

DWT Design Exploration via ROCCC

Benjamin Arai Conley Read

University of California, Riverside

Computer Science

{barai, cread}@cs.ucr.edu

Abstract

The speedup associated with software to FPGA (Field Programmable Gate Array) synthesization shows great improvement over pure processed software applications. The integration of software and FPGA microprocessor driven applications offers great possibilities. One problem with using pure FPGA driven applications is that the large source base may require more space than available on present FPGAs. In addition, the time required to program the FPGA based application is much greater than the time required to program an identical processed software implementation of the same application.

In this paper we evaluate the speedup that may be achieved by profiling an existing software implementation of the JPEG2000 still image compression algorithm and simulating replacements of terminal code in high frequency use source code execution paths with FPGA microprocessor implementations generated by ROCCC (Riverside Optimizing Compiler for Configurable Computing). In order to convert the high level software implementation to an FPGA based architecture, we use ROCCC. Our goal is to extract and convert the frequent execution paths in source code to FPGA leveraging parallelization and optimization, increasing overall performance, exploiting the unique properties of FPGAs.

1 Introduction

As the complexity of image compression algorithms increase, the speed of microprocessors need to increase in order to achieve the performance required for real-time decoding. In order to do this, hardware implementations are required. Not only do software implementations come short of reaching the performance required; they are often no cost effective. In order to combat this issue hardware implementations of algorithms have been created to speed up the performance of encoding and decoding image information. These methods, although fast, are not always viable due to algorithm space complexity, which may be larger than the available space on an FPGA. In order to address this issue we abstract the application at code level.

The first abstraction is the non-essential code, code which is not frequently used by the applications. This type of code

includes initialization information and other infrequently used instantiation. Then we identify the frequently executed portion of code. This type of code is found highly isolated in compression algorithms such as the JPEG2000. We exploit this code division by extracting high use code from the original code base and converting it to VHDL, hence circuits, for the FPGA.

2 Motivation

Design and space exploration of FPGA based hardware is an area of important research due to recent advances in our ability to create cheaper and faster FPGAs. Thus, it is becoming ever more important to automate methods of FPGA code creation for optimizing CPU and memory intensive applications and algorithms that find the general CPU environment an inefficient space. ROCCC minimizes the high cost of hand writing VHDL code for FPGAs and allows developers to focus on creating applications in high level languages like C and C++; using ROCCC for the compilation and linking of profiled highly executed code on the FPGA.

We explore design speedups and space requirements of using ROCCC for the JPEG2000 algorithm. We have chosen to explore the JPEG2000 algorithm for two main reasons. First, JPEG2000 is an emerging technology which solves many problems associated compression methods used by its predecessors. By attempting to optimize JPEG2000, we are creating a more optimized solution for the computationally intensive JPEG2000 algorithm. This improved performance will allow its migration into new and upgraded technologies with superior cost benefits in the near future.

The most notable use of JPEG2000 in new technology is the introduction of H.264 video format, which encodes video using the JPEG2000 technology [14]. This illustrates the importance of creating a high performance implementation of JPEG2000. This format must be capable of successfully decoding near thirty frames per second of compressed image data in order to keep up with motion video. This proposition proves computationally intensive when we consider the demands of High Definition video.

3 Overview of JPEG2000

The original JPEG standard, though efficient, is now almost a decade old. Unfortunately, the original JPEG standard does

not scale in a way required by modern imaging applications. JPEG2000 is an IEEE standard that takes advantage of advances in compression technologies and addresses many of the shortcomings associated with the original JPEG standard. For example, the new standard is better equipped to benefit modern imaging applications such as digital photography and mobile imaging [1].

JPEG2000 contains a host of improvements over its predecessor including yet not limited to the following:

- Superior low bit-rate performance
- Continues-tone and bi-level compression
- Lossless and lossy compression
- Progressive transmission by pixel accuracy and resolution
- Region-of-interest (ROI) coding
- Open architecture
- Robustness to bit errors
- Protective image security

These features are necessary for fulfilling the requirements of today’s emerging applications [1]. Ideally, we want to leverage FPGA architecture and parallelism capabilities as much as possible to improve the performance of these new features.

