Impact of High-Level Transformations within the ROCCC Framework

BETUL BUYUKKURT, JOHN CORTES, JASON VILLARREAL,
and WALID A. NAJJAR
University of California, Riverside

Reconfigurable computers, where one or more FPGAs are attached to a conventional microprocessor, are promising platforms for code acceleration. Despite their advantages, programmability concerns and the lack of efficient design tools/compilers for FPGAs are preventing the technology's widespread adoption. The traditional compiler technology is microprocessor-based-systems-specific and needs to be customized and augmented to address the needs in reconfigurable computing. The challenges are several due to the resources and performance constraints for FPGAs being drastically different than those of microprocessors, and also that compiling for FPGAs requires laying the computation in space by a circuit rather than in time by a sequence of instructions.

ROCCC is an optimizing C-to-VHDL compiler specifically targeting the reconfigurable computer platforms. ROCCC includes several high-level optimizations that parallelize and optimize the source code for minimized area and critical path length and maximized throughput. This article presents the effect of ROCCC's high-level transformations on the performance of the generated VHDL output. ROCCC utilizes: (1) several array access optimizations to eliminate redundant memory accesses, (2) procedure-level optimizations to achieve circuit area reductions of up to 88% compared to circuit areas generated from unoptimized codes, (3) loop-level optimizations to increase the throughput, and (4) transformations unique to certain classes of applications. The preceding listed features help ROCCC generate circuits with very large degrees of parallelism capable of very high computation rates.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Retargetable compilers*; *optimization*; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Compilers, code optimization, parallelization, ROCCC

This work is an extension of several works that precede this submission. The ROCCC compiler framework was in Guo et al. [2004, 2005] and Buyukkurt et al. [2005, 2006]; the concept of smart buffers was published in Guo et al. [2004]; impact of loop unrolling on area, clock cycle, and throughput within the ROCCC framework was published in Buyukkurt et al. [2006].
Authors' addresses: B. Buyukkurt (corresponding author), J. Cortes, J. Villarreal, and W. A. Najjar, Department of Computer Science and Engineering, University of California Riverside, Engineering BU-11, Room 351, Riverside, CA 92521; email: abuyukku@cs.ucr.edu.

DOI 10.1145/1880043.1880044 http://doi.acm.org/10.1145/1880043.1880044

ACM Reference Format:
Buyukkurt, B., Cortes, J., Villarreal, J., and Najjar, W. A. 2010. Impact of high-livel transformations within the ROCCC framework. ACM Trans. Architec. Code Optim. 7, 4, Article 17 (December 2010), 36 pages.
DOI = 10.1145/1880043.1880044 http://doi.acm.org/10.1145/1880043.1880044

1. INTRODUCTION

The growth in size and the speed of FPGA devices in recent years has led to the development of configurable computers, where one or more FPGA devices are used as hardware accelerators. The main advantage of configurable computing is that they combine the efficiency of customized data path implementation with the reprogramming flexibility of software. Indeed this has opened up, or reopened, a novel computing paradigm: spatial computing, where a computation is represented in space by a circuit rather than in time by a sequence of operations.

Spatial computing is extremely effective in streaming applications where a set of operations are applied to each element of a set of data. Examples of streaming applications include image and video processing, signal processing, cryptography, and certain classes of high-performance computing applications such as bioinformatics applications and molecular dynamics simulations.

Code acceleration in hardware is performed by expressing frequently executed code segments as hardware circuits, typically by writing them in some Hardware Description Language (HDLs) such as VHDL or Verilog. The data is then streamed through these circuits. Several projects [Dydel and Bala 2004; Puttegowda et al. 2003; Jacobi et al. 2005; Singpiel et al. 2000] have reported speedups in the 100s and 1000s. Such speedups come from large-scale parallelism, made possible by high-capacity FPGAs, as well as from customized circuit design. Since applications such as signal, image, and video processing exhibit very large amounts of parallelism, mapping such computations to circuits can drastically improve its efficiency as compared to running them on a traditional microprocessor.

Despite these advantages, FPGAs are not yet ideal platforms for application code development. One major drawback is the lack of programming tools that are accessible to traditionally trained application code developers. Currently, the developer must manually map the portion of the program that ought to be accelerated on the FPGA by rewriting it as a circuit and instantiating one or more buses to connect the microprocessor to the FPGA and both to the on-chip memory and the I/O devices. Currently all of these tedious tasks are done manually.

Optimizing compilers for traditional processors have benefited from several decades of extensive research that has led to extremely powerful tools. Similarly, Electronic Design Automation (EDA) tools have also benefited from several decades of research and development leading to powerful tools that can translate VHDL and Verilog code, and recently SystemC code, into efficient circuits. However, little work has been done to combine these two approaches. Several projects have implemented various types of High-Level Language (HLL) to HDL translations [Gokhale et al. 2000; Gupta et al. 2003; Bondalapati et al. 1999; Callahan et al. 2000]. For the most part, these efforts have focused on the translation of C or C++ to HDLs with optimizations supporting very simple loop nests and one-dimensional arrays.

ROCCC (Riverside Optimizing Compiler for Configurable Computing) is an optimizing C-to-VHDL compiler targeting FPGA and CSOC platforms. ROCCC optimizes and parallelizes the most frequently executed kernel loops in applications such as multimedia and scientific computing. Its objectives are:

(1) To bridge the gap between compiled and hand-written code. ROCCC-generated circuits [Guo et al. 2005] take around 2–3 times the area and run at comparable clock rated compared to that of XilinxIPs.
(2) To apply *extensive compile-time transformations* on multidimensional arrays and nontrivial loop nests. Such transformations would be too complex for a human programmer to handle in a reasonable time.

In this article we describe ROCCC's high-level transformations and report on the performance of the compiler generated code in applications involving both signal processing and supercomputing kernels. Work that is submitted for publication for the first time in this article are:

—The user interface of ROCCC;
—Array access optimizations of the ROCCC framework;
—Generation of hardware circuits from constant qualified array definitions for indirection involving nonlinear array accesses to constant qualified arrays;
—Impact of global optimizations on area, clock cycle, and throughput within the ROCCC framework;

Significant results that are reported for the first time in this article are:

—ROCCC's easy to use interface;
—The possibility of generating area- and clock-cycle-time-efficient hardware from constant qualified array definitions for various indirection involving array access patterns;
—ROCCC's procedure-level (i.e., global) optimizations achieving circuit area reductions of up to 88% compared to circuit areas generated from globally unoptimized codes.

The article is organized as follows: ROCCC framework is explained in Section 2. Section 3 introduces ROCCC's high-level transformations and optimizations. Section 4 discusses the impact of some of ROCCC's high-level transformations and optimizations on area, throughput, and clock cycle time within the ROCCC framework. Related work is described in Section 5. Finally, concluding remarks are offered in Section 6.

2. ROCCC OVERVIEW

ROCCC is a C to RTL-VHDL compilation framework for mapping application programs to FPGAs. The focus of ROCCC is generating highly parallel and

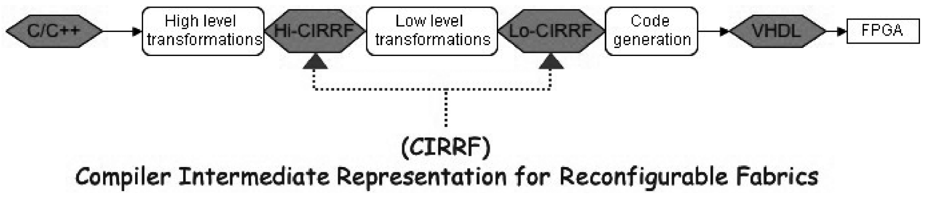

C/C++
High level transformations
Hi-CIRRF
Low level transformations
Lo-CIRRF
Code generation
VHDL
FPGA
(CIRRF)
Compiler Intermediate Representation for Reconfigurable Fabrics

Fig. 1. ROCCC framework.

optimized circuits rather than statement-by-statement translation of C programs to VHDL. One of the main distinguishing features of the ROCCC framework is its emphasis on compile-time loop transformations and optimizations. The objectives of ROCCC's optimizations are as follows: (1) maximize the parallelism in the circuit as well as the clock rate at which it operates and (2) minimize the number of off-chip memory accesses as well as the area of the overall circuit.

ROCCC is not designed to compile a whole program. It relies on the user to identify the functions that are the most compute-intensive in a given application. The functions are compiled to hardware and invoked via a specialized API that interfaces the FPGA fabric to the host processor. The remainder of the code executes in software on the host. We constrain the source code that will be translated to hardware as follows: no pointers, no break or continue statements, simple *for* loop headers where the loop counter of each loop level iterates from some lower bound to some upper bound in compile-time determinable steps, and that all array index expressions are in the form $loop\_counter \pm compile\_time\_constant\_stride$.

ROCCC is built on the SUIF2 [Aigner et al. 2010] and Machine-SUIF [Smith and Holloway 2010a] platforms. It compiles C code into synthesizable VHDL code for mapping onto FPGA fabrics. Information about loops and memory accesses is extracted from SUIF2's high-level Intermediate Representation (IR). Accordingly, most loop-level analysis and optimizations are done at the SUIF2 level. Most of the information needed to design high-level components, such as controllers and address generators, is also extracted from this level's IRs. ROCCC performs a very extensive set of loop analysis and transformations aiming at maximizing parallelism and minimizing area. Machine-SUIF is an infrastructure for constructing the back-end of a compiler. We modified Machine-SUIF's virtual machine (SUIFvm) IR [Smith and Holloway 2010b] to build our data flow code representation. All arithmetic opcodes in SUIFvm have corresponding functionality in IEEE 1076.3 VHDL with the exception of division.

We have added new analysis and optimization passes to SUIF2 and Machine-SUIF that target FPGAs. Specifically, taking the IR generated by SUIF2's front-end as input, our compiler detects and optimizes memory accesses. Our compiler also takes the IR generated by the Machine-SUIF front-end as input and generates the data flow. The IR of Machine-SUIF's IR is a low-level IR and resembles a three-address assembly language. The array access pattern information, which is obtained through memory reference analysis, combined

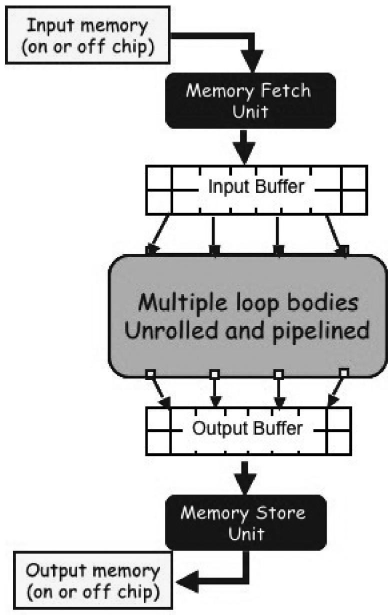

Input memory (on or off chip)
Memory Fetch Unit
Input Buffer
Multiple loop bodies Unrolled and pipelined
Output Buffer
Memory Store Unit
Output memory (on or off chip)

Fig. 2. ROCCC's execution model.

with the pipeline level each instruction belongs to on the datapath (which is created during data flow generation), is fed into the controller generation pass to generate controllers in VHDL. We rely on commercial tools to synthesize the VHDL code generated by our compiler.

ROCCC maps the computation onto reconfigurable logic in a decoupled manner as shown in Figure 2. For a given loop, ROCCC first decouples the memory accesses from computation using scalar replacement. Then from the optimized load and store array access expressions, smart buffers [Guo et al. 2004] are generated to store any input data that are already fetched and will also be used in later iterations, thereby reducing the number of memory accesses.

