Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences*

Etienne Gagnon and Laurie Hendren

Sable Research Group
Université du Québec à Montréal and McGill University
Montreal, Canada
[etienne.gagnon@uqam.ca,hendren@cs.mcgill.ca]

Abstract. Inline-threaded interpretation is a recent technique that improves performance by eliminating dispatch overhead within basic blocks for interpreters written in C [11]. The dynamic class loading, lazy class initialization, and multi-threading features of Java reduce the effectiveness of a straight-forward implementation of this technique within Java interpreters. In this paper, we introduce preparation sequences, a new technique that solves the particular challenge of effectively inline-threading Java. We have implemented our technique in the SableVM Java virtual machine, and our experimental results show that using our technique, inline-threaded interpretation of Java, on a set of benchmarks, achieves a speedup ranging from 1.20 to 2.41 over switch-based interpretation, and a speedup ranging from 1.15 to 2.14 over direct-threaded interpretation.

1 Introduction

One of the main advantages of interpreters written in high-level languages is their simplicity and portability, when compared to static and dynamic compiler-based systems. One of their main drawbacks is poor performance, due to a high cost for dispatching interpreted instructions. In [11], Piumarta and Riccardi introduced a technique called *inlined-threading* which reduces this overhead by dynamically inlining instruction sequences within basic blocks, leaving a single instruction dispatch at the end of each sequence. To our knowledge, *inlined-threading* has not been applied to Java interpreters before.

Applying this *inline-threaded* technique within an interpreter-based Java virtual machine (JVM) is unfortunately difficult, as Java has features that conflict with a straight-forward implementation of the technique. In particular, the JVM specification [9] mandates lazy class initialization, permits lazy class loading and linking, and mandates support for multi-threading. Efficient implementation of laziness requires in-place code replacement which is a delicate operation to do within a multi-threaded environment. In this paper, we introduce a technique

* This research was partly supported by NSERC, FCAR and Hydro-Québec.

called *preparation sequences* which solves the synchronization and shorter sequence problems caused by in-place code replacement within an inline-threaded interpreter-based JVM.

This paper is structured as follows. In Section 2 we briefly describe some interpreter instruction dispatch techniques, including inline-threading. Then, in Section 3 we discuss the difficulty of applying the inline-threaded technique in a Java interpreter. Next, in section 4 we introduce our *preparation sequences* technique. In Section 5, we present our experimental results within the *SableVM* framework. In Section 6 we discuss related work. Finally, in Section 7, we present our conclusions.

2 Dispatch Types

In this section, we describe three dispatch mechanisms generally used for implementing interpreters.

Switching A typical bytecode interpreter loads a bytecode program from disk using standard file operations, and stores instructions into an array. It then dispatches instructions using a simple loop-embedded *switch* statement, as shown in Figure 1(a). This approach has performance drawbacks. Dispatching instructions is very expensive. A typical compilation of the dispatch loop requires a minimum of 3 control transfer machine instructions per iteration: one to jump from the previous bytecode implementation to the head of the loop, one to test whether the bytecode is within the bounds of handled switch-case values, and one to transfer control to the selected case statement. On modern processors, control transfer is one of the main obstacles to performance [7], so this dispatch mechanism causes significant overhead.

Direct-Threading This technique was popularized by the Forth programming language [5]. Direct-threading improves on switch-based dispatch by eliminating central dispatch. In the executable code stream, each bytecode is replaced by the address of its associated implementation. This reduces, to one, the number of control transfer instructions per dispatch. Direct-threading is illustrated in Figure 1(b)[1].

Inline-Threading This technique, recently introduced in [11], improves upon direct-threading by eliminating dispatch overhead for instructions within a *basic block* [1]. The general idea is to identify instruction sequences forming basic blocks, within the code array, then to dynamically create a new implementation for the whole sequence by sequentially copying the body of each implementation into a new buffer, then copying the dispatch code at the end. Finally a pointer to

[1] Figure 1(b) uses the *label-as-value* GNU C extension, but direct-threading can also be implemented using a couple of macros containing inline assembly.

