Automatic Formal Verification for Scheduled VLIW Code *

Xiushan Feng
Department of Computer Science
University of British Columbia
xsfeng@cs.ubc.ca

Alan J. Hu
Department of Computer Science
University of British Columbia
ajh@cs.ubc.ca

ABSTRACT

VLIW processors are attractive for many embedded applications, but VLIW code scheduling, whether by hand or by compiler, is extremely challenging. In this paper, we extend previous work on automated verification of low-level software to handle the complexity of modern, aggressive VLIW designs, e.g., the exposed parallelism, pipelining, and resource constraints. We implement these ideas into a prototype tool for verifying short sequences of assembly code for TI's C62x family of VLIW DSPs, and demonstrate the effectiveness of the tool in quickly verifying, or finding bugs in, two difficult-to-analyze code segments.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]; D.2.5 [**Testing and Debugging**]: Symbolic execution

General Terms

Verification

Keywords

Formal Verification, Symbolic Execution, Theory of Equality with Uninterpreted Functions, VLIW, DSP

1. INTRODUCTION

VLIW processors are attractive for many embedded applications because of their promise of very high performance without the power- and die-area-consuming instruction-scheduling logic of superscalar processors, and because backward object-code compatibility is typically unnecessary for embedded applications. Unfortunately, the exposed parallelism, pipelining, and resource conflicts in an aggressive VLIW design, coupled with the lack of interlocks, makes generating and optimizing code for a VLIW processor extremely challenging. The result is often code that is buggy, or else excessively conservative and, therefore, sub-optimal.

*This work was supported in part by a research grant from Fujitsu Laboratories of America.

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

Formal verification can help address this problem, provided that the techniques are efficient and automatic enough for very frequent use. Obviously, formal verification can improve the quality of code, by verifying the correctness of optimizations performed by the compiler or by hand. Furthermore, formal verification can actually enable more efficient code, by proving the impossibility of certain resource conflicts, or confirming the correctness of highly unintuitive optimizations. Later in this paper, we will see examples of both.

In this paper, we extend previous work on verifying assembly code at the instruction set architecture (ISA) level. For a modern, aggressive VLIW design, the ISA actually exposes much of the underlying microarchitectural complexity. We introduce several innovations to handle this complexity. First, we model the parallel execution of multiple functional units. To accurately model pipeline delays, we introduce a simplified model of the pipeline that allows us to compute the correct result and also verify the presence/absence of resource conflicts. We also model predicated execution precisely, which permits a very detailed and accurate analysis of resource conflicts.

We have implemented our ideas in a prototype verification tool targeting the Texas Instruments C62x family of VLIW digital signal processors. Run times on short code segments are negligible, yet the tool is able to verify the absence of resource conflicts in a situation previously considered unanalyzable, and the tool also automatically found a bug in a published code optimization example written by a recognized expert. These examples demonstrate the potential usefulness of our ideas.

2. RELATED WORK

This work builds directly on the work of Currie et al. [3]. In that work, the authors demonstrated the feasibility of automatic verification of low-level software via symbolic execution and an automated decision procedure. The verification task was to compare two short assembly language routines for equivalence, assuming considerable similarity in control flow, as would occur, for instance, after hand-optimizing a performance-critical kernel. The verification method was demonstrated by building a verification tool targeting a very simple, 16-bit fixed-point digital signal processor. The present paper is the natural extension of Currie et al.'s work to a highly complex, modern VLIW processor.

Hamaguchi et al. [5] employed a similar approach to verify a high-level specification against a low-level implementation. Furthermore, their low-level implementation included a VLIW processor with assembly-level instructions. Subsequent work [4] 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 low-level programs. However, their VLIW processor was a simple, academic design (4-wide, 2-stage pipeline, no unusual architectural features), whereas we tackle a high-end commercial VLIW. The emphasis of our work is on how to deal with the architectural features of VLIW processors, rather than on improving the efficiency of the verification approach, and we believe both directions of research are needed for widespread, practical impact.

Blank et al. [2] have recently presented a broad survey of the general verification paradigm (symbolic simulation/execution with uninterpreted functions) along with techniques for greatly improved efficiency. Unfortunately, we became aware of their work after our project was well underway, so we have not harnessed their ideas yet. In particular, we are building symbolic expressions for results as functions of the original inputs, and these expressions can blow-up exponentially. In contrast, their approach introduces temporary variables for intermediate results, eliminating this expression-size blow-up entirely. To date, we have avoided the expression-size blow-up by some rewriting techniques, but their translation approach is likely the superior solution in general.

