Compiler Optimization for Configurable Accelerators

Betul Buyukkurt
University of California Riverside
Computer Science Department
Riverside, CA 92521
abuyukku@cs.ucr.edu

Zhi Guo
University of California Riverside
Electrical Engineering Department
Riverside, CA 92521
zguo@cs.ucr.edu

Walid A. Najjar
University of California Riverside
Computer Science Department
Riverside, CA 92521
najjar@cs.ucr.edu

ABSTRACT

ROCCC (Riverside Optimizing Configurable Computing Compiler) is an optimizing C to HDL compiler targeting FPGA and CSOC (Configurable System On a Chip) architectures. ROCCC system is built on the SUIF-MACHSUIF compiler infrastructure. Our system first identifies frequently executed kernel loops inside programs and then compiles them to VHDL after optimizing the kernels to make best use of FPGA resources. This paper presents an overview of the ROCCC project as well as optimizations performed inside the ROCCC compiler.

1. INTRODUCTION

FPGAs (Field Programmable Gate Array) can boost software performance due to the large amount of parallelism they offer. The area on the FPGAs can be configured to form the data path or a register file as the software necessitates. The only limiting factors to FPGA performance are the available area and the I/O pins. On the other hand, programming an FPGA is a very tedious task since FPGAs do not specify any control or data path mechanism or a fixed instruction set.

Researchers have studied mapping from high-level languages to HDLs. However, there is still not yet a single tool that is good enough to be adapted by the designers. ROCCC (Riverside Optimizing Configurable Computing Compiler) is an optimizing C to VHDL compiler targeting FPGA and CSOC platforms. ROCCC identifies, optimizes and parallelizes the most frequently executed kernel loops in applications such as multimedia and scientific computing.

ROCCC is built over SUIF2-MACHSUIF infrastructure. SUIF2 front end is easily extendible to support multiple source languages. SUIF2 also provides both high and low level IR (Intermediate Representation) necessary to support all high-level constructs found in most common languages such as C, Fortran, JAVA etc. On the other hand, MACHSUIF is a compiler infrastructure for building the back end of a compiler.

ROCCC performs both loop level transformations and circuit level optimizations. It first performs control and data flow analysis over its IR followed by global optimizations such as constant propagation, constant folding, dead code eliminations that enables other optimizations. Then, loop level optimizations such as loop unrolling, loop invariant code motion, loop fusion and other such transforms are applied.

This paper is organized as follows. Next section discusses on the related work. Section 3 gives an in depth description of the ROCCC system. SUIF2 level optimizations are described in Section 4 followed by the compilation done at the MACHSUIF level in Section 5. Section 6 mentions our early results. Finally Section 7 concludes the paper.

2. RELATED WORK

There are several projects and publications on translating C or other high level languages to different HDLs. Streams-C[7], Handel-C[8] and SA-C[6] generate HDL from C-variants, where the programmer is either in charge of specifying explicitly where the parallelism is such as in Streams-C or writing the application in this C-variant form or both.

Nimble[15] and GARP[13] target only specific hardware architectures. They generate specific configuration files instead of standard VHDL file that can be synthesized to readily available FPGA boards.

SPARK[14] generates VHDL from a subset of regular ANSI-C. It applies extensive front and back end optimizations such as loop unrolling/fusion, loop invariant code motion, common sub expression elimination, dead code elimination, etc. The two main objectives of SPARK are to reduce the number of states in the controller FSM and the cycles on the longest path. However, SPARK does not support two-dimensional arrays, neither performs optimizations on the data reuse which occurs regularly in image processing, DSP and scientific computing algorithms.

3. PROJECT OVERVIEW

ROCCC is a compiler tool that maps the most frequently executed regions of software written in a high level language such as C, onto configurable hardware. ROCCC is built over the SUIF2-MACHSUIF infrastructure. The main reason behind building our tool over an existing framework is code reuse. We mostly benefited from the

front end, IR and the auxiliary data structures/passes provided in these tools. ROCCC's main contributions on the other hand are its extensive optimizations and the data-transfer/execution model on the FPGA.

Not all of the C constructs are supported. The source program should not include pointers, floating point arithmetic nor any irregular control flow such as breaks and exits inside the source code. The code may contain function calls, however all the function calls are inlined prior to analysis and transformation passes.