The ROCCC compiler builds and pipelines the data path [Guo et al. 2005] from the scalar replaced code. The compiler groups the instructions into different execution levels to exploit instruction (operation)-level parallelism. The instructions grouped at the same level are executed simultaneously. Every level is marked as latched or not latched according to the sum of the estimated delays from the most recent latched node. The latched instructions are translated into a sequential logic circuit while the not-latched ones are translated into a combinational circuit. A pipeline stage corresponds to the instructions between two latched levels. For a loop body with no loop carried dependencies, every level of the dataflow graph corresponds to one loop iteration instance once pipelined.

ROCCC allows the user to specify bit-width information for each declared array or variable, which is then stored into the symbol table. ROCCC then automatically performs bit-width resizing (i.e., reducing the number of bits used for each variable to the minimum necessary) based on input data bit-size and instruction opcodes.

3. HIGH-LEVEL TRANSFORMATIONS IN ROCCC

Optimizing compiler technology faces several challenges when it comes to compiling for reconfigurable platforms. First, the resources and performance constraints for FPGAs are drastically different than those of microprocessors, and compiling for FPGAs requires laying the computation in space as a circuit rather than in time as a sequence of instructions. Moreover, in traditional processors any computation has to follow the fetch, decode, execute, write-back cycle, which additionally imposes restrictions on a traditional optimizing compiler. On an FPGA none of these restrictions holds.

Table I lists the transformations that ROCCC performs. Although most of these transformations are same as they are in traditional compilers, 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. On an FPGA the only constraints to loop unrolling are the area and data bandwidth from memory. Thus, as long as there is enough area and I/O bandwidth to fit another iteration, the loop could be rolled for another time. Due to this growth in the scale of the optimizations, new challenges arise such as the optimization of the transfer of large amounts of data to the data path and synchronization of data flow across pipeline iterations where a feedback variable needs to be fed back into the datapath in the subsequent cycle.

HLL's are designed to express algorithms in a way to help the compiler to generate an efficient *software* implementation. However, the most optimal implementation of a given algorithm in hardware is not always the same as the software implementation of that algorithm. For instance, wavefront algorithms are usually computed in software using a 2D matrix where the enclosing loop advances in a fixed direction to compute the current cell using neighboring values computed in previous iterations. A fast implementation of the same algorithm in hardware would be a systolic array. Thus, a powerful HLL-to-HDL compiler should recognize relevant patterns from the HLL source code and be able to map them to their best hardware implementation counterparts.

ROCCC tries to address these challenges by providing the user with predefined optimization packages such as unroll, tile, generate systolic array, etc. These packages define the type of loop-level transformations that the user wants to apply to parallelize a given loop-nest. Each package comes with a set of parameters such as the tile sizes, the particular loop in the loop-nest to be unrolled, the systolic array size, etc. Each package contains forty to fifty passes, most passes from Table I given before, where some are applied repetitively over the source code at various phases during the compilation process. The contents and the inner workings of the individual packages are entirely transparent to the user. The rationale of this approach is to enable the user to explore the design space of various hardware mappings.

3.1 The User Interface

ROCCC comes with an easy to use user interface. ROCCC expects two files that are provided from the command line. First is the C file that contains the

Table I. ROCCC's Available Optimizations

Array Access	—Scalar Replacement —Array RAW/WAW Elimination —Array Renaming —Constant Array Propagation
Procedure Level	—Code Hoisting —Code Sinking —Constant Propagation —Algebraic Identities Simplification —Constant Folding —Copy Propagation —Dead Code Elimination —Unreachable Code Elimination —Scalar Renaming —Reduction Parallelization —Approximation of Division/Multiplication by Constant —If Conversion
Loop Level	—Normalization —Invariant Code Motion —Peeling —Full & Partial Unrolling —Fusion —Tiling (a.k.a. Blocking) —Strip Mining —Interchange —Unswitching
Application Specific	—DFA State Table Expansion —Lookup Table Expansion —Systolic Array Generation

procedure/function, which has the loop to be sent to hardware together with all the *.h* files containing the declarations for all data types used inside the procedure, and second is a *.pass* text file, which specifies a list of loop-level transformations that the user wants to apply to the candidate loop nest. The C file must conform to the following specifications:

—No pointers, break, continue, return, or exit statements inside the candidate loop body;
—Simple for loop headers, where the loop lower bound, upper bound, and the step count are compile-time known constants;

SOURCE CODE	PASS COMMANDS

`begin_hw();`
`L1: for(i = 0; i < 1024; i=i+1)`
`  L2: for(j = 0; j < 1024; j=j+1){`
`        sum = 0;`
`        L3: for(n = 0; n < 5; n=n+1)`
`          L4: for(m = 0; m < 3; m=m+1)`
`                sum = sum + (image[i+n][j+m]*`
`                             filter[n])/8;`
`          output[i][j] = sum;`
`      }`
`end_hw();`
	`fully unroll L3`
	`fully unroll L4`
	`generate tile L1 L2 4 4`

`begin_hw();`
`L1: for(i=0; i<=511; i=i+1)`
`      B[i] = T[0]*A[i] + T[1]*A[i+1] + T[2]*A[i+2] +`
`             T[3]*A[i+3] + T[4]*A[i+4];`
`end_hw();`
	`partially unroll L1 15`

Fig. 3. Two sample C code snippets with ROCCC recognized macros and C programming labels (highlighted) and their corresponding *.pass* files listing loop-level transformation examples through the highlighted labels.

—All array index expressions are in form *loop_counter* ± *loop_step_size*;
—The candidate loop is enclosed in between *begin_hw*() and *end_hw*() calls.

Loop headers are labeled using regular C programming labels. Thus, the user compiles the ROCCC adapted code using their favorite C compiler. Figure 3 shows a sample labeling of loops in C. The candidate loop is wrapped between *begin_hw*() and *end_hw*() calls, which are two dummy calls that help ROCCC recognize where the candidate loop nest starts and ends, respectively.

In addition to the C file containing the procedure with the candidate loop-nest, the user needs to provide the compiler with a *.pass* file, which is a text file that consists of one or more of the following commands.

—*fully unroll loop_label*: Fully unrolls the loop that is labeled with *loop_label*. The loop specified in the parameter needs to have a compile-time constant trip count. The fully unroll command should be used with caution, as the final code after the optimizations should still be enclosed within at least one loop. ROCCC currently does not compile to VHDL code that is not enclosed in loops. So, one cannot fully unroll a loop, if that loop is the only loop in the code. Loop jamming is automatically done by ROCCC following loop unroll calls.
—*partially unroll loop_label unroll_factor*: Partially unrolls the loop labeled with *loop_label*, $unroll\_factor$ many times. An $unroll\_factor$ of three generates a total of four copies of the loop body in the output (i.e., original loop body + three unrollings) and a new step that is *original_step_size* + 3.

ROCCC automatically performs loop jamming over the outputs of the loop unroll commands.

—*generate tile loop_label1 loop_label2 tile_size_1 tile_size_2*: Generates a tiled version of the specified 2D loop. The parameters *tile_size_*1 and *tile_size_*2 specify the unroll factors of loops *loop_label*1 and *loop_label*2, respectively.

—*generate systolic array loop_label1 loop_label2 systolic_array_size*: Generates a systolic array of size *systolic_array_size* aligned along the loop labeled as *loop_label*2.

More packages are being developed and support for them is being added to the ROCCC front-end. Figure 3 shows two sample for-loops labeled and wrapped *begin_hw*() and *end_hw*() calls together with an accompanying sample *.pass* files. Using the *.pass* file, ROCCC customizes a phase-ordered, complete list of transformations to be applied to the C code. The final list is composed of forty to fifty passes, including several of the passes from the list provided in Table I. The contents and the inner workings of the formed list are entirely transparent to the user and automatically applied to the source code with a single command line call that includes the *.c* and the *.pass* files.

ROCCC does not employ an automatic parallelizer. There are few reasons for this. One of the reasons is that, depending on the specific FPGA family that the design is targeted at, the amount and the location of the resources on the FPGA change and vary greatly. Moreover, it would not always be possible to know at compile-time with how many other circuits the code would be sharing the FPGA at runtime. Giving the user the ability to modify the design parameters through the *.pass* file allows the user to have more control over the circuit size, algorithm design, and the final performance of the generated circuit, while making ROCCC portable across many FPGA platforms.

3.2 Memory Access Optimizations in ROCCC

Memory bandwidth is one of the two most important factors that can limit performance on an FPGA. Hence, compiler transformations that eliminate unnecessary memory accesses and promote data reuse are extremely valuable in increasing the performance and parallelism on the FPGA. ROCCC's array transformations eliminate most unnecessary memory accesses from the source code by: (1) propagating true data dependencies through scalar temporaries and (2) decoupling memory accesses from computation.

Figure 4 illustrates four cases where the highlighted array accesses are unnecessary. In Figure 4(a) the true data dependencies (RAW) and multiple stores (WAW) to the same address are eliminated through the use of scalar temporaries. The initial store and the load to $A[i]$ are both replaced by scalar variable t. In the second case illustrated in Figure 4(b), array read and write accesses to the same array carry no true data dependencies. Thus, ROCCC renames one of the arrays to a different name. This renaming process allows the second array to be assigned to a separate memory bank on the FPGA, thereby reducing the contention on the bandwidth of the bank where the read accesses are sent.

```
BEFORE                                     AFTER

for(...){                                  for(...){
   A[i] = ...;                                t = ...;
   ... = ... A[i] ...;          ⟹            ... = ... t ...;
   A[i] = ...;                                A[i] = ...;
}                                          }
```

(a) elimination of RAW and WAW memory accesses

```
for(...){                                  for(...){
   ... = A[i+1];                              ... = A[i+1];
   ...                          ⟹            ...
   A[i] = ...;                                B[i] = ...;
}                                          }
```

(b) array renaming to eliminate contention to the same on-chip memory bank

```
const int A[3] = {-1,0,1};                 const int A[3] = {-1,0,1};
for(...){                                  for(...){
   ... = ... A[1] ...;          ⟹            ... = ... 0 ...;
}                                          }
```

(c) Folding of constants from arrays defined with the *const* qualifier

```
for(i=1; ... ; i=i+1){                     t = A[0];
   ... = A[i-1];                           for(i=1; ...; i=i+1){
   ...                          ⟹            ... = t;
   A[i] = ...;                                ...
}                                             t = A[i] = ...;
                                           }
```

(d) elimination of feedback memory accesses for hardware systolic array generation

Fig. 4. Example cases of memory access optimizations available in ROCCC.

Figure 4(c) illustrates the case which mostly takes place in DSP-like and image processing applications. Such application codes contain *const* qualified array definitions, which are masks applied at every iteration to instances of a sliding window over some input stream. The index expressions of these *const* qualified array accesses are usually compile-time constants or become a constant once the innermost loop enclosing the array expression is fully unrolled. ROCCC replaces such array accesses with actual constants from the *const* qualified array, eliminating these memory accesses.

The case illustrated in Figure 4(d) is utilized during systolic array generation in ROCCC. Whenever a value that is written to an array location is accessed by the loop body in the subsequent iteration, a hardware register (i.e., t) is created for the stored value and the load from the memory is replaced by a load from the scalar register that is on the reconfigurable hardware. Later, accesses to variable t are enclosed within two macros, namely ROCCC_load_prev and ROCCC_store2next, which are internal to our compiler. These two macros are inserted into the final code to reflect the effects of the eliminated feedback

```
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] + 11*A[ i+4] ;
}
```

(a)

```
for(i=0; i<N; i=i+1){
    x0 = A[ i] ; x1 = A[ i+1] ; x2 = A[ i+2] ;      //decoupled
    x3 = A[ i+3] ; x4 = A[ i+4] ;                   //read accesses

    y = 3*x0 + 5*x1 + 7*x2 + 9*x3 + 11*x4;          //datapath

    C[ i]  = y;                                     //decoupled
}                                                   //write accesses
```

(b)

Fig. 5. Scalar replacement example over 5-tap FIR code.

load/store operations. ROCCC_load_prev loads a value from the iteration before and ROCCC_store2next saves its input to be read during the next iteration. Both macros are translated first into special machine instructions defined in MachineSUIF, followed by VHDL code at later stages. In the generated code, both macros read from and write to on-chip registers.

