Optimizing direct threaded code by selective inlining

Ian Piumarta and Fabio Riccardi

INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

email: ian.piumarta@inria.fr
fabio.riccardi@inria.fr

Abstract

Achieving good performance in bytecoded language interpreters is difficult without sacrificing both simplicity and portability. This is due to the complexity of dynamic translation ("just-in-time compilation") of bytecodes into native code, which is the mechanism employed universally by high-performance interpreters.

We demonstrate that a few simple techniques make it possible to create highly-portable dynamic translators that can attain as much as 70% the performance of optimized C for certain numerical computations. Translators based on such techniques can offer respectable performance without sacrificing either the simplicity or portability of much slower "pure" bytecode interpreters.

Keywords: bytecode interpretation, threaded code, inlining, dynamic translation, just-in-time compilation.

1 Introduction

Bytecoded languages such as Smalltalk [Gol83], Caml [Ler97] and Java [Arn96, Lin97] offer significant engineering advantages over more conventional languages: higher levels of abstraction, dynamic execution environments with incremental debugging and code modification, compact representation of executable code, and (in most cases) platform independence.

The success of Java is due largely to its promise of platform independence and compactness of code. The compactness of bytecodes has important advantages for network computing where code must downloaded "on-demand" for execution on an arbitrary platform and operating system while keeping bandwidth requirements to a minimum. The disadvantage is that bytecode interpreters typically offer lower performance than compiled code, and can consume significantly more resources.

Most modern virtual machines perform some degree of dynamic translation to improve program performance [Deu84]. Such techniques significantly increase the complexity of the virtual machine, which must be tailored for each hardware architecture in much the same way as a conventional compiler's back-end. This increases development costs (requiring specific knowledge about the target architecture and the time for writing specific code), and reduces reliability (by introducing more code to debug and support).

Some of these languages (Caml for example) also have more traditional compilers that produce high-performance native code, but this defeats the advantages that come with platform independence and compactness.

We propose a novel dynamic retranslation technique that can be applied to a certain class of virtual machines. This technique delivers high performance, up to 70% that of optimized C. It is easy to "retrofit" to existing virtual machines, and requires almost no effort to port to a new architecture.

This paper continues as follows. The next section gives a brief survey of bytecode interpretation mechanisms, providing a context for the remainder of the paper. Our novel dynamic retranslation technique is explained in Section 3. Section 4 presents the results of applying the technique to two interpreters: the small RISC-like interpreter that inspired this work, and a "production" virtual machine for Objective Caml. The last two sections contrast our technique with related work and present some concluding remarks.

2 Background

Interpreter performance can depend heavily on the representation chosen for executable code, and the mechanism used to dispatch opcodes. This section describes some of the common techniques.

2.1 Pure bytecode interpreters

The inner loop of a pure bytecode interpreter is very simple: fetch the next bytecode and dispatch to the implementation using a `switch` statement. Figure 1 shows a typical pure bytecode interpreter loop, and an array of bytecodes that calculate '`3 + 4`' (we will use this as a running example).

The interpreter is an infinite loop containing a `switch` statement to dispatch successive bytecodes. Each `case` in the body of the `switch` implements one bytecode, and passes control to the next bytecode by `break`ing out of the `switch` to pass control back to the start of the infinite loop.

Assuming the compiler optimizes the jump chains from the `break`s through the implicit jump at the end of the `for` body back to its beginning, the overheads associated with this approach are as follows:

compiled code:

```
unsigned char code[] = { ...,
  bytecode_push3,
  bytecode_push4,
  bytecode_add, ... };
```

bytecode implementations:

```
unsigned char *instructionPointer = code - 1;
for (;;) {
  unsigned char bytecode = *++instructionPointer;
  switch (bytecode) {
  /* ... */
  case bytecode_push3:
    *++stackPointer = 3;
    break;
  case bytecode_push4:
    *++stackPointer = 4;
    break;
  case bytecode_add:
    --stackPointer;
    *stackPointer += stackPointer[1];
    break;
  /* ... */
  }
}
```

Figure 1: Pure bytecode interpreter.

compiled code:

```
void *code[] = { ...,
  &&opcode_push3,
  &&opcode_push4,
  &&opcode_add, ... };
```

opcode implementations:

```
/* dispatch next instruction */
#define NEXT() goto **++instructionPointer

  void **instructionPointer = code - 1;
  /* start execution: dispatch first opcode */
    NEXT();
  /* opcode implementations... */
  opcode_push3:
    *++stackPointer = 3;
    NEXT();
  opcode_push4:
    *++stackPointer = 4;
    NEXT();
  opcode_add:
    --stackPointer;
    *stackPointer += stackPointer[1];
    NEXT();
  /* ... */
```

Figure 2: Direct threaded code.

- increment the `instructionPointer`;
- fetch the next bytecode from memory;
- a redundant range check on the argument to `switch`;
- fetch the address of the destination `case` label from a table;
- jump to that address;

and then at the end of each bytecode:

- jump back to the start of the `for` body to fetch the next bytecode.

Eleven machine instructions must be executed on the PowerPC to perform the `push3` bytecode. Nine of these instructions are dedicated to the dispatch mechanism, including two memory references and two jumps (among the most expensive instructions on modern architectures).

Pure bytecoded interpreters are easy to write and understand, and are highly portable — but rather slow. In the case where most bytecodes perform simple operations (as in the `push3` example) the majority of execution time is wasted in performing the dispatch.

2.2 Threaded code interpreters

Threaded code [Bel73] was popularized by the Forth programming language [Moo70]. There are various kinds of threaded code, the most efficient of which is generally *direct threading* [Ert93].

Bytecodes are simply integers: dispatch involves fetching the next opcode (bytecode), looking up the address of the associated implementation (either in an explicit table, or implicitly using `switch`) and then transferring control to that address. Direct threaded code improves performance by eliminating this table lookup: executable code is represented as a sequence of opcode implementation addresses, and dispatch involves fetching the next opcode (implementation address) and jumping directly to that address.