The type of source programs that would benefit from ROCCC's optimizations the most are the programs that would apply the same operations to each and every element of the input data. Image processing algorithms, DSP computations, scientific computation programs would be our best candidates since they are regular in their computation and easily parallelizable.

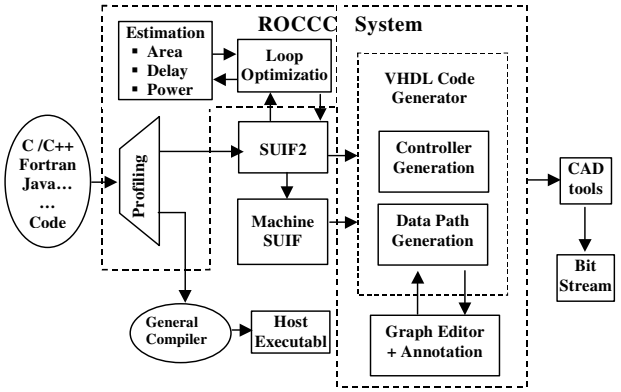

Figure 1 – ROCCC System Overview

Figure-1 shows an overview diagram of ROCCC system. Firstly, the source program is fed into a profiling tool. ROCCC uses profiler results to locate the most frequently executed kernels. The extracted software kernels then are fed into the SUIF2 environment, which would then analyze the code and apply global and loop level transformations. In SUIF2 we work on high-level IR as opposed to low-level IR as traditional compilers do. We eliminated the need for any additional coding due to working on high-level IR, by utilizing the auxiliary data structures in SUIF2, such as walkers, visitors, etc. which enabled us easily deal with various collections of IR classes at once. After all SUIF2 level transformations are completed, ROCCC decouples memory accesses from the computation through scalar replacement. SUIF2 then extracts the code that is going to form the data path on the FPGA in a file to MACHSUIF.

In our project we benefited from MACHSUIF's Control Flow Graph (CFG), Data Flow Analysis and the Static Single Assignment libraries. In MACHSUIF, we hoist independent instructions and assign them to pipeline stages.

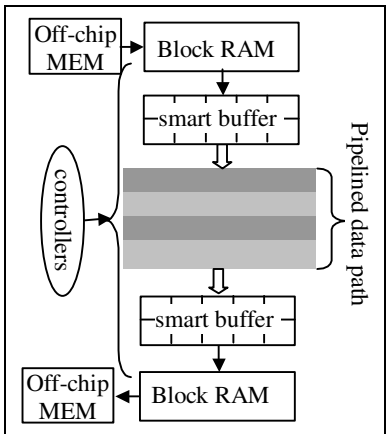

Figure 2 - The Execution Model

Then we map from MACHSUIF opcodes to IEEE 1076.3 VHDL library elements. All MACHSUIF opcodes except division have a corresponding functionality in the IEEE 1076.3 VHDL library. We extended MACHSUIF with few other opcodes to support for feedback variables and a multiplexor.

The Controller Generator module in Figure-1 would collect the array read/write operations to generate the address generators and smart buffers on the FPGA. Figure-2 depicts the execution model built on the FPGA by our compiler.

4. SUIF2 LEVEL OPTIMIZATIONS

ROCCC transformations mainly focus on loops operating on arrays. Prior to applying our transformation passes, control flow analysis and data flow analysis passes are executed over the SUIF IR. We developed our own control and scalar data flow analysis passes in SUIF2. These passes are built in an iterative fashion. Since ROCCC is not a production compiler we are currently running these analysis routines after every transformation that changes the IR.

4.1 Procedure Level Optimizations

The following passes are completed and being used in our compiler system.

- Constant Folding: This pass folds constants, simplifies algebraic identities such as multiplication by zero or division by one. The pass operates over the IR of the entire procedure until no more simplification occurs.
- Constant Propagation: This pass replaces the uses of variables whose values are constants by the constant values themselves. This pass works hand in hand with the constant folding pass, since one generates further optimization opportunities for the other.
- Division by constant approximation: Division in hardware is a costly operation and would consume large areas on FPGA's. This pass transforms the division into a sum of dividend right shifted by various powers of 2 that approximates to the constant divider. An example of this transformation is shown in Figure 3, where the sum_of_9 / 9 is replaced by a sequence of right shifts and additions.
- Copy Propagation: This pass eliminates the copy operations inserted by other transformations.