ROCCC maps the computation on the reconfigurable logic in a decoupled manner as shown in Figure 2. Once all the unnecessary array accesses are eliminated, all the memory accesses are decoupled from the actual computation using scalar replacement, which moves the array read accesses to the beginning of the loop and array writes to the end of the loop. Later all the array read and write accesses are replaced with smart buffer macros. Smart buffers [Guo et al. 2004] help exploit data reuse found in loops by storing on-chip the parts of the input data that will be reused in subsequent iterations. To illustrate further, the 5-tap-FIR code shown in Figure 5 reads five array elements per iteration, of which four were already fetched in the iteration before. Instead of issuing five new memory reads to the memory every iteration, smart buffer issues only one to the memory and retrieves the remaining four from its on-chip registers.

4. IMPACT OF HIGH-LEVEL TRANSFORMATIONS ON AREA, THROUGHPUT AND CLOCK CYCLE TIME

4.1 Procedure-Level Transformations

In instances of loops that do not carry dependencies from earlier iterations, the parallelism on an FPGA is limited only by the area available for the datapath on the FPGA considering there is sufficient on-chip I/O bandwidth. In most embedded applications, the critical parameter is the throughput. ROCCC's procedure-level transformations provide substantial help in minimizing the circuit area per loop iteration, which may allow the compiler to unroll and fit

Table II. Descriptions of Benchmark Codes Used in Evaluating Impact of Procedure-Level Transformations within the ROCCC Framework

Name	Dim.	Bitwidth	Description
Fir5	1D	32-bit	5-tap constant-coefficient finite-impulse-response filter
Mf9	1D	32-bit	9-tap moving average filter
Wf	1D	16-bit	Weighting filter kernel from Mi-bench
Bitcount1	1D	32-bit	Counts the number of 1's in an array element and saves the result into another array
9x7-Filter Code	2D	32-bit	9x7 Image processing filter
Face Recognition	2D	32-bit	Multiplication of a streaming window against two matrices

more iterations into the available area. Reduced circuit area per loop iteration may cause the clock cycle time to decrease, shorten the critical path, and thus could lead to substantial increases in throughput. ROCCC's extensive set of procedure-level transformations help reduce the circuit area in various ways. ROCCC:

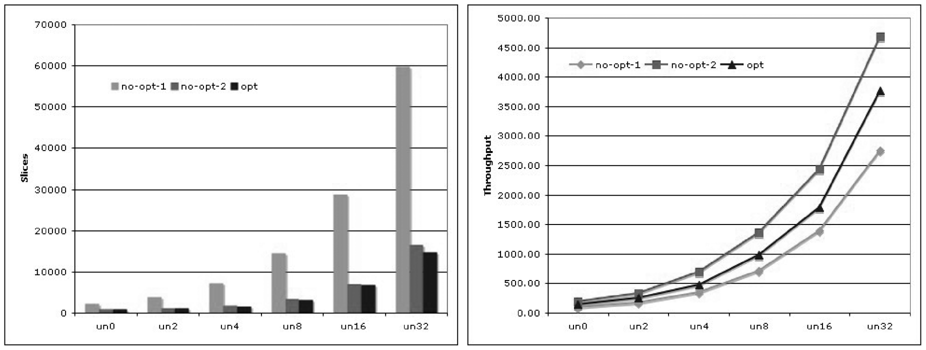
(1) Computes all arithmetic computations, whose operands are known or can be simplified, at compile-time (constant folding, algebraic identity simplification);
(2) Hoists code that computes the same value each time it's executed out loops and/or if statements (code hoisting, invariant code motion);
(3) Shortens the critical path by parallelizing reduction operations, where the operator is both commutative and associative, such as sum over an array (reduction parallelization);
(4) Transforms the expensive division/multiplication operations by constants into a sum of right/left shifts by various powers of 2 that approximates to the constant divider/multiplier (approximation of divisions/multiplications by constant);
(5) Simplifies the control flow by eliminating redundant code and converting branches to data flow operations (unreachable and dead code elimination, if conversion).

We used the six benchmarks listed in Table II to study the effect of ROCCC's procedure-level optimizations on the area, clock speed, and throughput of the ROCCC generated VHDL. All benchmark codes are kernel loops written in C and did not contain any code that is not part of the computation. ROCCC's procedure-level optimizations are applied as a set, therefore the effect of each optimization is not studied individually. The reason for this is that compiler optimizations usually trigger one another and one pass would help others find more opportunities in simplifying the code further. Most of the time, different sets of optimization passes are applied in cycles until no more changes are observed in the generated code. Hence, it was not possible in our experience to study the effect of each optimization individually on the VHDL generated from the listed benchmark codes in C.

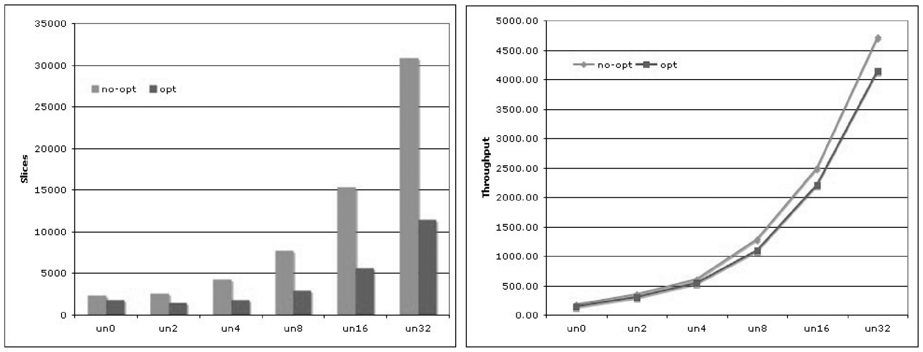
The source codes are directly read into the SUIF2 intermediate format and the optimizations described in the previous section are applied on the SUIF2 IR with the exception of the procedure-level optimizations. We generated two versions of each benchmark where in the opt case all the procedure-level optimizations are turned on and in the no-opt case all the procedure-level optimizations were turned off during the compilation process. In order to amplify the effect of the optimizations we unrolled both opt and no-opt cases several times. While performing unrolling we assumed that the I/O between the data path and the on-chip memory is sufficient, since the required data bus width increased with unrolling. The 1D benchmarks, which are single nest loops, are unrolled for 0, 2, 4, 8, 16, and 32 times, while the 9x7-Filter code is a doubly nested loop and is unrolled for 0x0 (no unrolling) and 2x2 (unrolled twice on the outer and twice on the inner loop) times. The Face Recognition benchmark consists of a doubly nested loop nest that had the inner loops fully unrolled and the outer loop unrolled 0, 2, 4, 8, and 16 times. unX labels on the figures indicate an unrolling factor of X applied to 1D benchmarks and unXxY labels indicate that an unrolling factor of X applied to the outer loop and an unrolling factor of Y to the inner loop, respectively. Throughput is measured in terms of number of elements from the input array processed per second. In the case of the face recognition code, throughput is measured in the number of 9x6 windows processed per second.

We synthesized and place-and-routed the generated VHDL for the Xilinx Virtex 4 LX200 FPGAs found on the SGI RASC RC100 blades [SGI Inc. 2010]. We used Xilinx ISE 8.2 to synthesize and place-and-route the generated VHDL code with the exception of the Bitcount and the Face Recognition benchmark, where the synthesize tool used was Xilinx ISE 9.1. Figure 6, Figure 7, and Figure 8 show our results. Our results show that the circuit area of the ROCCC-optimized C kernel codes was up to 88% smaller than the circuit area that is generated when the compiler optimizations are turned off.

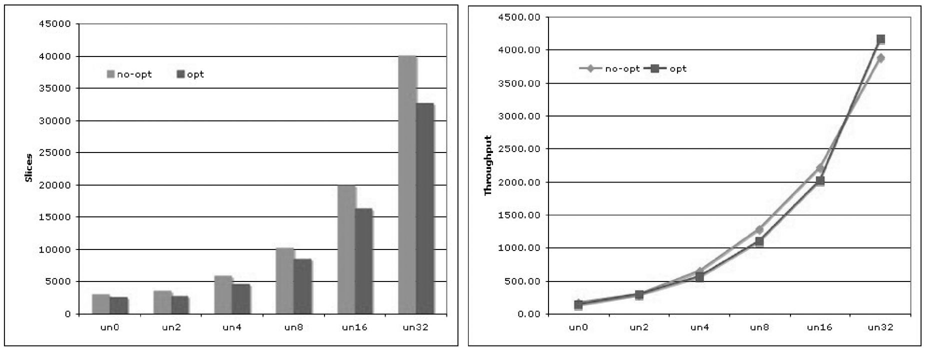
Figure 6(a) displays the Fir5 results. We wrote the Fir5 C code in two different forms. In the no-opt-1 case the constant-coefficients of the FIR are placed in a 1D const qualified array as a good software practice, whereas in the no-opt-2 case the constant coefficients were hard coded inside the loop body. The no-opt-2 code is shown in Figure 5. In the generated VHDL from no-opt-1 C code, the constant-coefficients were fetched from the host application to the reconfigurable hardware and then routed to the multipliers. The routing as well as the multiplier circuit caused the area of the generated circuit to increase. In the generated VHDL from no-opt-2, although the routing of constant-coefficients was avoided, the multipliers still occupied some area on the hardware. ROCCC compiled and optimized both C codes to the exact same VHDL code. The data legend opt shows the effect of the procedure-level optimizations. In the optimized version, the const qualified array entries are folded into the expression as compile-time constants and the multiplications by constants are transformed into shifts and addition sequences. The optimized circuit achieved area reduction up to 77% over the no-opt-1 and 10% over the no-opt-2. This is the only benchmark we had where we observed in the throughput a penalty of up to 32% over no-opt-2. We believe that the five multiplications by small



(a)



(b)



(c)

Fig. 6. Area and throughput results for optimized and nonoptimized (a) Fir5, (b) Mf9, and (c) Wf kernel.

constants produced a more efficiently synthesizable code than the sequence of 7 shifts and 11 adds, which replaced the five multiplications in the optimized circuit.