(a) Pure Switch-Based Interpreter

```
char code[CODESIZE];
char *pc = code;
int stack[STACKSIZE];
int *sp = stack;

/* load bytecodes from file and
   store them in code[] */
...
/* dispatch loop */
while(true) {
  switch(*pc++) {
  case ICONST_1: *sp++ = 1; break;
  case ICONST_2: *sp++ = 2; break;
  case IADD: --sp; sp[-1] += *sp; break;
  ...
  case END: exit(0);
}}
```

(b) Direct-Threaded Interpreter

```
/* code */
void *code[] = {
  &&ICONST_2, &&ICONST_2,
  &&ICONST_1, &&IADD, ...
}
void **pc = code;

/* dispatch first instruction */
goto **(pc++);

/* implementations */
ICONST_1: *sp++ = 1; goto **(pc++);
ICONST_2: *sp++ = 2; goto **(pc++);
IADD: --sp; sp[-1] += *sp;
      goto **(pc++);
...
```

Fig. 1. Switch and Direct-Threaded Interpreters

this sequence implementation is stored into the code array, replacing the original bytecode of the first instruction in the sequence. Figure 2 illustrates the creation of an instruction sequence implementation and shows an abstract *source code* representation of the resulting inlined instruction sequence implementation.

Inline-threading improves performance by reducing the overhead due to dispatch. This is particularly effective for sequences of simple instructions, which have a high *dispatch to real work* ratio. Unfortunately, not all instructions can be inlined. Inlining instructions that contain C function calls, hidden (compiler generated) function calls, or even simple conditional expressions (in presence of some compiler optimizations) can prevent inlining[2].

3 The Difficulty of Inline-Threading Java

Lazy Loading and Preparation In Java, classes are dynamically loaded. The JVM Specification [9] allows a virtual machine to *eagerly* or *lazily* load classes (or anything in between). But this flexibility does not extend to *class initialization*[3]. Class initialization must occur at specific execution points, such as the first invocation of a static method or the first access to a static field of a class. Lazily loading classes has many advantages: it saves memory, reduces network traffic, and reduces startup overhead.

Inline-threading requires analyzing a bytecode array to determine *basic blocks*, allocating and preparing implementation sequences, and lastly preparing a code array. As this preparation is time and space consuming, it is advisable to only prepare methods that will actually be executed. This can be achieved through lazy method preparation.

[2] The target of a relative branch instruction might be invalid in the inlined instruction copy.

[3] Class initialization consists of initializing static fields and executing static class initializers.

(a) Instruction Implementations

```
ICONST_1_START: *sp++ = 1;
ICONST_1_END: goto **(pc++);

INEG_START: sp[-1] = -sp[-1];
INEG_END: goto **(pc++);

DISPATCH_START: goto **(pc++);
DISPATCH_END: ;
```

(c) Inlined Instruction Sequence

```
ICONST_1 body:  *sp++ = 1;
INEG body    :  sp[-1] = -sp[-1];
DISPATCH body:  goto **(pc++);
```

(b) Sequence Computation

```
/* Implement the sequence ICONST_1 INEG */
size_t iconst_size = (&&ICONST_1_END - &&ICONST_1_START);
size_t ineg_size = (&&INEG_END - &&INEG_START);
size_t dispatch_size = (&&DISPATCH_END - &&DISPATCH_START);

void *buf = malloc(iconst_size + ineg_size + dispatch_size);
void *current = buf;

memcpy(current, &&ICONST_START, iconst_size); current += iconst_size;
memcpy(current, &&INEG_START, ineg_size); current += ineg_size;
memcpy(current, &&DISPATCH_START, dispatch_size);
...
/* Now, it is possible to execute the sequence using: */
goto **buf;
```

Fig. 2. Inlining a Sequence

Performance Issue Lazy preparation (and loading), which aims at improving performance, can pose a performance problem within a multi-threaded[4] environment. The problem is that, in order to prevent corruption of the internal data structure of the virtual machine, concurrent preparation of the same method (or class) on distinct Java threads should not be allowed.

The natural approach, for preventing concurrent preparation, is to use synchronization primitives such as *pthread mutexes*[5]. But, this approach can have a very high performance penalty; in a naive implementation, it adds synchronization overhead to every method call throughout a program's execution, which is clearly unacceptable, specially for multi-threaded Java applications.

Broken Sequences An important performance factor of inline-threading is the length of inlined instruction sequences. Longer sequences reduce the dispatch-to-real work ratio and lead to improved performance. Lazy class initialization mandates that the first call to a static method (or access to a static field) must cause initialization of a class. This implies (in a naive Java virtual machine implementation) that instructions such as `GETSTATIC` must use a conditional to test whether the target class must be initialized prior to performing the static field access. If initialization is required, a call to the initialization function must be made. The conditional and the C function call prevent inlining of the `GETSTATIC` instruction.

