Retargeting JIT compilers by using C-compiler generated executable code

M. Anton Ertl
TU Wien
anton@mips.complang.tuwien.ac.at

David Gregg
Trinity College, Dublin
David.Gregg@cs.tcd.ie

Abstract

JIT compilers produce fast code, whereas interpreters are easy to port between architectures. We propose to combine the advantages of these language implementation techniques as follows: we generate native code by concatenating and patching machine code fragments taken from interpreter-derived code (generated by a C compiler); we completely eliminate the interpreter dispatch overhead and accesses to the interpreted code by patching jump target addresses and other constants into the fragments. In this paper we present the basic idea, discuss some issues in more detail, and present results from a proof-of-concept implementation, providing speedups of up to 1.87 over the fastest previous interpreter-based technique, and performance comparable to simple native-code compilers. The effort required for retargeting our implementation from the 386 to the PPC architecture was less than a person-day.

1 Introduction

Different programming language implementation approaches provide different tradeoffs with respect to the following criteria:

- Portability (Retargetability)
- Execution Speed
- Compilation Speed

Existing language implementation techniques can satisfy only two of these criteria, and have disadvantages in the third: Native-code compilers are hard to retarget; interpreters execute slowly; and compilation-through-C results in slow compilation.

We propose to improve this situation with the following approach: we start with an interpreter written in C, modify it a little, then compile it to executable code, extract fragments from this code, and use them in a native-code compiler that generates code by concatenating and patching these fragments.

In this way the language implementor can start out with (and, if necessary, fall back to) an implementation that satisfies the requirements of portability and high compilation speed, and still achieve execution speed comparable to a simple native-code compiler (as used in many JITs).

You can view this approach in two ways:

- You can see it as a way to speed up an interpreter by turning it into a JIT compiler. This is the way the technique is presented in most of this paper.
- Or you can see it as a way of reducing the retargeting effort needed in a native-code compiler, with the additional benefit of having an interpreter to fall back to (e.g., if the language implementor does not have a machine with a specific architecture available for testing the port). It is important to remember this viewpoint when comparing with other work for generating compilers out of interpreters (Section 1.3).

We think that the main application area for this technique is for implementing new languages, and for speeding up existing languages that have been implemented as interpreters in the past (e.g., Perl, Python, Ruby, PHP).

In this paper we first present an overview of our approach (Section 2), then discuss some of the issues in more detail (Section 3), and present results for our proof-of-concept implementation (Section 4). Finally, in Section 5 we compare our work with related work in-depth.

The main contribution of this paper is in using this approach for a native-code compiler, including fall-back strategies for cases where our methods are not applicable (e.g., dealing with non-relocatable code, see Section 3.3), and the empirical results from a proof-of-concept implementation (Section 4). We also present a new and more general method for finding places to patch in the fragments (Section 3.5).

1.1 Why not write a compiler directly?

The native code resulting from the approach proposed in this paper is still quite a bit slower than the code from an optimizing native-code compiler such as gcc, and is comparable in speed to code from fast and simple native-code compilers like bigForth and iForth (see Section 4), so why would our approach be interesting? For the following reasons:

Portability With our approach, you start out with and can fall back to an interpreter that can run on a new architecture without any porting effort (e.g., Gforth-0.6.2 runs out of the box on IA64, AMD64, and ARM, three architectures that it has not been tested on before release), whereas if you write just a native-code compiler, it will not run on new architectures without retargeting (and that typically requires someone who is familiar with the compiler internals).

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Parallel Architecture and Compilation Techniques (PACT' 04), pages 41-50

If you implement an interpreter for portability and a native-code compiler for better speed on some platforms (the approach taken by, e.g., Ocaml), this will require more implemenation and maintenance effort than our approach, and there is also a higher potential for inconsistencies between the two implementations.

Implementation effort We implemented our approach for the 386 architecture on top of the Gforth interpreter in two person-weeks (resulting in about 200 lines of code specific to the native-code compiler), and retargeted it to the PPC architecture in less than a person-day (resulting in 91 new or changed lines compared to the 386 port). See Section 3.8 for details.

