Automatic Counterflow Pipeline Synthesis

Bruce R. Childers, Jack W. Davidson
Computer Science Department
University of Virginia
Charlottesville, Virginia 22901
{brc2m, jwd}@cs.virginia.edu

Abstract

The Counterflow Pipeline (CFP) organization may be a good target for synthesis of application-specific microprocessors because it has a regular and simple structure. This paper describes early work using CFP's to improve overall application performance by tailoring a CFP to the kernel loop of an application. A CFP is customized for an application using the kernel loop's data dependency graph to determine processor functionality and interconnection. Our technique builds the design space for a given a data dependency graph and explores the space to find the design with the best performance. Preliminary results indicate that speed-up for several small graphs range from 1.3 to 2.0 and that our design space traversal heuristics find designs that are within 10% of optimal.

1. Introduction

Application-specific microprocessor design is a good way to improve the cost-performance ratio of an application. This is especially useful for embedded systems (e.g., automobile control systems, avionics, cellular phones, etc.) where a small increase in performance and decrease in cost can have a large impact on a product's viability. A new computer organization called the *Counterflow Pipeline* (CFP), proposed by Sproull, Sutherland, and Molnar [6], has several characteristics that make it a possible target organization for the synthesis of application-specific microprocessors. The CFP has a simple and regular structure, local control, high degree of modularity, asynchronous implementations, and inherent handling of complex structures such as result forwarding and speculative execution.

This document summarizes our Counterflow Pipeline synthesis technique and preliminary results.

1.1. Synthesis Strategy

Most high-performance embedded applications have two parts: a control and a computation-intensive part. The computation part is typically a kernel loop that accounts for the majority of execution time. According to Amdahl's Law [3], increasing the performance of the most frequently executed portion of an application increases overall performance. Thus, synthesizing custom hardware for the computation-intensive portion may be an effective technique to increase performance.

Our synthesis system uses the data dependency graph of an application's kernel loop to determine processor functionality and interconnection. Processor functionality is determined from the type of operations in the graph and processor topology is determined by exploring the design space of all possible interconnection networks.

1.2. Counterflow Pipeline

The Counterflow Pipeline has two elastic pipelines flowing in opposite directions. One is the instruction

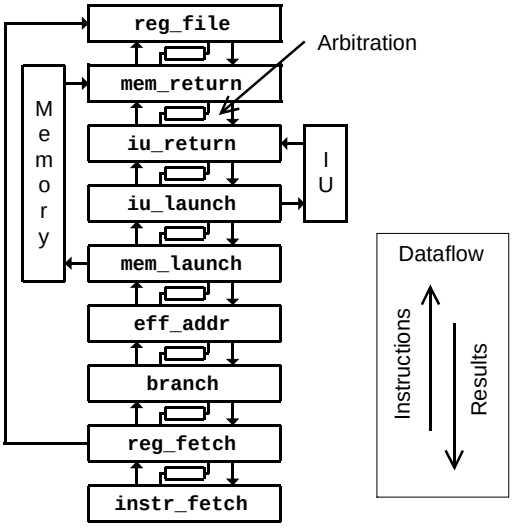

Figure 1: Example Counterflow Pipeline

pipeline. It carries instructions from an instruction fetch stage to a register file stage. When an instruction issues,

an *instruction bundle* is formed that flows through the pipeline. The instruction bundle has space for the instruction opcode, operand names, and operand values. The other is the results pipeline that conveys results from the register file to the instruction fetch stage. The instruction and results pipelines interact: instructions copy values to and from the result pipe.

Functional units (or *sidings*) are connected to the pipeline through *launch* and *return* stages. Launch stages issue instructions into functional units and return stages extract results from functional units. Instructions may execute in either pipeline stages or functional units.

A memory unit connected to a CFP pipeline is shown in Figure 1. Load instructions are fetched and issued into the pipeline at the `instr_fetch` stage. A bundle is created that holds the load's memory address and destination register operands. The bundle flows towards the `mem_launch` stage where it is issued into the memory subsystem.

