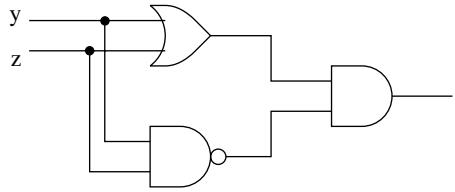

Fig. 1. A simple example: is this XOR?

2. BASIC CONCEPTS

We focus on the particular verification task of comparing two segments of assembly-language code to see if they have equivalent functionality. We further restrict ourselves to code that performs some computation as a function of inputs (e.g., a low-level computational kernel), rather than reactive software (e.g., the operating system), which would correspond to the harder problem of sequential hardware verification. As a further simplifying assumption, we assume that the two code segments are similar—our techniques would be unlikely to be successful on a major algorithmic restructuring, but experimentally, we have successfully verified code that had undergone aggressive optimizations (e.g., software pipelining).

The general approach is to compute a formal representation of what each code segment does, and then to compare the formal representations for equivalence. The questions that must be answered are: what sort of representation to use, how to compute the effect of a code segment, how to compare two representations for equivalence, and how to make the theory efficient enough to be useful in practice. Because our approach is based on ideas from combinational hardware verification, and because the ideas are easier to present in that context, we first illustrate the verification approach for hardware.

2.1. Symbolic Simulation for Hardware Verification

Consider the problem of verifying whether two combinational logic circuits are functionally equivalent. For example, consider verifying whether the simple circuit in Fig. 1 is equivalent to an XOR gate.

An obvious approach would be to simulate (model the behavior of) the circuit for all possible input combinations. For the simple circuit with two inputs, this takes 2^2 trials. For anything non-trivial, the exponential number of possible input combinations is impractical.

A better approach is *symbolic simulation*.[1] Rather than simulating the circuit with specific, concrete values on the inputs, we insert variables

at the inputs and compute symbolic expressions for what the circuit would compute at each point. In the example, we could label the primary inputs with the variables y and z, and then build an expression for each gate output as a function of its inputs[7]—labeling the OR gate with $(y \vee z)$, the NAND gate with $\neg(y \wedge z)$, and the AND gate with $(y \vee z) \wedge \neg(y \wedge z)$. We would then use an automatic decision procedure to check whether $(y \vee z) \wedge \neg(y \wedge z)$ is logically equivalent to $(y \oplus z)$. An alternative formalization would be to introduce a new variable for each gate output and add constraints that relate the inputs of each gate to its output—if we assign variable a to the output of the OR gate, b to the output of the NAND gate, c to the output of the AND gate, and d to the output of the specification XOR, we would get the expression $(a \equiv y \vee z) \wedge (b \equiv \neg(y \wedge z)) \wedge (c \equiv a \wedge b) \wedge (d \equiv (y \oplus z)) \wedge \neg(c \equiv d)$. A satisfying assignment to that expression is a witness to the inequivalence of the two circuits.

The two symbolic simulation formalizations have different characteristics. We dub the first one the "functional translation," because it computes all values as functions of the inputs. The expression size can grow exponentially, but the number of variables is fewer. This approach is commonly used with efficient representations for Boolean functions, such as BDDs.(2) We dub the alternative formalization the "relational translation," because it introduces a relation for each component in the circuit. The expression size is guaranteed to be linear, but the number of variables (and hence the complexity of deciding equivalence) is larger. The relational translation is typically used in practice with efficient SAT solvers.

With either translation style, we can see the same general advantages and disadvantages of symbolic simulation. The basic advantage is that the simulation covers all possible input combinations—it provides formal verification effectively by exhaustive simulation. Furthermore, it is clearly easy to write the simulator, because simple rules can be used to handle each gate or component of the system being verified. This simplicity holds even if the system is complex, e.g., a very large system, or one with unusual arbitration or timing rules. As long as it is easy to write a conventional simulator for the system, it is equally easy to write a symbolic simulator. The obvious disadvantage of symbolic simulation is the unavoidable worst-case exponential blow-up for the symbolic expressions or for the complexity of deciding equivalence of expressions.

Though these ideas are simple and limited, they form the core of modern, practical tools for verifying equivalence of combinational hardware—when strengthened with optimizations and heuristics to make the techniques scalable to practical problems. In the next subsection, we adapt

[7]We use $\oplus$ for XOR, $\equiv$ for equivalence/XNOR, $\wedge$ for AND, $\vee$ for OR, and $\neg$ for NOT.

symbolic simulation to embedded software, and later, we will introduce the optimizations and heuristics that we needed to verify practical software examples.

2.2. Symbolic Execution of Software

The exact same ideas from symbolic simulation of hardware can be used for symbolic execution of software. Instead of computing the values on wires in a circuit, we compute the values of the architectural state (registers, memory, flags) at each point in a program. Instead of a gate-level simulator to compute the effect of each gate in a circuit, we use an ISA (instruction-set architecture) simulator to compute the effect of each instruction.

For example, consider the following simple segment of code for the Fujitsu Elixir DSP:

```
        msm dx, a0, a1
        bge ok
        add dx, cx
ok:     mov (x0++1), dx
```

The first instruction multiplies registers `a0` and `a1` and adds the result to register `dx`. This instruction also sets condition codes, so the following branch instruction branches to label `ok` if the result was nonnegative. The `add` instruction adds register `cx` to `dx`. The `mov` instruction stores the contents of register `dx` into the memory location pointed to by index register `x0`, and then increments `x0` to point to the next memory location.

If we symbolically simulate the trace that skips the `add`, (We will consider handling control flow in a moment.) we find at the end of the trace that:

$$\begin{aligned} \texttt{dx} &= \mathit{initial\_dx} + f(\mathit{initial\_a0}, \mathit{initial\_a1}) \\ \texttt{x0} &= \mathit{initial\_x0} + 1 \\ \texttt{memory} &= \text{write}(\mathit{initial\_memory}, \mathit{initial\_x0}, \mathit{initial\_dx} + f(\mathit{initial\_a0}, \\ &\qquad \mathit{initial\_a1})) \end{aligned}$$

where the symbols that start with *initial_* denote the initial values at the start of the program, the function f computes the effect of the `msm` instruction, and the function write(m, a, v) computes the memory state that results from writing value v to address a in the memory state m.

To verify the equivalence of two segments of code, we symbolically execute each segment, and then use a decision procedure to check whether the results are provably equivalent.

The basic approach for handling control flow is to enumerate all possible execution paths through the code sequence. One could imagine the verification tool examining control flow paths one at a time, comparing them between the two programs being verified. Alternatively, one could imagine computing a large conditional expression indicating the function that would be computed for each register (and memory, etc.) depending on what conditions cause which control flow path to be executed. For example, returning to the simple code fragment above, there are two paths through the code, so the verification tool could analyze the two paths one at a time. For each path, the verification tool records the branch conditions that were assumed in order to follow that control path, and uses these conditions as assumptions during the equivalence check. Alternatively, the verification tool could compute a conditional expression for the values computed on both paths, e.g., for register `dx`, we would end up with:

$$\texttt{dx} = \begin{cases} \text{if } \mathit{initial\_dx} + f(\mathit{initial\_a0}, \mathit{initial\_a1}) \geqslant 0 \text{ then} \\ \quad \mathit{initial\_dx} + f(\mathit{initial\_a0}, \mathit{initial\_a1}) \\ \text{else} \\ \quad \mathit{initial\_dx} + f(\mathit{initial\_a0}, \mathit{initial\_a1}) + \mathit{initial\_cx} \end{cases}$$

Obviously, the number of paths can be extremely large (or infinite), so we make some simplifying assumptions and resort to some approximations. For example, at a branch, we can test if the formal verification decision procedure can prove whether the branch will be taken or not, based on the symbolic expressions computed for the machine state at that point. If so, we need explore only the path that is guaranteed to be taken. For branches where both paths must be explored, a heuristic we used was to insist that the branching structure be "compatible" in the two programs being compared—if one program reaches a branch, the other program must also reach a branch such that the branching condition is the same, or the negation (so that taken/not-taken paths can be reordered); if not, the two programs are declared to be "not verifiably equivalent" by the tool. In practice, these heuristics worked well for the examples we were targeting: the kernels involved mainly fixed-iteration loops, which our approach essentially unrolled, and the forward (non-loop) branching structure was similar because of our assumption that the two programs were structurally similar. However, more sophisticated control-flow analysis is a critical area for future work.

