AN ABSTRACT OF THE THESIS OF

Chung-lun Chan for the degree of Master of Science in Electrical and Computer Engineering presented on June 9, 2000. Title: Modeling Microarchitecture Simulator Using Object-Oriented Approach.

Redacted for privacy

Abstract approved: _____

Shih-Lien Lu

With the success of the CounterDataFlow Pipeline microarchitecture developed by Oregon State University, there is increasing demand for a highly flexible high-level simulator modeling tool to support the further expansions and studies of the Counterflow pipeline processors family. This work examines the implementation of a Java-based execution-driven simulator modeling tool, bBlocks, which gains flexibility by identifying the independent parts in a micro system and partitioning them into reusable blocks. Two simulators have been constructed to demonstrate the possibility of bBlocks.

Copyright by Chung-lun Chan
June 9, 2000
All Rights Reserved

Modeling Microarchitecture Simulator Using Object-Oriented Approach

By

Chung-lun Chan

A Thesis submitted

To

Oregon State University

In partial fulfillment of
The requirements for the
degree of

Master of Science

Presented June 9, 2000
Commencement June 2001

Master of Science thesis of Chung-lun Chan presented on June 9, 2000.

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

Chung-lun Chan, Author

ACKNOWLEDGEMENT

The research presented in this thesis would never have been possible without the tremendous effort of my teammate, Hua Ying. Special thanks to my major professor, Dr. Shih-Lien Lu, for guidance and assistance.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

LIST OF TABLES

Modeling Microarchitecture Simulator Using Object-Oriented Approach

CHAPTER 1. INTRODUCTION

The complexity of modern processors is growing exponentially causing architecture validation to become increasingly difficult. While simulation provides one effective way to verify correctness of a design at the high level, the growth of high-level simulators has not kept up with the development of new processors.

The Architecture-Blocks simulation package, or aBlocks, was a modeling tool for microarchitecture simulation developed three years ago by Oregon State University to address this demand [2-4]. At that time, there were no modeling tools for high performance computer architecture simulation available which provided enough flexibility to describe a variety of architectures or the capability to prototype microarchitectures rapidly.

A good simulator usually optimizes one or two of the following three categories: details, performance, and flexibility. aBlocks optimized toward flexibility. The primary objective for the aBlocks project was to develop a tool that allows rapid microarchitecture prototyping and performance evaluation.

While aBlocks is relatively flexible and easy to use, its structural shortcomings and trace-driven simulation nature limit its usability. This work inherits and expands the design goal from aBlocks. Lessons learned from aBlocks provided the foundation for the next generation simulator modeling tool, bBlocks.

1.1 MOTIVATION

The initial motivation for this simulator modeling tool project was to support research on the CounterDataFlow pipeline processor architecture (CDF). CDF is a cutting-edge microarchitecture design based on the counterflow pipeline concept originated by Sproull et. al. [1] and expanded by Oregon State University [2-4, 7]. As a result of the uniqueness of this architecture, it requires a simulator with high flexibility to achieve fast microarchitecture prototyping and performance evaluation, which was not available in the market. For this reason, aBlocks was developed.

With the success of CDF, a variety of research opportunities have opened, ranging from performance enhancements to feature add-ons. However, due to the lack of details from trace-driven simulation and its structure, the work required for further extension is expensive. After developing the counterflow pipeline architectures, the CDF research team was lacking a model of industry standard computer architecture (such as basic SuperScalar [6]) to serve as a benchmark for its results. To fill this need, it was decided to develop a second generation of aBlocks with the first aim targeted to SuperScalar architecture.

1.2 RELATED WORKS

This section describes prior works in microarchitecture simulator modeling. Three projects, including aBlocks, SimpleScalar [5], and Ptolemy [10], were studied.

1.2.1 aBlocks

The aBlocks simulation package was the ancestor to this work, with a goal to support the product lines of counterflow pipeline processors. It is a trace-driven simulator that runs the industry standard SimpleScalar simulator's program traces. As noted from the name, it is a block-based simulator where the components of microarchitecture are grouped into functional blocks and implemented as objects in the software. aBlocks provided Java's object-oriented advantages and adopted an environment for rapid simulator development. It's graphical support is also beneficial for debugging and prototyping various architectures. It is a cycle-timer simulator that tracks microarchitecture state for each cycle. A simulation cycle is initiated by a function call give(). This function call at the beginning of every simulation cycle initiates the execution chain by propagating this function call to other blocks. This call will not return until the return from the last blocks in the chain.

The advantages of aBlocks are:

- Objected oriented structure. The strength of abstraction, encapsulation, inheritance, polymorphism rewarded by object-oriented methodology allow high code reusability and in turn, gives the simulator high flexibility.
- GUI support. This helps significantly in debugging and prototyping a new microarchitecture (especially for the counterflow pipeline-based processors), because it is easier to visualize the flow of data as well as to check for correctness.
- Extremely portable Java-based executable. It provided the platform independent capability that allows our simulator to run on virtually any platform without modifying the source code or recompiling it.
- Trace-driven. It is simpler to implement in comparison to other simulation methods like execution-driven, which reduces the development time for new design. In addition, the performance of simulations can be better because it has fewer details.
- Simple software structure. The chain-linked give() function call standardizes the communication interface between blocks. This simple protocol allows the designers not to worry about timing problems that arise from data transactions.

The disadvantages of aBlocks include:

- Not enough details to demonstrate the correctness of a prototyped architecture. A trace driven simulator does not perform actual

computation. It is good at hiding minor problems that can easily be overlooked. In fact, we found a couple problems in the design that were hidden for this reason. It gives the designer less confidence that their design is functionally correct.

- Linear software structure. Although it adopted the look of object-oriented design from Java, it's more of a structural program. Note that it uses a give() method to communicate with adjacent blocks, and the adjacent blocks invoke their give() method to talk to other blocks. It is in fact applying structural methodology instead of object-oriented methodology. For this reason, it loses code reusability because every block tightly depends on the adjacent blocks. In another words, it gives away flexibility.

1.2.2 SimpleScalar

SimpleScalar is a SuperScalar architecture simulator developed by University of Wisconsin-Madison. It's currently the most widely used SuperScalar simulator in the academic area and the leading product in the market. It is a very sophisticated simulator that runs primarily in the UNIX environment. It has its own Instruction Set Architecture (ISA) derived from MIPS ISA, and it provides a lot of tools for simulations including a compiler for FORTRAN and C code, an assembler, a loader, and a debugger.

The advantages of SimpleScalar include:

- Comprehensive features. Having its own ISA and compiler, SimpleScalar has the ability to simulate both the software and hardware environment with great detail. It benefits researchers from research in software enhancement to hardware enhancement to improve performance. It benefits microarchitecture research from compiler enhancement to hardware add-on to study and improve performance of SuperScalar design.
- Simulate with varied details. SimpleScalar optimizes performance and flexibility, and in addition, it provides portability and simulators with different levels of detail, from one optimized in performance to one optimized in details.

The disadvantages of SimpleScalar are:

- Pipeline stage bases. It makes SimpleScalar incapable of describing new architectures that do not follow the traditional pipeline stages, e.g. CDF.
- Provides only source code portability, but the binary is still not portable (i.e. requires recompilation of source).
- Lack of flexibility in terms of architectural changes. When SimpleScalar works with a new architecture, the designer will need to "hack" the source code. It does not provide a way to "build" a simulator.

1.2.3 Ptolemy

Ptolemy is a heterogeneous concurrent modeling tool developed for embedded systems by University of California, Berkeley. It is a general-purpose modeling tool to describe systems in different domains, including continuous time, discrete-events, finite-state machines, and synchronous dataflow. It defines a modeling system and constructs an environment on which varies system architects can build their design. It also provides GUI, XML support, and some useful tools including plotters.

The advantages of Ptolemy are:

- Extremely flexible: The second generation of Ptolemy is written in Java. The component based Ptolemy applied OO techniques when possible, which empowered it to be as flexible as possible. One can even construct models through simple web-based scripting.
- Best at describing concurrent system. Its focus is on handling concurrency and time. Everything is built around that goal. Therefore it has the ability to describe a great variety of concurrent systems.
- Sophisticated infrastructure. They defined their own typing system for Ptolemy models. While it may seem unnecessary at first glance, it pioneered the possibility to effectively connect interacting components, which are heterogeneous by nature, together.

The disadvantage of Ptolemy is the great overhead from handling concurrency. It is especially critical to systems like microarchitecture simulation, when performance is important.

CHAPTER 2. BBLOCKS PRELIMINARY

2.1 GOALS

> "Why do I feel that flexibility is generally the most important quality you can give to your designs and code? The reason is that, as one of my managers used to put it, "software is a living product". Code isn't static. It is constantly being tweaked, enhanced, fixed, and so on, by a team of programmers, a team that is usually in constant flux itself" [8].

The primary goal for bBlocks, inherited from the ancestor aBlocks, is to assist microarchitecture researchers with rapid prototyping and performance evaluation by providing a simulator modeling tool with maximum flexibility. It should be able to comfortably accommodate any architectural changes or add-ons and should allow researchers to build simulators according to their design in the shortest amount of time while maintaining a satisfactory level of simulation details. While overall speed is also important to a simulator, it is of less importance to this project in comparison with flexibility and detail. In order to achieve rapid prototyping, it should also have graphical support to ease in the process of debugging. It's especially helpful for the product line of counterflow processors since counter-flowing data and instructions and on-the-fly matching are not trivial to trace without visual aids.

To achieve rapid prototyping, code reusability is highly important. The ideal scenario is that designers do not need to write code to describe new designs or features. While this may seem difficult to achieve, it is made easier by the fact that

most of the research has a tendency to modify or add to an existing design. By carefully partitioning a design into multiple components, it is possible to keep the majority of the components untouched, where changes only happen to a very small portion of the overall simulator. Two problems remain: how to identify what needs to be updated and how to add or extend features without affecting the existing functionality? These problems can easily be answered by applying object-oriented techniques, which promotes abstraction, encapsulation and information hiding.

