Architectural Considerations for Application-Specific Counterflow Pipelines

Bruce R. Childers, Jack W. Davidson
University of Virginia, Department of Computer Science
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., digital cameras, 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 evaluates CFP design alternatives and 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 evaluate CFP design trade-offs and show that CFP's provide speculative and out-of-order execution, and register renaming that is matched to an application. Fourth, we show that asynchronous counterflow pipelines achieve high-performance by reducing the average execution latency of instructions over synchronous implementations. Finally, we demonstrate that custom CFP's achieve cycles per instruction measurements that are competitive with 4-way superscalar out-of-order processors at a potentially low design complexity.

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. An innovative computer organization called the *Counterflow Pipeline* (CFP), proposed by Sproull, Sutherland, and Molnar [27], has several characteristics that make it an ideal target organization for the synthesis of application-specific 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 instruction-level parallel (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 [17] 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 other structures (e.g., register rename buff-

ers, 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 most importantly for this study, they complicate microarchitecture design. In contrast, the counterflow pipeline simply and naturally implements out-of-order and speculative execution without requiring hardware structures that require extensive study of design trade-offs. The combination of these features allows the counterflow pipeline to achieve high performance at a potentially lower design cost and quicker time to market 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 asychronous or a *double-clocked* (i.e., a synchronous design where a pipeline stage takes multiple clock cycles to do an instruction sub-operation) microarchitecture. In this paper, we show that this improves performance because average instruction latency (due to finer operation granularity) is reduced versus synchronous CFP's.

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 and the third section describes disadvantages of counterflow pipelines. The fourth 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 final section discusses 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.

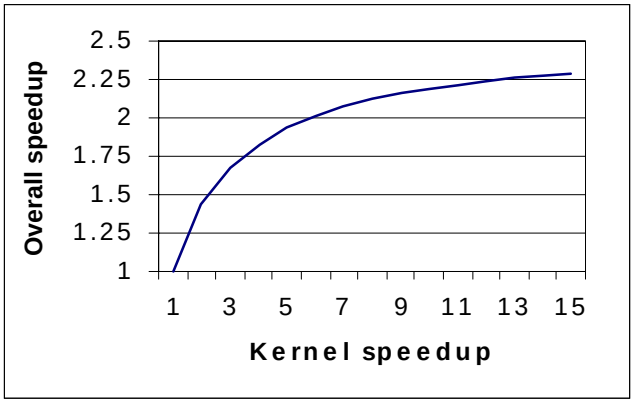

Figure 1: Overall speed-up for JPEG

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 approximately 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 [14] 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 possible.

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 networks.

There are two possible system architectures for a custom CFP. The first scheme has two processors: one for executing the control portions of an application and the other, a counterflow pipeline, for executing the kernel loop of an application. This scheme has the advantage that the kernel processor can be highly optimized for executing the kernel loop to get optimal performance. However, it has the disadvantage that support is needed to integrate the control and CFP processors (e.g., interface logic, handling of live-in/out data, etc.) Such an architecture may also have a high area cost. The second system architecture has a single CFP for executing both control and kernel code; this architecture has a potentially lower cost than a co-processor architecture. However, care must be taken to avoid slowing down the control code when tailoring a CFP to the kernel loop—the custom pipeline should have the functionality needed by *both* the kernel and control code. Although we are currently evaluating both system architectures, we assume the first scheme in this paper and generate a CFP optimized to an application's kernel loop.

1.2: Counterflow pipelin

This section presents a brief overview of counterflow pipelines. Sproull, Sutherland, and Molnar give a more detailed description of CFP's [27]. 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 pipeline 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. For example, load instructions are fetched and issued into the pipeline at the `instr_fetch` stage. This stage decodes instructions and determines which pipeline stage executes an instruction. 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.

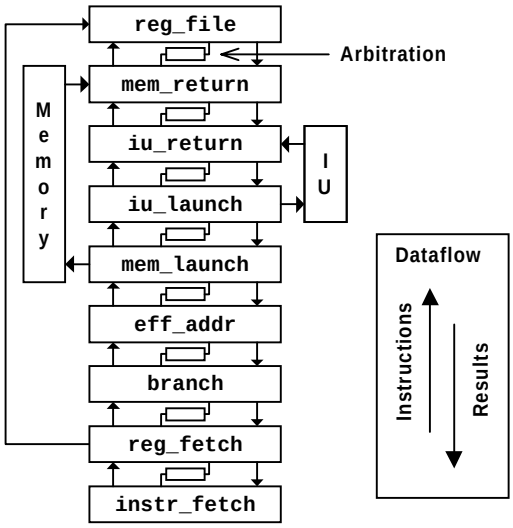

Figure 2: An example counterflow pipeline

The interaction between instruction and result bundles are governed by rules that ensure sequential execution semantics. There are four rules that control instructions: 1) instructions must stay in sequential order; 2) an instruction must acquire its source operands prior to executing (called *garnering*); 3) an instruction inserts a copy of its destination register in the result pipeline when it finishes executing (called *updating*); and 4) an unexecuted instruction may not move past the last stage capable of executing it.