To make the control-flow analysis tractable, we do not model interrupts, and we assume that no computation is performed on the program counter (other than branch offsets, which we ignore by analyzing at the assembly language level rather than encoded machine instructions).

We emphasize that architectural complexity of the target processor is *not* a difficult challenge for this approach. The simulator is only at the ISA level, which is generally quite easy to model. Writing an ISA simulator simply involves understanding the programmer-visible machine state (registers, flags), and then implementing the computation performed by each machine instruction listed in the programmer's reference manual. Making the simulator symbolic simply requires changing the data types of the variables in the simulator to hold symbolic expressions instead of numeric values. (The problem of verifying that a processor correctly executes instructions as specified by the ISA is the completely separate problem of *processor verification*, for which there is a rich literature.)

What *is* a challenge for our approach is the size of the symbolic expressions and the complexity of reasoning about them. If the symbolic simulator is bit-accurate, then the complexity will blow-up, much as brute-force approaches to combinational hardware verification do. The problem is especially bad for the types of software that we are targeting, because common computations like multiplication provably cause BDDs and most other efficient representations for Boolean functions to grow exponentially.[3] If we give up bit-accuracy and treat registers and memory locations as unbounded integers (as is typically done in software analysis), then the verification problem, even restricted to comparing the results of two control-flow path, becomes undecidable in general. What is needed is an abstraction that hides complexity, yet is accurate enough to analyze common optimizations correctly.

2.3. Uninterpreted Functions and Datapath Abstraction

Our main abstraction mechanism for hiding the complexity of operations is the use of *uninterpreted functions*, i.e., a function about which nothing is known other than its name and that it is a function in the mathematical sense (different calls with the same input values produce the same result). For example, if we know that $a = x$ and that $b = y$, then we know that $f(a, b) = f(x, y)$, regardless of what the function f is.

Uninterpreted functions naturally abstract away the details of hardware datapath operations (e.g., whether a multiplier actually multiplies), allowing the verification to focus on the higher-level problem of what operations are being applied to what operands. Uninterpreted functions have been widely and successfully used for formal hardware verification

for precisely this separation of control and datapath complexity, e.g., in semi-automated[4] and automatic[5] microprocessor verification, as well as for more general hardware.[6,7] In the example from the preceding subsection, we had built symbolic expressions assuming that f represented the processor's multiply operation, but if the two code segments being verified produced the same symbolic expression, then they are provably equivalent, regardless of the details of what operation f performs.

Note that the abstraction is safe—the abstraction will not cause two inequivalent programs to be declared equivalent—but is sometimes too conservative. For example, the abstraction loses the ability to prove that multiplying by 2 is equivalent to a left-shift, since multiplying a number x by 2 would be modeled using one uninterpreted function, e.g., $f(x, 2)$, whereas the left-shift would use a different uninterpreted function, e.g., $g(x)$, and the abstraction loses all information about how f and g might be related. Similarly, the theory loses any knowledge that, for example, multiplication is commutative or associative. We will discuss how we deal with these issues in practice in Section 3. Of course, on a real processor, the hardware implementation of multiplication might not be commutative or associative due to rounding or similar effects; if we wish the verification tool not to assume commutativity or associativity, the uninterpreted function abstraction, with no additional information, would likely be exactly the correct level of abstraction.

Not only do uninterpreted functions hide the complexity of data operations, they also make the verification problem itself computationally easier. In particular, the verification task consists of building symbolic expressions involving uninterpreted functions and comparisons, and the logic consisting of uninterpreted functions and equality is decidable.[8] Furthermore, practically efficient decision procedures exist for many variants of this logic.

2.4. Decision Procedures

The verification approach we have described so far is to use symbolic execution to build up symbolic expressions for what each code segment computes, using uninterpreted functions to hide datapath complexity. The verification task, then, consists of comparing two symbolic expressions to see if they are equal. For that, we need efficient decision procedures for the logic of uninterpreted functions and equality.

There is an extensive literature on automated decision procedures, which we cannot survey comprehensively. In this section, we give only some intuition about how the decision procedures work, some examples of

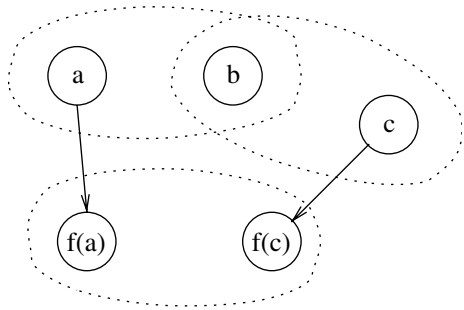

Fig. 2. Congruence closure example. Suppose we assert that $a = b$ and $b = c$, and wish to prove that $f(a) = f(c)$. When we assert $a = b$, the decision procedure creates nodes for the terms a and b, and groups them into an equivalence class. For $b = c$, the decision procedure does the same, then merges the two equivalence classes. To check whether $f(a) = f(c)$, the decision procedure creates nodes for $f(a)$ and $f(c)$. Since the two nodes have the same function symbol, and their arguments are already proven equivalent, they are grouped into an equivalence class, too.

the logical theories they can handle, and some pointers to the leading decision procedures currently used for automated verification.

Intuitively, a decision procedure for this logic must handle the usual reasoning for Boolean logic as well as support the reasoning needed to analyze equality comparisons and uninterpreted function calls. The Boolean aspects are handled much as in Boolean satisfiability solvers, e.g., case-splitting, backtracking, and conflict analysis.[9–13] For the uninterpreted functions and equality, there are currently two general approaches being used: congruence closure[14] and reduction to propositional logic.[8] In the congruence closure approach, the decision procedure tracks all terms in the formula, building up equivalence classes of terms proven to be equivalent. Two terms involving uninterpreted functions are proven equivalent iff they call the same function, and the corresponding arguments have already been proven equivalent. Figure 2 shows a short example of this procedure. In the alternative approach of reducing to propositional logic, new propositional variables are introduced to represent equality terms, and new propositional clauses are added to enforce symmetry and transitivity of equality, as well as functional consistency. The resulting propositional formula is solved using a Boolean SAT solver. Historically, the reduction-to-propositional-logic approach was invented first, but was too inefficient in practice; only recently have more efficient reductions been developed[15] that make both approaches effective on practical problems.