In the compiler-research community, Necula [8] has proposed a very similar approach to that of Currie et al., but targeting the verifier for use by the compiler 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 this translation-validation-during-compilation is a promising approach, and that our methods for dealing with the complexity of a high-performance VLIW processor could be integrated with Necula's methods for verifying large programs with complex control flow.

3. VERIFICATION METHOD

3.1 Basic Verification Approach

For completeness, we briefly review the basic verification approach we use, which is the same as that taken by Currie et al. [3]. The interested reader should consult that work for more details.

The verification task is to take two assembly-language routines, which compute some values and terminate, and verify that they are equivalent. The user specifies what inputs are initially equal and what outputs should be equal when the routines terminate. The assumption is that the two routines have very similar control-flow. If this assumption is violated, the verifier might declare inequivalent two routines that really compute the same value, but it will not claim equivalence for two routines that are not. As in [3], some additional simplifying assumptions are needed (e.g., no self-modifying code, no recursion, etc.) but for space reasons, we do not repeat them here.

The verification procedure requires a simple model of the processor at the instruction set architecture level, and then uses this model to simulate the two routines. However, instead of computing actual values, the simulator is symbolic and computes expressions that denote the values as a function of the initial inputs and states. For example, if we execute the two following instructions for a fictitious processor:

```
load 4(R2), R1  ; R1 := memory[R2+4]
add R3, R1, R3  ; R3 := R3 + R1
```

then we would compute the "value" of register R1 to be:

$$(\texttt{read init\_mem} \ (+\ 4\ \texttt{init\_R2}))$$

and then we would compute the "value" of register R3 to be:

$$(+\ \texttt{init\_R3}\ (\texttt{read init\_mem}\ (+\ 4\ \texttt{init\_R2})))$$

where the symbols that start with `init_` denote the initial values at the start of the program, "+" is addition, and "read" is a function used to access memory. To abstract away datapath complexity, most machine operations are treated as *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). This abstraction is safe, but is sometimes too conservative — being unable to prove the equivalence of a shift and a divide-by-2, for example — so additional domain-specific rewriting rules are needed to handle these cases. In general, the logic used for the expressions is the theory of uninterpreted functions with equality, augmented with linear arithmetic and the `read` and `write` functions to access memories. Efficient decision procedures exist for this logic; we use the Stanford Validity Checker (SVC) [1].

Forward and backward branching are fundamentally different. For backward branches, we essentially unroll the loop. The simple DSP used in Currie et al.'s work had explicit `for`-loop instructions, so it was trivial to unroll fixed-count loops. Other kinds of looping were not supported. In our case, we use a more general technique: if the decision procedure can prove that the a branch is taken, we take the branch; if the decision procedure can prove that a branch is not taken, we don't take it; otherwise, we declare that the code contains branching that we do not handle. All fixed-count loops, which are the common case in low-level DSP code, can be handled this way, but we cannot currently handle general control flow (e.g., `while`-loops, arbitrary `goto`s, etc.) Forward branching is handled by case-splitting, with the help of the decision procedure. Basically, when both routines reach a branch, the decision procedure is called to verify that the branch conditions are compatible — that the two programs must always branch the same way or always branch opposite ways. The verification proceeds to verify that the programs are equivalent for both the taken and not-taken paths. In the C6x family, all instructions are predicated, so we rarely encounter forward branches. Accordingly, our present tool does not implement the case-splitting and treats forward branches in the same manner as backward branches.

3.2 Texas Instruments' 320C62x VLIW DSP

The motivation for our research was to extend the basic verification approach described above to handle complex, modern VLIW processors. Texas Instruments' TMS320C6x family of VLIW digital signal processors [10] is both commercially important and also the epitome of this architectural style, so we have targeted this family for our research prototype. In particular, we are targeting 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 this subsection, we briefly highlight the salient architectural features that make verification difficult; subsequently, we present our solutions to these difficulties.

The C6x processors are 32-bit VLIW digital signal processors. Instructions are grouped into "execute packets" of up to eight instructions, and these packets can be executed one per clock cycle. Figure 1 shows the datapath. Of particular note are both the capability of very high performance, but also the striking non-orthogonality of the design. 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.