When the memory unit reads a value, it inserts the value into the result pipeline at the `mem_return` stage. In the load example, when the load reaches the `mem_return` stage, it extracts its destination operand value from the memory unit. This value is copied to the destination register value in the load's instruction bundle and inserted into the result pipe. A *result bundle* is created whenever a value is inserted into the result pipeline. A result bundle has space for the result's name (i.e., register name) and value. Results from sidings or other pipeline devices flow down the result pipe to the `instr_fetch` stage. Whenever an instruction and a result bundle meet in the pipeline, a comparison is done between the instruction operand names and the result name. If a result name matches an operand name, its value is copied to the corresponding operand value in the instruction bundle. When instructions reach the `reg_file` stage, their destination values are written back to the register file and when results reach the `instr_fetch` stage, they are discarded. In effect, the register file serves as a history buffer of results that have exited the pipe.

The interaction between instruction and result bundles are governed by special *pipeline* and *matching rules* that ensure sequential execution semantics. These rules govern the execution and movement of instructions and results and how they interact.

Arbitration is required between stages so that instruction and result bundles do not pass each other without a comparison made on their operand names. In Figure 1, the blocks between stages depict arbitration logic. A final mechanism controls purging the pipeline on an exception. A *poison pill* is inserted in the result pipeline whenever a fault is detected. The poison pill purges both pipelines of all instruction and result bundles. This purge mechanism can also be used for speculative execution when a branch target is mispredicted.

As Figure 1 shows, stages and functional units are connected in a very simple and regular way. The connections correspond to bundled interfaces of micropipelines. The behavior of a stage is dependent only on the adjacent stage in the pipeline, which permits local control of stages and avoids the complexity of conventional pipeline synchronization.

1.3. Experimental Framework

Figure 2 depicts the synthesis framework. The system accepts a C source file containing a loop that drives customizing a CFP processor. The loop is compiled by the optimizer *vpo* [1] into instructions for the SPARC architecture [5]. During optimization, *vpo* builds the

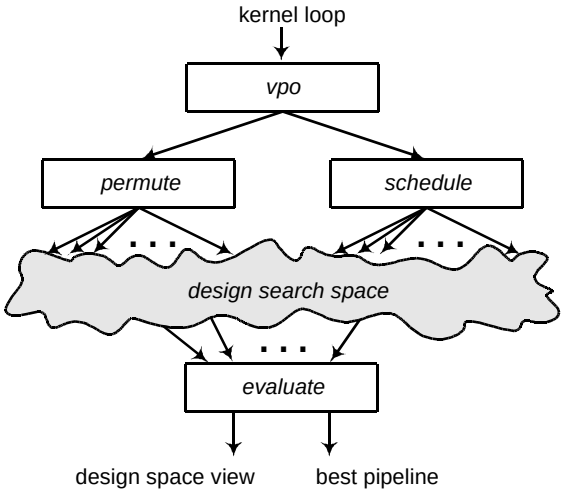

Figure 2: Synthesis framework

data dependency graph for the kernel loop. A separate synthesis module uses the graph to determine computational devices and emits a CFP specification indicating those elements. The specification is an input to a *permute* module that derives all legal CFP topologies for the listed processing elements. *vpo* emits optimized SPARC instructions that serve as input to a *schedule* module that determines all legal instruction schedules. The cross-product of pipelines from *permute* and instruction schedules from *schedule* defines the design search space. A final module *evaluate* traverses the search space to collect simulation statistics for every processor/schedule combination. These statistics are used to pick the best pipeline/schedule combination and to emit a visual representation of the design space. The visual representation shows the overall design space and simulation details for each design point.

