Design Space Exploration with A Stream Compiler

Oskar Mencer, David J. Pearce, Lee W. Howes, Wayne Luk

Department of Computing, Imperial College, London, UK.

Abstract

We consider speeding up general-purpose applications with hardware accelerators. Traditionally hardware accelerators are tediously hand-crafted to achieve top performance. ASC (A Stream Compiler) simplifies exploration of hardware accelerators by transforming the hardware design task into a software design process using only 'gcc' and 'make' to obtain a hardware netlist. ASC enables programmers to customize hardware accelerators at three levels of abstraction: the architecture level, the functional block level, and the bit level. All three customizations are based on one uniform representation: a single C++ program with custom types and operators for each level of abstraction.

This representation allows ASC users to express and reason about the design space, extract parallelism at each level and quickly evaluate different design choices. In addition, since the user has full control over each gate-level resource in the entire design, ASC accelerator performance can always be equal to or better than hand-crafted designs, usually with much less effort. We present several ASC benchmarks, including wavelet compression and Kasumi encryption.

1. Introduction

Traditionally computer systems consist of a microprocessor that pushes the limits of current technology, and an additional set of application or domain specific devices, or *hardware accelerators*, that accelerate certain functionality. Examples are: floating point co-processors in early microprocessor systems, 2D and 3D graphics accelerator cards, and combinations of software and hardware accelerators in embedded systems. However, all these hardware accelerators are tediously hand-crafted to achieve top performance.

The ASC (A Stream Compiler) system has been developed to automate the design of hardware accelerators. Previous publications have covered the spirit [20] and the module generation component [18] of our approach, together with its use for floating-point unit generation [16]. This paper explains how ASC can be used for design space exploration. In particular, it describes:

- the ASC system, including the stream architecture and the levels of abstractions,
- the hardware types and attributes for producing the ASC datapath and memory systems,
- a number of benchmarks, including wavelet compression and Kasumi encryption, that illustrate the ASC approach.

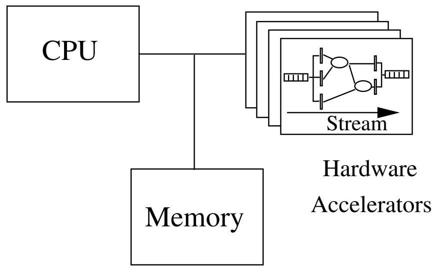

Figure 1: A computer system with hardware accelerators such as Stream Architectures.

In the following, we provide an overview and motivations for our work.

Figure 1 shows the general structure of a computer system with multiple application specific accelerators. The accelerator is located either on-chip with the processor such as today's floating point units, the Berkeley Garp Processor [7], the Xilinx Virtex Pro FPGAs (Field Programmable Gate Arrays) with on-chip PowerPC processors [27]; or such accelerators are combined with main memory [14], or on the peripheral bus[17][12]. These accelerators can be implemented in custom VLSI devices, or as FPGA configurations. Recent advances in FPGA technology enable the development of many hardware accelerators customised for specific applications and for particular input data-sets [19]. These accelerators can be generated and managed at compile time and at run time.

Building efficient hardware accelerators for a particular application, however, 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 results in the datapath part of the design. Second, a control block for this datapath needs to make sure that the timing of operations is correct. Third, an interface between the accelerator and the processor should maximize the data transfer rate and minimize latency. Fourth, a custom accelerator requires a custom memory system, consisting of on-chip registers, SRAM memory, and possibly DRAM memory. Fifth, there should be run-time routines to take care of sending the appropriate data back and forth between the processor and the accelerator.

ASC facilitates design space exploration in two ways. First, for the datapath, a single ASC description can be used to produce multiple datapath implementations at the micro architecture level with user-specified trade-offs. ASC also simplifies the process of selecting and possibly custom designing the functional units, by having descriptions in various levels of abstractions captured in a uniform, object oriented style.

Second, ASC automates the other tasks mentioned above, including control block generation, run-time routine generation, and interface generation; our purpose is to put the design space exploration under user control. 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.

