Counterflow Pipeline Architectures: A Survey

Venkata Suryadevara, Steven Kollmansberger, Charles Parker

June 6, 2003

Abstract

The couterflow pipeline processor has been viewed by research groups at Sun Microsystems and Oregon State University as a possible candiate for usage in a commercial capacity alongside superscalar processors. Its unique approach offers some new possibilities for avoiding pipeline stalls and exploiting instruction level parallelism. In this paper, we look at some of the work that has been done with CFPP and its successors, and offer comment on possible future work and the current state of the art.

1 The Basic Counterflow Pipeline

1.1 Introduction

Pipelines are used to accelerate execution by allowing several computer instructions to be completed at once. Pipelined processing has allowed tremendous performance increases in computing power. In this paper, we are presenting a summary of a pipelined architecture fundamentally different from traditional pipelines-the Counterflow Pipeline Architecture. The Counterflow Pipeline Processor (CFPP) architecture was a proposal for a simple and regular structure for processor pipelines. It handles in a uniform way a number of features that add complexity to conventional designs, including operand forwarding, register renaming, and pipeline flushing following branches and traps. A CFPP uses a bi-directional pipeline in which instructions flow and results counterflow to move partially-executed instructions and results in a regular way, subject to a few pipeline rules that guarantee correct operation. The CFPP structure has a number of properties that promise advantages:

1. **Local Control:** Only local information is required to decide whether an item in the CFPP pipeline should advance. There is no need to compute global pipeline stall signals and then distribute them to all pipe stages. The complexity of such a global stall computation and the time required to compute and distribute the signal are major headaches in current processor designs.

2. **Regular Structure and Simplicity:** The CFPP structure is very regular, which provides hope that it can be laid out on silicon in a simple, regular way. The simplicity and regularity should also help in devising correctness proofs. The overall simplicity of the structure may mean that CFPP processors are easier to design than their conventional counterparts, which might make them very attractive

3. **Local Communication:** Pipeline stages communicate primarily with their nearest neighbors, allowing short communication paths that can be very fast.

4. **Modular Structure:** The uniform communication behavior of pipeline stages admits variants that differ in the detailed design of individual stages or in the ordering of stages in the pipeline. For example, one design may call for a single ALU, another for separate adder/subtractor and multiplier units; yet both have the same overall pipeline structure.

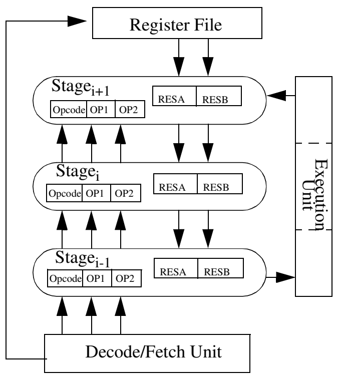

Figure 1: A Simple CFPP

5. **Simplicity:** Finally, the overall simplicity of the structure may mean that CFPP processors are easier to design than their conventional counterparts, which might make them very attractive.

The design of the CFPP is fundamentally different from traditional pipelines in two different ways:

- **Asynchronous Execution:** The CFPP does not use a clock to synchronize the movement of information.
- **Information Counterflow:** The CFPP uses a counterflow of information to ensure program execution. In traditional processors, the information flows only in one direction but in the CFPP, information is divided into two parts where each part flows against the other.

1.2 Structure

In a CFPP, instructions owe up through an instruction pipeline so that the sequence of instructions in the pipeline resembles a listing of that section of code. Actually,there are two pipelines that flow against each other in CFPP. In figure 1, the instructions move up the pipeline on the left side of the stages from the program counters to the register file. Results move in the opposite direction on the right side of the stages from the register file to the program to the program counters.

Instructions move up the pipe as far and as fast as they can, stalling only when the pipeline stage immediately above cannot yet accept a new instruction or when the instruction reaches the last pipeline stage that is equipped to execute it.

In CFPP, instruction and result packages are built of smaller units called bindings. Each instruction contains three bindings and an operation code. The first two bindings contain instruction operands while the last binding contains the result.

Whenever an instruction is executed, the outputs are used in two ways. First, they are entered in the instruction's destination bindings and are eventually retired into the register file. Second, each destination binding is inserted into the downward-owing results pipeline so that it may be observed by subsequent instructions.

