AN ABSTRACT OF THE THESIS OF

Kenneth J. Janik for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on February 27th, 1998. Title: A Microarchitecture Study of the Counterflow Pipeline Principle.

Abstract approved: Redacted for Privacy

Shih-Lien Lu

The counterflow pipeline concept was originated by Sproull et. al.[1] to demonstrate the concept of asynchronous circuits. The basic premise is that a simple architecture with only local communication and control and a simple regular structure will result in increased performance. This thesis attempts to analyze the performance of the basic counterflow pipeline architecture, find the bottlenecks associated with this implementation, and attempt to illustrate the improvements that we have made in overcoming these bottlenecks. From this research, three distinct microarchitectures have been developed, ranging from a synchronous version of the counterflow design suggested by Sproull to an all new structure which supports aggressive speculation, no instruction stalling and ultimately intrinsic multi-threading. To support high-level simulation of various architectures a Java based simulation environment has been developed which was used to explore the various design trade-offs and evaluate the resulting performance of each of the architectures.

©Copyright by Kenneth J. Janik
February 27th, 1998
All Rights Reserved

A Microarchitecture Study of the Counterflow Pipeline Principle

by

Kenneth J. Janik

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented February 27th, 1998
Commencement June 1998

Doctor of Philosophy thesis of Kenneth J. Janik presented on February 27th, 1998

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Kenneth J. Janik, Author

ACKNOWLEDGEMENT

The research presented in this thesis would never have been possible without the assistance and support of my major professor, Dr. Shih-Lien Lu. Over the past years, numerous students have contributed to the Counterflow project. My special thanks go to:

- Jeff Battles
- Ryan Carlson
- Che-Jen Chang
- Troy Conklin
- Rommel Dizon
- Helen Fong
- Dave Heckman
- Wuucheng Huang
- Milo Jueneman
- Yuichi Kishida
- Erik Landry
- Mike Miller
- Balaji Ramamoorthy
- Kim Smith
- Parag Shah
- Grace Teh
- Wendell Woo
- Ramsin Ziazadeh

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

This thesis is dedicated to Josephine Janik...

the strongest person I know.

A Microarchitecture Study of the Counterflow Pipeline Principle

1. INTRODUCTION TO THE COUNTERFLOW PIPELINE PRINCIPLE

The counterflow principle was originated by Sproull et. al. [1] as an architecture for asynchronous processor design. As such, it offers many useful properties including local control, local message passing, and an overall simple design methodology. These same ideas have been taken and used in the design of a synchronous processor [2]. Since the speed of today's processors is being limited more and more by the global signal routing [19], an architecture whose main premise is local control may indeed result in a significant speed increase. This thesis begins by attempting to explain the general counterflow pipeline principle. Intending to show that the local interchange of information and simple design can allow for longer pipelines and increased processor throughput.

1.1 General Pipeline Structure

The basic counterflow principle, illustrated in figure 1.1, has two pipelines flowing in opposite direction from one another [1] [2]. One pipeline carries the instructions up from the fetch or dispatch unit. This pipeline is referred to as the instruction pipeline or simply as the IPipe. The other pipeline carries the operands or results of previously executed instructions down toward the dispatch unit. This pipeline is referred to as the result pipeline or as the RPipe. The main idea behind

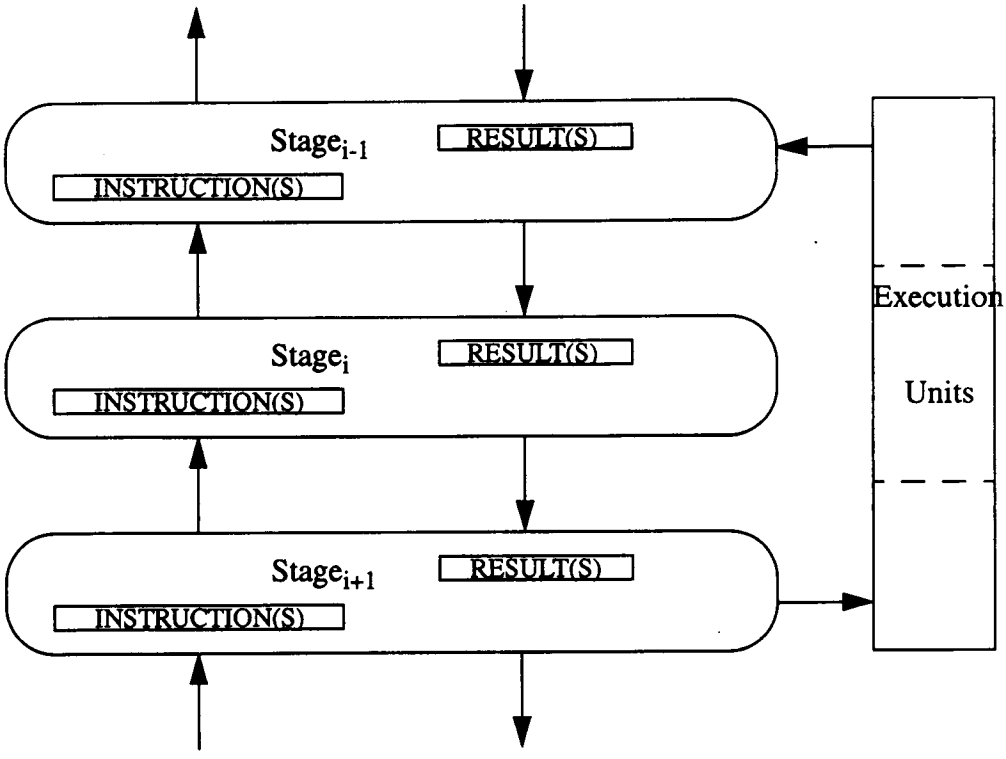

Figure 1.1 Counterflow Pipeline Structure

this structure is that when an instruction and data pass, they "inspect" each other. The instruction checks the operands to see if it needs any of the values. If it does, it takes the operand and carries it with as it proceeds up the instruction pipeline waiting to execute. The operands check the instruction's destination to see if the instruction is going to update their value. If this occurs, the operands have an old copy of the result and they invalidate themselves.

If an instruction reaches its corresponding execution unit launch stage and has all of its operands, it is sent off to the execution sidepanels. Consequently, if it has not received its operands by this stage, it must stall, possibly stalling the

instructions following it in the pipeline. Once the instruction has been sent off for execution, it may proceed up the pipeline. The execution sidepanels are clocked at the same rate as the instructions themselves. Therefore, an instruction's values are always at the same stage as the launching instruction. Upon reaching the associated recover stage, the result of the computation is loaded back into the instruction. The exception to this is the case where the execution unit has a variable latency, such as a memory execution unit. In this case, if the result has not yet been computed, the instruction has to stall at the recovery stage until the result is ready.

At any point after the instruction has retrieved it's result from the execution unit, it will be monitoring the result pipeline for an open slot. A slot is considered empty if it was invalidated by a previous instruction or it is simply empty because it hasn't been filled with anything yet. When an open slot is found, the result will be sent down the result pipeline. Having broadcast the result to the instructions in the pipeline behind it, the instruction will not send the result again.

1.2 Pipeline Internals

The generic pipeline stage acts as an information exchange unit and contains the associated storage and comparison logic for that stage of the instruction and result pipelines. Each pipeline compares what the other has, and takes one or more of three possible actions: [3]

1. The instruction pipeline can take any information it needs for its source operands that is coming down the result pipeline as long as the value in the result pipeline is valid.

2. Information in the result pipeline can be marked invalid if the current instruction is going to eventually write to that register.
3. If there is room in the result pipeline and the instruction pipeline has a newly computed result, the result is sent down the result pipeline.

An example showing these interactions follows. The definitions for the mnemonics used in the instruction and result pipelines are shown in tables 1.1 and 1.2 respectively. OP1 does double duty as both storage for the first operand

Table 1.1 Instruction Pipeline Fields

Mnemonic	Description
OP1	First operand/execution result storage.
OP2	Second operand.
dest	The destination register.
src1	The register for OP1.
src2	The register for OP2.
opcode	The opcode of the current instruction.

Table 1.2 Result Pipeline Fields

Mnemonic	Description
RES	Either an operand or a completed result.
rbind	The register associated with the value in RES.

and for storage of the result of the execution of this instruction. This is possible because once the result has been computed, the operands are no longer needed.

OP2 stores either the value of the second operand or an immediate value if the instruction only requires one operand. The pointer to the register that will eventually store the result of the execution is stored in dest. The src1 and src2 values hold the register numbers associated with the values in the OP1 and OP2 fields. Table 1.2 shows the fields held in the result pipeline. The RES field holds the value associated with the result or operand in the result pipeline. The rbind field holds the register number which the RES field is associated with. The rbind value is used to match with the src1, src2, and dest fields as well as to determine which register to write the value in RES to once the result reaches the register file. Although not shown, each of the values, OP1, OP2, and RES have valid bits associated with them. In general, the result and instruction pipelines can hold more than one result or instruction, but since the interactions are essentially the same, this example will be confined to the simplest case where both pipelines carry only one value.

Now consider an example showing how the instruction R1 = R2 + R3 is executed in a counterflow pipeline structure. Table 1.3 shows the instruction

Table 1.3 Counterflow Example (time t=0)

Pipe	OPC	dest	src1	OP1	src2	OP2	rbind	RES
n-4							R1	47
n-3							R12	4321
n-2							R2	3
n-1							R9	1234
n	ADD	R1	R2		R3		R3	2

entering from the bottom of the instruction pipeline and various values including R1, R2, and R3 flowing down the result pipeline. At this point, the instruction needs the values for src1 and src2 from registers R2 and R3 respectively. Since the result pipeline is holding the values for register R3 which the instruction needs, it will take the value, put it in it's OP2 field and mark the value as valid. In reality, the source operands will take both the value and valid bit directly from the result pipeline as long as the source operand itself is not already valid. This simplifies the comparison logic somewhat as the operand doesn't care if it takes invalid data. Since it is already invalid all that has happened is that the operand now holds different invalid data. It is important to note that since both the instruction and result pipelines are moving, the instruction at pipestage n will "inspect" the result pipeline's values in pipestages n and n-1. This is necessary to prevent instructions and results from synchronously passing each other on a clock cycle. Table 1.4 shows the pipeline states at the next time step. Here, the instruction has proceeded up to the pipestage n-1 and all of the results in the result pipeline have moved down one pipestage. The instruction now has the

Table 1.4 Counterflow Example (time t=1)

Pipe	OPC	dest	src1	OP1	src2	OP2	rbind	RES
n-4								
n-3							R1	47
n-2							R12	4321
n-1	ADD	R1	R2		R3	2	R2	3
n							R9	1234

value of R3 from the result pipeline in it's OP2 field. Correspondingly, the result pipe at this stage holds the value for register R2 which the instruction needs for it's first operand. Once the instruction has this value, it has all of the source operands that it needs to begin executing. Table 1.5 shows the pipeline states

Table 1.5 Counterflow Example (time t=2)

Pipe	OPC	dest	src1	OP1	src2	OP2	rbind	RES
n-4								
n-3								
n-2	ADD	R1	R2	3	R3	2	R1	47
n-1							R12	4321
n							R2	3

after the above actions have been performed. The instruction's OP1 field now holds the value from the result pipeline. At this point, the result pipeline holds the value of register R1. This instruction will create a new value for R1 once it executes, so it is important to invalidate the value in the result pipeline. If this action was not performed, any instructions coming up the pipeline behind the ADD instruction would take the old value of register R1 and would execute with the wrong values resulting in incorrect execution of the program. It is assumed at this point that the Add instruction will execute and that in the next cycle the result will be available. Table 1.6 shows the final state of the pipelines. At this time, three things have occurred. First, from the last clock cycle, the instruction invalidated the old copy of register R1 that was flowing down the result pipeline. This is

Table 1.6 Counterflow Example (time t=3)

Pipe	OPC	dest	src1	OP1	src2	OP2	rbind	RES
n-4								
n-3	ADD	R1	R2	5	R3	2	R1	5
n-2								
n-1								
n							R12	4321

shown by actually removing the value from the result pipeline, but in reality would only involve turning off the valid bit for that result. Second, the add instruction has been executed, and the result (5) has been placed in the OP1 field which now represents the value referred to by the dest field or register R1. Third, the result has been placed in the empty slot in the result pipeline. It will now flow down the result pipeline to give the result to any following instruction that might need the result. Depending upon how the processor is implemented, this result may also be written back to the register file or reorder buffer, but that will be covered more in the following chapters.

The circuitry for accomplishing these tasks is shown in figure 1.2 [2]. This diagram shows the internal circuitry between two stages. The instruction pipeline flows down the pipeline from $stage_{i-1}$ to $stage_i$ while the result pipeline flows up the pipeline from $stage_i$ to $stage_{i-1}$.

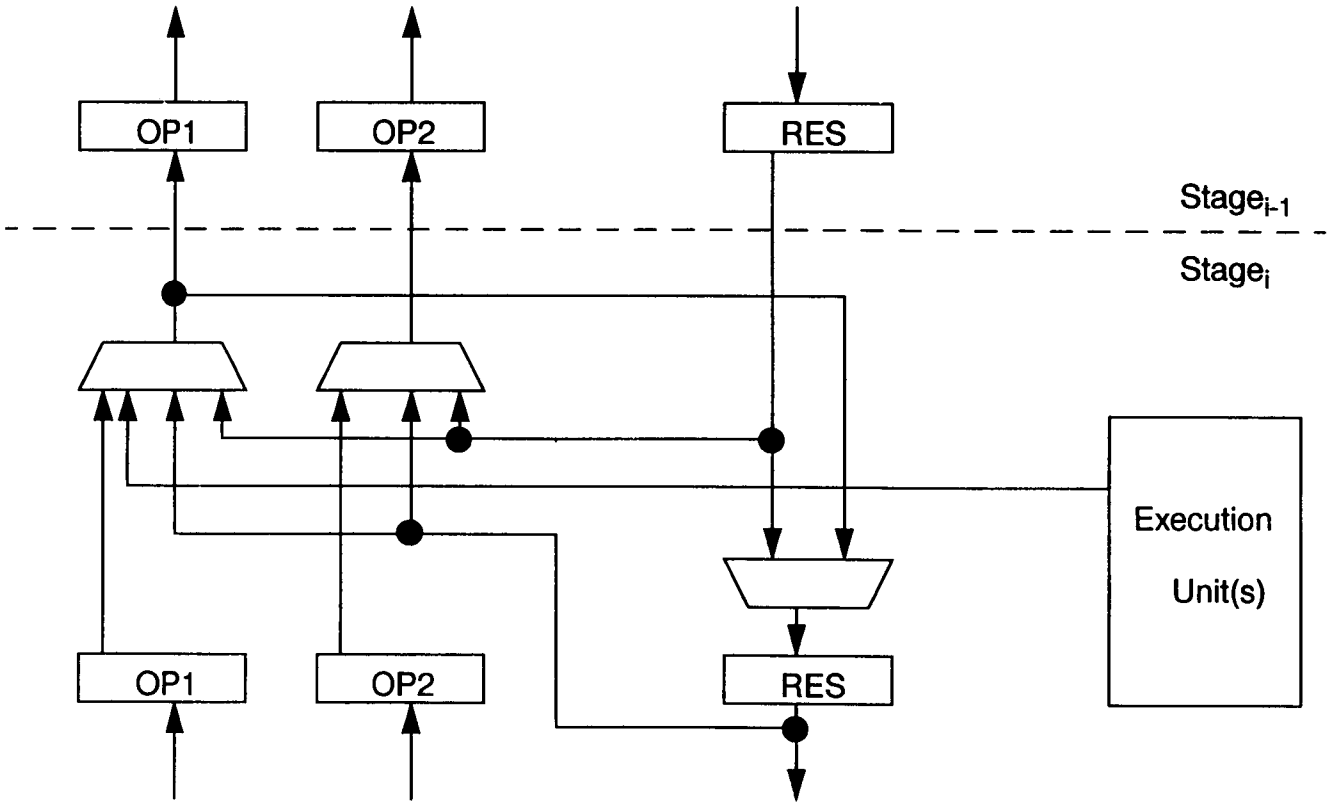

Figure 1.2 Counterflow Pipestage Internal Circuitry

$OP1_{i-1}$ =	RES_{i-1},	if ($src1_i$ == $rbind_{i-1}$) && (RES_{i-1} is valid)
	RES_i,	else if ($src1_i$ == $rbind_i$) && (RES_i is valid)
	{Execution Unit},	else if execution unit is done computing the results
	$OP1_i$,	else [default]

$OP2_{i-1}$ =	RES_{i-1},	if ($src2_i$ == $rbind_{i-1}$) && (RES_{i-1} is valid)
	RES_i,	else if ($src2_i$ == $rbind_i$) && (RES_i is valid)
	$OP2_i$,	else [default]

RES_i =	$OP1_i$,	if (RES_{i-1} is invalid) && ($OP1_i$ is a result)
	RES_{i-1} (invalid),	else if ($dest_i$ == $rbind_{i-1}$)
	RES_{i-1},	else [default]

The new first operand,$OP1_{i-1}$, will either be inspecting the result pipeline looking for results which it needs to execute, looking for the execution unit to return a result for it's instruction, or looking for an empty spot in the result pipeline to put it's completed result into. The new second operand, $OP2_{i-1}$, is only looking for operands in the result pipeline that it needs to execute. The new result, RES_i, is either taking a completed result from the instruction pipeline, or invalidating itself if it holds an old copy of a result that the current instruction will eventually overwrite.

1.3 Execution Unit Interactions

An example of the execution units' interactions with the pipeline is shown in figure 1.3 [13] [14]. In this example, there are two integer addition units, one

multiply unit, and one divide unit. At the bottom stage, the pipe can launch an add, multiply, or divide depending upon the opcode of the instruction.

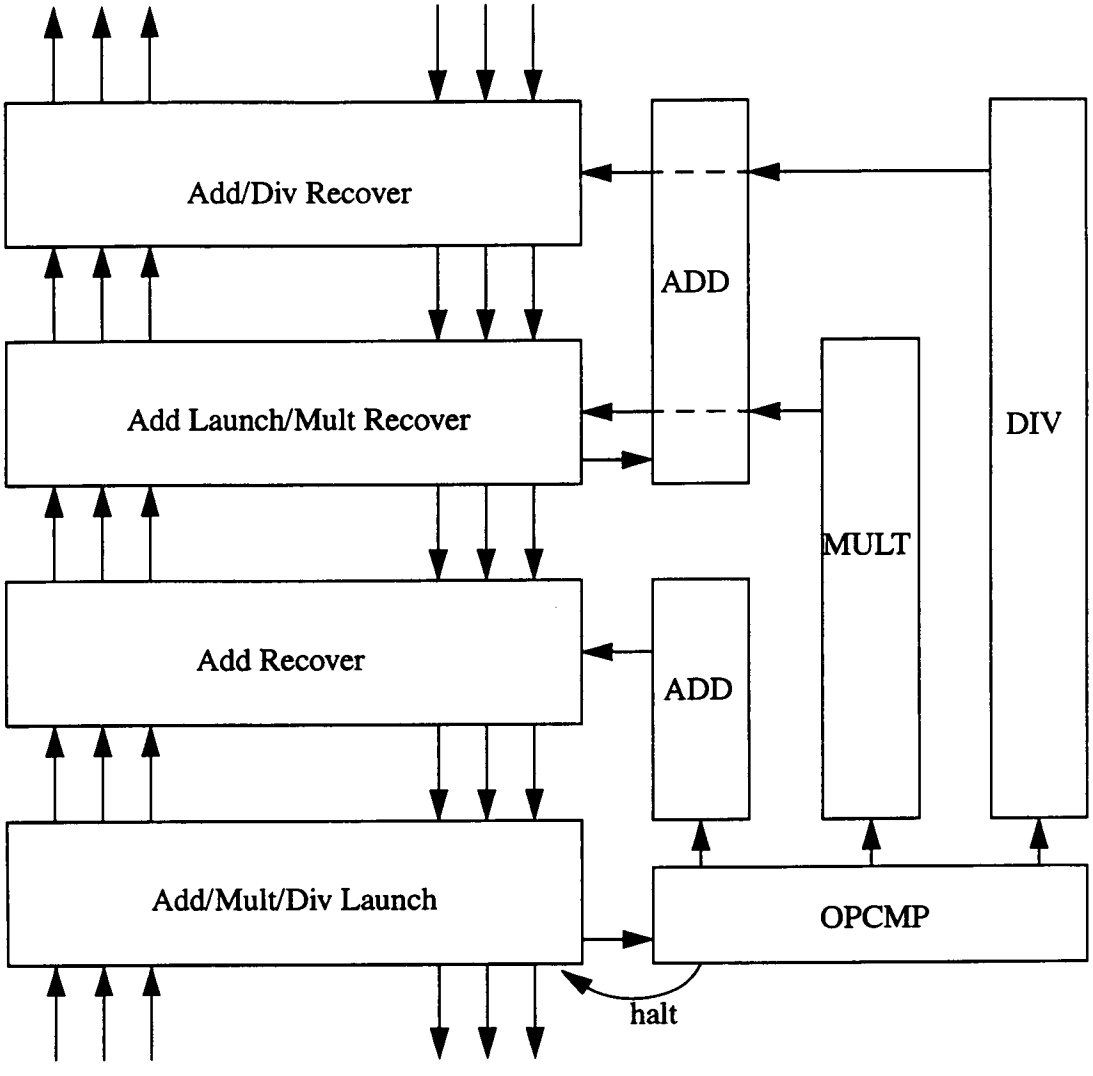

Figure 1.3 Example Execution Unit Interactions

The pipeline itself doesn't make the decision of whether or not to launch. This is the job of the OPCMP (OPerand CoMPare) unit [14]. This specialized unit

knows what execution units are available at this stage, and whether or not there are duplicate execution units at a later stage. The generic pipeline stage gives the OPCMP unit the opcode of the instruction and a one bit signal which is the logical AND of the two operand valid bits. The only signal returned from the OPCMP is a halt signal which is asserted if the unit realizes that this is the last stage from which the instruction can be launched and the instruction has not yet received both of its necessary sources. For example, if an add instruction were to arrive at the bottom stage and it didn't yet have both of its operands, the OPCMP would realize that there is another addition unit later on and wouldn't halt the instruction thus giving it more time to obtain its operands and not stalling the pipeline. However, if a multiply or divide instruction were to arrive at this stage without its operands, the OPCMP unit would be forced to stall the stage until the operands had been received from the result pipe since these are the only multiply and divide execution units in the processor.

1.4 Overall Counterflow Pipeline Structure

The structure show in figure 1.4 illustrates the general design of a counterflow processor [1] [2]. Depending upon the targeted software market, the ordering and number of execution units may vary, but the basic flow is still the same. There are several items that need special attention in this architecture.

First, note that the memory stages launch and recover (for a first level cache hit at least) before the branch execution unit [5]. Precautions need to be taken to ensure that a store is never written to permanent memory based upon an incorrectly predicted branch. This means either using a write-back cache and

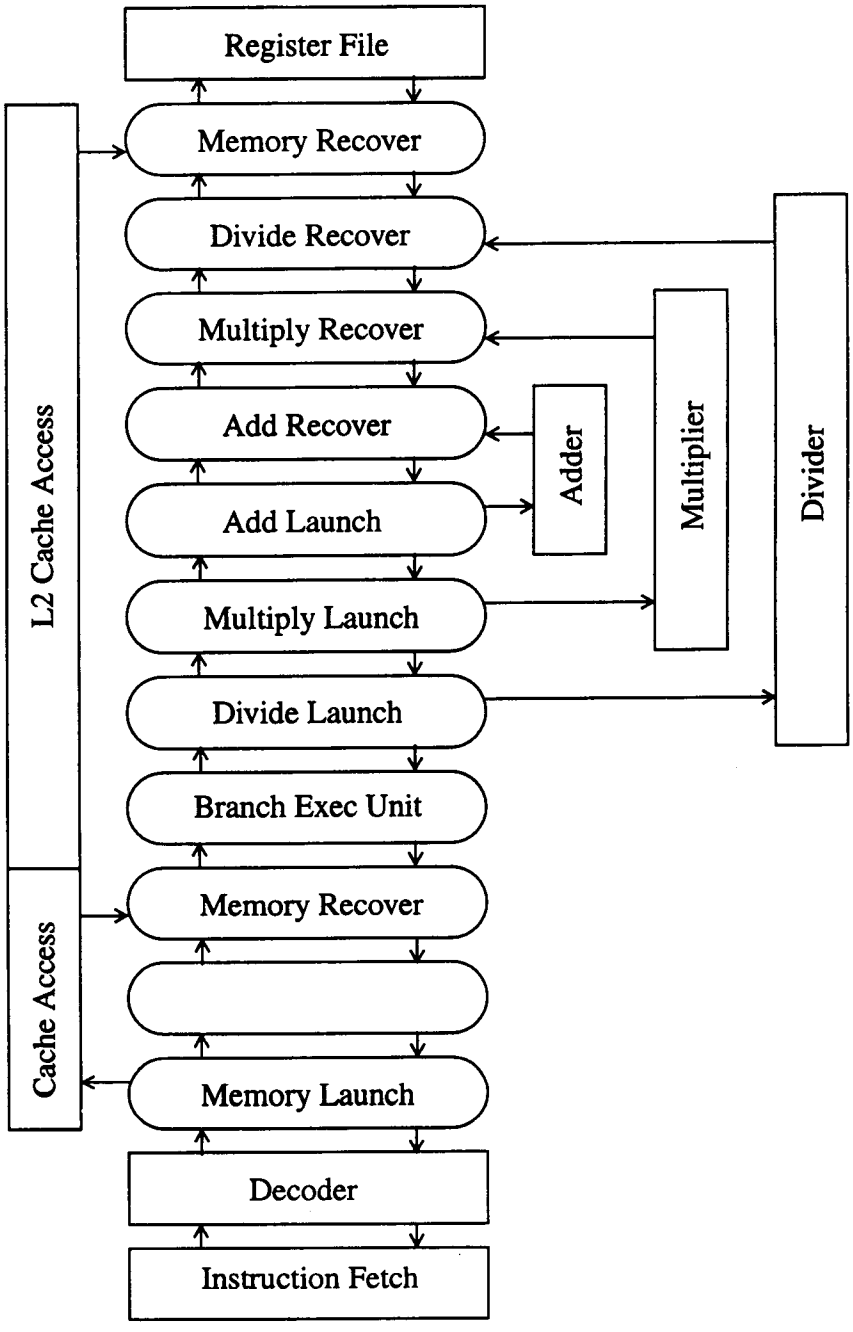

Figure 1.4 Overall Counterflow Pipeline Structure

flushing the cache block whenever there was a store and a bad branch taken or finding some other means of forcing the store operation to complete after the branch is known to be good. In other, more advanced versions of counterflow, a combination of reorder buffer [18] and memory order buffer are used to keep track of which stores are allowed to be written to permanent memory.

Second is the actual location of the branch execution unit. This is a unit which decides whether or not the branch direction chosen was indeed the correct one. If it is placed too far up the pipeline many instructions are needlessly executed when the prediction is incorrect, costing clock cycles to execute the instructions as well as to remove the instructions from the pipeline. If it is placed too soon in the pipeline, the results coming from the execution units higher up in the pipeline will take a long time to get to the branch unit and the instructions will have to stall, again harming performance. It is important to note that a branch prediction algorithm with a high prediction rate is mandatory for a pipeline as deep as counterflow [8].

Finally is the ability to use the pipeline to hide the latency associated with a cache miss. Given a first level cache [4] miss, the counterflow pipeline allows the instruction extra cycles to obtain the data from the level two cache. Even if the instruction must stall waiting for the level two cache to respond, instructions earlier in the pipeline can hopefully continue doing useful work. If the level two cache misses and main memory cannot respond in time such as during a page fault, it will become necessary to flush the pipeline and bring in a different process/thread to execute in the time needed to get the data from the other levels in the memory

hierarchy. Fortunately, most of the time, the needed data will be in the level one or level two caches [8].

1.5 Pipeline Flushing

In the case of a mispredicted branch instruction, all of the instructions that have been executed since the branch must be flushed from the pipeline [16]. In a counterflow pipeline, the implementation of this entails having a global signal coming from the pipestage with the branch execution unit and running down the pipeline back to the prefetch and branch prediction unit. Those pipestages contain the instructions in the shadow of the mispredicted branch. When the branch execution unit detects a misprediction, it immediately signals these stages who then invalidate the instructions that they are holding. The program counter, upon receiving this signal begins to fetch instructions from the branch path that it didn't take before, and the branch prediction unit updates its algorithm so that hopefully the next time it encounters the branch it predicts correctly.

1.6 Chapter Summary

The implementation details for a counterflow structure vary with each version of the microarchitecture, but the basic principles remain the same. There are two pipelines flowing opposite each other, one carrying the instructions up and the other carrying the results down. The length and number of elements that each of the pipelines can carry will change with each implementation. Since the communication within a counterflow pipeline is localized to only those stages directly adjacent, the clock frequency of the processor can be very high.

2. THE COUNTERFLOW PIPELINE PROCESSOR (CFPP)

The counterflow pipeline processor (CFPP) was developed as an architecture which lends itself to being implemented with asynchronous hardware [7]. The architecture exhibits properties including local control, regular structure, local communication, modularity, and overall design simplicity [1]. Although it was designed to be implemented with asynchronous circuitry, these characteristics also benefit a synchronous design. The decision to implement this version of the counterflow pipeline processor as a standard synchronous pipeline was based on several factors, not the least of which were familiarity with synchronous design techniques and availability of synchronous design tools. Figure 2.1 shows the basic architecture of a counterflow pipeline processor [2]. As the simulation studies performed were done at a microarchitecture level, they apply equally to both synchronous and asynchronous implementations, it is only the underlying circuit implementation that would change.

