

DELFT-JAVA Link Translation Buffer

John Glossner^{1,2}

¹Lucent / Bell Labs
Allentown, Pa
glossner@cardit.et.tudelft.nl

Stamatis Vassiliadis²

²Delft University of Technology
Electrical Engineering Department
Mekelweg 4, 2628 CD Delft, The Netherlands
stamatis@cardit.et.tudelft.nl

Abstract

We describe the hardware support in the DELFT-JAVA processor which enables efficient dynamic linking of JAVA programs. The proposed mechanism supports method invocation of dynamically linked classes through the use of a Link Translation Buffer (LTB). Since our Instruction Set Architecture directly supports dynamically linked method invocation, the Link Translation Buffer is architecturally transparent to the executing program. The operation of the LTB is described and preliminary performance results are reported. Method invocation differences between the C++ programming language and the JAVA programming language are outlined. Preliminary performance results for the Link Translation Buffer suggest that program performance may improve from 1.1x to 1.5x when a suitable LTB is used to cache frequently utilized methods.

1. Introduction

The JAVA programming language is a general-purpose, concurrent, class-based, object-oriented language[5]. It is strongly-typed so that common errors may be easily detected at compile-time but is also dynamically linked. Dynamic loading and linking resolves C++'s fragile class problem but imposes performance constraints on class accesses. In a C++ program, the actual method to be invoked may be deferred until run-time but the compiler must know the entire set of all possible classes during compilation. Therefore, when new objects are need, a new compilation must typically be performed. In Java, because all objects are dynamically linked, objects may be defined within a program which are not known at compile time. This resolves the fragile class problem but imposes a run-time penalty to dynamically link the classes.

In large class libraries, many classes may never be required in a particular application. In some cases, the class may not be available. The JAVA language does not require

that all the classes be available until a particular object calls it. In these cases, it may be desirable to delay the linking process until the class or method is required. A JVM may decide to pre-load many classes and their methods on the basis that they may be required. This policy incurs a large start-up penalty and may increase total execution time if some of the dynamically linked classes are not used. Alternatively, a JAVA Virtual Machine may delay loading and linking of a class or method until it is actually invoked.

1.1. Software Approaches To Bytecode Execution

Current implementations of the JAVA Virtual Machine take alternative approaches to JAVA bytecode execution. One solution is *interpretation*. This provides cross-platform portability but poses a number of performance issues. In this approach, a software program emulates the JAVA Virtual Machine. This requires software to execute multiple machine instructions for each emulated instruction. While this approach provides for maximum flexibility, the performance achieved can be as low as 5-10% the performance of natively compiled code [7]. *Native compilers* use JAVA as a programming language and generate native code directly from the high-level source. Even though this approach is contrary to the JAVA philosophy of "write once, run anywhere" [6], it may provide a good opportunity for speed improvement since no information is lost during high-level compilation. *Just-in-time compilers* (JIT) perform translation from JAVA bytecodes to native code just prior to executing the program. JITs have demonstrated 5-10x performance improvement over interpretation [7, 10]. However, the compilation is only resident for the current program invocation. Because they utilize processor resources, the number of optimizations that can be performed prior to execution is restricted [7]. *Off-line compilers* translate JAVA bytecodes to machine code prior to execution. This requires that programs be distributed and installed (e.g. compiled) prior to use. Since it is assumed that the compilation is performed once and maintained on a disk, additional time

may be devoted to optimizations. Except for loop information, the JAVA bytecodes contain nearly the same amount of information as the source itself [10]. Therefore, an off-line compiler should be nearly as efficient as a native JAVA compiler. Recently, hybrid approaches [10] have been investigated. In this approach, a highly optimizing JIT compiler and a JAVA Virtual Machine are integrated into a run-time library. This allows code to be loaded in an already compiled application. Stated performance improvements of 140x interpretive approaches and 13x JIT compilers have been reported.

