AN ABSTRACT OF THE THESIS OF

Ying Hua for the degree of Master of Science in Electrical and Computer Engineering presented on June 19, 2000. Title: bBlocks: A Flexible Object Oriented MicroArchitecture Simulation Framework.

Abstract approved: Redacted for Privacy

Shih-Lien Lu

This research develops an object-oriented approach of modeling microprocessor architecture. A generic modeling library, bBlocks, is proposed as a framework for constructing microprocessor simulation. bBlocks is a collection of predefined abstract components (blocks) implemented in Java, the object-oriented programming language.

Blocks are defined and used as the basic components in a variety of microprocessor operation modeling, simulations, and optimizations in bBlocks. The basic activities of a microprocessor are encapsulated into blocks. By using standard interfaces, blocks are integrated into the proposed modeling framework. Unlike traditional software simulators, this framework is cycle based, concurrent capable simulation, which may take advantage of multi-threading or parallel computation techniques.

Using an object-oriented approach, bBlocks adapts an open system policy, which gives end users the flexibility of incorporating user-defined components into end models and integrating the user-preferred architecture. Another purpose of this research is to provide a mechanism to model and study future microprocessors. bBlocks is implemented in Java, a truly cross-platform object-oriented language. It is conceivable that bBlocks could eventually run on any machine, regardless of the architecture or operating system as long as it is running the Java Virtual Machine. In addition to this, with Java GUI tools, bBlocks provides probes to investigate the activities inside the blocks.

To illustrate the effectiveness of bBlocks, two architectures, SuperScalar and CDF, have been implemented.

bBlocks: A Flexible Object Oriented MicroArchitecture Simulation Framework

By

Ying Hua

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 19, 2000
Commencement June 2001

Master of Science thesis of Ying Hua presented on June 19, 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

Ying Hua, Author

ACKNOWLEDGEMENT

I would like to express my sincerest thanks to Dr. Shih-Lien Lu, who serves as my major advisor, for his guidance and continuing support of my graduate study. Without his thoughtful advice and encouragement, this experimental research would have not been finished.

My thanks go to Dr. Gregg Rothermel for serving as my minor advisor. I benefit greatly from his Software Engineering class.

I would like to thank Dr. Alexandre F. Tenca for serving as my committee member and his technical advice.

I would like to thank Dr. Chih-hung Chang for serving as my Graduate Council Representative and his technical advice.

I would like to thank my teammate Chung-lun Chan with whom I have had many fruitful discussions during the course of this research.

I gratefully acknowledge the CFPP research group at Oregon State University. This research is based on the valuable results of the previous research.

I am deeply indebted to my parents, to my whole family, and specially, to my husband, Binglei Zhang, for their encouragement and dedicated support.

TABLE OF CONTENTS

Page

Chapter 1 INTRODUCTION 1

1.1 Background Overview 1

1.2 From aBlocks To bBlocks 3

1.3 Objective 5

1.4 Scope Of This Study 6

Chapter 2 MODELING CONCEPT 8

2.1 Object Oriented Technology 8

2.2 Modeling Methodology 10

2.2.1 Functional decomposition 11
2.2.2 Object decomposition 12
2.2.3 Comparison 13

2.3 First Level Abstraction 14

2.3.1 Block abstraction 15
2.3.2 Specific strategy 17

2.4 Decomposition 18

Chapter 3 MODELING LIBRARY 20

3.1 Notation 20

3.2 PreFetch 21

3.3 Decoder 22

3.4 Instruction Window 22

3.5 Execution Unit 23

TABLE OF CONTENTS

Page

3.6 Reorder Buffer 24

3.7 Register File (RF) 25

3.8 Memory Unit 26

3.9 Strategies 26

Chapter 4 SIMULATION ENGINE 28

4.1 Pack Blocks Into Simulation Engine 28

4.2 SuperScalar Simulation Engine 32

4.3 CDF Simulation Engine 35

Chapter 5 SIMULATION RESULT 37

5.1 Functional Verification 37

5.2 Statistic Verification 38

5.3 Additional Discussion 42

Chapter 6 CONCLUSIONS AND RECOMMENDATIONS 44

6.1 Conclusions 44

6.2 Recommendations For Further Study 46

6.2.1 Accomplish the modeling library 46
6.2.2 Optimize the performance 46

REFERENCES 47

LIST OF FIGURES

Figure Page

1. Austin's Triangle 3
2. Basic Block 15
3. Block Abstraction 16
4. Diagram Notation 20
5. PreFetch Block 21
6. Decoder Block 22
7. Instruction Block 23
8. Execution Unit 23
9. BEU Block 24
10. MEU Block 24
11. ROB 25
12. RF 25
13. Memory Unit 26
14. Simulation Framework 30
15. SuperScalar Simulation Manager 31
16. Basic SuperScalar Execution Model (Centralized) 33
17. SuperScalar Simulation Engine 34
18. CDF Architecture 36
19. Instruction Per Cycle 38

LIST OF FIGURES (CONTINUED)

Figure Page

20. Branch Prediction Miss Rate 39

21. Instruction Cache Miss Rate 40

22. Data Cache Miss Rate 41

LIST OF TABLES

Table — Page

1. Contrasting Traditional and Object-Oriented Paradigms for Simulation Modeling 13
2. Strategies 27
3. Software Structure 43
4. SuperScalar Configuration Parameter 43

bBlocks:
A Flexible Object Oriented MicroArchitecture Simulation Framework

Chapter 1 INTRODUCTION

1.1 Background Overview

This study is one of the projects being carried out by the CFPP (CounterFlow Pipeline Processors) research group at Oregon State University. The counterflow pipeline concept was originated by Sproull [1] as architecture for asynchronous processor design. It offers many useful properties including local control, local message passing, and an overall simple design methodology. The CFPP research group introduced the same idea to synchronous processor design and has made considerable progress by proposing the addition of two improved approaches, Virtual Register Processor (VRP) [2] and CounterDataFlow Processor [3] [4](CDF), to the original CFPP architecture. A general-purpose simulator was also proposed for measuring and evaluating various architectures.

In recent years, microarchitecture became more and more complicated. To construct a new processor or even part of it can be very expensive and time consuming. Thus, more and more microarchitecture researches depend on software simulation tools. Unfortunately, most of the simulation tools have been developed for some very specific research areas, and are difficult to modify to simulate things

that they were not designed to simulate [5], such as a different architecture, a different instruction set, or even multiple hardware contexts for multithreading. In this case, if you have a new idea that is significantly different from existing simulated designs, you are usually stuck with writing a simulator from scratch, or spending a lot of time attempting to modify an existing simulator to do something it was never intended to do. Diep and Shen [6] proposed a microarchitecture workbench called VMW in attempt to generalize simulator writing. However, their work is only suitable for modeling superscalar machines.

For the above reason, CFPP research group proposed a general purposed microarchitecture simulation package. The primary goal of this package is flexibility, which means it is easy to adapt to new microprocessor architectures. The performance of the simulator itself is important, but secondary. Todd Austin [7] clearly described the relationship of the three features to simulators. All simulators have three exclusive features, flexibility, speed and detail, which need to be balanced between as shown in Figure 1. Simulations can be optimized usually for one or possibly two of the three features. However as one optimizes for one feature, it may affect the other two features. Most simulators currently optimize for speed and to a less extent for detail and flexibility.

The simulator proposed here has a reasonable amount of detail, provides lots of flexibility and reduces development time at the expense of speed. Hopefully, researchers can concentrate on problem depth rather than on the development of simulation tools.

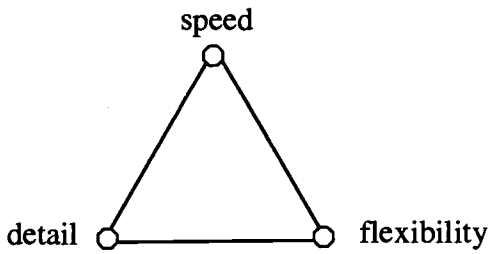

Figure 1. Austin's Triangle

1.2 From aBlocks To bBlocks

"Architectual Blocks" (thus "aBlocks"), the first version of simulation framework, was developed by Michael F. Miller and Kenneth J. Janik [8]. They successfully utilized the object-oriented concept in their design by declaring each significant structure in the processor as an aObject. aObject is the basic component to contain or hide a processor's data structure and operations. It was a good beginning of making the code reusable. In aBlocks framework, numerous general objects were made, including pipeline object, execution unit object, ROB object, cache object, decode and fetch object. Directly mapping hardware unit to software object makes it quite straightforward to understand the simulator code and structure.

Java [9] was used to implement aBlocks. There are following benefits of using Java according to Janik [10]. Java is a simple, object-oriented, distributed, architecture neutral, portable, high-performance, multithreaded, and dynamic language. This matches well with our requirements for a simulator. Also Java is

substantially easier to program because of the lack of several of C's "features" that are commonly abused or misused, such as pointers, preprocessing directives, "goto"s and lots of other "features" commonly found in the obscured C context entries. In addition, Java offers multi-platform binaries, a graphical API, and much easier debugging than most object-oriented languages. The tradeoff is that we do lose a substantial amount of performance, but we believe that the ease of programming (which significantly reduces development time) offsets this problem.