2.1 Architecture of a Counterflow Pipeline Processor

This section builds upon the general information covered in the previous introductory chapter, specializing in the implementation of the original counterflow pipeline processor. Referring to figure 2.1, there are two pipelines, the instruction pipeline and the result pipeline. The instruction pipeline carries instructions from the fetch/decode unit up toward the register file. The result pipeline takes results or source operands from the register file down the pipeline toward the fetch/ decode unit. Along the way, instructions and results interaction and inspect each

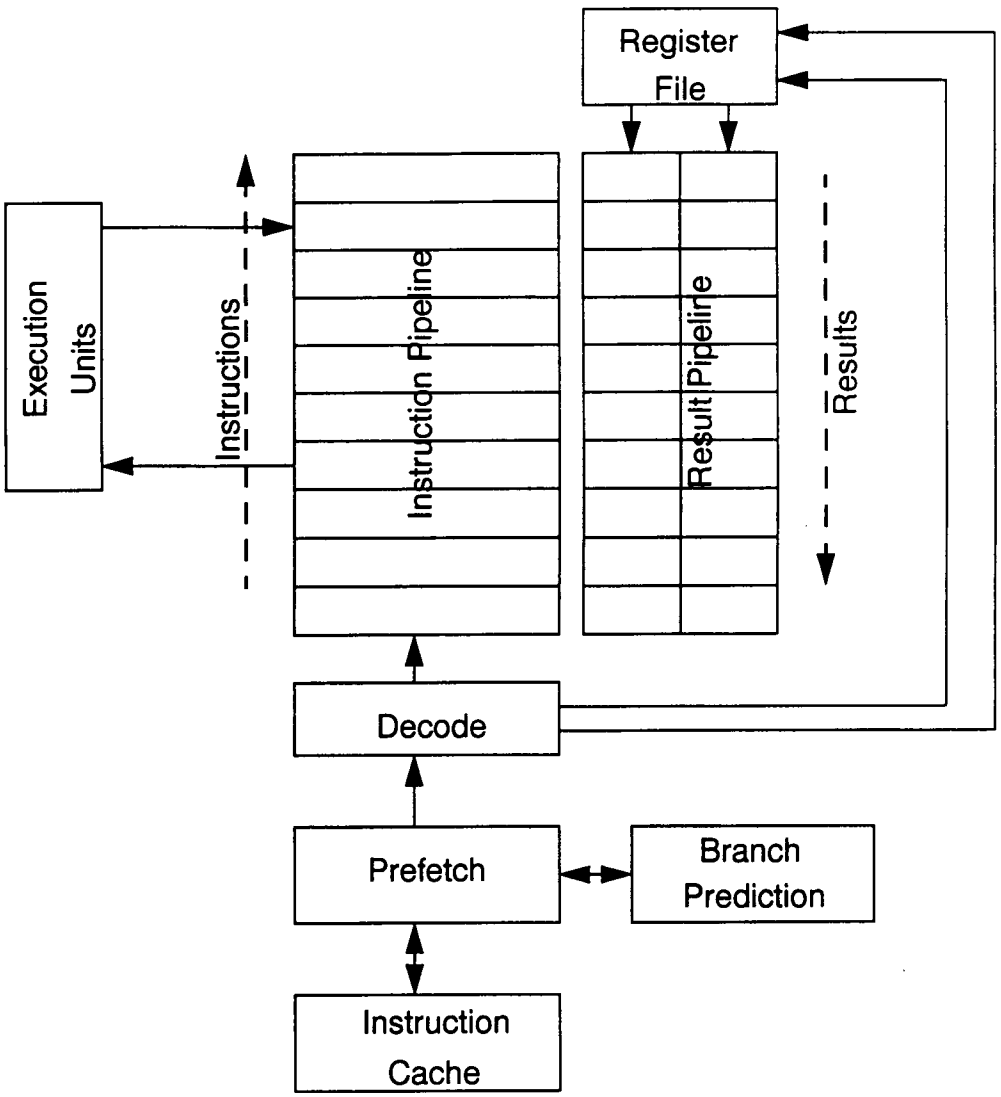

Figure 2.1 Counterflow Pipeline Processor Architecture

other. If an instruction needs an operand in order to execute, it watches the results that flow past it in the result pipeline and grabs whatever data it needs.

Once the instruction has all of the data that it needs to execute, it sends the instruction off to the execution units to calculate the result. As the result is being

executing, the instruction continues up the pipeline. When the instruction arrives at the execution unit's recovery point, it takes the result from the execution unit if the execution has completed. At this point, the instruction is not allowed to leave the pipeline even though it has technically completed it's execution. As the instruction continues up the instruction pipeline, it looks for empty spots in the result pipeline in which to put it's result. This is necessary so that any instructions following this instruction up the pipeline get the correct results [3]. While the instruction is flowing up the instruction pipeline, it is observing the result pipeline for a result which it is eventually going to overwrite. The value in this result is an old value from some previous computation, and is not valid for the instructions behind this instruction since they need the result of this instruction's execution. The instruction itself is responsible for invalidating any results which it sees that are old copies of the result it is generating. As a result of this interaction, the instruction only needs to send the result down the result pipeline once.

The instruction now continues up the pipeline, writing it's result into the register file once it reaches the top of the instruction pipeline. At this point, the instruction has finally finished executing. It is important to note that during this process the instruction pipeline can stall at three locations. The first location is at the launch point to the execution unit. If the instruction gets to the launch point of the last execution unit which can execute this type of instruction, but the instruction does not yet have all of it's required operands, it must stall waiting for the operands to show up from the result pipeline. The second time an instruction must stall is at the recovery point for execution units which have an unknown

execution latency (such as memory units). The instruction must stall at this stage and wait for the result to return. This causes bubbles, or empty slots, to be formed in the instruction pipeline above this instruction. The instructions following the stalled instruction must themselves stall unless there happens to be an empty bubble which can be "squashed" to allow the instructions to continue making forward progress and therefore doing useful work [1]. While the instruction pipeline can stall in these cases, the result pipeline is never allowed to stall. If it did, and the instruction pipeline also stalled, the stalled instruction could conceivably never receive it's result, and the entire processor would deadlock. The only other time an instruction can stall is at the top of the instruction pipeline. If the instruction gets to the top of the pipeline and has not yet been able to put it's result into the result pipeline, it must stall until it is able to do so. If the instruction were to just go ahead and write into the register file without sending its result down the result pipeline, there could be an instruction behind this instruction which needs the result. This other instruction would not be guaranteed to get the result it needs because the result will not come out of the register file again unless another instruction needs that result. If no other following instructions need this result, the result will sit in the register file, the instruction will stall waiting for the result, and the processor will again deadlock.

2.2 Simulation Results

The choice of execution units in a counterflow processor has never been addressed in published literature before. The number, placement, and latency of execution units to give the best performance is as yet an open question.

Therefore, to attempt to come up with an "acceptable" configuration began is a matter of trial and error and involved running many simulations with various configurations. The traces used consisted of the first two million instructions of ten Spec95 [20] traces which had been compiled using the SimpleScalar [10] [12] toolset. There are five integer benchmarks and five floating point benchmarks. The configuration which was eventually decided upon is simply the best solution that could be found with a reasonable amount of processor cycles for these particular benchmarks, it is in no way the absolute best solution. It may be possible to formally find the best configuration for a given category of programs, but that type of research is beyond the scope of this thesis.

During the course of experimenting with various configurations, some heuristic methods were recognized as leading towards a good configuration. Observing the locations of stalls is one very good way for finding where the bottlenecks are. If, for example, the last fast integer execution had an extraordinarily large number of launch stalls. There are several possible causes for this observation. There may be a need for another unit of this type. This is good up to a point where the area invested yields diminishing returns. Another possibility is that the last unit of this type needs to be moved farther up the pipeline. This is easy and cheap, but can cause other problems for instructions which have a strong dependency on this unit, causing the stalls for that unit to increase. The last possibility is that another unit which a lot of adds are dependent on is recovering too late in the pipeline to give the results back. That unit's recovery point can possibly be moved lower in the pipeline, but that can

cause more problems also. Generally, pipeline optimization is a gentle balancing act with no hard set rules.

The pipeline configuration which was decided upon is shown in Figure 2.2. This configuration consists of a one instruction wide instruction pipeline and a four result wide result pipeline. The result pipeline, although four results wide, should probably be considered only two results wide. The result pipeline width had to be doubled in the simulator to account for the SimpleScalar instruction set which for double wide floating point instructions can request four operands for one instruction [12]. In a real implementation, the number of bits in the operands themselves would probably be doubled instead of the result pipeline width. Nevertheless, the simulation results show that on average there are less than two results in any stage of the result pipeline.

There are three fast integer units (INTF01-INTF03). These units have a one cycle latency and handle instructions such as ADD, SUB, Shift, etc. There are two branch execution units (BEU01 and BEU02). The branch execution units have a one cycle latency, and communicate the results of the branch back to the branch prediction unit where the prediction algorithm is updated and if the prediction was incorrect, the recovery process is initiated. There is one slow integer unit (INTS01). It has a latency of four cycles, is fully pipelined, and handles slow integer instructions such as multiply and divide. There is one fast floating point unit (FPFAST). It has a latency of four cycles, is fully pipelined, and handles fast floating point instructions such as floating point addition and subtraction. There is one slow floating point unit (FPSLOW). It has a latency of

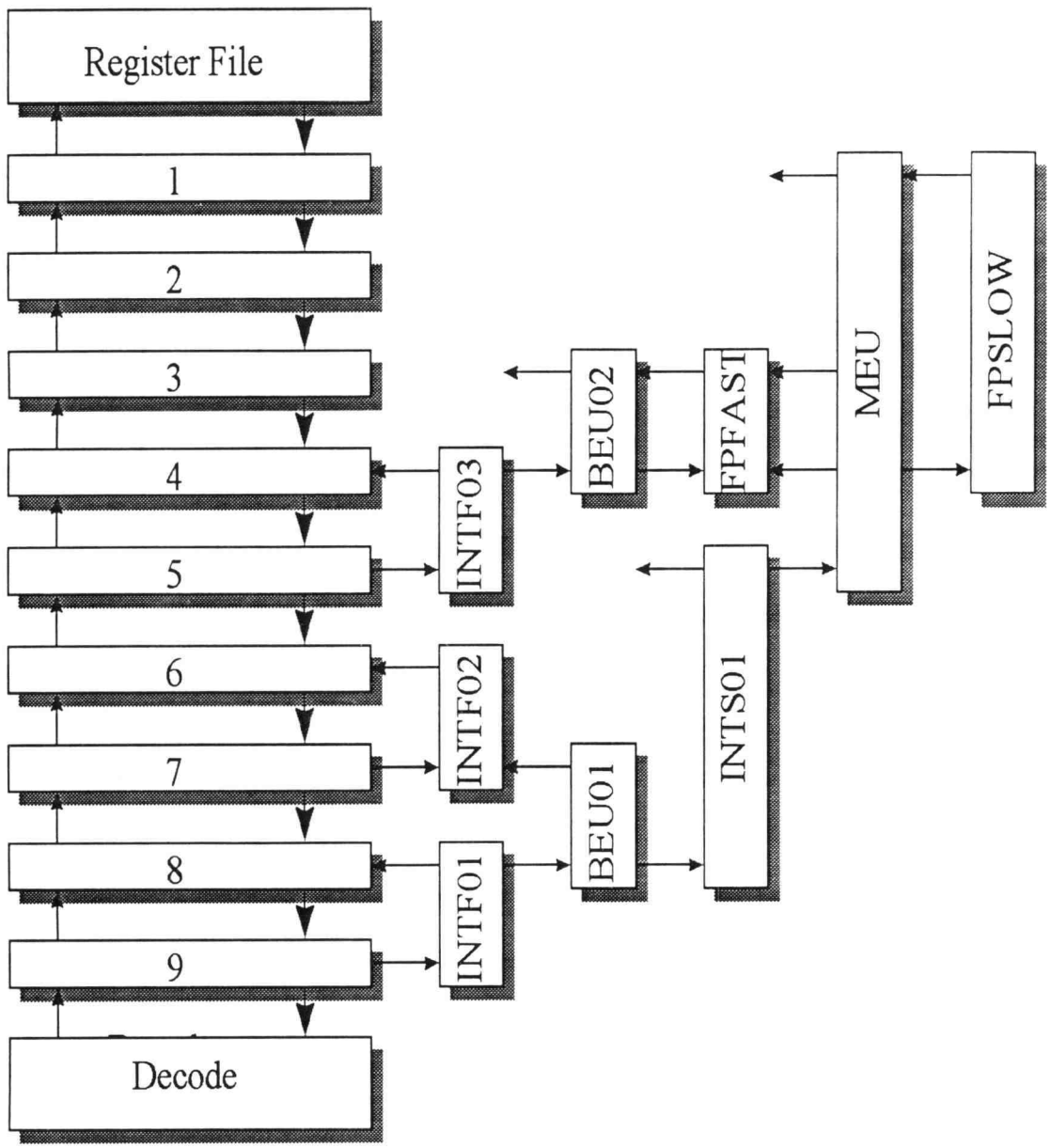

Figure 2.2 Simulated CFPP Pipeline Configuration

eight clock cycles, is fully pipelined, and handles slow floating point instructions such as floating point multiplication and division. There is a Memory Execution Unit (MEU) [4], which handles load and store operations. Since there is no

reorder buffer in a CFPP architecture, the MEU sees the instructions in order and doesn't allow dependent instructions to pass each other. There is one level one (L1) cache, not pictured, which is a 16KB, 4-way set associative data cache, with one cycle access time, and SLRU replacement policy.

The following assumptions have been made to allow for a higher level simulator. It is assumed that the L1 cache and main memory hold all necessary data. The main memory has a constant 10 cycle access latency. The branch prediction has a randomly predicted 94% correct branch prediction [8]. When recovering from a mispredicted branch, there is a one cycle "no fetch" penalty imposed on the prefetch of the next instruction after the misprediction.

There is a small trick which was used to increase the performance of this architecture. The fast floating point unit has an execution latency of four clock cycles, but in figure 2.2, the unit only spans two pipestages. This means that the instruction pipeline has to stall for two clock cycles every time a fast floating point operation is performed. Fortunately, the recovery point is beyond the launch point for all the other execution units. In this way, all the integer and control code can continue executing in the bottom half of the pipeline while the top half has stalled. Similarly, the slow floating point unit has an execution latency of 8 clock cycles, but spans four pipestages. This causes four stalls to occur, but they happen in the last pipestage allowing even more room for other instructions to continue work at the bottom.

Figure 2.3 shows the performance of the ten Spec95 traces on the CFPP configuration shown in figure 2.2. The performance is expressed in average

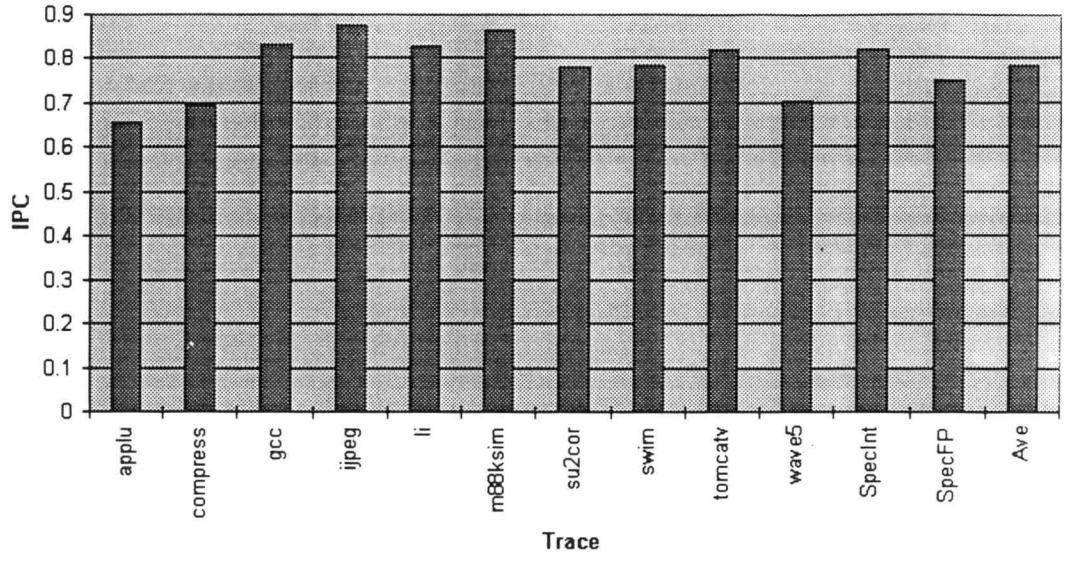

Figure 2.3 CFPP Average Instructions Per Clock Cycle by Trace

instructions per clock cycle for the first 2 million instructions of each trace. Average performance for the SpecInt95 traces at 0.82 was a reasonable amount higher than the SpecFP95 trace at 0.75. The program with the highest performance was the integer benchmark, *ijpeg*, with an IPC of 0.88. The lowest performance was by the floating point benchmark, *applu*, with an IPC of 0.66.

It is interesting to observe the execution unit usage. This shows the efficiency of the architecture as well as serving as a benchmark with which to compare the other architectures with to see how the changes made to the microarchitecture affect the efficiency with which the execution units are being used. Figure 2.4 shows the percent of time the various execution units were busy averaged across both the integer and floating point traces. The slow integer unit

Figure 2.4 CFPP Execution Unit Usage

(INTS01) was almost never used since there were almost no slow integer instructions present in any of the traces. The memory execution unit (MEU) and the last fast integer unit (INTF03) were the busiest for both the integer and floating point traces with an average of 23% and 22% respectively. The floating point traces also spent a good deal of time in the fast and slow floating point units, 16% and 8% respectively. The information gathered from the execution unit usage was used to decide on the placement and number of execution units. For instance, there are three fast integer execution units. By removing any one of the three, the load gets shifted too much to the remaining two. Removing one of the units causes instructions to stall unnecessarily early in the pipeline if one of the later units is removed or to be evaluated later than necessary if one of the early units is

removed. As always, this is a balancing act, with the ultimate aim being the best performance for the least hardware cost. In a similar manner, the branch execution units were placed. The first branch execution unit (BEU01) is placed very early in the pipeline. This unit executes all of the unconditional branches as well as most of the branches which depend on fast integer results. Since this unit is early in the pipeline, incorrectly predicted branches do not cost as much to recover from. Conversely, the last branch execution unit (BEU02) is placed relatively high in the instruction pipeline. By this point, most data dependencies have been resolved. Unfortunately, incorrectly predicted branches cost more to recover from, but since the branch prediction unit usually predicts correctly, it is better to allow the pipeline to continue doing hopefully useful work rather than stall and certainly do no useful work.

Figures 2.5 and 2.6 show the pipeline usage for the instruction and result pipelines respectively. The instruction pipeline usage shows that the instruction pipeline is heavily used over the entire length. Starting at pipestage nine next to the fetch/decode stage where there is almost always one instruction and dropping below 0.7 instruction on average only at the second to last pipestage. The average instructions go up in the last pipestage because on occasion, instructions may have to stall at the last pipestage if they haven't been able to deposit their results into the result pipeline. The pipestage usage is expected to be high for CFPP since instructions have to remain in the pipeline from start to finish to carry the results to the register file at the top of the pipeline. This hinders performance since there are very few empty locations to be "squashed" when an instruction

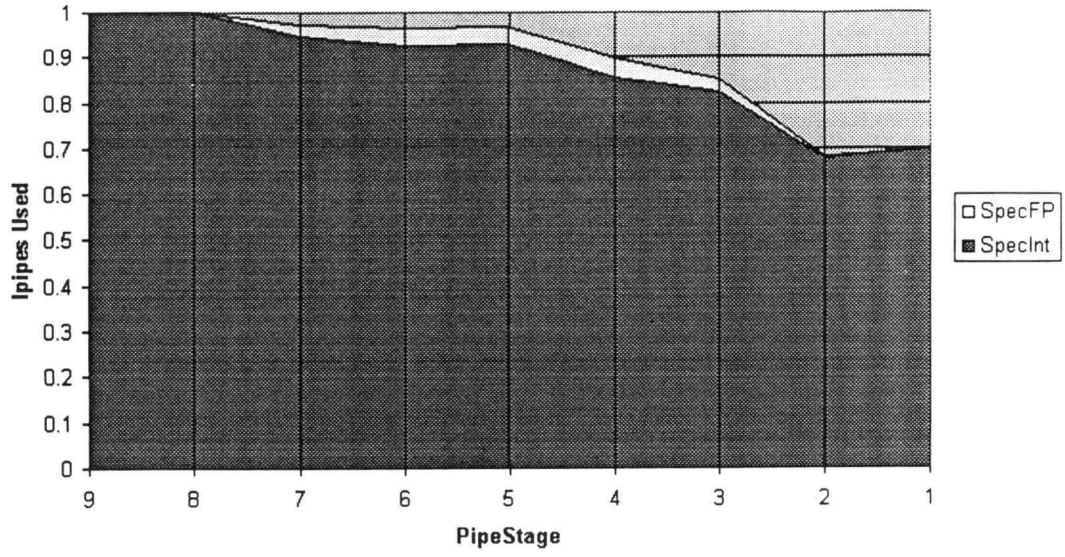

Figure 2.5 CFPP Instruction Pipeline Average Usage

stalls [3]. When there are no empty locations, the entire pipeline must stall behind any instruction which stalls.

The result pipeline's usage goes from a high of 1.8 results per stage at the bottom of the result pipe for floating point traces to a low of 1.2 at the top of the result pipe for integer traces. The result pipeline must carry the operands from the register file the entire length of the pipeline as well as the results of instructions from where ever they were computed to the bottom of the pipeline. Considering the amount of data which the result pipeline is being required to carry, the amount of usage is reasonable. Rising at the bottom where both the results and operands are in the pipeline and falling at the top where only the operands are in the pipeline. With all this traffic, it is sometimes difficult for an instruction to find an empty location in which to put it's computed results. This again, causes an

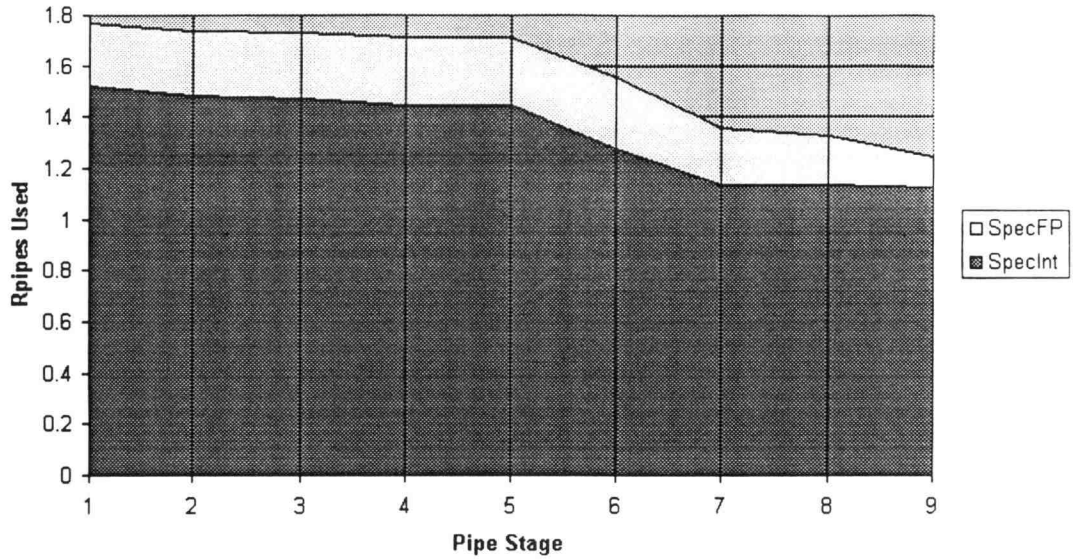

Figure 2.6 CFPP Result PIpeline Average Usage

increase in the usage of the last instruction pipestage as well as lowering the performance since the instruction pipeline must stall to wait for an empty result pipe location.

Figure 2.7 shows the percentage of time the instruction pipeline's stages are stalled for both integer and floating point traces. The first half of the pipeline is stalled about 23% of the time for floating point traces and 15% of the time for integer traces. The fifth pipestage is the last location in which to launch a fast integer or memory instruction. Since all instructions are required to stay in the pipeline in order to write their results to the register file, there is almost no chance that there is an empty location behind this stalling instruction and therefore every instruction stage lower than the fifth pipestage usually has to stall also. This is a

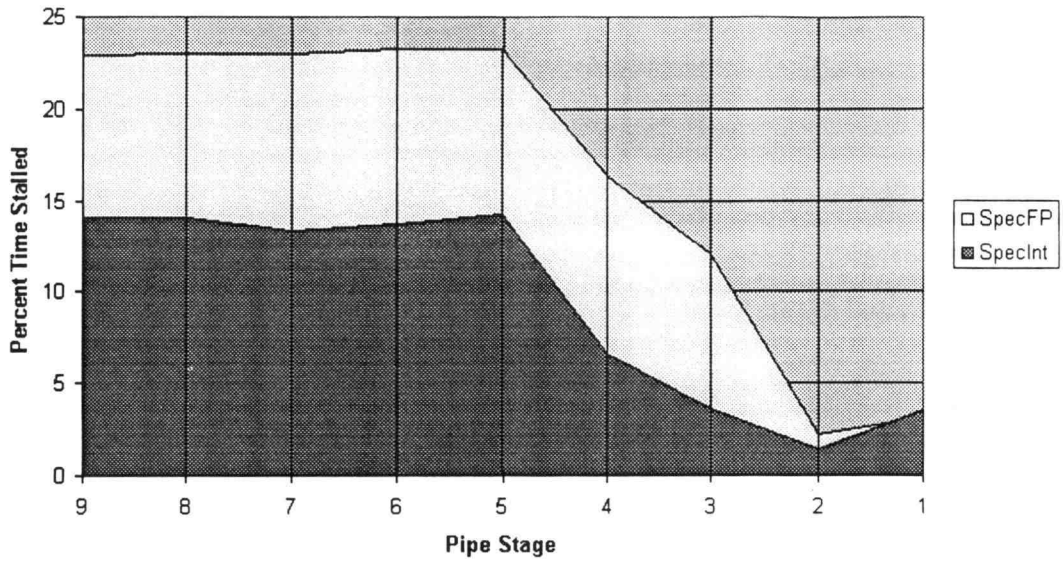

Figure 2.7 CFPP Stalling Locations

huge performance limitation and eliminates a major reason for using a counterflow pipeline. Fortunately, the next two chapters describe enhancements to the microarchitecture which overcome this problem. Once past this point, the number of stalls drops off drastically. The top two pipestages actually illustrate how a counterflow pipeline with empty locations should respond. The last pipestage, stage number 1, stalls about 3% of the time to write it's results into the result pipeline. Because of the large amount of stalling which occurs early in the pipeline, there are empty locations in pipestage number 2. This allows the instructions in stage 3 and below to continue to move and therefore do useful work while the instruction in stage 1 is stalled.

2.3 Problems with Original CFPP Implementation

As might be expected this being the first implementation of a new architecture, there are several problems. The first problem is that the register file is placed at the top of the pipeline, at the opposite end from where the instructions are first fetched [11]. The second problem is placement and use of global signals for the branch execution unit and halting mechanism. The last problem is a lack of tolerance of long latency operations such as memory instructions.

2.3.1 Register File Placement

In the original counterflow pipeline processor, the register file is at the far end of the processor away from the where the instruction is fetched from. This causes a long start-up delay since an instruction's operands have to come halfway across the pipeline before there is even the possibility of an instruction being able to execute. This leaves the first half of the pipeline empty whenever the pipeline has to flush because of a branch misprediction or a new thread starting up. As has been shown, this also contributes to congestion in the result pipeline since the operands from the register file must flow down the entire result pipeline, thus competing for space with newly computed results. To compensate, the result pipeline needs to be unnecessarily wide. If it were possible to move the register file from the top of the instruction pipeline to the bottom, the result pipeline width could be reduced. The register file placement also affects the instruction pipeline. Since instructions need to stay in the instruction pipeline just to carry their results to the register file, they take up space in the pipeline. Because of this, most of the time when an instruction stalls there are no empty slots available to be

"squashed" and all instructions behind the one which has stalled are also forced to stall. This negates one of the main benefits of a counterflow pipeline, the "squashing" of empty slots to allow computation to continue while an instruction is stalled [1] [2].

2.3.2 Branch Resolution and Placement

The branch execution unit's placement can cause problems with the processor's execution efficiency. Ideally it would be best if the branch execution unit could be placed near the top of the pipeline since the branch instruction would most likely have its operands and wouldn't have to stall the pipeline. Unfortunately, whenever there is a misprediction, the entire pipeline from the branch execution unit down to prefetch needs to be flushed. By spending the time to execute all of these instructions which should never have been brought into the pipeline a lot of work has been wasted which can get very expensive if the branch prediction rate is not high enough. On the other hand, placing the branch execution unit too near the bottom of the pipeline means that very often it won't have it's operands and will stall the entire pipeline. This can cause unwarranted stalling since a good branch prediction unit will guess correctly most of the time.

