Binary-Level Hardware/Software Partitioning of MediaBench, NetBench, and EEMBC Benchmarks

Greg Stitt and Frank Vahid*

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

Technical Report UCR-CSE-03-01
January 2003

ABSTRACT

Hardware/software partitioning can greatly reduce execution time and energy consumption of embedded systems. However, traditional source-level partitioning approaches have had limited success due in part to tool flow problems. Previous research introduced binary-level hardware/software partitioning as a solution to the tool flow problem, showing competitive speedups at the cost of almost double the hardware area. We incorporate powerful automated decompilation methods, previously developed by other researchers for binary translation tools, into a binary partitioning tool. Such incorporation eliminates the area overhead of previous approaches. Furthermore, we apply our tool to much larger examples than previous binary partitioning efforts, using examples from MediaBench, NetBench, and EEMBC benchmarks, and we show that speedups are still comparable with source-level partitioning even on these larger examples. Our results show that binary partitioning can result in average speedups of 3.0 and energy savings of 52% over software-only implementations, using an architecture similar to commercially available single-chip microprocessor/configurable-logic platforms.

Keywords

Hardware/software partitioning, decompilation, low power, speedup, FPGA, codesign, synthesis, platform, binary translation.

1. INTRODUCTION

Previous research has shown hardware/software partitioning to be very beneficial in embedded systems. Hardware/software partitioning divides an application's high-level description onto software running on a microprocessor and one or more hardware co-processors. Researchers have shown that such partitioning can yield large speedups and have thus sought to automate partitioning [3][10][11][12][15][17][21][23][32]. More recently, research has shown hardware/software partitioning to also greatly reduce energy consumption [18][19][28][29][34].

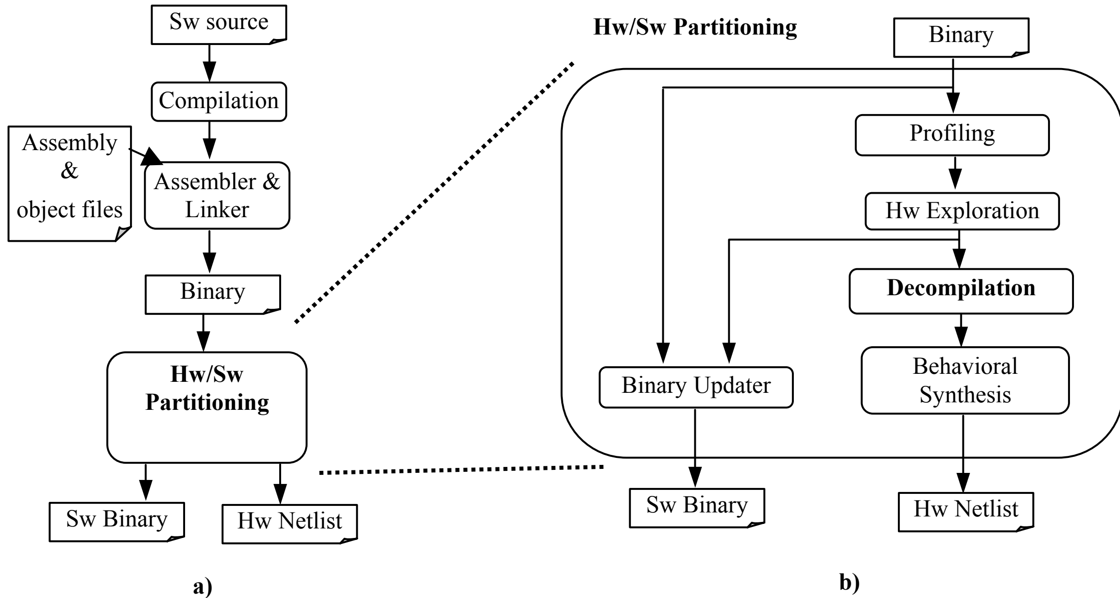
The recent appearance of single-chip platforms incoporating a microprocessor and field-programmable gate array (FPGA), such as the Triscend E5 and A7 [31], Xilinx Virtex II Pro [35], and Altera Excalibur [1], make hardware/software partitioning even more advantageous. These platforms can provide a time-to-market, cost, and size comparable to microprocessor-only designs, while achieving significantly better performance. These platforms also have more efficient hardware/software communication compared to multiple chip designs, often leading to reduced power and improved performance. While partitioning using off-chip configurable logic may increase energy compared to a software-only implementation due to expensive communication, utilizing on-chip configurable logic, which supports very efficient communication, can instead achieve large energy savings [28][29].

The traditional approach to hardware/software partitioning consists of partitioning code at the source-code level. This approach first compiles a high-level language to an intermediate format that is then explored for hardware candidates. Examples of source-level approaches include all the partitioning approaches referenced in the first paragraph, as well as Napa C [14], SA-C [4], Nimble [13], PRISM [2], DEFACTO [5], and Proceler [27]. At this time, source-level approaches have had limited commercial success, due in part to tool flow integration resistance. Most source-level approaches require replacing an existing compiler with a partitioning compiler, a replacement that is likely to be resisted by designers, who generally have trusted compilers and integrated development environments. The vast majority of compiler users want software only, so the strongest compiler vendors have little incentive to include partitioning in their tools.