In practice, additional theories beyond uninterpreted-functions-with-equality are very useful. For example, a theory of memories/arrays is crucial for reasoning about code that access large arrays in memory. The

theory consists of functions read(mem, $addr$), which given a memory and an address, returns a value, and write(mem, $addr$, val), which given a memory, address, and value, returns a new memory in which the write has taken place, and the key axiom is that

$$\text{read}(\text{write}(mem, addr1, val), addr2) = \begin{cases} val & \text{if } addr1 = addr2 \\ \text{read}(mem, addr2) & \text{otherwise} \end{cases}$$

Another invaluable theory for handling practical problems is linear (aka Presburger) arithmetic, consisting of addition of terms, and multiplication by constants. Support for linear arithmetic is especially important when verifying low-level software because of the need to reason about memory addressing calculations. (Standard Peano arithmetic, with multiplication of terms, is undecidable.) For applications that involve bit-level reasoning, a theory for handling bit-vectors is useful. Fortunately, all of these theories are decidable, and there exist standard methods for building decision procedures for combinations of decidable theories.[14,16,17]

There are several efficient implementations of these combined decision procedures publicly available. The most well-known currently are (in chronological order): the Stanford Validity Checker (SVC),[18] Integrated Canonizer and Solver (ICS),[19] Cooperating Validity Checker (CVC),[20] UCLID,[7] Simplify,[21] and CVC Lite (CVCL).[22] All of these tools support roughly the same combination of decidable theories, including uninterpreted functions and axioms for modeling memories. UCLID does not support full linear arithmetic, but the others do. Performance varies, with each tool being better than the others for some problems. De Moura and Rueß provide a recent performance comparison.[23] For our work, we used SVC. It supports the key theories that we needed: boolean logic, uninterpreted functions with equality, memory read/write, and linear arithmetic. Furthermore, it was stable and available at the time we commenced this research in 1998. For a project starting today, the other modern decision procedures should be considered as well.

3. FROM THEORY TO PRACTICE

The preceding section described our theoretical framework: symbolically execute the software using a decidable logic, with some abstractions to reduce complexity, and then compare the results with a decision procedure. Moving from theory to practice, required handling many practical details. Our improvements to the theoretical framework can be grouped into three general categories: handling specific architectural and

programming idiosyncrasies of real examples, improving the efficiency of the verification, and strengthening the decision procedure to reduce false errors.

3.1. Handling Real Processors and Real Coding Styles

Most of the complexity of real processors and code examples turned out to be straightforward to handle. For example, condition codes (which indicate whether the result of an operation is negative, zero, or overflow) can be handled in symbolic execution just as any other register is. Conditional branch instructions simply refer to the condition codes. No special modeling is required.

Accurately modeling memory addressing was important for computing correct verification results on real code examples. In particular, when verifying software written in a high-level language, arrays are assumed to be disjoint objects, and the read/write functions described earlier are typically applied to each array separately (e.g., a write to an array A does not change the state of array B). In contrast, we model all of memory (or each bank of memory in a system with multiple banks) as a single array with all reads and writes directed at this array using the read/write functions. This approach leads to larger symbolic expressions, but is needed because it correctly handles relative addressing (e.g., address arithmetic performed on index registers, based on knowing the layout of data in memory). Embedded software is often highly optimized based on the exact layout of data in memory.

Modulo addressing is one of the more complex addressing modes that we modeled. Many DSPs and embedded processors have modulo or circular addressing modes, which compute auto-increment and auto-decrement of index registers modulo a specified modulus. We handle these addressing modes with a simple if-then-else. For example, an auto-increment on an index register `x0` would be treated as:

$$\text{if } (\texttt{x0} + 1 \geqslant m) \text{ then } (\texttt{x0} + 1 - m) \text{ else } (\texttt{x0} + 1),$$

where m is the modulus. The tool is not actually computing the mathematical modulo operator (e.g., if $\texttt{x0} > 2m$), but fortunately, real processors typically perform the same computation as our tool for these addressing modes, rather than true mathematical modulo. Note that we handled this addressing mode using linear arithmetic; in general, having decision procedure support for linear arithmetic proved indispensible for address calculations.

Embedded processors have been quick to adopt modern ISA innovations such as predication and VLIW, because embedded software applications are typically recompiled, eliminating the need for legacy binary compatibility. The basic functionality of *predication* is easy to model: for each register modified by a predicated instruction, the verification tool generates a conditional expression that evaluates to the new value if the predicate is true or the old value if the predicate is false. (Predicated branches are conditional branches, handled as described earlier.) The *parallel execution* of instructions in a VLIW is also trivially handled by the symbolic execution. We symbolically execute each instruction in the very-long-instruction-word one-by-one, but don't update the register file until after all instructions have read their operands. The resulting behavior is identical to parallel execution.

Some embedded processors have lengthy *delay slots*, which result from removing interlocks and exposing the processor's pipeline to the programmer, the goal being to provide maximum performance at minimum cost even if it complicates the programmer's model. Fortunately, because the delay slots are the logical consequence of the processor's pipeline, the symbolic simulator can model delay slots easily using queues. For branch delay slots, we keep a small queue of pending branches. Each branch instruction gets queued, along with a counter indicating how many delay slots remain for that branch. Each clock cycle, all counters are decremented. Before each instruction fetch, we check whether a branch at the head of the queue is ready to be taken. Similarly, for read-after-write delay slots, we extend the data structure used to hold register values. In the basic verification approach, each programmer-visible register is modeled by a variable that can hold the symbolic expression for the value of that register. Instead, we replace this variable with a queue of symbolic expressions, which stores the pending writes to that register (see Fig. 3). Whenever a result is written to a register file, the latency of the operation is checked in a table, and the result is written into the appropriately delayed queue entry. Reads from the register file return the current value. After every clock cycle, all the register-value queues advance one step; if there is no write to a register in that cycle, it keeps its current value.

Dynamic resource conflicts. As shown in Fig. 3, the combination of predication, parallel instructions, and delay slots (with different latencies for different instructions) can result in potential conflicts, in which several instructions attempt to write a given register at the same clock cycle. Our verification approach can detect such situations by storing a *write history* in the pending write queue. Each write-history is a list of predicate-(symbolic)value pairs. The decision procedure can be used to

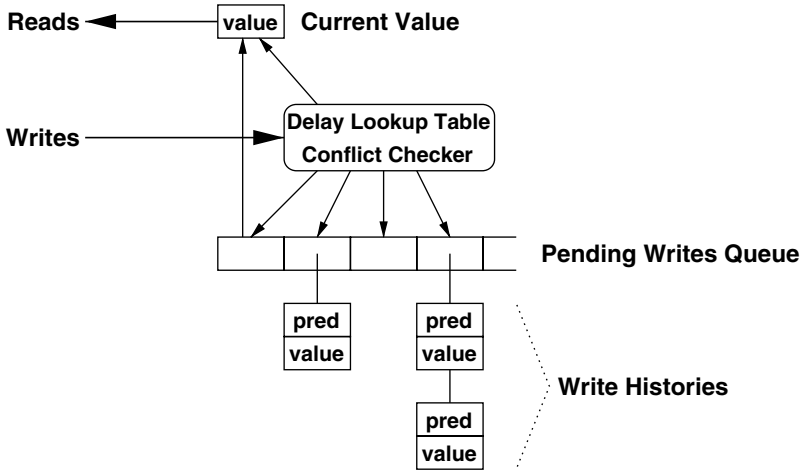

Fig. 3. Register model. This figure shows the data structures used to hold the "value" of each register. In the basic verification algorithm, only the "Current Value" variable would exist, and reads and writes would access and update this variable. To handle delay slots, we add a queue of pending writes. Reads still come from the Current Value, but values to be written are queued depending on their latency. The queue advances one position to the left at each clock cycle, updating the Current Value as appropriate. The attempt to write to a queue slot that is already full indicates a resource conflict. To handle predication with delay slots, we store *write-histories* in the pending writes queue, rather than values. Each write-history is a list of predicate-value pairs. We verify that at most one predicate in a write history can be true, otherwise there is a conflict. Since the simulator is symbolic, the "values" are actually symbolic expressions denoting the value as a function of the initial values.

verify whether any writes scheduled for a given clock cycle conflict with one another.