2. ASC – A Stream Compiler

This section provides an introduction to ASC, and explains how it can be used for design exploration. On the top level, the user writes ASC code which closely resembles C code. As such, existing C code can be used with some small modifications to generate ASC hardware accelerators. In order to express and explore the design space for a hardware accelerator, ASC code can be parameterized to generate a large selection of implementations. With these parameterizations the user can, for example, trade off silicon area for latency and/or throughput. As such, ASC is a tool for expressing and exploring the design space, rather than for providing the optimal solution.

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

The concepts of timing and architecture of the circuit can be expressed within the language constraints of C++ by using ASC "hardware types", implemented as C++ classes, and operators for these. These operators map to the module generation environment called PAM-Blox II [18]. PAM-Blox II is also implemented as a C++ class library. PAM-Blox II is built on top of PamDC [3], a gate-level design library implemented also as a C++ class library. At the gate level, PamDC provides the engine for gate-level simulation and creates a netlist in the EDIF format.

ASC facilitates design space exploration in the following way. It provides three intermediate representations, all in C++ syntax, to go from a software implementation all the

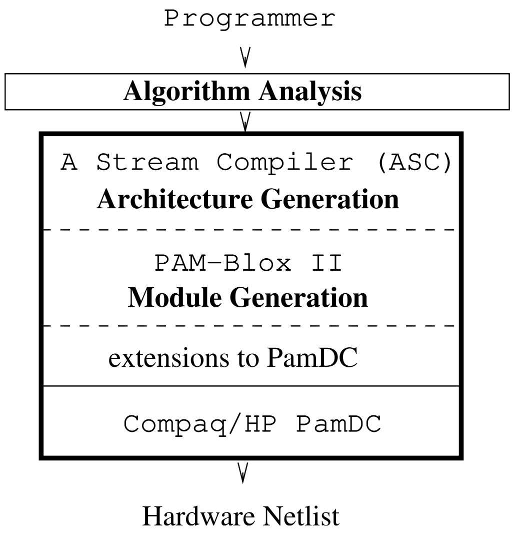

Figure 2: Levels of abstraction and structure of ASC. The bold box represents a single C++ program.

way down to the gate-level without the use of a single line of VHDL, Verilog, or IP libraries. Since each intermediate representation is human readable, it is possible to reason about optimizations at each of these levels and explore such optimizations within the ASC framework.

Conceptually, ASC follows the philosophy of the C programming language. The objective is to offer the potential for maximal performance, and at the same time provide a convenient language interface. The C language provides variable types which correspond directly to the actual number representations supported by the underlying hardware – the microprocessor. Examples for such C-style types are int, long, float, and double.

On the hardware side, implementations are not limited to any particular number representation or any particular bit width. Custom hardware allows 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 which allow the user to select specific number representations. Types and attributes provide a connection between the C++ description and the architecture generation layer. Figure 2 shows the levels of abstraction in ASC, described in more detail below.

- **Algorithm analysis layer.** Common tasks associated with this layer include: extracting compiler-controlled memory management [25][28], pointer analysis for hardware synthesis [21], loop transformations for hardware generation [7][10][26], precision analysis[4][6][24], data-structure transformations, and architecture selection. For the sake of this paper this step is handled manually, i.e. all algorithmic transformations are done by the programmer. ASC's task is to make this activity as easy as possible.

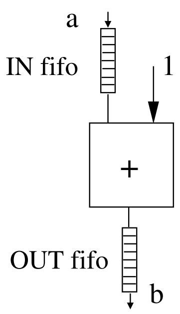

Figure 3: A basic Stream Architecture

- **Architecture generation layer.** This layer is the main focus of this paper. ASC code serves as the input to generating the accelerators architecture.

- **Module generation layer.** In contrast to most other 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 together with optimization on all the other levels. More details on PAM-Blox II can be found in a previous paper [18].

- **Gate Level to Netlist layer.** ASC does not utilize any VHDL or Verilog and instead uses PamDC [3] for gate level support, simulation, and EDIF generation.

In this paper we focus on a key step of design space exploration: converting an architectural description, in our case ASC code, to the gate level. 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, called 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 either throughput, latency or area. Since each of these three optimization modes can be selected separately for each expression in the ASC code, the user has the capability to optimize towards any objectives such as fiting into a specific area with minimal latency, or maximal throughput.

