# Cutpoints for Formal Equivalence Verification of Embedded Software [*]

Xiushan Feng Alan J. Hu
Department of Computer Science, University of British Columbia
{xsfeng, ajh}@cs.ubc.ca

## ABSTRACT

Like hardware, embedded software faces stringent design constraints, undergoes extremely aggressive optimization, and therefore has a similar need for verifying the functional equivalence of two versions of a design, e.g., before and after an optimization. The concept of cutpoints was a breakthrough in the formal equivalence verification of combinational circuits and is the key enabling technology behind its successful commercialization. We introduce an analogous idea for formally verifying the equivalence of structurally similar, "combinational" software, i.e., software routines that compute a result and return/terminate, rather than executing indefinitely. We have implemented a proof-of-concept cutpoint approach in our prototype verification tool for the TI C6x family of VLIW DSPs, and our experiments show large improvements in runtime and memory usage.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

## General Terms

Verification

## Keywords

embedded software, equivalence checking, formal verification

## 1. INTRODUCTION

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 artifacts (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. We focus on a particular verification problem: verifying the functional equivalence of two similar segments of low-level code, as would be needed, for example, after hand-tuning compiler-generated code.

Automatic formal verification of software has been enjoying a renaissance lately, with much of the focus on extending finite-state model checking [10] — which has been successfully applied to sequential circuits, protocols, and other reactive systems — to software, viewed at a system-level as a reactive (non-terminating) system (e.g., [2, 17, 27]). A complementary line of work, more relevant to this paper, has focused on formally verifying the equivalence of low-level code, e.g., to higher-level specifications [26, 1, 16], to other versions of low-level code [12, 15], or to hardware [9, 23]. This line of research typically verifies a relatively small segment of code as a transformational rather than reactive system, i.e., the code computes a result and terminates, analogous to a combinational circuit in hardware. The basic approach is to use symbolic execution of the code to compute the formal relationship between inputs and outputs, and then prove that the outputs are always equivalent. The lower-level emphasis is well-suited for the verification of optimizations needed for embedded systems, and indeed, we have demonstrated this approach successfully verifying (or finding bugs in) code optimization for complex embedded processors [12, 15]. Unfortunately, the basic approach is not scalable: the representation of the input/output relationship or the complexity of deciding equivalence blows up in memory, runtime, or both. In one embarrassing example, a 47-line assembly language routine required 15 hours to verify [12]! (Granted, the dynamic instruction count after loop-unrolling was a few thousand instructions, and the verification would have run much faster had certain expensive, but unnecessary, optimizations been disabled.)

The formal equivalence verification of combinational hardware went through a similar evolution. Symbolic simulation [8] auto-

[*]This work was supported in part by research grants from the Natural Sciences and Engineering Research Council of Canada and Intel Corporation.

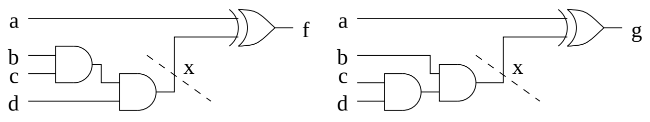


**Figure 1: Simple Cutpoint Example. To introduce cutpoint $x$, we first verify that $(b \wedge c) \wedge d$ is equivalent to $b \wedge (c \wedge d)$. Then, we can verify that $f$ is equivalent to $g$ because both are equal to $a \oplus x$.**

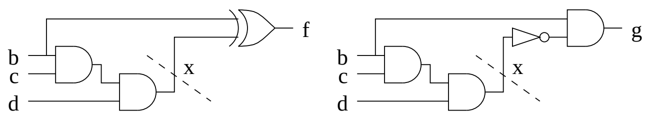


**Figure 2: False Inequivalence. Cutpoint verification fails because $f \neq g$ when $b = 0$ and $x = 1$. However, this is a false inequivalence, because if $x = 1$, then $b$ must be $1$.**

matically computes the input/output relationship for the circuits, which are then compared. BDDs [7] showed considerable promise as an empirically efficient, canonical representation for Boolean functions, but capacity limits prevented the basic approach from scaling to industrial-size problems.

A major practical breakthrough came with the introduction of cutpoints [4, 6]. Given two combinational circuits, presumed to be structurally similar, whose functional equivalence needs to be verified, the idea is to look for points in the two circuits that can be proven to be equivalent. The equivalent logic is cut out of the circuits and is replaced by a new primary input. (Figure 1.) If we can repeat this process all the way to the primary outputs, we have proven the two circuits equivalent, thereby reducing the original verification problem into a sequence of simpler verification problems. Note that the method is conservative: if we fail to prove the circuits equivalent, we cannot conclude that they are inequivalent without further computation. (Figure 2.) Minimizing the cases where the method is unable to prove the equivalence of equivalent circuits (called "false negatives" or "false inequivalence") has been an active research area. The general solution is to re-introduce constraints on the cutpoints, either in advance [6] or as needed [18, 19].

In this paper, we introduce cutpoint-style analysis to the formal equivalence verification of embedded software. Although many concepts are similar to their hardware-verification analogues, we address several novel problems as well: how to define cutpoints for software, how detailed will the cutpoint analysis be, how to find candidate cutpoints, and how to reduce false inequivalences. We have implemented the ideas in our proof-of-concept verification tool targeting the Texas Instruments C6x family of VLIW DSPs. Our preliminary experiments show large improvements in memory usage and runtime over earlier methods.

