重用 object fields 以加速 Java 處理器之執行效率

學生：林恩申　　　　指導教授：單智君　博士

國立交通大學資訊工程學系碩士班

摘要

在 Java 處理器執行 Java 程式時，需要很多的執行時間和計憶體的存取來完成 object field access 的動作。在此篇論文中，我們提出對於 object field access 的加速方法是利用一硬體的緩衝區，稱作 object cache，來儲存先前已經被存取過的 object field 的值。這種 object cache 不但能加速 object field access 的動作，並且支援 garbage collection，保證在發生 garbage collection 時，依然可由 object cache 拿到正確的資料。藉由透過 Trace-driven 的方式對 SPECjvm98 標竿程式進行模擬，結果顯示在昇陽的 picoJava-II 處理器中加入我們所提出的 object cache，將可使 object field access 的速度提升 3.3 倍。

Boosting Java Processor Performance by Reusing Object Fields

Student：En-Shen Lin　　　Advisor：Dr. Jean, J.J. Shann

Institute of Computer Science and Information Engineering
National Chiao-Tung University

ABSTRACT

The operations of object field accesses in the Java processor require many clock cycles and memory references. Our approach of improving the performance of object field accesses is to cache the values of object fields of previous execution in hardware buffers, called object cache. In this thesis, object cache that can reduce the object field access time and support garbage collection, is proposed. A detailed trace-driven simulation of the proposed method on SPECjvm98 benchmark show that our proposed method can achieve 3.3 speedup over Sun's picoJava-II on object field accesses.

Acknowledgment

首先感謝我的指導老師 單智君教授，在他的諄諄教誨、辛勤指導與勉勵下，而得以順利完成此論文。同時也感謝本論文所屬計畫老師、同時也是我的口試委員的鍾崇斌教授，以及另一位口試委員馬瑞良博士，由於他們的建議，使得這篇論文能更加完整。

感謝 Java 研究群的唐立人學長、張隆昌學長、喬偉豪同學、與周明俊、吳志宏兩位學弟，尤其是唐立人學長，這篇論文在他的指導與提供寶貴的意見下，才能更加完整。也感謝實驗室的學長們不厭其煩地與我討論許多問題，給予了我莫大的幫助。此外，感謝諸位同學與學弟們，你們的陪伴讓我的生活充滿歡樂；也讓這些年的研究生活更加多采多姿與充實。

感謝陪伴我走過這段時間的每一個人。讓我在這研究的路上走的更順利，進而能更無後顧之憂的從容學習，使我能堅持追求自己的理想。

所有支持我、勉勵我的師長與親友，奉上我最誠摯的感謝與祝福，謝謝你們。

林恩申

2000.6.25

Contents

摘要..I

ABSTRACT.. II

ACKNOWLEDGMENT III

CONTENTS..IV

LIST OF FIGURES VI

CHAPTER 1 INTRODUCTION..1

CHAPTER 2 BACKGROUND AND RELATED WORK 4

2.1 JAVA TECHNOLOGIES...5

2.1.1 Java Bytecode Manipulation Methods.....................................5

2.1.2 Java Class File Organization..6

Constant Pool and Class Descriptor...8

Fields, Methods, and Attributes 9

2.2 ARCHITECTURE OF JAVA VIRTUAL MACHINE 10

2.2.1 Method Area 14

2.2.2 Heap 14

CHAPTER 3 DESIGN AND SIMULATION OF OBJECT CACHE.... 18

3.1 OBJECT-FIELD ACCESS BEHAVIOR 19

3.1.1 An Example of Object-Field Access...20

3.1.2 Execution Flow of Object-Field Access 22

Constant Pool Resolution...23

3.1.3 Mechanism of rewriting Java bytecode by SUN 24

3.2 DESIGN OF OBJECT CACHE 26

3.2.1 Benchmark Behavior Analysis 27

Simulation Approach 27

Temporal Locality and Reusing Probability 29

Acceleration Approach — Object Cache...30

3.2.2 Design Issues 31
Issue I: Indexing Policy 32
Issue II: Pipeline Stage Design 34
Issue III: Cache Line Size and Cache Size 35
3.3 PERFORMANCE COMPARISON 36
CHAPTER 4 CONCLUSIONS AND FUTURE RESEARCH 38
REFERENCE 40

List of Figures

Fig. 2-1: Linear, record-based organization of a Java class file....................................7
Fig. 2-2: The internal architecture of the Java virtual machine 11
Fig. 2-3: Runtime data areas that are shared among all threads 12
Fig. 2-4: Runtime data areas that are exclusive to each thread...................................13
Fig. 2-5: Splitting an object across a handle pool and an object pool 16
Fig. 2-6: Keeping object data in one place 17
Fig. 3-1: The dynamic instruction mix of SPECjvm98 benchmark.............................19
Fig. 3-2: The format of opcode "*getfield*" and "*putfield*" and the changes of the operand stack before and after the execution of the bytecode.............................20
Fig. 3-3: An example of object-field access 21
Fig. 3-4: Execution flow of object field access for *getfield*..23
Fig. 3-5: Execution flow of object field access for *getfield_quick* 26
Fig. 3-6: Simulation approach...28
Fig. 3-7: Probability of an object field that to be reused in the next *n* object field accesses 29
Fig. 3-8: Hit rate of the LRU buffer with *m* entries 30
Fig. 3-9: Using an object cache to access object field data directly 31
Fig. 3-10: Range of the offsets for each benchmark 33
Fig. 3-11: Hit rates of the object cache under different indexing schemes and numbers of indexing bits. Assume that the object cache is direct-mapped with 8-word line size. 34
Fig. 3-12: Six pipeline stages of Sun's PicoJava-II 35
Fig. 3-13: Pipeline stage diagram 35
Fig. 3-14: Hit rates of the object cache with 8-word line size and index scheme 2 under different numbers of entries..36
Fig. 3-15: Speedup over Sun's picoJava-II for object field accesses by using a direct-mapped object cache with an 8-word line size, index scheme 2, and 1K entries...37

Chapter 1
Introduction

Java has become the most popular language to develop network programs. Its suitability for networked environments is inherent in its architecture, secure, platform-independent programs and run on a great variety of computers and devices [1,2]. A Java program is compiled to the class files of an abstract virtual machine , called bytecodes, to achieve its platform-independent feature. As an interpreted language, its disadvantage is slow performance. Thus, there are some solutions to enhance Java's execution performance such as Just-In-Time compilers and Java processors. A Just-In-Time (JIT) compiler translates Java bytecode dynamically to

native machine code to get partial speedup. But it needs more memory space to store the translated machine code. Java processor executes Java bytecode directly as its native code without interpreting [2,3,4]. With the appearance of Java processor, we have the new concept to be considered and studied.

Because a Java processor executes Java bytecode as its native code, to understand the features of Java language is important when we design it. One of Java's features is its cross-platform compatibility. A Java program is compiled to class file format. These class files may be loaded and executed by Java virtual machine (JVM) interpreters of various platforms without re-compilation, i.e., compiles once and runs everywhere. A Java processor is just the implementation of the abstract Java virtual machine. Another feature is its object-oriented programming model of Java. Java is an object-oriented language and it does not allow programmers to use pointers to access memory locations directly. Using object-oriented programming model advances the readability and maintainability of source code. The astriction of using pointers avoids the unpredictable errors of C programs and enhances the security.

Object-oriented programming model makes programmers maintain their source code easily, but with the penalty of more execution time of programs. We call those instructions used to access object data "object manipulation instructions". This kind of instructions is executed frequently and usually cost many clock cycles. We found that object field access instructions usually reference to the same entity. Therefore, we may accelerate the object access by recording and reusing some useful information. Sun's picoJava-II has proposed a rewriting method to accelerate the constant pool resolution and made it more simple to access object field [5]. Thus, we propose a hardware support to enhance the object access performance of a Java processor.

The purpose of this thesis is to accelerate the access of Java object fields. Our approach is to use a buffer, called object cache, to store the value of object fields. We

use an object virtual address instead of a physical memory address to index the object cache. This method will reduce unnecessary memory accesses and, thus, enhance performance.

The organization of this thesis is as follows. In Chapter 2, the background and related work are presented. In Chapter 3, the object field access behavior of the Java applications are investigated. Based on this study, a hardware acceleration mechanism of object field access is proposed. And then, the performance of the proposed acceleration mechanism is evaluated. Finally, conclusions and future research are presented in Chapter 4.

Chapter 2
Background and Related work

In this chapter, we describe the related background and researches of Java. First, Java technologies are discussed in detail, including several Java bytecode manipulation methods and Java class file organization. And then, the architecture of Java virtual machine is presented.

2.1 Java Technologies

Programs of traditional programming languages have only one form, a running program. Whereas Java programs come with two flavors: a stand-alone program to run as a separate unit or an applet to run from the Internet browser. The life cycle of a traditional language program is very simple. A programmer writes a program, which may consist of a number of modules but are all linked at compile time. The compiler then converts the program to the underlying machine assembly language. As far Java program modules, each consisting of one or more classes, they are compiled independently to Java Virtual Machine bytecode. At this stage, these modules which are called class files can be exchanged and transferred around the network. Users load the module into an implementation of a the JVM. JVM may then load additional ".class" files as needed, from the user or across the Internet. Only at this point, references between different modules are resolved. And a dynamic linking step performed by a linker before the user gets starting the program.

