Application-Specific Pipelines for Exploiting Instruction-Level Parallelism

Bruce R. Childers, Jack W. Davidson
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
{brc2m, jwd}@cs.virginia.edu

Abstract

Application-specific processor design is a promising approach for meeting the performance and cost goals of a system. Application-specific processors are especially promising 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. Sutherland, Sproull, and Molnar have proposed a new pipeline organization called the Counterflow Pipeline (CFP). This paper shows that the CFP is an ideal architecture for fast, low-cost design of high-performance processors customized for computation-intensive embedded applications. First, we describe why CFP's are particularly well-suited to realizing application-specific processors. Second, we describe how a CFP tailored to an application can be constructed automatically. Third, we present measurements that show CFP's elegantly and simply provide speculative execution, out-of-order execution, and register renaming that is matched to the application. These measurements show that CFP's speculative and out-of-order execution allow it to tolerate frequent control dependences and high-latency operations such as memory accesses. Finally, we show that asynchronous counterflow pipelines may achieve very high-performance by reducing the average execution latency of instructions over synchronous implementations. Application speedups of up to 7.8 are achieved using custom counterflow pipelines for several well-known kernel loops.

1. Introduction

Application-specific processor design is a promising approach for improving the cost-performance ratio of an application. Application-specific processors are 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 [25], has several characteristics that make it an ideal target organization for the synthesis of application-specific ILP-processors. The CFP has a simple and regular structure, local control, high degree of modularity, asynchronous implementations, and inherent handling of complex structures such as register renaming and speculative execution.

Modern ILP-processors must be able to tolerate high-latency operations and the frequent presence of control transfer operations. As an example, the 4-way superscalar HP PA-8000 microprocessor [15] tolerates a cache miss penalty of 50 clock cycles, which may cause the processor to stall for up to 200 instructions.

To keep aggressive superscalar designs busy requires large instruction windows and special structures (e.g., register rename buffers, data prefetching support) to overcome high-latency operations and control dependences. In the PA-8000, this is accomplished with a 56-entry instruction re-order buffer, data prefetch instructions, and branch prediction and history tables.

The typical hardware structures for implementing out-of-order and speculative execution are expensive; e.g., they consume a large portion of chip area and power budget, and they complicate microarchitecture design. In contrast, the counterflow pipeline cheaply and naturally implements out-of-order and speculative execution without requiring special hardware structures. The combination of these features allows the counterflow pipeline to achieve high performance at a lower cost than traditional processor organizations.

To get high-performance, modern superscalar processors include multiple functional units for exploiting instruction-level parallelism. The CFP has a simple and regular way to incorporate multiple functional units into a design, which permits fast, low-cost customization of counterflow pipelines to an application's resource and data flow requirements. Furthermore, the counterflow pipeline may be implemented as an asynchronous microarchitecture. This improves performance because average instruction latency is reduced versus synchronous counterflow pipelines.

This paper is organized as follows. The first section

has introductory material about our custom processor design strategy and the counterflow pipeline organization. The second section describes several design advantages of CFP's for automatic generation of application-specific ILP-processors. The third section contains an explanation of our pipeline customization technique and experimental results that demonstrate the effectiveness of speculative and out-of-order execution and custom asynchronous counterflow pipelines. The fourth section has related work and the fifth section concludes the paper.

1.1. Design 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. Increasing the performance of the most frequently executed portion of an application increases overall performance. Thus, synthesizing custom hardware for the computation-intensive portion of an application may be an effective technique to increase performance.

The type of applications we are considering need only a modest kernel speedup to effectively improve overall performance. For example, JPEG has a function `j_rev_dct()` that accounts for 60% of total execution time. This function consists of applying a single loop twice (to do the inverse discrete cosine transformation), so it is a good candidate for a custom counterflow pipeline. Figure 1 shows a plot of Amdahl's Law [12] for various speedup values of `j_rev_dct()`. The figure shows that a small speedup of the kernel loop of 6 or 7 achieves most of the overall speedup.

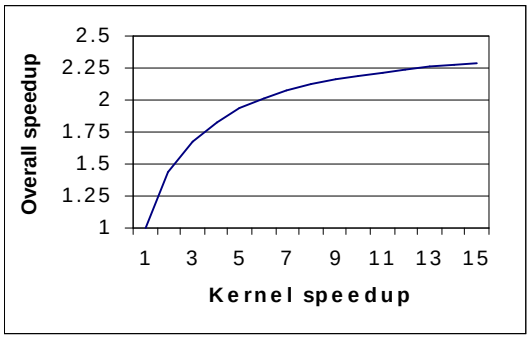

Figure 1: Overall speed-up for JPEG

We use the data dependency graph of an application's kernel to determine processor functionality and interconnection network. Processor functionality is determined from the type of operations in the graph and processor interconnection is determined by exploring the design space of all possible interconnection network.