A binary-level partitioning approach was proposed in [30] to solve the tool flow problems associated with a source-level approach. That approach has the advantage of being able to partition code compiled using any software compiler, and coming from any source language, including even assembly or object code, in contrast to source-level approaches, which are more restrictive. In addition, the binary-level approach has more accurate software performance and size estimation. The tradeoff is in less high-level information. Yet, the initial work in binary partitioning showed comparable speedups to a source-level approach [30]. However, these results were based on basic decompilation methods and used nearly double the area as a source level approach. Furthermore, those results were for small examples.

In our work, we describe our binary partitioning approach, whose novelty is the incorporation of fully automated, advanced decompilation techniques, which were designed by other researchers for binary translation purposes. We look at larger examples than previous work, drawn from the MediaBench [22], NetBench [26], and EEMBC [9] benchmarks. We show excellent speedups, with almost no area overhead compared to source-level partitioning.

Figure 1: Tool flow for binary-level hardware/software partitioning.



2. BINARY PARTITIONING OVERVIEW

Figure 1(a) shows our tool flow for binary-level hardware/software partitioning. The initial steps of the flow are exactly the same as a software design flow. We first compile high-level software source into object code, which we then link with other object code to create a software binary. Binary-level hardware/software partitioning then converts the binary into a custom hardware netlist for critical regions of the application, and a modified binary that utilizes the hardware.

Figure 1(b) illustrates binary-level hardware/software partitioning in more detail. The first step of binary partitioning consists of profiling the binary to identify regions of the original application that would benefit from hardware implementation. Hardware exploration uses the profiling results to determine the regions used for hardware implementation. Next, decompilation converts the regions destined for hardware into a high-level representation. Behavioral synthesis then converts the high-level representation into a hardware netlist. In addition to the previous steps, a binary updater modifies the binary so that the synthesized hardware will be used by the application.

The main goal of *profiling* is to identify regions of the application that contribute to a large percentage of execution time, and whose hardware implementation might therefore yield the greatest overall speedup. Loops are the most common regions that make up most of an application's execution time. In some cases, individual functions contribute to a large percentage of execution, but usually because the function is called from within a frequent loop. We have developed a profiler that works with an instruction set simulator to calculate the percentage of execution time for each loop and function in a program.

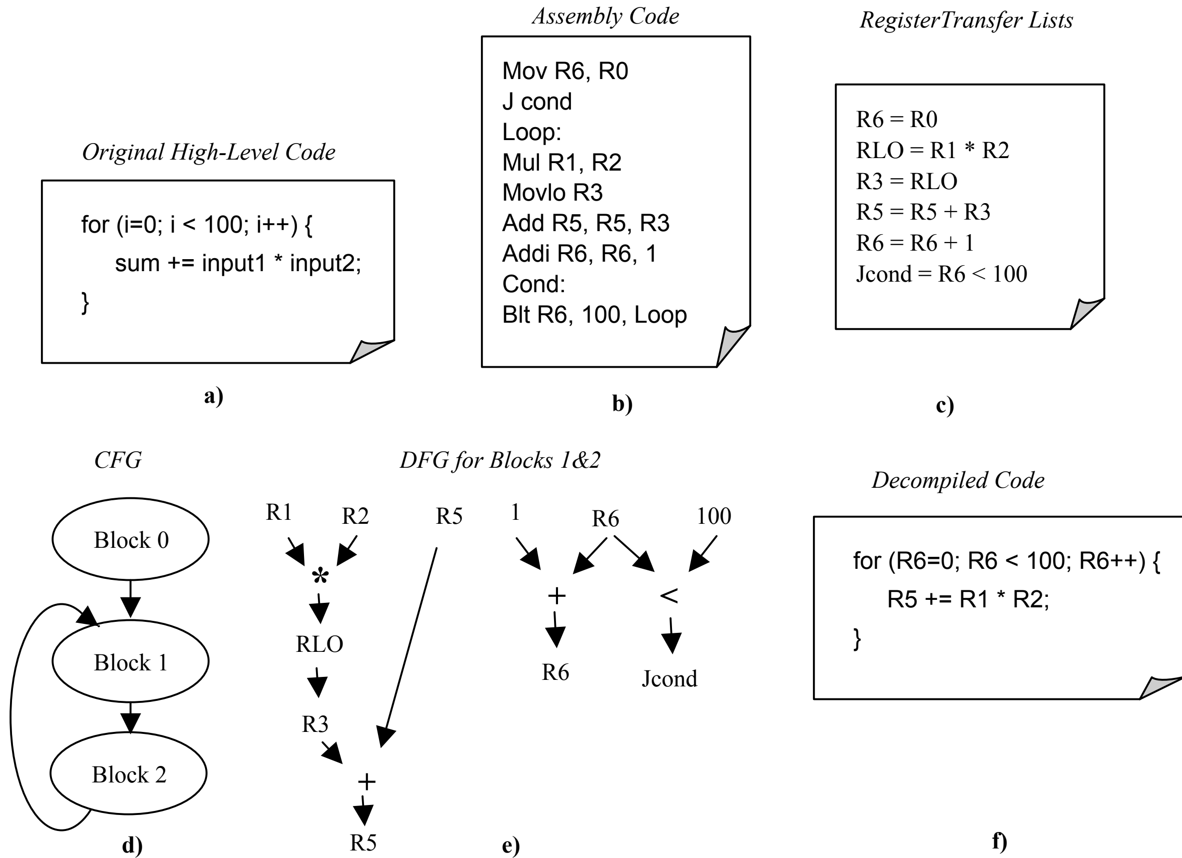
Hardware exploration uses the profiling results to select the particular binary regions that should be implemented in hardware. Hardware exploration is simplified because our studies of dozens of benchmarks from several different sources show that, for most programs, the several most frequent inner loops are often responsible for over 70% of program execution. Therefore, by limiting hardware exploration to the several most frequent inner loops, we can obtain large improvements without expensive and complicated exploration algorithms. Exploration can consider nearly all possible mappings of the frequent loops to hardware. We use an algorithm based on a heuristic for the 0-1 Knapsack problem.