A constructive way of visualizing 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 C code for vector increment, ASC code, and the resulting stream architecture implementation of that code:

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), output stream (b), and, by specifying the expression whose operator defines the function (add) to compute the output stream, 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;
```

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 `STREAM_OPTIMIZE` for each expression. Currently supported optimization values are `THROUGHPUT`, `LATENCY`, and `AREA`. Finally, a modified C program streams data through the hardware at runtime to compute the results. The modification consists of replacing the "for" loop above by a call to the ASC runtime library, for the example above:

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

This call sends `SIZE` data items from `buffer a` to the generated circuits, either in a gate level simulation mode or real hardware, and receives `SIZE` data items into `buffer b` back. On the accelerator, the input data enters a first-in first-out (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 figure 3.

In general, an ASC architecture consists of a multi-input, multi-output data flow graph. Each "wave" of input values flows through this implementation of the dataflow 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.

3. The ASC Datapath

This section describes the facilities that ASC provides to support exploration of datapath and memory system designs.

3.1. Hardware Types and Attributes

ASC uses types and attributes to hook the programmers' description of the algorithm to the architectural features of the datapath of the stream architecture.

As mentioned before, ASC provides hardware types and attributes which the programmer uses to specify number representations. 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 are: `HWint`, `HWfix`, and `HWfloat`.

3.2. ASC "Instructions": Module Generator Libraries

PAM-Blox II [18] consists of more than 170 integer arithmetic module generators for elementary operations in about 10K lines of C++ code, resulting in an average of less 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 trade-off of precision versus area by enabling the user to choose custom rounding and normalizing schemes.

3.3. The ASC Memory Systems

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. The elements for this memory system are: flip-flops and registers, FIFO buffers, small, multi-ported, on-chip SRAM blocks, large on-chip SRAM blocks, off-chip SRAM memory, and off-chip DRAM memory.

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, or INTMEM or EXTMEM for FPGA internal blockram memories and FPGA external memories. For multiple external memories ASC provides attributes EXTMEM0, EXTMEM1, etc.

4. Benchmarks

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

4.1. Wavelet Compression

The first benchmark we evaluate is Wavelet Compression 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 shows 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++ standard template library.

```
DefaultSign = TWOSCOMPLEMENT;
DefaultSize = 20;
DefaultFract = 14;

HWfix in1(IN),in2(IN),
      out1(OUT),out2(OUT);
HWfix low,high,temp,temp2,coefficient;