In figure 2, each stage of the results pipeline accommodates two bindings. Any later instruction that requires a source binding whose register name matches the register name of a result binding will garner the value by copying and retaining the value in the binding in the instruction pipe. Result bindings flowing down may be modified by a subsequent instruction. Each stage must detect matches between instruction and result bindings, i.e., cases when the register name in an instruction binding matches the register name in a result binding. An instruction that has already executed and has a

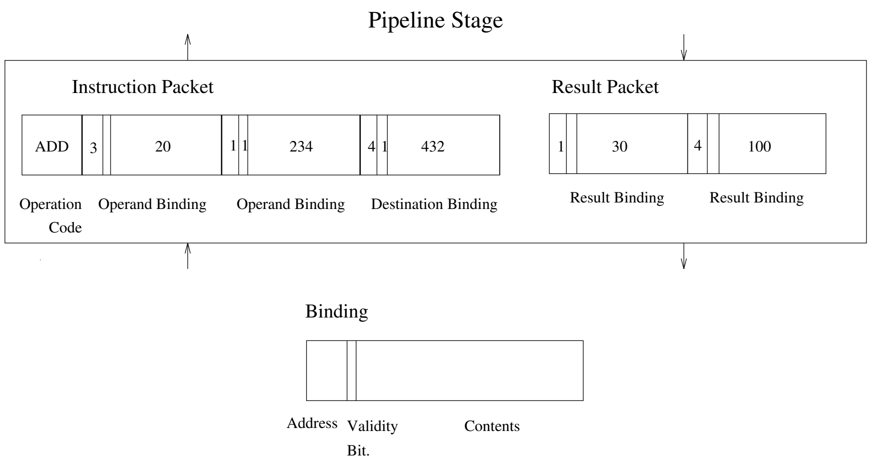

Figure 2: CFPP Stage Layout

destination binding that matches a result binding must copy the value from the destination binding into the result binding. Several different bindings for a register may be in transit in different parts of the pipeline at the same time. The register file is also a source of operands for instructions.Each binding contains a value from the register file needed by some instruction in the pipeline. Now, CFPP must detect matching bindings in each result flowing down with each instruction flowing up. Each result must meet every subsequent instruction in some pipeline stage, where the comparison is done. Thus, we must prevent adjacent pipeline stages from swapping an instruction and result at the same time to prevent detecting matches. The interface between stages may either pass an instruction up or a result down, but never both.

1.3 Information Flow in CFPP

- **Information flow between stages:** Instructions are inserted into the bottom of the pipeline and flow from stage to stage to reach the top likewise results are inserted at the bottom of the pipeline, move from stage to stage to the bottom of the pipeline.
- **Information flow within stages:** An instruction and result package occupying the same stage at the same time exchange information.

1.4 Pipeline Rules

The correctness of counterflow pipeline operation depends on a set of pipeline rules that each stage must obey. These rules, which have been introduced above, are stated explicitly in this section. The rules will be extended in our later discussion of traps and conditional branches.The pipeline rules make frequent reference to bindings: source bindings and destination bindings are held in instructions as they flow up the instruction pipeline, while result bindings flow down the results pipeline.

The first few pipeline rules concern instructions:

1. **No overtaking:** Instructions must stay in program order in the instruction pipeline
2. **Execution:** If all of the source bindings for an instruction are valid, and if the instruction is held in a stage that contains suitable computing logic, the instruction may execute. When an instruction completes execution, its destination bindings(s) are marked valid and value(s) are filled in.

3. **Insert Result:** When an instruction completes execution, one or more copies of each of its destination bindings, is inserted into the results pipeline.

4. **Stall for operands:** An un-executed instruction may not pass beyond the last stage of the pipeline capable of executing it also an instruction cannot be retired into the register file until it has executed.

The following are matching rules that apply when an instruction and result are present in the same pipe stage and have matching bindings:

1. **Garner Instruction Operands:** If a valid result binding matches an invalid source binding, copy the result value to the source value and mark the source valid.

2. **Kill Results:** If an invalid destination binding matches a valid result binding, mark the result binding as invalid.

3. **Update Results:** If a valid destination binding matches a result binding, copy the destination value into the result value and mark the result valid. To invalidate subsequent instructions, an additional pipeline rule is applied when a result and instruction meet in a stage(applied in branches)

4. **Kill Instruction:** If a result binding is either trap-result or wrong-branch-result, mark the instruction invalid. The instruction may proceed up the pipeline, but it will have no side effects on the results pipeline or the register file.