The most important step in binary-level hardware/software partitioning is *decompilation*. After hardware candidates are selected, decompilation converts the candidates into a high-level representation. During decompilation, we apply optimizations to remove the overhead introduced by the assembly code and instruction set. Decompilation will be discussed in more detail in the following section.

Behavioral synthesis uses the decompiled, high-level representation to create hardware for the given region. Before behavioral synthesis is performed, there may still be inefficiencies in the hardware representation. Behavioral synthesis removes many of these inefficiencies using standard compiler optimizations like constant propagation and temporary variable elimination. Constant propagation is necessary to remove the overhead from unnecessary operations, such as an add instruction with an immediate value of zero. Compilers commonly use these kinds of instructions instead of other more appropriate instructions, such as a move instruction. Temporary variable elimination is needed to remove registers that hold temporary values that are part of larger expressions. If not removed, these temporary registers can get synthesized to hardware, causing an area overhead.

For our experiments, we have developed our own behavioral synthesis tool that is integrated with the decompilation tool. Integrating the decompilation and synthesis tools allowed us to

Figure 2: Decompilation process: a) original code, b) binary/assembly code, c) conversion to register transfer lists, d) control flow graph generation, e) data flow graph generation, f) decompiled code.



have more control over the hardware that is created – in particular, achieving the desired clock frequency. In addition, this integrated approach eliminates the need for some high-level control structure detection that is necessary for other synthesis tools. Given that we had already implemented the decompilation tools, adding the behavioral synthesis tasks was quite feasible, due to the convenience of being able to directly access the high-level representation from the decompilation tool. However, future implementations could certainly consider using commercial behavioral synthesis.

Our behavioral synthesis tool performs a minimum latency, resource-constrained version of list scheduling. The tool then reduces the amount of hardware resources using a greedy algorithm. The tool currently supports adders, subtractors, multipliers, shifters, and any Boolean logic function. The output of the tool is register-transfer (RT) level synthesizable VHDL. We then use a commercial synthesis tool to synthesize the RT-level VHDL to a netlist. Since our tool outputs RT-level VHDL, we don't have to handle control logic synthesis and instead simply specify states and state transitions in the VHDL.

The *binary updater* modifies the original binary to utilize the newly created hardware. The binary updater modifies the binary by replacing the original region of code with a jump to hardware initialization code. The initialization code first reads any necessary values from memory, sends the values to the hardware, and then writes to a port or memory-mapped register that enables the hardware. The initialization code also contains instructions to put the microprocessor into a low-power sleep mode while the hardware is executing. The processor is woken up by an interrupt originating from the hardware, which specifies the completion of hardware execution. We assume that the time to wake up the processor is small (approximately 10 cycles). If the processor takes much longer to resume execution, the binary updater can simply leave the processor in a busy waiting loop that waits for the end of hardware execution. Busy waiting would resume software execution much faster, at the expense of more power during hardware execution. The initialization code also includes instructions that write back hardware results and then jump to the end of the original software region.

3. DECOMPILATION

Figure 2 illustrates the process of decompilation. We have created an automated decompilation tool that performs the described techniques to decompile regions of binary code that are to be implemented in hardware. Most of the techniques we utilize are from the University of Queensland Binary Translator (UQBT) [7]. The source code and assembly code for the example used in this section are shown in Figure 2(a) and Figure 2(b), respectively.