2.1.1 Java Bytecode Manipulation Methods

Java bytecodes in their way to run take one of three methods: interpreter, Just-In-Time (JIT) compiler, and Java processor. These methods connect the virtual machine to the actual machine, where Java software can run.

A Java interpreter, like a translator, can convert Java bytecodes on-the-fly (at run-time) into native codes. The interpreter must process the same code over and over again while a Java program is running. Interpretation is simple and does not require

much memory. It is relatively easy to be implemented on any processor. However, it involves a time-consuming loop to translate every Java bytecode, and, thus, affects performance signitificantly.

A Java Just-In-Time (JIT) compiler, like an interpreter, translates Java bytecodes into native code but it does not have to translate the same code over and over again as it cache the native code. This can result in significant speedup. However, sometimes a JIT compiler takes a large number of time to do its job and results in code size expansion and consuming more memory.

A Java processor natively understands Java bytecode without the overhead of an interpreter or a JIT compiler. We can take advantage of high performance by running Java programs on Java processors.

2.1.2 Java Class File Organization

Like any compiler, the Java compiler takes the source code of a program and translates it into machine code and binary symbolic information. In a traditional system, these data will be stored in an object file for later use or execution. In Java case, they are placed into a separate ".class" file for each Java class or interface in the source code.

The Java class file is a precisely defined binary file for Java programs. Each Java class file represents a complete description of one Java class or interface. There is no way to put more than one class or interface into a single class file. The precise definition for the format of the class file ensures that any Java class file can be loaded and correctly interpreted by any Java virtual machine, no matter which system produced the class file or which system hosts the virtual machine.

The Java class file is a binary stream of 8-bit bytes. Data items are stored sequentially in the class file, with no padding between adjacent items. The lack of padding helps to keep class files compact. Items that occupy more than one byte are split into several consecutive bytes that appear in big-endian order. The class files follow a rigid five-part format as shown in Figure 2-1. Each class file begins with a magic number and version information, followed by a constant pool, a class descriptor header, fields, methods, and finally an extension area. Because of Java's dynamically linked nature, each class file must contain a large amount of symbolic and typing information. This data informs the JVM about how to resolve internal and external class references, and also allows it to verify the security and integrity of classes.

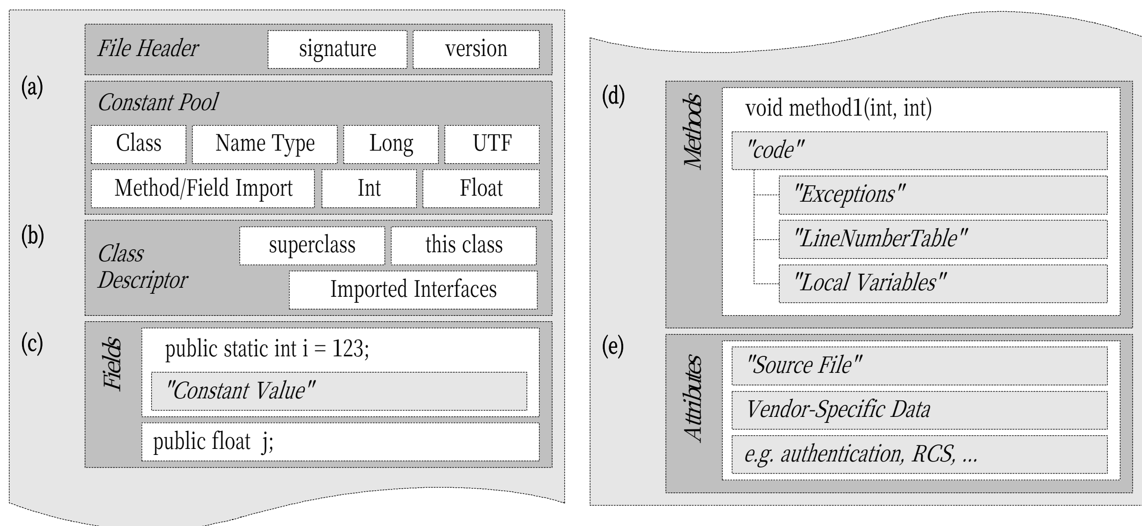


Fig. 2-1: Linear, record-based organization of a Java class file

Constant Pool and Class Descriptor

The constant pool of a class file is similar to the symbol table in a traditional object file; see Figure 2-1(a). The data with constant pool is referenced primarily by other structures and code within the class file, and thus contains a wealth of additional information beyond the usual symbol names. The pool is treated as an one-dimensional array of slots each containing one variable-length data type called a tag. The most common constant pool tags are strings. Strings are stored in the UTF-8 format in Unicode characters which are packed into bytes to save space.

The constant pool also integrates the aspects of traditional import/export and relocation tables. There are a number of special linking tags, which simply contain the indices of other pool slots. These tags (such as CLASS, METHOD, and NAMETYPE) are used to dynamically link Java classes. For example, a METHOD tag points to a CLASS tag (to specify an imported class), as well as a NAMETYPE tag (to identify a specific method in that class). Linking tags are also directly referenced by the bytecode of the class as a dynamic relocation table.

Following the constant pool, the class descriptor consists of several fields related to the entire class; see Figure 2-1(b). These fields include the access flags of the class (public, private, and so on), as well as constant pool indexes to the class and its superclass. An array of constant pool indexes to any interfaces implemented by the class also appears here.

Fields, Methods, and Attributes

Following the class descriptor, there are two arrays that describe fields (Figure 2-1(c)) and methods (Figure 2-1(d)). Both arrays have an identical structure, but they describe different types of class members. Each variable length entry identifies the access flags, name, and signature of the member, as well as a list of associated "attributes".

An attribute is a basic component of the class format, and is merely a special type of record that provides additional information in a more flexible format. For instance, each method descriptor contains a nested Code attribute that fully describes the actual bytecode for that method. Similarly, a field descriptor may contain a *ConstantValue* attribute, which points to a constant pool entry that describes a "static" constant in a class. In addition, several attributes are optional and are related to debugging. You can include your own attributes in class files to extend the class format without breaking existing code or Java Virtual Machine.

The Code attribute is especially important because it contains the actual Java bytecode (along with stack and local variable information). It can also contain nested attributes. For example, it can nest an *Exceptions* attribute (to list any exceptions thrown by the method owning the Code attribute) as well as several debug attributes, such as *LineNumberTable*, *LocalVariables*, and *SourceFile*.

At the end of the class file (Figure 2-1(e)) is a separate section for other attributes that apply to the class as a whole. The *SourceFile* attribute is placed here by the Java compiler, and vendors are free to put additional attributes in this section as well. For instance, the *Attributes* section is a good place to put class authentication or security information, or perhaps revision control system data.

2.2 Architecture of Java Virtual Machine

In the Java virtual machine specification, the behavior of a virtual machine instance is described in terms of subsysytems, memory areas, and instructions. These components describe an abstract inner architecture for the abstract Java virtual machine. The purpose of these components is not so much to dictate an inner architecture for implementations but to provide a way to strictly define the external behavior of implementations. The specification defines the required behavior of any Java virtual machine implementation in terms of these abstract components and their interactions.

Figure 2-2 shows a block diagram of the Java virtual machine that includes the major subsystems and memory areas described in the specifications. Each Java virtual machine has a class loader subsystem, which is a mechanism for loading types (classes and interfaces) when given fully qualified names. Each Java virtual machine also has an execution engine, which is a mechanism responsible for executing the instructions contained in the methods of loaded classes. When a Java virtual machine runs a program, it needs memory to store many items—including bytecodes and other information that it extracts from loaded class files, objects that the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several runtime data areas.

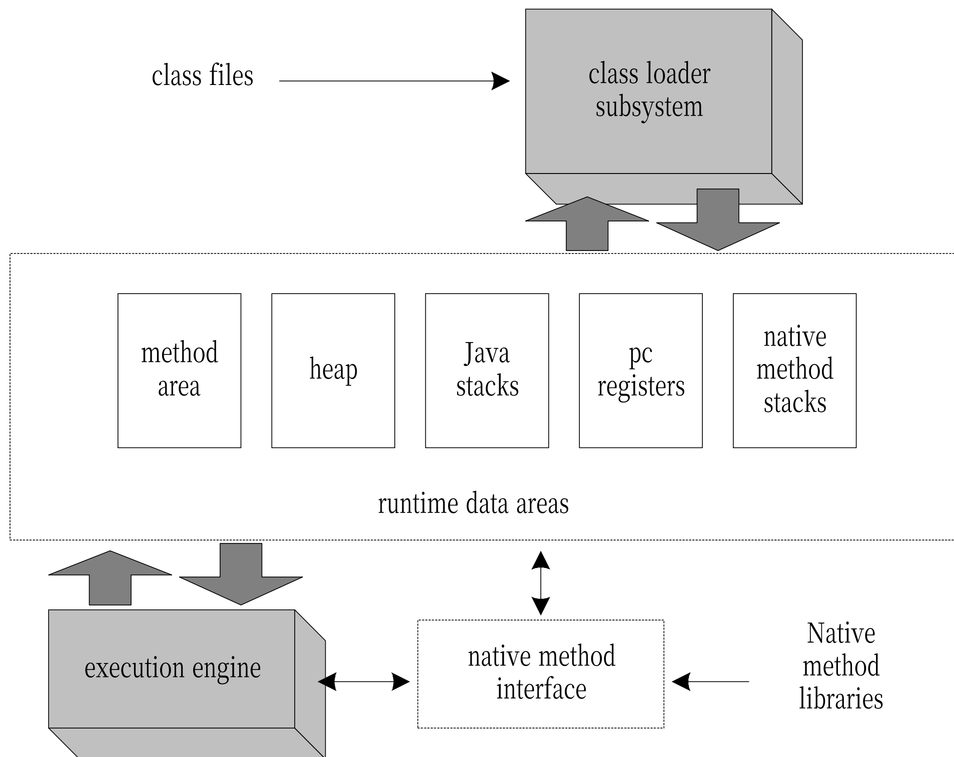


Fig. 2-2: The internal architecture of the Java virtual machine

Some runtime data areas are shared among all threads of an application, and others are unique to individual threads. Each instance of the Java virtual machine has one method area and one heap. These areas are shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses the information about a type from the binary data contained in the class file, then places this type information into the method area. As the program runs, the virtual machine places all objects that the program instantiates onto the heap. Figure 2-3 shows a graphical depiction of these memory areas. More details about the memory area and the heap are described in subsection 2.2.1 and 2.2.2.