The native-code compilers we compare with (bigforth, iforth, and gcc) have certainly required more implementation and retargeting effort. In particular, bigForth has only has two targets so far (68k and 386) and iForth only one (386), in both cases without fallback to an interpreter; we believe that this lack of portability is caused by the higher effort that retargeting would require. Retargeting gcc has been reported to require several person-months of effort.

As a further data point, let us look at Ocaml, a system having both an interpreter and a native-code compiler with relatively many targets: The Ocaml-3.06 native-code compiler and its run-time system have 8700 lines of target-independent code in addition to the 6000 lines of code common to the bytecode interpreter and the native-code compiler. In addition, they have 1200–3100 lines of code for each target.

Compilation speed During compilation, our approach just copies existing code fragments and patches constants into the appropriate places. It is hard to beat such an approach on compilation speed, if you want the resulting code to be faster. By contrast, gcc produces significantly better code, but does not compile fast enough to serve as a load-and-go or JIT compiler.

The Ocaml-3.06 byte-code compiler takes 5.5s of CPU time to compile a 4300-line file on an 800MHz Alpha 21264B machine; the native-code compiler takes 40.2s (factor 7). For comparison, our prototype Gforth compiler compiles 3500 lines in 86ms on a 1200MHz Athlon; this is slightly faster than the Gforth threaded-code compiler.

1.2 Why not write a source-to-source translator?

A popular approach for language implementation is to compile a source language into "source" code for another language and then compiling that into machine code with an optimizing compiler; C is most popular as the intermediate language, because C imposes fewer restrictions and has better compilers than most other languages.

The disadvantages of this approach compared to our approach are the *lower compilation speed* (unacceptable for an interactive or JIT compiler[1]), and that some language features (e.g., run-time code generation and guaranteed tail-call elimination) cannot be implemented directly in C, whereas they can be implemented in an interpreter (even one written in C), and in our interpreter-based compiler.

[1] One reader praised the compilation speed of MSVC compared to gcc. However, even if MSVC was fast enough, that would be of little use to someone porting a system to, say, Linux-PPC. Moreover, if MSVC is fast enough, why is the .NET JIT compiler not implemented as a translator-through-C?

1.3 Why not use partial evaluation?

Producing compilers from interpreters has been proposed before, especially in the context of partial evaluation and program specialization: Specialize an interpreter for a source program, and you get a compiled version of that program.

However, these proposals solve a different problem than our approach; in particular, they do not attack the retargeting issue, they merely push it down into the specializer, or even further:

- Most partial evaluators output source code for a programming language, so the resulting compiler would actually be a source-to-source translator, with the corresponding compilation speed disadvantage.
- Some systems, like Tempo [NHCL98] and DyC [GMP$^+$00], are able to produce native code directly. But these systems have to be retargeted themselves: E.g., DyC is based on a conventional compiler (Multiflow) and retargeting DyC is at least as hard as any compiler. Tempo pioneered some of the techniques we use for retargeting, but we introduce additional techniques that make retargeting easier and applicable to more architectures (see Section 3.5).

On the practical side, specializers are complex systems that are currently mostly available as research prototypes, without the long-term maintenance guarantee that language implementors require of their tools.

In contrast, our approach is simple enough that it can be implemented and maintained with relatively small effort (compared to the rest of the language implementation) by the language implementors themselves, without creating a dependence on the developer of a partial evaluator or an interpreter retargeting library.

2 Basic Idea

This section demonstrates our approach on a running example. We start with a plain interpreter, then add several previously proposed optimizations, and finally (Section 2.6) perform a new step that results in the complete elimination of interpreted code.

2.1 Running Example

Consider the Java expression `x?y+1000:z`. The corresponding JVM code produced by the `javac` compiler is:

```
      iload_1     ; x
      ifeq else   ; ?
      iload_2     ; y
      sipush 1000 ; 1000
      iadd        ; +
      goto cont   ; :
else: iload_3     ; z
cont:
```

Our running example will be the following piece of that JVM code:

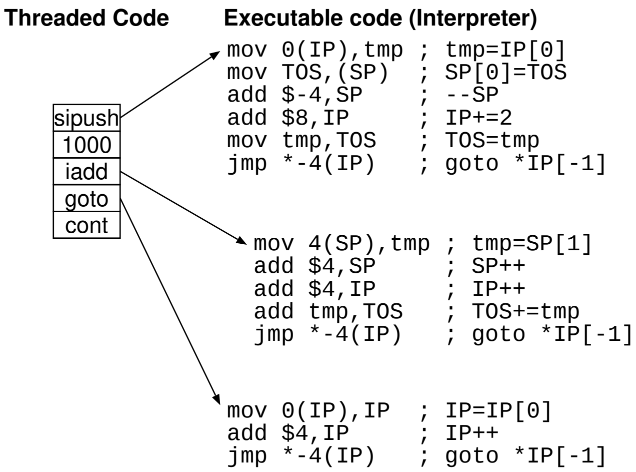

Figure 1. Threaded code and the corresponding parts of the interpreter

```
sipush 1000 ; 1000
iadd        ; +
goto cont   ; :
```

2.2 Threaded Code

The fastest interpreters use threaded code [Bel73] to minimize dispatch overhead. Threaded code represents each VM instruction as address of the routine that implements the instruction; the code for dispatching the next instruction consists of fetching the VM instruction, jumping to the fetched address, and incrementing the instruction pointer. This technique cannot be implemented in ANSI C, but it can be implemented in GNU C using the labels-as-values extension.[2]

Figure 1 shows threaded code for our example, and the routines of the interpreter that are called to execute the VM instructions (as 386 assembly code, with VM register names instead of real register names; note that `%esp` is used by GCC for its own purposes, so our `SP` is in some other register and we cannot use the `push` and `pop` instructions).

In Fig. 1 the machine code enters each routine with the instruction pointer (IP) pointing to the memory location just after the VM instruction (e.g., in our example IP points to 1000 when the `sipush` code is entered); each routine updates IP to point just after the next VM instruction, and then jumps to the address pointed to by the VM instruction. The stack grows down. The top-of-stack element (TOS) is kept in a register. The stack pointer (SP) points to where the top-of-stack element would be if it were in memory.

2.3 Static Superinstructions

One way to improve performance is to provide VM superinstructions that have the effect of a sequence of simple VM instructions, and to replace sequences of simple VM instructions with appropriate superinstructions [Pro95, HATvdW99, EGKP02]. Figure 2 shows the result of using a superinstruction `sipush_iadd` instead of the sequence of `sipush` followed by `iadd`.

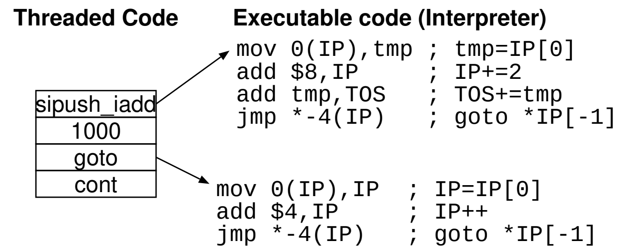

Figure 2. Threaded code with static superinstructions

In this case the code for the superinstruction is shorter than the code for each of the simple VM instructions, because the superinstruction does not change the stack depth (no SP update necessary) and only accesses TOS (no stack memory access necessary).

2.4 Dynamic Superinstructions and Replication

The interpreter cannot provide a static superinstruction for every possible sequence of simple instructions; in fact, practical considerations (in particular, the space required for compiling the interpreter) limit the number of static superinstructions to several hundred. Another way to create superinstructions is by simply concatenating the executable code of the VM instructions, and leaving the dispatch code between the VM instructions away (see Fig. 3) [PR98].

These dynamic superinstructions are not as well optimized as static superinstructions, but they reduce the dispatches more effectively, and dispatches are expensive [EG03].

Dynamic superinstructions can be reused if the same sequence occurs several times [PR98], but having a separate instance of the superinstruction for each occurence of a sequence in the threaded code (replication) is both simpler to implement and improves the branch prediction accuracy of the remaining dispatches [EG03].

2.5 Do we need IP?

With dynamic superinstructions and replication, the dynamically generated code is already quite close to what a simple native-code compiler would produce. In particular, this code needs VM dispatches only when a VM branch is taken. However, a lot of the remaining code deals with IP, i.e., the pointer to threaded code.

The threaded code and IP are still needed for the following purposes:

- To find the native code to execute after a dispatch (i.e., after a taken VM branch).
- To access immediate arguments of VM instructions (in our example, 1000 and cont).

How can we eliminate these uses, and thus the threaded code and IP?