1.5 Traps and Conditional Branches

In a CFPP, a specially-marked result traveling down the results pipeline invalidates instructions following a trap or wrongly-predicted branch. If the execution of an instruction causes a trap, the stage that generates the trap introduces a special trap-result binding into the results pipeline instead of the normal destination bindings.

When the trap result reaches the bottom of the results pipeline, it is interpreted specially by the stage responsible for program-counter control which changes the program counter so as to start fetching instructions from a suitable trap handler. As these new instructions enter the pipeline, they will not meet the trap result, and so will remain valid.

A conditional-branch instruction which cites the condition code register as a source, goes up the instruction pipeline, just like any other instruction. Meanwhile, the program-counter control makes a branch prediction and continues to launch instructions into the pipeline from the predicted instruction stream. When the conditional-branch instruction executes, if it determines that the branch was predicted incorrectly, it sends down the results pipeline a wrong-branch-result binding, which contains a value for the correct program counter target of the branch. This special result will kill all subsequent instructions it meets in the pipeline. When the branch result reaches the program counter control stage, the program counter is adjusted to reflect the result of the branch, and instructions will now be fetched from the proper path.

Figure 3 represents a step by step execution of instructions in a five stage pipeline. Two columns show what is held in the instruction and result registers of each pipeline stage(assuming each stage holds two resultant bindings at once). The upward and downward pointing arrows indicate the movement of instructions.

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R		$B[2]C[3]\downarrow$	Registers contain: $A[14]B[2]C[3]D[21]$
0			
1			
2	$A[] := B[] + C[]\uparrow$		
I	$PC = 102\uparrow$		Fetch, send source names A, B to reg file

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R		$A[14]B[2]$	Registers contain: $A[14]B[2]C[3]D[21]$
0		$B[2]C[3]$	Mustn't swap with instruction below.
1	$A[] := B[] + C[]\uparrow$		Mustn't swap with result above.
2	$B[] := A[] + B[]\uparrow$		
I	$PC = 103$		Fetch delayed due to cache miss.

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R		$A[14]B[2]$	Registers contain: $A[14]B[2]C[3]D[21]$
0	$A[] := B[2] + C[3]$	$B[2]C[3]\downarrow$	Garner B, C; execute
1	$B[] := A[] + B[]$		
2			
I	$PC = 103\uparrow$		Fetch, send source name C to reg file

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R		$A[14]B[2]$	Registers contain: $A[14]B[2]C[3]D[21]$
0	$A[5] := B[2] + C[3]\uparrow$	$A[5]$	Insert result
1	$B[] := A[] + B[2]\uparrow$	$B[2]C[3]$	Garner B
2	$D[] := C[] - 1\uparrow$		Literal –1 held in binding value
I	$PC = 104$		Fetch, send source names to reg file

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R	$A[5] := B[2] + C[3]\uparrow$	$A[]B[2]$	Registers contain: $A[5]B[2]C[3]D[21]$
0	$B := A[5] + B[2]\uparrow$	$A[5]$	Garner A, execute
1	$D := C[3] - 1\uparrow$	$B[2]C[3]$	Garner C, execute
2	. . .		
I	$PC = 105$		Fetch, send source names to reg file

Figure 3: A Step-by-step Execution of Instructions in a CFPP

1.6 Function Units and Sidings

In a CFPP architecture, the processing work gets done in the functional side units. There are two stages associated with each side unit-launch and return.In the launch stage, operands and operations are sent to the side unit, and in the return stage, the side unit returns the result obtained by performing the operation. Figure 4 displays multiplication side unit and comparison side unit. The sidings themselves are pipelined so that several operations can be in concurrent progress. While siding is performing its job, the instruction that launched the operation proceeds normally along the instruction pipeline so that it can recover and handle results in proper sequence with other instructions in the pipe.

1.7 Register Files and Register Caches

The CFPP architecture permits several separate register files to co-exist along the pipeline, but any instruction that may alter the contents of a register in the file must execute before passing the corresponding register file. One can locate separate fixed and floating point register files at different places along the pipeline. The register file is equivalent to a series of stages that hold recently completed instructions. Register cache helps in reducing the latency entailed in fetching the values from the register file and passing them down the results pipeline. It is located above the instruction decoder and it enters values into source bindings, for any registers, that have valid entries in the cache.