What we would like, is to use two versions of the `GETSTATIC` instruction, as shown in Figure 3 and replace the slow synchronized version by the fast version

[4] Note that *multi-threading* is a concurrent programming technique which is inherently supported in Java, whereas *inline-threading* is an instruction dispatch technique.

[5] POSIX Threads mutual exclusive locks.

after initialization. Unfortunately this does not completely solve our performance problem. Even though this technique eliminates synchronization overhead from most executions of the `GETSTATIC` instruction, it inhibits the removal of dispatch code in an instruction which has very little *real work* to do. In fact, the cost can be as high as the execution of two additional dispatches. To measure this, we compare the cost of two inline-threaded instruction sequences that only differ in their respective use of `ILOAD` and `GETSTATIC` in the middle of the sequence.

Synchronized `GETSTATIC`	Unsynchronized `GETSTATIC`

`GETSTATIC_INIT: /* pseudo-code */`
 	
`pthread_mutex_lock(...);`
 	
`/* lazily load class */`
`...`
`/* conditional & function call */`
`if (must_initialize)`
`  initialize_class(...);`
`/* do the real work */`
`*sp++ = class.static_field;`
`/* replace by fast version */`
`code[pc -1] = &&GETSTATIC_NO_INIT;`
 	
`pthread_mutex_unlock(...);`
 	
`/* dispatch */`
`goto **(pc++);`
	`GETSTATIC_NO_INIT: /* pseudo-code */`
	
	`/* do the real work */`
	`*sp++ = class.static_field;`
	
	`/* dispatch */`
	`goto **(pc++);`

Fig. 3. `GETSTATIC` With and Without Initialization

Broken Sequence Cost If we had the sequence of instructions `ICONST2--ILOAD-IADD`, we could build a single inlined sequence for these three instructions, adding a single dispatch at the end of this sequence. Cost: $3\times realwork+1\times dispatch$. If, instead, we had the sequence of instructions `ICONST2-GETSTATIC--IADD`, we would not be allowed to create a single inlined sequence for the three instructions. This is because, in the prepared code array, we would need to put 3 distinct instructions: `ICONST2`, `GETSTATIC_INIT`, and `IADD`, where the middle instruction cannot be inlined. Even though the `GETSTATIC_INIT` will eventually be replaced by the more efficient `GETSTATIC_NO_INIT`, the performance cost, after replacement, will remain: $3 \times realwork + 3 \times dispatch$. So, the overhead of a broken sequence can be as high as two additional dispatches.

Two-Values Replacement In reality, the problem is even a little deeper. The *pseudo-code* of Figure 3 hides the fact that `GETSTATIC_INIT` needs to replace two values, in the code array: the instruction opcode and its operand. The idea is that we want the address of the static variable as an operand (not an indirect pointer) to achieve maximum efficiency, as shown in Figure 4. But this pointer is unavailable at the time of preparation of the code array, as lazy class loading only takes place later, within the implementation of the `GETSTATIC_INIT` instruction.

Fast Instruction	Code Array

GETSTATIC_NO_INIT:
{ int *pvalue =
(pc++)->pvalue;
*sp++ = *pvalue;
}
/* dispatch */
goto **(pc++);
	/* Initially */
	...
	[GETSTATIC_INIT]
	[POINTER_TO_FIELD_INFO]
	...
	/* After first execution */
	...
	[GETSTATIC_NO_INIT]
	[POINTER_TO_FIELD]
	...

Fig. 4. Two-Values Replacement in Code Array

Replacing two values without synchronization creates a race condition. Here is a short illustration of the problem. A first Java thread reads both initial values, does the instruction work, then replaces the first of the two values. At this exact point of time (before the second value is replaced), a second Java thread reads the two values (instruction and operand) from memory. The second Java thread will thus get the fast instruction opcode and the old field info pointer. This can of course lead to random execution problems.

4 Preparation Sequences

In this section, we first introduce an incomplete solution to the problems discussed in Section 3, then we introduce our *preparation sequences* technique.

Incomplete Solution The two problems we face are *two-values replacement*, and *shorter sequences* caused by the slow preparation version of instructions such as `GETSTATIC`. Of course, there is a simple solution to two-values replacement that consists of using single-value replacement[6] and an indirection in the *fast* version of instructions, as shown in Figure 5. Note how this implementation differs from Figure 4; in particular the additional `fieldinfo` indirection. This simple solutions comes at a price, though: that of an additional indirection in a very simple instruction. Furthermore, this solution does not solve the shorter sequences problem.