Figure 6(b) shows Mf9 results. Mf9 is a 9-tap moving-filter code computing the arithmetic average of nine values. In the no-opt case the division by 9 is

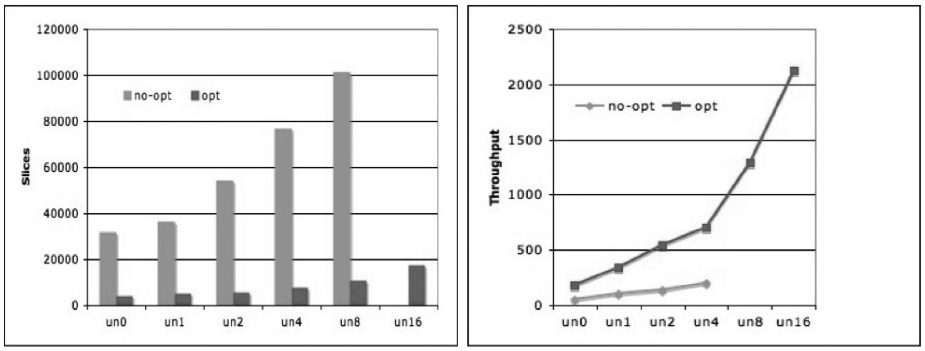


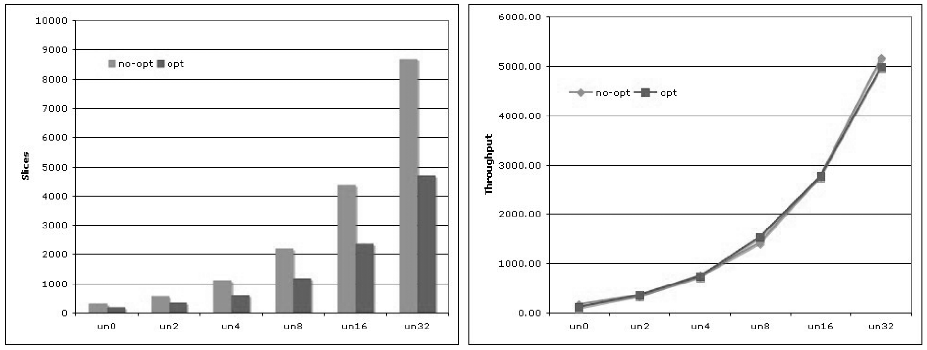
Fig. 7. Area and throughput results for optimized and nonoptimized Face Recognition codes.

left unoptimized and ROCCC's back-end replaced it with a single-cycle-delay integer division circuit, whereas in the optimized version, the division by 9 got transformed into a right-shift/add sequence. This single optimization reduced the area by 63% over the no-opt case. For larger unroll factors the throughput observed a slight penalty over the no-opt-2 for the same unroll factors. However, since the optimized circuit of un32 occupied less area than the did the unoptimized un16, the throughput per area overall increased.

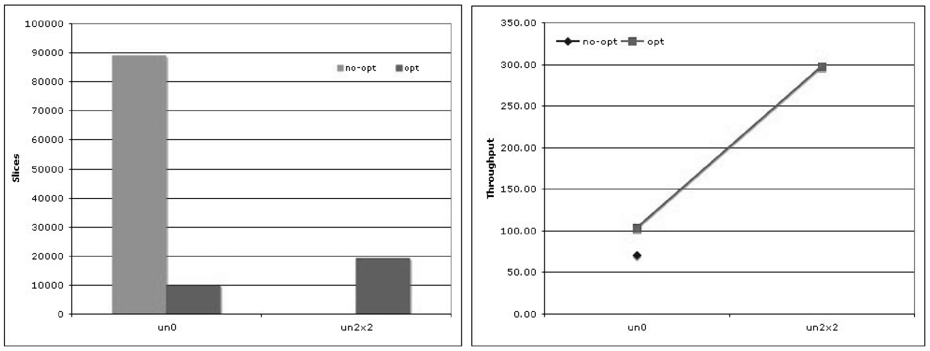
Figure 6(c) shows the Wf results. Wf is a weighting-filter code taken from the GSM benchmark in MiBench benchmark suite [Guthaus et al. 2001]. The original source code contained multiplications by quite large constants. ROCCC replaced these multiplication by constants with shifts and adds during the compilation process. Moreover, ROCCC also detected few opportunities for constant propagation and dead code elimination as a consequence of the former transformation. The ROCCC-optimized circuit achieved an area reduction up to 23% over the no-opt case. The throughput stayed almost the same across same unroll factors.

Figure 7 shows the face recognition results. The face recognition code performs matrix multiplication between a 1x9 matrix and a 9x6 matrix, followed with a multiplication of the result with a 6x1 matrix, resulting in a single value for each window. The 1x9 matrix and 6x1 matrix are constant while the 9x6 is a streaming window over a much larger matrix. In the no-opt case, the constant matrices are not propagated into the datapath code and are read as input from the host program, in the opt case all constant values are propagated and folded into the datapath generated. The original C code consists of two consecutive inner loops to perform the matrix multiplications inside of a larger loop that streams the window data. We fused and fully unrolled the inner loop in all cases and unrolled the outer loop 0, 1, 2, 4, 8, and 16 times to process more windows at once. ROCCC detected many chances for constant propagation and folding and replaced many multiplications with shifts and adds, resulting in a large reduction of area.

The nonoptimized version of the Face Recognition code when unrolled 8 times took up 112% of the LX200 chip and as such could not fit on the FPGA. The number of slices shown in Figure 7 for the unrolled 8 times case represents



(a)



(b)

Fig. 8. Area and throughput results for optimized and nonoptimized (a) Bitcount and (b) 9x7-Filter kernel codes.

the number reported by synthesis and not place-and-route. When unrolled 16 times, the optimized version still fit on approximately 52% of the FPGA while the unoptimized version did not fit on the FPGA. The difference in area on average is 88% between the optimized and nonoptimized circuits.

Throughput results reflect a 50 MHz clock for the nonoptimized hardware versus a 180 MHz clock for the optimized hardware. Throughput results could not be determined for the unroll 8 and unroll 16 instances which did not fit on the LX200 FPGA.

Bitcount results are provided in Figure 8(a). Bitcount is kernel code taken from Bitcount application in MiBench benchmark suite. It counts number of 1's in the elements of the input array and saves the result into another array. Source code contains shift, bitwise-and, and accumulation of the detected 1's. ROCCC automatically detected the accumulation as a reduction operation and applied reduction parallelization to the source code. ROCCC also detected couple constant and copy propagation and dead code elimination opportunities over the result of the reduction parallelization pass. The ROCCC-optimized circuit achieved an area reduction of up to 46% over the no-opt case. We noticed that the main savings in area was due to ROCCC's reduction parallelization pass. The throughput again stayed almost the same across same unroll factors.

9x7-Filter was implemented as a doubly nested loop. The results for 9x7-Filter are provided in Figure 8(b). The original source code applied a mask that is of size 9x7 to every 9x7 window over a 2D image. This particular kernel code contained several procedure-level optimization opportunities. First, the mask array is read from a const qualified array, which later got automatically folded into the expressions by ROCCC. Second, the constant mask was used to multiply the imput image elements and these multiplications are either eliminated incases where the constants were 0s and 1s or were transformed into shift/add sequences. Third, the result of the multiplications were adjusted by dividing the multiplication result to some constant number. This division by constant also got transformed into shifts and adds. Finally, the computed values were summed up and a single value was produced for each 9x7 window of the image. ROCCC automatically detected the summation as a reduction operation and applied reduction parallelization to the IR resulted from the previous transformations. Our ROCCC-optimized circuit achieved an area reduction of 88% over the no-opt un0x0 uncase. The unoptimized un0x0 circuit area occupied almost the entire FPGA area, as a consequence we were not able to synthesize and place-and-route the unoptimized un2x2 circuit onto our target platform.

Our results show that even tight kernel loops contain vast opportunities for optimizations when compiling to hardware. ROCCC first unrolls the loops to increase optimization opportunities and then applies procedure-level transformations over the unrolled code. Our results in this section prove that our approach successfully compiles loops taken from DSP and MiBench applications to area-efficient, high-throughput circuits.

4.2 Loop Unrolling

Loop-level optimizations help achieve most of the parallelism in ROCCC. Loops that do not carry dependencies from earlier iterations can theoretically be fully unrolled to achieve maximum parallelism. Since full unrolling may not always be feasible due to the limited resources on the FPGAs, ROCCC provides several passes to the user for optimal unrolling such as unroll-jam, tiling (a.k.a. blocking), and strip-mining. 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. ROCCC currently gets the unrolling factor from the user as a optimization package parameter.

Loop unrolling techniques help expose data- and instruction-level parallelism opportunities inside the loop, yet the same techniques increase 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.

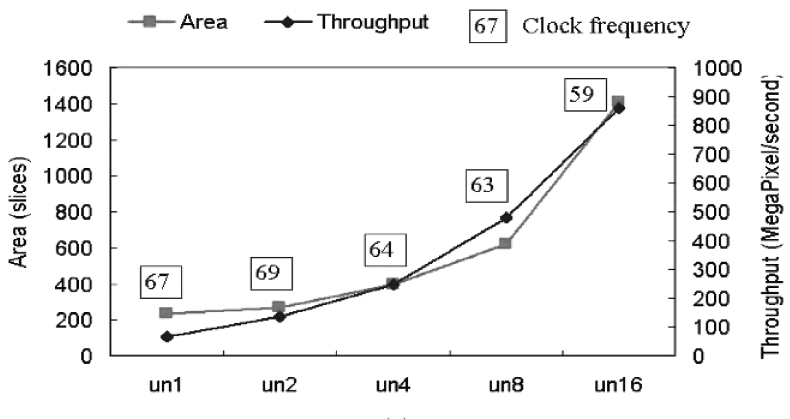
In this section we study the effect of loop unrolling on the area, clock speed, and throughput within ROCCC, C-to-VHDL compilation framework. Our results indicate that due to the unique design of the ROCCC compilation framework, FPGA area either shrinks or increases at a very low rate for the first few 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.

We used five benchmarks to show the effect of unrolling in area, clock cycle, and throughput. 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. 5x3-Filter code is a doubly nested loop operating on a 5x3 block of pixels. Moravec 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 5x3-Filter and Moravec operate on 2D arrays. Fir5, Fir15, Mf9, 5x3-Filter and Moravec are all kernel loops.

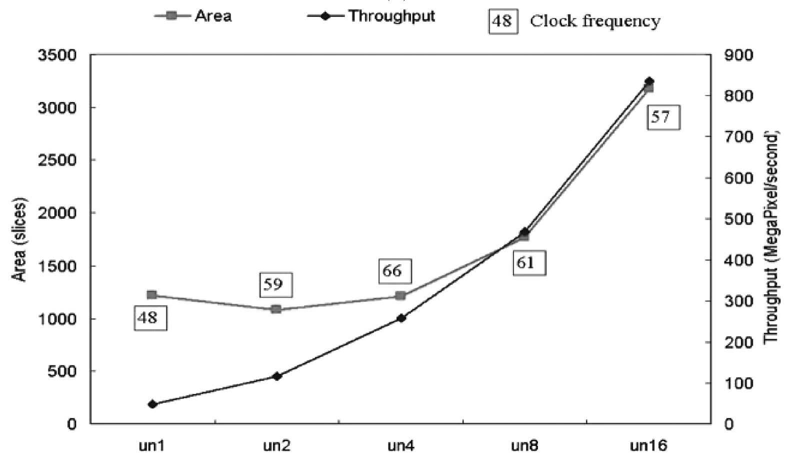
We used Xilinx ISE 6.2.03i to synthesize and place-and-route the generated VHDL code. Our target was the Xilinx Virtex-II xc2v8000-5 FPGA. The source codes are directly read into the SUIF2 intermediate format and ROCCC's procedure-level and array access optimizations described in previous sections are fully applied. Further, we assumed that the I/O between the data path and the on-chip memory is sufficient, when performing unrolling, since the required data bus width increases with unrolling. The 1D benchmarks are unrolled for 2, 4, 8, and 16 times. Moravec is unrolled for 2x2 and 4x4 times. Finally, the 5x3-Filter 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.

Figure 9 displays our area results for the 8-bit data size versions of the benchmarks that execute on one-dimensional arrays. The area results on these figures show the combined areas of the data path, 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 9(b), and 9(c) the not unrolled cases are not the minimal area points. This shows that there exist optimal times to unroll.