- Dead and Unreachable Code Elimination: Dead code elimination removes all code that has no contribution in computing the result or the return value of the program. Whereas unreachable code elimination eliminates blocks where control would never reach due to other transformations over the IR.
- Common Sub Expression Elimination: This pass eliminates exactly matching expressions or expression sub trees over the entire program.
- Propagation of Constant Arrays: It is common in image processing or DSP applications that some constant mask or array of constants is applied to the input data over and over again. These masks are usually applied to input data in loops with known at compile time constant bounds. ROCCC fully unrolls innermost loops with constant bounds. Then wherever the mask indices turn into a constant, ROCCC replaces the reference to the constant array with the constant itself.

4.2 Loop Level Optimizations

Loop level optimizations are the transformations, which help achieve most of the parallelism in ROCCC. Although most of these transformations are same as traditional compiler optimizations, the scale of the optimizations would differ. Since, traditional compilers target processors with a defined ISA and a rigid data path, unrolling by small amounts would provide enough parallelism to keep the processor busy. However on an FPGA as long as there is enough area and I/O bandwidth to fit another iteration, the loop can be unrolled for another time.

Moreover, in traditional processors any computation has to follow the fetch, decode, execute, write back cycle which additionally imposes memory access pattern restrictions on an optimizing compiler. On an FPGA none of these restrictions would hold, however due to the growing scale of the optimizations new challenges arise such as the transfer of data to the data path and synchronization of data flow across pipeline iterations incase of any backward dataflow as in feedback variables. ROCCC performs the following loop level optimizations.

- Loop Normalization: This pass rearranges the loop headers so that the indices of the loops start from one and incremented by one after an iteration completes.
- Invariant Code Motion: This pass hoists any statement/expression that computes the same value across iterations of the loops outside of the loop.
- Loop peeling: Loop peeling helps remove first or last few iterations of a loop by copying its body before or after the loop as many times as the loop is peeled. This operation helps bring the iteration count of a loop to a desired value enabling other optimizations such as loop fusion. In addition this pass can be used to remove dependencies that only bound the first or last few iterations of a loop.
- Loop Unrolling: Loop unrolling is the main compiler technique that allows reconfigurable architectures to achieve large degrees of parallelism. ROCCC has two unrolling passes. One of the two passes is used to fully unroll loops with known constant upper bounds. The other unroll pass unrolls a given loop at a given unroll factor. The unrolling factor can be a user-defined value as well as a value computed, based on the memory and the area limitations of the FPGA. At this time we are assuming that ROCCC gets the unrolling factor from the user, however once we incorporate our area estimation tool, we predict that ROCCC will be able to determine the unroll factor based on the available area on the FPGA using the feedback from the estimation tool.

  Figure-3 shows moving filter code that is unrolled twice by our compiler. As seen in the unrolled code there are common sub expressions appearing between the different iterations of the loop. These expressions appear in different places in their respective expression trees. As the scale of unrolling grows, such partial expressions would appear in many other expressions whose trees are different and similarity varies. There are many variations to CSE in the literature such as CSE, partial CSE, value-numbering etc. each targeting different, but overlapping cases of CSE occurrences in programs and dealt at the low level IR. We have not found CSE algorithms except [16] addressing the CSE problem shown in Figure-3. Such common sub expression problems are ROCCC's targeted optimizations.
- Loop Fusion: This pass combines the bodies of two consecutive loops under the header of one of the loops. In order to be able to fuse two loops, following conditions must hold. The iteration counts of the loops should be the same. The loops should not contain statements that would form a dependency cycle between the statements of the two loops after they are fused. For instance, the first loop is reading array A and writing array B while the second loop is reading B but writing back on A, then when these two loops are fused the new body would form a dependence cycle that perhaps did not exist prior to the transformation. At this point though our fusion algorithm is assuming both loops do not form new dependencies, when they are fused. Loop peeling transformation can be used to bring the iteration counts of any two consecutive loops, if the iteration counts of the loops are apart by a constant amount.
- Loop Unswitching: This optimization transforms for-loops having only an if-statement in its body, whose condition test is a loop invariant, into an if-statement whose then and else statements are enclosed in a for statement. This pass helps reduce unnecessary control flow from loop bodies. ROCCC can apply this pass to a nest of if-statements, if all if conditions are loop invariants.

```
void main(){
    int sum_of_9, i;
    int A[256], X[256];
    for(i = 0; i < 247; i=i+1) {
        sum_of_9 = A[i] + A[i+1] + A[i+2] + A[i+3]
+ A[i+4] + A[i+5] + A[i+6] + A[i+7] + A[i+8];
        X[i] = sum_of_9 / 9;
    }
}
```