The first step in decompilation is the conversion of assembly instructions into register transfer lists. A register transfer is an assignment statement for a particular register (or memory location). Decompilation techniques typically use semantic strings to represent the expression for each register transfer. A

Table 1: Benchmark overview.

Example	Benchmark Suite	Total Ins	Loop Ins	Loop Time%	Loop Size%	Ideal Speedup
g721	MediaBench	11,878	132	55%	1.1%	2.2
adpcm	MediaBench	9,302	153	99%	1.6%	100.0
pegwit	MediaBench	24,990	219	78%	0.9%	4.5
dh	NetBench	21,678	250	75%	1.2%	4.0
tl	NetBench	12,140	9	51%	0.1%	2.0
url	NetBench	13,526	17	80%	0.1%	5.0
AIFFTR01	EEMBC	16682	316	80%	1.9%	5.1
AIFIRF01	EEMBC	15832	150	93%	0.9%	13.3
BITMNP01	EEMBC	17400	2121	100%	12.2%	2272.7
IDCTRNO1	EEMBC	17,136	222	77%	1.3%	4.4
PNTRCH01	EEMBC	15,554	284	100%	1.8%	714.3
TTSPKR01	EEMBC	16,558	39	65%	0.2%	2.9
dither	EEMBC	15,342	48	100%	0.3%	249.7
	Average:	16,001	305	81%	1.8%	260.0
					Median:	5.0

register transfer list is a series of register transfers. Register transfers provide an instruction-set independent representation of the program. To convert a region of code to register transfer lists, decompilation converts each individual assembly instruction into a series of register transfers. For the example shown in Figure 2(c), decompilation converts each instruction into a single register transfer. For more complex instructions, several register transfers may be required. Decompilation handles instruction side effects by converting any side effects into explicit register transfers. For example, a pop instruction requires one register transfer to modify the stack pointer and another register transfer to read from memory.

After a region of code is converted to register transfers, the next step in decompilation is control flow graph (CFG) generation, as shown in Figure 2(d). The first step in CFG generation is basic block determination, for which there are several standard algorithms. Once the basic blocks have been determined, CFG generation connects the basic blocks to form the control flow graph. One of the drawbacks to a binary-level approach exists during CFG generation. If an indirect jump exists in a region, we may not be able to decompile that region of code. An indirect jump generally uses the value of a register as the target of the jump. The most common use of indirect jumps is for implementing function pointers and switch statements. There are several ways of dealing with indirect jumps [7], but in the worst case, we must exclude a region of code with an indirect jump from hardware implementation.

A data flow graph (DFG) is generated for the register transfers in each block. DFG generation is performed by parsing the semantic strings and creating a tree that defines each register transfer. DFG generation performs definition-use and use-definition analysis on the register transfer lists to connect the trees for each register transfer into a full data flow graph. Figure 2(e) shows the DFG generation process.

After a control flow graph has been determined, the next step in decompilation is the recovery of high-level control structures (loops, if statements, etc.). We use interval theory [8] to determine loops and nesting orders of loops. Two-way conditionals (if-then, if-then-else) can be recovered by a method described in [8]. For our decompilation tool, we do not explicitly determine two-way conditional statements. The decompilation tool detects that a two-way conditional exists, but does not determine the high-level type. During subsequent synthesis, we simply treat two-way conditionals as two possible state transitions. Figure 2(f) shows the decompiled representation of the program after we have recovered high-level control structures.

After a high-level representation of a region of code has been recovered, we apply optimizations to remove overhead introduced by the instruction set and assembly code. The behavioral synthesis tool handles most of the required optimizations. However, the decompilation tool performs several necessary optimizations. An example of a decompiler optimization occurs for an instruction set using compare instructions that have an implicit zero operand. In this situation, a comparison of two values first requires a subtract instruction followed by a compare with zero. In a high-level representation, the subtract operation is unnecessary and must be removed to create efficient hardware.

4. EXPERIMENTS

4.1 Setup

For our experiments, we assume a single-chip platform having a MIPS microprocessor with on-chip cache and an on-chip Xilinx FPGA configurable logic fabric. We utilized that particular microprocessor and FPGA fabric in part because we had good compilation, simulation, synthesis and estimation tools for each. While we have performed physical measurements of partitioning results using real platforms [28][29], we can examine examples far more quickly using simulation-based methods. We have previously found our simulation-based methods to yield similar power and performance as our physical measurements [29]. Our platform's system architecture is very similar to existing commercial architectures [1][31][35]. The MIPS runs at 200 MHz (which is reasonably fast for an embedded processor) with a CPI (cycles per instruction) of 1.5. We assume that the microprocessor and configurable logic share a data cache, which is similar to the GARP [16] system. Sharing the data cache allows us to implement more flexible memory accesses than would be possible with a DMA and avoids cache coherency problems that may occur.