An additional optimization eliminates the centralized dispatch. Instead of returning to a central dispatch loop, each direct threaded opcode's implementation ends with the code required to dispatch the next opcode. The direct threaded version of our '`3 + 4`' example is shown in Figure 2.[1]

Execution begins by fetching the address of the first opcode's implementation from the compiled code and then jumping to that address. Each opcode performs its own work, and then dispatches to the next opcode implied by the compiled code. (Hence the name: control flow "threads" its way through the opcodes in the order implied by the compiled code, without ever returning to a central dispatch loop.)

The overheads associated with threaded code are much lower than those associated with a pure bytecode interpreter. For each opcode executed, the only additional overhead is dispatching to the next opcode:

- increment the `instructionPointer`;
- fetch the next opcode address from memory;
- jump to that address.

Five machine instructions are required to implement `push3` on the PowerPC. Three of these are associated with opcode dispatch, with only one memory reference and one jump.

We have saved six instructions over the "pure bytecode" approach. Most importantly we have saved one memory reference and one jump instruction (both of which are expensive).

2.3 Dynamic translation to threaded code

The benefits of direct threaded code can easily be obtained in a bytecoded language by translating the bytecodes into

[1] The threaded code examples are written using the first-class labels provided by GNU C. The expression "`void *addr = &&`*label*" assigns the address (of type "`void *`") of the statement attached to the given *label* to `addr`. Control can be transferred to this location using a `goto` that dereferences the address: "`goto *addr`". Note that `gcc`'s first-class labels are *not* required to implement these techniques: the same effects can be achieved with a couple of macros containing a few lines of `asm`.

translation table:

```
void *opcodes[];
/* ... */
opcodes[bytecode_push3] = &&opcode_push3;
opcodes[bytecode_push4] = &&opcode_push4;
opcodes[bytecode_add]   = &&opcode_add;
/* ... */
```

dynamic translator:

```
unsigned char *bytecodePointer = firstBytecode;
void **opcodePointer = translatedCodeForFunction;
while (moreBytecodesToTranslate)
  *opcodePointer++ = opcodes[*bytecodePointer++];
```

Figure 3: Dynamic translation of bytecodes into threaded code.

direct threaded code before execution. This is illustrated in Figure 3. The translation loop reads each bytecode, looks up the address of its implementation in a table, and then writes this address into the direct threaded code.

The only complication is that most bytecode sets have *extension bytes*. These provide additional information that cannot be encoded within the bytecode itself: branch offsets, indices into literal tables or environments, and so on. These extension bytes are normally placed inline in the translated threaded code by the translator, immediately after the threaded opcode corresponding to the bytecode.

Translation to threaded code permits other kinds of optimization. For example, Smalltalk provides four bytecodes for pushing an implicit integer constant (between `-1` and +2) onto the stack. The translator loop could easily translate these as a single `pushInteger` opcode followed by the constant to be pushed as an inline operand. The same treatment can be applied to other kinds of literal quantity, relative branch offsets, and so on. Another possibility is "partial decoding", where the translator loop examines an "overloaded" bytecode at translation time, and translates it into one of several threaded opcodes.

The translator loop must be aware of the kind of operand that it is copying. A relative offset, for example, might require modification or scaling during the translation loop.

It is possible to make an approximate evaluation of this approach in a realistic system. Squeak [Ing97] is a portable "pure bytecode" implementation of Smalltalk-80; it performs numerical computations at approximately 3.7% the speed of optimized C. BrouHaHa [Mir87] is a portable Smalltalk virtual machine that is very similar to the Squeak VM, except that it dynamically translates bytecodes into direct threaded code for execution [Mir91]. BrouHaHa performs the same numerical computations at about 15% the speed of optimized C. Both implementations have been carefully hand-tuned for performance; the essential difference between them is the use of dynamic translation to direct threaded code in BrouHaHa.

2.4 Optimizing common bytecode sequences

Bytecodes can typically only represent 256 operations. Threaded opcodes can represent many more, since they are encoded as pointers. Translating bytecodes into threaded code therefore gives us the opportunity to make arbitrary transformations on the executable code. One such transformation is to detect common sequences of bytecodes and translate them as a single threaded "macro" opcode; this macro opcode performs the work of the *entire* sequence of original bytecodes. For example, the bytecodes "push literal, push variable, add, store variable" can be translated into a single "add-literal-to-variable" opcode in the threaded code.

Such optimizations are effective because they avoid the overhead of the multiple dispatches that are implied by the original bytecodes (but elided within the macro opcode). A single macro opcode that is translated from a sequence of N original bytecodes avoids $N-1$ opcode dispatches at execution time.

This technique is particularly important in cases where the bytecodes are simple (as in the '`3 + 4`' example), when the implementation of each bytecode can be as short as a single register-register machine instruction. The cost of threading can often be significantly larger than the cost of "useful" execution. If three instructions must be executed to dispatch to the next opcode then the overhead for this threading for '`3 + 4`' is 75% (four useful instructions executed and 12 instructions for dispatching the threaded opcodes). This overhead drops to 43% when the operation is optimized into a single macro opcode (four useful instructions and 3 instructions for threading).[2]

Dispatching to opcode implementations at non-contiguous addresses also undermines code locality, causing unnecessary processor pipeline stalls and inefficient utilization of the instruction cache and TLBs. Combining common sequences of bytecodes into a single macro opcode considerably reduces these effects. The compiler will also have a chance to make inter-bytecode optimizations (within the implementation of the single macro opcode) that are impossible to make between the implementations of the individual bytecodes.

Determining an appropriate set of common bytecode sequences is not difficult. The virtual machine can be instrumented to record execution traces, and a simple offline analysis will reveal the likely candidates. The corresponding pattern matching and macro opcode implementations can then be incorporated *manually* into the VM. For example, such analysis has been applied to an earlier version of the Objective Caml bytecode set, resulting in a new set of bytecodes that includes several "macro-style" operations.

2.5 Problems with static optimization

The most significant problem with this static approach is that the number of possible permutations of even the shortest common sequences of consecutive bytecodes is prohibitive. For example, Smalltalk provides 4 bytecodes to push the most popular integer constants (minus one through two), and bytecodes to load and store 32 temporary and 256 "receiver" variables. Manually optimizing the possible permutations for incrementing and decrementing a variable by a small constant would require the translator to implement 2304 explicit special cases. This is clearly unreasonable.

