

Automated Dynamic Throughput-constrained Structural-level Pipelining in Streaming Applications

Mark Muir ⁽¹⁾

¹ The University of Edinburgh
Mayfield Road, Edinburgh, EH9 3JL
United Kingdom
Mark.Muir@ed.ac.uk

Tughrul Arslan ^(1,2)

² Institute for System Level Integration
Alba Centre, Livingston, EH54 7EG
United Kingdom

Iain Lindsay ⁽¹⁾

Abstract

Stream processing applications such as image signal processing demand high throughput. However, customers increasingly demand runtime flexibility in their designs, which cannot be provided by custom ASIC solutions. Currently, reconfigurable processors tend to offer insufficient throughput for widespread use in streaming applications. This paper demonstrates how structural-level pipelining techniques can be applied to rapidly dynamically reconfigurable computing architectures, in order to increase throughput. This is done by automatically inserting registers into the data path of performance critical code sections that have already been optimised into a single configuration context. A new algorithm is presented to choose the insertion point of pipeline stage registers in order to meet a specified throughput whilst minimising register resource usage. The paper then demonstrates a new approach where properties of dynamic reconfiguration can be utilised to perform the tasks of pipeline stage initialisation and flushing. The technique is demonstrated on a real-life application: the demosaic filter in a standard image signal processing pipe used in modern digital cameras, and can be seen to boost the throughput from 16MPixels/s to 51MPixels/s on an example reconfigurable processor.

1. Introduction

The choice of platform for many modern digital signal processing tasks in embedded systems is often limited to application-specific integrated circuits (ASICs), since off-the-shelf programmable architectures such as DSPs and microprocessors cannot meet the throughput requirements, whereas reconfigurable hardware such as field-programmable gate arrays (FPGAs) require too much area and power. However, for applications that demand an element of reprogrammability, streaming processors (such as those offered by Ambric [1] and SPI [2]) are becoming an increasingly attractive solution, which improve on throughput by providing multiple processing elements/cores with

an interconnect structure suited to streaming. However, these processing elements—usually based on regular DSP designs—often equate to significant silicon area. Coarse-grained dynamically reconfigurable architectures (DRAs) offer a high degree of parallelism, sufficient to achieve high throughput [3][4]. Thus fewer cores are required for a given application, leading to a much lower area overhead. These coarse-grained architectures are reconfigured very rapidly (e.g. millions of times per second), in order to achieve control flow similar to a regular microprocessor. This paper focuses on maximising the performance of programs running on a single core. However, the techniques can be directly applied to programs running on additional cores in a complete streaming application.

Coarse-grained DRAs, such as instruction cell based computing architectures [5][6], provide a high degree of instruction chaining inside the core, by allowing arbitrary connections to be made between the various functional units via a configurable routing network. This allows quite complex data paths to be rendered onto the fabric and executed in a single configuration. This makes these architectures particularly suitable to stream processing, as fewer fetches from program memory are required. Performance is optimised by attempting to match the size of each kernel (inner loops where most of the execution time is spent) to the available resources, allowing them to fit into a single configuration context. This allows the configuration to persist for many clock cycles, operating on new data on each cycle. This increases throughput, since no time is spent having to reconfigure the core between successive iterations. It also decreases power consumption, as the configuration only needs to be fetched from program memory (or cache) once—upon first entering the kernel—rather than on every iteration. However, the resulting data paths can often have a long critical path, leading to poor temporal utilisation of the functional units, since they have to wait until all functional units have completed before operating on the next batch of data, which limits the throughput.

Pipelining provides a way of starting to operate on a new

batch of data before an old one has completed, so that the functional units of multiple stages of the kernel can be active concurrently; each operating on a different batch of data. This paper describes how structural-level pipelining can be applied *dynamically* to architectures that support a high degree of operation chaining. This is done as part of the configuration—i.e. pipelines tailored to the particular kernel are rendered onto the core at run-time. This has the same effect as adding pipelining in hardware, but can be changed at run-time. Furthermore, these custom pipelines can be initialised and flushed in separate configuration contexts, reducing the resource requirement of the pipelined kernel.