[2] With a little bit of conditional compilation, it is easy to write an interpreter such that it uses threaded code with gcc, and uses some ANSI C-compliant technique with other compilers.

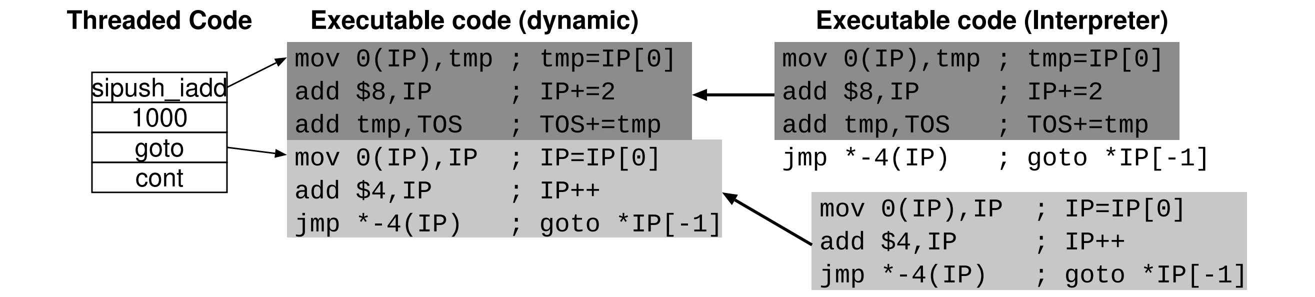

Figure 3. Dynamic superinstructions

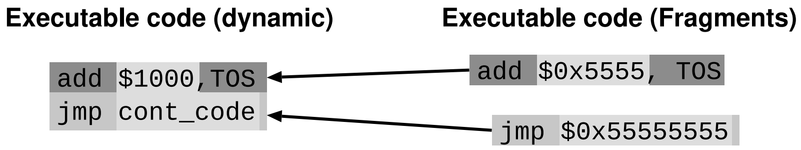

Figure 4. Generating code by concatenating and patching fragments

2.6 Fragments and Patching

The approach we propose here is to have code fragments that contain immediate arguments instead of references through IP. When copying the code fragment during the dynamic code generation, we patch the desired immediate argument into the fragment in place of the original immediate argument. And instead of using threaded code dispatch for VM branches, we just use direct branches; the target addresses are patched just like other immediate arguments. Figure 4 shows how this approach works in our example.

How do we get the fragments and the patch information? A fragment is basically the executable code for a VM instruction; if the VM instruction has no inline argument, it is basically the same code as in the dynamic superinstruction case. The patch information is found by comparing two fragments for VM instructions that differ only in constants used in the VM instruction. This is discussed in more detail in Section 3.5.

2.7 Benefits

The benefit of this approach over the fastest previous interpreter-based techniques is faster code, for three reasons:

- All accesses to the interpreted code are eliminated, in particular accesses to immediate arguments and accesses to threaded code pointers for control flow.
- Our approach uses direct jumps instead of performing control flow through threaded code dispatch (load and indirect jump). For conditional VM branches, this results in a direct conditional branch instead of a conditional branch followed by a dispatch.
- The VM instruction pointer is eliminated, freeing a register, and eliminating all the instructions for updating the VM instruction pointer.

```
start_sipush_iadd_n:
  TOS += 0x5555;
end_sipush_iadd_n:
  goto *next;
...
memcpy(dest, &&start_sipush_n,
   (&&end_sipush_n)-(&&start_sipush_n));
```

Figure 5. The GNU C code for a fragment, and code for copying the fragment to `dest`

The benefits of this approach over writing a JIT compiler in the usual way are that it needs very little platform-specific code; and if even that is not available, we can fall back to an interpretive implementation, and thus achieve full portability.

Another way of looking at this is: if you implement a language through an (efficient) interpreter, you do not need to redo a lot of work if you want to improve performance with a simple compiler; you can just extend the work you did on the interpreter.

3 Details

This section discusses a number of specific issues that we ignored in Section 2. Some of the techniques presented in this section are not original to this paper and we present them for completeness. Read Section 5 for a detailed account.

