Non-Stalling CounterFlow Architecture

Michael F. Miller, Kennneth J. Janik, and Shih-Lien Lu:
mikem@ichips.intel.com, kjanik@icihps.intel.com, sllu@ece.orst.edu
Dept of Electrical and Computer Engineering, Oregon State University

***Abstract*-- The counterflow pipeline concept was originated by Sproull et al.[1] to demonstrate the concept of asynchronous circuits. This architecture relies on distributed decision making and localized clocking and data movement. We have taken these ideas and reformulated them into a substantially faster more scalable architecture that has the same distributed decision making and locality for clocking and data, but adds very aggressive speculation, no stalls, and other desirable characteristics. A high level Java simulator has been build to explore the design tradeoffs and evaluate performance.**

***Keywords*-- counterflow, CFPP, pipeline, dataflow, virtual register, VRP, counterdataflow, CDF, multithreading, architecture.**

I. Introduction

The current trend in microprocessors is to provide maximum speed by exploiting instruction level parallelism (ILP) both to hide long latency operations like memory accesses, and to execute multiple instructions at once. Currently the primary mechanism for doing this is an out-of-order superscalar processor. This solution is usually constructed with renaming registers, reservation stations, and reorder buffers which rely on multiple content addressable memories (CAMs) that tend to be slow, area intensive, and expensive. Such solutions also require accurate global timing and global communication between the various structures across the entire chip. This is likely to become problematic since recent technology advances provide very high clock rate logic. This will eventually result in it being physically impossible to send signals from one side of the die to the other in a single clock cycle.

Counterflow processors are able to provide a competitive alternative to the superscalar approach by using highly localized communication to resolve the scheduling issues and resolution of data dependencies. In this paper we will discuss the fundamental concepts behind counterflow pipelines, then briefly cover the development of counterflow pipelines from the first design by Sun (CFPP), to Oregon State's conversion of the architecture to the virtual register processor (VRP[1])[2] and it's powerful successor, counterdataflow (CDF). We will also discuss the present findings that suggest that CDF can be used to produce high performance processors with high parallelism, low inner-chip communication cost and good latency hiding.

II. Basic Counterflow Concepts

The basic principle of counterflow pipelines is that there are two pipelines flowing in opposite directions, as shown in a block diagram of a CFPP pipeline in Fig. 1. One of them carries instructions up from the decode unit, while the other carries data down from either the register file or earlier instructions. The key concept is that as the instructions and data pass each other, they "look" at each other. The instruction checks to see if it needs the value(s) in the data pipe (read after write hazard resolution). If it does, it copies the piece(s) of data and continues along the pipe. The data values check the instructions and mark themselves "invalid" if the instruction will update the register they will write to (write after write hazard resolution). Unlike other pipelines, the pipe stages do not perform the operations. The pipe stages merely resolve scheduling and data dependencies. To the sides of the pipelines are the execution units (called sidepanels) which execute the actual instructions. The stage that allows instructions to transition from the instruction pipe to a sidepanel is called the launch point for that sidepanel. The stage in which the instruction can transition from the sidepanel back to the instruction pipe is called the recovery point.

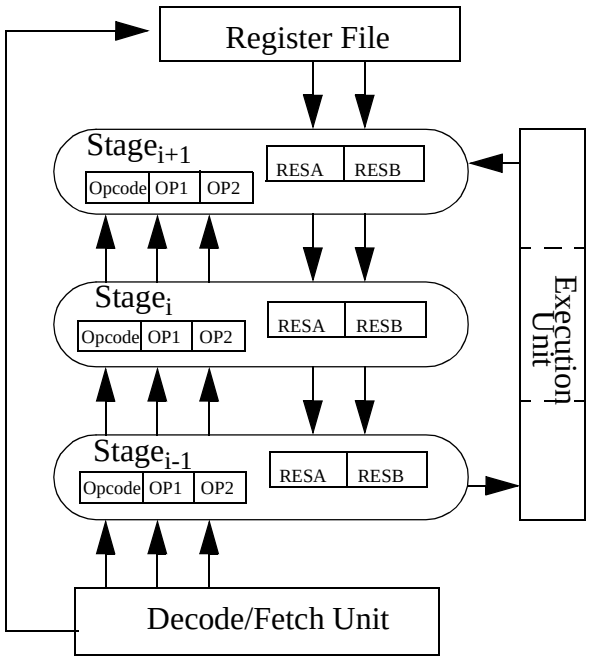

Fig. 1. Basic CFPP Architecture

When an instruction reaches a launch point where it can be executed, a check is made to see if it has all its operands. If it has all the operands, the instruction is sent into the sidepanel and executed. If not, it must either wait for the necessary operands to arrive from the data pipeline (a "last usable" sidepanel stall) or continue up the pipe if there is another sidepanel capable of executing the instruction further up in the pipeline. After an instruction has completed execution, its result is copied into the data pipeline so that subsequent instructions will receive the value as soon as possible[1],[3].

As a matter of terminology, we refer to both instructions and data in the pipeline as tokens, borrowing these terms from dataflow. An instruction token carries with it an opcode, an identifi-

1. Originally there were two VRP processors, VRP, and VRP+ in this paper we will be referring only to the latter of the designs as VRP

cation tag of some kind, and a valid bit which indicates if the instruction is a valid instruction, and a set of consumer and producer data tokens. The consumers are the data tokens that the instruction will need to execute. The producers are the data tokens that are the result of the instruction. In a similar way, the data tokens carry a value, an identification tag and a valid bit. Some architectures add additional bits to the token that are specific to that architecture's needs.

