Architectural Evaluation of a Programmable Accelerator for Baseband, Phy and Video Applications using High Level Synthesis

Andy Fox, Tigran Sargsyan
RUSHC
San Jose, CA, USA

{andy, tigran}@RUSHC.com

Steven Anderson
Forte Design Systems
San Jose, CA, USA

sanderson@forteds.com

***Abstract*—Realizing software algorithms with hardware cores is a proven technique [1],[2] for power reduction. Evolving standards coupled with the enormous cost of chip design demand that the accelerators be programmable. In this paper we describe the development of a new low power programmable accelerator for Baseband, Phy and Video applications which fits into an LLVM compiler tool chain.**

The candidate architectures were expressed in SystemC. We chose SystemC for our source code because it provided a quick way to express design intent with a much simpler model than an equivalent RTL model. And there are several commercial HLS tools that could generate the RTL for us. Our SystemC source code also integrated well with our software tools and was simulate-able.

In order to measure the effectiveness of a candidate architecture we first compiled an application to assembler using a configurable LLVM compiler tool chain. The compiled program was then run on the SystemC model of the accelerator for functional verification. We then used a HLS tool to create a RTL implementation and ran the same compiled program on the RTL. The RTL simulations gave us accurate estimates of area, performance and power with existing RTL design tools.

Through a series of many benchmarks we discovered that typically 2-4 instructions would be executed in parallel per basic block in the C code. Architectural performance was measured by running common DSP and baseband (Viterbi/Turbo) algorithms. A small sample of the results that were generated for a 28 nm library is shown below:

(a) 16 Giga Sample per second 128 tap FIR required 512 8-tap FIR blocks and consumed 4 Watts.
(b) Turbo (LTE) "C" code after compilation required 217390 instructions. It mapped to 18 accelerators and consumed 152mW.
(c) Viterbi (k==8) required 6 accelerators each running at 500 MHz and consumed 24mW.
(d) 8x8 Discrete Cosine Transform (DCT) mapped to 8 accelerators and consumed 28.12mW

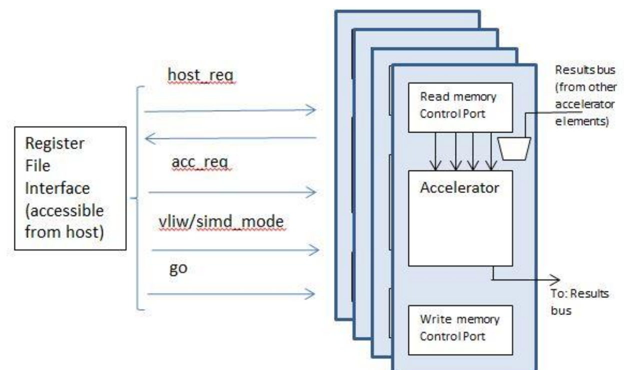
A programmable accelerator is expected to be less efficient than a "hardcoded" RTL implementation for any particular software application. As part of this project we explored the overhead of the programmable accelerator relative to an application specific RTL implementation and we offer comments about this in our results section.

I. INTRODUCTION

Our architectural evaluation methodology comprised two parts: A hardware evaluation flow for exploring the cost of the various instruction/memory configurations, and a flexible compiler software flow for mapping various applications to the candidate architectures.

We started with the base hardware architecture shown in Figure 1.

Figure 1: Base Architecture



This comprises a host CPU and an accelerator unit. Each accelerator has two memory controller interfaces (a read and write port controller) and is controlled by a host. The accelerator comprises a simplified processing element with 1k words of instruction memory, 1k words data memory, 8 registers, and a 40 bit accumulator with a DSP oriented instruction set. The result from the execution stage of the accelerator processor can be fed to the other accelerators via a "results bus". The host controls the accelerator via a register file. The accelerators can be arranged in Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) under the control of the host. In SIMD mode each accelerator operates on the same instruction, and data is streamed from the memory as a block. In MIMD mode the

