New Decompilation Techniques for Binary-level Co-processor Generation

Greg Stitt, Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
{gstitt, vahid}@cs.ucr.edu
http://www.cs.ucr.edu/{~gstitt, ~vahid}
*Also with the Center for Embedded Computer Systems, UC Irvine

Abstract—**Existing ASIPs (application-specific instruction-set processors) and compiler-based co-processor synthesis approaches meet the increasing performance requirements of embedded applications while consuming less power than high-performance gigahertz microprocessors. However, existing approaches place restrictions on software languages and compilers. Binary-level co-processor generation has previously been proposed as a complementary approach to reduce impact on tool restrictions, supporting all languages and compilers, at the cost of some decrease in performance. In a binary-level approach, decompilation recovers much of the high-level information, like loops and arrays, needed for effective synthesis, and in many cases yields hardware similar to that of a compiler-based approach. However, previous binary-level approaches have not considered the effects of software compiler optimizations on the resulting hardware. In this paper, we introduce two new decompilation techniques, strength promotion and loop rerolling, and show that they are necessary to synthesize an efficient custom hardware co-processor from a binary in the presence of software compiler optimizations. In addition, unlike previous approaches, we show the robustness of binary-level co-processor generation by achieving order of magnitude speedups for binaries generated for three different instruction sets, MIPS, ARM, and MicroBlaze, using two different levels of compiler optimizations.**

I. INTRODUCTION

In order to meet the increasingly large performance and energy requirements of embedded applications, designers often partition critical software regions into custom hardware. The use of ASIPs (application-specific instruction processors) [4][8] is a common approach for partitioning, which tunes the instruction set of a microprocessor to a particular application. Although ASIPs achieve large performance benefits, the modification of a standard instruction set eliminates the possibility of using standard compilers, profilers, and simulators. Tensilica's Xtensa processor [5] is an ASIP approach that causes less impact of tool flow by automatically generating software development tools for the modified instruction set, while achieving good performance, including a speedup of 6.5 compared to a MIPS processor on EEMBC benchmarks. Recently, Tensilica introduced the XPRES compiler [16], which takes C/C++ code and automatically generates a customized microprocessor, thus eliminating the need for a designer to define new instructions manually. Stretch [14] has introduced a similar approach that combines the Xtensa processor with the Stretch Instruction Set Extension Fabric (ISEF), a specialized configurable logic fabric that designers can use to integrate new instructions into the Xtensa pipeline without having to fabricate an application-specific integrated circuit.

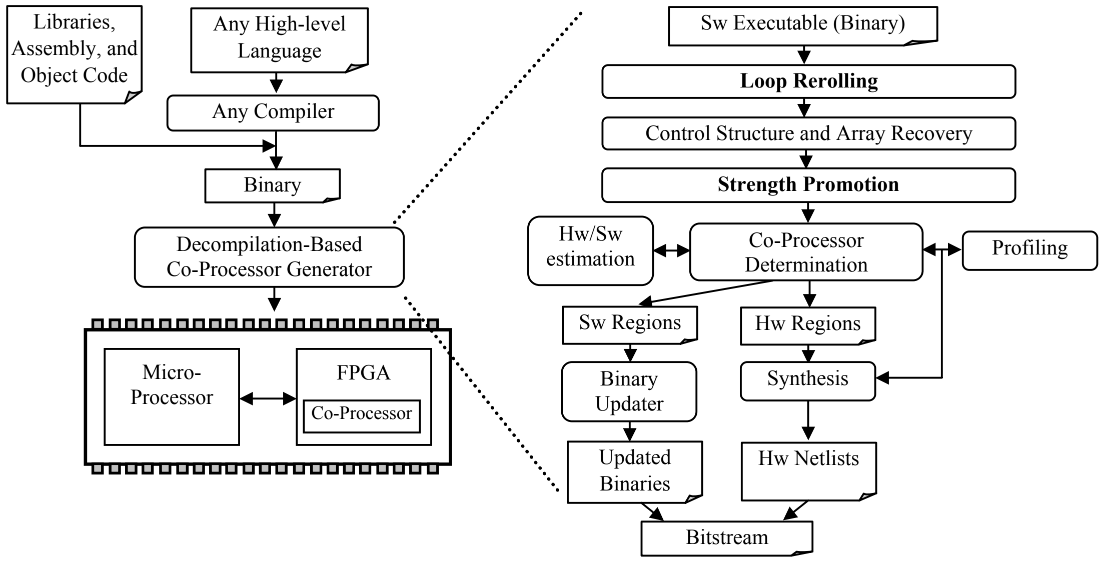
Other partitioning approaches have reduced the impact on tool flow by generating a co-processor, instead of modifying the instruction set. Critical Blue [3] generates a very-large instruction-word co-processor based on the software binary for an application. In this approach, the customized coprocessor executes performance critical regions of the application while the remaining regions of the application execute on a standard microprocessor.

Although these previous approaches have lessened the impact of tool flow integration, the approaches still impose some restrictions on tool flow by requiring a specific language or specific compiler. These restrictions may be undesirable to many software designers who have well-established compilers and developing environments. In order to further reduce tool flow restrictions at the cost of decreased performance, we proposed hw/sw partitioning of software binaries [13]. That binary-level approach partitioned a software binary onto a platform with a microprocessor and field-programmable gate array (FPGA), utilizing a limited form of decompilation to recover high-level control structures for critical kernels, and then synthesizing the kernels to hardware. By partitioning after software compilation and linking, binary partitioning supports all compilers and languages and possibly even multiple languages linked together at the object level. In addition, a binary-level approach supports the partitioning of library code and hand-optimized assembly, which may be important for designers using precompiled libraries – portions of code in embedded systems often come in the form of object code from third-party vendors. Mittal el al. [10] also utilized a similar approach for translating digital signal processor binaries onto an FPGA, useful for example to convert legacy code to an FPGA implementation. In [12], we showed that by recovering arrays and memory access patterns in addition to control structures, binary synthesis could often create hardware with almost identical performance compared to hardware generated from a compiler-based synthesis approach.