III. Original CFPP

The original counterflow architecture proposed by Sproull et.al.[1] put the Register File (RF) at the top of the pipeline and necessitated a global communication line from the decode unit to the RF to communicate which registers should be put into the result pipeline to satisfy the instructions data dependencies. Instruction tokens would remain in the instruction pipeline after executing to carry their results up to the register file. While some instructions can be executed out of order, the instruction pipe does not allow passing, so they are retired into the register file at the top of the pipeline in the same order they are issued in.

This design has three significant problems. First, instructions requiring registers that had not been used before, or recently, would be forced to travel up half of the pipeline before meeting their operands traveling down from the register file. This problem, called the half-pipe problem, results in relatively high startup cost, and adds to any branch misprediction costs. The second problem is that instructions are forced to remain in the instruction pipe until they reach the top of the pipe. This combined with the "last usable" sidepanel rule, means that if any instruction stalls, usually every instruction following must stall as well. This means that even if stalls are rare, they will have a very strong impact on performance. Finally, in order to issue more than one instruction per cycle, dependencies between the instructions being issued at the same time must be made to prevent deadlocks. The rules to prevent deadlock are relatively extensive and depend on the configuration of the pipe.

IV. Rough Draft: VRP

We originally concluded that there were two key aspects that needed to be addressed with CFPP. First, the halfpipe problem is quite significant. Second, since instructions remain in the pipeline, any stalls are detrimental to performance. Therefore, we moved the register file to the bottom of the pipeline, where instruction are issued, as shown in Fig. 2. This completely removes the halfpipe problem, and if there is a reorder buffer (needed for precise interrupts) then instructions do not need to remain in the pipeline after they have finished executing.

The addition of a reorder buffer has several pleasant effects. Instructions can now be removed from the pipeline creating more bubbles so that the stalls that do occur won't propagate as far, and also gives us perfect renaming since the tags are no longer the register tags, but are converted to be ROB entry numbers for the producers of the value. A large disadvantage that remains in VRP is that constructing an instruction pipe wider than one requires that either the instructions being issued at the same time be independent and needing different execution units, or the pipe stages must be far more complex to support moving only part of a pipe-

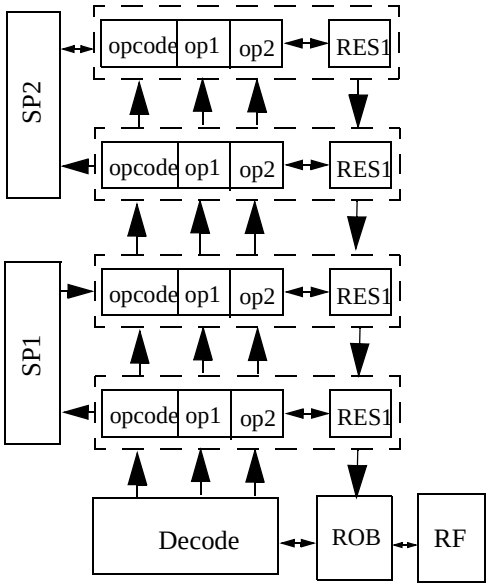

Fig. 2. Basic VRP Architecture

stage at a time.

The main problem besides the issue width limitation stalling is that while the stalls in the pipeline are less common than CFPP, the data dependencies and variable-latency instructions shift the work to the upper section of the pipe, where stalls are more common. The stalls then propagate swiftly back down the pipe and have a strong impact on performance.

V. The Next Step: CDF

The fundamental problem with the VRP processor is that the instruction pipeline is allowed to stall. To fix this problem, we wrapped the instruction pipeline back onto itself through the decode unit. This greatly simplifies the individual pipe stages because now instructions don't ever need to stall because there is always a sidepanel capable of executing them 'further down' the pipeline. Additionally, the need to check for dependencies between instructions concurrently launched is eliminated as well since there is no 'last sidepanel' stall to deadlock the pipeline.

To better understand the differences between VRP and CDF, a short example is in order. Let's say that three instructions are introduced into each of the processors, a load, and two arithmetic instructions one of which is dependent on the load. In both of the processors, the load will go into the memory sidepanel to be executed since its operands are available. With VRP, all dependent instructions on the load will fill up the pipestages behind the load recovery point. This blocking will have a ripple effect that will prevent instructions, like the independent load, that could execute from reaching sidepanels. On the other hand, with CDF, the instructions are keep moving in the pipeline (remember that it forms a circle) so that only the dependent instructions that cannot begin to execute will take up space, and independent instructions that get into the pipe are not prevented from reaching the sidepanels they require. So for our example, the load and the independent math instruction would begin execution and leave the pipeline. The dependent math instruction would circle the pipeline until the result from the load instruction was provided by

memory.

The pipeline behaves almost identically to VRP but with three important differences. First, as mentioned before, there is no concept of a ‘last sidepanel’ so there is no stalling in the instruction pipe. The instructions are merely reissued back into the bottom of the pipe if they reach the top of the instruction pipe without being executed. Second, after instructions are launched into a sidepanel, they are not required to remain in the pipe. This is important since the slots are needed to insert new instructions into the pipeline. Finally, results are recovered into the result pipe, not the instruction pipe, so that lockstepping the sidepanels to the pipeline is unnecessary, and variable-length execution is no longer a problem.

