Dynamic Linking on a Shared-Memory Multiprocessor

Bowen Alpern* Mark Charney* Jong-Deok Choi† Anthony Cocchi* Derek Lieber*

IBM T. J. Watson Research Center, PO box 218, Yorktown Heights, NY 10598

Abstract

This paper presents a technique for back-patching instructions in an SMP environment. This technique is used by the Jalapeño virtual machine to support dynamic class loading in Java. There is a small runtime overhead the first time a back-patch site is executed. Thereafter, it executes at the same speed as equivalent sites not requiring back-patching.

1 Introduction

The phenomenal success of the Java^{TM} [12] programming language is not merely a reflection of the phenomenal hype behind it. Java addresses some very real requirements of the programming community. Among these is the need for applications (browsers being a paramount example) that extend their functionality during execution. Java's provision for dynamic class loading satisfies this requirement: a class is linked the first time a bytecode referring to it is executed.

In the Java environment, bytecodes are either interpreted or compiled to machine code by a Just-In-Time (JIT) compiler and then executed. If bytecodes are compiled, mention of not-yet-linked classes is problematic. The mentioned class cannot be linked at compile time, since the bytecode that mentions it may not be executed immediately. (Indeed, it may *never* be executed.)

One scheme for compiling bytecodes that mention classes that have not been linked is to call a method that conditionally links the desired class. A performance drawback of this scheme is that this method will be called repeatedly, even after the class has been linked.

In a uniprocessor (Von Neumann architecture) computing environment, this drawback can be addressed in a straightforward manner: the method that conditionally links a class overwrites its call site with the code that the compiler would have emitted had the class been present at compile time. The backpatching method gets called at most once per dynamic link site. However, in a shared-memory multiprocessor computing environment, this approach is fraught with difficulties. What happens when two processors try to call the link method at the same time? Can a processor execute partially overwritten code? If so, what happens?

This paper shows how the Jalapeño JVM [2, 3, 4, 7] overcomes the difficulties inherent in concurrent backpatching on PowerPC multiprocessors [14]. Section 2 details Java's requirements for dynamic linking. Section 3 reviews the multiprocessing features of the PoverPC architecture. Section 4 presents Jalapeño's dynamic linking technique. Section 5 shows that this technique is correct. Section 6 discusses related work. And, section 7 concludes.

2 Dynamic linking in Java

Jalapeño is a Java virtual machine for server environments. As such, it is not designed to support browsers. However, dynamic class linking is an important feature of the Java language. On a server, it might be needed to execute uploaded *servlets* for instance. In any case, it must be supported by *any* Java virtual machine.

Java classes are linked in three phases [13]. In the first phase, *class loading*, the class is located and read into memory. In the second phase, *class resolution*, slots are laid out in an object template for instance fields, and space is set aside for static fields and for references to virtual and static method bodies. Field and method offsets are computed at this time. In the third phase, *class instantiation*, the class's methods may be compiled, and the class initializer is run. This initializer must not run more than once.

When a compiler encounters a bytecode (`putstatic`, `getstatic`, `putfield`, `getfield`, `invokestatic`, `invokevirtual`, or `invokespecial`) that mentions a class that has not been linked, it is not permitted to link the class immediately. (If the class is missing a `ClassNotFoundException` cannot be thrown until an appropriate bytecode is *executed*.) However, the offset of the field or method in question cannot be determined until the class is resolved. Also, the compiler must emit code that, unless the class has been linked before the bytecode

*{alpern, charney, tony, derek}@watson.ibm.com.
†jdchoi@us.ibm.com.

is executed, causes the class to get loaded, resolved, and instantiated at runtime and then performs the indicated operation. Jalapeño's baseline compiler[1] emits code that calls runtime methods that ensure the required class instantiation has happened. These methods then overwrite their call sites with the code the baseline compiler would have produced had the class initially been resolved.

3 The PowerPC SMP architecture

Jalapeño is targeted to shared-memory-multiprocessor (SMP) servers with PowerPC processors[14] running the AIX operating system [1]. Although the processors share the same main memory, each processor can have its own data and instruction caches. This paper discusses only split cache (Harvard architecture) implementations.