2.3.3 Global Signals Limit the Clock Speed

One of the main premises for using a counterflow pipeline as a high speed processor microarchitecture is the removal of global control signals [1]. Indeed local control and communications are some of the basic founding principles behind counterflow. Unfortunately, the microarchitecture of CFPP as it stands

currently relies on a few critical global signals which run right down the middle of the pipeline logic. The first signal, is the halt signal. This signal is in the instruction pipeline. Whenever an instruction stalls, it must tell the pipestage behind it that it is stalling. This pipestage in turns looks at it's contents and decides if it too must stall. It then propagates this signal to the pipestage behind it. This signal can potentially run the entire length of the pipeline if the instruction at the top of the pipeline must stall and there are no empty slots in the pipeline. Since this worst case can occur, this is delay path must be accounted for. This would almost certainly be the most critical signal in the processor and would have to be routed from one end of the pipeline to the other [2].

The other main global signal is the pipeline flush signal. Whenever there is a mispredicted branch, the branch execution unit must flush the pipeline to clear out all of the incorrectly speculated instructions. This signal is less critical than the halt signal since it only has to go from the last branch execution unit back to the start of the instruction pipeline. Also, since there is no associated logic at each stage this signal doesn't need to propagate like the halt signal does. It is however a global signal which must travel long distances over the die and negates some of the benefits of having local control and communications everywhere else on the processor.

2.4 Chapter Summary

In this chapter, the original counterflow pipeline processor (CFPP) suggested by Sproull [1] has been simulated. By investigating the characteristics of CFPP, some hidden problems in implementing such a microarchitecture have

been uncovered. While the performance is respectable, there are several global signals which would defeat some of the reasons for developing the counterflow architecture. With the register file at the top of the instruction pipeline, the result pipeline has to be wider than otherwise necessary. Instructions also have to carry the results the entire length of the instruction pipeline to write the results into the register file. Since the instructions remain in the pipeline the entire length of the instruction pipeline, CFPP does not deal well with long-latency instructions. If a load or store misses the first level cache, most of the instructions behind it will have to stall until the next level of memory can respond since there are very few empty locations in the pipeline. These problems cause the performance to be less than expected while causing the pipelines to be used relatively inefficiently. Fortunately, many of these shortcomings have been overcome in the next architectural implementation, the virtual register processor [3] [11].

3. THE VIRTUAL REGISTER PROCESSOR (VRP)

The virtual register processor (VRP) came about by attempting to overcome the problems with the original CFPP design [3] [11]. The most notable limitation being having to wait half the length of the pipeline to get operands from the register file. Although the operand has already been computed and can be used, it must travel at least half the length of the pipeline to meet its instruction. If, as in figure 3.1, there is an instruction which has an available execution unit, but is only waiting on its operands, the entire pipeline may stall needlessly. In this case,

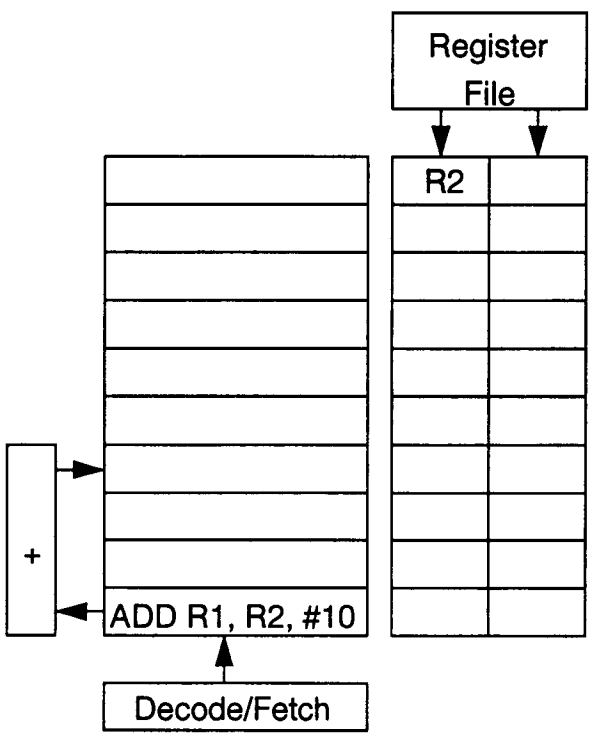

Figure 3.1 Register File Placement Problem in CFPP

the instruction needs to add the value in register R2 to the immediate value 10 and place the result of the execution in register R1. Since the value of register R2 has been computed much earlier, there is no reason this instruction cannot complete and give the result to a future instruction. Unfortunately, in CFPP, the instruction and data must meet in the pipeline for the transfer of information to occur. For the situation illustrated in figure 3.1, this can cause this instruction and all following instructions to stall at or near the bottom of the pipeline waiting for this already valid result to make it's way down the result pipeline.

3.1 Rationale For Changing

The problems associated with CFPP in the previous chapter severely limit the performance achievable with this architecture. By moving the register file from the top of the pipeline down to the bottom, many of the ill-effects were removed [3]. With the register file at the bottom, values already computed are available at the time of instruction launch. Additionally, the instruction pipeline no longer needs to carry the completed instructions the entire length of the pipeline just to be able to write the result into the register file. Unfortunately, there are some changes that need to be made to overcome the problems created by moving the register file.

3.2 Architectural Changes From CFPP

As was stated in the previous sections, the main architectural change associated with the virtual register processor is the location of the register file. While this appears to be a straightforward change on the surface, it has

ramifications which cause some problems and change the way that the processor functions. Figure 3.2 shows the microarchitectural layout of a virtual register pipeline processor. Note the addition of a re-order buffer and location of the register file.

3.2.1 Register File and ROB

The register file has been moved from the top of the pipeline to the bottom. Now, operands in the register file are available immediately. Unfortunately, the register file now needs to function more as a register cache. To facilitate this change, a reorder buffer (ROB) [18] has been added. The functioning of the ROB will be described in detail later. The addition of a ROB adds hardware as well as complexity, but a ROB is no longer considered extravagant hardware in today's microprocessors [8]. The ROB is needed to force the instructions to retire in the order in which they were issued into the processor. It also serves to keep instructions which were incorrectly speculated from writing their results to the register file. This can occur after a mispredicted branch instruction as well as after a fault or interrupt has been triggered. While a ROB is not extraordinarily complex, it is still worth noting that this design is beginning to edge away from the inherent simplicity that the original counterflow pipelined processor started out with.

The ROB has a complicated job to perform, handling instructions both when they are entering the pipeline and retiring instructions which have completed. Figure 3.3 shows an example. An instruction ADD R8, R1, R3 arrives at the decode stage at time (t). This instruction needs to perform the action of

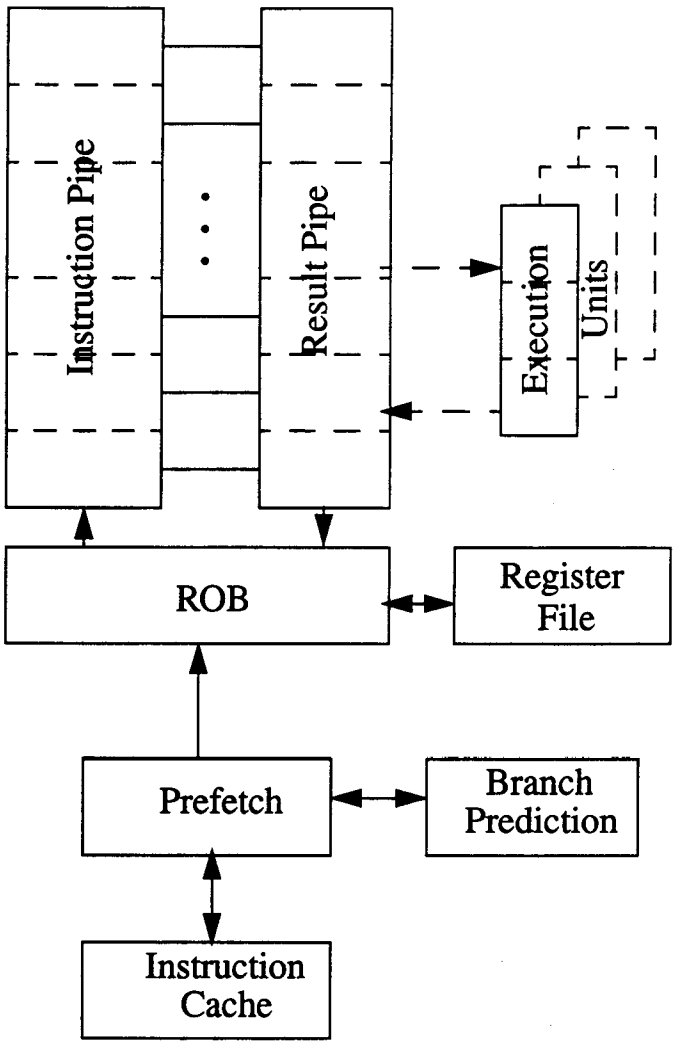

Figure 3.2 Virtual Register Processor Architecture

adding the contents of register 1 (R1) and register 3 (R3) and putting the results in register 8 (R8). The ROB contains registers R1, R5, and R2 from work done previously. R1 has been in the ROB the longest and has been allocated the ROB entry number zero (T0). R2 was the most recent instruction to be put in the ROB and has been allocated the ROB entry number two. Now that the ADD instruction

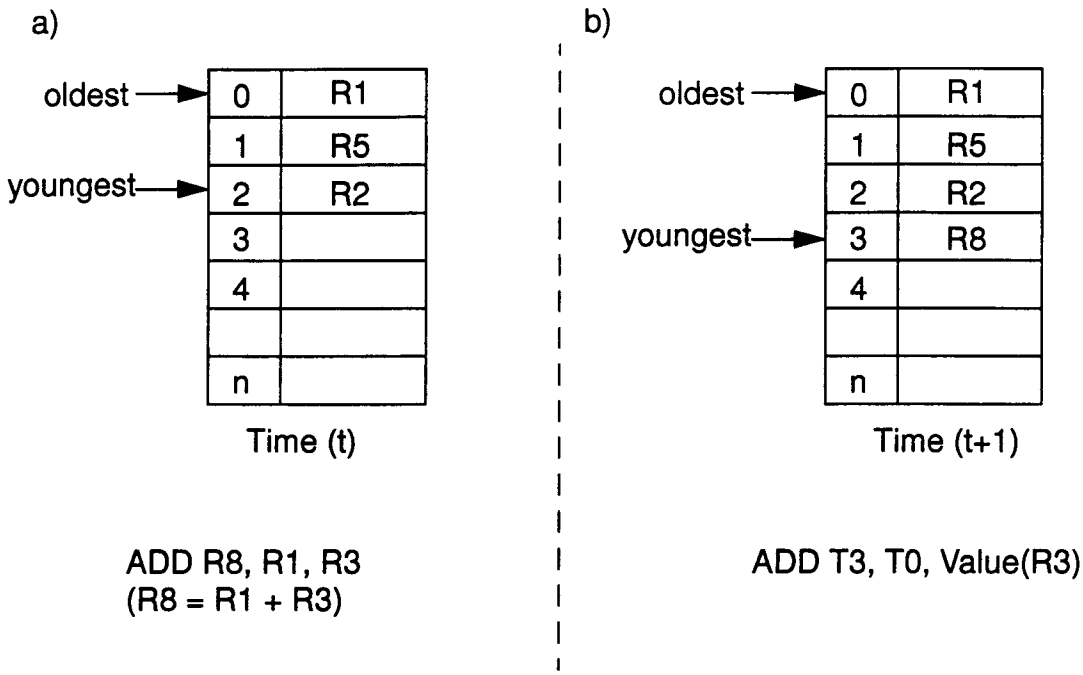

Figure 3.3 ROB Interactions (a) before allocating entry (b) after allocation

has arrived at the decode stage, it asks the ROB to allocate an entry for it. If the ROB is full, and cannot give the instruction an entry, the instruction must stall at the prefetch unit and wait until another older instruction completes execution and is retired from the machine. Figure 3.3b shows the ROB and instruction after the instruction has been allocated an entry. The instruction will eventually create a value for R8, so the ROB puts the tag for R8 into it's next available entry which in this case happens to be tag number three. The instruction's result register has been renamed and therefore it now indicates that it will write back to tag entry three (T3). During this time, the ROB has also checked to see that this instruction wants the values of R1 and R3. It finds that it has renamed R1 to tag zero and

replaces the R1 with T0 in the instruction. The ROB also finds that it doesn't have a tag for R3 and therefore the value must be in the register file. The ROB gets the value for R3 from the register file and forwards this value to the instruction. At this time, the instruction is now ready to enter the pipeline to be executed. Eventually, the instruction will finish executing and a result tagged T3 will come down the result pipeline. The ROB will take this tag, match it with the original register R8 and will write the result back to the correct register file entry.

3.2.2 Result Pipeline Width

Now that the register file has been moved to the bottom of the pipeline, the operands no longer need to travel down the result pipeline. As a result of this, there is less competition for available slots in the result pipeline [11]. The instructions need these empty slots to send completed results to instructions earlier in the instruction pipeline. The effect of this is to shorten the time between a result being calculated and the result being used as an operand in a subsequent instruction. One benefit of this new result pipeline is that the instruction pipeline never needs to stall because of an instruction getting to the top of the instruction pipeline and not being able to find an empty slot in the result pipeline to send it's results down. Since there are no results coming in at the top of the result pipeline, the last pipeline stage is guaranteed to be empty even in the worst case. In CFPP, the result pipeline needed to be at least as wide as the maximum number of operands that an instruction can have. Since the register file is at the bottom of the instruction pipeline in VRP, the result pipeline doesn't need to be as wide, thus reducing the amount of hardware necessary.

3.2.3 Instruction Removal

One of the biggest improvements in VRP is at what time the instructions are removed from the pipeline. This is actually very subtle, but can lead to a substantial increase in performance. In CFPP, the instructions must stay in the instruction pipeline from start to finish. This is because once the instruction has completed executing, it has to continue to the end of the pipeline to write its result into the register file. In a VRP processor, since the register file is at the bottom of the pipeline, once an instruction has finished executing and has placed its result into the result pipeline, it can be removed from the instruction pipeline. This creates more bubbles or empty locations in the instruction pipeline which can be used to absorb some of the costs of a stall. If an instruction near the top of the instruction pipeline stalls and there are bubbles below it, the instructions can "squash" the bubbles and continue up the pipeline. This has the effect of hiding some or all of the latency involved with the instruction stall allowing other instructions to continue doing useful work while other instructions are stalled. If the instructions at the bottom of the instruction pipeline do not need to stall, more instructions can be issued into the pipeline.

3.3 Simulation Results

Various simulations were run using the Spec95 traces to get an estimate of the performance of a VRP processor. Figure 3.4 shows the configuration that was eventually decided upon after many simulations. This is not to say that this is the best configuration possible, with more time and a lot more compute cycles a better solution most certainly could be found. In the configuration shown, the

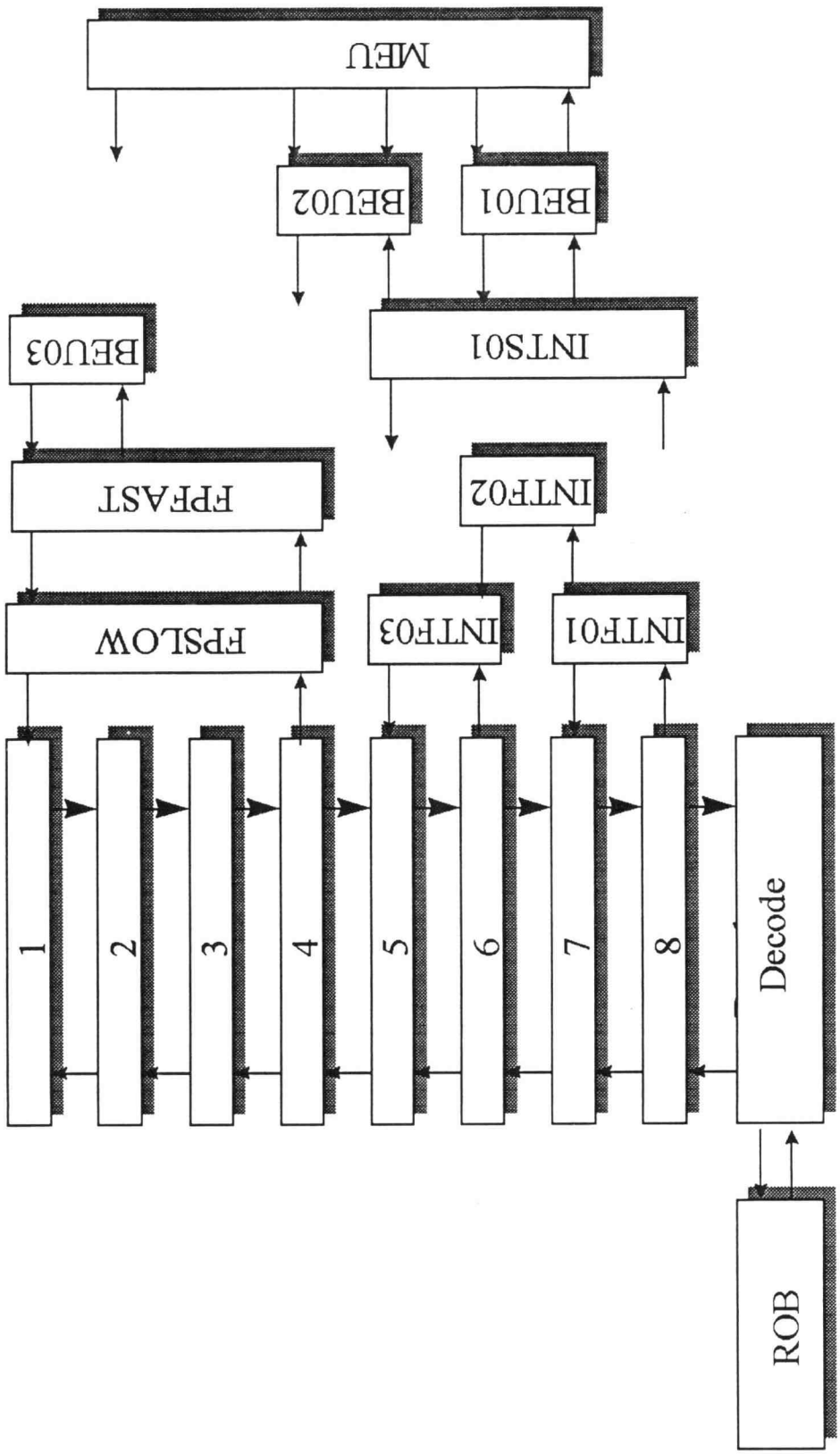

Figure 3.4 VRP Configuration

instruction pipeline can hold one instruction per pipestage and the results pipeline can hold two results per pipestage.

There are three fast integer units (INTF01-INTF03). These units have a one cycle latency and handle instructions such as ADD, SUB, Shift, etc. There are three branch execution units (BEU01-BEU03). These units have a one cycle latency, and communicate the results of the branch back to the branch prediction unit. There is one slow integer unit (INTS01). It has a latency of four cycles, is fully pipelined, and handles slow integer instructions such as multiply and divide. There is one fast floating point unit (FPFAST). It has a latency of four cycles, is fully pipeline, and handles fast floating point instructions such as floating point addition and subtraction. There is one slow floating point unit (FPSLOW). It has a latency of eight clock cycles, is fully pipeline, and handles slow floating point instructions such as floating point multiply and divide. There is a memory execution unit (MEU), which handles load and store instructions and communicates with the ROB to maintain proper ordering of loads and stores. There is one level one (L1) data cache, not pictured, which is a 16KB, 4-way set associative data cache, with one cycle access time, and SLRU replacement policy [4].

The following assumptions have been made to allow for a higher level simulator. It is assumed that the L1 cache and main memory hold all necessary data. The main memory has a constant 10 cycle access latency. The branch prediction has a randomly predicted 94% correct branch prediction. When recovering from a mispredicted branch, there is a one cycle "no fetch" penalty [5].

To maintain precise interrupts, store instructions are not allowed to complete until they are the oldest instruction in the ROB [18]. Also, the ROB is allowed to retire as many instructions as it needs to in one clock cycle.

Figure 3.5 shows the performance of the ten Spec95 traces on the VRP

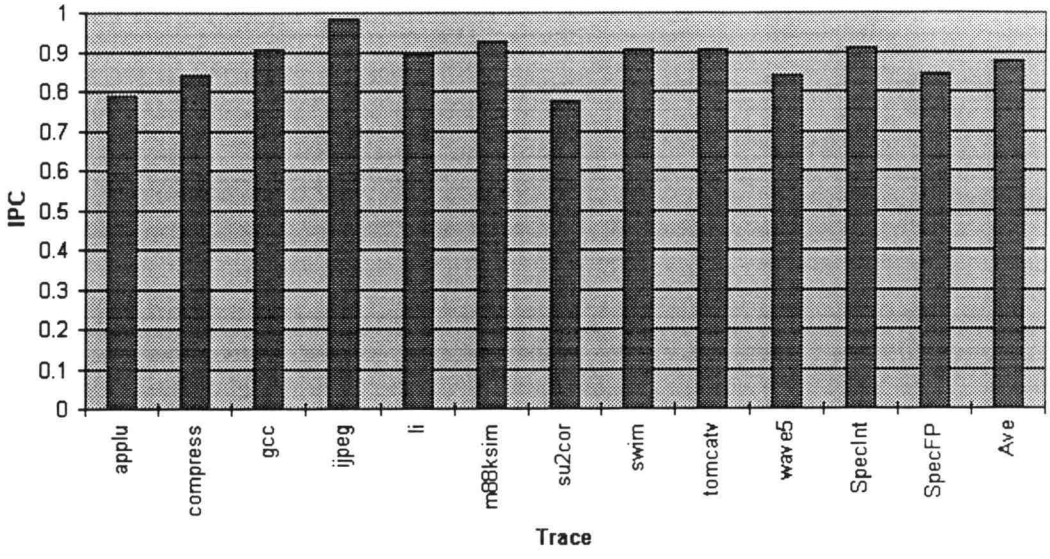

Figure 3.5 VRP Average Instructions Per Clock Cycle by Trace

configuration shown in figure 3.4. The performance is expressed in average instructions per clock cycle (IPC) for the first 2 million instructions of each trace. Performance for the SpecInt traces was slightly higher at 0.91 versus the SpecFP traces with an IPC of 0.84. The average IPC for VRP on all ten Spec95 traces is 0.88.

The execution unit usage data was collected to analyze how efficiently the units were being used as well as to compare against the other counterflow

architectures to see how the additions which were made to the architecture affected the efficiency. Figure 3.6 shows the percent of time each execution unit

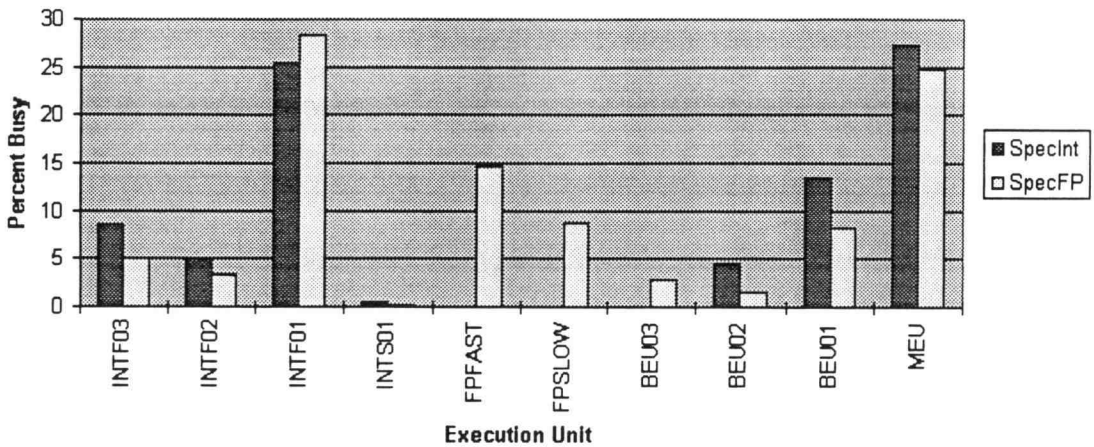

Figure 3.6 VRP Execution Unit Usage

was kept busy for both integer and floating point traces. The memory and first fast integer units were kept busy approximately 25% of the time. Note that the first fast integer unit's usage number would be higher if one of the other two fast integer units were eliminated. While this is true, allowing these instructions to continue up the pipeline to one of the other two units prevented instructions from stalling and effectively halting the processor at the very beginning of the pipeline. A similar effect was observed with the branch execution units. Most of the branches are executed by the first BEU. This aids performance since it clears these instructions out of the pipeline earlier and if the branch was incorrectly predicted, limits how many instructions enter the pipeline from down the bad branch path. If however the branch cannot execute until later in the pipeline due

to a data dependency, the second and third branch execution units allow the branch to proceed up the pipeline without causing the pipeline to stall. There is a higher penalty associated with this case since more incorrect instructions were brought into the processor, but since the branch predictor usually guesses correctly and the pipeline wouldn't have been doing useful work if a guess hadn't been made, this is considered a good trade-off [11].

Figures 3.7 and 3.8 show the pipeline usage for the result and instruction

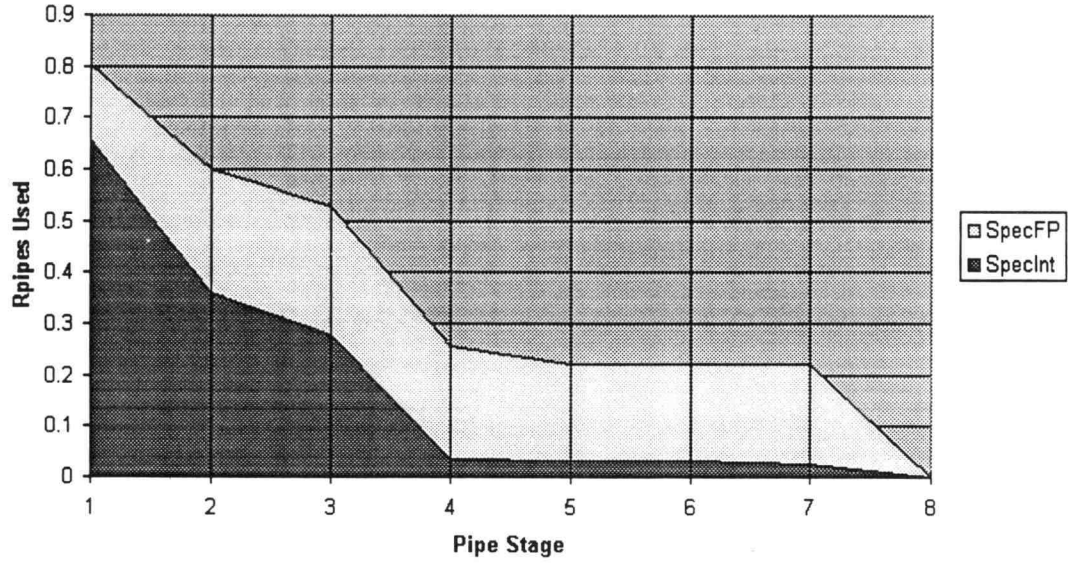

Figure 3.7 VRP Result Pipeline Usage

pipelines respectively. The result pipeline usage shows that, as expected, a majority of the result pipeline's usage occurs in the bottom half of the pipeline. Since all of the results have to flow from their originating point and proceed down to the bottom of the pipeline to the register file this is an expected result. What it

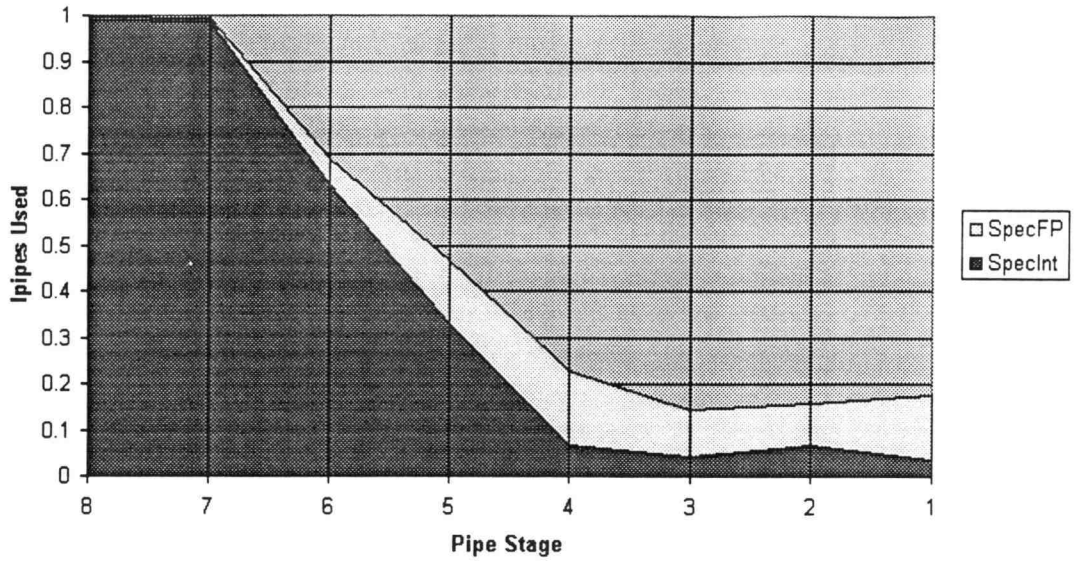

Figure 3.8 VRP Instruction Pipeline Usage

does show is that when a result is generated, there is almost always an empty slot in the result pipeline available. Since the instruction cannot leave the instruction pipeline until it has managed to put it's results into the result pipeline this is important for the overall throughput of the machine. In a similar manner, the usage of the instruction pipeline shows that most instructions execute and place the results in the result pipeline in the bottom half of the pipeline. It is important to recognize that this is one of the major improvements of VRP over the original CFPP implementation. In the original CFPP implementation, instructions were required to remain in the instruction pipeline for the entire length of the pipeline since they needed to carry their results to the register file. With VRP, once an instruction's results have been placed in the result pipeline, the instruction has

completed and becomes a bubble [3]. If an instruction farther up in the pipeline stalls, these bubbles can be squashed allowing instructions near the bottom of the pipeline to continue doing useful work thus hiding some of the penalty associated with stalling.