Section 2 reviews existing pipelining techniques, and relevant software optimisation techniques. Section 3.1 describes an algorithm to perform pipeline stage allocation, and section 3.2 shows how properties of dynamic reconfiguration can be used to fill and flush the resulting pipeline. Section 4 shows the result of applying this technique to a real-life kernel used in image processing.

2. Previous work

For architectures that support instruction chaining, scheduling involves mapping as many dependent and independent data paths into as few configuration contexts as possible [7]. Independent data paths run in parallel, so the time for which a configuration persists is determined by the maximum critical path length of these data paths. If sufficient functional unit resources are available, loops can be optimised by loop unrolling [8]—i.e. placing multiple iterations as independent data paths in the same configuration. This allows multiple iterations to begin and end at once. This does not change the original critical path length, yet can increase the throughput. The throughput is determined by the critical path length of a loop iteration and the number of iterations that can be performed at once. During each execution of the loop configuration context, data propagates through the operation chains until the final result is ready. This means that the functional units involved in that chain are only performing useful work for a fraction of the time. This is where structural-level pipelining of these data paths comes in—to artificially reduce the critical path length by allowing new iterations to begin without waiting for the completion of previous iterations.

Various approaches of pipelining data paths have been proposed [9]. These require that the designer specifies a throughput constraint, in order to allow the algorithm to best make the choice between throughput and the area overhead each pipeline stage introduces. These approaches describe various algorithms for the task of pipeline stage allocation, applied to a number of different levels in a design. On reconfigurable architectures such as FPGAs, custom pipelines can be rendered as part of the configuration, leading to sig-

nificant increases in throughput [10].

Performing this pipelining dynamically as part of the configuration allows the throughput of a given DRA core to be increased, without reducing its flexibility. A generic stream processing engine built from these cores would therefore be able to achieve much higher throughputs over a wide range of streaming applications. For a given throughput requirement, fewer cores are required with this approach, which reduces the area and also the complexity of application development.

3. Dynamic pipelining

Conventional structural-level pipelining can be applied to single configuration context kernels with long critical data paths, in order to reduce the critical path, and thus increase throughput. This is done as part of the configuration—i.e. pipelines tailored to the particular kernel are rendered onto the core at runtime. This is done using existing register resources in the core to delay values for a single execution cycle, allowing values to be bridged across pipeline stage boundaries.

Structural pipelining is applied to the kernel basic block by first assigning each operation in the original data flow graph to a pipeline stage. Then, registers are introduced to store values over boundaries between pipeline stages. Figure 1 shows an example kernel before and after structural-level pipelining.

3.1. Pipeline stage allocation

First, constraints are defined between operations, where the order of execution is important. Examples include ‘same stage or earlier’ constraints between operations reading from input registers and operations that have those same registers marked as global output registers, and ‘same stage or earlier’ constraints between data memory read operations and potentially aliasing data memory write operations. All operations in a feedback chain must be placed in the same pipeline stage, since such chains require single-step total latency in order to keep the pipeline full.

The algorithm is a form of list scheduling. Only operations whose predecessors (in the data path) have already been assigned a pipeline stage may be considered for insertion on each pass. In order to minimise the register count, operations should be placed in as late a pipeline stage as possible. Operations that must be placed in the same stage are dealt with together. Operations are considered for placement in the latest pipeline stage containing any of their predecessors. Then, the insertion point is moved towards later pipeline stages until all constraints have been satisfied. Once a valid insertion point has been identified, the critical path is calculated for the resulting (incomplete) configuration context with the operation in that pipeline stage. If the critical path meets the target value, the operation is placed