The problem is made more acute since different applications running on the same virtual machine will favor different sequences of bytecodes. Statically chosing a single "optimal" set of common sequences is therefore impossible.

Our technique focuses on making this choice at *runtime*, which allows the set of common sequences to be nearly optimal for the particular application being run.

[2] "Instruction counting" is not a very accurate way to estimate the savings, since the instructions that we avoid are some of the most expensive to execute.

```
dynamic_opcode_push3_push4_add:
  *++stackPointer = 3;
  *++stackPointer = 4;
  stackPointer--;
  *stackPointer += stackPointer[1];
  goto **++instructionPointer;
```

Figure 4: Equivalent macro opcode for `push3`, `push4`, `add`.

```
int nfibs(int n)
{
  return (n < 2)
    ? 1
    : nfibs(n - 2) + nfibs(n - 1) + 1;
}
```

Figure 5: Benchmark function in C.

```
nfibs:  push    r1      ; r1 saved during call
        move    #2 r1   ; if (arg < 2)
        jge     r0 r1 @
        =cont
        pop     r1      ; restore r1
        return  #1      ; return 1
cont:   move    r0 r1   ; else arg -> r1
        sub     #1 r0   ; call nfibs(arg-1)
        call    @
        =nfibs
        swap    r0 r1   ; nfibs(arg-1) -> r1, arg -> r0
        sub     #2 r0   ; call nfibs(arg-2)
        call    @
        =nfibs
        add     r1 r0   ; nfibs(arg-2) + 1 -> r0
        add     #1 r0   ; nfibs(arg-1) + nfibs(arg-2) + 1 -> r0
        pop     r1      ; restore r1
        return  r0      ; return nfibs(arg-1) + nfibs(arg-2) + 1

start:  move    #32 r0  ; call nfibs(32)
        call    @
        =nfibs
        print   r0      ; print result
        halt            ; stop
```

Figure 6: Threaded code for `nfibs` benchmark, before inlining.

3 Dynamically rewriting opcode sequences

We generate implementations for common bytecode sequences *dynamically.* These implementations are available as new macro opcodes, where a single such macro opcode replaces the several threaded opcodes generated from the original common bytecode sequence. These dynamically generated macro opcodes are executed in precisely the same manner as the interpreter's predefined opcodes; the original execution mechanism (direct threading) requires no modification at all. The transformation can be performed either during bytecode-to-threaded code translation, or as a separate pass over already threaded code.

Figure 4 shows the equivalent C for a dynamically generated threaded opcode for the sequence of three bytecodes needed to evaluate the '`3 + 4`' example.

The translator concatenates the compiled C implementations for several intrinsic threaded opcodes, each one corresponding to a bytecode in the sequence being optimized. Since this involves relocating code, it is only safe to perform this concatenation for threaded opcodes whose implementation is position independent. In general there are three cases to consider when concatenating opcode implementations:

- A threaded opcode cannot be inlined if its implementation contains a call to a C function, where the destination address is relative to the processor's PC. Such destination addresses would be invalidated as they are copied to form the new macro opcode's implementation.
- Any threaded opcode that changes the flow of control through the threaded code must only appear at the end of a translated sequence. This is because different paths through the sequence might consume different numbers of inline arguments.
- Any threaded opcode that is a branch destination can only appear at the beginning of a macro opcode, since incorporating it into the middle of a macro opcode would delete the branch destination in the final threaded code.

The above can be simplified to the following rule: we only consider *basic blocks* for inlining, where a basic block begins with a jump destination and ends with either a jump destination or a change of control flow. For inlining purposes, opcodes that contain a C function call are considered to be single-opcode basic blocks. (This restriction can be relaxed if the target architecture and/or the compiler used to build the VM uses absolute addresses for function call destinations.)

Our technique was designed for (and works best with) fine-grained opcodes, where the implementations are short (typically a few machine instructions) and therefore the cost of opcode dispatch dominates. The next section presents an example in such a context.

3.1 Simple example

We will illustrate our technique by applying it to a simple "RISC-like" virtual machine executing the "`nfibs`" function, as shown in Figure 5.[3]

Our example interpreter implements a register-based execution model. It has a handful of "registers" for performing arithmetic, and a stack that is used for saving return addresses and the contents of clobbered registers during subroutine calls. The direct threaded code has two kinds of inline operand: instruction pointer-relative offsets for branch destinations, and absolute addresses for function call destinations.

The interpreter translates bytecodes into threaded code in two passes. It makes a first pass over the bytecodes, expanding them into threaded opcodes with no inlining, exactly as explained in Section 2.3. Figure 6 shows a symbolic listing of the `nfibs` function, implemented for our example interpreter's opcode set, after this initial translation into threaded code.

Bytecode operands are placed inline in the threaded code during translation. For example, the offset for the `jge` opcode and the `call` destinations are placed directly in the opcode stream, immediately after the associated opcode. These are represented as the pseudo-operand '`@`' in the fig-

[3]This doubly-recursive function has the interesting property that its result is the number of function calls required to calculate the result.

```
nfibs: %1  → { push r1, move #2 r1, jge r0 r1 @, <thr> }
       =cont
       %2  → { pop r1, return #1, <thr> }
cont:  %3  → { move r0 r1, sub #1 r0, call @, <thr> }
       =nfibs
       %4  → { swap r0 r1, sub #2 r0, call @, <thr> }
       =nfibs
       %5  → { add r1 r0, add #1 r0, pop r1, return r0, <thr> }
```

Figure 7: Threaded code for `nfibs` benchmark, after inlining. The implementations of the new macro opcodes are shown on the right.

ure, and appear on a separate line in the code prefixed with '='.

After this initial translation to threaded code, a second pass performs inlining on the threaded code: basic blocks are identified, used to dynamically generate new threaded macro opcodes, and the corresponding original sequences of threaded opcodes are replaced with single macro opcodes. The rewriting of the threaded code can be performed *in-situ*, since optimizing an opcode sequence will always result in a shorter sequence of optimized code; there is no possibility of overwriting an opcode that has not yet been considered for inlining.

