ASC: A Stream Compiler for Computing With FPGAs

Oskar Mencer

***Abstract*—A stream compiler (ASC) for computing with field programmable gate arrays (FPGAs) emerges from the ambition to bridge the hardware-design productivity gap where the number of available transistors grows more rapidly than the productivity of very large scale integration (VLSI) and FPGA computer-aided-design (CAD) tools. ASC addresses this problem with a software-like programming interface to hardware design (FPGAs) while keeping the performance of hand-designed circuits at the same time. ASC improves productivity by letting the programmer optimize the implementation on the algorithm level, the architecture level, the arithmetic level, and the gate level, all within the same C++ program.**

The increased productivity of ASC is applied to the hardware acceleration of a wide range of applications. Traditionally, hardware accelerators are tediously handcrafted to achieve top performance. ASC simplifies design-space exploration of hardware accelerators by transforming the hardware-design task into a software-design process, using only “GNU compiler collection (GCC)” and “make” to obtain a hardware netlist. From experience, the hardware-design productivity and ease of use are close to pure software development.

This paper presents results and case studies with optimizations that are: 1) on the gate level—Kasumi and International Data Encryption Algorithm (IDEA) encryptions; 2) on the arithmetic level—redundant addition and multiplication function evaluation for two-dimensional (2-D) rotation; and 3) on the architecture level—Wavelet and Lempel–Ziv (LZ)-like compression.

***Index Terms*—Design space exploration, FPGAs, hardware design.**

I. Introduction

TRADITIONALLY, computer systems consist of a microprocessor and an additional set of application- or domain-specific devices, or hardware accelerators, which accelerate a certain functionality. Some examples are floating-point coprocessors in early microprocessor systems, two-dimensional (2-D) and three-dimensional (3-D) graphics accelerator cards, and combinations of software and hardware accelerators in embedded systems. However, all these hardware accelerators are tediously handcrafted to achieve top performance. If we consider a field programmable gate array (FPGA) with 10M customizable gates, which could be reconfigured every 100 ms; we could generate circuits of up to 100M gates/s to keep the chip busy. Therefore, the more we can increase the productivity of our hardware-design system, the better use we can make of reconfigurable technology.

The ideal programming solution needs to automate the generation of hardware and, at the same time, achieve top performance of hand-designed circuits. A stream compiler (ASC) is a general-purpose hardware-generation system with a special support for generating stream architectures. ASC achieves top performance with low programming effort by providing access on all three levels of abstraction. Thus, ASC bridges the hardware-design gap between the ever-increasing number of transistors on a chip and the much slower increase of productivity delivered by hardware-design tools and methodologies. Previous publications have covered the principle of our approach [20]. The key points about ASC that are discussed in this paper are as follows:

1) programming interface, hardware-variable types and attributes on various levels of abstraction (Section III);
2) details of custom stream architecture generation, in particular, the datapath part of the design (Section IV);
3) details of module generation/“instruction set’” for hardware accelerators on FPGAs (Section V);
4) evaluation and test using a number of benchmarks from small to large sizes such as encryption, compression, and elementary arithmetic (Section IX).

Various aspects of ASC are published in conference papers [16], [18]. This paper selectively combines and extends previous publications, adding the test methodology employed to ASC and ASC user programs and an extended comparison to related work.

Fig. 1 shows the general structure of a computer system with multiple application-specific accelerators. The accelerator can be located on-chip with a processor such as today’s floating-point units, the Berkeley Garp Processor [7], or the Xilinx Virtex Pro FPGAs with on-chip PowerPC processors [25]. Also, such accelerators can be combined with the main

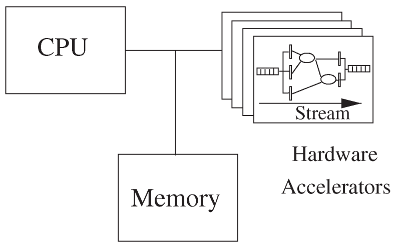

Fig. 1. Computer system with hardware accelerators such as stream architectures.

Manuscript received October 15, 2004; revised January 21, 2005 and May 14, 2005. This work was supported in part by the Computing Sciences Center at Bell Labs and Imperial College and in part by the U.K. Engineering and Physical Sciences Research Council under Contract GR/R 55931. This paper was recommended by Associate Editor A. Raghunathan.

The author is with the Department of Computing, Imperial College, London SW7 2BZ, U.K., and Maxeler Technologies, Inc. (e-mail: o.mencer@imperial.ac.uk).

Digital Object Identifier 10.1109/TCAD.2005.857377

memory [14] or on the peripheral bus [17], [34]. Furthermore, accelerators can be implemented in custom very-large-scale-integration (VLSI) devices or as FPGA configurations. In either case, there are two memory systems: 1) a compile-time memory system on or around the accelerator, i.e., memories inside the FPGA or directly attached to it and 2) the processors memory system, which is managed at run time.

On the FPGA side, recent advances in FPGA technology enable the development of many hardware accelerators customized for specific applications and for particular input-data sets [19]. These accelerators can be generated and managed at compile time and at run time.

However, building efficient hardware accelerators for a particular application consists of many challenging tasks. First, the programmer can explore four degrees of freedom: the system architecture, the micro architecture, the functional units, and the level of programmability or granularity of configuration. This exploration of the structure of computation results in the datapath part of the design. Second, a custom accelerator requires a custom memory system consisting of on-chip registers, an on-chip and off-chip static random access memory (SRAM), and possibly a dynamic random access memory (DRAM). Third, run-time software routines take care of sending the appropriate data back and forth between the processor and the hardware accelerator. Fourth, a control block for this datapath makes sure that the timing of operations is correct. Fifth, an interface between the accelerator and the processor maximizes the data transfer rate.

With ASC, the programmer can focus on the first three items while ASC provides facilities to save the programmers time and automates the fourth and fifth tasks.

Traditionally, low-level hardware-design tools focus on creating one hardware design, while high-level design tools focus on design-space exploration. By combining these activities, ASC simultaneously provides both top performance and easy design-space exploration.

ASC facilitates design-space exploration in two ways. First, for the datapath, a single ASC description produces multiple datapath implementations at the micro architecture level with user-specified tradeoffs. ASC also simplifies the process of selecting and possibly custom designing of the functional units by having the descriptions on various levels of abstractions captured in a uniform object-oriented style. The object-oriented implementation of ASC also enables us to easily support several families of Xilinx FPGA devices such as Xilinx 4000, Xilinx Virtex, Virtex 2, Virtex 4, Spartan 2, and Spartan 3.

Second, ASC automates the generation of the control block, the run-time routines, and the CPU–FPGA interface based on user specifications. Our purpose is to put the design-space exploration under user control. For example, by specifying the algorithm in C++ syntax and ASC semantics, the user also controls the memory system that ASC generates for the application at hand.

II. A Stream Compiler (ASC)

On the top level, the user writes an ASC code that closely resembles C code. As a consequence, existing C/C++ software can be seamlessly transformed to ASC. In order to express and explore the design space of a hardware accelerator, ASC code is parameterized to generate a large selection of implementations. With these parameterizations, the user trades off, for example, silicon area for latency, throughput, and/or precision.

In essence, ASC is a C++ library and, as such, can be compiled by a standard C++ compiler. Thus, ASC code is simply C++ that makes use of the ASC library in a compliant manner. When compiled, the ASC code becomes an executable that either acts as a word-level simulation or a bit-level (RT-level) simulation, or produces a circuit in the form of a hardware netlist.

The concepts of timing and architecture of the circuit map to user-defined types or ASC "hardware-types," implemented as C++ classes and operators. These hardware operators map to the module-generation layer, PAM-Blox II [18]. PAM-Blox II is also implemented as a C++ class library built on top of PamDC [26], which provides the engine for gate-level simulation and supports output in EDIF netlist format.

For design-space exploration, ASC provides three intermediate representations, all in C++ syntax, to transform a software implementation all the way down to the gate level without the use of a single line of very-high-speed-IC hardware description language (VHDL), Verilog, or IP libraries. Since each intermediate representation is a human readable language, it is possible to reason about optimizations at each of these levels and explore such optimizations within the ASC framework.

Conceptually, ASC follows the underlying methodology of the C programming language. The objective is to offer the potential for maximal performance and, at the same time, to provide a convenient language interface. On the hardware side, implementations are not limited to any particular number representation or to any particular bitwidth. Custom hardware provides a substrate for the programmer to tailor the number representation to the specific application. In order to simplify this process, the ASC description provides hardware types and attributes that select specific number representations. Types and attributes provide either a connection or hooks between the C++ description and the architecture generation layer. Fig. 2 shows the levels of abstraction in ASC, which are described in detail below.

1) Algorithm analysis layer. The common tasks associated with this layer include: extracting compiler-controlled memory management [38], [39], pointer analysis for hardware synthesis [37], loop transformations for hardware generation [7], [11], [24], precision analysis [4], [6], [35], data-structure transformations, and architecture selection. This layer is currently handled manually, i.e., all algorithmic transformations are done by the programmer. The task of ASC is to make this activity as easy as possible and to support research on hardware algorithm analysis and transformations.
2) Architecture generation layer. ASC code serves as the input to generate the hardware architecture. The ASC-type system provides the mapping of sequential code to a custom-hardware architecture.