There are three other rules that ensure result values are current for their position in the pipeline and not values from previous operations with the same name. The rules are: 1) an instruction copies a result's value if a result register name matches one of its source operands; 2) a result register matching an unexecuted instruction's destination register is invalidated; and 3) a result register matching an executed instruction's destination is updated with the destination's value.

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 [18, 26]. 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 branch resolution stage and instruction fetch stage.

2.2: Out-of-order execution

An important issue for instruction-level parallel micro-architectures 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 [18, 26].

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 [14, 20]. 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 orga-

nization 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 [12]. 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 [19, 22, 27], our research focuses on how to construct custom ILP-processors for embedded systems.

3: CFP design disadvantages

To understand the trade-offs associated with counterflow pipelines, it is useful to examine some of the disadvantages of these structures. The first disadvantage is that the CFP requires arbitration between adjacent pipeline stages. Because the arbiter controls the advancement of results and instructions in the pipeline, it should be made as fast as possible. In practice, it has proven difficult to build fast (and correct) CFP arbiters and control circuits because of race conditions, circuit hazards, and handshaking, although some designs have been proposed [21].

A second disadvantage is that enforcing the pipeline matching rules may be expensive. The matching rules require examining an instruction's local state (e.g., whether it has been killed, launched, executed, etc.) and comparing an instruction's operands to a result bundle (the action associated with the comparison outcome also must be done.) Enforcing the matching rules can become a performance bottleneck because it affects the speed at which results are sent to their consumer instructions. However, careful instruction scheduling by a compiler and arrangement of pipeline stages can reduce the performance impact of the matching rules.

A final disadvantage is that CFP's may use more chip area than traditional architectures. This is especially true for asynchronous designs because control signals are needed to synchronize computational elements (i.e., request and acknowledge signals) and extra logic is needed to ensure glitch-free circuits [5]. Also contributing to CFP chip area is the width of pipeline registers. These registers are very wide since they hold instantiated instruction and result bundles in each pipeline stage. An approximation for the width of an instruction bundle register is:

$$(n_{srcs} + n_{dests}) \times (w_{reg} + w_{spec} + w_{rflags}) + w_{op} + w_{flags}$$

where n_{srcs} is the number of source operands (e.g. 2), n_{dests} is the number of destination operands (e.g., 1), w_{reg} is the width of a register value (e.g., 32), w_{spec} is the width of a register name (e.g., 5), w_{rflags} is the width of result flags (e.g., 2 for valid and garnered bits), w_{op} is the width of a decoded opcode (e.g., approximately 10 for a 12 stage pipeline), and w_{flags} is the width of status flags (e.g., 2 for instruction killed and executed bits.) As an example, a 32-bit triadic instruction requires a bundle width of:

$$(n_{srcs} + n_{dests}) \times (w_{reg} + w_{spec}) = 3 \times (32 + 5) = 111 \text{ bits}$$

for just its source and destination operands. An approximation for the width of result registers is:

$$n_{result} \times (w_{reg} + w_{spec} + w_{rflags})$$

where n_{result} is the number of individual results in a result bundle. The minimum value for n_{result} is n_{dests} and a reasonable value is n_{srcs}, which allows a single result bundle to carry all the sources necessary for an instruction.

High-performance CFP's need two instruction and result registers per pipeline stage to ensure maximum throughput [27], doubling the cost of pipeline registers. The point-to-point connections between pipeline stages are also very wide since they transmit instantiated instructions and results between pipeline registers. Finally, multiple comparators are needed in each pipeline stage to enforce the matching rules. Although each of these comparators is small (their size is w_{spec}), a CFP design can have:

$$(n_{srcs} + n_{dests}) \times n_{result}$$

comparators per stage. For triadic instruction sets with $w_{spec} = 5$ and $n_{result} = 2$, this is approximately equivalent to one 32-bit comparator per stage ($3 \times 2 \times 5 = 30$ bits.) Although the area cost of CFP's is high, this concern is becoming less important with the advent of 100+ million transistor chips. Indeed, time to market is the most important issue in today's embedded systems. Thus, an architecture (such as the CFP) that is well suited for quick turn-around design is very attractive despite a potentially high area cost.

4: Experimental result

In this section we show how to construct counterflow pipelines automatically 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.