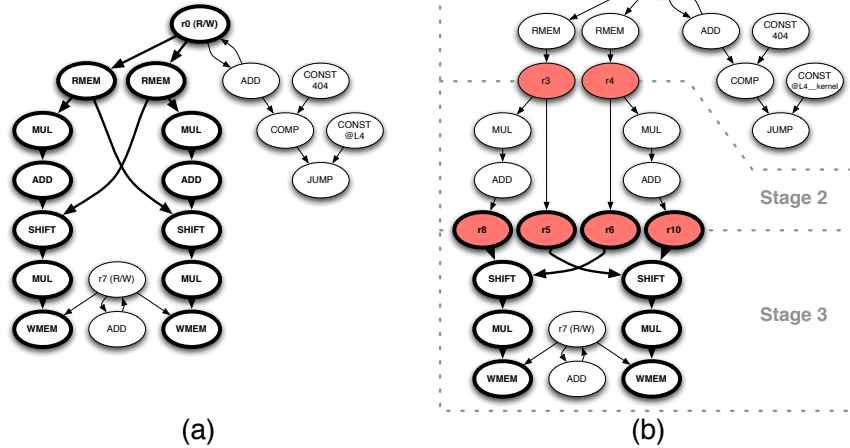


Figure 1: Example kernel data flow graph, (a) before pipelining, (b) after pipelining (kernel loop context). The inserted pipeline stage registers are shown in red. The per-cycle critical path is shown in bold, and is shorter in (b), which allows for a higher throughput.

in that pipeline stage. Otherwise, the operation is added to the next pipeline stage (creating it if it does not exist).

Once the pipeline stages have been determined, pipeline stage registers are assigned as follows: for each pipeline stage in sequence, assign a new register storing the value produced by each operation in all previous pipeline stages that need to be stored for use in this or any later stage.

3.2. Dynamic initialisation and clean-up

Normally, a pipelined design would require additional logic to take care of initialising the pipeline stages, or to suppress the operations in later pipeline stages until the previous stages have filled (predication), so that they do not operate on garbage. However, since the pipelines in a coarse-grained DRA are themselves rendered as part of the configuration context. Provided that the configuration time is not significantly larger than the execution time of each step, dynamic reconfiguration can be used to render different configurations before the main kernel loop configuration, to fill successive stages of the pipeline, and similarly to flush the pipeline after exiting the kernel loop. This allows the kernel loop configuration to assume that the pipeline stages are always full.

Prologue: New configuration contexts are created to initially fill each successive stage of the pipeline. For n pipeline stages, $n - 1$ pipeline filling contexts are created.

Loop: A single configuration context is created for the kernel loop, which includes all pipeline stages.

Epilogue: New configuration contexts are created to flush successive stages of the pipeline. For n pipeline stages, $n - 1$ pipeline flushing contexts are created.

The core is dynamically reconfigured to first perform pipeline initialisation, then reconfigured to execute the kernel loop, then finally reconfigured to flush the pipeline—as demonstrated in figure 2. This is similar to the epilogue and prologue in software pipelining [11].

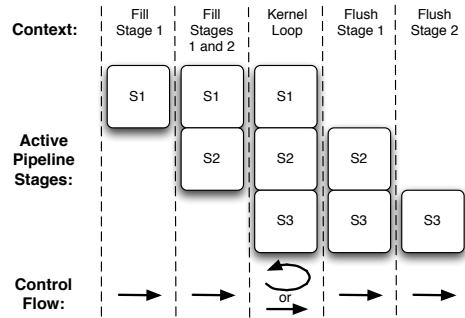


Figure 2: Control flow for a 3-stage pipelined kernel, showing which stages are active in each context (and moment in time). Execution flows from one context to the next, except in the kernel loop, which loops back to itself (holding the same context) until the end condition is satisfied.

Figure 2 shows which stages of the pipeline are active during execution for a 3-stage pipeline. As the target architectures may not be state free (e.g. memory access), it is important to not allow any operation in any pipeline stage to operate on garbage, and to preserve the execution count. With the arrangement shown in the figure, all pipeline stages will be executed the same number of times irrespective of the number of iterations performed in the kernel loop.

4. Application to streaming

The algorithm described in this paper was applied to a real-life application: a 3-line demosaic filter [12], which in-

volves interpolating missing colour components from the Bayer output of a colour filter array sensor. This is a high-throughput task normally done on-chip (integrated into the sensor) as part of a custom image signal processing pipeline, used in modern digital cameras and mobile phones. This is typically the most computationally intensive part of a standard Image Signal Processor (ISP). The filter was re-implemented on a reconfigurable instruction cell-based processor [5], using the C language. Software optimisation techniques were used to reduce the filter kernel into a single basic block, small enough to fit onto the target architecture in a single configuration context. The throughput of the resulting filter is given in table 1.