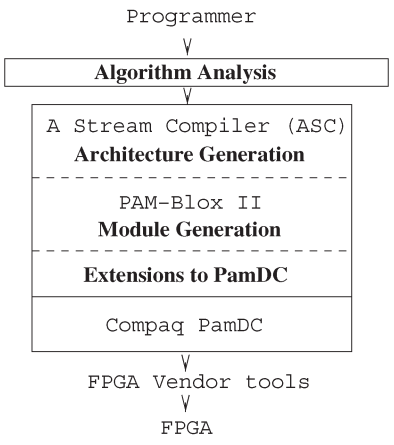

Fig. 2. Levels of abstraction and the structure of ASC. The largest box represents a single C++ program.

3) Module-generation layer. In contrast to most other hardware compiler efforts, ASC contains its own integrated module-generator libraries, PAM-Blox II. PAM-Blox II offers the ASC user easy exploration of bit-level parallelism (BLP) in conjunction with optimizations on various other levels of abstraction.
4) Gate Level to netlist layer. ASC does not utilize any VHDL or Verilog. Instead, it uses PamDC [26], a C++ library for gate-level FPGA design, simulation, and EDIF-netlist generation.

In order to meet the above requirements for module generation, we apply an object-oriented-design methodology. Object-oriented software design is a well established technology in the software world. The hardware world is slowly adopting the advances made by object-oriented languages such as C/C++ [45], [46] and Java [47]–[49]. Object-oriented design leads to an efficient solution of the module-generation problem by focusing on the requirements for module generation mentioned above, such as scalability and code sharing. Inheritance and hierarchical class structures match the requirements of creating a large library of module generators with the logic expressed as computation (methods) and the module abstraction parameters described as internal state (local variables) of the generated object.

III. Comparison With Other Approaches

As for related tools and approaches, the commercial module-generator library available from Xilinx (CoreGen) contains module generators that can be instantiated through a standalone GUI. This approach is very well suited for the computer-aided-design (CAD) tool flow but less ideal for a programming environment. A direct comparison of the performance values from the Xilinx CoreGen data sheets is complicated since the numbers in this paper are real design results, while Xilinx values are maximal (best case) values.

Pebble [58], a language designed at Imperial College, generates VHDL modules for a conventional CAD flow but requires the user to learn a new language syntax and to use the CAD design methodology. The Java hardware description language (JHDL) [49] is a similar effort to PAM-Blox/ASC. Besides thearguments for and against Java, JHDL also integrates module generation with the higher compilation layers. Additionally, JHDL contains a run-time system, a port to Virtex II, and a large set of modules. Similarly, a commercial effort by Celoxica [46] provides the "programming feel" to FPGA design, mostly targeting embedded systems.

One difference of the approach proposed in this paper to these related projects is the emphasis on handling different number representations to tap into the full potential of the FPGAs flexibility on the bit level.

The key benefit of architecture-level ASC as compared to the C-to-FPGA approaches below is that ASC enables the programmer to generate optimal circuits by programming on the bit level, while at the same time making it easy to explore a large design space and program noncritical parts of the applications on a very high level.

The design environment for adaptive computing technology (DEFACTO) system [23] supports the hardware-design-space exploration based on parallelizing compiler technology and high-level synthesis tools. A key element in DEFACTO is the use of synthesis estimation techniques, possibly from behavioral synthesis tools [22], to quantitatively evaluate alternative designs for a loop nest computation. Other researchers have also proposed estimation-based exploration methods, such as the heuristics-based allocation based on communication cost reduction [5]. In contrast, ASC operates on a lower level and could be targeted by a DEFACTO-style layer.

The Nimble framework [15] extracts loops from applications and generates a hardware accelerator for an FPGA. Similar to DEFACTO, much of Nimble is actually above the ASC level, as its main focus is on hardware/software partitioning. As a consequence, Nimble is limited to high-level transformations, particularly those exploring architectural and instruction-level parallelism. The focus with ASC is to bring all relevant levels of abstraction together in a coherent framework, from bit level to algorithm level.

The Stream-C [11] and malleable-architecture-generator (MARGE) [12] systems compile C code to multi-FPGA hardware accelerators. Similar to Nimble above, Stream-C operates mostly at a higher level than ASC. However, Stream-C is more hands-on than Nimble, requiring user programming to explore the design space. Stream-C follows the traditional behavioral synthesis approach of adding annotations with compiler directives to the code in the form of comments. Instead, ASC includes compiler "directives" into the structure of the description within the type system, object classes, function calls, and macros, offering a richer scope of expression to the programmer.

Celoxica [8] provides Handel-C, a C derivative language for high-level hardware design. Handel-C can be used to design hardware accelerators for FPGAs at a similar level as ASC. Like ASC, Handel-C provides the hardware designer with control and opportunities for optimization. The main difference from ASC is that the entire compiler code and most module libraries are proprietary and thus off limits to the user. In comparison: 1) ASC is truly general purpose, while Handel-C does not support the generation of arbitrary circuits; 2) ASC uses a conventional GNU-compiler-collection (GCC) compiler, which makes ASC more compatible with standard C++ software practices; 3) Handel-C restricts the user by declaring a clock cycle as one expression, i.e., the assignment operator "=" specifies a clock cycle—clearly making it difficult to create large combinational circuits—while ASC supports the generation of arbitrary circuits; and 4) ASC enables the user to generate many designs with a single source file and an experimental setup in makefiles. To our knowledge, neither Handel-C nor any other high-performance hardware-design environment support similar productivity in exploring the design space.

Similar efforts also exist in the custom VLSI world. For example, ShiftQ [2], the nonprogrammable accelerator (NPA) for program-in-computer-out [1] (PICO) systems enables the user to quickly find an optimum hardware solution.

Tensilica [41] provides a similar processor generation system. Sherwood and Calder [21] provide higher level algorithms to search through the processor design space for a PICO or a Tensilica-like system. Also, Dhodapkar and Smith [10] show a dynamic method to manage different configurations of a computer system. Even though their method is targeted at configurable resources in a conventional processor, their method could be applied to dynamically control configuration options of domain- and application-specific compilers.

More generally, the idea of data-centric computation is the key component in dataflow [3] systems. ASC uses similar principles like static dataflow but customizes the architecture to a particular application or application domain.

Why did we choose object-oriented C++? C++ is one of the richest object-oriented languages sometimes criticized for the complexity arising from this richness of features. On the other hand, once an optimal mapping of the problem space to C++ features is established, the software design and maintenance task is greatly simplified.

Why did we choose C++ as opposed to Java [47]–[49]? C++ offers operator overloading (not available in Java), which is one of the most convenient features for adding application-specific semantics to a programming language. In our case, these semantics include Boolean logic equations and in fact any expressions/operations on user-defined classes that specify hardware-variable types. The second reason for using C++ is the standard template library (STL) [50].

Why did we not choose SystemC [45]? SystemC is optimized for the hardware-design process by mirroring the philosophy of simulation languages such as VHDL or Verilog, i.e., providing an additional layer on top of very large legacy code. ASC provides the user with simultaneous access to all levels of abstraction.

The major general advantage of ASC is the combination of general-purpose low-level optimization with high-level design-space exploration; which, as far as we know, is not supported by other currently available tools.

IV. ASC Architecture Generation

ASC architecture generation deals with the mapping of an architectural description, in our case ASC code, to a structure consisting of a custom datapath, a control, and various interface blocks. How does an ASC description deal with timing, parallelization, and pipelining of an algorithm? The big picture is that ASC contains an underlying parametrizable and moldable architecture—the stream architecture. ASC extends the C++-type system using user-defined classes as hooks to map the algorithm to a particular instance of a stream architecture. In addition, for each piece of code, ASC can be directed by the user to optimize throughput, latency, or area. Since each of these three optimization modes can be selected separately for each expression in the ASC code, the user can optimize towards any objective such as area, latency, or throughput. This means that ASC allows for optimization towards a combination of all three optimizations.

A constructive way to visualize stream architectures, assuming a simple feed-forward dataflow graph of a loop body, is to imagine taking the dataflow graph, inserting flip-flops to generate a pipeline, and streaming data in at one end while letting the data flow out on the other end of the pipeline.

The following example shows a C code for vector increment, an ASC code, and the resulting stream architecture.

in C (software):
```
int i, a[SIZE], b[SIZE];
for (i = 0; i < SIZE; i++) {
  b[i] = a[i] + 1;
}
```

The C loop above is expressed in ASC by declaring an input stream `a` and an output stream `b` and by specifying the expression whose operator defines the function (add) to compute the elements of the output stream when given the input stream.

ASC code:
```
STREAM_START;
  // variables and bitwidths;
  HWint  a (IN, 32), b (OUT, 32);
STREAM_LOOP (SIZE);
  STREAM_OPTIMIZE = THROUGHPUT;
  b = a + 1;
STREAM_END;
```

The ASC code is a correct C++ with user-defined types, operators, and a library of macros and function calls. Therefore, the ASC code is compiled with GCC like any other C++ program, libASC is linked, and running the executable produces an EDIF netlist or a gate-level simulation of the circuit. Consequently, programming with ASC is similar to programming in C++; and thus, we have a true softwarelike (suffix, close up) hardware development process.

For the simple example above, the seven lines of ASC code run on an FPGA by typing "make run" in the command line. The ASC makefile system automatically compiles the code, generates the EDIF netlist, runs Xilinx place and route tools, gets the timing information, sets the clock on the FPGA card, downloads the bit stream, and runs the program on the FPGA, displaying the result. By typing "make sim," the program above is executed in a gate-level simulation in C++.