3.1 Finding Fragments

How do we find the start and the end of each fragment? We put labels before and after each fragment, and we use GNU C's labels-as-values extension to convert these labels into the start and end addresses of the machine code for the fragment[3] (ANSI C only

[3] Recent GCC versions need the `-fno-reorder-blocks` flag in order to guarantee that the order of labels and code in the source is preserved in the native code.

allows using labels in `goto` statements); eventually these labels are used in a `memcpy()` call. Figure 5 shows the code for a fragment, and how code for copying the fragment might look (in reality the copying code will be more general, though).

3.2 Context

How do we ensure that the code fragments that we use fit together? In particular, how do we ensure that the register allocation is the same between fragments?

We put all the fragments in the same C function, and we put an indirect jump after each fragment (see Fig. 5). This indirect jump can jump to any fragment; this forces the compiler to make sure that fragments can follow each other in any order.

For fragments containing patchable direct jumps, we initially let the direct jump branch to an indirect jump, in order to ensure the correct register allocation (the actual target is set later, during patching).

3.3 Non-Relocatable Code

Not all executable code can be copied easily to another location. E.g., absolute references within the fragment (e.g., conditional branches on the MIPS architecture) or PC-relative references to code outside the fragment (e.g., calls on 386 or SPARC) require more complex relocation capabilities than just copying the code. How do we deal with such fragments?

We can try to use the same techniques for relocating this code that we use for patching fragments with embedded immediate arguments (see Section 3.5).

However, if all else fails, we have to classify the fragment as non-relocatable, and we have to find a way to still use it.

Our approach in this case is to call the fragment in its original location (in the executable code of the C function containing all the fragments), using the indirect jump at the end of the fragment to return to our dynamically generated code. To achieve this, we load the variable `next` with the address of the code after the non-relocatable fragment, and then jump to the non-relocatable fragment[4] (see Fig. 6). The indirect jump at the end of the fragment will then jump back to the dynamic code.

3.4 Relocatability information

How do we find out if a fragment is non-relocatable? We compare two fragments produced from the same C source code.

We have tried several variants of implementing this idea, but the one that we have finally settled on is to have two instances of the function containing the fragments. The benefits over having all fragment instances in the same function are: 1) no spurious differences from different stack offsets of equivalent fragment-local variables; 2) we can use more unique fragments (the number of fragments per function is limited by gcc's memory usage).

One of these functions contains additional padding between the fragments (created with `asm(".skip 16");`), to make relative inter-fragment jumps different.

3.5 Patch information

How do we know where to patch the fragments? For each fragment, we produce two instances that differ only in the embedded constants; then we compare their machine code to produce the entry for the fragment in the patch table (see Fig. 7).

When comparing the fragments, we have to check for all the ways in which the fragment-producing compiler encodes the constants. For each architecture, there are only a few such ways. E.g., on the 386 architecture constants are encoded as sign-extended 1-byte values or as 4-byte values, either PC-relative (for branch and jump targets) or absolute (for other immediate arguments); on the PowerPC architecture constants are either encoded as 16-bit or 26-bit PC-relative addresses, or as absolute values that are split into two 16-bit chunks or (if representable) are stored in one sign-extended 16-bit field.

By wisely choosing the constants that we put into the fragments, we can ensure that we recognize all the places that contain the constants, that the encodings we want to have are used, and that we recognize the encodings and offsets correctly, and which constants in the fragments correspond to which logical argument.

In particular, in our prototype we use word-size arguments (where the most significant bits are either 10 or 01) for non-branches to ensure that we get full-length encodings; the two constants for the same argument of the two instances of the same fragments are bitwise complements of each other to ensure that they are recognized as being different. And the two 16-bit halves are different to ensure that we recognize which half is which.

In addition to having fragments for word-size constants, we could provide alternative fragments for smaller constants (e.g., within the signed 8-bit range for the 386, and the signed 16-bit range for PPC), to allow to use better code; our prototype implementation does not do this yet.

For patchable branches, our current prototype uses just some targets within the same function (different targets for different instances), with the targets being not-too-close to avoid getting short operand encodings.

On some platforms, large constants (e.g., 64-bit constants on the Alpha architecture) are not embedded in the code, but referenced through a constant table. In these cases, the varying bits in the machine code have no obvious relation to the constant, and we will have to find another way to deal with such constants. One way is to use a constant table ourselves, which is accessed through a constant table pointer and offsets; the offsets are small enough to be created without the constant table. This is not yet implemented in our prototype implementation.