As a first version of simulation framework, aBlocks was successful but it also has a lot of shortcomings.

Although the object-oriented language Java was used to implement aBlocks, the design didn't follow OO design rules. The object concept is obscure in aBlocks. The relationship among objects is even more confusing. Each object involves a give() method, which is used to "call" other object. One object is triggered when it got a call and then it continuously triggers other objects by issuing a new call. Give() method linked all objects working into an inseparable chain. That greatly violates the data separation principle in object oriented design and eventually makes the reuse of code impossible.

aBlocks was designed to be a trace-driven simulator. It has few details in most of the blocks. The deficiency in details makes it hard to evaluate lots of architecture features, such as branch prediction miss, instruction cache miss, etc. To implement a microarchitecture involves many blocks working under various assumptions. This also departs from the primary goal of the simulator.

A new simulation framework – "bBlocks" was proposed and developed by this study to solve the above problems. bBlocks is not simply upgraded from aBlocks. It uses different solutions from design to implementation although it keeps the primary goals unchanged.

1.3 Objective

The objective of this study is to establish a simulation framework to be used in variety of microarchitecture analysis and performance evaluation. The proposed framework will be flexible allowing rapid microarchitecture prototyping and quick modifications to represent various architecture studies by advocating heavy object reusability.

Execution-based simulation engines are developed to give the architecture researchers sufficient details to analyze their architecture structure, verify their theory, find the bottleneck and evaluate the performance. The simulation engine need to be encapsulated by a graphic user interface, which has probes penetrating into each simulator unit.

The overall speed of the simulator is of course important, but is considered of less importance to the flexibility, speed of prototyping, and even to detail, in this study. That is speed was sacrificed for flexibility and detail.

From the standpoint of users, multi-platform capabilities are considered important. This allow users to port simulators to different hardware platforms as

new and faster machines are made available. Similarly the capabilities of supporting multithreading and parallel computing to reduce overall simulation time are also desirable, although they are also beyond the scope of this study.

1.4 Scope Of This Study

This thesis focuses on implementing an open-ended simulation toolkit, which exploits the benefit of object-oriented technologies for promoting reusability, flexibility and integrity, factors that are crucial for ensuring software quality and productivity. Due to time constraint, we define the following limits to this study.

- This study is to propose a feasible modeling methodology together with a generic microarchitecture prototyping library. But accomplishing and optimizing this library may need years of work. It's out of the range of this study.
- This study is to provide a simulation toolkit, instead of doing research on different architectures. Thus, simulation result is collected just for illustrating and verifying the correctness and efficiency of our simulator.
- This study is a good attempt of using Object Oriented approach in an application. It's not our goals to perform OO methodology research.

- The simulation framework developed in Java is theoretically platform independent. However, it is not our responsibility to do the maintenance and troubleshooting of different platforms.
- The SimpleScalar ISA developed by Todd Austin and Doug Berger [7] is chosen as the default instruction set. It has all the necessary features of a modern ISA, without any strange "leftover" bits that are in most of today's ISAs (register windows in SPARC, variable length instructions in x86, strange floating point coprocessor math in MIPS and x86) [10]. It also has an impressive tool based on gcc to produce SimpleScalar binaries, and a simulator capable of producing traces. All the implementation is based on SimpleScalar ISA, although we provide users a possibility to put in a new ISA by simply adding a new ISA interpreter.

This thesis is organized according to the bBlock software architecture, which has three levels of abstraction: block concept, general modeling library and simulation engine. Chapter 2 discusses the first level abstraction — block abstraction, starting from object-oriented concept. Chapter 3 describes the general modeling library and its basic abstraction models. Chapter 4 demonstrates how to build a simulation engine with the existing models and uses SuperScalar and CDF as examples. Chapter 5 discusses simulation result. Chapter 6 concludes the work.

Chapter 2 MODELING CONCEPT

2.1 Object Oriented Technology

Object-oriented technology emerged in the mid- to late 1980s as businesses began to seriously consider object-oriented programming languages for developing systems. Even though Simula is credited as being the first object-oriented language, popular object-oriented languages such as Smalltalk, C++, Objective C, and Eiffel came into their own in the 1980s [11]. All of these object-oriented languages approach programming from a significantly different paradigm than previous programming languages. Rather than follow the structured, deterministic, and sequential programming paradigm associated with languages such as COBOL, Fortran, C, Basic and others, these languages follow the approach pioneered by Simula based on object, attributes, responsibilities.

The Java language, as the latest development in commercial object-oriented technology, has combined many of the good features from all previous ones [12]. For example, while C++ was just a programming language, Java, with its large standard libraries, is a complete object-oriented system, just as Smalltalk is. However, compared to Smalltalk, Java programming is less interactive and more rigid, and Java libraries give better support for the creation of distributed and multi process systems. It also has a much larger and greatly increasing user community and considerable commercial push.

There are several key characteristics forming the fundamental building blocks for Object Oriented technology: 1) abstraction, 2) data encapsulation, 3) inheritance, 4) polymorphism and 5) reusability of code. The following paragraphs briefly describe these characteristics according to R. J. Norman [11].

- **Abstraction** is the principle of ignoring those aspects of a problem domain that are not the current purpose in order to concentrate more fully on those that are. It has to do with the amount of detail you care to get involved in. In systems analysis and design, this is called levels of abstraction. A problem can have several levels abstraction. Each level concentrates on a given level details.
- **Data Encapsulation** is the notion that a software component should isolate or hide single design decision. It requires the problem domain to be decomposed into small encapsulated-units. Encapsulation helps to localize the volatility when changes and maintenance are required. With the object-oriented methodologies, encapsulation incorporated both functions and data together into objects. It also provides special protection to data that belong to the object or a set of objects. This means an object can prevent some other unrelated part of the program from accidentally modifying it or from incorrectly using the private parts of the object [13]. The private parts can only be changed or retrieved by authorized functions.
- **Polymorphism** is the term used to describe variables that may refer at run-time to objects of different classes.

- **Inheritance** is the property that allows one object to acquire the properties from another one. The inheritance mechanism makes it possible for one object to be an "instance" of a more general case.
- **Reusability** In OOP, objects are classified, written, created, and debugged in advance. Then they can be distributed or used by other programmers in their own programs. Further more, because of inheritance, a programmer can take an existing class and, without modifying it, add additional features and capabilities to it.

Contrast to traditional function-oriented technology, object-oriented technology describes the real world as an integration of some individual objects. It coheres the people's view of the realistic world [13]. In object oriented analysis, objects are finite data models with functions representing states of a physical object in the real world. The object perspective, rather than the functional perspective of traditional methodology, is used to approach the software problem of object-oriented programming.

2.2 Modeling Methodology

The most promising way to develop a complex software system is to break a large system up into manageable components [14]. Decomposition provides a very useful strategy to solve complex problems by dealing with the much simpler

components and then integrating them together. There are several advantages in component composition:

- **Reusability** a well-designed component can be used repeatedly, saving the redesign, reprogramming and retest time.
- **Reliability** it is safer to use a component from some other applications, which have been tested and errors removed.
- **Extendibility** applications and systems composed from software components can be extended by substituting existing components with new ones.

2.2.1 Functional decomposition

Traditionally, in function-oriented technology, a system is described by functions and data structures with data flows connecting them. The decomposition is done functionally — a high level function is broken into sub-functions that are further broken down until the bottom level is reached. The sub-functions are usually developed to fit one or several functional needs. They are called in some designated places to perform certain previously defined processes.

To enhance the ability to compose and reuse software, the experienced designers classify components by the components collaborative behavior. Similarity of components is based on the principle that two similar components can be substituted without the rest of the software in the system observing any difference. Additionally, the other components must keep in a similar way when

some components in the system are substituted by components with similar behavior.

Since there is no efficacious guidance, the designer's experience is highly desired here.

2.2.2 Object decomposition

The object decomposition approach suggests breaking systems down by using higher-level objects rather than higher-level functions. These higher level objects are further broken down to lower-level objects, each level objects dealing with different level of details. That is, the object of one level focuses on modeling some specific level of detail of the real world physical object's behavior. The higher-level objects then communicate and use each other in roughly the same way that the lower-level objects do.

The object is the basic component in object-oriented technology. With the characteristics of encapsulation, inheritance and polymorphism, we have good reasons to believe that object decomposition makes more sense for creating an extendible system. At present the most successful extensible systems are built using object-oriented components [15].

2.2.3 Comparison

Traditional modeling paradigm and object-oriented modeling paradigm are compared by Mize [15]. Table 1 contains a brief summary of the comparison.