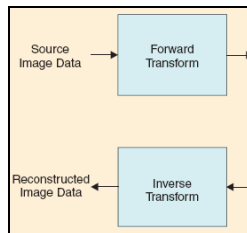


Figure 1 – Simple Image Reconstruction in JPEG2000 [1]

The JPEG2000 standard involves transformation including quantization, and encoding/decoding. We will be focusing on the transformation portion of the JPEG2000 standard. The algorithm first breaks an image into tiles, then for each tile, performs the transformation. The tiling refers to the breaking up of the image in to n non-overlapping sections. The tiling phase is used to reduce memory requirements of the algorithm. Each of the tiles is decomposed into levels of resolution. Once the DWT (discrete wavelet transformation) has been performed on each of the tiles, the algorithm moves the quantization phase. We will focus on the DWT portion only because of the intensive code execution as a result of the tiling [1]. Since the DWT execution can be broken up into an arbitrary number of tiles, we can exploit parallelism using the FPGA.

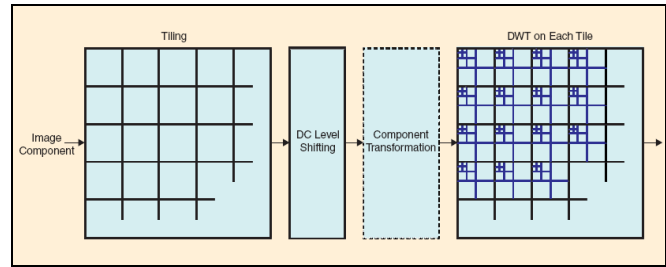


Figure 2 – Tiling and DWT in JPEG2000 [1]

As shown in figure 2, the original image is broken up into an $n \times n$ matrix of tiles. Then, the DWT is performed on each of the individual tiles independent of each other.

Each of the stages that we identify in section 5 is performed on an image independently. The tiling of the image serves to lower the overall memory requirements of the algorithm. The sizes of the individual tiles affect the visual quality of the image. Smaller tiles, though low in memory cost do create a visible tiling effect on the image [1]. The relationship of tile size, parallelization, and memory requirements are tightly coupled in the JPEG2000 algorithm performance and contribute to image quality.

The JPEG 2000 algorithm is a new standard, which provides an outline for current and future compression schemes focusing on scalable, high-quality image compression [1]. This high level of compression comes at the cost of increase computation, therefore further optimization on both the software and hardware realm are required.

4 Overview of ROCCC

ROCCC, also known as the Riverside Optimizing Compiler for Configurable Computing is a compiler developed for converting C and C++ code to VHDL. ROCCC provides a progressively more automated method to convert C and C++ code into optimized VHDL.

ROCCC distinguishes itself from other compilers by focusing on optimization of performance and size. These two factors play an important role in creating a competitive VHDL source code generator. In addition, the design time required to produce hand crafted VHDL code is up to twice as long as the comparable high level implementation. Therefore, ROCCC is an important time saving tool for testing and implementing partially embedded applications [5].

The ROCCC implementation focuses on three categories of optimization including loop transformation – optimal parallelization, storage optimization, and optimization of circuit expression. The goal of the loop transformation is to maximize parallelism while not increasing the required circuit area. Storage optimization attempts to minimize the number of fetches from memory by the FPGA. These optimizations combined are used to create VHDL output [5].

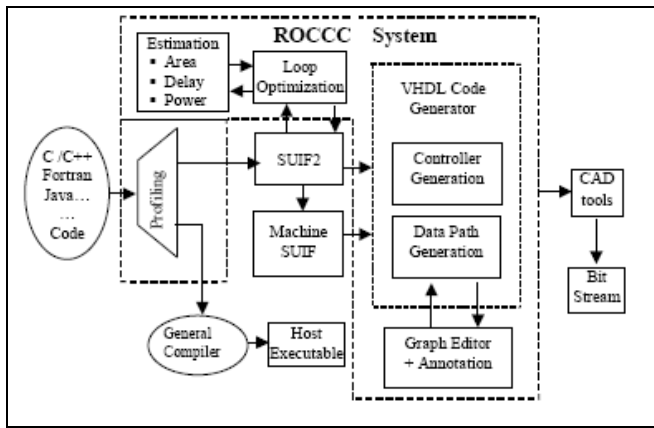


Figure 3 – ROCCC System Overview [4]

The goal of ROCCC is not to compile entire applications to VHDL but to instead compile the highly executed divisions of a program and gain the speedup of a highly specialized ASIC. The compiler relies on developers' use of profiling tools to identify frequently executed code in the application.