While binary-level partitioning may initially seem a sub-optimal approach, dynamic binary-level translation methods do represent a trend in modern processors, enabling binary portability for execution on processors with very different architectures. In

This research was supported in part by the National Science Foundation (CCR-0203829) and by the Semiconductor Research Corporation (2005-HJ-1331).

Figure 1: An approach for synthesizing a custom co-processor to speedup software binaries, utilizing the new decompilation techniques loop rerolling and strength promotion.



fact, recent work has even examined dynamic binary partitioning [11]. Nevertheless, we acknowledge that source level methods represent a superior technical solution in general. Binary level methods instead represent a means for expanding the benefits of partitioning-based co-processor design methods to a broader range of software designers.

Previous binary-level approaches have not addressed the problems associated with software compiler optimizations, which may obscure the representation of the original code within the binary, making decompilation more difficult. In this paper, we improve upon previous approaches by introducing two new decompilation techniques, *strength promotion* and *loop rerolling,* to eliminate the problems caused by strength reduction and loop unrolling. These new decompilation techniques result in speedups of up to 2.9 compared to previous binary synthesis approaches. We also show the robustness of binary-level co-processor generation by applying our techniques to two levels of compiler optimizations for three different instruction sets: MIPS, ARM, and MicroBlaze. Previous techniques have only considered a single instruction set.

II. BINARY-LEVEL CO-PROCESSOR GENERATION

In our approach to co-processor generation, shown in Figure 1, high-level software source code from any source language is first compiled using any software compiler to object code and linked with object files obtained from other means (assembly code or library files) to create a software binary. The decompilation-based co-processor generator then analyzes the software binary and creates a custom co-processor to speedup critical regions of the binary. The co-processor generator initially performs the new loop rerolling technique to convert unrolled loops within the binary into a non-unrolled representation. The co-processor generator then utilizes existing decompilation techniques [2][12] to perform control structure and array recovery. The co-processor generator then applies the new strength promotion technique to further recover high-level information needed to synthesize efficient hardware in the presence of aggressive compiler optimizations. Co-processor determination uses profiling and hw/sw estimation to select performance-critical regions of the application for implementation in the co-processor. We then use existing behavioral synthesis techniques to convert all critical regions into a custom hardware implementation. The binary updater modifies the software binary to add instructions for communication with the hardware. Finally, the co-processor generator combines the updated binary and netlists into a bitstream that configures the microprocessor and co-processor.

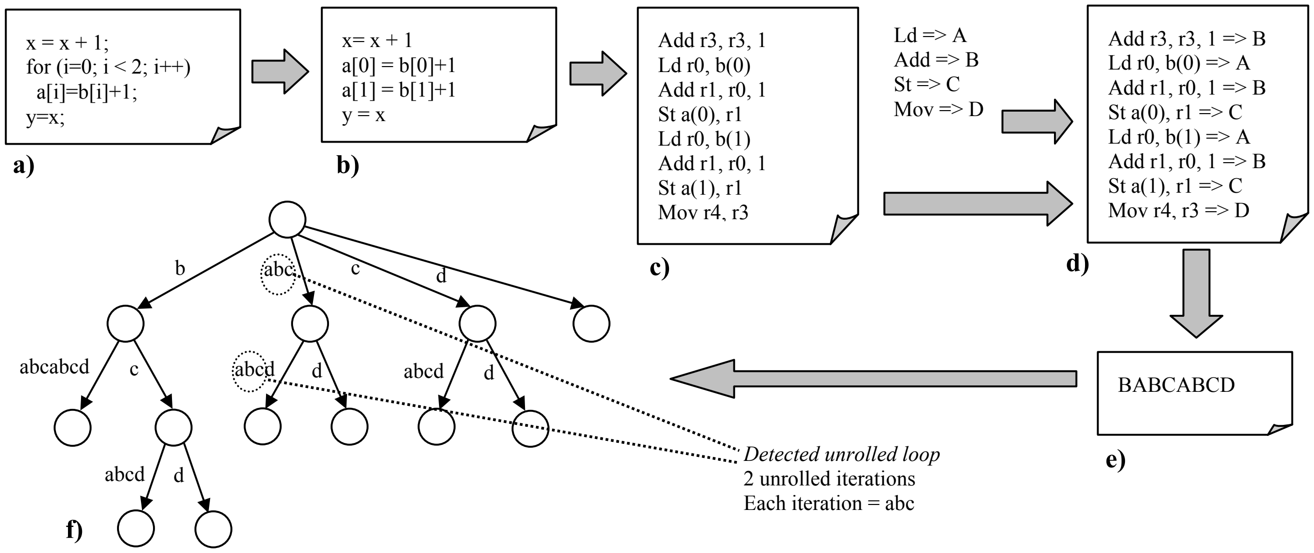
III. LOOP REROLLING

Loop unrolling is a common optimization that a software compiler uses to reduce branch penalties and increase instruction level parallelism. Loop unrolling can cause several problems for binary-level synthesis approaches. One potential problem is that loop unrolling can greatly reduce the usefulness of many profilers. Most profilers specifically determine the execution time of loops by monitoring branch instructions that have a small, negative target. If a loop is completely unrolled, then these branch instructions do not appear in the software binary. This problem may result in profiling information that does not include some of the most critical regions of an application.