Table 1. Contrasting Traditional and Object-Oriented Paradigms for Simulation Modeling

Factors	Traditional Modeling Paradigm	Object-Oriented Modeling Paradigm
Model Construction:		
Software	Based on procedural programming style	Based on object-oriented programming style
Problem description	Abstract	Natural and intuitive
Level of detail	More detail means much more complexity in code	Abstraction level is the key to solve complexity problem
Effort/time/cost	Moderate costs of model development, but a "throwaway" type	Initial cost of establishing detailed model is very high, but cost of subsequent reuse is relatively low
Model Attributes:		
Purpose	Usually a unique model is created for a specific purpose	More general models possible for multiple purposes
Usage	Single usage, throw-away models	Repeated usage and continuous refinement
Flexibility	Highly inflexible; Changes almost always result in a complete rewrite of program	Highly flexible, due to the ability to modify fundamental building blocks; Quick reconfiguration is possible

Object-oriented technology has opened a new avenue in the rapid prototyping and development of complex software. Now, more and more system designers realized the significant advantages of object-oriented technology in describing large software problem, simplifying code maintenance and providing extensibility for future enhancements. The following chapters will show how to design reusable and maintainable components and give more efficient to system development and maintenance.

2.3 First Level Abstraction

This section focuses on the first level abstraction of microprocessor architecture. Block and strategy are principle concepts used to describe the microarchitecture being modeled.

There are many substantially different modeling approaches to fulfill the decomposition task. However, to achieve the primary goals — getting the most flexibility and reusability, the modeling abstraction need to be carefully considered. The central idea in making components reusable is to involve the stable behavior inside it and exclude the variable features outside it. Another useful idea is that features tending to change at the same time should be grouped together. After inspecting several microarchitectures, we refined two basic abstractions, block and strategy, to model the system. The following will describe these two abstractions in detail.

2.3.1 Block abstraction

Block is the key concept in bBlocks. It inherits the idea of "creating each significant structure in the processor as an object" [8] from aBlocks. It is the basic unit to encapsulate the data structure and simulation activities, also the basic reusable and extendible component to construct the simulator, as the brick to the building. Figure 2 shows the basic block diagram in bBlocks.

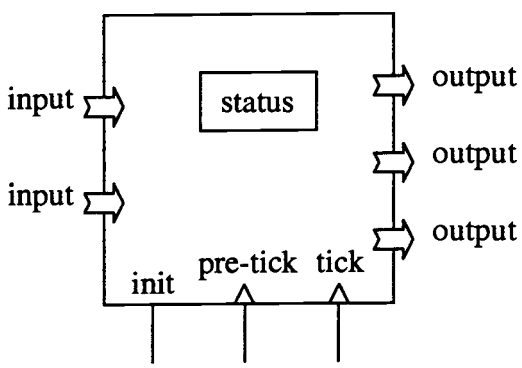

Figure 2. Basic Block

The basic block is defined as a module which, having some kinds of standard inputs and outputs, with the ability to memorize its own status, can do some pre-designed operations to the inputs and get the output when it is triggered. It potentially corresponds to the microprocessor hardware modules. It has two synchronized behavior: pre-tick() and tick(). In pre-tick() period, a block communicates with other blocks and prepares the input into the input buffer. In

tick() period, a block manipulates the input and gets the output ready into the output buffer. A block also has some standard information probe engines, such as toString(), report() and status(), which provide methods to dump the current status of itself. The basic block is created as an abstract class in bBlocks (shown in Figure 3). Any other simulator blocks are extended from it.

```
public abstract class Block implements Testable{

      private String name;
      private Simulator owner;
      private StatCalculator statistic; …

      public Block( String name, Simulator owner ){
            this.name = name;
            this.owner = owner;
      }

      public abstract boolean preTick() throws SimException;

      public abstract boolean tick() throws SimException;

      public String toString(){
            return "I am "+ this.getClass().getName()+" "+ name +"\n";
      }

      public String report(){
            return "I am "+ this.getClass().getName()+" "+ name +"\n";
      }

      public String status(){
            return "I am "+ this.getClass().getName()+" "+ name +"\n";
      }
}
```

Figure 3. Block Abstraction

So far, in bBlocks library, some essential blocks are provided, including MemoryUnit, PreFetch, Decoder, Instruction Window, Reorder Buffer, Instruction Pipe, etc. In this way, the basic microprocessor activities were decomposed and encapsulated into blocks.

2.3.2 Specific strategy

It is the fact that every simulator is constructed under some specific conditions and assumptions. There is no exception for bBlocks. These conditions and assumptions may vary in different architecture research. For instance, bBlocks uses SimpleScalar ISA as defaulted ISA. The executive binary depends on SimpleScalar's `gcc` compiler. That's the specific mechanism bBlocks can not avoid even as a generic modeling library. Strategy is used here to group and represent those features liable to change. The goal we are seeking is to give it the most possibility to involve different strategies or to change from one to another.

Strategy is not a hardware-associated object. Instead, it is more reasonable to think it as a protocol or rule the hardware (block) need to follow. The block access the protocol by declaring a standard interface associated to the strategy provider. The separated strategy provider is the place to implement whole strategy with detail. Therefore, the block keeps being a generic one, not relying on any specific strategies. In the above example, a specific strategy called SimpleScalar ISA interpreter is provided to implement all the methods concern to SimpleScalar

ISA. The blocks using this strategy only need to declare an ISA function interface in their blocks. The simulation engine is responsible for associating the ISA function interface to SimpleScalar ISA interpreter.

When a new ISA is needed, the only thing you need to do is to provide a new ISA interpreter in correct directory and associate your simulation engine to your new interpreter. (That can be done in a definition file.) The polymorphism characteristic of object-oriented technology offers that benefit.

2.4 Decomposition

The "clock" plays an important role in modern RISC microprocessor. Most of the hardware units are synchronized by clock. That means the operation of those hardware units happen simultaneously, synchronous with the clock signal. And further more, lots of factors, likes IPC (Instructions Per Cycle), which we are trying to evaluate with our simulator, is relatively concerned with clock cycles. Just like the hardware, bBlocks uses "clock" to drive and synchronize the block, which has two synchronized phase, pre-tick() and tick(). The former, communicating phase, gets inputs from other blocks and puts them into input buffer. The later, data processing phase, operates input data, changes current status and puts the outputs into the output buffer. The input buffer and output buffer are not only separations between blocks but also used for synchronizing clock activities.

Now the block concept can be updated. It is a software model to imitate the behavior of a piece of hardware unit which identifies itself by lying between two synchronized buffers and using clock to synchronize its behavior. For example, in five-stage MIPS architecture, five blocks are deduced accordingly. Those are IF, ID, EX, MEM and WB. With same theory, SuperScalar architecture is decomposed to PreFetch, Decoder, Instruction Window, Execution Unit (EU, BEU, MEU), Reorder Buffer, Register File and Memory Unit. CDF architecture has most of the components from SuperScalar except Instruction Window. It has Instruction Pipe instead. The following chapter reveals the detail of each block.

Although in the software simulator, blocks are triggered sequentially for pre-tick() and tick(), it is obvious that the concurrent concept is implicitly involved in the design of bBlocks. Block, which has input and output buffer to isolate itself from outward and uses clock signal to synchronize the behavior, is the basic concurrent component in bBlocks.

Chapter 3 MODELING LIBRARY

This chapter deals with the second level of abstraction — the generic modeling library. In brief, this second level of abstraction is using block and strategy concept to model the microarchitecture. Some reusable components are constructed, which create the stem of a generic modeling library.

3.1 Notation

The following notations are used in diagrams showing architecture details:

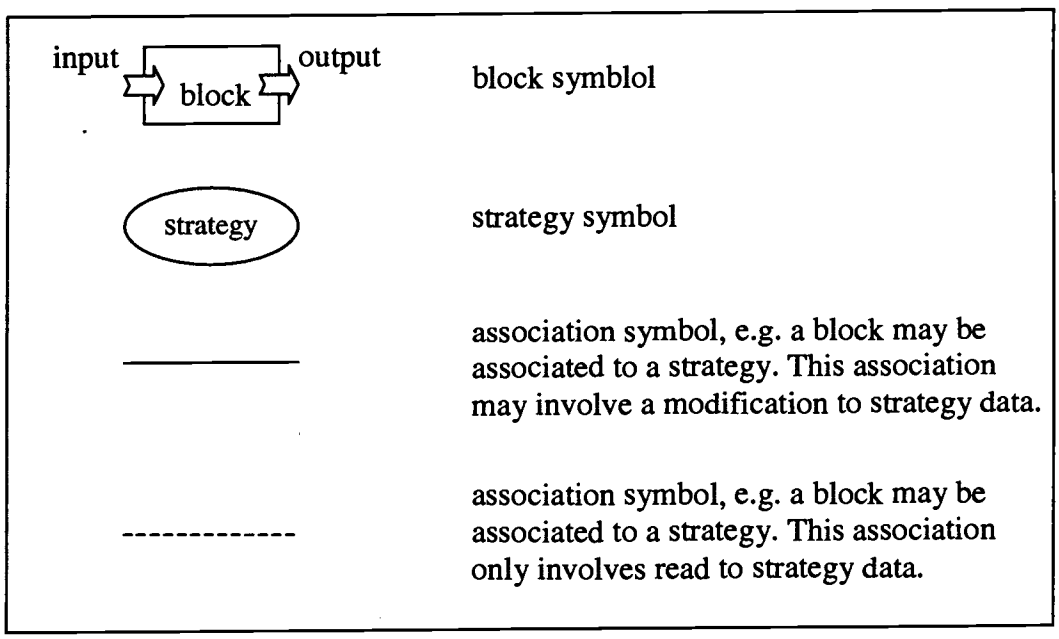

Figure 4. Diagram Notation

3.2 PreFetch

PreFetch generates instruction-loading requests and parses the loaded instructions for next simulation stage. The instruction loading address is determined according to current program counter (PC) and branch prediction information. The branch prediction information is provided by a branch predictor strategy, which is separated from PreFetch block and ready to change. Now, bBlocks only involves a very simple branch predictor strategy. It uses a hash table to record the recent branch history. The instruction is considered to be a branch, if its address has a match in this hash table. Register File strategy is connected to PreFetch for supplying program counter (PC) information.