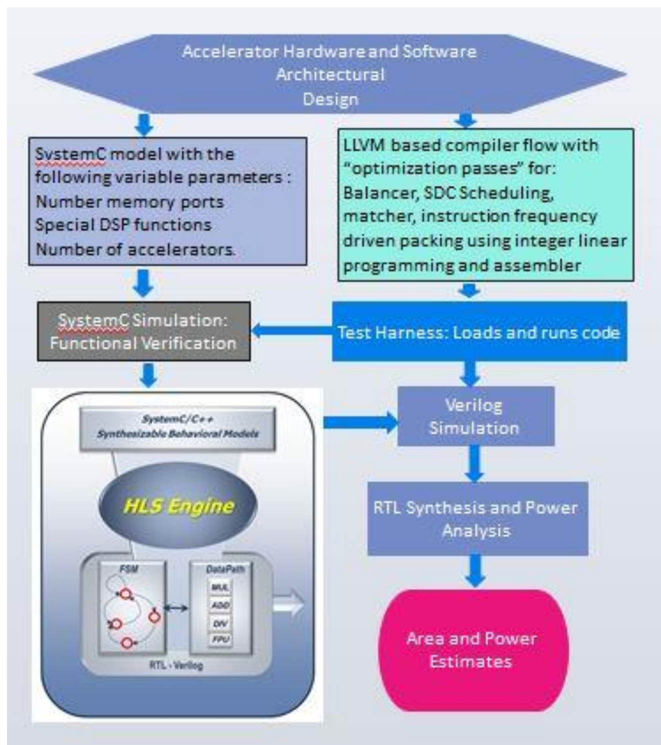
accelerators operate independently and arbitrary memory reads/writes permitted. In our target fabric many host/accelerator pairs are replicated and a memory architecture, not outlined in this paper, is used to allow efficient memory to memory transfers.

To develop an optimal hardware accelerator we experimented with the following architectural options:

(a) Varying the number of accelerators to match the amount of parallelism we could extract from the algorithms
(b) Varying the memory interface
(c) Varying the instruction set to include advanced autonomous DSP instructions.

The candidate architectures were expressed as a SystemC model. The flexibility of the SystemC model with generated RTL gave us a much faster turn-around vs. hand coded RTL. The behavioral SystemC simulations were used for design verification prior to synthesis into RTL. We were also able to execute the same compiled program on the generated RTL Verilog models as shown in Figure 2.

Figure 2: The Evaluation



The evaluation process is shown in Figure 2. The first step is to design an instruction set to suit individual applications. The application is then compiled into assembly code which was run against our SystemC model by the test harness. We can then create a Verilog RTL model of the hardware with the HLS tool and run the same asm code against that Verilog model. This gives us very accurate switching characteristics for doing power analysis. We were also able to synthesize that Verilog into a gate level implementation using an existing RTL synthesis tool. After running all of the tools we get a very accurate estimate of power and area for our chosen application.

II. Software Flow Details

The software flow comprised the following components:

(a) Library, expressed in XML, was devised which enumerates the number of instructions each accelerator can execute, the size of the memories available, and the rates at which the accelerators can operate
(b) Front end which restructures the control data flow graph produced by LLVM using: balancing, and word sizing for minimum bit count.
(c) Scheduler which exposes instruction level parallelism within a basic block of instructions.
(d) Matcher which maps groups of instructions onto a library of predefined "macro" instructions expressed in the library.
(e) Resource driven packer which assigns instructions to the accelerators based on measured instruction execution using sample data from applications.

The balancing and word sizing operations use the approach taken in [6]. The goal of balancing is to make the circuit graph more amenable to exposing parallelism, so long series of tree nodes are replaced by height balanced more parallel operations. The idea of word sizing is to find the smallest machine word representation for each operation.

Each accelerator has a number of communicating functional units which may be executing in parallel. A key architectural exploration experiment is the number of functional units each accelerator should have for the target applications. The scheduler component of the software flow is devised to help determine this count. The scheduler is based on the difference constraints approach outlined in [3]. The goal of the scheduler was set to minimize the latency (cycle count) for any basic block while maximizing the number of functional units executed in parallel.