5 Algorithm Profiling

In order to localize the areas of highest execution, we employed a multi-level profiling system for both coarse and fine grained profiling. The first profiling method performs a profile of the JPEG2000 algorithm by separating it into components and finally profiling the components, reporting statistics for both. The second stage profiles the identified code portion, which contains the highest level of execution. This allows us to locate both the highly traversed primary execution path and the code modules that are executed most often. The JPEG2000 algorithm is first broken up into coarse grained pieces based on the different algorithms internal to JPEG2000.

5.1 JPEG2000 Profiling

We have broken up the algorithm into six main parts. The results below are based upon execution time of a JPEG2000 C and C++ implementation provided by the University of California, Irvine [15].

Division	Norm.	% of Exec.
Tile Initialization:	0.02	1.57%
DC Level Shifting:	0.09	7.08%
Component Transformation:	0.04	3.15%
Wavelet Transformation:	0.581	45.71%
Arithmetic Coding (T1):	0.53	41.70%
Rate Control (T2):	0.01	0.79%
Tile Encode Total:	1.251	98.43%
Grand Total:	1.271	100.00%

Figure 4 – JPEG2000 Profiling Summary

The sample image used was a 16x32 pixel, 24-bit, bitmap image. As shown on the table in Figure 4, the DWT portion of the JPEG2000 takes the most significant amount of execution time.

5.2 DWT Profiling

Now that we have identified a candidate for performance improvement, the area of greatest execution time, we attempt to profile the chosen area of highest execution in order to get a fine grained localized portion of code to optimize.

For profiling the identified code, we have chosen to use Microsoft Visual Studio Team Edition 2005. This suite of tools is required for fine grained profiling of unmanaged C and C++ code in an existing Visual Studio project. Function level profiling for the JPEG2000 source code is as follows:

Function Name	Exclusive Percent	Inclusive Percent
xJ1D_FILTD	24.479	35.455
_ftol2_pentium4	10.141	10.141
_int_abs	7.594	7.594
_fix_mul	4.86	4.86
xJ2D_DEINTERLEAVE	4.432	4.446
t1_enc_refpass_step	4.321	10.141
t1_enc_clnpass_step	4.183	9.126
DC_LS	3.935	5.012
xHOR_SD	3.845	23.754
t1_enc_sigpass_step	3.569	3.707
xJ1D_EXTD	3.486	3.541
xFDWT	3.058	53.528
_int_max	2.389	2.389
MQ_ENCODE	2.278	2.34
FICT	2.264	2.264
xVER_SD	1.961	22.27
t1_enc_clnpass	1.319	11.521
t1_getctxno_mag	1.291	1.291
xDC_Level_Shifting	1.077	1.077
...

Figure 5 –Function Profiling Summary

Functions beginning with an underscore on the table in Figure 4 are requisite library calls and are not part of the JPEG2000 implementation. The exclusive percent is the percent of clocks that the CPU spent processing that function only. The inclusive percent is a statistic of the percent of clocks that the CPU spent processing that function or a function called from that scope or deeper.

The function with the highest exclusive percentage is the function xJ1D_FILTD. This is the function that performs the JPEG2000 DWT tile filtering algorithm. The code within this function consists of a case statement for each filter type to be executed on a tile, which is then computed and returned. The function is executed often because of the n tiles created for the DWT execution.

We break up the code at two levels. The first Function Level, is coarse grained and identifies functions that may benefit from compilation to FPGA. Due to the constraints of both the compiler and, more important, space limitations on the FPGA, we further break the selected code into areas of highest execution in Loop Level Filtering. This allows us to exploit not only the areas of highest execution but also to optimize benefits considering the size of the synthesized code on the FPGA.

The two point approach we provide is a solid framework for automating the conversion of high level implementations to synthesized partially embedded solutions. The profiling technique provides both a scalable and integratable solution for most software requirements.

6 Related Work

Several high level language translators have been implemented using various languages including Verilog, VHDL, and C and C++. Some of these compilers include SA-C [6, 7], Streams-C [8], GARP's [9], and SPARK [10]. All of these implementations have target hardware languages but all of them fail to offer the level of optimization and integration provided by ROCCC.

7 Method

Once the division of C and C++ code has been chosen for compilation using ROCCC, we can measure performance by running the DWT division of the JPEG2000 compression algorithm using only the compiled C and C++ source code. We then compare the performance of the software algorithm with the modified ROCCC compiled replacements. This will allow us to compare both the performance relative to the embedded DWT and to the entire execution time of the application.