Figure 3.9 shows the percentage of time that the instruction pipeline's

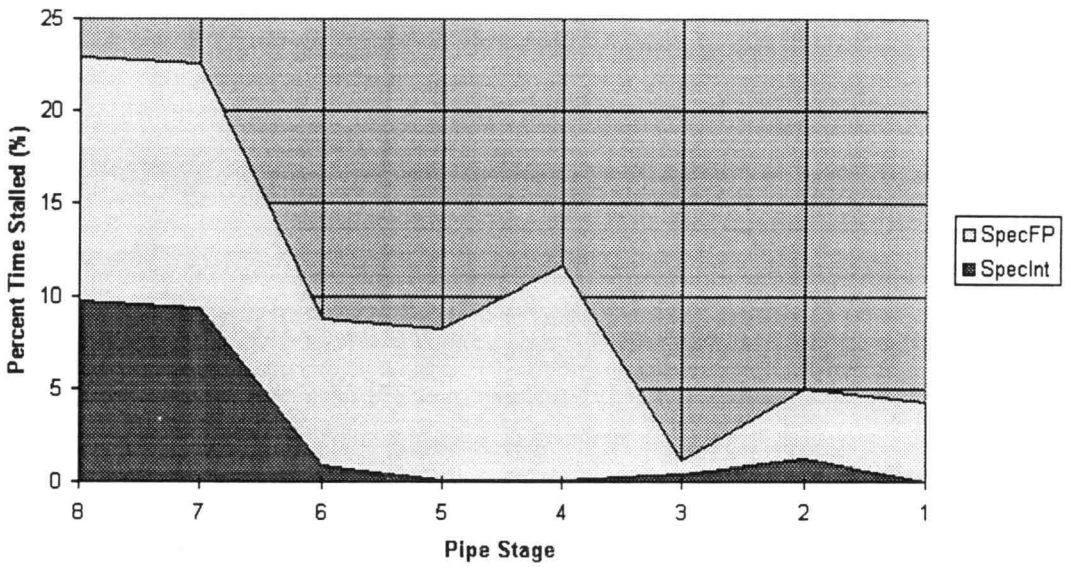

Figure 3.9 VRP Stalling Locations

stages are stalled. Unfortunately, the first two stages of the pipeline are stalled over twenty percent of the time for the floating point traces and ten percent of the time for the integer traces. This is not desirable behavior. If an instruction must stall, it would preferable if it stalled in the later half of the pipeline. If the instruction stalls later, some of the bubbles created by completed instructions can be used to hide the latency of the stall. Most of the stalls in pipestages 7 and 8 are caused by

pipestage 7 being the memory execution unit's only launch point. The launch point could be moved back to a later pipestage, but doing so causes other arithmetic instructions, dependent on the memory instruction, to stall causing the overall performance to drop. The increase in stalls at pipestage 4 in the floating point traces is caused by the launch point of both the fast and slow floating point execution units at that pipestage.

Figure 3.10 shows the average number of available ROB entries for both

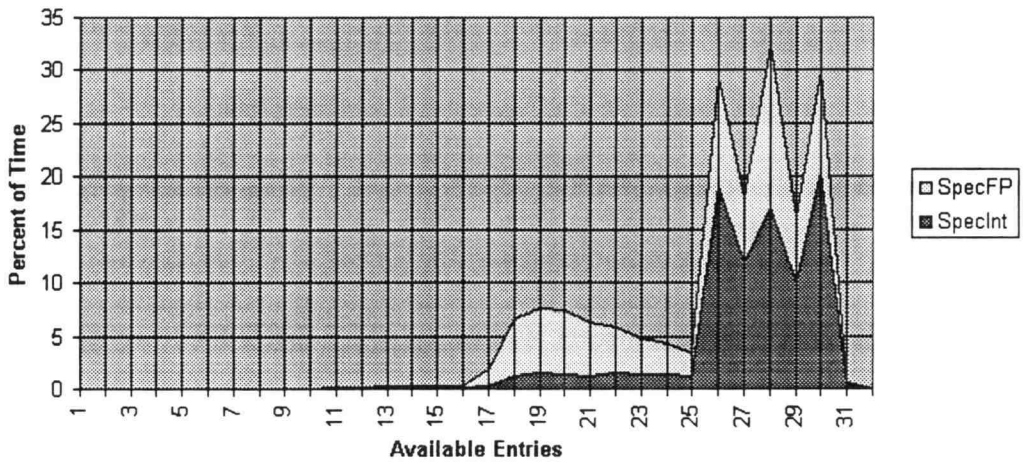

Figure 3.10: VRP ROB Available Entries

floating point and integer traces. Since there are only 8 pipestages, it is surprising at first to see that VRP is using more than eight entries in the ROB at all. However, the ROB entry must stay allocated from the time the instruction enters the pipeline until it's results return to the ROB. If no stalling occurred, the maximum time that a ROB entry would be allocated would be 16 cycles. For the integer traces, this is almost exactly what occurs. The ROB almost never uses

more than 16 entries. The floating point traces, while still rarely using more than 16 entries, uses far more entries than the integer traces. This is partly because floating point instructions take longer to execute and thus the time for inter-instruction dependencies to be resolved is longer. This causes more instructions to stall waiting for their results from earlier instructions and thus stay in the pipeline longer. Another reason the floating point traces use more ROB entries is that the slow floating point execution unit has a longer latency than the number of pipestages between it's launch and recover points. Therefore, whenever a slow floating point instruction is executed, it guarantees that the instruction will stall at the recover point. Fortunately the recovery point is at the last stage of the pipeline where it rarely stalls any other instructions. For the pipeline configuration used, there would be no reason to increase the size of the ROB since the pipeline never used up all of the ROB entries. It may even be feasible to decrease the ROB size so that it only has 16 entries. The number of times that more than 16 entries were allocated is so small that the effect on performance would be minimal.

3.4 Chapter Summary

In this chapter, the microarchitecture of the virtual register processor has been described. This microarchitecture offers the main improvement over the original counterflow pipelined processor of moving the register file from the top of the pipeline to the bottom so that the previously calculated results are available at the start of the pipeline. As a consequence of moving the register file, a reorder buffer has been added. This addition has the by-product of solving several other problems which prevented the previous architecture from being attractive. Now,

with the ROB, recovering from a mispredicted branch becomes much easier. There is no longer a global signal running from the branch execution unit to every pipestage below to inform those stages to invalidate the instructions that they are carrying. In VRP, the branch execution unit only communicates to the ROB and branch prediction units. When a misprediction occurs, the branch execution unit sends a signal to the ROB telling it that the branch and all instructions issued after the branch need to be removed from the pipeline. Instead of actually removing the instructions however, the ROB simply lets the instructions complete their execution, but when the results come back to be retired, the ROB ignores them. This is simpler, but does cause the pipeline and execution units to be used to calculate results that will never be used. Effectively wasting time and resources that could be better used doing useful work. The simulation results for VRP show that the improvements made have resulted in work being computed more efficiently, but there are still improvements which can be made to get rid of the remaining bottlenecks.

4. THE COUNTERDATAFLOW PROCESSOR (CDF)

There are two main drawbacks to the virtual register processor. The most serious limitation is that architecture is still limited to only launching one instruction per clock cycle. Given the fact that current processors are already capable of executing more than one instruction per clock cycle [8], this limitation rules VRP out of being used as a general purpose processor. The other drawback is also related to this limitation of launching only one instruction per cycle. Even if it were possible for VRP to launch more than one instruction per clock, the pipeline still stalls, and would quickly clog the instruction flow since instructions would not be able to pass each other. The counterdataflow processor design overcomes both of these difficulties and as will be shown may be a viable alternative to current microprocessor designs [13] [14].

4.1 Architectural Description

The main improvement in counterdataflow over the virtual register processor comes by allowing both the instruction and result pipelines to wrap around thereby creating a circular structure. The two pipelines are now two counter-rotating queues. Referring to figure 4.1, the instruction pipeline moves up while the result pipeline moves down just as in the previous two architectures. Only now, if an instruction gets to the end of the pipeline and hasn't executed, it simply wraps around to the beginning of the pipeline and continues up the pipeline. The result pipeline acts similarly, only for slightly different reasons. The results, upon reaching the bottom of the pipeline do not necessarily need to wrap

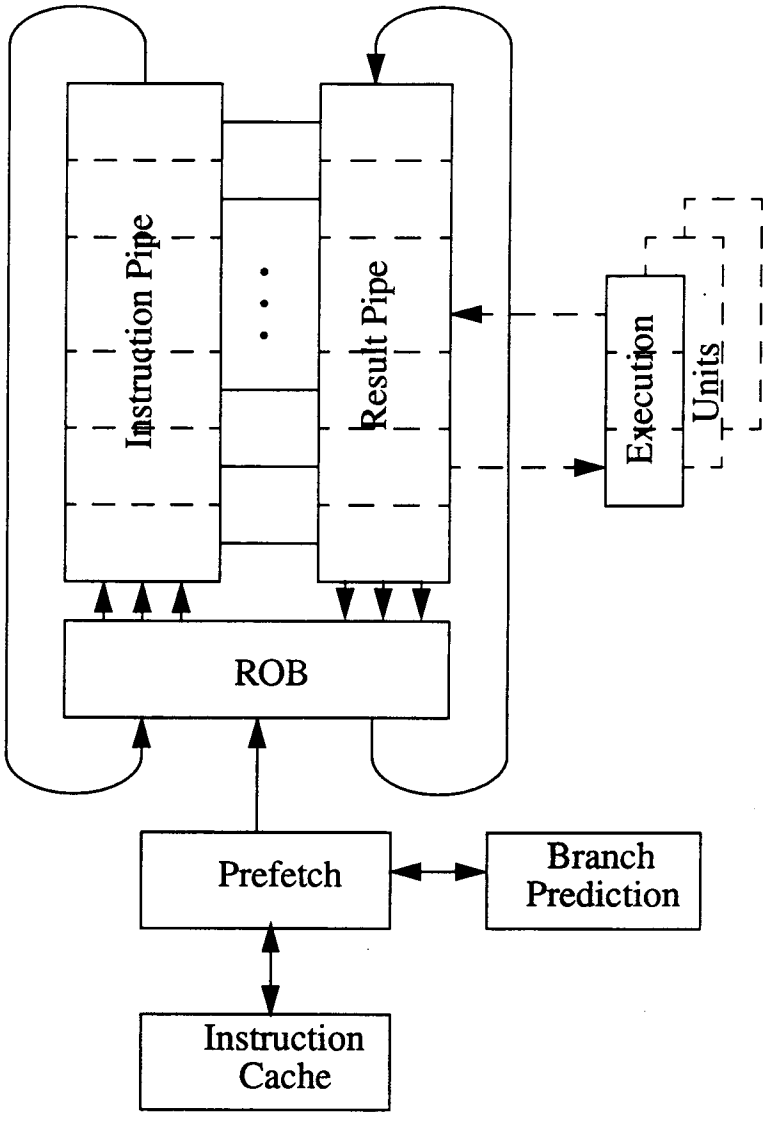

Figure 4.1 The Counterdataflow Architecture (CDF)

around. They could just write their values into the ROB, and exit the pipeline. The results are forced to wrap around for performance reasons. In most cases, the results that an instruction generates are used by the instructions immediately following. If this happens, the following instruction must go all the way around the

pipeline just to read the value from the ROB. This both increases the latency to execute instructions as well as puts additional read ports on the ROB. By forcing the results to make one extra trip around the pipeline, the worst case delay waiting for a result will be half the length of the pipeline since the instruction and result pipeline are moving in opposite directions [14]. Since neither of the pipelines are required to stall, by having the results make the one extra trip around the pipeline, it is guaranteed that all instructions will pass the result and will read it if they need the value.

4.2 Architecture Changes From VRP

The main architectural change to CDF from VRP is the ability to wrap both the instruction and result pipelines around. This deceivingly simple change brings out various enhancements, resulting in both performance improvements as well as simplifications in implementation. Since the instruction pipeline no longer stalls, the remaining global signal, the pipeline halt signal, has successfully been removed and one of the main premises of the counterflow architecture, local control, has been achieved.

4.2.1 Multiple Instructions Per Clock Cycle

Now that the instructions can wrap around to the start of the pipeline once they reach the end, multiple instructions per clock cycle can be issued. This can be done by making each stage of the instruction pipeline wider. It doesn't make any difference if instructions in the same stage are dependent on each other since if the dependency isn't resolved by the end of the pipeline, the instruction or

instructions will simply wrap around and potentially execute in this next pass of the pipeline. In VRP and CFPP, the dependent instruction would have to stall the entire pipeline waiting for its operands. In theory, the width of the instruction pipeline is unlimited, it is possible to launch unlimited instructions per clock cycle. In all practicality, the number of instructions issued per clock cycle is bounded by area available, and the amount of logic which can be executed during one clock cycle. Currently, a width of four to eight instructions wide seems feasible, but with future advances in processing technology, that number can be expected to increase [18] [19]. In this thesis, the instruction pipeline width was limited to four instruction per pipestage.

4.2.2 Out-Of-Order Execution

By it's vary nature, CDF executes instructions out of order. Any instructions which are not able to execute in their first revolution of the pipeline will wrap around and be surrounded by younger instructions just being issued. Even for a processor where the instruction pipeline is one instruction wide, the instructions are fetched in order, but the instructions will be executed regardless of order, chosen only by which instructions are ready to be executed. With a wider instruction pipeline, deep speculation occurs exposing more available parallelism than would otherwise be possible. Since more instruction level parallelism is exposed, the number of instructions executing at any given time increases thus increasing performance. This will be illustrated later in the simulation results section.

4.2.3 Fast Clock Cycle

The counterflow pipeline principle was first developed for it's use of local control. This allows a very fast clock cycle since there are no global signals which take a relatively long to cross the chip. Unfortunately, there has always been one signal which needs to propagate through the pipeline from start to finish. Up until now, the instruction pipeline has always needed to stall. It is possible for an instruction at the very end of the pipeline to stall thereby needing to stall every instruction down the pipeline back to the fetch unit. This has been the bottleneck in maximum clock speed for the CFPP and VRP processor.

Since CDF's instruction and result pipelines wrap around, there is no longer any reason to stall. This lessens the logic complexity in the basic pipeline cells as well as in the pipeline to execution unit logic. With this innovation, the architecture has returned to counterflow's basic premise of obtaining high clock speeds by having local control of information and a simple pipeline.

4.2.4 Easy and Inexpensive Recovery from Incorrect Speculation

Modern microprocessor have very high branch prediction rates, greater than 90% [8]. However, 10% of the time, the prediction is still wrong. It is important to be able to recover from these incorrect speculations quickly and inexpensively. Since the speculation in CDF is even higher than other processors, it is even more important for this recovery to be efficient. CDF accomplishes this in much the same way that other modern processors (including VRP) do, by using a ROB. When a mispredicted branch is detected, all instructions after the branch are invalidated from the ROB. In most other

architectures, the instructions are either forced to complete execution using up valuable resources, or are explicitly removed from the processor with extra hardware [8] [18]. In CDF, a hybrid approach is taken. If an instruction can execute, it is allowed to. When the results return to the ROB, they are ignored and deleted from the pipeline. If the instruction has not managed to execute, when it wraps around and passes the ROB, it sees that it has been invalidated in the ROB, and deletes itself from the pipeline. This allows at least some instructions to not have to execute. This is important for an architecture such as CDF where aggressive speculation occurs. When an incorrectly speculated branch has been identified, all of the incorrect instructions in the shadow of the branch are not required to complete their execution and can leave the machine without having taken up time and space in the execution units.

4.2.5 Tolerance of Long Latency Execution Units

In CFPP and VRP, instructions that have begun executing remain in the pipeline until they have completed. For long latency instructions like loads, stores, and floating point operations, these instructions can stall the entire pipeline keeping unrelated instructions from executing even though resources are available. In CDF, once an instruction has begun executing, it leaves the pipeline. The results of the execution do not need to be matched with the corresponding instruction until they arrive at the ROB. Therefore, the results are simply sent down the result pipeline whenever they complete. By doing so, load and store instructions are saved from having to wrap around the pipeline several times in the case of a cache miss. This frees up space in the instruction pipeline for

another instruction to enter the pipeline, thus increasing the instruction window and therefore the amount of available parallelism exposed in the pipeline.

4.2.6 Support for Multithreading

One of the benefits which was never anticipated was the ease with which CDF supports multithreading. Multithreading or shared resource multiprocessing (SRMP) [18] is implemented quite naturally with CDF, as is illustrated in figure 4.2. With multithreading, the same execution units can be shared among instructions from all of the threads. Therefore, one version of area expensive execution units such as memory order buffers (MOBs) and floating point units can be used by all of the different threads. The only hardware that absolutely has to be duplicated are the ROBs and fetching logic. Since all matching is done based on tags, each ROB would have it's own unique tag which would get appended to the normal instruction tag. The matching which occurs in the pipeline is based entirely on the tags. Since the instructions from different threads would have different tags and therefore would not match, the pipeline's logic will continue to do the same job of matching as before. The analysis and simulation of multithreading is beyond the scope of this thesis and will not be covered.

4.3 Simulation Results

To analyze the CDF architecture, many different pipeline configurations were simulated. The configuration in figure 4.3 had the best average performance and was chosen for full analysis. This is not meant to indicate that this is the one and only best configuration only that this was the best configuration that could be

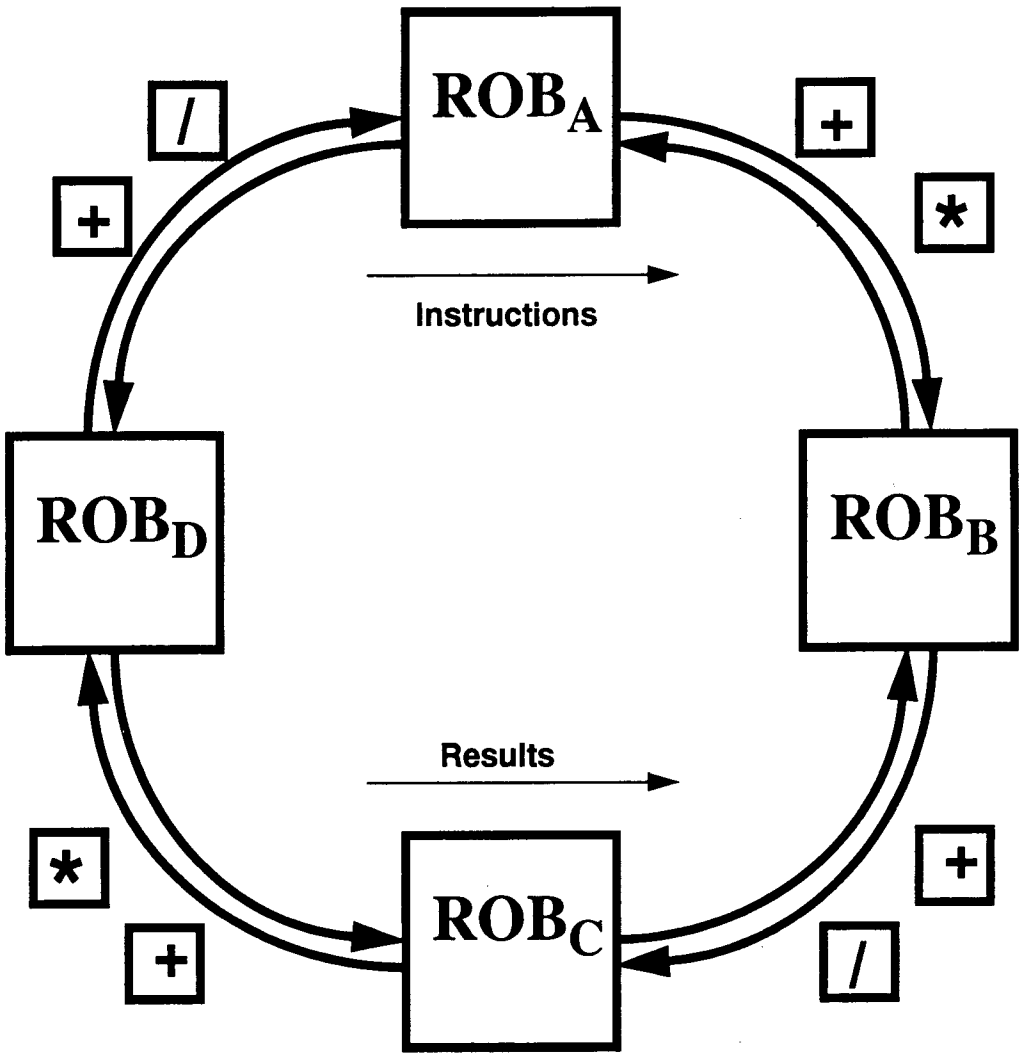

Figure 4.2 Multithreading with CDF

found with a given amount of time and computer cycles. In the configuration shown, the instruction pipeline can hold four instructions in each pipestage and the result pipeline can hold eight results in each pipestage. In figure 4.3, the top four pipestages have been replicated at the bottom of the pipeline in order to

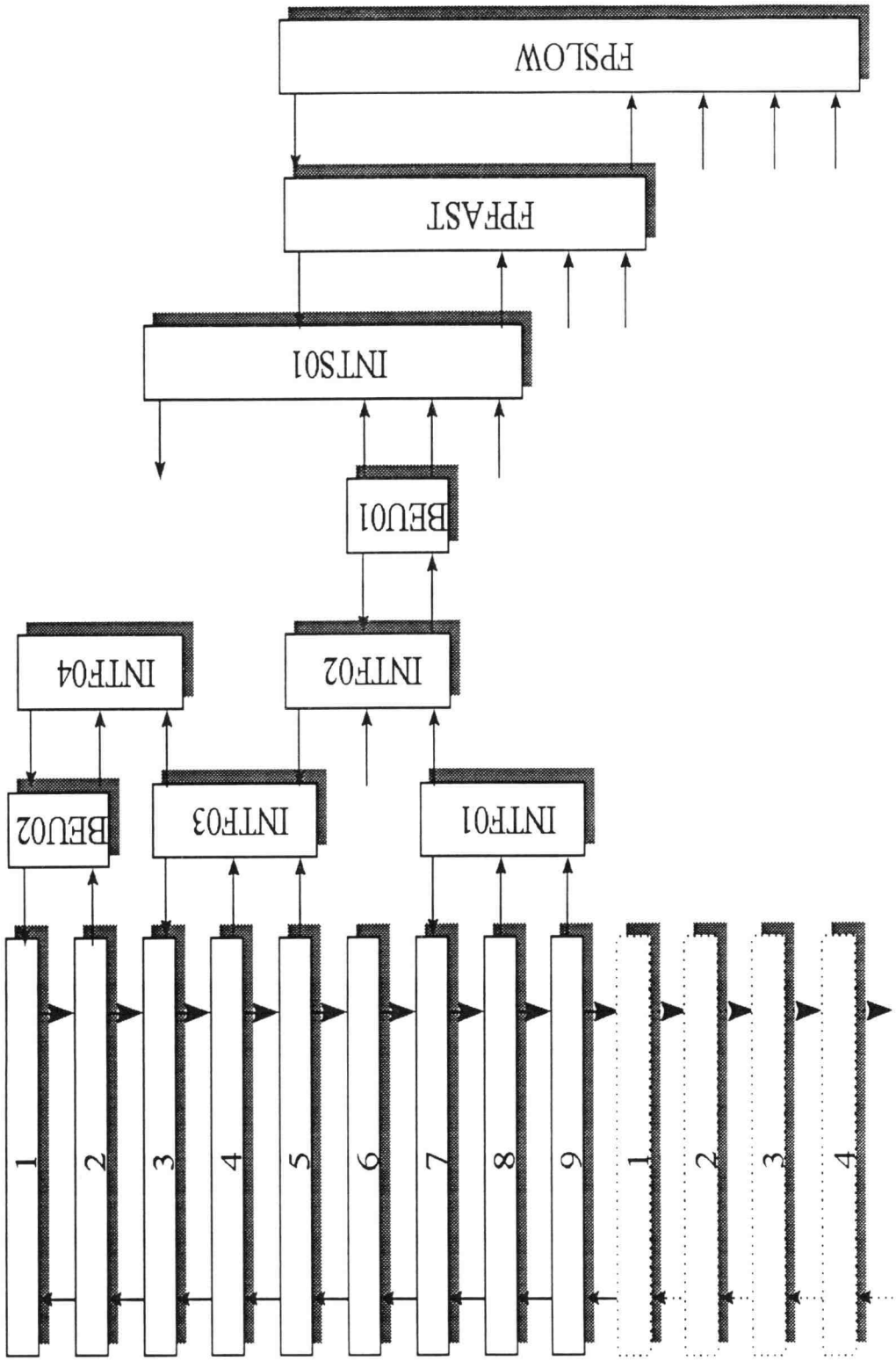

Figure 4.3 CDF Simulation Configuration

illustrate that the two floating point units launch on one cycle of the pipeline and recover on the next cycle. Since these execution units have such long latencies, it was observed to be better for overall performance to recover on the next revolution at the location where the instructions that immediately followed the launching instruction will be. These instructions are the mostly likely ones to be waiting for the result of the execution and they therefore receive the result the earliest [18]

There are four fast integer units (INTF01-INTF04). These units have a one cycle latency and handle instructions such as addition, subtraction, logical operations, etc. There are two branch execution units (BEU01-BEU02). These units have a one cycle latency, and communicate the results of a branch back to the branch prediction unit. There is one slow integer unit (INTS01). It has a latency of four cycles, is fully pipelined, and handles slow integer instructions such as multiply and divide. There is one fast floating point unit (FPFAST). It has a latency of four cycles, is fully pipeline, and handles fast floating point instructions such as floating point addition and subtraction. There is one slow floating point unit (FPSLOW). It has a latency of eight clock cycles, is fully pipeline, and handles slow floating point instructions such as floating point multiply and divide. There is a memory execution unit (MEU), not pictured, which handles load and store instructions and communicates with the ROB to maintain proper ordering of load and stores. There is one level one (L1) data cache, not pictured, which is a 16KB, 4-way set associative data cache, with one cycle access time, and SLRU replacement policy [4].

The following assumptions have been made to allow for a higher level simulator. It is assumed that the L1 cache and main memory hold all necessary data. The main memory has a constant 10 cycle access latency. The branch prediction unit has a randomly predicted 94% correct branch prediction rate. When recovering from a mispredicted branch, there is a one cycle "no fetch" penalty. To maintain precise interrupts, store instructions are not allowed to complete until they are the oldest instruction in the ROB. Also, the ROB is allowed to retire as many instructions as it needs to per clock cycle.

A total of ten Spec95 traces were run. Since the number of cycles needed to simulate a processor of this complexity is large, only the first 2 million instructions of each trace were run. Five of the traces (*m88ksim*, *gcc*, *compress*, *li*, and *ijpeg*) were integer traces, and the other five (*tomcatv*, *swim*, *su2cor*, *applu*, and *wave5*) were floating point (FP) traces [20]. Figure 4.4 shows the performance in average instructions executed per clock cycle for each trace. While the average integer performance is slightly higher than the average FP performance, it is interesting to note that the floating point traces had widely varying performances. Resulting in both the highest performance, *swim* with an IPC of 2.5, and the lowest performance, *su2cor* with an IPC of 1.1. The average performance is 2.0 instructions per clock cycle.