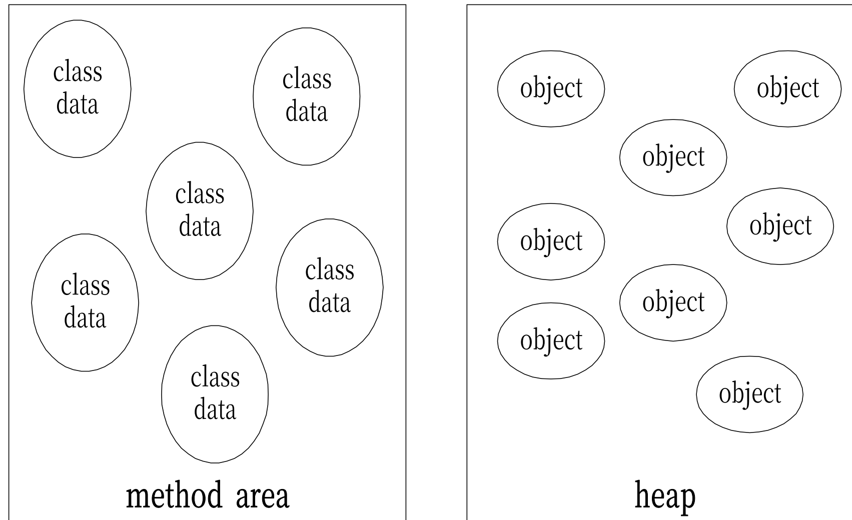


Fig. 2-3: Runtime data areas that are shared among all threads

As each new thread comes into existence, it receives its own PC register (program counter) and Java stack. If the thread is executing a Java method (not a native method), the value of the PC register tells the next instruction to execute. The Java stack of a thread stores the state of Java method invocations (not native invocations) for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation- dependent way in native method stacks, as well as possibly in registers or other implementation-dependent memory areas.

The Java stack is composed of stack frames (or frames), which contain the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto the Java stack of a thread. When the method completes, the virtual machine pops and discards the frame for that method. The Java virtual machine has no registers to hold intermediate data values. The instruction set uses the Java stack for storage of intermediate data values. This approach was taken by Java designers to keep the JVM instruction set compact and to facilitate

implementation on architectures with few or irregular general-purpose registers. In addition, the stack-based architecture of the JVM instruction set facilitates the code optimization work done by just-in-time and dynamic compilers that operate at run time in some virtual machine implementations.

Figure 2-4 shows the memory areas that the Java virtual machine creates for each thread. These areas are private to the owning thread, and no thread can access the PC register or Java stack of another thread. In the figure, threads one and two are executing Java methods, while thread three is executing a native method.

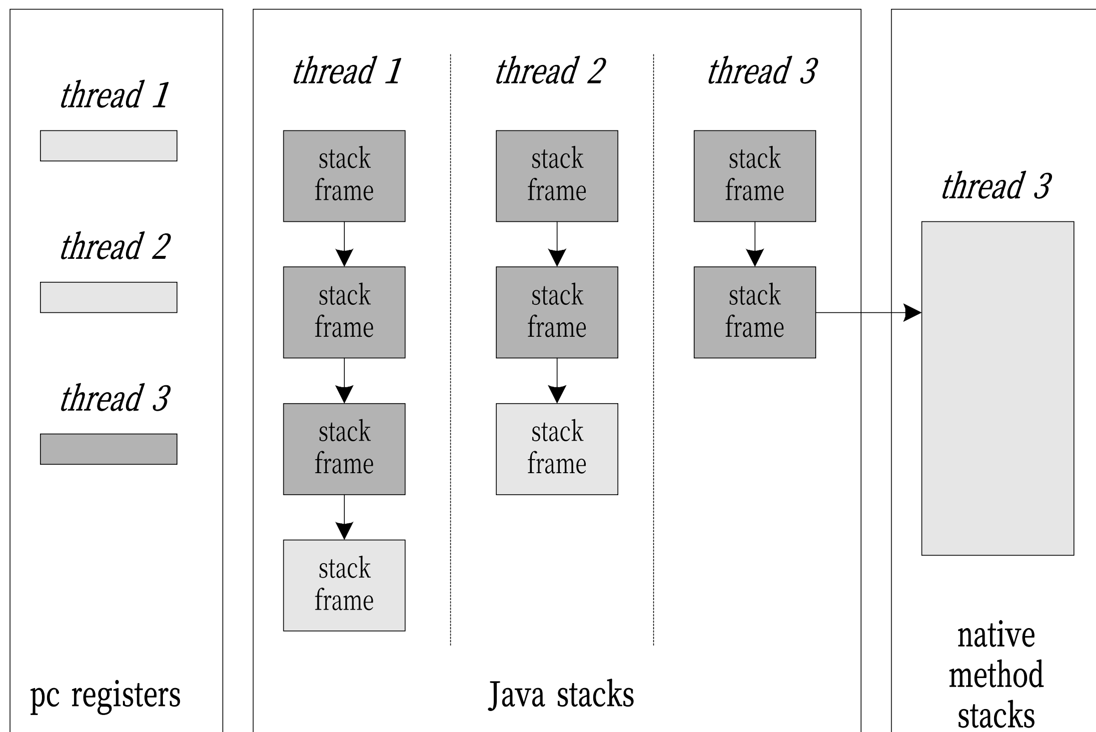


Fig. 2-4: Runtime data areas that are exclusive to each thread

2.2.1 Method Area

Inside a Java virtual machine instance, information about loaded types is stored in a logical area of memory called the method area. When the Java virtual machine loads a type, it uses a class loader to locate the appropriate class file. The class loader reads the class file—a linear stream of binary data—and passes it to the virtual machine. The virtual machine then extracts information about the type from the binary data and stores the information in the method area. Memory for class (static) variables declared in the class is also taken from the method area. All threads share the same method area, so accessing the data structure of the method area must be designed to be threadsafe. The class data that store in method include type information, constant pool, field, method information, class variables, and method tables.

2.2.2 Heap

Whenever a class instance or array is created in a running Java application, the memory for the new object is allocated from a single heap. Because there is only one heap inside a Java virtual machine instance, all threads share the heap. Since a Java application runs inside its own exclusive Java virtual machine instance, there is a separate heap for every individual running application. Two different Java applications can not access each other's heap data. Two different threads of the same application could access each other's heap data. For this reason, we must concern about the proper synchronization of multi-threaded access to objects (heap data) in

Java programs.

The Java virtual machine has an instruction that allocates memory on the heap for a new object but has no instruction for freeing that memory. Just as you can not explicitly free an object in Java source code, you can not explicitly free an object in Java bytecodes. The virtual machine itself is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application. Usually, a Java virtual machine implementation uses a garbage collector to manage the heap.

In Java virtual machine, there is no specification in regard to how objects should be represented on the heap. Object representation—an integral aspect of the overall design of the heap and garbage collector—is a decision left to implementation designers. The instance variables declared in the object's class and all of its superclasses make up the primary data that must be represented for each object. Given an object reference, the virtual machine must have the capability to quickly locate the instance data for the object. In addition, there must be some way to access the object's class data (stored in the method area) when given a reference to the object. For this reason, the memory allocated for an object usually includes some kind of pointer to the method area.

One possible heap design divides the heap into two parts: a handle pool and an object pool. An object reference is a native pointer to a handle pool entry. A handle pool entry has two components: a pointer to instance data in the object pool, and a pointer to class data in the method area. The advantage of this scheme is that the virtual machine can easily combat heap fragmentation. When the virtual machine moves an object in the object pool, it only needs to update one pointer with the object's new address: the related pointer in the handle pool. The disadvantage of this approach is that every point of access to an object's instance data requires

dereferencing two pointers. This approach to object representation is shown in Figure 2-5.

