Deriving an Efficient, Application-Specific, FPGA-Based Pipelined Processor

Jonathan Phillips
Utah State University
j.d.phillips@aggiemail.usu.edu

Aravind Dasu
Utah State University
dasu@engineering.usu.edu

Abstract

Hardware accelerators outperform equivalent software implementations through the exploitation of both spatial and temporal parallelism. Determining the correct levels of each in order to maximize system performance is a challenging task. We present techniques for exploring the design space of a high-level pipelined architecture to balance pipeline stages and optimize performance. Consideration is given to both memory-intensive and compute-intensive stages. As an example, a custom FPGA-based architecture for a simulated annealing algorithm which runs as a time-critical in-space application is presented. Performance of the novel architecture is compared with that of traditional space-based microprocessors.

1. Introduction

Many compute-intensive, time-critical software applications can benefit from being ported to custom accelerator hardware, exploiting characteristics of the software algorithm to speed up execution through the use of both spatial and temporal parallelism. Determining appropriate levels of parallelism to extract in each stage of a high-level pipeline can be a daunting task, especially on an FPGA where resource constraints are fixed. Techniques are presented for the derivation of both memory-intensive and compute-intensive pipeline stages within an architecture targeted for a space-ready, radiation hardened Xilinx Virtex 4 FPGA (XQR4VSX55 [1]). As unmanned space vehicles become more autonomous, the onboard processors are forced to perform complex, time-critical applications such as event scheduling and route planning. Simulated annealing [2]is a widely-used technique for solving these optimization problems. For example, NASA uses the CASPER [3] and ASPEN [4] tools to design software code for iterative repair [5], a simulated annealing algorithm that derives complicated event schedules onboard spacecraft.

In section 2, a summary of recent advances in the design of application-specific processors is presented. In section 3, the design methodology is presented. In section 4, results comparing the performance of an example custom circuit for accelerating simulated annealing with that of traditional space-based processors are provided, followed by conclusions and plans for future work.

2. Background

Application-Specific Processors (ASPs) are a relatively new method of hardware architecture design. In [6] some of the benefits of ASPs over standard processors, including execution time and power consumption, are explained. The configurability of FPGAs, when combined with the strength of ASPs, offers huge benefits for application specific programmable accelerators. As FPGA technology improves in terms of clock speeds and power consumption, reconfigurable ASPs are likely to gain popularity as an alternative to traditional ASIC designs [7]. Design space exploration (DSE) is used to determine the characteristics of an ASP. In [8] the steps necessary to derive an efficient ASP using DSE are identified.

3. Methodology

This section describes a methodology for the derivation of efficient, FPGA-based high-level pipelined processing stages. Fig. 1 outlines the

```
generate minimal architecture for all stages
compute latency of all stages
Loop
     identify worst-performing stage
     reduce latency of worst stage by either
          a.   reducing latency enough to pass "worst
               stage" label to a different stage OR
          b.   reducing latency as much as possible
               while retaining "worst stage" label
EndLoop (when worst stage label cannot be passed)
```

Figure 1: Algorithm for generating a pipelined processor.

```
temperature ← INITIAL_TEMP
generate initial solution
compute score of initial solution
while temperature > STOP_THRESHOLD
     copy: next_solution ← current_solution
     alter: modify next_solution
     evaluate: compute score of next_solution
     accept: probabilistically accept next_solution
     adjustTemperature
```

Figure 2: Simulated annealing pseudocode. An optimal solution is derived by repeatedly executing five steps.

traditional algorithm used to improve pipeline performance. A pipelined processor can only run as fast as the latency of the slowest stage. The algorithm iteratively reduces the latency of the worst stage through DSE until (a) another stage becomes the slowest, (b) no more parallelism can be derived, or (c) FPGA resources are exhausted. FPGA resource (LUTs, flip-flops, and Block RAMs) utilization is estimated using post-place-and-route data for different arithmetic modules, registers, multiplexers, and memories.