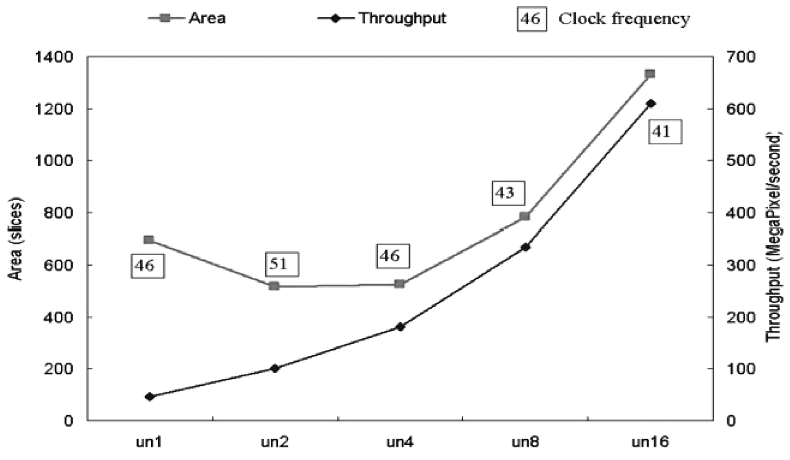
Figure 10(a) shows the results of unrolling a 5x3-Filter code. Note that an unrolling of 8x8 means that the 5x3 block is replicated eight times in each direction. In other words 64 windows of 5x3 are operated simultaneously. From no unrolling to 64 concurrent loops the area grows by 12 while the throughput 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.



(a)

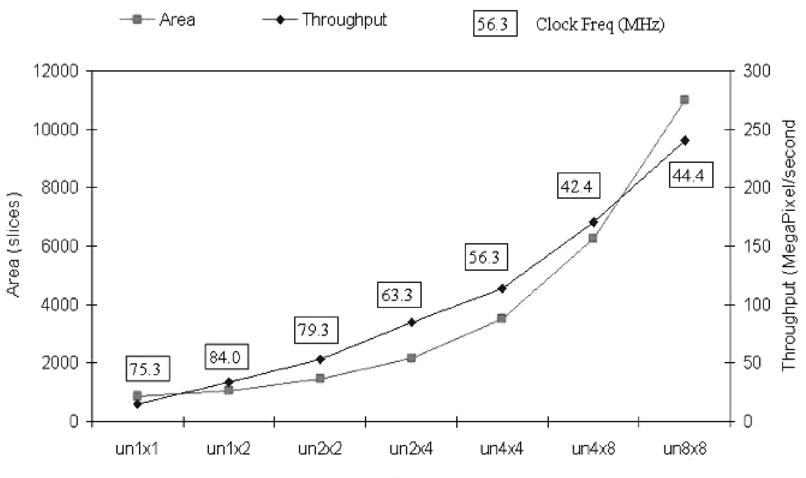


(b)

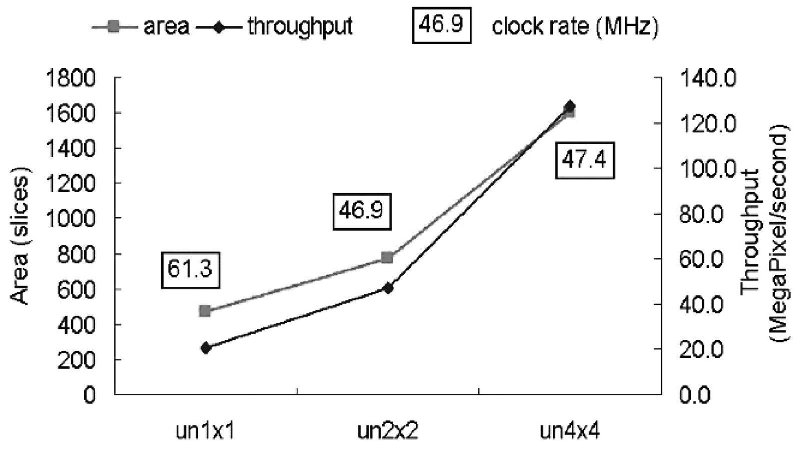


(c)

Fig. 9. Area, clock frequency, and throughput for (a) Fir5, (b) Fir15, and (c) Mf9.



(a)



(b)

Fig. 10. Area, clock frequency, and throughput for (a) 5x3-Filter and (b) Moravec.

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 5x3-Filter code. Thus, the unrolled version operates not just on one more set of input data as it would be over a 1D array, but three more sets of input data since it is twice unrolled towards both directions over a 2D array.

Table III 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 3 and 4. 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

Table III. 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
5x3-Filter	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

smart buffer circuits onto 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. This procedure would not give a correct area estimate of the area occupied by the control logic, however, it shows where the reduction in area came from. We did not think 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 preexisting parameterized FSMs in a VHDL library, whose size does depend on the unrolling factor.

According to the figures in Table III, the circuit area of the data path increased almost at a linear rate, although it is known that loop unrolling introduces more opportunities for optimizations especially on the data path. The reason for the linear increase is that all the data path codes are mapped after being decoupled from all memory accesses and all address computation code, and after 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 data path. For the un1 case, the number of windows in the smart buffer is large, although at any given time 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 data path per clock cycle. Since the control logic size diminishes, the overall area for the smart buffer decreases.

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 the overall throughput decreasing. The number of parallel iterations generated by high unroll factors implies that the number of outputs generated per clock cycle on a pipelined data path increases. Thus, the effect of the clock rate decrease with increased unrolling is overcome with the increased parallelism in the unrolled codes.

4.3 Hardware Circuit Generation for Applications Using A[B[index_expr]]-Style Nonlinear Array Accesses

A[B[index_expr]]-style indirect, nonlinear array accesses are frequently used in many CPU-intensive applications such as in cryptography as key dependent substitution boxes, in image processing as color palettes, in bioinformatics as score matrices in sequence alignment, and in various other scientific computing applications. Such array accesses are usually made into constant qualified arrays where the values of an incoming data stream are used to index into the constant table. This style of array or table lookup helps speed up programs by retrieving a costly-to-compute function's precomputed values from memory instead of recomputing them each time a value is needed. Such table lookups either map the actual index number of the value in the array or some specific keys to values. In the latter case an associative array is used in order to represent the mapping from the collection of keys to the collection of output values. An example of an associative array lookup is found in protein sequence alignment applications where two input streams composed of characters from a protein alphabet are compared against one another. The characters read from the two streams are first converted into numeric indices of a 2D lookup table, that is, the scoring matrix. Then, a value is returned from the scoring matrix using the derived indices.

ROCCC allows users to declare and compile table lookups to hardware using two macro calls, that is, ROCCC_create_lookup_table() and ROCCC_lookup_in_table(). ROCCC_create_lookup_table() is placed somewhere above the candidate loop nest but not outside of the scope of the procedure enclosing the candidate loops. ROCCC_ lookup_in_table is placed within the loop body and replaces the actual array access (i.e., the actual lookup) operation. Figure 11 lists the parameters of both and Figure 12 and Figure 13 give examples on how to use the macro calls.

The first argument to the ROCCC_create_lookup_table() is a user-specified id, which links the specified table description to the actual lookup operation (i.e., the ROCCC_lookup_in_table() use). The second argument specifies a constant,

```
void ROCCC_create_lookup_table(int lookup_table_id,
                               const <type> lookup_table_values,
                               int row_count,
                               int is_row_an_associated_array,
                               const <type> row_key_collection,
                               int column_count,
                               int is_column_an_associated_array,
                               const <type> column_key_collection);

int ROCCC_lookup_in_table(int lookup_table_id,
                          <type> row_index_stream,
                          <type> column_index_stream);
```

Fig. 11. Generic prototypes of ROCCC macros for defining and using table lookup operations in user-level C source code; (`<type>` to be replaced by any primitive data type in C).

initialized array which contains the constant table contents and can either be a 1D or a 2D array. The third argument tells the compiler the number of elements or the row count if the lookup operation is to be done on a 2D array. The presence or the absence of the fourth argument specifies if the input stream indexes the constant table directly or needs to go through an associative array to find out, to which index the input key corresponds. In the presence of associative arrays, the input stream is first processed by a form of CAM (Content Addressable Memory) on the hardware to find out to which integer index the input data corresponds. Whenever the fourth argument is one, the fifth argument specifies the name of the constant initialized array of key values, which is assumed to perfectly align with the constant table values specified in the first argument. The remaining three arguments to the ROCCC_create_lookup_table() call carry the same meanings as those of the third through fifth arguments, except that they describe the column properties of 2D table lookups.

ROCCC takes the high-level description of an indirect array lookup described through macro calls and replaces them with MUX/CAM circuits defined through lower-level macros that are internal to ROCCC. The arguments of the ROCCC_ lookup_in_table() forms the select lines to the MUXes and CAMs and the inputs of the MUX/CAM circuits are taken from the constants listed in the constant qualified 1D or 2D lookup table arrays. Figure 14, Figure 15, and Figure 16 illustrate what the ROCCC generated circuits look like for the five classes of indirect table lookups ROCCC can map to hardware.

Figure 17 shows the area growth of the ROCCC generated circuits with 1D nonassociative table lookups. 1D table lookups are mostly found in cryptography applications as key-dependent substitution boxes. Figure 17 displays the growth in circuit area of a loop body that streams in elements of size 32 bit. The input stream indexed into a 1D array and returned a compile-time constant value from the hardware circuit, every clock cycle. The area is measured in slices and the horizontal axis indicates the number of elements stored inside the hardware circuit. To efficiently and effectively map the 1D nonassociative table lookup descriptions to hardware, only the necessary least significant bits of the index expressions are used to index the hardware table lookup descriptions. For instance, the least significant five bits of any index expression are

```
// SOURCE CODE IN C
const int A[4]= {92, 54, 91, 83};
int S[N], R[N];

begin_hw();
ROCCC_create_lookup_table(1, A, 4);
L1: for(i=0; i<N; i=i+1)
    R[i] = ROCCC_lookup_in_table(1, S[i]);
end_hw();

                    // ROCCC GENERATED LOWER LEVEL MACRO SEQUENCE
                    for(i=0; i<N ; i=i+1){
                        // input buffer definitions
                        row_tmp0 = ROCCC_convert(S_i, 2);
                        R_i = ROCCC_mux4(92, 54, 91, 83, row_tmp0);
                        // output buffer definitions
                    }
```

(a) example of 1D nonassociative indirect array lookup definition in C and its corresponding ROCCC generated lower-level macro sequence

```
// SOURCE CODE IN C
const int KEY[4]= {33, 24, 31, 23};
const int A[4]= {92, 54, 91, 83};
int S[N], R[N];

begin_hw();
ROCCC_create_lookup_table(1, A, 4, 1, KEY);
L1: for(i=0; i<N; i=i+1)
        R[i] = ROCCC_lookup_in_table(1, S[i]);
end_hw();

                    // ROCCC GENERATED LOWER LEVEL MACRO SEQUENCE
                    for(i=0; i<N ; i=i+1){
                        // input buffer definitions
                        row_tmp0 = S_i;
                        idx0 = ROCCC_cam4(33, 24, 31, 23, row_tmp0);
                        R_i = ROCCC_mux4(92, 54, 91, 83, idx0);
                        // output buffer definitions
                    }
```

(b) example of 1D associative indirect array lookup definition in C and its corresponding ROCCC generated lower-level macro sequence

Fig. 12. 1D indirect array lookup definitions in C and the accompanying ROCCC generated lower-level macros of each.

adequate to index a 1D nonassociative table lookup description of 32 elements. The hardware circuit then returns the location addressed by the index expression. The generated circuit is only a simple MUX whose elements are initialized from the const qualified array specified in the macro calls. Figure 14(a) illustrates what the ROCCC generated circuits look like for the 1D nonassociative