Loop unrolling causes additional problems for binary-level synthesis by resulting in large control/data flow graphs. Most synthesis algorithms have super-linear complexity, which implies that the large graphs created from unrolling can greatly increase the execution time and memory requirements of binary-level synthesis. For traditional synthesis, increased time and memory are not serious problems, but for binary-level synthesis, this increase eliminates the possibility of performing binary-level synthesis dynamically [11]. Even in the case where a synthesis tool would unroll the loops, the amount of unrolling applied by the software compiler is unlikely to match the amount of unrolling that would be applied by the synthesis tool.

Also, previous binary synthesis techniques [12] have shown that recovering loop structure is important for utilizing custom memory structures, such as smart buffers [7], to support data reuse within the FPGA. Smart buffers utilize memory access patterns to determine an efficient structure for caching data needed by the FPGA, effectively increasing the memory bandwidth, which is commonly the main performance bottleneck [7]. When a loop is unrolled, the memory access patterns are no longer explicit. To be able to utilize smart buffers, loop rerolling recovers the memory access patterns by recreating the loop structure.

Figure 2: Loop rerolling using suffix trees.



Loop unrolling is also problematic for approaches that utilize specialized loop buffers such as a loop cache or filter cache [6], which store small loops to prevent fetching from a larger, higher-power cache or memory. Unrolled loops are likely too large to fit in these buffers, even if the software compiler only partially unrolls the loop.

To fix the problems caused by loop unrolling in binary-level synthesis, we perform *loop rerolling*. Whereas loop unrolling replicates a loop body to increase the number of sequential instructions within a software binary, loop rerolling detects these replicated loop bodies and recreates a loop that is similar to the loop in the original high-level source code. Note that loop rerolling will never decrease performance of synthesized hardware because the synthesis tool can always unroll the loops again. Although rerolling and then unrolling loops may seem redundant, this process is required to eliminate the problems caused by loop unrolling in a software binary.

Our loop rerolling technique consists of two steps. The first step of loop rerolling consists of analyzing the software binary and identifying all unrolled loops. The second step of loop rerolling consists of compacting the unrolled loops that were identified in the previous step into a more concise representation consisting of a single loop body that is equivalent to the behavior of all the unrolled iterations.

A. Identifying unrolled loops

Identifying an unrolled loop in a software binary is similar to the problem of finding consecutively repeated sequences of instructions in the software binary. Each repeated sequence represents a single iteration of the unrolled loop. Note that in the case of aggressive compiler optimizations, an unrolled loop may not contain the same instructions in all iterations. We discuss this potential problem in section III.C.

The process of identifying unrolled loops is illustrated in Figure 2. The high-level function we use for this example is shown in Figure 2 (a). This function consists of a simple for loop and two separate expressions. During software compilation, the compiler unrolls the loop into a representation similar to the code shown in Figure 2 (b) by applying constant propogation to remove the induction variable. Figure 2 (c) shows the code after the compiler converts the function into assembly instructions. The compiler converts each iteration of the loop into a series of three instructions, consisting of a load instruction, an add instruction, and a store instruction. Figure 2 (d) converts the entire assembly region into a string representation by mapping each assembly instruction to a character from an alphabet the size of the instruction set. Using the string representation simplifies the search for consecutively repeated sequences of instructions. Figure 2 (e) shows the resulting string representation of the function from Figure 2 (a).

After obtaining the string representation of the function, we identify all unrolled loops by determining all the repeated substrings that occur at consecutive locations in the string. The time and space complexity of exhaustively determining all substrings is O(n^2). To more efficiently determine repeated substrings, we use suffix trees.

A suffix tree is a Patricia tree that stores all possible suffixes of a string. Each path from the root node to a leaf node represents a unique suffix. Suffix trees allow for efficient pattern matching and can also be constructed in linear time [15]. Figure 2 (f) shows the corresponding suffix tree for the string in Figure 2 (e). Although substrings are shown on the edges, we also refer to nodes as representing the substring specified by the node's incoming edge.

Every internal node of a suffix tree represents a substring that is repeated at some point in the original string. We determine consecutively occurring substrings by traversing the suffix tree and checking adjacent nodes for the same substring. If two or more adjacent nodes represent the same repeated substring, then the concatenation of all these substrings represents an unrolled loop.

Once we have identified an unrolled loop, we determine the total number of iterations from the number of matching internal nodes. Figure 2 (f) highlights the two iterations of the loop from Figure 2 (a). We determine the number of instructions per iteration from the string length of each internal node. We determine the beginning and ending address boundaries of the loop by annotating each node of the suffix tree with instruction addresses from the original function.

The previous steps identify regions that are potentially an unrolled loop. Many of these regions may simply be instructions that happened to repeat with some regularity. To eliminate non-loop regions from consideration, we perform definition-use and use-definition analysis on each iteration of the potential unrolled loop. If the definition-use and use-definition chains are the same for each iteration, then the region is likely to be an unrolled loop. After definition-use analysis, we must also verify that there are no jump instructions into the middle of the potential loop. If such a jump exists, then the region is not an unrolled loop.

The example shown in Figure 2 assumed that the loop was completely unrolled. In many cases, compilers only partially unroll a loop. To handle partially unrolled loops, we apply our unrolled loop identification technique to every loop found during the control structure recovery step shown in Figure 1.

Although a compiler will typically only unroll the innermost loop of a nested loop structure, we can identify the nesting order of unrolled loops in a straightforward manner after we have determined each individual unrolled loop. To determine the nesting order, we examine the address boundaries of each loop. If a loop is completely contained within another loop, then the loop is an inner loop. The control structure recovery step in Figure 1 recovers the nesting order of partially unrolled loops.

B. *Compacting loop iterations*

After identifying an unrolled loop, we can compact the unrolled iterations into a smaller loop body that closely represents the loop in the original high-level description.