Different types of processing stages require different techniques for latency reduction. To illustrate, C code for the classic Traveling Salesperson Problem (TSP), solved using simulated annealing, is presented as an example. Throughout the example, data widths are 16 bits. While TSP is a simple algorithm, it is relevant as it has direct ties to mission planning and event scheduling algorithms. Fig. 2 shows pseudocode for the classic simulated annealing algorithm [2]. The loop shown in fig. 2 contains five tasks that can serve as the basis for a pipelined hardware implementation. This architecture is depicted in fig. 3. The five processing modules shown in fig. 3 (*Copy*, *Alter*, *Evaluate*, *Accept*, and *Adjust Temperature*) can be divided into three types, based upon the manner in which parallelism can be exploited. *Accept* and *Adjust Temperature* are simple modules with no available parallelism, and are not interesting in this discussion.

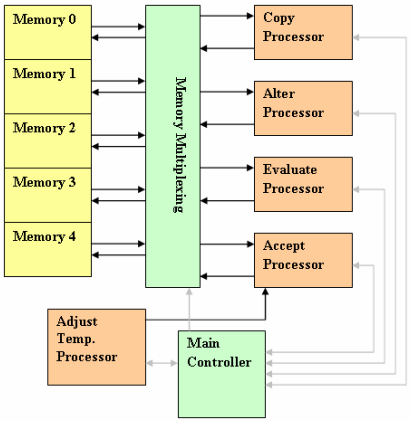

Figure 3: The simulated annealing pipelined architecture.

Alter and *Evaluate* are computationally intensive stages that can benefit from the extraction of both temporal and spatial parallelism through the introduction of additional resources. *Copy* is a memory-intensive stage that can be accelerated by widening memory read and write ports. These different processing stages are discussed in the following subsections.

3.1 Acceleration of Memory-Intensive Stages

The *Copy* stage shown in fig. 3 is a hardware representation of the C code shown here:

```
for (i=0; i<100; i++)
     dest[i] = source[i];
```

This function is realized in hardware by allocating separate memory banks for the source and destination arrays and copying data from one the source bank to the destination bank. The rate at which data can be transferred is a function of both the number of words to be transferred and the width of memory ports. The simplest architecture uses single-word read/write ports and transfers the data one word at a time. A transfer of n words takes $n+1$ clock cycles. In theory, accelerating this transfer would be as simple as creating additional read and write ports. Xilinx Virtex 4 BRAMs provide two read and two write ports each; however, a limitation of Xilinx ISE allows only one write port when BRAM usage is to be inferred from a VHDL memory description. Thus, speedup is only attainable by widening the ports and providing external multiplexing and control logic to provide both single-word and multi-word data accesses. Fig. 4 details how BRAMs are used to create a memory system with double-word read (fig. 4a) and write (fig. 4b) capability. Two copies of the data are maintained, thus doubling BRAM usage. Circuits can also be implemented with four-word or eight-word wide accesses, incurring proportional increases in BRAM usage. Additional FPGA resources are consumed by the multiplexers and control units needed to govern data flow.

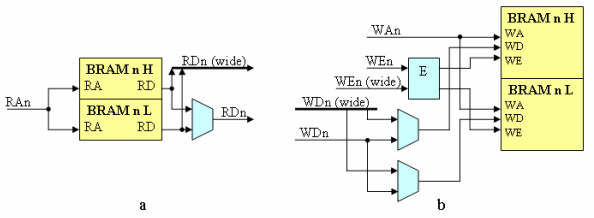

Figure 4: Memory circuit that allows for double-wide data transfers in addition to word-sized accesses.

3.2 Acceleration of Compute-Intensive Stages

Other stages, such as *Alter* and *Evaluate* in this example, are compute-intensive and are excellent candidates for extracting spatial and temporal parallelism. For example, C code for the *Evaluate* stage is as follows:

```
distance = 0;
for (i=0; i<99; i++)
{
    distance += (abs(x_pos[next[i]]
          - x_pos[next[i+1]])
          + abs(y_pos[next[i]]
          - y_pos[next[i+1]]));
}
```

This code loops through all cities in order, accumulating the total distance that the traveling salesperson must travel. One possible architecture for this stage is shown in fig. 5. The architecture shown in fig. 5 corresponds to the internals of the loop, thus data could be streamed through on every clock cycle for 99 straight cycles. Including pipeline draining, this architecture runs in 106 cycles and consumes 176 LUTs, 176 flip-flops, and 8 Block RAMs. The search space of possible architectures for this stage is very large. The fastest architecture would consist of 98 additional copies of the architecture shown in fig. 5, and the slowest would consist of one adder, one subtractor, one read port, etc. A design space explorer repeatedly generates and evaluates different architectures, searching for the smallest footprint that meets a given timing constraint. The DSE is based upon a modified form of Force-Directed Scheduling [9] coupled with an iterative resource allocator.