The goal for this project is to develop a modeling tool with flexibility to develop new microarchitectures that SimpleScalar doesn't offer, detail that aBlocks lacks, and low overhead and block reusability that Ptolemy does not emphasize.

2.2 DESIGN PROBLEMS

In reality, a microarchitecture usually contains a very large amount of digital logic. To ease the complexity in development, we partition a large design into small entities. A typical entity would have a set of inputs to receive data, a body to perform functions, and a set of outputs to pass the result to other entities.

This concept of an entity is the same as in software. Entities are not limited to a fixed number of input/output ports, nor restricted to a certain data type. For instance, entity A might have output(int, int, float, boolean, double), while entity B might have input(double, long, float, char). For a modeling tool, it is fundamental

to identify commonalities of entities and encapsulate them into a class, so that only the non-reducible, design-specific porting of the code is left to the designer.

aBlocks works around the heterogeneous nature of functional blocks by generalizing every block to have the same I/O interface and same data type. The other heterogeneous system modeling tool, Ptolemy, has a better solution to this problem by developing a data type system. By taking advantage of data polymorphism offered by an object-oriented approach, it defined a token to be the common type for data. A tree of data types is then derived from the object token. The actual attribute of a token depends on what it is created for. bBlocks noticed the loosely coupled nature of the entity. No attempt was made to generalize the data type and solve the type problem caused by the generalization. Instead, the problem was solved in a different way.

2.3 EVOLUTION BEYOND ABLOCKS

As mentioned in the previous chapter, aBlocks' blocks rely on the give() method to communicate with other blocks. This method for bi-directional communication can be interpreted as "I am giving something to you", or "please give something to me". It generalizes all blocks to have the same input/output interface. The cycle-based aBlocks performs a call every simulation cycle to the give() method of the first block in the calling tree to initiate the chain reaction. The give() method of the first block will make a call to the give() method of the

adjacent block during its linear execution. Similarly, within the give() method of the adjacent block, it will make a call to its adjacent block. This motion continues until a block reaches a steady condition (i.e. doesn't make call to a give() method of other blocks). It can also be viewed as a hybrid recursive call, with the difference that the method is not calling itself, but rather the method with same declaration from other blocks. There are two major points that the user of aBlocks should be aware of. First, there must be a block in the calling tree that serves as terminator, otherwise a give() method will keep looping and cause a deadlock. Second, calling order is essential. While the declaration of give() methods are the same across the simulator, their requirement and functionality is not. When a call is made is as important as passing the correct information to a block. aBlocks give() method passes information by using an data object called aToken, which is a container (or data structure) of information. The meaning of the data stored varies based upon which block it is locating and when it is received.

While it may not be obvious, aBlocks structure does not take much advantage from object-oriented methodology. First of all, it defined that every block has the same input/output interface (i.e. every block uses the give() method to communicate), and the type of information is generalized as aToken. Blocks are heterogeneous. To implement heterogeneous blocks with a homogeneous structure, aBlocks defines the generalized aToken be used for all data types. In another words, the meaning of aToken varies, from block to block, depending on where it is and when it is being received. This implementation placed a constraint to the

aBlocks structure that the blocks give() method has to be connected and called in a certain order to allow the blocks to receive the correct aToken. Thus, every block is depending on other blocks, in the sense that it makes assumptions about who is connecting it in what order.

To illustrate this problem, consider the amount of work that must be done to add a new block to an existing design. Ideally the new block would be implemented and add it to the current design, without modifying or affecting the existing design. In the aBlocks system, the first step is to examine the block in the existing design that the new block is going to attach to. Since blocks are source and call-order dependent, adding a block very likely requires modifying the existing design. The modification in the current design may affect the blocks adjacent to the modified block, because the call-order is changed and that could affect the overall functionality, requiring examination or modification of other blocks. In another words, adding a block to the system affects the existing system, which reduces the level of flexibility and increases the time for prototyping.

2.4 SYNCHRONOUS LOGIC SUPPORT

bBlocks works the best with synchronous logic, i.e. a block where most of the activities happen at the beginning of the machine cycle. For asynchronous logic that does not depend on the clock, a cycle-based simulator will need to allow data to iterate and oscillate until it reaches a steady state. However, this task is

extremely time consuming, as signals can oscillate for many simulation cycles before stabilizing. Since it would badly hurt the simulation performance, asynchronous logic is not supported in our current release.

CHAPTER 3. SOFTWARE ARCHITECTURE OF BBLOCKS

bBlocks is a system that takes the component view of design, where interactive blocks are defined and built. It governs the interaction and execution of blocks under the system.

The bBlocks design was borrowed from JavaBean, where an event-listener model is used for loosely coupled bean communications. The JavaBean system has a very similar nature as our computer architecture block: both beans and blocks are loosely coupled. However, unlike GUI, computer architecture has extremely busy traffic, which makes the event-listener model too expensive to use. To fix this problem, the interface to bBlocks was designed differently than JavaBean. An interface based on the concept of the jigsaw puzzle is developed.

3.1 "JIGSAW PUZZLE" INTERFACE

This "jigsaw puzzle" structure is based on a couple observations. First, recall that a block has a set of inputs and a set of outputs. While the block's computation depends on its input and the internal state, output format is not a requirement for a block to process. The output is served solely for the other blocks that it connects to. Second, a block does not depend on a particular block, but it only needs a certain data type at its inputs to allow computations. In this sense, a block is a completely independent unit. It is a very important key to lead to better flexibility,

because as long as a block is independent of others, it can be added or removed easily without affecting the system.

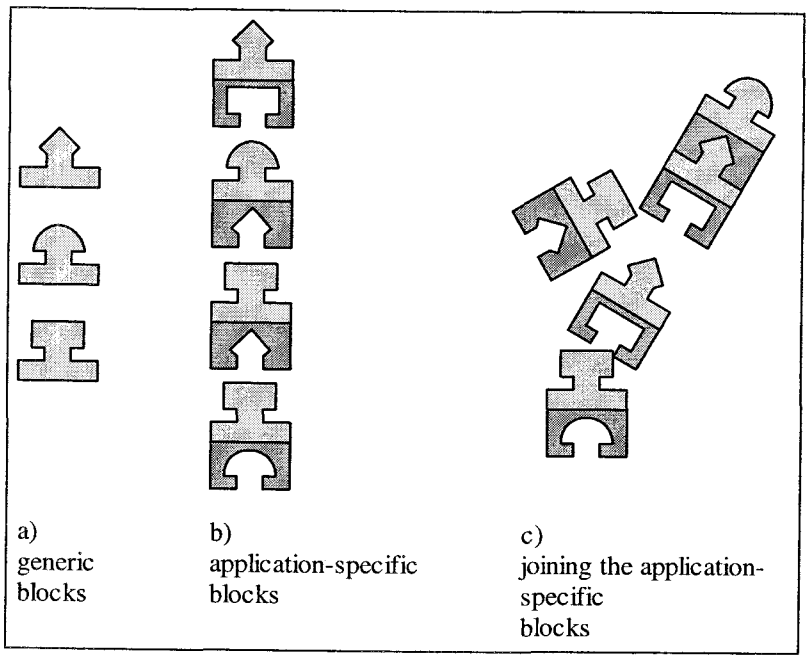

Figure 3.1 bBlocks "jigsaw puzzle" structure

Based on these observations of the blocks, bBlocks allows a block to define who can be connected to its input. Only those whom meet all the criteria set by the by the block can be the provider of inputs. A block that defines inputs and performs computations is called a generic block, as shown in Figure 3.1a. When a block implements the input criteria defined by another block, it becomes an application-specific block, as shown in Figure 3.1b. Application-specific blocks are put together to construct a simulator like jigsaw puzzle pieces fit together to make a picture (Figure 3.1c).

3.2 WHY JAVA

To achieve the flexibility that other structured based simulators do not offer requires an object-oriented based simulator. Considering flexibility of the language, graphical support, the bonus of cross-platform binary independence that comes without extra efforts, and the choice of the ancestor aBlocks, Java was an obvious choice for the second generation of simulator modeling tool.

3.3 OVERALL STRUCTURE

bBlocks comprises four kinds of classes: Block, Provider, Simulator, and data. With these four basic kind of classes, you can build simulators for any microarchitecture. Block is a class that describes an entity in the microarchitecture. It is responsible for performing computations. Provider is an interface used by blocks to define the communication topology between the outputs of one block to the inputs of another block. Simulator is a class that defines a simulation system. By joining blocks in a certain manner, a simulator class describes the behavior of a microarchitecture.

3.3.1 Simulator

The simulator class defines a simulation system. It is also a system manager that disciplines and monitors the blocks registered to it, and governs the interactions between blocks and executions.

For a cycle-based simulation for a synchronous system, data should be moved from the source blocks to the destination blocks at the beginning of the cycle, all at once. While it's trivial to achieve in hardware, software does not work the same. A software block cannot update internal status while it is still being retrieved from the other blocks. This causes a race condition.

bBlocks simulates the concurrency by dividing the action of retrieving input and perform computation on input into two different stages. In other words, a simulation cycle has two stages. We called the stage where a block retrieves inputs and lets other blocks acquire outputs "pre-tick", and called the stage where the actual computation take place "tick".

Every machine cycle, the simulator will go around and execute twice the blocks that are registered to it. The first simulation cycle is to invoke the preTick() method of all blocks to initiate them to collect their inputs. The second simulation cycle is to invoke the tick() method, which is used to instruct the blocks to execute with the input.

Simulator is an abstract class in bBlocks. To define a simulation system, a user should create a new class, have it extend or inherit from the simulator class,

and direct it to register the necessary block and connect them together according to the microarchitecture.

Method	Description
setup()	It defines blocks and how they are connected. A concrete simulator class (e.g. SuperScalar) should implement this method inherit from the abstract Simulator class in order to complete the functionality. Code to instantiate, register, and connect blocks should be put in this method.
Main()	To make the java program executable. The designer should invoke run() method to start a simulation.