To compact unrolled loop iterations we first determine the relationship of constants in each unrolled iteration. In most cases of unrolling, a loop induction variable becomes a constant with value equal to the current unrolled iteration. After performing constant propogation, the compiler converts many of the original expressions into constants. In order to reroll a loop, we must determine the original expressions. We limit this determination to constants that have a linear relationship with a loop induction variable. We could detect other relationships, but the linear relationship allows us to detect the majority of loops. For nested loops, we must also detect row-major ordering calculations. We detect such calculations using techniques from [12]. Although these techniques do not guarantee the detection of row-major-ordering, we have successfully detected row-major-ordering calculation for all of our tested examples.

For completely unrolled loops, once we have determined the relationships of constants, we create an induction variable for the rerolled loop and replace each constant value with an expression that calculates the constant value using the induction variable. We can then create a compacted loop body that replaces all of the unrolled iterations.

For partially unrolled loops, we must determine the existing induction variables and any calculations that use the induction variables. We can then eliminate the expressions that are part of unrolled iterations. Finally, we adjust either the exit condition or the code that updates the induction variables. If the induction variable is changed for each iteration of the unrolled loop, then we adjust the exit condition. If the exit condition is modified for the unrolled loop, then we change the amount that the induction variable is updated by.

C. *Limitations*

A limitation of our loop rerolling technique is that in some situations, not all iterations of an unrolled loop will contain the same instructions. Different iterations of an unrolled loop commonly contain different instructions after the compiler applies constant propogation on the loop's induction variables. The first iteration of a loop will typically have different instructions because the compiler will be able to perform optimizations based on the induction variable having a value of zero. For example, if the code performs a multiplication using the induction variable, the compiler might replace a multiplication instruction with a move using an immediate value of zero. In our experiments, typically only the first two iterations of an unrolled loop use different instructions. In cases where the initial iterations are different, loop rerolling rerolls the remainder of the unrolled iterations.

IV. STRENGTH PROMOTION

Strength reduction is the process of converting an expensive operation into a series of less expensive operations. The most common use of strength reduction is the conversion of multiplications by a constant value into equivalent shifts. For example: a*8 could be converted to a << 3, thereby eliminating the expensive multiplication. Strength reduction is more beneficial when the operation being reduced has a constant input that is a power of two. If the constant is not a power of two then add operations are required. For example: a*11 would be converted to (a << 3) + (a << 1) + a.

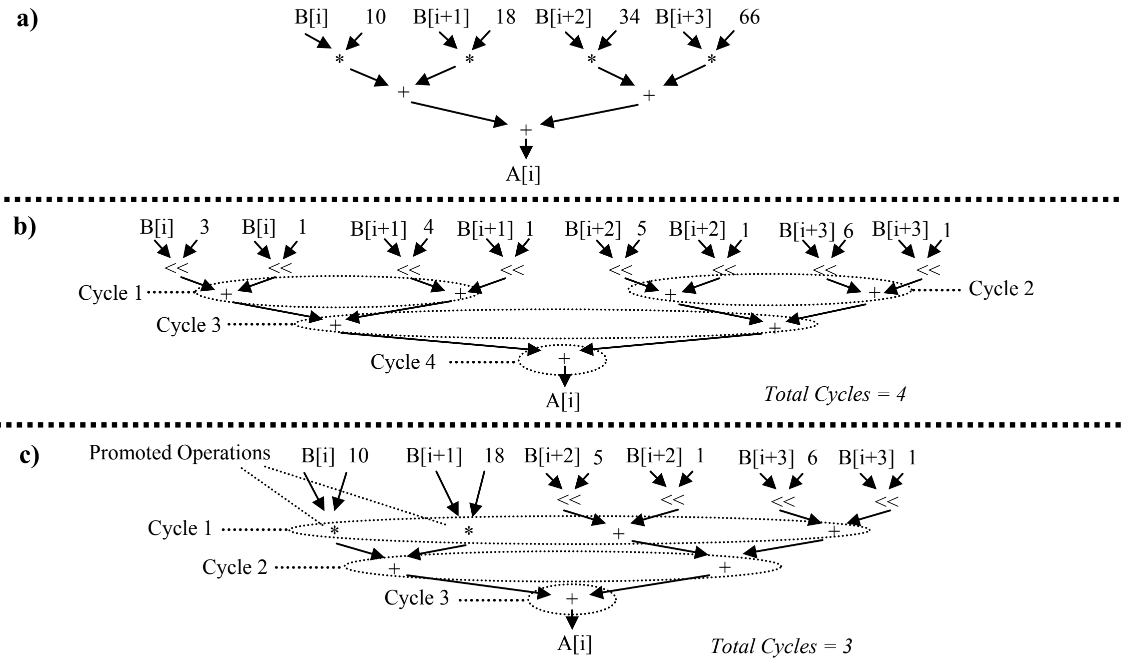
Synthesis tools need to analyze the available hardware resources before performing strength reduction. In most cases, a synthesis tool performs strength reduction because multipliers typically have long latencies, resulting in a slower clock frequency, and require large amounts of area. Therefore, a series of shift and add operations is likely to be faster than a multiply operation. However, there are situations where using shift/add operations results in decreased performance. For resource-constrained synthesis, the extra add operations created from strength reduction may exhaust all existing adder resources, causing the cycle latency from scheduling to be increased. In a situation where adders are exhausted and multipliers are available, the synthesis tool should not perform strength reduction, and instead should use the available multipliers to increase parallelism. Even if the synthesis being performed is not resource-constrained, strength reduction is not always beneficial. Strength reduction may greatly increase the area by creating unnecessary adders, resulting in hardware that will not fit within the targeted FPGA. In addition, the clock speed may suffer due to the difficulty of routing large FPGA designs. These issues with strength reduction imply that a synthesis tool must decide to perform strength reduction based on the available resources of the target platform and the current size of the design.