(a) Moving filter 9

```
void main(){
    int  i;
    __ar_2 A;
    __ar_2 X;
    int  sum_of_9;

    for(i = 0;i<247- 1; i = i+1+1) {
        sum_of_9    =    A[i]+A[i+1]+A[i+2]+A[i+3]+
A[i+4]+A[i+5]+A[i+6]+A[i+7]+A[i+8];
        X[i] =sum_of_9/ 9;
        sum_of_9 = A[i+1]+A[(i+1)+1]+A[(i+1)+2]))+
A[(i+1)+3]+A[(i+1)+4]+A[(i+1)+5]+A[(i+1)+6]+A[(i+1
)+7]+A[(i+1)+8];
        X[(i+1)] =sum_of_9/ 9;
    }
}
```

(b) Moving filter after being unrolled twice

```
void main(){
    int  i;
    __ar_2 A;
    __ar_2 X;
    int  sum_of_9;

    for(i = 0; i < 246; i = 2+i) {
        sum_of_9 = A[i]+A[1+i]+A[2+i]+A[3+i]+
A[4+i]+A[5+i]+A[6+i]+A[7+i]+A[8+i];
        X[i]  =  (sum_of_9>>  12)+(sum_of_9>>  11)
+(sum_of_9>>  10)+(sum_of_9>>  6)+(sum_of_9>>  5)
+(sum_of_9>>4);
        sum_of_9 = A[1+i]+A[2+i]+A[3+i]+A[4+i]
+A[5+i]+A[6+i]+A[7+i]+A[8+i]+A[9+i];
        X[1+i]  =(sum_of_9>>  12)+(sum_of_9>>  11)
+(sum_of_9>> 10)+(sum_of_9>> 6)+(sum_of_9>> 5)
+(sum_of_9>> 4);
    }
}
```

(c) Moving filter code after constant folding and division by constant elimination passes

```
    for(i = 0; i < 246; i = 2+i) {
        A0 = A[i]; A1 = A[1+i]; A2 = A[2+i];
        A3= A[3+i]; A4 = A[4+i]; A5 = A[5+i];
        A6 = A[6+i]; A7 = A[7+i]; A8 = A[8+i];
        A9 = A[9+i];
        sum_of_9 = A0+A1+A2+A3+A4+A5+A6+A7+A8;
        TO  =  (sum_of_9>>  12)+(sum_of_9>>  11)
+(sum_of_9>>  10)+(sum_of_9>>  6)+(sum_of_9>>  5)
+(sum_of_9>>4);
        sum_of_9 = A1+A2+A3+A4+A5+A6+A7+A8+A9;
        T1  =  (sum_of_9>>  12)+(sum_of_9>>  11)
+(sum_of_9>> 10)+(sum_of_9>> 6)+(sum_of_9>> 5)
+(sum_of_9>> 4);
        X[i] = T0;
        X[1+i] = T1;
    }
```

(d) Moving filter loop after scalar replacement

Figure 3 - Moving filter code after ROCCC transformations

However, this pass should be used with care, since the condition variables, which are loop invariants from the software analysis view, could be hardware signals modified by external sources as the loop executes. To remedy this situation and still be able to benefit from this pass, users are advised to use the C keyword volatile to express that a variable should be read from the memory each time it is accessed. Then our data flow analysis library would not mark it as loop invariant.

4.3 SUIF2 to MACHSUIF Data Path Generation

In SUIF2 we apply all the passes in high-level IR as opposed to low-level IR as traditional compilers do. One of the justifications for traditional compilers' working on low-level IR is that once the high level IR is lowered close to the level of ISA some of the optimizations can get lost. For example, new common sub expression elimination opportunities may arise due to array address calculation arithmetic, strength reduction or operator substitutions due to underlying ISA not supporting the operator existing in the high-level language. However, in our case we're bounded by neither an ISA nor memory. Rather we're faced with configuring bare hardware where a variable is a value not a memory reference.

After we apply all our passes, we ran other passes to prepare our output to be processed by our MACHSUIF backend.

