A Survey of Instruction Dispatch Techniques for Byte-Code Interpreters

Markku Rossi Kengatharan Sivalingam
mtr@iki.fi siva@iki.fi

May 6, 1996

Abstract

Some interpreters use an intermediate machine code, *byte-code*, for program execution. This paper surveys techniques used to implement instruction dispatching in byte-code interpreters. For comparison purpose, we implemented a sample interpreter which includes different instruction dispatch techniques. In this article, we compare machine and compiler dependencies and the efficiency of these methods and analyse the results.

1 Introduction

Some interpreters evaluate source code directly without transforming them into any other form. Here, the source code is parsed one string at a time and evaluated by the interpreter. Even if this is considerably slower than native code execution, there are many advantages, such as no compilation and portability, in using this technique. UNIX™ shells, for example, use this method for evaluating their source code.

Another kind of language interpreters transform their source code into an intermediate format, called *byte-code*. The byte-code format differs from the native code in being simple and machine-independent. Also, compiling source code into byte-code is faster than into native code. The execution of byte-code is more efficient than the above mentioned direct interpretation technique. Java™ and GNU-Emacs are among many implementations which use the byte-code technique.

Byte-code interpreters define an abstract machine, also referred to as *virtual machine*, to execute byte-code programs. A virtual machine and its byte-code are analogous to a CPU and it's machine code. The interpretation of a virtual machine instruction consists of the following steps [1]:

- accessing instruction arguments
- performing the function of the instruction
- *dispatching* the next instruction

An instruction dispatching process includes the following functions: fetching, decoding and starting. Accessing and dispatching instructions play a main role in the efficiency of an interpreter.

In this paper, we study various instruction dispatch techniques currently in use and compare their efficiency. In Section 2, we explore different instruction dispatch techniques in detail, and in Section 3 we compare the efficiency and various dependencies of these techniques. Section 4 concludes the study.

2 Instruction Dispatch Techniques

This section presents a few traditional instruction dispatch techniques with sample implementations. The final part explores an exotic new technique, called *memcpy method*, proposed by Kenneth Oksanen [2].

2.1 Indirect Dispatch (Trampoline & Function Pointer)

In trampoline & function pointer method, a trampoline array is initialized with primitive function pointers. The array is later used in a loop to execute byte-code program instructions.

```
void (*primitives[])() = {/* pointers for primitive functions */};
unsigned char *program = {/* program */};
unsigned int pc = 0;

void interpreter()
{
  while (1)
    (*primitives[program[pc++]])();
}
```

2.2 Direct Threading Using Tail-Calls

In direct threading, the instructions of a program are replaced with the addresses of routines that implement them. During instruction dispatch, the address of an instruction routine is fetched and the program calls the corresponding routine.

A direct implementation of this method is possibly very efficient for instruction dispatching. But in ANSI C or other languages, the direct implementation of this method is not possible. Because these languages do not have first-class labels and do not guarantee tail-call optimization. The following code segment shows how this method could be implemented in C using tail-calls.

```
void (*program[])() = {/* program */};
unsigned int pc = 0;

void prim_add()
{
  /* Do addition. */
  (*program[pc++])();
}

void prim_sub()
{
  /* Do subtraction. */
  (*program[pc++])();
}
...

void interpreter()
{
  pc = 0;
  (*program[pc++])();
}
```

2.3 Instruction Dispatch Using Switch

In this method, instructions are represented as integer tokens and a big `switch` statement within the `main()` function is used to select the appropriate routine for an instruction. The whole interpreter, with the implementations of all instructions, must be in one function.

```
unsigned char *program = {/* program */};
unsigned int pc = 0;

void interpreter()
{
  while (1)
    {
      switch (program[pc++])
        {
```

```
            case add:
              /* Do addition. */
              break;

            case sub:
              /* Do subtraction. */
              break;
              ...
            }
        }
    }
```

2.4 Direct Call Dispatch: JUMPS1 & JUMPS2

The direct threading can also be implemented using GNU C's *labels as values* [4] as shown below.

```
    void interpreter()
    {
      void *program[] = {&&add.../* rest of the program */};
      void *ip = program;

      goto *ip++;

      add:
       /* Do addition. */
       goto *ip++;

      sub:
       /* Do subtraction. */
       goto *ip++;
           ...
    }
```

For comparison, we implemented two different versions of this technique, jumps1 and jumps2. Both versions use a jump-table which consists of the op-codes of the program. The jumps1 version loops through the table and executes corresponding primitives found in the program. But the jumps2 uses GNU C's `goto` and jumps directly to the next primitive.

2.5 Instruction Dispatch Using `memcpy`

The memcpy dispatching method gets rid of almost all dispatching overhead yet without using a native code compiler. It uses dynamically allocated memory regions to achieve high execution efficiency. First, sequences of code between pairs of labels of corresponding primitive implementations are copied to a contiguous memory area. Then the data of processor's write buffer is flushed and the operating system is requested to treat the copied code as executable program memory. Finally, the program jumps to the address of the copied code [2].