## 2. BASIC VERIFICATION APPROACH

The present work is built on an existing formal software verification paradigm, which we briefly review here. More extensive introductions are available elsewhere (e.g., [5, 11]).

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 our previous work [12, 15, 11], some additional simplifying assumptions are needed (e.g., no self-modifying code, no recursion, no arithmetic performed on the program counter, etc.); 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, consider the following (TI C6200) code segment:

```
ADD .L2 B1, B0, B2  ; B2:=B1+B0
ADD .L2 B2, B0, B3  ; B3:=B2+B0
```

If we denote the initial values of registers B0, B1, and B2 as $B0_0$, $B1_0$, and $B2_0$, then after these two instructions execute, the simulator will compute the "values" in registers B2 and B3 to be symbolic expressions "$B1_0 + B0_0$" and "$(B1_0 + B0_0) + B0_0$".

We dub the above style of symbolic simulation the "functional translation", because it computes the values at each point as a function of the initial values. An alternative, which we dub the "relational translation", computes for each instruction a clause that relates the values before and after execution. For example, for the above code, we would generate $(B2_1 = B1_0 + B0_0) \wedge (B3_1 = B2_1 + B0_0)$, where the subscripts indicate different versions of the registers at different times.[1] The functional translation can have worst-case exponentially-sized expressions; the relational translation guarantees an expression size linear in the length of the execution sequence, but at the cost of many more variables, which blows up the complexity of deciding equivalence. Others have argued for the superiority of the relational translation [5]; we will revisit this issue later.

To keep the equivalence of symbolic expressions decidable, only constant propagation and linear arithmetic (i.e., symbolic expressions can be added together and multiplied by constants) are interpreted. More complex operations (e.g., multiplication of symbolic expressions, or any arbitrarily complex 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 hides datapath complexity and is safe, but 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 those cases. We also use special interpreted functions `read` — which given a memory and an address, denotes the value at that address — and `write` — which given a memory, an address, and a value, denotes an updated memory in which the value has been written to the address. The key axiom is that

$$\mathrm{read}(\mathrm{write}(m, a1, v), a2) = \begin{cases} v & \text{if } a1 = a2 \\ \mathrm{read}(m, a2) & \text{otherwise} \end{cases}$$

To successfully verify low-level code, we have found it necessary to model memory layout accurately. In particular, when verifying software written in a high-level language, arrays are often assumed

[1]The relational translation may remind some readers of the well-known static single assignment (SSA) form [13], but recall that the simulation is of a dynamic execution trace. A closer analogy is dynamic single assignment form, but since we are considering a single execution trace at a time, the translation is trivial versus the standard computation of DSA [14].

to be disjoint, so the read/write functions can be 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. This approach leads to large symbolic expressions, so we rely on some rewriting optimizations to try to keep expression size manageable [12, 11]. Efficient decision procedures exist for this combined logic (linear arithmetic, uninterpreted functions, and read/write); we use the Stanford Validity Checker (SVC) [3].

We use simple techniques to handle control flow. These techniques proved adequate to handle the bottom-level, highly-optimized computational kernels we are targeting. For backward branches, we essentially unroll loops: if the decision procedure can prove that the branch is taken, we take the branch; if the decision procedure can prove that the 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. For forward branches, we again first try to prove the branch certainly taken or certainly not taken. Otherwise, we case-split. Based on our assumption that the two routines being compared have similar control structure, we require that the two routines encounter "compatible" forward branches in the same order: the two branches must always branch the same way or always branch opposite ways (to allow reordering taken/not-taken paths). If so, our tool proceeds to verify that the routines are equivalent along both paths. If not, our tool declares that it cannot verify the routines equivalent. In the C6x family, all instructions are predicated, so we rarely encounter forward branches.

Overall, we have found it straightforward to build symbolic simulators, even for complex processors [15]. The basic verification approach works well on small, intricately optimized code segments. However, as mentioned earlier, the basic approach does not scale well to longer segments of code.

## 3. CUTPOINTS FOR SOFTWARE

Analogously to formal equivalence verification of combinational circuits, we would like to use cutpoints to gain scalability. Obviously, the method needs to be conservative — it should not declare equivalence when the code segments are inequivalent — but we must also avoid introducing too many false inequivalences.

The most fundamental question is how to define cutpoints for software. In a combinational circuit, values flow along wires, from the inputs to the outputs, with gates performing computation along the way. Similarly, in software, values flow through the code in the program state (variables for high-level software; registers, internal buffers, memory, and other machine state for low-level software), with each instruction performing some computation on the values as they pass by. Thus, a cutpoint in software is some part of the program state at some point in a program, which is provably equal to some part of the program state at some point in the other program. In a combinational circuit, we can ignore the logic driving the cutpoint and insert a new primary input. Similarly, for software, we can discard the symbolic expression we computed for the value at the cutpoint and replace it with a new symbolic variable. If we can verify equivalence using the cutpoint, then the original circuits or programs were equivalent.