The Basic Idea Instead, we propose a solution that solves both problems. This solution consists of adding *preparation sequences* in the code array. The basic idea of preparation sequences is to duplicate certain portions of the code array, leaving fast inlined-sequences in the main copy, and using slower, synchronized, non-inlined preparation version of instructions in the copy. Single-value replacement is then used to direct control flow appropriately.

[6] Single-value replacement does not require synchronization when there is a single aligned word to change.

Fast Instruction with Indirection	Code Array

GETSTATIC_NO_INIT:
{ int *pvalue =
(pc++)->fieldinfo->pvalue;
*sp++ = *pvalue;
}
/* dispatch */
goto **(pc++);
	/* Initially */
	...
	[GETSTATIC_INIT]
	[POINTER_TO_FIELD_INFO]
	...
	
	/* After first execution */
	...
	[GETSTATIC_NO_INIT]
	[POINTER_TO_FIELD_INFO]
	...

Fig. 5. Single-Value Replacement of GETSTATIC

Single-Instruction Preparation Sequence *Preparation sequences* are best explained using a simple illustrative example. We continue with our GETSTATIC example. We assume, for the moment, that the GETSTATIC instruction is preceded and followed by non-inlinable instructions, in the code array. An appropriate instruction sequence would be MONITORENTER-GETSTATIC-MONITOREXIT, as neither monitor instruction is inlinable.

Figure 6, (a) and (b), illustrates the initial content of a prepared code array containing the above 3-instructions sequence. The GETSTATIC *preparation sequence* appears at the end of the code array. The initial content of the code array is as follows. After the MONITORENTER, we insert a GOTO instruction followed by two operands: (i) the address of the GETSTATIC preparation sequence, and (ii) an additional word (initially NULL) which will eventually hold a pointer to the static field. At the end of the code array, we add a preparation sequence, which consists of 3 instructions (identified by a *) along with their operands.

Figure 6, (c) to (f), shows the implementation of four instructions: GOTO, REPLACE, GETSTATIC_INIT, and GETSTATIC_NO_INIT. Notice that in the preparation sequence, the GETSTATIC_NO_INIT opcode is used as an operand to the REPLACE instruction.

We used labels (e.g. SEQUENCE_1:) to represent the address of specific opcodes. In the real code array, absolute addresses are stored in opcodes such as [@ SEQUENCE_1].

Here is how execution proceeds. On the first execution of this portion of the code, the MONITORENTER instruction is executed. Then, the GOTO instruction is executed, reading its destination in the following word. The destination is the SEQUENCE_1 label, or more accurately, the GETSTATIC_INIT opcode, at the head of the *preparation sequence*.

The GETSTATIC_INIT instruction then reads two operands: (a) a pointer to the field information structure, and (b) a destination pointer for storing a pointer to the resolved static field. It then proceeds normally, loading and initializing the class, and resolving the field, if it hasn't yet been done[7]. Then, it stores the address of the resolved field in the destination location. Notice that, in the present case, this means that the pointer-to-field will *overwrite* the NULL value

[7] Each field is only resolved once, yet there can be many GETSTATIC instructions accessing this field. The same holds for class loading and initialization.

(a) Original Bytecode

```
...
...
MONITORENTER
GETSTATIC
INDEXBYTE1
INDEXBYTE2
MONITOREXIT
...
...
```

(b) Initial Content of Code Array

```
                ...
                ...
                [MONITORENTER]*
OPCODE_1:       [GOTO]*
                [@ SEQUENCE_1]
OPERAND_1:      [NULL_POINTER]
NEXT_1:         [MONITOREXIT]*
                ...
                ...
SEQUENCE_1:     [GETSTATIC_INIT]*
                [POINTER_TO_FIELDINFO]
                [@ OPERAND_1]
                [REPLACE]*
                [GETSTATIC_NO_INIT]
                [@ OPCODE_1]
                [GOTO]*
                [@ NEXT_1]

Opcodes followed by a * are
instructions.
```

(c) GETSTATIC_INIT

```
GETSTATIC_INIT:
{ fieldinfo_t *fieldinfo =
    (pc++)->fieldinfo;
  int **destination =
    (pc++)->ppint;
  pthread_mutex_lock(...);
  /* lazily load and initialize
     class, and resolve field */
  ...
  /* store field information in
     code array */
  *destination =
     fieldinfo->pvalue;
  /* do the real work */
  *sp++ = *(fieldinfo->pvalue);
  pthread_mutex_unlock(...);
}
/* dispatch */
goto **(pc++);
```