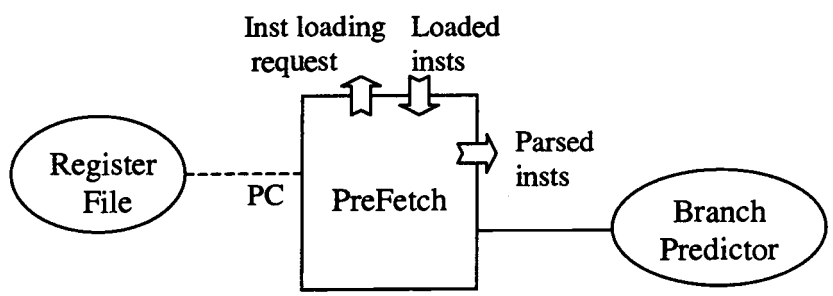

Figure 5. PreFetch Block

3.3 Decoder

Decoder translates original instruction into instruction token, which carries all the necessary information for execution. A separate ISA interpreter provides the translation strategy. Register File provides the register value for decoder. Register renaming also happens inside the Decoder.

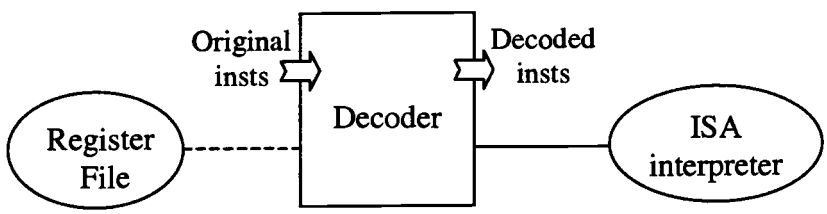

Figure 6. Decoder Block

3.4 Instruction Window

Instructions wait in Instruction Window till they are ready for execution. Then they are issued to available execution units. It can issue multiple instructions out of the original program order. Once the instruction is executed, the result is forwarded to Instruction Window to resolve the instruction dependency. Instruction Window also guarantees every instruction in it gets an entry reserved in Reorder Buffer.

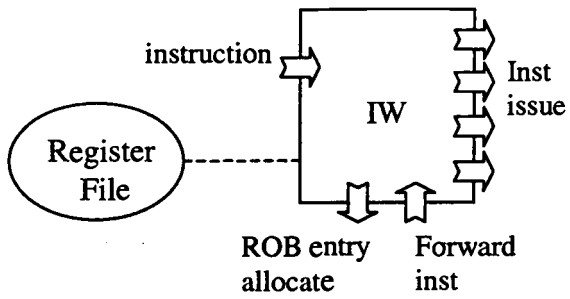

Figure 7. Instruction Block

3.5 Execution Unit

Execution Unit is the place where instructions are executed. There are two extension blocks derived from Execution Unit: BEU and MEU. BEU generates branch interrupt if it finds a branch prediction miss. MEU handles all the memory access. It has an interface with memory hierarchy.

Figure 8. Execution Unit

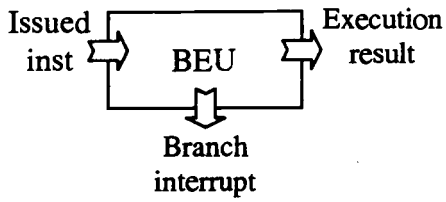

Figure 9. BEU Block

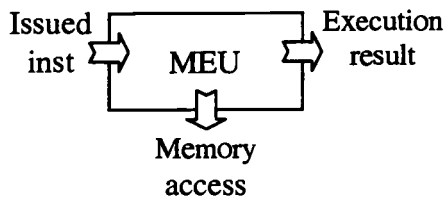

Figure 10. MEU Block

3.6 Reorder Buffer

Reorder Buffer reserves an entry for every instruction entered into Instruction Window. This entry is used to store its execution result. The execution results of the instructions are retired in order after they are made ready.

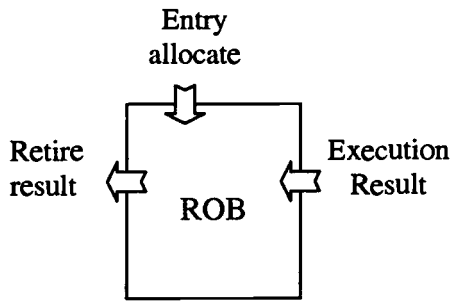

Figure 11. ROB

3.7 Register File (RF)

RF accepts retired execution results and modifies the register values accordingly. The register values are saved in the register file, which is another strategy dependent on ISA.

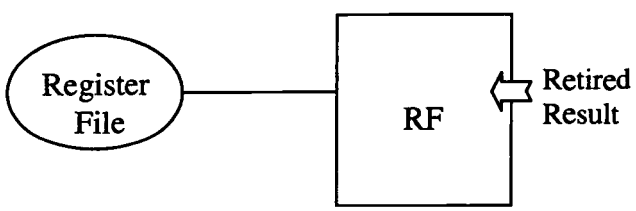

Figure 12. RF

3.8 Memory Unit

This is the generic memory/cache object. It can act as any level of cache, with any size (but all 'dimensions' must be in powers of 2) and any latency. It checks the input from lower level cache. If it hits in the cache set table, result is returned to lower level output, otherwise it is sent to next level. The data returned from high level is passed to lower level and loaded into the cache set table at the mean time.

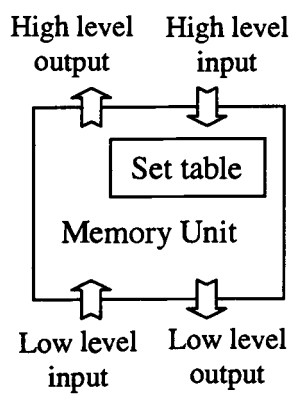

Figure 13. Memory Unit

3.9 Strategies

So far, in bBlocks, several strategies are provided. Table 2 lists the strategies.

Table 2. Strategies

Strategy	Description
SimpleScalar ISA Interpreter	Implement all mechanisms associate to ISA
Brach Predictor	Implement branch prediction strategy
Register File	Implement registers structure
Memory File	Implement simulation virtual memory

Chapter 4 SIMULATION ENGINE

In the object modeling framework described above, the microarchitecture is divided into individual blocks such as PreFetch, Decoder, Instruction Window, Reorder Buffer, Register File and Memory Unit. Each of these blocks represents a physical object in the real world. This chapter will describe how to integrate these models into a simulation engine — the third level of abstraction.

4.1 Pack Blocks Into Simulation Engine

All features discussed above are architecture independent. But to construct a simulation engine, the architecture dependent features can not be avoided.

First, to create a simulation engine, connectors are needed to link the blocks together. Connector also serves as an adapter to transfer data type when two interfaces with different data types are linked. Connectors are architecture dependent features. For Example, in SuperScalar, Execution Units are connected to Instruction Window and in CDF, they are connected to Instruction Pipe. For this reason, connectors are separated from original blocks and implemented in the simulation engine level to protect the original blocks to be a generic one.

ABlocks used a fixed interface (aToken) to connect blocks. It offers aBlocks a very simple connector, but at the same time brings some problems. aToken is the most confusing part in aBlocks. The over generalized structure needs

a lot of assumption to make it understandable. It greatly hurts the data encapsulate theory. bBlocks avoids it by choosing several well designed interface types. For instance, in bBlocks, the "CacheLine" type serves for the interfaces between memory units, while "InstructionToken" type for interfaces of the blocks after decoder and "Result" type for interfaces of the blocks after execution units. Here is a trade off between the complexity of connectors and the functionality of interface types.