Control flow adds a wrinkle to the above definition. In combinational hardware, every wire always has a value for every possible input value. In software, some instructions may never be executed for some input values, and other instructions may be executed multiple times; the value of the program state at a given point in a program isn't always well-defined. The solution is to define software cutpoints *dynamically*, based on each dynamic execution path, rather than on the static code. We will use our existing verification approach to enumerate paths, and we will attempt to use cutpoints to make the verification of each path more efficient.

**Example:** For a simple example, consider a short loop that zeroes out a 1024-word block of memory. To further simplify the example, we will verify the equivalence of the loop to itself. Ignoring the loop induction variable, the dynamic instruction stream is just 1024 store instructions:

```
STW .D1 A0, *A4++
STW .D1 A0, *A4++
...
STW .D1 A0, *A4++
```

where register A0 has been initialized to 0, and register A4 indexes through the memory block using auto-increment. Using our basic verification approach, after *i* iterations, the symbolic expression for memory will be:

$$\text{write}(\ldots\text{write}(\text{write}(m_0, A4_0, 0), A4_0+4, 0)\ldots, A4_0+4(i-1), 0)$$

where $m_0$ and $A4_0$ are the initial values of memory and register A4. The expression is growing linearly with iteration count. Using the relational translation would also give linear-size expressions (linear number of constant-size clauses), plus a linear number of new variables. However, if we use cutpoints, we find that, amazingly enough, the machine states of the "two" programs (the two copies of itself) agree completely after each instruction. Hence, after each instruction, we could introduce a new cutpoint memory variable $m_i$ and a new cutpoint address value $A4_i$, and then at the next instruction have to prove only the equivalence of $\text{write}(m_i, A4_i, 0)$ in the two code segments.

Granted, the above example is contrived, but it serves to highlight the key design decisions in trying to apply cutpoints to software:

- *Where and how fine-grained to look for cutpoints?* In the above example, after every instruction, the entire machine state matched between the programs being compared, so we could cut the entire state between instructions. That would work for the example, but would produce false inequivalence for anything non-trivial. On the other extreme, we could try to match each register and memory location, or even each bit of each register and memory location for interpreted values, as a possible cutpoint. Finer-grained cutpoints allow more flexibility, improving the possibility of matches, but also exploding the set of possible matches to be considered. Also, we may not want to look for or insert cutpoints for some parts of the state: for example, in the simple example, if we make the loop induction variable a cutpoint, we lose the ability to prove termination.

- *How to find cutpoints?* In the simple example, the two programs were synchronized in lock-step, so we could execute a single instruction from each and find matching cutpoints. In general, however, computations will be reordered and instructions will be optimized away, so we need techniques to look for possible cutpoints.

- *Whether to do the cut?* This is the dynamic version of the first question. Once we find (and prove) a cutpoint, we may heuristically choose not to use it, perhaps to avoid false inequivalences.

- *How to do the cut?* By definition, we create a new symbolic variable to take the place of the expression computed for the cutpoint. But how aggressively do we propagate this new cutpoint variable? By the time the tool discovers a cutpoint, the symbolic simulator may have already computed other symbolic expressions, for other parts of the machine state, based on the symbolic expression being cut out. Should we track down these dependent expressions? How?

- *How to reduce false inequivalences?* The previous questions will affect the false negative rate, but this question is important enough to consider independently. Should we add constraints on newly introduced cutpoint variables? Are there other ways to reduce false inequivalences?

Any implementation of software cutpoints must answer the above questions. Ultimately, the real question is "Do the answers to the above questions allow verifying real code more efficiently and with an acceptable level of false inequivalences?"

## 3.1 Proof-of-Concept Implementation

To test the effectiveness of software cutpoints, we have implemented an instance of the idea. Our proof-of-concept implementation is just an initial exploration of the cutpoint idea, so we have strived for the simplest heuristics that seemed reasonable for each of the design questions raised above. The implementation is built on top of our existing tool, which uses the basic verification approach from Section 2 and targets assembly code for the Texas Instruments' C6x family of VLIW DSPs [15].

*Where and how fine-grained to look for cutpoints?* We check the symbolic expressions for only the memory. We do not look for cutpoints between registers or other parts of the machine state. Furthermore, we treat the entire memory as a single possible cutpoint. Our experience indicated that the symbolic expressions for memory are the primary source of blow-up in the basic verification approach, so we chose to focus on memory. Treating the memory as a single possible cutpoint greatly simplified the task of searching for cutpoints. Leaving registers out of the cutpoint analysis avoids the possibility of loop induction variables being cut.

*How to find cutpoints?* Checking only the entire memory makes this task much easier. The symbolic expression for memory changes only after a store instruction, so we keep a history buffer of the memory expressions from the last $k$ stores, for some depth $k$. The verification tool simulates one program through $k$ stores, then the other program through $k$ stores, then calls the decision procedure to find the most recent match (if any) of the $k^2$ possibilities. A larger value of $k$ handles greater reordering (thereby reducing false inequivalences), but can slow down the tool if there aren't any recent matches. We chose $k = 10$ arbitrarily, and it seemed work reasonably.

*Whether to do the cut?* When we find a cutpoint, we always do the cut. Again, this is motivated because memory expressions tend to blow up, and because we are matching only memory, so loop induction variables in registers won't be cut.