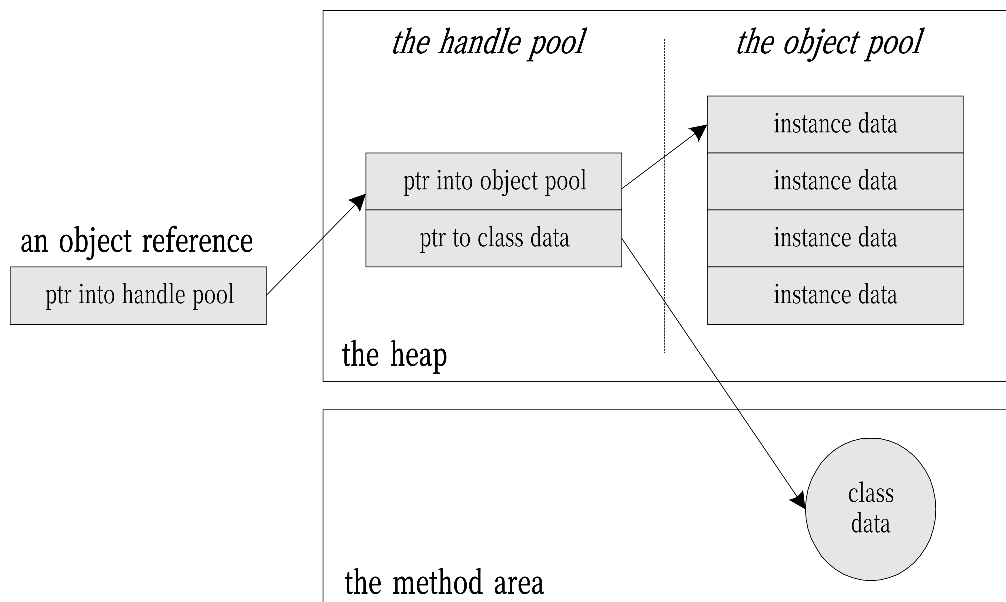


Fig. 2-5: Splitting an object across a handle pool and an object pool

Another design of heap makes an object reference a native pointer to a bundle of data that contains the object's instance data and a pointer to the object's class data. This approach requires dereferencing only one pointer to access an object's instance data but makes moving objects more complicated. When the virtual machine moves an object to combat fragmentation of this kind of heap, it must update every reference to that object anywhere in the runtime data areas. This approach to object representation is shown in Figure 2-6.

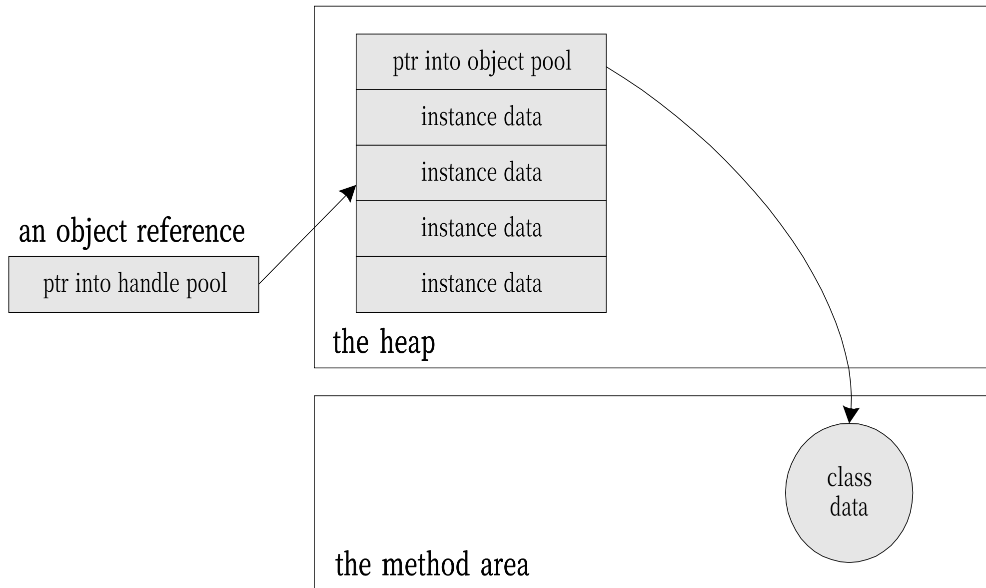


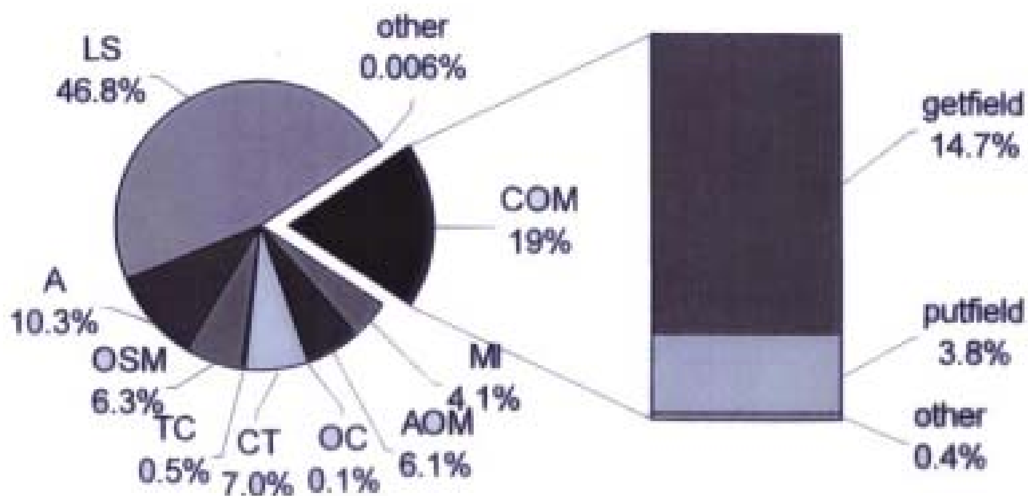
Fig. 2-6: Keeping object data in one place

Chapter 3
Design and Simulation of Object Cache

In this chapter, the analysis of the object-field access behaviors is presented. First, an example and execution flow of the object-field access are presented. And then, an optimization method of rewriting bytecode proposed by Sun is shown. Next, the benchmark behavior analysis is given. Base on this result, a new acceleration mechanism for object-field access is proposed. Then, the issues that affect our proposed mechanism and the operations of the mechanism are presented. Finally, the performance of the proposed accelerated mechanism is evaluated.

3.1 Object-Field Access Behavior

Java is an object-oriented language. One of its important feature is the data encapsulation. The data and methods of a sturcture are encapsulation into a class. We have to access object data or method through object manipulation instructions. Traditionally, this kind of instructions are performed by traps and always cost lots of cycles to execute. Figure 3-1 shows the dynamic instruction mix of SPECjvm98 benchmark [6]. We find that class object manipulation (COM) insturctions constitute 19% of total instruction counts. Therein, opcode “getfidle” and “putfield” constitute most of this kind of instructions. In this section, we explain the detail execution flow of object-field access instructions ,especially getfield and putfield, and declare that we want to accelerate the speed of these two instructions.



LS: load and store
A: arithmetic
OSM: operand stack management
TC: type conversion
CT: control transfer
OC: object creation
AOM: array object manipulation
MI: method invocation
COM: class object manipulation

Fig. 3-1: The dynamic instruction mix of SPECjvm98 benchmark

3.1.1 An Example of Object-Field Access

Figure 3-2 shows the formats of bytecode "*getfield*" and "*putfield*" and the changes of the operand stack before and after the execution of the bytecode. These two instructions are used to access object-field data. There are two indexbytes follow the opcode and are used to index into the constant pool. Bytecode "getfield" is used to fetch a field data from an object. Before the opcode "getfield" be executed, the object reference of the target field must be put on the top of stack (TOS). After execution, the value of target field is on top of stack. Bytecode "putfield" is used to set a field value in an object. Before the opcode "putfield" be executed, the object reference of the target fieldand the value must be put on the top of stack (TOS). After execution, the value is set in the target field.