4.1: Methodology

The goal of our present work is to see how far the counterflow pipeline in its original form can be pushed to get good performance in an application-specific setting. To that end, the experiments in this paper use CFP's customized to the resource and data flow requirements of benchmark applications. The customization process operates at the architectural-level on pre-designed functional devices such as pipeline stages, register files, and functional sidings.

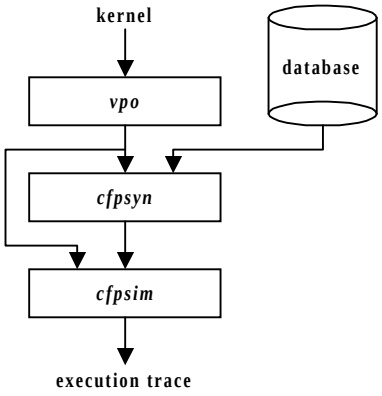

Figure 3: The workflow of the customization process. The kernel loop is optimized by *vpo* to serve as a specification for a custom counterflow pipeline determined by *cfpsyn*. The custom CFP is simulated and analyzed by *cfpsim*.

The design space of counterflow pipelines is defined by processor functionality and topology. Processor functionality is the type and number of computational elements in a pipeline and topology is the interconnection of those elements. In our work, processor functionality is charac-

terized by an user-supplied database of computational elements that indicates device type (siding or stage) and semantics (as instruction opcodes) for each database entry.

CFP topology is determined by the order of pipeline stages because CFP functional devices are interconnected via stages. Thus, given what functional devices a CFP contains, the topology space is all combinations of pipeline stages, excluding some combinations that do not make sense (e.g., placing a siding's return stage before its launch stage.)

Figure 3 is a diagram of the customization process. The customization system accepts a kernel loop (in C) as an input to the code improve *vpo* [1], which transforms the loop using classic optimizations such as strength reduction, induction variable elimination, global register allocation, loop invariant code motion, etc. The optimized instructions are the input to the synthesis phase *cfpsyn*, which selects and instantiates computational devices from the design database for each kernel loop operation. It also determines the processor interconnection network. The synthesis step emits a description of the custom pipeline that is used by the counterflow pipeline simulator, *cfpsim*, to collect performance statistics and a program execution trace.

Customization technique: Although we have studied search-based pipeline customization techniques [3], we use an 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 [20] 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. Low latency operations are assigned unique pipeline stages so that they may possibly execute independently of one another (assuming no dependences between two instructions.) Although we do not currently cast multiple low latency operations onto a single stage (e.g., a chain of two additions could use the same stage), we intend to address cost reduction in the future. 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.

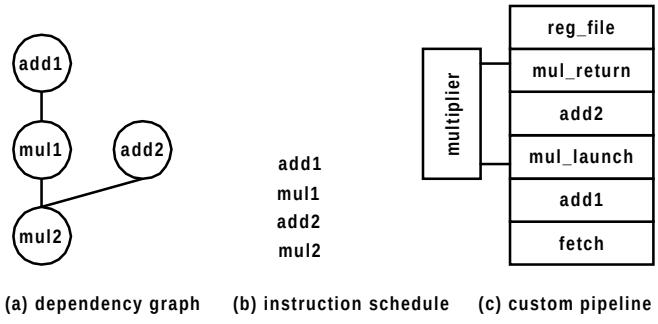

Figure 4: An example of pipeline customizing.

An example of the customization process is shown in Figure 4. The dependency graph in (a) has two addition and two multiplication instructions. The first customization step generated two addition stages and one multiplication siding.

The second customization step arranges pipeline stages. Using path latency as scheduling pri-

ority, local instruction scheduling produces 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`.

The technique described above uses the instruction schedule generated by the compiler as a specification for a counterflow pipeline organization. The compiler uses local instruction scheduling to separate the definition and use of register values based on the latency of the defining operation. In effect, this keeps producer and consumer instructions relatively close to one another, while allowing independent instructions to be scheduled in the delay between the producer and consumer

This closely matches what happens in a counterflow pipeline during program execution. An execution trace can be thought of as a *very* long counterflow pipeline—instructions march through time producing values that are consumed by following instructions; precisely the same behavior exhibited by the counterflow pipeline. This suggests that arranging pipeline stages in the order of the instruction schedule (which reduces the impact of operation latency) will give good performance. Using small dependence graphs, we have found that this technique generates pipelines that are within 7–19% of the optimal pipeline arrangement. We use the above technique because it is able to quickly and easily generate pipeline organizations for reasonably complex benchmarks. More details about the interaction of counterflow pipeline stage arrangement and instruction scheduling can be found in [3].

