Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs

Betul Buyukkurt, Zhi Guo and Walid A. Najjar

Department of Computer Science and Engineering
University of California - Riverside
Riverside CA 92507, USA

Abstract. Loop unrolling is the main compiler technique that allows reconfigurable architectures achieve large degrees of parallelism. However, loop unrolling increases the area and can potentially have a negative impact on clock cycle time. In most embedded applications, the critical parameter is the throughput. Loop unrolling can therefore have contradictory effects on the throughput. As a consequence there exists, in general, a degree of unrolling that maximizes the throughput per unit area.

This paper studies the effect of loop unrolling on the area, clock speed and throughput within the ROCCC, C to VHDL compilation framework. Our results indicate that due to the unique design of the ROCCC compilation framework, FPGA area either *shrinks* or *increases at a very low rate* for the first few times the loops are unrolled. This reduced area causes the clock cycle time to decrease and thus a great gain in throughput. Our results also show that there are different optimal unrolling factors for different programs.

1 INTRODUCTION

Loop unrolling is the main compiler technique that allows reconfigurable architectures achieve large degrees of parallelism. Loops that do not carry dependencies from earlier iterations can theoretically be fully unrolled to achieve maximum parallelism. However due to the adverse impact of loop unrolling on clock cycle time, there exists, in general, a degree of unrolling that maximizes the throughput per unit area. Since in most embedded systems, the critical parameter is the throughput, this implies that there should be different optimal unrolling factors for different programs.

This paper studies the effect of loop unrolling on the FPGA area, clock speed and throughput within the ROCCC C to VHDL compiler framework. Our results indicate that the consumed FPGA area either shrinks or grows at a very low rate for the first few times the loops are unrolled. In most cases, decrease in area leads to a decrease in the clock cycle time thus a great gain in throughput. Such results indicate that a design space exploration in the loop-unrolling factor vs.

performance would indicate an optimal number of times to unroll for maximum throughput.

The impact of loop unrolling on FPGA area has been reported in [1] and [2]. Crowe et. al. [1] implements a AES symmetric key cryptosystem, SHA-512 secure hashing algorithm and a public key cryptography algorithm on a single FPGA. The area increases from no unrolling to an unrolling factor of 2 by 15% and to an unrolling factor of 4 by 75%. Park et. al. [2] implements binary image correlation on an FPGA which is used in template matching computations in image processing systems. Binary image correlation operates a window over a 2D array. This study reports FPGA areas for unrolling factors of 2x16, 4x16, 8x16 and 16x16. The overall area increases less than 50% and the area of the datapath shrinks, when the design is moved from no unrolling to an unrolling factor of 2. For unrolling factor 4x16, the area increases by little over 50% compared to no unrolling and their datapath area increases around 25%.

Both of the above studies did not report generating their VHDL from high-level languages, whereas in our work the VHDL is generated using our ROCCC, C to VHDL compiler system. Both [1][2] included the areas of the entire datapath, the controller logic and the memory interface area in their results. Our work reasons about the advantages of our ROCCC system that lead to the shrinkage of area on the FPGA. We give a breakdown of the area as the loops are unrolled and how the FPGA real estate is allocated between two main components of our design: 1) the datapath & the controller and 2) the smart buffer, which helps maximize data reuse across iterations.

This paper is organized as follows. Next section describes the ROCCC system. Section 3 talks about the experimental setup and results. Section 4 talks about other existing HLL to HDL compilers. Finally, Section 5 concludes the paper.