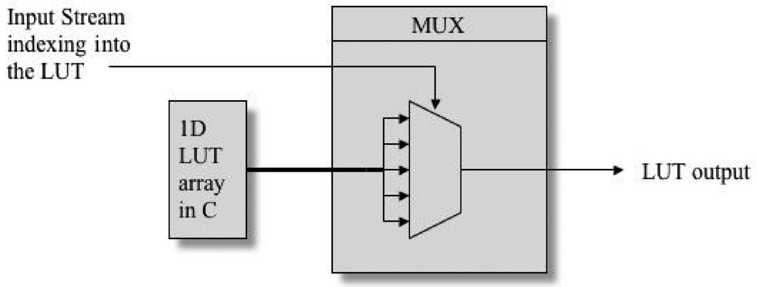
```
// SOURCE CODE IN C
const int A[2][2]= {{92, 54},
                    {91, 83}};
int S[N], T[N], R[N];

begin_hw();
ROCCC_create_lookup_table(1, A, 2, 2);
L1: for(i=0; i<N; i=i+1)
    R[i] = ROCCC_lookup_in_table(1, S[i], T[i]);
end_hw();

                    // ROCCC GENERATED LOWER LEVEL MACRO SEQUENCE
                    for(i=0; i<N ; i=i+1){
                        // input smart buffer macros
                        row_tmp0 = S_i;
                        col_tmp0 = T_i;
                        tmp0 = ROCCC_mux2(92, 54, col_tmp0);
                        tmp1 = ROCCC_mux2(91, 83, col_tmp0);
                        R_i = ROCCC_mux2(tmp0, tmp1, row_tmp0);
                        // output buffer definitions
                    }
```

(a) example of 2D nonassociative array lookup definition in C and its corresponding ROCCC generated lower-level macro sequence

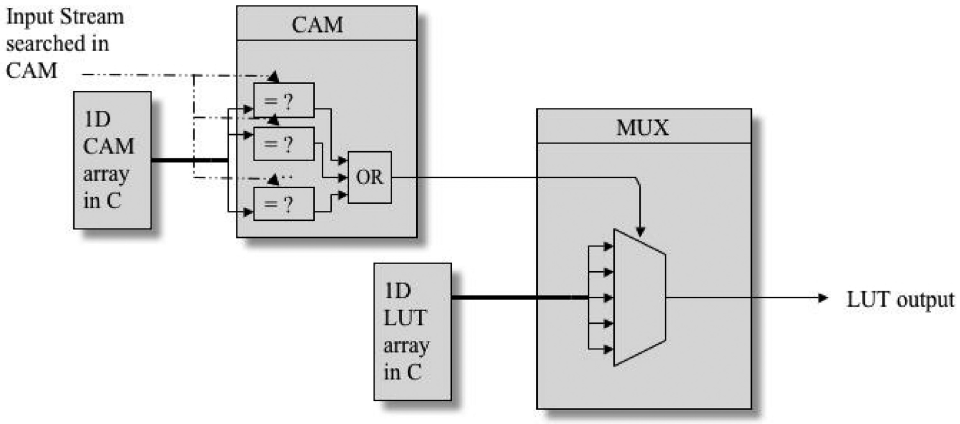
```
// SOURCE CODE IN C
const int KEY[2]= {33, 24};
const int A[2][2]= {{92, 54},
                    {91, 83}};
int S[N], T[N], R[N];

begin_hw();
ROCCC_create_lookup_table(1, A, 2, 1, KEY, 2);
L1: for(i=0; i<N; i=i+1)
    R[i] = ROCCC_lookup_in_table(1, S[i], T[i]);
end_hw();

                    // ROCCC GENERATED LOWER LEVEL MACRO SEQUENCE
                    for(i=0; i<N ; i=i+1){
                        // input buffer definitions
                        row_tmp0 = S_i;
                        col_tmp0 = T_i;
                        idx0 = ROCCC_cam2(33, 24, row_tmp0);
                        tmp0 = ROCCC_mux2(92, 54, col_tmp0);
                        tmp1 = ROCCC_mux2(91, 83, col_tmp0);
                        R_i = ROCCC_mux2(tmp0, tmp1, idx0);
                        // output buffer definitions
                    }
```

(a) example of 2D 1-way associative array lookup definition in C and its corresponding ROCCC generated lower-level macro sequence

Fig. 13. Sample 2D table lookup definitions in C and the corresponding ROCCC generated lower-level macros of each.



(a) 1D nonassociative array lookup

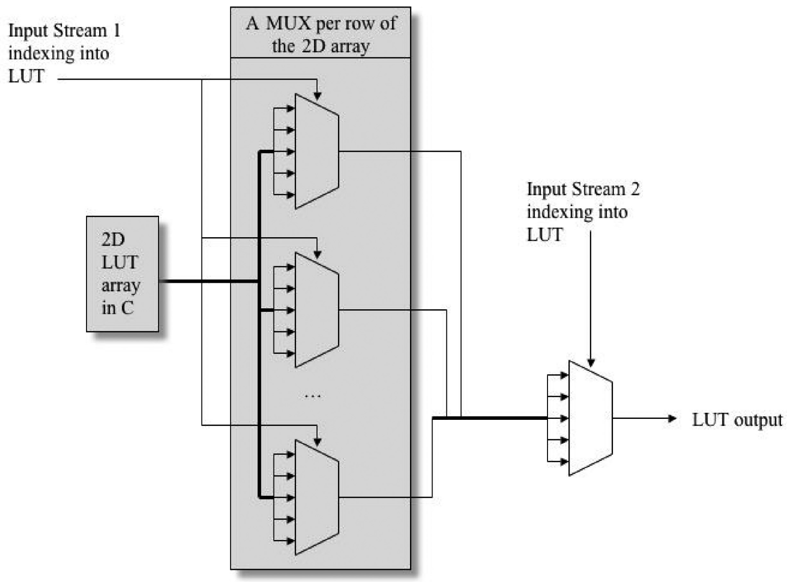


(b) 1D associative array lookup

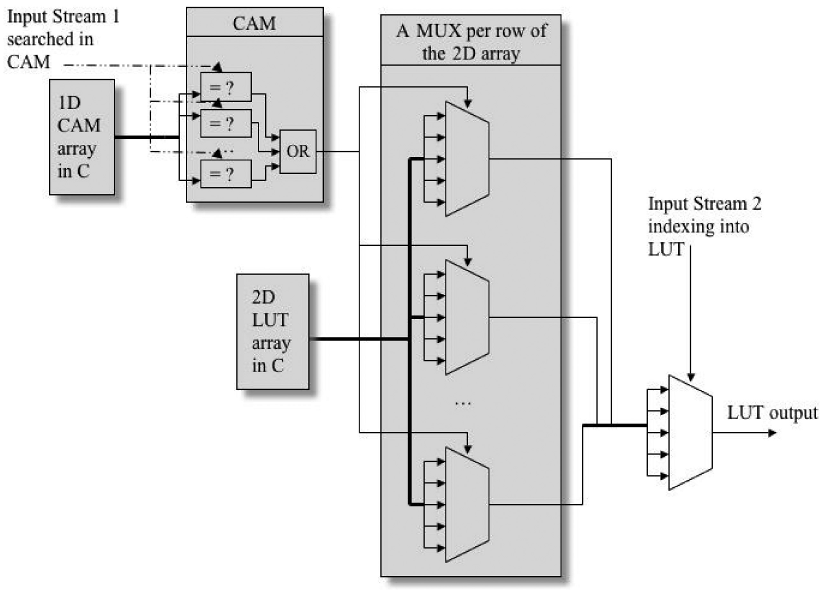
Fig. 14. Schematic view of the ROCCC generated circuits for 1D table lookups.

table lookups. The clock cycle time varied from 5.1ns to 8ns linearly as the circuit area increased. The steady increase in the clock cycle was due to the current version of our compiler generating a single multiplexor taking as many inputs as the number of elements in one row or column of the table description. The clock cycle time could be reduced by generating a collection of multiple small size multiplexors instead of one big one and pipelining through them.

Figure 18 displays the growth in circuit area of circuits with 1D associative table lookups. Examples of this kind of array lookups are sine, cosine, etc., tables in scientific computing. We compiled a loop body that streams in 8-, 16- or 32-bit elements, to be looked up in a CAM composed of the input stream bit-size elements. The index information returned from the CAM is used to index into a 1D array and return a compile-time constant value, every clock cycle. Figure 14(b) illustrates the ROCCC generated circuit design for the 1D associative table lookups. The clock cycle time for this design varied from 5.3ns to 8.3ns growing almost linearly with the circuit area. Again, the increase in



(a) 2D nonassociative array lookup



(b) 2D associative array lookup with CAM alongside one of the dimensions

Fig. 15. Schematic view of the ROCCC generated circuits for 2D table lookups.

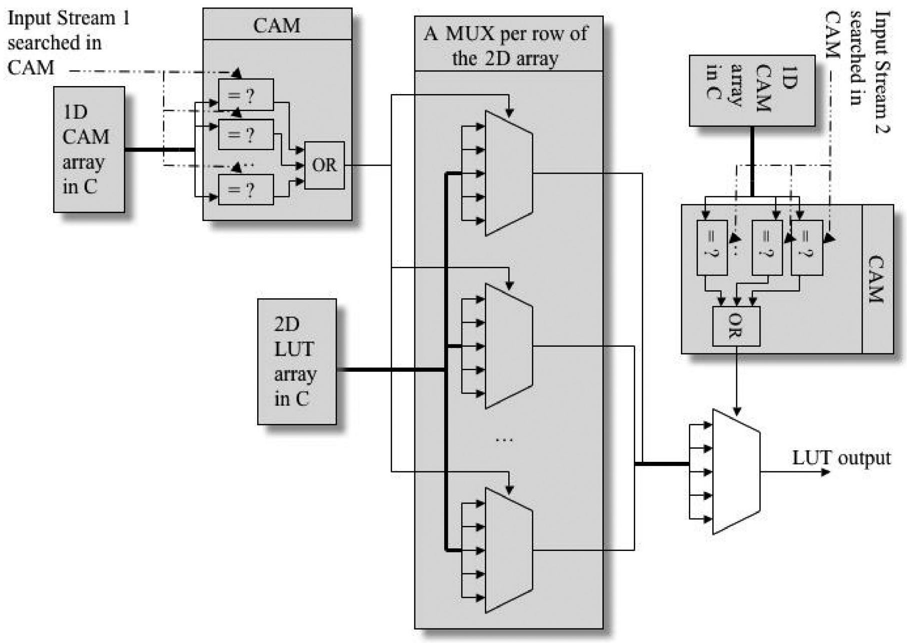

Input Stream 1 searched in CAM
CAM
1D CAM array in C
= ?
OR
A MUX per row of the 2D array
2D LUT array in C
...
1D CAM array in C
Input Stream 2 searched in CAM
CAM
LUT output

Fig. 16. Schematic view of the ROCCC generated circuit for 2D associative table lookup with CAM alongside both dimensions.

