A Code Refinement Methodology for Performance-Improved Synthesis from C

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

ABSTRACT
Although many recent advances have been made in hardware synthesis techniques from software programming languages such as C, the performance of synthesized hardware commonly suffers due to the use of C constructs and coding practices that are not appropriate for hardware. Most previous approaches to addressing this problem require drastic changes to coding practice. We present an approach that instead requires only minimal changes but yields significant speedups. In this approach, a software developer initially writes C code as they normally would, and then applies simple refinement guidelines to only the performance-critical code regions, which are the regions most likely to be synthesized to hardware. Alternatively, if a designer is aware of performance-critical parts of the application, the guidelines could be followed during development. In this study, we analyze dozens of embedded benchmarks to determine the most common C coding practices that limit hardware performance, and introduce coding guidelines to make the code more amenable to synthesis. Those guidelines typically require minimal coding effort, generally consisting of less than ten lines of code for each guideline. The guidelines typically represent modifications that require designer knowledge, making the guidelines difficult or impossible for synthesis tools to automate. We apply these guidelines to six benchmarks, resulting in average speedups of 3.5x compared to synthesis from the original code with a negligible software size and performance overhead.

Categories and Subject Descriptors
B.5.2 [**Register-Transfer Level Implementation**]: Design Aids – *automatic synthesis, hardware description languages.*

General Terms
Performance, Design.

Keywords
Synthesis, hardware/software partitioning, coding guidelines, code refinement, embedded systems, FPGA, compilation.

ICCAD'06, November 5-9, 2006, San Jose, CA

1. INTRODUCTION
Numerous research and commercial efforts deal with synthesis and hardware/software partitioning of high-level code to allow a designer to specify an application using a single high-level representation, as opposed to specifying hardware regions using HDLs (hardware description languages). Although register-transfer level specification using HDLs is appropriate for designers concerned with obtaining maximum performance, high-level synthesis is a complementary approach that trades off hardware performance for reduced design time. Several approaches introduced new HDLs with C-like syntax, such as HardwareC [9], HandelC [13], and SystemC [6]. Other approaches introduced new software-programming languages to be used with synthesis, such as NapaC [4], Streams-C [7], and SA-C [1]. Several synthesis tools, such as ROCCC [12], SPARK [5], and CatapultC [2], have attempted to synthesize standard C onto FPGAs.

Although advances have been made in synthesis from software code, many software constructs remain inappropriate for existing synthesis techniques. For example, many synthesis tools do not effectively support pointers [5][15] due to alias problems and the difficulty of scheduling irregular memory access patterns. Function pointers, global variables, and recursion are also problematic for synthesis.

One potential solution to the problems with synthesis from software code is for a software developer to simply not use problematic constructs. However, this approach is unlikely to be widely accepted by many developers because many of these constructs are useful for software development. In addition, these problematic constructs are widely used in existing code that may be adapted for synthesis. Furthermore, a software developer may not even be certain that his/her code will have portions synthesized to hardware in FPGAs.

Many previous works have solved these problems by introducing C-based languages [1][6][7][9][13] that eliminate problematic constructs, typically yielding more parallelism and faster hardware. Although these languages are good technical solutions, they have the disadvantage of using non-standard software code. In this paper, we instead focus on refining C coding practices to improve synthesis from standard C, at the possible cost of decreased performance compared to C-based HDLs. If software developers have flexibility in how they can write code, and need only perform minimal changes to create fast hardware, synthesis from software code may gain wider acceptance.

In this study, we analyzed C-code benchmarks in light of synthesis concerns to identify common coding practices that are

