Parallelizing DSP Nested Loops on Reconfigurable Architectures using Data Context Switching

Kiran Bondalapati
Chameleon Systems, Inc.
161 Nortech Parkway
San Jose, CA 95134, USA
kiran@cmln.com

ABSTRACT

Reconfigurable architectures promise significant performance and flexibility advantages over conventional architectures. Automatic mapping techniques that exploit the features of the hardware are needed to leverage the power of these architectures. In this paper, we develop techniques for parallelizing nested loop computations from digital signal processing (DSP) applications onto high performance pipelined configurations. We propose a novel data context switching technique that exploits the embedded distributed memory available in reconfigurable architectures to parallelize such loops. Our technique is demonstrated on two diverse state-of-the-art reconfigurable architectures, namely, Virtex and the Chameleon Systems Reconfigurable Communications Processor. Our techniques show significant performance improvements on both architectures and also perform better than state-of-the-art DSP and microprocessor architectures.

Keywords

Configurable Computing, Mapping Techniques, Loops

1. INTRODUCTION

Current and future signal processing applications have significant performance and flexibility demands. Algorithmic performance demands in future wireless communication, voice and video processing, etc. are evolving at a faster rate than the performance of computational platforms. Rapidly changing dynamic performance requirements and frequently changing application features and standards demand flexible hardware architectures. Reconfigurable hardware promises to deliver the required performance, flexibility, and cost by combining aspects of microprocessors, ASICs, and DSPs.

Automatic mapping of applications onto configurable hardware is necessary to deliver high performance for applications. Pipelining and parallelizing are the two main techniques employed to exploit reconfigurable hardware. Loops with simple control and no data dependencies between different iterations of the loop are easy to compile/synthesize for high performance on a variety of architectures. Mapping loops onto various configurable architectures has been studied extensively in research. In this paper, we focus on mapping nested loops that have dependencies. These dependencies do not permit existing parallelization techniques to be applied. The dependencies in such computations limit the throughput of the pipelined computations. Some examples of such loops are Infinite Impulse Response (IIR) Filters and adaptive filters. We will use the IIR as a motivating example in the remainder of the paper.

In this paper, we develop an approach to map nested loops by using a combination of pipelining, parallelization and our proposed optimization - *data context switching*. Each iteration of the outermost loop in the computations defines a *data context*. Data context switching interleaves the execution of the iterations of the loops. The data dependency in the inner loop reduces the throughput that can be achieved due to the inherent pipeline delays. We use embedded memory blocks in the architecture to parallelize the outer loop of the computation to increase the throughput. The resulting designs have the optimal throughput of one output per cycle, utilizing reduced hardware. The mapping scales with the number of loop iterations and the amount of hardware resources. We compare the performance benefits of experimental mappings using our approach with performance that can be achieved on state-of-the-art microprocessors and DSPs.

Section 2 describes the nested loop mapping problem and our *data context switching* approach. Section 3 illustrates the performance benefits that can be achieved on the reconfigurable architectures using our approach and Section 4 draws conclusions.

2. PARALLELIZING NESTED DSP LOOPS

Loop computations provide an opportunity for parallelizing the computations on reconfigurable architectures. Reconfigurable architectures have a large number of functional units which can be utilized for concurrent computations (parallel or pipelined). Pipelined functional units support high clock frequency and hence high performance. The rich dynamic interconnection network provides enough data bandwidth for parallel and pipelined computations. We first clarify some of the nomenclature used in the remainder of the paper by introducing some definitions.

38th DAC, June 18-22, 2001, Las Vegas, Nevada, USA.

2.1 Nested Loops

Loops which have loop carried dependencies are difficult to execute concurrently. In this paper, we focus on the parallelization and pipelining of nested loop computations that have loop carried dependencies. Such loops occur in several classes of applications including signal processing. An example of such a computation is an Infinite Impulse Response (IIR) filter:

$$y(n) \ = \ a_0 * x(n) \ - \ \Sigma_{k=1}^{10} a_k * y(n-k)$$

In typical signal processing applications, this filter is executed for a frame (a specified set of samples such as 80) and for a large number of channels (e.g. 100). The filter coefficients are different for each channel and the sampled waveform is different for each channel. The complete code where the i loop denotes different channels is given below:

```
for i=1 to 100 do
   for j=1 to 80 do
      sum = 0;
      for k=1 to 10 do
S1:      sum = sum + a[i,k]*y[i,j-k-1];
      endfor
S2:   y[i,j] = a[i,0] * x[i,j] - sum;
   endfor
endfor
```

In the following sections, we show how such loops can be mapped onto reconfigurable architectures by using pipelining and *data context switching*. The source code for the example will be used as a running example to illustrate our techniques.