Finally, *indirect branches and jump tables* required special handling. A key simplification in our verification approach is the separation of our analysis of control flow from our analysis of data computation. In low-level code, however, a common idiom for control flow is the indirect branch—branching to an address that had been read or written to memory or register as a data value. We found an easy solution to handle the common, practical case, in which branch targets are only loaded and stored, but not used in computation. The solution is to expand the concept of a "value" to be two-sorted: data values (including addresses used to index into memory), and control values, which can be only labeled control flow points in the program. Computation is not allowed on the control values, but they can be read and written from memory, stored

in registers, and, most importantly, used as branch targets. This solution handles common cases of indirect branches, such as subroutine/coroutine linkage, jump tables, or virtual function dispatch.

3.2. Improving Verification Efficiency

Context management. Our verification method is based on a depth-first exploration of possible control flow of the programs. For the tool to run efficiently, it must quickly recall the previous state of the partially simulated trace during backtracks. We call this state a *context*. When a branch (that cannot be proven to be unconditional) is encountered, the current context is pushed onto a stack. When backtracking, the symbolic execution can continue from this point by popping to the previously saved context. In our initial implementations, our tools maintained the stack of contexts. In later implementations, SVCs internal context management had become sufficiently stable that we used it instead. This change greatly sped up our verifier, since using the decision procedure's context management allowed the decision procedure to reuse its own computations.

Constant propagation was an obvious and simple optimization that sometimes resulted in large speed gains. Our general principle was to have the symbolic execution engine do as much non-symbolic execution as possible, to minimize the burden on the decision procedure.

Rewriting memory expressions. The main source of expression-size blow-up in our industrial examples came from memory accesses. For example, consider a loop (for the Fujitsu Elixir) that copies data from one array to another:

```
        do  endloop, 512 ;; repeat 512 times to label endloop
        mov a0, (x1++1)  ;; a0 gets next word from memory
        mov (x2++1), a0  ;; write a0 to memory
endloop:
```

After the first iteration, the memory will contain the symbolic expression $\text{write}(m, x_2, \text{read}(m, x_1))$, where m, x_2, and x_1 denote the contents of the memory and index registers at the start of the loop. After another iteration, the symbolic expression for memory will grow to:

$$\text{write}(\underbrace{\text{write}(m, x_2, \text{read}(m, x_1))}_{\text{previous iteration's memory}},\ x_2+1,\ \text{read}(\underbrace{\text{write}(m, x_2, \text{read}(m, x_1))}_{\text{previous iteration's memory}}, x_1+1)).$$

The expression is growing exponentially! To reduce the blow-up of memory expressions, we introduced some sound-but-incomplete rewriting rules:

1. When a write is performed, the existing memory is scanned to check if the address being written has already been used. Since the new write overrides the older write, the older one can be removed, e.g., write(write(m, a, v_0), a, v_1) becomes write(m, a, v_1). This simplification is extremely useful for loops that repeatedly write to a temporary storage location.
3. When a read is performed, we "peel off" any writes that can be proven not to be to the same location as the address being read, e.g.,read(write(write(m, a_0, v_0), a_1, v_1), a_2) becomes read(write(m, a_0, v_0), a_2), if the decision procedure can prove the a_1 can never be equal to a_2.
3. Similarly, when a read is performed, if the decision procedure can prove that the address being read is always equal to the address that has been written, the result is simply the value that was written, e.g., read(write(m, a_0, v_0), a_2) simplifies to v_0, if the decision procedure can prove that $a_0 = a_2$.

In the array-copying example above, if the verification tool can prove that the arrays don't overlap (based on knowledge of the memory layout), then these rewriting rules will cause the expression size to grow linearly rather than exponentially.

Functional vs. relational translation. Recall from Section 2.1 that there are two ways to build the symbolic expression for a program or circuit: the functional or the relational translation. The functional translation uses fewer symbolic variables, but potentially generates exponential expression size; the relational translation generates linear-size symbolic expressions, but requires many more symbolic variables. Other researchers have argued for the superiority of the relational translation.[24] Our experience was different, and we continued to use the functional translation. Our domain-specific expression-rewriting was able to keep our symbolic expressions reasonably small in most practical cases, and the additional variables introduced by the relational translation severely impacted the performance of our decision procedure. Perhaps a newer decision procedure would be better able to handle the relational translation, but the ability of ad hoc, sound-but-incomplete rewriting to reduce expression size should not be ignored. In the longer term, we expect the right approach will be to use a functional translation as long as possible, with rewriting to reduce expressions size, and then introducing intermediate variables as in the relational translation only when needed.

3.3. Reducing False Inequivalence

In addition to rewriting to improve efficiency, we also relied on domain-specific expression rewriting to strengthen the decision procedure. The basic problem is that we rely on uninterpreted functions to model most operations on data (almost everything except addition and subtraction), and the theory of uninterpreted functions with equality is too weak to capture many properties we need to verify equivalence. For example, common optimizations include using shifts to replace multiplication/division by powers of 2, or exploiting commutativity/associativity to group and reorder computations. Using only uninterpreted functions, our verification approach would falsely declare the optimized code inequivalent, because uninterpreted functions do not in general obey the properties used in the optimization.

Axioms. Our solution is to allow the user to specify additional properties, which we dub "axioms", that specific uninterpreted function symbols are assumed to obey. For example, if we use an uninterpreted function symbol `MULT` to denote multiplication, we can specify that `MULT` is commutative by

```
MULT assert (= (MULT argMULT1 argMULT2)
               (MULT argMULT2 argMULT1))
```

In general, user-specified axioms could make the logic undecidable. Instead, our approach is simply to use these axioms as rewriting rules to generate additional facts in an ad hoc manner. For example, whenever the verification tool generates an expression involving `MULT` like[8] `(MULT a b)`, the verification tool will use the axiom as a pattern to generate the assertion `(= (MULT a b) (MULT b a))`. The result is a sound application of the user-specified axioms, but is not complete: not all implications of the axioms will be computed. The ability to specify axioms was crucial to strengthen the decision procedure enough to produce useful results, and fortunately, we found that only a few axioms were enough in our experiments (e.g., commutativity of multiplication, equivalence of shift to multiplication by 2, etc.).

Expression normalization. This optimization can be considered a special, extra powerful axiom. The basic problem is that arithmetic with multiplication is undecidable, so we cannot expect the decision procedure to handle arbitrary arithmetic expressions. On the other hand, programs often generate complex arithmetic expressions, so we want a more powerful means to check equivalence. The solution is to rewrite arithmetic

[8]SVC uses LISP-like syntax: the first symbol after the parentheses is the function being called.

expressions heuristically into a normal form, to increase the likelihood that mathematically equivalent expressions can be proven equivalent by the decision procedure.

The heuristics consist of a number of transformations and simplifications in an attempt to bring the two expression to a reduced form. Constants in additions and in multiplications are evaluated as much as possible. Address expressions are simplified based on the memory layout (e.g., if memory location $a + 3$ is being accessed, and we know that $a + 3 = b$, then we replace the reference to $a + 3$ by b). In a sequence of multiplications and divisions, the divisions are moved to the end. Most importantly, the arguments of multiplications are reordered according to a fixed order. Although these transformations are obviously not complete, they are sound and were sufficiently powerful to handle all the instances that arose in the industrial examples.

