Automatic Customization of Embedded Applications for Enhanced Performance and Reduced Power using Optimizing Compiler Techniques[1]

Emre Özer, Andy P. Nisbet, and David Gregg

Department of Computer Science
Trinity College, Dublin, Ireland
E-mails: {emre.ozer, andy.nisbet, david.gregg}@cs.tcd.ie

Abstract. *This paper introduces a compiler framework that optimizes embedded applications written in C, and produces high-level hardware descriptions of the applications for customization on Field-Programmable Gate Arrays (FPGAs). Our compiler performs machine-specific and machine-independent optimizations in order to increase the performance of an embedded application and reduce area/power requirements without explicit programmer intervention. Our experimental results show that our compiler framework can increase performance by 38% with loop and expression parallelism for eight embedded benchmarks. Also, area usage and power consumption are reduced by 69% and 55%, respectively through the efficient utilization of on-chip FPGA resources for* ***Xilinx*** *Virtex-II FPGA chip.*

1. Introduction

The attraction of using FPGAs is the ability to generate application specific logic where the balance and mix of functional units can be altered. This has the potential to generate orders of magnitude speedup for computationally intensive algorithms. However, the time taken to develop and optimize an application for execution on an FPGA using a traditional hardware description language (HDL) design flow may be prohibitive.

Our goal is to develop a compilation framework (see **Fig. 1**) that takes an application written in *ANSI-C*, translates it into an intermediate representation (IR) for performing optimizations, and produces *Handel-C* code. *Handel-C* [12] is a high-level HDL language that can reduce both the time and the expertise required to develop embedded FPGA applications. However, the programmer must explicitly specify code parallelism, the storage type (memory or registers), operand bit-widths, resource sharing or private resource allocation using special language constructs. The detection and exploitation of parallelism in an application and explicit specification of operand bit-widths and decisions about how to allocate resources are tedious, error-prone tasks which can be largely automated.

[1] The research described in this paper is supported by an Enterprise Ireland Research Innovation Fund Grant IF/2002/035.

Our compilation framework is built upon the *Stanford SUIF1* compiler [8]. It performs several architecture-independent optimizations as well as architecture-specific optimizations on the *SUIF IR* such as loop parallelization, expression scheduling, shared hardware structuring, operand bit-width specifications, the efficient utilization of on-chip RAM and ROM, multi-ported RAM and dedicated multipliers in the target FPGA chip. The final step of our compiler is to generate *Handel-C* code from the optimized IR. The *Handel-C* compiler reads the generated *Handel-C* code and produces *EDIF netlists* for **Xilinx Virtex-II XC2V6000** FPGA chip [13].

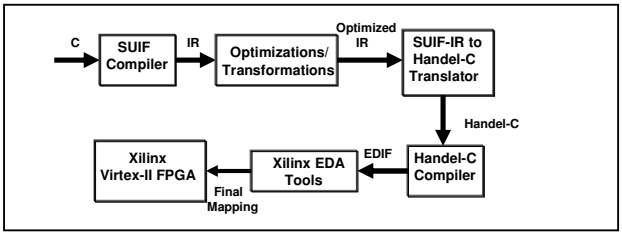

Fig. 1. Our compilation framework

The organization of the paper is as follows: *Section 2* explains compiler optimizations for improving performance and analyzes performance results. *Section 3* presents optimizations and experimental results of area and power consumption. *Section 4* discusses the related work in hardware compilation, and finally, *Section 5* concludes the paper with a discussion of future work.

2. Optimizations for Performance

This section introduces compiler optimizations to increase the performance of embedded applications by taking advantage of the *Handel-C* parallelism constructs. Each assignment statement in *Handel-C* takes exactly one clock-cycle to execute. The operand reads of variables are initiated at the start of a clock cycle and a write to the destination variable is finalized by the end of the clock cycle. This means that it is permissible to have multiple reads and a single write to a variable occurring in the same clock cycle.