The operations of the resulting kernel were then pipelined using the algorithms described in this paper, for several target critical path lengths (timing constraints), the results of which are also given in table 1.

Target critical path (ns)	None	40.0	30.0	20.0	19.0	16.0
Actual critical path (ns)	60.3	38.4	29.2	21.3	20.3	19.6
Throughput (MPixels/s)	16.4	26.0	34.2	47.0	49.3	51.2
Pipeline stages	-	2	3	4	6	6
Additional registers	0	10	17	27	31	34
Additional contexts	0	2	4	6	10	10

Table 1: Performance of the demosaic filter kernel before pipelining, and after pipelining. Throughput and additional register and program memory resource requirements are shown.

Pipelining can be seen to increase the throughput, at the expense of extra registers, and additional program memory for the prologue and epilogue. The last column in table 1 shows that a natural throughput limit is reached, determined in this case by the length of the feedback chains present in the kernel data flow graph. Note that for a target of 19.0ns in this case, the resulting critical path is greater than that obtained for a target of 16.0ns. This represents boundary noise in the register stage allocation algorithm, where depending on previous choices, a different local minimum may be found.

5. Conclusions

This paper demonstrates that structural-level pipelining techniques can be applied via software to rapidly reconfigurable/programmable architectures supporting operation-chaining, where complete kernels can be mapped into a single configuration context/cycle. This improves throughput by reducing the critical path length of the looping kernel. This makes such architectures ideal candidates for use as the cores in a stream processing engine, as fewer cores are needed to meet a particular throughput. This work concentrated on reconfigurable instruction cell processors, which support a high degree of operation chaining. However, the same techniques could be applied to other quite different architectures that support operation chaining, such as upcoming VLIW/ULIW processors.

Furthermore, this paper introduced the idea of achieving pipeline filling and flushing through dynamic reconfiguration, in a manner similar to that used in software pipelining. Pipelining was shown to increase the register requirement, and uses more program memory to store the additional configuration contexts (prologue and epilogue). However, the program memory overhead can be largely avoidable, since the additional contexts are suited to temporal compression. The potential throughput is limited by the number of registers available for use in connecting the pipeline stages, and by the presence of feedback loops that demand single cycle latency (e.g. when updating the value of a register). The algorithm was applied to a demosaic filter for a variety of target throughput constraints, and achieved a maximum throughput of more than three times that of the original kernel.

References

- [1] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM*, 2007, pp. 55–64.
- [2] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. Daly, "A programmable 512 GOPS stream processor for signal, image, and video processing," in *Solid-State Circuits Conference*, 2007, pp. 272–602.
- [3] A. Major, T. Arlsan, *et al.*, "H.264 decoder implementation on a dynamically reconfigurable instruction cell based architecture," in *International SOC Conference*, 2006, pp. 49–52.
- [4] Z. Khan, T. Arlsan, *et al.*, "Implementation of a real time programmable encoder for low density parity check code on a reconfigurable instruction cell architecture," in *Design Automation Conference, Asia and South Pacific*, 2007, pp. 583–588.
- [5] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 1–11, 2008.
- [6] "Loosely-biased heterogeneous reconfigurable arrays," U.S. Patent 20 050 257 024, 2005.
- [7] Y. Yi and I. Nousias, "System-level scheduling on instruction cell based reconfigurable systems," in *Design Automation and Test in Europe, International Conference on*, 2006, pp. 381–386.
- [8] J. Sanchez and A. Gonzalez, "The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures," in *ICCP Parallel Processing, International Conference on*, 2000, p. 555.
- [9] S. Bakshi and D. Gajski, "Partitioning and pipelining for performance-constrained hardware/software systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 7, no. 4, pp. 419–432, 1999.
- [10] S. Silva and S. Bampi, "Area and throughput trade-offs in the design of pipelined discrete wavelet transform architectures," in *Design Automation and Test in Europe, International Conference on*, 2005, pp. 32–37.
- [11] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *ACM SIGPLAN conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 318–328.
- [12] J. Mukherjee, M. Moore, and S. Mitra, "Color demosaicing with constrained buffering," in *Signal Processing and its Applications, Sixth International Symposium on*, vol. 1, 2001, pp. 52–55.