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. The latencies in the table are expressed relative to how long it takes an instruction or result to move between adjacent pipeline stages. These latencies were originally supplied by Sun Microsystems based on their CFP work [28]. 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.

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

Our timing assumptions do not account for overhead due to asynchronous protocol signalling (i.e., two-phase vs. four-phase signalling.) Such low-level implementation detail should not affect the architectural design conclusions we draw in this paper. Furthermore, we believe it is difficult to make timing evaluations about protocol signalling without a low-level implementa-

tion. Under our simulation assumptions, our asynchronous counterflow pipelines are equivalent to *double-clocked* synchronous CFP implementations [27], where a device takes multiple clock cycles to do a single pipeline operation (e.g., garner or kill.) Indeed, this makes our work also applicable to synchronous CFP implementations. In the future, we will investigate low-level implementation trade-offs and their impact on architecture design.

Evaluation: 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.

4.2: 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 effect 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 5 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. We use a statistical method to evaluate branch prediction because we are interested in determining the impact of different prediction rates on the arrangement of pipeline stages. From statistical experiments, we can identify the needed prediction accuracy and select a branch prediction scheme that achieves that accuracy. The data in Figure 5 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 adjacent 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 become known.

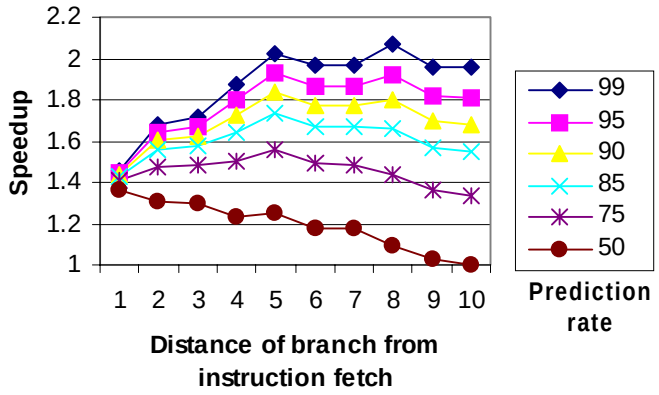

Figure 5: Speedup for *dotprod* using different branch stage positions and prediction rates.

The graph in Figure 6 shows the average branch misprediction penalty fo *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.)

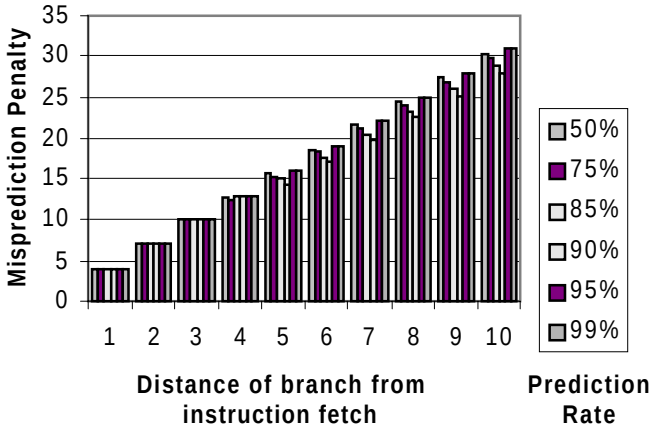

Figure 6: Branch misprediction penalty *dot-product* with different branch resolution stage positions.

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 fo *dotprod* is long (12 stages), the misprediction penalties in Figure 6 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 near instruction fetch to achieve reasonable misprediction penalties.) Indeed, like *dotprod*, the best position for branch resolution in *matmult* is near comparison operations.

Figure 7 shows an alternative CFP branch architecture that reduces the misprediction penalty for pipelines with late branch resolution. This architecture uses a bypass network to forward poison pills to stages that are before the branch. The figure shows two paths that are added to the basic counterflow pipeline. The ''kill wire'' signals *kill event* to a stage, indicating a branch misprediction. When a stage receives a kill event, it marks the instruction in the stage *killed* (a stage that does not have an instruction ignores the event) in a way similar to the standard scheme. The second path in the figure sends the correct target address to the instruction fetch stage on a branch misprediction. The instruction fetch stage updates the program counter and begins fetching from the new target.

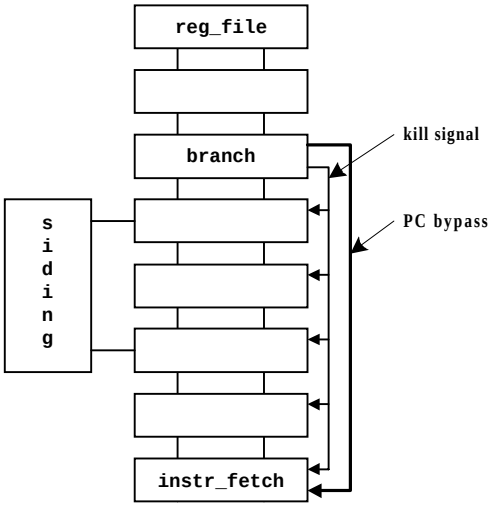

Figure 7: A CFP branch architecture with bypass paths for forwarding poison pills to reduce the branch misprediction penalty.