Table 3.1 Key methods of Simulator class

3.3.2 Provider

Provider is simply an interface that a block defined to guarantee that whomever connects to its input agrees to provide the necessary input data. It can be viewed as a contract – a promise that the acceptor must have implemented certain methods, where methods in this case are the means to feed inputs. This interface structure is the key to meeting the primary goal of maximizing the reusability of code. With a provider, input to a block is no longer limited to a specific block. Instead, any blocks that meet the requirement of a particular provider can be used to feed inputs of that block.

A provider is not restricted to a certain data type or data object. It can be of any data type or object.

3.3.3 Block

A block in bBlocks is defined as a synchronous computation module that performs a core computation that is a function of only its inputs and the current state.

Due to the race condition problem mentioned earlier, a block should not update the internal state when getting input or being queried for output. Therefore there should always be an input buffer and output buffer for each input and output interface.

In the "jigsaw puzzle" structure, a block has two forms: a generic block, and an application-specific block. A generic block is a block that carries out the computation, while an application-specific block is responsible to take care of communications.

3.3.3.1 Generic block

A generic block is an abstract class. It is simply called "Block" in the software. Since the computations of a block only depend on its input, a generic block defines inputs only, but no outputs. At this level, blocks are completely independent from other blocks, in the sense that it does not know any other blocks. The only knowledge it has about the block connecting to its input is that block has the ability to provide the necessary data that it needs. This confidence is obtained from an agreement of input, which we called "Provider". This agreement is

basically a contract from other blocks, saying they must provide the necessary input in the proper format. In other words, only the blocks that are providers of a certain data can connect to that particular block. Because a generic block is independent from others, this object is highly reusable.

As mentioned in the earlier sections, due to race conditions, there are two phases in the execution of a block. The first phase is so called "pre-tick". It is when a block collects inputs from other blocks. When a block is ready to receive an input, it invokes the provider-defined methods of the connected blocks, requesting the connected blocks to feed the necessary data, if they have any ready. The received input is then stored into an input buffer, waiting to be processed in the next phase, "tick".

The second phase, tick, is the body of a block. It processes the inputs in the input buffer stored in the pre-tick phase and performs computations. Outputs are stored in output buffers, waiting for the other blocks to collect them.

Methods	Description
connectTo()	It defines which provider to connect to
preTick()	First phase of the two phases execution. A simulator modeler should fill out this method with calls to provider's method to collect inputs, and code to load the data into the input buffer.
tick()	Second phase of the two phases execution. The actual computation happens in this phase. When computation is finished, outputs are stored in the output buffer.

Table 3.2 Key methods of Block class

3.3.3.2 Application-specific block

An application-specific block is a concrete class that inherits from the generic block. It specifies which input interface it agrees to implement, and it implements the necessary methods to serve as a provider. It can be view as a wrapper – to wrap a generic block in a form that another block can use. Code at this level is design specific, so the level of reusability is limited. When there are architectural changes in the computer design, code may no longer reusable if the connection is altered. But it is comforting to know that it only shares a very small portion of the total amount of code.

3.3.4 Data Type

Unlike aBlocks or Ptolemy, bBlocks input and output is not limited to a fixed data type or object. It can be a data structure that contains a group of data. So it is practically possible to transmit anything, including both data type and objects, from one block to another block. For example a result object would contain instruction id, instructions address, results, etc.

3.4 EXAMPLE

At this point, an example would help to explain the architecture more clearly. This example demonstrates how to model an adder, integrate it to the system and

connect it to other blocks to complete the design. Our adder has the properties described in Table 3.3.

Input	2 integers
Body	Add the two integers
Output	1 integer

Table 3.3 Specification of an adder

Let's assume that generic blocks IntegerFeeder and OutputPrinter are available to feed input to the adder and take the output from adder to some storage media.

3.4.1 Modeling a Block

To model a block, the first step is to define a provider, the input interface of adder. In this example, a provider to the adder is anyone who can provide two integers. To ease the data transmission and make it clearer, a unique data type is defined (Figure 3.2). The code of AdderProvider is shown in Figure 3.3.

```
public class TwoIntegers{
      public int a;
      public int b;
}
```

Figure 3.2 TwoIntegers class

```
public interface IntegerProvider implements Provider{

      TwoIntegers giveTwoIntegers();
}
```

Figure 3.3 IntegerProvider interface

The next step is to code the generic adder block. A generic block should have a constructor to initialize the member variables, a connectTo() method to get a handle to the block connecting to it, and preTick() and tick() for loading input and processing the data, respectively. Figure 3.4 shows the code for the generic adder block.

```
public class Adder extends Block{


	protected AdderProvider adderProvider;

	protected TwoIntegersQueue inputBuffer;
	protected TwoIntegersQueue outputBuffer;


	public Adder(){

		inputBuffer = new TwoIntegersQueue(1);
		outputBuffer = new TwoIntegersQueue(1);
	}

	public void connectTo( AdderProvider adderProvider ){

		this.adderProvider = adderProvider;
	}

	public boolean preTick(){

		if( !inputBuffer.isFull() ){

			TwoIntegers tmp = adderProvider.giveTwoIntegers();
			InputBuffer.push(tmp);
		}
		return true;
	}

	public boolean tick(){

		while( !inputBuffer.isEmpty() && !outputBuffer.isFull() ){

			TwoIntegers tmp = inputBuffer.pop();

			int sum;
			If( tmp != null ){
				sum = tmp.a + tmp.b;
				outputBuffer.push(sum);
			}

		}
		return true;
	}
}
```

Figure 3.4 Adder class

3.4.2 Modeling a Simulator

After completing the blocks for the simulator, it's time to put things together and build a simulator. First of all, we need a wrapper to dress IntegerFeeder in a way that Adder block can use. IntegerFeeder needs to implement AdderProvider, as shown in Figure 3.5.

```
public class SsIntegerFeeder extends IntegerFeeder implements
AdderProvider{

      public TwoIntegers giveTwoIntegers(){

            TwoIntegers out = new TwoIntegers();

            out.a = myOutputBuffer.pop();
            out.b = myOutputBuffer.pop();

            return out;
      }
}
```

Figure 3.5 SsIntegerFeeder class

To allow the generic adder block to be connectable, it also needs an application-specific block (Figure 3.6).

```
public class SsAdder extends Adder implements OutputPrinter.OutputProvider{

      public int giveOutput(){

            return outputBuffer.pop();
      }

}
```

Figure 3.6 SsAdder class

Now that all the application-specific blocks are done, they must be assembled to form the simulator system. Figure 3.7 shows the code for the simulator.

```
public class SampleScalar extends Simulator{

      SsIntegerFeeder        intFeeder;
      SsAdder                adder;
      SsOutputPrinter        outPrinter;

      public void setup(){
            intFeeder = new SsIntegerFeeder();
            adder = new SsAdder();
            outPrinter = new SsOutputPrinter();

            adder.connectTo(intFeeder);
            outPrinter.connectTo(adder);

            add(intFeeder);
            add(adder);
            add(outPrinter);
}

      public static void main( String[] args ){
            Simulator sim = new SampleScalar();

            sim.run();
      }
}
```

Figure 3.7 SampleScalar class

CHAPTER 4. SIMULATIONS AND RESULTS

To demonstrate the possibilities with bBlocks, we built two simulators: SuperScalar and CDF. They both support dynamic scheduling and have an out-of-order execution microarchitecture. In this chapter, we'll show how flexible bBlocks is to move from the basic SuperScalar microarchitecture to a CouterDataFlow pipeline microarchitecture.

4.1 SIMULATION METHODOLOGY

We build a set of generic blocks based on the basic SuperScalar microarchitecture. This set of generic blocks is independent to simulator structure and is reusable across different simulators (Table 4.1).

bBlocks does not yet have its own ISA and compiler. In the current release it relies on the front end of SimpleScalar to provide ISA definitions and binary generation capability. In spite of the fact that blocks were constructed around SimpleScalar's definition, blocks are independent to this definition and have the ability to adopt other definitions.

There is one drawback of using SimpleScalar's ISA and compiler. Recall that syscalls are instructions that provide operating-system-like services. These services include interfacing with the I/O. Since the syscall instructions were designed specifically for the UNIX environment using C, it is extremely difficult, if

not impossible, to completely port every syscall functionality to the java-based bBlocks. For this reason, we only implemented a few syscalls that are critical to us, including the stdout write function, and simulated the rest of the syscall by some constant return values.

Block	Description
PreFetch	It fetches instructions every cycle.
MemoryUnit	A basic unit of memory unit. Cache and Memory are built from that.
Cache	It can serve as either instruction or data cache, and it can be use for any level of cache.
Memory	Main memory. It has access to the virtual memory (which are files on disk).
Decoder	It decodes raw instructions.
IW	Instruction Window. It keeps the instructions that are pending to be executed.
EU	Execution Unit. It can serve as any functional unit, e.g. INT ALU, FP ALU, etc.
BEU	Branch Execution Unit. It is a child of EU. It is responsible for executing branch or jump instructions.
MEU	Memory Execution Unit. It is responsible for executing instructions that access to the memory. It also handles syscall instructions.
ROB	Re-Order Buffer. It's a buffer to maintain the retiring order of the instructions.
RF	Register File.

Table 4.1 Descriptions of generic blocks

4.2 SUPERSCALAR SIMULATION

The following discussion shows the simulator that was constructed to model a SuperScalar microprocessor.