The target system architecture for our synthesis technique has a single CFP processor that executes the control and computation portions of an application. Our work customizes a general-purpose CFP for the kernel computation to improve performance. This is similar to software acceleration using a co-processor; however, there is only *one* processor in this scheme and, as a result, overall cost should be lower than with a co-processor scheme while still improving performance. A single CPU architecture has the advantage that it does not need any support for synchronization with an attached coprocessor (e.g., interface logic, handling of live-in/out data, etc.).

For applications where there is not a clearly identifiable kernel, the above strategy will not be as effective. However, most applications we have examined have execution profiles similar to JPEG—one kernel that consists of over 50% of the overall execution of the application. Thus, in this paper we consider only kernel loops. However, to accommodate the full application, the custom pipeline should have the functionality needed by both the kernel and control code.

1.2. Counterflow Pipeline

The counterflow pipeline has two pipelines flowing in opposite directions. One is the instruction 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. When an instruction copies a value from the results pipeline, it is called a *garner operation*, and when an instruction copies a value to the results pipeline, it is called an *update operation*.

Pipelined functional units, called *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 2. 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. That is, the instruction *garners* its source operands. 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 stores results that have exited the pipe.

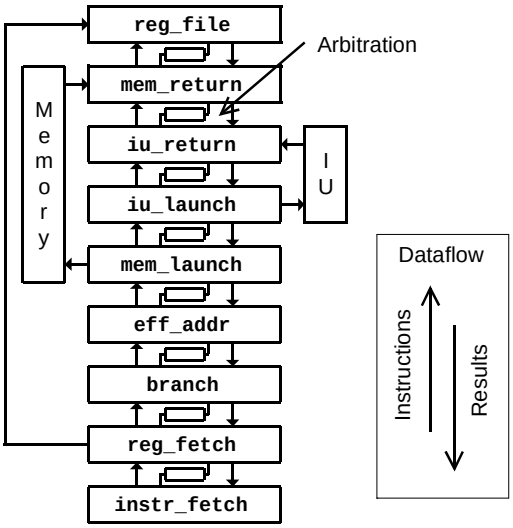

Figure 2: An example counterflow pipeline

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 2, 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 2 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.

2. CFP Design Advantages

Counterflow pipelines have several characteristics that make them suitable for custom ILP-processors: a simple and regular structure, local control, modularity, and asynchronous implementations. These characteristics can be used to achieve higher performance with custom designs than with general-purpose ones.

2.1. Speculative Execution

Traditional dynamically scheduled ILP microarchitectures use branch prediction and speculative execution to keep execution pipelines full [16, 23]. This reduces the impact of control dependences and exposes more instruction-level parallelism to the hardware.

The CFP handles speculative execution in an elegant and simple way. The outcome of branches is predicted at the beginning of the pipeline during the insertion of new instruction bundles. Instructions following a branch are speculatively fetched and inserted into the pipeline. Branch predictions are resolved by a branch resolution stage in the pipeline. When the branch resolution stage detects a misprediction, it inserts a poison pill into the results pipeline. The poison pill kills all instructions it meets while flowing down the result pipeline, and when it reaches the instruction fetch stage, the program counter is changed to the correct branch address carried by the poison pill. The degree of speculative execution is determined by the distance between the instruction fetch stage and branch resolution stage.

2.2. Out-of-Order Execution

An important issue for instruction-level parallel microarchitectures is how to tolerate high-latency operations; especially, memory accesses. Keeping an aggressive ILP-processor busy during memory accesses is becoming difficult as processor widths and memory latencies (relative to processor speed) increase. Indeed, some current superscalar processors have data cache miss penalties of up to 50 clock cycles, and future implementations are likely to see penalties in excess of 100 cycles [16, 23].

Superscalar processors use out-of-order execution to keep functional units busy during high-latency operations. To achieve high performance with out-of-order execution requires reservation stations and re-order

buffers with register renaming [12, 18]. In a conventional microarchitecture, these structures introduce much complexity; however, the CFP inherently handles speculative execution and register renaming in a very simple way.

In the CFP, instructions are kept in order of issue, but they may execute out of order. Two instructions can be executing in different stages of the pipeline at the same time as long as there is no dependency between them. There is no order imposed on which instruction finishes first. Sequential execution semantics are preserved by writing results back to the register file in instruction issue order (and by register renaming in the result pipeline). To enable out-of-order execution, the results pipeline of the CFP implements a type of register renaming. There can be multiple values with the same register name in different places of the result pipeline at the same time. This has the same effect as register renaming: instructions with anti- and output dependencies may execute concurrently with their dependent instructions. This type of out-of-order execution and register renaming is an effective way to hide memory access latency.