Figure 14 uses SuperScalar simulation engine as an example. A set of "ssBlock" is derived from the original blocks in the generic modeling library. Besides the characteristics inherited from original blocks, ssBlocks implement the connectors. The ssBlocks are called application-specific blocks. These blocks may add in any application related features or use a specific feature to substitute the generic one.

At this point, all the components for constructing a simulation engine are available. A simulation manager is used to assemble the existing components actively into an architecture simulation engine. The simulation manager is responsible for setting up simulation components, parsing parameters to each component and triggering each component's activity during its lifetime. Figure 15 shows part of SuperScalar simulation manager as an example.

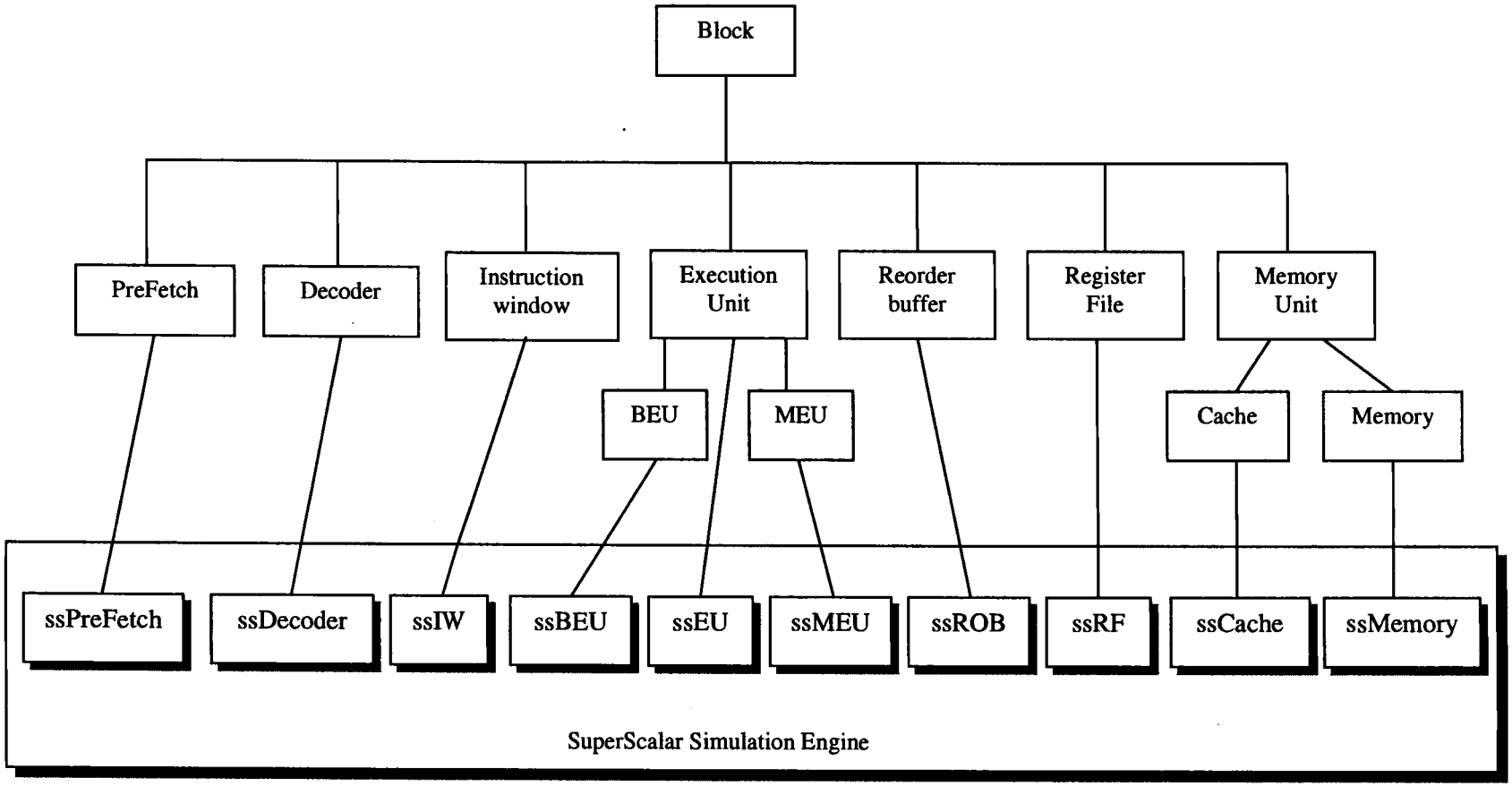

Figure 14. Simulation Framework

```
public class SuperScalar extends Simulator {

     SsPreFetch preFetch;
     SsDecoder  decoder;
     SsIW       iw;
      …

      public void setup(){
             preFetch   = new SsPreFetch(FetchParam1, FetchParam2 …);
             decoder    = new SsDecoder(decodeParam1, decodeParam2 …);
             iw         = new SsIW(IWParam1, IWParam2 …);
             …

             preFetch.connectTo(cache1);
             decoder.connectTo(preFetch, getRegFile());
             iw.connectTo(decoder, forwardProvider);
             …

             addBlock(preFetch);
             addBlock(decoder);
             addBlock(iw);
             …
      }

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

             sim.run();
      }
}
```

Figure 15. SuperScalar Simulation Manager

In bBlocks, the users are granted the privilege to setup the parameters for hardware configuration in a definition file. The simulation manager reads in the parameters when it is started.

Therefore, using the block models in simulation library and adding some connectors, it's easy to construct a new architecture simulation engine. Like your toy bricks, with all the bricks (block models), using some well-designed

connectors, you can construct whatever (architecture) you like. To demonstrate the efficiency of above analysis, two simulation engines — SuperScalar and CDF, are implemented.

4.2 SuperScalar Simulation Engine

SuperScalar refers to microprocessor architectures that enable more than one instruction to be executed per clock cycle, and may internally reorder instruction execution. Nearly all of modern microprocessors, including the Pentium, PowerPC, Alpha, and SPARC microprocessors are SuperScalar.

The SuperScalar simulation engine is modeled after the centralized SuperScalar execution model [16], shown in Figure 16. Figure 17 shows the diagram of SuperScalar simulation engine.

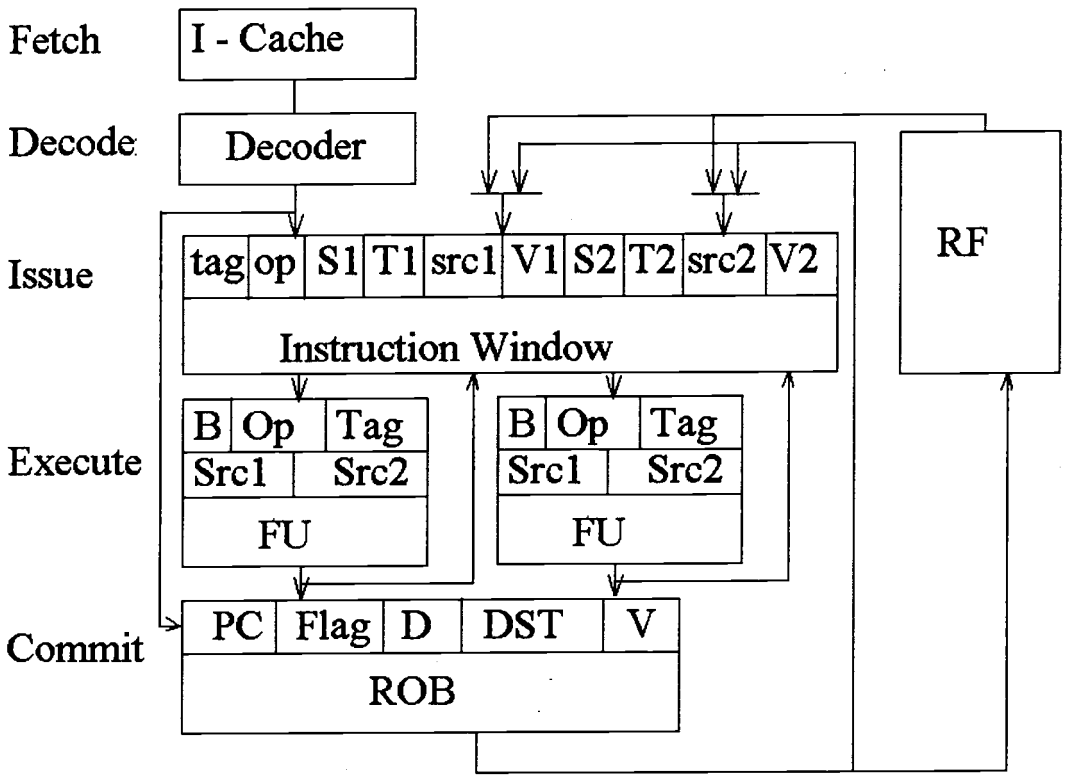

Figure 16. Basic SuperScalar Execution Model (Centralized)