Current technology allows many execution units to be fabricated. The problem is that up until now, processors haven't been able to utilize these execution units. For much of the time, machines with five or six executions have these units busy performing useful work only 20-30% of the time [8]. With CDF,

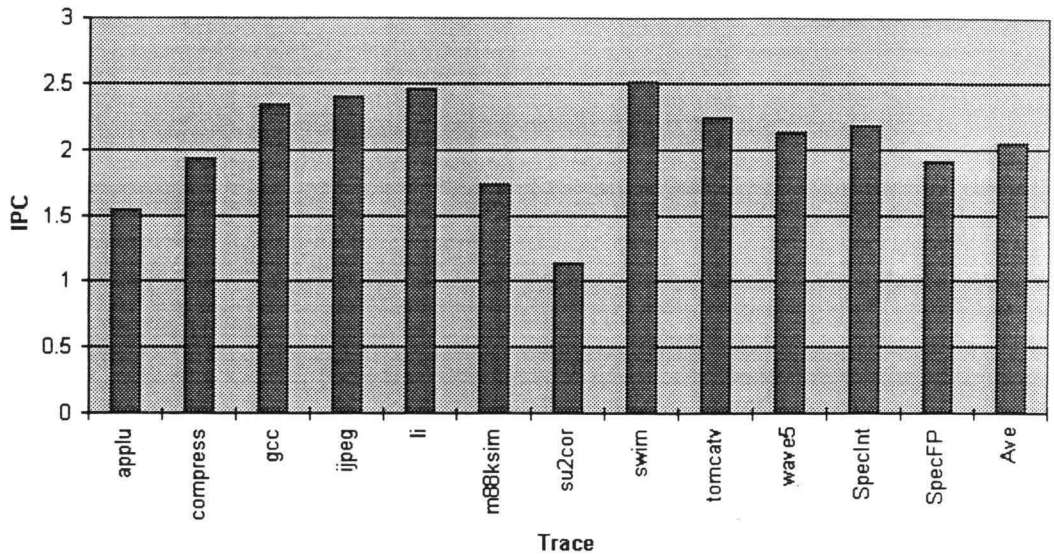

Figure 4.4 CDF Average Instructions Per Clock Cycle by Trace

many instructions are brought into the machine to be potentially executed, thus increasing the effective instruction window. Also, by allowing instructions to launch to the execution units at multiple locations, it looks (from the viewpoint of the instructions) as though there are more execution units. Figure 4.5 shows the percent of time the execution units were kept busy for both integer and floating point traces. Since the memory execution unit (MEU) was able to accept up to five instructions, and these are capable of taking long periods of time, it is not surprising that the MEU was busy most of the time, 85% for SpecInt and 68% for SpecFP. The fast and slow floating point units (FPFAST and FPSLOW) are similar with 57% and 38% respectively. Since SpecInt traces have negligible numbers of floating point instructions, they essentially do not use the floating point

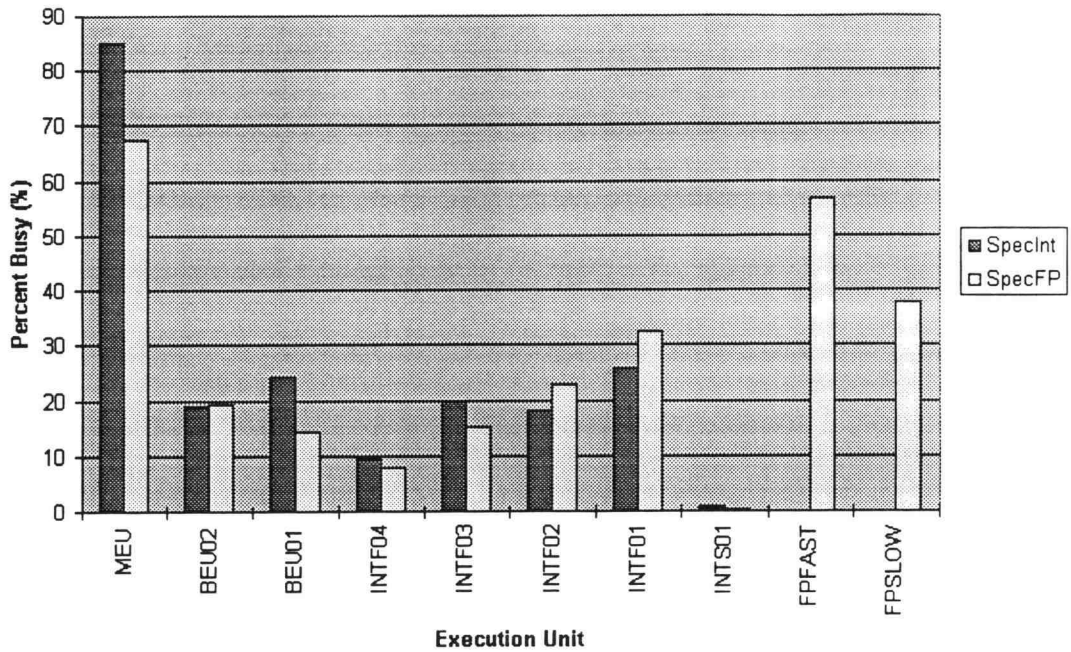

Figure 4.5 CDF Execution Unit Usage

execution units. The compiler used to generate the traces unfortunately had the penalty for using slow integer operations set high and therefore generated few of these instructions, causing the slow integer unit to be rarely used.

The utilization of the pipelines is of prime interest in attempting to optimize a CDF processor's performance. The ultimate goal in CDF is to get the maximum number of instructions into the machine as possible. This increases the exposed available parallelism in the code and allows useful work to be done even when older instructions may be waiting for their data. The number of instructions that can be launched is the same as the number of instructions which do not wrap

around the pipeline. This shows that while it is advantageous for instructions to be able wrap around if they cannot execute, performance suffers if instructions wrap around too much. For example, if the instruction pipeline is four instructions wide and when a set of instructions get to the top of the pipeline none of the instructions have been able to launch, no new instructions are able to enter the machine during that clock cycle. However, if two of those instructions did manage to launch, two new instructions can enter the machine and have a chance to execute. It is noteworthy that it is not very important how long the instructions take to execute and return their results. It is far more important that the instructions launch and leave the instruction pipeline allowing new instructions to enter. Figure 4.6 shows the instruction pipeline utilization for the pipeline

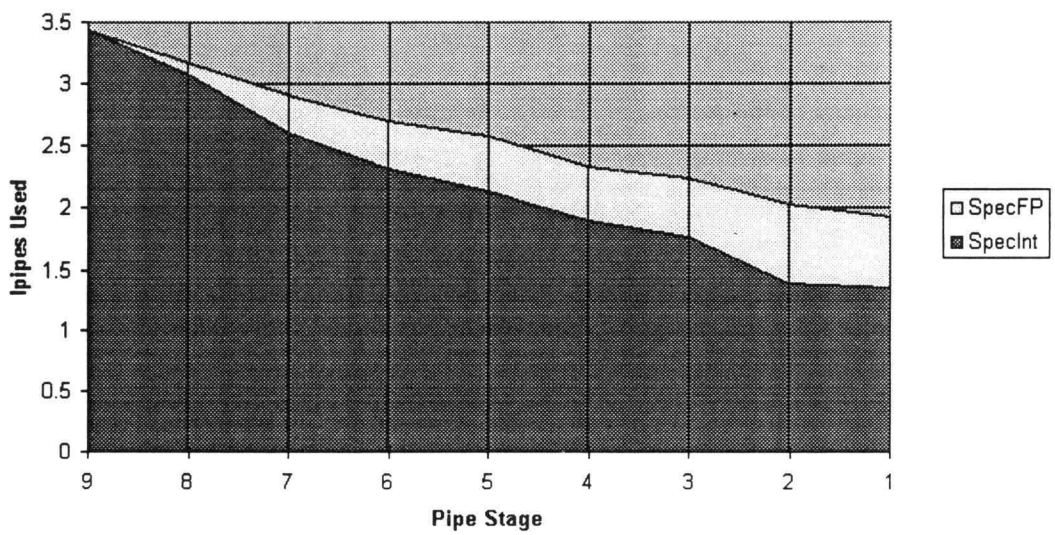

Figure 4.6 CDF Instruction Pipeline Usage

configuration depicted in figure 4.3. Pipestage 9 is the bottom of the pipeline where instructions enter, and pipestage 1 is the top of the pipeline where instructions wrap back to stage 9. At stage 9, the average number of instruction pipelines used is approximately 3.4 for both the SpecInt and SpecFP traces. It is less than the ideal case of all four instruction pipelines being used because prefetch is assessed a one cycle penalty every time a branch is taken. Therefore, everytime a branch is taken, the other instructions prefetched after the branch are discarded, to be fetched in the next clock cycle. The more important number is the average number of instruction pipelines used at the top of the pipeline. The SpecFP traces have just under two instruction pipelines used and the SpecInt traces have around 1.5 instruction pipelines used. The SpecFP instructions stay in the pipeline longer because of the longer latency of the floating point execution units. Still, an average of 2 new instructions have an empty slot to enter the pipeline at each clock cycle. Ideally, this results in an IPC of 2. From figure 4.4, it is shown that the SpecFP traces had an actual IPC of approximately 1.9. For the SpecInt traces, the IPC will ideally be 2.5 while the actual IPC observed was 2.2. These differences can be attributed to incorrect branch prediction, and the fact that instructions cannot always be fetched to fill the empty slots.

The result pipeline utilization is less important than the instruction pipeline utilization. It only affects performance if reduced considerably. Still, the results in figure 4.7 are useful to estimate how wide to make the result pipeline so that it does not become the performance bottleneck. In the pipeline configuration simulated, the result pipeline was made eight results wide. This, as it turns out, is

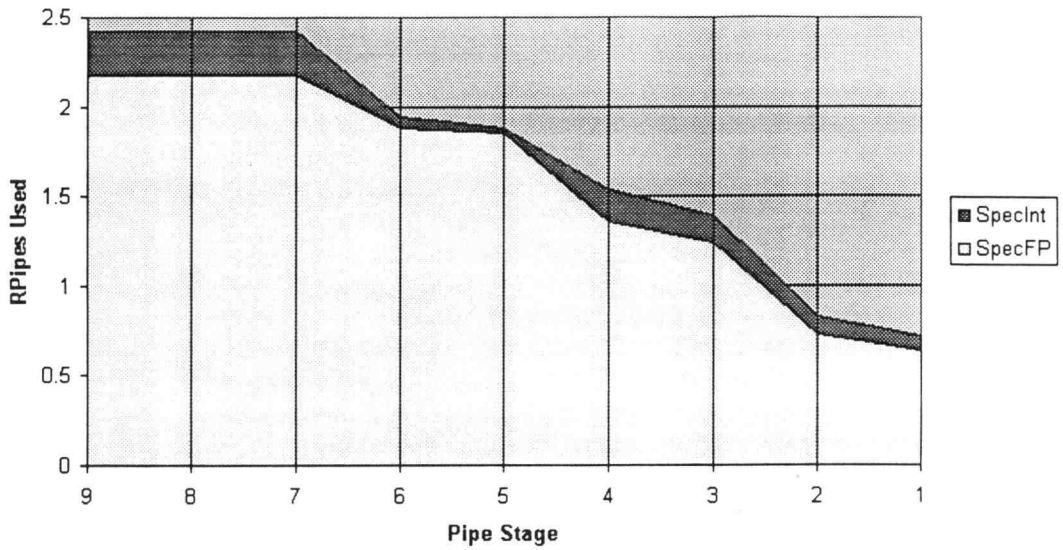

Figure 4.7 CDF Result Pipeline Usage

considerable overkill. Again, pipestage 9 is the bottom of the pipeline where the results are written back to the ROB, and pipestage 1 is the top of the pipeline where some results wrap around. At the bottom of the pipeline, an average of 2.3 results are in the pipeline. It is important to note that the usage of the result pipeline is constant at the bottom of the result pipeline. This occurs because no execution units recover at these pipestages. That, and since the result pipeline does not stall, means that no results enter or leave at these stages. The majority of the congestion occurs in the middle of the pipeline. If the result pipeline is not wide enough, results will not be able to find empty locations in which to write their results back in to, and they will not be able to leave the execution units, thus causing the execution units to stall. Although the execution units stall, this still

does not cause either of the result or instruction pipelines to stall. Since the instructions will just continue passing the execution units until they are no longer stalled.

When attempting to optimize the pipeline configuration for maximum performance, it was found useful to observe the average number of times various types of instructions wrap around the pipeline. It was shown in figure 4.6 that the average performance is strongly dependent on how many instructions are left in the instruction pipeline at the top of the pipeline. To minimize this number, the instructions must launch into the execution units. Figure 4.8 shows the

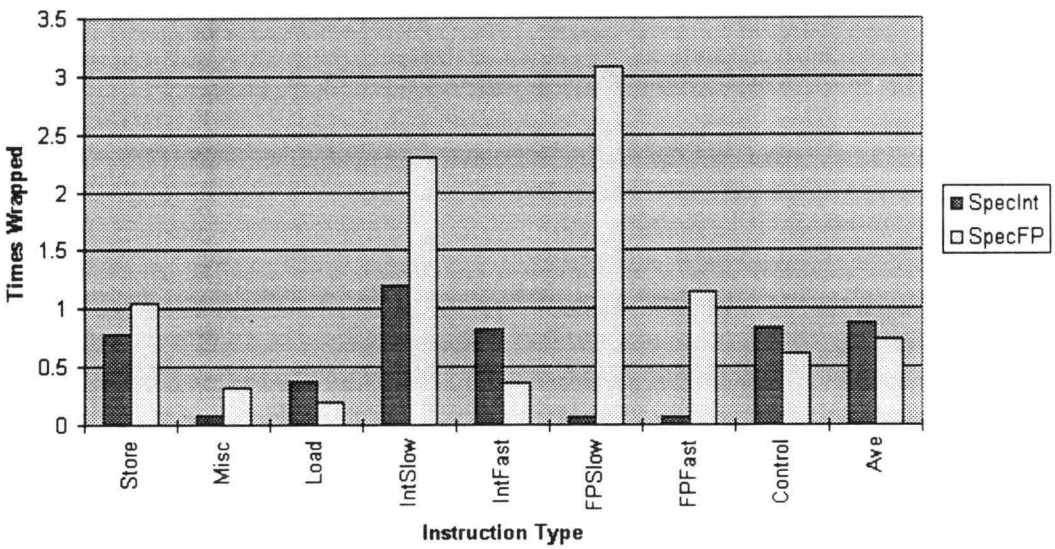

Figure 4.8 CDF Instruction Wrapping by Instruction Type

breakdown of the average times an instruction wraps by the type of execution unit. This graph can be misleading since it does not take into account the number

of instructions of each type. For example, the slow integer instructions (INTSLOW) wrap an average of 1.7 times. While this is relatively high, there are almost no slow integer instructions, so it's reasonable to ignore optimizing these in favor of decreasing the number that wrap of a more common instruction type. The floating point instructions wrap considerably more than integer instructions because of the fact that their execution latencies are longer. The effects of the very high amount of wrapping for slow floating point instructions (FPSLOW) is somewhat offset by the fact that they are usually the end result of a long computation and other instructions are generally not dependent on the result.

Minimizing the number of instructions wrapping has been used as the main metric for optimizing the performance of a CDF processor. It is interesting to analyze the data to see how well this data correlates to actual performance on a trace by trace basis. Figure 4.9 shows the average number of times instructions wrapped by trace. It does indeed show that those traces with the lowest performance wrapped the most while those with the highest performance wrapped the least. Not surprisingly, the two traces whose instructions wrapped the most, *applu* and *su2cor*, are floating point benchmarks since the latencies involved in floating point operations are higher. To aid in analysis, figure 4.10 shows a scatterplot of IPC versus average wrapping by trace. There is essentially a linear relation between performance, expressed in instructions per clock cycle, and the average number of instructions wrapping. It follows that a major aim of improving performance is to try to have as few instructions as possible wrap around the pipeline.

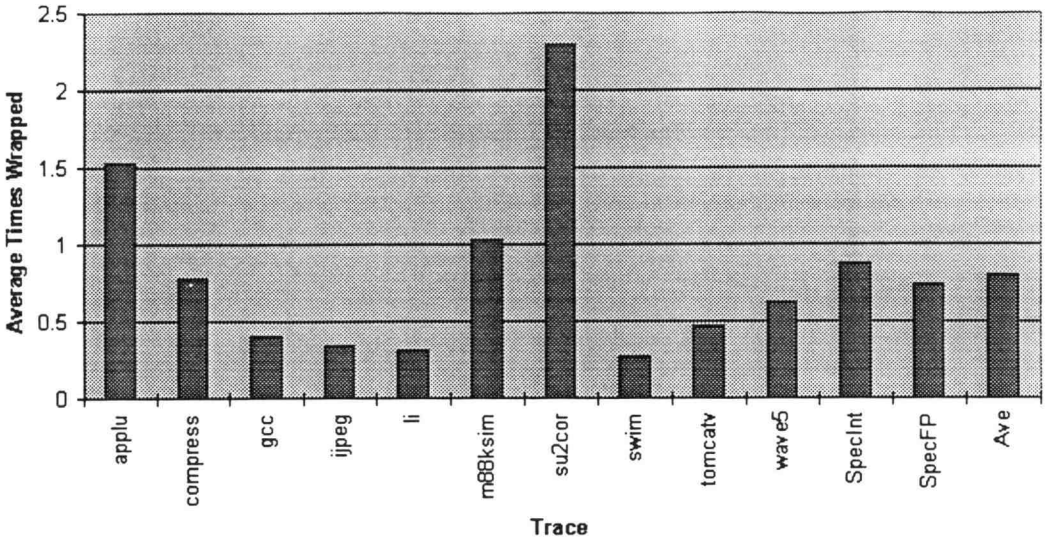

Figure 4.9 CDF Instruction Wrapping by Trace

Another useful metric is the average number of entries that are available in the ROB. Figures 4.11 and 4.12 show the distribution of available entries for both the SpecInt and SpecFP traces. These give an idea of the size of the instruction window being exposed during execution. It is impressive to note that in the pipeline configuration that was simulated there are only locations for 36 instructions (9 pipestages with a width of 4 instructions), but there averages over 60 instructions in the machine at a time for SpecInt traces and 80 instructions for SpecFP traces. Both sets of traces reach the limits of the 128 entry ROB, but the SpecFP traces actually appear to be limited by having only 128 entries. Indeed, 0.7% of the time the ROB size is limiting the number of instructions that can enter the processor. More instructions can be in the machine at a time due to the fact

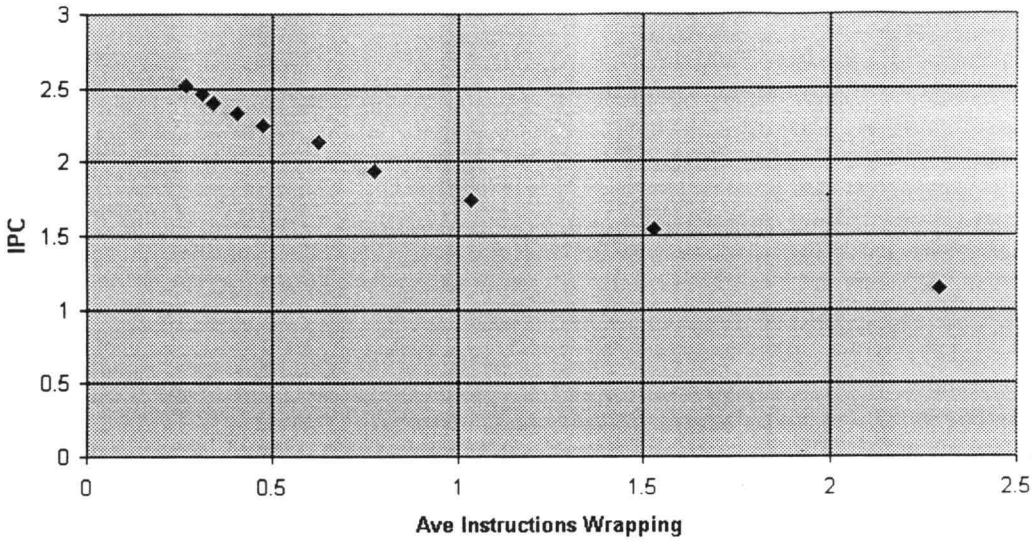

Figure 4.10 Instructions Per Clock Cycle vs. Instruction Wrapping

that once an instruction has entered an execution unit, it no longer needs to be kept in the instruction pipeline. When the results are eventually computed, the instruction's results are sent down the result pipeline tagged with the ROB tag of their originating instruction. These large instruction windows allow more of the program's inherent parallelism to be exposed and thus improving performance.

4.4 Chapter Summary

In this chapter, the microarchitecture of the counterdataflow processor has been described, shown a possible pipeline configuration, and given simulation results for 10 Spec95 traces. It's been shown that the CDF microarchitecture is an efficient means of exposing the available parallelism in a given program and

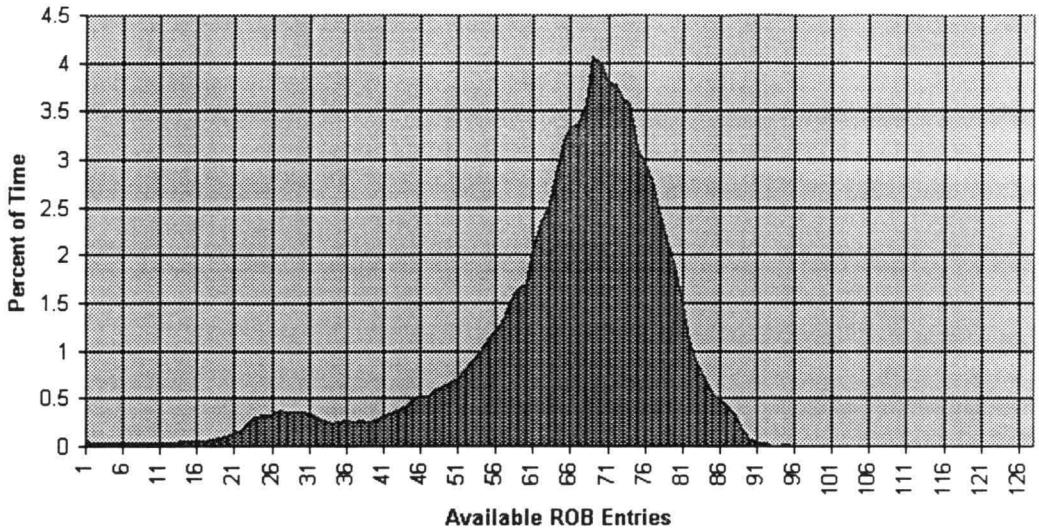

Figure 4.11 CDF ROB Available Entries for SpecInt

dispatching instructions to multiple execution units. The problems of the earlier CFPP architectures, pipeline stalling and single instruction issue, have effectively been solved and are no longer a bottleneck to higher processor performance.

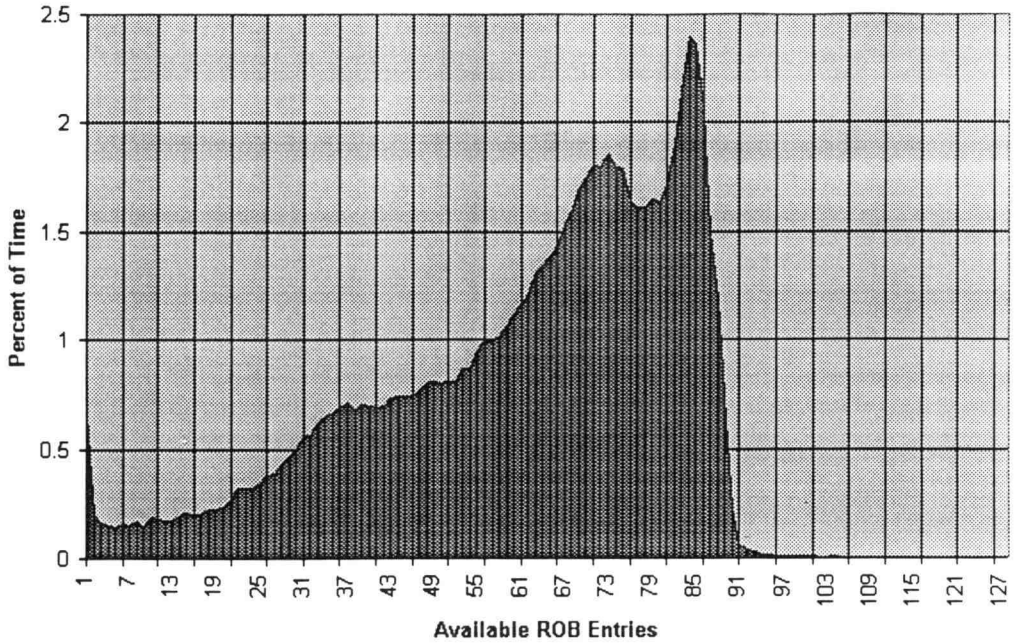

Figure 4.12 CDF ROB Available Entries for SpecFP

5. THE ARCHITECTURE-BLOCKS (aBlocks) SIMULATION PACKAGE

5.1 Goals of the Simulator

The aBlocks simulation package has been developed to give microarchitecture researchers, a package of tools that allows rapid microarchitecture prototyping and performance evaluation. This package advocates heavy object reuse, and provides users the flexibility to try out various subsystems without rewriting large sections of code. In most educational environments there is usually a wide variety of computing resources available. If it is possible to utilize all of the different types of computers, the amount of computational horsepower available can be considerably large. While it is theoretically possible to support multiple versions of software on different platforms, it is not our goal as designers to be involved in software support. The overall speed of the simulator is of course important, but is considered of secondary importance to the flexibility and speed of prototyping. Having graphical support in a language can be very useful for debugging as well as prototyping various architectures. As the requirements are platform independence, an object-oriented language, and graphical support, Sun Microsystems' object-oriented language, Java, was chosen with which to write the simulator [15].

5.2 Implications of Using Java

Java is a simple, object-oriented, distributed, , architecture neutral, portable, high-performance, multithreaded, and dynamic language [15]. This matches well with our requirements for a simulator. Since our goal is to be end

users of this product and not tool writers, the fact that the language is simple allows us to write quality code without spending too much time fighting against the language and tools. Java does not have pointers. This alone justifies using it in many people's minds. There is also automatic garbage collection. This frees the programmer from having to keep track of all of the memory used and making sure it gets deallocated. The programmer simply allocates whatever structures are needed. When they structures are no longer being used, the programmer simply lets go of them and the garbage collection thread cleans up and deallocates the memory that was being used.

Since Java is an object-oriented language, it encourages object reuse. Many different types of architectures share common functional units. For instance, many architectures have a ROB or reorder buffer unit. There is very little difference in a ROB from one architecture to the next. So, once the time writing and testing a ROB for one architecture has been invested, it would be preferable to reuse this section of code for another architecture. Since Java is object-oriented, it is straightforward to reuse the same or similar code in different simulations.

As educational environments have many varying software and hardware platforms, it would be desirable to use all of the available computational resources possible. Since Java compiles and runs on all of the major hardware platforms (Sun, Intel, HP, Apple) once a simulator is functioning it is possible to run it on all the computers available. An added benefit is that Java is aware of the World Wide Web. Since most of the currently available web browsers are capable of

running Java code, it is possible to use computing resources that are not even physically at the researcher's institution. Essentially, anyone willing to go to a web page can run simulations. It conceivable to think of the entire internet as one large distributed computation engine. With a Java based simulator the idle resources on the internet could be harnessed and used for many useful purposes.

Since the simulator will be used to debug and prototype new architectures, it must be capable of graphically displaying the simulator's information. Since Java comes with it's own built-in platform independent windowing and graphics library [17], it is possible to display all the necessary information in a platform-independent way without having to overly burden the programmer. This is another added advantage of using Java.

5.3 Simulator Methodology

One of the main goals of the simulator is flexibility. To this end, a few basic interface guidelines were decided upon to ensure that any object written for this simulator will be able to be reused transparently with any other object.

5.3.1 The Superclass Object - *aObject*

Every object/class in the aBlocks simulation package is a descendent of the *aObject* class. Any physical piece of hardware or information is represented as an *aObject*. This object defines the basic functions that allow all objects written to be used in an essentially plug-n-play manner. Replacing an object written for one architecture directly into another with little or no rewriting allows rapid prototyping and easy experimentation.

The *toString()* function is one function that all objects are required to implement, either by inheriting the function from an object that it subclasses or by it's own implementation. The *toString()* function takes no arguments and returns a string containing the state or status of the object. Any object can call any other object and request this string. This can be used when debugging to print out the state of an object. For example, when debugging the ROB, it's state could be output by calling *aROB.toString()*. In this manner, the intermediate state can be displayed to see that if perhaps something is going wrong internal to a given object.

The *give()* function is the fundamental interface between all objects. This is the manner by which all objects communicate. The *give* function takes a list of *aTokens* and returns a list of *aTokens*. An *aToken* is the basic unit of information in the simulator and will be described in the next section. The *give* function is easily misunderstood and is even more easily abused. The interface is bidirectional in nature, meaning either "I give to you" or "you please give me" depending upon the context of the communication and the object being called. It is very strictly enforced that the list of *aTokens* returned is the same length as that passed. While this may seem unorthodox, it has proven to be incredibly convenient and more importantly generic enough that all objects can perform the communication necessary to perform their duties.

5.3.2 The Information Object - *aToken*

As alluded to in the previous section, one of the basic *aObjects* in the simulator is the *aToken* object. This object is literally the information object.

Anything that is not actually a piece of physical hardware is simulated as an *aToken*. An *aToken* is used to represent either an instruction, such as ADD R1, R2, R3, or a piece of data (i.e. 0x1234, 45, etc.). Throughout this document, an *aToken* being used as an instruction will be referred to as an instruction token (*iToken*) while an *aToken* being used as a data element will be referred to as a data token (*dToken*).

Since the *aToken* has to play a variety of roles it is represented in a very generic manner. Table 5.1 shows the fields available in *aToken*. Most field's

Table 5.1 aToken Fields and Values

Type	Name	Description
long	value	an actual numeric value or the opcode if is an instruction
long	address	an address in memory
int	tag	the identifier for this token
int	phase	the state or status of this token
aToken[]	producers	for instructions, the tokens that this token produces
aToken[]	consumers	for instructions, the tokens that this token consumes

meanings change based upon whether the token is an instruction or data. For an instruction token (*iToken*), the value stores the opcode. If a functional simulation is being run, the value of a data token (*dToken*) holds any calculated values. In general, an *iToken* will use the producers array to hold the register(s) that it will produce when it is done executing and will use the consumers array to hold the

register(s) that it needs in order to execute. At this point, an example might be helpful. If the instruction ADD R1, R2, R3, meaning take the values held in registers 2 and 3, add them together and put the result in register 1, is to be represented with *aTokens*. Figure 5.1 shows how the *aToken* would be used to

iToken format

Value	Opcode for Instruction (i.e. ADD)
Address	Program Counter for this Instruction
Tag	Unique Identifier for this Instruction
Phase	{Identifies aToken as an iToken}
Producers[]	List of dTokens that will be produced
Consumers[]	List of dTokens that will be consumed

dToken Format

Value	In Execution Based Simulator, the register's value
Address	For memory operations, the memory address
Tag	The register number (i.e. R1, R2, R3)
Phase	{Identifies aToken as a dToken}

Figure 5.1 Instruction and Data Token Formats

hold all the necessary information. While the *iToken* and *dToken* are in reality the same object, the Producers and Consumers fields are generally unused in a *dToken*. This is not necessarily true however, and in certain dataflow type architectures it may be advantageous to use *dTokens* that consume and produce other *dTokens* (or for that matter *iTokens*). The many possible implementations of this object are left up to the end user to decide what is the most efficient method to represent the flow of information.