*How to do the cut?* We could conceivably match a memory expression $k$ stores earlier, which could be an unbounded number of (non-store) instructions in the past. It's hard to imagine trying to compute directly the effect of introducing the cut variable on all the values (and control flow!) that may have been computed subsequently. Furthermore, the C6x family have very deep pipelines, so searching through all the symbolic expressions in the pipeline and reasoning about any pipeline interactions is a daunting task. Instead, we simply leverage the fact that we already have a symbolic simulator for the processor. After each store instruction, we record in the history buffer the entire machine state, not just the expression for memory. When we find a cutpoint, we roll back the simulation to the cutpoint, and re-simulate any subsequent instructions.

*How to reduce false inequivalences?* This is the most complex question to answer. The simplest answer is to do nothing special. We initially implemented that choice and found that it worked successfully, and very efficiently, on a few examples (e.g., the embarrassing 47-line industrial example mentioned in the introduction, ported to the TI C6x), but produced too many false inequivalences in general (e.g., on the software pipelining example in Section 4). The fundamental problem is the inability to handle reordering of independent memory accesses. For example, consider verifying

```
LDW .D1 *A3++, A1
NOP 4  ; 4 cycle NOP for load to complete
STW .D1 A0, *A4++
```

versus

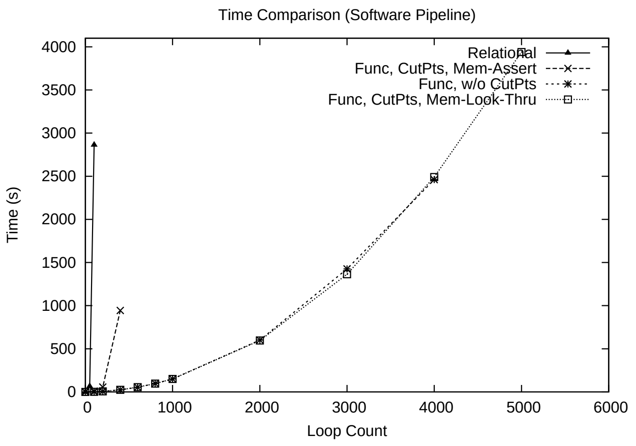
```
STW .D1 A0, *A4++
LDW .D1 *A3++, A1
NOP 4  ; 4 cycle NOP for load to complete
```

If we know that registers A3 and A4 point to different locations, then the two code segments are equivalent, and the basic verification approach would successfully verify that. Using our simple cutpoint approach, however, we would introduce a cutpoint after the STW instructions. The value of A1 at the end of the first code segment, therefore, will be based on the pre-cutpoint memory expression, e.g., $\text{read}(m_{\text{old}}, A3_0)$, whereas the value of A1 at the end of the second code segment will be based on the post-cutpoint memory expression, e.g., $\text{read}(m_{\text{new}}, A3_0)$, which aren't equivalent. The verification returns a false inequivalence.

We introduced two ways to eliminate these false inequivalences. We call the first "memory look-through". In this approach, the verification tool records the address written for every store instruction. For each load instruction, the tool tries to prove the independence of the address being read from the addresses that have been written. The read expression that is generated can read from any version of memory back to the most recent store that cannot be proven independent of the address being read. For our implementation, it turned out to be faster to first ask SVC to prove that the load address is independent of *all* stores, in a single decision procedure call. If this succeeds, the read expression reads from the initial memory. In the example above, if the address ranges for A3 and A4 provably never overlap, then the value loaded into A1 will always be $\text{read}(m_0, A3_i)$, where $m_0$ is the initial memory. This method reduces, but does not eliminate false inequivalences.

The other approach we tried completely eliminates false inequivalence (from the cutpoints — obviously, false inequivalence from other aspects of the verification approach, such as the uninterpreted functions, remain). When a new cutpoint variable is introduced, we add an assertion to the decision procedure that the new cutpoint variable is equal to one of the two (proven equivalent) expressions that it is replacing. This assertion guarantees that the cutpoint variable will always be properly constrained. We call this approach "memory assertions", and it is analogous to the combinational circuit equivalence technique in which, rather than introducing a new primary input at the cutpoint, we simply drive the cutpoints in both circuits from the same circuitry in one [6]. In the example above, we would assert that $m_{\text{new}} = \text{write}(m_{\text{old}}, A4_0, A0_0)$; this constraint preserves the relationship between $m_{\text{old}}$ and $m_{\text{new}}$, eliminating false inequivalences, but complicating the task of the decision procedure.

With plausible answers to all the design decisions, we can proceed to the real question: does it work on real code?