To execute an instruction, a processor fetches the instruction from its instruction cache. If the instruction is not in this cache, the cache line containing it is brought into the cache from main memory. The PowerPC architecture also allows instructions to be prefetched into a processor's *instruction prefetch buffer*. Instructions in this buffer may be executed without refetching them from the instruction cache. PowerPC processors normally execute several instructions simultaneously. Data dependencies among the instructions executing *on the same processor* are obeyed.

When a data value is loaded into a register, the cache line containing the value must be in the data cache of the processor. The same cache line may be in the data caches of other processors at the same time. When a register is stored into a cache line, if the cache line is not already held *exclusively* by the processor, a message is sent to the other processors telling them to invalidate their copy of the cache line. Such messages may be buffered up in an *invalidation buffer* on each processor.

The PowerPC architecture provides several instructions to explicitly manage these caches. Those that will be needed below are now described.

- The `dcbst` (data cache block store) instruction causes the line of cache containing a specified address to be written to main memory. This instruction has no effect on other processors.

- The `icbi` (instruction cache block invalidate) instruction causes the cache line containing a specified address to be invalidated in every instruction cache. These invalidation messages may get buffered in a processor's invalidation buffer. There is no guarantee as to when the processor will act on this message.

- The `isync` (instruction synchronization) instruction waits for all previous instructions on the processor to complete and the processor's instruction prefetch buffer is purged before any subsequent instruction is executed. This instruction has no effect on other processors.

- The `sync` (synchronization) instruction causes a memory barrier between previous and subsequent instructions issued by the processor. When another processor observes the effects of one of the latter, all of the former are guaranteed to have taken effect. Typically, this instruction ties up each of the processors. On a multiprocessor, this is an expensive instruction that should be avoided where possible.

Periodically, the AIX operating system performs a task switch. This must include reflecting in main memory any recently written data in the data cache of the processor that was executing the task and purging any stale data in the data or instruction cache of the processor that is about to resume executing the task. Note, however, that the new processor may still have stale data in its instruction prefetch buffer unless and `icbi` instruction explicitly purges such data.

4 Dynamic linking in Jalapeño

Jalapeño's implementation of dynamic linking will be demonstrated by detailed consideration of the `putfield` bytecode, which is typical and fully illustrates the general technique.

The `putfield` bytecode connotes writing a value to a field of an object. It takes a `FieldRef` as an argument. The FieldRef has three parts: the name of the class to which the field belongs, the name of the field, and its type. The offset of the field in the object is unknown until the class has been resolved. In Jalapeño, a field is either four bytes (`int`, `short`, `char`, `byte`, `boolean`, `float` or `Object`) or eight bytes (`long` or `double`). If the class has already been incorporated into the running Jalapeño image, then a compiler can emit code that directly writes the value to the appropriate location. Hereafter, we assume the class has not yet been so incorporated and that the field is four bytes long.

The code emitted by Jalapeño's baseline compiler for an unresolved `linkPutfield` bytecode is depicted on the "Before" side of figure 1. In this figure, `r3` and `r4` denote scratch registers, `r30`, a stack-top pointer, and `r31`, a pointer to a global table that includes references to static method bodies. The constant `@putfield` is the offset of

[1] Jalapeño has three different compilers. This paper only discusses the operation of the *baseline* compiler. It differs from the others in its restricted use of machine registers. It faithfully emulates the stack-based JVM specification [13]: the arguments to every bytecode are popped from a stack in main memory and the results of every bytecode are pushed onto the stack. Limiting consideration to this compiler simplifies the presentation.

```
          Before                                     After
          ------                                     -----

1)        l     r3,@putfield,r31                     b    zzz            (*)
2)        mtlr  r3                                   mtlr r3
3)        blrl                                       blrl
4)  xxx:  DATA(fieldId)                              DATA(fieldId)
5)  yyy:  isync                                      isync
6)        b     yyy                            zzz:  l    r3,0,r30       (*)
7)        b     yyy                                  l    r4,4,r30       (*)
8)        b     yyy                                  st   r3,@field,r4   (*)
9)        b     yyy                                  cal  r30,8,r30      (*)
```

Figure 1. A dynamic `putfield` site before, and after, it is backpatched.

the `linkPutfield` method in this global table. The constant `@field` is the appropriate field offset in the object. This is the value that is not available until the class has been resolved.