2.3. Asynchronous Custom Pipelines

Local control and the simple and regular structures of counterflow pipelines makes the organization very modular. This means that CFP designs can be easily modified for different application requirements and design goals. For example, modules that have been optimized differently (e.g., for speed, area, or power) may be interchanged as long as they maintain the same communication interface.

Asynchronous CFP's have the advantage that computation proceeds at average-case speed instead of worst-case speed in synchronous designs [10]. The combination of local control, the absence of global signals, and asynchronous implementations leads to short communication distances between functional devices in CFP's. This suggests that counterflow pipelines may have very high performance, especially when tailored to an application's resource and data flow requirements.

Although counterflow pipelines may be appropriate for general-purpose processors [17, 19, 25], our research focuses on how to construct custom ILP-processors for embedded systems.

3. Experimental Results

In this section we show how to automatically construct counterflow pipelines using an application's data dependency graph. We also demonstrate that the counterflow pipeline's elegant mechanisms for speculative and out-of-order execution are effective, and that asynchronous custom CFP organizations can significantly improve an application's performance.

3.1. Methodology

The performance statistics in this paper were collected using several common benchmarks. The benchmarks have three Livermore loops (*kernel 1*, *kernel 5*, and *kernel 12*), vector dot product (*dotprod*), the finite impulse response filter (*fir*), memory copy (*memcpy*), and matrix multiplication (*matmult*). Some of our benchmark kernels were extracted from large applications. These loops include the 2-D discrete cosine transformation (*dct*) used in image compression and an implementation of the Floyd-Steinberg image dithering algorithm (*dither)*. We also extracted the vector computation $a = b^c \bmod d$ from RSA encryption (*modexp*). The benchmarks were compiled using the optimizing C compiler *vpcc*-*vpo* [1] for the SPARC architecture [24].

3.1.1 Pipeline Simulation

We have built a behavioral microarchitecture simulator for asynchronous counterflow pipelines. The simulator is highly reconfigurable to permit microarchitecture experimentation, and it generates a detailed program execution trace that is post-processed by a separate analysis tool to collect performance statistics.

To model asynchronous counterflow pipelines our simulator varies computational latencies. Table 1 shows the latencies we use in our simulation models.

Operation	Latency
Stage copy	1 time unit
Garner, kill, update	3 time units
Return, launch	3 time units
Instruction operation	5 time units

Table 1: Computational latencies

The latencies in the table are expressed relative to how long it takes an instruction or result to move between adjacent pipeline stages. Using the base values from Table 1, we derive other pipeline latencies. For example, a simple instruction operation such as addition takes 5 time units. High latency operations are scaled relative to low latency ones, so an operation such as multiplication—assuming it is four times slower than addition—takes 20 time units.

3.1.2 Customization Technique

The experiments in this paper use counterflow pipelines customized to the resource and data flow requirements of each benchmark. Although we have studied search-based pipeline customization techniques [2], we use a different approach in this paper that does not rely on

searching a design space. This approach uses a benchmark's instruction dependency graph to determine processor functionality and interconnection network. The customization process has two steps:

1. *Allocate: Every low latency operation in the instruction dependency graph is assigned an unique pipeline stage and every high latency operation is assigned a (possibly shared) functional siding.*

2. *Arrange: The instruction dependency graph is scheduled using priority-based list scheduling [18] and the pipeline stages determined by step 1 are arranged in reverse order of the instruction schedule.*

Step 1 assigns high latency operations to functional sidings to move their computation out of the main pipeline. This avoids stalling subsequent instructions that may otherwise advance in the pipeline and execute. In step 2, pipeline stages are placed in the reverse order of the instruction schedule to ensure that successive loop iterations overlap in the pipeline. Arranging stages in reverse order lets the pipeline speculatively issue one loop iteration while another is finishing.

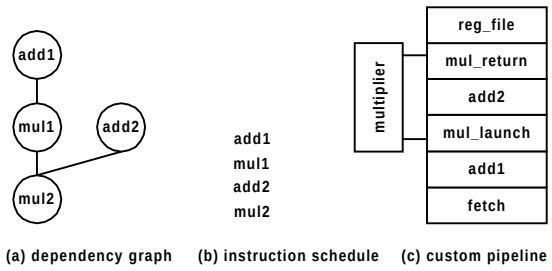

Figure 3: An example of pipeline customizing.

An example of the customization process is shown in Figure 3. The dependency graph in (a) has two addition and two multiplication instructions. According to the first customization step, two addition stages and one multiplication siding are generated.