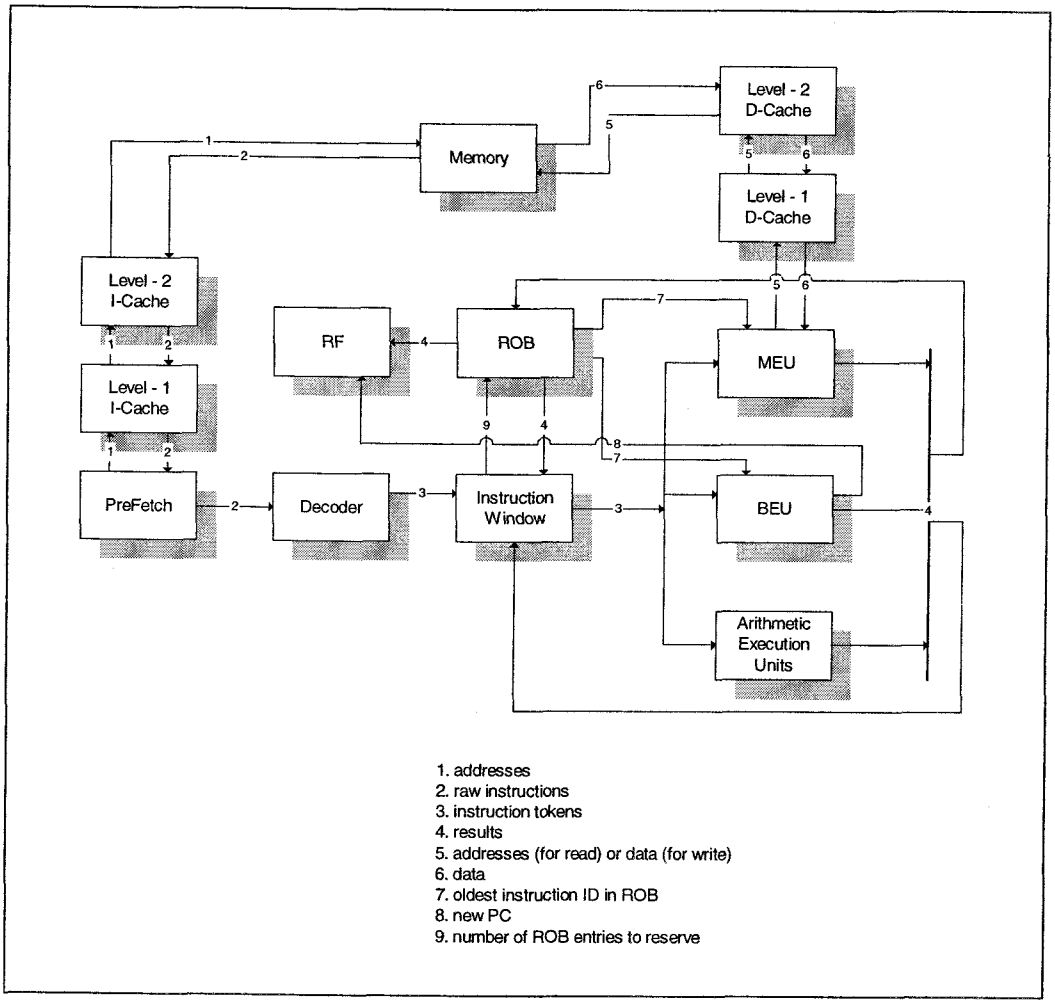

Figure 4.1 The SuperScalar structure that bBlocks implemented

4.2.1 Structure

The SuperScalar structure we built is presented in Figure 4.1. It consists of the application-specific blocks shown in Table 4.2.

The simulation starts with loading the program binaries to the memory. Once the binaries are loaded, the simulation cycle will begin. The following will demonstrate how an instruction travels through each block and retires to the Register File. (Note that in bBlocks, every transition is completed by putting data into a buffer to let the destination pick it up during the next pre-tick. Later, when we say passing data from one block to the other, it implies that the data is stored in a buffer and being picked up at the beginning of the next cycle).

1. The SsPreFetch generates an address according to the PC, and stores the address into its output buffer for the cache.
2. The first level cache, an instance of SsCache, takes the address from SsPreFetch. It looks up its content for a match. If it cannot find a match, it passes the address to the next level cache by storing it at the output buffer for the next level. Otherwise, it puts the raw instruction that matches the address to the output queue for the SsPreFetch. Let's assume we have a miss at this cache.
3. The second level cache, also an instance of SsCache, takes the address from the first level cache's output buffer, and repeats the same searching process as in the first level cache. Again, let's assume we have a miss at this cache.

4. The memory, represented by SsMemory, takes the address from the second level cache's output buffer. Then it brings the data from the instruction's virtual memory (implemented by a file), and stores it to the output buffer for the higher level (second level cache).
5. The second level cache gets the instruction from the Memory's output buffer. It updates the cache line with the instruction, and passes it on to the output buffer for the higher level (first level cache).
6. The first level cache repeats the same procedures as in second level and passes the instruction back to the SsPreFetch by putting the instruction into the output buffer.
7. SsPreFetch gets the instruction from cache, and passes it on to the next block, SsDecoder.
8. SsDecoder gets the raw instruction and decodes it into an instruction token. As opposed to a raw instruction that is nothing more than a fix size set of bits, an instruction token is an object that holds all the information about an instruction that must be present prior to execution. Surprisingly, it also takes extra effort to rename registers and attempts to load the necessary register values to the operand. After these tasks are done, it passes the instruction token to the instruction window.
9. SsIW gets the instruction token from Decoder and stores it into the instruction window. A just arrived instruction cannot be passed to the functional unit to execute until it has an entry in the re-order buffer reserved. For this reason,

SsIW will make an entry request to the SsROB after it receives an instruction token.

10. After an entry is allocated in SsROB, an instruction token may be fetched if all the operands are ready. If the operands are not ready, it will stay in the instruction window until the required results come back from the execution units or from SsROB.
11. When an instruction token is ready (when it has all its operands), the execution units (instances of SsEU) can take the instruction token from the instruction window. Note that if it is a memory access instruction, it will be executing in order. Results will be generated after execution, and they will be stored in an output result buffer, and forward buffer. The former buffer is for the result retiring (results will go to SsROB in this case), the later buffer is to perform result forwarding (going to SsIW in this case).
12. SsROB takes the results from the functional units, and fills them into their reserved entry. Every cycle, SsROB looks from the bottom of the buffer where the oldest results are located, and marks the results retired in descending order.
13. SsRF takes results in the SsROB that are marked retired, and writes them to the register file.

Applicatons Specific Blocks	Generic Block Inherit From:	Implemented Providers to connect to:
SsPreFetch	PreFetch	MemoryUnit
SsCache	Cache	Decoder
SsMemory	Memory	MemoryUnit
SsDecoder	Decoder	IW
SsIW	IW	ROB, EU
SsEU	EU	IW, ROB
SsBEU	BEU	IW, ROB, RF
SsMEU	MEU	IW, ROB, MemoryUnit
SsROB	ROB	IW, RF, MEU, BEU
SsRF	RF	

Table 4.2 Descriptions of application specific blocks in SuperScalar simulator

4.2.2 Configurations

To understand the performance of SuperScalar, we ran simulations for some SPEC95 [9] benchmarks and programs. The simulation results obtained from these four programs will be used as the baseline to compare the performance with CDF.

The SuperScalar simulator can fetch, issue, and complete up to four instructions per clock. It has four fast integer units, one slow integer unit, one fast floating-point unit, and one slow floating-point unit. They have a latency of one, four, four, and four, respectively. There is one memory execution unit to take care of the memory access instructions, and one branch execution unit to execute branch or jump instructions. It has two levels of instruction cache as well as a data cache, and one memory unit. The cache size for level one instruction cache, level two instruction cache, level one data cache, and level two data cache are 32, 128, 16,

and 64 bytes, respectively. It takes two cycles to access the first level cache, four cycles to access the second level cache, and six cycles to access the memory. Both instruction windows and the re-order buffer have 32 entries.

4.3 COUNTERDATAFLOW SIMULATION

To support future studies in CounterDataFlow microarchitecture and to demonstrate the flexibility of bBlocks to adopt different designs, a CDF simulator is built. The following sections discuss the construction of this simulator.