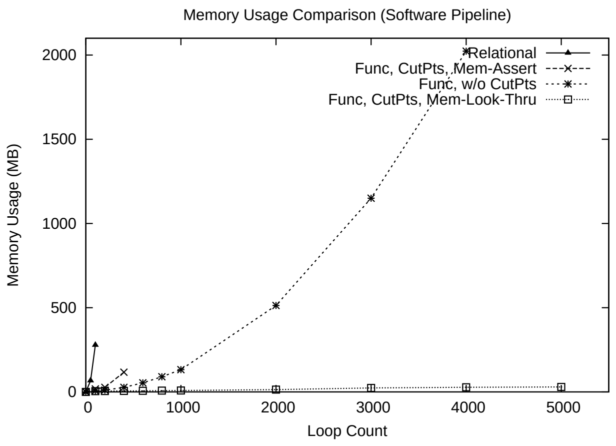


**Figure 3: Software Pipeline Results. The relational translation times out even for minuscule numbers of iterations, and the functional translation with cutpoints and memory assertions times out quickly, too. Our previous functional translation method without cutpoints is fastest, but the memory usage blows up. The new cutpoint method with memory look-through is almost as fast and uses very little memory.**

## 4. EXPERIMENTAL RESULTS

We have run experiments using several test cases. They are all small computational kernels, performing computations over arrays, where we can scale the difficulty of the example by adjusting the loop count. In each case, we verified the equivalence of unoptimized and highly optimized versions of the code. We compare the performance of four different methods: the basic verification approach, using the functional translation; the basic verification approach, using the relational translation; the functional translation with cutpoints, using memory look-through; and the functional translation with cutpoints, using memory assertions. For the functional translation methods, we use memory rewriting optimizations to try to reduce blow-up [12, 11], but we enable only the rewrites that actually help in the examples. Doing so helps the basic functional translation, but makes no difference for the cutpoint methods, so we have a fair baseline to compare the cutpoints against.

The first test case is taken from an article written by an expert on DSP code optimization, explaining how to optimize code for high-performance DSPs [22]. The example demonstrates software pipelining a short loop, targeting the C67x. Software pipelining is a powerful instruction scheduling technique that exposes additional parallelism in loops, thereby improving performance [20]. The basic idea of software pipelining 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 4 shows the unpipelined code, and Figure 5 gives the software pipelined code. The task is to verify the equivalence of the two.

We had previously been able to verify this example, using the basic verification approach (without cutpoints). Using cutpoints, we were still able to verify equivalence of the two versions, taken unmodified from the article. Because the cutpoint methods are conservative, successfully proving equivalence shows the cutpoints were sufficiently accurate and did not create false inequivalences. Figure 3 shows the performance trends as we scaled the number of loop iterations. The relational translation performs strikingly poorly: the run time blows up immediately, but surprisingly, the so does the memory usage. We do not have enough data to extrapolate the growth rate, but apparently the theoretically linear expression size growth of the relational translation is not competitive with the savings possible with the memory rewriting tricks of the functional translation. The method using cutpoints and memory assertions also performs disappointingly. Apparently, forcing the decision procedure to reason about all the cutpoint variables is causing blow-up, similar to the relational approach. Perhaps a newer decision procedure would help, as SVC is several years old. Nonetheless, cutpoints appear to provide a vast improvement in memory usage at a small cost in run time.

The preceding experiment used compiler-optimized code. For a harder experiment, we ran experiments on expert, hand-optimized code. Texas Instruments provides the TMS320C67x DSP Library (DSPLIB), a freely downloadable library of commonly-used DSP signal-processing routines hand-tuned by TI experts to achieve optimal execution speed [25]. Furthermore, each library function includes an equivalent C reference model, which we can compile using TI’s TMS320C6x ANSI C compiler to get an equivalent, non-optimized version. For our experiments, we selected three simple routines with numerous memory writes (the main source of expression size blow-up) from the library: block move (`DSPF_sp_blk_move`, 43 lines of code), convolution (`DSPF_sp_convol`, 101 lines of code), and FIR filtering (`DSPF_sp_fir_r2`, 270 lines of code).

On our initial attempt to verify these examples, the cutpoint methods failed immediately with false inequivalences. The problem is that the hand-tuned code’s subroutine linkage is different from the compiler-generated code: caller state is saved and restored slightly differently. This highlights the immaturity of our initial heuristics. Trivial, obviously unimportant changes were able to foil our cutpoint implementation. More sophisticated heuristics will obviously be needed in practice.

Fortunately, the subroutine linkage code was easy to remove, so we were able to try the verification (expert-hand-tuned vs. compiler-generated) on only the computational kernels of each routine. We manually replaced the linkage code with NOPs, added assertions to the decision procedure to initialize the two routines in the same way, and re-ran the verification tool. This time, we were able to verify equivalence fully automatically, demonstrating that our cutpoint heuristics were accurate enough for the computational kernels of our test cases. Figures 6, 7, and 8 show the performance

```
(... linkage and initialization omitted.
     B0 is the loop counter ...)
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++
(... subroutine return omitted ...)
```

**Figure 4: Unpipelined Assembly Code. The vertical bars indicate instructions executed in parallel. `LDW` (load word) has 4 delay slots, branches have 5 delay slots, and `MPYSP` (single-precision multiply) has 3 delay slots. The code point-wise multiplies two arrays, storing the result in a third array. The code takes 10 cycles per iteration. (Listing taken from [22].)**

```
(... linkage and initialization omitted.
     B0 is the loop counter ...)
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++  ;@@@@
(... subroutine return omitted ...)
```