We have extracted the frequently executed division of the DWT code, the function `xJ1D_FILTD`, and encapsulated it for execution on the FPGA. This is done to allow ROCCC to create an image which can be used on a single FPGA. The selected source code is also modified to remove C and C++ features that are incompatible with ROCCC.

If the source code being replaced with ROCCC code contains various pointer references to arrays and other reference to incompatible memory, these references will be replaced with reference based code to ensure compatibility.

7.1 Code Parallelism

The performance gained by transferring the most highly executed path of an algorithm is obtained in two main areas. The first gain is in the area of FPGA generation. The fact that an FPGA generated for a particular application can complete in a single cycle the same amount of work that a general CPU could take thousands of cycles to complete. Therefore, even without disregarding the slower clock speed of a FPGA, the algorithm can complete at a much faster rate than the

completely general CPU based applications. The second gain is the performance improvement associated with execution parallelism [12, 13]. The ability to parallelize execution is crucial for software driven applications, but only easily implemented in hardware based applications. Sometimes this performance benefit is essential to justify extra hardware cost and development. We enable this optimization through compilation with ROCCC on a simple FPGA. The ability to parallelize an application using hardware based FPGA's can achieve unparalleled speedups with complete execution times equal to the time a single un-parallelized block requires to complete in software.

We address parallelization on two levels. Our initial step is to locate areas of high repetitive execution. If a division of code is found which meets these requirements then we move to compilation with ROCCC. Otherwise, the algorithm can still benefit from FPGA based execution but the expected gains are lower. Once we have found a code candidate then we proceed by first examining the code and searching for areas of parallelizable execution. In the piece of code we have chosen there are many areas of potential parallelization such as the "for" loops which compute each of the pixel values.

Parallelization can also be the source of possible bottlenecks. If a piece of code is very parallelizable but is part of a pipeline where one stage in the execution path is extended, then the parallelization may be useless because the optimized division of the algorithm in hardware will continually wait for additional data.

8 Experiments

After solving initial problems compiling the JPEG2000 implementation application provided by the Irvine team, we began evaluating the primary path of execution of the implementation for a candidate to compile with ROCCC.

8.1 Irvine Implementation Issues

Initial problems with the Irvine implementation were solved by hard coding source and destination image data for conversion. The initial problems appeared to be mainly due to platform assumptions.

After initial testing and simple troubleshooting we identified that the implementation appears to support input images of any size, however the unedited application faults on image input of size greater than 16x32 pixels of 24bit BMP images.

We leave unexplored the actual application crash which limits the output size of images for JPEG2000 conversion. The fault occurs on data flush of the transformed bitmap image. As a workaround, we set a breakpoint in `my_bmp.c` at line 134 to enable reporting of results for the Irvine simplified C implementation.

8.2 Empirical Profiling JPEG2000

Fortunately for our uses, the DWT code is not a limiting factor. The fDWT runs correctly on images larger than the 16x32 pixel, 1:2 ratio, 24bit BMP image and appears highly scalable.

Our next problem is to locate the primary path to image tile transformation. The following, Figure 6, includes the path through the source code to the implementation of the fDWT (Forward Discrete Wavelet Transformation) for JPEG2000. We will show the process we used to identify the most executed code divisions and the steps we used to modify the identified code for compilation with ROCCC.

File	Function	Depth
My_bmp.c:	main	0
My_jp2k.c:	MyJP2	1
My_jp2k.c:	jp2k_encode	2
My_jp2k.c:	jp2k_write_sod	3
My_tile_enc.c:	tile_encode	4
My_xfdwt.c:	xFDWT	5
My_xfdwt.c:	xJ2D_SD	6
My_xfdwt.c:	xHOR_SD	7
My_xfdwt.c:	xJ1D_SD	8
My_xfdwt.c:	xJ1D_FILTD	9

Figure 6 – JPEG2000 Primary Execution Path

For profiling and code coverage we use the Microsoft Visual Studio Team System 2005 to generate Caller-Callee and Call-Tree statistics at the function layer of abstraction. This information allows us to identify the paths of execution of which the primary path is noted Figure 6.

After breaking up the JPEG2000 implementation into each stage of the algorithm, we look for functions along the path of execution that can benefit from synthesizing with ROCCC. At this stage we begin design exploration to evaluate tradeoffs of selecting a division of code for synthesizing. These tradeoffs include data memory space requirements, circuit space requirements, and synthesized circuit depth impact.