The fact that a VRP pipeline has difficulties launching multiple instructions at once is surprisingly important; if you do not allow multiple instructions to be issued each clock cycle, your instructions per cycle (IPC) can never become greater than one. This was a significant restriction for all previous counterflow-based designs. The older designs either needed complex hardware to detect and prevent potential deadlocking pairs or some kind of VLIW support. CDF requires neither, the issues of data dependencies are completely resolved in the pipeline.

In our simulations, we force all the data tokens to make at least one half complete trip around the data pipeline. While this seems like somewhat of a waste of resources since extra result pipelines will be needed, there are two compelling reasons for doing this. First, if we leave it to the ROB to fill in the consumers as instructions pass the ROB, then the ROB will be more complex, and associative searches to find the consumers will be needed (recall that the tag on the incoming data token is for the producer instruction). We consider this to be bad, since the ROB needs to be both large and fast. If the data tokens are required to make at least a half circuit, then the ROB is not required to fill in values as instruction tokens go by. The second reason is that data dependencies actually get resolved faster. Because both the instruction pipeline and the data pipeline are moving in opposite directions, their relative speed is twice that of a pipeline versus the ROB. So forcing the data tokens to go through a minimum half loop resolves the data dependencies up to twice as fast as letting it stop in the ROB.

This is implemented in a slightly unusual manner since all data tokens are required to finish their journey at the ROB. We have set up the simulation so that the data tokens that are recovered into the result pipe after the halfway point are marked as having not needing to pass the ROB, but those recovered before the halfway point are marked as needing to pass the ROB in order to make their half circuit. So the distance traveled for a token recovered in the first half of the pipe is greater than 1 length, but less than 1.5 pipe lengths. But a token recovered in the second half of the pipe travels greater than.5 lengths, but less than one length. Because recovery points usually are later in the pipe, and long latency instruction recover points like floating point and memory the latter timing is more common.

The CDF architecture, like other counterflow designs, has some significant advantages when it comes to circuits and layout. Because of the regular structure of the pipeline, it makes sense to highly optimize the structure, and then replicate as many stages as are necessary for the processor. Also, as can be seen from the diagrams, there are no giant cross-chip busses. All communication on the chip is localized. This is great for modern designs with clockrates increasing dramatically since the actual distance that can be traveled in a clock is finite. In addition, the CDF architecture is much more flexible in terms of variable-time execution. previous designs required blocking at specific pipestages if an execution unit had failed to complete some operation. With CDF, the result is entered into the pipeline whenever the sidepanel has completed the instruction. This makes it much more tolerant of memory latencies, and opens the way for using execution units that have non-fixed latencies, such as asynchronous execution units.

VI. Simulations: aBlocks

To analyze the performance of a CDF processor (and counterflow processors in general) it was necessary to construct a simulator. We felt that the previously existing simulator, VRPSim[6] was neither robust enough or well enough implemented to continue working with. We examined a number of simulators that were available at the time and reached an interesting conclusion after conversing with Todd Austin. He characterized simulation design as having three design factors: performance, flexibility, and detail as shown in Fig. 3. To optimize for one, you must give up something else in one or both of the other categories[12]. Most of the simulators available have optimized for either speed or detail. Very few put much consideration into flexibility.

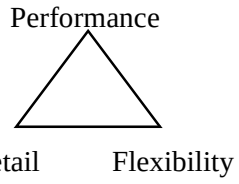

Fig. 3. Austin’s Triangle

We decided that since we needed to simulate a family of processors, and also need to make a large number of architectural tweaks and changes we should construct a simulator family that was optimized for flexibility and visualization instead of simulation performance. The simulator family is now called architecture-blocks (aBlocks) and is written in Java. The goal of the aBlocks project is not to provide a hyperacurate ultrafast processor simulator, but rather to provide an easy to use set of simulator objects that can be simply “plugged together” like Legos to simulate different processor architectures. At the time of writing, aBlocks is available only as a prerelease toolset. For up to date information about the status and availability of aBlocks please visit http://ece.orst.edu/~sllu/cfpp/java

The current instruction set (ISA) is Todd Austin’s SimpleScalar ISA[13] because it is a clean, simple ISA with good support for generating traces on multiple platforms. The simulators are currently trace-based, but the simulator components are written so that converting to an execution-based model would not be difficult. We did not write our simulator as part of the SimpleScalar simulator because our design goals of flexibility and visualization are not compatible with SimpleScalar.

We are using thirteen of the SPEC95 benchmarks as a general indicator of performance gcc, compress and swim. We are running only the first 2 million instructions through our simulator. We believe that this provides us with a reasonable indicator of overall performance for the core of the processor. Because we are not running very much of the actual program we have made some

simplifications to the memory subsystem. We are approximating steady state behavior of the level 2 data cache and the level 1 instruction cache as always containing the data requested. While this is somewhat imprecise, the extremely high hit rates over the SPEC95 benchmarks for a reasonably sized L2 data cache and L1 instruction cache[8] makes this a reasonable approximation. If we did not approximate this, our numbers would be heavily tainted by the compulsory misses that the caches would experience.