2.1. Loop Iteration Parallelization

Handel-C has three different loop constructs: *for*, *par* and *seq*. Each construct has the same syntax:

for|par|seq(*loop_index*=LOWER_BOUND;*loop_index*<|>|=|!=UPPER_BOUND;*step*)
{Statements}

The way that the *for* loop in *Handel-C* works is the same as in C, i.e. the loop is executed by the amount of the loop trip count. In a *par* loop, every iteration and every statement in the iteration is executed in parallel. On the other hand, the *seq* loop fully unrolls or replicates the loop and executes the iterations sequentially. Loop-carried

data dependency analysis can determine if loop iterations are independent and suitable for parallel execution. A *for* loop whose iterations are independent can be replaced by a *Handel-C par* construct. Note that the statements in the loop body are by default put into a *seq* construct in our system and will execute sequentially. For example, the C sample code on the left-hand side can be parallelized using *Handel-C* as shown on the right-hand side below.

```
/* C sample for loop */
for (i = 0; i<10; i++) {
        a[i] = b[i] + c[i];
        d[i] = e[i] * f[i];
}
```

```
/* Handel-C par loop */
par (i = 0; i<10; i++){
        seq {
            a[i] = b[i] + c[i];
            d[i] = e[i] * f[i];
        }
}
```

The sample loop has independent iterations since there are no data-carried dependences between iterations. The loop can be transformed into a *par* loop, and the loop body is conservatively contained in a *seq* statement that guarantees that the two statements execute sequentially. In this example, the two statements have no data dependency and can run simultaneously. However, our parallel expression scheduling algorithm described in the next section can detect this case and replace the *seq* with a *par* construct.

The loops that cannot be parallelized can be fully unrolled by using a *seq* loop although this may be of little benefit. The *Handel-C* compiler places each iteration sequentially, and separate hardware is synthesized for each iteration in the final FPGA. The advantage of fully unrolling is that there is no need for a loop increment and comparison operations. The disadvantage is the amount of FPGA area consumed by unrolled iterations. If area is of concern, then the *for* statement should not be replaced by a *seq* statement. The logic created by a *for* statement consists of the loop body and the loop test. It is important to note that the loop bounds must be known in order to translate a *for* into a *par* or a *seq*. Other loops such as *do..while* and *while* are translated into *par* and *seq* loops if they can be transformed into a canonical loop form.

2.2. Parallel Expression Scheduling

Expressions having no true data dependencies amongst them can be executed in the same cycle by enclosing them in a *par* block. A variable can be read multiple times and assigned to at most once in a *par* block. For instance, the three statements on the left-hand side of the following code are data-independent and can execute in parallel using a *Handel-C par* statement as shown on the right-hand side. All three statements in this *par* block will execute in a single cycle, whereas it takes three cycles to execute the code if no *par* is used since a statement in *Handel-C* takes a cycle to execute.

```
A = B * C;
D = E + F * G;
H = M;
```

```
par {
    A = B * C;
    D = E + F * G;
    H = M;
}
```

```
foreach Procedure begin
   build_DAGs( );
   foreach DAG in Procedure begin
      topological_sort(DAG);
      foreach Node in DAG begin
         compute_depth(Node);
         foreach Predecessor of Node begin
            if (Is Edge(Node, Predecessor) anti-dependency?) then begin
              if (depth(Node) < depth(Predecessor))  then depth(Node) = depth(Predecessor);
            end
         end
      end
      sort_by_depth(DAG);
      schedule(DAG);
   end
end
```

Fig. 2. Pseudo-code for expression scheduling

Data dependency analysis is performed to find data-independent expressions and put them in *par* blocks. A directed acyclic graph (DAG) scheduling technique is used to schedule independent expressions into several parallel expression groups as shown in **Fig. 2**. Each node in the DAG represents an expression and an edge represents *true, output* or *anti-dependency* between two nodes. Anti and output dependencies are false data dependencies that can be eliminated by variable renaming. However, renaming uses temporary variables that demand more FPGA resources.

True and output dependencies restrict expressions to be grouped in the same *par* block. On the other hand, anti-dependencies may or may not restrict the scheduling of expressions. The expression that writes the anti-dependent operand (i.e. *the sink expression*) may be scheduled in an earlier cycle than the expression that reads the anti-dependent operand (i.e. *the source expression*). The depth of the sink expression must be made equal to the depth of the source one to guarantee that they will be at least in the same *par* block.