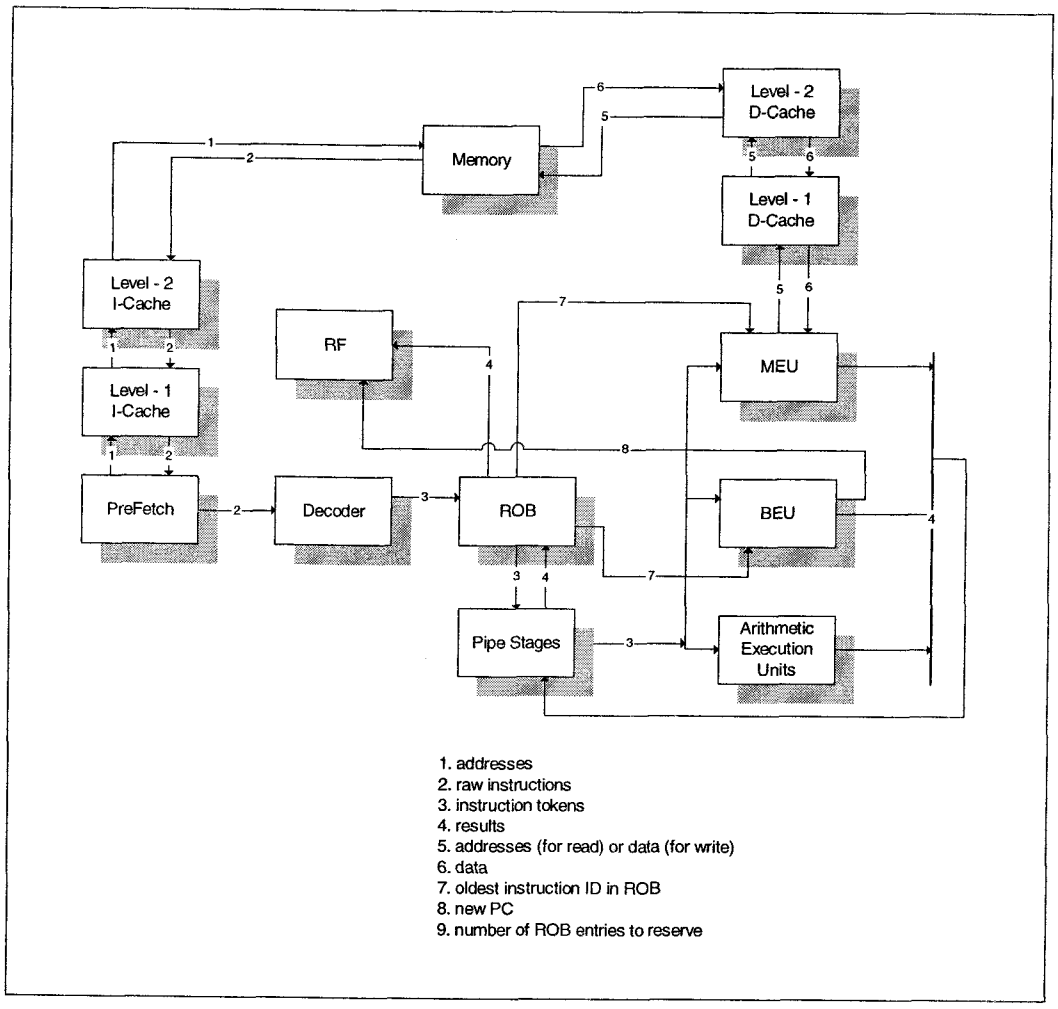

Figure 4.2 The CDF structure that bBlocks implemented

4.3.1 Structure

CounterDataFlow pipeline microarchitecture is an architecture derived from the counterflow pipeline concept. CDF differs from the basic SuperScalar structure by having a set of instruction pipelines and a set of result pipelines as opposed to SuperScalar's instruction window. Instructions are passed from Decoder to ROB, and from ROB to the instruction pipeline stage that is closest to the ROB (the first

instruction pipeline stage). Every cycle, instructions in the instruction pipeline will shift one pipeline stage away from the ROB. When instructions reach their launch points, instructions will be launched to the corresponding execution units, if they have all the operands ready. Instructions continue their adventure in the pipeline otherwise. When results come back from the execution unit, they are transferred to the result pipeline. Results in the result pipeline shift one stage toward the ROB every cycle. The name "CounterDataFlow" describes the fact that instructions and results are flowing in opposite directions in the pipeline stages. If instructions find a match of results that they are waiting for at the same stage of pipeline, they take the values from the result pipeline. When the results reach the ROB, the ROB will retire them in order. On the other hand, when instructions reach the last stage without launching, they will wrap around to the ROB and be fetched again to the first stage. One keynote is that the CDF pipeline does not stall.

The simulator that we built for CounterDataFlow pipeline microprocessor reused most of the objects from SuperScalar simulator. Table 4.3 and Table 4.4 show the generic blocks and the application specific blocks implemented for CDF.

Block	Description
CdfROB	The completion unit for CDF. Besides the basic re-order buffer functionalities to retire instructions in order, its tasks include passing instruction tokens to the pipeline, taking care of wrap around instructions, and perform values matching to the instructions.
CdfPipe	It represents the pipeline stages in CDF, both instruction and result pipelines.

Table 4.3 Descriptions of generic blocks implemented for CDF

Applicatons Specific Blocks	Generic block inherit from:	Providers implemented:
SsCdfROB	CdfROB	CdfPipe, RF, BEU, MEU
SsCdfPipe	CdfPipe	EU, BEU, MEU
SsCdfEU	EU	CdfPipe
SsCdfMEU	MEU	CdfPipe, MemoryUnit
SsCdfBEU	BEU	CdfPipe, RF

Table 4.4 Descriptions of application specific blocks for CDF simulator

4.3.2 Configurations

The same integer and floating-point programs used for the SuperScalar simulations were also used for CDF simulations. In order to collect comparable simulation results, all the CDF configurations are set to the same as SuperScalar. While CDF does not have an instruction window, it has pipeline stages. It has eight pipeline stages. Each stage can hold up to four instructions and eight results. This configuration of pipeline stages accommodates the 32-entry instruction window in SuperScalar. CDF simulator has the same number of executions with the same

amount of latency. The positions of each execution unit are listed in Figure 4.3. This sidepanel setup is similar to the setup presented in [7], except that it only has eight stages instead of nine, and one branch execution unit as opposed to two BEU used in [7].

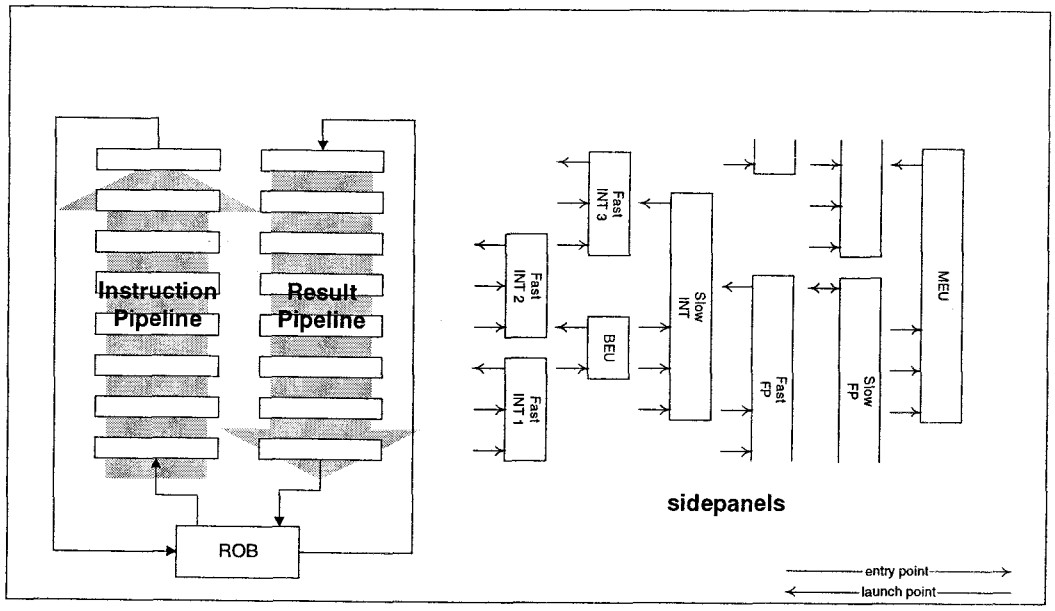

Figure 4.3 Simulated CDF sidepanels placement

During the development of bBlocks, our simulator uncovered a design flaw in the CDF microarchitecture. This flaw is caused by memory access instructions that should be executed in order. The original design did not guarantee the execution order of these instructions, so only the memory access instructions in the buffer of the BEU are kept.

```
1:    Store   R3, Location_1
2:    Load    R4, Location_1
3:    Load    R5, Location_1
```

Figure 4.4 Sample code segment

Consider the code shown in Figure 4.4. Assume the MEU has one buffer. In the case where R3 is being used by other instructions but R4 and R5 are ready, Load2 and Load3 will be fetched to the MEU, while Store1 remains in the pipeline. There are two preferences to execute these instructions: perform the memory access only when the corresponding instruction is retiring from the ROB, or allow "load passes store" if the load instruction is not reading from the memory location that the previous store is writing to.

For the former case, Load2 will be in the MEU pending for the signal from ROB to perform the memory access, and Load3 waiting in the buffer of the MEU. As one may observe, Load2 will not be retired until after Store1 is retired. However, Store1 will never have a chance to get to the MEU, because the MEU is occupied by Load2, and the buffer is filled by Load3. In another words, Store1 will never be retired and a deadlock is generated.

For the later case, load instructions can pass store instructions. In the example, if Load2 reaches the MEU before Store1 because MEU is lacking information about the earlier store instructions, it will allow Load2 to be executed, before Store1 comes to the MEU and writes to Location_1. It is obvious that the

instruction execution order is no longer maintained and will result in incorrect execution.

The solution that we applied to our CDF simulator is so called "dynamic dependency adding". At the ROB, it keeps track of the memory access instructions. When it sees one, it will dynamically attach an operand to it, an operand that's waiting for the previous memory access instruction. When this instruction travels in the pipeline, it will not be launched to its execution unit until it finds the result from the previous memory access instruction, solving the dependency.

Since it is not the goal of this research to investigate microarchitecture designs, this topic will not be discussed any further. But it should be noted that the "dynamic dependency adding" does affect the performance of CDF microarchitecture. Since those instructions depend on the previous memory access instructions and it's not trivial to anticipate when the instruction will solve the dependency, there is a good chance that the instructions may not be able to solve the dependency before they reach the end of the pipeline stages and wrap around. The designer of CDF in the future should pay more attention to the placement of execution units to take this issue into account.

4.4 SIMULATION RESULTS: CDF VERSUS SUPERSCALAR

This section investigates the results obtained from the simulations. Since the purpose of these simulations is mainly to demonstrate the potential of bBlocks, we are not going to study the results from the two microarchitectures extensively. Instead, we will discuss only some of the major observations, and focus on studying the performance differences between SuperScalar and CDF.

The following tables present the simulation results using different configurations. Every simulation ran two thousand instructions. The purpose for obtaining these results is to demonstrate the capabilities of bBlocks. It should be noted that the simulator was not fine tuned, because optimizing a microarchitecture is beyond the scope of this research; therefore the microarchitecture performance is not necessarily optimal. But every effort was made to keep all simulation parameters the same across SuperScalar as well as CDF simulations to generate comparable results.