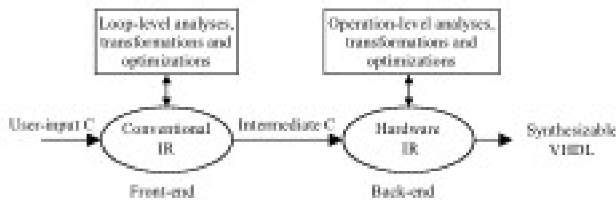


Fig. 1. ROCCC system overview

2 THE ROCCC SYSTEM

ROCCC is a compiler system built on top of the SUIF2 [3] and MACHSUIF [4][7] compiler infrastructures. The ROCCC system, shown in a diagram in Figure 1,

is composed of two main components: a front-end that applies the high-level transformations and a back-end that handles the low-end optimizations as well as the VHDL code generation. The objectives of the ROCCC optimizations are threefold:

1. **Parallelism.** Maximize the throughput by exploiting the largest amount of loop and instruction level parallelism within the resource constraints of the FPGA.
2. **Storage.** Optimize the use of on-chip storage by effecting smart re-use of data and minimizing the accesses to memory.
3. **Pipelining.** Generate an efficient, pipelined datapath within the resource constraints in order to minimize clock cycle time.

Among the main strengths of the ROCCC system is the number of loop-level transformations it implements. The ROCCC system currently performs the following loop transformations: Invariant code motion, partial and full loop unrolling, loop peeling, loop un-switching, loop tiling, strip-mining and loop fusion. At the procedure level ROCCC performs the following optimizations: Constant propagation of scalars and constant array masks, constant folding, elimination of algebraic identities, copy propagation, dead and unreachable code elimination, code hoisting, code sinking, scalar renaming and division by constant approximation using shifts and additions. ROCCC also generates reduction on scalars that accumulate values through associative and commutative operations on themselves. Although some of the above analysis/passes exists inside the SUIF2 framework, we wrote our own passes to be able to easily gather and annotate the IR with all the ROCCC specific information. Our analysis and optimization passes use the high-level IR in SUIF2, where all the control structures and arithmetic expressions are preserved as close to their format in the original source code as possible.

Once the above passes are executed, the optimizer output is then analyzed and prepared to generate the data-path and controller information. There are few passes that ROCCC runs on the optimizer output to extract and format the datapath of the loop bodies. These passes are as follows:

- **Scalar I/O detection:** This pass marks all scalar variables that are computed or updated with in the loop body and referenced by the rest of the code once the loop completes execution.
- **Scalar replacement:** This pass decouples array accesses from computation. Figure-2(b) shows the moving filter code after the scalar replacement pass. The middle code block is isolated from memory accesses and is used to form the datapath. The top array read and bottom array write chunks are analyzed to form the smart buffer [18], address generation and the controller circuits.
- **Feedback variable detection:** This pass annotates the scalar variables, which are dependent upon their values from the loop's previous iteration.

The extracted datapath code is then transferred to Machine-SUIF IR, where most of the instruction level parallelism is brought up and pipelines are formed.

```
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) 9-tab Moving filter