HWvector<HWfix> v_temp1(4, new HWfix(TMP));
HWvector<HWfix> v_temp2(5, new HWfix(TMP));
HWvector<HWfix> lcoeff1(4, new HWfix(TMP));
HWvector<HWfix> lcoeff2(5, new HWfix(TMP));
HWvector<HWfix> hcoeff1(4, new HWfix(TMP));
HWvector<HWfix> hcoeff2(5, new HWfix(TMP));
```

The algorithm consists of two loops, one after the other. 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 following piece of 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);
#else
  int idx1=0;
  UNROLL_LOOP(int i=0;i<size1_2;i++){
#endif

      temp2 = v_temp1[idx1<<1];

      coefficient = IF(idx1,
                       lcoeff1[3],
                       lcoeff1[1]);
      low = low+(coefficient*temp2);

      coefficient = IF(idx1,
                       hcoeff1[3],
                       hcoeff1[1]);

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

      coefficient = IF(idx1,
                       lcoeff1[2],
                       lcoeff1[0]);
      low = low+(coefficient*temp2);

      coefficient = IF(idx1,
                       hcoeff1[2],
                       hcoeff1[0]);

      high = high+(coefficient*temp2);

      idx1++;
#ifndef UNROLL1
  LOOP_END();
#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.

In case of a dynamic loop in hardware, ASC generates array indexing hardware, which is basically a multiplexor-tree. This tree can be implemented in Xilinx FPGAs either by using tristate buffers and a bus, or by a tree of lookup tables. In general, the tree of lookup tables is faster, but requires precious FPGA resources. The ASC user can control tristate usage by setting `FASTINDEX` to true or false within the program. In addition, ASC knows from experience that designs with too many tristate buffers have a problem routing with current tools, so it will limit the number of tristate buffers to half the number of tristate buffers that are available on the particular FPGA.

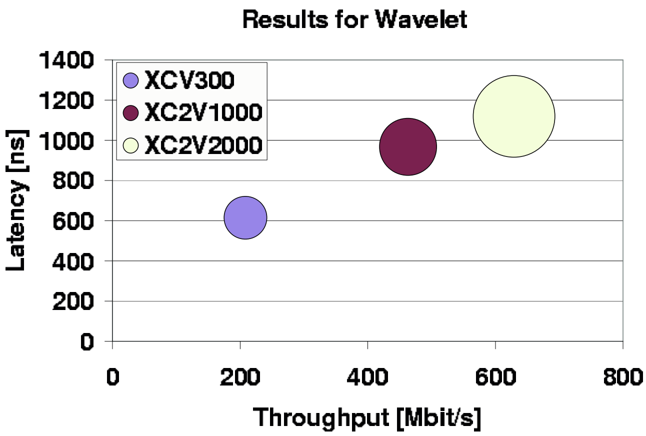

Figure 4: Results for the Wavelet design space exploration showing the best performers for each of the three FPGA sizes. The size of the circle indicates the area of the design.

4.2. 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. Two bits of one S-box are defined as:

$$\begin{aligned} y0 &= x1x3 \oplus x4 \oplus x0x1x4 \oplus x5 \oplus \\ &\quad x2x5 \oplus x3x4x5 \oplus x6 \oplus x0x6 \oplus x1x6 \oplus \\ &\quad x3x6 \oplus x2x4x6 \oplus x1x5x6 \oplus x4x5x6 \end{aligned}$$

$$\begin{aligned} y1 &= x0x1 \oplus x0x4 \oplus x2x4 \oplus x5 \oplus x1x2x5 \oplus \\ &\quad x0x3x5 \oplus x6 \oplus x0x2x6 \oplus x3x6 \oplus x4x5x6 \oplus 1 \end{aligned}$$

ASC allows the user to exploit bit-level parallelism 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 (sub-class) and creating a function call that access that particular new module from the ASC code level, as shown in the code below.

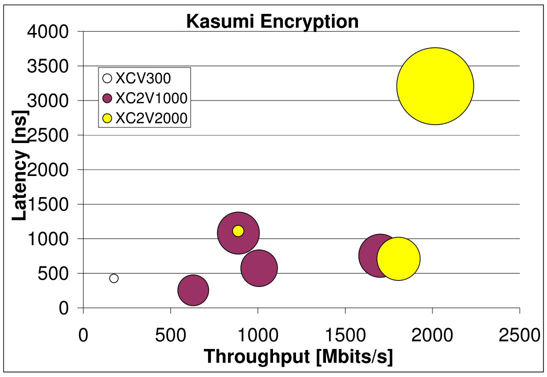

Figure 5: Kasumi design space exploration with ASC.

```
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();
#else
  }
#endif

  data[0] = l;
  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, when porting the code the user can decide to use available block RAMs to save area or create the custom logic to achieve maximal performance.

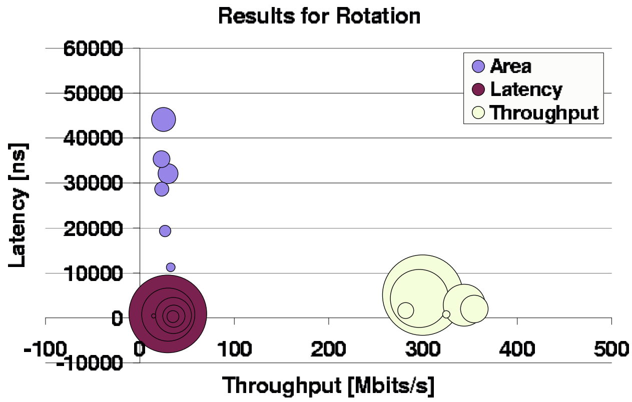

Figure 6: Rotation example, exploration of design space.

4.3. Rotation and Elementary Functions

The third application is that of elementary functions, such as sine and cosine, and their use in a coordinate rotation unit. We use polynomial approximations to generate sine and cosines. The coordinate rotation performs a pair of 2D rotations through input angles written to memory-mapped registers. The coordinates are then streamed in, and the rotated coordinates streamed out.

Use of `Default` variables and `STREAM_OPTIMIZE` 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 to test.