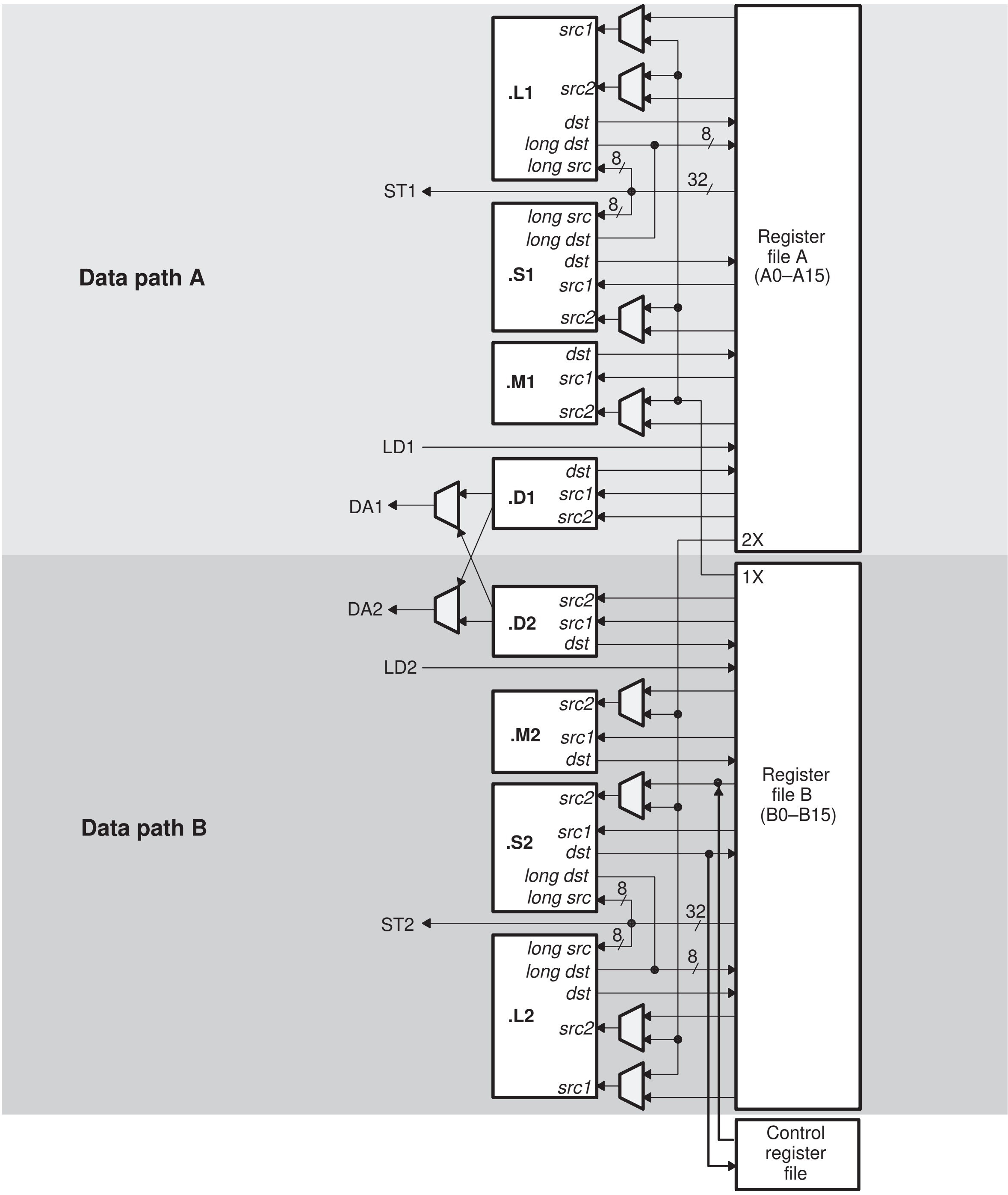

Figure 1: TI C62x Datapath Block Diagram. Registers are in two banks, with limited pathways connecting them (1X and 2X). The .L and .S units are integer ALUs with slightly different properties. The .D units are also integer ALUs, but designed for address calculation. The .M units are multipliers. LD and ST are load and store paths, with DA being data address paths. This figure is from the processor reference guide [10, Figure 2-1], and is ©Texas Instruments 2000, used here courtesy of Texas Instruments.

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.