```
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;
    T0 = (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;
}
```

(b) Moving filter loop after being unrolled twice, and applied the constant folding, division by constant elimination and scalar replacement transformations

Fig. 2. Moving filter code before and after ROCCC transformations

We modified Machine-SUIF's virtual machine (SUIFvm) IR to build our data flow. All arithmetic opcodes in SUIFvm have corresponding functionality in IEEE 1076.3 VHDL with the exception of division. Machine-SUIF's existing passes, like the Control Flow Graph (CFG) library [8], Data Flow Analysis library [9] and Static Single Assignment library [10] provide useful optimization and analysis tools for our compilation system. Our compiler at this level automatically places latches in the data flow graph to pipeline the datapath.

ROCCC analyzes the array accesses at the SUIF2 level and generates the smart buffer, which is a storage mechanism that helps minimize the accesses to off-chip memory bandwidth for programs that operates on windows sliding over arrays such as signal/image processing applications. The smart buffer stores the input data for future iterations and removes old data to save room for new input data.

The controllers generated by ROCCC include address generators, which export a series of memory addresses according to the memory access pattern of the loop, and a higher-level controller, which controls the address generators. They are all implemented as pre-existing parameterized FSMs (finite state machine) in a VHDL library.

3 EXPERIMENTAL EVALUATION

3.1 Experimental Set-Up

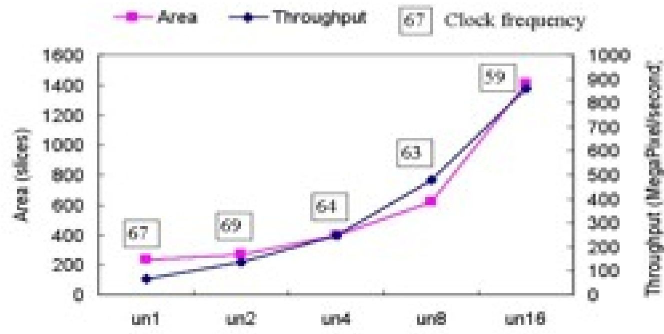
We used Xilinx ISE 6.2.03i to synthesize and place-and-route the generated VHDL code. Our target was the Xilinx Virtex-II xc2v8000-5 FPGA. We used five benchmarks to collect our data. fir5 and fir15 are 5-tap and 15-tap constant-coefficient finite-impulse-response filters and mf9 is a 9-tap moving average filter. fir5, fir15 and mf9 all operate on one-dimensional arrays. dwt (Discrete Wavelet Transform) is part of the JPEG 2000 compression standard. It is a doubly nested loop operating on a 5x3 block of pixels. mvc computes the first step of the three step Moravec corner detection algorithm, which computes the variance of the center pixel within a 3x3 window of 9 pixels. Being image-processing kernels, both dwt and mvc operate on 2D arrays.

fir5, fir15, mf9, dwt and mvc are all kernel loops. The source codes are directly read into the SUIF2 intermediate format and the ROCCC optimizations described in the previous section are applied. Further, we assumed that the I/O bandwidth between the datapath and the on-chip memory is sufficient, when performing unrolling, since the required data bus width increases with unrolling. The 1-D benchmarks are unrolled for 2, 4, 8 and 16 times. mvc is unrolled for 2x2 and 4x4 times. Finally, the dwt code is unrolled at different unrolling factor combinations ranging from 1 to 8 in powers of 2 in either dimension. un1, un2, un4 and un8, un16 labels on the figures indicate the unrolling factors of none, 2, 4, 8 and 16 for benchmarks operating on one dimensional arrays and unxXy labels indicate an unrolling factor of x applied to the outer loop and an unrolling factor of y to the inner loop respectively. We collected data for 8-bit data size. The reported area and clock frequency results are place-and-route results.

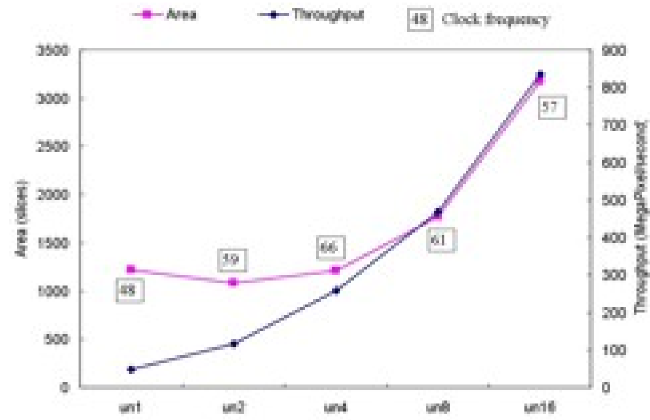