A drawback to previous binary-level synthesis approaches is that the software compiler forces the synthesis tool into a particular implementation, whereas giving the synthesis tool a choice would result in better hardware. This problem occurs because the software compiler makes optimization decisions based on the architecture of the target microprocessor. However, the microprocessor's architecture may be completely different from the target architecture of the synthesis tool.

To give the synthesis tool the choice of whether or not to perform strength reduction, a binary-level synthesis tool must first detect occurrences of strength reduction and then "promote" these operations back to the more expensive representation. We refer to this process as *strength promotion*. Note that strength promotion will never hurt the performance of an application because the synthesis tool can always convert the promoted operation back into the reduced form. Strength promotion may even promote shift/add operations that were never multiplications in the original code. This type of promotion may actually improve performance even further by giving the synthesis tool more options for synthesis.

Figure 3 illustrates the benefits of strength promotion for the binary-level synthesis of a 4-tap FIR (finite-impulse-response) filter, assuming resource constraints of 2 adders and 2 multipliers. Figure 3 (a) shows the unoptimized data flow graph for the FIR filter. Figure 3 (b) shows the data flow graph for the filter after

Figure 3: Several implementations of a FIR filter, assuming resource constraints of 2 adders and 2 multipliers. a) an unoptimized FIR filter, b) a FIR filter with strength-reduced multiplications, c) a FIR filter after "promoting" 2 strength-reduced multiplications.



software compilation, and the corresponding scheduling during synthesis. In this example, the software compiler has performed strength reduction on the multiplication operations, converting them to equivalent shift/add operations. Figure 3 (c) shows the data flow graph and corresponding schedule for the same data flow graph as shown in Figure 3 (b) after promoting two of the shift/add operations back to multiplications. Note that after performing strength promotion, we are able to achieve a scheduling that is one cycle faster. The data flow graph in Figure 3 (b) requires an extra cycle because the four add operations at the top level of the data flow graph require two separate cycles, due to the availability of only two adders. The strength-promoted graph in Figure 3 (c) is able to schedule these corresponding operations to the same cycle because of the availability of the two multipliers.

We perform strength promotion by traversing the data flow graph in search of subgraphs that match the pattern "input << const_amount" or "input*const_amount1 + input*const_amount2". These patterns are common implementations of strength-reduced multiplications. When these patterns are found, the corresponding subgraphs are replaced with semantically equivalent subgraphs for the expressions "input * $2^{\text{const\_amount}}$" or "(const_amount1 + const_amount2) * input", respectively. The algorithm continues in this manner until no more changes can be made to the data flow graph.

V. Experimental Results

A. Experimental setup

In our experiments, we use benchmarks from Powerstone [9], EEMBC, and our own benchmark suite. The Powerstone examples include: crc (cyclic redundancy check), des (data encryption standard), summin (handwriting recognition), and brev (bit reversal). The EEMBC examples include: BITMNP01 (bit manipulation), IDCTRN01 (inverse discrete cosine transform), PNTRCH01 (pointer chasing), AIFIRF01 (FIR filter), AIFFTR01 (fast fourier transform), rotate (90 degree image rotation), and IIRFLT01 (IIR filter). For our own benchmark suite, we chose examples not provided by other suites. Our benchmarks include a 5-tap FIR filter, a beamformer, and a viterbi decoder. All examples were compiled with gcc using –O1 optimizations. In section D, we also tested binaries generated using –O3 optimizations.

Our target architecture is a hypothetical platform consisting of a microprocessor and FPGA on a single chip. Although a wide variety of microprocessor/FPGA platforms are commercially available, such as the Triscend E5/A7, the Xilinx Virtex II Pro, and the Altera Excalibur, we use a hypothetical platform to more easily evaluate the benefits of our approach on different platforms. To evaluate a wide variety of different potential platforms, we include experiments in section D for three different microprocessors, including a MIPS, ARM, and MicroBlaze. We use the MicroBlaze to show that our approach can also be applied to platforms with only an FPGA. The FPGA used in all experiments is the Xilinx Virtex II Pro. The communication model used by the microprocessor and FPGA consists of shared memory and memory-mapped registers within the FPGA. The microprocessor can communicate with the FPGA by writing/reading the memory-mapped registers. The communication model allows the FPGA to read from memory either through the data cache or by using a DMA. The communication model maintains data coherency by requiring mutually exclusive execution for the microprocessor and FPGA. When writing to memory, the FPGA writes through the cache to prevent the microprocessor from reading stale values after resuming execution.

To generate hardware for each example, we developed binary-level tools that implement the approach shown in section II. The tools consist of approximately 30,000 lines of C code. The output of the tools is register transfer level VHDL code. We then generate a netlist for the FPGA using Xilinx ISE.

We determine the performance of the custom hardware co-processor, including the communication between the microprocessor and co-processor, using VHDL simulation. We determine hardware clock frequency from Xilinx ISE after placement and routing.

Table 1: Profiling results showing the percentage of execution time spent in the most frequent loop of each example both without and with loop rerolling.

	No Rerolling		Rerolling	
Example	***%Time***	***Speedup***	***%Time***	***Speedup***
AIFIRF01	4.9%	1.1	91.2%	11.4
AIFFTR01	10.9%	1.1	55.4%	2.2
rotate	1.2%	1.0	97.6%	41.7
bitmnp01	35.2%	1.5	53.7%	2.2
IIRFLT01	9.7%	1.1	83.6%	6.1
beamformer	5.2%	1.1	94.7%	18.9
viterbi	8.4%	1.1	89.2%	9.3
Average	10.8%	1.1	80.8%	13.1

B. *Loop rerolling*

Table 1 shows profiling results, both with and without rerolling, assuming that the software compiler unrolled the most frequent loop in the application. *%Time* is the percentage of execution time, determined by the profiler, that was spent in the most frequent loop. *Speedup* is the corresponding ideal speedup from implementing this loop in hardware, assuming the hardware executes in zero time. The *No Rerolling* columns represent profiling done without rerolling. The *Rerolling* columns represent profiling results after we performed rerolling.