1.2. Hardware Approaches To Bytecode Execution

Hardware approaches to improving JAVA performance have been proposed. Sun's *picoJava* [11, 13] implementation directly executes the JAVA Virtual Machine Instruction Set Architecture (ISA) but incorporates other facilities that improve the system level aspects of JAVA program execution. The *picoJava* chip is a stack-based implementation with a register-based stack cache. Support for garbage collection, instruction optimization, method invocation, and synchronization is provided. Sun states that this provides up to 5x performance improvement over JIT compilers [12].

Another approach to hardware acceleration is *dynamic instruction translation*. To the best of our knowledge, the DELFT-JAVA processor [3] is the only processor to incorporate this feature. In hardware assisted dynamic translation, JAVA Virtual Machine instructions are translated on-the-fly into the DELFT-JAVA instruction set. The hardware requirements to perform this translation are not excessive when support for JAVA language constructs are incorporated into the processor's ISA. This technique allows application level parallelism inherent in the JAVA language to be efficiently utilized as instruction level parallelism while providing support for other common programming languages such as C/C++. In addition to dynamic translation, a special Link Translation Buffer (LTB) can be used to improve the performance of dynamic linking.

In the following sections, we present an architecturally transparent mechanism for accelerating JAVA method invocation which supports dynamic linking. In Section 2 we briefly describe the issues associated with dynamic linking. In Section 3 we describe the operation of the Link Translation Buffer. In Section 4 we present a description of a DELFT-JAVA processor which incorporates a Link Translation Buffer. In Section 5 we will describe the results based on the simulations which characterize the Link Translation Buffer performance. Finally, in Section 6 we present our conclusions.

2. Background

Dynamic method invocation is a technique whereby a program may invoke a method with the same name and parameters but execute a different sequence of code depending upon the run-time type of the object invoking the procedure. The JAVA programming language supports generalized use of dynamic method invocation. The C++ language also supports a more limited form of this behavior through the virtual keyword. As in C++, JAVA method invocation generally involves an indirection through a method dispatch table.

Program 1 C++ Method Invocation

```
class MyClass {
    public:
        virtual void instanceMethod() {};
};

class MySubClass : public MyClass {
    public:
        virtual void instanceMethod() {};
};

void main() {
    MyClass mc = MyClass();
    MyClass msc = MySubClass();
    mc.instanceMethod();
    msc.instanceMethod();
}
```

Program 2 JAVA Method Invocation

```
public class MyClass {
    void instanceMethod() {}
}

public class MySubClass extends MyClass {
    void instanceMethod() {}
}

class Test {
    public static void main(String args[]) {
        MyClass mc = new MyClass();
        MyClass msc = new MySubClass();
        mc.instanceMethod();
        msc.instanceMethod();
    }
}
```

In the programs shown in Program 1 and Program 2, both the C++ and JAVA statements for `msc.instanceMethod()` invoke `MySubClass::instanceMethod()`. In the C++ version, the virtual keyword informs the compiler that the method `instanceMethod()` will have runtime binding behav-

ior. The capability of calling different methods at runtime is known as *late binding* because the method to invoke is not known until the program executes. In C or Pascal, by contrast, a function call always resolves to a specific, compile-time known, location. This is known as *early binding* because physical addresses can be associated with the function during compilation and linking. The advantage of early binding is that the only run-time overhead is argument passing, performing the function call, and cleaning up the frame stack. The advantage of late-binding is that the mix of objects in a system is not required to be fixed at compile-time. The cost of this additional flexibility is the run-time efficiency of deducing which methods to invoke. C++ is a hybrid language and only uses late binding when the virtual keyword is utilized but still requires the set of all potential objects which may be invoked to be known at compile time. JAVA, because of its dynamic linking facility, is inherently a late-binding language that allows an arbitrary set of objects, which may not all be known at compile time, to be invoked at run-time.