1.8 Pipeline Control

The pipeline control in CFPP is elastic: new packets can be inserted into or removed from a stream of packets. When an instruction executes, it inserts a new result into the result path. This result could occupy a previously invalid result binding in a result packet or it might require inserting a fresh result packet into the result pipe. If an instruction is killed by a trap or conditional branch, it is marked as invalid and is propagated through the rest of the pipe without causing side effects. Several results are packed into a result packet and allowed to flow down the pipeline as a unit. A result packet contains two full word operands and a condition code. Another packet called instruction packet, carries its opcode ,source and destination bindings and other data like program counter that fetched the instruction. Each stage of the pipeline contains latches for holding an instruction packet and a result packet which provide storage that is essential for pipelined processing. Circuits are connected to these latches to enforce pipelines rules, including comparators that detect when instruction and result bindings match.

Easy modification is one of the main advantage of CFPP design. The design is easy to modify because each stage is basically the same. Communication happens only between stages while short communication connections prevent changes made in one part from extending to other parts of the pipeline.

CFPP has performance-wise advantages also. The CFPP does not directly solve structural hazard problems but its uniformity leads to fast design modifications that would allow the addition of more hardware to handle structural hazards without increasing overall processor complexity. However, the existence of side units might lead to some scheduling problems. Counterflow instructions and results eliminates data hazards elegantly in a CFPP. As instructions update results and garner operands, they are guaranteed to carry correct information that reflects changes made by prior instructions. Branch prediction and recovery schemes are used to handle control hazards. A scheme is developed for recovering from incorrect branches that only requires communication between adjacent stages which preserves the simplicity of the CFPP design.

