Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decode

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

Gordon McGregor and Brian Einloth
Freescale Semiconductor
Gordon.Mcgregor@freescale.com
Brian.Einloth@freescale.com

ABSTRACT
We describe results of a case study whose intent was to determine whether new techniques for hardware/software partitioning of an application's binary are competitive with partitioning at the C source code level. While such competitiveness has been shown previously for standard benchmark suites involving smaller or unoptimized applications, the case study instead focuses on a complete 16,000-line highly-optimized commercial-grade application, namely an H.264 video decoder. The several month study revealed that binary partitioning was indeed competitive, achieving nearly identical 2.5x speedups as source level partitioning, compared to a standard microprocessor. Furthermore, the study revealed that several simple C-level coding modifications, including pass by value-return, function specialization, algorithmic specialization, hardware-targeted reimplementation, global array elimination, hoisting and sinking of error code, and conversion to explicit control flow, could lead to improved application speedups approaching 7x for both source level and binary level partitioning.

Categories and Subdescriptors
C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems.*

General Terms
Performance, Design.

Keywords
Hardware/software partitioning, FPGA, synthesis, embedded systems, binaries, H.264.

1. INTRODUCTION
Microprocessor platforms with on-chip field-programmable gate arrays (FPGAs), such as the Xilinx Virtex II Pro, Altera Excalibur, Triscend E5/A7, and Atmel FPSLIC, provide the opportunity for significant performance improvement compared with software execution on an embedded microprocessor alone. The improvement comes from partitioning critical software kernels to the FPGA hardware. Designers traditionally perform such hardware/software partitioning manually, perhaps using a hardware/software focused high-level language such as SystemC [8], SA-C [2], HandelC [12], or SiliconC [1]. Recently, automated compilers with built in partitioning capabilities have appeared, such as CatapultC [4] and XPRES [16], and research approaches such as [7][10]. These compiler-based approaches provide an excellent technical solution for hardware/software partitioning, commonly achieving order of magnitude performance improvements.

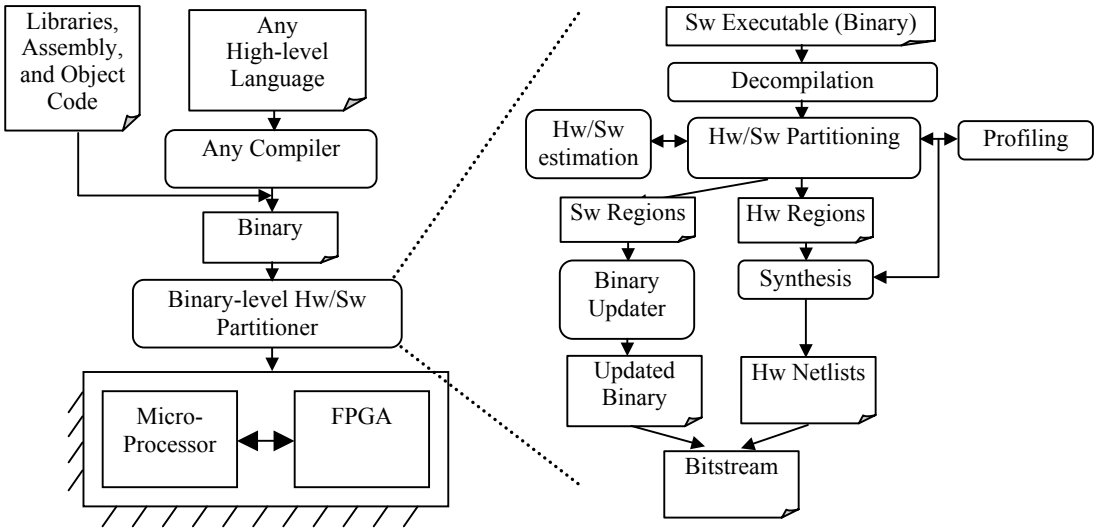
A partitioning approach that operates on the binary, rather than source, may expand the applicability of hardware/software partitioning to an even broader market. Binary partitioning operates on the executable generated after compilation and linking. From a tool flow perspective, a binary approach avoids placing restrictions on the language, compiler or integrated design environment to be used, which are often well established in industry software design flows. A binary approach supports applications built from multiple source languages, built using third party object code for which no source is available, or with parts written in assembly language for optimization purposes or due to use of legacy code. We originally introduced binary partitioning in [15], trading off reduced speedup compared to source partitioning, for easier tool flow integration. Mittal used binary partitioning to map legacy digital-signal processor binaries onto FPGAs [11]. A commercial product from Critical Blue [6] uses binary partitioning to create a custom very-large instruction word co-processor that speeds up software execution on a standard embedded microprocessor.