3.4. False Equivalence

We have strived to be conservative in all of our abstractions and approximations, so that our verification approach will not erroneously declare two programs equivalent if they are not. Our effort, then, focuses on reducing the false inequivalences—cases where the verification approach declares two programs inequivalent, even though they are actually equivalent.

However, because we do not model the processor at a bit-accurate level, with finite word sizes and precisions, there do exist some aspects of our verification approach where we might falsely declare programs to be equivalent. For example, our theory will declare $(x + y) - z$ to be equivalent to $(x - z) + y$, although the two expressions have different overflow behavior on finite-sized words. Similarly, we do not model a finite-size memory, so our verification approach might declare two programs equivalent even though in reality, one of them might take a memory protection fault when the other wouldn't. In general, the use of pure uninterpreted functions is safe, even when considering exceptions or finite-precision, but all the other theories (linear arithmetic, memory, user-defined axioms) are potentially problematic.

This is the most serious weakness of our verification method. If we consider the tool a debugging aid rather than a certification of correctness, however, and given that the verification is completely automatic, the tool should still be useful despite this theoretical limitation.

4. EXAMPLE EXPERIMENTS

To assess the practical usefulness of our approach, we have applied it over the years to a few different embedded processors and DSPs. In this section, we review these results.

4.1. Fujitsu Elixir DSP

Our first effort to apply this verification approach[25,26] was for the Fujitsu Elixir, a 16-bit, fixed-point DSP used in cellular telephones. We were fortunate to obtain a set of four matched pairs of code.[9] Each pair consists of two subroutines that were believed to be functionally equivalent.[27] One subroutine was production code taken from a cellular telephone application, hand-written by experts. The other was compiled from an allegedly equivalent C program, using a highly optimizing compiler.[27] The examples range in size from 37 lines to 190 lines of code for the compiled version.

All errors discovered were found completely automatically. When an error was discovered, we generally considered the hand-written code to be "golden", modified the other program to fix the problem, and repeated the verification.

Yhaten: Yhaten was the smallest example and had no errors.

Hup: The Hup example contained only one branch to check a rounding flag and set the appropriate register, followed by a fixed-count loop to calculate a sum and multiplication.

The tool found two errors. The first is that the compiled code lacked the aforementioned branch to check the rounding flag. The tool detected this error due to discrepancies between the two control-flow graphs. The second error was that a register used in modulo addressing was set to a fixed value in the generated version, but was set from a memory location in the hand-coded version.

Kncal: Kncal contained a number of branches, a fixed count loop to calculate a division as well as a much more interesting array of operations, such as shifting, negating and logical ANDing.

The first error located was that the compiled code used a logical shift where the hand code used an arithmetic shift. Depending on the value of the input this could generate different results. Another error was that, along a particular trace through several branch choices, the compiled code attempted to access a temporary memory location that had not been set properly. Errors of this sort (missed initialization along one particular

[9]The code is proprietary, so we cannot give listings or detailed descriptions of functionality.

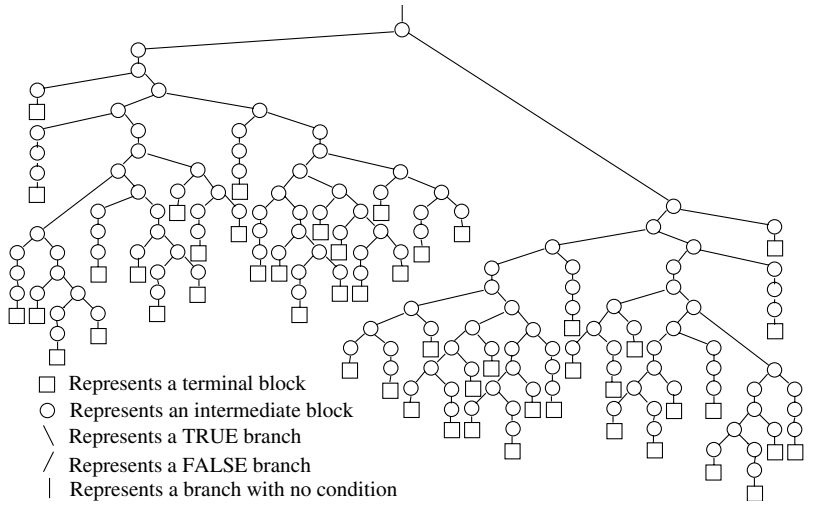

Fig. 4. Branching structure for Dt_pow. This diagram is the CFG unfolded into a tree, showing all possible paths through the routine.

trace) are common and can be extremely difficult to detect if the branching structure is complex.

Dt_pow: Dt_pow was the largest example and the most complex in terms of branching structure (Fig. 4).

Further complicating this example was the extensive use of auto-incrementing and decrementing in the hand-coded version, making it difficult to determine which memory addresses were being accessed by visual inspection alone.

The tool found several errors in this example. There is a trivial computation error (order of operations), apparently resulting from a typo in the C code. In the compiled code, several computed results were not being stored properly. More interestingly, there are several distinct cases of missed initializations along particular traces. For example, both programs contain a branch to allow the program to skip initialization of several memory locations. In the hand-written code, if this branch is taken, those addresses would have default values instead. In the compiled code, however, if the branch is taken, default values are not properly supplied. The most interesting bug is one that we believe is a missed initialization along an involved sequence of branch choices in the expert hand-written code.

In summary, we emphasize that in all examples, all bugs were discovered automatically using the tool. The examples were small, but were sufficiently

Table I. Tool Performance on Fujitsu Elixir Examples

Example	Size	Time	SVC Time	Mem
Yhaten	37	39 s	25 s	38 MB
Hup	47	8 h 36 m	7 h 56 m	69 MB
Kncal	69	1 s	1 s	3.4 MB
Dt_Pow	190	12 s	9 s	3.9 MB

tricky that the bugs had eluded previous detection. We also note that most bugs resulted from errors in the C program, and not the compiler.

Performance: The performance for each example is given in Table I. "Size" is the number of lines of code in the compiler-generated version. "Time" is the total wall clock time taken by the program to verify the assembly code, after all errors were corrected. "SVC Time" is the time that the program spent in SVC. "Mem" gives the maximum memory used during the verification. The memory usage was shared between SVC and our program, but most of the execution time was spent in SVC. Only Hup performed poorly, because of the rewriting—the more expensive rewriting rules were ineffective, but were repeatedly called while the program's main loop was unrolled. Turning off the useless optimizations would have accelerated the runtime substantially. Overall, we find the performance for verifying small, intricate code to be reasonable. The tests were run on a Sun Ultra 60 (360 MHz) with 1280 MB of memory.

4.2. Texas Instruments' 320C62x VLIW DSP

Our next effort to apply our verification approach targeted the more complex ISAs of modern VLIW processors.[28,29] Texas Instruments' TMS320C6x family of VLIW digital signal processors[30] is both commercially important and also the epitome of this architectural style, so we targeted the family for our research. In particular, we targeted the C62x family, which are 32-bit fixed-point DSPs that are code-compatible with the more powerful C64x fixed-point family and the C67x floating-point family.