We analyzed software performance by execution on SimpleScalar [6], a cycle accurate simulator for a MIPS-like processor. We multiplied the dynamic instruction count from SimpleScalar by our assumed CPI and clock cycle time to get software execution time. We used the gcc compiler with optimization set to level 1. We analyzed hardware performance by synthesizing our partitioner's RT-level output to a Xilinx FPGA netlist using Xilinx ISE, which tells us the critical path and hence the fastest FPGA clock frequency. Our partitioning tool outputs the latency, measured in clock cycles, of the longest path through a single iteration of each loop that is implemented in hardware. We then multiply this latency by the total iterations of the loop to get hardware cycles. We also add in the time required to enable and initialize the hardware, the time required to write back any results upon completion, and the time required by the microprocessor to resume execution.

We determined the application's software size from the static instruction count reported by SimpleScalar. Our profiling tools reported the software size of the loops that we implemented in hardware. We obtained hardware area from Xilinx ISE.

We computed software power by using the reported power of the 0.18 micron MIPS 4KP core [25], which would be 220 mW at 200 MHz. We determined FPGA power using the Xilinx Virtex Power Estimator [33], for the Xilinx xvc300e. To determine total power of the system, we used the following formula:

$$\%Sw*P_{Sw} + \%Hw*(P_{Hw} + .25* P_{Sw}) + P_I + P_Q,$$

where *%Sw* is the percentage of time spent executing in software, P_{Sw} is the power of the microprocessor, *%Hw* is the percentage of the time spent executing in configurable logic, P_{Hw} is the power of the configurable logic, P_I is the power of the system interconnect and memories, and P_Q is the quiescent power. We estimate that the power of interconnect and memory is 100 mW, which is based on physical measurements we made of the Triscend platforms. The formula for total power represents a weighted average of the system power during software execution and the system power during hardware execution. The formula assumes that the power-down sleep mode of the microprocessor consumes 25% of the power during normal execution (e.g., [20]). The formula also assumes that the only significant power of the configurable logic during software execution is quiescent power, as is typically the case. This formula has been compared to actual implementations on the Triscend A7 platform in [28], which showed estimated results and actual results to be similar.

Our decompilation and behavioral synthesis tools consist of approximately 7,000 lines of C code. The average run time for the partitioning tools was approximately 2 seconds, running on a 550 MHz Pentium III. The average size of the register transfer VHDL was between 1000 and 2000 lines – a reasonable size for synthesis using Xilinx tools.

Table 1 contains an overview of the benchmarks used in our experiments. *Total Ins* is the total number of instructions in the example. *Loop Ins* is the number of instructions in the loops that were implemented in hardware. *Loop Time%* is the percentage of execution time spent in the loops that were implemented in hardware. *Loop Size%* is the percentage of the total instructions used by the loops. *Ideal Speedup* is the largest possible speedup if the loops were implemented in zero time. Note that the loops that were implemented in hardware contributed to an average of 81% of execution time. This percentage corresponds to an average ideal speedup of 260. The median for the speedup is 5.0, which is significantly lower than the average due to the existence of several outliers. In addition to large potential speedups, these loops consisted of only 1.8% of the total program size, implying that little area may be required for hardware implementation. For the examples that have a reported *Loop Time%* of 100%, the percentage is actually slightly less and rounds to 100%.

4.2 Partitioning results for MediaBench, NetBench, and EEMBC benchmarks

To show that binary-level hardware/software partitioning is beneficial, we partitioned the binaries of many examples from the popular benchmark suites MediaBench, NetBench, and EEMBC. Table 2 shows the results for the partitioned examples. *Time* is the total time of the example when running in software. *Loop Time* is the time required by the implemented loops when running in software. *Hw Loop Time* is the time required by the implemented loops when running in hardware. *Hw Clock* is the maximum clock frequency used by the configurable logic, reported in megahertz. *Hw/Sw Time* is the execution time of the partitioned example. *Ideal Speedup* is the maximum possible speedup. *Speedup* is the performance increase of the partitioned example. *Area* is the required amount of configurable logic, given in equivalent ASIC gates. *Energy Savings* is the percentage energy savings of the partitioned example. All times values are reported in seconds.

Binary-level partitioning achieved an average speedup of 3.0 for the tested examples. For several of the examples, the speedup was far below the ideal speedup. The main reason for this difference is that our partitioning tool limits optimizations to individual basic blocks. In several cases, this limitation causes a large decrease in speedup. We are currently working to improve the optimizations in our partitioning tools. Despite the large difference between ideal speedup and actual speedup for several examples, we were still able to achieve significant speedups.

The average area required was 14,772 logic gates – a reasonable amount for current FPGAs. AIFFTR01 required an excessive 54,852 gates, due to the use of several multipliers. AIFFTR01 could be improved by modifying the amount of available multipliers during scheduling.

Binary partitioning achieved a significant energy savings of 52%. The main reason for this large savings was the performance improvement. Power consumption was approximately 5% lower after partitioning, implying that use of the configurable logic was not a power overhead.