3.2 Area

Figure three displays our area results for our benchmarks that execute on one-dimensional arrays. The area results on these figures show the combined areas of the datapath, the controller and smart buffers. As the results indicate, the overall area shrinks from the original version to un2, even un4 for some cases such as that of mf9. In figure three (b) and (c) the not unrolled cases are not the minimal area points. This fact shows that there exist optimal times to unroll.

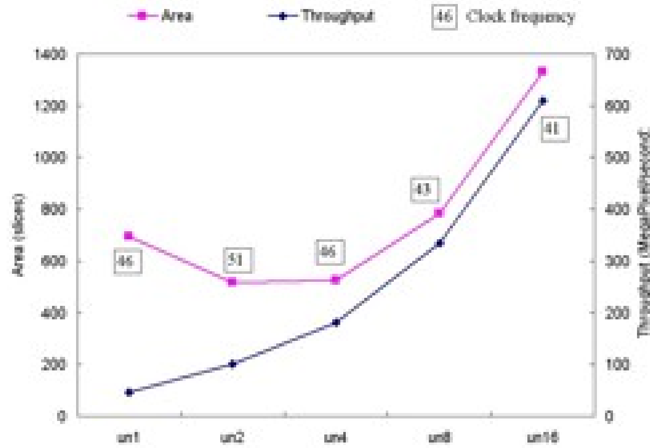
Figure four (a) shows the results of unrolling a dwt code. Note that an unrolling of 8x8 means that the 5x3 block is replicated eight times in each direction. In other words 64 windows of 5x3 are operated simultaneously. From no unrolling



(a)



(b)



(c)

Fig. 3. Area, clock frequency and throughput for (a)fir5, (b)fir15 and (c)mf9

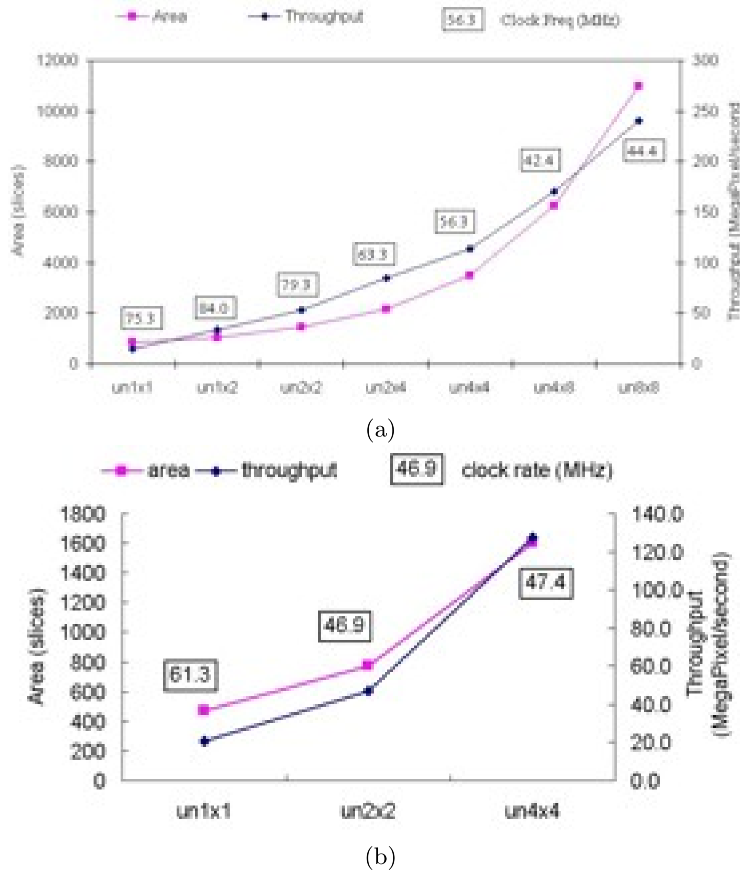


Fig. 4. Area, clock frequency and throughput for (a)dwt and (b)mvc

to 64 concurrent loops the area grows by 12 while the throughput grows by 16 in spite of the clock cycle time being about twice as long. Here also it seems that the 1x2, 2x2, 2x4, 4x4 and 4x8 unrolling factors achieve a better throughput per area. Note that the 8x8 unrolling achieves a throughput of 240 MegaPixels/sec, which is more than twice the rate necessary for high-definition TV.

Moravec's pixel variance computation kernel results indicate an almost linear increase in area. However, for this example the operator has a doubly nested loop, as does the DWT code. Thus, the unrolled version operates not just on one more set of input data as it would be over a 1-D array, but three more sets of input data since it is twice unrolled towards both directions over a 2-D array.

3.3 Breakdown of the FPGA Area

Table-1 shows the slice and percentage breakdown of the FPGA area into datapath, smart buffer and controllers. To produce the table, we first mapped the

entire circuit on to the FPGA, which gave us the results in figures three and four. Since place and route merges the circuits to use the FPGA area as efficiently as possible, it is not possible to distinguish which slices on the FPGA belonged to which component of the design. Thus to be able to obtain an estimate of the area breakdown, we separately mapped the datapath and the smart buffer circuits on to the FPGA, through which we obtained areas of the datapath and smart buffers. Then, to obtain the controller area we summed up the datapath and the smart buffer results and subtracted it from the total area shown in the figures. Obviously, this procedure would not give a correct area estimate of the area occupied by the control logic - the subtraction operation resulted in negative computed area values in some cases as in the case of fir15 -, however it adequately shows where the reduction in area came from. We did not see it necessary to map the controllers separately, since the areas of the controllers stay around the same due to the fact that the controllers are all implemented as pre-existing parameterized FSMs in a VHDL library, whose size does not depend on the unrolling factor.

Table 1. The Area Breakdown (8-bit)

		Datapath		Smart Buffer		Control Logic		Total
		slices	%	slices	%	slices	%	slices
Fir5	un1	42	17.8	172	72.9	22	9.3	236
	un2	80	29.5	164	60.5	27	10.0	271
	un4	159	40.2	201	50.8	36	9.1	396
	un8	317	51.2	250	40.4	52	8.4	619
	un16	769	54.6	483	34.3	157	11.1	1409
Fir15	un1	168	13.7	1086	88.9	-32	-2.6	1222