The phase field is the most complicated of all fields and truly holds most of the information of the token. The phase is stored as an integer, and forms a sort of tree of values, see figure 5.2. The diagram in figure 5.2 illustrates a typical

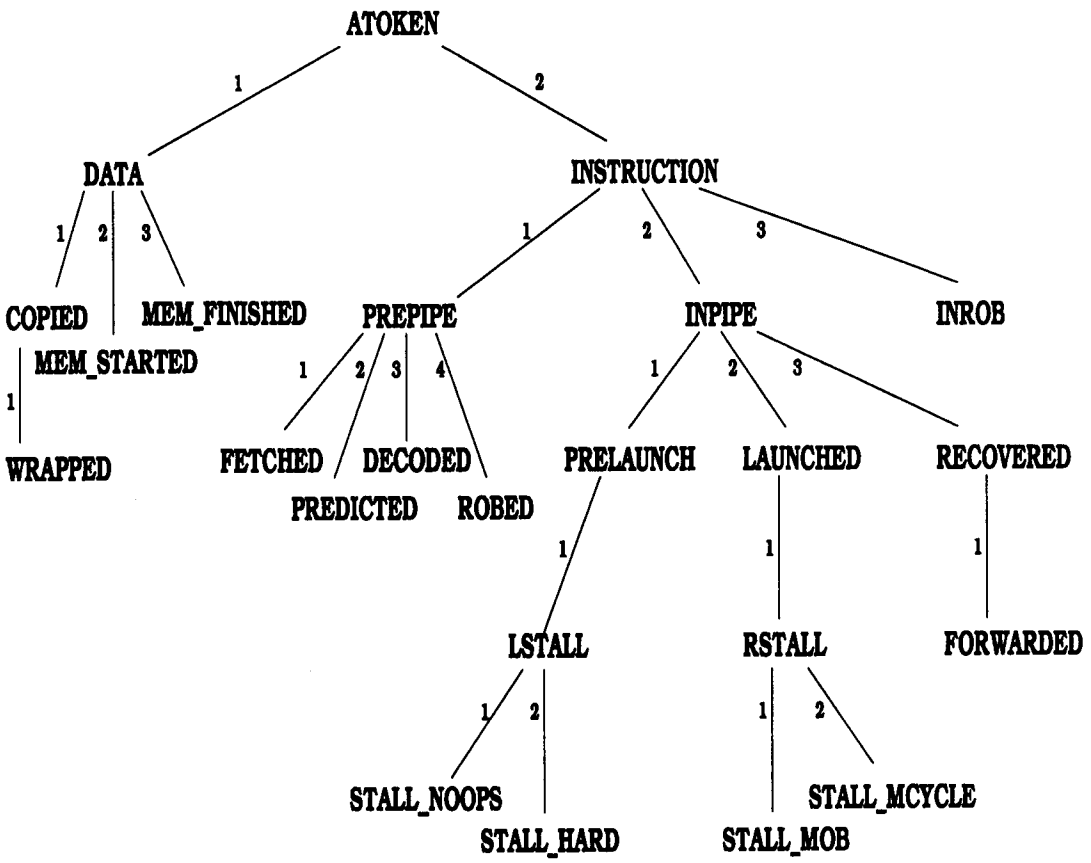

Figure 5.2 A Typical Phase Tree

phase tree. The *aToken* definition specifies that the most significant (MSB) is the valid/invalid bit. When set to '0' it indicates an invalid token and when set to '1' it indicates a valid token. Bits 30-27 store the data length. This value is used in a memory transfer operation and indicates the number of bytes that are to be

transferred. The rest of the word, bits 26-0 form the user definable part of phase. While this can indeed be defined as anything the programmer wants, the most flexible use of these bits is to form a phase tree. The phase tree is represented in

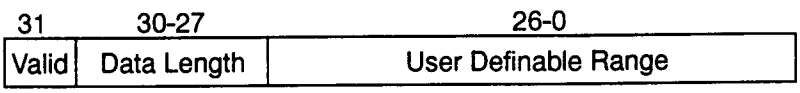

Figure 5.3 Phase Representation

an integer assuming 4-bits per tree depth and proceeding from LSB to MSB. To specify that a token is in the ROBED phase, one would specify that it is an INSTRUCTION (2), that it is PREPIPE (1), and that it is ROBED (4). This would be represented as 0x0412. Now, given this phase, suppose you want to ask if this token has phase PREPIPE (0x0012). The *aToken* method *isPhase* would proceed from the LSB, matching sets of 4 bits. If the phase in question reaches zero and the two phases match, then the phase in question is of that type and true is returned. This method of handling information has been found to be flexible, expandable, and allows various questions to be asked without having to change the representation of the tree.

5.3.3 The Statistics Interface - *stats*

The *stats* interface specifies the interface which an object must implement if it is to collect and return simulation statistics. The object must implement a *setStatLevel* function. This function takes an integer which specifies how much

Table 5.2 Phase Descriptions

Phase	Description
ATOKEN	The token is a token.
DATA	The token is a data token.
COPIED	The token has been copied to the result pipeline.
MEM_STARTED	The token's memory access has begun.
MEM_FINISHED	The token's memory access has completed.
INSTRUCTION	The token is an instruction token.
PREPIPE	The token hasn't entered the pipeline.
FETCHED	The token has been fetched.
PREDICTED	The token has been through branch prediction.
DECODED	The token has been decoded.
ROBED	The token has been through the ROB.
INPIPE	The token has entered the pipeline.
PRELAUNCH	The token has yet to launch.
LSTALL	The token has stalled waiting to launch.
STALL_NOOPS	The token has stalled on an operand hazard.
STALL_HARD	The token has stalled on a hardware hazard.
LAUNCHED	The token has launched to an execution unit.
RSTALL	The token has stalled on recovery from execution.
STALL_MOB	The token has stalled waiting on the MOB.
STALL_MCYCLE	The token has stalled due to a multicycle sidepanel.
INROB	The token has returned to the ROB.
RECOVERED	The token has recovered into the pipeline.
FORWARDED	The token has forwarded its results.

and what type of statistics to collect. The internal representation of what these values mean is left up to the individual objects. There is a corresponding

getStatLevel function which returns this integer so that other objects can find out what statistics are being collected and possibly use the statistics collected by another object. The *getStats* function recursively calls *getStats* on any objects that are below this object in the dependency tree of the simulator and returns it's own statistical information as well as the other information in the form of a string as output.

5.3.4 Other Basic Objects

In building this simulator, numerous objects have been built, all conforming to the aBlocks specifications. Table 5.3 lists some of the main objects that are in the package along with a brief description of their functionality. Not all of these objects were used in the development of the counterflow architectures and most were subclassed to account for small differences in their functioning in a counterflow environment.

5.4 Using *aBlocks* to Simulate Microarchitectures

The *aBlocks* package was used to simulate the three CFPP microarchitectures, the original counterflow pipelined processor (CFPP), the virtual register processor (VRP), and the counterdataflow processor (CDF). To create these architectures the basic *aObjects* were used and in some cases subclassed to create more specific "counterflow" objects. As examples of how to use *aBlocks*, the CFPP and VRP simulators will be explained. The CDF simulator is largely identical to the VRP simulator and will not be shown.

Table 5.3 Descriptions of Basic aObjects

aObject	Brief Description
aBranchExecUnit	Communicates with a Branch Predict Unit to decide the correctness of a branch instruction.
aBranchPredictUnit	Basic branch prediction. Randomly decide a branch's correctness.
aCache	Any level cache unit, implements LRU, SLRU, FIFO, RAND replacement algorithms.
aDecodeObject	Communicates with the prefetch unit, ISA Decode unit and processor to decode instructions.
aExecUnit	Generic fixed latency execution unit.
aMOBExec	Memory Order Buffer. Talks to ROB to decide on memory ordering.
aMemory	Generic main memory with a fixed latency.
aNBitBPU	Branch prediction unit implementing an N-bit history algorithm.
aObject	The Superclass object.
aPrefetch	Prefetch unit. Talks to branch prediction and fetch units to prefetch instructions.
aRFObject	Generic register file.
aROB	ReOrder Buffer.
aToken	The Information Object

5.4.1 Counterflow Pipelined Processor - CFPP

As the first implementation of a counterflow processor, CFPP encompasses most of the basic principles of the other architectures. Shown in figure 5.4 is the pipeline calling tree of a counterflow processor [11]. The calling tree is the order in which calls to the give function of objects is made. In most cases, only one external call from the top level of the simulator is needed to create

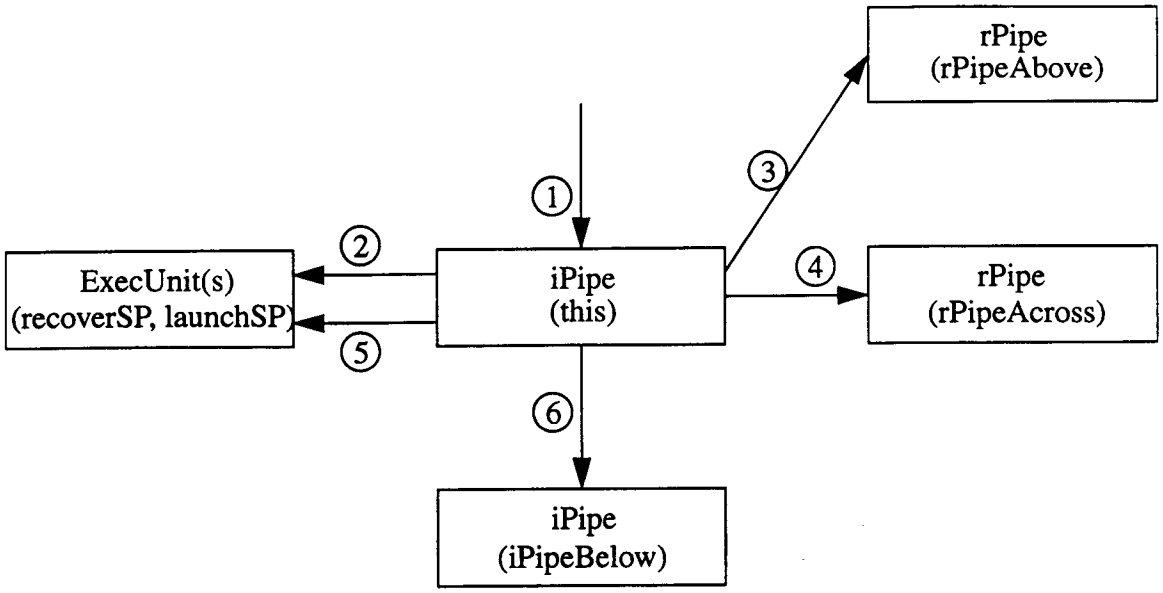

Figure 5.4 Basic Counterflow Pipeline Calling Tree

the calling tree and the resulting calls are then made by the objects themselves to generate the actions required. In this example, the top level simulator makes one call to the iPipe asking it to "give" some tokens. The iPipe then takes care of calling the other four objects with actions 2 through 6. This basic tree remains essentially unchanged for all three architectures. The calling tree is numbered in order from 1 to 6, with the numbers circled in the figure.

1- Some object calls the instruction pipeline (iPipe) (this) with *give* with a list of tokens requesting a number of aTokens. It is important to note that the requested object must return the same number of tokens as has been requested. If the number requested is zero, it means that the requesting object cannot accept any new tokens at this time. This is the mechanism by which stalling occurs. If

the number requested is nonzero and the requested object cannot return that many objects, these spaces must be filled with null tokens. It is the requesting object's responsibility to check for nulls and process them accordingly.

2- The instruction pipeline (this) calls all of the execution units which recover at this point and asks if they have any tokens which have finished being executed. If they have, the instruction pipeline takes the tokens. This is one of the two points where the instruction pipeline can be told to stall. If the execution unit at this stage sees that the instruction pipeline has a token which it is executing, but which has not completed executing, the instruction pipeline must stall. The execution unit takes the token, changes the phase to RSTALL for recover stall, and returns the token to the instruction pipeline. The instruction pipeline seeing that it must stall does not request any new tokens in step number six.

3- The instruction pipeline checks the result pipeline (rPipe) above it for multiple items. It compares the tokens in the result pipeline for results that it might need in it's consumers arrays. If it finds any that have the same tag match and are valid, it takes the values from the result pipeline and marks it's consumer token as valid. At the same time, if the iToken has completed executing and there are empty locations in the result pipeline, it puts it's producer tokens in the empty spots, and marks the producer as phase FORWARDED.

4- The same actions that occurred in step number 3 are repeated for the result pipeline across from this instruction pipeline. Because both the instruction pipeline and result pipeline are moving at the same time, the instruction pipeline

has to check with both the result pipeline above and across from it to make certain that data doesn't pass without being inspected, resulting in incorrect execution.

5- At this step, the instruction pipeline calls all of the execution units which can launch at this point and asks them if they can execute any of the tokens that are being held. If the execution unit can execute the token and all of the token's consumers are valid it takes the token, marks the phase as LAUNCHED, keeps a copy of the token, and returns the original to the instruction pipeline. This is also the other time which the instruction pipeline may be required to stall. If the execution unit is the last execution unit which can process this type of instruction and the instruction does not yet have all of it's consumer tokens, the instruction pipeline will be required to stall. In this case, the execution unit will detect this case, mark the token's phase as LSTALL for launch stall and return the token to the instruction pipeline. There it will remain until it receives all of it's tokens when it will be allowed to execute normally.

6- At this point, all of the necessary internal transferring of information has taken place. If the instruction pipeline was not stalled, it will request a list of tokens from the instruction pipeline below it to replace the tokens that it will give to the object which called it in step number 1. The instruction pipeline below will go through these same steps on it's own calling tree, eventually returning a list of tokens. These will become this instruction pipeline's new tokens and it will return it's old tokens to the object which called it in step number 1.

The calling tree described above shows only the interactions between the two pipelines and the execution units. Figure 5.5 shows the full calling tree of a

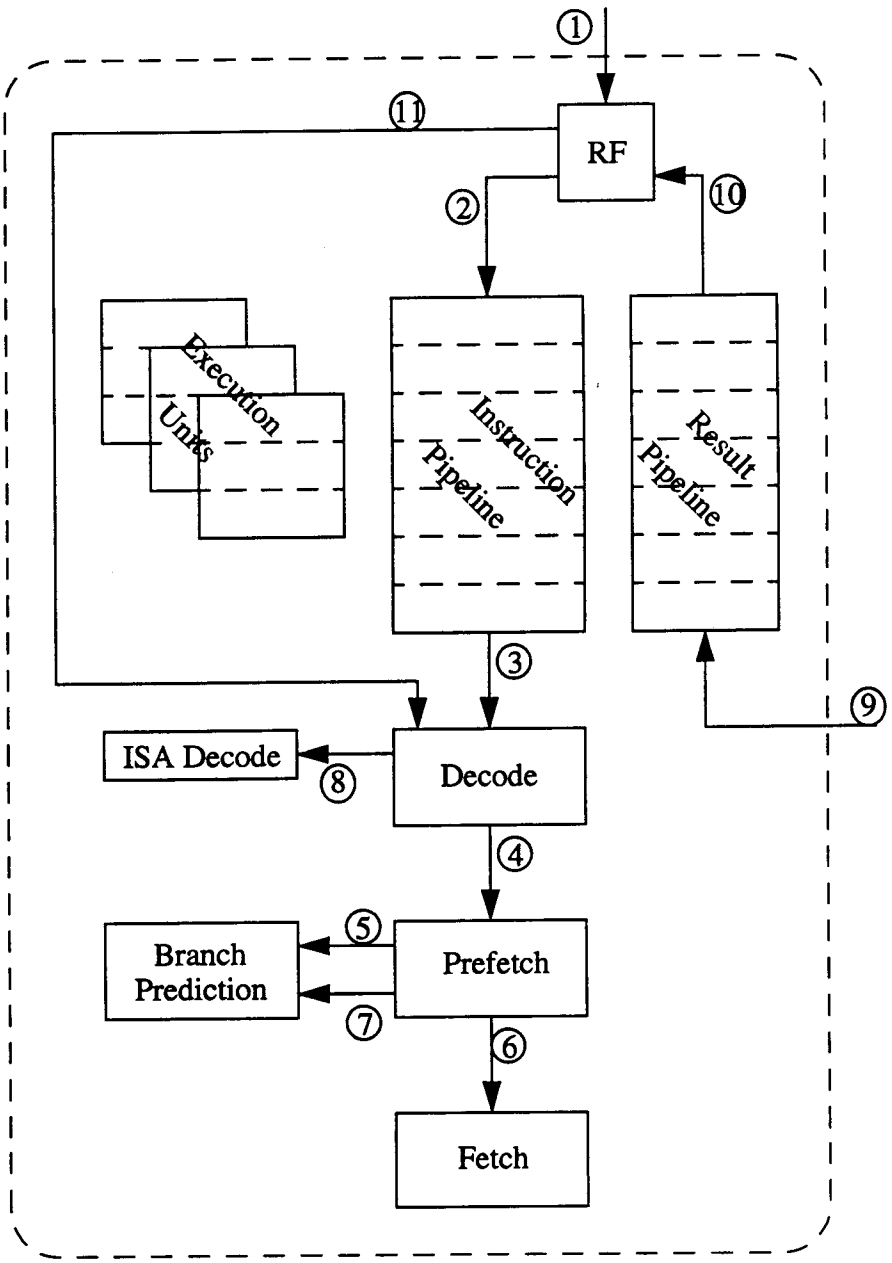

Figure 5.5 CFPP Microarchitecture Calling Tree

CFPP processor implementation. It needs to be mentioned that this is only one possible implementation of this microarchitecture. As long as the full cycle of actions is completed, this specific calling tree does not have to be followed. This

tree just happens to be the implementation that seemed to make the most sense at the time. The following steps, labelled from 1 to 11 in figure 5.5 make up one full clock cycle.

1- This is the start of one clock cycle for the processor. The top-level of the simulator calls into the register file to begin the calling cycle.

2- The register file calls the top of the instruction pipeline requesting a list of tokens the width of the instruction pipeline. The tokens eventually returned are the instructions which have completed and are then written back into the register file. In the case of CFPP, which can only execute a maximum of one instruction per clock cycle, this is a list of one token. This call results in a cascade of calls described in figure 5.4 that calls down the instruction pipeline, launching tokens, recovering tokens, and communicating with the result pipeline. However, once the pipeline calling tree has been debugged, it can be treated as one entity and used transparently.

3- The bottom of the instruction pipeline calls to the decode object to get a new list of iTokens that have been decoded and are ready to enter the pipeline.

4- The decode object calls the prefetch object for a list of iTokens that have been fetched and are ready to be decoded.

5- The prefetch object calls the branch prediction for a list of dTokens to be fetched. These dTokens contain the addresses from which to fetch instructions.

6- The prefetch object uses the dTokens from the branch prediction object to request instructions from the fetch object. Depending on whether the simulator is trace based or execution based, the fetch object either reads from a trace file or

calls a memory and requests the instructions at the addresses in the dTokens.

7- The prefetch object takes the list of iTokens from the fetch object and presents them to the branch prediction object. The branch prediction object looks through the iTokens for branches. If it finds a branch, it uses its prediction algorithm and guesses the direction it believes the branch will take. If the branch was not taken, the rest of the iTokens in the list are returned to the prefetch object. If the prediction determines that the branch was taken, the rest of the iTokens are invalidated and the predicted PC is calculated to be used during the next cycle.

8- At this time, the decode object has received a list of tokens to be decoded. It now calls the ISA specific decode object to have the iTokens decoded into the ISA being used. Currently the only ISA that has been written is for the SimpleScalar ISA. This ISA specific decode object must be handwritten for each ISA that needs to be run. At this point, the full iToken is formed, with the value field filled in with the integer value of the opcode, lists of consumers and producers formed, and any other ISA specific functions performed.

9- This step is another top-level simulator call. A call is made to the bottom of the result pipeline. This call ripples up the result pipeline essentially causing all data to move down one stage.

10- The top of the result pipeline makes a call to the register file to request a list of dTokens to pass down the pipeline to waiting instructions in the instruction pipeline.

11- The register file needs to know what dTokens to read in order to fill the request from the result pipeline in step number 10. Therefore it calls the decode

object which by this time holds the iTokens which have just been fetched. The consumers of these iTokens are returned to the register file which in turn gives the values to the result pipeline.

5.4.2 Virtual Register Processor - VRP

The virtual register processor is built in largely the same manner as the counterflow pipelined processor. The basic pipeline calling tree is in principle the same, however the calling order is vastly different. The main hardware addition is the reorder buffer or ROB, however since the objects all use the standard aBlocks interface most of the same objects as CFPP are used, they are only called in a different manner. Figure 5.6 shows the full calling tree of a virtual register processor. It must again be stated that this is only one method of arranging the tree. Other organizations could be just as correct, it was simply felt that this was the most straightforward tree at the time.

1- The top-level simulator makes a call to the top of the instruction pipeline to begin the calculations for one clock cycle of the simulator. This cascades into the same set of calls that the CFPP pipeline calling tree generates in the previous section.

2- The bottom of the instruction pipeline makes a call to the decode object to get a new set of decoded tokens to bring into the pipeline.

3- The decode object makes a call to the result pipeline object to get the results for this clock cycle. This step is necessary to prevent data from passing at this boundary between putting an iToken into the instruction pipeline and removing a dToken from the result pipeline. If this step is omitted instructions

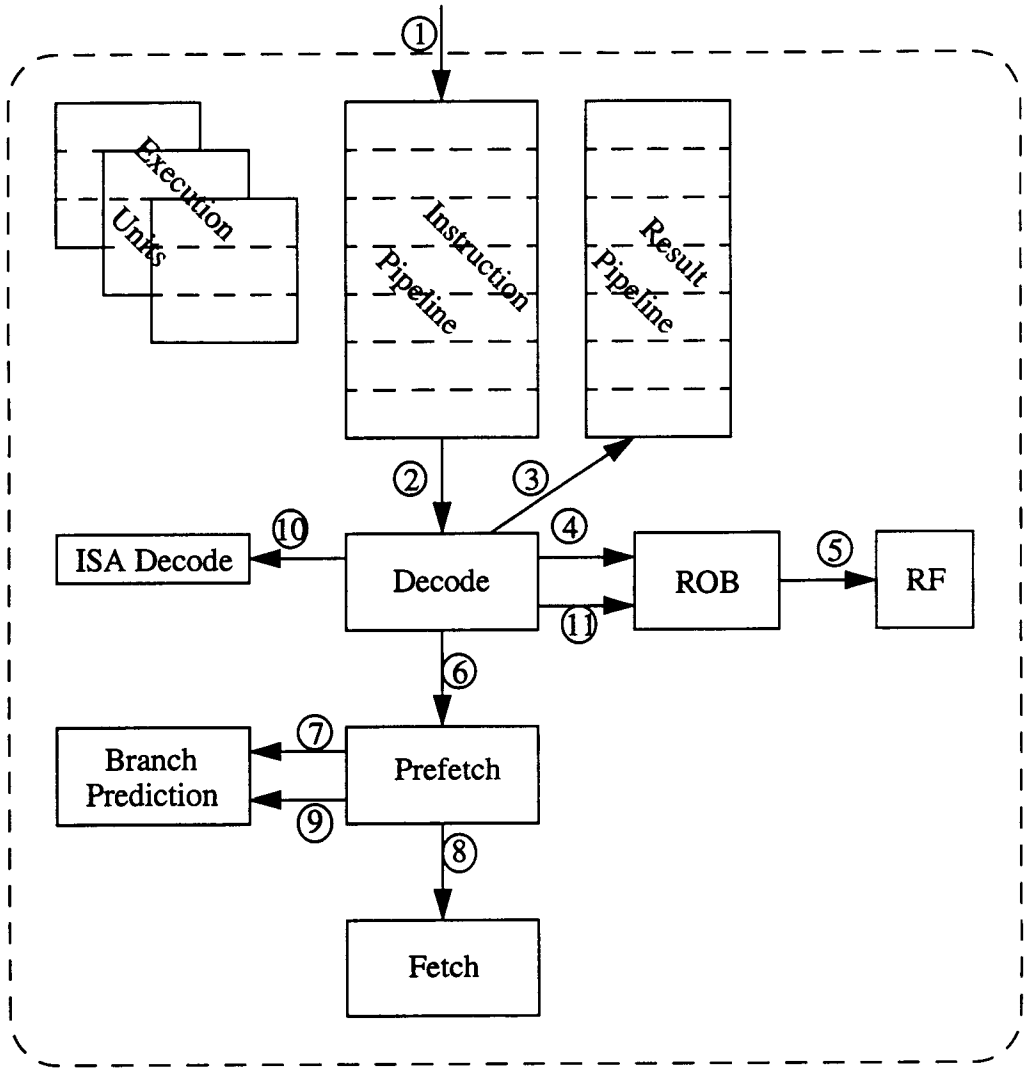

Figure 5.6 VRP Microarchitecture Calling Tree

entering the instruction pipeline can miss their required data that is leaving the result pipeline, resulting in incorrect execution.

4- The decode object takes the results from the result pipeline in the form of a list of dTokens and gives them to the ROB to be retired.

5- The ROB takes the list of completed dTokens and if they belong to the oldest instructions in the machine retires them by writing their values to the register file for permanent storage.

6- The ROB and register file have done their respective jobs by this point, and the decode now returns to it's original job of getting new instructions to be decoded. To this end, it requests a list of iTokens from the prefetch object to decode.

7- The prefetch object calls the branch prediction for a list of dTokens to be fetched. These dTokens contain the addresses from which to fetch instruction.

8- The prefetch object uses the dTokens from the branch prediction object to request instructions from the fetch object. Depending on whether the simulator is trace based or execution based, the fetch object either reads from a trace file or calls a memory and requests the instructions at the addresses in the dTokens.

9- The prefetch object takes the list of iTokens from the fetch object and presents them to the branch prediction object. The branch prediction object looks through the iTokens for branches. If it finds a branch, it uses its prediction algorithm and guesses the direction it believes the branch will take. If the branch was not taken, the rest of the iTokens in the list are returned to the prefetch object. If the prediction determines that the branch was taken, the rest of the iTokens are invalidated and the predicted PC is calculated to be used during the next cycle.

10- At this time, the decode object has received a list of tokens to be decoded. It now calls the ISA specific decode object to have the iTokens decoded into the ISA being used. Currently the only ISA that has been written is for the

SimpleScalar ISA. This ISA specific decode object must be handwritten for each ISA that needs to be run. At this point, the full iToken is formed, with the value field filled in with the integer value of the opcode, lists of consumers and producers formed, and any other ISA specific functions performed.

11- The now decoded iTokens are given to the ROB. The first thing the ROB does is attempt to give the iToken an entry. If it fails because the ROB is full, the unchanged iToken is given back to the decode unit who has to hold onto it and the decode unit is forced to try again the next clock cycle. If there is an available entry, the iToken gets renamed to the ROB entry's tag. The ROB also searches the iToken's consumers to see if it is holding a renamed producer. If it is, the consumer is renamed to the associated ROB entry that will eventually produce it's value. If not, the value is read from the register file and given to the consumer. At this point, the new iToken is returned to the instruction pipeline and one clock cycle has been completed.

5.4.3 CounterDataflow Processor - CDF

The counterdataflow processor has also been implemented in the aBlocks simulations package. It is, however, very similar to the VRP processor. The main differences being that the instruction pipelines can be wider than one iToken and that the instruction pipelines and result pipelines wrap around. The calling graphs are almost identical except for the fact that at the end of the results and instruction pipelines one more call is made to cause the values to wrap around the pipelines. Since the calling trees are so similar, the CDF tree will not be covered here.

5.5 Chapter Summary

An architectural simulation suite, aBlocks, has been written by microarchitects for microarchitects. A variety of common objects have been written in Java. Since Java is an "architecture neutral" language, these objects can be used on just about any modern processor without any porting. While Java is slower than other compiled languages, the ability to run on multiple platforms and "just in time" compiling make up some of the lost speed.

6. FUTURE EXTENSIONS OF CDF

Each time a new architecture was developed it was always in response to a shortcoming in the previous architecture. At the moment, CDF appears to have overcome all of the major shortcomings of a new architecture. It handles long latency instructions without a problem by removing them from the pipeline until they complete. This allows other instructions to continue executing without getting backed up waiting for this operation to finish. It facilitates deep instruction speculation by acting as a distributed reservation station where instruction dependencies are resolved within the pipeline. There are some areas which still need to be investigated, both from an implementation standpoint and to find further performance increases.