A further direction of binary partitioning is dynamic binary partitioning, known as warp processing, which we have proposed [13]. In warp processing, on-chip tools dynamically and transparently detect an executing binary's kernels, decompile each kernel to a control/dataflow graph, and then synthesize, place and route the kernel onto a configurable logic fabric. While implementing all such tools on-chip sounds infeasible, careful creation of lean algorithms and a highly-simplified fabric enable execution of those tools on a low-end ARM embedded processor (i.e., an ARM7) in less than two seconds for MediaBench, PowerStone, EEMBC, and other benchmarks [13]. Warp processing potentially opens up hardware/software partitioning to the entire microprocessor market, including server, desktop, mobile, and other microprocessor platforms, just as dynamic binary translation and optimization presently enables use of underlying architectures radically different from x86 instructions in nearly all modern x86 microprocessor architectures. A major U.S. desktop microprocessor vendor is presently fabricating a prototype warp processing chip.

To compensate for the speedup gap between source and binary partitioning, we have applied existing advanced decompilation methods, and developed several new synthesis-focused methods to recover high-level constructs, such as loops and arrays, from the binary [14]. We have shown that such decompilation makes binary partitioning competitive with source

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

Figure 1: Binary-level hardware/software partitioning.



level partitioning for a wide variety of benchmarks drawn from MediaBench, PowerStone, and EEMBC [14]. Of course, we acknowledge that binary level partitioning will likely never equal source level partitioning for all benchmarks, especially for highly-advanced source-level methods. But by closing the gap, binary partitioning can at least broaden the market for hardware/software partitioning by resulting in respectable speedups in binary-based approaches.

However, a limitation of previous binary partitioning studies (and in fact of previous source level partitioning studies too) is that the benchmarks used tend to be small benchmarks (kernels) and/or unoptimized software (reference code). Speeding up a kernel alone may result in misleadingly-high speedups as the surrounding support code, which often can't be sped up, is excluded. Speeding up unoptimized code also may yield misleadingly-high speedups, as clearly a slow original execution makes hardware implementation look even faster. A concern is whether binary partitioning would achieve speedups for real commercial highly-optimized embedded applications, which are often painstakingly hand-optimized for performance. We wondered whether such optimized code could still be sped up.

Through our collaboration with Freescale Semiconductor [9], we obtained the 16,000-line C source code for an H.264 video decoder, which unlike publicly available reference code has been highly hand-optimized for performance. We then performed a several month case study that involved profiling the application to find the most critical functions (of which there were nearly 50, rather than just 2-5 as in most benchmarks) out of the many hundreds of total functions. The study also involved performing binary and source level partitioning, involving extensive manual analysis to look beyond present synthesis limitations of source-level tools. In addition to finding the two approaches competitive, we found several simple modifications to the C code that would enable even greater speedups, for either source or binary partitioning.