(d) GETSTATIC_NO_INIT

```
GETSTATIC_NO_INIT:
/* skip address */
pc++;
{ int *pvalue =
    (pc++)->pvalue;
  /* do the real work */
  *sp++ = *pvalue;
}
/* dispatch */
goto **(pc++);
```

(e) GOTO

```
GOTO:
{ void *address =
    (pc++)->address;
  pc = address;
}
/* dispatch */
goto **(pc++);
```

(f) REPLACE

```
REPLACE:
{ void *instruction =
    (pc++)->instruction;
  void **destination =
    (pc++)->ppvoid;
  *destination =
    instruction;
}
/* dispatch */
goto **(pc++);
```

Fig. 6. Single GETSTATIC Preparation Sequence

at label OPERAND_1. Finally, it executes the *real work* portion of the instruction, and dispatches to the next instruction.

The next instruction is a special one, called REPLACE, which simply stores the value of its first operand into the address pointed-to by its second operand. In this particular case, a pointer to the GETSTATIC_NO_INIT *instruction* will be stored at label OPCODE_1, overwriting the former GOTO instruction pointer. This constitutes, in fact, our *single-value* replacement.

The next instruction is simply a GOTO used to exit the *preparation sequence*. It jumps to the instruction following the original GETSTATIC bytecode, which in our specific case is the MONITOREXIT instruction.

Future executions of the same portion of the code array will see a GETSTATIC_NO_INIT instruction (at label OPCODE_1), instead of a GOTO to the *preparation sequence*. Two-values replacement is avoided by leaving the GOTO operand address in place. Notice how the implementation of GETSTATIC_NO_INIT in Figure 6 (d) differs from the implementation in Figure 4, by an additional pc++ to skip the address operand.

Some Explanations Our single-instruction preparation sequence has avoided two-values replacement by using an extra word to permanently store a *preparation sequence address* operand, even though this address is useless after initial execution.

This approach adds some overhead in the fast version of the *overloaded* instruction; that of a program-counter increment, to skip the preparation sequence address. One could easily question whether this gains any performance improvement over that of using an indirection as in Figure 5. This will be answered by looking at longer preparation sequences.

The strangest looking thing, is the usage of 3 distinct instructions in the preparation sequence. Why not use a single instruction with more operands? Again, the answer lies in the implementation of longer *preparation sequences*.

Full Preparation Sequences We now proceed with the full implementation of preparation sequences. Our objective is two-fold: (a) we want to avoid two-values replacement, and (b) we want to build longer inlined instruction sequences for our inlined-threaded interpreter, for reducing dispatch overhead as much as possible.

To demonstrate our technique, we use the three instruction sequence: `ICONST2-GETSTATIC-ILOAD`.

Figure 7, (a) and (b), shows the initial state of the code array, the content of the dynamically constructed `ICONST2-GETSTATIC-ILOAD` inlined instruction sequence, some related instruction implementations, and the content of the code array after first execution.

This works similarly to the single-instruction preparation sequence, with two major differences: (a) the jump to the *preparation sequence* initially replaces the `ICONST_2` instruction, instead of the `GETSTATIC` instruction, and (b) the `REPLACE` instruction stores a pointer to an *inlined instruction sequence*, overwriting the `GOTO` instruction.

Here is how execution proceeds in detail. On the first execution of this portion of the code, the `GOTO` instruction is executed. Its destination is the `ICONST_2` opcode, at the head of the *preparation sequence*.

Next, the `ICONST_2` instruction is executed. Next, the `GETSTATIC_INIT` instruction reads two operands: (a) a pointer to the field information structure, and (b) a destination pointer for storing a pointer to the resolved static field. It then proceeds normally, loading and initializing the class, and resolving the field, if it hasn't yet been done. Then, it stores the address of the resolved field in the destination location. Finally, it executes the *real work* portion of the instruction, and dispatches to the next instruction.

The next instruction is a `REPLACE`, which simply stores a pointer to the dynamically *inlined instruction sequence* `ICONST2-GETSTATIC-ILOAD` at label `OPCODE_1`, overwriting the former `GOTO` instruction, and performing a *single-value* replacement.

Next, the `ILOAD` instruction is executed. Finally, the tail `GOTO` exits the *preparation sequence*.

Future executions of the same portion of the code array will see the `ICONST2--GETSTATIC-ILOAD` instruction sequence (at label `OPCODE_1`), as shown in Figure 7(f). Notice that the *inlined implementation* of `GETSTATIC_NO_INIT` in Figure 7(c) does not add any overhead to the fast implementation shown in Figure 4.