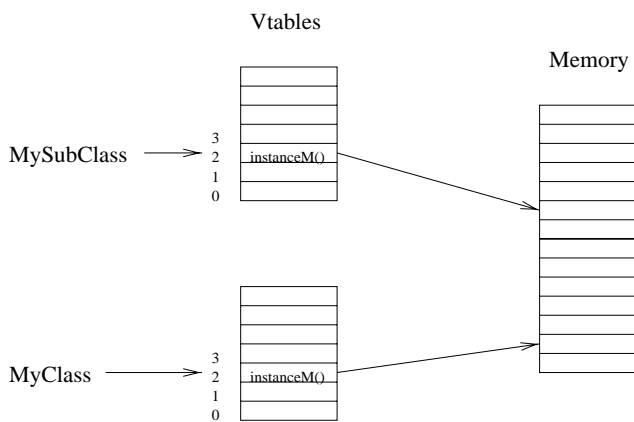


Figure 1. C++ Virtual Table Implementation.

In Figure 1 we show a possible implementation of C++ virtual tables. During compilation, two tables can be created - one for MyClass and one for MySubClass. These are stored in memory. When an object invokes the function `instanceMethod()`, they both have the same offset in their virtual tables. The contents of that memory offset is used to reference the actual location of the code in memory.

To understand how dynamic linking and late-binding are performed in JAVA, it is instructive to see the compiled bytecodes. As shown in Program 3, the first instruction (`new #3`) creates a MyClass object on the heap. Line 3 invokes its constructor. Line 5 does similarly for MySubClass. The interesting cases are found at lines 10 and 12. The `invokevirtual` call for both the `MyClass::instanceMethod()` and `MySubClass::instanceMethod()` are called with the same Constant Pool index (e.g. #5). The only way to distinguish them is through the object reference that is loaded in Lines 9 and

Program 3 JAVA Method Invocation Bytecode

```
Method void main(java.lang.String [])
Line InstrAddr Instr
1      0      new #3 <Class MyClass>
2      3      dup
3      4      invokenonvirtual #7 <Method
                          MyClass.<init>()V>
4      7      store_1
5      8      new #4 <Class MySubClass>
6     11      dup
7     12      invokenonvirtual #6 <Method
                          MySubClass.<init>()V>
8     15      astore_2
9     16      aload_1
10    17      invokevirtual #5 <Method
                          MyClass.instanceMethod()V>
11    20      aload_2
12    21      invokevirtual #5 <Method
                          MyClass.instanceMethod()V>
13    24      return
```

11. In line 9, the method dispatch table to use is the one for a MyClass object. In line 11, it is for the MySubClass object. As in C++, the offsets into the Constant Pool are the same. The actual method to call is disambiguated by the object reference which is loaded onto the stack at run-time. This is exactly analogous to Figure 1, except that the method dispatch tables are built at run-time by the runtime system.

3. Dynamic Linking Acceleration

The JAVA Virtual Machine [9] contains support for runtime bound method invocation. Because the DELFT-JAVA processor incorporates invocation instructions directly into its ISA, architecturally transparent techniques can be used to accelerate dynamic linking and method invocation. In this section we briefly describe method resolution and invocation. We then introduce an architecturally transparent technique, the Link Translation Buffer, and explain its operation.

3.1. Method Invocation

In a JAVA program, the Constant Pool contains the names of the methods to be invoked. These are stored as strings and function much like a symbol table. When a method is invoked, the JAVA Virtual Machine searches the Constant Pool for the name of the method to invoke. Then, based on the run-time type of the object invoking the method, it determines if the method has already been resolved [5]. If the method has not been resolved, the runtime system searches for the method. If the method is found, the address of where the method is loaded is returned and execution continues from the new address. If the method has been resolved, the

name contained within the constant pool can be associated with a physical location in memory for each object.

3.2. Link Translation Buffer Operation

A Link Translation Buffer is a buffer which accelerates late-binding of names with locations in memory. It is properly characterized as an organizational technique although some architectural support can be provided. In particular, a kernel program may need to enable, disable, lock, or judiciously flush the buffer.