The first three instructions (lines 1, 2, and 3) call `linkPutfield` which will backpatch this `putfield` site. The fourth value (line 4) is not in fact an instruction, it is an encoding of the information in the `FieldRef` argument. The purpose of the `isync` (lines 5) will be explained in the next section. The branches to it (lines 6, 7, 8, and 9) are stand-ins for instructions to be backpatched.[2] Note that, as the code is written, the last five instructions are unreachable unless the `linkPutfield` method were to return normally, which it must *not*.

The (unsynchronized) `linkPutfield` method behaves as follows. First, it uses the most resent return address from the method invocation stack to retrieve the `FieldId` from address `xxx` (line 4). This id is an index into a table of `FieldRef`'s. The appropriate `FieldRef` is obtained from the table. If the class designated by the `FieldRef` is not yet a part of the running image, a synchronized `link` method loads, resolves, and instantiates the class.[3] Synchronization is required to ensure that this happens exactly once (the `link` method returns if the class has already been instantiated). Notice that the `link` method is only called the first time a dynamic link site for a class is executed, most dynamic link sites are backpatched without any call to a synchronized method.

After ensuring that the class has been instantiated, five instructions of the `putfield` site are overwritten. These are marked (*) in the "After" side of figure 1. The first of these (line 1) causes execution to bypass the call to `linkPutfield`. The other four are the instructions the compiler would have emitted for the `putfield` had its class been present at compile-time.[4] The first two of these obtain the value to be stored (line 6) and a reference to the `Object` containing the field (line 7) from the stack. The third (line 8) performs the store at an offset `@field` obtained from the now loaded class. The final instruction (line 9) adjusts the stack pointer to reflect popping two values from the stack. (Notice that lines 6, 7, and 8 can be executed repeatedly without adverse effect on the program state.) After the `putfield` site has been backpatched, the seven synchronization instructions of figure 2 are executed by the backpatching code. Finally, control is transferred to address `zzz`.

```
1)      dcbst xxx-12
2)      dcbst xxx+ 8
3)      dcbst xxx+20
4)      sync
5)      icbi  xxx-12
6)      icbi  xxx+ 8
7)      icbi  xxx+20
```

Figure 2. Synchronization instructions executed after a backpatch.

The `dcbst` instructions (lines 1, 2, and 3 of the synchronization code in figure 2) cause the cache line, or cache lines,[5] containing the backpatched code to be transferred

[2] In fact, only the value of `@field` (line 8 of "After") is not known at compile time. Lines 6, 7, and 9 could be the same before and after backpatching. Only line 8 need start out as a branch. (Overwriting all four instructions helps illustrate the general technique.)

[3] Since the stack should contain a reference to an `Object` of this class one might suspect that the class must have been instantiated. However, if the reference is `null` and the class does not exist, the JVM must throw a `ClassNotFoundException` rather than a `NullPointerException`.

[4] On a PowerPC, an unconditional branch (line 1, in "After") is normally free. Thus, the backpatched code will run at the same about the same speed as equivalent code not requiring backpatching.

[5] If cache lines are at least 36 bytes long, lines 2 and 6 can be omitted. If

from the cache of the backpatching processor to main memory. The `icbi` instructions (lines 5, 6, and 7) purge any stale copies of these cache lines from all the instruction caches. The expensive `sync` instruction (line 4) makes sure that the purged cache lines are not replaced with stale data from memory that is about to be overwritten by the `dcbst` instructions.

5 Correctness

The correctness of the dynamic `putfield` algorithm detailed in the previous section follows from three lemmas.