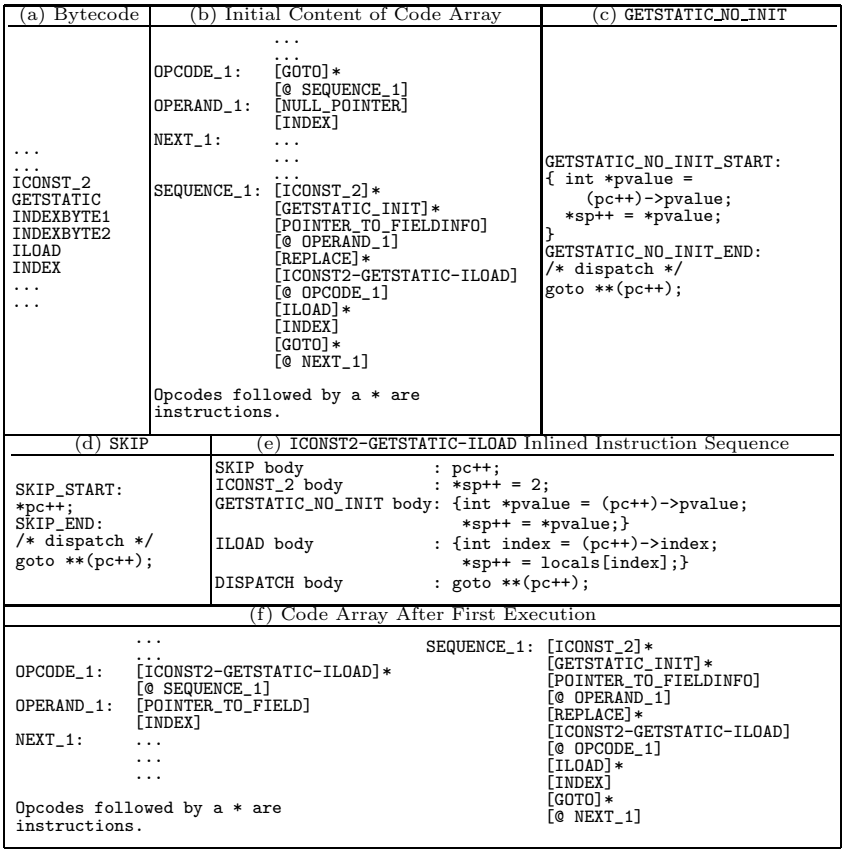

Fig. 7. Full Preparation Sequence

Thus, we have achieved our goals. In particular, we have succeeded at inlining an instruction sequence, even though it had a complex two-modes (preparation / fast) instruction in the middle, while avoiding two-values replacement. All of this with minimum overhead in post-first execution of the code array.

Detailed Preparation Procedure Preparation of a code array, in anticipation of inline-threading, proceeds as follows:

1. Instructions are divided in three groups: inlinable, two-modes-inlinable (such as `GETSTATIC`), and non-inlinable.
2. Basic blocks (determined by control-flow and non-inlinable instructions) are identified.
3. Basic blocks of inlinable instructions, without two-modes-inlinable instructions, are inlined normally.
4. Every basic block containing two-modes-inlinable instructions causes the generation of an additional *preparation sequence* at the end of the code array, and the construction of a related *inlined instruction sequence*.

The construction of a *preparation sequence* proceeds as follows:

1. Instructions are copied sequentially into the preparation sequence.
   - Inlinable instructions and their operands are simply copied as-is.
   - The *preparation* version of two-modes-inlinable instructions is copied into the preparation sequence, along with the destination address for resolved operands.
2. A `REPLACE` instruction with appropriate operands is inserted just after the last two-modes-inlinable instruction.
3. A final `GOTO` instruction with appropriate operand is added at the end of the preparation sequence.

The motivation for adding the replace instruction just after the the last two-modes-inlinable instruction, is that it is the earliest safe place to do so. Replacing sooner could cause the execution (on another Java thread) of the fast version of an upcoming two-modes instruction before it is actually prepared. Replacing later can also be a problem, specially if some upcoming inlinable instruction is a conditional (or unconditional) branch instruction. This is because, if the branch is taken, then single-value replacement will not take place, forcing the next execution to take the slow path[8].