Sequences of LLVM instructions were matched onto custom instructions for the accelerator: each target custom instruction was expressed as an equation which was matched against an equation extracted from a cut in the graph of instructions taken from LLVM. The approach is "functional" because rather than syntactically match the instructions we use an equation which denotes the behavior. So, for example, the candidate x*2 could match with x<<1 or x+x. This "functional" matching is accomplished by using a solver designed for matching arithmetic expressions – we used STP [4]. For each candidate matching we generated the clauses required by the solver and expressed the match as a miter. Using this approach the relative gains of various customized instructions (including DSP type operations such as Sum of Differences and various FIR combinations) were measured.

The target fabric comprises multiple accelerators which can run at programmable frequencies (ranging from 1GHz to 200 MHz). The assignment of the scheduled basic blocks to the fabric is done using a "packing" stage. We formulate this as an integer linear programming problem. We measure the frequency of instruction usage by executing the application (using the LLVM utility lli) with sample data and observe the invocation count and instruction counts per basic block. The objective function of ILP is to minimize the total number of accelerator units needed while sustaining overall system performance.

III. Hardware Experiments

We started from the base architecture, comprising 2 accelerators. The results for this architecture are shown below in Table 1.

Table 1: Hardware Experiments

Baseline	Area (mm*2)	Power (mW)
2 accelerators, results bus, single port memory interface	0.51	35
Experiment		
4 accelerator elements	1.1	71
Support for the accumulator bus + DSP FIREVER, FIRCONT, SAD, SOD, instructions, sum of differences, 2 accelerator elements	0.52	35
Additional data memory port for FIR instructions	0.54	36
Additional wait state logic for 2-port accelerator memory	0.52	35

Note that these results are for a 45nm process with 500 MHz target frequency.

We then conducted the following experiments:

(a) Varying the number of accelerator elements. The base architecture has just two accelerator elements, we experimented with 4. This required increasing the number of memory ports and replicating the accelerators.

(b) We added specialized DSP FIR instructions for building filters, including FIRIN and FIREVERIN instructions. In these the accumulator of one accelerator is passed to the next allowing a daisy chain construction of FIR components. These instructions are also autonomous in the sense that they pull data out of local instruction and data memory to process buffers of data.

(c) We added specialized instructions for sum of differences (SOD), sum of sums (SOS) and sum of absolute differences (SAD), saturation. These used register pair encoding to keep the instruction set architecture binary representation compact. We experimented with a "double register" addressing mode (2 bits used to represent 4 register pairs).

(d) We experimented with adding an extra data memory port to allow for coefficients to be pulled from memory for the specialized FIR instructions.

(e) Adjusting the memory interface. The memory interface to the accelerator has a flexible number of ports, we experimented with four versus two. In SIMD mode the data is streamed from the memory in 4 word chunks so reducing the memory bandwidth would be expected to halve the throughput. This experiment required us updating the controller to the memory by adding a wait state.

IV. Software Experiments

In order to measure the effectiveness of candidate architectures we first compiled an application to assembler with our LLVM compiler tool chain. Through a series of many benchmarks we discovered that typically 2-4 instructions would be executed in parallel per basic block in the C code. Table 2 shows sample data (we ran our scheduler across numerous C programs for video, base band and Phy applications). Architectural performance was measured by running common DSP and baseband (Viterbi/Turbo) algorithms.

Algorithm	Instruction Count	Latency	Average Number of Instructions in Parallel	Accelerators needed	Memory accesses
FIR-10 (10 tap fir)	116	7	3	6	501
Turbo decoder (k=64, 2 iterations, 8 states)	217390	14	2	18	134144
FFT	1077	15	2.5	8	286
DCT (8x8)	14754	22	2	8	4496

Table 2: Software Experiments

From Table 2 we observe approximately 2.5 instructions per cycle that can be easily extracted to run in parallel. This is in line with experiences reported elsewhere and informed our decision to associate two accelerators with each host. By using predicate bits [7] to collapse basic blocks we would expect further parallelization to be possible but did not explore this. We report the number of memory accesses: this is the total number of word level accesses for the sample data applied.

