

Symbolic Verification of Dynamic Optimization in Microprocessors

Steven S. Lumetta, Sanjay J. Patel, Brian Fahs, Satarupa Bose
University of Illinois at Urbana-Champaign Department of ECE
Coordinated Science Laboratory, 1308 W. Main St, Urbana, IL 61801
{*steve,sjp,bfahs,sbose*}@crhc.uiuc.edu

Dynamic optimization exploits stabilities not identifiable at compile time to overcome the barriers to static optimization posed by modern code deployment techniques. The rePLay framework monitors control flow and value information to aid in the construction of long sequences of dynamic instructions (termed frames) that execute either to completion or not at all. Frames typically contain hundreds of instructions and offer many optimization opportunities. Simple, fast optimizations can yield large benefits, as a frame represents a single code path. The optimization process, however, must ensure that an optimized frame has precisely the same effects as the unoptimized frame on the architectural state. We apply formal symbolic verification to this problem by constructing an abstract syntax tree for each version and checking the equivalence of the two AST's. Interestingly, the need for a canonical form of AST for comparisons reduces the validation process in practice to one closer to N-version programming.

The rePLay Framework: In rePLay, the candidate regions of optimization are called *frames*. A frame is logically atomic: execution always starts at the first instruction in the frame and continues through to the last instruction. A frame also encapsulates a single flow of control: if any instruction within a frame executes, all instructions within the frame execute. A basic block is an example of a frame, albeit a small one. This atomicity property provides more flexibility for applying optimizations than if the frame were not atomic: instructions within a frame are not control dependent on one another and can be moved freely within the frame within the confines of data dependencies. Atomicity also reduces the complexity of the optimization algorithms. These algorithms need not consider multiple paths of execution as there are no side exits, side entrances, or divergent flows of control within a frame.

To support frame atomicity, rePLay employs a hardware recovery mechanism that reverts architectural state to the beginning of a frame when not all instructions in a frame should execute (*e.g.*, because an early exit condition is detected). This approach enables aggressive optimization of basic blocks that are likely to execute together without generation of the recovery code necessary to several other approaches to optimization. In order for frame optimization to have a significant impact on performance, frames must

contain many instructions and span multiple basic blocks. The longer the frame, the greater the opportunity for optimization and the greater the boost to fetch bandwidth.

Control instructions within a frame, such as branches between basic blocks in the frame, are changed to *assertion* instructions. An assertion instruction is similar to a conditional branch in that both test a condition, but the two differ in the actions taken after the condition is tested. The outcome of a conditional branch instruction determines the address of the next instruction. In contrast, an assertion has no effect on the address of subsequent instructions, and has no effect whatsoever if the condition is true. If the condition is false, the assertion fires, triggering a recovery action that discards all instructions in the frame and begins execution of the original, unoptimized instruction sequence.

Sample Optimization: Figure 1 illustrates optimization on a frame from the SPEC2000 *crafty* benchmark compiled with the Compaq Alpha C compiler DEC C V5.9 using optimization level 4. At this level of optimization, the compiler performs inlining, loop unrolling, and code replication to eliminate branches. The left column of the figure represents the frame constructed automatically through observation of the dynamic instruction stream and branch behavior. The frame contains eight basic blocks, separated by one procedure call and six assertions, and terminated by a branch instruction. Square brackets represent load operations, and the subscripts ("32") indicate sign-extension from the low 32 bits (Alpha is a 64-bit architecture).

Our frame optimizer is still under development, but the existing version is fairly effective at propagating constants and eliminating dead code. Compared with compiler optimizations, the task of optimizing a rePLay frame is relatively simple and fairly flexible. A frame consists of a single dynamic path through the code; in complexity, the frame is equivalent to a single basic block. Speculative execution of frames is supported in hardware using extended versions of mechanisms already in use in modern processors to support speculative prediction of branches. The middle column of Figure 1 presents the results of the automated optimization: five instructions are eliminated, and seven others are rewritten to use known constants or registers available earlier in the execution, thus reducing the dependency height of those instructions (the number of other instructions that must complete before an instruction can begin execution).