The L1 data cache has a single cycle access time, and is a 4 way set associative 16Kb cache with 32 bytes per line. The replacement policy is SLRU similar to the mechanism used in the Intel i486 [7]. The L2 cache (acting as main memory) has a constant pipelined access time of 10 cycles. The memory consistency model is relaxed so that loads can pass a limited number of stores, providing that they are accessing different locations in memory. Fetching is restricted to contiguous access only in a single cycle. If a taken branch is encountered, then fetching stops after the taken branch and resumes on the next cycle.

The branch prediction is simulated by randomly predicting incorrectly 5% of the time. While this is imprecise, we believe that it offers a reasonable approximation to an aggressive speculation mechanism running in steady-state. When a misprediction occurs, the branch prediction unit queues up the instructions that are issued speculatively, and when it the simulation “finds out” that the branch was wrong, there is a slight delay, and the queued instructions are reissued. While this is not as precise as traversing the incorrect branch path, we believe that it is sufficiently accurate to measure the effects of misspeculation pollution in the core of the processor.

Of course all of these configuration choices can be modified in the simulator. For example, there are several prediction mechanisms and cache replacement policies, but for the purposes of this paper, we are holding these choices constant so that comparing the architectures will be easier.

VII. Simulation Results for CDF

For describing the performance of the CDF processors, we have chosen to show five configurations that are very similar, but yet yield substantially different results. The number, type, and placement of the sidepanels is held constant, but we vary the width of the instruction and result pipes as well as the number of entries in the ROB. The first two, have one instruction pipe, 32 entries in the ROB, CDF0 has one result pipe, CDF1 has two. The next two have two instruction pipes and 64 ROB entries. CDF2 has three result pipes and CDF3 has four. The final configuration has four instruction pipes, eight result pipes and 128 entries in the ROB. This final configuration (CDF4) is intended simply to show how scaling the width affects the performance and behavior. These values are summarized in Table 1 below.

Table 1: CDF Pipe configurations

Name	Inst Pipes	Result Pipes	ROB Entries
CDF0	1	1	32
CDF1	1	2	32

Table 1: CDF Pipe configurations

Name	Inst Pipes	Result Pipes	ROB Entries
CDF2	2	3	64
CDF3	2	4	64
CDF4	4	8	128

The sidepanel configuration as shown in Fig. 4 was designed to be conservative in the amount of hardware used. The only functional units that are replicated are the single-cycle integer units. The pipeline is kept relatively short as well. More aggressive designs would include a second set of floating point units, a second or third branch execution unit, several multicycle integer units, and many single-cycle integer units. The pipeline would probably be extended as well to allow for more instructions to be active in the pipeline as well...

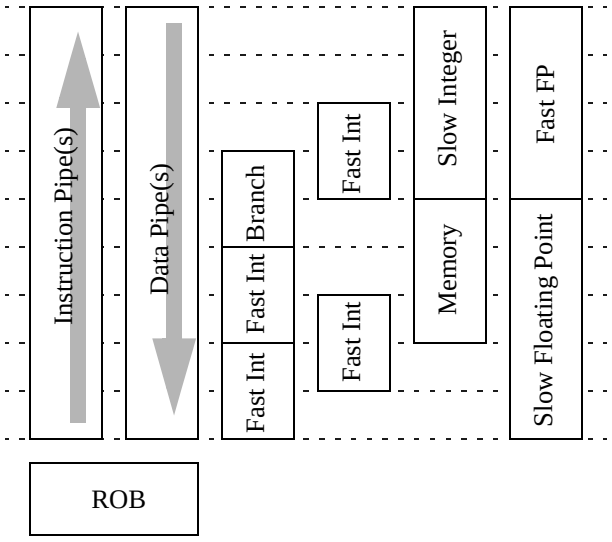

Fig. 4. Layout of sidepanels

The diagram of the sidepanel placement shows the 9 stages (separated along the dotted lines) with the instruction pipe headed away from the ROB. At the top of the pipeline, the instruction pipe “reconnects” to form a circle. The data pipe also forms a circle, but the flow direction is reversed. The sidepanels, labeled by functionality are shown lined up with their launch/recover points.

It should also be noted that this is not a very optimal placement of the sidepanels. Later research by Ken Janik has shown that another 50% or more performance can be gained by more optimal placements with about the same amount of hardware, as well as providing comparisons that quantify the performance gains of CDF over previous counterflow designs. Also, our research strongly indicates that the optimizations that the compiler was making were suboptimal for any modern out-of-order speculative processor with long latency memory.

While the performance of the single instruction pipelines seems to be disappointing, the performance of the dual-instruction pipelines more than compensates. Below in Table 2 is the instructions per cycle (IPC) for each of the five models for sev-

eral of the SPEC95 benchmarks.