2. BINARY-LEVEL HW/SW PARTITIONING

Binary-level hardware/software partitioning is the process of partitioning computation kernels of a software binary into regions that will be implemented in custom hardware and regions that will execute in the existing binary format, with possibly some added instructions to communicate with the hardware regions. Figure 1 illustrates a binary-level hardware/software partitioning approach. The software developer initially writes the application to be partitioned in any high-level language, such as C, and then compiles the high-level code using any software compiler. During compilation, the compiler links in additional object code from libraries and possibly hand-optimized assembly, forming a software binary. The *binary-level hw/sw partitioner* then uses *decompilation* to recover high-level information that is necessary for hardware/software partitioning and synthesis. *Hw/sw partitioning* determines the regions of the binary that will be implemented in hardware, possibly using profiling information and hardware/software performance estimation. *Synthesis* then converts the regions selected for hardware into a hardware netlist. The *binary updater* modifies the original software binary to remove the software implementation of regions moved to hardware in addition to adding new instructions to communicate with the hardware regions. Finally, the binary-level hw/sw partitioner combines the updated binary with the hardware netlist to form a bitfile that configures the microprocessor/FPGA platform.

2.1 Decompilation

Binary partitioning uses decompilation to recover high-level information that the synthesis tool needs to synthesize efficient hardware. Without the necessary high-level information available in the original code, synthesis of software binaries would produce hardware much slower than would be synthesized by compiler-based approaches. For example, compilers commonly unroll loops to expose parallelism. Without decompilation, binary synthesis approaches cannot unroll loops, losing potential parallelism, because the loop structure is not explicit at the binary level.

Decompilation for binary synthesis consists of several steps. Initially, binary parsing converts the binary into an instruction set independent representation that allows all of the following steps to be performed for any instruction set [5]. The decompiler next analyzes the instruction-set independent representation and creates a control/data flow graph (CDFG). The decompiler then performs control structure recovery [5] to identify regions of the CDFG that correspond to loops and if statements. Next, the decompiler performs array recovery to determine the regions of memory that correspond to arrays. The decompiler also performs optimizations that remove overhead introduced by the instruction set, in addition to undoing optimizations that were

applied by the software compiler that may obscure some of the high-level constructs in the binary.

Previous work [14] has shown that in some cases, decompilation is successful enough to recover enough high-level information to allow binary partitioning to achieve identical results as a compiler-based approach. In fact, if decompilation is able to recover the original representation, then binary synthesis is equivalent to synthesis from C.

Although decompilation can commonly recover all necessary information for partitioning, decompilation does have limitations that can result in an incomplete recovery of high-level information. One limitation of decompilation is the inability to statically analyze control flow in the presence of indirect jumps. Indirect jumps typically result from function pointers and switch statements. Another limitation of decompilation is the inability to guarantee array recovery. Array recovery relies on memory access patterns to determine regions of memory that correspond to arrays. If an array is not accessed in a regular pattern, the decompiler may fail to identify the array. The inability to recover all arrays may be a disadvantage for binary partitioning because ideally these arrays could be stored in multiple memories on the FPGA, allowing for parallel accesses to individual array elements. Fortunately, much of the code typically considered for hardware implementation is written in a way that is ideal for decompilation.

2.2 Hardware/Software Partitioning

Due to Amdahl's Law, selecting hardware regions based on percentage of execution time guarantees the largest potential speedup. Although selecting hardware regions solely based on percentage of execution time does not guarantee satisfaction of area constraints, in this study we are mainly concerned with obtaining the maximum speedup. Therefore, we used a simplified partitioning technique that uses profiling results to sort the functions of the application based on the percentage of total execution time. We also consider loops in addition to functions, but for the H.264 decoder, the body of most of the functions was just a loop, so we chose to partition the entire function. Our partitioning technique selects functions for hardware implementation in order of the profiling results until the performance benefits from additional functions is negligible. If a function doesn't benefit from hardware implementation, our partitioning technique skips the inappropriate function and moves to the next most frequent function. We could of course extend our approach to meet specific area constraints by investigating different hardware implementations of each function or by using partial implementations of functions that have certain regions not appropriate for hardware.

3. H.264

H.264 is a video codec included in the MPEG4 standard as MPEG4-Part 10, also known as advanced video coding (AVC). H.264 is capable of compressing a video using up to three times fewer bits than MPEG2 and results in much higher quality for encoded videos. This section provides a brief overview of the differences between H.264 and previous standards.