In the C6x processors, instructions are grouped into "execute packets" of up to eight instructions, and these packets can be executed one per clock cycle. The design can attain very high performance, but is highly non-orthogonal. Each of the eight instructions goes to a specific functional unit. The functional units have specific capabilities, and there are limited resources for routing data among the functional units and register files. Careful code-tuning is imperative to achieve maximum performance. However, the processor has no interlocks—most potential conflicts in resource

utilization are disallowed statically during code generation, but some cases produce undefined results. We will elaborate on this point below.

The other main architectural feature that both enhances performance and complicates code generation and verification is the pipeline. The processor is heavily pipelined, with instructions taking up to 11 cycles to complete. Table II summarizes the pipeline. The pipeline has no interlocks, which simplifies the hardware (more performance at lower cost) but complicates the code. In particular, multiply and load instructions have long latencies and require 1 and 4 delay slots, respectively. Instructions in these delay slots see the old value of the register. Multiple writes to a register in a single clock cycle are illegal, and in most cases, this can be detected during code generation.

Another artifact of the programmer-visible pipeline is that there is a long branch delay—five delay slots (i.e., the next five packets fetched following a branch always execute, regardless of whether the branch is taken or not)—because the branch doesn't affect the Program Address Generation stage until it reaches the Execute 1 stage. Branch instructions may appear in the delay slots of other branches, so the five packets fetched following a branch might not be contiguous in memory. The results may be unintuitive, but they follow naturally from the pipeline definition.

To complicate matters further, but also to allow very compact and efficient code, all instructions are predicated, i.e., each instruction in each execute packet specifies a register and a condition (equal-zero or not-equal-zero), and only executes if that register is zero or not zero. Predication can eliminate many branches, increasing the size of basic blocks and the amount of instruction-level parallelism available.[31] Unfortunately, predication and the absence of pipeline interlocks means that many register-write conflicts may or may not happen, depending on the values of registers at runtime. The processor manual specifically states that these situations cannot be detected, but that the result is undefined when they happen.[30,Section3.7.6] Our verification tool was able to detect (or verify the absence of) these dynamic conflicts, using the write-history technique described in Section 3.1.

Overall, the architecture follows the VLIW philosophy and is optimized for maximum performance with minimal hardware, with no consideration for easy code generation or verification. The apparent complexity of the programmer's model, however, is not arbitrary, but the logical consequence of the exposed parallelism and pipeline. Accordingly, we found that although the code is error-prone for a human to read or write, it was relatively easy to create a simulator for the processor—even a symbolic simulator—that captures the correct semantics.

Table II. C62x Processor Pipeline

Stage	Description
Program Address Generate	Determine address of the fetch packet
Program Address Send	Send address of the fetch packet to memory
Program Wait	Program memory access happens
Program Data Receive	CPU receives fetch packet from memory
Dispatch	Determine the next execute packet in the fetch packet and send it to the appropriate functional units
Decode	Decode instructions in functional units
Execute 1	Evaluate predicates. Read operands. For load and store instructions, do address generation and write address modifications to register file. For branch instructions, affect the Program Address Generate stage. For single-cycle instructions, write results to a register file
Execute 2	Load and store instructions send address (and data for store) to memory. Most multiply instructions complete
Execute 3	Data memory accesses are performed. Store instructions complete
Execute 4	For load instructions, CPU receives data
Execute 5	For load instructions, write the data into a register

The processor pipeline is 11 stages deep. Instructions are grouped into "execute packets" of up to eight instructions that proceed through the pipeline in parallel. Note that different instruction types write results with different latencies. This table is summarized from Ref. 30, Table 6-1.

As the challenge problem for our verification tool, we found an article, written by an expert, explaining how to optimize code for high-performance DSPs.[32] The most difficult example in the article demonstrates software pipelining[33] a short loop, with code supposedly for the TI C6200. The basic idea is to rearrange the computation such that portions of different loop iterations execute at once, similarly to hardware pipelining. A prologue is required to start the pipelined computation, and an epilogue is required to "flush the pipeline" at the end of the computation.

Figure 5 gives the desired functionality in C. Figure 6 gives partially optimized, but unpipelined assembly code, which is reasonably readable.

Figure 7 gives the software pipelined code presented in the article. Intuitively, the @-signs in the comments indicate which iteration of the original loop is being processed by which instructions. For example, the first iteration of the unpipelined loop corresponds to the LDW instructions on lines 17 and 18 of Figure 7, followed by the SUB instruction on line 25, the branch on line 27, the multiply on line 31, and so forth. It is also crucial to remember the five branch delay slots. The performance improvement is that the loop kernel now runs in 2 cycles instead of 10.

We ran our tool on this example, comparing the pipelined and unpipelined programs. To our surprise, the tool discovered a bug: the result of the first iteration is never written, and there is an extra result written at the end. A fix, which our tool verified as correct, is to move a STW instruction from the epilog to the prolog. We published our results,[28]

```
void example1(float *out,float *input1,float *input2)
{
    int i;

    for(i= 0 ; i < 100;i++)
    {
        out[i]=input1[i] * input2[i];
    }
}
```

Fig. 5. Software pipelining example. This C code specifies the desired functionality. (Listing taken from.[32])

```
(... linkage and initialization omitted.
     B0 is the loop counter ...)
13 L12:    ; PIPED LOOP KERNEL
14      LDW    .D2    *B5++,B4   ; 2 loads in parallel
15 ||   LDW    .D1    *A3++,A0
16      NOP    2          ; 2 cycle NOP for load delay
17 [ B0]SUB    .L2    B0,1,B0
18 [ B0]B      .S2    L12       ; 5 branch delay slots
19      MPYSP  .M1X   B4,A0,A0
20      NOP    3                  ; wait for multiply
21      STW    .D1    A0,*A4++
(... subroutine return omitted ...)
```

Fig. 6. Unpipelined assembly code. According to the article,[32] the compiler was unwilling to pipeline the loop because of possible aliasing between the inputs and the output. The correspondence with the C code is fairly clear.

```
(... linkage and initialization omitted
     B0 is the loop counter ...)
15 L8:    ; PIPED LOOP PROLOG
16
17          LDW   .D2   *B5++,B4  ;
18 ||       LDW   .D1   *A3++,A0  ;
19
20          NOP   1
21
22          LDW   .D2   *B5++,B4  ;@
23 ||       LDW   .D1   *A3++,A0  ;@
24
25     [B0] SUB   .L2   B0,1,B0   ;
26
27     [B0] B     .S2   L9        ;
28 ||       LDW   .D2   *B5++,B4  ;@@
29 ||       LDW   .D1   *A3++,A0  ;@@
30
31          MPYSP .M1X  B4,A0,A5  ;
32 || [B0]  SUB   .L2   B0,1,B0   ;@
33
34     [B0] B     .S2   L9        ;@
35 ||       LDW   .D2   *B5++,B4  ;@@@
36 ||       LDW   .D1   *A3++,A0  ;@@@
37
38          MPYSP .M1X  B4,A0,A5  ;@
39 || [B0]  SUB   .L2   B0,1,B0   ;@@
40
41 ;** ----------------------------*
42 L9:     ; PIPED LOOP KERNEL
43
44     [B0] B     .S2   L9        ;@@
45 ||       LDW   .D2   *B5++,B4  ;@@@@
46 ||       LDW   .D1   *A3++,A0  ;@@@@
47
48          STW   .D1   A5,*A4++  ;
49 ||       MPYSP .M1X  B4,A0,A5  ;@@
50 || [B0]  SUB   .L2   B0,1,B0   ;@@@
51
52 ;** ----------------------------*
53 L10:    ; PIPED LOOP EPILOG
54          NOP   1
55
56          STW   .D1   A5,*A4++  ;@
57 ||       MPYSP .M1X  B4,A0,A5  ;@@@
58
59          NOP   1
60
61          STW   .D1   A5,*A4++  ;@@
62 ||       MPYSP .M1X  B4,A0,A5  ;@@@@
64          NOP   1
65          STW   .D1   A5,*A4++  ;@@@
66          NOP   1
67          STW   .D1   A5,*A4++  ;@@@@
    (... subroutine return omitted ...)
```