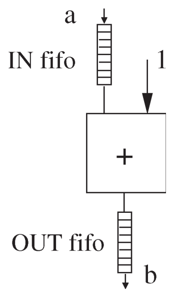

Fig. 3. Simple ASC stream architecture (circuit on the hardware accelerator/FPGA) with one input FIFO, one output FIFO, and a possibly pipelined datapath (in this case just one adder) absorbing inputs from the input FIFO and producing outputs for the output FIFO.

Note that the "for" loop in C code translates to a declaration of `STREAM_LOOP` in ASC code, the variable-type changes to `HWint`, and the variables get "architectural attributes" `IN` and `OUT`. From a vector processor perspective, streams are a generalization of vectors. We express algorithms in terms of streams (or arrays in C). ASC then generates a stream architecture based on the `STREAM_OPTIMIZE` for each expression. Currently supported optimization values inclue `THROUGHPUT`, `LATENCY`, and `AREA`.

1) `THROUGHPUT`: (default) In a throughput mode, all flip-flops are being used, and the resulting circuit is balanced (scheduled) by using first-in-first-out (FIFO) buffers in-between the arithmetic units.
2) `LATENCY`: In a latency mode, no flip-flops are being inserted; and as a consequence, the resulting circuit is purely combinational.
3) `AREA`: In an area mode, ASC uses sequential arithmetic units, e.g., ASC selects an add-accumulate unit for multiplication.

At run time, a C program with modified ASC run time calls streams data through the hardware to compute the results. An example for an ASC run time call could replace the "for" loop (`STREAM_LOOP`) above by a call to the ASC run-time library.

```
ascrt_stream_int (a, b, SIZE, SIZE);
```

This call sends `SIZE` data items from buffer `a` in main memory to the generated circuits, either in a gate-level simulation mode or real hardware, and receives `SIZE` result items into buffer `b`. At the hardware accelerator, the input data enters an FIFO buffer and flows through the stream architecture until it arrives at the output FIFO buffer. The above ASC code results in the implementation shown in Fig. 3.

In general, an ASC architecture consists of a multiinput multioutput data flow graph. Each "wave" of input values flows through this implementation of the data flow graph. An implementation of a data flow graph involves delay FIFO buffers, which balance the movement of the various operands through the compute engine. The delay inserted by each buffer is set by the scheduling phase of ASC.

A. ASC Scheduling And Control-Block Generation

ASC generates statically scheduled architectures. While the ASC user focuses attention on generating the datapath, ASC automatically schedules all computations and generates a custom control block for the particular pipeline. This control logic sets the enable signals for all flip-flops and controls all FIFO buffers and memories to latch the correct values at the right time.

From the scheduler's perspective, a stream architecture is a graph where the nodes have a particular latency (pipeline depth) and a minimal number of clock cycles between successive inputs. This minimal number of clock cycles between successive inputs is the latency of sequential units. ASC schedules the operations of the data flow graph resulting from the C++ code by inserting delay FIFO buffers between producers and consumers of data values, to ensure that the operands of each operation arrive together. As long as there are no cycles in the data flow graph, the resulting implementation can be fully pipelined regardless of local data dependencies and runs at a throughput equal to the data rate, since it can absorb one set of input values at each clock cycle. This pipelined mode of operation represents the computation that is parallelized in time (as opposed to parallelization in space, which would mean replicating stream architectures).

V. ASC Datapath

This section describes the facilities that ASC provides for design-space exploration of the datapath and the custom-memory-system blocks of the stream architecture.

A. Hardware Types and Attributes

ASC uses custom types and attributes as the means of conveying timing and structure to the compiler. Each hardware type denotes a family of related representations. For example, `HWint` denotes the integer family of representations. In addition, the user specifies attributes to select more specific details, such as sign representation (e.g., two's complement or sign magnitude), bit width, or memory-type (e.g., register, temporary, stream input, or FPGA internal memory block). These attributes are parameters stored within the state of the hardware-variable class. Available data types include `HWint`, `HWfix`, and `HWfloat`. For example, the following code uses fixed-point variables to compute a running average.

```
STREAM_START;
// var x - iiiiiiii.ffffffff
HWfix x (IN, 16, 8, UNSIGNED);
HWfix y (TMP, 64, 8, UNSIGNED);
HWfix sum (MAPPED_REGISTER, 64, 8,
   UNSIGNED);
HWfix av (MAPPED_REGISTER, 24, 12,
   UNSIGNED);
HWint l (TMP);
STREAM_LOOP (1000000)
STREAM_OPTIMIZE = LATENCY;
LoopIndex (l);
y = x + sum;
sum + = y;
av = sum/l;
STREAM_END;
```

Notice that `HWint`/`HWfix` are streams of numbers rather than single data items. Streams are like vectors with flexible length. The length of a stream can be varied at run time. ASC stream variables reflect the data streams that come through a port/bus on a chip.

The attribute `MAPPED_REGISTER` maps the ASC variable into the host processors memory space making it read/writable at run time.

B. ASC "Instructions": Module-Generator Libraries

The ASC module-generation layer, PAM-Blox II [18], consists of more than 170 integer arithmetic module generators for elementary operations in about 10 000 lines of C++ code, resulting in an average of fewer than 60 lines of code per module generator.

ASC arithmetic-unit generators include flip-flops and thus timing in the generated unit. For all operations, the user chooses an appropriate implementation by selecting one of three optimization modes: latency, area, or throughput. As a consequence, ASC chooses the appropriate module for the particular optimization: a plain combinational arithmetic unit for latency minimization, a sequential arithmetic unit for area minimization, and a fully pipelined arithmetic unit for throughput maximization.

ASC also contains floating-point module generators [16] capable of generating over 200 distinct floating-point units. The generated floating-point units differ in their algorithm, architecture, and timing (pipelining) and thus represent over 200 design points in the area, latency, and throughput design space. In addition, each of these floating-point units can be generated with a variable number of bits for the mantissa and the exponent. Furthermore, our arithmetic-unit generators enable a tradeoff of precision versus area by enabling the user to choose custom rounding and normalizing schemes.

C. ASC Memory Systems

The compile-time memory system in ASC supports flip-flops and registers FIFO buffers, small multiported on-chip SRAM blocks, large on-chip SRAM blocks, off-chip SRAM memory, and off-chip DRAM memory. At run time, there is also the processor's memory system, which is managed by the ASC run-time system. In this section, we will focus on the compile-time part of the memory hierarchy.

One key advantage of having flexibility at the bit level is that we can generate an application-specific memory system all the way down to the bit level. ASC does not automatically generate the optimal memory system. Instead ASC provides a notation to express application-specific memory systems, in order to enable the exploration of and reasoning about memory-system optimizations. As before, we utilize types and especially "architectural attributes" to assign algorithmic variables to the various physical components of the generated memory system. Thus, ASC variables can be TMP variables as described before, and INTMEM or EXTMEM for FPGA internal block RAM memories or FPGA external memories, respectively. For multiple external memories, ASC provides attributes EXTMEM0, EXTMEM1, etc.

VI. Implementation of ASC Module Generation

A conventional hardware-module library stores the implementations of a large set of hardware modules. A module-generation library distinguishes itself from a conventional library of hardware modules by storing the algorithm that generates a set of hardware modules based on input parameters such as bitwidth of inputs and outputs and sign representation of inputs and outputs. For example, the parameterized array multiplier occupies an area of $m \times n$ cells, where m and n are the bitwidths of the multiplicand and multiplier, respectively. In this sense, module generation is really a software system that designs hardware, rather than an extension of a hardware description system.

The module-generation framework of ASC, PAM-Blox II, contains: 1) extensions to the underlying gate-level layer, PamDC; and 2) an updated methodology for utilizing object-oriented features of C++ to module generation for the purpose sof computing with FPGAs.

Bit-level features are as follows.

1) `class Net`: The generic `class Net` encapsulates a set of wires of variable size and thus enables width inference at the module-generator level. This class simplifies the C++ code required to describe the generators. In addition, `class Net` contains a set of user defined operators that further simplify the description of operations on entire sets of wires, such as assignment, indexing, and concatenation. A key feature of `class Net` is compatibility with the STL of C++, which is not compatible with PamDC objects such as `Bool`, `Wire`, or `WireVector`.
2) Support for various sign-representation modes on the bit level: In order to support multiple sign representations such as twos-complement, sign magnitude, and unsigned numbers.
3) Xilinx Virtex support includes wrappers for generating large block RAMs available in the Xilinx Virtex FPGA family as dedicated parametrizable blocks of memory. The gate-level designer has the option to select the width of the constant-size block RAM within the limits of the particular underlying FPGA technology.
4) In order to make the ASC project and, in particular, PAM-Blox II more accessible, PamDC is ported from Compaq ALPHA cxx to GNU GCC version 2.95.2 or higher. Even though C++ is standardized, porting software between platforms is still a major challenge because most of the C++ compilers do not implement a stable set of the C++ standard.

PAM-Blox II is implemented on top of these bit-level features. Object-oriented features of C++ correspond to the tasks involved in describing hardware-module generators as follows.

1) Encapsulation of a module generator in a C++ class. An object state represents the internal wires and parameters of the module. These parameters can be accessed by various other components of the architecture generation environment such as the scheduler or, possibly, a high-level area and timing estimator. The object

functions or methods describe the logic parametrically, generating the hardware module based on the input parameters.

2) The code reuse is supported by a C++ class hierarchy with explicit inheritance controlled by defining virtual functions and function overloading. Child objects inherit all public methods (functions) and variables (state). For example, all objects with a carry chain (such as adders, counters, and shifters) inherit the carry-chain definition functions from their common parent. This particular example of code reuse is paramount to porting the module generators from one FPGA family to another. Details on porting Xilinx XC4000 carry-chain generators to Xilinx Virtex devices using inheritance and code reuse are summarized at the end of this section.