Our experiments assumed 1k 16-bit word instruction memory and 1k 16-bit word data memory per accelerator. The accelerator memories can be filled under the control of the host but we do not detail that in this paper. The various algorithms memory requirements are summarized below in Table 3.

Table 3: Memory Requirements

Design	Memory Requirements	Discussion
Turbo	2336 words	Forward trellis: 16 words Backward trellis: 16 words Alpha, beta, Gamma: 520,520,16 words Alpha2,beta2,Gamma2: 520,520,16 words Interleaver: 64 words
FFT (1024 points)	6144 words	Coeff: 5120 words Points: 1024 words

We experimented with various memory port combinations but found the results converged with a single memory port per operand. Table 4 shows the effect in the reduction in latency with the addition of two more memory ports per operand for a CIC. The scheduler was able to slightly reduce the latency through the critical code block with the additional port but we found on the average the gain was not justified.

Table 4: Results for Latency Reduction by Memory Port Addition

Basic Block	Accelerator frequency	Instruction count	Invocation Count	Latency 1,2 Ports
BB1	300	4	8	3,3
BB2	200	12	8	8,8
BB3	100	111	64	32,30
BB4	300	10	8	4,4
BB5	200	42	8	22,22
BB6	100	111	64	33,31

V. Architectural Tradeoffs

Based on these experiments we were able to make the following tradeoffs. We observe that the cost of adding the accumulator bus and FIREVER, FIRCONT, SAD, SOD DSP instructions is very small so we elect to include them. Likewise we notice the gain in adding a wait state to the read memory interface to reduce the memory bandwidth had a negligible gain, so we discard that decision. As expected increasing the number of accelerator elements linearly increased area and power.

This architecture was arrived at based on the following conclusions from our software and hardware experiments:

(a) On average we found between 2 and 3 instructions which could be parallelized.

(b) The cost of the accumulator bus was very low and allowed us to pass unsaturated FIR results and construct FIR filters readily.

(c) The cost of 4 memory ports per accelerator was low and allowed us to sustain the 500 MHz target frequency. Our experiments revealed we could easily add an additional wait state in the accelerator memory interface but the performance degradation was not justified by any significant area savings. This experiment merely revealed the relative ease of adjusting the memory port arrangements within the HLS environment.

We found that we could save data memory port accesses using our autonomous DSP FIR instructions. In these the program counter is suspended while the FIR instruction is being computed allowing the instruction memory to serve as a source of coefficients thereby allowing two memory accesses (data and coefficient) per cycle. Moreover the propagation of the accumulated results from the adjacent accelerator allows in 1 cycle a full multiply accumulate. The semantics of the Added DSP FIR instructions are:

```
FIRIN:
for (temp=0; temp < immediate ;) {
        OUT = A;
        //IN is the accumulator input from the
        //adjacent accelerator.
        if (temp ==0) { A = IN + imem[IMAR +
        temp] * dmem[DMAR + temp];
        REQ = 1} //Note how the instruction
        //memory is read to get the coefficients.
        else {A += imem[IMAR + temp] *
        dmem[DMAR + temp]}
        temp = temp + 1;
        PC = PC;  //Note that the program counter
        //never changes
}
```

```
FIREVERIN:
for (temp=0; temp < immediate ;) {
        OUT = A;
        if (temp == 0) { A = IN + imem[IMAR +
        temp] * dmem[DMAR + temp]; REQ = 1}
        else { A += imem[IMAR + temp] *
        dmem[DMAR + temp]}
        temp = temp + 1;
        if (temp == immediate) temp = 0;
        PC = PC;  // Note that the program counter
        //never changes
}
```

These FIR instructions proved efficient because:

(a) Unsaturated FIR results were propagated through the accumulator bus (in this case the accumulator from the adjacent accelerator is via the IN port).

(b) During the DSP FIR loop the program counter is suspended and the instruction memory is used as the source of the coefficients. This saved an additional read on the data memory and so obviated the need for an additional data memory port. Our experiments indicated that this port (which would be needed for every accelerator) was unjustified.