<pre> 00 r20 ← (r09 / 8) ; signed 01 r21 ← (r09 & 7) 02 r21 ← (r21 * 8)32 03 r13 ← 0 04 r22 ← [r29 - 26072] 05 r12 ← [r30 + 64] 06 r14 ← r11 07 r15 ← r20 & 7 08 r10 ← r21 * 8 + r22 09 prefetch [r30 & ~7] 10 r16 ← r15 11 r17 ← r13 12 r18 ← 8 13 r26 ← fixed address ; call instruction 14 r16 ← (r16)32 15 r02 ← 1 16 r03 ← (r16 + 1)32 17 r17 ← (r17)32 18 r03 ← r02 << r03 ; left shift 19 r18 ← (r18)32 20 r00 ← (r16 < 7) 21 r01 ← 0 22 assert r00 ≠ 0 ; asserts r16 < 7 23 r03 ← (r03)32 24 r00 ← r03 - 256 25 assert r00 < 0 26 r01 ← r01 r03 27 r00 ← r03 & r17 28 assert r00 = 0 29 r04 ← (r03 * 2)32 30 r00 ← r04 - 256 31 r03 ← (r04 * 2)32 32 assert r00 < 0 33 r01 ← r01 r04 34 r00 ← r04 & r17 35 assert r00 = 0 36 r00 ← r03 - 256 37 assert r00 < 0 38 r01 ← r01 r03 39 r00 ← r03 & r17 40 branch r00 ≠ 40 </pre>	<pre> 00 r20 ← (r09 / 8) 01 r21 ← (r09 & 7) 02 r21 ← (r21 * 8)32 03 r13 ← 0 04 r22 ← [r29 - 26072] 05 r12 ← [r30 + 64] 06 r14 ← r11 07 r15 ← r20 & 7 08 r10 ← r21 * 8 + r22 ; prefetch removed ; register motion propagated ; register motion propagated ; register load propagated 13 r26 ← fixed address 14 r16 ← (r15)32 15 r02 ← 1 16 r03 ← (r16 + 1)32 17 r17 ← 0 18 r03 ← r02 << r03 19 r18 ← 8 20 r00 ← (r16 < 7) ; register load propagated 21 r01 ← 0 22 assert r00 ≠ 0 23 r03 ← (r03)32 24 r00 ← r03 - 256 25 assert r00 < 0 26 r01 ← r03 27 r00 ← r03 & 0 28 assert r00 = 0 29 r04 ← (r03 * 2)32 30 r00 ← r04 - 256 31 r03 ← (r04 * 2)32 32 assert r00 < 0 33 r01 ← r01 r04 34 r00 ← r04 & 0 35 assert r00 = 0 36 r00 ← r03 - 256 37 assert r00 < 0 38 r01 ← r01 r03 39 r00 ← r03 & 0 40 branch r00 ≠ 40 </pre>	<pre> 00 r20 ← (r09 / 8) 01 r21 ← (r09 & 7) 02 r21 ← r21 * 8 ; never negative 03 r13 ← 0 04 r22 ← [r29 - 26072] 05 r12 ← [r30 + 64] 06 r14 ← r11 07 r15 ← r20 & 7 08 r10 ← r21 * 8 + r22 13 r26 ← fixed address 14 r16 ← r15 ; never negative 15 r02 ← 1 16 r03 ← r15 + 1 ; propagated, never neg. 17 r17 ← 0 18 r03 ← r02 << r03 19 r18 ← 8 20 r00 ← (r15 < 5) ; propagated, cond. from 37 22 assert r00 ≠ 0 ; instruction had no effect ; dead code ; redundant with 20' ; register move propagated ; dead code ; always true 29 r04 ← r03 * 2 ; never negative ; dead code 33 r01 ← r03 r04 ; propagated, reordered ; redundant with 20' 31 r03 ← r04 * 2 ; never negative, reordered ; dead code ; always true ; dead code ; redundant with 20' 38 r01 ← r01 r03 39 r00 ← 0 ; special case for & 40 branch ; unconditional </pre>
---	--	--

Figure 1. Optimization of a frame from *crafty*. The left column is unoptimized; the middle column was optimized automatically; the right column was hand-optimized. Primes indicate changes.

The right column of the figure illustrates the effects of more aggressive optimization, performed by hand for this example. The optimizer missed several straightforward opportunities. Elimination of the pair 27' and 28, for example, does not occur because the optimizer fails to recognize that AND'ing a register with zero produces zero. Another missed opportunity appears in 26': instruction 33 reads and overwrites r01, allowing us to eliminate one instruction by switching the order of 31 and 33 and replacing r01 with r03 in 33 to form 33'.

The other unexploited optimizations are more subtle, and require a small amount of reasoning about sets of possible values. Generation of r15 in 07, for example, produces a value in the range [0, 7]. Propagating this range through the frame enables elimination of twelve more instructions, primarily redundant assertions and the instructions used to generate their conditions. Several sign extensions can also be dropped, and more propagations performed once the sign extensions are removed.

Symbolic Verification: Proper functioning of a rePLAY-based microprocessor demands that the optimizer preserve the input-to-output behavior of each frame and retain any side effects. Our simulator currently verifies only that optimizations do not effect any simulated instances of the frame, in particular by executing both optimized and unoptimized versions and comparing the results. This approach verifies only register inputs and memory states that occur during the course of the execution, however, and is thus not a viable approach for an actual processor.

To improve the coverage of our verification, we have begun to develop a symbolic verifier using abstract syntax trees based on the two versions of the frame. Using the tree abstraction, we build up expressions for each live-out register in terms of the input registers. We also track assertions and memory operations (and order) in terms of the input registers. If the two frames match exactly in all three categories, the optimizations are valid.

The difficulty of such an approach lies in the comparison, which requires that the two results be placed in a canonical form. In practice, this canonicalization replicates the optimization process, and the resulting implementation is thus more akin to comparing two versions of the optimizer than to formal symbolic verification. In a straightforward construction of the output expression for r03 from the unoptimized frame in Figure 1, for example, we obtain $((1 < < (((r09/8) \& 7)_{32} + 1)_{32})_{32} * 2)_{32} * 2)_{32}$. From the right column, we obtain $((1 < < (((r09/8) \& 7) + 1)) * 2) * 2$. Only by again recognizing the possible results of AND'ing a register with the constant 7 can we reduce both cases to a canonical form, such as $(1 < < (((r09/8) \& 7) + 3))$. The final multiplications must be folded into or brought out of the shift completely to avoid the possibility of a false negative should the optimizer take advantage of this equivalence. Of course, the AST canonicalization could ignore optimizations not recognized by the optimizer, but sharing such information is likely to increase the chances of undetected errors through common mistakes in optimization/canonicalization.