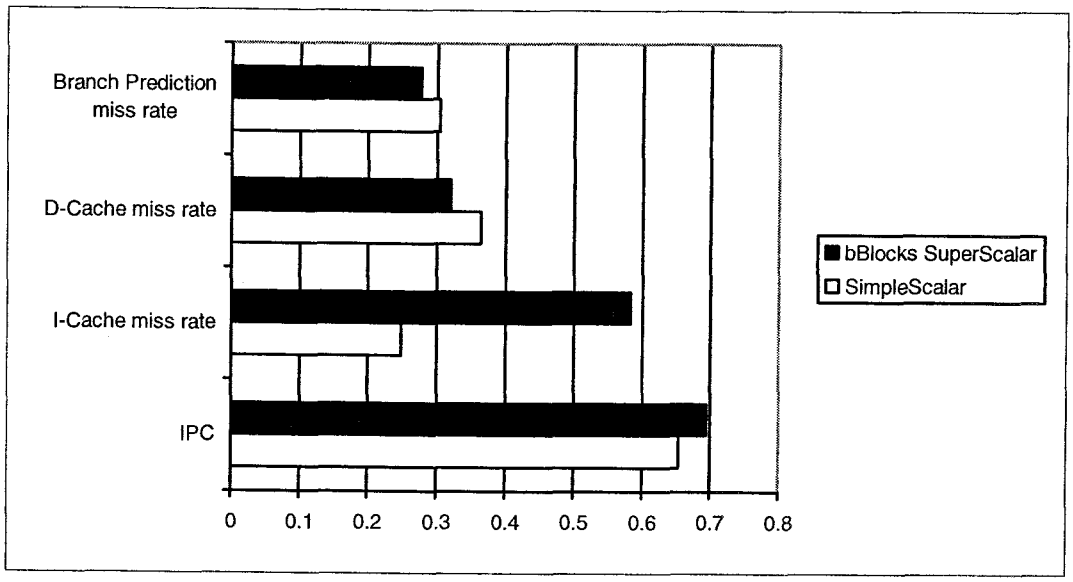

Figure 4.5 applu simulation results

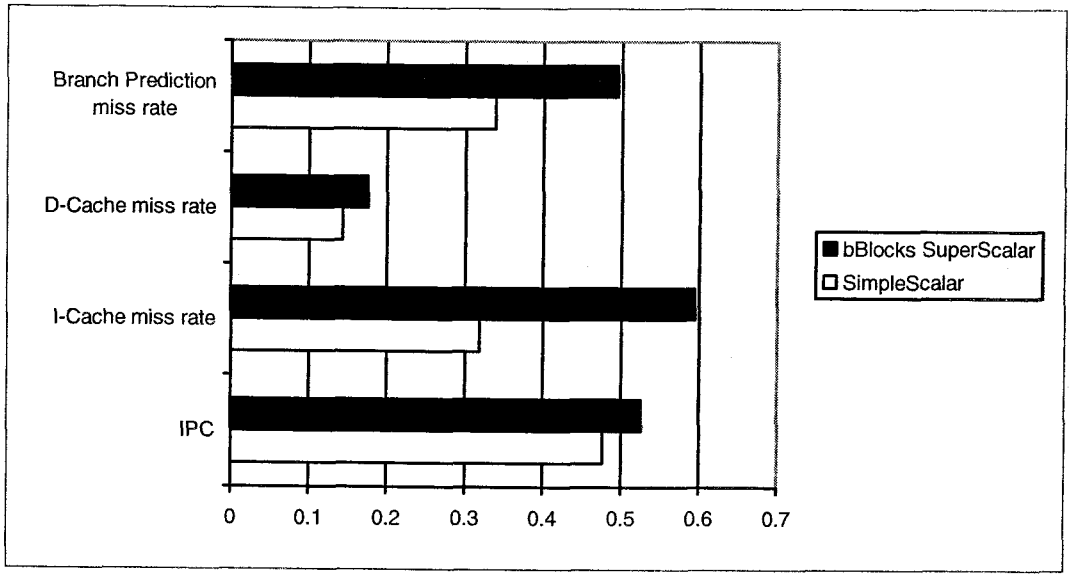

Figure 4.6 swim simulation results

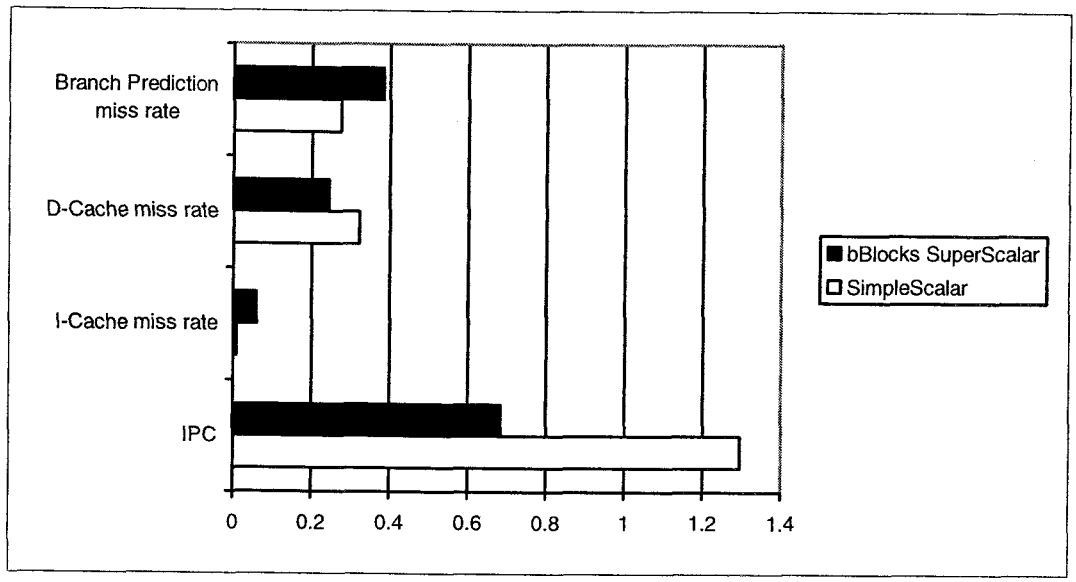

Figure 4.7 compress95 simulation results

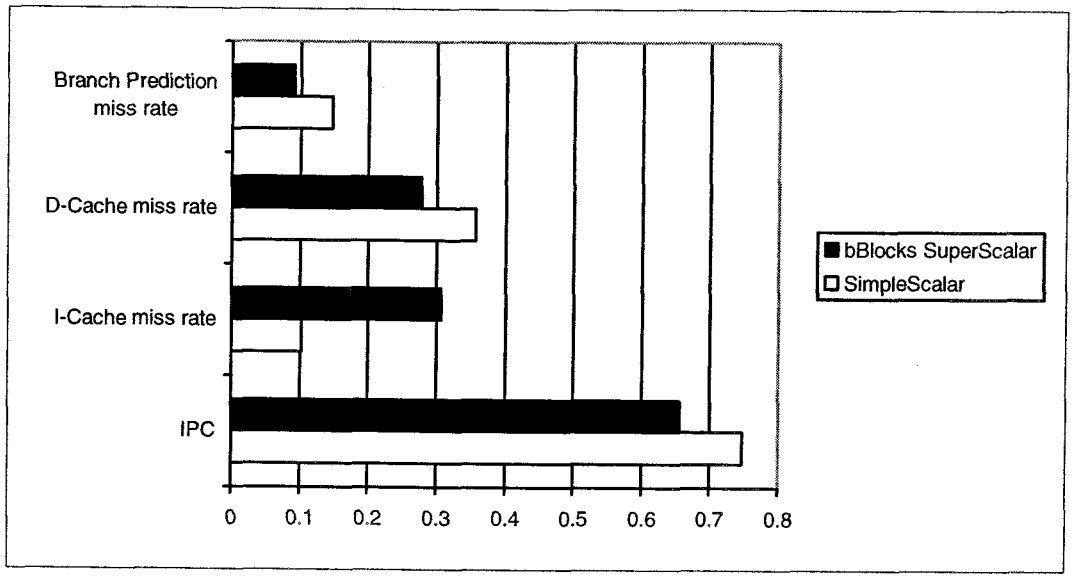

Figure 4.8 fpppp simulation results

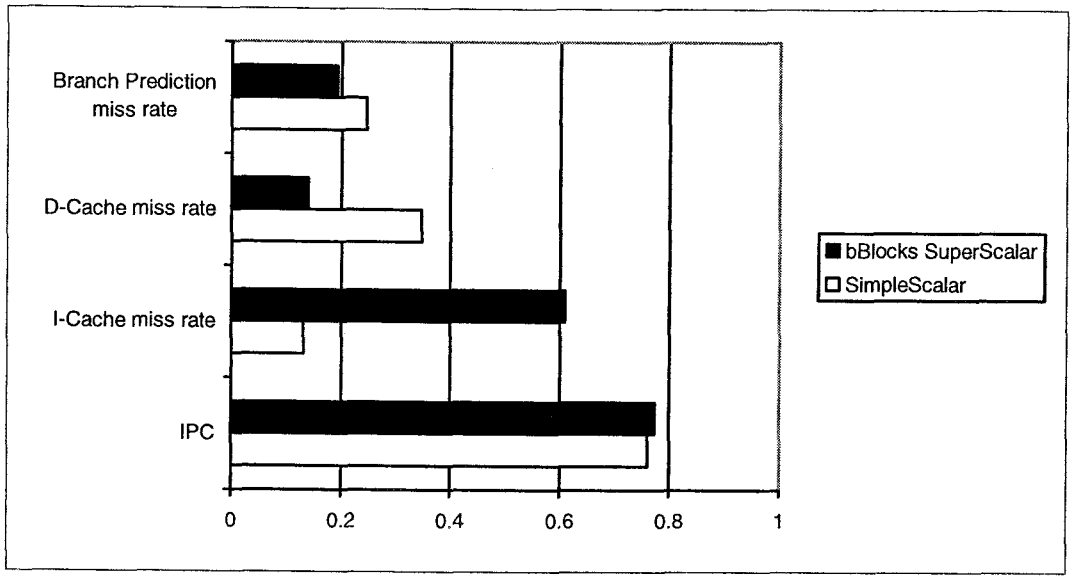

Figure 4.9 turb3d simulation results

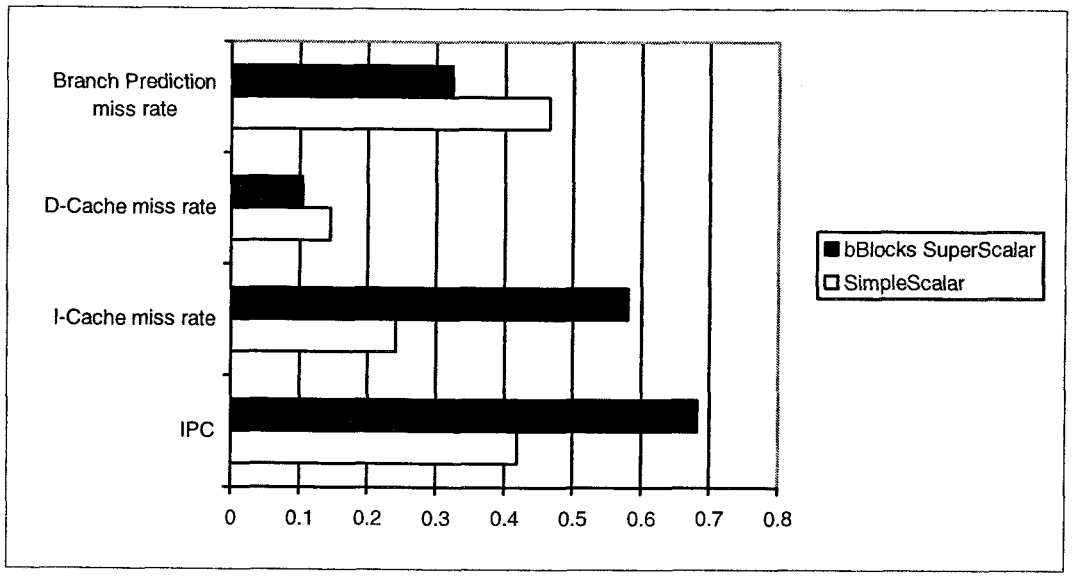

Figure 4.10 test-printf simulation results

The simulation results point out that CDF does not have a better performance in terms of instructions per cycle (IPC) against SuperScalar under the same conditions. CDF is about two times slower than SuperScalar, averaging about 0.2 instructions per cycle, while SuperScalar has over 0.5 IPC on average.

Readers may find the results differ from that obtained in the earlier research on CDF using aBlocks [4, 7]. Although lower cache and branch prediction miss rates do make a contribution to the better performance, the major factors that offer better results in aBlocks simulations are the large ROB size and the neglect of the memory access instructions execution order. Without taking care of instructions execution order carefully in aBlocks, there is little true data dependency between instructions. Thus, instructions can be executed earlier and in turn, more instructions can be retired in a cycle. For the result from [4, 7], aBlocks simulations used a ROB size of 128 entries to get the best performance. The role of a large ROB is to allow new instructions to issue from the decoder without stalling while the older instructions are still traveling in the pipeline. In addition, the ability to retire an indefinite number of instructions per cycle also benefits aBlocks IPC.

Bblocks' CDF simulations implemented "dynamic dependency adding" to maintain the order of memory access instructions, and used a ROB size of 32 entries only. In this scheme, a load instruction that reads from a location where an earlier store instruction is writing will not be launched to the execution unit until the store instruction finishes execution. When the store instruction is executed, the result will enter the result pipe at the corresponding entry point. There is the