The second customization step arranges pipeline stages. Using path latency as scheduling priority, local instruction scheduling gives the instruction sequence in (b) for the dependency graph. The pipeline stage order is derived from the reverse order of the schedule, as shown in (c). Thus, the first stage after instruction fetch is `add1` followed by `mul_launch` and `add2`. The second multiplication instruction is skipped since it shares a siding with the first multiplication.

A final issue is where to return multiplication results into the main pipeline. We use the heuristic that the number of stages inclusively between launch and return equals the siding's pipeline depth. This ensures that the siding can be fully utilized. In this example, if the multiplication siding has a depth of 3, then `mul_return` is placed two stages after `mul_launch`.

3.2. Speculative Execution

The location of branch resolution in a counterflow pipeline determines the amount of speculative execution. If branches are resolved early in the pipeline, then very little speculative execution is possible and if branches are resolved late in the pipeline, then much speculative execution is possible. However, late branch resolution impacts the misprediction penalty, which may lead to overspeculation and an adverse affect on performance.

Deep counterflow pipelines require accurate branch prediction. Our CFP designs use dynamic branch prediction to predict branches as they are issued into the pipeline. The program counter is maintained in the instruction fetch stage and updated to the appropriate branch target address whenever a branch is predicted.

The CFP designs we use tag control transfer instruction bundles with their *taken* and *not-taken* target addresses. For most branch instructions, the taken address is encoded directly in the instruction (i.e., the taken target address is PC-relative or absolute). The not-taken address is the address of the instruction following the branch. Both target addresses are needed by the branch resolution stage so that it is able to transmit the correct target address on a branch misprediction to the instruction fetch stage. When the branch stage detects a mispredicted branch, it inserts a poison pill into the results pipeline that contains the address of the correct branch target address. The poison pill flows to the instruction fetch stage carrying the correct target address, and when it reaches instruction fetch, the program counter is updated with the target address.

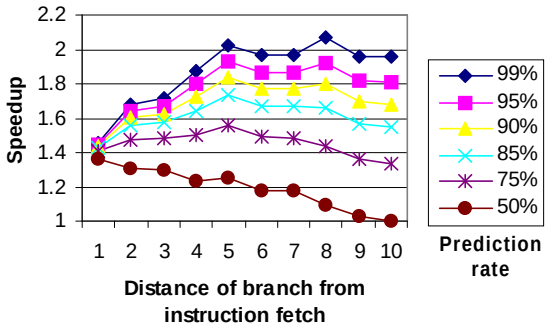

Figure 4: Speedup for *dotprod* using different branch stage positions and prediction rates.

Figure 4 shows the effect of branch prediction accuracy on performance for a custom asynchronous CFP

for *dotprod*. The graph plots performance using several branch prediction rates and branch resolution stage placements. The prediction rates were varied from 50% accuracy (i.e., 50 of 100 branches were predicted correctly) to 99% accuracy and the position of branch resolution was varied from the first pipeline stage to the last pipeline stage. The data in the figure was collected using a custom counterflow pipeline and instruction schedule for *dotprod* determined by our design methodology. The instruction schedule was not changed based on the position of branch resolution.

The figure verifies the intuitive notion that prediction accuracy must increase as pipeline length increases to attain good performance. The figure also shows that performance levels off at branch position 5. This is the point at which overspeculating instructions begins to impact performance. It is also the position that places the branch resolution stage next to the stage that determines the loop exit condition (i.e., a comparison stage). This is typically the best position for branch resolution since there is no need to speculatively execute instructions beyond the point at which branch outcomes are known.

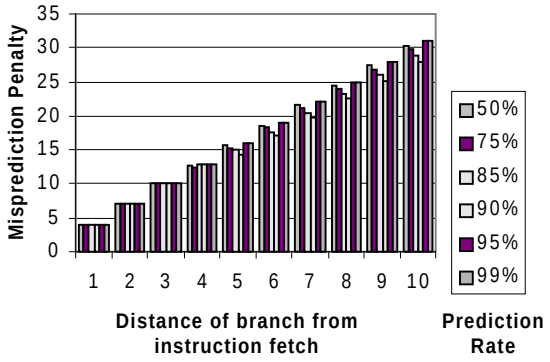

Figure 5: Branch misprediction penalty *dot-product* with different branch resolution stage positions.

The graph in Figure 5 shows the average branch misprediction penalty for *dotprod*. For counterflow pipelines, the misprediction penalty is the time from when a mispredicted branch is resolved until the instruction fetch stage begins fetching from the correct branch target.

As expected, the graph shows that the misprediction penalty increases as the distance between instruction fetch and branch resolution increases. However, the branch prediction penalty differs at several branch positions depending on prediction accuracy (e.g., positions 5 through 10). For example, when branch predictions are resolved at position 8, the misprediction penalty varies according to prediction rate. In a traditional microprocessor organization, the misprediction penalty is static and would be the same regardless of prediction accuracy (e.g., the cluster of bars for position 8 would have the same values).