After the computation of all depths in the DAG, the DAG is converted into a flat list of nodes sorted in ascending order of their depths. The expressions with the same depth are put into the same *par* block. If there is only one expression associated with a depth, then a *par* block is not necessary.

Table 1. Benchmarks

Benchmark	**Description**
adpcm encoder	16-bit PCM to ADPCM speech encoder
matrix multiplication	8-by-8 matrix multiplication
shellsort	Shell Sort algorithm of 32 integer numbers
2D convolution	8-by-8 2D convolution algorithm with 3-by-3 kernel matrix
FIR	32-tap Finite Impulse Response filtering
IDFT	32-point Integer Inverse Discrete Fourier Transform
huffman encoder	32-character Huffman Encoding algorithm
g721decoder	CCITT G.721 ADPCM decoding

2.3. Experimental Results

The benchmark programs used in this study are shown in **Table 1**. They represent various embedded computing fields such as DSP, image processing, telecommunications and scientific computing. All benchmarks perform integer arithmetic because *Handel-C* supports only integer arithmetic. For each benchmark, our compiler carries out loop parallelism and expression scheduling and finally generates the *Handel-C* code. The *Handel-C* code is then passed to the *Handel-C* compiler, which in turn generates an EDIF netlist for the **Xilinx Virtex-II XC2V6000** FPGA device. **Xilinx Virtex-II XC2V6000** is a 6M-gate chip with 8448 Configurable Logic Blocks (CLBs). Each CLB is made of four slices, each of which has two 4-input look-up tables (LUTs), two 16-bit D-type registers, dedicated multiplexers and fast carry look-ahead chain. In addition to an array of CLBs, the **Xilinx Virtex-II XC2V6000** FPGA chip has also 324KByte block RAM/ROM and 144 18x18-bit multipliers. We used **Xilinx ISE 6.2** tool set with standard placement/route effort levels.

In order to form a baseline model for performance comparison, each benchmark is also transformed into *Handel-C* without performing any of the aforementioned optimizations. Our metric for performance comparison is *the total execution time* for each benchmark. The total execution time is determined by multiplying *the total cycles* required to execute the benchmark by *the minimum clock period*. The total cycles is computed as the sum of every individual or parallel statement. The minimum clock period is the clock period required to operate the FPGA after each benchmark is mapped and implemented onto the FPGA.

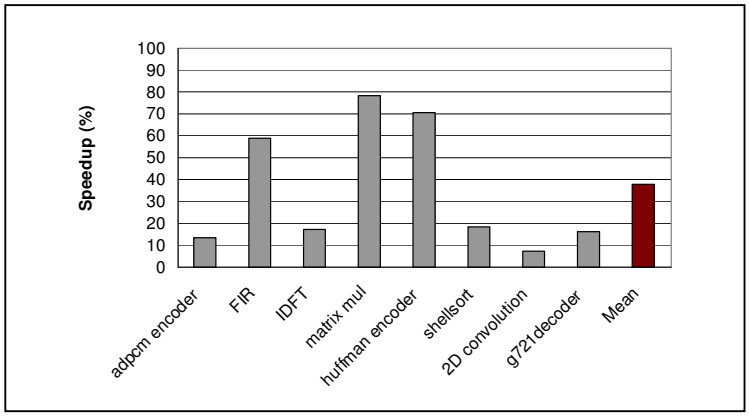

Fig. 3. Speedup results in the total execution time

Fig. 3 presents the speedups in terms of percentage reduction in the total execution time for each benchmark using two parallelization techniques. The highest speedups come from *matrix multiplication*, *huffman encoder* and *FIR* by *78%*, *70%* and *59%*, respectively. *matrix multiplication* has three nested loops from which the outermost two loops can be translated into *par* loops because there are no loop-carried data dependencies across their iterations. For *huffman encoder* and *FIR*, the loops that are converted into *par* loops have a great amount of expression parallelism. The speedups

of the other benchmarks vary between *7%* and *18%*. Smaller performance improvements in these benchmarks are caused by the fact that none of the loops can be parallelized. Only parallel expressions contribute to the speedups. In summary, an average of *38%* improvement in performance can be obtained over eight benchmarks.