Figure 7 shows the code for the `nfibs` function after inlining has taken place. The function has been reduced to five threaded macro opcodes (shown as '`%1`' through '`%5`'), each replacing a basic block in the original code. The implementation of each new macro opcode is the *concatenation* of the implementations of the opcodes that it replaces. These new implementations are written in a separate area of memory called the *macro cache*. Five such implementations are required for `nfibs`, and are shown within curly braces in the figure. Each one ends with a copy of the implementation of the pseudo-opcode `<thr>`, which is the threading operation to dispatch the next opcode.

Inline arguments are copied verbatim, except for `cont` (a jump offset) which is adjusted appropriately by the translator. (These inline arguments are used by the macro opcode implementations at the points marked with '`@`' in the figure.)

To help with the identification of basic blocks, we divide our threaded opcodes into four classes, as follows:

> `INLINE` — the opcode's implementation can be inlined into a macro opcode without restriction (the arithmetic opcodes belong to this class);
>
> `PROTECT` — the implementation contains a C function call and therefore cannot be inlined (the `print` opcode belongs to this class);
>
> `FINAL` — the opcode changes the flow of control and therefore defines the end of a basic block (e.g. the `call` opcode);
>
> `RELATIVE` — the opcode changes the flow of control and therefore defines the end of a basic block (e.g. the conditional branch `jge`).

The only difference between `FINAL` and `RELATIVE` is the way in which the opcode's inline operand is treated. In the first case the operand is absolute, and can be copied directly into the final translated code. In the second case the operand is relative to the current threaded program counter, and so must be adjusted appropriately in the final translated code.

Figure 8 shows the translator code that initializes the threaded opcode table, along with representative implementations of several of our threaded opcodes (each of the four classes of threaded opcode is represented).

```
#define PUSH(X) (*++sp = (long)(X))
#define POP()   (*sp--)
#define GET()   ((long)(*++ip)) /* read inline operand */
#define NEXT()  goto **++ip     /* dispatch next opcode */

#define PROTECT  (0x00) /* never expanded */
#define INLINE   (1<<0) /* expanded */
#define FINAL    (1<<1) /* expanded, ends a basic block */
#define RELATIVE (1<<2) /* expanded, ends a basic block,
                           offset follows */

#define OP(NAME, NARGS, FLAGS) \
 case NAME: \
 { \
  info[op].nargs = NARGS;  info[op].flags = FLAGS; \
  info[op].addr = &&start_##NAME; \
  info[op].end = &&end_##NAME; \
  info[op].size = (int)&&end_##NAME - (int)&&start_##NAME; \
  if (!initialIP) break; \
 start_##NAME:
  /* opcode body */

#define END(NAME) \
 end_##NAME: NEXT() \
 }

/* initialize rather than execute (see macro 'OP') */
initialIP = 0;
for (int op = FIRST_OPCODE; op <= LAST_OPCODE; ++op)
 switch (op) {
  OP(add_1_r0,  0, INLINE)   { r0 += 1;  END(add_1_r0); }
  OP(mul_r1_r0, 0, INLINE)   { r0 *= r1; END(mul_r1_r0); }
  OP(jge_r0_r1, 1, RELATIVE) { register long offset = GET();
                               if (r0 >= r1) ip += offset;
                               END(jge_r0_r1); }
  OP(call,      1, FINAL)    { register long dest = GET();
                               PUSH(ip);
                               ip = (void **)dest - 1;
                               END(call); }
  OP(return_r0, 0, FINAL)    { ip = (void **)POP();
                               END(return_r0); }
  OP(print_r0,  0, PROTECT)  { printf("%ld\n", r0);
                               END(print_r0); }
 default:
  fprintf(stderr, "panic: op %d is undefined!\n", op);
  abort();
 }
```

Figure 8: Opcode table initialization.

The translator's inlining loop is shown in Figure 9. It is not as complex as it might first appear. `code` is a pointer to the translated threaded code, which is rewritten *in-situ*. `in` and `out` are indices into `code` pointing to the next opcode to be copied (or inlined) and the location to which it will be copied, respectively (`in >= out` at all times).

The loop considers each `in` opcode for inlining: the inlining loop is entered only if both the current opcode and the opcode following it can be inlined. If this is not the case, the opcode at `in` is copied (along with any inline arguments) directly to `out`.

`nextMacro` is a pointer to the next unused location in the macro cache. The inlining loop first writes this address to `out` (it represents the threaded opcode for the macro implementation that is about to be generated), and then copies the compiled *implementations* of opcodes from `in` into the macro cache. The inlined threaded opcodes are not copied, although any inline arguments that are encountered *are* copied directly to `out`.

The inlining loop continues until it copies the implementation of an opcode that explicitly ends a basic block (`FINAL` or `RELATIVE`), or until the next opcode is either non-inlinable

```
int in = 0, out = 0;
while (thisOp = code[in]) {
  int  nextIn = in + 1 + info[thisOp].nargs;
  long nextOp = code[nextIn];
  relocations[in] = out;
  if (info[thisOp].flags == INLINE  &&
      info[nextOp].flags != PROTECT &&
      !destination[nextIn]) {
    /* CAN INLINE: create new macro opcode at nextMacro */
    void *ep = nextMacro;
    code[out++] = (long)ep;  /* new macro opcode */
    while (info[thisOp].flags != PROTECT) {
      icopy(info[thisOp].addr, ep, info[thisOp].size);
      ep += info[thisOp].size;
      ++in;  /* skip opcode */
      if (info[thisOp].flags == RELATIVE) {
        patchList[patchIndex++] = out;    /* locn of offset */
        code[out++] = in + 1 + code[in];  /* original destn */
        ++in;
      } else {
        for (int i = info[thisOp].nargs; i > 0; --i)
          code[out++] = code[in++];
      }
      if (info[thisOp].flags == FINAL    ||
          info[thisOp].flags == RELATIVE ||
          destination[in])
        break;  /* end of basic block */
      thisOp = code[in];
    }
    /* copy threading operation */
    icopy(info[thr].addr, ep, info[thr].size);
    ep += info[thr].size;
    nextMacro = ep;
  } else {
    /* CAN'T INLINE: copy opcode and inline arguments */
    code[out++] = (long)info[thisOp].addr;
    ++in;  /* skip opcode */
    if (info[thisOp].flags == RELATIVE) {
      patchList[patchIndex++] = out;
      code[out++] = in + 1 + code[in];
      ++in;
    } else {
      /* copy literal arguments */
      for (int i = info[thisOp].nargs; i > 0; --i)
        code[out++] = code[in++];
    }
  }
}
```

Figure 9: Dynamic translator loop.