The Link Translation Buffer provides an architecturally transparent means to accelerate the JVM's late-bound method invocations. When a DELFT-JAVA processor executes a method invocation instruction, it first dynamically translates the instruction into an equivalent DELFT-JAVA invocation instruction. When the instruction is executed, it checks the Link Translation Buffer to determine if the current object reference and its associated constant pool address are in the Link Translation Buffer. If the information is resident in the LTB, a new frame is created and the method is directly invoked. If the information is not in the LTB, the instruction is forwarded to the Control Unit. The Control Unit may be implemented as a state-machine, microcode, or be a separate processor executing a thin interpretive JVM layer. The Control Unit is responsible for resolving the method name and placing the physical method invocation address into the Link Translation Buffer.

In designing the Link Translation Buffer, there are two cases to consider: 1) dynamic method invocation and 2) static method invocation. In dynamic method invocation, a JAVA Virtual Machine invoke instruction takes a 16-bit Constant Pool value from the instruction format and searches the current object's Constant Pool for the name of the method to invoke. The current object's *this* pointer and the Constant Pool index from the instruction field provide a unique identifier to the name of the method to invoke. The name of the method includes the class name, the method name, and the method signature (e.g. the argument and return types). The DELFT-JAVA processor must then determine if the class is already loaded. If it is not, the Control Unit searches for the class and loads it. The instance reference is then retrieved from the stack and the list of methods defined by that class (and possibly its super-classes) is searched. If a method is found that matches the name and descriptor, it is invoked. Once this relationship is resolved to a physical address, additional invocation instructions may use the previously resolved address directly. Thus, the caller's object id (this pointer and Constant Pool location) and the callee's object reference provide sufficient information to directly invoke the method. This relationship is stored in the LTB. In static method invocation, the method to be invoked is a class method (e.g. not an in-

stance method) and does not need an explicit object reference. In this case the caller's object id is sufficient to invoke the method.

The Link Translation Buffer can support a variety of entries and associativities depending upon the desired implementation cost. We note that since the runtime can define the object ids, depending upon the actual associativity of the LTB, we may optimize the runtime to produce object ids which minimize cache conflicts. This is not true, however, for the 16-bit Constant Pool offset location.

In the design of the Link Translation Buffer some further enhancements can be made that reduce the impact of creating new frames. Our initial design places a small amount of additional data into the LTB. In the DELFT-JAVA processor, a frame is created which holds the 32-bit base values of the Constant Pool and Local Variables. Because each invoke instruction causes the creation of a new frame, we would like the frame creation to have as minimal overhead as possible. To assist this, we designed all execution frames within a thread to be contiguous. The Heap, Local Variables, and Constant Pools do not have this requirement. Making the frames contiguous allows us to use the operand stack as the frame stack. Because there can be 2^{16} Local Variables in a JAVA frame, we do not place this data in the execution frame. By not placing them in the frame stack, we avoid excessive flushing of the register file stack cache. In Java, only the actual operands are considered to be placed on the stack. In our method, the calling parameters also traverse through the stack so that the values are cached by the register file. Because the register file operates as a type of cache, if we were to change the operand stack's offset register, the modified register file locations would be required to be flushed on every method invocation. This requires no more additional cycles than a caller-save calling convention however we can gain the advantages of using a register file with no register saving overhead and variable length parameter passing. By using contiguous frames, the only information that needs to be stored when a new frame is created is the base addresses of the Local Variables and Constant Pool and the instruction address to return to the previous frame. These additional pushes and pops are performed transparently to the JAVA programmer and are logically considered to be part of the Operating System. As shown in Figure 2, some other performance enhancements can also be made by associating additional data with the LTB. Examples include synchronization locks, garbage collection reference counts, actual data, and other information to accelerate dynamic invocation.

The JAVA Virtual Machine also defines other operations which require late-binding. The `getstatic`, `putstatic`, `getfield`, and `putfield` instructions may work in an analogous manner except that the value stored in the LTB is the data field itself rather than the address of the method to invoke. In addition,

these instructions do not cause the creation of a new frame.