Table 2: IPC for SPEC95

Benchmark	CDF0	CDF1	CDF2	CDF3	CDF4
applu	0.459	0.604	0.819	0.850	0.960
compress	**0.522**	**0.699**	**1.040**	**1.080**	**1.456**
gcc	**0.651**	**0.752**	**1.106**	**1.120**	**1.330**
hydro	0.652	0.756	1.090	1.098	1.300
ijpeg	0.706	0.704	1.039	1.039	1.069
li	0.691	0.738	1.085	1.089	1.231
m88k	0.582	0.748	0.949	0.948	1.051
su2cor	0.470	0.547	0.526	0.526	0.534
swim	**0.651**	**0.799**	**1.351**	**1.406**	**2.191**
tomcatv	0.649	0.759	1.085	1.097	1.305
turb3d	0.583	0.687	0.854	1.003	0.873
vortex	0.711	0.763	1.028	1.030	1.166
wave5	0.570	0.700	1.001	1.026	1.315

Of the SPEC95 set of benchmarks we chose three that we believe are representative of the entire group to show graphically the performance improvements and other features. These are compress, gcc and swim. There are two important trends to note from the graph showing the IPC of the processors. First, adding more pipes generally improves the performance significantly. This is a clear indication that neither the functional units or the ROB are bottlenecks when there is one or two instruction pipes. This is due largely to the fact that we put very large ROBs on the simulated processors and the number of functional units is relatively high for a processor with a fetch bandwidth of one or two

A notable exception to this is are the specFP applications, specifically, su2cor, While there are some other contributing factors, like poor sidepanel placement for floating point code in general, the high content of back-to-back long-latency operations doesn't provide much parallelism to be exploited. Hence, those programs are bottlenecked by the availability and latency of execution units until data speculation is used.

There is also a trend that swim does better than gcc and gcc does generally better than compress. This actually due in a large part to the data dependencies within the programs. To measure these data dependencies, we wrote a small metricing program with aBlocks to measure the distances between data dependencies. Compress has a very tight chain of data dependencies. On average, instructions are dependent on the instruction 1.3 instructions in front of them (average RAW hazard distance). This tight data chaining is not only the worst of the SPEC95 set, but it means that there is not very much instruction level parallelism available to exploit.

On the other end of the spectrum, swim's instructions are dependent on an instruction 13 instructions in front of them. This allows for a great deal more computation to be done in parallel. As can be seen in Fig. 5, as the effective instruction window

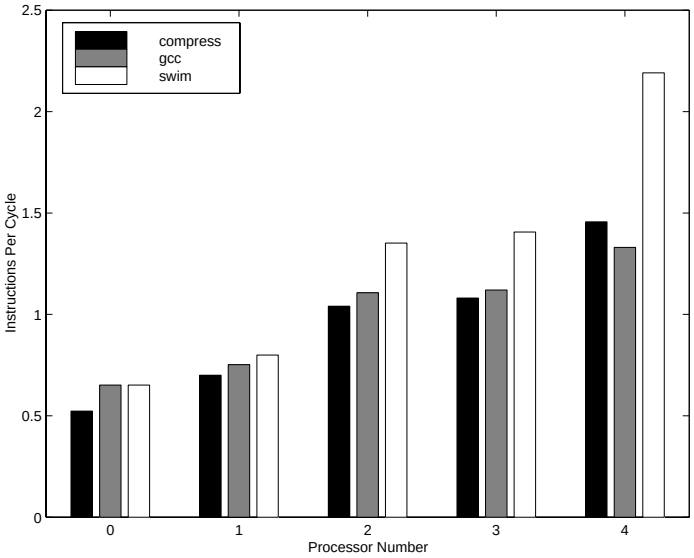

Fig. 5. IPC for Selected Benchmarks

increases, swim is able to run more in parallel, and therefore in much fewer cycles. Also of note is that in gcc the average RAW hazard distance is 3, so it does not benefit as much as swim from the hardware additions in CDF4. Fortunately there are ways of improving performance in spite of tight data dependence chaining. Some of these methods, like data speculation and multithreading will be discussed relative to CDF in section VIII

The utilization of the pipelines is one of the most important factors in counterflow designs. It gives an image of what is happening inside the processor, and is one of the key tools for identifying and removing bottlenecks by repositioning sidepanels. To keep the graphs simple, only processors 1,3 and 4 will be displayed (shown in dashed, dot-dashed, and connected), and only gcc and swim will be used. The graphs show on average how many slots in a pipe are being used for each stage of both the instruction pipe and data pipe. The launch numbers do not include instructions that were launched at that particular stage, but the recover numbers include recovers that occurred at that stage.

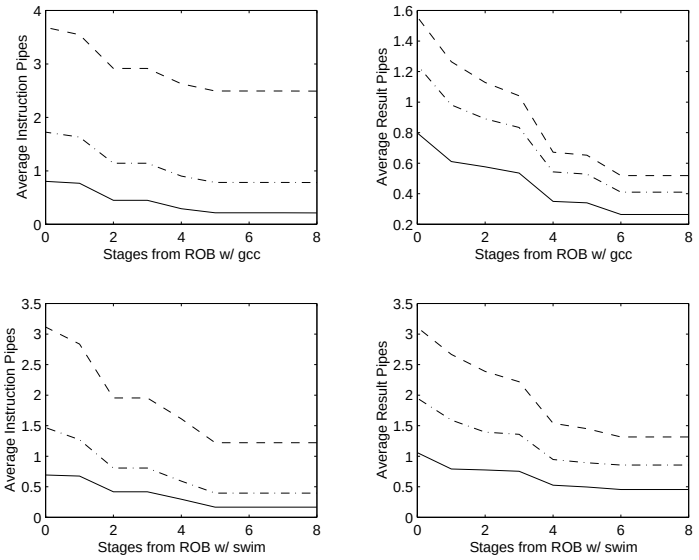

Fig. 6. Graphs of Pipeline Utilization

From these graphs, several things become obvious. The upper half of the pipeline isn't used much for execution, but mostly as a waiting period for long-latency operations. Also, the profiles are similar for each width, because these graphs are dependent on the sidepanel placement. Also of note is the amount of wrapping taking place. Whatever is furthest from the ROB will be wrapped around (or has been wrapped in the case of result tokens). As a general rule, we have found that if you cannot achieve less than half wrapping, you will wrap too much material and prevent fetching from taking place which will cripple performance as can be seen with CDF4 running gcc.