Without loop rerolling, the profiler detects that the most frequent loop uses an average of 10.8% of execution time, resulting in a maximum possible speedup of 1.1. On average, the profiling results with loop rerolling show that the most frequent loop is actually responsible for 80.8% of execution time, corresponding to an ideal speedup of 13.1. The large difference in results is caused by the failure of the profiler to identify the most frequent loop after that loop is unrolled. Therefore, without rerolling, the profiler actually detects the second most frequent loop as being the most frequent loop. Of course, most profilers also provide percentages of execution time for functions. In this case, the execution time for an unrolled loop would be shown as part of a function. However, this function might contain other regions that are not appropriate for hardware implementation, which makes loop rerolling necessary to obtain efficient hardware. For all of these examples, the most frequent loop that we unrolled is likely to be unrolled by any compiler because each loop has a small loop body and small constant bounds.

Table 2 compares the synthesized hardware performance of the most frequent loop of each example, both with and without loop rerolling. $Cycles_{NoRerolling}$ is the number of cycles required to execute the synthesized hardware for the unrolled loop without rerolling. $Cycles_{Rerolling}$ is the number of cycles required to execute the synthesized hardware after we have performed loop rerolling on the unrolled loop. *Speedup* is the performance improvement of the hardware synthesized from a rerolled loop compared the performance of the hardware synthesized without rerolling. We compare performances in terms of clock cycles because for all examples, Xilinx ISE achieved the same clock frequency for both the unrolled and rerolled loop representations. Loop rerolling resulted in an average speedup of 1.6. For the beamformer example, loop rerolling resulted in a large speedup of 2.9. All of the performance improvements were the result of using smart buffers [7] to reuse data within the FPGA, which was made possible by using loop rerolling to recover the loop structure. For several examples, loop rerolling achieved no speedup. These examples exhibited no potential data reuse, which eliminated the possibility of using smart buffers. If we had utilized other loop-based synthesis optimizations, speedups from loop rerolling would likely be much greater. The area for both the unrolled and rerolled examples was almost identical. The reason for the similar area requirements is that the binary-level synthesis tool unrolled all the loops that we rerolled.

Table 2: Performance improvements when using loop rerolling.

Example	$\mathbf{Cycles_{NoRerolling}}$	$\mathbf{Cycles_{Rerolling}}$	**Speedup**
AIFIRF01	28326358	18884239	1.5
AIFFTR01	27720000	27720000	1.0
rotate	112000	112000	1.0
bitmnp01	23619014	23619014	1.0
IIRFLT01	2100000	1020000	2.1
beamformer	1251000	435000	2.9
viterbi	1460000	980000	1.5
Average:	12084053	10395750	1.6

C. *Strength promotion*

In our experiments for strength promotion, we synthesized a 5-tap FIR filter under a variety of resource constraints. By using different resource constraints, we were able to see the effects of strength promotion on a variety of different configurable architectures. We present results with constraints on the number of adders and multipliers, ranging from 5 to 20.

In addition to using multiple resource constraints, we also present results using different constant values as inputs to the multiplications. As a constant moves farther from a power of two, more adders are required for strength reduction, and performing strength promotion becomes more important. We present results using constants that require 0, 1, 2, or 3 adders to perform a strength-reduced multiplication.

Table 3 illustrates the benefits of performing strength promotion during resource-constrained binary-level synthesis. *Adders* specifies the constraint on the number of adders when performing synthesis. *Multipliers* specifies the constraint on the number of multipliers when performing synthesis. $Adders_{SR}$ specifies the amount of adders that are required to perform the strength-reduced multiplication for the current example. We vary this value by changing the constants used as inputs to the multipliers. *Clock* specifies the clock frequency in megahertz of the synthesized hardware, obtained after placement and routing. *Latency* is the number of clock cycles needed for the synthesized hardware to execute to completion. *Time* is the actual execution time of the synthesized hardware. *LUTs* is the number of lookup tables used within the FPGA. *Speedup* is the corresponding speedup achieved when using strength promotion.

Using strength promotion greatly reduced the latency of the hardware to an average of 16 cycles, compared to almost 31 cycles without strength promotion. The average clock frequency when using strength promotion was 192 MHz compared to 226 MHz when not performing strength promotion. The clock frequency is slower for strength promotion because the hardware generated using strength promotion utilizes multipliers, which typically have much longer latencies than adders. However, overall execution time of the hardware generated using strength promotion was approximately 1.5 times faster. In general, the hardware using strength promotion achieves a larger speedup as the number of adders required by strength reduction increases. When the constant values for multiplication were powers of two ($Adders_{SR} = 0$), both sets of hardware had identical performance because strength promotion converted the strength-reduced code back into multiplication operations, which was strength-reduced again by the synthesis tool after the synthesis tool determined that there would be no benefit from using multipliers. In terms of overall FPGA

Table 3: Comparison of resource-constrained binary-level synthesis of a 5-tap FIR filter without and with strength promotion, for a variety of different constant inputs.