The major improvements in PAM-Blox II over the initial PAM-Blox [42] implementation, in addition to the object-oriented-design decision mentioned above, are as follows.

1) Use of template classes. A template class is a description of a class that can be instantiated with different variable types as inputs. The most common use of template classes is in the STL. An STL class such as a `vector` can be instantiated as a vector of integers (`vector <int>`), a vector of floats (`vector <float>`), or a vector of any other user-defined class such as `vector <Net>`. The initial PAM-Blox implementation uses template classes to distinguish hardware integers with different bitwidths as different types. PAM-Blox II uses `class Net`. As a consequence, PAM-Blox II treats variables with different bitwidths as variables of the same type with a different attribute (or object state).
2) An object-specific "enable" for control. Sequential modules iterating in parallel for a specific number of clock cycles require a control input to coordinate the number of iterations. For example, a one-cycle adder followed by an N-cycle sequential multiplication requires separate control lines for the two units to be pipelined correctly. *A priori* options are:
   a) provide separate clocks;
   b) add an enable signal to the logic equations (LUT) of the module;
   c) use the enable input of the flip-flop.

   Providing separate clocks is impractical due to the latency of going between clock domains and the limitation of FPGAs to few clock buffers. Enable inputs as part of the object logic (b) are used in the initial PAM-Blox implementation. PAM-Blox II provides a more efficient separate enable line for flip-flops (c) of each hardware object.

In summary, the state of a PAM-Blox II hardware object consists of latency, number of sequential cycles, a list of nested sequential objects, a maximal sequential cycle within the object (for nested objects), size (bitwidth), a hierarchical name for debugging, an enable signal, a clock signal, and an "inputs valid" signal.

A. Portability of Object-Oriented Module Generation

Object-oriented design of hardware-module generators enables code reuse. As a consequence, if a particular feature on the FPGA changes from one product line to another, such as the carry chain, it is easy to adapt the library to a new carry chain by overloading a single method. Overloading this one method then changes the carry chains of all generated modules that require a carry chain, regardless of the function that the module computes. The following discussion describes the object-oriented method of porting FPGA features from one FPGA family to another by using the carry-chain example.

Carry chains form the basis of almost all arithmetic circuits from adders, subtracters, multipliers, and dividers to more specialized units such as counters, comparators, and leading-one-detect circuits. A conventional binary full adder with inputs A and B has the following well-known logic equations:

$$\mathtt{sum_i} = A_i \text{ xor } B_i \text{ xor } \mathtt{carry_{i-1}} \tag{1}$$

$$\mathtt{carry_i} = (A_i B_i) \text{ or } (A_i \mathtt{carry_{i-1}}) \text{ or } (\mathtt{B_i} \mathtt{carry_{i-1}}). \tag{2}$$

For all FPGAs with a dedicated carry chain, the above equations have to be mapped to a four-input lookup table (the lookup table available for logic in the cell) plus some dedicated custom carry logic. The various FPGA families vary in the precise way that this partition is accomplished.

In order to simplify porting PAM-Blox to new carry-chain organizations, the two equations above are described by two separate virtual functions that can be overloaded and inherited. The next step lies within the details of the partition of the carry-chain equations for the two technologies at hand, Xilinx XC4000 and Xilinx Virtex devices.

From Xilinx documentation, we learn that the equations for addition in C++ for Xilinx XC4000 FPGAs are as follows:

$$\mathtt{sum[i]} = \mathtt{A[i]} \mathbin{\hat{}} \mathtt{B[i]} \mathbin{\hat{}} \mathtt{carry[i-1]} \tag{3}$$

$$\begin{aligned}\mathtt{carry[i]} &= (\mathtt{A[i]}\ \&\mathtt{B[i]}) \mid (\mathtt{A[i]}\ \&\mathtt{carry[i-1]}) \mid \\ &\quad (\mathtt{B[i]}\ \&\mathtt{carry[i-1]}) \\ &= \mathtt{mux}\left(\mathtt{A[i]\hat{}B[i]}, \mathtt{carry[i-1]}, \mathtt{ZERO}\right).\end{aligned} \tag{4}$$

For Xilinx XC4000 devices, the dedicated carry chain is inferred by the Xilinx place and route tools based on relative placement constraints that lock the particular wires to positions relative to each other.

For Virtex devices, the equations for addition are as follows:

$$\mathtt{sum[i]} = \mathtt{xorcy}\left(\mathtt{LUT[i]}, \mathtt{carry[i-1]}\right) \tag{5}$$

$$\mathtt{carry[i]} = \mathtt{muxcy}\left(\mathtt{LUT[i]}, \mathtt{ZERO}, \mathtt{carry[i-1]}\right). \tag{6}$$

Specific function calls `muxcy(select, input1, input0)` and `xorcy()` instantiate dedicated carry-chain logic primitives available inside the Virtex logic blocks. The `LUT[]` array describes the logic that goes into the Virtex adders lookup

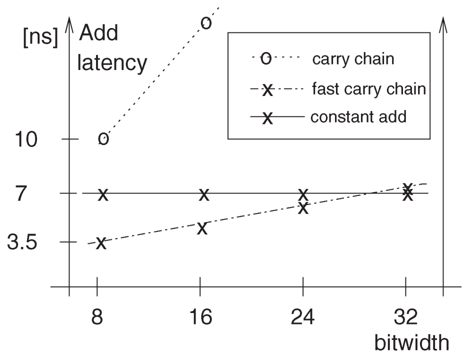

Fig. 4. Latency of addition given three implementation choices: carry chain, (dedicated) fast carry chain, and constant time addition.

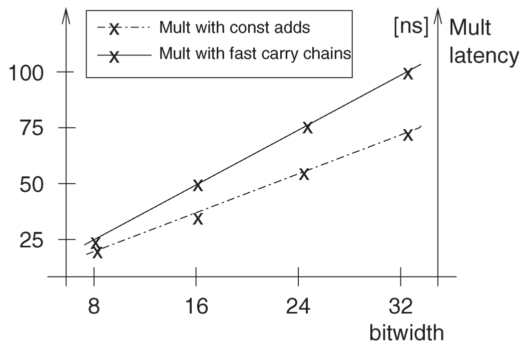

Fig. 5. Latency comparison of two multiplier implementations: with (dedicated) fast carry chains and with internal redundant representation.

table. In the case above, the lookup table holds the exclusive-or of the two inputs, or `LUT[i]` $=$ `A[i]` ^ `B[i]`. Since carry chains use dedicated blocks explicitly, there is no need for relative placement constraints to infer a carry chain such as that necessary for XC4000 FPGAs.

Since the only difference between the two technologies lies in the above two equations, declaring each one of these equations in a separate virtual function enables porting PAM-Blox II by overloading the carry-chain functions of the top ancestor class. Thus, partitioning the logic into appropriate virtual functions is the key to portability of an object-oriented module-generation environment and also provides the key advantages for using object-oriented technology.

VII. Examples of PAM-Blox II Module Generators

In order to demonstrate the custom-designed module generators for computing with FPGAs, we explain the design of a few sample module generators and the impact of having such custom modules available in the module-generator library. The tradeoffs for the module generators are based on trading area for speed, hand-optimizing technology mapping to the specific FPGA microarchitecture, and utilizing a redundant number representation.

The results for latency and area are based on Xilinx VirtexE devices (speedgrade -6) and standard Xilinx Foundation series v3.2 place and route tools.

A. Addition and Subtraction

Addition and subtraction are the most important module generators for computing with FPGAs. The results in this section quantify the advantages of the FPGA's fast carry chain versus redundant representations. Fig. 4 shows the latency of addition given three implementation choices: carry chain, (dedicated) fast carry chain, and constant-time addition.

1) Using Redundant Representations: Redundant representations are one of the key methods to speed up arithmetic circuits in VLSI [52]. Redundant encodings are defined by Omondi [54]. Such redundant digits enable us to tradeoff area (more bits) for time by eliminating the carry chain and obtaining "constant-time addition," where addition time does not depend on the bitwidth of the operands.

Fig. 4 compares carry-chain adders with and without the dedicated fast carry chain and a constant-time adder using the carry-save redundant representation. The carry-save representation requires two bits to represent each digit and, thus, results in a doubling of the required bits to represent a value. The graph in Fig. 4 shows the order of magnitude speedup of carry-chain addition provided by the Xilinx fast carry chain. A single redundant addition is comparable to a 32-bit carry-chain add. Despite that fact, a collection of adders such as that present in an array multiplier (results in Fig. 5) shows significant time savings for redundant adders even for bitwidths smaller than 32 bits. Interestingly, not only does the redundant implementation outperform the multiplier with fast carry chains, but even scaling turns out to work in favor of redundant digits resulting in a smaller slope of the redundant multiplication line in Fig. 5. This surprising result is due to the structure of redundant representations. Most of the delay is in the interconnect to and from the unit. By placing multiple units together, Xilinx place and route tools can minimize this interconnect delay and thus optimize the performance of the combined circuit.

As for area, redundant multipliers are about 5% smaller than carry-chain-based implementations. The area advantage results from a slightly higher utilization of FPGA resources due to technology mapping of conventional (3, 2) counters [53], which are the basic building blocks for computing with a redundant representation. A further optimization of multipliers for computing with FPGAs can be applied to constant multiplication, as shown in a previous paper [42].