(`PROTECTED`) or a branch destination (implicitly ending the current basic block). The translator then appends the implementation of the pseudo-opcode `thr`, which is the "threading" operation itself. Finally, the `nextMacro` location is updated ready for the next inlining operation.

The translator loop uses an array of flags "`destination`" to identify branch destinations within the threaded code. This array is easily constructed during the translator's first pass, when bytecodes are expanded into non-inlined threaded code. The loop also creates two arrays, `relocations` and `patchList`, that are used to recalculate relative branch offsets.[4]

The inlining loop concatenates opcode implementations using the `icopy` function, shown in Figure 10. This function is similar to `bcopy` except that it also synchronizes the processor's instruction and data caches to ensure that the new macro opcode's implementation is executable. It contains the only line of platform-dependent code in our interpreter.

[4] The branch destination identification and relative offset recalculation are not shown here. These can be seen in the full source code for the example interpreter (see the Appendix).

```
static inline void icopy(void *source, void *dest, size_t size)
{
  bcopy(source, dest, size);
  while (size > 0) {
#if defined(PPC)
    asm ("dcbst 0,%0; sync; icbi 0,%0; isync" :: "r"(p));
#elif defined(__sparc)
    asm ("flush %0; stbar" :: "r"(p));
#elif defined(__i386)
    /* no-op */
#elif defined(...)
    ...
#endif
  dest += 4;  size -= 4;
  }
}
```

Figure 10: The `icopy` function, containing the single line of platform-dependent code.

3.2 Saving space

Translating multiple copies of the same opcode sequences would waste space. We therefore keep a cache of dynamically generated macro opcodes, keyed by a hash value computed from the incoming (unoptimized) opcodes during translation. In the case of a cache hit we reuse the existing macro opcode in the translated code, and immediately reclaim the macro cache space occupied by the newly translated version. In the case of a cache miss, the newly generated macro opcode is used in the translated code and the hash table updated to include the new opcode. This ensures that we never have more than one macro opcode corresponding to a given sequence of unoptimized opcodes.

4 Experimental results

We are particularly interested in the performance benefits when dynamic inlining is applied to interpreters with fine-grain instruction sets. Nevertheless, we were also curious to see how the technique would perform when applied to an interpreter having a more coarse-grained bytecode set. We took measurements in both of these contexts, using our own RISC-like interpreter and the widely-used (but less suited) interpreter for the Objective Caml language.

4.1 Fine-grained opcodes

Our RISC-like interpreter has an opcode set similar to that presented in Section 3.1. It can be configured (at compile time) to use bytecodes, direct threaded code, or direct threaded code with dynamically-generated macro opcodes. The performance of two benchmarks was measured using this interpreter: the function-call intensive Fibonacci benchmark presented earlier (`nfibs`), and a memory intensive, function call free, prime number generator (`sieve`).

Table 1 shows the number of seconds required to execute these benchmarks on several architectures (133MHz Pentium, SparcStation 20, and 200MHz PowerPC 603ev). The figures shown are for a simple bytecode interpreter, the same interpreter performing translation into direct threaded code, direct threaded code with dynamic inlining of common opcode sequences, and the benchmark written in C and compiled with the same optimization options (`-O2`) as our interpreter. The final column shows the performance of the inlined threaded code compared to optimized C.

nfibs

machine	*bytecode*	*threaded*	*inlined*	*C*	*inlined/C*
Pentium	63.2	37.1	22.3	11.1	49.8%
Sparc	93.6	51.4	24.9	18.1	72.7%
PowerPC	40.6	20.3	10.4	6.0	57.7%

sieve

machine	*bytecode*	*threaded*	*inlined*	*C*	*inlined/C*
Pentium	25.1	17.6	13.2	4.6	34.8%
Sparc	41.5	23.9	15.1	4.4	29.1%
PowerPC	24.0	12.8	8.7	2.4	27.6%

Table 1: `nfibs` and `sieve` benchmark results for the three architectures tested. The final column shows the speed of the inlined threaded code relative to optimized C.

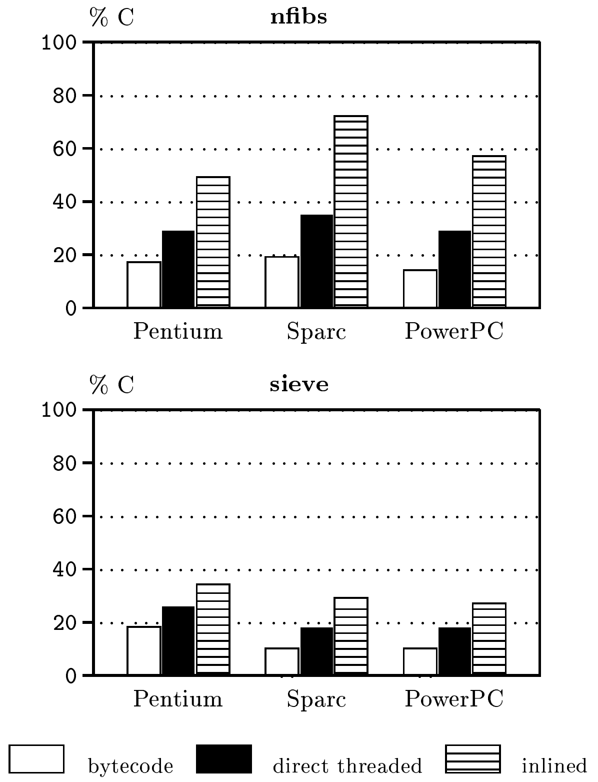

Figure 11: Benchmark performance relative to optimized C.

`nfibs` spends much of its time performing arithmetic between registers. Memory (stack) operations are performed only during function call and return.