We select the division from function xJ1D_FILTD for a number of reasons that include the fact that this function can be easily flattened by in-lining the functions that it calls from its scope. Also, xJ1D_FILTD is the most executed division of the JPEG2000 DWT. You can see in Figure 7, from section 8.4 that the filter stage of the DWT accounts for 5136 clocks of 7754 total clocks spent in the DWT.

We told you earlier that the DWT was the most executed division of the JPEG2000 algorithm. To give you even more perspective, with reference to actual clocks spent execution code on a general CPU, the DWT accounts for 7754 clocks of 14482 total clocks spent executing the main function of JPEG2000! Main even includes such code as initialization impact, which you may reference in Figure 4.

8.3 Compiling with ROCCC

At this point in the development of ROCCC the entire compilation process is not yet automated. However, this point is not meant to identify a weakness, but only to acknowledge a fact which may not be obvious as we begin to describe the typical compilation.

ROCCC, Figure 3, is currently implemented in a two step process: High level code optimization and VHDL circuit generation. These two steps of the process can be fused to create a single automated step, but is currently counter productive due to limitations of both steps. These limitations are related to currently unsupported high-level code in the optimization step and memory management in the synthesizing step.

To successfully compile the DWT division with ROCCC we must make a series of modifications to the chosen xJ1D_FILTD high level code while maintaining semantic correctness.

Modification requirements for ROCCC optimization pass:

- Convert high level C++ code to C strict
- Convert while loops to for loops
- Convert to integer or fixed point
- Remove type casting
- Convert pointers to array notation
- Fuse multiple loops into single for loop
 - ROCCC will do this in future version

Modification requirements for ROCCC synthesizing pass:

- ROCCC can create one circuit per loop
 - Loop fusion requirement
- Array indexes must be i or i+1
- Bit shifts operate on immediate constant
 - FPGA may shift differently than CPU
- Reading array must be different from writing array
- Loop control variables must be pre-computed

After these modifications, the high level code will compile to FPGA. Currently, this process can prove tedious when editing existing, highly abstracted code. Fortunately, during profiling we observed that divisions that benefit most from synthesizing appear in code almost un-abstracted due to existing algorithm and development constraints. We postulate that this is the general case.

8.4 Performance Improvement

At this time, physical results are not available from simulation of a hardware implementation of our generated DWT xJ1D_FILTD circuit. This performance improvement analysis begins to evaluate design exploration tradeoffs, but much more is left for future evaluation. ROCCC has proven to be an excellent design-space exploration tool.

Function	Inclusive	Exclusive
main	14482	0
xFDWT	7754	443
xJ2D_SD	7311	0
xJ1D_SD	5826	82
xJ1D_FILTD	5136	3546/0*

* We flatten xJ1D_FILTD as part of the modifications we make to compile the division with ROCCC.

Figure 7 – DWT Execution Path

From previous work we estimate that our circuit will run at approximately 40 – 50 MHz. We exploit the benefits of partial embedding here. Once we have a circuit for a particular parallelizable application, we can put as many of these on our FPGA as benefit us. In the best case we can decide to put the maximum number of DWT filter circuits on the FPGA and gain the maximum speedup.

Function	Parallelizable Calls
tile_init	7
xFDWT	7
xJ1D_FILTD	113

Figure 8 – DWT / filter Parallelizable calls

Time on 1GHz* CPU

$$CPUtime = 14482cycles \times \frac{1sec}{266Mcycles} = 5.4 \times 10^{-5} sec$$

Using FPGA with multiple DWT filters

$$CPUtime = 9346cycles \times \frac{1sec}{266Mcycles} = 3.5 \times 10^{-5} sec$$

$$FPGAtime = 1cycles \times \frac{1sec}{45Mcycles} = 2.2 \times 10^{-8} sec$$

$$TOTALtime = 3.5 \times 10^{-5} sec + 2.2 \times 10^{-8} sec = 3.5 \times 10^{-5} sec$$

* We limit the CPU to 266 MHz memory bandwidth for JPEG2000

Figure 9 – Possible Speedups from Compilation

In our example, a 16x32 pixel BMP image, we have created 7 tiles. The DWT division is completely parallelizable, so with 7 tiles, we can execute all DWTs in 1 cycle on a FPGA imaged with 7 or more DWT circuits.

Similarly, we can parallelize our DWT filter execution with an FPGA imaged with multiple circuits in the same way. In our case, utilizing an FPGA image of greater than 16 DWT filter circuits ensures that no stalls occur between filters on each tile and that execution completes in a single cycle.