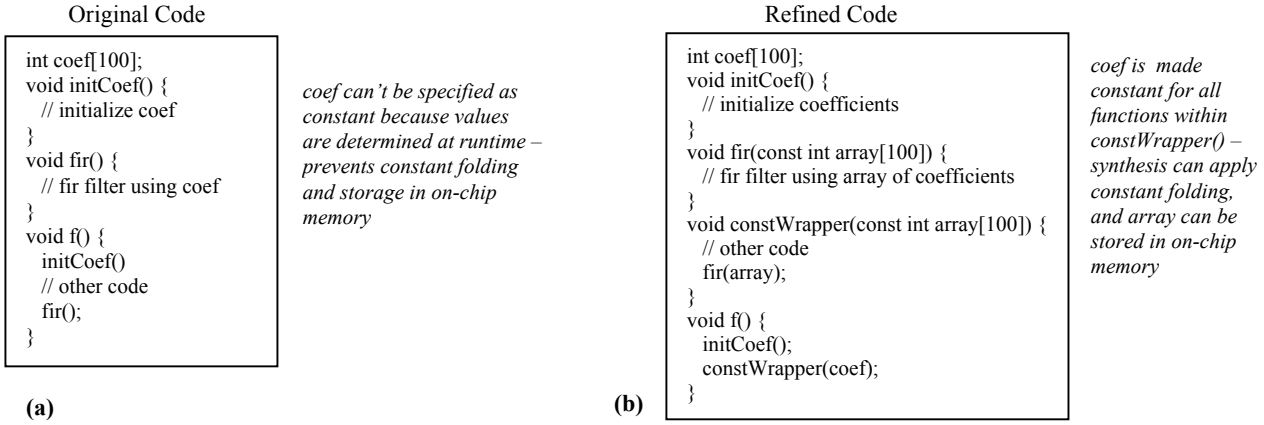

Figure 1: *Conversion to constants* of an array of coefficients. a) Original code that initializes an array of coefficients for a FIR filter. These coefficients are actually constants but cannot be specified as constants because they are determined at runtime. b) After initializing the array of constants, the wrapper function specifies from that point on that the array is constant, allowing the synthesis tool to perform constant folding in the fir() function, and for the array to be stored in on-chip memory.

problematic in hardware. Based on this analysis, we developed guidelines for refining critical regions of C code to improve the quality of synthesized hardware. To use these guidelines, a software developer would have the flexibility to write the application using all C constructs and would then refine performance critical regions using the guidelines. Alternatively, a designer with knowledge of performance-critical parts of an application could follow the guidelines during development. The main advantage of this approach is that the performance critical regions typically consist of approximately only 10% of the code, corresponding to the well-known 90-10 rule. Therefore, only a small percentage of the code would have to be refined. Furthermore, the guidelines typically consist of simple rewrites that require minimal designer effort, generally taking minutes per guideline. We focus on code refinements for the C language due to the language's popularity. Some of our refinements are C-specific, while others are applicable to other software languages.

One potential problem with a guideline-based refinement approach is that in some cases software developers may write one software application that later gets mapped to different architectures, which may or may not contain FPGAs. Even on architectures with FPGA, hardware/software partitioning may select different functions for hardware implementation based on available FPGA resources. Therefore, in some situations, the refined regions may not be mapped to hardware. Due to the differences in spatial and temporal computing, the code that is refined for hardware implementation could potentially have a software performance overhead. We therefore analyze this software overhead and present a methodology that provides a good tradeoff between improved hardware performance and software performance overhead.

Theoretically, one could argue that the need for coding guidelines could be obviated by improved synthesis optimizations. However, most of the guidelines we propose require designer knowledge that cannot reasonably be expected to be extracted from the code by a synthesis tool. Therefore, we claim that these guidelines are complementary to improved synthesis optimizations. Nevertheless, we discuss the possibility and hurdles of automating each guideline during synthesis.

The refinement guidelines are not meant to be exhaustive, but are instead a solution to commonly occurring problems in standard embedded systems benchmark code. Analysis of other applications, especially from different domains, may yield new guidelines.

To our knowledge, coding guidelines for high-level synthesis have been limited to avoiding pointer operations and recursion [5][15]. Many previous works have presented coding guidelines for register transfer-level synthesis [3], but those guidelines do not apply to high-level synthesis.

2. CODING GUIDELINES

This section describes each of the proposed coding guidelines. Each subsection discusses the problem in C that the guideline solves and how the guideline solves the problem. The potential software overhead of each guideline is also discussed along with the possibility of automating the guideline in a synthesis tool. We initially proposed the guidelines presented in sections 2.7 to 2.10 in other work [17] to improve the performance of an h.264 decoder synthesized from C code. In this paper, we evaluate the performance improvements of those previous guidelines on multiple applications, consider their software overhead, and incorporate them in a refinement methodology.

2.1 Conversion to Constants (CC)

Constant folding is a standard synthesis optimization that improves hardware performance by evaluating constant expressions at compile time. In addition to constant folding, constants can also improve hardware by enabling use of on-chip memory, which increases memory bandwidth by reducing accesses to main memory. From our benchmark analysis, we have found that in many cases, software developers do not use constants appropriately, eliminating these potential performance improvements. In many cases, scalars and arrays that are actually constant are not specified as constants in the C code.

The refinement guideline *conversion to constants* explicitly informs the synthesis tool that an array or scalar is constant. In the simplest case, the refinement consists simply of adding the keyword *const* to the arrays or individual variables that are

constants. This simple refinement has no software performance overhead but potentially yields large hardware performance improvement.

In many situations, simply adding the keyword *const* to an array of constants is not possible because the software developer initializes the array at runtime using an initialization function. Software developers commonly use initialization functions when manually specifying a large number of constants is too time-consuming, such as for lookup tables of sine values. Such arrays are still essentially constant, but cannot be declared using the keyword *const*. We extend the conversion of constants refinement guideline to handle such cases. The refinement involves adding a wrapper function that takes as a parameter a *const* array. After the initialization code for the constant array, the software developer adds code that passes the array to the new wrapper function, which a synthesis tool can then recognize as constant for all functions contained within the wrapper function. In the worst case, the wrapper function would replace the body of main(). However, in most situations, the wrapper function can be isolated to a small section of code that uses the array.