chance that the following load instruction will either miss the result, meet the result immediately, or hit the result in a few cycles. In any case, however, the following load instruction will have a good possibility to travel in the pipeline for a few more cycles before it can meet the launch point again. In the case that an instruction misses the result, it will take even longer. Assume that a memory access instruction take three cycles to get to the launch point after resolving the data dependency on average, executing a memory access instruction will be three times slower than SuperScalar. The performance of CDF will therefore be greatly reduced, especially for programs with a large percentage of memory access instructions.

Even for a program with fewer memory access instructions, it is not very optimistic. The long latency memory access instructions cause more instructions to stay in the ROB without retiring. For this reason, the ROB is easily filled up. For a four issue CDF microprocessor, a full ROB prevents instructions from issuing even though it is very likely that there are spaces in the pipeline. Consequently, a four issue microprocessor wastes the chance to issue four instructions for every cycle the ROB is full. Unfortunately, this happens very often in a 32-entry ROB as shown in Figure 4.11 and Figure 4.12.

The performance of CDF greatly relies on optimal sidepanel placement and ROB size, according to [7]. While optimal sidepanel placement is dependent on the nature of the program, it is feasible to enlarge the ROB to some extent. Increasing the ROB size from 32 to 128 entries should help to improve the IPC.

However, the simulation results in Figure 4.13 do not conform that. It is obvious that CDF performance is limited by factors besides ROB sizes. This research will leave the investigation of CDF performance bottlenecks to the future studies. Future research should run simulations with more instructions to obtain results with better accuracy.

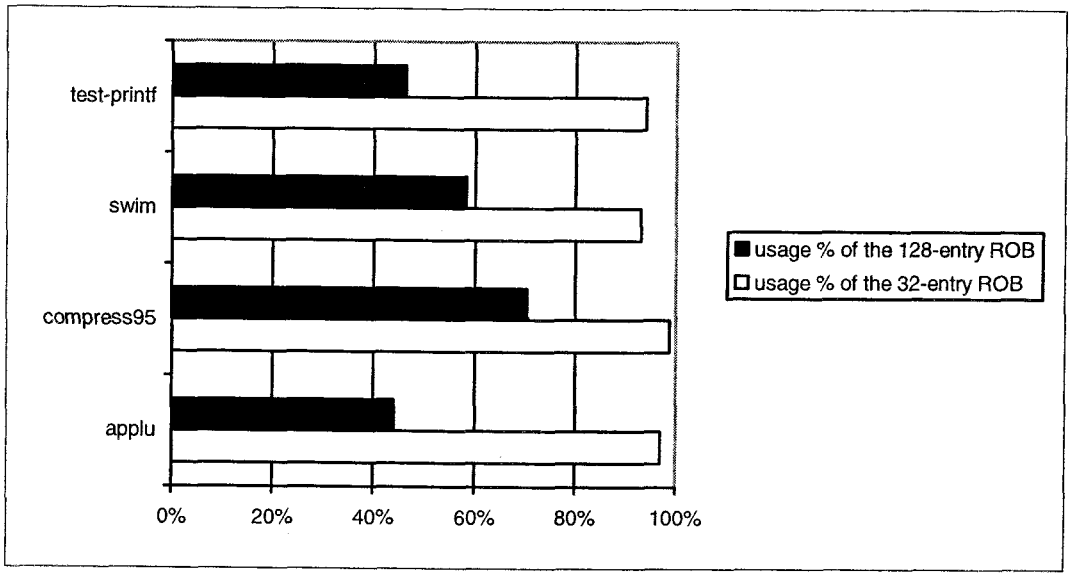

Figure 4.11 Usage percentage of CDF ROB with 32-entry and 128-entry.

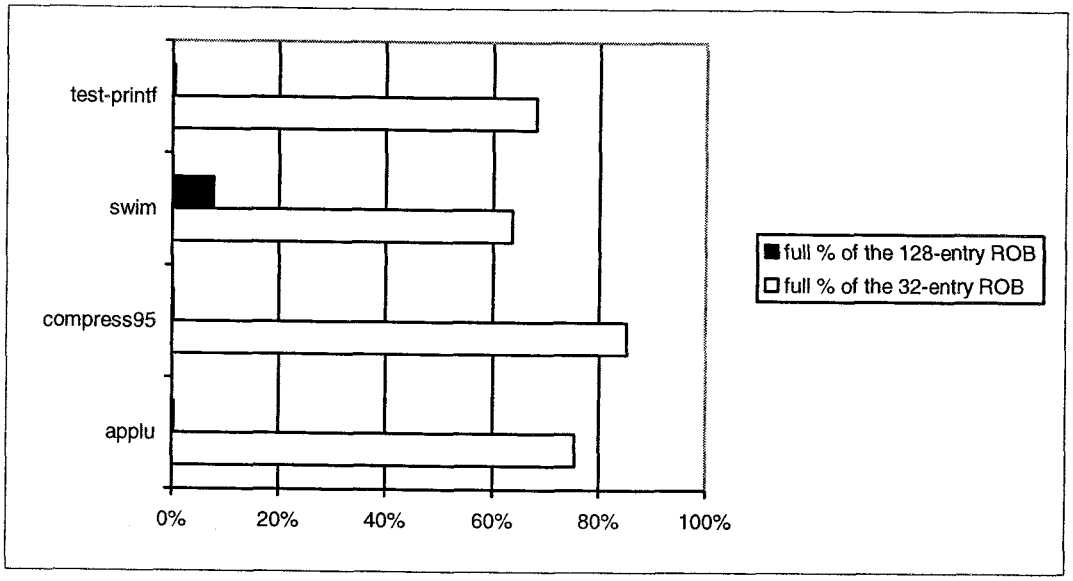

Figure 4.12 Full percentage of CDF ROB with 32-entry and 128-entry.