(c) The memory fetches are embedded efficiently into the instruction. For example the two memory fetches from imem and dmem (coefficient and data) and the increment of the addresses are embedded in the instruction execution. This improves code density by at least 4 instruction (loads, stores, memory increment).

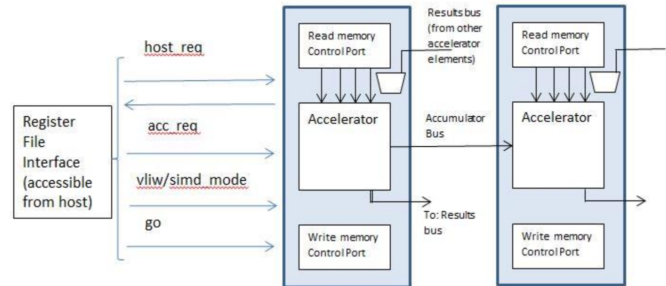
(d) In one of our target applications (filtering of base station antenna data) we had a need to set up the accelerators to do channel filtering and Digital Up Converter Nyquist rate filtering. In this application the accelerators were configured and then never changed (except by interruption from the host). The autonomous FIREVER instructions proved useful for this mode.

We were surprised at the low overhead of introducing the complex DSP operations. We had expected the addition of these instructions to require additional hardware resources in the resultant RTL. However closer inspection revealed that the high level synthesis tool was successfully finding a schedule to minimize the use of the hardware functional units.

VI. Final Architecture

The candidate architecture that we chose is shown in Figure 3.

Figure 3: Final Architecture



In the diagram each accelerator comprises an array of processing elements (PE's) each one of which is in essence a mini-DSP. Each of these processing elements has a memory interface and is also mapped into the address space of a host.

Accelerator features:

(a) SIMD/MIMD modes

(b) Each accelerator is a mini DSP with local Instruction Memory and Data Memory.

(c) Support for autonomous FIR instructions (FIREVER,FIRIN) including a memory interface which pumps data through the FIR instructions.

(d) Propagation of unsaturated accumulator result through dedicated "accumulator bus".

(e) Simplified host interface (The host can read/write to any of the registers for each accelerator).

The preliminary results for our final architecture are shown in Table 5.

Table 5: Preliminary Results

Design	Power (mw)	#Accelerators	Area (mm*2)
16GSPS 128 tap	4000	208	126
Turbo	152	18	5.2
Viterbi	24	6	1.48
DCT 8x8	28.12	8	1.94
Smith Waterman	24	6	1.58

Notes:

(a) The results are scaled for a 28 nm process. Each design uses the same target library.

(b) The 16 Giga Sample per second 128 tap FIR required 512 8-tap FIR blocks. Each 8 tap FIR mapped onto 1 accelerator (2 PE's).

(c) The Turbo (LTE) required 18 accelerators and 6 memories. The turbo "C" code after compilation required 217390 instructions.

(d) The Viterbi (k==8) required 6 accelerators each running at 500 MHz.

(e) The DCT result is for an 8x8 DCT.

VII. Lessons Learned

In this section we offer some lessons learned during the course of the experiments about SystemC modeling, simulation and RTL generation from SystemC.

1) ***Programmable vs. Hard Coded accelerators:*** We were interested in gauging the cost of programmability versus a hardcoded implementation. To achieve this we ran the "C" algorithm for the Turbo decoder directly through HLS. This hardcoded RTL implementation resulted in a massive reduction in power (7.6 mW down from 152 mW) and area (0.21mm**2 versus 5.2mm**2) versus the programmable accelerator. We note that the programmable solution required the instantiation of more than 20 1k word memories roughly 4mm*2) which dramatically skew's the result in favor of the hardcoded RTL. We conservatively estimate the cost of the programmability in this case to be a factor of 5x.