To maximize the benefit of this guideline, a software developer should avoid using pointers to a constant array because a synthesis tool may not be able to statically determine the value of the pointer. Thus, even though the pointer may point to a constant value, the synthesis tool is unlikely to know what that constant value is until runtime and will not be able to apply constant folding.

Figure 1 shows an example of conversion to constants. Figure 1 (a) shows the unmodified code for an FIR filter that uses an array of coefficients, which is initialized and then never modified. A synthesis tool is unlikely to know that this array is not modified because of aliases (not shown in figure). Figure 1(b) shows the refined code after applying the conversion to constants guideline. The wrapper function constWrapper() with the constant array parameter specifies that the *coef* array is actually constant. The synthesis tool can now prefetch the *coef* array if necessary and can also potentially apply constant folding within the fir() function.

Automation: A synthesis tool could potentially automate conversion to constants for simple examples, by performing definition-use analysis on scalars and arrays. However, in the general case, the guideline cannot be automated because the presence of pointer operations may complicate data flow analysis, or even make such analysis impossible.

Overhead: The software overhead of conversion to constants when using a wrapper function consists of the extra instructions needed to call the wrapper function and to pass the address of the array parameter to any additional functions.

2.2 Conversion to Explicit Data Flow (CEDF)

The common use of global variables causes problems for synthesis. In cases of task-level parallelism, scheduling accesses to global variables can be difficult. Even without task-level parallelism, global variables may make complicated global alias and data flow analysis necessary, which may result in long compilation runtimes and in some cases is impossible.

The guideline *conversion to explicit data flow* eliminates problems caused by global variables. This guideline consists of replacing accesses to a global variable with extra parameter passing for functions that access the global variable. This parameter passing is beneficial for synthesis because the global data flow is explicit. Such explicit data flow may result in large hardware parallelism increases because a simpler data flow analysis may show that multiple functions do not have any dependencies.

Automation: In general, conversion to explicit data flow cannot be automated because of pointers that may alias a global variable. Although alias analysis may be able to handle simple cases, designer knowledge is typically required to specify the lack of aliases.

Overhead: The software overhead of conversion to explicit data flow consists of the extra instructions required for the additional parameter passing. Typically, this overhead is small unless the code uses large amounts of global variables.

2.3 Conversion to Fixed Point (CF)

The inefficiency of floating point operations in configurable logic is widely known. Therefore, an obvious optimization is to convert floating-point operations to fixed point, resulting in significant area and performance improvements.

The *conversion to fixed point* guideline can yield further improvements when paired with the other guidelines. For example, when combining conversion to fixed point with conversion to constants, a synthesis tool can generally perform optimizations such as constant folding for the fixed-point operations, which would not have been as beneficial for floating point operations.

Automation: Although in some cases a synthesis tool could automate conversion to fixed point [14], determining the required amount of precision is difficult.

Overhead: Conversion to fixed point typically has no software overhead and in most cases will improve software performance by replacing floating-point instructions with much faster integer instructions. However, this improvement is obtained by trading off higher precision in floating point.

2.4 Conversion to Explicit Memory Accesses (CEMA)

Software developers commonly use pointer operations as a substitute for array indices, to save lines of code, or in some cases to improve code performance. Although these pointer operations correspond to array accesses, a synthesis tool may not be able to determine the memory access patterns.

Conversion to explicit memory accesses makes array accesses explicit by replacing uses of pointers with array indices when possible. Such array use enables the synthesis tool to more easily determine memory access patterns, which allows data to be prefetched and also eases alias analysis.

Automation: A synthesis tool is unlikely to be able to automate conversion to explicit memory accesses, because the tool would have to determine memory access patterns, which is difficult or impossible in the presence of pointer operations.

Overhead: The software overhead of conversion to explicit memory accesses is generally small. In some situations, use of pointer operations may slightly improve performance, but many compilers convert array accesses to pointers automatically.

2.5 Constant Input Enumeration (CIE)

In many situations, software developers write high-level functions that are intended to be as general as possible. For example, h.264 supports multiple block sizes. Therefore, a single function in an h.264 decoder is likely to take the block size as a parameter. This coding practice makes the software code concise and readable, but may limit the performance of synthesized hardware. For example, if the bounds of a loop are parameters to a function and those parameters' possible values are not determined by the synthesis tool, then the loop will not be unrolled because the bounds are not known statically.

Constant input enumeration helps solve this problem. In many cases, a parameter has a small set of possible values. A software developer can enumerate all possible input values for a particular parameter using the enum type. The advantage to this guideline is that the synthesis tool knows all possible values of the parameter and can create specialized versions of the function for each possible value.

Automation: In certain situations, a synthesis tool may be able to automate this guideline by determining all possible values for a parameter, or by performing partial evaluation. However, in general, statically determining the possible values is difficult, and is therefore unlikely to be added to a synthesis tool.

Overhead: Input enumeration has no software overhead and can sometimes improve software performance when the compiler can create specialized functions for each input value.