Caller's Reference	CPool Entry	Callee's Object Ref	LV[0]	CP[0]	Other
32-bit	16-bit	32-bit	32-bit	32-bit	

Figure 2. DELFT-JAVA Link Translation Buffer Layout

4. Performance Model

In this section, we describe the performance model used to generate the results. We do not describe details of the architecture unless they are relevant to the operation of the Link Translation Buffer. The primary purpose of the experiments is to characterize the performance of the Link Translation Buffer and to isolate the performance improvement of the LTB from other processor features. The only parameters which are varied are of interest in characterizing the Link Translation Buffer.

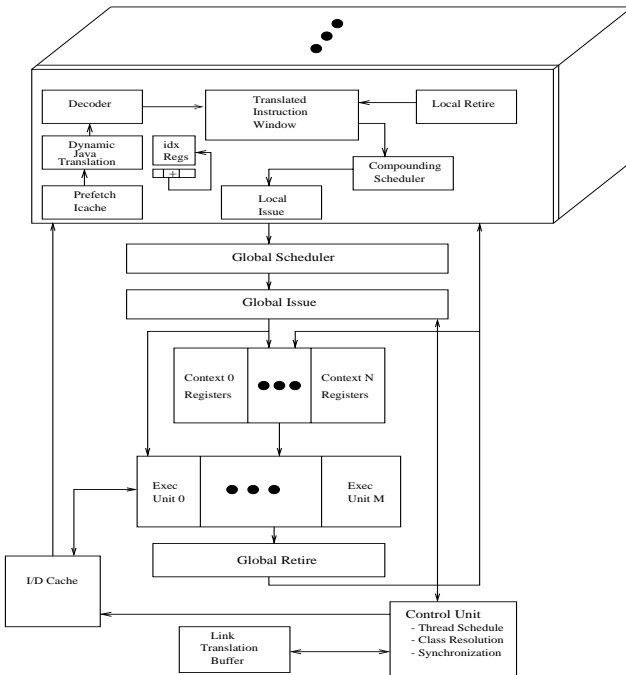


Figure 3. DELFT-JAVA Concurrent Multi-threaded Model

An organization of the DELFT-JAVA architecture which supports multiple concurrent execution of threads and

shared global execution units is shown in Figure 3. We define a *context* as a hardware supported thread unit. A context does not include shared resources such as a first level (L1) cache, execution units, a register file, global instruction schedulers, nor global issue units. The term thread is generally used to refer to the programmer's view of a thread - a possibly concurrent stream of independent executing instructions [8]. In this paper, the term context denotes the hardware on which a thread may run.

4.1. Operation

All instructions are fetched from global shared memory and placed into a global L1 on-chip instruction cache. Each context also assumes a zero level (L0) prefetch instruction cache to provide concurrent per context *instruction fetch* capacity. During normal user-level operation, all instructions are fetched as JAVA instructions. After being fetched, instructions are *dynamically translated* into the DELFT-JAVA instruction set. This is accomplished by making a circular buffer out of the register file [3]. Because the instructions are stored in cache memory as JAVA instructions, branching and method invocation code produced by JAVA compilers will execute properly on the DELFT-JAVA architecture. After translation, the instructions are decoded and placed into a *local instruction window* which keeps track of issued and pending instructions. The *local instruction scheduler* takes translated instructions in a RISC form and schedules them for execution. The *local issue unit* determines if the instructions that have been locally scheduled can be issued to the *global instruction scheduler*. All instructions which require access to shared resources must be forwarded to the global instruction scheduler. This unit schedules the aggregated instructions destined for execution units. The JAVA language specifies that in the absence of explicit synchronization, a JAVA implementation is free to update the main memory from multiple threads in any order [5]. This relaxed memory consistency model allows the scheduler to reorder the instructions from individual contexts to optimize the utilization of the shared execution units. The *global issue unit* ensures that global resources are available prior to issuing instructions. Instructions may be issued in one of two forms: single independent instructions and compound parcels [14]. After execution, all results are forwarded to the *global retire unit* which writes the results to the register file. In addition, this unit removes the requirement for a general interconnection unit between all contexts and execution units. The *local retire unit* removes the instruction from the window and commits state in the context. Each context may retire multiple instructions per cycle. The control unit is responsible for synchronization, cache locking, initial dynamic linking, I/O, loading instructions, and general system functions. Since the JAVA Virtual Machine does not

provide all the functionality generally required by a full operating system, many of these functions have been grouped into a special control unit. A control unit is analogous to a context except that it contains additional resources that are not necessarily required within a Java context.