The construction of an *inlined instruction sequence* containing two-modes-inlinable instructions proceeds as follows:

1. The body of the `SKIP` instruction is copied at the beginning of the sequence implementation.

[8] Multiple executions of the same *preparation sequence* is allowed, but suffers from high dispatch overhead. It *can* happen in the normal operation of the inline-threaded interpreter as the result of an exception thrown before single-value replacement, while executing a *preparation sequence*.

2. Then, all instruction bodies are sequentially copied.
3. Finally, the body of the `DISPATCH` instruction is copied at the end of the sequence implementation.

Note that a single preparation sequence can contain multiple two-modes instructions. Yet, on the fast execution path, there is a single program-counter increment (i.e. `SKIP` body) per inlined instruction sequence.

5 Experimental Results

We have implemented 3 flavors of threaded interpretation, in the *SableVM* framework [6]: switch-threading, direct-threading and inline-threading. *Switch-threading* differs from simple switch-based bytecode interpretation in that it is applied on a *prepared* code array of word-size elements. To avoid the two-values replacement problem, *single-instruction preparation sequences* are in use within the switch-threaded and direct-threaded engines.

We have performed execution time measurements with *SableVM* to measure the efficiency of inline-threading Java, using our technique. We have performed our experiments on a 1.5 GHz Pentium IV based Debian GNU/Linux workstation with 1.5 Gb RAM, and a 7200 RPM disk, running SPECjvm98 benchmarks and two object-oriented applications: Soot version 1.2.3[9] and SableCC version 2.17.3[10].

In a first set of experiments, we have measured the relative performance of the switch-threaded, direct-threaded and inline-threaded engines. Results are shown in Table 1. To do these experiments, three separate versions of *SableVM* were compiled with identical configuration options, except for the interpreter engine type.

Table 1. Inline-Threading Performance Measurements

benchmark	switch-threaded	direct-threaded	inline-threaded
compress	317.72 sec.	281.78 sec. (1.13)	131.64 sec. (2.41) (2.14)
db	132.15 sec.	119.17 sec. (1.11)	87.64 sec. (1.51) (1.36)
jack	45.65 sec.	46.78 sec. (0.98)	38.16 sec. (1.20) (1.23)
javac	110.10 sec.	105.24 sec. (1.05)	89.37 sec. (1.23) (1.17)
jess	74.79 sec.	68.12 sec. (1.10)	53.57 sec. (1.40) (1.27)
mpegaudio	285.77 sec.	242.90 sec. (1.18)	136.97 sec. (2.09) (1.77)
mtrt	142.87 sec.	115.34 sec. (1.24)	100.39 sec. (1.42) (1.15)
raytrace	166.19 sec.	134.06 sec. (1.24)	113.55 sec. (1.46) (1.18)
soot	676.06 sec.	641.96 sec. (1.05)	548.13 sec. (1.23) (1.17)
sablecc	40.12 sec.	36.95 sec. (1.09)	26.09 sec. (1.54) (1.41)

[9] `http://www.sable.mcgill.ca/soot/`
[10] `http://www.sablecc.org/`

Columns of Table 1 contain respectively: (a) the name of the executed benchmark, (b) the execution time in seconds using the switch-threaded engine, (c) the execution time in seconds using the direct-threaded engine, and the speedup over the switch-threaded engine in parentheses, and (d) the execution time in seconds using the inline-threaded engine, and the speedup over both switch-threaded and direct-threaded engines respectively in parentheses.

The *Inline-threaded* engine does deliver significant performance improvement. It achieves a speedup of up to 2.41 over the switch-threaded engine. The smallest measured speedup, over the *fastest* of the two other engines on a benchmark, is of 1.15 on the *mtrt* benchmark, where it still delivers a speedup of 1.42 over the second engine.

It is important to note that the switch-threaded engine already has some advantages over a pure switch-based bytecode interpreter. It benefits from word alignment and other performance improving features of the *SableVM* framework. So, it is likely that the performance gains of inline-threading over pure bytecode interpretation are even bigger than those measured against switch-threading.

In a second set of tests, we measured the performance improvement due to the inlining of two-modes instructions (e.g. `GETSTATIC`), within the inlined-threaded engine. To do so, we compiled a version of SableVM with a special option that prevents inlining of two-modes instructions, and compared its speed to the normal inline-threaded engine. Results are shown in Table 2.