The misprediction penalty varies dynamically in a CFP according to prediction accuracy because the penalty is sensitive to activity in the pipeline. The reason for the misprediction penalty variance is that a branch's target instruction blocks have different instructions, which affect the pipeline differently. For example, suppose one branch target has an instruction that stalls in the pipeline during a garner operation. This blocks results, including poison pills, from flowing through the stalled pipeline stage until the garner operation completes. However, the opposite branch target may not have this behavior. In this case, results would flow directly through the pipeline.

Although the pipeline for *dotprod* is long (12 stages), the misprediction penalties in Figure 5 are small enough that speculative execution is effective. This has also proven true for other benchmarks. For example, *matmult* has 26 stages and the branch misprediction penalty does not impact performance so significantly that the position of branch resolution is tightly constrained (i.e, it does not have to be placed closely to instruction fetch to achieve reasonable misprediction penalties). Indeed, like *dotprod*, the best position for branch resolution in *matmult* is near comparison operations.

Figures 4 and 5 demonstrate that speculative execution in counterflow pipelines is one effective way of achieving high performance in an application without additional hardware such as history buffers and complex control mechanisms.

3.3. Out-of-Order Execution

The counterflow pipeline uses out-of-order execution to tolerate high-latency operations, and as an example of this, we consider memory accesses in this section.

In our custom CFPs, memory accesses are launched early in the pipeline into an attached memory siding. This moves memory accesses out of the main pipeline so that subsequent instructions can continue to flow through the pipeline to a stage where they may execute. This serves the same purpose as an instruction re-order buffer, allowing independent instructions to begin executing before a memory access completes.

Figure 6 demonstrates how the counterflow pipeline tolerates increasing memory latency for five benchmarks. In this experiment, a custom pipeline was generated for each benchmark using our customization methodology. The initial pipeline for each benchmark has a non-pipelined memory siding and a latency of 5. We assume that the memory siding has no data cache (i.e., all memory accesses cause a cache miss). In this experiment, we pipeline the memory siding and vary the pipeline depth from 2 to 10 stages. Thus, memory

access latency varies from 10 to 50 time units. The instruction schedule and main pipeline configuration are not changed in this experiment; only the memory siding pipeline depth is changed.

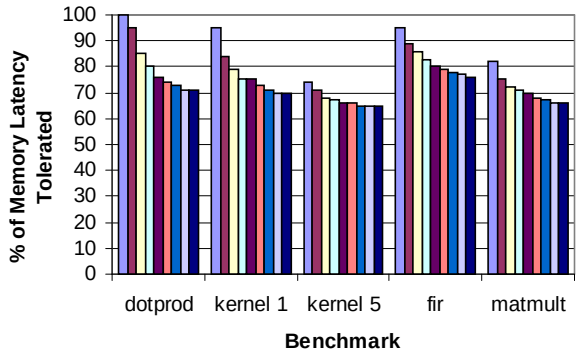

Figure 6: Percentage of memory latency tolerated when increasing memory pipeline siding depth. For each benchmark, the columns vary left to right from a siding depth of 2 to a depth of 10.

The graph in Figure 6 shows the percentage of memory latency tolerated by the custom pipelines for the five benchmarks and 9 different memory siding pipeline depths (the columns in the figure are arranged left to right with a depth of 2 to 10 for each benchmark). The percentage of latency tolerated is the amount of total memory latency that is hidden by the application. The percentage is calculated using the equation:

$$1 - \frac{depth \times latency \times accesses}{observed - baseline}$$

The term *depth* is the length of the memory pipeline siding, *latency* is the stage latency of a memory pipeline stage, and *accesses* is the total number of dynamic memory accesses. The term *observed* is the execution latency for a particular benchmark run and *baseline* is the execution latency for each benchmark's initial pipeline configuration. The equation calculates memory latency tolerance by a particular pipeline when *depth* is varied.

Figure 6 shows that a large portion of memory latency is tolerated for the benchmarks. The high tolerance is due to the memory siding moving memory accesses out of the main pipeline. This allows subsequent instructions to be inserted into the pipeline and begin execution. The memory latency is also partly hidden by the increase in the number of memory accesses that can be "in-flight" in the memory siding. As siding pipeline depth is increased, a siding can accommodate more accesses, which reduces resource contention for the siding.

The percentage of latency toleration in Figure 6 decreases as memory latency increases because there is not enough instruction parallelism in the loops to cover the change in latency. To increase instruction-level parallelism, program transformations such as software pipelining, if-conversion, etc. [20] could be applied and the resulting instruction dependency graph could be used as the basis for pipeline customization.