Binary partitioning was unable to create hardware for two EEMBC examples. CACHEB01 contained function pointers in the frequent loop that was selected for hardware implementation. The compiler converted function-pointer calls into indirect jumps. As previously stated, we are unable to handle indirect jumps during decompilation. CANRDR01 resulted in the same problem, due to the use of a switch statement in the frequent loop.

4.3 Source vs. binary partitioning comparison

In the previous section, we showed that binary-level partitioning can achieve large performance and energy improvements. We now compare results for source-level and binary-level partitioning approaches to show that binary-level partitioning can in fact achieve results similar to source-level partitioning. For the source-level experiments, we manually translated the C code for

Table 2: Binary partitioning results for MediaBench, NetBench, and EEMBC benchmarks.

	SW		Binary-Level Partitioning						
Ex	Time	Loop Time	Hw Loop Time	Hw Clock	Hw/Sw Time	Ideal Speedup	Speedup	Area	Energy Savings
g721	4.191	2.288	0.53	115	2.43	2.2	1.7	4,779	44%
adpcm	0.164	0.164	0.05	73	0.05	100.0	3.0	15,240	71%
pegwit	0.214	0.166	0.03	115	0.07	4.5	2.9	15,386	64%
dh	8.965	6.745	2.31	63	4.53	4.0	2.0	16,607	53%
tl	0.287	0.146	0.10	124	0.24	2.0	1.2	2,020	23%
url	0.137	0.109	0.01	138	0.03	5.0	4.0	1,234	76%
AIFFTR01	0.623	0.501	0.33	85	0.45	5.1	1.4	54,852	15%
AIFIRF01	0.403	0.37	0.22	107	0.29	13.3	1.4	13,911	34%
BITMNP01	1.745	1.744	0.20	119	0.20	2272.7	8.8	15,535	88%
IDCNTRN01	0.75	0.579	0.07	46	0.24	4.4	3.1	20,439	70%
PNTRCH01	0.577	0.576	0.11	134	0.11	714.3	5.1	8,595	82%
TTSPKR01	0.352	0.23	0.15	136	0.27	2.9	1.3	8,249	25%
dither	1.744	1.737	0.62	140	0.62	249.7	2.8	15,194	35%
Average:			0.36	107.3	0.74	260.0	**3.0**	**14,772**	**52%**

Table 3: Comparison of source-level and binary-level hardware/software partitioning.

	SW		Source-Level Partitioning						Binary-Level Partitioning					
Ex	Time	Loop Time	Hw Loop Time	Hw Clock	Hw/Sw Time	S	A	E%	Hw Loop Time	Hw Clock	Hw/Sw Time	S	A	E%
g721	4.191	2.288	0.45	98	2.35	1.8	8,394	45%	0.53	115	2.43	1.7	4,779	44%
adpcm	0.164	0.164	0.03	40	0.03	5.5	14,132	87%	0.05	73	0.05	3.0	15,240	71%
pegwit	0.214	0.166	0.05	48	0.09	2.3	18,150	61%	0.03	115	0.07	2.9	15,386	64%
dh	8.965	6.745	2.18	40	4.40	2.0	21,383	57%	2.31	63	4.53	2.0	16,607	53%
tl	0.287	0.146	0.08	61	0.22	1.3	5,478	31%	0.10	124	0.24	1.2	2,020	23%
url	0.137	0.109	0.01	156	0.03	4.1	2,929	76%	0.01	138	0.03	4.0	1,234	76%
AIFIRF01	0.403	0.371	0.25	112	0.28	1.4	22,361	31%	0.22	107	0.29	1.4	13,911	34%
IDCTRN01	0.75	0.579	0.07	49	0.24	3.2	15,136	71%	0.07	46	0.24	3.1	20,439	70%
TTSPKR01	0.352	0.23	0.14	143	0.26	1.4	9,171	34%	0.15	136	0.27	1.3	8,249	25%
dither	1.744	1.737	0.63	127	0.63	2.8	16,093	38%	0.62	140	0.62	2.8	15,194	35%
Average:			0.39	87	0.85	**2.6**	**13,323**	**53%**	0.41	106	0.88	**2.3**	**11,306**	**50%**

each example into VHDL. We manually optimized the VHDL to create the best performing hardware that we could achieve. We are currently in the process of finishing the source-level results for EEMBC.

We restate that binary-level partitioning is not intended to achieve superior results compared to source-level partitioning. The advantages of binary-level partitioning are a more transparent integration into standard tool flows, without sacrificing the quality of results.

Table 3 compares the results of source-level partitioning and binary-level partitioning. *Time* is the total time of the example when running in software. *Loop Time* is the time required by the implemented loops when running completely in software. *Hw Loop Time* is the time required by the implemented loops when running in hardware. *Hw Clock* is the maximum clock frequency used for the configurable logic, given in megahertz. *Hw/Sw Time* is the execution time of the partitioned example. *S* is the speedup of the partitioned example. *A* is the area required for the configurable logic, given in equivalent ASIC gates. *E%* is the percentage of energy savings. All times values are reported in seconds.