2.2 Pipelining

In the first phase, the inner loops are transformed into a pipelined datapath. The datapath is constructed for the body of the innermost loop. In the absence of loop-carried dependencies, the loop will have a data flow graph with no cycles. The single stage datapath itself can have internal pipeline registers and internal pipeline delay. This delay is referred to as the single stage pipeline delay, δ. The example shows a datapath with a delay of two.

The innermost loop is unrolled to generate the pipelined datapath for loops with feedback. The depth of the pipeline is the number of iterations of the innermost loop. As shown in Figure 1, unrolling the example results in a depth of ten stages. If there was no loop-carried dependency in the j loop then this pipeline can be mapped and executed on the hardware to generate results at a throughput of one output/cycle. But, the loop-carried dependency results in a feedback path from the last stage of the pipeline to the first stage. This feedback path and the multiplexer to accommodate the selection of the feedback and the input are shown in Figure 1.

2.3 Limitations on the Throughput

In the sample code, there is a loop-carried dependency in the j loop. Each pipeline stage has inherent delay buffers (registers) to satisfy timing constraints of the functional units and the routing. The design operates at a much lower frequency without these delay buffers. A new sample cannot be fed into the pipeline every cycle due to this dependency. Hence, the throughput of the pipeline reduces to the delay of each pipeline stage. Also, the outermost loop (i loop) will have to be executed sequentially after finishing the complete j loop. The computation cannot be interleaved due to the loop-carried dependency.

Let δ denote the pipeline stage delay and N_r denote the number of iterations of the iterations of the r loop ($r \in \{i, j, k\}$). The throughput of the pipeline is $\frac{1}{\delta}$ and total time to execute the loop in number of cycles is given by

$$T_{pipe} \ = \ \delta * (N_k + N_j) * N_i$$

On typical DSP engines and microprocessors, loop transformation does not provide any performance benefits. Loop unrolling of the k loop can provide additional instructions in the basic block for more Instruction Level Parallelism (ILP). But, the memory bandwidth required for executing each computation, S_1, limits the performance even if there are multiple functional units. Simple mapping onto reconfigurable architectures will also face similar limitations in spite of multiple functional units. The embedded memory in reconfigurable architectures is exploited to perform *data context switching* to eliminate the memory bandwidth problem.

2.4 Data Context Switching (DCS)

The outermost loop (i loop) in the example does not have any loop-carried dependencies. Using the pipelined datapath, the outermost loop can be executed sequentially. The loop can be parallelized by replicating the hardware mapping and executing a subset of the iterations on each replicated pipeline. But, there is a limit on the hardware resources that are available on most reconfigurable architectures, including Virtex and Chameleon RPF. We developed an alternate technique to improve the throughput of the pipelined datapath - *data context switching*.

Each iteration of the outermost loop defines a different *data context*. Each data context differs in the data inputs and constants that are used in the computation. In the example computation, each context differs in the x and y input data and the filter coefficients a. By using data context memories, we simulate multiple versions of the pipeline computing on distinct data sets. *Data context switching* uses the embedded and distributed local memory to store the context information and retrieve it at appropriate cycles in the computation. *Data context switching* is achieved by using three steps:

I. Rescheduling input data: The pipelined design in the previous section computes the full frame (j loop) for one channel (i loop) at a time. In our *data context switching* approach we compute one sample of each channel at a time. This interleaves the execution of the iterations of the outermost loop (i loop) with the execution of the j loop. In conventional architectures this increases the memory bandwidth requirements. In reconfigurable architectures, the inherent pipeline registers in each stage and the data context memories are utilized to store intermediate results. The inherent pipeline delays in each stage are exploited to switch the data context on which the pipeline is operating in each cycle.

II. Memories for constants: The constants used in computations of all the outer loop iterations (such as filter coefficients) are stored in local on-chip memories. This necessitates additional logic for addressing the local memories and accessing the correct constants for each operation in each