3. Optimizations for Area and Power

This section discusses machine-specific optimizations to reduce area and power consumption in the **Xilinx Virtex-II XC2V6000** FPGA chip. Such optimizations typically attempt to utilize on-chip resources of block RAM/ROM and multipliers present on the FPGA chip so that precious FPGA area can be allocated to more compute-intensive operations or tasks.

3.1. On-chip RAM and ROM Utilization

Arrays are implemented using look-up tables and flip-flops in the FPGA slices. For FPGA devices that support on-chip block RAM and ROM such as in the **Xilinx Virtex-II** architecture, these arrays can be declared as *ram*, *rom* and *mpram* (multi-ported ram) in *Handel-C* to use on-chip RAM/ROM hardware resources so that the FPGA slices can be allocated for other uses. The difference between an array and a *ram* variable declaration is that any number of array elements can be read or written in a single clock cycle in an array, whereas only a single location in a single ported, single bank RAM array can be read or written in a single clock cycle. Similarly, ROM can be read only once in a clock cycle. If an array is to be used more than once in the same cycle, it can be declared as *mpram* to allow simultaneous read/writes through multiple ports in the same cycle. *mpram* supports only dual-port memory accesses.

A compiler algorithm is written to analyze the global and local array declarations and to find out whether they can be declared as *ram*, *rom* or *mpram*. An array variable can be declared as *rom* if it is not accessed more than **once** in a statement or a *par* block and all accesses are **reads**. The array variable can be declared as *ram* if it is not accessed more than **once** in a statement or a *par* block and **not** all accesses are **reads**. If it is accessed at most twice in a statement or a *par* block, then it can be declared as dual-port *mpram*. Array variable names can also be passed as arguments in procedure calls, and alias names can be used at the call sites. Thus, argument-parameter alias analysis is also performed for the whole program to determine the set of alias names for each array name passed as an argument to procedures. The array variable has to be kept as an array declaration if it is accessed more than **twice** in a statement or a *par* block.

3.2. On-chip Dedicated Multiplier Utilization

Customized multiplication units can take up a vast amount of slices in the FPGA. If the widths of multiplicands are wider, the multiplication operation takes longer and this can cause a drop in the overall clock frequency of the FPGA since every instruc-

tion in *Handel-C* takes exactly one cycle. Hence, it is requisite that the compiler must assign some of multiplication operations to the dedicated fast on-chip 18x18-bit multipliers in the FPGA chip. If the widths of the multiplicands are larger than 18 bits, several multipliers can be tied together to form wider multipliers. A multiplier can be assigned to more than one operation but only one of them can access it at any cycle. This will create a multiplexer logic in front of the multiplier to route only one set of multiplicands to the multiplier. As more sets of multiplicands share a multiplier, the multiplier becomes slower due to its wide multiplexer. The compiler must intelligently distribute multiplication operations among the dedicated multipliers so that none of the multipliers has prohibitively wide multiplexers. **Fig. 4** shows the on-chip 18 by 18 bit multiplier assignment algorithm.

```
foreach Procedure
  begin
    MULOP_LIST = find_multiplication_operations();
    sort_by_bitwidth(MULOP_LIST);
    initialize_multiplier_weights(MULTIPLIER_LIST);
    while (not end of MULOP_LIST)
      begin
        OP = pick_operation(MULOP_LIST);
        n  = bitwidth(OP_operand1);
        m  = bitwidth(OP_operand2);
        if (max(n,m) > 18) then number_of_required_multipliers = ⌈max(n,m)/18⌉;
        else number_of_required_multipliers = 1;
        MUL_SET = find_smallest_weighted_multipliers(MULTIPLIER_LIST,number_of_required_multipliers);
        assign(OP, MUL_SET);
        increment_multiplier_weights(MUL_SET,(n+m));
      end
  end
```