B. Comparison Operator $==$

A common computation is to check if two values are equal. Looking at the problem in a top down approach, one might consider using a subtracter and checking if the result is zero. Given the flexibility at the bit level, there are two interesting solutions; one for checking the equality of a variable and a constant, and one for checking the equality of two variables. Optimizing for area and latency, respectively, one could implement the comparison operation: 1) with a carry chain or 2) with a parallel treelike (suffix, close up) implementation.

A closer look at the implementation for comparing a variable with a constant shows that the carry-chain version can be a subclass of an adder. Such a modified adder then simply requires the overloading of the carry chain's LUT functionality, which is a separate virtual function within the adder. The code fragment below shows one version of the PAM-Blox II code defining a comparison between a variable `A` and a constant `K` at bit position `i`

```
virtual EquationHandler LUT(int i){
return ((((K>>i)&1) ? A[i]   :~A[i])&
    (((K>>(i+1))&1) ? A[i+1] :~A[i+1])&
    (((K>>(i+2))&1) ? A[i+2] :~A[i+2])&
    (((K>>(i+3))&1) ? A[i+3] :~A[i+3]));
}
```

This code implies that a single four-input LUT compares up to four bits against a constant value. As a consequence, the area of the resulting unit is four times smaller than a subtracter and delivers the result of the comparison on the carryout wire of the unit. A similar construction for comparing two variables leads to a unit of half the size of a subtracter.

A treelike (suffix, close up) implementation still reduces up to four bits per lookup table; but instead of a carry chain, the result is obtained by reducing the input in a treelike fashion. The PAM-Blox II code for such a reduction tree is slightly more arduous.

Comparing two variables limits the number of bits that can be compared in one lookup table to two bits of each input variable. As a consequence, for the carry-chain solution, comparing two variables takes about twice the area of comparing a variable to a constant and about half the area of a subtracter.

From standard VLSI experience, we expect a circuit with a hierarchical or tree-based solution to be faster than a carry chain. From an FPGA designer's view, we expect any solution that uses the fast carry chain to be superior. The results show that the dedicated fast carry-chain solution is in fact faster than the hierarchical solution.

One of the conclusions from this result is that knowledge from VLSI design is not directly applicable to FPGA design on the module-generation level despite the fact that both are hardware-design methodologies. The difference arises from the particular LUT and interconnect structure of FPGAs and the associated technology mapping, placement, and routing.

VIII. Testing ASC and ASC Programs

ASC provides a test infrastructure that automates testing and precision analysis of the hardware generated by ASC. This testing feature leads to a regression test suite and a simple mechanism for the ASC programmer to write or utilize an existing software version of the ASC program. ASC automatically runs a whole series of tests, which can be defined and parametrized in the makefile.

A test consists of executing: 1) a pure software version of the code and 2) either a gate-level simulation (PamDC/C++ simulation of circuit on the gate level) or the actual hardware running on an FPGA in real time. The outputs of the two executions are automatically compared against each other. The tests can either be specified to check for equivalence of software and simulation/hardware, or the user can specify an error bound. With the error bound, ASC ensures that the error of finite precision arithmetic (e.g., 12-bit multiplication) in the hardware does not exceed the error limit when compared to the software version. The software versions can be written using the processor's data-types such as the double precision IEEE floating point or 32/64-bit integers. The result of a test is a message that the test succeeded or failed. In case of a failure, additional information about the failure case is provided.

We identified verification as an imperative task, and ASC contains substantial support and infrastructure for regression testing and verification of resulting circuits. For example, to illustrate the accuracy of the hardware, ASC enables plotting error graphs that show the error of hardware/simulation over the software version as a function of input values.

IX. Design-Space-Exploration Case Studies

In this section, three benchmarks—wavelet compression, Kasumi encryption, and rotation and elementary functions—are used to illustrate and to evaluate our approach. The first few benchmarks demonstrate three main kinds of design-space exploration: loops (architecture level), the arithmetic-unit level, and the bit level.

A. Wavelet Compression

The first benchmark we evaluate is wavelet compression, which is based on a piece of code from a wavelet library [9]. The code is implemented using `HWfix` variables of 20 bits with the binary point after the 14th fractional bit. The declarations of the variables show the usage of default values for variable attributes, such as sign-mode and bitwidth, and the `HWvector` declaration, which mirrors the functionality of `vector` in the C++ STL

```
DefaultSign = TWOSCOMPLEMENT; // sign
DefaultSize = 20; // bitwidth
DefaultFract = 14; // fractional bits

HWfix in1 (IN), in2 (IN), // declare IO
        out1 (OUT), out2 (OUT);
HWfix low, high, temp, temp2, coeff;

// vectors of HWfix streams
HWvector<HWfix> v_temp1(4, new HWfix
   (TMP));
HWvector<HWfix> v_temp2 (5, new HWfix
   (TMP));
HWvector<HWfix> lc1 (4, new HWfix (TMP));
HWvector<HWfix> lc2 (5, new HWfix (TMP));
HWvector<HWfix> hc1 (4, new HWfix (TMP));
HWvector<HWfix> hc2 (5, new HWfix (TMP));
```

The algorithm consists of two consecutive loops. Each loop can be unrolled in hardware, or ASC can generate an actual feedback loop in the hardware. ASC provides two main loop constructs `LOOP` and `UNROLL_LOOP`, which explicitly create a feedback connection or unroll the loop body. The control flow can be handled by the functional-style `IF` construct, which stands for `IF(condition, true, false)`. If the condition is true, the second argument streams to the output; while if the condition is false, the third argument proceeds. The following ASC code shows how the user can explore the design space for loops in ASC

```
#ifndef UNROLL1
  HWint idx1 (TMP, 5);
  idx1 = 0;
  LOOP (size1_2); // hardware loop
#else
  int idx1 = 0; // fully unrolled
  UNROLL_LOOP(int i = 0;i < size1_2;i++){
#endif
    temp2 = v_temp1 [idx1<<1];

    coefficient = IF(idx1,lc1[3],lc1[1]);
    low = low + (coeff*temp2);

    coefficient = IF(idx1,hc1[3],hc1[1]);

    high = high + (coefficient*temp2);
    temp2 = v_temp1 [(idx1<<1)+1];

    coefficient = IF(idx1,lc1[2],lc1[0]);
    low = low + (coefficient*temp2);

    coefficient = IF(idx1,hc1[2],hc1[0]);
    high = high + (coefficient*temp2);
    idx1++;
#ifndef UNROLL1
  LOOP_END(); // feedback hardware loop
#else
  }
#endif
```

Notice that in the case of unrolling, the loop index variable is an integer. In the case of a loop in hardware, the index variable is a `HWint`. A major consequence of unrolling is that all array indexing can be done at compile time, thus saving a lot of area for dynamic array accessing. Also, all arithmetic involving the integer `idx1` can now be implemented as constant arithmetic, i.e., PAM-Blox modules for constant multipliers and adders, etc.

B. Kasumi Encryption

The second application we examine is Kasumi encryption [13], which is part of the 3G standard for wireless communication.

Key opportunities for exploring parallelism at the bit level are in the `FL()` and `FO()` function calls (S-boxes), which are implemented as table lookups in the software version. In the standard specification, these are provided as both lookup tables and logic functions. When creating application-specific hardware, we convert these tables into Boolean equations, which can be minimized with a logic minimization algorithm. Given enough symmetries in these tables, the resulting circuit can be made smaller and faster than the corresponding hardware tables.

ASC allows the user to exploit BLP by creating custom PAM-Blox modules at the bit level. The user creates modules by extending the PAM-Blox class library with a new module (subclass) and creating a function call that access that particular new module from the ASC code level, as shown in the code below

```
void
kasumi(Kstate *ks,HWvector<HWint> &data){
  HWint &l(*new HWint(TMP,32,UNSIGNED));
  HWint &r(*new HWint(TMP,32,UNSIGNED));
  HWint &t1(*new HWint(TMP,32,UNSIGNED));
  HWint &t2(*new HWint(TMP,32,UNSIGNED));
  l = data[0];
  r = data[1];

#if USE_LOOP
  HWint i(TMP,6,UNSIGNED);
  i = 0;
  STREAM_OPTIMIZE = AREA;
  LOOP(4);
#else
  unsigned int i;
  UNROLL_LOOP (i = 0;i < 8;) {
#endif

  t1  = FL(ks, l, i);
  r ^ = FO(ks, t1, i);
  t2  = FO(ks, r, i+1);
  l ^ = FL(ks, t2, i+1);
  i   = i + 2;

#if USE_LOOP
  LOOP_END(); // feedback
#else
  }
#endif

  data[0] = l; // assign outputs
  data[1] = r;
}
```

Our implementation of the `FL()` and `FO()` functions has a user configurable parameter to indicate whether the circuit should use a lookup table (held in on-chip SRAM such as Xilinx block RAMs) or a direct implementation of the above. Thus, the user can decide to use available block RAMs when porting the code to save area or create the custom logic to achieve maximal performance.