The misspeculation issue is quite important since we have a relatively high cost of misspeculation. Fortunately misspeculation only represents adding at most an addition 10% of invalid instructions into the processor. Even better is that unlike VRP, CDF is not required to execute these misspeculated instructions. If the instruction is known by the ROB/BPU to be invalid when it wraps by the ROB it is removed from the pipeline.

VIII. Advanced Features of CDF

The distributed architecture of CDF lends itself well to allowing a number of interesting features and modifications that can improve performance with minimal area additions. In comparison, most of these techniques add significant additional hardware and complexity to current superscalar designs. In this section we will discuss some of these features but not the actual performance benefits as aBlocks is not yet advanced enough to support simulating these ideas. First we'll examine what can be done about the ROB in terms of making is a non-associative structure to reduce the die area and decrease access time. Second, we'll examine how multithreading can be very beneficial because of the way the instruction pipeline is utilized. Third, we'll discuss the ability of CDF to manage data prediction in a very efficient manner. Next, we'll look at how complex instructions can be mapped onto this RISC style architecture. Finally, we'll conclude with multi-clocked CDF.

Currently, out-of-order processors tend to have some kind of reorder mechanism to support precise interrupts. In both superscalar designs, VRP and CDF this usually takes the form of content addressable memories (CAMs) to determine the data dependencies. Ideally, these structures should be as large as possible, as they usually determine the instruction window size, but being CAMs, they tend to be both expensive and slow. The CDF's ROB can be constructed with non-associative memory. When there instruction pipe width is greater than one, then the ROB can be divided into as many sections as there are horizontal entries in the instruction pipeline. Then, the actions taken by the ROB can be done exclusively by indexing and "last modified by" tags for the register file as shown in Fig. 7. This also helps reduce the number of read and write ports required on the ROB.

The penalty for creating a non-associative ROB is that when a branch misprediction occurs, the table containing the ROB entries for instructions writing to the RF must be reconstructed. While instructions can be allowed to continue to execute during this reconstruction time, no new instructions can be issued. We believe that the penalty is somewhere in the neighborhood of 4-32 cycles, depending on the size of the ROB and the degree of segmentation. This is acceptable if the branch prediction rate is sufficiently high, similar to the Intel Pentium Pro design which compensates for a high mispredict cost with excellent branch prediction [8].

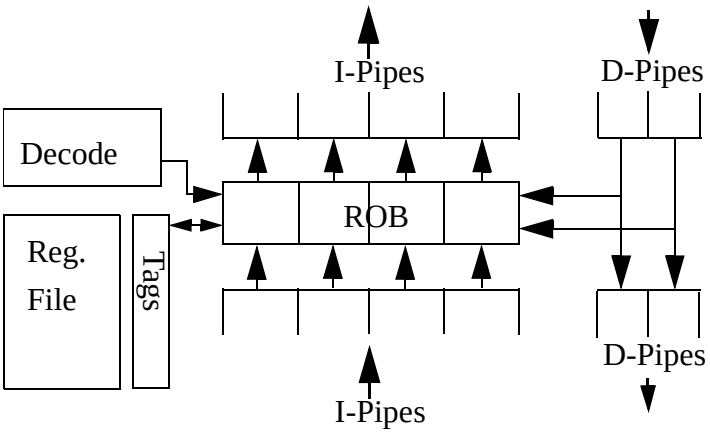

Fig. 7. Diagram of Segmented ROB

One of the problems facing high performance processor architects is that there is a limited amount of IPC available in programs. To increase the amount of parallelism available, many groups have turned to hardware mulithreading to hide latency of memory and provide additional parallelism since the threads have very little interaction between each other [9]. There are a variety of ways to add mulithreading to a CDF architecture, but one particular way takes advantage of the properties of CDF.

Since the instruction pipeline utilization drops off the further away one gets from the fetch/decode area as seen in Fig. 6, letting each thread have it's own ROB/Fetch/Decode (or perhaps a multiplexed fetch/decode) spaced evenly around the pipe should provide excellent performance over more traditional designs. The reason is that the instructions that can be executed quickly (like ones based on immediates or values computed far in the past) will be executed by the first execution unit encountered, leaving holes in the pipeline for the thread to use. Functional units close to a thread's ROB will be used more by that thread than the other threads. This results in that while the threads still compete for resources in the processor, the 'prime resources' (resources close to the ROB) are not the same for all the threads, each has it's own set. This is very different from a superscalar design where all threads are constantly competing for exactly the same resources so there is less competition.