Fig. 7. Software pipelined assembly code. If the inputs are declared to be `const`, the compiler performs software pipelining, resulting in much more efficient code. But, does this do the same thing as Fig. 6? (Listing taken from Ref. 32.)

and an astute reader subsequently discovered that our verification tool was wrong, and that the example is actually correct![34]

What happened? The code in the article turned out to be for the floating-point C67x family, although it was described as code for the fixed-point C62x family. We wrote our tool based on the C62x, where multiplies take 2 cycles; on the C67x, multiplies take 4 cycles. In retrospect, an example allegedly for a fixed-point DSP obviously should not use floating-point data, but since our tool uses uninterpreted functions for data operations, we did not notice. We made a trivial change to our verifier (changing the latency table entry for the multiply), and it immediately verified the original example correctly.

Runtime for our tool was less than a second for all runs (on an Intel Pentium III at 733 MHz): finding the "bug", verifying our fix, and verifying the original code with the correct latency. Our tool easily handled both

correct and incorrect versions of intricate, highly optimized code, and was easily modified for different instruction latencies.

4.3. Fujitsu FR500 VLIW Processor

As a further test case, we applied our verification approach to building a tool for the Fujitsu FR500,[35] a high-performance VLIW processor for media-intensive, embedded applications.[29] The FR500 is a 4-wide VLIW, with two instruction slots per cycle for two integer units, and two instructions slots per cycle for floating-point or media instructions. The two floating-point units are each 2-way SIMD 32-bit floating point units, allowing a total of four floating-point operations per cycle. The two media units are each able to execute two 2-way SIMD 16-bit operations, allowing a total of eight operations per cycle.

Although the FR500 has many architecturally interesting features, building a verification tool for it, based on our approach, proved to be very straightforward. No further verification techniques were needed beyond those already described.

As a challenge problem, we were given assembly code output[10] from two different optimizing compiler, which used different optimization levels. The program computes a convolution, with 256 samples and 16 coefficients. The two assembly language programs contained 63 and 100 instructions. The runtime to verify equivalence was 2.6 s on an Intel Pentium III at 733 MHz with 128 MB of memory. Total memory usage was approximately 12 MB. As in the preceding experiments, we see that our verification approach is easily able to handle small segments of code, even with lengthy loop unrolling.

5. RELATED WORK

We do not have space to survey the long research tradition of non-automatic, formal verification of software. Recently, there has also been considerable research on automatic formal verification of software, with the emergence of systems such as Slam,[36] Java Pathfinder,[37] CMC,[38] and CBMC.[39] These works focus on model checking properties of high-level language programs, whereas our focus is on the detailed complexity of embedded software.

There has been some prior work on automatically verifying assembly language programs for embedded systems. For example, Thiry and Claesen[40] proposed a model-checking approach based on BDDs. Balakrishnan

[10]This example is also proprietary.

and Tahar[41] proposed a similar approach based on the more general multiway decision graph to avoid some BDD-size blow-up. Both were able to verify a mouse controller and find inconsistencies between the assembly code and flow chart specifications.

Contemporaneously with us, and using a similar verification approach to ours, Hamaguchi et al.[42] have verified the equivalence of higher-level specifications against lower-level implementations. Furthermore, their lower-level implementation included a VLIW processor with assemblylevel instructions. Subsequent work[43] enhanced performance with better heuristics. Their work addresses the harder problem of high-level-versus-low-level verification, whereas we consider only the problem of comparing two similar low-level programs. On the other hand, their VLIW processor was a simple, academic design (4-wide, 2-stage pipeline, no unusual architectural features), whereas the strength of our work is on handling the complexity of commercial processors and highly optimized code.

In the compiler-research community, Necula[44] has proposed a very similar approach to ours, but targeting the verifier for use on the compiler's intermediate code between optimization passes. His work uses hints from the compiler and more sophisticated control-flow analysis, and was demonstrated by verifying the compilation of the Gnu C compiler itself. We believe that our methods for dealing with the complexity of real assembly language could be augmented by Necula's methods for handling larger programs with more complex control flow.

6. CONCLUSION AND FUTURE WORK

We have reviewed our approach to the formal equivalence verification of embedded software, based on symbolic execution and uninterpreted functions. The approach is flexible and easily adapted to different, even highly complex, instruction set architectures. To make the abstract theory useful in practice requires the use of domain-specific rewriting and other optimizations. In particular, we have found that performing sound-but-incomplete rewriting using user-specified axioms is crucial to being able to successfully verify correctness of real optimized code. Simple rewriting rules can also greatly improve the efficiency of the decision procedure, for example, allowing our functional translation to avoid generating exponential-sized symbolic expressions. Overall, our approach is very promising for the verification of small segments of complex, highly optimized code.

Further research is needed to improve the efficiency, scalability, and accuracy of the verification approach. An obvious way to improve efficiency is to research more efficient decision procedures, since most of the

runtime is spent in the decision procedure. Another important next-step is to exploit better static analysis, especially to avoid loop unrolling.[45] For scalability, research is needed to find ways to decompose a larger verification problem into smaller, more tractable ones—for example, by finding equivalent cut points between two programs, similar to what is done in combinational hardware equivalence verification. For improved accuracy, the most important need is for more accurate bit-level reasoning. Fully bit-accurate verification is likely computationally hopeless, but there may be ways to improve accuracy by combining uninterpreted functions with a bit of bit-level analysis. Another challenge for improving accuracy is to find additional, useful, but decidable theories, or to find more powerful ways to handle user-specified axioms. In general, future research must carefully tune the level of abstraction: too much abstraction yields inaccurate results, but too little results in complexity blow-up.

ACKNOWLEDGMENTS

This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada and from Fujitsu Laboratories of America. Authors David Currie and Mark Kwan participated in this work while students at UBC. We thank Koichiro Takayama of Fujitsu Laboratories of America for encouraging the continuation of this work, and Francky Catthoor of IMEC for pointing out the importance of static analysis techniques for handling higher-dimensional loops. Special thanks to Brett Huber of Texas Instruments for not only spotting our error on the C62x/C67x mix-up, but also figuring out exactly how we went wrong.

REFERENCES

1. R. E. Bryant, A Methodology for Hardware Verification Based on Logic Simulation, *J. ACM*, **38**(2):299–328 (April 1991).
2. R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. Computers*, **C-35**(8):677–691 (August 1986).
3. R. E. Bryant, On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication, *IEEE Trans. Computers*, **40**(2):205–213 (February 1991).
4. J. Joyce, G. Birtwistle, and M. Gordon, *Proving a Computer Correct in Higher Order Logic*, Technical Report UCAM-CL-TR-100, University of Cambridge Computer Laboratory (December 1986).
5. J. R. Burch and D. L. Dill, Automatic Verification of Pipelined Microprocessor Control, *Computer-Aided Verification: Sixth International Conference*, Lecture Notes in Computer Science, Vol. 818, pp. 68–80, Springer (1994).