			No Strength Promotion				Strength Promotion				
Adders	*Multipliers*	*Adders $_{SR}$*	*Clock*	*Latency*	*Time*	*LUTs*	*Clock*	*Latency*	*Time*	*LUTs*	*Speedup*
5	5	0	228	20	8.77	2912	228	20	8.77	2912	1.00
10	10	0	234	12	5.13	2930	234	12	5.13	2930	1.00
15	15	0	236	8	3.39	3376	236	8	3.39	3376	1.00
20	20	0	236	8	3.39	3376	236	8	3.39	3376	1.00
5	5	1	221	47	21.27	2840	176	29	16.48	2838	1.29
10	10	1	221	25	11.31	2872	178	17	9.55	2860	1.18
15	15	1	225	18	8.00	2898	180	13	7.22	2880	1.11
20	20	1	229	14	6.11	2924	181	11	6.08	2912	1.01
5	5	2	225	68	30.22	2856	176	29	16.48	2838	1.83
10	10	2	225	36	16.00	2890	178	17	9.55	2860	1.68
15	15	2	227	25	11.01	2924	180	13	7.22	2880	1.52
20	20	2	221	20	9.05	2958	181	11	6.08	2912	1.49
5	5	3	221	87	39.37	2840	176	29	16.48	2838	2.39
10	10	3	221	46	20.81	2874	178	17	9.55	2860	2.18
15	15	3	221	32	14.48	2908	180	13	7.22	2880	2.00
20	20	3	221	25	11.31	2936	181	11	6.08	2912	1.86
		Average:	226	31	13.73	2957	192	16	8.67	2942	**1.47**

Table 4: Comparison of latency-constrained binary-level synthesis of a 5-tap FIR filter without and with strength promotion, for a variety of different constant inputs.

	No Strength Promotion				Strength Promotion				
Adders$_{SR}$	*Clock*	*Latency*	*Time*	*LUTs*	*Clock*	*Latency*	*Time*	*LUTs*	*Speedup*
0	223	4	1.79	2940	223	4	1.79	2940	1.00
1	169	4	2.37	3432	169	4	2.37	3432	1.00
2	132	4	3.03	3556	153	4	2.61	3026	1.16
3	131	4	3.05	3936	153	4	2.61	3026	1.17
Average:	164	4	2.56	3466	175	4	2.35	3106	1.08

resources, the hardware generated using strength promotion required more area because this hardware utilized the dedicated multipliers. In terms of LUTs, both approaches required very similar area, which is to be expected due to the same adder constraints.

In addition to the resource-constrained experiments, we also performed latency-constrained experiments, shown in Table 4, where we attempted to achieve the smallest possible latency in terms of clock cycles. Strength promotion achieved a speedup of 1.08 and reduced the number of LUTs from 3466 to 3106. For the examples where strength reduction required zero or one adders, the results were identical because the synthesis tool reapplied strength reduction on the promoted operations. Strength promotion required 30 multipliers to achieve a latency of four cycles for the examples where more than one adder was necessary for strength reduction. Note that in contrast to the resource-constrained synthesis results, the hardware using strength promotion was able to achieve a faster clock. The reason for the faster clock is that because the number of adders wasn't constrained, the strength-reduced code used a larger amount of configurable logic blocks, making routing more difficult, resulting in a longer critical path.

D. Results for different optimization levels and different instruction sets

To show the robustness of our binary-level co-processor generation approach when using loop rerolling and strength promotion, we tested examples on three different instruction sets: MIPS, ARM, and MicroBlaze. We also used binaries generated with two different levels of compiler optimizations: –O1 and –O3. The –O1 flag informs gcc to apply a low-level of optimizations. The –O3 flag informs gcc to apply aggressive optimizations.

To evaluate software performance of each application we use an instruction set simulator for each microprocessor. For MIPS and ARM, we use versions of SimpleScalar [1] ported to these instructions sets. The MIPS results are actually based on the PISA instruction set, which is a superset of the MIPS instruction set. We determine MicroBlaze software performance using the simulator in Xilinx Platform Studio. For all experiments, the MIPS and ARM use a clock frequency of 200 MHz and the MicroBlaze uses a 150 MHz clock.

Table 5 shows execution times and speedups of our approach for a MIPS, ARM, and MicroBlaze using binaries generated using the –O1 and –O3 optimization levels. *Sw* is the execution time of the example when running in software on the specified microprocessor. *Hw/Sw* is the execution time of the example after applying our approach to generate a custom co-processor to speedup the software. All execution times are normalized to the execution time of software compiled with –O1 optimizations. *S* is the speedup from using the co-processor, compared to software compiled with –O1 optimizations. Our approach achieved significant speedups for all three microprocessors, averaging over an order of magnitude on each instruction set. In addition, note that in most cases when using aggressive optimizations (–O3), the resulting execution time is similar or better than when performed on binaries with less optimization. These results imply that when using loop rerolling and strength promotion, aggressive software compiler optimizations may not negatively impact binary synthesis. Several of the examples did experience a slight increase in execution time when using aggressive optimizations. However, the

Table 5: A comparison of execution times for software and software with custom co-processor for several microprocessors and several levels of optimization. *(Sw is the execution time of software execution. Hw/Sw is the execution time with a customized co-processor, and S is the speedup when using the co-processor compared to –O1 software. All execution times are normalized to the execution time of –O1 software.)*