H.264 achieves significant compression improvements by refining each step from previous standards, as opposed to adding entirely new steps. Unlike MPEG2, H.264 can perform both intraprediction and interprediction to predict the samples in macroblocks. For intraprediction, H.264 utilizes nine prediction modes for luma values and four prediction modes for chroma values in order to predict a macroblock based on other macroblocks within the same frame. Intraprediction can predict either entire 16x16 macroblocks or smaller 4x4 blocks. Interprediction uses a more flexible form of motion compensation than in previous standards that is capable of predicting regions ranging from 16x16 samples to 4x4 samples. Also, unlike MPEG2, interprediction in H.264 may utilize up to 32 reference frames. H.264 also supports sub-pixel motion compensation up to one-quarter pixel for luma samples and one eighth of a pixel for chroma samples. In addition to predicting macroblocks using motion compensation, H.264 utilizes the high correlation of motion vectors to predict the values of motion vectors, further reducing the size of each predicted frame.

Whereas previous standards utilized the discrete cosine transform on 8x8 samples, H.264 supports three different transforms based on the data being coded. H.264 utilizes a 4x4 transform for luma DC coefficients, a 2x2 transform for chroma DC coefficients, and an additional 4x4 transform for all other data. These transforms are integer transforms and can be implemented using only shifts and adders. The transforms can also be inverted exactly without mismatches.

H.264 applies a deblocking filter during decoding to improve blocking distortion that occurs from transforms using smaller block sizes. In addition to improving image quality, the deblocking filter is applied by the encoder to all reference frames used in interprediction to help reduce the size of residual data in predicted frames.

H.264 supports two possible entropy coding techniques. Context-adaptive variable length coding (CAVLC) encodes transform coefficients by adaptively choosing from multiple variable-length code word tables. CAVLC encodes non-transform coefficient data using a single table. Context-adaptive binary arithmetic coding (CABAC) is a more complex method of entropy coding capable of achieving better compression at the cost of longer decoding/encoding execution times.

4. CASE STUDY OF H.264 DECODE

4.1 Experimental Setup

The target architecture for our experiments is a hypothetical platform consisting of an ARM9 microprocessor running at 200 MHz and a Xilinx Virtex II FPGA running at 100 MHz. The communication model used by the microprocessor and FPGA uses shared memory and memory-mapped registers within the FPGA. Hardware within the FPGA accesses main memory using DMA or by directly reading from the data cache. The communication model maintains data coherency by requiring the execution of the microprocessor and FPGA to be mutually exclusive. The FPGA writes to memory through the cache to prevent the microprocessor from reading old values after resuming software execution. The microprocessor copies all registers that are needed as input to the hardware to registers in the FPGA before hardware execution and reads modified registers back before resuming software execution.

We generated a software binary by compiling the H.264 decoder using gcc ported to the ARM, using the highest level of software optimizations (-O3).

We performed profiling of the decoder using the LOOAN profiler, which we developed ourselves. This profiler uses instruction traces from an ARM version of the SimpleScalar [3] simulator. The output of the profiler is a list of all functions and loops in the application and their corresponding percentages of total execution time.

To perform binary partitioning of the decoder, we developed tools that implement the steps described in Section 2, requiring approximately 30,000 lines of C code. The output of the tools is register-transfer-level (RTL) VHDL. The execution time of the tools running on a 2.8 GHz Pentium IV is typically several

seconds. After obtaining the RTL VHDL, we use Xilinx ISE to synthesize the code to a netlist for the Virtex II FPGA.

To obtain C-level partitioning results, we manually created hardware for each of the functions to guarantee the best possible hardware for the C code. Current commercial tools are limited to the use of specific C constructs and may not create efficient hardware if other constructs appear in the code. Instead, our 2-month manual analysis ensured the hardware represented close to the ideal hardware that could be synthesized from C. Manual analysis was possible because the average function consisted of only 20-50 lines of code. To manually generate the hardware, we first created a control/data flow graph and then analyzed all dependencies to determine the potential parallelism of the code. We unrolled loops to increase parallelism if dependencies were not violated. When possible, we copied data from main memory into multiple FPGA memories to increase the memory bandwidth during computational kernels that needed fast access to multiple array elements. We performed standard compiler optimizations such as strength reduction, tree-height reduction, etc. We scheduled each operation in the CDFG, and then created a controller and datapath to implement the operations.