The code below is the rotation function, demonstrating how the `Default` and `STREAM_OPTIMIZE` variables can be used to explore the design space for the FPGA:

```
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);

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*cosP-y*sinP+z*cosD*cosP;
STREAM_END;
```

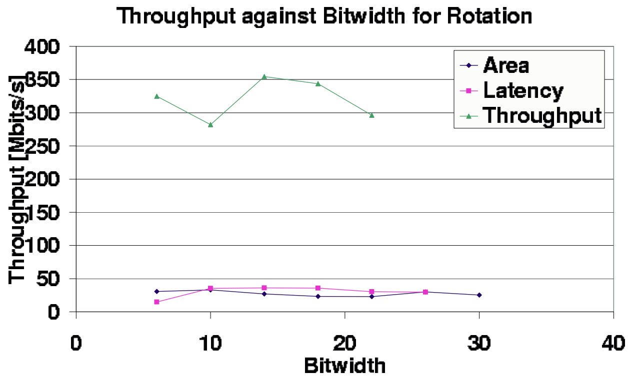

Figure 7: Bitwidth / Throughput tradeoff.

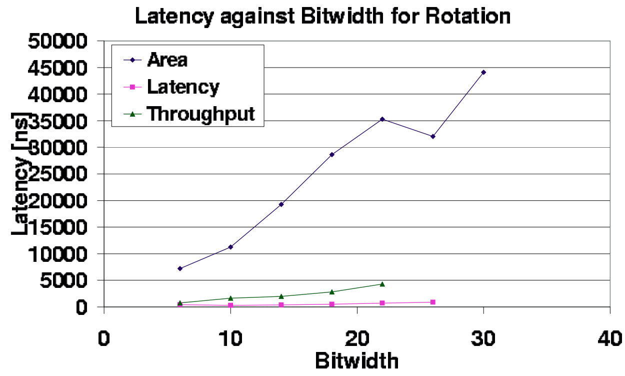

Figure 8: Bitwidth / Latency tradeoff.

5. Results

Results are obtained with ASC v0.4, gcc v2.95, and current Xilinx tools. We simulate the implementations on the gate-level by compiling ASC code with gcc and running the program in simulation mode. 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 bubble chart in figure 4 shows the design space for the wavelet compression example. We explore latency, throughput, and 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 one unrolls once and the large implementation is fully unrolled for maximal throughput. Figure 5 shows the results of design space exploration for Kasumi encryption using ASC.

The third set of results shows the design space for the rotation example. Just as for the previous two examples, a bubble chart in figure 6 shows the design space. In addition, figure 8 and figure 7 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.

6. Related Work

The key benefit of 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 non-critical parts of the applications on a very high level.

The DEFACTO system [23] supports 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 behavioural 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 [10] and MARGE [11] 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, rather than including design parameters into the language such as the type system in ASC.

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.

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.

7. Conclusions

ASC enables the exploration of area, latency, and throughput trade-offs for hardware design, and accelerator generation especially. Moreover, ASC is a platform for tools that automate the exploration of the space-time design space. In particular, since the entire source code is available, ASC enables research of the space-time design space at the architecture level, the functional unit level, and the bit level.

Our experience shows that ASC simplifies hardware design: in fact most of the ASC application code presented in this paper is developed by C++ programmers after reading the ASC manual. Current and future research on ASC includes automating some of the tasks in the algorithm analysis layer (Figure 2), and evaluating our approach using further benchmarks.

8. Acknowledgements

We thank Paul Kelly, Miron Abramovici, Cliff Young, Martin Morf, and Michael J. Flynn, for discussions and support of ASC efforts. Jian Liang, Gary Huang, and Henry Styles helped to advance ASC and wrote some of the code examples. The support of the UK Engineering and Physical Sciences Research Council (Contract GR/R 55931) and Xilinx Inc. is gratefully acknowledged.

9. References

[1] S.G. Abraham, B.R. Rau, *Efficient design space exploration in PICO*, Proc. CASES, International Conference on Compilers, Architecture and Synthesis for Embedded Systems, San Jose, California, Nov. 2000.

[2] S. Aditya, M. S. Schlansker, *ShiftQ: A bufferred interconnect for custom loop accelerators*, Proc. CASES, International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Atlanta, Georgia, Nov. 2001.