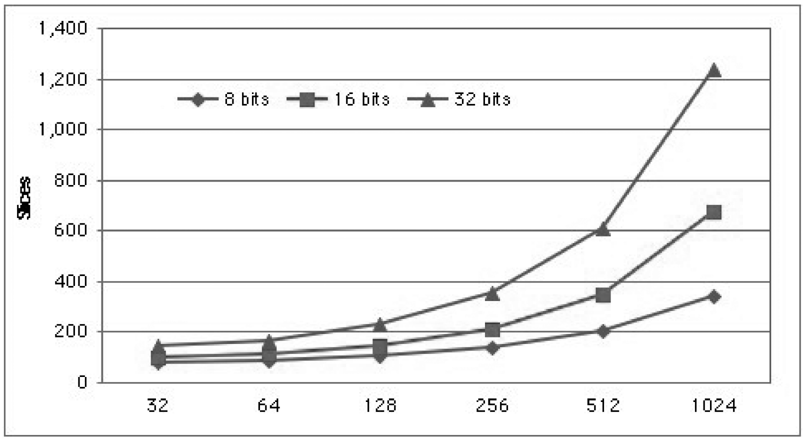
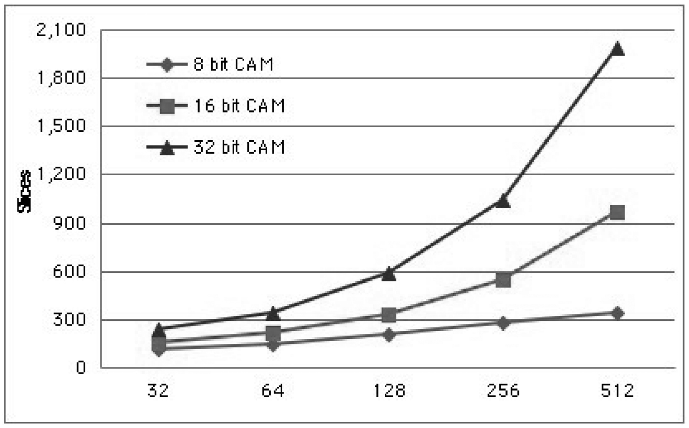
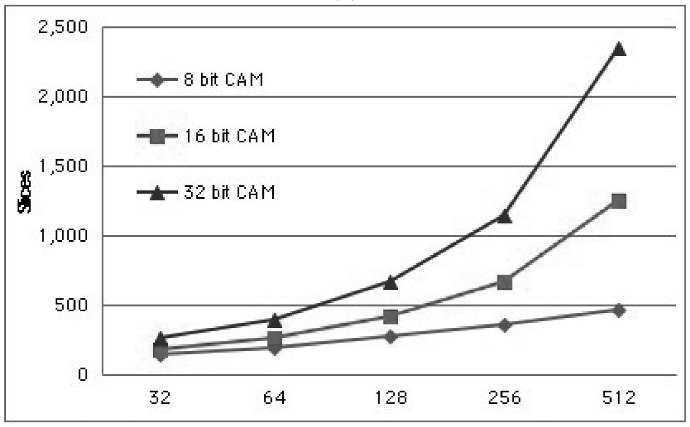


Fig. 17. Area growth of ROCCC generated 1D nonassociative table lookups up to 1024 elements for 8-, 16-, and 32-bit data types.

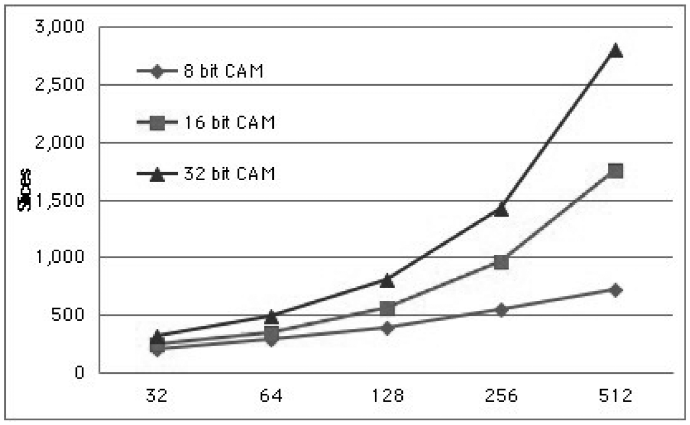
the clock cycle was due to the multiplexors as well as the CAM circuits taking as many inputs as the number of elements in one row or column of the table description. It's possible to reduce the clock cycle time for large table sizes by forming a collection of smaller size CAMs and multiplexors, and pipelining them.



(a)



(b)



(c)

Fig. 18. Area growth of ROCCC generated 1D associative table lookups of up to 512 elements for (a) 8-, (b) 16-, and (c) 32-bit data types.

Figure 19 displays the growth in circuit area of circuits with various size 2D nonassociative table lookups. The two input streams providing the row and column indices are 32-bit input streams. The contents of the 2D table are set as 8, 16, and 32 bits. Only the necessary least significant bits of the 32-bit input stream are used to index into the table. We tried 8, 16, and 32-bit data types for the contents of the 2D tables. Figure 15(a) illustrates the ROCCC generated circuit design for the 2D nonassociative table lookups. The clock cycle time for this design varied from 5.4ns to 11.9ns growing almost linearly with the circuit area, where the clock cycle time can easily be reduced by partitioning the MUXes into collection of smaller ones and pipelining more aggressively.

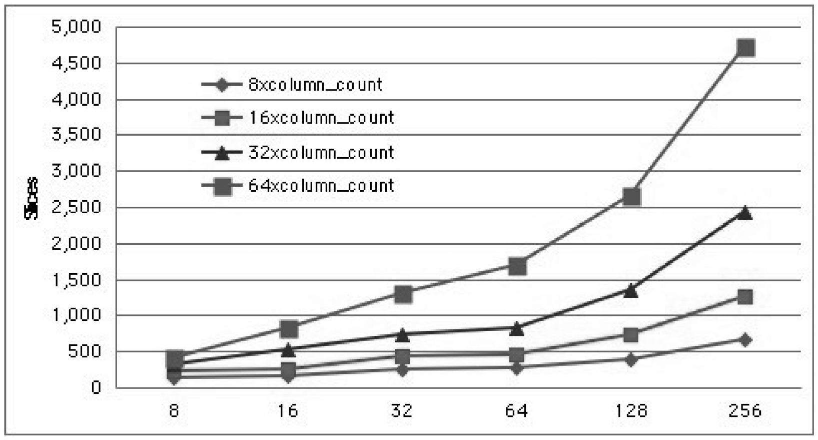
Finally, Figure 20 displays the growth in circuit area of circuits with various size 2D associative table lookups. This type of lookup operations usually exists in bioinformatics string matching applications as score matrix lookups for DNA and protein comparisons. The two input streams providing the row and column indices have the same bit-size as the CAM contents. The contents of the 2D associative table lookup are set as 8 bits. The CAM circuit is aligned alongside the row, hence per graph the size of the CAM does not change but the bitwidth of its contents only changed. Figure 15(b) and Figure 16 illustrate the ROCCC generated circuit designs for the 2D table lookup with a CAM along the row and two CAMs along both the row and the column. The clock cycle time for this design varied from 5.4ns to 8.9ns growing almost linearly with the circuit area.

5. RELATED WORK

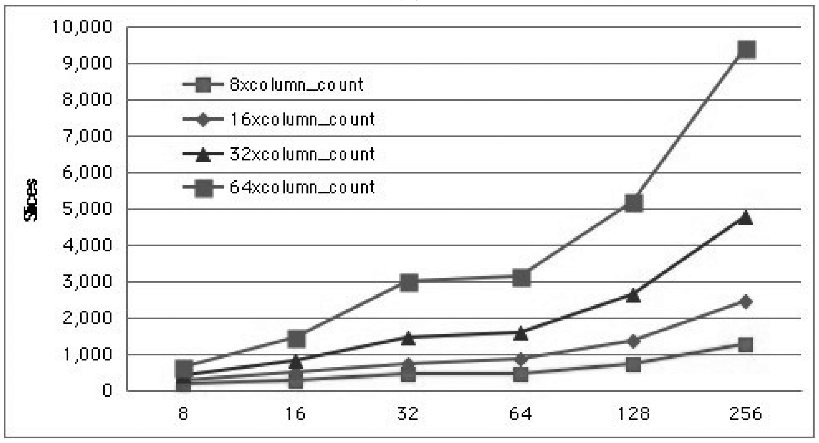
Many projects have worked on translating high-level languages into hardware using various approaches. SystemC [SystemC 2010] is designed to provide roughly the same expressive functionality of VHDL or Verilog and is suitable to designing software hardware synchronized systems. Handel-C [Celoxica Inc. 2010], 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 [Najjar et al. 2003] 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.

There are few commercial tools similar to ROCCC: Mitrion [Dellson et al. 2006], ImpulseC, PICO and Altera's C2H. The Mitrion's approach is to instantiate a Mitrion Virtual Processor (MVP) on the FPGA, a massively parallel core that is programmed using the Mitrion-C language, a single assignment flavor of C. The Mitrion-C language has a special loop statement using the key word foreach. In Mitrion-C, memory interfaces have to be defined by the user through particular key words such as _memread and _memwrite. The keyword, _wait provides timing information.

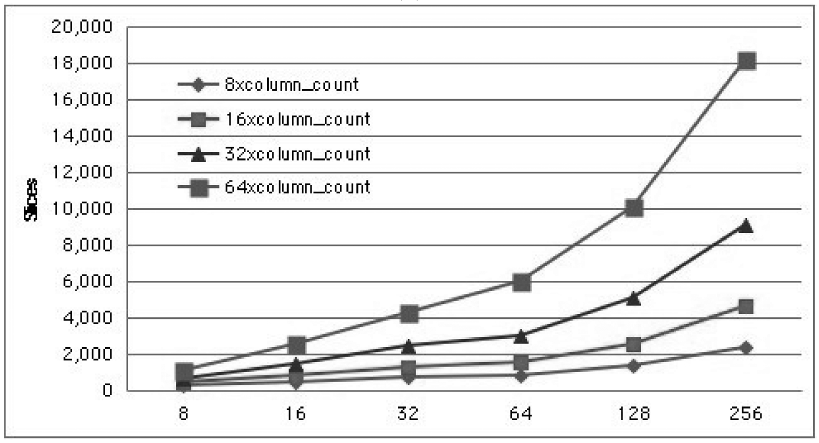
ImpulseC [2010] is the commercialization of Streams-C [Frigo et al. 2001; Gokhale et al. 2000]. Streams-C relies on the user explicitly partitioning the



(a)

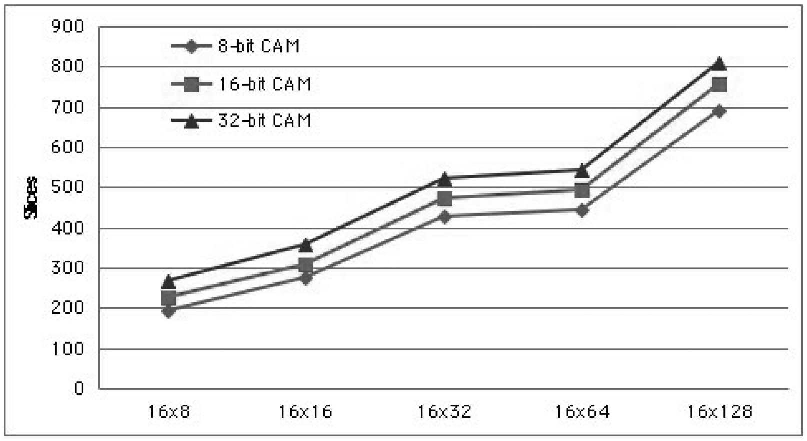


(b)

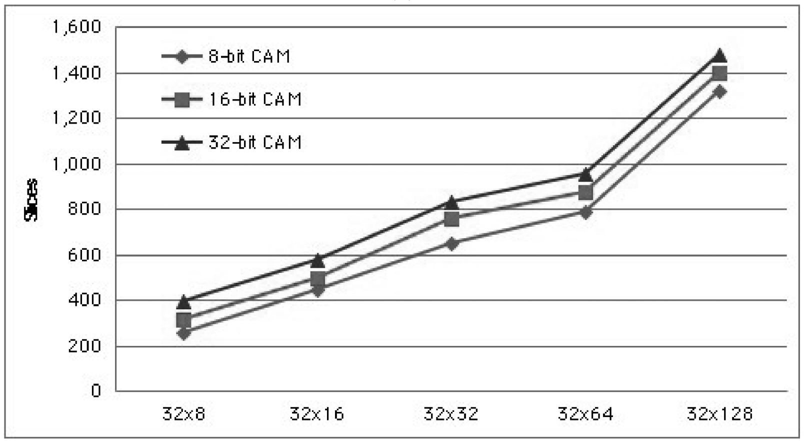


(c)

Fig. 19. Area growth of ROCCC generated nonassociative 2D table lookups of up to 64x256 elements for (a) 8-bit, (b) 16-bit, and (c) 32-bit data types (horizontal axis displays the column_count's).