4.2 Profiling Results

Table 1 shows a summary of the profiling results for the H.264 decoder. *Function Name* specifies the name of the function, *%Size* specifies the percentage of the total application size in terms of assembly instructions for the specified function, $\%Time_I$ is the percentage of total execution time spent in the individual function, $\%Time_C$ is the cumulative percentage of total execution time spent in the specified function and every more frequent function. S_{Ideal} is the ideal speedup assuming that the specified function and all more frequent functions execute in zero time.

Many embedded applications typically spend the majority of execution time in a few small loops [14], implying that a partitioning tool can obtain the majority of speedup by implementing the several most frequent loops in hardware. However, the profiling results shown in Table 1 for the H.264 decoder show that the several most frequent regions are responsible for less than 20% of total execution time, corresponding to an ideal speedup of only 1.25.

Upon further analysis of the profiling results, we found that the H.264 decoder does follow the well-known 90-10 rule, which states that 90% of execution time is typically spent in 10% of the code. More specifically, H.264 decode spends 93% of execution time in 14% of the code. The 14% percent of the code represents 6,110 instructions out of a total of 42,835 instructions. We determined that most of the speedup could be obtained by partitioning the 52 most frequent functions to hardware, allowing for an ideal speedup of 21.2. The relationship between the ideal speedup and the number of functions implemented in hardware is shown in Figure 2. The ideal speedup begins to increase very rapidly after approximately 43 functions are implemented in hardware, which corresponds to approximately 90% of execution time. Although the ideal speedup continues to increase past the 52 most frequent functions that we implement in hardware, the actual speedups tend to level off around 52 hardware functions, as will be shown in the following sections.

4.3 Partitioning Results

The actual speedups obtained compared to a 200 MHz ARM9 by partitioning the H.264 decoder at both the binary level and the C level are shown in Figure 2. The figure shows how the actual speedup increases as more functions are implemented in hardware. The overall application speedup from binary partitioning was 2.48 when the 52 most frequent regions were implemented in hardware. C-level partitioning achieved only a slightly larger speedup of 2.53.

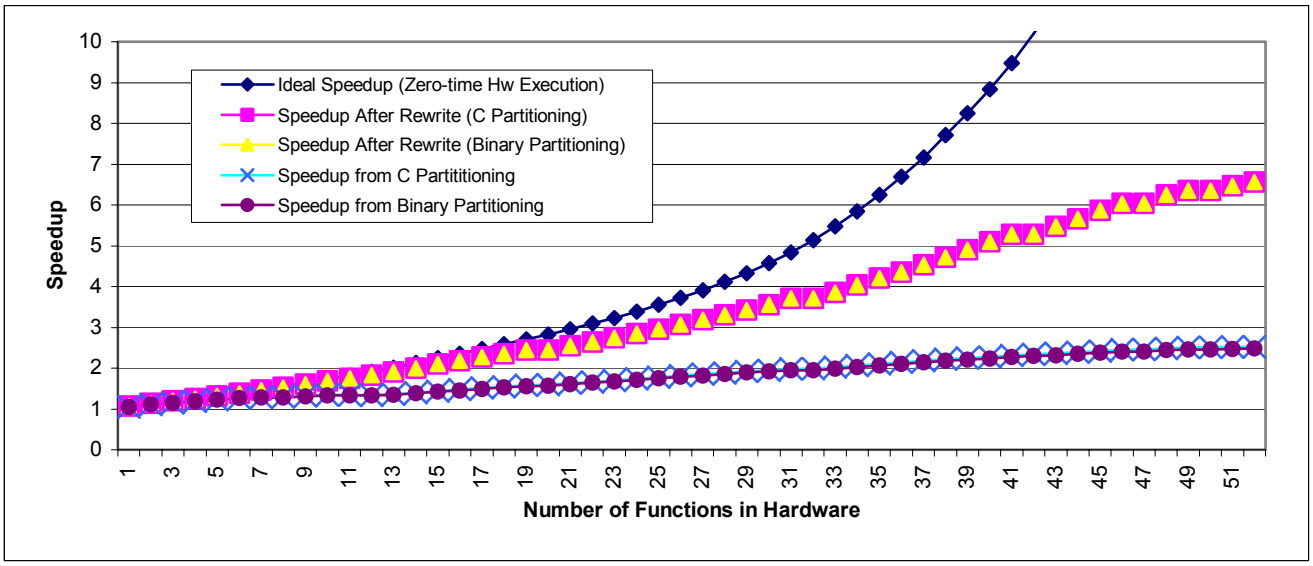
Table 1: Profiling results showing the percentage of total size, percentage of total execution time, cumulative percentage of execution time, and ideal speedup for the most frequent functions in H.264 decode. The total size of the decoder was 42,835 instructions.