**Figure 5: Software Pipelined Assembly Code. If the inputs are declared to be `const`, the compiler does software pipelining, improving performance to 2 cycles per iteration. But, does this do the same thing as Figure 4? (Listing taken from [22].)**

trends for these examples. On the block move example, the performance is very similar to the software pipelining example: the relational translation times out immediately; the cutpoints method with memory assertions times out quickly, too; the basic functional translation is fastest, but blows up in memory; and the cutpoint method with memory look-through is almost as fast and doesn't suffer memory blow up. On the convolution and FIR examples, though, the results are even more interesting: now, the cutpoint method with memory look-through is roughly twice as fast as the basic functional translation! The reason for this performance difference appears to be that in the hand-optimized convolution and FIR examples, the computation is much more highly reordered, resulting in a much harder equivalence expression for the decision procedure. The overhead of finding cutpoints is swamped by the savings of a simpler final verification problem. To test this hypothesis, we ran experiments using larger convolutions and more FIR filter coefficients and found that the performance advantage of the cutpoint method increased. Conversely, the block move example has no computation at all, simplifying the final verification problem, so the relative overhead of the cutpoints is higher.

In all the test cases, the cutpoint method with memory look-through vastly reduced memory usage. Run time ranged from a minor increase to a significant decrease. Accuracy was good enough to verify all of the computational kernels. Clearly, cutpoints can be very effective.

Tables 1, 2, 3, and 4 give detailed results comparing the two competitive methods: our original basic verification approach, without cutpoints, using the functional translation and memory rewriting, and our new cutpoint-based method, using the functional translation and memory look-through. All experiments were run on a 2.6Ghz Pentium 4 with 4GB of RAM. We have set the run time limit to 1 hour.

## 5. CONCLUSION AND FUTURE WORK

We have introduced the concept of cutpoints to the formal equivalence verification of low-level software. We have instantiated the theory in a proof-of-concept implementation and demonstrated large improvements in memory usage and comparable-or-better run time versus previous approaches. Our heuristics for reducing false inequivalences are immature, yet effective on the kernels of our examples.

Future work must try to further improve scalability, handle more general control-flow differences, and reduce false inequivalences. For scalability, our research has focused on improving efficiency and accuracy for verifying paths in programs. An obvious direction for future work is integration with complementary work on reducing the number of paths explored, e.g., by static analysis, analyzing well-structured loop bodies [24], or exploiting information from the compiler [21]. Such techniques will also be helpful for handling greater control-flow differences. Beyond reducing the number of paths explored, static analysis might provide other useful information. For example, a fast, approximate points-to analysis might be able to quickly guarantee that certain loads and stores are non-interfering, allowing faster confirmation of cutpoints.

For reducing false inequivalences, the main lines of future work will be finer-grained analysis for cutpoints, and heuristics for quickly finding candidate cutpoints. The instantiation of the theory we have presented here is vulnerable to false inequivalences if the programs make extraneous writes to memory or lay out data in memory differently. Looking for cutpoints for only specific addresses or ranges of memory would be more accurate. User guidance could be helpful, for example, by defining don't-care regions of memory. Finer-grained analysis, however, makes finding cutpoints much more expensive, necessitating better ways for finding good candidates. For combinational circuits, the standard technique is to simulate the two circuits on a sequence of random inputs; wires that always agree during simulation are candidate cutpoints. Unfortunately, with uninterpreted functions, such an approach is meaningless. However, an exciting direction for future work is to try our method on bit-accurate, fully interpreted (i.e., all operations are fully defined, rather than being abstracted) software models. Not only would such analysis be much more accurate, but there is also potential synergy between bit-accurate modeling and cutpoints: the efficiency improvement of cutpoints makes it conceivable to verify bit-accurate software models, and the bit-accuracy might greatly simplify reasoning about cutpoints.

## Acknowledgments

We would like to thank Shu Li, who was very helpful in generating the examples for our experiments. We thank the anonymous reviewers for encouraging us to try public, hand-tuned examples for our experiments and for suggesting several promising ideas for future work.

## 6. REFERENCES

[1] 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.

[2] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Computer-Aided Verification: 13th International Conference*, number 2102 in Lecture Notes in Computer Science, pages 260–264. Springer, 2001.

[3] 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://chicory.stanford.edu/SVC`.

[4] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for VLSI circuits. In *International Conference on Computer-Aided Design*, pages 456–459. IEEE, 1989.

[5] 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.

[6] D. Brand. Verification of large synthesized designs. In *International Conference on Computer-Aided Design*, pages 534–537. IEEE/ACM, 1993.

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

[8] R. E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, 38(2):299–328, April 1991.

[9] E. Clarke and D. Kroening. Behavior consistency of C and Verilog programs using bounded model checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, May 2003.

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as Lecture Notes in Computer Science Number 131.