In the worst case, we complete execution of all 113 filters in 113 cycles, or in 7 cycles on a FPGA with multiple images of the circuit. In the best case, we can run the entire DWT on each of the 7 tiles using multiple images of the DWT circuit to complete execution in a single cycle using the modified

DWT from the JPEG2000 implementation [15], or using the UCR DWT [16] for comparison.

9 Conclusion

Our method is highly scalable allowing us to improve the performance of code divisions implemented in a high level language. In Figure 9 we show that the performance improvement is significant. Our research shows that the speedup gained by the compilation to FPGA is definitely worth the time taken to modify the original code to that supported by ROCCC. We found multiple candidates for compilation with ROCCC that afford us immediate performance and parallelization opportunities.

We have shown that the two point approach to profiling and analysis can identify divisions that are most suited to modification. We have described the benefits and current limitations for ROCCC. Our exploration shows that at even the current ROCCC version, the benefits of synthesizing for parallelization outweigh the difficulty of code modification.

As the ROCCC implementation grows more mature, design-space exploration will become even easier.

10 Future Work

The speedup gained using our optimized hardware approach, offers promising results for not only the JPEG2000 algorithm, but also any algorithm that is parallelizable and contains areas of high repetitive execution.

In future work we hope to continue our design exploration of the additional stages of the JPEG2000 algorithm. This includes the remaining tiers of quantization and entropy coding. We venture that in these additional segments of the JPEG2000 algorithm we will see similar speedups and be able to better gauge and predict the speedups and space requirements to create a variety of optimized partially embedded solutions.

References

- [1] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG2000 Still Image Compression Standard", *IEEE Signal Processing Magazine* 36-58, September 2001.
- [2] W. Najjar, W. Böhm, B. Draper, J. Hammes, R. Rinker, R. Beveridge, M. Chawathe and C. Ross. *From Algorithms to Hardware – A High-Level Language Abstraction for Reconfigurable Computing*. *IEEE Computer*, August 2003.
- [3] Z. Guo, W. Najjar, F. Vahid and K. Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs over Processors, *Symp. on Field-Programmable gate Arrays (FPGA)*, Monterey, CA, February 2004.
- [4] Z. Guo, A. B. Buyukkurt and W. Najjar. Input Data Reuse In Compiling Window Operations Onto Reconfigurable Hardware, *Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES 2004)*, Washington DC, June 2004.

- [5] ROCCC, Riverside Optimizing Compiler for Configurable Computing. <http://roccc.cs.ucr.edu> 2005.
- [6] W. Najjar, W. Böhm, B. Draper, J. Hammes, R. Rinker, R. Beveridge, M. Chawathe and C. Ross. From Algorithms to Hardware - A High-Level Language Abstraction for Reconfigurable Computing. *IEEE Computer*, August 2003.
- [7] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems, *The Journal of Supercomputing*, Volume 21, pages 117-130, 2002
- [8] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Lalinowski. Stream-oriented FPGA computing in the Streams-C high level language. *IEEE Symp. on FPGAs for Custom Computing Machines (FCCM)*, 2000.
- [9] T. J. Callahan, J. R. Hauser, J. Wawrzynek. The Garp Architecture and C Compiler. *IEEE Computer*, April 2000.
- [10] SPARK. <http://www.cecs.uci.edu/~spark/> 2004.
- [11] M. Boliek, C. Christopoulos, and Eric Majani, JPEG2000 Part I Final Committee Draft Version 1.0, April 2000.
- [12] X. Wang and S. Ziavras. Parallel Direct Solution of Linear Equations on FPGA-Based Machines. *Symposium-IPDPS2003*, Nice, France, April 22-26, 2003.
- [13] X. Wang and S.G. Ziavras. Performance Optimization of an FPGA-Based Configurable Multiprocessor for Matrix Operations. *Proc. of IEEE International Conference on Field-Programmable Technology (FPT'03)*, Tokyo, Japan, Dec.15-17, 2003.
- [14] T. Halbach and M. Wien. Concepts and Performance of Next-Generation Video Compression Standardization. *Proc. Nordic Signal Processing Symposium (NORSIG)*, Hurdigruten (Norway), October 2002.
- [15] JPEG2000 Implementation. Chunhui Zhang. University of California, Irvine.
- [16] UCR DWT Implementation. Zhi Guo. University of California, Riverside.