The experiment in Figure 6 changes only memory latency by increasing the siding's pipeline depth. We do not re-schedule or re-customize a benchmark's instruction dependency graph or pipeline. Although this is not important for these benchmarks because they are small, it may be profitable to change the instruction schedule and pipeline configuration for larger benchmarks that have many memory accesses per iteration.

The pipeline configurations used in Figure 6 have separate stages for initiating load and store instructions: one stage handles loads and another handles stores. This permits customizing pipeline stage order to the relative position of loads and stores in the instruction dependency graph. In most graphs, load operations occur early in the graph and store operations occur late and are typically dependent on significant computation. By separating memory operations into distinct stages, loads can be launched early in the pipeline and stores can be launched late after they have garnered their source operands. In a pipeline configuration that combines load and store launch stages into a single stage, stores may stall waiting for their source operands early in the pipeline. This can degrade performance because instructions following the store can not reach their execution stage.

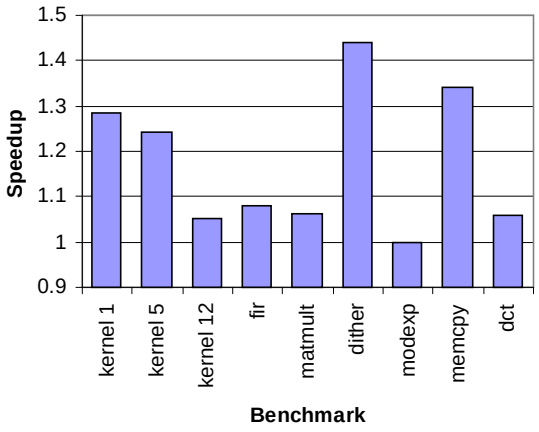

Figure 7: Benchmark speedup for separate load and store stages versus a combined memory launch stage.

Figure 7 shows the speedup obtained from separate load and store launch stages over a combined load and store launch stage (*dotprod* is not shown because it does not have a store instruction in its loop). The *kernel 1*,

kernel 5, *dither*, and *memcpy* benchmarks show a significant improvement in performance. For *fir*, *matmult*, *kernel 12*, *modexp*, and *dct*, it appears that the store instruction in the combined stage arrangement acquires its source operand at about the same point as it does in the separate load and store stage arrangement. This means there is little advantage to having a separate stage for a store.

Separate load and store launch stages creates a problem for memory addresses that alias the same location. The hardware must ensure that a load aliasing a memory address written to by a store does not launch before the store completes. In our current system, if we identify that there are no memory aliases between loads and stores in a kernel loop, we use separate load and store stages and when we can not determine that there are no memory aliases, we use a single stage to launch loads and stores. However, even in the presence of aliases, it is possible with the aid of a store address buffer to have separate load and store stages.

Based on the experiments in this section, we conclude that functional sidings are an effective means for overlapping the execution of high latency operations with other processing in the pipeline. Indeed, this type of out-of-order execution is especially attractive because it does not require hardware structures such as instruction re-order buffers and register rename tables.

3.4. Asynchronous Custom Pipelines

Counterflow pipelines can be implemented as synchronous and asynchronous microarchitectures. An asynchronous implementation has the advantage that micro-operations can be separated into several lightweight functions. For example, a synchronous CFP must be able to garner source operands and execute an instruction in a single cycle. This has the disadvantage that an instruction will see the worst case cycle time regardless of whether an instruction only garners a source operand. In an asynchronous implementation, micro-operations can be separated into distinct phases and an instruction can be advanced out of a pipeline stage as soon as possible. For example, if an instruction only needs to pass through a pipeline stage, it can proceed directly through the stage very quickly. In a synchronous design, it would take a full cycle for the instruction to move through the stage.

Figure 8 shows the speedup of custom asynchronous counterflow pipelines over a general-purpose synchronous pipeline. The general-purpose pipeline has functional sidings for integer operations, memory operations, and multiplication. The custom pipelines are tailored to the data dependency graph using our customization technique.

The figure indicates that asynchronous custom pipelines achieve a speedup of up to 6 times over a synchronous general-purpose pipeline. The speedup can be attributed to three reasons. First, the custom pipelines are tailored to the resource requirements of the graphs, which eliminates resource contention. Second, the custom pipelines have their stages arranged to minimize the latency of conveying source operands. Finally, the asynchronous pipelines achieve average case execution time.

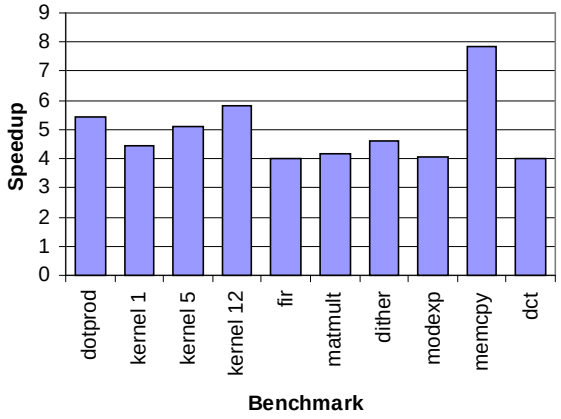

Figure 8: Speedup of custom asynchronous CFPs over a general-purpose synchronous CFP.