Function Name	%Size	$\%Time_I$	$\%Time_C$	S_{Ideal}
MotionComp_00	0.1%	6.76%	6.76%	1.1
InvTransform4x4	0.1%	5.77%	12.53%	1.1
FindHorizontalBS	0.1%	4.15%	16.68%	1.2
GetBits	0.1%	4.09%	20.78%	1.3
FindVerticalBS	0.1%	3.93%	24.70%	1.3
MotionCompChromaFullXF	0.1%	3.91%	28.61%	1.4
FilterHorizontalLuma	1.3%	3.91%	32.52%	1.5
FilterVerticalLuma	1.1%	3.32%	35.84%	1.6
FilterHorizontalChroma	0.3%	3.12%	38.96%	1.6
CombineCoefsZerosInvQua	0.2%	3.06%	42.02%	1.7
memset	0.0%	2.85%	44.87%	1.8
MotionCompensate	0.4%	2.79%	47.66%	1.9
FilterVerticalChroma	0.3%	2.66%	50.32%	2.0
MotionCompChromaFracX	0.1%	2.66%	52.98%	2.1
ReadLeadingZerosAndOne	0.1%	2.60%	55.58%	2.3
DecodeCoeffTokenNormal	0.2%	1.97%	57.54%	2.4
DeblockingFilterLumaRow	0.6%	1.88%	59.42%	2.5
DecodeZeros	0.2%	1.87%	61.29%	2.6
MotionComp_23	0.7%	1.67%	62.96%	2.7
DecodeBlockCoefLevels	0.1%	1.61%	64.57%	2.8
MotionComp_21	0.7%	1.60%	66.17%	3.0
FindBoundaryStrengthPMB	0.1%	1.49%	67.66%	3.1

C-level partitioning outperformed binary partitioning because several of the functions implemented in hardware used switch statements, resulting in indirect jumps that the decompiler was unable to remove. Binary partitioning was unable to obtain any speedup for these functions. Also, for several of the functions, the decompiler was unable to recover arrays successfully. Array recovery failed because the functions accessed global arrays using indices that were parameters to the function. At the binary level, there is no known way of determining the existence of an array in such an instance. These arrays could potentially be implemented in memories on the FPGA during C-level partitioning. However, a binary partitioning approach must fetch the appropriate array elements from main memory each time the code accesses an array element. One would likely expect the speedup of binary partitioning to be much less because of the incomplete recovery of high-level information. However, the results were similar because the functions where high-level information could not be recovered consisted of a small percentage of total execution time.

For both binary partitioning and C-level partitioning, the main limitation of speedup was memory bandwidth. Many of the functions implemented in hardware contained parallelizable loops. However, the synthesis tools could not take advantage of this parallelism because the hardware had to wait for the data needed by the loops to be fetched from memory. Our binary synthesis tools fetch data for the hardware by copying the data from main memory into multiple memories in the FPGA, as early as dependencies allow. However, much of the data needed by these loops is passed to a function as a pointer. If other pointers are passed into the function, local alias analysis is

Figure 2: Speedup when implementing multiple functions in hardware. The figure shows that binary partitioning is competitive with C-level partitioning and that simple rewrites of the original code can improve speedups for both binary partitioning and C-level partitioning.



unable to determine if copying the data to FPGA memories will violate the dependencies of the original code. Global alias analysis may allow for further optimization, but global analysis is very difficult and time consuming and is rarely performed by existing synthesis tools. In order to obtain larger speedups, we have to modify the original C code to allow for the synthesis tool to determine aliases. Improved alias analysis could allow for data to be fetched by the FPGA much earlier and in some cases the data could be completely moved to the FPGA without ever having to refetch the data from main memory.