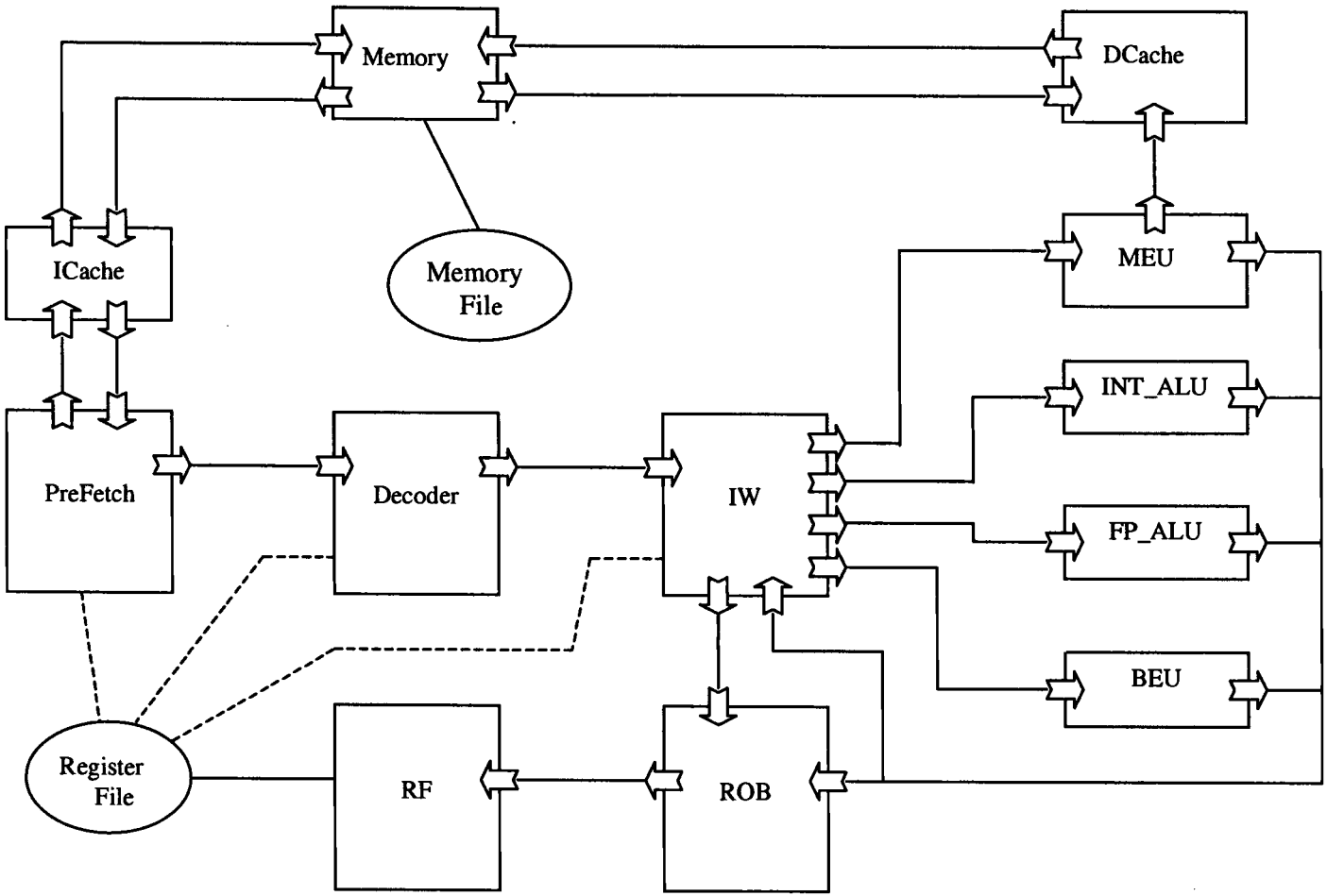

Figure 17. SuperScalar Simulation Engine

4.3 CDF Simulation Engine

The CounterDataFlow (CDF), described by Janik [10], is an improved architecture from original CFPP (counterflow pipeline processor) suggested by Spoull [1]. Referring to Figure 18, there are two pipelines, the instruction pipeline and the result pipeline. The instruction pipeline carries instructions from ROB up toward the top of the instruction pipeline. If an instruction gets to the end of the pipeline and hasn't been executed, it simply wraps around to the beginning of the pipeline and continues up the pipeline. Along the way, instructions and results interact and inspect each 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. When the instruction arrives at the execution unit's recovery point, it takes the result from the execution unit if the execution has completed. As the instruction continues up the instruction pipeline, it looks for empty spots in the result pipeline in which to put its results. The result pipeline carries results down to the ROB. When it gets to ROB, the result could be written to ROB and exit the pipeline, or be forced to wrap around for performance reasons.

In CDF simulation engine, about 85% codes, which include all the architecture models except Instruction Pipe and ROB, are reused from the existing modeling library and SuperScalar simulation engine. The fact that it only takes

about one week to generate the CDF simulation engine demonstrates the efficiency of the simulator design. The OOM feature in bBlocks allows the users assemble their models with much fewer redundant efforts, which are definitely needed if models are developed from scratch.

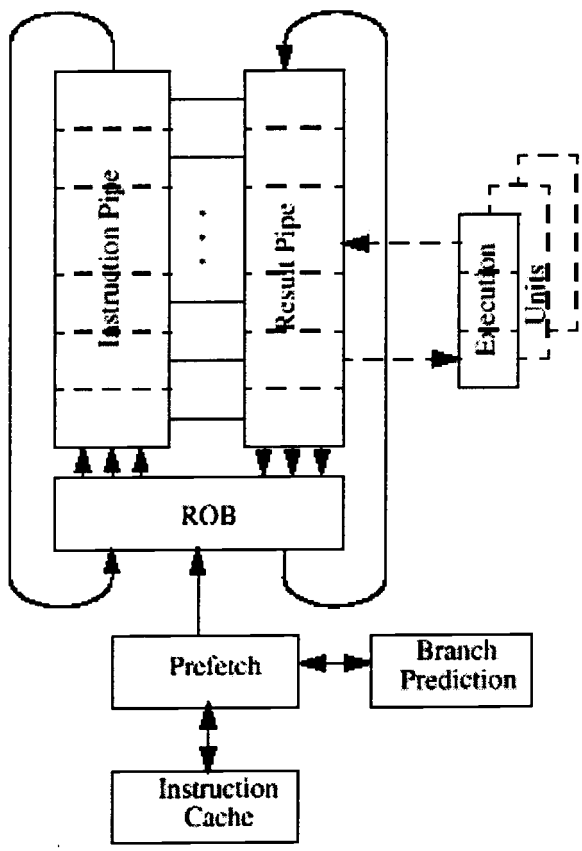

Figure 18. CDF Architecture

Chapter 5 SIMULATION RESULT

To verify the correctness of simulators, lots of test programs were loaded in and executed. A set of simulation result was collected for analysis.

5.1 Functional Verification

As an execution based simulator, correctly running program is the first test feature. Both SuperScalar and CDF simulation engine was fully tested and validated for functional correctness by some gradually advanced steps:

- Test by simple programs. We wrote some simple C programs for the first step verification.
- Test by SimpleScalar test programs. SimpleScalar toolkit has a set of precompiled binaries that give more thoroughly test for different types of operations including integer, floating point, long integer, short integer and characters.
- Test by SPEC95 benchmark. SPEC95 is a worldwide standard for measuring and comparing computer performance across different hardware platforms.

Our confidence comes from fully passing all the tests. All the outputs are correct.

5.2 Statistic Verification

Statistic correctness comes after functional correctness for a simulation tool. To inspect the statistics, simulation results got from SuperScalar architecture (shown in Figure 16) are compared with the results got from SimpleScalar "sim-outorder". SimpleScalar's "sim-outorder" can be used as a SuperScalar simulator. With similar configuration parameter, the two simulator's results should be comparable. Due to time constraint, seven benchmarks from SPEC95 are selected for performance comparison and one million instructions are executed for each benchmark.