One potential disadvantage to this guideline is that if there are many parameters that use a small number of possible values, then refining the code may require a large number of *enums*. However, in most situations, we have found that only several *enums* are necessary. Also, a software developer could potentially break this guideline by casting an integer variable as an enumerated type. That coding practice should be avoided because the synthesis tool can no longer determine all possible values of a parameter used in such a way.

2.6 Loop Rerolling (LR)

In some situations, software developers manually unroll a loop several times to expose concurrency. While this practice may seem unusual due to unrolling being a well-known compiler transformation, not all compilers unroll loops – hence, we have observed that manual unrolling is a widespread practice. Manual unrolling may improve software performance, but may result in extra area overhead, especially when synthesis is not resource constrained, in addition to greatly increasing synthesis execution times. The *loop rerolling* refinement fixes the problems with manual loop unrolling, converting the loop back into a non-unrolled representation. Loop rerolling has a software performance overhead only if the compiler fails to unroll.

Automation: Loop rerolling has been automated in previous approaches [16] and could be added to a synthesis tool. However, because rerolling requires minimal effort by the software developer, we propose loop rerolling as a guideline to ensure good results on any synthesis tool.

2.7 Conversion to Explicit Control Flow (CECF)

Synthesis tools can rarely statically analyze control flow in the presence of function pointers because the target of a function call using a function pointer is not known until runtime. To fix this problem, we applied *conversion to explicit control flow* to eliminate function pointers from critical regions of code. This guideline works by replacing a function call using a function pointer with a series of if-else statements, where the body of each if statement is an explicit call to one of the possible targets of the function pointer.

Automation: Compilers and synthesis tools can potentially automate this guideline by tracking targets of function pointers through control-flow analysis. However, control-flow analysis techniques are only likely to work under simple situations because in the general case, statically determining all possible targets of a function pointer is impossible [15].

Overhead: The software overhead of conversion to explicit control flow consists of the extra instructions required to implement the if statements. Using input enumeration or function specialization, which is discussed in section 2.8, can reduce this overhead.

2.8 Function Specialization (FS)

In [17], we proposed *function specialization* as a guideline for writing C for synthesis. Function specialization is applied when a parameter to a function has a frequent value, which can then be treated as a constant and optimized. This guideline is similar to input enumeration, except that instead of only having to support a small set of inputs, function specialization optimizes a function for a particular input value but still supports any possible input.

Automation: Function specialization can be automated by performing profiling to determine values for each parameter. Although profiling is common to determine execution time of each function, profiling to determine possible values for each parameter is much less common. Also, for large applications, the amount of data that has to be profiled is large. However, the software developer may be aware of several common values. Therefore, we include function specialization as a guideline to make this approach applicable to the majority of software and synthesis tool flows.

Overhead: Function specialization has a software overhead consisting of extra instructions to determine which function to call at runtime. However, if applied to a frequent input value, function specialization typically improves software execution in addition to hardware execution.

2.9 Algorithmic Specialization (AS)

Algorithmic specialization replaces a software algorithm with an algorithm that is more appropriate for hardware, possibly achieving order of magnitude improvements. However, the software overhead of algorithmic specialization is potentially large because this guideline replaces a software efficient algorithm with a less efficient algorithm. Applying this guideline requires more consideration than the other guidelines, which is discussed further in section 3.

Automation: Algorithmic specialization cannot be automated by a synthesis tool because no tool is currently able to replace one

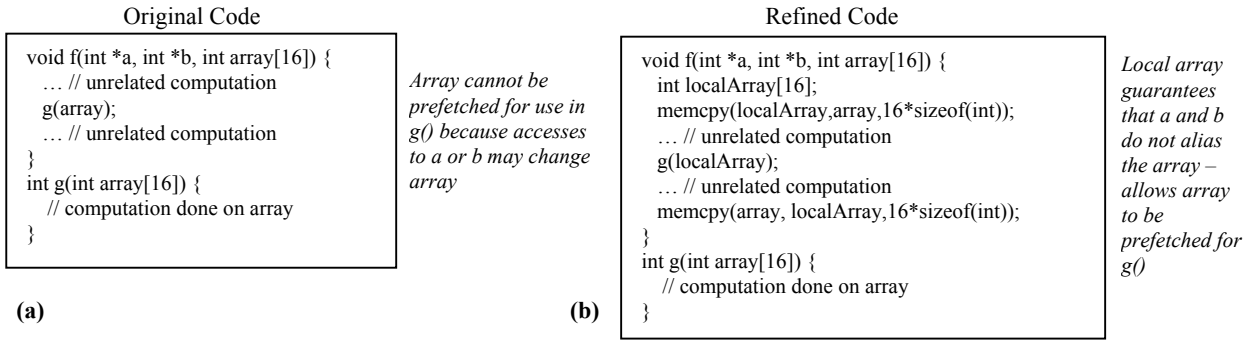

Figure 2: *Pass-by-value return* for arrays. a) The array used in function g() cannot be prefetched during f() because the pointers a and b may modify elements of the array. b) Copying the array into a local array guarantees that a and b are not aliases, allowing the array to be prefetched into on-chip memory before execution of g().

algorithm with another without using a higher-level specification such as intrinsics.