5. Results

This section contains preliminary performance results for the DELFT-JAVA Link Translation Buffer. We investigate a range of performance the LTB can provide and then characterize the projected actual performance on an DELFT-JAVA processor. For the results presented, we assume: 1) all instructions have unit latency except for invocation instructions, 2) perfect branch prediction, 3) perfect L1 and L2 caches, 4) a single-context DELFT-JAVA processor, and 5) one instruction issued in-order per cycle.

We have built a C++ model which is capable of simulating JAVA programs that do not make system calls. The model supports dynamic translation and the Link Translation Buffer. It currently does not perform garbage collection or simulate I/O. We have used a program that produces synthetic workloads that can execute within our model. The workload generator allows the number of objects that are created and invoked to be varied. It also supports adjusting the dynamic instruction mix between invocation instructions and other instructions. Our results are based on the synthetic workloads generated by the program.

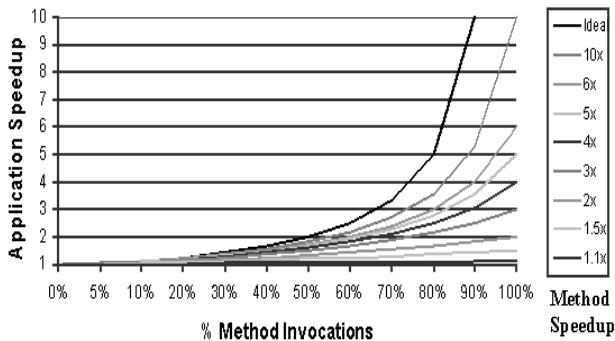


Figure 4. Application Speedup Versus Method Speedup (A)

5.1. Available Performance

We have isolated all performance results to be relevant to a DELFT-JAVA processor with and without a Link Translation Buffer. This allows us to compare application speedup due to the LTB mechanism. Figure 4 shows application

speedup versus the percent of invocation instructions. This relationship is based on Amdahl's law [1]. Ideal speedup refers to code that has all invocation instructions removed from the instruction stream (e.g. they execute in zero time). As an example, if 50% of the dynamic instructions are invocation instructions, then an ideal DELFT-JAVA processor would accelerate the application by 2x. In the above example, if we accelerate method invocation by 10x, we anticipate an application speedup of about 1.8x.

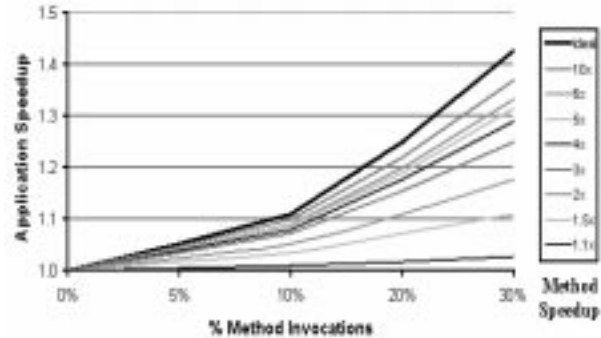


Figure 5. Application Speedup Versus Method Speedup (B)

Not all applications have large opportunities for acceleration. Sun has shown an instruction distribution with less than 10% of all instructions being invocations [11, 12]. Figure 5 highlights a portion of Figure 4 and is intended to represent a more typical distribution of invocations.

Work Load	Objects Created	Percent Dynamic Instr	Ideal Speedup
WL1	2048	40%	1.67
WL2	32	10%	1.11
WL3	512	20%	1.25
WL4	1024	30%	1.43

Table 1. Workload Characteristics

5.2. Workload Characterization

Since our simulator does not yet execute API or System calls, we have generated four synthetic workloads that are intended to represent a reasonable range of JAVA programs. Table 1 shows the characteristics of the workloads. The first workload, WL1, creates many objects and has a high percent of dynamic instructions which access the Link Translation Buffer. An example of this type of program is object-oriented Tensor Fast Fourier Transforms (FFTs) [4].