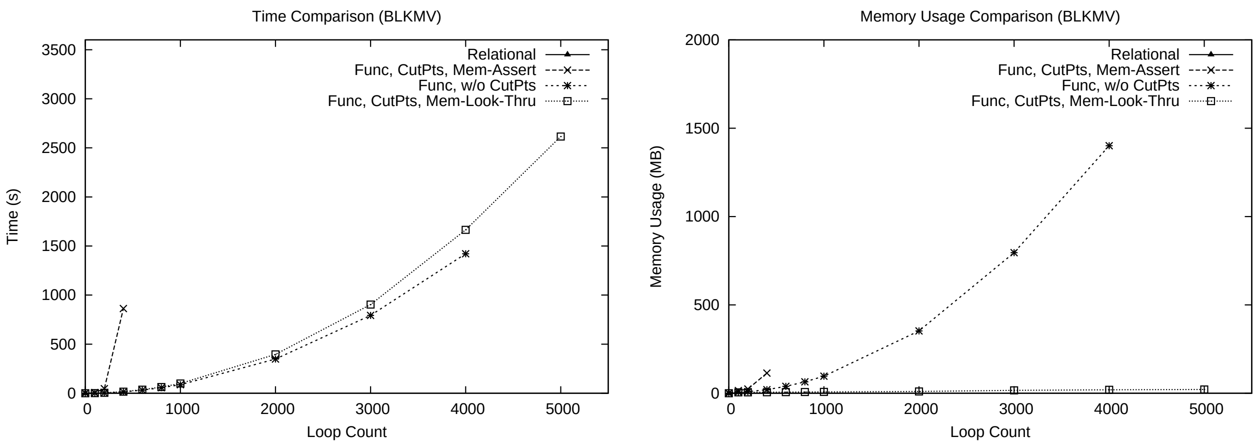


**Figure 6: Block Move Results. Performance trends are similar to Fig. 3, except that the time overhead of cutpoints is larger.**

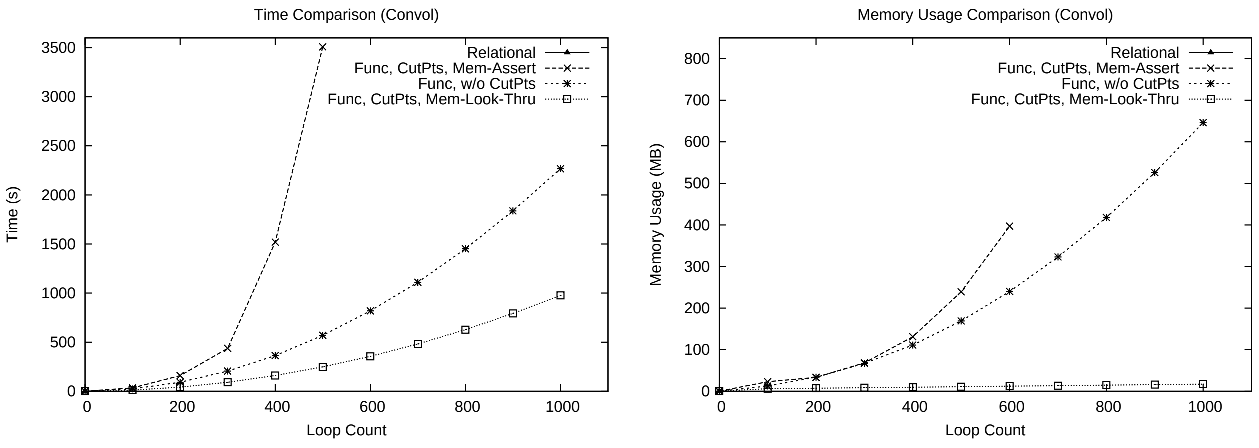


**Figure 7: Convolution Results. Here, not only does the cutpoint method with memory look-through have the lowest memory consumption, but it is fastest, too. The runs were with the number of impulse response samples set to 8.**

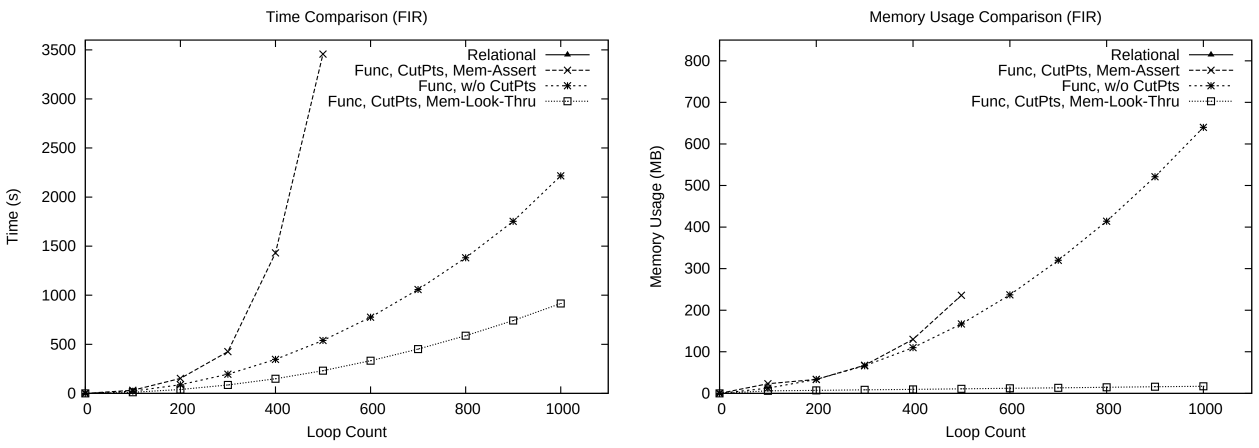


**Figure 8: FIR Filter Results. Performance trends are similar to Fig. 7: cutpoints with memory look-through was fastest and used vastly less memory. We set the number of filter coefficients to 8.**