Lemma 5.1 *If a process ever executes the final version of line 9, it performs the correct computation.*

> To reach the final version of line 9, the processor must execute the final versions of lines 6 through 8 (because the initial versions of any of these lines would cause the executing processor to loop, not reaching line 9). By construction, these are the same instructions the compiler emits for `putfield` bytecodes when the class is available at compile-time. Thus, they had better be correct. □

Lemma 5.2
Execution of a partially backpatched `putfield` *site will not cause permanent damage.*

> If the partially backpatched code does not include the branch at line 1, the `linkPutfield` method will be called a second time. Because the `link` method that actually links a class is synchronized, it will not link a class twice. The instructions at the `putfield` site may be overwritten again, but since the values are the same the second time as the first, this is benign.
>
> If one or more of lines 6, 7, or 8 is still in its original state, the code will loop without executing line 9. Similarly, the code loops if line 9 is in its original state.
>
> The final version of line 9 is the only line that has a destructive impact on the programs state: it changes the stack top register (R30). If this instruction were to be executed twice, the stack would be corrupted. However, this is the last instruction and it is executed at most once. □

Backpatch code must be idempotent. Only the final instruction (if any) of a backpatch site may be destructive of the program state. Only this last instruction cannot be executed more that once.

cache lines are less that 16 bytes long extra `dcbst` and `icbi` instructions will be needed.

Lemma 5.3 *If a processor ever executes line 1 it will eventually execute the final version of line 9 (assuming it doesn't throw a* `ClassNotFoundException` *or the like).*

> Whether it executes line 1 in its original or final state, the processor must eventually get to line 6 after *some* processor has completed the backpatch. *This* processor may have stale values for some subset of lines 6, 7, 8, and 9 in its instruction cache. However, if it does, then it has, or will soon, have an invalidate message for the cache line containing these stale instructions in its invalidation buffer. After it processes that invalidation (as eventually it must), it may still have stale values in its prefetch buffer. If so, it will execute the `isync` instruction at line 5 purging the prefetch buffer and reloading the instruction cache with the final values from main memory. □

It remains only to argue that the instructions in our synchronization code are necessary as well as sufficient. Without the `icbi` instructions, some processors instruction cache might contain stale values for one or more of the instructions at lines 6, 7, 8, and 9 forever (or until the next context switch). The `dcbst` instructions provide that the values that correct final values are in main memory when the `icbi` invalidation causes new values to get loaded into an instruction cache. The `sync` instruction is needed to prevent a race condition in which the invalidation gets processed and the cache line reloaded before the `dcbst` instructions complete. It also prevents the instruction cache from being reloaded from a stale data cache.

6 Related work

The Paradyn system [15, 17] uses dynamic backpatching for dynamically inserting and deleting instrumentation code in user applications (or operating system kernels). To instrument an instruction in the user application, the Paradyn system first writes a stub of instrumentation code at a location different from the application code. The first instruction of this stub emulates the instrumented instruction. The body of the stub implements the instrumentation. The stub ends with a branch to the instruction following the instrumented instruction. Then the system atomically replaces the instrumented instruction with a branch to the stub. When the stub is too far from the instrumented code for a single-instruction branch statement to reach, an intermediary stub, called *springboard*, is allocated at a location to which a single-instruction branch can reach from the instrumented code. The springboard has multiple-instruction branch statements that can reach the real target stub.

This technique would be difficult to adapt for Jalapeño because its code can be relocated and the relative distance between the original code and its instrumenting stub may change. In Paradyn, the application code is always executable: there is no data (invalid code) embedded in the instruction stream. Therefore, failing to purge stale instructions potentially prefetched by other, non-backpatching, processors will not cause program failures (although it might result in improper measurements).

Dynamic translation of Smalltalk-80 [11] also dynamically generates native code (called *n-code*) from byte-code (called *v-code*) during execution. When a procedure is about to be executed, but is not in n-code form, the call faults. The translator then finds the corresponding v-code routine, translates it, and completes the procedure call. Smalltalk-80, however, does not address the issue of how to ensure correct translation in the presence of multiple threads running on a shared-memory multiprocessor.