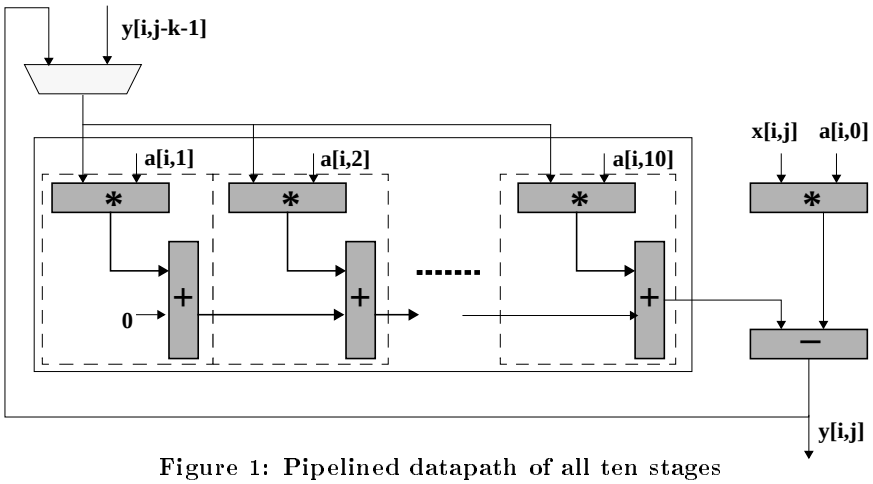

Figure 1: Pipelined datapath of all ten stages

stage. Since these are in local distributed memories, they can be updated by using distributed computational units. This can be exploited for computations in which the constants change, such as adaptive filters.

This memory for constants is shown in Figure 2. A context index counter is utilized to extract the correct constants for each step of the computation as shown in the figure.

III. Data context memories: At any cycle in the computation, the corresponding functional unit in each pipeline stage operates on the same data context (iteration of outermost loop). The intermediate results flow through the pipeline stage and arrive at the next stage in the pipeline after a delay δ. To compute on the same data context in the corresponding functional unit in each stage, this pipeline stage delay should equal the number of contexts that are being computed. To match these, we introduce distributed memories as FIFO buffers which store the context information in each pipeline stage. The re-timed data flow ensures that the correct constants and input operands for each context appear at the inputs of each functional unit in the pipeline.

The resulting datapath of one pipeline stage after applying the data context switching optimization is shown in Figure 2. The single pipeline stage shows how the data context information is distributed throughout the single stage.

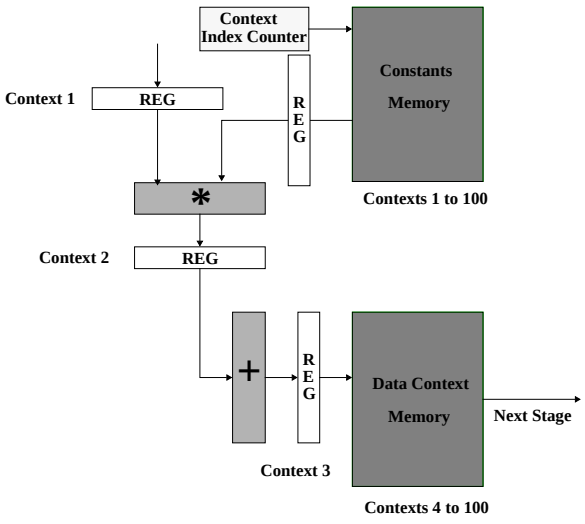

Figure 2: Optimized datapath for one stage

The latency of the pipeline increases to $N_k * N_i$ but the total number of computation cycles becomes:

$$T_{dcs} = (N_k + N_j) * N_i$$

The speedup achieved for executing all the iterations of the loops using *data context switching* is δ. On the Chameleon RPF, the most aggressive pipelined design has a δ of 6. In typical reconfigurable hardware (such as Virtex FPGAs) implementations δ is usually much higher due to pipelined functional units with several stages (such as 5-stage pipelined multiplier).

2.5 DSP/Microprocessor Implementations

The instruction schedule that can be obtained on a RISC or DSP processor is limited by the bandwidth that can be achieved from memory/registers. The number of data values operated on in the nested loop computation make it difficult to store all values in registers. The filter coefficients in the example computation are the values typically expected to be available in registers. If the innermost loop body is mapped onto a schedule of delay T_{body} then the computation on a DSP or microprocessor will run in the following number of cycles:

$$T_{dsp} = T_{body} * N_k * N_j * N_i$$

where N_i,N_j, and N_k denote the number of iterations of the i,j and k loop respectively. On a microprocessor the example loop body can take up to 20 cycles to execute. Reconfigurable architectures attain considerable speedup over DSPs and microprocessors as evident from the number of cycles.

3. PERFORMANCE RESULTS

We performed experiments on various platforms to validate the performance benefits of *data context switching*. The example DSP nested loop discussed in Section 2 is a 10-tap Infinite Impulse Response (IIR) filter that occurs in several signal processing applications including the Voice over IP (VoIP) standards. Examples of similar computations included cryptographic engines that use multiple rounds with feedback and iterative encoders and decoders for compression and error correction.