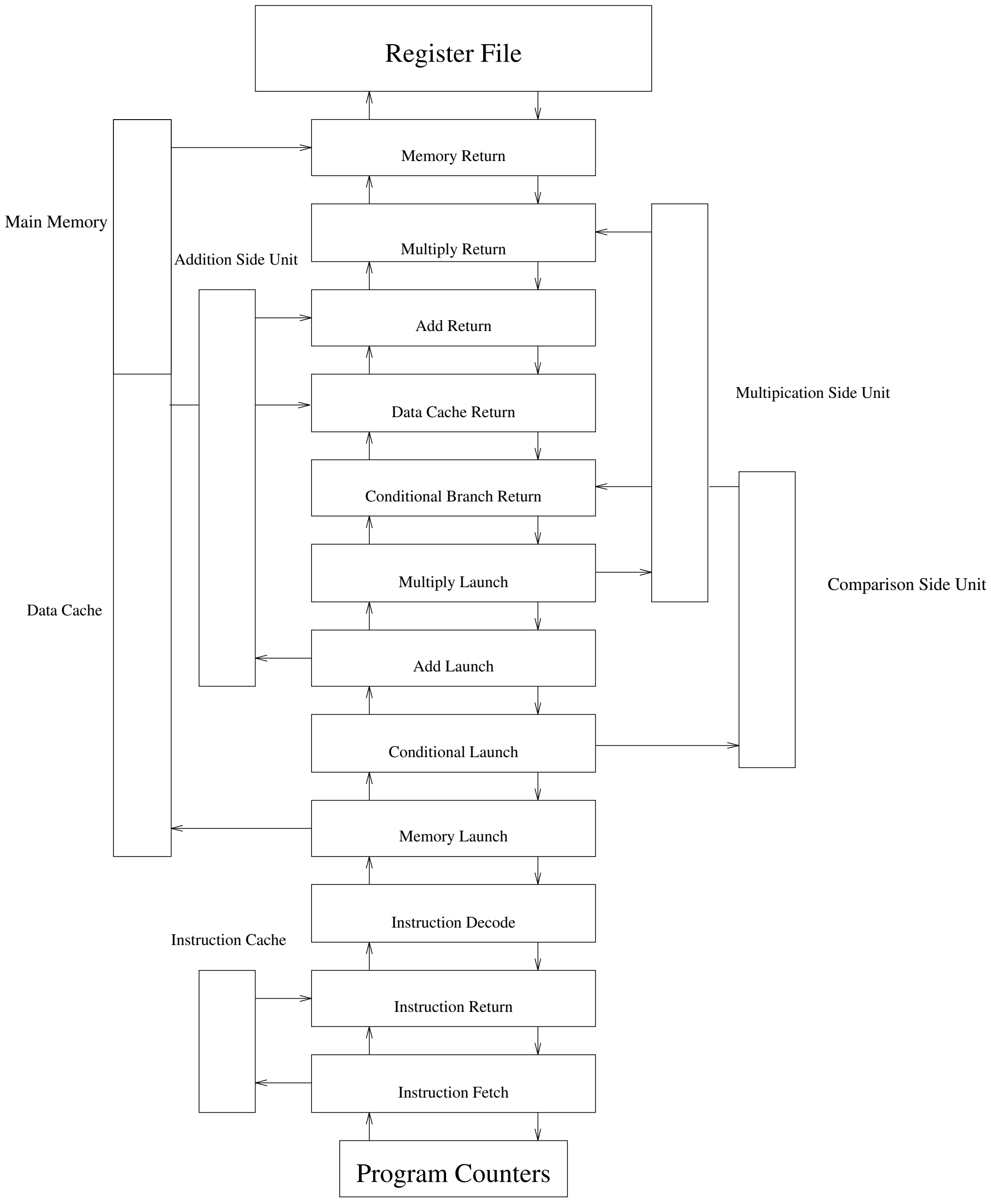

Figure 4: Block Diagram of a CFPP with Functional Units and Sidings

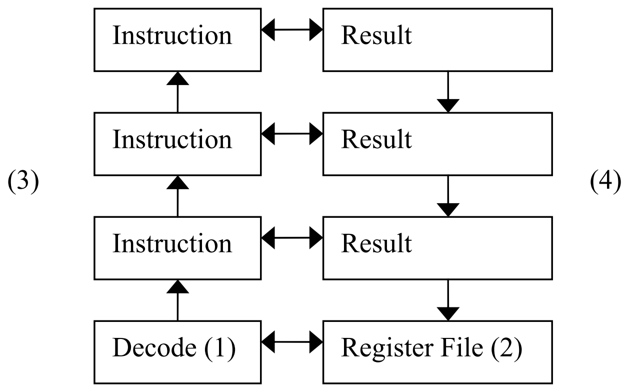

Figure 5: Basic VRP Design

2 The Virtual Register Processor

2.1 The Need for Virtual Register Processor Design

The counterflow pipeline processor introduced in the previous section shows significant promise for increasing pipeline efficiency and simplifying design. However, the original design cannot be competitive against current super-scalar architectures for four major reasons.

1. When a new set of instructions accessing different registers or memory locations begins, they will have to traverse half way to the top of the pipe to receive their data. If side panels branch off before this midpoint, stalls will further decrease pipeline efficiency.
2. When the system makes an incorrect branch prediction, all operands and instructions have to be marked invalid and flushed from the register. This imposes a significant penalty every time a branch mis-prediction occurs.
3. Completed instructions are not removed from the pipeline, so if a stall occurs high in the pipeline, lower instructions may be stuck behind completed instructions. If they were allowed to overwrite completed instructions, the impact of high-level stalls may be greatly decreased.
4. No allowance is made for multi-issue architectures, thus creating an absolute service ceiling of one instruction per cycle, which is far below common super-scalar architectures.

2.2 Virtual Register Processor Architecture

Obviously, these problems are significant roadblocks to the acceptance of the counterflow design. The virtual register processor design was the first major step to resolving these performance issues. The primary change in this design is to move the register file to the bottom of the pipeline, next to the decode unit. The register file adds a validity bit for each register, to indicate if the value in the file is valid, or if the appropriate value is coming down the pipe somewhere. When instructions are processed by the decode unit, they receive valid register operands immediately from the register file. Any destination registers are marked invalid, and the instruction begins upward through the pipeline. Invalid registers in the file will be created by instructions higher in the pipe, thus leading to results coming down the pipe. As they encounter newer results while traversing the pipe, their operands will be updated. This change eliminates the start up cost and opens the door to multi-issue designs.