Fig. 4. The pseudo-code for the dedicated multiplier assignment algorithm

The algorithm puts each multiplication operation into a list by traversing each procedure. Then, the list is sorted by the sum of bit-widths of multiplicands of each operation in descending order. Each multiplier is assigned a weight that shows the estimated length of its multiplexer if more than one set of multiplicands share the multiplier. After all the weights are initialized to zero, an operation is popped off the list. If the bit-width of any of its multiplicands is less than or equal to 18, then only one multiplier is needed. Otherwise, more than one multiplier is needed to implement the current multiplication. The number of the required multipliers is computed by dividing the maximum bit-width by 18. After determining the exact number of multipliers, the multiplier list is searched to select the multipliers with the lowest weights. Then, the multiplication operation is assigned to these selected multiplier(s) and their weight(s) are incremented by the sum of the bit-widths of the two multiplicands. The same steps are applied to all operations in the multiplication operation list until all of them are assigned to the dedicated multipliers. The objective of the algorithm is to ensure that the multipliers to which long multiplications (i.e. the ones with wider operand bit-widths) are assigned are not multiplexed with many different sets of multiplicands of other multiplications.

3.3. Experimental Results

Area usage and power consumption are measured for each benchmark. Similar to *Section 2.3*, our base model for comparison are the same benchmarks with no on-chip RAM/ROM and multiplier utilization optimizations. For the area usage, we measure *the percentage reduction* in the number of FPGA slices on the chip. The *dynamic logic power consumption* consumed by the logic implemented on the FPGA chip is measured after a complete placement and routing of the design on the FPGA using **Xilinx's** *XPower* estimation tool. We set the FPGA's clock frequency and source voltage to 100MHz and 1.5V, respectively. Also, circuit switching activity rate is set to 100% or ½ the clock frequency, which is the rate at which a logic gate switches.

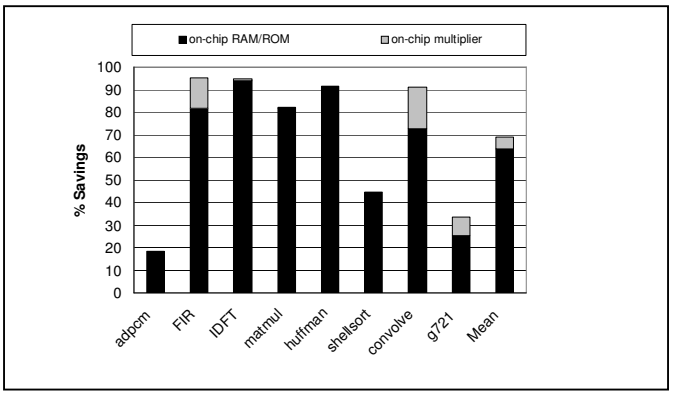

Fig. 5 The percentage reduction in the number of FPGA slices using on-chip RAM/ROMs and multipliers

Fig. 5 shows the percentage reduction in the number of FPGA slices on the chip. The black bar denotes the reduction contributed by only on-chip RAM/ROM and the grey bar represents the reduction contributed by only on-chip multiplier utilizations. It is possible to save an average of *69%* of FPGA slices for all benchmarks as shown in the last column. Of this average, *64%* comes from the on-chip RAM/ROM utilization and only *5%* savings from the slices are due to the on-chip multiplier utilization. The arrays in all benchmarks can fit into the on-chip RAM/ROM structures. Although the majority of FPGA slice savings is caused by efficient utilization of on-chip RAM/ROMs, a reasonable amount of reduction has been made using on-chip multiplier optimizations such as in *FIR*, *2D convolution* and *g721decoder*. These three benchmarks have several multiplication operations that are allocated to on-chip multipliers. On the other hand, *adpcm, shellsort. huffman, matrix multiplication* and *IDFT* have either no multiplication or only one multiplication operation. Thus, the contribution of on-chip multiplier utilization to the area reduction for these benchmarks remains less than 1%.

Fig. 6 presents the percentage reduction in dynamic logic power consumption of on-chip RAM/ROM and multiplier utilizations for eight benchmarks. The large number of FPGA slice savings allows energy-efficient designs by reducing the logic power consumption proportional to the number of slices. Over eight benchmarks, an

average of *55%* (i.e. 51% due to on-chip RAM/ROMs and 4% due to on-chip multipliers) reduction in dynamic logic power consumption is possible by utilizing on-chip FPGA resources.