Figure 3 shows the visual representation of a single design point. The top half of the figure shows resource usage for every instruction and result per cycle. The

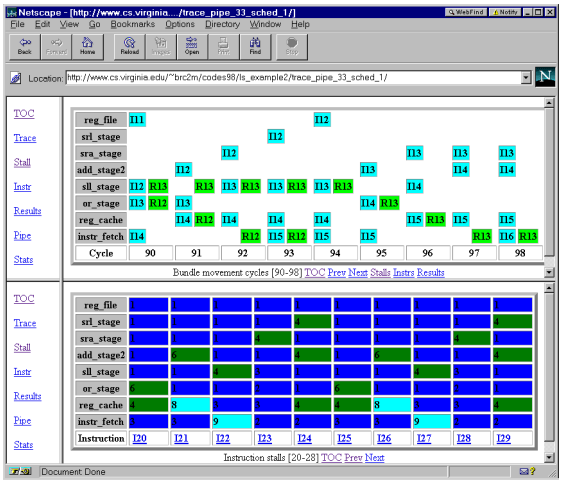

Figure 3: Visual representation of a design point

boxes labeled with **I** indicate instructions as they flow from the instruction fetch stage toward the register file. The boxes labeled with **R** indicate results flowing from the stage where they are produced to where they are consumed. The second half of the figure shows the *stall density* of instructions. This graph shows how long an instruction spends in each pipeline stage. It is especially useful for understanding where instructions stall in the pipeline.

2. Synthesis Methodology

We are exploring techniques for synthesizing a Counterflow Pipeline customized to a kernel loop. Our first experiments used an iterative refinement methodology to derive a pipeline layout [2]. The iterative approach picks a pipeline layout, executes the kernel loop using the layout, collects an execution trace, and refines the layout using the trace. This is repeated iteratively until there are no further performance gains.

The iterative refinement work verified that customizing a CFP to a kernel loop can improve performance. It also demonstrated that minimizing the distance results flow between their production and consumption affects performance because the further a result flows, the greater the latency of conveying the result. If there are many instructions between the set and use of a source operand, the latency can be very high since a comparison of the operand's name is required with every intervening instruction's source and destination register names.

The iterative refinement technique suffers from the disadvantage that it does not lend itself well to automation because it requires careful hand-tuning of a CFP design. Our experience with this technique suggested a second synthesis methodology. Instead of iteratively refining a design, we generate all processor topologies for a given set of functional units and pipeline stages to pick the best one.

Although it not apparent how to build the full design space for a traditional microprocessor organization, it is straightforward for the CFP since pipeline stage order specifies processor topology. This is true for all Counterflow Pipeline computational elements since they are connected via the pipeline. For example, a memory unit has launch and return stages that connect it to the pipeline. A *CFP design space* is all permutations of stages for a given partitioning of functionality. A *partitioning of processor functionality* is an assignment of data dependency graph nodes to computational elements such as pipeline stages and sidings.

Given a data dependency graph $G = (N, E)$, where N is a set of nodes and E is a set of edges, our synthesis system proceeds in three steps:

1. *Partition:* $\forall n \in N$, assign n to a pipeline stage or functional unit. If n is assigned stage s, add s to the set of stages PS. If n is assigned a functional siding, add the functional unit's launch stage to PS. Low latency operations are assigned unique pipeline stages, while high latency operations may share sidings.

2. *Permute:* For the set PS of pipeline stages and the set S of legal instruction schedules, construct the design space $D = perm(PS) \times perm(S)$.

3. *Evaluate:* $\forall d \in HD \subseteq D$, simulate d to determine performance, where HD is a subspace determined by applying some *search heuristic*.

It is not practical to exhaustively search the entire design space for most dependency graphs. Although the space could be small, it is not likely with aggressive instruction-level parallelism transformations such as speculative and predicated execution, software pipelining, *if*-conversion, etc. are applied [4].

Result flow distance and critical path stage order are important factors in obtaining good performance from a CFP design. A heuristic that uses these to guide the exploration of a CFP design search space may narrow the space sufficiently and accurately so a good stage order and schedule are found. Our present heuristics do not consider instruction schedule because the data dependency graph constrains the number of schedules.