This alternative scheme has the potential to reduce the branch penalty dramatically over the original approach because it ''instantly signals'' a misprediction (ignoring circuit delays.) The original scheme propagates a misprediction signal over time through the use of a poison pill.

Figure 8 illustrates the effect of forwarding poison pills. The figure shows speedup for a forwarding scheme over a standard counterflow pipeline for the *kernel1* benchmark. The figure indicates that for a high misprediction rate and late branch resolution, forwarding can have a significant impact on performance. For a misprediction rate of 50% and a branch position of 11, performance is improved by 17% over a standard CFP. However, at low misprediction rates, the forwarding scheme does not significantly improve performance. For a prediction accuracy of 99%, the performance improvement is essentially non-existent, except for a branch position of 11. In this case, the improvement is less than 1%. The overall improvement, however, depends

on branch frequency. The other benchmarks show trends similar to *kernel1* with an average maximum speedup of 1.19 for a 50% misprediction accuracy and late branch resolution.

Despite the performance potential of forwarding poison pills, we believe that such a scheme is unnecessary for counterflow pipelines. Instead, it is better to implement a highly accurate branch prediction scheme such as correlating prediction, which achieves very high accuracies (especially on loops.) Furthermore, the introduction of global signals for forwarding poison pills makes processor design significantly more complex—the very problem we are trying to avoid. This may prove especially difficult for asynchronous implementations and limit the design advantages of device regularity and simplicity.

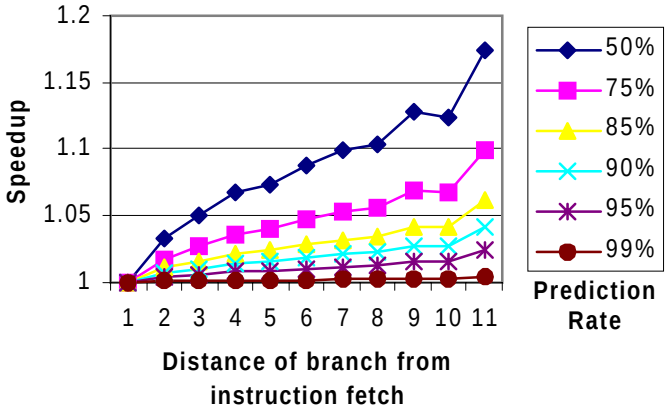

Figure 8: The speedup of poison pill forwarding versus the original CFP branch misprediction purge scheme.

Figures 5 and 6 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.

4.3: 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 9 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.

The graph in Figure 9 shows the percentage of memory latency tolerated by the custom pipelines for the five benchmarks and nine 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 9 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.

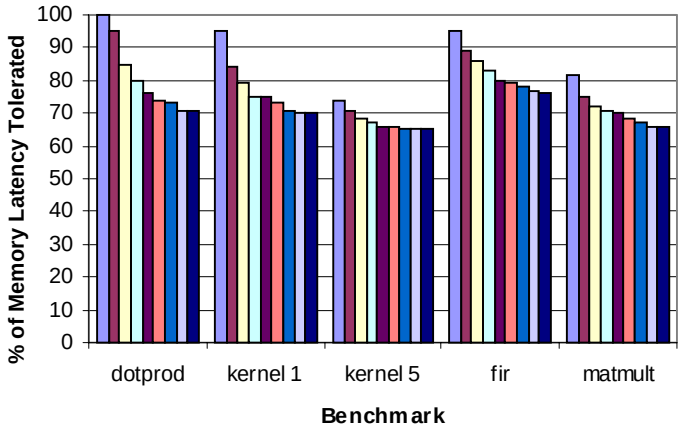

Figure 9: Percentage of memory latency tolerated when increasing memory pipeline siding depth. For each benchmark, the columns vary left to right from a siding depth of 2 to a depth of 10.

The percentage of latency toleration in Figure 9 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. [23] could be applied and the resulting instruction dependency graph could be used as the basis for pipeline customization.

The experiment in Figure 9 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 9 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 10 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.) Th *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.