Table 2. Preparation Sequences Performance Measurements

benchmark	shorter sequences	full sequences	speedup
compress	195.50 sec.	131.64 sec.	1.49
db	108.22 sec.	87.64 sec.	1.24
jack	40.46 sec.	38.16 sec.	1.06
javac	99.99 sec.	89.37 sec.	1.12
jess	62.91 sec.	53.57 sec.	1.17
mpegaudio	157.38 sec.	136.97 sec.	1.16
mtrt	105.39 sec.	100.39 sec.	1.05
raytrace	133.12 sec.	113.55 sec.	1.17
soot	617.42 sec.	548.13 sec.	1.13
sablecc	32.35 sec.	26.09 sec.	1.24

Columns of Table 2 contain respectively: (a) the name of the executed benchmark, (b) the execution time in seconds using the special inline-threaded engine that does not inline two-modes instructions, (c) the execution time in seconds using the normal inline-threaded engine implementing full preparation sequences, and (d) the speedup achieved by the normal inline-threaded engine over the atrophied version.

Our performance measurements show that the speedup due to longer sequences ranges between 1.05 and 1.49, which is quite significant.

6 Related Work

The most closely related work to the work of this paper is the work of I. Piumarta and F. Riccardi in [11]. We have already discussed the inline-threading technique introduced in this paper in Section 2. Our work builds on top of this work, by introducing techniques to deal with *multi-threaded* execution environments, and inlining of *two-modes* instructions.

Inline-threading, in turn, is the result of combining the Forth-like *threaded interpretation* technique [5] (which we have already discussed in Section 2) with the idea of *template-based* dynamic compilation [2, 10]. The main advantage of inline-threading over that of template based compilation is its simplicity and portability.

A related system for dynamic code generation is that of *vcode*, introduced by D. Engler [4]. The *vcode* system is an architecture-neutral runtime assembler. It can be used for implementing *just-in-time* compilers. It is in our future plans to experiment with *vcode* for constructing an architecture-neutral *just-in-time* compiler for *SableVM*, offering an additional choice of performance-portability tradeoff.

Other closely related work is that of *dynamic patching*. The problem of potential high cost synchronization costs for concurrent modification of executed code is also faced by dynamically adaptive Java systems. In [3], M. Cerniac *et al.* describe a technique for *dynamic inline patching* (a similar technique is also described in [8]). The main idea is to store a self-jump (a jump instruction to itself) in the executable code stream before proceeding with further modifications of the executable code. This causes any concurrent thread executing the same instruction to spin-wait for the completion of the modification operation.

Our technique of using explicit synchronization in preparation sequences and single value replacement has the marked advantage of causing no spin-wait. Spinning can have, in some cases, a highly undesirable side effect, that of *almost* dead-locking the system when the spinning thread has much higher priority than the *code patching* thread. This is because, while it is spinning, the high priority does not make any progress in code execution and, depending on the thread scheduling policy of the host operating system, might be preventing the patching thread from making noticeable progress.

7 Conclusions

In this paper we have explained the difficulty of using the *inline-threaded* interpretation technique in a Java interpreter. Then, we introduced a new technique, *preparation sequences*, that not only makes it possible, but also effective. This technique uses efficient single-word replacement for managing lazy class-loading and preparation in a multi-threaded environment, and increases the length of *inlined instruction sequences*, reducing *dispatch* overhead.

We then presented our experimental results, showing that an inline-threaded interpreter engine, implementing our technique, achieves significant performance

improvements over that of switch-threaded and direct-threaded engines. Our results also show that longer inlined instructions sequences, due solely to preparation sequences, can yield a speedup ranging between 1.05 and 1.49.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.
2. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pages 149–159. ACM Press, 1996.
3. M. Cierniak, G.-Y. Lueh, and J. N. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 13–26, Vancouver, British Columbia, June 2000. ACM Press.
4. D. R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pages 160–170. ACM Press, 1996.
5. A. M. Ertl. A portable Forth engine. http://www.complang.tuwien.ac.at/forth /threaded-code.html.
6. E. M. Gagnon and L. J. Hendren. SableVM:A Research Framework for the Efficient Execution of Java Bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM-01)*, pages 27–40. USENIX Association, Apr. 2001.
7. J. L. Hennessy and D. A. Patterson. *Computer architecture (2nd ed.): a quantitative approach*. Morgan Kaufmann Publishers Inc., 1996.
8. K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the ACM SIGPLAN '00 conference on Object-oriented programming, systems, languages, and applications*, pages 294–310. ACM Press, 2000.
9. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
10. F. Noel, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based runtime specialization: Implementation and experimental study. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages 1998*, pages 132–142. IEEE Computer Society Press, Apr. 1998.
11. I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300. ACM Press, June 1998.