C. Rotation and Elementary Functions

The third application computes elementary functions sine and cosine for a coordinate-rotation unit. We use polynomial approximations to generate sine and cosines. The coordinate

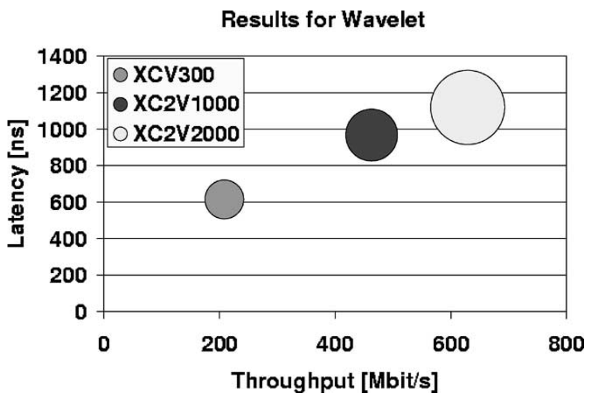

Fig. 6. Results for the wavelet design space exploration showing the best throughput performers for each of the three FPGA sizes. The size of the circle indicates the area of the design.

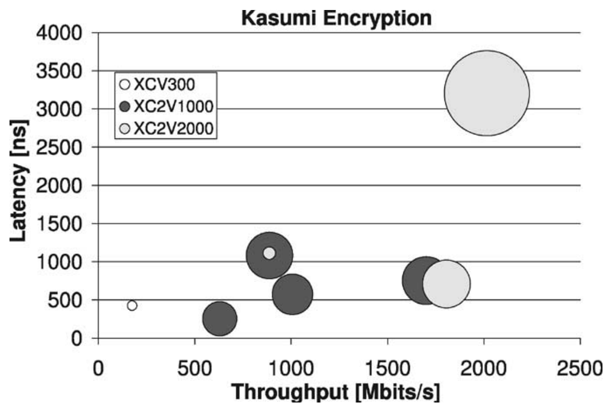

Fig. 7. Kasumi design space exploration with ASC, using a bubble chart. The size of the bubble corresponds to the area of the circuit. The color of the bubble shows the particular FPGA (XC...) used.

rotation performs a pair of 2-D rotations through input angles written to memory-mapped registers. The coordinates are then streamed in, and the rotated coordinates are streamed out of the ASC pipeline.

The use of `Default` and `STREAM_OPTIMIZE` variables enables exploration of the design space. Changing these options alters the size of hardware variables or the optimization mode of the logic blocks; this creates a widely differing range of hardware implementations.

The code below is the rotation function, demonstrating how the `Default` and `STREAM_OPTIMIZE` variables can be used to explore the design space. In the case below, we vary bitwidth for each of the optimization modes

```
STREAM_START;
DefaultSign = SIGNMAGNITUDE;
// THROUGHPUT, LATENCY or AREA
STREAM_OPTIMIZE = THROUGHPUT;
DefaultSize = 26;
DefaultFract = 21;
HWfix x(IN), y(IN), z(IN);
HWfix outx(OUT), outy(OUT), outz(OUT);
HWfix phi (MAPPED_REGISTER);
HWfix delta (MAPPED_REGISTER);
HWfix cosP(TMP), cosD(TMP);
HWfix sinP(TMP), sinD(TMP);

// runtime stream length parameter
STREAM_LOOP (10);

cosP = cos(phi);
cosD = cos(delta);
sinD = sin(delta);
sinP = sin(phi);
outx = x*cosD-z*sinD;
outy = y*cosP+x*sinP*sinD+z*sinP*cosD;
outz = x*sinD-z*cosP-y*sinP+z*cosD*cosP;
STREAM_END;
```

The bubble chart in Fig. 6 shows the design space for the wavelet compression example. We explore the latency, the throughput, and the FPGA area, which is shown as the size of the bubbles. The tradeoffs between the various implementations are based on different loop-unrolling decisions. The smallest design has no unrolling, the middle design unrolls once, and the large implementation is fully unrolled for maximal throughput. Since each of the bubbles corresponds to the maximal throughput for a particular FPGA size, we observe the general activity of trading area for performance. ASC enables us to obtain a larger FPGA and increase performance by recompiling to a larger area, without changes to the source code. The modifications are limited to the parametrizations of the source code, which can be located in the makefile.

Fig. 7 shows the results of design-space exploration for Kasumi encryption using ASC. The bubbles in the figure correspond to a complete design with a particular set of parameters; which includes loop unrolling and optimization modes such as latency, area, and throughput. The area restrictions for each particular FPGA limit the number of optimizations that can be employed. Also, the figure shows only a part of the complete design space across all levels of abstraction.

The third set of results shows the design space for the rotation example. As in the previous two examples, a bubble chart in Fig. 8 shows the design space for varying precision (bitwidth) and different optimization modes. Given the user's precision requirements, it is possible to optimize down to the individual variable bitwidth. By reducing the bitwidth of each variable, especially for multipliers and table indices, considerable savings in time and space can be observed.

In addition, Figs. 9 and 10 show the design-space tradeoff when varying the bitwidth of the variables. The graphs show the impact of optimizing latency, throughput, or area across different bitwidths. Note an interesting artifact in the throughput result in Fig. 9: When increasing bitwidth with mainly constant multiplication (e.g., `cosP`), the throughput remains close to flat despite increasing complexity of the multiplication. Logic minimization seems to get us to the same clock frequency for different bitwidths in this particular case. Therefore, in this case, we can conclude that bitwidth is not forming a critical bottleneck at these particular bitwidth values.

Also, while throughput optimization clearly increases the throughput of implementations, there are still surprises

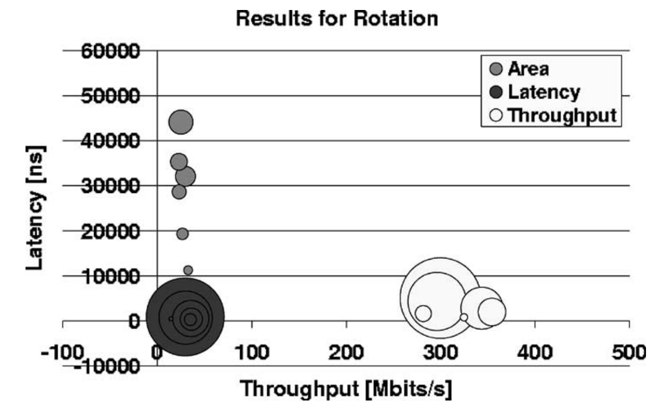

Fig. 8. Rotation example—exploration of design space—using a bubble chart. The size of the bubble corresponds to the area of the circuit, when optimizing for area, latency, or throughput (agenda). The different bubbles of the same color correspond to different bitwidths.

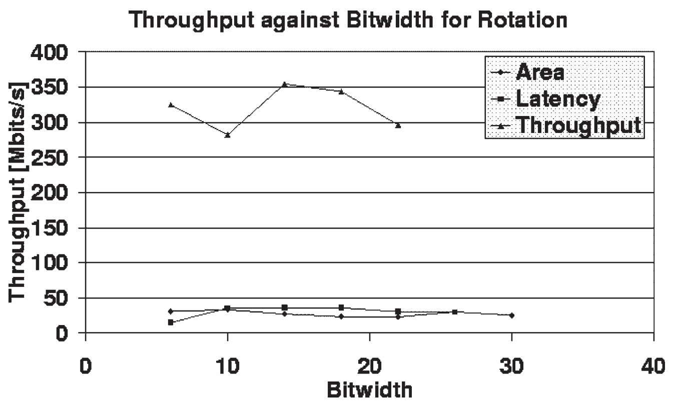

Fig. 9. Impact of bitwidth on the throughput of the implementation, when optimizing for area, latency, or throughput (agenda).

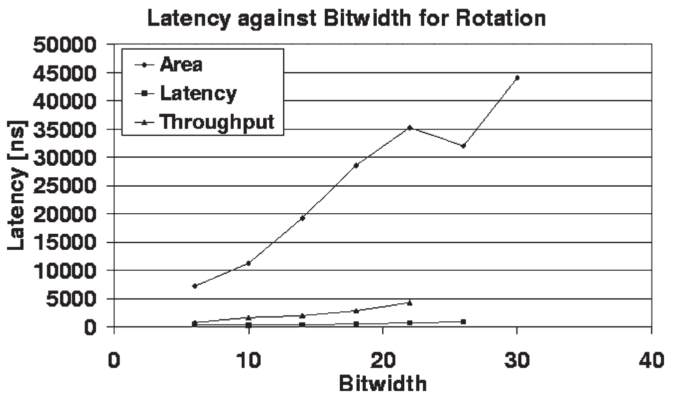

Fig. 10. Impact of bitwidth on the latency of the implementation, when optimizing for area, latency, or throughput (agenda).

sometimes; such as the throughput for different bitwidths, which exhibits artifacts from the discrete nature of technology mapping and place and route. An example of such deviation from the general shape is in the latency figure for 26 bits: the throughput line shows a lower value than expected.

X. Performance Results

Results are obtained using a conventional GCC compiler and current Xilinx tools under Windows. We run ASC on

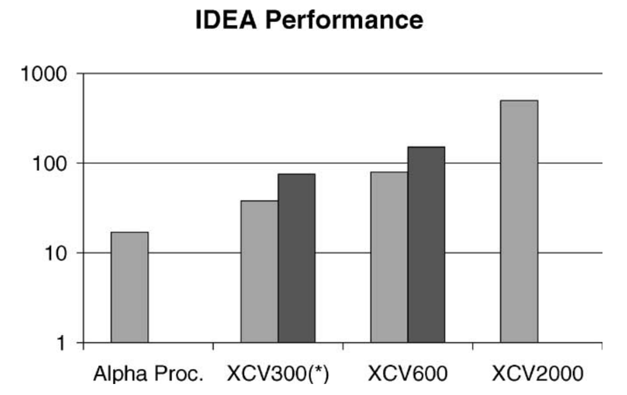

Fig. 11. Performance [Mb/s] of IDEA Encryption on a Compaq Alpha processor and a range of Xilinx Virtex devices. (*)This implementation is run on the Wildcard board.