We consider two heuristics. The first confines the search space to designs that have pipeline stages in order of the critical path:

> *Heuristic 1: For all nodes* $\{n_1, n_2, \ldots, n_k\}$ *and edges* $\{(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)\}$ *on the critical path, evaluate only designs that have the partial order* $\{n_1 \ll n_2, n_2 \ll n_3, \ldots, n_{k-1} \ll n_k\}$ *wrt. pipeline stages.*

This order overlaps the execution of instructions from different loop iterations (loop-carried dependences may affect this, of course). Stages that execute non-critical path instructions may occur any place in the pipeline (no constraints are placed on the order). Figure 4 shows

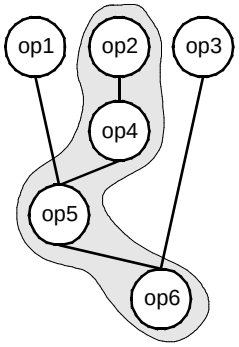

Figure 4: Example data dependency graph

an example data dependency graph with the critical path highlighted. The partial order for this example is:

$$\{op2 \ll op4, op4 \ll op5, op5 \ll op6\}$$

For example, *op1* can appear any place, while *op5* must appear after *op4* and before *op6*. Although this heuristic determines good pipeline layouts, in many cases it must consider a large number of design points.

A second heuristic preserves critical path order using the instruction dependency graph to define operation partitions by drawing cuts across each level of the graph.

> *Heuristic 2: For the graph* $G = (N, E)$*, divide N into K partitions, where K is the maximum path length in E. Use the partitions to impose a partial order on pipeline stages such that* $\forall n_i \in partition_k$ *and* $\forall n_j \in partition_{k+1}$ *then* $n_i \ll n_j$ *is in the partial order. Assign nodes to a partition according to some assignment heuristic and the dependence edges E. Evaluate only designs that have the partial order wrt. pipeline stages.*

The cuts determine a partial order that places operations from level *n* before operations from level *n+1* (root is level 0) in a pipeline. For example, Figure 5 shows two possible assignments of operations to graph cuts. The partial order for the greedy assignment is:

$$\{op1 \ll op4, op2 \ll op4, op3 \ll op4, op4 \ll op5, op5 \ll op6\}$$

In this example, *op1*, *op2*, and *op3* all must occur within the first three pipeline stages (in any order), *op4* occurs in the fourth position, *op5* in the fifth, and *op6* in the sixth position.

Nodes can be assigned to different instruction partitions. Greedy assignment may work well since assigning nodes to early instruction cuts ensures those operations begin executing as soon as possible. Late assignment may also work since it can minimize the distance results flow between their definition and use. Figure 5 also shows the dependency graph with late assignment of nodes to partitions. Late assignment

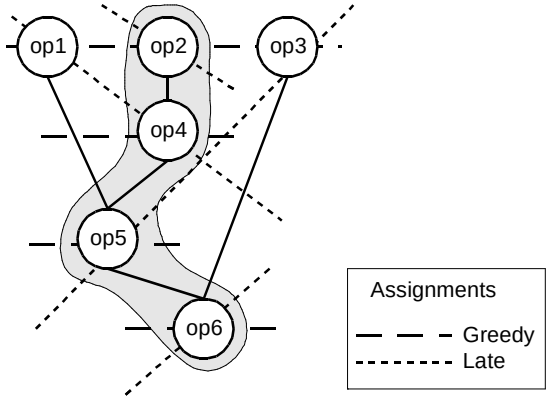

Figure 5: Greedy assignment of graph cuts

works best for this graph since it minimizes the distance results move between their production and consumption. The partial order for late assignment is:

$$\{op1 \ll op5, op1 \ll op3, op2 \ll op4, op2 \ll op1,\ op3 \ll op6, op4 \ll op5, op4 \ll op3, op5 \ll op6\}$$

3. Results

Preliminary results for several small graphs are shown in Figure 6. The speed-up in the figure is relative to a general-purpose pipeline that has separate sidings for memory, multiplication, and integer operations and a pipeline stage for branch resolution. The figure shows speed-up for three pipeline orders: *optimal*, *heuristic 1*, and *heuristic 2*. In all cases, the partitioning of functionality is the same: every graph node is assigned an unique pipeline stage or siding. The *optimal* pipeline had the best performance from all pipeline stage permutations.