We observe that our approach to programmability involved using a lightweight processor (complete with instruction decode, ALUs, memory interface) for application in multiple domains with consequent overhead. Another approach would be to target the application domain and offer only programmability needed for that domain. A practical example is DSP filtering. A target programmable accelerator could maintain a filtering tap structure (eg support the construction of various multi-rate filters) and provide programmability for the coefficients only (eg as host programmable registers). We expect that by using such application specific programmable accelerators the cost of programmability could be substantially reduced. The FIREVER class of DSP instructions that we experimented with would be good candidates for this "target" domain acceleration (recall the FIREVER instruction is set up and then never reconfigured except if the host resets the accelerator). A practical approach to fine grained programmability control from SystemC would be the user nomination of variables to be placed in registers programmable from the host.

2) ***SystemC Modeling:*** Coding style as it relates to several factors was very important to us for this project. First was the question of coding efficiency. Could we write code in a natural "C" coding style and still get good synthesis results? And how did the coding style work for control vs. data path portions of the design?

To illustrate the coding style for control we can use a portion of the Instruction dispatch model:

```
switch(opcode){
  case DMAA:{
    HandleDmaa(earg0, earg1, earg2, earg3);
    break;
  }
  case ADD1:{
    HandleADD1(earg0, earg1, earg2, earg3);
    break
  }
  case ...
}
```

For our programmable accelerator it was easy to code the instruction dispatch model as a simple switch statement depending on the instruction op-code. And each case then represented one of the op-codes for our accelerator.

This coding style is natural and extensible. If we decided to add an opcode we could simply add another case to the switch. In fact that is exactly what we did when adding the FIRIN and FIREVERIN instructions. And the HLS tool handled all of the details of the modified state machine for us.

This is a good example to show the simplicity for coding control in a SystemC model. But what about synthesis efficiency with this coding style? To illustrate synthesis efficiency consider the source code for the HandleDmaa (Double multiply and add to accumulator) function:

```
sc_uint<ACCUM_WIDTH>  alu::HandleDmaa(
sc_uint<DATA_WIDTH> arg0,
sc_uint<DATA_WIDTH> arg1,
sc_uint<DATA_WIDTH> arg2,
sc_uint<DATA_WIDTH> arg3)
{
   sc_uint<DATA_WIDTH> rs3 = rf[arg0]; //read
   //from register file at address arg0
   sc_uint<DATA_WIDTH> rs2 = rf[arg1];
   sc_uint<DATA_WIDTH> rs1 = rf[arg2];
   sc_uint<DATA_WIDTH> rs0 = rf[arg3];
   sc_uint<ACCUM_WIDTH> val = rs0 * rs1 + rs2
      * rs3;
   return(val);
}
```

In this you see that it is implementing a multiply accumulate. There are 2 multipliers plus one addition operator being used to calculate the result. In our accelerator there were several other op-codes that also used multipliers and adders. And it turned out that the HLS tool was easily able to determine that those multiply and add hardware elements were in mutually exclusive control branches and therefore were share-able. And it produced an RTL implementation that did the sharing with whatever control and muxing that was required for correct operation. We were happy to find that it did this sharing without any additional hints from us in the form of a tool directive or some other user intervention.

For an architectural exploration project it is critical to have source code that is easily configurable. The C++ language has many constructs like templates that are very useful in this regard. We also found it very useful to declare hardware modules as pointers and then use for loops to construct the netlist.

For instance in the declarative region we can declare a pointer to our module:

```
module_type * module_name[ACCEL_NUM];
```

Then in the constructor we can instantiate the models and hook up the netlist with a for loop:

```
for(int num = 0; num < ACCEL_NUM; num++){
        module_name[num] = new
            module_type(const char* name =
            sc_gen_unique_name("this_module"));
        module_name[num]->clk(sys_clk);
        module_name[num]->rst(reset);
        etc.
}
```

As long as ACCEL_NUM is static at compile time the HLS tool can create an RTL design for us from this code. This feature allowed us to test many different hardware options from a single set of source code.