The implementation of this method uses system calls provided by modern UNIX systems. The required memory can be allocated by `mmap(2)` and bytecode instructions are copied using `memcpy(3)`. The method of flushing the write buffer of processor varies from system to system.

```
unsigned char *program = {/* program */};
unsigned int codesize;

struct
{
  void *start;
  void *end;
 } primitives[256] = {/* offsets for primitive implementations */};

void interpreter()
{
  void *code = *allocate_executable_memory();
  void *code_end = code;
  int i;

  /* Compile program. */
  for (i = 0; i < codesize; i++)
    {
      unsigned char prim = program[i];
      unsigned int delta = primitives[prim].end - primitives[prim].start;

      memcpy(code_end, primitives[prim].start, delta);
      code_end += delta;
    }
  flush_data_cache_writes();

  /* Execute program. */
  goto *code;

  /* Define primitives. */
op_add_start:
  /* Do addition. */
op_add_end:

op_sub_start:
```

```
  /* Do subtraction. */
op_sub_end:
  ...
}
```

The scheme given here eliminates the memory traffic and data cache occupation caused by the accesses to the code in traditional threading techniques. Because this method does not have parameter/return value passing overhead, might suit even better for short sequences of code.

The memcpy method does not require a separate program counter (*pc*) which has to be incremented after every instruction. But since copying does not handle immediate values particularly well, it has to maintain a data pointer (*dp*) rather frequently. In these cases, the maintenance of the real hardware *pc* and *dp* causes a few machine instructions overhead for some byte-code primitives [3]. Note that the handling of constant data and corresponding *dp* updates have been omitted in the sample code given for this method.

Figure 1 shows the results obtained from the performance test, a Fibonacci machine that uses memcpy for dispatching, conducted by Kenneth Oksanen [3]. The virtual machine of this test caches the topmost item of the stack in a register. Because of this optimization, the execution time of primitives gets smaller and the effect of instruction dispatching becomes crucial in program execution. As a consequence of this, the differences between the memcpy and other techniques grow even further.

3 Comparison of Dispatch Techniques

We implemented some of the instruction dispatch techniques, discussed in the previous section, and compared them on different platforms. In this section, we analyse the results taken from the sample runs of our byte-code virtual machine which consists of three different tests: a factorial calculation, a Fibonacci number calculation and a prime counting program.

3.1 Requirements

The system requirements of the dispatch methods can be divided into three different categories: processor, compiler and operating system-related dependencies. Table 1 lists the requirements for each of the explained dispatch methods; not all dispatch methods have any special requirements for the underlying system, they can be implemented with standard system utilities. However some methods, like memcpy, require features from all categories.

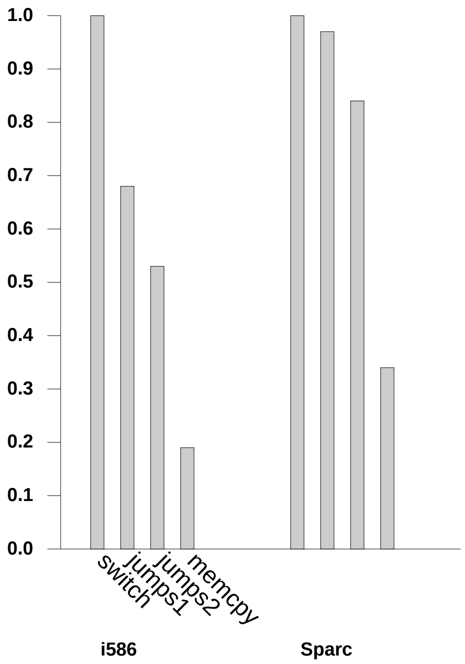

Figure 1: Relative Performance of Memcpy Method

3.1.1 Processor

The memcpy dispatch method relocates the actual machine code during the program execution. This approach requires that the processor can execute position independent code; the code cannot have any relative references away from the moved code-block. A care must also be taken when performing function calls from the moved code-block. In some cases, the functions calls must be avoided or they must be performed through function pointers.

METHOD	GCC	RELOCATABLE CODE	EXECUTION OF DYNAMICALLY ALLOCATED MEMORY REGIONS
trampoline	-	-	-
switch	-	-	-
computed goto	required	-	-
memcpy	required	required	required

Table 1: System Requirements

3.1.2 Compiler

The direct call threading and the memcpy methods require a C-compiler which supports computed-goto statements, gotos to offsets which are computed during the program execution. The trampoline and switch approaches do not have any special compiler dependencies, they can be implemented using any usable C-compilers.

3.1.3 Operating System

The memcpy approach requires an operating systems which allows executing dynamically allocated memory regions. On Unix-like systems this can be implemented with the `mmap(2)` function; a continuous executable memory region is mapped to the memory space of the process and the generated code is copied to and executed in this region.