Source-level partitioning achieved an average speedup of 2.6. Binary-level partitioning achieved an average speedup of 2.3. *Adpcm* performed much better at the source level, achieving a speedup of 5.5 that was almost twice the binary-level speedup. The main reason for this difference is that *adpcm* has many small basic blocks that can be optimized away. As previously stated, our partitioning tool limits optimizations to individual blocks and is unable to schedule operations from different blocks to the same clock cycle. Therefore, the source-level approach currently can exploit more parallelism than our binary partitioning tools, causing decreased performance. We are currently implementing optimizations in our partitioning tools to fix the performance difference. Another interesting result is that the binary-level

version of *pegwit* actually outperforms the source-level version. The reason for this performance difference is that the manually written VHDL was unable to match the clock frequency for the binary-level results. We could of course continue to revise the manually written VHDL to reach the desired clock frequency, but our initial efforts were unable to achieve the frequency of the binary example. At the same clock frequency, the source-level results for *pegwit* would have been better than the binary-level.

The source-level hardware required an average area of 13,323 gates. The binary-level results required slightly less area, averaging 11,306 gates. The main reason for the area differences is that the source-level examples outperformed the binary-level examples, doing so by using more resources. The average clock frequency of 87 MHz for source-level partitioning was lower than the 106 MHz for binary-level partitioning because of these area differences. The source-level examples performed more computation in each cycle, which required a slower clock. Our area results improved greatly compared to previous binary partitioning work [30], where the area of the binary-level examples was more than twice that of source-level examples.

Energy savings for the source-level examples was 53%. The binary-level energy savings was 50%. The main reason for the difference is that the source-level examples had smaller execution times and lower clock frequencies, leading to reduced power. The source-level results reduced overall power by 11%. The binary-level results reduced overall power by 5%. Previous efforts on partitioning for single-chip platforms [28][29] experienced a power increase when using the configurable logic. Our experiments experienced a power reduction because we are using a microprocessor that consumes more power (because the microprocessor is much faster) than in previous efforts. Therefore, shutting down the microprocessor and executing in the configurable logic was actually able to reduce power.

5. FUTURE ISSUES

Unlike source-level approaches, binary-level partitioning is dependent on the instruction set of the microprocessor and any compiler optimizations that are performed while creating the binary. The dependence on instruction sets and optimizations are possible drawbacks to hardware/software partitioning at the binary level. We plan to investigate these dependences in future work.

Loop unrolling is the most likely optimization to affect binary partitioning. The largest problem with unrolled loops is detecting the existence of an unrolled loop during profiling. Previous work [24] in loop re-rolling can be used to solve this problem. Loop re-rolling is a computationally expensive technique. However, we are only concerned with detecting unrolled loops, which can be done with pattern matching. Synthesizing an unrolled loop can be handled during behavioral synthesis using an appropriate scheduling algorithm. By limiting the number of available resources to the amount used by a single iteration of the loop, we can generate efficient hardware for the unrolled loop. Therefore, by using pattern matching algorithms to detect unrolled loops and sophisticated scheduling algorithms to synthesize unrolled loops, we can generate hardware that is very similar to a normal loop.

We have recently begun to look at the effects of different instruction sets on the results of binary partitioning. Initially, instruction side effects seemed to potentially degrade the quality of hardware, but we found that decompilation techniques fixed this problem. We plan on handling predicated instructions either by extracting the conditition from the predicated instruction and including the condition in the control-flow graph, or by using a conditional write to the destination specified by the predicated instruction. For all the instruction sets that we have currently looked at, register windows in the SPARC instruction set seem to be the hardest to overcome during decompilation. We are currently working on a solution to register windows. We are also currently working on converting our partitioning tools to the ARM instruction set, which will show that binary partitioning is possible on a variety of popular embedded processors.

6. CONCLUSIONS

Hardware/software partitioning is an effective technique to increase performance and reduce energy in embedded systems. Traditional source-level partitioning approaches have the drawback of requiring an inconvenient integration into a tool flow. Binary-level partitioning largely solves the tool flow problem while achieving comparable results. Binary partitioning has the added benefits of more accurate estimation and the ability to create hardware for assembly and object code.

In this paper, we showed that by incorporating powerful automatic decompilation methods, along with behavioral synthesis, into a binary-level partitioner, excellent partitioning speed, power, and area can result. We examined examples from MediaBench, NetBench, and EEMBC benchmarks. Binary-level partitioning achieved an average speedup of 3.0 and energy savings of 52%, while only requiring 14,772 gates on average. We also compared source-level partitioning results with automated binary partitioning results to show that similar results can in fact be achieved. Speedups averaged 2.6 for source-level examples and 2.3 for binary-level examples. Energy savings were also similar, with source-level partitioning achieving 53% savings and binary-level partitioning achieving 50% savings. The binary-level results required less area than the source-level results.

7. REFERENCES

[1] Altera Corporation, *ARM-Based Embedded Processor PLDs*, August, 2001.