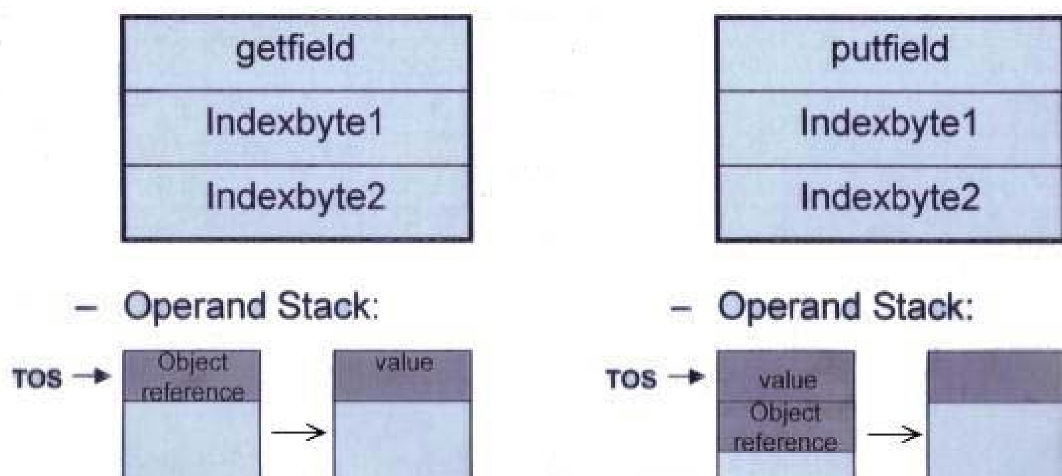
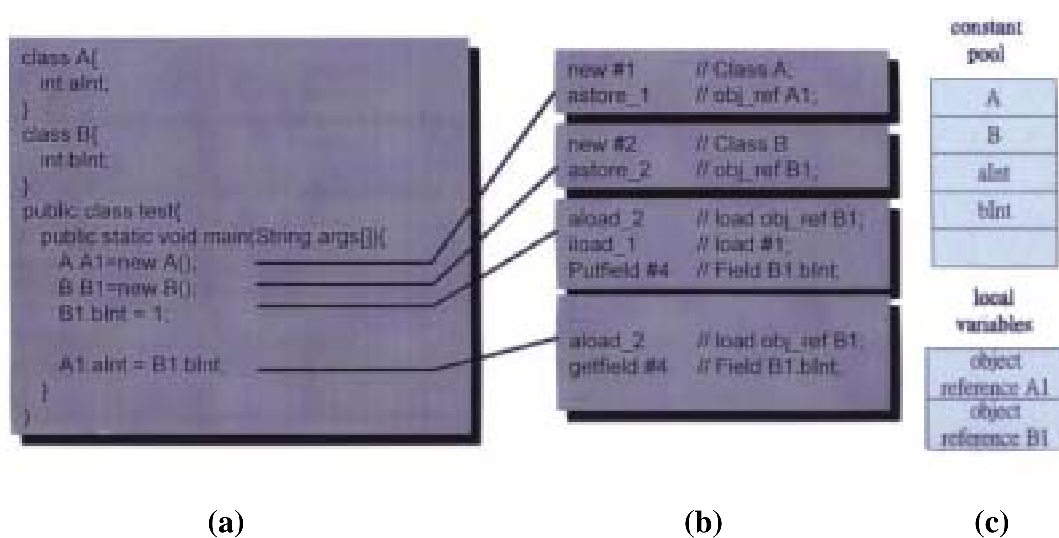


Fig. 3-2: The format of opcode "*getfield*" and "*putfield*" and the changes of the operand stack before and after the execution of the bytecode

An example of object-field access is shown in Figure 3-3. Figure 3-3(a) is the example source code. We declare two classes (class A and B), each contains one field (field aInt and bInt). We use the keyword "new" to create the object instance from a

class. In this example, A1 is an instance created from class A and B1 is an instance created from class B. Figure 3-3(b) shows the Java bytecode compiled from the source code. It will call the opcode "*new*" to create the object instance. There is one indexbyte following the opcode. This byte is used to index into the constant pool. Figure 3-3(c) shows the state of constant pool and local variables. When the bytecode "new #1" is executed, it will go to constant pool entry #1 to get the necessary information. After execution, the created object reference is on the top of stack. It will call the opcode "*astore*" to save the object reference to local variables. When we want to access object field data, it will call the opcode "*getfield*" or "*putfield*" and need to load some information from constant pool or local variables.



(a) **(b)** **(c)**

Fig. 3-3: An example of object-field access

3.1.2 Execution Flow of Object-Field Access

In the Java virtual machine, memory is allocated on the garbage-collected heap only as objects. You can not allocate memory for a primitive type on the heap, except as part of an object. On the other hand, only object references and primitive types can reside on the Java stack as local variables. Objects can never reside on the Java stack. The architectural separation of objects and primitive types in the Java virtual machine is reflected in the Java programming language, in which objects can not be declared as local variables—only object references and primitive types can. Upon declaration, an object reference refers to nothing. Only after the reference has been explicitly initialized—either with a reference to an existing object or with a call to new, the reference refer to an actual object.

When we want to access an object method or field, some sequential actions will be executed. Opcode “getfield” and “putfield” are used to get and put object fields. There are 2-byte operands called “indexbytes” followed the opcodes used to index to constant pool. Constant pool resolution is executed to find the physical memory location of the referenced field or method. It is the process of dynamically determining concrete values from the symbolic references in the constant pool. The 2-byte operand is used to index to constant pool to find offset. It may need to involve loading one or more classes or interfaces, binding several types, and initializing types. This process always cost many execution cycles. And then, it is needed to translate the object reference on the top of stack and offset to the physical memory address to access data. This process may need another memory access of handle table to get the object base memory address. The flow of the object field access for *getfield* is shown in Figure 3-4.

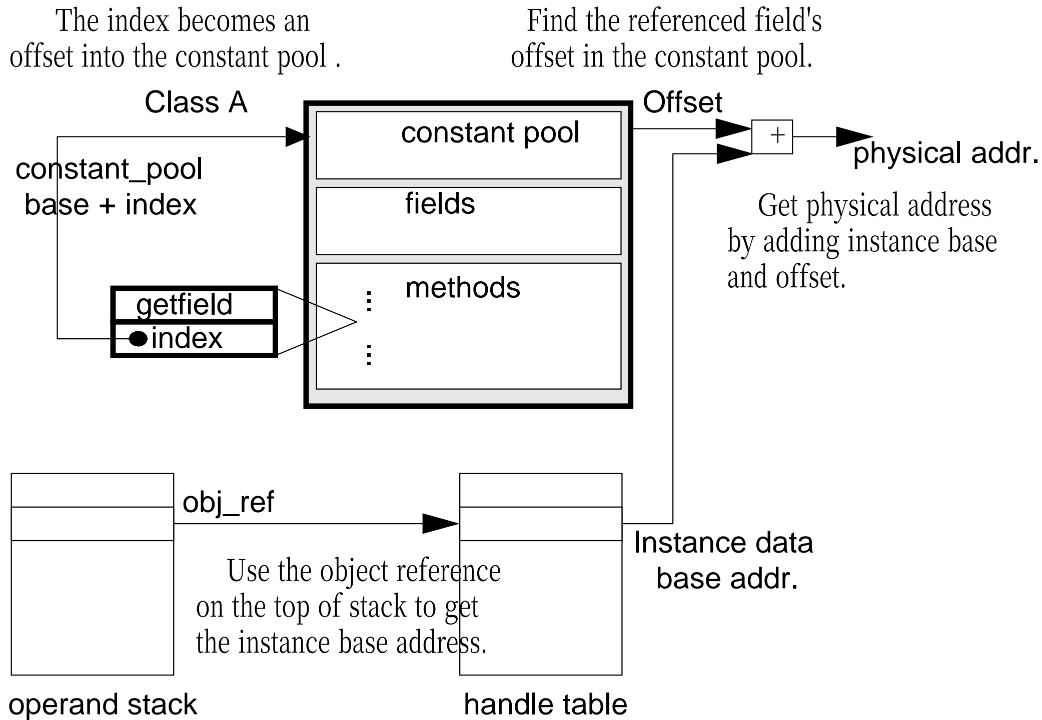


Fig. 3-4: Execution flow of object field access for *getfield*

Constant Pool Resolution

Java classes and interfaces are dynamically loaded, linked and initialized. Loading is the process of finding the binary form of a class or interface type with a particular name and constructing a class object to represent the class or interface. Linking is the process of taking a binary form of a class or interface type and combining it into the runtime state of the Java Virtual Machine so that it can be executed. Initialization of a class consists of executing its static initialization and the initialization for the static fields declared in the class.

A Java compiler does not presume to know the way in which a Java Virtual Machine lays out classes, interfaces, class instances, or arrays. References in the

constant pool are always initially symbolic. At run-time, the symbolic representation of the reference in the constant pool is used to work out the actual location of the referenced entity. The process of dynamically determining actual locations from symbolic references in the constant pool is known as constant pool resolution or dynamic linking. Constant pool resolution may involve loading one or more classes or interfaces, binding several types, and initializing types. This process always costs lots of cycles. After resolution, the useful information, such as the offset and type of the referenced target, will be placed in the corresponsive constant pool entry. We can get the resolved information when reference to the constant pool entry.

3.1.3 Mechanism of rewriting Java bytecode by SUN

In the optimization implemented in Sun's version of Java Virtual Machine, compiled Java code is modified at run-time for better performance. The optimization works by dynamically replacing certain instructions by more efficient variants at the first time they are executed. The new instructions take advantage of loading and linking work done the first time the associated normal instruction is executed. For instructions that are rewritten, each instance of the instruction is replaced on its first execution by a *_quick* pseudo-instruction. Subsequent execution of that instruction instance is always the *_quick* variant.

In all cases, the instructions with *_quick* variants reference the constant pool. The *_quick* pseudo-instructions save time by exploiting that, while the first time an instruction referencing the constant pool must dynamically resolve the constant pool

entry, subsequent invocations of that same instruction must reference the same object and need not resolve the entry again. The rewriting process is as follows:

1. Resolve the specified item in the constant pool.
2. Throw an exception if the item in the constant pool can not be resolved.
3. Overwrite the instruction with the *_quick* pseudo-instruction and any new operands it requires.
4. Execute the new *_quick* pseudo-instruction.

For instance, if we execute the bytecode *getfield* to access object field data, only the first execution goes through the process as shown in Figure 3-4. Subsequent executions become much faster because the Java Virtual Machine substitutes *getfield_quick* (or *getfield2_quick*, depend on its type) in place of the *getfield* bytecode. The index bytes after this *_quick* pseudo-instruction already becomes the offset of the target object as shown in Figure 3-5.

The benefit of dynamic linking via rewriting is that “Rewrite once, profited forever.” However, Java Virtual Machine must perform mechanism, such as coherency keeping between instruction cache and data cache, flushing the contents of the instruction buffer or instruction pipeline, to ensure correct functionality.