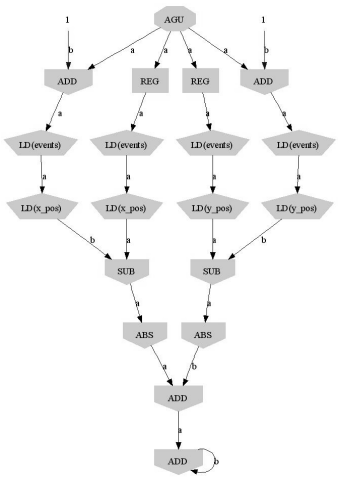

Figure 5: One possible architecture for the TSP *Evaluate* function.

Table 1: DSE Progression

	ASP1	ASP2	ASP3	ASP4
Copy Latency	101 cycles	101 cycles	51 cycles	51 cycles
Alter Latency	24 cycles	24 cycles	24 cycles	24 cycles
Evaluate Latency	409 cycles	78 cycles	78 cycles	45 cycles
Accept Latency	54 cycles	54 cycles	54 cycles	54 cycles
Adjust Temperature Latency	12 cycles	12 cycles	12 cycles	12 cycles
Circuit Size (LUT/FF/BRAM)	4,552 2,668 23	4,856 2,972 28	5,341 2,972 49	5,729 3,324 97

4. Results

When the pipeline derivation algorithm is applied to the 100-city TSP problem, it proceeds as shown in Table 1. *ASP1* designates the minimal architecture. The latency of a pipeline stage for *ASP1* is limited by the worst-performing *Evaluate* stage, 409 cycles. The goal of the DSE is to identify an improved architecture for *Evaluate* with the smallest possible footprint that reduces the latency to less then 101, the latency of the second-worst performing stage (*Copy*). *ASP2* is the result of this effort, with *Copy* now the critical stage. DSE is performed again, creating a double-word data transfer memory, and producing *ASP3*, where *Evaluate* is once again the worst stage. Finally, the latency of *Evaluate* is reduced below that of *Accept* (*ASP4*). Because *Accept* cannot be improved, the algorithm terminates. The bottom line of Table 1 gives resource utilization in LUTs, flip-flops, and Block RAMs for the different circuits.

Table 2 compares the performance in time of each ASP on a Xilinx Virtex 4 XQR4VSX55 with that of the

Table 2: Speedup Comparison (100 MHz clock freq.)

	Clock Cycles per Iteration	Execution Time per Iteration	Speedup
PowerPC Processor	9,530.2	95.3 us	1.0
ASP1 (Minimal Architecture)	410	4.10 us	23.25
ASP2 (ASP1 with Eval. Improved)	102	1.02 us	93.41
ASP3 (ASP2 with Copy Improved)	79	790 ns	120.61
ASP4 (ASP3 with Eval. Improved)	55	550 ns	173.23

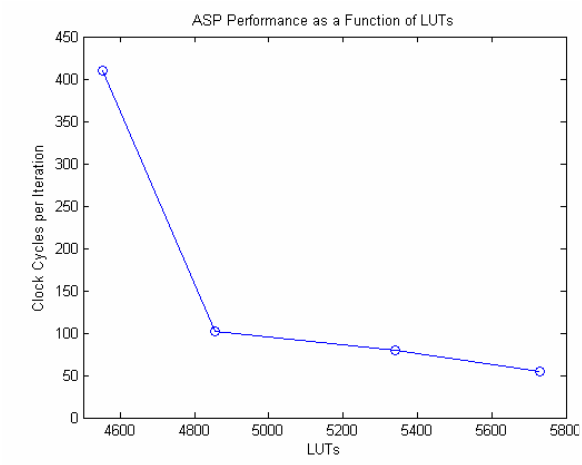

Figure 6: Custom architecture performance as a function of LUT usage.

traditional PowerPC 750 with a floating-point coprocessor. Clock frequency has been normalized across all circuits to 100 MHz, which is a typical clock rate for space-based applications. Speedup ranges from 23 times (*ASP1*) to 170 times (*ASP4*). Figs. 6, 7, and 8 plot latency as a function of resource usage for LUTs, flip-flops, and Block RAMs, respectively. Notice that in all cases, the law of diminishing returns applies as more resources are used by the circuit.