Performance of both C-level hardware and binary-level hardware was also limited by the use of software algorithms when hardware algorithms would be more appropriate. One of the functions, "GetBits", handled fetching a specified number of bits from a buffer. This function could easily be implemented as a shift register or even as wires if the synthesis tool applied function specialization. However, based on the C description, the code synthesizes to hardware that requires many additional cycles. A synthesis tool could potentially implement another function, ReadLeadingZerosAndOne, as a priority encoder, requiring only a single cycle. However, the C description results in a state machine that requires seven cycles.

Another reason for poor hardware performance was because the C code contained constructs that were inappropriate for synthesis. The C code contained multiple uses of function pointers and global variables, which complicated dependency analysis and limited the amount of parallelism in the hardware.

4.4 Partitioning Results after Code Rewrite

The limited speedups reported in the previous sections were caused largely in part by the C constructs and coding style of the H.264 decoder. To see the potential speedup of H.264 decode in general, we performed an in-depth study of how the code could be rewritten to achieve improved performance for hardware synthesized from both the C code and the software binary. We developed guidelines for improving C code used for synthesis, and applied these guidelines to the H.264 decoder.

We addressed the memory bandwidth issues by rewriting small sections of the code to implement *pass by value-return* in order to make alias analysis easier for the synthesis tool, which allowed data to be efficiently moved to the FPGA. We first identified groups of functions that executed consecutively and operated on the same data. We then declared a local array and explicitly copied all data that was needed by the hardware into this local array, allowing for alias analysis to more easily determine the validity of moving the data to the FPGA. We minimized the overhead from copying data into the local array by amortizing the overhead over multiple functions that needed the data. Theoretically, a synthesis tool may be able to automate this process, but existing tools would require an explicit rewrite. This simple rewrite resulted in large speedups in many of the functions. For example, many of the motion compensation functions performed operations on a macroblock. By copying the data for the macroblock into a local array and then using the data in the array for several functions, we were able to unroll the loops in the motion compensation functions and obtain much more parallelism. The speedups of the motion compensation functions increased from an average of 6.5 to 89.5 after the rewrite.

We manually performed *function specialization* in cases where specific input values resulted in efficient hardware. Many of the motion compensation functions contained loops that could be completely unrolled. However, the bounds of the loops were passed to the function containing the loop. To be able to determine the bounds of the loops statically, we had to perform function specialization for common bounds. The synthesis tool could theoretically perform this function specialization by performing value profiling, but manually performing the specialization guaranteed that the loops were unrolled.

We also rewrote the code to perform *algorithmic specialization* for hardware, by replacing algorithms more appropriate for software with more parallel algorithms. For the function ReadLeadingZerosAndOne we rewrote the code to model a priority encoder, resulting in a speedup of 7 for that function.

In some cases, we performed *hardware-targeted reimplementation* of the existing algorithm to obtain more efficient hardware. For the GetBits function, the code originally used a struct that contained a variable to store a buffer of bits and an integer to store the number of valid bits within the buffer. Ideally, the code could be implemented in hardware as a shift register. However, the synthesis tool was unable to determine a

relationship between the variables that stored the buffer and the size of the buffer. We rewrote the code to modify the value of the buffer when bits were read from the buffer. Through data flow analysis, the synthesis tool was then able to determine that the buffer could be implemented as a shift register, resulting in a speedup of more than 5 for that function.

We also performed *global array elimination* to replace global arrays of constants with logical expressions that could calculate the appropriate constant value based on the index of the array. For example, one global array was used to efficiently implement clipping a value between 0 and 255. We rewrote accesses to this array with if statements that assigned values greater than or less than 0 and 255 to be 0 or 255. When synthesized, this code only required a comparator and a mux. A synthesis tool could further optimize this code by performing logic synthesis to convert the code to Boolean expressions. We removed other global variables from the code by adding additional inputs to the functions that used global variables.