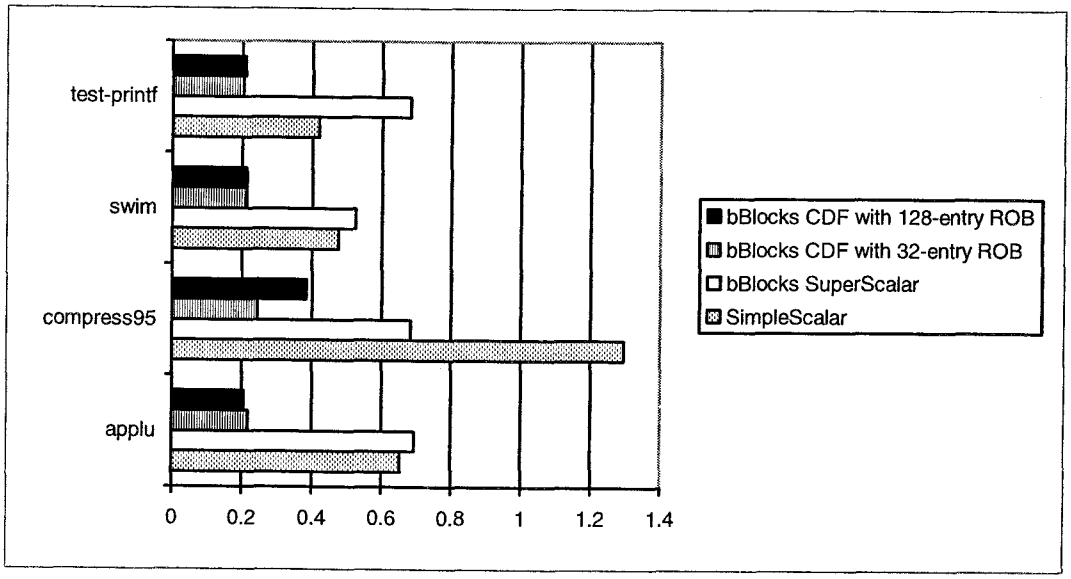

Figure 4.13 IPC comparison

CHAPTER 5. CONCLUSIONS

We have designed and developed a simulator modeling tool for microarchitecture, bBlocks. It is a Java-based application with 19,000 lines of code that takes advantage of object-oriented methodology to obtain better flexibility, and code reusability as well as object reusability.

The development of bBlocks provides a sophisticated simulation tool to the microarchitecture designer to prototype designs with the minimum amount of time. The flexibility offered by bBlocks opens the possibility for researchers to study and optimize designs by adjusting different parameters of the simulator.

It has a re-designed structure that is completely different from its ancestor aBlocks. While bBlocks is also a component-based simulator like aBlocks, it has a "jigsaw puzzle structure", which consists of four kinds of base objects: Simulator, Provider, Block, and Data objects. This structure allows improved code reusability, and provides high flexibility to integrate new blocks to the simulator system to adopt new designs. Execution-driven property also gives advantages to bBlocks over aBlocks, offering greater details and more accurate simulation results. It found design problems in the microarchitecture that aBlocks did not reveal. The simulations also indicate that CDF requires more hardware than SuperScalar to attain the same level of performance.

The experience of building two simulators for SuperScalar and CDF microarchitectures demonstrated how bBlocks achieves great code reusability as

promised. Implementing the CDF simulator only required adding 2000 lines of new code to the SuperScalar components already available. It has only 13% of the total number of lines of code in the CDF simulator. In other words, over 86% of the code in CDF simulator is reused from the SuperScalar simulator. This flexibility not only accelerates a simulator development, it also relieves the burden of debugging as the majority of the code is tested.

CHAPTER 6. FUTURE EXTENSIONS

Based on experience gained in this research, a number of useful extensions to bBlocks can be envisioned.

6.1 PERFORMANCE ENHANCEMENT

While bBlocks has great flexibility and details, the simulation performance is slower than aBlocks. It averages about 5 simulation cycles per second on Pentium II-266MHz running Windows NT 4.0. It is much slower than the trace-driven aBlocks, which can cycle about 50 times in a second.

To obtain better flexibility, we chose the object-oriented language Java to develop bBlocks. Object-oriented is by definition less efficient than structural methodology. An interpreted language is also significantly slower than machine language. Therefore in terms of simulation time, bBlocks is no match for a simulator like SimpleScalar that is built on the structural language C. The simulation performance could be improved by porting bBlocks to C++, an object-oriented language that generates platform dependent machine code. While a C++ implementation would lack platform independent capability and trouble-free GUI support, it would give an order of magnitude improvement to bBlocks.

6.2 FAST FORWARDING

There are times when the simulating program is large, but we are only interested in investigating how a portion of the code behaves in the simulator. Currently this requires executing all programs from the beginning. However, since sometimes a simulation with all details will take a long time to run, it might take days to reach the code that we are interested in. A fast forwarding feature would be helpful if it executes the code that is less important to us as fast as possible.

It would be difficult to perform the exact functionality of fast forwarding, in the sense that the simulator fast forwards to anywhere in the program, and all the blocks have all the state information once the fast forward is stopped. But the simulator can do the simplest, fastest, in-order execution to skip to the section that we are interested in. During the transition that fast forward has just stopped and out-of-order execution is beginning, blocks in the simulator will not have state information. It is as if the program has just started.

With bBlocks flexibility, this functionality is not impossible to implement. One possible way to implement it is to allow dynamically adding, removing, connecting or disconnecting blocks. Then implement a block for fast forwarding, and replace all blocks from PreFetch to ROB with it. The memory and Register File blocks will need to be kept throughout fast forwarding or normal simulation in order to maintain data consistency.

6.3 SYSCALL

bBlocks borrows the ISA definition from SimpleScalar, including the syscall definition. Due to the fact that SimpleScalar was developed in C for the UNIX environment, it is tedious to completely port all syscall functionalities to Java-based bBlocks. In the future, effort could be spent on developing a custom ISA definition and compiler for the long run, or modifying SimpleScalar's syscall definition and compiler as a short-term goal.

6.4 STALLING

Since information and controls signals are highly localized in bBlocks, it is quite tricky to handle stalling. Because bBlocks does not have a global controller to control the dataflow – blocks make decisions themselves – it is not possible to propagate the stall information all the way back in one simulation cycle. The current version of bBlocks uses buffers to work around this problem. A block does not worry whether the next block will take the results or stall. It just produces a result and puts it into the buffer, as long as the buffer is not full.

While this work around seems feasible, there are cases that it cannot handle (a two-stage pipeline for example). To move data from one pipeline stage to the next stage, the first pipeline puts the data in the output buffer and waits for the next pipeline to catch it at the beginning of the next cycle. The problem reveals itself when a stall happens. Consider the example below with two pipelines. During pre-

tick, a block takes inputs from the previous block if its input buffer is not full. During tick, a block processes its input and puts the result in the output buffer. Assume the block connected to Pipe-2 input is stalled at cycle three and does not take the data from Pipe-2's output buffer. Pipe-2 does not sense the stall at cycle three and continues to take input. Now, there are three data values in the pipeline in two pipeline stages.

Simulation cycle	Pipe –1		Pipe – 2	
	Input buffer	Output buffer	Input buffer	Output buffer
1a. pre-tick	A			
1b. tick		A		
2a. pre-tick	B		A	
2b. tick		B		A
3a. pre-tick	C		B	A

Table 6.1 Example to demonstrate the stalling problem

Extensions to bBlocks can be implemented to fix this problem. Along with local signals, global signals can be added to signify to all blocks that there is a stall happening. A block reacts to this signal and decides how many inputs it should take for the next cycle. This implementation may require complex algorithms and structural modification in bBlocks, however.

BIBLIOGRAPHY

1. R.F. Sproull and I.E. Sutherland and C.E. Molnar, "The Counterflow Pipeline Processor Architecture," IEEE Design and Test of Computers, vol. 11, no. 3, pp. 48-59, Fall 1994.

2. K.J. Janik and S. Lu, "Synchronous Implementation of a Counterflow Pipeline Processor," ISCAS, vol. 4, pp. 69-72, May 1996.

3. K.J. Janik and S. Lu and M.F. Miller, "Advances to the Counterflow Pipeline Microarchitecture," in HPCA-3, February 1997, pp. 230-236.

4. K.J. Janik and M. Miller and S.L. Lu, "Non-Stalling Counterflow Architecture," in Proceedings of HPCA-4, February 1998, pp. 334-341.

5. D.C. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin Computer Science, Madison, Wisconsin, Tech. Rep. #1342, June 1997.

6. J.E. Smith and G.S. Sohi, "The Microarchitecture of Superscalar Processors," in Proceedings of the IEEE, in Proceedings of the IEEE, December 1995, vol. 83, no. 12, pp. 1609-1624.

7. K.J. Janik, "A Microarchitecture Study of the Counterflow Pipeline Principle," Ph.D. Dissertation, Oregon State University, Corvallis, Oregon, February 1998.

8. B. Venners, "Design Techniques," April 1999, http://www.javaworld.com/javaworld/jw-04-1999/jw-04-techniques-2.html.

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

10. E. Lee, "The Ptolemy Project II – Heterogeneous Concurrent Modeling and Design in Java," January 2000, http://www.ptolemy.eecs.berkeley.edu.

APPENDIX

This appendix briefs the system requirements and usage of bBlocks.

A.1 SYSTEM REQUIREMENTS

bBlocks runs on any platform that support JDK 1.2.2 or above. Although it is not required, it is highly recommended to run bBlocks on a virtual machine that has Java HotSpot™ (performance engine) installed. It greatly reduces the elapsed time of the bBlocks simulations.

A.2 INPUT FILES PREPARATION

bBlocks runs only the big-endian binary generated by SimpleScalar. Prior to performing a bBlocks simulation, the simulating program should be compiled on a big-endian machine by the SimpleScalar compiler.

bBlocks needs a definition file before running a simulation. This definition file tells bBlocks how the simulator should be configured. It also tells bBlocks which binary file to simulate. Currently bBlocks has two definition files (superscalar.def and cdf.def) for SuperScalar and CDF simulations.

A.3 STARTING THE SIMULATION

There are two "jar" files for SuperScalar and CDF simulation. To run a simulation, one may simply execute the corresponding jar file, with the definition file name as the argument. For example, if a designer were to run SuperScalar simulation, the following command should be invoked at the command prompt:

```
java -jar superscalar.jar superscalar.def
```

Depending on the configuration in the definition file, bBlocks runs in either batch mode or graphic mode. In graphic mode, each block has a window. At the end of each simulation cycle, all blocks dump the internal state information to the corresponding window.

A.4 COLLECTING SIMULATION RESULTS

At the end of the simulation, bBlocks will generate a report file. This file contains all the simulation results, including block usage, branch miss rate, and cache miss rate. BBlocks also generates a log file for all the retired instructions if the option in the definition file is set to active. The name of the report file and log file are specified in the definition file.