Windows/Cygwin and Linux, since these are the platforms for which we can get Xilinx tools. ASC itself requires GCC and can be compiled on any system supporting GCC.

We simulate the implementations on the gate level by compiling ASC code with GCC and by running the program in simulation mode. The gate-level simulation is provided by PamDC. Since ASC can target any FPGA board, the reported results show FPGA peak performance without taking into account board level bottlenecks.

The two case studies are IDEA encryption and lossless compression.

1) Idea Encryption: The IDEA encryption serves to demonstrate the effects of the above redundant multipliers on the performance of an application. The IDEA encrypts or decrypts 64-bit data blocks, using symmetric 128-bit keys. The 128-bit keys are expanded further to 52 subkeys, with 16 bits each. A single algorithm uses different keys for encryption and decryption. The inner loop is repeated eight times and consists of operations: XOR, multiplication, addition, and $(\bmod\ 2^{16}+1)$.

Fig. 11 shows a performance comparison of running the inner loop of IDEA encryption on an Alpha EV5.6 (21164A) processor operating at 532 MHz, compiled with the native Alpha C compiler, and a series of Xilinx VirtexE FPGAs (speedgrade -6). The FPGA designs include glue logic for the Wildcard [56] from Annapolis Microsystems with a Xilinx XCV300E device. The implementation for the Wildcard (XCV300E) utilizes 99% of the FPGA's lookup tables.

The performance results show a speedup of about two times for the conventional XCV300E implementation without "redundant multipliers" and another factor of two speedup with "redundant multipliers" for the XCV300E and the XCV600E, using the redundant adders from Section III. In the case of the XCV2000E, the design is fully unrolled and thus throughput does not depend on the latency of the operations. Since redundant multipliers only help with latency, and the XCV2000E implementation is fully unrolled and pipelined; latency does not impact performance/throughput, and thus redundancy does not generate a performance bar for the XCV2000E. Since redundant multipliers only help with latency, there is only one performance bar for the XCV2000E.

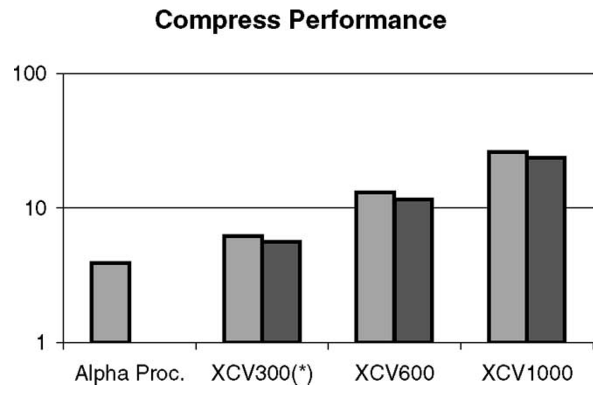

Fig. 12. Performance [Mb/s] of compression on an Alpha processor and a range of Virtex devices. (*)This implementation is run on the Wildcard board.

2) Lossless Compression: The results for compression demonstrate the effects of optimal comparison units on compression performance.

Lempel–Ziv (LZ) compression has many variations. In this example, we implement a very simple form of LZ-like compression where we look at D bytes of history and try to match a string up to length D into the future. As a consequence, the implementation consists of a 2-D array of comparison units.

Fig. 12 shows the performance comparison of our variant of LZ compression with $D = 26$, using the same methodology as in the previous example. The implementation for the Wildcard/XCV300E utilizes 99% of the configurable logic blocks (CLBs). The results show that the 10% improvement in cycle time of the stand-alone compare units, described in Section III, scales to a 10% performance improvement for our variant of LZ compression. In general, ASC allows us to explore low-level optimizations and quickly study their impact on complete application performance.

XI. Conclusion

The results presented above show a wide range of optimizations that can be undertaken within the ASC system. Optimizations on the algorithm level, the architecture level, the arithmetic level, and the bit level can be explored within the same C++ program. On the architecture generation level, ASC enables the exploration of area, latency, and throughput tradeoffs for hardware design and accelerator generation especially. Moreover, ASC is a platform for tools that automate the exploration of the area-time design space. On the module-generation level, PAM-Blox II is a core enabling technology for computing with FPGAs. It enables the programmer to take full advantage of the bit-level flexibility of FPGAs. This flexibility enables us to explore BLP, in addition to parallelism on higher levels of abstraction.

Although the user does have expanded design space with variable granularity (bit level to architecture level), obviously, bit-level and architecture-specific optimizations require the user to change the source code for changes in the target technology family—such as if Virtex4 was the desired target architecture. This may require considerable effort but is not avoidable due to the nature of manual bit-level optimization. Also, ASC is technology specific to Xilinx, a limitation that other similar tools do not have but which enables the user to optimize for the Xilinx architecture-specific low-level features. Clearly, this is a limitation of project resources rather than the methodology itself.

On the language side, careful utilization of C++ features yields an efficient abstraction for the development, maintenance, and extension of a large module-generator library and an architecture generation layer such as present in ASC. Concrete conclusions from the sample module generators shown in this paper are as follows.

1) A single redundant (constant latency) addition is comparable to a 32-bit carry-chain add. Despite that fact, a collection of adders such as present in an array multiplier shows significant time savings for redundant adders even for bitwidths smaller than 32 bits. This surprising result can be explained by finding that most of the delay of a stand-alone constant-time adder can be optimized away when compiling a whole set of such adders, while the delay through the carry chain is fixed by the technology.
2) Knowledge from VLSI design is not directly applicable to FPGA design despite the fact that both are hardware-design methodologies. In particular, design tradeoffs depend largely on the available resources in the FPGA cell and on the optimality of technology mapping, which can be controlled within the module-generation layer.
3) The object-oriented design of module generators allows us to retarget ASC to multiple Xilinx FPGA families such as the Xilinx 4000, Virtex, Virtex 2, Virtex 4, Spartan 2, and Spartan 3. In addition to Xilinx FPGAs, we also consider porting PAM-Blox II to Altera devices. However, any such effort is complicated by the artificial incompatibility of Xilinx and Altera netlists on the AND/OR gate level, even though both netlists are in standard EDIF format. As a consequence, this effort is left for future work.

Our experience substantiates that ASC simplifies hardware design. In fact, most of the ASC application codes presented in this paper are developed by C++ programmers rather than hardware designers. With ASC, the hardware-design productivity and the complexity of the description are close to software development.

We foresee another layer of software on top of ASC architecture generation, which will automate tasks such as precision analysis, loop transformations, memory management generation, and partitioning of an application into software and hardware accelerators. In addition, such a high-level transformation layer will be able to deal efficiently with data structures. The combination of these techniques has the potential to attack the memory wall [57] and result in productive interactions between FPGA research results and microprocessor-centered research. Clearly, some of the high-level transformations will not be fully automatable in the near future. Some transformations will have to be partially automated in conjunction with user hints. Minimizing the user hints necessary for successful acceleration across a wide range of applications is one of the long-term goals.

Acknowledgment

The author would like to thank W. Luk, M. J. Flynn, M. Morf, and C. Young for the discussions and support of ASC efforts; D. Pearce, J. Liang, G. Huang, H. Styles, and L. Howes for helping to advance ASC and for writing some of the code examples; and M. Shand for keeping PamDC up to date with current Xilinx technology, for his efforts in porting PamDC to GNU GCC, and for his help with various PamDC related issues.

References

[1] S. G. Abraham and B. R. Rau, "Efficient design space exploration in PICO," in *Proc. Int. Conf. Compilers, Architecture and Synthesis Embedded Systems (CASES)*, San Jose, CA, Nov. 2000, pp. 71–79.

[2] S. Aditya and M. S. Schlansker, "ShiftQ: A bufferred interconnect for custom loop accelerators," in *Proc. Int. Conf. Compilers, Architecture and Synthesis Embedded Systems (CASES)*, Atlanta, GA, Nov. 2001, pp. 158–167.

[3] Arvind, "Can dataflow subsume von Neumann computing?" in *16th Int. Symp. Computer Architecture (ISCA)*, Jerusalem, Israel, May 1989, pp. 262–272.

[4] K. Bondalapati and V. K. Prasanna, "Dynamic precision management for loop computations on reconfigurable architectures," in *IEEE Symp. FPGAs Custom Computing Machines*, Napa, CA, Apr. 1999, pp. 249–258.

[5] L. Bossuet, G. Gogniat, and J.-L. Philippe, "Fast design space exploration method for reconfigurable architectures," in *Proc. Int. Conf. Engineering Reconfigurable Systems and Algorithms*, Las Vegas, NV, 2003, pp. 65–71.

[6] M. Budiu, S. C. Goldstein, K. Walker, and M. Sakr, "BitValue inference: Detecting and exploiting narrow bitwidth computations," in *Eur. Conf. Parallel Processing (Euro-Par)*, Munich, Germany, Aug. 2000, pp. 969–979.

[7] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *IEEE Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[8] Celoxica. *Handel-C Language Reference Manual*. [Online]. Available: http://www.celoxica.com/

[9] G. Davis, J. Danskin, and R. Heasman, *Wavelet Image Compression Construction Kit, Version 0.3*. [Online]. Available: http://www.geoffdavis.net/dartmouth/wavelet/wavelet.html

[10] A. Dhodapkar and J. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proc. Int. Symp. Computer Architecture (ISCA)*, Anchorage, AK, May 2002, pp. 233–244.