6. D. Cyrluk and P. Narendran, Ground Temporal Logic: A Logic for Hardware Verification, *Computer-Aided Verification: Sixth International Conference*, Lecture Notes in Computer Science, Vol. 818, pp. 247–259, Springer (1994).
7. R. E. Bryant, S. K. Lahiri, and S. A. Seshia, Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions, *Computer-Aided Verification: 14th International Conference*, Lecture Notes in Computer Science, Vol. 2404, pp. 78–92, Springer (2002).
8. W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland (1954).
9. M. Davis and H. Putnam, A Computing Procedure for Quantification Theory, *J. ACM*, **7**(3):201–215 (July 1960).
10. M. Davis, G. Logemann, and D. Loveland, A Machine Program for Theorem Proving, *Commun. ACM*, **5**(7):394–397 (July 1962).
11. J. P. Marques Silva and K. A. Sakallah, GRASP—A New Search Algorithm for Satisfiability, *International Conference on Computer-Aided Design*, pp. 220–227, IEEE/ACM (1996).
12. H. Zhang, SATO: An Efficient Propositional Prover, *14th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 1249, pp. 272–275, Springer (1997).
13. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, Chaff: Engineering an Efficient SAT Solver, *38th Design Automation Conference*, pp. 530–535, ACM/IEEE (2001).
14. G. Nelson and D. C. Oppen, Fast Decision Procedures Based on Congruence Closure, *J. ACM*, **27**(2):356–364 (1980).
15. R. E. Bryant, S. German, and M. N. Velev, Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions, *Computer-Aided Verification: 11th International Conference*, Lecture Notes in Computer Science, Vol. 1633, pp. 470–482, Springer (1999).
16. R. E. Shostak, Deciding Combinations of Theories, *J. ACM*, **31**(1):1–12 (January 1984).
17. D. Cyrluk, P. Lincoln, and N. Shankar, On Shostak's Decision Procedure for Combinations of Theories, in M. A. McRobbie and J. K. Slaney (eds.), *Automated Deduction—CADE-13*, number 1104 in Lecture Notes in Artificial Intelligence, pp. 463–477, Springer-Verlag, New Brunswick, NJ (July/August 1996).
18. C. Barrett, D. Dill, and J. Levitt, Validity Checking for Combinations of Theories with Equality, *Formal Methods In Computer-Aided Design: First International Conference*, *Lecture Notes in Computer Science*, Vol. 1166, pp. 187–201, Springer (1996), currently, software is available at `http://chicory.stanford.edu/SVC`.
19. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar, ICS: Integrated Canonizer and Solver, *Computer-Aided Verification: 13th International Conference*, Lecture Notes in Computer Science, Vol. 2102, pp. 246–249, Springer (2001).
20. A. Stump, C. W. Barrett, and D. L. Dill, CVC: A Cooperating Validity Checker, *Computer-Aided Verification: 14th International Conference*, Lecture Notes in Computer Science, Vol. 2404, pp. 500–504, Springer (2002).
21. D. Detlefs, G. Nelson, and J. B. Saxe, *Simplify: A Theorem Prover for Program Checking*, Technical Report HPL-2003-148, HP Labs (2003).
22. C. Barrett and S. Berezin, CVC Lite: A New Implementation of the Cooperating Validity Checker, *Computer-Aided Verification: 16th International Conference*, Lecture Notes in Computer Science, Vol. 3114, pp. 515–518, Springer (2004).
23. L. de Moura and H. Rueß, An Experimental Evaluation of Ground Decision Procedures, *Computer-Aided Verification: 16th International Conference*, Lecture Notes in Computer Science, Vol. 3114, pp. 162–174, Springer (2004).

24. C. Blank, H. Eveking, J. Levihn, and G. Ritter, Symbolic Simulation Techniques—State-of-the-Art and Applications, *International Workshop on High-Level Design, Validation, and Test*, pp. 45–50, IEEE (2001).
25. D. W. Currie, *A Tool for Formal Verification of DSP Assembly Language Programs*, Master's thesis, University of British Columbia (August 1999).
26. D. W. Currie, A. J. Hu, S. Rajan, and M. Fujita, Automatic Formal Verification of DSP Software, *37th Design Automation Conference*, pp. 130–135, ACM/IEEE (2000).
27. A. Sudarsanam, S. Malik, S. Rajan, and M. Fujita, Development of a High-Quality Compiler for a Fujitsu Fixed-Point Digital Signal Processor, *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, pp. 2–7, ACM SIGDA, Rome (May 1999).
28. X. Feng and A. J. Hu, Automatic Formal Verification for Scheduled VLIW Code, *Joint Conference on Languages, Compilers, and Tools for Embedded Systems, and Software and Compilers for Embedded Systems*, pp. 85–92, ACM SIGPLAN (2002).
29. X. Feng, *Automatic Formal Verification for Scheduled VLIW Code*, Master's thesis, University of British Columbia (August 2002).
30. Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide* (October 2000), literature Number SPRU189F.
31. W.-M. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, Compiler Technology for Future Microprocessors, *Proc. IEEE*, **83**(12):1625–1640 (December 1995).
32. R. Oshana, Optimization Techniques for High-Performance DSPs, *Embedded Systems Programming* (March 1999), we accessed the on-line article at `http://www.embedded.com/1999/9903/9903osha.htm`.
33. M. S. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Conference on Programming Language Design and Implementation*, pp. 318–328, ACM SIGPLAN (1988).
34. B. Huber, private communication, 27 August 2002.
35. T. Sukemura, FR500 VLIW-architecture High-Performance Embedded Microprocessor, *Fujitsu Scientific Technical J.*, **36**(1):31–38 (June 2000).
36. T. Ball and S. K. Rajamani, The SLAM Toolkit, *Computer-Aided Verification: 13th International Conference*, number 2102 in Lecture Notes in Computer Science, pp. 260–264, Springer (2001).
37. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, Model Checking Programs, *Automated Software Engineering*, **10**(2):203–232 (April 2003).
38. M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, CMC: A Pragmatic Approach to Model Checking Real Code, *Symposium on Operating Systems Design and Implementation*, pp. 75–88, ACM SIGOPS (2002).
39. E. Clarke, D. Kroening, and F. Lerda, A Tool for Checking ANSI-C Programs, *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science*, Vol. 2988, pp. 168–176, Springer (2004).
40. O. Thiry and L. Claesen, A formal verification technique for embedded software, *IEEE International Conference on Computer Design*, pp. 352–357, IEEE Computer Society Press, New York, USA (1996).
41. S. Balakrishnan and S. Tahar, *On the Formal Verification of Embedded Systems Using Multiway Decision Graphs*, Technical Report TR-402, Concordia University, Montreal, Canada (1997).
42. K. Hamaguchi, H. Urushihara, and T. Kashiwabara, Symbolic Checking of Signal-Transition Consistency for Verifying High-Level Designs, *Formal Methods in*

Computer-Aided Design: Third International Conference, Lecture Notes in Computer Science, Vol. 1954, pp. 455–469, Springer (2000).
43. K. Hamaguchi, Symbolic Simulation Heuristics for High-Level Design Descriptions with Uninterpreted Functions, *International Workshop on High-Level Design, Validation, and Test*, pp. 25–30, IEEE (2001).
44. G. C. Necula, Translation Validation for an Optimizing Compiler, *Conference on Programming Language Design and Implementation*, pp. 83–94, ACM SIGPLAN (2000).
45. K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens, Automatic Functional Verification of Memory Oriented Global Source Code Transformations, *International Workshop on High-Level Design, Validation, and Test*, pp. 31–36, IEEE (2003).