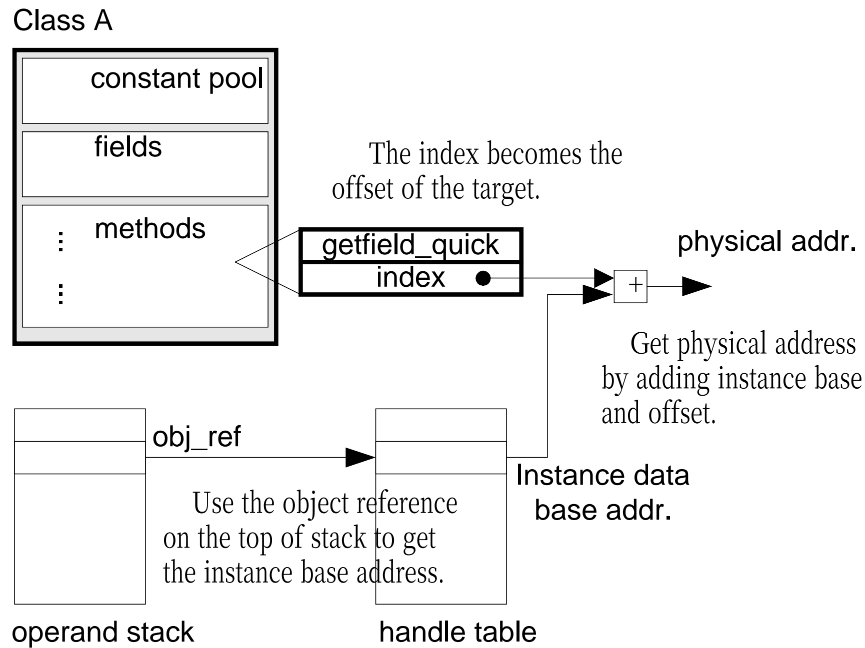


Fig. 3-5: Execution flow of object field access for *getfield_quick*

3.2 Design of Object Cache

In this thesis, we intend to accelerate the execution of object-field access instructions (*getfield* and *putfield*). In Section 3.1.2, we have given the detail execution flow of these instructions. To reach our purpose, we divide the design process into two parts: one is benchmark behavior analysis, the other is design issue consideration and simulation. In the first part, benchmark behavior analysis, we analyze the actual execution state of the SPECjvm98 benchmark. We have detailedly investigated the execution features of the benchmark suite, including temporal locality and reusing probability. Base on these results, we propose our acceleration technique—using of object cache. In the second part, cache design issues, design issues that may affect our acceleration mechanism are discussed and simulated. The design issues include the pipeline stage, index policies, cache line size, and cache

size.

3.2.1 Benchmark Behavior Analysis

In this subsection, we analyze the execution of SPECjvm98 benchmark. First, we describe our simulation approach including the benchmark and simulation environment. Then, some important features of benchmark behavior is presented. Base on these features, we propose our acceleration approach.

Simulation Approach

We choose the SPECjvm98 benchmark as our testing program. A brief explanation of the SPECjvm98 benchmark suite is given below:

- **_200_check** is a simple program to test various features of the JVM to ensure that it provides a suitable environment for Java programs.
- **_201_compress** implements file compression and decompression. It performs five iterations over a set of five tar files, each of them between 0.9 Mbytes and 3 Mbytes large. Each file is read in, compressed, the result is written to memory, then read again, uncompressed, and finally the new file size is checked.
- **_202_jess** is an expert system that reads a list of facts about several word games from an input file and attempts to solve the riddles.
- **_209_db** simulates a simple database management system with a file of persistent records and a list of transactions as inputs. The task is to first build up the database by parsing the records file and then to apply the transactions to this

set.

- **_213_javac** is the JDK 1.0.2 compiler iterating four times over several thousand lines of Java code; the source code of jess serves as input for javac.
- **_222_mpegaudio** is an application that decompresses 4 Mbytes of audio data that conform to the ISO MPEG Layer-3 audio specification.
- **_228_jack** is a Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). This is an early version of what is now called JavaCC. The workload consists of a file named jack.jack, which contains instructions for the generation of jack itself. This file is fed to jack so that the parser generates itself multiple times.

The class files of the benchmarks were executed on a modified JDK 1.0.2 interpreter to obtain the traces of the instrumented execution characteristics. These traces were then analyzed to identify the behavior of object-field access. Moreover, architectural components to support object-field access were proposed and simulated. The benchmark traces were also used to evaluate the proposed architectural components. Figure 3-6 shows this approach.

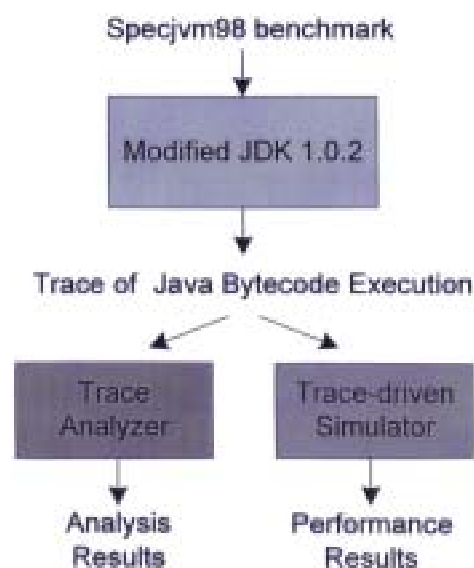


Fig. 3-6: Simulation approach

Temporal Locality and Reusing Probability

To accelerate object-field access, we analyzed the actual execution of the instructions of SPECjvm98 first. We find that most of these instructions usually access the same fields. Figure 3-7 shows the probability of an object field that will be reused in the next n object field accesses. We can see that over 70% of the object fields will be reused in the next 100 times access. Then, we make a simple simulation. We use an LRU buffer to store these object fields. Figure 3-8 shows the hit rate of the LRU buffer with *m* entries. We find that over 80% of the object fields can be hit in a 256-entry LRU buffer except for benchmark **_209_db.** Because **_209_db** simulate a large number of database records, only 60% of object fields can be hit in a 256-entry LRU buffer. Base on the analysis, we conclude that object fields have good temporal locality and are good for reuse.

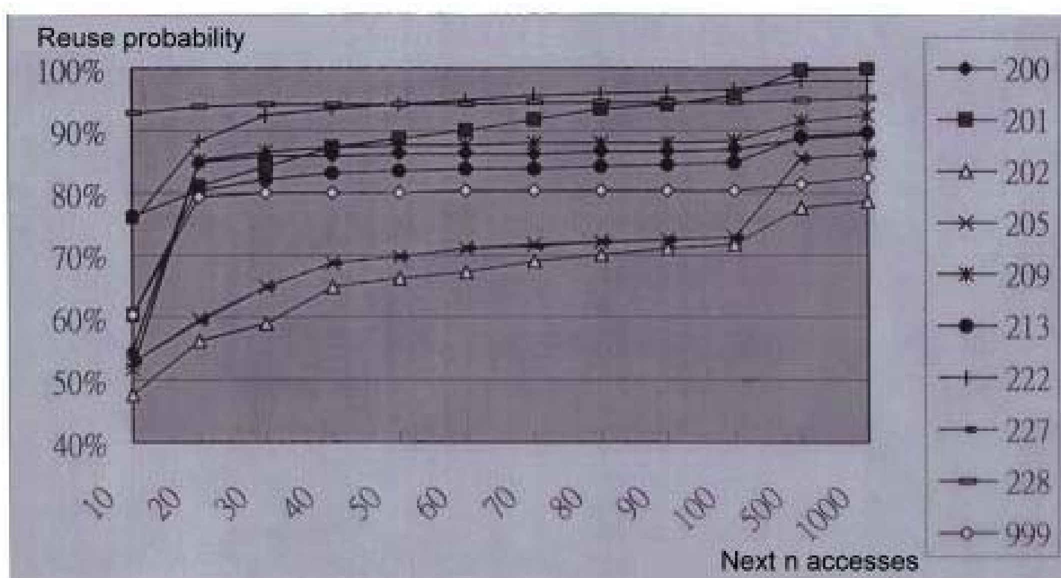


Fig. 3-7: Probability of an object field that to be reused in the next *n* object field accesses