Table 1: C62x Processor Pipeline. The processor pipeline is 11 stages deep. (TI literature refers to the pipeline stages as "phases", but we adopt standard terminology here.) 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 [10, Table 6-1].

The other main architectural feature that both enhances performance and complicates code generation and verification is the pipeline. For maximum throughput, the processor is heavily pipelined, with instructions taking up to 11 cycles to complete. Table 1 summarizes the pipeline. The pipeline has no interlocks, which simplifies the hardware (more performance at lower cost) but complicates the code. For most instructions, the operand read and result write occur in a single stage, so there are no hazards. However, 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. In particular, branches have five delay slots (i.e., the next five execute packets following a branch always execute[1], 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. Unlike many processors, branch instructions may appear in the delay slots of other branches. The results in that case may be unintuitive, but they follow naturally from the pipeline definition. To ease coding, the architecture provides a multicycle NOP instruction, which is equivalent to a series of *n* NOPs executed in sequence.

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 is known to eliminate many branches, increasing the size of basic blocks and the amount of instruction-level parallelism available [6]. 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 [10, Section 3.7.6]. We will demonstrate that these situations actually can be analyzed.

In sum, this 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 claim that although the code is very error-prone for a human to read or write, it is relatively easy to create a simulator for the processor — even a symbolic simulator — that captures the correct semantics. The following subsections describe how we extended the basic verification approach to handle the challenges of this architecture.

3.3 Parallel Execution

The most obvious difference of a VLIW is the multiple instructions executing simultaneously. This is trivially handled in the symbolic simulator. We simply simulate each instruction in an execute packet one-by-one, [2] but we don't update the register file until after all instructions have read their operands. The resulting behavior is identical to parallel execution.

This is also the logical point in the tool to check for statically detectable resource conflicts. We didn't bother to implement these checks, however, since they are a straightforward check of the execute packet against various simple rules, and existing tools handle these checks already, so the problem was uninteresting from a research perspective.

3.4 Predication

Three issues arise in dealing with predication: the basic functionality, conditional branching, and resource conflicts. The basic functionality of predication is easily expressible in our logic. Indeed, the SVC logic provides an if-then-else operator, so the tool simply generates the expression that evaluates to the new value if the predicate is true, or the old value if the predicate is false. For example, the value of the register `A3` after this instruction:

```
[B0] ADD .L1 A1, A2, A3
```

is

```
(ITE (/= cur_B0 0) (+ cur_A1 cur_A2) (cur_A3))
```

where `/=` denotes not-equal and the `cur_` symbols are abbreviations for the current expressions for the values of the registers.

[1] Unless we're already in the delay slots of a preceding branch.

[2] Our current tool implements only a subset of common instructions, since the point of our research is to explore the verification technique rather than build a production tool.

Predicated branches are handled exactly as conditional branches are handled in the basic verification approach. The only difference is the lengthy branch delay, which we describe in the next subsection. We will also deal with resource conflicts later in the paper.

3.5 Branches

Because branches have five delay slots, there exists the possibility that other branch instructions will occur in those slots. In fact, this occurs frequently in tight loop kernels. For example, consider the following code fragment:

```
        B .S1 Loop1

        MVK .S2 10, B1
||      MVK .S1 0, A0

Loop2:
        ADD .L1 A0, 1, A0
|| [B1] B   .S1 Loop2
|| [B1] SUB .S2 B1, 1, B1

        STW .D1 A0. *A1++
        NOP 2
Loop1:
   [B1] ADD .L1 A0, 1, A0
|| [B1] SUB .S2 B1, 1, B1
        NOP 5
```

(The || separates instructions in the same execute packet.) In this example, the branch to `Loop2` occurs in the delay slots of the branch to `Loop1`, as well as in its own delay slots on subsequent iterations. The net effect is that the code loops until register `B1` reaches 0. Although this example is contrived, coding in this style is not atypical.

We would like to avoid explicitly modeling the instruction packet fetch mechanism, which is actually somewhat more complex than we have described. Instead, we introduce a small queue for 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.

3.6 Read-After-Write Delay Slots

As with branch delay slots, delay slots resulting from different instruction latencies are the result of the programmer-visible pipeline, and we handle them in a similar fashion. Conceptually, these delay slots simply delay the assignment of results to the register file. Accordingly, we can avoid modeling the full pipeline, and use small queues to delay the writes to the register file.