5. Conclusions and Future Work

A technique has been presented for deriving efficient high-level pipelined processors. The technique employs a combination of Design Space Exploration and resource utilization estimation. This algorithm can be automated to quickly derive architectures that outperform conventional processors by orders of magnitude.

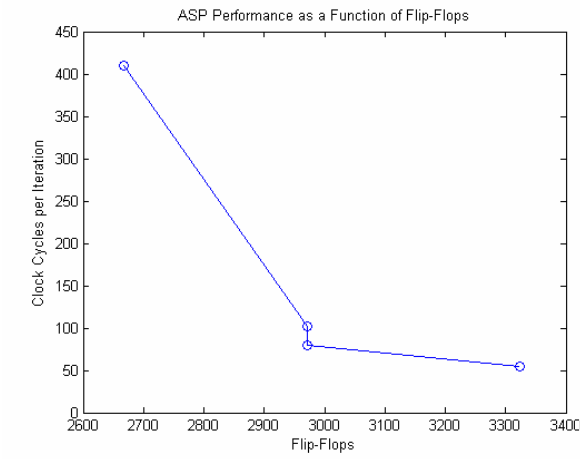

Figure 7: Custom architecture performance as a function of flip-flop usage.

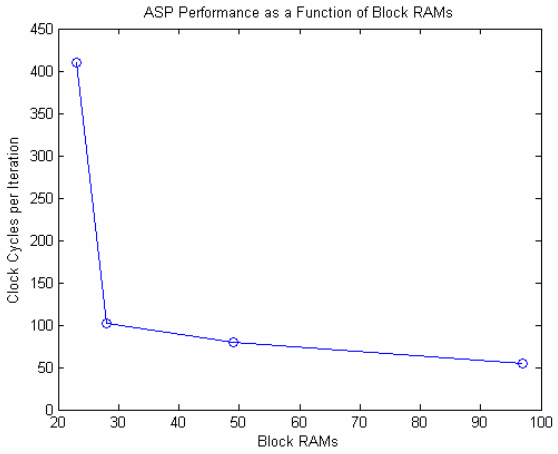

Figure 8: Custom architecture performance as a function of Block RAM usage.

This work is currently in a semi-automated form. Future work involves automating the process to derive architectures directly from C code. Enhancements to the Design Space Explorer to further optimize circuit performance are also underway.

References

[1] Xilinx, "http://www.xilinx.com/publications/prod_mktg/virtex4qv_flyer.pdf," 2008.

[2] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, pp. 671-680, 1983.

[3] S. Knight, G. Rabideau, S. Chien, B. Engelhardt, and R. Sherwood, "Casper: space exploration through continuous planning," *IEEE Intelligent Systems,* vol. 16, pp. 70-75, 2001.

[4] R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, and A. Fukunaga, "Using ASPEN to automate EO-1 activity planning," in *Proceedings of the IEEE Aerospace Conference*, 1998, pp. 145-152 vol.3.

[5] M. Zweben, M. Deale, and R. Gargan, "Anytime rescheduling," in *Proceedings of the DARPA Workshop on Innovative Approaches to Planning and Scheduling*, 1990.

[6] C. Shekhar, S. Raj, A. S. Mandal, S. C. Bose, R. Saini, and P. Tanwar, "Application Specific Instruction Set Processors: redefining hardware-software boundary," in *Proceedings of the 17th International Conference on VLSI Design*, 2004, pp. 915-918.

[7] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: the next design discontinuity," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, pp. 84-90.

[8] M. K. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: survey and issues," in *Proceedings of the Fourteenth International Conference on VLSI Design*, 2001, pp. 76-81.

[9] P. G. Pauline and J. P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," in *24th Conference on Design Automation*, 1987, pp. 195-202.