The Self compiler [8, 10] employs a specialization, called *customization* [9], that dynamically generates specialized versions of a method optimized for different calling context. Self, however, does not address the issue of a shared-memory multiprocessor, either.

Dynamic link libraries (DLL) allow for dynamically linking of libraries when the application executes. Accesses to (static) variables or methods in DLL are generally performed indirectly through entries at known indices of a table whose location is known once static linking is performed. The entries at these indices are filled out at dynamic linking. However, no modifications of the running code are involved [6].

7 Conclusions

Dynamic code patching is a well-known runtime instrumentation technique used to support a variety of tools, such as debuggers and performance monitors. More recently, it has emerged as an important technique for efficient implementation of virtual machines for Java and other object-oriented programming languages. The effect of dynamic code patching is to modify binary code at execution-time. However, dynamic code patching poses many challenges in a multithreaded Java program, because of the potential asynchrony between a thread that is patching a code object and a thread that is executing the same code object. These challenges are compounded in a multiprocessing environment with multiple CPUs.

The backpatching algorithm presented here is necessitated by the relaxed memory consistency model of the PowerPC architecture. Similar algorithms would be required on other microprocessor architectures that use relaxed memory consistency models. Sun's V9 architecture [18] requires the use of a `flush` instruction for each modified doubleword. The `flush` behaves similarly to store instruction and is ordered by their `membar` memory barriers. DEC Alpha [5] requires an `imb` instruction between the write and the instruction fetch.

On architectures, such as the IBM S/390 [16], where a stronger memory consistency model is employed, the cache coherence mechanism ensures that modifications to the instruction stream are seen in the order in which they are performed. Stronger memory consistency models can eliminate the need for explicit cache management instructions for successful backpatching, while still allowing the reordering of certain operations for performance reasons. (However, even on S/390 when the backpatcher and the processor executing the backpatched code are using different addressing modes to access memory, a serializing instruction is required.)

This paper presents a technique for backpatching instructions in an SMP environment. The final code must be idempotent. Only the first word of the original code is changed. A synchronization protocol is used to insure that all processors eventually see the backpatched code.

This technique is used by the Jalapeño virtual machine to support dynamic class loading in Java. In this context, backpatch sites are about twice a big as equivalent sites where dynamic linking is not required. There is a runtime overhead to backpatch a site the first time it is executed. Subsequent executions run at the same speed as equivalent sites.

References

[1] *IBM AIX Version V4.3 Technical references*, 1998. IBM Corporation order number: SBOF-1878-00.

[2] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeno virtual machine. *IBM Systems Journal special issue on Java performance*, 39(1), 2000. (see also http://www.research.ibm.com/jalapeno).

[3] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. Shepherd, and S. Smith. Implementation of Jalepeño in java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[4] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño — a Compiler-Supported Java Virtual Machine for Servers. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999. Also available as INRIA report No. 0228, March 1999.

[5] Alpha architecture handbook. Digital Equipment Corporation, 1992.

[6] M. Auslander. Managing programs and libraries in AIX Version 3 for RISC System/6000 processors. *IBM Journal of Research and Development*, 34(1), January 1990.

[7] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and

J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[8] C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Mar. 1992. Published as technical report STAN-CS-92-1420.

[9] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–160, July 1989. *SIGPLAN Notices, 24(7)*.

[10] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self – a dynamically-typed object-oriented language based on prototypes. In *Proceedings OOPSLA '89*, pages 49–70, Oct. 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[11] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, Jan. 1984.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.

[14] C. May, E. Silha, R. Simpson, and H. Warren. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

[15] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. In *IEEE Computer*, November 1995.

[16] Enterprise Systems Architecture/390 Principles of Operation, sixth edition, 1998. IBM Corporation order number: SA22-7201-05.

[17] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernel. In *Third Symposium on Operating System Design and Implementation*, New Orleans, February 1999.

[18] D. Weaver and T. Germond. The SPARC Architecture Manual, Version 9. Prentice Hall, 1994.