[11] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the streams-C C-to-FPGA compiler: An applications perspective," in *Proc. IEEE Field Programmable Gate Arrays (FPGA) Conf.*, Monterey, CA, Feb. 2001, pp. 134–140.

[12] M. Gokhale, J. Kaba, A. Marks, and J. Kim, "Malleable architecture generator for FPGA computing," *Proc. SPIE*, vol. 2914, pp. 208–217, Oct. 1996.

[13] Kasumi encryption algorithm. *3G Wireless Standard*. [Online]. Available: http://www.3gpp.org/

[14] P. H. W. Leong, M. P. Leong, O. Y. H. Cheung, T. Tung, C. M. Kwok, M. Y. Wong, and K. H. Lee, "Pilchard—A reconfigurable computing platform with memory slot interface," in *Proc. IEEE Symp. FPGAs Custom Computing Machines*, Apr. 2001, pp. 170–179.

[15] Y. Li *et al.*, "Hardware-software co-design of embedded reconfigurable architectures," in *Proc. Design Automation Conf.*, Los Angeles, CA, 2000, pp. 507–512.

[16] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for FPGAs," in *Proc. IEEE Symp. FPGAs Custom Computing Machines*, Napa, CA, Apr. 2003, pp. 185–194.

[17] M. Macedonia, "The computer graphics war heats up," *IEEE Computer*, vol. 35, no. 10, pp. 97–99, Oct. 2002.

[18] O. Mencer, "PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs," in *Proc. IEEE Symp. FPGAs Custom Computing Machines*, Napa, CA, Apr. 2002, pp. 67–76.

[19] O. Mencer and W. Luk, "Tutorial: Computing with FPGAs," presented at the Int. Symp. Computer Architecture (ISCA), Anchorage, AK, May 2002.

[20] O. Mencer, M. Platzner, M. Morf, and M. Flynn, "Object-oriented domain-specific compilers for programming FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (Special issue on Reconfigurable Computing)*, vol. 9, no. 1, pp. 205–210, Feb. 2001.

[21] T. Sherwood and B. Calder, "Automated design of finite state machine predictors for customized processors," in *Proc. Int. Symp. Computer Architecture (ISCA)*, Göteborg, Sweden, Jun. 2001, pp. 86–97.

[22] B. So, P. Diniz, and M. Hall, "Using estimates from behavioral synthesis tools in compiler-directed design space exploration," in *Proc. ACM/IEEE 40th Design Automation Conf.*, Anaheim, CA, Jun. 2003, pp. 514–519.

[23] B. So, M. Hall, and P. Diniz, "A compiler approach to fast design space exploration in FPGA-based systems," in *Proc. ACM Conf. Programming Language Design and Implementation (PLDI)*, Berlin, Germany, Jun. 2002, pp. 165–176.

[24] M. Weinhardt and W. Luk, "Pipeline vectorization," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 2, pp. 234–248, Feb. 2001.

[25] Xilinx, *Virtex-E and Virtex II Pro FPGA Datasheet*. [Online]. Available: http://www.xilinx.com/

[26] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable active memories: A performance assessment," in *ACM Field Programmable Gate Arrays (FPGA)*, Feb. 1992.

[27] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *Splash-2, FPGAs in a Custom Computing Machine*. Los Alamitos, CA: IEEE Comput. Soc. Press, 1996.

[28] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, and H. Spaanenburg, "Seeking solutions in configurable computing," *IEEE Computer*, vol. 30, no. 12, pp. 38–43, Dec. 1997.

[29] O. Mencer, M. Morf, and M. Flynn, "Hardware software tri-design of encryption for mobile communication units," presented at the Int. Conf. Application Specific Signal Processing, Seattle, WA, May 1998.

[30] O. Mencer and M. Morf, "CORDICs for reconfigurable computing," presented at the *6th FPGA/PLD Design Conf. and Exhibit*, Yokohama, Japan, Jun. 24–26, 1998.

[31] M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," in *Proc. 11th IEEE Symp. Computer Arithmetic*, Windsor, ON, Canada, 1993, pp. 252–259.

[32] F. F. Lee, "A scalable computer architecture for lattice gas simulation," Ph.D. thesis, Dept. Elect. Eng., Stanford Univ., Stanford, CA, Jun. 1993.

[33] H. Styles and W. Luk, "Customising graphics applications: Techniques and programming interface," in *Proc. IEEE Symp. Field Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr. 2000, pp. 77–87.

[34] S. D. Haynes, P. Y. K. Cheung, W. Luk, and J. Stone, "Video image processing with the SONIC architecture," *IEEE Computer*, vol. 33, no. 4, pp. 50–57, Apr. 2000.

[35] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation," in *Proc. ACM Conf. Programming Language Design and Implementation*, Vancouver, BC, Canada, Jun. 2000, pp. 108–120.

[36] R. Razdan, "PRISC: Programmable reduced instruction set computers," Ph.D. thesis, Div. Appl. Sci., Harvard Univ., Cambridge, MA, May 1994.

[37] L. Semeria, "Applying pointer analysis to the synthesis of hardware from C," Ph.D. thesis, Dept. Elect. Eng., Stanford Univ., Stanford, CA, Jun. 2001.

[38] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz, "Cool-cache for hot multimedia," in *Proc. MICRO-34 Conf.* Austin, TX, Dec. 2001, pp. 274–283.

[39] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee, "The impulse memory controller," *IEEE Trans. Comput.*, vol. 50, no. 11, pp. 1117–1132, Nov. 2001.

[40] O. Mencer, H. Huebert, M. Morf, and M. J. Flynn"StReAm: Object-oriented programming of stream architectures using PAM-blox," in *Field-Programmable Logic and Applications*. Berlin, Germany: Springer-Verlag, 2000, vol. 1896, pp. 595–604.

[41] *The Xtensa Processor*. [Online]. Available: http://www.tensilica.com/

[42] O. Mencer, M. Morf, and M. J. Flynn, "PAM-Blox: High performance FPGA design for adaptive computing," in *Proc. IEEE Symp. FPGAs Custom Computing Machines*, Napa, CA, 1998, pp. 167–174.

[43] Synopsys. [Online]. Available: http://www.synopsys.com/products/fpga/fpga_express.html

[44] P. Bertin and H. Touati, "PAM programming environments: Practice and experience," in *Proc. IEEE Workshop FPGAs Custom Computing Machines*, Napa, CA, Apr. 1994, pp. 133–138.

[45] J. Kunkel and K. Kranen, "SystemC demonstrates rapid progress," *Electron. Eng. Times*, Sep. 2000.

[46] ——, "Celoxica adds simulator, debugger to Handel-C compiler," *Electron. Eng. Times*, Feb. 2001.

[47] S. A. Guccione and D. Levi, "XBI: A java-based interface to FPGA hardware," in *Proc. SPIE Photonics East Conf.*, J. Schewel, Ed, Nov. 1998, pp. 97–102.

[48] A. Frey, G. Berry, P. Bertin, F. Bourdoncle, and J. Vuillemin, *Jazz is a high-level programming language for expressing ... large digital synchronous circuits*. [Online]. Available: http://www.exalead.com/jazz/

[49] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A CAD suite for high-performance FPGA design," presented at the IEEE Symp. Field Programmable Custom Computing Machines (FCCM), Napa, CA, Apr. 1998.

[50] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Reading, MA: Addison-Wesley, 1997.

[51] H. Boehm, "Space efficient conservative garbage collection," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Albuquerque, NM, Jun. 1993, vol. 28, 6, pp. 197–206.

[52] D. S. Phatak, T. Goff, and I. Koren, "Constant-time addition and simultaneous format conversion based on redundant binary representation," *IEEE Trans. Comput.*, vol. 50, no. 11, pp. 1267–1278, Nov. 2001.

[53] I. Koren, *Computer Arithmetic Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[54] A. Omondi, *Computer Arithmetic Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[55] O. Mencer and W. Luk, "Parameterized high throughput function evaluation for FPGAs," *J. VLSI Signal Process. (Special Issue on Field Programmable Logic)*, vol. 36, no. 1, pp. 17–25, 2004.

[56] Annapolis Microsystems. *Wildcard, A Cardbus Based FPGA Accelerator card*. [Online]. Available: http://www.annapmicro.com/

[57] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[58] W. Luk and S. McKeever, "Pebble: A language for parametrised and reconfigurable hardware design," in *Proc. Field-Programmable Logic and Applications (FPL) Conf.,* Tallinn, Estonia, 1998, pp. 9–18.

Oskar Mencer received the B.Sc. degree in computer engineering at the Technion—Israel Institute of Technology, Haifa, Israel, and the M.S. and Ph.D. degrees from the Computer Systems Laboratory at Stanford University, Stanford, CA.

His early work on object-oriented hardware design was mentioned in an EE-Times top technology story in 1998. In 2000–2003, he was a member of Technical Staff at the Computing Sciences Center at Bell Labs, where he led the development of a stream compiler (ASC) for computing with field programmable gate arrays (FPGAs) and founded Maxeler Technologies Inc. in 2003. He holds two patents and is the author or coauthor of over 30 publications. His main interests are in the field of computer architecture, computer-aided-design (CAD), very large scale integration (VLSI), and FPGAs.

Dr. Mencer currently holds a 5-year Engineering and Physical Sciences Research Council (EPSRC) Advanced Fellowship.