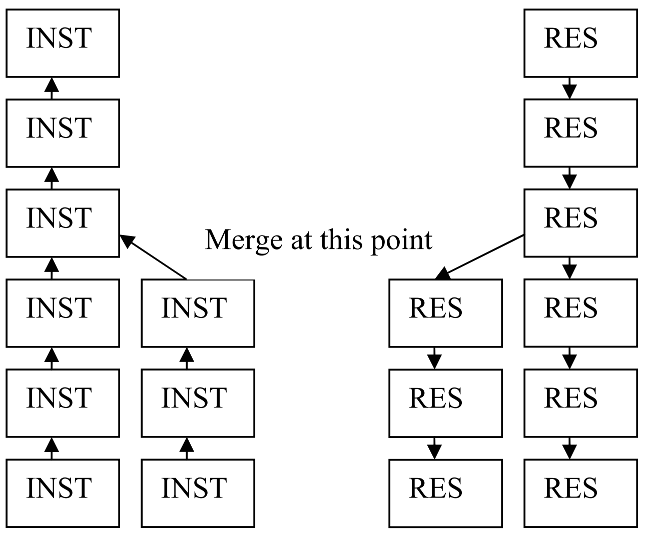

Figure 6: Multi-Issue VRP with Merging

Another advantage of having the register file at the bottom is also apparent: once an instruction has completed and turned the final values over to the downward result pipeline, the instruction can be dropped from the pipeline. In the original counterflow system, the finished instruction was forced to proceed all the way to the top of the pipe to deliver its result. The advantage of being able to remove instructions may not be immediately apparent. The concept is to turn finished instructions into bubbles which can collapse, preventing high-level stalls from clogging up the entire pipeline. Consider this example: rising up the pipe are a multiply, an add and another multiply instruction. The add completes early, leaving the two multiply instructions with a bubble between them. The top-most multiply instructions stalls, waiting for an operand or side-bar availability. At this point, in the old counterflow method, the multiply instruction two steps down, and another below it, would have all stalled. As it is, the lower multiply instruction and anything below rise up, consuming the bubble, decreasing the stall by one cycle. If the top multiply stall is only one cycle, then the pipeline overall experiences no stall. Likewise, when an instruction stalls, instructions above it continue upward, leaving bubbles in their wake. In this way, an elastic pipeline is created wherein the distances between instructions can increase or decrease as needed to permit maximum flow.

The remaining problem is the problem of pipeline flushing on incorrectly predicted branch instructions. In order to deal with this problem, the authors introduced revalidation. Revalidation adds a flag to each instruction in the pipeline such that an instruction can be told not to pick up newer looking operands in the pipe. In this way, when a branch occurs, all existing instructions from before the branch will not be confused by new operand results coming from instructions after the branch. This solves the pipeline flush on branch misprediction problem.

2.3 Multiple Issue VRP

Consider that many instructions will be completed before they reach the top of the pipeline. Using this, we can create multiple partial instruction pipes to parallelize the most common (lowest) instructions. These pipelines can then merge, higher in the pipe, after many instructions have been reduced to bubbles. The results pipeline mirrors the instruction pipeline, expanding from the top down to accommodate the results from the more parallel pipeline sections.

However, this implementation is not as straightforward as it may seem. In a single pipe implementation, no instruction is able to pass another instruction. In this way, data dependencies of that sort are impossible. However, with multiple pipes this could happen. One solution is to stall horizontally i.e., whenever one instruction stalls, all instructions at that level stall. This is simple, but computationally expensive. Another option is to check all data dependencies to decide if a stall is necessary. This requires much more complex hardware, but could improve performance.

An alternate approach is to use VLIW to have multiple, specific-use instruction pipes. VLIW, however, has not seen much support and so was not discussed in detail.

2.4 Shortfalls of Virtual Register Processor Architecture

By permitted instructions to be overwritten as soon as they are complete, we also eliminate in-order retirement, which is required for precise interrupts. This has a very real significance in the handling of branch mispredictions. Specifically, all instructions above the first sidepanel return point must be revalidated, since we do not know which ones are before or after the branch. This can cause very long stalls as a result must travel all the way from a source instruction back to the branch instruction. While this is occurring, the entire CPU is stalled; unable to even speculate (i.e. predict) next instructions. These long stalls have the potential to make this architecture impractical.

In order to eliminate the giant stalls associated with revalidation, it is necessary to ensure in-order retirement. To accomplish this, instructions gained a reorder flag which allows instructions which should be stopped to finish executing in order, but not enter any values into the register file. This is a very simple change which eliminates the need for revalidation or pipeline flushing.

2.5 Simulation Performance

No actual processor designed in this manner actually existed at the time the paper was written, so the authors created a simulation to measure performance. This simulation used a single issued pipeline which lacked accurate branch misprediction handling and cache misses, but the authors later claimed that the results were appropriate adjusted to account for these things. The best performance achieved was an IPC of 0.85, after several optimizations which seem to address the specific matrix code being used for benchmarking. Original results were an IPC of 0.4. The authors also found if they designed a perfect branch predictor that the IPC rose to 0.99, almost perfect. However, this value seems to be of little worth, since at this point the simulated CPU was designed solely to execute the particular benchmark.