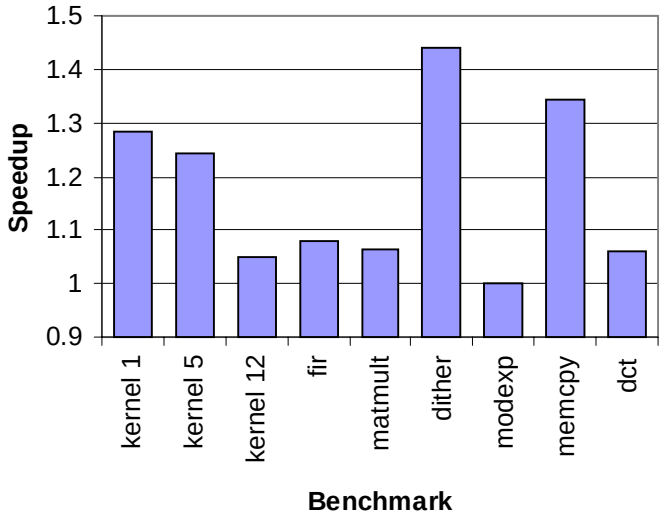

Figure 10: Benchmark speedup for separate load and store stages versus a combined memory launch stage.

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. When we can not determine that there are no memory aliases, we use a single stage to launch loads and stores.

In a single processor scheme (such as that outlined in Section 1.1), it is likely that the control code has memory aliases, which requires a unified load/store scheme. However, in a system architecture that uses a CFP as a kernel co-processor (i.e., a co-processor for executing the kernel loop), it will be easier to separate loads and stores into individual stages by transforming the kernel loop to eliminate aliases.

A final issue with a split load/store scheme is how to avoid updating memory (i.e., by a store) until branch prediction outcomes are known. There are two straightforward approaches to solving this problem. The first is the simplest: it constrains the position of the store stage to occur after the branch. This ensures that the store will be killed before committing its destination value to memory on a branch misprediction. The second scheme has a store *commit stage* at the end of the pipeline [27]. In the absence of memory aliases (i.e., having separate load and store launch stages), this is equivalent to the first approach. However, when memory aliases are present and a single store/load launch stage is included in the pipeline, a commit stage is needed to indicate

when memory can be updated. A store queue is also needed in the memory unit to check for aliases and to buffer in-flight stores until they are safe to commit (which also allows speculative loads of uncommitted addresses.) In such an approach, whenever a store passes through the commit stage, a signal is sent to the store queue to write the last queue element to memory. In the experiments presented in Figure 10, we use the first approach and constrain the store to occur after the branch stage (it is placed immediately after branch resolution.)

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.

4.4: 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 (or a double-clocked 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.

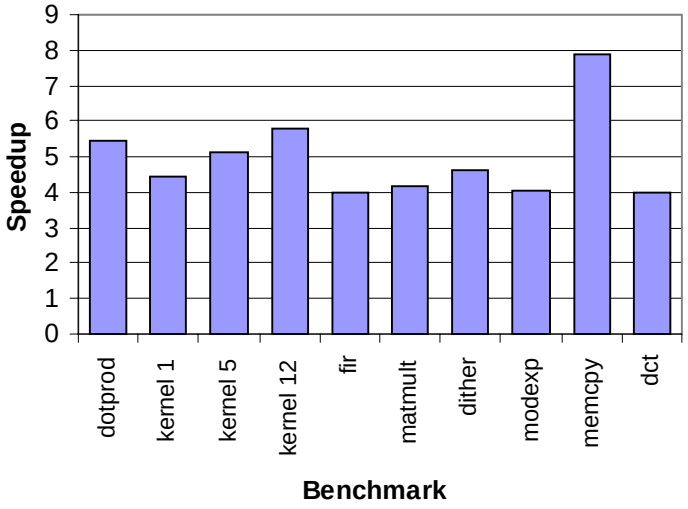

Figure 11: Speedup of custom asynchronous CFPs over a general-purpose synchronous CFP.

Figure 11 shows the speedup of custom asynchronous CFP's over a general-purpose synchronous CFP. 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 4 to nearly 8 times over a synchronous general-purpose pipeline with the average being 4.4. 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. That is, the asynchronous pipelines better overlap pipeline operations because they have a small operation granularity (as described below.) We do not currently use any techniques to shorten the actual execution latency of an operation (e.g., short-circuiting carry propogation in an asynchronous ripple-carry adder.)

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: *compar* , *launch*, *return*, and *execut* . 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

The results in Figure 11 show that custom asynchronous counterflow pipelines achieve higher performance than synchronous pipelines. This performance improvement is impressive considering how easily and quickly counterflow pipelines can be customized to the resource and data flow requirements of a kernel loop.