The *heuristic 2* pipelines use late assignment to allocate graph nodes to partitions.

The benchmarks in the figure are small data dependency graphs that have less than 8 nodes. The graphs contain mostly low latency integer operations, although graph 10 has two memory references and a multiplication. We selected these initial benchmarks because they were small enough to generate the full design space and demonstrate the effectiveness of the search heuristics.

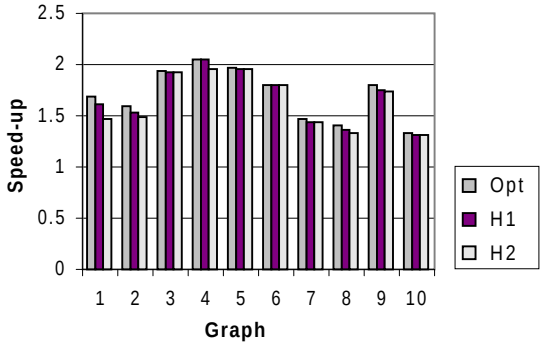

Figure 6: Speed-up for several small graphs

The figure shows that the search heuristics find pipelines that are nearly as good as optimal. The performance difference between the heuristically determined pipelines and the optimal pipeline is generally less than 10%. This difference is partly influenced by start-up cost: The optimal stage orders usually have a lower start-up penalty because they order stages to favor requesting source operands from the register file

Both search heuristics work well. *Heuristic 2* does nearly as well as *heuristic 1*, while evaluating fewer designs. Indeed, for graphs 3, 5, 7, and 10, *heuristic 2* finds the same pipelines as *heuristic 1*.

Table 1 shows a comparison of the number of designs evaluated by each heuristic. The first column of the table is the total number of points in each design space, while the second and third columns is the number of points each heuristic evaluates. Both heuristics reduce the search space; however, the amount of the reduction is dependent on the shape of the dependency graph.

Graph	Total Designs	Heuristic 1	Heuristic 2
1	72	12	6
2	960	160	32
3	480	20	8
4	240	10	4
5	360	15	6
6	960	160	32
7	10,800	450	60
8	7,920	330	44
9	17,280	720	96
10	33,696	7,056	288

Table 1: Number of design points

Table 1 and Figure 6 show that the heuristics constrain the search space to a small number of pipelines and find designs that are nearly as good as optimal. Although these initial experiments are small, we expect the heuristics to also work for full applications.

4. Conclusion

This paper describes preliminary experiments that indicate the Counterflow Pipeline organization is a flexible target for high-level synthesis of application-specific microprocessors. We are continuing to research the performance potential of custom CFP's, including micro-architecture extensions that may greatly improve performance without sacrificing ease of design.

References

[1] Benitez M. E. and Davidson, J. W., "A Portable Global Optimizer and Linker", *Proc. of the SIGPLAN Notices 1988 Symposium on Programming Language Design and Implementation*, pp. 329–338, Atlanta, Georgia, June 1988.

[2] Childers B. R., Davidson J. W., and Wulf W. A., "Synthesis of Application-Specific Counterflow pipelines", *Workshop on the Interaction between Compilers and Computer Architecture*, held during *ACM HPCA-2*, San Jose, Ca., February 3–7, 1996.

[3] Hennessy J. L. and Patterson D. A., *Computer Architecture: A Quantitative Approach*, *2nd edition*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[4] Rau B. R. and Fisher J. A., "Instruction-level parallel processing: History, overview, and perspective", *J. of Supercomputing*, Vol 7, pp. 9–50, May 1993.

[5] SPARC International, Inc., *The SPARC Architecture Manual, Version 8*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[6] Sproull R. F., Sutherland I. E., and Molnar C. E., "The Counterflow Pipeline Processor Architecture", *IEEE Design and Test of Computers*, pp. 48–59, Vol. 11, No. 3, Fall 1994.