Our interpreter allocates the first few VM registers in physical machine registers whenever possible. The opcodes that perform arithmetic are therefore typically compiled into a single machine instruction on the Sparc and PowerPC. These two architectures show a marked improvement in performance when common sequences are inlined into single macro opcodes, due to the significantly reduced ratio of opcode dispatch to "real" work. The effect is less pronounced on the Pentium, which has so few machine registers that all the VM registers must be kept in memory. Each arithmetic opcode compiles into several Pentium instructions, and therefore the ratio of dispatch overhead to real work is lower than for the RISC architectures.

We observe a marked improvement (approximately a factor of two) between successive versions of the interpreter for `nfibs`.

`sieve` shows a less pronounced improvement because it spends the majority of its time performing memory operations. The contribution of opcode dispatch to the overall execution time is therefore smaller than with `nfibs`.

It is also interesting to observe the performance of each version of the interpreter relative to that of optimized C. Figure 11 shows that `nfibs` gains approximately 14% the speed of optimized C when moving from a bytecoded representation to threaded code. The gain when moving from threaded to inlined threaded code is more dependent on the architecture: approximately 20% for the Pentium, and 38% for the Sparc. The gains for `sieve` are both smaller and less dependent on the architecture: approximately 9% at each step, for all three architectures.

4.2 Objective Caml

We also applied our technique to the Objective Caml bytecode interpreter, in order to obtain realistic measurements of its performance and overheads in a less favorable environment.

Objective Caml was chosen because the design and implementation of the interpreter's core is clean and simple, and so understanding it before making the required modifications did not present a significant challenge. Furthermore it is a fully-fledged system that includes a bytecode compiler, a benchmark suite, and some large applications. This made it easier to collect meaningful statistics.

The interpreter is also equipped with a mechanism to bulk-translate the bytecodes into threaded code at startup (on those platforms that support it).[5] We needed only to extend this initial translation phase to perform the analysis of opcode sequences, generate macro opcode implementations, and rewrite the threaded code *in-situ* to use these dynamically-generated macro opcodes. Implementing our technique for the Caml virtual machine took one day. There were only two small details that required careful attention.

The first was the presence of the `SWITCH` opcode. This performs a multi-way branch, and is followed in the threaded code by an inline table mapping values onto branch offsets. We added a special case to our translator loop to handle this opcode.

The second was the existence of a handful of opcodes that consume two inline arguments (a literal and a relative offset). We introduced a new opcode class `RELATIVE2` for these, which differs from `RELATIVE` only by copying an additional inline literal argument before the offset in the translator loop.

Our translation algorithm was identical in all other respects to the one presented in Section 3.

We ran the standard Objective Caml benchmark suite[6] with our modified VM (see Table 2). The VM was instrumented to gather statistics relating to execution speed,

[5] It uses `gcc`'s first-class labels to do this portably.

[6] `ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/benchmarks/objcaml.tar.gz`

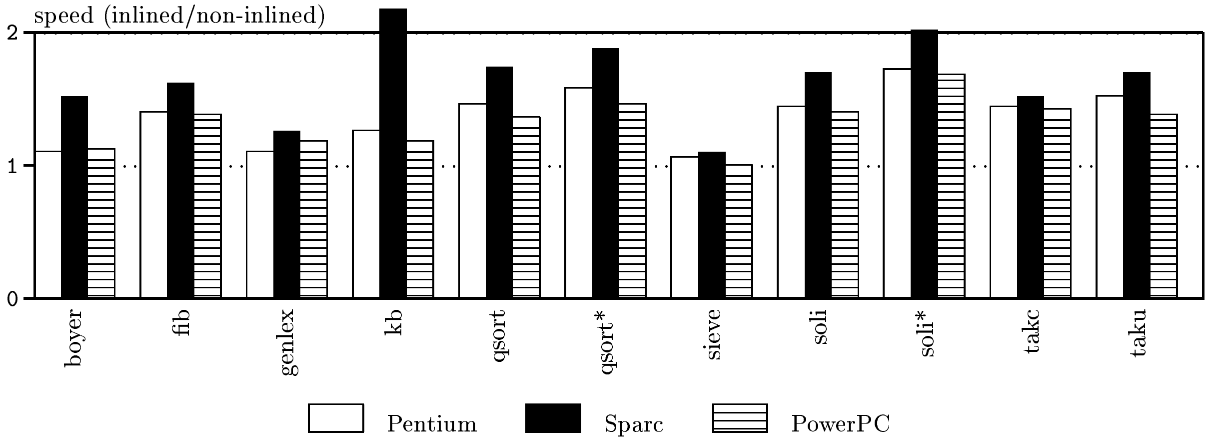

Figure 12: Objective-Caml benchmark results for the three architectures tested. The vertical axis shows the performance relative to the original (non-inlining) interpreter. Asterisks indicate versions of the benchmarks compiled with array bounds checking disabled.

boyer	term processing, function calls
fib	integer arithmetic, function calls (1 arg)
genlex	lexing, parsing, symbolic processing
kb	term processing, function calls, functionals
qsort	integer arrays, loops
sieve	integer arithmetic, list processing, functionals
soli	puzzle solving, arrays, loops
takc	integer arithmetic, function calls (3 args, curried)
taku	integer arithmetic, function calls (3 args, tuplified)

Table 2: Objective Caml benchmarks.

memory usage, and the characteristics of dynamically generated macro opcodes.

Figure 12 shows the performance of the benchmarks after inlining, relative to the original performance without inlining.

It is important to note that the Objective Caml bytecode set has already been optimized statically, as described in Section 2.4 [Ler98]. Any further improvements are therefore due mainly to the elimination of dispatch overhead in common sequences that are particular to each application. Virtual machines whose bytecode sets have not been "statically" optimized in this way would benefit more from our technique.

We can see from the figure that the majority of benchmarks benefit from a significant performance advantage after inlining. In most cases the inlined version runs more than 50% faster than the original, with two of the benchmarks running twice as fast as the original non-inlined version on the Sparc.

It is clear that the improvements are related to the processor architecture. This is probably due to differences in the cost of the threading operation. On the Sparc, for example, avoiding the pipeline stalls associated with threading seems to make a significant difference.