6.1 Distributed Reorder Buffer

In the current implementation of CDF, the reorder buffer (ROB) is in great demand. For a CDF pipeline, illustrated in figure 6.1, with four instruction pipelines and four result pipelines, the requirements on the ROB are eight write ports and eight read ports. This assumes an instruction format where there is a maximum of two operands. This is the maximum number of ports that the ROB may need to be able to handle. Unfortunately, in CDF the instruction and result pipelines do not stall, so it is possible that all of the pipelines could be full and need processing by the ROB. In this case, the ROB has to be able to handle all of these requests simultaneously. It is possible that the pipelines could be altered to allow instructions and data to flow past the ROB, but the ROB then becomes the

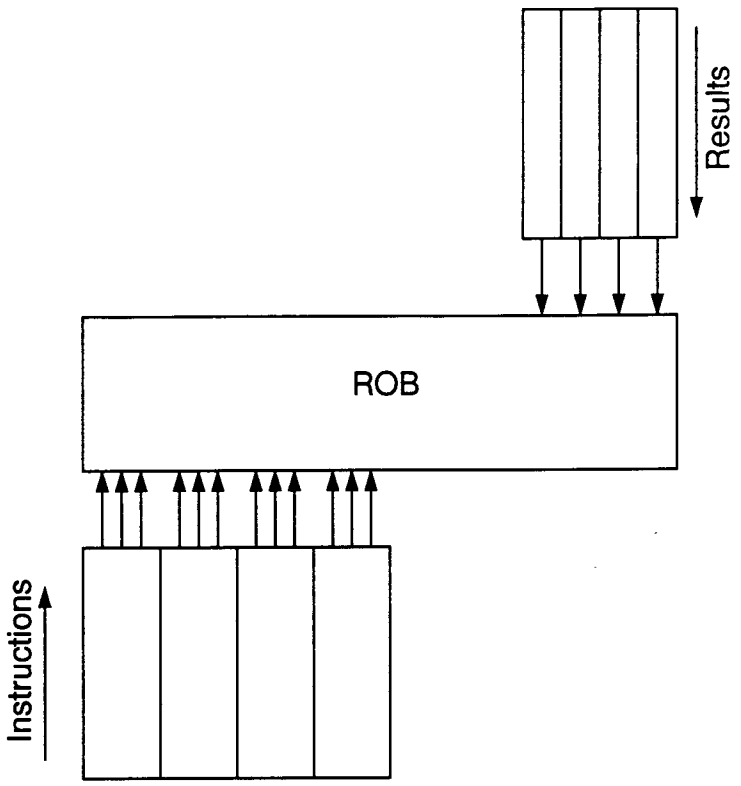

Figure 6.1 Reorder Buffer Interactions with the CDF Pipelines

bottleneck for new instructions to enter the processor. Having many read and write ports on the ROB increases the ROB's complexity and increases the time required to access the data. This limits the maximum clock speed at which the processor can run.

CDF acts as essentially a distributed reservation station, where instructions and data are matched as they both flow through the pipelines [18]. The natural extension of this paradigm is to attempt to distribute the ROB around the pipeline. In effect, segmenting the ROB and doing some matching at various locations around the pipeline. The only extra hardware needed to implement this scheme is

a table to hold the register aliases and adding a field to the register file to hold the ROB entry which will eventually write the data back.

The extra hardware needed is shown in figure 6.2. Shown are the register alias table (RAT) and the modified register file. For this illustration, it is assumed that there are two instruction pipelines and that the ROB can hold, in total, four instructions at any given time. The RAT is organized as a circular buffer, with new entries being added at the head pointer and old entries being retired from the tail pointer. When the head and tail pointers point to the same location, the RAT is full, and can hold no more instructions until the oldest instruction completes and can be retired. The "pipe" field shows which instruction pipeline the instruction was dispatched into. The "register" field shows which register, in the register file, that this instruction will write its results into when it completes. The "last" field points to the RAT entry which previously was going to write to the same register. This is used in case of an incorrectly speculated branch. The instructions after the branch must be removed from the pipeline and the RAT and RF must be returned to the state they were in before the branch occurred. The "last" field is used in this case so that the RAT does not need to be associatively searched. If this is the only instruction which is going to write to this register, this entries own RAT number is put in the last field. The register file performs the same functions as a standard register file, with the exception of the addition of the "alias" field. This holds the RAT entry which will eventually write into this register. This field is provided to allow the "last" field of the RAT to be updated by reading it directly from the register file.

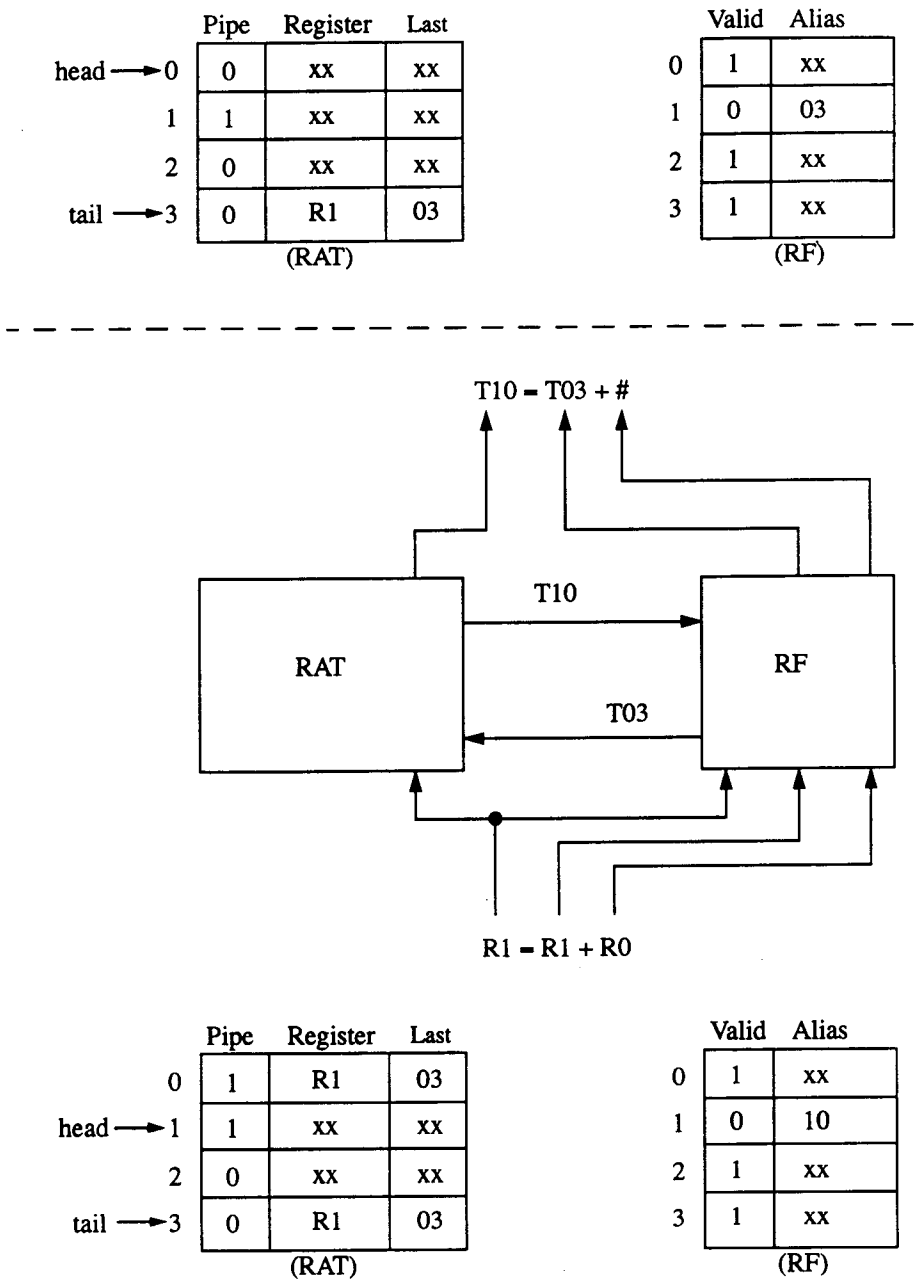

Figure 6.2 Register Alias Table and Modified Register File

Figure 6.2 shows an example of how the RAT and RF together are used to process an instruction. At the top, the RAT and RF are shown initially. They have

only one outstanding instruction. Some instruction is in the pipeline and will eventually write to register R1. This instruction has been put into the instruction pipeline 0 and given the RAT tag of T03. The "0" in T03 indicates that the instruction is in instruction pipeline 0 and the "3" indicates that it has been put into the third RAT entry.

At this point, a new instruction needs to be issued to the pipeline. Assuming that this instruction performs the function, R1 = R1 + R0 and there is room in instruction pipeline 1 for this instruction, the following actions occur. The register file is read to see what the values of R1 and R0 (the consumers) are. Since R0 is already valid, the actual numerical value is given out. R1 is to be processed by the first instruction and so that instruction's alias, T03, is given in the place of R1. This new instruction will eventually write it's result to register R1. The head of the RAT is pointing to entry 0 and since this new instruction is going into pipeline number 1, the instruction is given the tag T10. At the same time, the old instruction's alias is read out of the RF and written into the "last" field of this instruction's entry. After being processed by the RAT and RF, the translated instruction looks like, T10 = T03 + #. This new instruction is launched into instruction pipeline 1. This entire lookup process was accomplished without making any associative memory accesses, therefore this step in the pipeline can be fast and not limit the performance of the processor.

At this point, a short example may help to clarify just how this whole process allow the ROB to be broken up and still maintain consistent data across the individual ROBs. Figure 6.3, shows a small pipeline which will be used to step

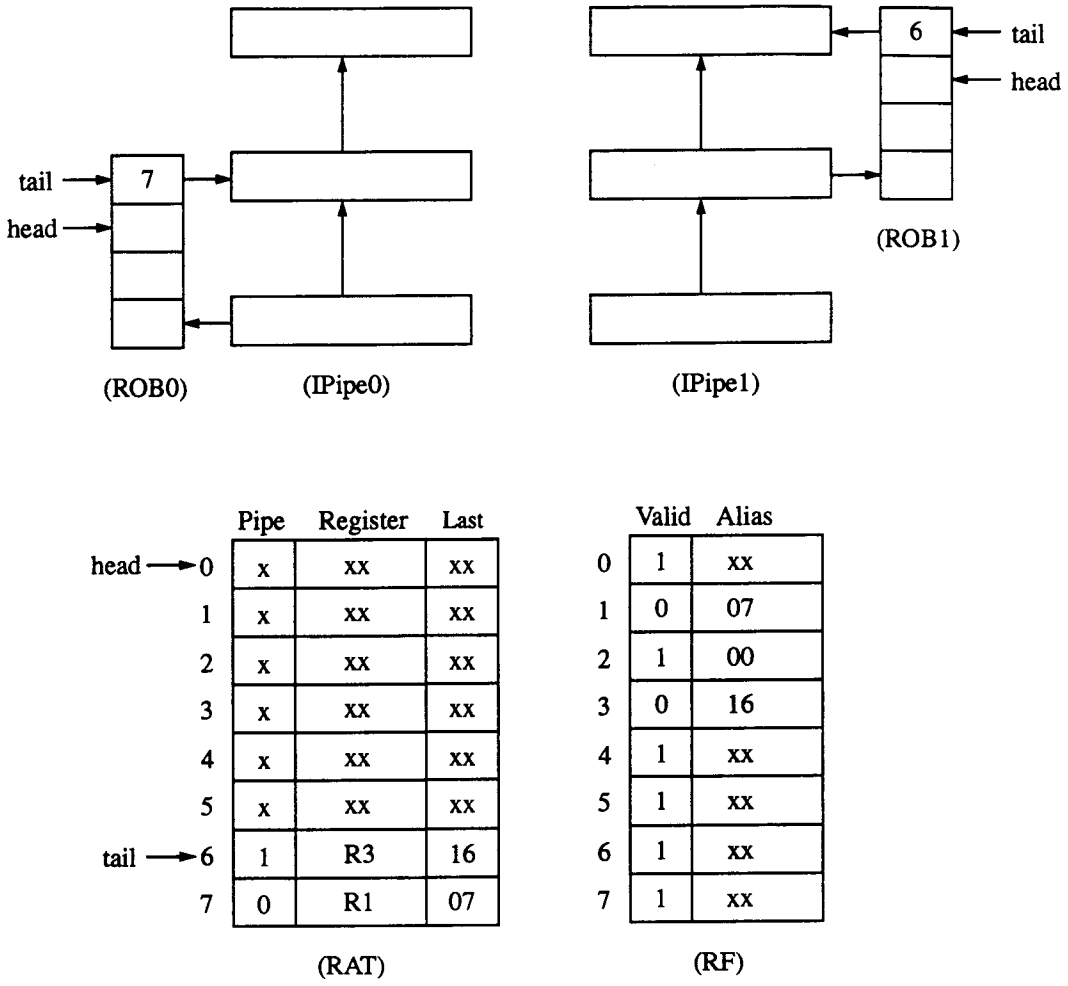

Figure 6.3 ROB Example (Initial State)

through an example. This example has two instruction pipelines, IPipe0 and IPipe1, each of them being three stages long. There are two ROBs which hold four entries each as well as a RAT which holds eight entries. At this starting point, two instructions are in the processor somewhere. One instruction has been dispatched to IPipe0 and will eventually write back to register R1. This instruction has RAT/ROB tag of T07. The other instruction has been dispatched to IPipe1 and will eventually write back to register R3. This instruction's RAT/ROB tag is

T16. Notice that ROB0 has an entry for T07 and ROB1 has an entry for T16. ROB0 only holds entries for IPipe0. Similarly, ROB1 only holds entries for IPipe1. Since all instructions know which pipeline the instruction they are looking for were dispatched into, they also know which ROB will hold that instruction. In this way, the number of times an individual ROB needs to be accessed is reduced. If an instruction is looking for a result tagged T13 for example, it knows by definition that it doesn't have to bother checking any other ROB other than ROB1.

In the next clock cycle, two instructions go through the RAT/RF renaming process. The first instruction, R2 = R1 + R0, will be dispatched to IPipe0. The second instruction, R5 = R4 + R3, will be dispatched to IPipe1. Figure 6.4 shows the actions which occur to begin the processing of these instructions. Starting with the first instruction's operands, R1 and R0. These operands are read out of the register file, since R0 is already valid its value is given. The register R1 is going to be generated by the instruction which has tag T07, so that tag is given in place of R1. The head of the RAT points to entry number 0 and since the instruction is being issued to IPipe0, this instruction gets tag T00. This can be observed at register R2's location in the register file where the alias gets set to 00 as well as in entry 0 of the RAT itself. The second instruction, R5 = R4 + R3, occurs at the same time as the first instruction with the same actions occurring. The operands R4 and R3 get their values from the RF. The values for R4 and R3 are the value in R4 and the tag T16 respectively. Since the second instruction is being issued to IPipe1 and the RAT's head pointer effectively points to entry 1, this instruction gets tag T11. The register file records that the instruction with tag T11

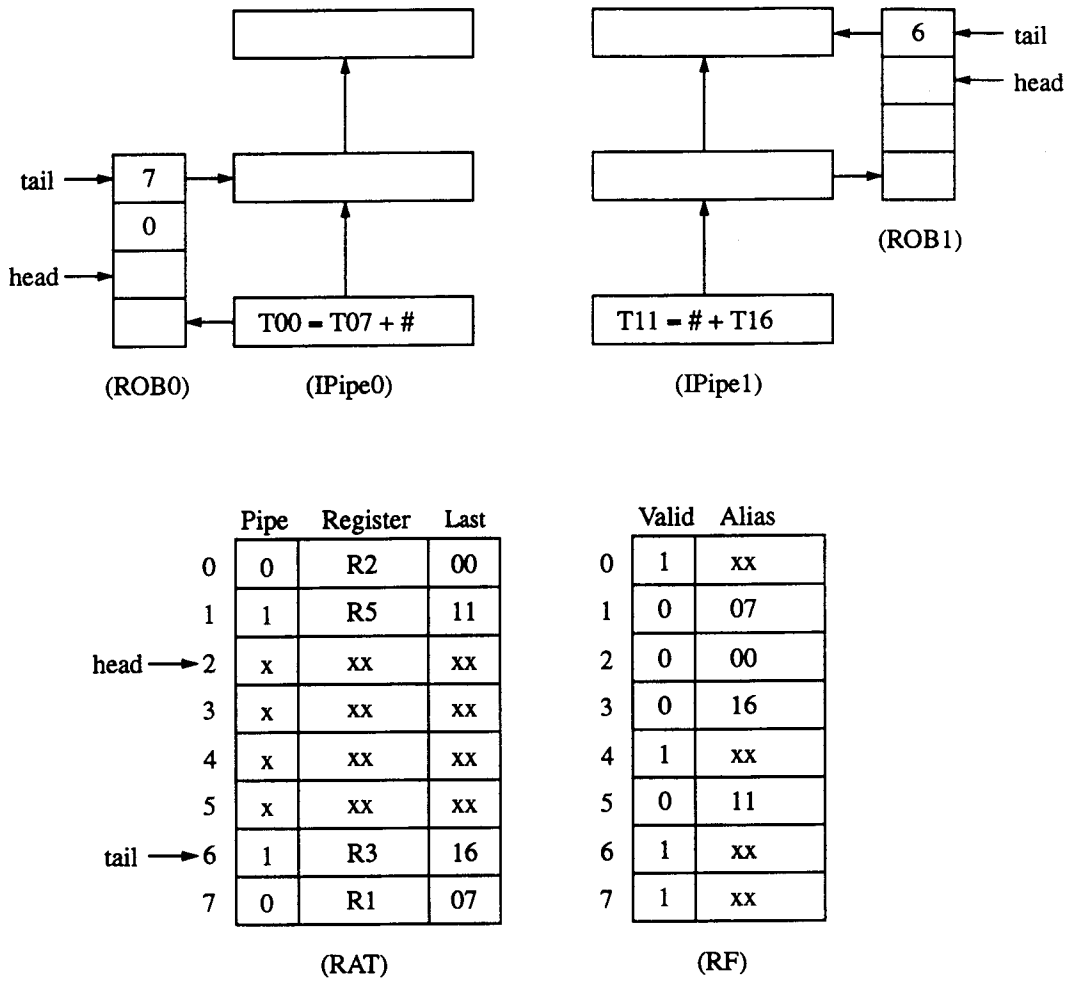

Figure 6.4 ROB Example (Two Adds Enter Pipeline)

will eventually write to register R5. In the pipeline itself, ROB0 has seen the first instruction. It puts the instruction tag 0 into it's smaller ROB and updates its head pointer. The second instruction's ROB is located farther up in the pipeline, so ROB1 has not yet seen the instruction tagged T11. This completes the first clock cycle.

In the next clock cycle, two more instructions enter the processor, figure 6.5 shows the state after they have been processed. The first instruction is another

ADD operation performing the function, R6 = R2 + R5, which will be issued to IPipe0. The second instruction is a branch which will be mispredicted, labelled Branch R5. At a later time, when this misprediction is realized, this branch and all other instructions issued after it will need to be removed from the processor and the state of all ROBs, the RAT, and the RF will need to be returned to their state from before the branch.

The first instruction needs to read R2 and R5 from the RF getting the tags of the instructions which will be generating these register's values as T00 and T11 respectively. This instruction is being issued to IPipe0 and the RAT's head pointer points to entry 2, so the generated tag for this instruction is T02. Since this instruction gets to its ROB in the first stage, ROB0 takes the tag and updates its head pointer. The second instruction, the branch, is assumed to not need to read any values from the RF, but for some reason it writes a value back to register R5. Even if the branch didn't have a result to write back, it still needs a RAT/ROB entry number so in this example, R5 is used. Since the branch is being issued to IPipe1 and the RAT's head pointer essentially points to entry number 3, the generated tag is T13. It is important to note that since the branch, with tag T13, and the instruction with tag T11 are both writing to register R5, the "last" field is filled in appropriately. The branch's "last" field points to tag T11 as the instruction which was going to write to register R5 before. This is important because when the branch is removed later, this value will have to be replaced in the register file so that R5 will be updated by the instruction with tag T11. This will be explained in more detail later. As the second ADD, the one that will write to tag T11, has

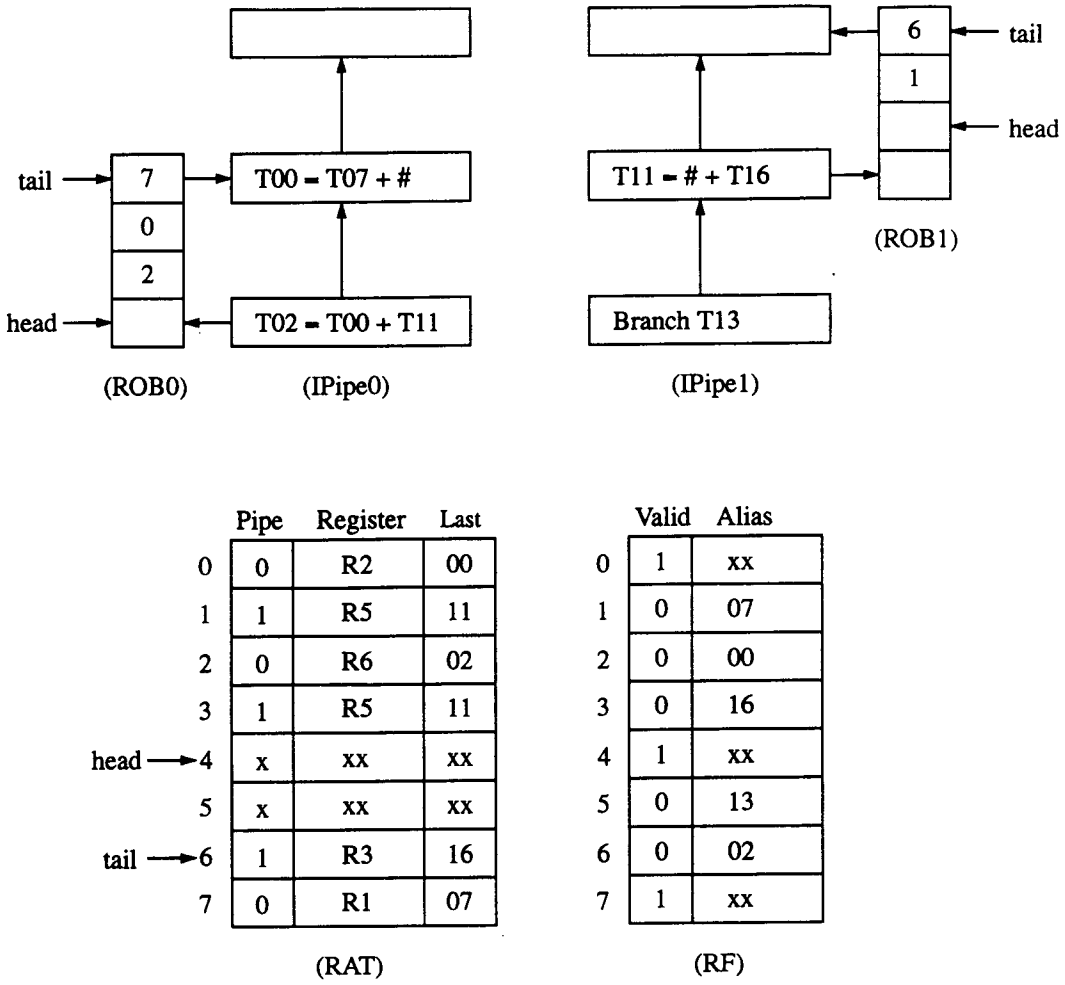

Figure 6.5 ROB Example (Add and Bad Branch Enter Pipeline)

advanced a pipestage since the last clock cycle, it has now been written into ROB1.

Figure 6.6 shows the machines state after another ADD instruction, R7 = R2 + R5, enters the processor before the wrong branch has been detected. This instruction is in the shadow of the wrongly speculated branch and therefore should never have been executed. It enters the processor because this is a speculative architecture and most of the time the branch prediction guesses

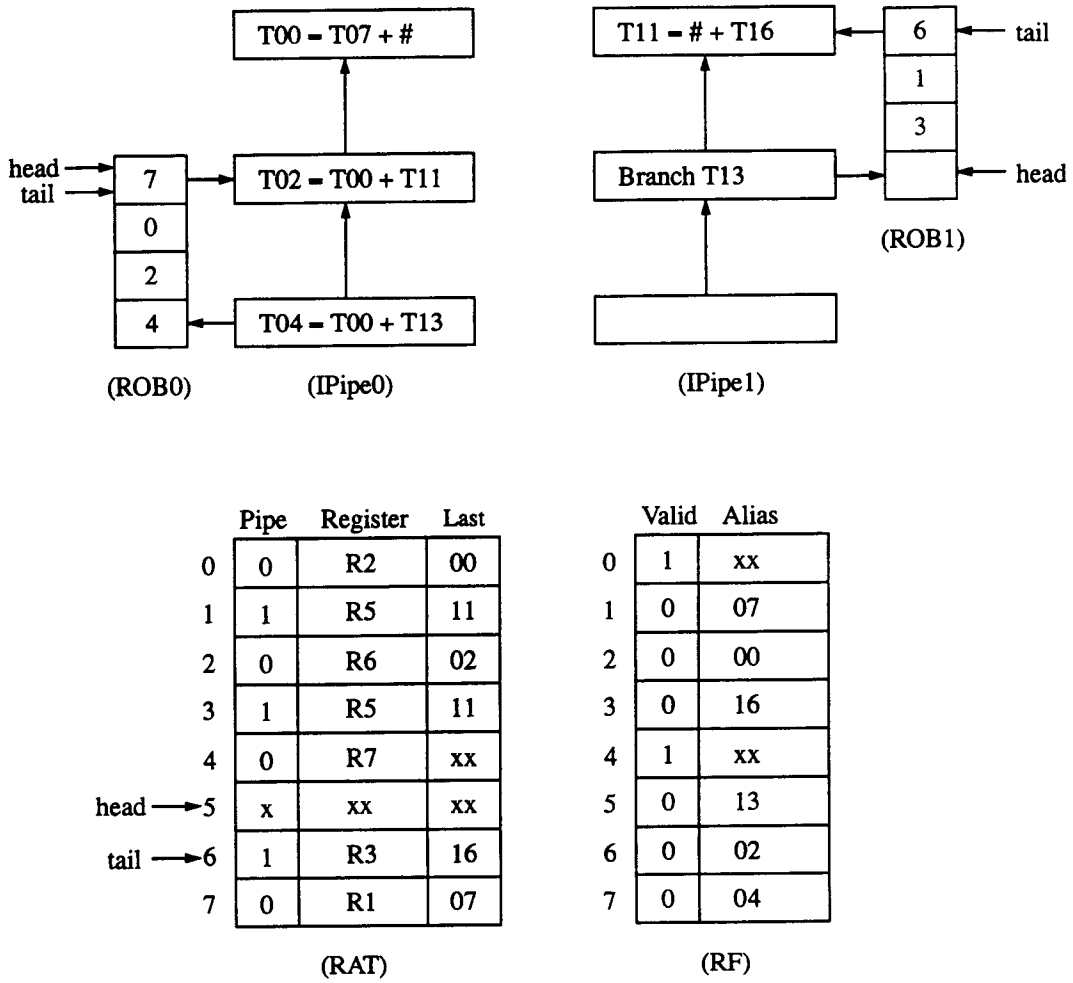

Figure 6.6 ROB Example (A Speculated Add Enters Pipeline)

correctly. In the case where the branch prediction guesses correctly, no work was lost while the branch was being processed. Unfortunately, in the incorrect prediction case, any actions caused by this instruction need to be undone. The ROB itself makes certain that the result of this instruction is never written back to permanent storage, but now the RAT and RF have to clean up their tables when the branch gets resolved. For now, this instruction is treated as any other. It

reads its operands R2 and R5 from the RF and gets the tags T00 and T13 respectively. This instruction is being issued to IPipe0 and the RAT's head pointer is at entry 4, so the generated tag is T04. ROB0 writes this instruction's tag into itself. At the same time all this has occurred, the branch instruction has moved up a pipestage and ROB1 has written the branch's tag into itself.

At this point, it is assumed that the wrongly predicted branch has been discovered. To make things simpler, it is also assumed that all of the other instructions have not executed and are still in the pipeline as shown in figure 6.6. The branch execution unit, which discovered the mistake, tells the RAT that the instruction with tag T13 was a wrong branch. The RAT now knows that all the instructions between entry 3 and its head should not have been in the pipeline and must be cleared. In this case, this amounts to removing entries 3 and 4 from the RAT. It sends a message to the individual segmented ROBs telling them to invalidate the instructions in that range. All that is left to do is put the register file back in order. To do this, it looks at the "last" field in the entries of the RAT that it is clearing. Entry four's "last" field points to itself, so register R7 is marked valid and the value contained in the RF is the correct value from before this instruction. Entry three's "last" field points to tag T11. By checking the RAT's entry 1, it is observed that this instruction has not yet written back. Since the instruction has not yet completed, the RF entry for R5 has its "alias" field set to tag T11 since it's value will now be coming from that instruction. Register R5's valid bit is not set in this case. If the instruction with tag T11 had completed, the correct value would have already been written back to the RF, and the valid bit would need to be set.