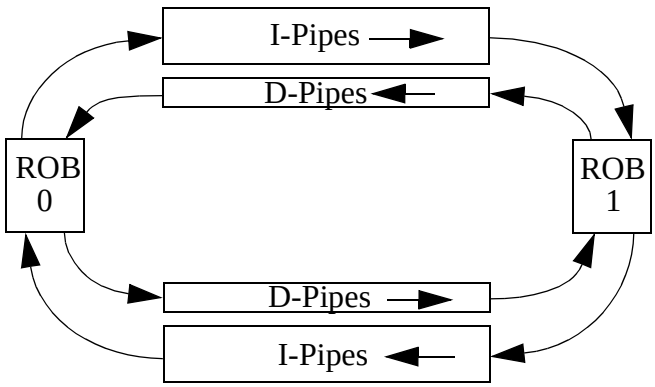

Fig. 8. Diagram of a 2-thread CDF System

The real benefit of multithreading in CDF is that more parallelism is available to the scheduling core. Because the threads are independent, there are no data dependencies between them. This means that there will be more instructions available for execution at any point in time. This is very beneficial, since otherwise execution units might be idle waiting for data dependencies to be resolved.

The resource contention that remains could be managed more efficiently by assigning a priority to the instructions. For example if one thread was speculating, it might mark it's instructions as being a lower priority than usual to allow non-speculative instructions (even from another thread) to be executed first. Each time a instruction makes a loop around the processor it's priority could be increased to indicate that it should be executed soon since there are probably a number of other instructions waiting for it to complete. One might even go so far as to have the compiler add 'hints' to the processor indicating which instructions are in the critical path of a computation, or have a large number of instructions dependent on them.

An interesting benefit of the pipeline is that instruction reuse becomes straightforward. For example, if you wanted to do value prediction, simply make the prediction at some point, and allow instructions with predicted operands to execute (except for stores of course) but not leave the pipeline. They can even put their results in the result pipeline as long as they are marked as speculative data. Then, when the actual value of the operand passes the instruction with a speculated consumer, the check to see if they are the same. This comparison can be done right there in the pipe and the instruction can then issue a producer that is not speculative if the speculation was correct, or take the correct value of the consumer and get reexecuted. Deep speculation and recovery from misspeculation is simple and elegant.

Entire data dependency trees can be speculated and reexecuted multiple times without expensive customized hardware to recover from misprediction. It is somewhat similar to the way branch prediction occurs, except that we are able to reuse the instructions, similar to a trace cache [10]. There is, however a risk of overspeculation if you combine this technique with mulithreading or some other mechanism where the speculated instructions could prevent other instructions that are more likely to be useful not to be issued or executed. However, if you only allow data-speculated instructions to be executed when there is nothing else to be done in a pipelined execution unit, then all you loose is some of the result pipe bandwidth, a relatively inexpensive resource.

Even though RISC seems to have been declared the winner in the instruction set wars, more complex instructions keep getting added to the relatively large existing instruction sets. Sparc's V9 adds SIMD instructions to the ISA as does Intel's MMX. In general, these more complex instructions are either dealt with by custom hardware or by breaking the instruction into RISC-like microops and then executing the sequence of microops

There are three ways to deal with instructions that can be broken down into smaller components in a CDF architecture. The first is simply to devote dedicated execution hardware in a sidepanel. This can be expensive if the instruction is complex and cannot share it's functionality with other instructions. The other two options rely on the characteristics of a CDF pipeline to operate.

One alternative is to actually issue the micro-ops separately into the pipeline. Careful choice of tags can allow all the micro-ops in an instruction to share the same ROB entry. If there is any parallelism in the instruction itself, it can easily be exploited, and the execution units can be kept very simple, as all they need to execute are simple micro-ops. On the down side, a substantial amount of the pipe itself will be taken up by the instruction's micro-ops and communication within the instruction. If there is little or no parallelism within the instructions, then the best that can be hoped for is that the parallelism between instructions will be improved by having more, smaller operations active in the pipeline at once.

The other alternative to dedicated hardware is to have the instruction execute in multiple sidepanels. For example, a Multiply-Add would first execute the multiply in a multi-cycle integer unit to perform the multiplication. Then the result would be placed in the consumers array of the instruction and it would then be launched into an single-cycle integer unit to execute the add. This has the advantage of not polluting the pipe with extra micro-ops, but it means that if there is any parallelism within the instruction, it cannot be used.

Finally, there is the issue of ultra-high speed clocking of the pipeline. There has been some research into running different segments of a processor at different speeds to get performance gains[11]. For example, it may not be feasible to run an entire chip at the technologies fastest available speed due to power or heat reasons. Also, from an architectural standpoint it doesn't make sense to run the fetch unit at 600Mhz if you can only pull in instructions from the cache at a rate of 300Mhz. But it does make sense to run the execution core at 600Mhz even if the cache can only sustain 300Mhz because it usually takes more than one cycle to complete an instruction due to data dependencies and hazards.

CDF designs lend themselves very well to a multi-clocked scheme because the execution core has only a few links to the "outside world". Also the communication within the pipe is all localized, so it can support very high clock speeds because there is no need to transmit any information in the core a significant distance between clock cycles. Moreover, because the pipe stages are all the same, they can be carefully optimized for excellent performance.

Consider the case where fetch/decode is running at half the speed of the CDF core pipeline. So every other pipeline clock, new instructions are added into the core. If the number of stages in the pipeline is relatively prime to the number of cycles it takes to fetch (one fetch per 2 pipeline cycles and 9 pipe stages for example) then the pipestage that the fetched tokens will be going into, will be the least recently fetched to location in the pipe, and thus the most probable to have places to accept the new instructions. This concept can also be extended to Globally Asynchronous Locally Synchronous (GALS) systems with localized, independent clocks.

So now that you have seen a overview of some of the things that CDF pipelines can do because of their unique structure. The direction of at least part of our future research should be obvious, but there are some specific things that we believe are important to look at.