Figure 13 shows the final size of the macro cache for each benchmark on the Sparc, plotted as a factor of the size of the original (unoptimized) code. The final macro cache

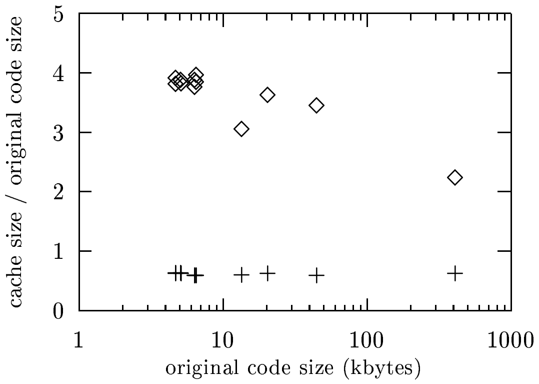

Figure 13: Macro cache size (diamonds) and optimized threaded code size (crosses), plotted as a factor of the original code size.

sizes vary slightly for each architecture, since they depend on the size of the bytecode implementations. However, the shape is the same in each case. The average ratios of original bytecode size to the macro cache size show that the cost is between three and four times the size of the original code on the Sparc. (The ratio is almost identical for the PowerPC, and slightly smaller for the Pentium.)

We observe that this ratio decreases gradually as the original code size increases. This is to be expected, since larger bodies of code will tend to reuse macro opcodes rather than generating new ones. We tested this by translating the bytecoded version of the Objective Caml compiler: 421,532 bytes of original code generated 941,008 bytes of macro opcode implementation on the Sparc. This is approximately 2.2 times the size of the original code, and is shown as the rightmost point in the graph.

Inlined threaded code is always smaller than the original code from which is generated. Figure 13 also shows the final optimized code size for each benchmark. We observe that the ratio is independent of the size of the benchmark. This is also to be expected, since the reduction in size is dependent on the average number of opcodes in a common sequence and the density of the corresponding macro opcodes in the final code. These depend mainly on the characteristics of the language and its opcode set.

Some systems have a long-lived object memory, and generate new executable code at runtime. A realistic implementation for such systems would recycle the macro cache space, and possibly use profiling to optimize only popular areas of the program. For example, the 68040LC emulator found on Macintosh systems performs dynamic translation of 68040 into PowerPC code; it normally requires only 250Kb of cache in which the most commonly used translated code sequences are stored [Tho95]. A similar (fixed) cache size is effective in the BrouHaHa Smalltalk system [Mir97].

Translation speed is also an important factor. To measure this we ran the Object Caml bytecode compiler (a much larger program than any of the benchmarks) with our modified interpreter. The 105,383 opcodes of the Objective Caml compiler are translated in 0.22 seconds on the Sparc, a rate of 480,000 opcodes per second. The inlining interpreter executes the compiler at a rate of 2.4 million opcodes per second. Translation is therefore approximately five times slower than execution.[7]

5 Related work

BrouHaHa and Objective Caml have both demonstrated the benefits of creating specialized macro opcodes that perform the work of a sequence of common opcodes. In Objective Caml this led to a new bytecode set. In BrouHaHa the standard Smalltalk-80 bytecodes are translated into threaded code for execution; the detection of a limited number of pre-determined common bytecode sequences is performed during translation, and a specialized opcode is substituted in the executable code. Our contribution is the extension of this technique to dynamically analyze and generate implementations for new macro opcodes at runtime.

Several systems use concatenation of pre-compiled sequences of code at runtime [Aus96, Noe98], but in a completely different context. Their precompiled code sequences are generic "templates" that can be parameterized at runtime with particular constant values.

A template-based approach is also used in some commercial Smalltalk virtual machines that perform dynamic compilation to native code [Mir97]. However, this technique is complex and requires a significant effort to implement the templates for a new architecture.

An interesting system for portable dynamic code generation is vcode [Eng96], an architecture-neutral runtime assembler. It generates code that approaches the performance of C on some architectures. Its main disadvantage is that retrofitting it to an existing virtual machine requires a significant amount of effort — certainly more than the single day that was required to implement our technique in a production virtual machine. (Our simple `nfibs` benchmark runs about 40% faster using vcode, compared to our RISC-like inlined threaded code virtual machine.)

Superoperators [Pro95] are a technique for specializing a bytecoded C interpreter according to the program that it is to execute. This is possible because the specialized interpreter is generated at the same time as the compiled (bytecoded) representation of the program. A compile-time analysis of the program chooses likely candidates for superoperators, which are then implemented as new interpreter bytecodes.

Superoperators are similar to our macro opcodes. One advantage is that their corresponding synthesized bytecodes can benefit from some of the inter-opcode optimizations that our simple concatenation of implementations fails to exploit. However, superoperators require bytecodes corresponding precisely with the nodes used to build parse trees — which might not always be the best choice of bytecode set. It would also be tricky to use superoperators in an incremental system such as Smalltalk, where new executable code is generated at runtime. Nevertheless, an investigation of merging some of the techniques of superoperators and dynamically-generated macro opcodes might be very worthwhile.

6 Conclusions

This work was inspired by the need to create an interpreter with a very fine-grain RISC-like opcode set, that is both general (not tied to any particular high-level language) and amenable to traditional compiler optimizations. The cost of opcode dispatch is more significant in such a context, compared to more abstract interpreters whose bytecodes are carefully matched to the language semantics.

The expected benefits of our technique are related to the average semantic content of a bytecode. We would expect languages such as Tcl and Perl, which have relatively high-level opcodes, to benefit less from macroization. Interpreters with a more RISC-like opcode set will benefit more — since the cost of dispatch is more significant when compared to the cost of executing the body of each bytecode. The Objective Caml bytecode set is positioned between these two extremes, containing both simple and complex opcodes.[8]

Vcode has better performance than our technique because its instruction set matches very closely the underlying architecture. It can exert very fine control over the code that is generated, such as performing some degree of reordering for better instruction scheduling. We believe that similar results can be achieved with our RISC-like inlining threaded code interpreter, but in a more portable manner.