If we are not able to determine the encoding and position of the constants (e.g., if the compiler used an encoding we did not expect), or if a fragment containing an inline constant turns out to be

[4] On many architectures we will have to use an indirect jump to perform this (long) jump; GCC-3.2 and later rearrange indirect branches so we cannot copy them (even with `-fno-reorder-blocks`, see http://gcc.gnu.org/bugzilla/show_bug.cgi?id=15242). However, the gcc maintainers are working this problem, and hopefully it will be fixed in GCC-3.5.

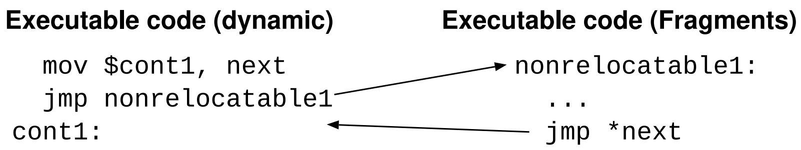

Figure 6. Calling a non-relocatable fragment

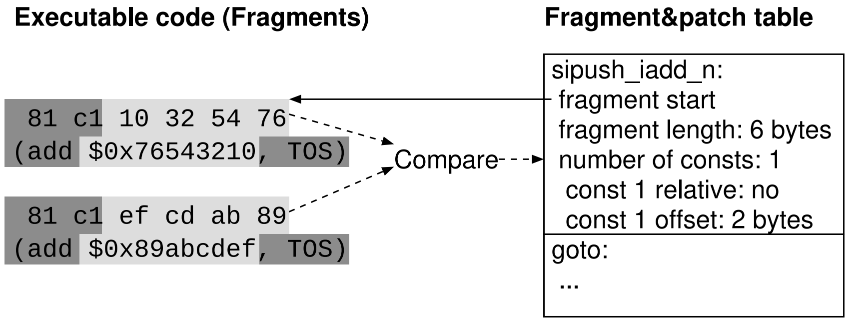

Figure 7. Generating patch information by comparing fragments

non-relocatable, we have to give up and fall back on the interpreter. However, this has not happened in our prototype implementation.

3.6 Determining the encodings

We hardcoded the information about the various constant encodings used in the architecture into our prototype. In a production version, we will abstract this information out into architecture-specific include files; we expect each of these files to contain a few dozen lines of code.

However, the encodings can also be determined automatically by comparing fragments: For this approach, we need more instances of the same fragments with different constants. E.g., for each bit in the constant, we might have an additional fragment whose constant differs from the constant of the previous fragment only in that bit, allowing to determine which bit in the constant corresponds to which bit in the fragment.

This approach would be able to deal with arbitrary splitting of the constant into parts (e.g., 16-16 for PPC, 19-13 for SPARC), as well as nonsequential distribution of the bits of the constant (e.g., in HPPA branches). Its limitations are that it can only recognize encoding schemes we expect (such as absolute and PC-relative), and the scheme has to be simple enough that changing one bit in the constant changes one bit in the fragment.

One problem with this idea is that it is probably easier to write a dozen or so encoding descriptions for the architectures we are interested in than to implement this idea. While complete machine independence is a worthwhile goal, the encoding description is not the only thing missing for this goal; we also have to provide code for ensuring instruction cache consistency (typically a few lines of code), and we know of no automatic way to generate that code.

```
;a call to foo
  add    $-4,SP          ;SP--
  movl   $retaddr,(SP) ;SP[0]=&&retaddr
  jmp    foo             ;goto foo
retaddr:
;and a return
  mov    (SP),%eax       ;tmp=SP[0]
  add    $4,SP           ;SP++
  jmp    *%eax           ;goto *tmp
```

Figure 8. A simple call (no parameter or frame handling) to foo and a simple return

3.7 Calls and Returns

Of course, we can only use features in the generated code for which we can generate fragments from C source code. A particularly nasty problem in this respect are VM-level calls and returns.

One might think that we can just use fragments generated from C code for a C call or a `return`, but this does not work, because this does not call or return to code in the context of the function we are working in; instead, the calling sequence typically changes the C-stack pointer (invalidating all references through that pointer), and the return sequence also tries to restore registers.