	un2	310	28.6	721	66.5	54	5.0	1085
	un4	604	49.9	528	43.6	79	6.5	1211
	un8	1186	66.9	490	27.7	96	5.4	1772
	un16	2358	74.2	566	17.8	254	8.0	3178
Mf9	un1	45	6.5	620	89.5	28	4.0	693
	un2	83	16.1	406	78.5	28	5.4	517
	un4	166	31.6	324	61.6	36	6.8	526
	un8	332	42.5	399	51.0	51	6.5	782
	un16	662	49.7	513	38.5	156	11.7	1331
Moravec	un1x1	33	7.0	181	38.3	258	54.7	472
	un2x2	133	17.1	368	47.4	275	35.4	776
	un4x4	532	33.3	714	44.7	352	22.0	1598
DWT 5x3	un1x1	205	23.7	425	49.1	235	27.2	865
	un2x1	351	31.1	496	43.9	283	25.0	1130
	un1x2	338	31.9	515	48.5	208	19.6	1061
	un2x2	590	40.4	618	42.3	252	17.3	1460
	un2x4	1068	49.3	750	34.6	347	16.0	2165
	un4x4	2003	57.5	996	28.6	487	14.0	3486
	un8x4	3875	63.1	1498	24.4	767	12.5	6140

We could not collect the smart buffer's place and route data, for the case of DWT unrolled 8x8 times, due to the fact that the smart buffer's ports exceeded the chip I/O capacity. When the smart buffers are mapped together with the datapath and the controllers, smart buffers' I/O ports become just on-chip wires connected to the datapath and the controllers.

According to the figures in Table-1, the circuit area of the datapath increased almost at a linear rate, although it is known that loop unrolling introduces more opportunities for optimizations especially on the datapath. The reason for the linear increase is that all the data path codes are mapped after being decoupled from all memory accesses, all address computation code, and being applied an extensive set of procedure level optimizations.

The gain in area on the FPGA comes mainly from the shrinkage of the circuit size of the smart buffers due to its unique design. Smart buffer organizes the data that is received from the memory in windows. Each window has its own control logic enabling when and which sets of windows are to be exported to the datapath. For the un1 case, the number of windows in the smart buffer is large, although anytime only one of the windows is active. When we unroll the loop, the buffer size increases to hold more loops of input data, however the control logic cost decreases since the number of windows decreases due to the increase in the window size. Window size represents the amount of data that has to be dispatched to the datapath per clock cycle. Since the control logic size diminishes, the overall area for the smart buffer decreases.

To illustrate, if the not unrolled 5-tap-FIR in Figure-5 has a smart buffer whose size is seven words (Figure 6), then every five words constitutes a window, since the loop stride is one. Thus, this seven-word smart buffer contains seven windows. In Figure 6-left (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 datapath. To illustrate better, Figure 6-right 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-right, 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. As a result, the overall area shrinks due to the control logic shrinkage.

3.4 Clock Cycle & Throughput

A circuit's clock rate is affected by many factors. The smaller a design is, the easier it is for the synthesis tool chain to generate a faster circuit for it. The data points on the figures where clock frequency increases are the points where the design area shrinks. However, the overall decrease in clock speed for higher unroll factors should not be taken as that the overall throughput is decreasing. The number of parallel iterations generated by high unroll factors imply that the number of outputs generated per clock cycle on a pipelined datapath increases.