The performance of macro opcodes is limited by the inability of the compiler to perform the inter-opcode optimizations that are possible when a static analysis is performed and new macro opcodes implemented manually in the interpreter. We believe that these limitations are less important when using a very fine-grain opcode set, corresponding more closely to a traditional RISC architecture. Most opcodes will be implemented as a single machine instruction, and new opportunities for inter-opcode optimization will be available to the translator's code generator.

Our technique is portable, simple to implement, and orthogonal to the implementation of the virtual machine's opcodes. In reducing the overhead of opcode dispatch, it helps to bring the performance of fine-grained bytecodes to the same level as that of more abstract, language-dependent opcode sets.

[7] Since translation is performed only once for each opcode, the "break-even" point is passed in any program that executes more than six times the number of opcodes that it contains.

[8] Significant overheads are associated with the technique used to check for stack overflow and pending signals in Objective Caml, but a discussion of these is beyond the scope of this paper.

	speed (seconds)						space (bytes)		
	Pentium		**Sparc**		**PowerPC**		**Sparc**		
benchmark	*original*	*inlined*	*original*	*inlined*	*original*	*inlined*	*original*	*inlined*	*cache*
boyer	2.0	1.81 (111%)	2.3	1.50 (154%)	1.4	1.19 (113%)	13800	8324	42012
fib	2.0	1.44 (140%)	4.0	2.47 (163%)	1.6	1.12 (139%)	5288	3320	20160
genlex	1.0	0.93 (110%)	1.1	0.84 (127%)	0.7	0.59 (118%)	45696	26856	156892
kb	10.3	8.15 (126%)	16.9	7.71 (219%)	6.3	5.36 (118%)	20968	13048	75868
qsort	5.8	3.95 (146%)	9.5	5.39 (175%)	4.1	2.98 (137%)	6676	3932	26416
qsort*	4.8	3.04 (158%)	8.0	4.26 (188%)	3.3	2.27 (147%)	6532	3884	25280
sieve	3.0	2.79 (107%)	2.5	2.22 (110%)	1.9	1.86 (100%)	5200	3312	20124
soli	3.1	2.18 (144%)	5.1	2.98 (170%)	2.1	1.50 (142%)	6644	3952	25516
soli*	2.4	1.38 (172%)	4.0	2.00 (202%)	1.6	0.93 (168%)	6544	3908	24548
takc	2.8	1.91 (144%)	5.0	3.26 (152%)	2.1	1.47 (142%)	4784	3012	18652
taku	4.9	3.20 (152%)	7.0	4.14 (170%)	3.2	2.33 (139%)	4812	3036	18296

Table 3: Raw results for the Objective-Caml benchmarks.

Acknowledgements

The authors would like to thank Xavier Leroy, John Maloney, Eliot Miranda, Dave Ungar, Mario Wolczko and the anonymous referees, for their helpful comments on a draft of this paper.

References

[Arn96] K. Arnold and J. Gosling, *The Java Programming Language*, Addison Wesley, 1996. ISBN 0-201-63455-4

[Aus96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers and Brian Bershad, *Fast, Effective Dynamic Compilation*, Proc. PLDI '96. Published as SIGPLAN Notices 31(5):149–159.

[Bel73] James R. Bell, *Threaded Code*, Communications of the ACM, 16(6):370–372, 1973.

[Deu84] L. Peter Deutsch and Alan M. Schiffman, *Efficient Implementation of the Smalltalk-80 System*, Proc. POPL '84, pages 297–302.

[Eng96] Dawson R. Engler, VCODE*: A Retargetable, Extensible, Very Fast Dynamic Code Generation System*, Proc. PLDI '96. Published as SIGPLAN Notices 31(5):160–170. `http://www.pdos.lcs.mit.edu/~engler/vcode.html`

[Ert93] M. Anton Ertl, *A Portable Forth Engine*, Proc. euroFORTH '93, pages 253–257. `http://www.complang.tuwien.ac.at/forth/threaded-code.html`

[Gol83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983. ISBN 0-201-11371-6

[Ing97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay, *Back to the Future: the Story of Squeak, a Usable Smalltalk Written in Itself*, Proc. OOPSLA '97. Published as SIGPLAN Notices 32(10):318–326.

[Ler97] Xavier Leroy, *The Objective Caml system release 1.05*, INRIA, 1997.

[Ler98] Xavier Leroy, personal communication.

[Lin97] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997. ISBN 0-201-63452-X

[Mir87] Eliot Miranda, *BrouHaHa — A Portable Smalltalk Interpreter*, Proc. OOPSLA '87. Published as SIGPLAN Notices 22(12):354–365.

[Mir91] Eliot Miranda, *Portable Fast Direct Threaded Code*, posted to `comp.compilers`. `http://cuiwww.unige.ch/OSG/people/jvitek/Compilers/Year91/msg00215.html`

[Mir97] Eliot Miranda, personal communication.

[Moo70] Charles H. Moore and Geoffrey C. Leach, *FORTH — A Language for Interactive Computing*, Technical Report, Mohasco Industries, Inc., 1970. `http://www.dnai.com/~jfox/F70POST.ZIP`

[Noe98] François Noël, Luke Hornof, Charles Consel and Julia L. Lawall, *Automatic, Template-Based Run-Time Specialization: Implementation and Experimental Study*, Proc. ICCL '98. `http://www.irisa.fr/compose/papers/rt_bench.ps.gz`

[Pro95] Todd A. Proebsting, *Optimizing an ANSI C Interpreter with Superoperators*, Proc. POPL '95, pages 322–332.

[Tho95] Tom Thompson, *Building the Better Virtual CPU*, Byte Magazine, August 1995.

Appendix

Table 3 shows the raw results for the Objective Caml benchmarks. The execution speed (in seconds) is shown for all three architectures, for both the original interpreter and the inlined interpreter. The inlined interpreter speed is shown both as an absolute figure and as a percentage relative to the original interpreter's speed. The final three columns show the sizes of the original threaded code, the threaded code after inlining, and the final size of the macro cache for the Sparc only. All are measured in bytes.

The sources for the RISC-like interpreter and the modified Objective-Caml interpreter used to generate the benchmark data are available from `http://www-sor.inria.fr/~piumarta/pldi98`.