	MIPS						ARM						MIcroBlaze					
	O1			O3			O1			O3			O1			O3		
Example	Sw	Hw/Sw	S	Sw	Hw/Sw	S	Sw	Hw/Sw	S	Sw	Hw/Sw	S	Sw	Hw/Sw	S	Sw	Hw/Sw	S
FIR Filter	1.000	0.089	11.2	0.923	0.070	14.2	1.000	0.085	11.8	0.999	0.084	11.9	1.000	0.040	25.3	0.549	0.015	68.4
Beamformer	1.000	0.074	13.5	0.853	0.071	14.0	1.000	0.149	6.7	1.018	0.172	5.8	1.000	0.031	32.3	0.647	0.032	31.4
Viterbi	1.000	0.136	7.4	0.891	0.152	6.6	1.000	0.131	7.6	0.957	0.126	7.9	1.000	0.060	16.7	0.765	0.017	59.0
Crc	1.000	0.030	33.8	0.967	0.019	53.6	1.000	0.020	49.5	1.105	0.007	134.8	1.000	0.012	80.3	0.995	0.011	88.6
Des	1.000	0.275	3.6	0.990	0.310	3.2	1.000	0.360	2.8	1.028	0.401	2.5	1.000	0.205	4.9	0.998	0.218	4.6
Summin	1.000	0.111	9.0	0.899	0.145	6.9	1.000	0.183	5.5	0.684	0.128	7.8	n/a	n/a	n/a	n/a	n/a	n/a
Brev	1.000	0.120	8.3	0.976	0.129	7.7	1.000	0.156	6.4	1.476	0.153	6.5	1.000	0.011	90.2	0.951	0.009	106.5
BITMNP01	1.000	0.114	8.8	0.985	0.113	8.8	1.000	0.188	5.3	0.988	0.186	5.4	1.000	0.112	8.9	0.999	0.115	8.7
IDCTRN01	1.000	0.323	3.1	0.975	0.323	3.1	1.000	0.230	4.4	1.005	0.230	4.3	1.000	0.258	3.9	0.885	0.150	6.7
PNTRCH01	1.000	0.196	5.1	0.945	0.196	5.1	1.000	0.325	3.1	0.963	0.313	3.2	n/a	n/a	n/a	n/a	n/a	n/a
Average:	1.000	0.147	***10.4***	0.940	0.153	***12.3***	1.000	0.183	***10.3***	1.022	0.180	***19.0***	1.000	0.091	***32.8***	0.849	0.071	***46.7***
Geo.Mean:	1.000	0.124	***8.4***	0.939	0.122	***8.7***	1.000	0.150	***7.0***	1.008	0.134	***8.3***	1.000	0.053	***19.0***	0.831	0.037	***27.4***

increase is typically insignificant because large speedups were still obtained compared to software execution. Note that for the ARM experiments, the performance of software was actually slower in many cases when using –O3 optimizations. However, despite the slower software the performance of the co-processor system generated from –O3 was similar to –O1. The co-processor achieved the largest speedups for the MicroBlaze, mainly due to the fact that the MicroBlaze is a slower microprocessor with a higher CPI and slower clock. The geometric mean is included in the table because of several outliers in terms of speedup. We omit results for summin and PNTRCH01 on the MicroBlaze because we could not get them to simulate correctly in software.

The average area required for the co-processors generated from the MIPS binaries was 26,109 gates for the –O1 binaries and 33,574 for the –O3 binaries. For the ARM, 34,553 gates were required for the –O1 binaries and 34,712 were required for the –O3 binaries. The MicroBlaze co-processor required similar area of 34,791 gates for the –O1 binaries and 38,458 gates for the –O3 binaries. The ARM and MicroBlaze co-processors required more area because of unneeded instruction side effects that our decompilation and synthesis tools were unable to optimize away. The average clock frequencies for the MIPS co-processors were 121 MHz and 112 MHz, for –O1 and –O3. For the ARM co-processors, the average clock frequency for both –O1 and –O3 was 111 MHz. For the MicroBlaze, both –O1 and –O3 co-processors had an average clock frequency of 121 MHz.

VI. Conclusions

Previous binary-level co-processor generation techniques have achieved results competitive with high-level compiler-based approaches. However, no previous technique has accounted for potential software compiler optimizations that may obscure the binary, making decompilation more difficult. In this paper, we improved previous binary-level co-processor generation techniques by introducing two new decompilation techniques: strength promotion and loop rerolling. These new techniques, used in conjunction with existing decompilation methods, significantly improved the quality of the synthesized co-processor when using a binary generated with aggressive software compiler optimizations. Also, whereas previous approaches have been limited to a particular microprocessor, we show the robustness of a binary-level co-processor generation by synthesizing efficient co-processors for a MIPS, ARM, and MicroBlaze microprocessor, using binaries generated with two different optimization levels.

References

[1] D. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342. June, 1997.

[2] C. Cifuentes, M. Van Emmerik, D.Ung, D. Simon, T. Waddington. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. Proceedings of the Workshop on Binary Translation, Newport Beach, USA, October 1999.

[3] CriticalBlue. http://www.criticalblue.com.

[4] J. Fisher. Customized Instruction-Sets for Embedded Processors. Design Automation Conference (DAC) 1999. pg. 253-257.

[5] R. Gonzalez, R.E. Xtensa: A Configurable and Extensible Processor. IEEE Micro, pp. 60-70, 2000.

[6] A. Gordon-Ross, S. Cotterell, F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. IEEE Computer Architecture Letters, Vol I, January 2002.

[7] Z. Guo, A. B. Buyukkurt and W. Najjar. Input Data Reuse In Compiling Window Operations Onto Reconfigurable Hardware, Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES 2004), Washington DC, June 2004.

[8] K. Kucukcakar. An ASIP Design Methodology for Embedded Systems. International Symposium on Hardware/Software Codesign, May 1999.

[9] A. Malik, B. Moyer., and D. Cermak. 2000. A low power unified cache architecture providing power and performance flexibility. In Proceedings of the International Symposium on Low Power Electronics and Design.

[10] G. Mittal, D. Zaretsky, X. Tang and P. Banerjee. Automatic Translation of Software Binaries onto FPGAs. Design Automation Conference (DAC) 2004. June 2004.

[11] G. Stitt, R. Lysecky, F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. IEEE/ACM 40th Design Automation Conference (DAC), June 2003.

[12] G. Stitt, Z. Guo, F. Vahid, W. Najjar. Techniques for Synthesizing Binaries to an Advanced Register/Memory Structure International Symposium on Field Programmable Gate Arrays (FPGA) 2005. pp. 118-124.

[13] G. Stitt and F. Vahid. Hardware/Software Partitioning of Software Binaries. IEEE/ACM International Conference on Computer Aided Design, November 2002.

[14] Stretch, Inc. http://www.stretchinc.com.

[15] E. Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3):249-260, September 1995.

[16] XPRES Compiler. http://www.tensilica.com/html/xpres.html.