2.10 Pass-By-Value Return (PVR)

Main memory bandwidth is commonly the main performance bottleneck for synthesized hardware. In many situations, loops can potentially be unrolled and parallelized, but because data cannot be delivered fast enough the parallelism is wasted. To eliminate this problem, synthesis will prefetch the required data and store the data in on-chip memory to increase memory bandwidth. However, the common use of pointers in C code makes the determination of when to prefetch data without violating dependencies difficult. An example of this problem is shown in Figure 2(a). In this example, an array is passed to function f() which is then used by function g(). Theoretically, the data in the array can be preteched immediately upon execution of f(). However, because pointers are passed to f(), the values in the array may be modified through these pointers.

In many cases, a software developer may know that aliases do not exist. In this situation, a software developer can apply *pass-by-value return* for arrays by declaring a local array within the calling function and manually copying all data from the array parameter into the local array. The local array is then used for all computation. The advantage of this guideline is that the local array eliminates all aliasing problems because the pointer parameters cannot possibly alias the local array.

Figure 2(b) shows the example from Figure 2(a) after applying pass-by-value return. The refined code immediately copies data from *array* into *localArray* at the beginning of function f(). Synthesis can now prefetch data starting at this point so that the data is available when g() is called. After execution of g(), the data is copied back from *localArray* into *array*. These copying operations are eliminated during synthesis and are simply used to eliminate aliases.

Automation: This guideline is unlikely to be automated because most synthesis tools cannot perform the alias analysis necessary to guarantee that the array can be prefetched. Of course, improved alias analysis could eliminate the need for this guideline. However, global alias analysis is difficult and in the general case is impossible [15].

Overhead: Pass-by-value return can potentially have a large software overhead if the amount of data in the copied array is large. However, in many situations these arrays will be small, such as in DSP code that operates on small blocks of data. Also, in many situations the overhead of copying data can be amortized over many functions that use the same data.

As an alternative to pass-by-value return, a designer could use a compiler directive to specify the non-existence of aliases. However, this practice makes code compiler-specific, reducing the portability of the code.

3. REFINEMENT METHODOLOGY

Although the proposed coding guidelines may greatly improve hardware performance, the guidelines may introduce software performance overhead if the refined regions are not implemented in hardware. In addition, the guidelines must be manually applied to the code, which could potentially greatly increase design time if applied to too many regions. We therefore propose a methodology for applying the guidelines that seeks to minimize the effort and overhead.

To develop the methodology, we analyzed benchmarks to determine which guidelines were applicable and how the guidelines would best be applied. We found that for most examples, most speedup could be obtained by only rewriting a small number of performance critical regions, corresponding to the well-known 90-10 rule. Rewriting those regions is generally a simple task and can yield large performance improvements. Exceptions exist, such as the h.264 decoder from [17], which spent 90% of execution time in more than 50 functions. However, for most benchmarks we observed, refining just a few critical regions results in large performance improvements. In addition, most of the guidelines take only several minutes to implement. The exceptions are conversion-to-fixed point, which may require modifying a large section of code, and algorithmic specialization, which may be difficult to perform in the absence of a well-known hardware algorithm.

Figure 3 illustrates the proposed methodology for refining C code for synthesis. The first step of the methodology is to profile the application to determine critical regions where the majority of execution time is spent. The next step is to select the most critical region, generally a function or loop, to which to apply the guidelines. To optimize a function or loop, the guidelines may have to be applied to the containing function, or possibly even to an earlier function. The next step is to apply conversion to constants, conversion to fixed point, conversion to explicit data flow, conversion to explicit control flow, conversion to explicit memory accesses, constant input enumeration, function specialization, and loop rerolling, but only if these guidelines are

applicable for the application. Although this step may seem to require a large amount of designer effort, in most cases, only two or three of these guidelines are applicable. Also, in almost all cases, the software overhead of these guidelines is small, allowing them to be safely applied without impacting software performance. Before implementing pass-by-value return, the software developer first needs to determine the software overhead. To determine the overhead, the developer analyzes the critical region to determine the data used by the region. If aliases prevent prefetching of data for the critical region and the overhead of copying the data in software is acceptable (which may have to be determined by profiling), then the software developer should implement pass-by-value return for the data. The next step of the methodology is to consider whether or not to perform algorithmic specialization. If an appropriate hardware algorithm exists for the current critical region and the performance of the hardware algorithm is sufficient in software (which also may have to be determined with profiling), then the software developer should perform algorithmic specialization. In the next step, the software developer selects another critical region and repeats the same steps until the most critical regions are rewritten.

Of course, if an entire application is to be synthesized or if the designer can select synthesized functions, then software overhead can be ignored and these guidelines can be applied to as many regions as possible.