Although the results in Figure 11 demonstrate that custom asynchronous counterflow pipelines have good performance, it is important to compare our results to conventional processor organizations. Figure 12 shows a comparison of custom asynchronous CFP's to three traditional architectures, including a single-issue in-order processo *conv*), a 2-way superscalar out-of-order processor (*2-way*), and a 4-way superscalar processor (*4-way*). The 2-way processor has 2 integer ALU's, 1 branch unit that uses 2-bit branch prediction and a table size of 2048, and 1 memory unit. The 4-way processor is similar to the 2-way processor, except it has 4 integer ALU's and correspondingly larger issue and retirement bandwidth. The conventional architectures are based on the SimpleScalar processor from the University of Wisconsin [2].

Figure 12 plots clocks per instruction (CPI) for each benchmark. The measurements for the conventional architectures were collected using the SimpleScalar toolkit. For the counterflow pipelines, we calculated *effective CPI* (ECPI) using our tools. The formula for ECPI is:

ECPI = (latency/ECC)/instruction count.

This formula normalizes execution time by the effective clock cycle (*ECC*) length. We use an ECC length of 8 as previously discussed.

Figure 12 gives insight into the performance of counterflow pipelines relative to modern computer architectures. The figure shows that custom asynchronous counterflow pipelines have CPI's on par with modern 4-way superscalar processors. It is also our belief that simple CFP structures will lead to very fast effective clock cycle speeds, possibly faster than traditional

architectures (despite potential ECC penalties due to asynchronous implementation.) The differences in CPI may be partly influenced by the use of a different instruction set architecture and compiler for the conventional processors than what was used for the counterflow pipeline. We did not have access to a simulator to collect CPI numbers for conventional architectures based on the SPARC ISA. Although the instruction sets are different, the performance trend should not be significantly influenced by this difference (we examined the code generated for both the conventional and CFP processors to ensure there were minimal differences.)

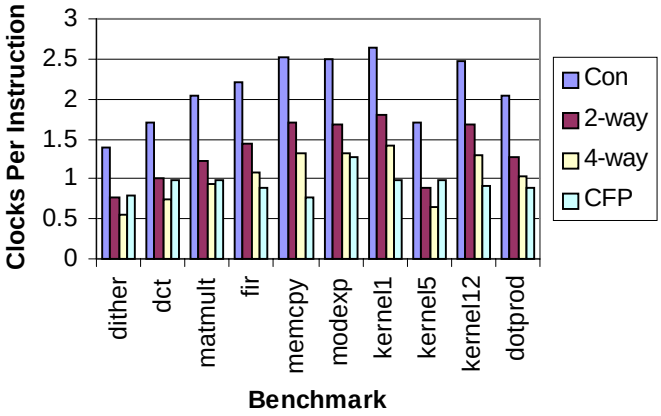

Figure 12: A comparison of the performance of conventional processors to counterflow pipelines. *Conv* is a single issue and in-order processor, *2-way* is a 2-way superscalar processor, *4-way* is a 4-way superscalar processor, and *CFP* is a custom asynchronous CFP. The figure indicates that the effective CPI of an asynchronous custom CFP is competitive with 4-way superscalar processors.

The CFP organizations in Figure 12 were derived using a simple customization process that did not require evaluating complex design trade-offs, such as simultaneous instruction decoding, issue rules, instruction re-order buffer sizes, reservation table sizes, bus network structure, etc. All of these issues introduce significant complexity to the design of modern superscalar processors. Our work demonstrates that simple application-specific counterflow pipeline structures are able to achieve equivalent performance through the use of composable high-level computational elements. Furthermore, the advantages of asynchronous processors, such as low power consumption and design composability, make custom CFP's attractive for embedded systems.

5: 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 [29], a decoupled access-execute microarchitecture from the University of Utah [25], and a low-power implementation of the ARM architecture from Manchester University [9, 10].

Although the counterflow pipeline was proposed as an asynchronous organization for general-purpose microprocessors [27], there has also been a proposal for synchronous version [22]. 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 CFP's 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 pro-

cessors (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 [15, 16]. 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 [6, 13, 24]. Another microarchitecture synthesis approach tailors a single processor to the resource requirements of the target application [4, 8]. Although instruction set and microarchitecture synthesis can be treated independently, many co-design systems unify them in a single framework [11].

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 micro-architecture 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.

6: Summary

The experimental results presented in this paper demonstrate that counterflow pipelines are well-suited for automatic design of application-specific processors. The paper describes why CFP's are an ideal architecture for custom processors, 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. This work further demonstrates how asynchronous counterflow pipeline implementations can lead to high performance. Finally, we show that custom CFP's achieve cycles per instruction measurements that are competitive with modern 4-way superscalar processors.