```
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];
}
```

Fig. 5. 5-tap-FIR in C

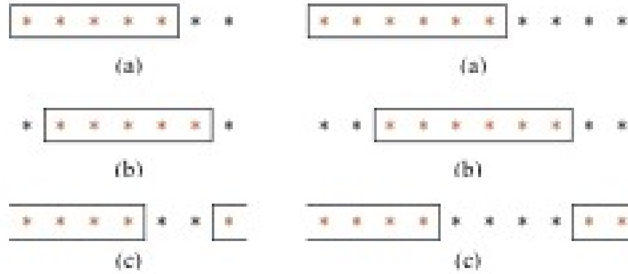


Fig. 6. Windows No.0, No.1 and No.6 in the smart buffer for the not unrolled 5-tap-FIR (left) and the windows No.0, No.1 and No.4 in the smart buffer for the twice-unrolled 5-tap-FIR (right)

Thus, the effect of the clock rate decrease with increased unrolling is overcome with the increased parallelism in the unrolled codes.

4 RELATED WORK

Many projects have worked on translating high-level languages into hardware using various approaches. SystemC [12] is designed to provide roughly the same expressive functionality of VHDL or Verilog and is suitable to designing software-hardware synchronized systems. Handel-C [13], a low level hardware/software construction language with C syntax, supports behavioral descriptions and uses a CSP-style (Communicating Sequential Processes) communication model.

SA-C [14] is a single-assignment, high-level, synthesizable language. Because of special constructs specific to SA-C (such as window constructs) and its functional nature, its compiler can easily exploit data reuse for window operations. SA-C does not support while-loops. ROCCC compiler transforms the IR into single-assignment form at back-end. Users are not required to write algorithms in a single-assignment fashion.

Streams-C [15] relies on the CSP model for communication between processes, both hardware and software. Streams-C can meet relatively high-density control requirements. The compiler generates both the pipelined datapath and the corresponding state machine to sequence the basic and pipeline blocks of the datapath. ROCCC supports two-dimensional array access and performs input data reuse analysis on array accesses to reduce the memory bandwidth requirement. Streams-C does not handle 2D arrays.

The DEFACTO [16] system takes C as input and generates VHDL code. It allows arbitrary memory accesses within the datapath. The memory channel architecture has its FIFO queue and a memory-scheduling controller. ROCCC has abundant loop transformations to increase parallelism and performs data reuse using the smart buffer.

GARP's [17] compiler is designed for the GARP reconfigurable architecture. The compiler generates a GARP configuration file instead of standard VHDL. GARP's memory interface consists of three configurable queues. The starting and ending addresses of the queues are configurable. The queues' reading actions can be stalled. The GARP-C compiler is specific to the GARP reconfigurable architecture while ROCCC targets commercial available configurable devices and generates synthesizable VHDL. GARP does not handle 2D arrays.

SPARK [11] is another C to VHDL compiler. Its optimizations include code motion, variable renaming and loop unrolling. The transformations implemented in SPARK reduce the number of states in the controller FSM and the cycles on the longest path. SPARK does not perform optimizations on input data reuse. Thus, ROCCC explores more parallelism than SPARK. ROCCC performs loop pipelining if there are no loop carried dependencies. SPARK handles 2D arrays by converting them into a one-dimensional array and computes memory addresses on the datapath, however ROCCC decouples computation from address calculation using scalar replacement.

CASH [19] is a C to Verilog compiler that generates a hardware dataflow machine that directly executes the input program. It targets asynchronous ASIC implementations. Catapult C [20] is a C++ to RTL compiler that generates hardware for ASICs/FPGAs. The compiler performs loop unrolling, loop pipelining and bit-width resizing. ROCCC harnesses its smart buffer architecture to increase the throughput by reusing input data.