[2] P. Athanas, H. Silverman: Processor Reconfiguration Through Instruction-Set Metamorphosis. IEEE Computer, March 1993.

[3] A. Balboni, W. Fornaciari and D. Sciuto. Partitioning and Exploration in the TOSCA Co-Design Flow. International Workshop on Hardware/Software Codesign, pp. 62-69, 1996.

[4] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems. The Journal of Supercomputing, vol. 21, pp. 117-130, 2002.

[5] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler. DEFACTO: A Design Environment for Adaptive Computing Technology. In Reconfigurable Architectures Workshop, RAW'99, April 1999.

[6] 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.

[7] 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.

[8] C. Cifuentes. Structuring Decompiled Graphs. In Proceedings of the International Conference on Compiler Construction, volume 1060 of Lecture Notes in Computer Science, pages 91--105. April 1996.

[9] EEMBC. http://www.eembc.org/.

[10] P. Eles, Z. Peng, K. Kuchchinski and A. Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. Kluwer's Design Automation for Embedded Systems, vol2, no 1, pp. 5-32, Jan 1997.

[11] R. Ernst, J. Henkel, T. Benner. Hardware-Software Cosynthesis for Microcontrollers. IEEE Design & Test of Computers, pages 64-75, October/December 1993.

[12] D.D. Gajski, F. Vahid, S. Narayan and J. Gong. SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. IEEE Transactions on VLSI Systems, Vol. 6, No. 1, pp. 84-100, 1998.

[13] R. Goering. Compiler project marks Synopsys' step into post-ASIC world. EE Times, August 28, 2000, http://www.eedesign.com/story/OEG20000828S0020.

[14] M. Gokhale, J. Stone. NAPA C: Compiling for hybrid RISC/FPGA architectures. IEEE Symposium on FPGAs for Custom Computing Machines, FCCM '98.

[15] R. Gupta, G. De Micheli. Hardware-Software Cosynthesis for Digital Systems. IEEE Design & Test of Computers, pages 29-41, September 1993.

[16] J. Hauser, J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. IEEE Symposium on FPGAs for Custom Computing Machines, pages 12-21, Napa Valley, CA, April 1997.

[17] J. Henkel and R. Ernst. A Hardware/Software Partitioner using a Dynamically Determined Granularity. Design Automation Conference, 1997.

[18] J. Henkel, Y. Li. Energy-conscious HW/SW-partitioning of embedded systems: A Case Study on an MPEG-2 Encoder. Proceedings of Sixth International Workshop on Hardware/Software Codesign, March 1998, pp. 23-27.

[19] J. Henkel. A low power hardware/software partitioning approach for core-based embedded systems. Proceedings of the 36th ACM/IEEE conference on Design automation conference, pp. 122 – 127,1999.

[20] Intel StrongArm 1110 Processor, http://developer.intel.com/design/strong.

[21] A. Kalavade and E. Lee. A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem. International Workshop on Hardware/Software Codesign, 1994, pp. 42-48.

[22] C. Lee, M. Potkonjak and W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, MICRO 1997.

[23] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, Hardware-Software Co-Design of Embedded Reconfigurable Architectures. Proceedings of Design Automation Conf. (DAC), 1999.

[24] R. Metzger, Automated Recognition of Parallel Algorithms in Scientific Applications. In IJCAI-95 Workshop Program Working Notes: "The Next Generation of Plan Recognition Systems". Sponsored jointly by IJCAII/AAAI/CSCSI, Aug. 1995.

[25] MIPS Technologies, Inc., http://www.mips.com.

[26] NetBench, http://cares.icsl.ucla.edu/NetBench/.

[27] Proceler, http://www.proceler.com.

[28] G. Stitt, B. Grattan, J. Villarreal and F. Vahid. Using On-Chip Configurable Logic to Reduce Embedded System Software Energy. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 2002.

[29] G. Stitt, F. Vahid. Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic. IEEE Design & Test of Computers. November-December 2002. pp.36-43.

[30] G. Stitt, F. Vahid. Hardware/Software Partitioning of Software Binaries. IEEE/ACM International Conference on Computer Aided Design (ICCAD), November 2002. pp. 164-170.

[31] Triscend Corporation, http://www.triscend.com/. 2002.

[32] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels and I. Bolsens. Hardware/Software Partitioning of Embedded System in OCAPI-xl. International Symposium on Hardware/Software Codesign, pp. 30-35, 2001.

[33] Virtex Power Estimator, http://support.xilinx.com/cgi-bin/powerweb.pl.

[34] M. Wan, Y. Ichikawa, D. Lidsky, J. Rabaey. An energy conscious methodology for early design exploration of heterogeneous DSPs. Proceedings of the IEEE 1998 Custom Integrated Circuits Conference, p.111-117, Santa Clara, May 1998.

[35] Xilinx Corporation, *Virtex-II Pro Platform FPGA Handbook*, January 31, 2002.