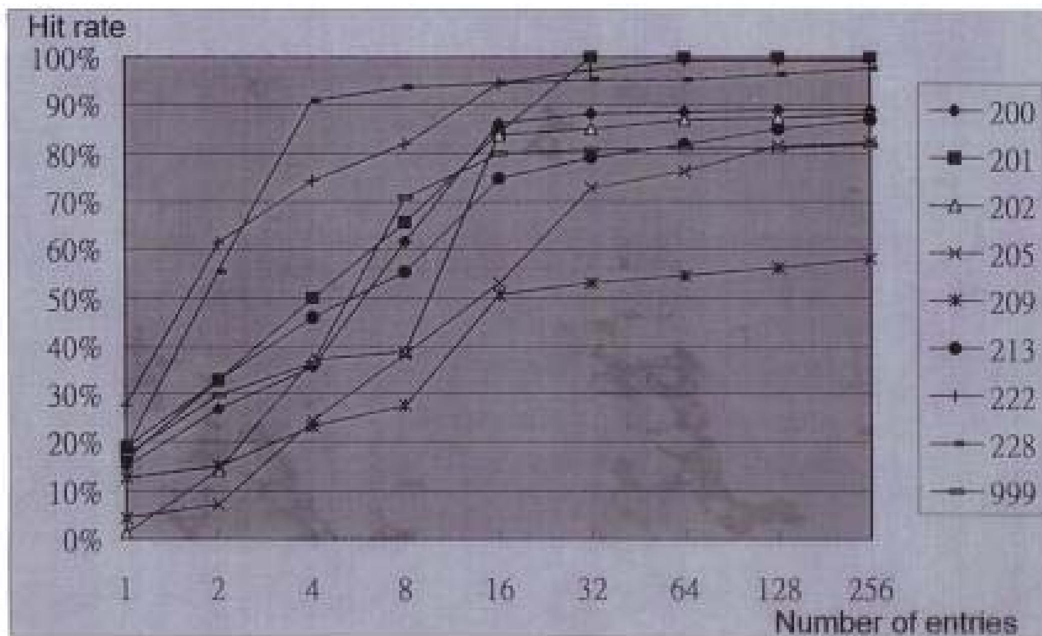


Fig. 3-8: Hit rate of the LRU buffer with *m* entries

Acceleration Approach — Object Cache

Through the analysis in the previous subsection, we know that object fields have good temporal locality and are good for reuse. Because of this feature of object fields, obviously, there is a good chance to earn speedup of performance by using a dedicated cache to store the data of referenced object fields. Therefore, we attach a cache, called object cache, in Java processor to execute *_quick* code directly without accessing constant pool and handle table. See Figure 3-9.

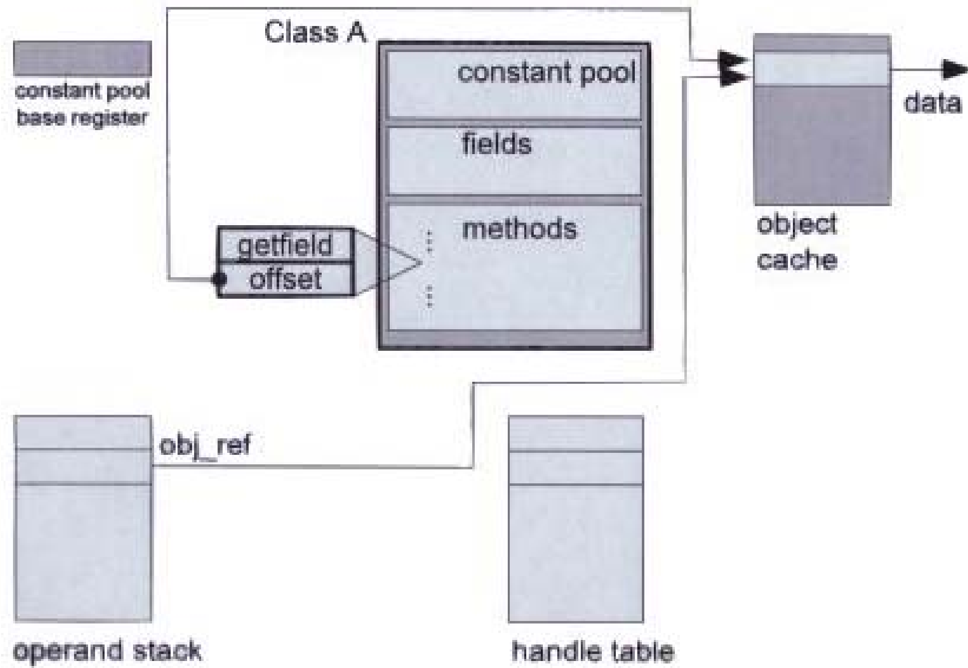


Fig. 3-9: Using an object cache to access object field data directly

3.2.2 Design Issues

The issues that affect the performance and the operations of our proposed mechanism for optimizing object field accesses are discussed in this subsection. The design issues of the object cache include indexing policy, pipeline stage design, cache line size, and cache size.

Issue I: Indexing Policy

Now we have to choose how to index the object cache entry, that is, how to identify a referenced object field. In the original bytecode "*getfield*" and "*putfield*", the value of the constant pool base register is added with the index byte of the bytecode to access constant pool. It means that the constant pool base register and index byte pair is used to identify a referenced object field. However, when garbage collection happened, constant pool base address may be changed, i.e., the physical memory address of an object field is not always invariable. In other words, we can not use physical memory address to index object cache because of garbage collection. Therefore, we consider another choice. The choice is to use object reference and offset supported by *_quick* code to index the object cache. Object reference is the unique id of an object and will never be changed. Offset is the location of the field inside the class instance and will never be changed, either. These mean that one object reference and offset pair map to only one object field. Hence, we use object reference and offset to index object cache.

To reduce the overhead of the tag field in the object cache, we analyzed the range of offset. An offset is an 8-bit value, $0 \sim 2^8 - 1 (=255)$ but is usually much smaller than 255. Figure 3-10 shows the range of the offsets for each benchmark. We obtained that the average value of offsets is 8.954. We can see that only 5 bits are required to almost 100% of the offsets. For those *_quick* codes with offset >= 32, we may use normal *_quick* execution without accessing object cache.

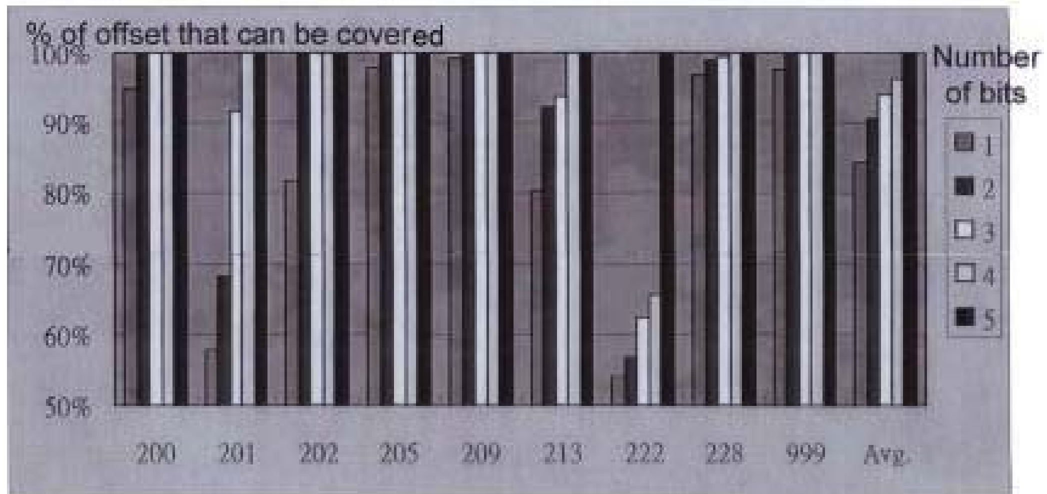
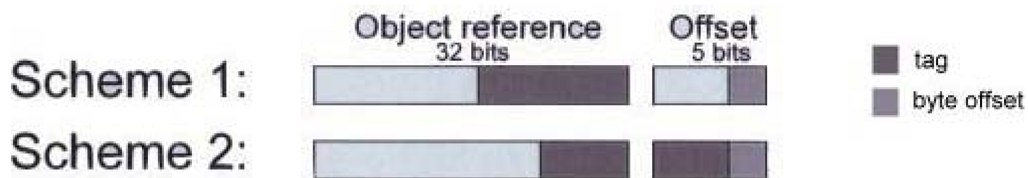


Fig. 3-10: Range of the offsets for each benchmark

We consider two schemes for the index bits of the object cache. These two schemes are shown in Figure 3-11. Scheme 1 uses object reference only to index object cache; scheme 2 uses the concatenation of object reference and offset to index object cache. The hit rates of the object cache under these two schemes are shown in Figure 3-11, too. We find that when the number of the object cache entries is larger than 1K, scheme 2 has better performance and has over 90% hit rate. Base on this, we use scheme 2 to index our object cache.



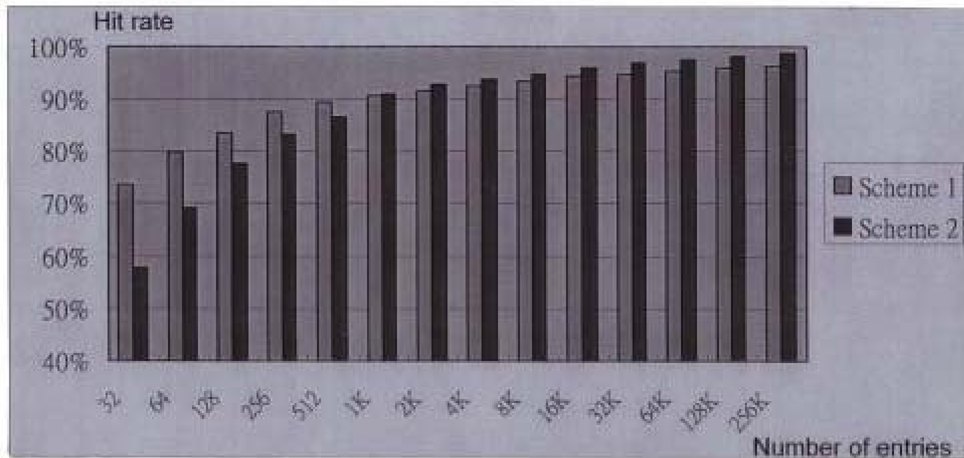


Fig. 3-11: Hit rates of the object cache under different indexing schemes and numbers of indexing bits. Assume that the object cache is direct-mapped with 8-word line size.