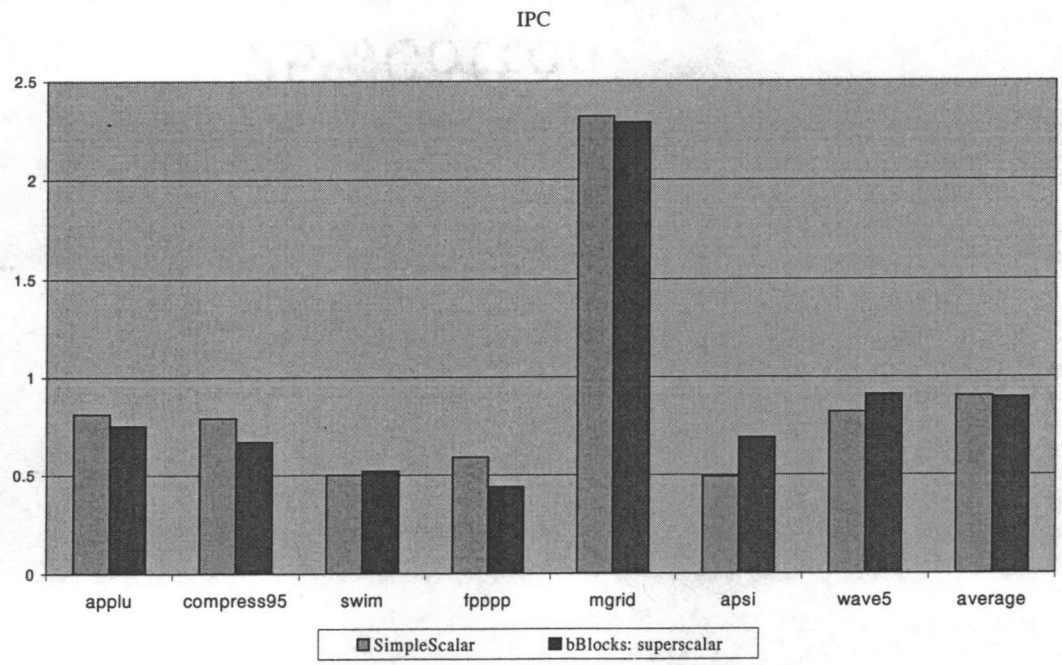

Figure 19. Instruction Per Cycle

Figure 19 shows the comparison in average IPC (Instruction Per Clock cycle) for the first 1 million instructions of each benchmark. From the above figure, the results of these two simulators are very close. Actually the average IPC is almost same.

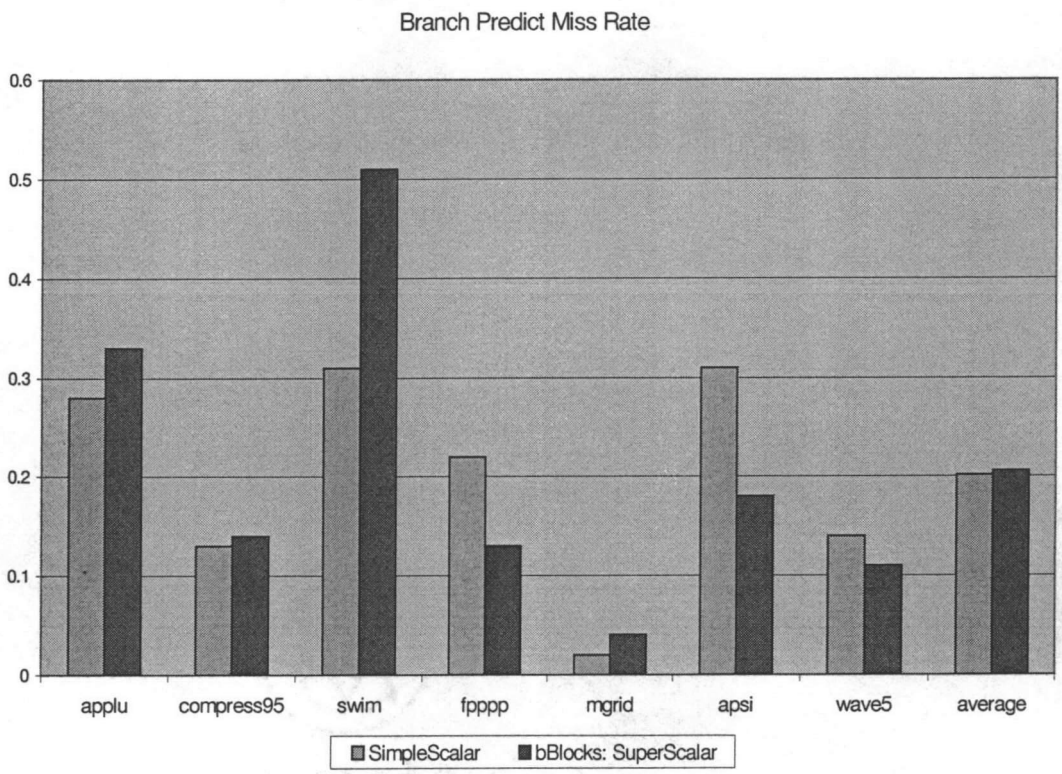

Figure 20. Branch Prediction Miss Rate

Now, bBlocks only has a very simple branch prediction strategy. It uses a hash table to record the recent branch history. The instruction is considered to be a branch instruction, if its address has a match in this hash table. It is found similar to the combined predictor in SimpleScalar toolkit, although the later is much complicate. Figure 20 compares the branch prediction miss rates got from those two simulations. The average branch prediction miss rates are very close.

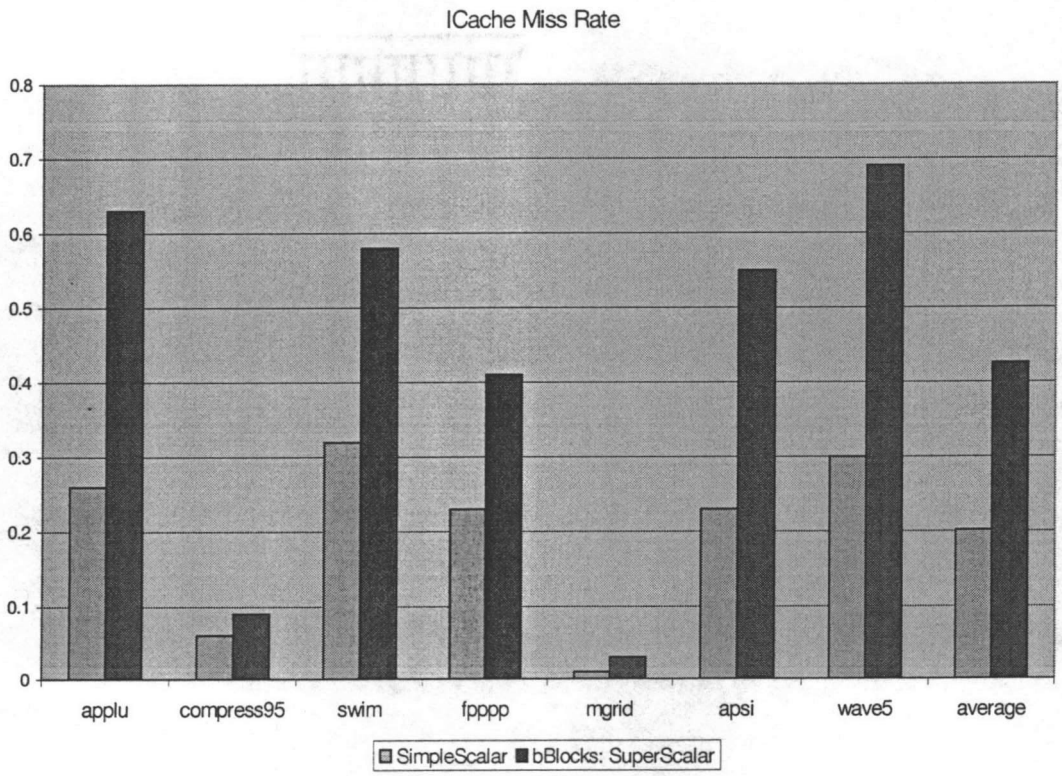

Figure 21. Instruction Cache Miss Rate

Instruction cache miss rate is an incomparable feature for these two simulators. The reason is that the instruction cache in bBlocks is a non-blocking cache, whereas the instruction cache in SimpleScalar "blocks on an I-cache miss until the miss completes" [7]. Non-blocking cache means it still can accept following cache access after it gets a cache miss (obviously, that introduces additional miss possibility). Figure 21 shows the difference in instruction cache miss rate. It is reasonable that the instruction cache miss rate of bBlocks is always larger than that of SimpleScalar.