We mapped the nested loop onto different architectures to obtain performance results. Chameleon Systems Reconfigurable Communications Processor (RCP) [3, 4] integrates

Table 1: Performance Results and Speedups

Platform	Frequency MHz	Approach	Cycles	Speedup (in cycles)	Time (μsec)	Speedup (in time)
UltraSPARC-II	450		800000	1.0	2000	1.0
DSP	300		200000	4.0	660	3.0
Virtex	56.7	Standard	81000	9.8	1426	1.4
Virtex	56.7	DCS	9000	88.9	158	12.7
Chameleon	125	Standard	54000	14.8	432	4.6
Chameleon	125	DCS	9000	88.9	72	27.8
Chameleon	125	DCS+Double	4500	177.8	36	55.6

a custom reconfigurable coarse-grain datapath with a 32-bit ARC RISC processing core. The reconfigurable processing fabric consists of 32-bit programmable arithmetic units and 16x24 multipliers in addition to distributed local storage memories (LSM). Virtex is a fine-grain FPGA from Xilinx that provides a lookup table based architecture. Virtex also supports distributed memory, either as BlockRAM memory blocks or distributed RAM which is built using the lookup table logic.

Table 1 shows the performance of various architectures and the different mapping techniques on the reconfigurable architectures. The results marked as DCS indicate the results obtained using *data context switching*. The base case for comparing the speedup is the UltraSPARC-II implementation. The basic loop body timing, T_{body}, for the microprocessor and DSP is 10 and 2 cycles, respectively. This was obtained based on the most aggressive scheduling of the instructions.

The mapped design on the Virtex had a pipeline single stage delay of 9 cycles. The design runs at 56.7MHz on a -6 speed grade part which is the fastest Xilinx device. The local memory blocks were implemented as a combination of distributed logic cells and the BlockRAM available on the Virtex. This was necessary to balance the usage of the BlockRAM and the distributed memory. A design using only BlockRAMs would fit only on the largest Virtex device, V1000. On a V600 device, 91% of the BlockRAMs and 43% of the logic cells are utilized by combining distributed RAMs and BlockRAMs.

The Chameleon implementation was developed using the C∼SIDE software tools [4]. Two stages of the pipeline (with one multiply-accumulate each) are mapped onto one tile achieving maximum tile usage. The complete design was mapped onto two slices of the Chameleon chip. The design uses 50% of the DPUs and 31% of the LSM memories. The control FSM is simple and is the same for each pipeline stage. Two versions of the design can be mapped onto the Chameleon chip with the available reconfigurable resources. These can operate in parallel to achieve twice the speedup. This speedup is reflected in the last row (DCS+Double) in Table 1.

Using standard pipelining approach on reconfigurable architectures, we obtain speedup of 4.6 over UltraSPARC-II. Using our *dynamic context switching* (DCS) approach, we obtain speedup of up to 27.8 over UltraSPARC-II implementation in actual execution timings (in spite of lower clock speed). The optimized Chameleon mapping achieves a speedup of 9.2 over state-of-the-art DSP architecture which is extensively optimized for such nested loops. By fully utilizing the resources and using two duplicate versions of the mapping, it is possible to further improve the performance by a factor of 2. This is illustrated in the last row (DCS+Double) of Table 1. Chameleon chip can achieve a speedup of 55.6 over UltraSPARC-II implementation.

The results indicate that reconfigurable architectures can achieve impressive speedups over microprocessors by exploiting the reconfigurable logic resources. Our novel *data context switching* approach can significantly enhance the speedup that can be obtained on reconfigurable architectures by exploiting the distributed memory resources. The class of signal processing computations we considered are word-oriented. Chameleon architecture has coarse-grain functional units and performs better than Virtex architecture for such signal processing applications.

4. CONCLUSIONS

We propose a novel *data context switching* technique to map nested DSP loops onto reconfigurable architectures. Nested loops with feedback dependencies occur in several signal processing applications. *Data context switching* overcomes this feedback limitations by switching between different contexts of the outermost loop. Embedded local memory found in most reconfigurable architectures is utilized for *data context switching*. We demonstrate speed-ups using our technique on diverse reconfigurable architectures.

5. ACKNOWLEDGMENT

I am grateful to several of my colleagues at Chameleon Systems (http://www.chameleonsystems.com) who provided significant inputs in developing the above techniques. The techniques developed have been built on the foundation developed while doing my Ph.D. work at University of Southern California. Special thanks go to my mentor at Chameleon Systems, Michael Raam, and my advisor at USC, Prof. Viktor Prasanna.

6. REFERENCES

[1] K. Bondalapati and V.K. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.

[2] K. Bondalapati and V.K. Prasanna. Loop Pipelining and Optimization for Reconfigurable Architectures. In *Reconfigurable Architectures Workshop (RAW '2000)*, May 2000.

[3] L. Caglar and B. Salefski. Reconfigurable Computing in Wireless. In *38th Design Automation Conference*, June 2001.

[4] Chameleon Systems. http://www.chameleonsystems.com/.

[5] Xilinx Inc.(www.xilinx.com). *Virtex Series FPGAs.*