More precisely, we extend the data structure used for 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. These queues are very short — the longest latency is four delay slots for loads, so the queue is only five entries long (current value and four delay slots). (See Figure 2.) 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.

3.7 Resource Conflicts

We've already mentioned the uninteresting resource conflicts that are easily checked whether in our tool or statically. The interesting case is when multiple instructions attempt to write to the same register. This could happen because of multiple instructions within one execute packet, or because of instructions with different latencies trying to write in the same cycle. For example,

```
MPY .M1 A0, A1, A2
ADD .L1 A4, A5, A2
```

will result in both instructions trying to write register `A2` simultaneously, because the multiply instruction has one delay slot.

These register-write conflicts could still be checked statically during code generation, or in our tool by disallowing multiple assignments to the same queue position. The situation becomes more complex with predication. Indeed, the processor reference manual explicitly states that examples like:

```
   [!B1] ADD .L2 B5, B6, B7
|| [B0] SUB .S2 B8, B9, B7
```

are legal if zero or one predicates are true, but will produce undefined results if more than one predicate is true, and that the presence or absence of a conflict cannot be detected [10, Section 3.7.6].

Our work contradicts this claim; in most instances, these conflicts can be checked. To handle this case, we extend the verifier further. Instead of a queue of symbolic expressions for each register, we have a queue of *write-histories* for each register. Each write-history is a list of all the (symbolic expression) values that are scheduled to be assigned to that register in that cycle, along with the predicate condition required for the assignment to take place. In the simplest cases, the write-history is either empty (no write to the register) or has a single, unpredicated expression. Whenever a new expression is scheduled to be assigned in a given cycle, it is first checked against all entries already in the write-history. The decision procedure determines if the predicate is mutually exclusive with all the other writes. If so, the tool has **proven** that this assignment is conflict-free; if not, the tool flags the potential conflict to the user.

4. EXAMPLES

4.1 Dynamic Resource Conflicts

Our first example is a small one to illustrate our ability to detect/disprove dynamic resource conflicts.

Consider the following code segment:

```
         ADD .L2  B1, B0, B2
         ADD .L2  B2, B0, B3

   [!B1] ADD .L1  A0, A1, A0
|| [!B2] ADD .S1  A0, A2, A0
|| [!B3] ADD .D1  A0, A3, A0
```

The first two `ADD` instructions initialize `B1` through `B3` to be equally spaced `B0` apart. The next three `ADD` instructions are in a single execute packet and might conflict in writing to `A0` in the same cycle. Our tool flags this as a potential conflict, because `B0` might be equal to zero.

If we prepend an instruction guaranteeing that `B0` is not zero:

```
         MVK .S2  1, B0
         ADD .L2  B1, B0, B2
         ADD .L2  B2, B0, B3
```

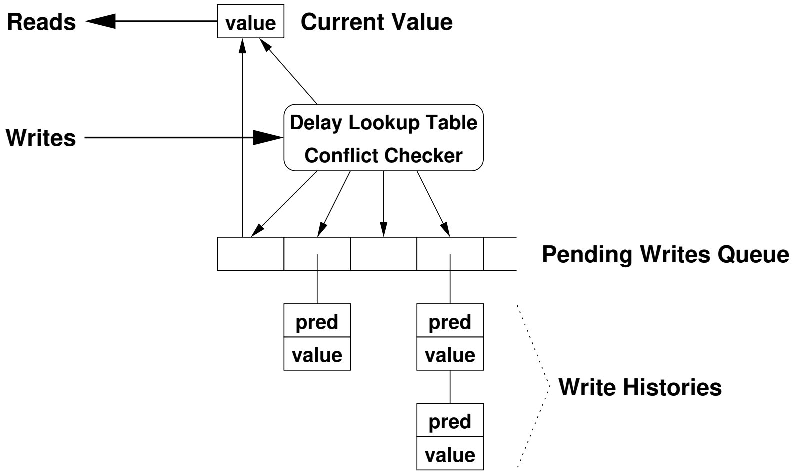

Figure 2: Register Model. This figure shows the data structures used to model each register. In the basic verification algorithm, only the "Current Value" variable would exist, and reads and writes would access and update this variable. To handle delay slots, we add a queue of pending writes. Reads still 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.

```
   [!B1]  ADD .L1  A0, A1, A0
|| [!B2]  ADD .S1  A0, A2, A0
|| [!B3]  ADD .D1  A0, A3, A0
```

the tool verifies that a conflict cannot happen. This class of resource conflict was previously considered to be unanalyzable, but runtime on our examples was a tiny fraction of a second. Integrating the analysis we are proposing into a compiler should enable generating more efficient code with greater confidence in its correctness.