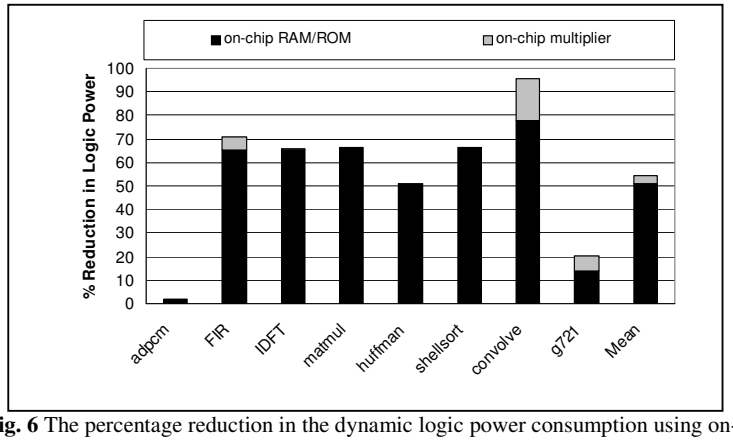

Fig. 6 The percentage reduction in the dynamic logic power consumption using on-chip RAM/ROMs and multipliers

4. Related Work

There have been several projects [1] [2] [3] [4] [5] [6] [7] [9] [10] [11] that attempt to transform modified C programs to custom hardware by way of producing low-level HDL languages such as Verilog or VHDL. In spite of their portable nature, these languages are not suitable to explore instruction-level and loop-level parallelism. Hence, we use *Handel-C* as our backend language whose high-level constructs allow us to exploit both instruction and loop-level parallelism with explicit control.

5. Conclusion

We have introduced a compilation framework that optimizes embedded applications written in C to improve performance, and reduce area/power requirements. Loop parallelism and expression scheduling are applied to decrease the total execution time of the programs. We have shown that an average of **38%** improvement in performance is possible with these parallelizations for all benchmarks. We have also presented the experimental results of machine-specific optimizations for utilizing on-chip block ROM/RAMs, and dedicated multipliers to reduce area and power consumption of the **Xilinx Virtex-II XC2V6000** FPGA chip. Our results have showed that an average of **69%** reduction in area and an average of **55%** reduction in logic power consumption over eight benchmarks can be attained through the efficient utilization of on-chip resources.

We are exploring various compiler optimizations such as shared expressions, storage reuse, expression splitting and combining, and software pipelining. Our ultimate aim is to develop an iterative hardware compilation environment that can apply compiler transformations using feedback-directed data from timing, resource mapping and power estimation tools to optimize for combinations of performance, power consumption and area.

References

[1] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. "PRISM-II Compiler and Architecture", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, California, April 1993.

[2] B. A. Draper, A. P. W. Böhm, J. Hammes, W. Najjar, J. R. Beveridge, C. Ross, M. Chawathe, M. Desai, J. Bins, "Compiling SA-C Programs to FPGAs: Performance Results", *International Conference on Vision Systems*, Vancouver, July, 2001.

[3] M. Hall, P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain, and J. Granacki, "DEFACTO: A Design Environment for Adaptive Computing Technology", Proceedings *of the 6th Reconfigurable Architectures Workshop* (RAW'99), 1999.

[4] J. Frigo, M. Gokhale, and D. Lavenier "Evaluation of the Streams-C C-to-FPGA Compiler: An Application Perspective", *9th ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, February, 2001.

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

[6] M. Budiu, and S. C. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics", *7th ACM International Symposium on Field-Programmable gate Arrays*, 1999.

[7] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and Compiling Applications for RaPiD", *Field-Programmable Custom Computing Machines*, 1998.

[8] R. P. Wilson, R. S. French, C. S. Wilson, S. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers", *Tech. Report*, Computer Systems Laboratory, Stanford University, CA, USA, 1994.

[9] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations", *the 16th International Conference on VLSI Design*, New Delhi, India, Jan. 2003.

[10] Jonathan Babb, Martin Rinard, Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe, "Parallelizing Applications Into Silicon", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines '99* (FCCM '99), Napa Valley, CA, April 1999.

[11] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist and M. Sivaraman, "PICO: Automatically Designing Custom Computers", *IEEE Computer*, vol. 35, no. 9, pp. 39-47, September 2002.

[12] Celoxica, *Handel-C Language Reference Manual*, Version 3.1, 2002.

[13] Xilinx, *Xilinx Virtex-II Architecture Manual*, Sep. 2002.