The difference in performance between the asynchronous custom pipelines and the synchronous general-purpose pipeline comes partly from the difference in the way they handle pipeline operations. The synchronous pipeline takes a full cycle to complete all operations needed by an instruction in a pipeline stage regardless of whether an instruction only needs part of a cycle. For example in a synchronous CFP implementation, if an instruction only garners a source operand in a pipeline stage, it is held in the stage for the full cycle. In an asynchronous pipeline, the instruction would be allowed to proceed as soon as the garner operation completes.

For our simulations, garnering a source operand takes 3 time units and executing an instruction takes 5 time units. This implies that the clock cycle length in the synchronous pipeline is 8 time units. Thus, a garner operation in the asynchronous pipeline takes 3/8 of the time that a synchronous CFP takes. Asynchronous custom pipelines take advantage of micro-operation parallelism to improve performance.

Asynchronous CFP implementations are able to exploit micro-operation parallelism because they have a smaller operation granularity than synchronous pipelines. The counterflow pipeline has four types of micro-operations: *compare*, *launch*, *return*, and *execute*. The *compare* operation corresponds to garnering source operands, updating or killing destination registers, and handling poison pills. The *launch* and *return* operations

correspond to initiating a pipeline siding operation and returning a result from a siding. The *execute* operation corresponds to executing an instruction (whether in a siding or in a pipeline stage). For example, an instruction that executes in a pipeline siding does all four micro-operations. It first garners its source operands and then launches, executes, and returns a result.

We have also experimented with asynchronous general-purpose counterflow pipelines. Although we do not report the results here, the asynchronous custom pipelines achieve speedups of up to 2 times over the general-purpose asynchronous architecture. The combination of asynchronous CFP implementations and pipeline customization leads to a significant performance improvement over synchronous pipelines.

The experiments in this section demonstrate that custom asynchronous counterflow pipelines achieve higher performance than synchronous pipelines. This performance improvement is very impressive considering that counterflow pipelines can be customized simply and quickly to the resource and data flow requirements of a kernel loop. Furthermore, the advantages of asynchronous processors, such as low power consumption, make custom high-performance CFP's an attractive microarchitecture for embedded systems.

4. Related Work

In the last ten years, asynchronous microprocessors have gained much attention because of their promise for design ease, high performance, and low cost. There have been several asynchronous microprocessor proposals, including a design from the California Institute of Technology [26], a decoupled access-execute microarchitecture from the University of Utah [22], and a low-power implementation of the ARM architecture from Manchester University [7, 8].

Although the counterflow pipeline was proposed as an asynchronous organization for general-purpose microprocessors [25], there has also been a proposal for synchronous version [19]. However, this work adds significant hardware structures to the original design to get good performance on a wide variety of applications. In our work, we customize counterflow pipelines to a single application to get high performance without introducing new microarchitecture enhancements.

There has also been much interest in automated design of application-specific integrated processors (ASIPs) because of the increasing importance of high-performance and quick turn-around in the embedded systems market. ASIP techniques typically address two broad problems: instruction set and microarchitecture synthesis. Instruction set synthesis attempts to discover micro-operations in a program (or set of programs) that can be combined to create instructions [13, 14]. The synthesized instruction set is optimized to meet design goals such as minimum program size and execution latency. Microarchitecture synthesis derives a microprocessor implementation from an application (or set of applications). Many microarchitecture synthesis systems use a co-processor strategy to synthesize custom logic for a portion of an application and to integrate the custom hardware with an embedded processor core [4, 11, 21]. Another microarchitecture synthesis approach tailors a single processor to the resource requirements of the target application [3, 6]. Although instruction set and microarchitecture synthesis can be treated independently, many co-design systems attempt to unify them into a single framework [9].

Our current research focus is microarchitecture synthesis. We do not presently synthesize an instruction set for an embedded application. Instead, we customize a counterflow pipeline microarchitecture to an application using a standard RISC instruction set and information about the data flow of the target application. Our microarchitecture synthesis technique has the advantage that the design space is well defined (although potentially very large), making it easier to derive custom pipeline configurations that meet design goals.