As an added note, to achieve the performance benefits of the guidelines that deal with improving data prefetching, a software developer should apply the guideline as far before the data is needed as possible to guarantee that there is sufficient time to prefetch the data. This may not always be possible, based on the dependencies of the code.

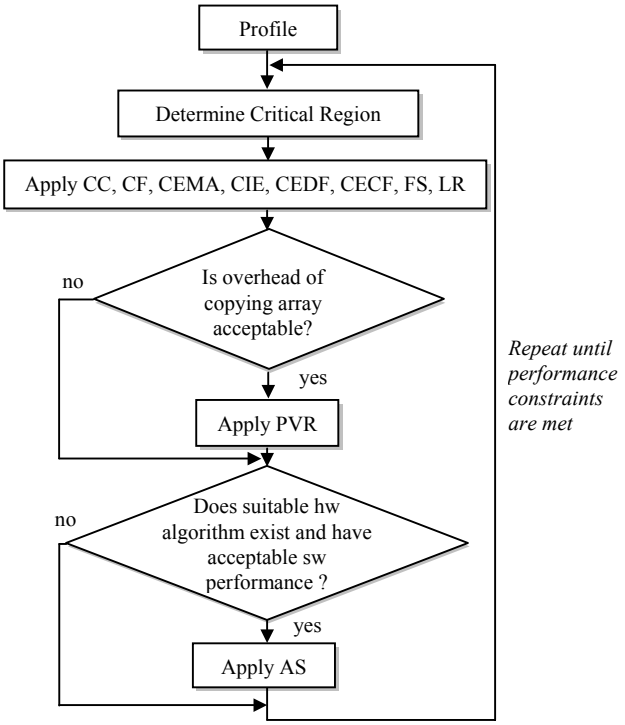

Figure 3: Methodology for applying coding guidelines to achieve hardware performance improvements while minimizing software overhead.

4. EXPERIMENTS

To evaluate the coding guidelines, we applied the guidelines to six benchmarks, using the methodology described in section 3. *G3fax* is a group 3 fax decode benchmark. *Mpeg2* is an mpeg2 decoder. *Jpeg* is a jpeg decoder. *Brev* performs a bit reversal of each element from an array. *Fir* is a finite impulse response filter. *Crc* performs a cyclic redundancy check. *G3fax*, *brev*, *fir*, and *crc* are from the Powerstone [11] benchmark suite. *Mpeg2* and *jpeg* are from the MediaBench [10] benchmark suite. We selected these benchmarks because they are representative of high-level applications, such as DSP code and bit manipulation code, which are commonly synthesized.

For all examples, we measured software performance using the SimpleScalar simulator, ported to the ARM instruction set. Each example was compiled using gcc with –O1 optimizations.

We selected regions to synthesize to hardware by utilizing a greedy hardware/software partitioning heuristic. The heuristic first sorts each function and loop based on the percentage of execution time. The heuristic then selects regions for hardware implementation in sorted order until area is exhausted. We could of course have utilized existing hardware/software partitioning techniques [8], but for the selected examples, the greedy heuristic achieves similar results.

To synthesize hardware, we considered using existing C-level synthesis tools [5], but because of the variety and relative immaturity of these tools, we developed our own behavioral synthesis tools to have more control over the synthesized hardware, and then performed standard behavioral synthesis optimizations manually to prevent the benefits of the guidelines from being exaggerated – our goal was to make the hardware synthesized from the original code as fast as possible. To perform synthesis, we developed a parser to convert the code into a control/data flow graph representation. The tool then performed data flow analysis to determine potential parallelism. Next, we applied standard synthesis optimization techniques such as loop unrolling, strength reduction, constant propagation, tree-height reduction, etc. After optimizing the control/data flow graph, the synthesis tool applied as-soon-as-possible scheduling to obtain the lowest cycle latency for the control/data flow graph. Whenever possible, we prefetched data from main memory into on-chip memory to increase memory bandwidth. After scheduling each operation, the synthesis tool created a controller and datapath, at which point we could determine the cycle latency for each region. We were able to manually optimize the control/data flow graph because we only had to analyze a small number of regions and those regions generally only consisted of a few dozen lines of code.

The target architecture in our experiments was a microprocessor/FPGA platform having an ARM9 microprocessor running at 200 MHz and a Xilinx Virtex II FPGA running at 100 MHz. Communication between the FPGA and ARM9 was implemented using shared memory. The FPGA accessed memory using a DMA.