We performed *hoisting and sinking of error checking code* to remove error checking within computation kernels. In many cases, the error checking greatly limited the parallelism of the synthesized hardware. Whenever possible, we moved the error checking before or after the computation. When the error checking had to be performed during the computation, we simply set a flag to signal that an error occurred, and then checked the flag after the computation completed to see if the output of the computation was valid.

Whenever possible, we performed *conversion to explicit control flow* to replace all function pointers with if-else statements. After the rewrite, functions with modified control flow obtained an average speedup of 4.0.

Ideally, we would rewrite the code so that each stage of the H.264 decoder could be pipelined. Such an implementation would result in a large increase in throughput. However, synthesis tools generally cannot determine coarse-grained parallelism that is not explicit in the code. We do not consider explicit coarse-grained parallelism as part of the rewrite because there are multiple thread packages for C. We could of course implement synthesis for a specific thread package, but we consider this to be a restriction on tool flow, which contradicts the purpose of binary partitioning.

The increased overall application speedups from rewriting the code are shown in Figure 2 for both binary partitioning and C-level partitioning. The speedups for binary partitioning and C-level partitioning were almost identical, with binary partitioning achieving a speedup of 6.55 and C-level partitioning achieving a speedup of 6.56. Binary partitioning was able to achieve a similar speedup compared to C partitioning because decompilation recovered almost all important high-level information from the original C code. These results imply that if code is written in a way that is suitable for hardware synthesis, then the performance of hardware generated from binary partitioning and C-level partitioning should be almost identical.

5. CONCLUSIONS

In this paper, we presented a case study showing that binary hardware/software partitioning is competitive with source-level compiler-based partitioning. Unlike previous work, which performed binary partitioning on simple benchmarks, we partitioned large, highly-optimized commercial code for an H.264 decoder. Binary partitioning and compiler-based partitioning achieved almost identical speedups of 2.5 compared to an ARM9 microprocessor. In addition, we also found that simple rewrites to the original code following several proposed guidelines resulted in much larger speedups of 6.5, both at the binary level and source level. These guidelines included using pass by value-return, function specialization, algorithmic specialization, hardware-targeted reimplementation, global array elimination, hoisting and sinking of error checking, and conversion to explicit control flow. Of course, rewriting the original code is not possible for all binary partitioning approaches, such as partitioning of legacy code and hand-optimized assembly.

Binary partitioning will likely never achieve equal results to compiler-based partitioning on all benchmarks. However, by achieving reasonably competitive results, binary partitioning can expand the market of hardware/software partitioning to include more software developers.

6. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation (CCR-0203829) and by the Semiconductor Research Corporation (2003-HJ-1046G). The source code for the H.264 decoder was provided by Freescale Semiconductor.

7. REFERENCES

[1] C. Scott Ananian. SiliconC: A Hardware Backend for SUIF. http://flex-compiler.lcs.mit.edu/SiliconC.

[2] 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.

[3] 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.

[4] CatapultC. http://www.mentor.com/products/c-based_design/

[5] 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.

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

[7] 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.

[8] A. Fin, F. Fummi, M. Signoretto. SystemC: A Homogenous Environment to Test Embedded Systems. CODES 2001.

[9] Freescale Semiconductor. http://www.freescale.com/.

[10] 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, June 1999.

[11] G. Mittal, D. Zaretsky, X. Tang and P. Banerjee. Overview of the FREEDOM Compiler for Mapping DSP Software to FPGAs. IEEE Symposium on Field-Programmable Custom Computing Machines. April 2004.

[12] OXFORD Hardware Compilation Group, The Handel language, Technical Report, Oxford University 1997.

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

[14] G. Stitt and F. Vahid. A Decompilation Approach to Partitioning Software for Microprocessor/FPGA Platforms. Design and Test Europe Conference (DATE), 2005.

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

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