The Tensor program is distinguished by a complex type implemented in JAVA. Many objects are created in this program. Additionally, many instance methods are recursively invoked to perform the FFT.

The second workload, WL2, creates very few objects and contains very few method invocations. This is intended to simulate code which may have been ported from C. An example of this type of program is the Press FFTs [4]. A key feature of this type of code is the lack of method invocation and the use of static (e.g. class) method invocations where possible.

The final two workloads, WL3 and WL4, represent intermediate points between the extremes of workload WL1 and WL2. They may be typical of object-oriented code that uses most of the features of the JAVA language.

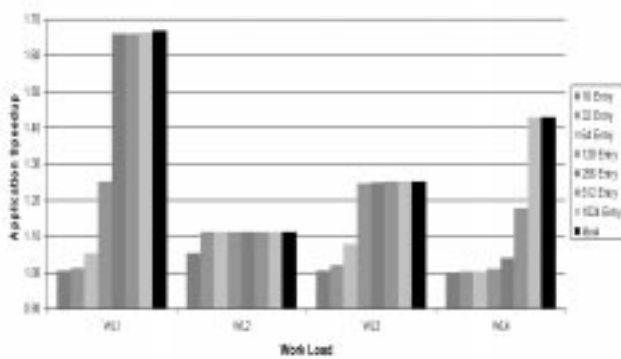


Figure 6. Application Speedup Versus LTB Cache Size for Workloads

5.3. Link Translation Buffer Characterization

Figure 6 shows overall application speedup for the four workloads versus the number of LTB entries. We assume a fully associative Link Translation Buffer with greater than 100x method invocation speedup. In addition, we use a random replacement policy. This allows us to minimize the effect of degenerate loops which invoke more objects than can fit within a particular Link Translation Buffer working set. Furthermore, these should be considered optimistic since the effect of I/O and garbage collection is ignored. We also note that these numbers include only the instructions of the simulated workload. Sun’s JVM, for example, implements some of the JAVA Virtual Machine functions in JAVA. This creates additional objects in their JVM. Our simulator is implemented in C++ but is not fully compliant and does not yet execute system calls. Therefore, our model does not create any additional objects.

For workload WL1, the largest performance gain is achieved. Since this workload had the most opportunity for

method acceleration, the results improve to nearly the ideal maximum if the LTB contains sufficient entries to hold the most frequently used objects. Workload W2 achieves near optimal application speedup as soon as the LTB is of sufficient size to hold the most frequently used objects.

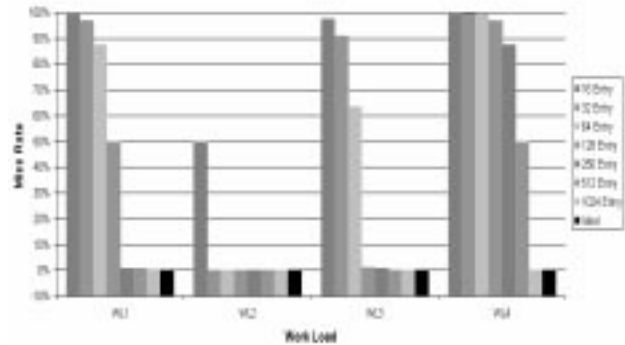


Figure 7. LTB Miss Rate Versus Entries for Workloads

Figure 7 shows the Link Translation Buffer miss rates for the four workloads. By miss rate, we mean the traditional definition - the probability of a reference made to the buffer is not in the buffer [2].

Notably, workload WL4 is the only workload that requires more than 512 entries. As Table 1 shows, 1024 objects are created for this workload. However, a large number of the objects are required for the working set. Workload WL1 has 2048 objects created but its working set of objects is much smaller and therefore achieves lower miss rates at 256 LTB entries.