3 The Counter-Dataflow Architecture

3.1 Motivation

Even with the improvements made over traditional CFPP with the VRP approach, we can still see some substantial room for improvement. First we note that in VRP as in all previously discussed CFPP implementations, multiple issue is a problem: If an instruction is executing on a certain sidepanel execution unit, and another instruction needing that unit wishes to execute, we have no choice but to stall the pipe and wait for the execution to complete. To get around this is very costly as the only solution seems to be to design hardware that only moves along instructions not waiting for the said sidepanel, or to employ multiple execution units.

Another drawback is the stalls in the pipe. VRP manages to reduce these stalls, but they will still occur on instruction requiring a large number of cycles, or on memory accesses. This will require stalling at the top of the pipeline and the delays will move to the bottom, cloging the pipe and ultimatly slowing execution.

Name	Inst. Pipes	Result Pipes
CDF0	1	1
CDF1	1	2
CDF2	2	3
CDF3	2	4
CDF4	4	8

Table 1: Tested CFP Configurations

A solution to these problems that has been proposed is the so-called "Counter DataFlow" (CDF) architecture. In the counter dataflow architecture, we simply wrap the ends of the instruction and data units back into the decode unit, thus instructions that cannot execute because of data dependancies or resource conflicts are simply "reissued" as they reach the top of the pipeline. Obviously this solves the two issues mentioned above: Since instructions are forced (and are able) to constantly move through the pipeline, the pipe becomes quasi-infinate and we no longer have to worry about reaching "the end" of the pipe and needing to stall for a result. High-latency and resource starved instructions no longer bottle up the pipe.

The CDF pipeline differs from the in two additional ways: First, results of computation go into the result pipe rather than the instruction pipe. since these pipes are now seperate, we can now have instructions with differing execution times, whereas before this was a problem due to the consistancy required between the entry and recovery points in the pipeline. Second, instructions are removed from the pipeline upon execution in order to make room for new instructions in the pipeline.

3.2 Experiment and Results

The authors of one of the papers reviewed constructed a simulation of several candidate CDF processors, each having a different number of instruction/data pipes. These different processors, labeled 0 through 4, are summarized in 1. Each processor was simulated on the first 2,000,000 instructions in the SPEC95 suite of benchmarks.

In the paper mentioned above, the results of the compress, gcc, and swim benchmarks were deemed representative of the group in general. We see the results in figure 7. As is expected, a general trend is that the larger the number of pipes, the better the performance. Another general trend is that the swim benchmark is generally easier than the other two, and that gap increases with the number of pipes. A closer look at the three benchmarks reveals the reason: The swim benchmark has an average hazard distance (that is, the average distance to the next hazard) of 13 inststructions, with the other two benchmarks at 1.3 and 3 instructions for compress and gcc respectively. This allows for a high level of instruction-level parallelism in the swim benchmarks not possible in the other two, which in turn explains the outstanding performance of the CFP4 design on that benchmark.

Another factor measured was the average pipeline utilization. Note the this measurement can be taken in two different ways: On one hand, we would prefer that our entire pipeline would be utilized all the time as this would point to high effiency. On the other hand, we realize that if the entire pipe is utilized all the time, there will be no room for new instructions to enter! The compromise is hope for high utilization in early pipe stages and lower utilization in later ones. This is exactly what the authors found for most cases with both the data and instruction pipes. In the case of gcc, however, the high degree of data dependancy caused many instructions to remain in the pipe for a great deal of time, thus "over-utilizing" the pipe and preventing new instructions from entring. Obviously, this has a disasterous effect on performance, althougth leaving empty pipe stages at the top of the pipe leaves one feeling unsatisfied. We resolve this dissatisfaction in the next section.