- Scalar Replacement: This pass helps isolate memory references from computation by moving all memory read/write operations to the beginning/end of the loop body. The values of the memory reads are saved into automatically generated scalar temporaries prior to the first statement in the original loop body. Then the array references in the original loop body are replaced with the generated scalar temporaries holding array values. Finally, any values that are to be saved into an array cell locations are saved from scalar temporaries back into array locations after the last statement inside the original loop body.

  Scalar replacement decouples array references from the computation in loop bodies and control generator uses its results to form the read and write buffers. These two buffers are placed before and after the pipelined execution unit on the FPGA.

- Scalar I/O Detection: The loop outputs are not always arrays. They could be scalars computing a sum or a max or min of an array that has to be communicated to the software running on the host processor. This pass marks those variables whose values are initialized outside the loop body, modified within the loop and referenced by the host executable after the loop finished its execution. Such variables are marked and communicated to the

MACHSUIF environment so that they may be saved for use later by the host executable.

- Feedback Variable Detection: There could also be other scalar variables whose values are incremented or updated per iteration of the loop body such as a scalar variable holding a sum over an array. This pass identifies those variables whose values are dependent upon their past values. Although the execution model of ROCCC necessitates that every so many cycles new values are read and written, ROCCC supports the case of feedback variables by saving them in registers generated on the data path to be referenced by following pipeline cycles.

4.4 Smart Buffer

Smart buffer [10] is a custom generated buffer aiming at maximizing data reuse. It acts as a buffer between the Block RAM on the FPGA board and the data path. Therefore, there are two of them placed at the entry and the exit of the data path circuitry. The data that is going to be stored in the smart buffer is determined by the scalar replacement phase of our compiler. The scalar replacement phase shifts the memory read/write operations to the two ends of the loop body. The addresses for these decoupled memory references are generated by the Address Generator to be brought into the Block RAM.

Block RAM is a piece of SRAM, which can be configured to provide the data using various numbers of ports at will. Since the number of ports to both the Smart Buffer and the Datapath are application-specific, for ROCCC only the bandwidth with the external memory can be the main limitation to parallelism.

Smart buffer organizes the data that is received from the memory in windows. Most applications in multimedia and high performance scientific computing process input data by sliding a window over it. In window operations, a portion of the window data, as it is sliding over the input array, is always shared across successive iterations of the loop body.

Each window has its own control logic enabling when and which sets of windows are to be exported to the data path. To illustrate, if the not unrolled 5-tap-FIR in Figure-4 has a smart buffer whose size is seven words, then, if the loop stride is one, every five words constitutes a window. Thus, this seven-word smart buffer contains seven windows. In Figure 5 (a), (b) and (c) shows the No. zero, No. one and No. six windows. However, at any clock cycle at most one window's data is valid. The inactive words in the smart buffer could be receiving new data while the active window is sending its data to the data path. To illustrate better, Figure 6 shows smart buffer for the twice unrolled 5-tap-FIR. If we say the smart buffer size is two-word larger than the not-unrolled one, then, since and the loop stride is two, the size of each window is now six-word. Therefore inside the buffer in Figure 6, there are only five windows in total. Although the buffer size increased a bit, the number of windows decreases and so does the control logic.

```
for (i=0; i<N; i=i+1) {
     C[i] = 3*A[i] + 5*A[i+1] + 7*A[i+2] +
     9*A[i+3] - A[i+4];
}
```

Figure 4 - A 5-tap FIR in C

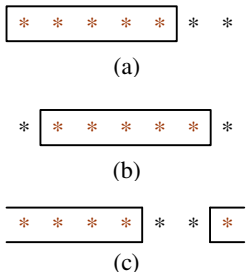

Figure 5 – Windows No.0, No.1 and No.6 in the smart buffer for the not unrolled 5-tap-FIR

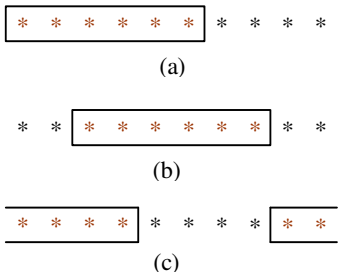

Figure 6 – Windows No.0, No.1 and No.4 in the smart buffer for the twice-unrolled 5 –tap-FIR

5. MACHSUIF LEVEL OPTIMIZATIONS

Once ROCCC passes are executed over the SUIF2 IR, the output is then sent to MACHSUIF environment. We extended the MACHSUIF IR with three new opcodes to describe MUX (multiplexor), LPR (load previous) and SNX (store to next) operations.