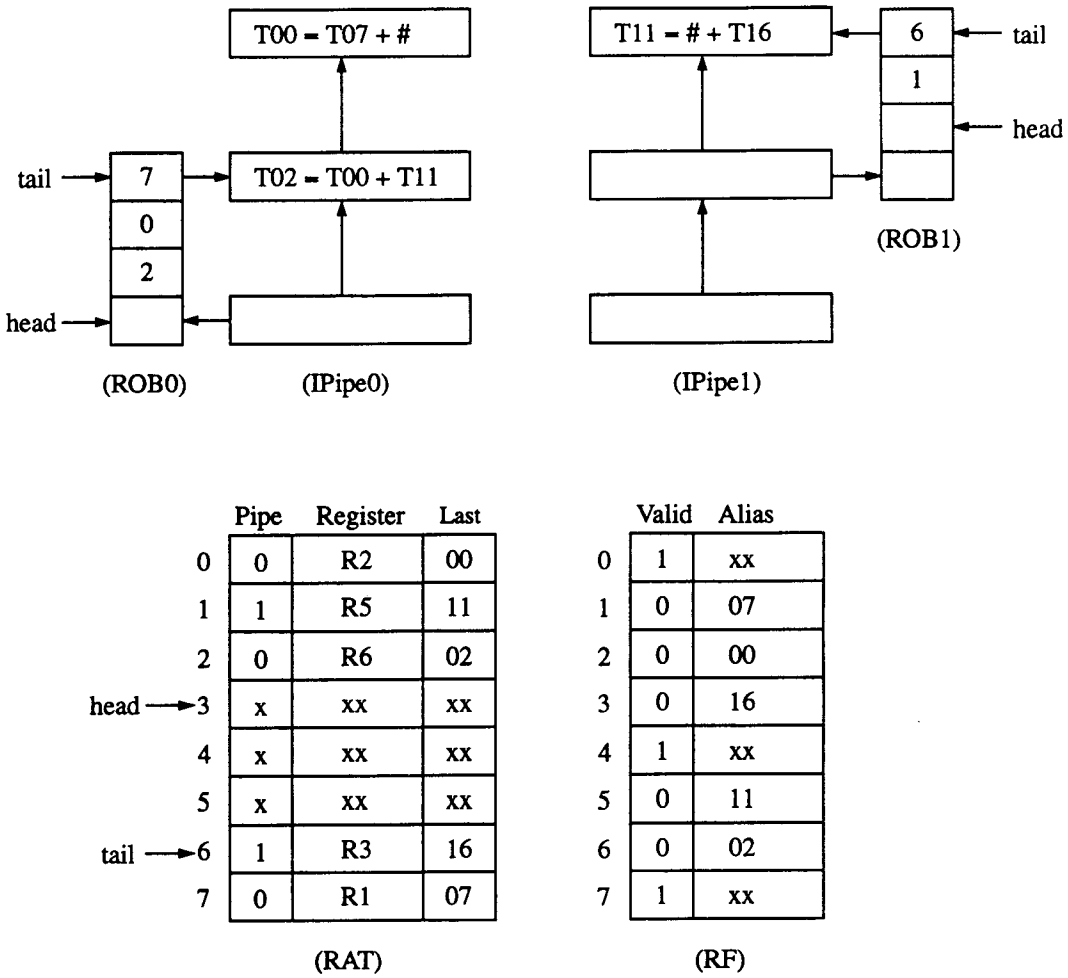

Figure 6.7 ROB Example (Incorrect Speculation Cleanup)

The tables are now back to the state they were in before the wrong branch and figure 6.7 shows the final state of the machine. As the remaining instructions complete, they are written back to the register file and removed from the RAT. The instructions must be retired in the order in which they were issued to guard against interrupts or faults. In this manner, the machine state can be saved so that it can be restarted if necessary after performing whatever operation is

required. This same mechanism is used to recover from incorrect branches, page faults, interrupts, and any other type of asynchronous event.

By segmenting the ROBs, the size of the individual ROBs have been reduced by an amount equal to the number of instruction pipelines. For example, a machine which originally had one 128 entry ROB with four instruction pipelines can now have four 32 entry ROBs. The segmented ROBs are still created with associative memory, but they are considerably smaller. When instructions are added to a ROB, they are added in order. At times when wrongly speculated instructions need to be removed, a start range and end range can be specified and since the entries are in order, they are easily found and removed.

The number of read and write ports can be reduced also. Since each instruction pipeline has its own dedicated ROB, the individual ROBs only need to have one write port for the IPipes regardless of how many IPipes there are. It is possible that all of the instructions in all of the instruction pipes need to read from one particular ROB. This is not likely however since in the case of there being four instruction pipelines the odds of an operand being in a given ROB are 25%. This probability decreases as the number of IPipes increases. Since it is known ahead of time whether or not the operand could possibly be in the ROB, there is no need to query any other ROBs. The worst case for when there are four instruction pipelines is eight read ports from the instruction side. The worst case for when there are four result pipelines is still four write ports, but again the probability for each result is only 25% and it is known which ROB needs to be written to. So, for the worst case, the ROB needs 5 write ports and 8 read ports

versus 8 write ports and 8 read ports for the non-segmented ROB. Assuming that the values are equally distributed amongst the four ROBs, the average number of reads per stage is 2. Since an instruction pipeline has an associated ROB, it always makes an access if there is an instruction in that pipestage. Correspondingly, the average number of writes is 1+(4*.25) = 2. If it is taken into account that not all of the stages are filled, that some of the operands have been read from the register file, that some of the instructions have been processed on previous times of having passed the ROB, and that not all instructions have two operands, the number of ports could possibly be lowered. The appropriate number of ports will depend on simulation runs for the type of benchmarks the architecture is being marketed to run. On top of all this, if on some cycles, there are not enough ports to perform all of the required actions the data can simply recycle around the pipeline and perform the necessary actions on the next pass of the ROB. This differs from the non-segmented case because in that case the ROB processed all instructions which entered the pipeline. In the segmented case, the RAT and RF can process the instructions as they enter the pipeline since they are non-associative structures. Then, if the need arises, the ROB can take extra time and force the instruction to make another revolution of the pipeline before doing the processing since the issuing of instructions isn't being stalled.

For the generic case, there are i instruction pipelines and r result pipelines where i and r are assumed to be binary multiples. The ROB can be segmented into i pieces. Each segmented ROB has the worst case number of ports as r+1 write ports and 2*i read ports. Assuming that the operands are distributed equally

across the ROBs, the probability that a given operand is in a given ROB is 1/i. Therefore, the number average number of reads for a given pipestage will be (2*i)/i = 2. The average number of writes for a given pipestage will be 1 + (r/i). Again, these numbers will be lower in practice since not all stages will be filled and not all instructions will have two operands.

6.2 Multithreading

In the counterdataflow chapter it has already been alluded to that CDF readily supports multithreading. Figure 6.8 shows a simplified version of a

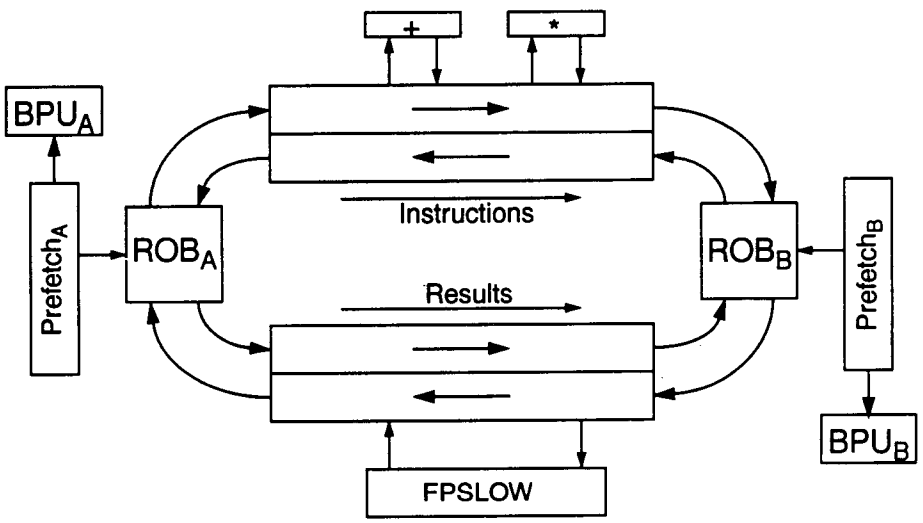

Figure 6.8 Multithreading with Counterdataflow

multithreaded CDF implementation which can handle two threads. The threads need to each have their own prefetch, branch prediction and ROB units. In addition, the branch execution units and memory units need to be thread aware or

have separate instances for each thread. They may have separate instruction caches or a unified cache, but that is left up to the specific implementation.

The instructions from the individual threads act just as they do in the non-multihreaded CDF pipeline. The difference occurring only in the matching logic. When an instruction gets a ROB entry, an extra bit is added to the tag that the instruction is given based on which thread it is from. For the case where there are two threads, it can be assumed that thread A gets a 0 and thread B gets a 1. Now, instructions from both threads can be in the pipeline at the same time and the standard tag matching logic will take care of matching tags. Since the instructions from different threads are defined as having different tags they will never match.

There are several advantages to using multithreading. Some execution units are area expensive and yet are not used very often [18]. With multithreading, instructions from both threads can share these execution units. This lowers the overall cost of having the unit while increasing the amount of time the unit gets used because both threads will use the same execution unit. Another advantage is that the same instruction and result matching logic can be shared by both threads, giving an effectively larger reservation station without doubling the number of stages. If one thread is not making forward progress for some reason, the other thread can use more of the resources and keep the overall throughput high [9]. This throttling effect can be used when one thread needs to be replaced due to a page fault or other fault. While the one thread is being flushed and replaced, the other thread can use all of the available resources

thereby somewhat offsetting the performance lost from the other thread. Multithreading in CDF is inherently scalable. By adding $\log_2 n$ bits, where n is the number of threads, to the tag a large number of threads can be supported. Of course, a linear number of ROBs, prefetch, and branch prediction units need to be added, so the hardware needed does still increase substantially.

6.3 Data Speculation

The idea behind data speculation is that now that instructions are being speculatively executed the next logic step is to speculatively execute instructions based on guesses of what the data values will be. If some sort of an educated guess can be made, the thinking goes, it is better to guess and hopefully perform useful work than to do nothing and definitely not perform useful work.

Counterdataflow gracefully supports data speculation. Currently, each result and instruction can be in one of two different states, either valid or invalid. For data speculation, that will need to be changed to valid, invalid, and speculated. With speculated data, an instruction can launch to an execution unit and produce a speculated result. In the normal case, once an instruction has been launched to an execution unit, it is removed from the pipeline. This will need to be changed in the case where speculation is being performed. The instruction will need to remain in the instruction pipeline. While the instruction is circulating, it is inspecting the result pipeline just as it usually does. In this case, however, it is watching for its operands to pass. If the operands pass and have the same value as was speculated, the same result is dispatched down the result pipeline, only this time not marked speculated but simply valid. The instruction is now free to be

removed from the pipeline. If on the other hand, the speculation was incorrect, the instruction will take the correct value and when an execution unit is available will launch eventually creating the real result. In this way, speculated results can be created and used by subsequent instructions while maintaining correct operation. The ROB will never allow a speculated result to be retired to permanent storage [18]. Either a new valid result will be sent, or a confirmation that the speculated result is indeed the correct result will be sent.

With all of these speculated results and instructions, some sort of control must be implemented or the pipelines will be flooded with only speculated values and no real work will be done. The first step is to implement a priority to decide which instructions get access to an execution unit. Obviously, if two instructions want to execute and one has real values while the other has speculated values, the real valued instruction should get priority. Second, speculation will need to be intelligently applied, only guessing when there is a reasonable probability of being correct or when nothing else would be executing anyway. Again, it is better to do something and hopefully accomplish some work than to do nothing and definitely accomplish nothing.

6.4 Chapter Summary

Counterdataflow has all of the capabilities of a modern microarchitecture. It is capable of high performance, scalable multithreading, and data speculation all without exponentially increasing the amount of hardware necessary. Additionally, the one potential bottleneck of CDF, the ROB, has had a solution presented which not only reduces the number of ports needed, but increases the

possible size of the ROB while reducing the complexity and increasing the speed of access. More simulations are needed to quantitatively show that CDF is indeed this scalable.

7. CONCLUSION

7.1 From CFPP to CDF

The evolution of the counterflow architecture from the original counterflow pipeline processor to the virtual register processor and finally ending with the counterdataflow processor has now been covered. Each new architecture has overcome some of the limitations of the previous one and in some cases introduced some bottlenecks of its own. There are definitely limitations to our simulations' accuracy, but the limitations have been consistently applied to all of the architectures. Therefore, it is now possible for to make a direct comparison between the three architectures to see just how much the changes made have affected the performance.

7.2 Execution Unit Usage

How much of the time a processor's execution units are busy can be used as a measure for how efficiently an architecture is using it's resources. Figure 7.1 shows how much of the time the various execution units were in use for the three counterflow architectures for the SpecInt95 traces. The first item to note is that this metric is just the percent of the time the traces took to execute that the execution units were busy. Since CDF takes less time per trace than VRP, and VRP takes less time per trace than CFPP, this metric doesn't take into account that each trace actually takes less time to execute on CDF and VRP. Considering that CDF takes less than half the time to execute the same number of instructions and is busy more of the time illustrates just how far the counterflow architecture

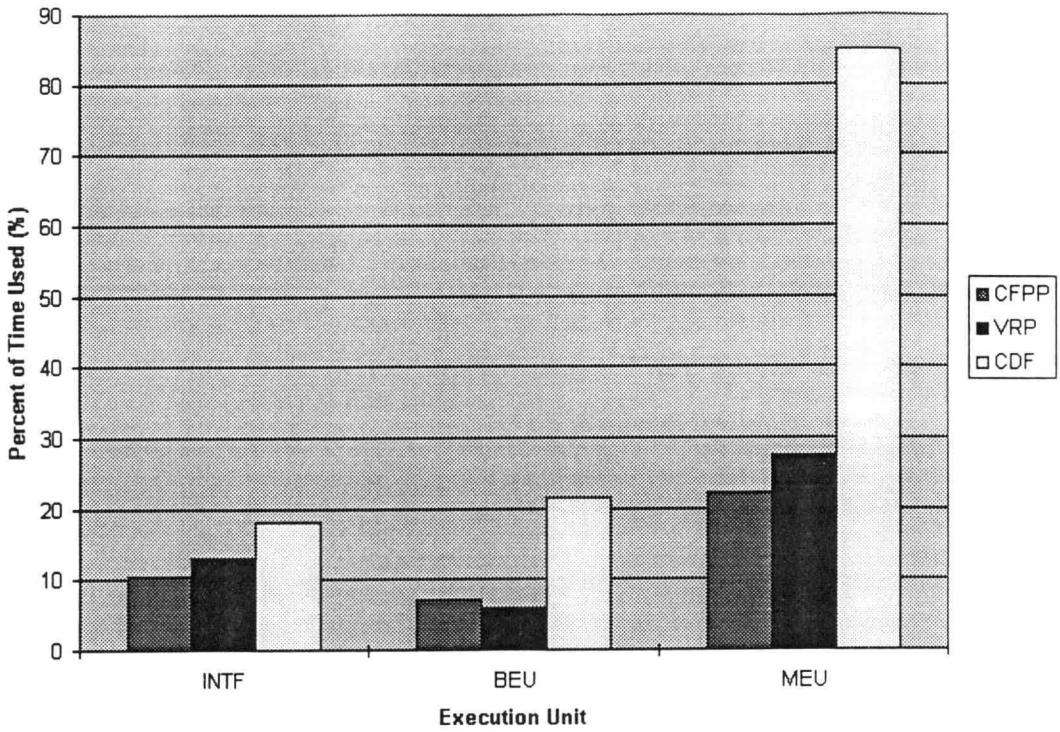

Figure 7.1 Execution Unit Usage for Integer Traces

has improved. Since these are the results for integer benchmarks, the floating point units were hardly ever used and have therefore been left off of figure 7.1 for clarity. Similarly, the compiler used to generate the traces gave heavy penalties to the use of slow integer instructions, therefore there are almost no slow integer instructions present in any of the traces. Each of the architectures has one slow integer unit, one fast floating point unit and one slow floating point unit. All of these units were in use for less than 1% of the time for the integer traces.

The values represented in the graphs are the average amount of time the execution units were in use during a simulation run. In the cases where there was

more than one execution unit of a given type, the usage values were averaged. In this case, all architectures had multiple fast integer (INTF) and branch execution units (BEU). CFPP and VRP had three fast integer units while CDF had four. Even with an extra unit, CDF managed to have an average usage of 18% compared to 13% and 10.6% for VRP and CFPP respectively. For the branch execution unit, CFPP and CDF both had two units while VRP had three. Considering that VRP's average was lowered by the extra unit, CFPP's usage of 7.2% and VRP's usage of 6% are fairly similar. CDF's average of 22% illustrates the amount of speculation that the CDF architecture was performing. Each of the architectures had only one memory execution unit. Considering that the latencies involved with load and store operations can be considerably higher than other instructions, it is not all that unexpected that the MEU was busy more often than the other units. What was unexpected was just how often CDF managed to keep this unit busy. CDF kept the MEU busy 85% of the time compared to 22% and 27% for CFPP and VRP. This can be attributed to the fact that in CDF, instructions never have to stall. In CFPP and VRP, once a load or store instruction stalls, it tends to backup the instruction pipeline preventing other instructions from being launched. So, in CDF, many memory instructions could be in the MEU at the same time, thereby increasing the amount of time the MEU was kept busy.

Figure 7.2 shows the execution unit usage for the SpecFP95 traces. The reasoning for the differing values of the fast integer and branch execution unit are essentially the same as for the integer traces. The usage of the memory

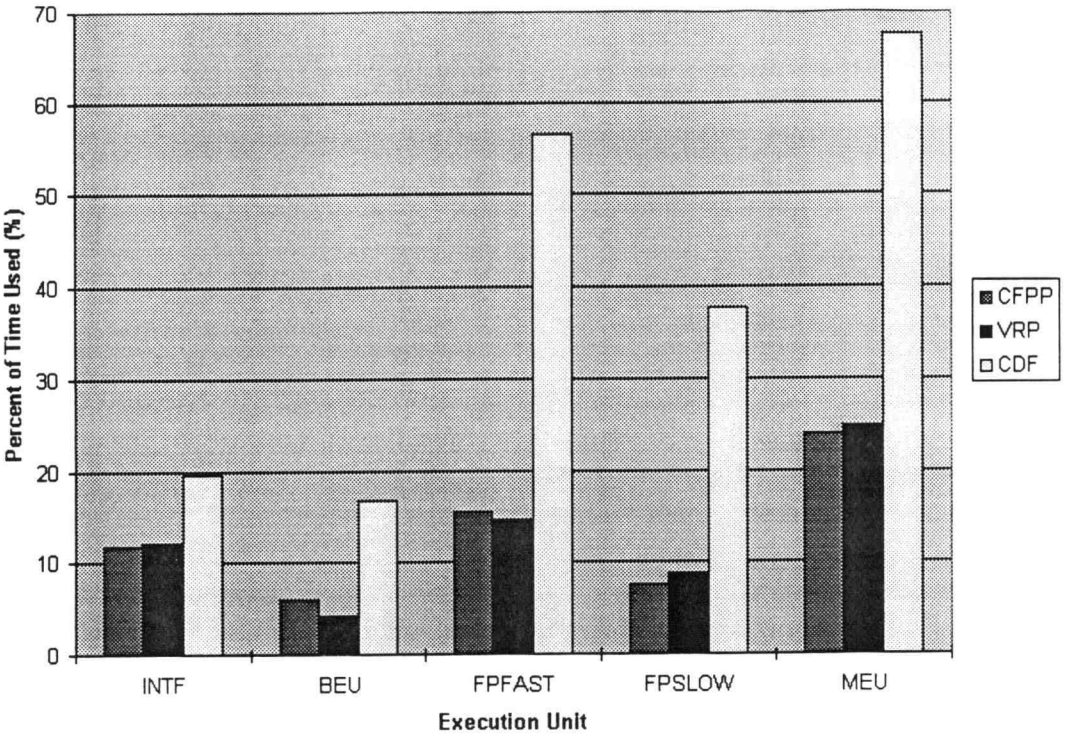

Figure 7.2 Execution Unit Usage for Floating Point Traces

execution unit is slightly lower for CDF on the floating point traces. This can most likely be attributed to the fact that integer operations are slightly more memory intensive than floating point operations. All architectures have only one fast floating point unit and one slow floating point unit. CDF really shows it's ability here, keeping the fast unit busy 57% of the time and the slow unit busy 38% of the time. CFPP and VRP are considerably less efficient at keeping their units busy being at best a third of the amount of CDF. This is directly related to the fact that their instruction pipeline's stall. Any instructions that depend on the execution of a long-latency operation like floating point will wind up stalling the instruction

pipeline waiting for the result. CDF gracefully handles longer, or unknown, latency operations much better than its predecessors. Because CDF's instruction pipeline does not stall, unrelated instructions can continue in the pipeline and do useful work keeping the execution units busy.

7.3 Effective Instruction Window

The effective instruction window, as measured here, is the average number of instructions which are in the machine at any given time. For CDF and VRP, this is calculated from the reorder buffer (ROB). Since the ROB holds all instructions from the time they enter the instruction pipeline until the time they finish executing and are retired, the average number of instructions in the ROB is the average size of the instruction window. This number also includes incorrectly speculated instructions, which take up space in the ROB even though they have actually accomplished no real work. Since CFPP has no ROB, the effective instruction window is estimated as the distance from the instruction entering the instruction pipeline to the last branch execution unit. This is actually the best case value, but will suffice for a rough comparison. Figure 7.3 shows the effective instruction window for the three architectures. The instruction window is important because it has the opportunity to expose as much available parallelism in the code as possible. CDF, with its deep speculation, and ability to remove instructions once they have begun to execute has an effective instruction window almost ten times the size of CFPP or VRP.

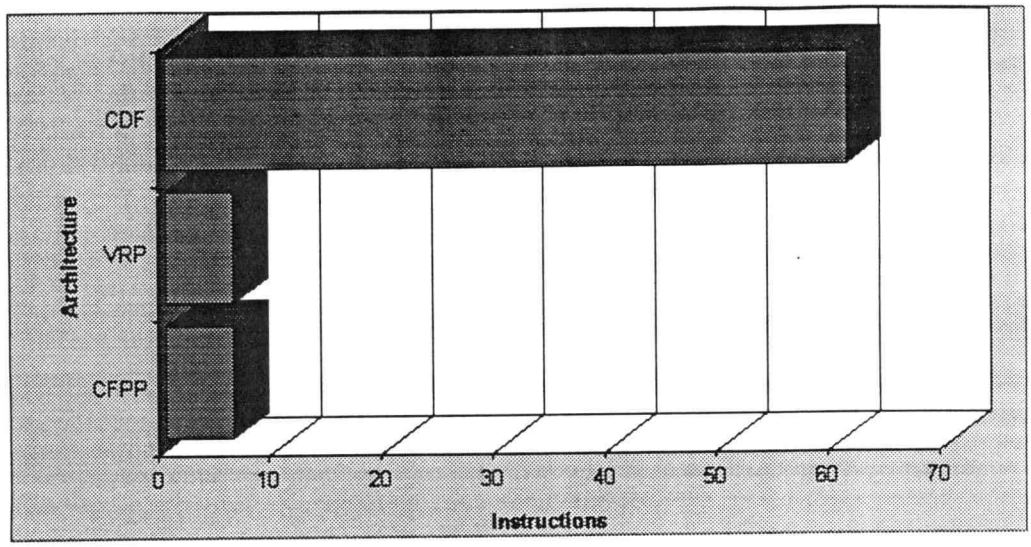

Figure 7.3 Effective Instruction Window

7.4 Instructions Per Clock Cycle

The average instructions executed per clock cycle are summarized in figure 7.4 for the three counterflow architectures. On average performance in instructions per clock cycle, VRP shows a 12% improvement over CFPP. CDF has a 133% improvement over CFPP and a 161% improvement over VRP. It may be obvious to say but CDF has drastically improved the available performance over its predecessors.

7.5 Summary

Several years ago, the counterflow pipeline process was developed by researchers at Sun Microsystems to demonstrate the concept of asynchronous

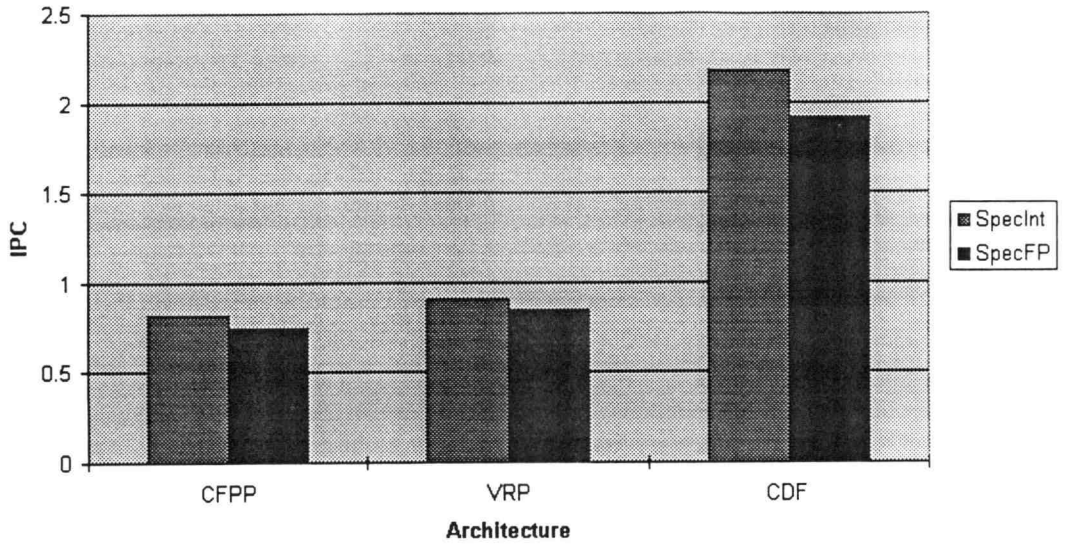

Figure 7.4 Average Instructions Per Clock Cycle by Architecture

circuits [1]. This architecture uses local control and clocking to allow distributed decision making. Unfortunately, this first design suffered from several severe limitations. The register file was at the opposite end of the pipeline from the issuing unit resulting in the pipeline being used inefficiently. While the basic architectural premise was to use local control, there were several global signals which ultimately limited the minimum clock cycle time. As a result, a new architecture was developed, the virtual register processor, which overcame the problem of the register file being far away from the issuing unit. This improved the performance somewhat, but the global signals remained as did the fact that this architecture still could not issue more than one instruction per clock cycle. Since other commercially available processors were already capable of issuing more than one instruction per clock cycle, the virtual register processor was not

scalable enough to be used as a general purpose processor. In order to get rid of the global signals, the pipelines would have to not be required to stall. Eventually the idea was struck upon that if the instruction and result pipelines were to wrap around and become circular queues the instructions would never have to stall. This became the counterdataflow processor or CDF. Now that instructions can wrap around the pipeline, it is possible to issue more than one instruction per clock cycle since instructions in the same pipestage can be dependent upon each other. CDF appears to be scalable and efficiently lends itself to advanced microarchitectural techniques such as multithreading and data speculation. Further investigations are required to see just how much performance can be extracted from the counterflow architecture, but currently the outlook is promising.

BIBLIOGRAPHY

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 and S. Lu, "Synchronous Implementation of a Counterflow Pipeline Processor" Proceeedings of the 1996 International Symposium on Circuits and Systems, May 1996.

3. K.J. Janik and S. Lu and M.F. Miller, "Advances to the Counterflow Pipeline Microarchitecture" High-Performance Computer Architecture - 3, February 1997.

4. J.P. Hayes. Computer Architecture and Organization. New York, NY: McGraw-Hill, 1988.

5. J.L. Hennessy and D.A. Patterson. Computer Architecture: A Quantitative Approach. San Mateo, CA: Morgan Kaufman, 1990.

6. M.D. Jones, "CFPP: A New Approach to Pipelined Microprocessors", Dec. 1994. http://lal.cs.byu.edu/people/jones/latex/sproull.html/sproull.html.html.

7. 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.

8. D. Bhandarkar and J. 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 and S.J. Eggers and J.S. Emer and H.M. Levy and 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. D.C. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin Computer Science Technical Report #1342, June 1997.

11. R. Carlson and M.F. Miller, "VRP Simulator", April 1996. http://www.ece.orst.edu/~sllu/cfpp/vrpsim/docs/vrpsim.html.

12. T.M. Austin, "Simplescalar Tools Hacker's Guide," talk given to the Electrical and Computer Engineering Department, Oregon State University, 1996.

13. M.F. Miller, "CounterDataFlow Architecture: Design and Performance," M.S. Dissertation, Oregon State University, August 1997.

14. M. Miller and K.J. Janik and S.L. Lu, "Non-Stalling Counterflow Architecture," to be presented at the 4th Annual Conference on High Performance Computer Architecture, Las Vegas, Nevada, February 1998.

15. G. Cornell and C.S. Horstmann, Core Java, Mountain View, CA: SunSoft Press, 1996.

16. J.E. Smith and G.S. Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, December 1995.

17. D.M. Geary and A.L. McClellan, Graphic Java Mastering the AWT, Mountain View, CA: SunSoft Press, 1997.

18. M.J. Flynn, Computer Architecture Pipelined and Parallel Processor Design, Boston, MA: Jones and Bartlett Publishers International, 1995.

19. J.M. Rabaey, Digital Integrated Circuits A Design Perspective, Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

20. Standard Performance Evaluation Corporation, "SPEC Describes SPEC95 Products And Benchmarks," SPEC Newsletter, September, 1995.