Figure 4 shows the speedup of a C-based hardware/software solution compared to a software-only solution both without and with the coding guidelines. Without the coding guidelines, the speedups typically ranged from 1x (no speedup) to 2x, though

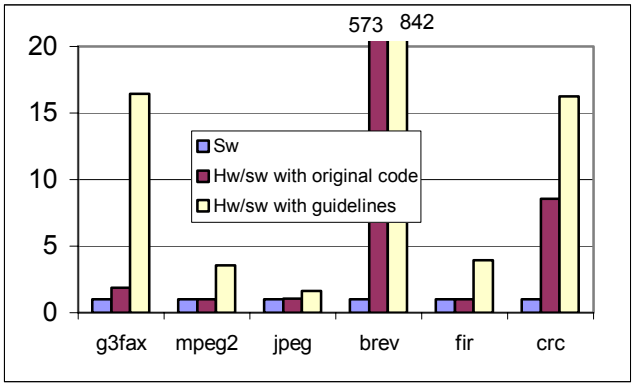

Figure 4: Speedups of hardware/software solution compared to software both without and with coding guidelines.

brev and *crc* yielded speedups of 573x and 8.5x. With the guidelines, the speedups were significantly larger. The speedup of *g3fax* increased from approximately 2x to 16x. The speedup of *mpeg2* increased from 1x to 3.4x. The speedup of *fir* and *jpeg* increased from 1x to 4x and 1.6x, respectively. Even though *brev* and *crc* already achieved significant speedups, the guidelines increased these speedups even further to 842x for *brev* and 16.7x for *crc*. On average, the guidelines resulted in a partitioned system that was 3.5x faster than the partitioned system created from the original code.

Figure 5 shows the overall software performance and size overhead (in % of lines of C code) from applying the guidelines for each example. For *g3fax* and *mpeg2*, there was a small 4%-5% software performance overhead. For *jpeg*, the guidelines caused almost no overhead. For *brev*, *fir*, and *crc*, the software performance actually improved because of function specialization. The size overhead ranged from almost no overhead for *mpeg2* and *jpeg*, to 28% for *crc*, and –47% for *brev*. *Crc* experienced a large increase because the original code only consisted of 143 lines of code, to which we added 40 lines. *Brev* actually was smaller after being refined, because the use of a specialized algorithm was more concise than the original code. On average, the guidelines required only 23 lines of additional code.

Applying these guidelines took approximately between one and two hours for each example. Most of the guidelines required little analysis of the code. Pass-by-value return, conversion to fixed point, and algorithmic specialization required the most time.

We attempted to determine the contribution to overall speedup for each individual guideline. However, we found that in many cases each individual guideline provided no speedup, but when combined with other guidelines resulted in a large speedup. For *g3fax*, we applied conversion to constants to convert several global variables whose values never changed into constants. Because these variables were used as bounds for a critical loop, after also applying pass-by-value return, these guidelines allowed for the loop to be unrolled, resulting in increased speedups of 3.6x compared to 2x without the guidelines. We also applied conversion to explicit data flow to eliminate other global variables, and algorithmic specialization to optimize the reading of a bit, increasing the speedup to 16x.

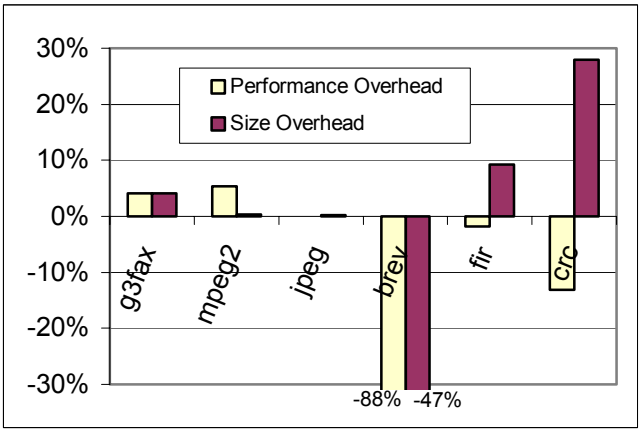

Figure 5: Software performance and size overhead of the code refinement guidelines, when hardware/software partitioning is *not* performed. In several cases, software performance and size improved and are shown as *negative* overhead.

For *mpeg2*, we applied conversion to fixed point, conversion to constants for an array of IDCT coefficients, and pass-by-value return so that the hardware could prefetch blocks to be decoded. These three guidelines allowed synthesis to unroll critical loops that were part of the IDCT code, resulting in a speedup increase from 1x to 3.4x. Without either conversion to constants or pass-by-value return, the speedup would have been 2.3x. Without conversion to fixed point, there would have been no speedup.

For *jpeg*, we applied conversion to constants, conversion to explicit memory accesses, conversion to explicit control flow, function specialization, and pass-by-value return. These guidelines allowed us to parallelize much of the IDCT and dequantization code in the JPEG decoder, resulting in a speedup increase from 1.1x to 1.6x. Without any of these guidelines, the hardware would have achieved a speedup of 1.1x. The lack of a larger speedup was caused by other critical functions that could not be synthesized to efficient hardware, suggesting that other guidelines may be necessary.

For *brev*, we applied loop rerolling to eliminate manual unrolling that had been performed. We also performed algorithmic specialization to perform a bit reversal using a series of logic operations. In this case, algorithmic specialization resulted in an increased speedup of 842x compared to 573x, and also improved software performance by 88%. Loop rerolling did not have an effect on performance because we completely unrolled the loop during synthesis. However, in other situations, loop rerolling could result in a speedup of over 2x [16].