Another language feature that simplifies architectural exploration is the handling of arrays. From a simple array dereference it is possible to infer any type of memory or register mapping for that array. For instance in our source code if we have:

```
int x = my_array[addr];
```

The HLS tool can automatically map my_array to a set of registers or single port memory. Or with the help of a directive it could also be mapped to a dual port memory. And all of these architectural trade-offs can be made with no change to the users source code.

Synthesizable SystemC can be written with methods or clocked threads. Since methods execute every clock cycle they are analogous to writing Verilog. Clocked threads on the other hand can be suspended for multiple clock cycles. And it is possible to aggregate more functionality into a clocked thread. So early on we made the choice to use clocked threads for our models.

*3) **Simulation Performance:*** The other important factor related to coding style was simulation speed. For a full-up simulation of the SOC we would have to simulate hundreds of processors and accelerators. So we needed to ensure that the coding style enabled sufficient simulation speed to make this feasible.

For faster simulation it is handy to aggregate a bunch of individual signals into a packet so that they can all be read or written with a single event in the simulator. This is easily done by using a C++ struct or class as the type for an I/O signal. Combining a bunch of individual signals into a single data type can significantly reduce switching activity, particularly for modules with high pin counts and lots of simultaneous switching.

For simulating a massive system we had several options available to us. There was the SystemC simulator, an internally developed simulator, or compiled code using Pthread's. We benchmarked the performance of each for 100 instances of a FIR program with each accelerator assigned a multiply accumulate running in a loop 10,000 times. The results of each FIR were communicated to the other PE's using a shared memory array in the Pthread program. The results in Table 6 show the relative performance comparison on a quad core machine running at 3GHz.

Table 6: Simulation Results

Simulation Model	Number of instructions executed	Total simulation time (second)	Instructions per second (million)
SystemC. Each accelerator modeled using SC_THREAD	26,215,200	22	1.2
Pthread's	26,215,200	0.03	873
Internal tool	26,251,200	84	0.32

We were initially disappointed at the SystemC performance; it was only 4 times faster than our internal simulator. However closer inspection revealed that the SystemC scheduler was not threaded so the gain from using SC_THREAD was null. We had hoped to see a thread become a parallel thread on the simulating machine as suggested by [8].

When using a Pthread model we saw a substantial simulation performance improvement. Note that this is a compiled simulation model (it truly executes the instructions as opposed to decoding them and mimicking the full instruction pipeline) so we expected to see a significant performance gain. This Pthread simulation is also not sufficient for design verification since it does not accurately model hardware details such as data types or signal interfaces or the stalling behavior of the pipelines. It is however useful as a software development platform.

*4) **Synthesis to RTL:*** The HLS tool that we used for this project was Cynthesizer from Forte Design Systems. In addition to synthesizing the RTL from our SystemC source it also provided integrations for running simulation and all of the RTL synthesis, simulation and power analysis tools. The primary benefit was to automate data transfer between all of the individual tools. It also provided some default scripts for running the other tools which gave us a starting point to customize our own scripts. This made it very easy to create a working design flow.

Cynthesizer also includes a complete IDE for SystemC code development including the SystemC simulator. We used this early in the project to write and debug much of our code. As the project progressed it was beneficial to do many of the same tasks from the equivalent Makefile based command line interface. During source code development we continually ran SystemC synthesis. So once our source code was complete we knew that we could quickly get the RTL that we needed to complete our analysis.

One of the design constraints for the ALU module was that it needed to be pipelined so that a new input could be accepted every clock cycle and a new output produced every clock cycle after some latency. This was further complicated by the fact that the number of cycles required for various instructions differed. For instance a 32 bit unsigned add might require just one cycle but a floating point add required 3 cycles. Fortunately there is no need to modify the behavioral code to make this pipelining possible. Cynthesizer has a simple

directive that did the pipelining for us, including the balancing of latencies.

Once we had the generated RTL for the complete system we were able to simulate it and extract the switching activity for power analysis. These simulations simply consisted of running the same assembled program on the Verilog RTL that we had already run on the SystemC model. And we were able to use the same test harness that had been used to run the program on our SystemC source code. The tool integrations provided by Cynthesizer made the task of switching between SystemC models and generated RTL completely transparent for simulation.