6. Conclusions

In most Instruction Set Architectures, it is not possible to directly encode high-level operations in a single instruction. In particular, late binding method invocation is generally not supported. In this case, a C++ compiler would be required to emit a sequence of instructions that would point to a dispatch table based on the object calling the function. The dispatch tables are set up so that the offset of the function is the same regardless of the actual class. However, at the instruction level, because a sequence of memory and arithmetic operations is required, the information concerning the high-level operation is lost. Therefore, it is difficult for a traditional processor to optimize this operation. Because the DELFT-JAVA processor architecture retains the high-level information, it is possible to optimize for dynamic operations.

Currently, we are extending our model of the Link Translation Buffer to quantify performance degradation under various parameters including LTB associativity, multithreading, and non-unit latency memory access. In addition, we are assessing the use of the instruction address as a caller's object id. This may provide a benefit similar to Sun's quick instructions without requiring additional opcodes. Another advantage of using the instruction address is that the 16-bit constant pool location does not need to be stored in the Link Translation Buffer. However, potentially more entries may be registered for the same object since more than one instruction address may invoke the same method.

In conclusion, we have presented the operation of the DELFT-JAVA Link Translation Buffer. The purpose of this buffer is to accelerate Java's dynamic linking capability. This buffer is architecturally transparent but requires Instruction Set support for dynamic method invocation. If the target architecture contains these types of instructions, this technique may accelerate the performance of a Java application from 1.1x to 1.5x depending upon the number of objects utilized. The DELFT-JAVA processor architecture contains instruction set support for dynamic method invocation and may utilize a Link Translation Buffer to accelerate Java applications.

References

- [1] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings AFIPS National Computer Conference*, pages 483–485, 1967.
- [2] M. J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, Boston, Mass., 1995.
- [3] C. J. Glossner and S. Vassiliadis. The Delft-Java Engine: An Introduction. In *Lecture Notes In Computer Science. Third International Euro-Par Conference (Euro-Par'97 Parallel Processing)*, pages 766–770, Passau, Germany, Aug. 26 - 29 1997. Springer-Verlag.
- [4] J. Glossner, J. Thilo, and S. Vassiliadis. Java Signal Processing: FFT's with bytecodes. In *Proceedings of the 1998 ACM Workshop on Java for High-Performance Network Computing*, February 28 and March 1 1998.
- [5] J. Gosling, B. Joy, and G. Steele, editors. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.
- [6] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, Mountain View, California, October 1995. Available from <ftp.javasoft.com/docs>.
- [7] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. mei W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In *Proceeding of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, pages 90–97, Los Alamitos, CA, USA, December 2-4 1996. IEEE Computer Society Press.
- [8] B. Lewis and D. J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. SunSoft Press - A Prentice Hall Title, Mountain View, California, 1996.
- [9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.
- [10] G. Muller, B. Moura, F. Bellard, and C. Consel. JIT vs. Offline Compilers: Limits and Benefits of Bytecode Compilation. Technical Report 1063, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, December 1996. <http://www.irisa.fr>.
- [11] Sun Microelectronics. picoJava I Microprocessor Core Architecture. Technical Report WPR-0014-01, Sun Microsystems, Mountain View, California, November 1996. Available from <http://www.sun.com/sparc/whitepapers/wpr-0014-01>.
- [12] Sun Microelectronics. Sun Microelectronic's picojava I Posts Outstanding Performance. Technical Report WPR-0015-01, Sun Microsystems, Mountain View, California, November 1996. Available from <http://www.sun.com/sparc/whitepapers/wpr-0015-01>.
- [13] M. Tremblay and M. O'Connor. picoJava: A Hardware Implementation of the Java Virtual Machine. In *Hotchips Presentation*, 1996.
- [14] S. Vassiliadis, B. Blaner, and R. J. Eickemeyer. SCISM: A Scalable Compound Instruction Set Machine. *IBM Journal of Research and Development*, 38(1):59–78, January 1994.