| | Functional w/o Cutpoints | | Functional with Cutpoints | |
|---|---|---|---|---|
| Loop Count | Time(s) | Memory(MB) | Time(s) | Memory(MB) |
| 200 | 6.13 | 10.6 | 6.29 | 5.3 |
| 400 | 24.32 | 27.1 | 24.49 | 6.0 |
| 600 | 54.50 | 53.7 | 54.65 | 6.7 |
| 800 | 97.22 | 90.0 | 96.84 | 7.6 |
| 1000 | 149.96 | 132 | 150.13 | 8.7 |
| 2000 | 600.98 | 513 | 596.47 | 13.7 |
| 3000 | 1425.63 | 1150 | 1363.85 | 23.4 |
| 4000 | 2461.32 | 2023 | 2490.58 | 27.6 |
| 5000 | | mem out | 3939.21 | 29.3 |

**Table 1: Software Pipeline Detailed Results**

| | Functional w/o Cutpoints | | Functional with Cutpoints | |
|---|---|---|---|---|
| Loop Count | Time(s) | Memory(MB) | Time(s) | Memory(MB) |
| 200 | 3.59 | 8.7 | 4.15 | 5.0 |
| 400 | 14.05 | 20.3 | 15.99 | 5.6 |
| 600 | 31.53 | 39.1 | 35.90 | 6.1 |
| 800 | 56.00 | 64.6 | 63.65 | 6.7 |
| 1000 | 87.22 | 96.5 | 99.04 | 7.3 |
| 2000 | 349.06 | 353 | 395.70 | 10.2 |
| 3000 | 793.32 | 796 | 903.46 | 16.1 |
| 4000 | 1420.66 | 1401 | 1665.15 | 19.3 |
| 5000 | | mem out | 2615.88 | 21.0 |

**Table 2: Block Move Detailed Results**

| | Functional w/o Cutpoints | | Functional with Cutpoints | |
|---|---|---|---|---|
| Loop Count | Time(s) | Memory(MB) | Time(s) | Memory(MB) |
| 100 | 23.32 | 12.5 | 11.34 | 6.0 |
| 200 | 91.64 | 33.7 | 41.96 | 7.2 |
| 300 | 204.96 | 67.1 | 91.42 | 8.5 |
| 400 | 363.38 | 111 | 159.91 | 9.7 |
| 500 | 569.05 | 169 | 247.60 | 10.9 |
| 600 | 818.84 | 240 | 354.77 | 12.2 |
| 700 | 1109.54 | 323 | 481.52 | 13.4 |
| 800 | 1451.09 | 418 | 627.45 | 14.6 |
| 900 | 1836.78 | 526 | 792.95 | 15.9 |
| 1000 | 2267.42 | 646 | 976.55 | 17.1 |

**Table 3: Convolution Detailed Results**

| | Functional w/o Cutpoints | | Functional with Cutpoints | |
|---|---|---|---|---|
| Loop Count | Time(s) | Memory(MB) | Time(s) | Memory(MB) |
| 100 | 22.83 | 12.4 | 10.72 | 6.0 |
| 200 | 88.01 | 33.6 | 39.38 | 7.3 |
| 300 | 195.28 | 66.4 | 85.21 | 8.6 |
| 400 | 345.95 | 110 | 149.41 | 9.7 |
| 500 | 539.08 | 167 | 231.48 | 11.0 |
| 600 | 776.26 | 237 | 332.31 | 12.2 |
| 700 | 1058.00 | 320 | 451.26 | 13.4 |
| 800 | 1381.92 | 414 | 587.61 | 14.6 |
| 900 | 1752.31 | 521 | 742.46 | 15.9 |
| 1000 | 2216.26 | 640 | 915.77 | 17.1 |

**Table 4: FIR Filter Detailed Results**

[11] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming*. To appear.

[12] 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.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[14] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[15] X. Feng and A. J. Hu. Automatic formal verification for scheduled VLIW code. In *Joint Conference on Languages, Compilers, and Tools for Embedded Systems, and Software and Compilers for Embedded Systems*, pages 85–92. ACM SIGPLAN, 2002.

[16] 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.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Conference on Principles of Programming Languages*, pages 58–70. ACM SIGPLAN-SIGACT, 2002.

[18] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli. Formal verification of combinational circuits. In *International Conference on VLSI Design*, 1997.

[19] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *34th Design Automation Conference*, pages 263–268. ACM/IEEE, 1997.

[20] 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.

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

[22] 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`.

[23] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya. An equivalence checking methodology for hardware oriented C-based specifications. In *International High-Level Design, Validation, and Test Workshop*, pages 139–144. IEEE, 2002.

[24] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Automatic functional verification of memory oriented global source code transformations. In *International Workshop on High-Level Design, Validation, and Test*, pages 31–36. IEEE, 2003.

[25] Texas Instruments. *TMS320C67x DSP Library*. Version 1.00, February 17, 2003. Part Number SPRC121. `http://focus.ti.com/docs/toolsw/folders/print/sprc121.html`.

[26] O. Thiry and L. Claesen. A formal verification technique for embedded software. In *IEEE International Conference on Computer Design*, pages 352–357, New York, USA, 1996. IEEE Computer Society Press.

[27] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.