For *fir*, we applied conversion to constants for an array of filter coefficients, conversion to fixed point, conversion to explicit memory accesses, and function specialization, which were all necessary in order to unroll a critical loop within the FIR filter, resulting in a speedup increase from 1x to 4x.

For *crc*, we applied conversion to constants and input enumeration, which allowed for synthesis to unroll a loop and to create a specialized version of the most critical function, resulting in an increase in speedup from 8.6x to 16.7x. Without input enumeration the speedup was 14.6x. Without conversion to constants, the speedup was 12.2x. We also considered applying algorithmic specialization so that we could perform the crc using

xor operations. Algorithmic specialization did slightly increase the overall speedup to 19x compared to 16.7x, but had a software overhead of over 6000%. By following the methodology in section 3, we decided to exclude algorithmic specialization.

5. CONCLUSIONS

Although synthesis from C code has made many advances, the use of standard software constructs and coding styles commonly results in inefficient hardware. To deal with this problem, we proposed a code refinement approach that requires little design effort and results in minimal software overhead. In this approach, the designer first profiles the code to determine critical regions, and then applies simple refinement guidelines to make those regions more amenable to hardware synthesis tools. Those critical regions generally correspond to a small number of loops, which can be rewritten in 1-2 hours, resulting in hardware/software partitioned applications that ran 3.5x faster than when partitioning was done on the original unrefined code.

Due to the common use of hardware-inappropriate C code, one may wonder if more appropriate languages should be utilized instead of applying coding guidelines. While new languages will certainly provide better technical results, the widespread use of C code in embedded systems suggests that a C-based synthesis approach is likely to be widely accepted even with worse hardware performance. The proposed refinement approach helps reduce this performance difference with minimal designer effort. Although the refinement approach does require the design time overhead of modifying the code, for many designers this overhead is likely much less than the overhead of learning new languages.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Böhm, W., Hammes, J., Draper, B., Chawathe, M., Ross, C., Rinker, R., and Najjar, W. Mapping a single assignment programming language to reconfigurable systems. *The Journal of Supercomputing*, Vol. 21, 2002, 117-130.

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

[3] Coding guidelines for datapath synthesis. www.synopsys.com/products/designware/dwtb/articles/coding_guidelines/coding_guidelines.html

[4] Gokhale, M.B. and Stone, J.M. NAPA C: compiling for a hybrid RISC/FPGA architecture. *In proceedings of IEEE symposium on FPGAs for custom computing machines (FCCM)*, 1998, 126-135.

[5] Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. *In proceeding of international conference on VLSI Design*, 2003, 461-466.

[6] Fin, A., Fummi, F., and Signoretto, M. SystemC: a homogenous environment to test embedded systems. *In proceedings of the international symposium on hardware/software codesign (CODES)*, 2001, 17-22.

[7] Frigo, J., Gokhale, M., and Lavenier, D. Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. *In proceedings of the international symposium on field programmable gate arrays (FPGA)*, 2001, 134-140.

[8] Kalavade, A. and Lee, E. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. *In proceedings of the international symposium on hardware/software codesign (CODES)*, 1994, 42-48.

[9] Ku, D. and DeMicheli, G. HardwareC -- a language for hardware design (version 2.0). *Technical Report: CSL-TR-90-419*, *Stanford University*, 1990.

[10] Lee, C., Potkonjak, M., and Mangione-Smith, W. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. *In proceedings of international symposium on microarchitecture* (MICRO), 1997, 330-335.

[11] Malik, A., Moyer, B., and Cermak, D. A low power unified cache architecture providing power and performance flexibility. *In proceedings of international symposium on low power electronics and design (ISLPED)*, 2000, 241-243.

[12] Najjar, W., Böhm, W., Draper, B., Hammes, J., Rinker, R., Beveridge, R., Chawathe, M., and Ross, C. From algorithms to hardware – a high-level language abstraction for reconfigurable computing. *IEEE Computer*, Vol. 36, Issue 8, August 2003, 63-69.

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

[14] Roy, S. and Banerjee, P. High-level techniques for signal processing: an algorithm for converting floating-point computations to fixed-point in MATLAB based FPGA design. *In proceedings of the design automation conference (DAC)*, 2004.

[15] Séméria, L. and De Micheli, G. SpC: synthesis of pointers in C. *In proceedings of the international conference on computer-aided design (ICCAD)*, 1998, 8-12.

[16] Stitt, G. and Vahid, F. New decompilation techniques for binary-level co-processor generation. *In proceedings of the international conference on computer-aided design (ICCAD),* 2005.

[17] Stitt, G., Vahid, F., McGregor, G., and Einloth, B. Hardware/software partitioning of software binaries: a case study of h.264 decode. *In proceedings of international conference on hardware/software codesign and system synthesis (CODES/ISSS)*, 2005, 285-290.