3.2 Performance

Table 2 shows the results from the performance tests. Five dispatch methods were implemented and they were tested on five different processors. The dispatch methods were tested by using a simplified Java-bytecode which implements integer arithmetics and comparisons, gotos and subroutines. Table 2 shows the relative speeds of the different methods. The slowest method is given the value 1 and other faster methods have values less than 1, representing the relative time used to complete the test task.

Processor	trampoline	switch	jumps1	jumps2	memcpy
i586/120	1.00	0.81	0.71	0.62	0.63
PowerPC	1.00	0.75	0.82	0.76	0.43
Sparc 10/402	1.00	0.81	0.86	0.77	0.55
Alpha	1.00	0.84	0.73	0.67	0.41
Mips	1.00	0.85	0.88	0.85	0.48

Table 2: Relative Performance

For all processor architectures, the trampoline method was the slowest and the memcpy method was the fastest. The performance of the other dispatch methods varied between these two extremes, but the switch method was almost always slower than the direct call threading methods jumps1 and jumps2.

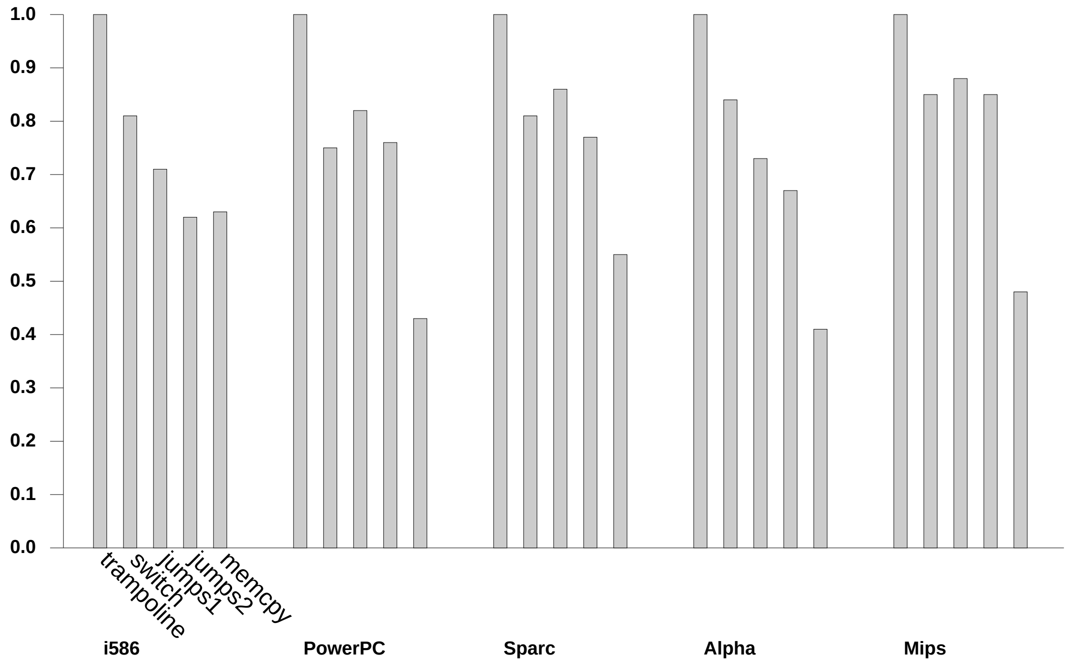

Figure 2: Relative Performance

4 Conclusion

Our comparison shows that the memcpy dispatch technique is the fastest in most of the platforms we used. However, the implementation of this technique is very much dependent on compiler, operating system and processor. The switch method does not depend on any of these categories but is proved to be slow.

Our test clearly reveals that the selection of instruction dispatch technique for a byte-code interpreter depends on the demands of the program. The designer might prefer the computed goto or memcpy method if the performance of the interpreter is critical but not the portability. The trampoline or switch method can be chosen as the dispatch technique if the performance of the byte-code interpreter is not crucial in the overall performance of the application.

The test also showed that the instruction dispatching is vital for the efficiency of interpreters. By carefully selecting the instruction dispatch technique, the execution efficiency of an interpreter can be improved considerably. Also, an interpreter can increase it's efficiency by exploiting more than one dispatch technique for program execution. For example, the interpreter may use memcpy method to execute time-critical parts of the program, and use a more traditional technique for primitives that cannot be copied in the particular computer.

References

[1] A. Ertl, Stack Caching for Interpreters, *SIGPLAN '95 Conference on Programming Language*, 1995.

[2] Personal communication with Kenneth Oksanen `<cessu@cs.hut.fi>`, 1996.

[3] Notes from the discussion between Xavier Leroy and Kenneth Oksanen on instruction dispatch techniques, 1996.

[4] R. Stallman, Using and Porting GCC.