This paper explores the potential of counterflow pipelines for application-specific integrated processors. However, there are many unanswered questions about custom CFP's, and we are continuing to study these structures. Our work is proceeding in several directions:

- Extending the original counterflow pipeline to handle more instruction level parallelism, including a new microarchitecture proposal called ''wide counterflow pipelines,'' which are based on VLIW techniques;
- Reducing the hardware cost of custom pipelines by assigning multiple dependence graph nodes to a single functional device;
- Applying aggressive ILP compiler optimizations such as software pipelining and if-conversion to statically expose more parallelism to the synthesis system.

References

[1] M.E. Benitez and J.W. Davidson, "A portable global optimizer and linker", *SIGPLAN Notices 1988 Symp. on Programming Language Design and Implementation*, pp. 329–338, Atlanta, Georgia, June 1988.

[2] D. Burger and T.M. Austin, "The SimpleScalar tool set, version 2.0", Technical Report #1342, Computer Science Department, University of Wisconsin-Madison, June 1997.

[3] B.R. Childers and J.W. Davidson, "A design environment for counterflow pipeline synthesis", *Workshop on Languages, Compilers, and Tools for Embedded Systems* (*LCTES '98*), held in conjunction with *PLDI '98*, Montreal, Canada, June 19-20, 1998.
[4] H. Corporaal and J. Hoogerbrugge, "Cosynthesis with the MOVE framework", *Symp. on Modelling, Analysis, and Simulation, CESA '96*, pp. 184–189, Lille, France, July 1996.
[5] A. Davis and S.M. Nowick, "An introduction to asynchronous circuit design", Technical Report UUCS-97-013, University of Utah, Sept. 1997.
[6] C. Ebeling 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.
[7] J.H. Edmondson 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.
[8] J.A. Fisher et al., "Custom-fit processors: Letting applications define architectures", Technical Report HPL–96–144, Hewlett-Packard Laboratories, Palo Alto, California, 1996.
[9] S.B. Furber et al., ''AMULET1: A micropipelined ARM", *Proc. of the IEEE Computer Conf.*, March 1994.
[10] S.B. Furber 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.
[11] R.K. Gupta and G. Micheli, "Hardware-software co-synthesis for digital systems", *IEEE Design and Test of Computers*, Vol. 10, No. 3, pp. 29–41, Sept. 1993.
[12] S. Hauck, ''Asynchronous design methodologies: An Overview'', *Proc. of the IEEE*, Vol. 83, No. 1, January 1995, pp. 69–93.
[13] J.R. Hauser and J. Wawrzynek, "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.
[14] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach, 2nd edition*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.
[15] B. Holmer *Automatic Design of Instruction Sets*, Ph.D. thesis, University of California, Berkeley, 1993.
[16] I-J Huang and A.M. Despain, "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.
[17] D. Hunt, "Advanced performance features of the 64-bit PA-8000", *COMPCON '95 Digest of Papers*, pp. 123–128, March 1995.
[18] D. Hunt and G. Lesartre, "PA-8500: The continuing evolution of the PA-8000", *COMPCON '97.*
[19] K.J. Jank, S-L Lu, and M.F. Miller, "Advances of the counterflow pipeline microarchitecture", *Proc. 3rd Int'l Symp. on High-Performance Computer Architecture*, pp. 230–236.
[20] M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.
[21] W.H. Korver and I.M. Nedelchev, "Asynchronous implementation of the SCPP-A counterflow pipeline processor", *IEE Proc.—Computers and Digital Techniques*, Vol. 143, No. 5, pp. 287–294, Sept. 1996.
[22] M.F. Miller, K.J. Janik, and S.L. Lu, "Non-stalling counterflow architecture", *Proc. 4th Int'l Symp. on High-Performance Computer Architecture*, pp. 334–41, Las Vegas, Nevada, Feb. 1–4, 1998.
[23] B.R. Rau and J.A. Fisher, "Instruction-level parallel processing: History, overview, and perspective", *J. of Supercomputing*, Vol 7, pp. 9–50, May 1993.
[24] R. Razdan and M.D. Smith, "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.
[25] W. Richardson and E. Brunvand, "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.
[26] D. Leibholz and R. Razdan, "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.
[27] R.F. Sproull, I.E. Sutherland, and C.E. Molnar, "The counterflow pipeline processor architecture", *IEEE Design and Test of Computers*, pp. 48–59, Vol. 11, No. 3, Fall 1994.
[28] I.E. Sutherland, private e-mail communication, Spring 1994.
[29] J. Tierno, et al., "A 100-MIPS GaAs asynchronous microprocessor", *IEEE Design and Test of Computers*, Vol. 11, No. 2, pp. 43–49, 1994.