VIII. Conclusions

We observed that the SystemC simulation model was substantially slower than a Pthread equivalent (by a factor of more than 100). This was significant for us because we had hoped to use our SystemC source code as the basis for our software simulator. Unfortunately we could not achieve the goal of having a single model for both hardware and software development. But our result with Pthread's suggest a significant performance improvement should be possible for the SystemC simulator if it was multi-threaded as suggested by [8].

All of our benchmark designs originated from C descriptions. With small modification we were able to run one of them (turbo trellis decoding) through the high level synthesis tool and directly translate to RTL. This generated a substantially (5x) smaller and more power efficient design than the programmable alternative and gave some hint of the true cost of programmability.

We found several efficient DSP instructions that could be effectively exploited by the design flow. Our chosen HLS flow was able to extract parallelism and efficiently implement these DSP functions with minimum hardware. This made it very easy to modify the instruction set and test it against our set of applications. And it allowed us to experiment with complicated instruction control and operations (such as the autonomous FIREVER instruction expressed in terms of C) and let the tool do the work of extracting the hardware parallelism.

Using the combination of the Forte Design Systems Cynthesizer HLS tools and an LLVM compiler software tool chain we were able to quickly experiment with various accelerator and instruction primitives. Based on our experiments we concluded that an array of two accelerator combinations was adequate for the problems we explored: any further parallelism was not exploitable by the software flow.

Acknowledgment

We thank the anonymous referees for their constructive feedback and acknowledge the helpful discussions with Edvard Ghazaryan.

References

[1] G. Venkatesh, J. Sampson, N. Goulding, S Garcia, V Brysin, J Lugo Martinez, S Swanson, M Bedford Taylor, *Conservation Cores: Reducing the Energy of Mature Computations*, Proceedings of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems.

[2] H. Esmaeilzadeh, E. Blem, R Amant, K Sankaralingam, D Burger, *Dark Silicon and The End of Multicore Scaling*, Proc of the 38th International Symposium on Computer Architecture, 2011.

[3] J. Cong, Z. Zhang *An Efficient and Versatile Scheduling Algorithm based on SDC formulation.* Proceedings of the 43rd Annual Design Automation Conference.

[4] V Ganesh, D Dill, *A Decision Procedure for Bit-Vectors and Arrays.* Proc Computer Aided Verification 2007.

[5] LLVM: An Infrastructure for Multi-Stage Optimization, C Lattner, MS Thesis, CS Dept, University of Illinois at Urbana Champaign, Dec 2002.

[6] Nadav Rotem, Haifa University, Presentation titled "*High Level Synthesis Using LLVM*", http://llvm.org/devmtg/2010-11/Rotem-CToVerilog.pdf

[7] Scott Mahkle et al, *Effective Compiler Support For Predicated Execution Using the HyperBlock.* Micro 25, Proceedings of the 25th annual international symposium on Microarchitecture, pages 45-54, 1992.

[8] Aline Mello et al, "Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations", DATE Conference, 2010.

RUSHC

RUSHC is a consulting company with a 20 year history of specializing in both EDA software development and hardware projects for clients. RUSHC has strong algorithmic knowledge and experience in logic synthesis, formal verification and technology mapper CAD tool development. RUSHC has engineers located in its offices in Yerevan, Armenia as well as the San Francisco Bay Area. More information can be found at www.RUSHC.com.

Forte Design Systems

Forte Design Systems™ is the #1 provider of electronic system-level (ESL) synthesis software, confirmed by Gary Smith EDA, provider of market intelligence for the global Electronic Design Automation (EDA) market. Forte's software enables design at a higher level of abstraction and improves design results. Its innovative synthesis technologies and intellectual property offerings allow design teams creating complex electronic chips and systems to reduce their overall design and verification time. More than half of the top 20 worldwide semiconductor companies use Forte's products in production today for ASIC, SoC and FPGA design. Forte is headquartered in San Jose, Calif., with additional offices in England, Japan, Korea and the United States. For more information, visit www.ForteDS.com.