4.2 Software Pipelining

Our second example is somewhat larger and demonstrates the verification of a non-trivial code optimization. The example is taken from an article, written by an expert on DSP code optimization, explaining how to optimize code for high-performance DSPs [9]. This article appeared in a trade magazine, and also on the magazine's website.

The most difficult example in the article demonstrates software pipelining a short loop, targeting the C6x. Software pipelining is a powerful instruction scheduling technique that exposes additional parallelism in loops, thereby improving performance [7]. 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 3 gives the desired functionality in C. Figure 4 gives partially optimized, but unpipelined assembly code, which is reasonably readable. Figure 5 gives the software pipelined code presented in the article. According to the article, this code was generated by the compiler if the input vectors are declared to be constants. This code is quite hard to read, but the article clearly explains the principles involved. 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 5, 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. Another way to understand the code is to note that the loop kernel on lines 44-50 of Figure 5 performs all the same operations as the original, unpipelined loop kernel, but can do far more operations in parallel because different loop iterations are being processed at the same time. The performance improvement is that the loop kernel now runs in 2 cycles instead of 10.

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

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

Figure 3: Software Pipelining Example. This C code specifies the desired functionality. (Listing taken from [9].)

We ran our tool on this example, comparing the pipelined and unpipelined programs. To our surprise, the tool discovered the pipelined code has a bug. In particular, the result of the first iteration is never written, and there is an extra result written at the end. The fix is to add an additional `STW` instruction to the packet at lines 38–39, and to delete the `STW` instruction at line 67. Runtime for our tool was less than a second both for finding the bug as well as for verifying our fix. We have confirmed our findings on real hardware.

```
_example1:
;** ------------------------------*
     MVK    .S2    0x64,B0

     MVC    .S2    CSR,B6
 ||    MV   .L1X   B4,A3
 ||    MV   .L2X   A6,B5
     AND    .L1X   -2,B6,A0
     MVC    .S2X   A0,CSR
;** -----------------------------*
L11:    ; PIPED LOOP PROLOG
;** -----------------------------*
L12:    ; PIPED LOOP KERNEL
      LDW    .D2    *B5++,B4
||    LDW    .D1    *A3++,A0
      NOP    2
[ B0]SUB    .L2    B0,1,B0
[ B0]B      .S2    L12
      MPYSP  .M1X   B4,A0,A0
      NOP    3
      STW    .D1    A0,*A4++
;** -----------------------------*
      MVC    .S2    B6,CSR
      B      .S2    B3
      NOP    5
; BRANCH OCCURS
```

Figure 4: Unpipelined Assembly Code. According to the article [9], this code was compiler-generated, and 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.

We are in no way criticizing the author of the article. Indeed, the article very clearly explains and illustrates a difficult code optimization. We do not know if this error arose during compilation, during the writing of the article, or somewhere in the editorial process of publication. The important point is that the coding style that attains maximum performance is so difficult to comprehend that a bug crept into automatically generated code and then eluded detection by the expert who was using the code as an example, as well as by the countless readers who studied the article. This clearly demonstrates both the applicability of our techniques, and the importance of using verification as processor architectures and compiler optimizations become increasingly complex.

5. CONCLUSION AND FUTURE WORK

We have extended previous work on verification of low-level code to handle the complexity of modern, high-performance VLIW processors. Our proof-of-concept tool demonstrates that our approach can easily find bugs or confirm correctness in situations that are extremely challenging without automated assistance.

The most obvious direction for future work is to continue development on our tool to fully support the C6x architecture. This entails better control-flow analysis and support, as well as implementing all instructions and checks for static resource conflicts. Eventually, such a tool should be integrated into compilers or integrated development environments.

The direction of future development hinges largely on the techniques available for dealing with control flow. If we are able to find effective means for analyzing programs, then it is possible to develop a highly useful stand-alone verification tool. If the control flow analysis remains a difficult challenge, then the more likely path for future impact is to integrate the verification into the compiler, which has more information available to it on the control flow of the program.