Issue II: Pipeline Stage Design

In our design, we use the six pipeline stages of PicoJava-II, as shown in Figure 3-12, without modification. The pipeline stage diagram is shown as figure 3-13. At decode stage, the indexbyte followed the quick code is decoded. Our object cache resides at register stage because all the information used to index object cache are ready then. In fact, our object cache can reside at either register or execute stage for forwarding.

Fetch	Decode	Register	Execute	Cache	Writeback
Fetch 1~8 bytes from cache to 16 byte instruction buffer	Decode top entries from instruction buffer	Fetch operands from stack cache; access microcode ROM	Execute; detect branches; bypass operands	Access data cache	Retire instructions; write results to stack

Fig. 3-12: Six pipeline stages of Sun's PicoJava-II

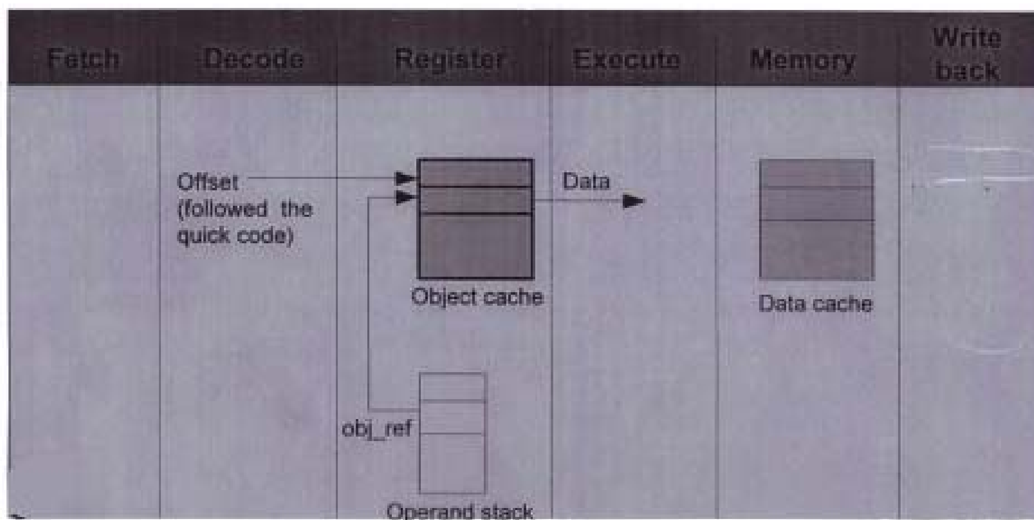


Fig. 3-13: Pipeline stage diagram

Issue III: Cache Line Size and Cache Size

From the previous analysis of the offsets, we obtained that the average value of offsets is 8.954. It means that the average size of an object is 8.954 word. For simplifying our design, we chose the line size of the object cache as 8 words. Figure 3-14 shows the hit rate of the object cache with different entry numbers under index policy of Scheme 2. We can see that when the size of the object cache is larger than

1K entries, the hit rate does not increase so notedly. Therefore, We choose an 1K-entry object cache in our design.

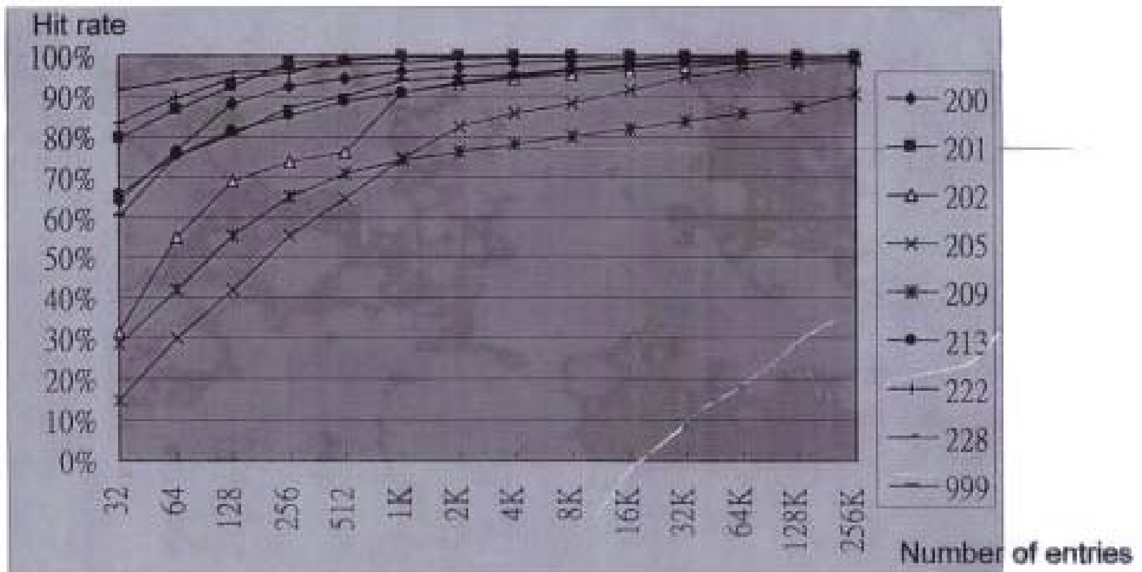


Fig. 3-14: Hit rates of the object cache with 8-word line size and index scheme 2 under different numbers of entries.

3.3 Performance Comparison

The speedup over Sun's picoJava-II for object field accesses by using the proposed object cache is shown in Figure 3-15. We can see that a direct-mapped object cache with 8-word line size, index scheme 2, and 1K entries can get the average speedup of 3.3.

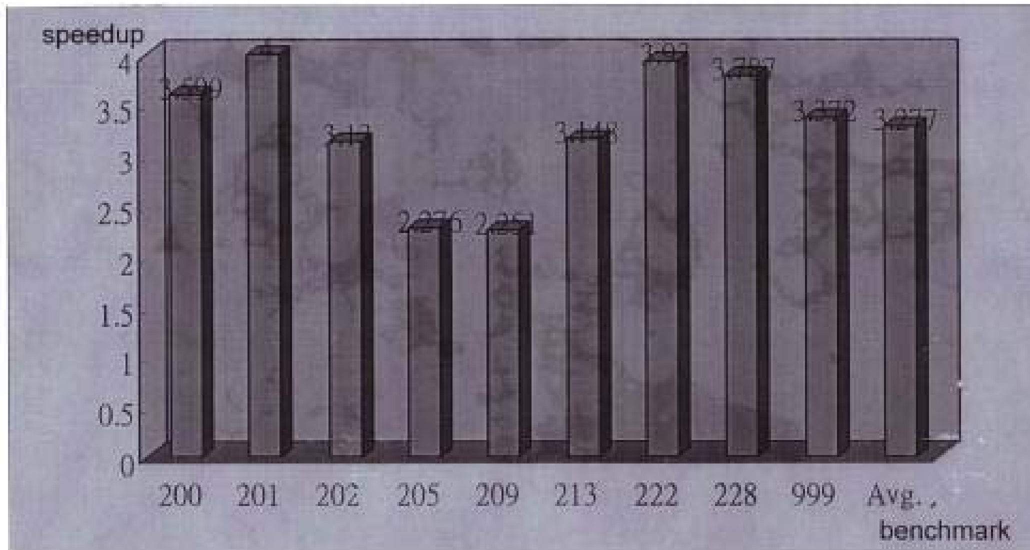


Fig. 3-15: Speedup over Sun's picoJava-II for object field accesses by using a direct-mapped object cache with an 8-word line size, index scheme 2, and 1K entries.

Chapter 4
Conclusions and Future Research

In this thesis, we presented the simulation studies to document how the performance of Java benchmark is impacted by the inclusion of a hardware acceleration mechanism for object field accesses into Sun's picoJava-II pipelined stack based architecture. We presented how an object field access is executed. The characteristics of object fields were identified by analyzing the benchmark programs.

Our simulation results show that the use of our proposed mechanism can be quite effective in speeding up the object field accesses by caching the field data in the object cache. For the benchmark programs we have studied, the inclusion of an

8-word line size, 1K-entry direct-map object cache can achieve 91% hit rate and 3.3 speedup over Sun's picoJava-II on object field accesses.

Future research efforts can address techniques to use object cache on superscalar Java processor. The object cache we proposed can speedup the performance of superscalar Java processor because its faster object field accesses may make more instructions be executable earlier.

reference

[1] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, see also http://www.javasoft.com/docs/books/jls/index.html.

[2] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1997.

[3] Bill Venners, *Inside the Java 2 Virtual Machine,* McGraw-Hill, June 1996

[4] N. Vijaykrishnan, *Issues In The Design Of A Java Processor Architecture.* Ph.D.'s Thesis, University of South Florida, Dec. 1998

[5] Sun Mircosystems, *PicoJava-II Mircoarchitecture Guide*, March 1999

[6] " Spec JVM 98 Benchmarks " http://www.spec.org/osh/jvm98 , 1998

[7] N. Vijaykrishnan, N. Ranganathan, "Supporting Object Accesses In A Java Processor" Computers and Digital Techniques, IEE Proceedings-, Volume: 147, Nov. 2000