We execute the then and else branches of if-statements in parallel. MUX opcode is introduced to copy the output signals of the alternative branches of if-statements to the next common successor node. Our approach of executing the then and the else branches of if-statements in parallel is completely safe; since no memory is written until the pipeline is over. The LPR and SNX are used in MACHSUIF to indicate that the load operation is loading from and storing to a feedback variable respectively. When translated to VHDL these opcodes ensure that the read and write operations occur from/to a hardware register on the FPGA to hold the value of the feedback variable across iterations.

5.1 Pipelined Datapath Generation

ROCCC, due to its pipelined execution, is able to instantiate a new iteration every cycle if there are no preventing loop carried dependencies across iterations. The Datapath Generation Module part of our VHDL code generator first builds the control flow graph over the MACHSUIF IR and analyzes for data dependencies. Then the module hoists instructions that are not data or control dependent on the earlier instructions as high as possible within the local control block and every instruction is assigned a location in the hardware data path. Instructions that are grouped together are executed in parallel. ROCCC automatically places latches in a data path to pipeline it.

The alternate branches of if-statements are also executed in parallel. [9] describes the data path generation in ROCCC in detail.

6. CONCLUSION & FUTURE WORK

This paper gives an overview and update on the current status of the ROCCC project. However, the project is currently functional but yet far from completion.

We are currently developing our data dependence analysis and common sub expression elimination passes. Array data dependence analysis, in its generalized form, is an NP complete problem. Thus, we started building our analysis for a restricted version of the problem. Our data dependence analysis code at this time aimed at extracting dependence information in perfectly nested loops and for array subscripts that are linear functions of the loop indices. The transformations we seek benefit from our dependence analysis tool are currently loop fusion and loop interchange. We are also in the process of developing a common sub expression elimination pass that would eliminate the occurrences of sub expression cases elaborated on in Section 4.2.

Another future work is to integrating our profiler [12] and adding an accurate resource estimation module into the current ROCCC system and update necessary ends to bring all modules shown in Figure-1 to full functionality.

7. REFERENCES

[1] *SUIF Compiler System.* http://suif.stanford.edu, 2004

[2] Machine SUIF. http://www.eecs.harvard.edu/hube/research/machsuif.html, 2004

[3] G. Holloway and M. D. Smith. Machine SUIF Control Flow Graph Library. Division of Engineering and Applied Sciences, Harvard University 2002.

[4] G. Holloway and A. Dimock. The Machine SUIF Bit-Vector Data-Flow-Analysis Library. Division of Engineering and Applied Sciences, Harvard University 2002.

[5] G. Holloway. The Machine-SUIF Static Single Assignment Library. Division of Engineering and Applied Sciences, Harvard University 2002.

[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] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Lalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In IEEE Symp. on FPGAs for Custom Computing Machines (FCCM), 2000.

[8] Handel-C Language Overview. Celoxica, Inc. http://www.celoxica.com, 2004.

[9] Z. Guo, B. Buyukkurt, W. Najjar and K. Vissers. Optimized Generation of Data-Path from C Codes. In ACM/IEEE Design Automation and Test Europe (DATE), Munich, Germany, March 2005.

[10] Z. Guo, 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.

[11] Z. Guo, W. Najjar, F. Vahid and K. Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs over Processors, In. Symp. on Field-Programmable gate Arrays (FPGA), Monterrey, CA, February 2004.

[12] D. C. Suresh, W. A. Najjar J. Villareal, G. Stitt and F. Vahid. Profiling Tools for Hardware/Software Partitioning of Embedded Applications, Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES 2003), San Diego, CA, June 2003.

[13] T. J. Callahan, J. R. Hauser, J. Wawrzynek. The Garp Architecture and C Compiler. IEEE Computer, April 2000.

[14] SPARK. http://www.cecs.uci.edu/~spark/ 2004

[15] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In Design Automation Conf.(DAC), 1999.

[16] P. Briggs and K. D. Cooper, Effective Partial Redundancy Elimination, Proceedings of the SIGPLAN 94 Conference on Programming Language Design and Implementation, SIGPLAN Notices 29(6), June 1994, pages 159-170.