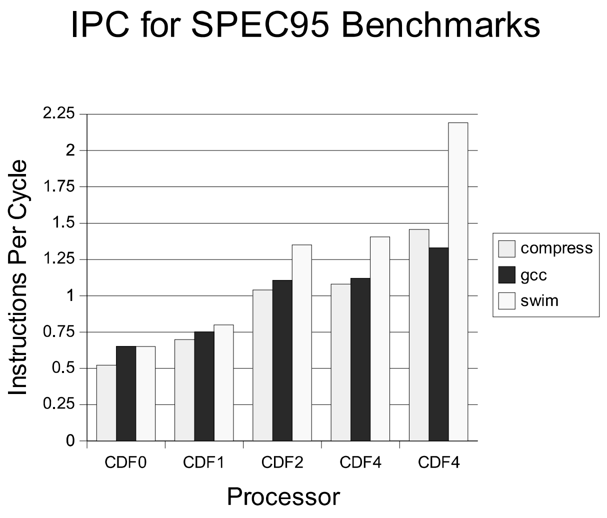

Figure 7: Results of Benchmarks

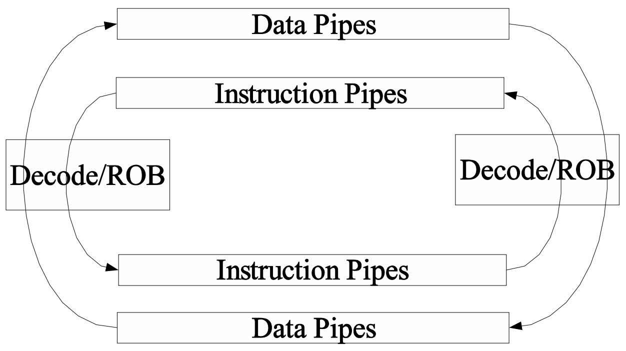

Figure 8: Multi-threaded CDP Architecture

3.3 Advanced Applications

We have seen that the two main apparent areas for possible improvement in the CDF architecture are, first, finding new areas of parallelism to exploit, and second, utilizing the unused upper stages of the pipe. We exploit both of these by attempting hardware multithreading.

In the absence of very little available space left to exploit in the realm of instruction-level parallelism, many processor architects have begun to do multithreading within the hardware. Executing instructions from different threads is especially attractive because one need no worry about interdependancies between threads on an instruction level as there are none! In traditional architectures, this involves scheduling instructions which compete for resources within the processor.

Here we see that CDF is uniquely suited for this problem. Since the upper stages of the pipe are currently underutilized, we can push instructions from another thread into the pipe at just the point where we see a sharp drop in instructions from the current thread. We thus create a pipe with two junctures at which instructions are inserted, as shown in figure 8. In this way, we create new parallelism and use our unused pipe stages effectively with one elegant construction.

Another interesting area for improvement over traditional processing is in the area of data speculation. In previous architectures, speculating a value involved reissuing the instruction or stalling the pipeline if the value proved incorrect. In the CDF architecture, we see that we are once again on fortuitous ground. If we speculate a value, we can simply mark that value as it enters the result pipe. Instructions that use the speculated result can simply remain in the pipe until the actual value is computed. If the actual vaue is different, the instruction can be executed again at the next available sidepanel, if it is correct, the instruction gives the "all clear" signal and is removed from the pipe. The simplicity here provides another strong indicator that the CDF architecture has an eye towards the future.

4 Conclusion

The papers surveyed offer insight into the new pipelining approach known as CFPP. This architecture and its successors have certainly given us a great deal to think about. Though this is a radical departure from traditional processor design, the theoretical evidence supporting the design is hard to ignore. In CDP, the most recent incarnation of CFPP, we see that in addition to minimizing stalls we are able to do multiple issues and even have a way of easily doing data speculation and hardware multithreading. On this level of analysis, the future seems bright.

However, the approach so far fails to convince on the level of empirical practicality. Many of the comments in the reviewed papers indicate that an attempt is being made to introduce a CDF processor alongside (or in replacement of) the current market's slew of superscalar chips. Yet with all the talk of "inexpensive resources", "small size" there is no comparison, even at a very rough level, of the cost of fabricating, or the reletive performance of a CDF processor versus a traditional superscalar IC.

To us, this indicates that this technology is at best still in its infancy and certainly several years yet away from the broader marketplace. However, with a host of new ideas on its side, and the possiblity that such ideas will look even more attractive as we move forward, the counterflow processor architecture seems to be in a prime position to capture the attention of major manufactures in due course.

References

[1] John. L. Hennessey and David A. Patterson. Computer Architecture: A quantitative approach. Third edition.2003

[2] Robert F. Sproull, Ivan E. Sutherland and Charles E. Molnar. Counterflow pipeline processor architecture. Technical Report TR-94-25, Sun Microsystems Laboratories, Inc., April 1994

[3] Michael F. Miller, Kennneth J. Janik, and Shih-Lien Lu. Non-Stalling CounterFlow Architecture. Dept of Electrical and Computer Engineering, Oregon State University.

[4] 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

[5] Michael D. Jones. A New Approach to Microprocessors. Laboratory for applied logic, Department of Computer Science, Brigham Young University. Nov. 14, 1994.