[3] P. Bertin, D. Roncin, J. Vuillemin, *Programmable Active Memories: A Performance Assessment*, ACM FPGA, February 1992.

[4] K. Bondalapati, V.K. Prasanna, *Dynamic Precision Management for Loop Computations on Reconfigurable Architectures*, In IEEE Symposium on FPGAs for Custom Computing Machines, April 1999.

[5] L. Bossuet, G. Gogniat, J.-L. Philippe, *Fast Design Space Exploration Method for Reconfigurable Architectures*, Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms, 2003.

[6] M. Budiu, S. C. Goldstein, K. Walker, M. Sakr, *BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations*, Europar Conf., Munich, Germany, Aug. 2000.

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

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

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

[10] J. Frigo, M. Gokhale, D. Lavenier, *Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective.*, IEEE FPGA Conference, Monterey, CA, Feb. 2001.

[11] M. Gokhale, J. Kaba, A. Marks, J. Kim, *Malleable architecture generator for FPGA computing*, Reconfigurable Logic, Proc. SPIE 2914, Bellingham, WA, Oct. 1996.

[12] S. D. Haynes, P.Y.K. Cheung, W. Luk, J. Stone, *Video Image Processing with the SONIC Architecture*, IEEE Computer, April 2000, pp 50 - 57.

[13] Kasumi Encryption Algorithm, *3G Wireless standard*, http://www.3gpp.org/

[14] P. H. W. Leong, M. P. Leong, O. Y. H. Cheung, T. Tung, C. M. Kwok, M. Y. Wong, K. H. Lee, *Pilchard - A Reconfigurable Computing Platform with Memory Slot Interface,* Proc. IEEE Symp. on FPGAs for Custom Computing Machines, Apr. 2001.

[15] Y. Li, et. al., *Hardware-software co-design of embedded reconfigurable architectures*, Design Automation Conference, 2000.

[16] J. Liang, R. Tessier, O. Mencer, *Floating Point Unit Generation and Evaluation for FPGAs* Proc. IEEE Symp. on FPGAs for Custom Computing Machines, Apr. 2003.

[17] M. Macedonia, *The Computer Graphics War Heats Up,* IEEE Computer Magazine, October 2002.

[18] O. Mencer, *PAM-Blox II: Design and Evaluation of C++ Module Generation for Computing with FPGAs,* Proc. IEEE Symp. on FPGAs for Custom Computing Machines, Apr. 2002.

[19] O. Mencer, W. Luk *Tutorial: Computing with FPGAs*, International Symposium on Computer Architecture (ISCA), Anchorage, May 2002.

[20] O. Mencer, M. Platzner, M. Morf, M. Flynn, *Object-oriented Domain-Specific Compilers for Programming FPGAs,* IEEE Transactions on VLSI, special issue on Reconfigurable Computing, Feb. 2001.

[21] L. Semeria and G. De Micheli, *Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C*, IEEE Transactions on Computer-Aided Design, February 2001.

[22] B. So, P. Diniz, M. Hall, *Using Estimates from Behavioral Synthesis Tools in Compiler-Directed Design Space Exploration*, Proc. ACM/IEEE 40th Design Automation Conference, June 2003.

[23] B. So, M. Hall, P. Diniz, *A Compiler Approach to Fast Design Space Exploration in FPGA-based Systems,* Proc. ACM Conference on Programming Language Design and Implementation (PLDI'2002), ACM Press, June 2002.

[24] M. Stephenson, J. Babb, S. Amarasinghe, *Bitwidth Analysis with Application to Silicon Compilation*, Proc. of the ACM Conf. on Programming Language Design and Implementation, Vancouver, BC, June 2000.

[25] O.S. Unsal, I. Koren, C. M. Krishna, C. A. Moritz, *Cool-Cache for Hot Multimedia*, MICRO-34 Conference, Austin, Texas, Dec, 2001.

[26] M. Weinhardt, W. Luk, *Pipeline Vectorisation*, IEEE Transactions on Computer-Aided Design, February 2001.

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

[28] L. Zhang, et. al., *The Impulse Memory Controller*, IEEE Trans. on Computers, Nov. 2001.