In the long term, the broad trend in compiler technology is towards ever more sophisticated program analysis to enable better and better compilation. The verification style we are proposing is, in some sense, just a more detailed, careful analysis of the code. Our work shows that such analysis is becoming feasible and useful. In the future, such techniques may become standard components of optimizing compilers.

Acknowledgments

We would like to thank Masahiro Fujita, Sreeranga Rajan, and Koichiro Takayama for suggesting that we investigate VLIW processors, and Jizhong Han for access to a C6x development system and assistance in testing our results.

6. REFERENCES

[1] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design: First International Conference*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 1996. Currently, software is available at `http://sprout.stanford.edu/SVC`.

[2] C. Blank, H. Eveking, J. Levihn, and G. Ritter. Symbolic simulation techniques — state-of-the-art and applications. In *International Workshop on High-Level Design, Validation, and Test*, pages 45–50. IEEE, 2001.

[3] D. W. Currie, A. J. Hu, S. Rajan, and M. Fujita. Automatic formal verification of DSP software. In *37th Design Automation Conference*, pages 130–135. ACM/IEEE, 2000.

[4] K. Hamaguchi. Symbolic simulation heuristics for high-level design descriptions with uninterpreted functions. In *International Workshop on High-Level Design, Validation, and Test*, pages 25–30. IEEE, 2001.

[5] K. Hamaguchi, H. Urushihara, and T. Kashiwabara. Symbolic checking of signal-transition consistency for verifying high-level designs. In *Formal Methods in Computer-Aided Design: Third International Conference*, pages 455–469. Springer, 2000. Lecture Notes in Computer Science Vol. 1954.

[6] 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. *Proceedings of the IEEE*, 83(12):1625–1640, December 1995.

[7] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Conference on Programming Language Design and Implementation*, pages 318–328. ACM SIGPLAN, 1988.

[8] G. C. Necula. Translation validation for an optimizing compiler. In *Conference on Programming Language Design and Implementation*, pages 83–94. ACM SIGPLAN, 2000.

[9] 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`.

[10] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, October 2000. Literature Number SPRU189F.

```
_example2:
;** ------------------------------*
        MVK   .S2   0x64,B0
        MVC   .S2   CSR,B6
||      MV    .L1X  B4,A3
||      MV    .L2X  A6,B5
        AND   .L1X  -2,B6,A0
        MVC   .S2X  A0,CSR
||      SUB   .L2   B0,4,B0
;** ------------------------------*
L8:    ; PIPED LOOP PROLOG
        LDW   .D2   *B5++,B4  ;
||      LDW   .D1   *A3++,A0  ;
        NOP   1
        LDW   .D2   *B5++,B4  ;@
||      LDW   .D1   *A3++,A0  ;@
   [B0] SUB   .L2   B0,1,B0   ;
   [B0] B     .S2   L9        ;
||      LDW   .D2   *B5++,B4  ;@@
||      LDW   .D1   *A3++,A0  ;@@
        MPYSP .M1X  B4,A0,A5  ;
|| [B0] SUB   .L2   B0,1,B0   ;@
   [B0] B     .S2   L9        ;@
||      LDW   .D2   *B5++,B4  ;@@@
||      LDW   .D1   *A3++,A0  ;@@@
        MPYSP .M1X  B4,A0,A5  ;@
|| [B0] SUB   .L2   B0,1,B0   ;@@
;** ------------------------------*
L9:    ; PIPED LOOP KERNEL
   [B0] B     .S2   L9        ;@@
||      LDW   .D2   *B5++,B4  ;@@@@
||      LDW   .D1   *A3++,A0  ;@@@@
        STW   .D1   A5,*A4++  ;
||      MPYSP .M1X  B4,A0,A5  ;@@
|| [B0] SUB   .L2   B0,1,B0   ;@@@
;** ------------------------------*
L10:    ; PIPED LOOP EPILOG
        NOP   1
        STW   .D1   A5,*A4++  ;@
||      MPYSP .M1X  B4,A0,A5  ;@@@
        NOP   1
        STW   .D1   A5,*A4++  ;@@
||      MPYSP .M1X  B4,A0,A5  ;@@@@
        NOP   1
        STW   .D1   A5,*A4++  ;@@@
        NOP   1
        STW   .D1   A5,*A4++  ;@@@@
;** ------------------------------*
        MVC   .S2   B6,CSR
        B     .S2   B3
        NOP   5
; BRANCH OCCURS
```

Figure 5: 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 Figure 4? (Listing taken from [9].)