Compared to previous efforts in translating C to HDLs, ROCCC's distinguishing features are its emphasis on maximizing parallelism via loop transformations, maximizing clock speed via pipelining and minimizing area and memory accesses, a feature unique to ROCCC. ROCCC handles 2D arrays and can optimize memory accesses for window operations. On an image processing code operating over an image using a 3x3 window, unrolling and reusing of already fetched data from the smart buffers reduces the memory re-fetches from 800% (without any optimizations) down to 6.25% (when the unrolling factor is 32x32).

5 CONCLUSIONS

Compilers for reconfigurable architectures achieve parallelism through unrolling and optimizing the kernel loops inside source codes. This paper studied the effect of loop unrolling on the FPGA area within the ROCCC compiler system. Our results indicate that the relation between the unrolling factor and the overall area growth on the FPGA is non-linear for the ROCCC, C to HDL compiler. This indicates that for systems where area is a constraint, using ROCCC's technology designers can gain more throughout with less area applying loop unrolling. We

observed that overall FPGA area either shrinks or increases at a very low rate for the first few times the loops are unrolled. This shows that there are different optimal times to unroll for different programs.

References

1. Crowe. F., Daly A., Kerins T. and Marnane W.: Single-Chip FPGA Implementation of a Cryptographic Co-Processor. International Conference on Field Programmable Technology, Brisbane, December 2004
2. Park J., Diniz P.C., Shayee K.R.S.: Performance and Area Modeling of Complete FPGA Designs in the Presence of Loop Transformations. IEEE Transactions on Computers, Nov. 2004, pgs. 1420-1435, Vol 53, Issue 11, ISSN: 0018-9340
3. SUIF Compiler System. http://suif.stanford.edu, 2004
4. Machine-SUIF. http://www.eecs.harvard.edu/hube/research/machsuif.html, 2004
5. 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
6. M. D. Smith and G. Holloway. An introduction to Machine SUIF and its portable libraries for analysis and optimization. Division of Engineering and Applied Sciences, Harvard University.
7. G. Holloway and M. D. Smith. Machine-SUIF SUIFvm Library. Division of Engineering and Applied Sciences, Harvard University 2002.
8. G. Holloway and M. D. Smith. Machine SUIF Control Flow Graph Library. Division of Engineering and Applied Sciences, Harvard University 2002.
9. G. Holloway and A. Dimock. The Machine SUIF Bit-Vector Data-Flow-Analysis Library. Division of Engineering and Applied Sciences, Harvard University 2002.
10. G. Holloway. The Machine-SUIF Static Single Assignment Library. Division of Engineering and Applied Sciences, Harvard University 2002.
11. SPARK project. http://mesl.ucsd.edu/spark/, 2005.
12. SystemC Consortium. http://www.systemc.org, 2005.
13. Handel-C Language Overview. Celoxica, Inc. http://www.celoxica.com. 2004.
14. W. Najjar, W. Bohm, 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.
15. 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.
16. P. Diniz, M. Hall Park, J. Park, B. So and H. Ziegler. Bridging the Gap between Compilation and Synthesis in the DEFACTO System. Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing Synthesis (LCPC'01), Oct. 2001.
17. T. J. Callahan, J. R. Hauser, J. Wawrzynek. The Garp Architecture and C Compiler. IEEE Computer, April 2000.
18. 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.
19. M. Budiu, G. Venkataramani, T. Chelcea and S. C. Goldstein. Spatial Comput-ing. ASPLOS 2004.
20. http://www.mentor.com/products/c-based_design/