(a)



(b)

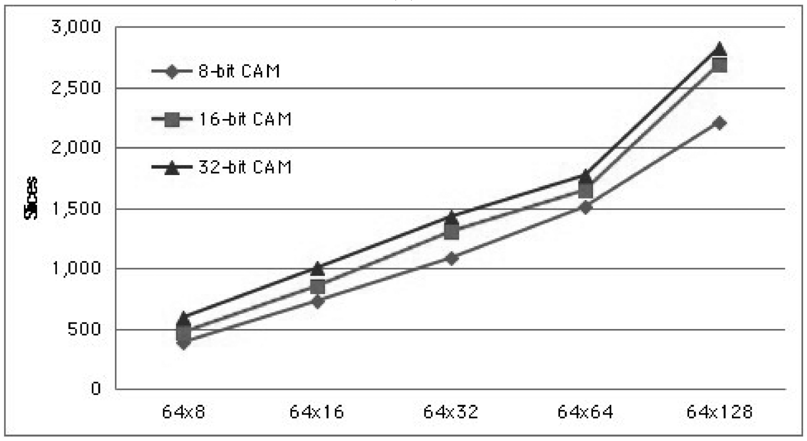


Fig. 20. Area growth of ROCCC generated associative 2D table lookups for the added (a) 16, (b) 32, and (c) 64 element CAMs with 8-, 16-, and 32-bit CAM contents.

code into hardware and software processes and setting up CSP (Communicating Sequential Processes)-based communication channels between them. 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.

Altera's C2H [Altera 2010a, 2010b], compiles ANSI C functions as a component within an existing Nios II system. C2H compiler assumes that the C code already runs successfully on a Nios II system. The compiler gives support to most C constructs including pointers, arrays, and structures, global and local variables. C2H does not do optimization on removing redundant memory accesses. It does not parallelize the code at the loop level, but recognizes and schedules the computation that can occur in parallel simultaneously and pipelines the loop body. ROCCC instead comes with several loop-, procedure-, and application-specific optimizations.

PICO(Program In, Chip Out) [Kathail et al. 2002; Synfora 2010b], being productized by Synfora [Synfora 2010a], started as a research project at HP Labs. PICO takes in C code together with user-specified constraints on area and clock cycle. PICO then maps the compute-intensive sections of the inputed C code onto configurable PPAs (Pipeline of Processor Arrays) and the control-intensive parts of the algorithm onto a configurable VLIW processor architecture. PICO explores the design space for designs that are better than any other in at least one measure of quality. PICO then presents a set of pareto-optimal designs to the user. ROCCC is different than PICO in what it's trying to achieve. PICO's target is to generate fast, complete-solution, custom-made circuits for an entire application in the embedded systems domain, while ROCC aims to accelerate HPC, DSP, and scientific computing applications running on existing servers and/or desktop machines.

An academic work that stemmed from the PICO study is the work Fan et al. [2006]. Their work does not aim to generate complete-solution, custom-made circuits for an entire application, rather it takes an application's long-running kernels and maps them to hardware accelerators. The novelty of their work is in that they generate one unified hardware circuit for multiple kernels and aim at hardware reuse. They map the kernels to a customizable VLIW processor architecture and modulo-schedule the loop bodies so that multiple iterations are executing at the same time at different functional units, which are connected to one another through shift registers. Customized are the widths and/or depths of the functional units and the shift registers connecting the units. ROCCC's approach is different in that it does aggressive compiler transformations to eliminate all the redundancies (memory access and computation) within and across the iterations.

The DEFACTO [Bondalapati et al. 1999; Diniz and Park 2000; Diniz and Park 2002] 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 [Callahan et al. 2000] 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 [Gupta et al. 2003] 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 [Budiu et al. 2004] 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 [Mentor Graphics 2010] 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.

Optimus [Hormati et al. 2008] is a StreamIt [Thies et al. 2002]-to-Verilog compiler that is built on top of the Trimaran [2010] compiler. Optimus first compiles the input language to a canonical IR, where a program is composed of interconnected filters. Next, it applies different sets of optimizations targeting interfilter (macrofunctional) and inner-filter (microfunctional) operations. Optimus then uses a specialized filter template to synthesize the resulting IR's components.

Compared to previous efforts in translating HLLs to HDLs, ROCCC's distinguishing features are its emphasis on maximizing parallelism via extensive set of high-level transformations, maximizing clock speed via pipelining, minimizing area, and minimizing memory accesses using smart buffers, a feature unique to ROCCC. ROCCC handles 2D arrays and can optimize memory accesses for window operations. Moreover, ROCCC replaces the software implementations of the algorithms with hardware-efficient counterparts where selected by the user as a package and the necessary patterns are detected.

6. CONCLUSION

In this article we have presented the high-level transformations of ROCCC (Riverside Optimizing Compiler for Configurable Computing), a C-to-VHDL compiler whose focus is on extensive compile-time transformations and optimizations intended to maximize parallelism and clock rate and minimize memory accesses and circuit area. We used several applications to show the potential of ROCCC's transformations. In this article we showed how ROCCC utilizes: (1) several array access optimizations to eliminate redundant memory accesses, (2) procedure-level optimizations to achieve circuit area reductions of up to 88% compared to circuit areas generated from unoptimized codes, (3) loop-level optimizations in increasing the throughput, and (4) application-specific transformations, such as the hardware lookup table generation, in mapping software implementations of various classes of algorithms most effectively to hardware. The previously listed features help ROCCC generate circuits with very large degrees of parallelism capable of very high computation rates.

REFERENCES

AIGNER, G., DIWAN, A., HEINE, D. L., LAM, M. S., MOORE, D. L., MURPHY, B. R., AND SAPUNTZAKIS, C. 2010. An overview of the SUIF2 compiler infrastructure. Tech. rep., Computer Systems Laboratory, Stanford University.

ALTERA. 2010a. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. http://www.altera.com/literature/wp/wp-aghrdwr.pdf.

ALTERA. 2010b. Nios II C2H user guide. http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf.

BONDALAPATI, K., DINIZ, P., DUNCAN, P., GRANACKI, J., HALL, M., JAIN, R., AND ZIEGLER, H. 1999. Defacto: A design environment for adaptive computing technology. In *Proceedings of the IPPS/SPDP Workshops*. 570–578.

BUDIU, M., VENKATARAMANI, G., CHELCEA, T., AND GOLDSTEIN, S. C. 2004. Spatial computation. *SIGPLAN Not. 39,* 11, 14–26.

BUYUKKURT, B., GUO, Z., AND NAJJAR, W. 2005. Compiler optimization for configurable accelerators. In *Proceedings of the Conference on Optimizations for DSP and Embedded Systems (ODES) In conjunction with International Symposium on Code Generation and Optimization (CGO'05)*.

BUYUKKURT, B., GUO, Z., AND NAJJAR, W. 2006. Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. In *Proceedings of the Workshop On Applied Reconfigurable Computing (ARC'06)*.

CALLAHAN, T. J., HAUSER, J. R., AND WAWRZYNEK, J. 2000. The Garp architecture and C compiler. *IEEE Comput. 33,* 4, 62-69.

CELOXICA INC. 2010. Handel-C language. http://www.celoxica.com.

DELLSON, A., SANDBERG, G., AND MOHL, S. 2006. Turning fpgas into supercomputers—Debunking the myths about fpga-based software acceleration. Tech. rep.

DINIZ, P. AND PARK, J. 2000. Automatic synthesis of data storage and control structures for fpga-based computing engines. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*. 91.

DINIZ, P. AND PARK, J. 2002. Data reorganization engines for the next generation of system-on-a-chip fpgas. In *Proceedings of the ACM/SIGDA 10th International Symposium on Field-Programmable Gate Arrays (FPGA'02)*. 237–244.

DYDEL, S. AND BALA, P. 2004. Large scale protein sequence alignment using fpga reprogrammable logic devices. In *Field Programmable Logic and Application*. Springer, 23–32.

FAN, K., KUDLUR, M., PARK, H., AND MAHLKE, S. 2006. Increasing hardware efficiency with multifunction loop accelerators. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'06)*. ACM, New York, 276–281.

FRIGO, J., GOKHALE, M., AND LAVENIER, D. 2001. Evaluation of the streams-c c-to-FPGA compiler: An applications perspective. In *Proceedings of the ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays(FPGA'01)*. 134–140.

Gokhale, M., Stone, J., Arnold, J., and Kalinowski, M. 2000. Stream-Oriented fpga computing in the streams-c high level language. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*. 49–56.

Guo, Z., Buyukkurt, B., and Najjar, W. 2004. Input data reuse in compiling window operations onto reconfigurable hardware. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*. 249–256.

Guo, Z., Buyukkurt, B., Najjar, W., and Vissers, K. 2005. Optimized generation of data-path from C codes for FPGAs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*. 112–117.

Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. 2003. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceeding of the International Conference on VLSI Design (VLSI'03)*.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., and Brown, R. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC'01)*. 3–14.

Hormati, A., Kudlur, M., Mahlke, S., Bacon, D., and Rabbah, R. 2008. Optimus: Efficient realization of streaming applications on fpgas. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'08)*. ACM, New York, 41–50.

ImpulseC. 2010. ImpulseC homepage. http://www.impulsec.com.

Jacobi, R., Ayala-Rincon, M., Carvalho, L., Llanos, C., and Hartenstein, R. 2005. Reconfigurable systems for sequence alignment and for general dynamic programming. *Genet Mol. Res. 4,* 3, 543–552.

Kathail, V., Ahitya, S., Shreiber, R., Ramakrisha, B., Cronquist, D., and Sivaraman, M. 2002. PICO (program in, chip out): Automatically designing custom computers. *IEEE Comput. 35,* 9, 39–47.

Mentor Graphics. 2010. Catapult C synthesis. http://www.mentor.com/products/c-based.design/catapult.c.synthesis/index.cfm.

Najjar, W. A., Böhm, A. P. W., Draper, B. A., Hammes, J., Rinker, R., Beveridge, R., Chawathe, M., and Ross, C. 2003. From algorithms to hardware—A high-level language abstraction for reconfigurable computing. *IEEE Comput. 36,* 8, 63–69.

Puttegowda, K., Worek, W., Pappas, N., Dandapani, A., Athanas, P., and Dickerman, A. 2003. A run-time reconfigurable system for gene-sequence searching. In *Proceedings of the 16th International Conference on VLSI Design (VLSID'03)*. 561–566.

SGI Inc. *SGI RASC RC100 Blade*. http://www.sgi.com/products/rasc/datasheets.html.

Singpiel, H., Simmler, H., Kugel, A., Manner, R., Vieira, A. C. C., Galvez-Durand, F., de Alcantara, J. M. S., and Alves, V. C. 2000. Implementation of cryptographic applications on the reconfigurable fpga coprocessor microenable. In *Proceedings of the 13th Symposium on Integrated Circuits and Systems Design (SBCCI'00)*. IEEE Computer Society, 359.

Smith, M. D. and Holloway, G. 2010a. An introduction to Machine SUIF and its portable libraries for analysis and optimization. Tech. rep., Division of Engineering and Applied Sciences, Harvard University.

Smith, M. D. and Holloway, G. 2010b. Machine-SUIF SUIFvm Library. Tech. rep., Division of Engineering and Applied Sciences, Harvard University.

Synfora, I. 2010a. Synfora homepage. http://www.synfora.com.

Synfora, I. 2010b. PICO technology white paper (v.10). http://www.synfora.com/about/files/PICO_technology_whitepaper_v1.0.pdf.

SystemC. 2010. SystemC consortium. http://www.systemc.org.

Thies, B., Karczmarek, M., and Amarasinghe, S. 2002. Streamit: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction*. 179–196.

Trimaran. 2010. An infrastructure for research in ILP. http://www.trimaran.org.

Received March 2007; revised April 2009; accepted August 2010