IX. Future Research

The research at OSU is at the point where we believe that we have shown that counterflow pipelines like CDF can be effectively used as general purpose processors if they are properly constructed. We also believe that there is a great deal of additional work that can be done in showing that the advanced features of CDF, as mentioned in this paper are viable and provide improved performance. This would help show that CDF processors are a reasonable alternative to modern superscalar designs given the new constraints of higher clock speeds and longer memory latencies.

We fully intend to continue developing aBlocks not just for use in counterflow research but as a platform for prototyping and visualizing new processors, and better understanding existing processors. Some additions on the horizon include execution-based simulation, a 5-stage pipeline and a generic superscalar processor, along with more advanced bus models supporting SMP. We also intend to take advantage of the JavaBeans API in the hopes that aBlocks will eventually develop to the point where generating or modifying a existing processor simulator can be done with a few mouseclicks instead of hours of tedious programming.

We are also concerned with the amount of ILP currently available in modern programs. The basic block size for SimpleScalar binaries is around 4-5, and the average number of instructions between data-dependent instructions is around 3 for the SPEC95 benchmarks. Research into more aggressive compiler techniques to increase available ILP will benefit not only CDF processors but other out-of-order processors like superscalar designs. We would also like to explore alternative instruction sets like the Java bytecodes to see if their CISC-like structure would allow us to reveal more parallelism within and between instructions while reducing memory bandwidth.

Finally, there is the issue of sidepanel placement. While it is impossible to achieve an optimum choice of sidepanels, pipe-widths, and placements, careful choices can make the difference between poor performance and excellent performance. We have developed some rules of thumb by trial and error while simulating various counterflow pipelines. These need to be reexamined and performance tradeoffs quantified for number, type and placement of sidepanels, pipe widths and reorder buffer sizes.

X. Conclusion

Our research has shown that the original counterflow designs had fundamental limitations on performance for general purpose computing. We have also developed solutions for most of these limitations and given the current trends towards high internal clock speeds, high memory latencies, multithreading and value prediction, the CDF processor will be a viable, scalable alternative to traditional superscalar designs in the near future. The distributed scheduling mechanism and localized clock and data transfers makes it ideal for ultra-high clock speed implementations. The current simulation results show that good performance with reasonable hardware can be achieved if care is taken in designing the layout of the sidepanels. To see the simulators yourself or take a look at a more complete set of simulation data, please visit our web page at http://ece.orst.edu/~sllu/cfpp

Acknowledgments

This project would not have been possible without the hard work of all the members of the CFPP design team at OSU over the past several years. We would also like to thank Dr. Sproull and Dr. Sutherland for their original ideas in counterflow processor architecture; GNU software for their hard work and excellent products; Todd Austin and Doug Berger for their work with SimpleScalar; SunSoft for Java, Ben Lee, for providing insights into many aspects of processors, and Intel and HP for donating equipment necessary to perform this research.

References

[1]. R.F. Sproull and 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.

[2]. K.J. Janik, S. Lu, M. Miller, “Advances of the Counterflow Pipeline Microarchitecture” Presented at HPCA3, Feb 1997. http://www.ece.orst.edu/~sllu/cfpp/papers/vrp1/vrp1.ps

[3]. K.J. Janik and S. Lu, “Synchronous Implementation of a Counterflow Pipeline Processor” http://www.ece.orst.edu/~janikk/iscas96.ps

[4]. M.D. Jones, “A New Approach to Microprocessors”, Nov. 1996. http://lal.cs.byu.edu/people/jones/latex/sproull.html/sproull.html.html

[5]. M.B. Josephs and P.G. Lucassen and J.T. Udding and T. Verhoeff, ``Formal Design of an Asynchronous DSP Counterflow Pipeline: A Case Study in Handshake Algebra", *Proc. Int'l Sym. on Advanced Research in Async. Circuits and Systems*, pp. 206-215, November 1994.

[6]. R. Carlson and M.F. Miller, “VRP Simulator” Apr. 1996. http://www.ece.orst.edu/~sllu/cfpp/vrpsim/docs/vrpsim.html

[7]. Intel Corp., i486 Microprocessor Databook (240440-001), Santa Clara, CA, April 1989.

[8]. Dileep Bhandarkar and Jason Ding, “Performance Characterization of the Pentium(R) Pro Processor,” Proceedings of the 3rd International Symp. on High Performance Computer Architecture, Feb. 1997, San Antonio, TX, pp. 288-297.

[9]. J.L. Lo, S.J. Eggers, J.S. Emer, H.M. Levy, R.L. Stamm, and D.M. Tullsen, “Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading,”Transactions on Computer Systems (August, 1997).

[10]. E. Rotenberg, S. Bennett, J.E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. Proceedings of the International Symposium on Microarchitecture, pages 24-34, December 1996.

[11]. P. P. Vaidyanathan, “Multirate digital filters, filter banks, polyphase networks, and applications: A tutorial,” Prodeeings of IEEE, Vol. 78, No. 1, pp. 56-93, 1990.

[12]. Todd Austin, “SimpleScalar Tools Hacker’s Guide,” talk given at Dept. of ECE Oregon State University, 1996.

[13]. D.C. Burger and T. M. Austin. “The SimpleScalar Tool Set, Version 2.0,” University of Wisconsin Computer Sciences Technical Report #1342, June, 1997.