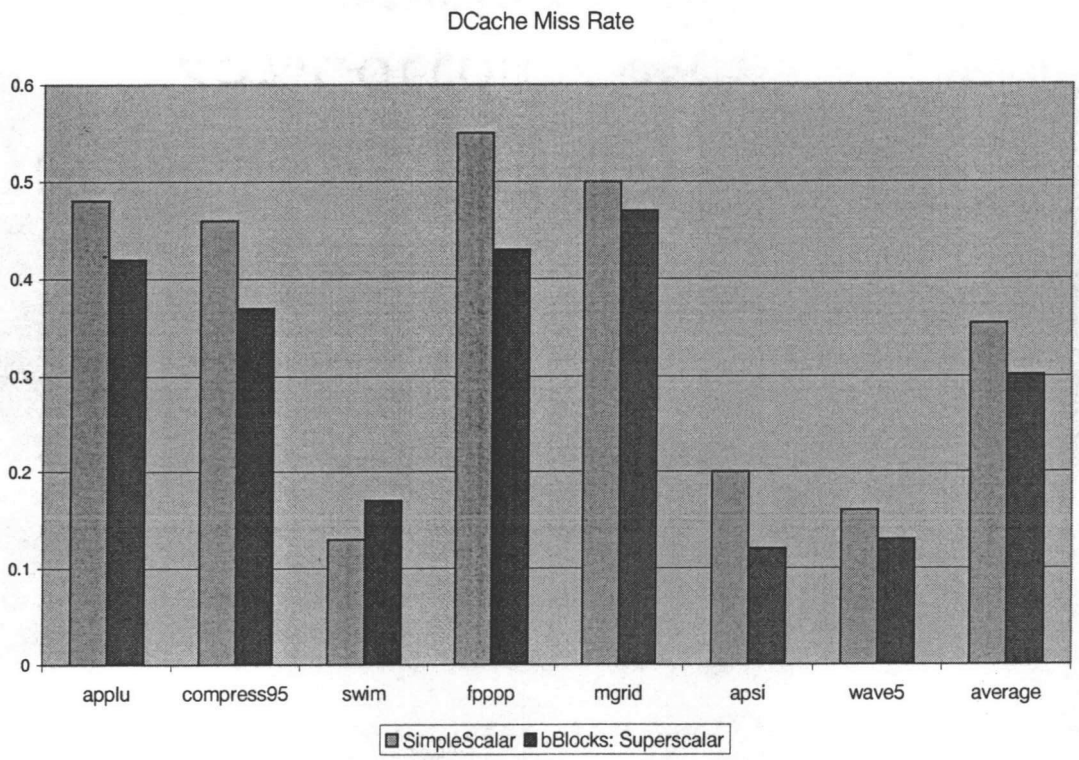

Figure 22. Data Cache Miss Rate

Data cache miss rates are compared in Figure 22. The results from those two simulators have a rational difference.

5.3 Additional Discussion

It is admitted that bBlocks with object-oriented approach design is much slower than SimpleScalar with traditional design (implement with C). From the above experiment, the average execution time of bBlocks is 120000 seconds, and the average execution time of SimpleScalar is 39 seconds. That is bBlocks is 3000 times slower than SimpleScalar.

Another fact is that, CDF simulation engine is generated in a very short time (about one week) based on SuperScalar engine. About 85% codes of CDF simulation engine are reused from SuperScalar engine. CDF is new microprocessor architecture proposed by CFPP group at Oregon State University. There is no execution-based simulation for CDF before bBlocks. From our experience, it takes months or even years of work to build a new microprocessor simulator. Therefore, the great benefit of bBlocks is manifested. bBlocks favors flexibility and reusability at the expense of speed.

bBlocks software structure is shown in Table 3. Table 4 lists the configuration parameter used in above experiment.

Table 3. Software Structure

Level	Directory name	Program lines	Explanation
1	sim	1681	Definition of block,simulator,etc
2	block	4486	Blocks like prefetch,decoder,etc
	supplement	8196	Strategies and data type definition
	util	1470	Utility used by whole project
3	superscalar	1261	Specific blocks for SuperScalar
	cdf	2188	Specific blocks for CDF
Total		19282	

Table 4. SuperScalar Configuration Parameter

Block name	Parameters
PreFetch	width=4
I-cache 1	sets=8,lineLength=32,assoc=2,latency=1,width=4,LRU, writeback
I-cache 2	sets=8,lineLength=128,assoc=2,latency=1,width=4,LRU ,writeback
D-cache1	sets=8,lineLength=16,assoc=2,latency=1,width=4,LRU, writeback
D-cache2	sets=8,lineLength=64,assoc=2,latency=1,width=4,LRU, writeback
Memory	pages=8,pageLength=256,latency=1,width=4
Decoder	width=4
IW	size=32
ROB	size=32,width=4
FastIntAlu * 3	latency=1
SlowIntAlu	latency=4
FastFPAlu	latency=4
SlowFPAlu	latency=8
BEU	latency=1
MEU	latency=1
RF	width=4

Chapter 6 CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

Modeling real world systems with classes and components is the preferred modeling technique for new generation microprocessor simulation software. The advantages of using OOP and OOM is significant in achieving code reusability, easing development times, improving portability and avoiding obsolescence. In this research, a generic modeling library, bBlocks, was constructed from scratch allowing microarchitecture analysts to quickly assemble new prototyping models. Although it is not fully accomplished, the library has already demonstrated some outstanding qualities.

- Correctness. Because the simulation package is execution based, the correctly running programs itself gives us more confidence for the functional correctness. In fact, during the implementing of CDF, we found a great mistake lying in aBlock simulator. As a trace-based simulation, it is hard for aBlocks to identify the error.
- Reliability. The fact that result got from our SuperScalar simulation is very similar to that got from SimpleScalar simulation announces the reliability.
- Reusability. This is our primary goal. And we do achieve it by successfully using the OOP and OOM methodology in simulation design and developing. The generic modeling library is the essential part advocating large amount code

reuse. From this library, any new architecture is readily to be generated. Actually, our second simulation engine for CDF (a recently proposed architecture, no existing simulator for it), was generated in a very short term.

- User friendliness. bBlocks provides a graphic user interface, which let users dynamically probe into any block and get the detail. With it, the user can trace every instruction loaded into the simulation engine. It did give us a great help in program debugging. It will show its advantage in helping architecture validating and bottleneck detecting.
- Portability. Theoretically, application developed with Java will run on any computer that has a Java virtual machine.

From this study, Java, although not perfect, is still a good choice for implementing microarchitecture simulation libraries. As an object-oriented programming language, Java offers several advantages over C++ such as a true cross-platform characteristic, it is network ready, has many easy to use features and has good GUI ability. The library developed using Java offers clear opportunities of portability and offers an unlimited audience through the Internet. All the good features can compensate the only drawback, the sacrifice of computational speed. Nowadays, the improvement of computer hardware, Java compiler techniques and distributed computing also negative this drawback. Java multi-thread provides a great potentiality to enhance the speed and capacity of the simulation.

6.2 Recommendations For Further Study

6.2.1 Accomplish the modeling library

Due to time limit, the modeling library is not fully finished. For example, a complex branch prediction strategy is necessary for branch study. The "syscall" instruction is not completely decoded now. The Memory Execution Unit needs to handle "instruction load pass store". New microprocessor models may be required for future study. Numerous works need to be done to fulfill a comprehensive simulation package.

6.2.2 Optimize the performance

Java virtual machine offers the platform-independent ability at the expense of runtime performance. The simulation speed is major problem. Sun's HotSpot technology, a just-in-time compiler, provides a notable improvement. But there is still a big gap, which can be filled by code optimization. Another choice is to use Java multi-thread.

REFERENCES

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

2. K. J. Janik and S. Lu, "Synchronous Implementation of a Counterflow Pipeline Processor," *Proceedings of the 1996 International Symposium on Circuits and Systems*, pp. 69-72, May 1996.

3. K. J. Janik and S. Lu and M. F. Miller, "Advances to the Counterflow Pipeline Microarchitecture," *High-Performance Computer Architecture – 3*, pp. 230-236, February 1997.

4. M. F. Miller and K. J. Janik and S. Lu, "Non-stalling Counterflow Architecture," *High-Performance Computer Architecture – 4*, pp. 334-341, February 1998.

5. J. Huang and D. J. Lilja, "An Efficient Strategy for Developing a Simulator for a Novel Concurrent Multithreaded Processor Architecture," *IEEE Design and Test of Computers*, pp. 185-191, Vol.2, 1998.

6. T. A. Diep and J. P. Shen, "VMW: A Visualization-based Microarchitecture Workbench," *Computer*, pp. 57-64, Vol. 28 12, Dec. 1995.

7. D. Burger and T. Austin and S. Bennett, "Evaluating Future Microprocessors: The Simple Scalar Tool Set," *http://www.cs.wisc.edu/~mscalar/simplescalar.html.*

8. M. F. Miller and K. J. Janik and S. Lu, "aBlocks Users Manual and Reference Guides".

9. G. Cornell and C. S. Horstmann, *Core JAVA*, Sun Microsystems, Second Edition, 1997.

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

11. R. J. Norman, *Object-Oriented Systems Analysis and Design,* Prentice-Hall, 1998.

REFERENCES (CONTINUED)

12. J. Waldo, "Tutorial Java: A Language for Software Engineering," Proceedings of the 1997 International Conference on Software Engineering, pp. 630-630, 1997.

13. H. Schildt, *C++: The Complete Reference,* Osborne McGraw-Hill, 1991.

14. R. C. Covintton, "The Rice Parallel Processing Testbed," *In ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 4-11, 1998.

15. J. H. Mize and H. C. Bhuskute and D. B. Pratt and M. Kamath, "Modeling of integrated manufacturing systems using an object-oriented approach," *IIE Transactions,* v. 24, n. 3, p. 14-26, July 1992.

16. J. L. Hennessy and D. A. Patterson, *Computer Organization and Design,* Morgan Kaufmann, Second Edition, 1998.