5. Summary

The experiments in this paper demonstrate that counterflow pipelines are well-suited for automatic design of application-specific microprocessors. In the paper, we describe why counterflow pipelines are an ideal architecture for custom microprocessors, and we present an effective and simple approach for customizing counterflow pipelines to an application. We also show that counterflow pipelines handle speculative and out-of-order execution in a low-cost and elegant way that allows custom CFP's to tolerate control dependences and high-latency operations such as memory accesses. Finally, this work demonstrates how asynchronous counterflow pipeline implementations can lead to high performance. We are continuing to research the performance potential of custom CFP's, including microarchitecture 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 Symp. on Programming Language Design and Implementation*, pp. 329–338, Atlanta, Georgia, June 1988.

[2] Withheld for blind review.

[3] Corporaal, H. and Hoogerbrugge J., “Cosynthesis with the MOVE framework”, *Symp. on Modelling, Analysis, and Simulation, CESA ‘96*, pp. 184–189, Lille, France, July 1996.

[4] Ebeling C., Cronquiest D. C., Franklin P., et al., “Mapping applications to the RaPiD configurable architecture”, *IEEE 5th Annual Symp. on Field-Programmable Custom Computing Machines*, pp. 106-115, Napa Valley, California, April 16–18, 1997.

[5] Edmondson J. H., Rubinfeld P. I., Bannon P. J., et al., “Internal organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC microprocessor”, *Digital Technical Journal*, pp. 119–135, Vol. 7., No. 1, 1995.

[6] Fisher J. A., Faraboschi P., and Desoli G., “Custom-fit processors: Letting applications define architectures”, Technical Report HPL–96–144, Hewlett-Packard Laboratories, Palo Alto, California, 1996.

[7] Furber S. B., Day P., Garside J. D., Paver N. C., and Woods J. V., ‘‘AMULET1: A micropipelined ARM’’, *Proceedings of the IEEE Computer Conference*, March 1994.

[8] Furber S. B., Garside J. D., Temple S. et al., “AMULET2e: an asynchronous embedded controller”, *Int’l Conf. on Adv. Research in Asynchronous Circuits and Systems (Async97)*, pp. 290–299, Eindhoven, The Netherlands, April 1997.

[9] Gupta R. K., and Micheli G., “Hardware-software co-synthesis for digital systems”, *IEEE Design and Test of Computers*, Vol. 10, No. 3, pp. 29–41, Sept. 1993.

[10] Hauck S., ‘‘Asynchronous design methodologies: An Overview’’, *Proc. of the IEEE*, Vol. 83, No. 1, January 1995, pp. 69–93.

[11] Hauser J. R., and Wawrzynek J., “Garp: A MIPS processor with a reconfigurable coprocessor”, *IEEE 5th Annual Symp. on Field-Programmable Custom Computing Machines*, pp. 12–21, Napa Valley, California, April 16-18, 1997.

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

[13] Holmer B., *Automatic Design of Instruction Sets*, Ph.D. thesis, University of California, Berkeley, 1993.

[14] Huang I-J and Despain A. M., “Synthesis of application specific instruction sets”, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol 14, No. 6, pp. 663–675, June 1995.

[15] Hunt D., “Advanced performance features of the 64-bit PA-8000”, *COMPCON ‘95 Digest of Papers*, pp. 123–128, March 1995.

[16] Hunt D. and Lesartre G., “PA-8500: The continuing evolution of the PA-8000”, *COMPCON ‘97*.

[17] Jank K. J., Lu S-L, Miller M. F., “Advances of the counterflow pipeline microarchitecture”, *Proc. 3rd Int’l Symp. on High-Performance Computer Architecture*, pp. 230–236.

[18] Johnson M., *Superscalar Microprocessor Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[19] Miller M. F., Janik K. J., and Lu S-L, “Non-stalling counterflow architecture”, *Proc. 4th Int’l Symp. on High-Performance Computer Architecture*, pp. 334–41,

[20] 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.

[21] Razdan R. and Smith M. D., “A high-performance microarchitecture with hardware-programmable functional units”, *Proc. of 27th Annual Int’l Symp. on Microarchitecture*, pp. 172–180, Dec.1994, San Jose, California.

[22] Richardson W. and Brunvand E., "Fred: A Decoupled Self-Timed Computer," *Int’l Conf. on Adv. Research in Asynchronous Circuits and Systems (Async96)*, pp. 60–68, Aizu, Japan, March 1996.

[23] Leibholz D. and Razdan R., “The Alpha 21264: A 500 MHz out-of-order execution microprocessor”, *42nd IEEE Computer Society Int’l Conf. Proc. (COMPCON Spring 97)*, pp. 28–36, 1997.

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

[25] 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.

[26] Tierno J., Martin A. J., Borkovic D., and Lee T. K., “A 100-MIPS GaAs asynchronous microprocessor”, *IEEE Design and Test of Computers*, Vol 11., No 2., pp. 43–49, 1994.