There is no other way to get the C compiler to produce call and return instructions in a portable way, so we have to emulate calls and returns using ordinary jumps: the calling code provides the return address as literal and saves it somewhere, then jumps to the target; the return code just performs an indirect jump (`goto *...` in GNU C) to the return address (see Fig. 8).

Of course, this approach does not provide the best performance; the calling code is larger than otherwise, and the return code does not utilize the branch prediction usually provided by a processor's return stack.

If we want performance to be better, we can provide machine-specific call and return fragments, and fall back to the generic approach for machines without special support.

3.8 Implementation Effort

Implementing the techniques described in this paper in Gforth required detailed knowledge of the Gforth VM, and the threaded-code generator, and of the techniques themselves. The changes needed were mostly local to the part of Gforth that is also involved when implementing dynamic superinstructions.

In addition, we had to change a few other things in the code generator where (originally) data was mixed with the threaded code and accessed through the return address of Forth functions. We modified such code so that it explicitly takes a pointer to the data as a literal; this works both for the native-code and the threaded-code compiler. While this particular issue is probably relatively Gforth-specific, we expect that other language implementations also have issues where threaded-code addresses may be needed instead of native-code addresses or vice versa (e.g., exception tables in the JVM).

Overall, implementing this approach required about two person-weeks, and resulted 200 lines of code specific to the native-code compiler, plus small changes in other code generation parts to eliminate threaded/native code-address mixups.

Retargeting the prototype to PowerPC required a knowledge of the part of the Gforth system that deals with patching (i.e., very localized knowledge), and a knowledge of the way that constants and branch targets are represented in the 386 and PPC architecture. This required less than a person-day of work and resulted in 91 new or changed lines compared to the 386 port.

4 Experimental Results

We implemented the approach presented above in Gforth, a product-quality Forth system[5], for the 386 and the PowerPC architecture.

We ran a number of benchmarks on this system, on a number of Gforth variants and other Forth systems, and, for some benchmarks, GCC:

gforth-plain Gforth without static or dynamic superinstructions (`gforth-fast --no-dynamic --ss-number=0`). This is an efficient classic threaded-code interpreter.

gforth-super Gforth with static and dynamic superinstructions and replication (`gforth-fast`). 13 static superinstructions are used.

gforth-native Our prototype implementation. It uses the same 13 static superinstructions as gforth-super.

bigforth (version 2.0.11). A fast and simple native-code compiler (uses peephole optimization).

iforth (version 1.12.1125). Another simple native-code compiler.

[5] You may be wondering why we have not used this approach with a JVM implementation. The reason is that we currently do not have a useful JVM interpreter to which we could apply this approach (the Cacao interpreter that we used in earlier work was very limited in the set of benchmarks it could run).

Program	Version	Lines	Description
cross	0.6.9	3793	Forth cross-compiler
tscp	0.4	1625	chess
brainless	0.0.2	3519	chess
vmgen	0.6.9	2641	interpreter generator
bench-gc	1.1	1150	garbage collector
CD16sim	1.1	937	CPU emulator
pentomino		516	puzzle solver
sieve		23	prime counting
bubble		74	bubble sort
matrix		55	integer matrix multiply
fib		10	double-recursive function
mm-rtcg		109	run-time code generation
compile		3519	brainless compile-only

Figure 9. Benchmark programs used

gcc (version 2.95.1 on the Athlon, 3.3.2 on the PPC). This uses hand-written C code for the benchmarks, compiled with `gcc -O3`. The compile time is not included in the timings for the gcc results (in contrast to the Forth systems).

Figure 9 shows the benchmarks we used for our experiments. We used the following platforms: a 1200MHz Athlon (Thunderbird, VIA KT133 chipset, 384MB PC100 SDRAM, Linux-2.4.18, glibc-2.1.3) and a 450MHz PPC7400 (PowerMac3.1 system, 256MB RAM, Linux-2.4.22, glibc-2.3.2).

Figure 10 and 11[6] show how the various systems perform compared to gforth-native. Not all benchmarks run on all systems (in particular, we only have C code for sieve, bubble, matrix, and fib), so we show different sets of bars for different benchmarks.

On the Athlon the speedup of gforth-native over gforth-plain is between 1.4 (mm-rtcg) and 5.2 (matrix), median 2.7. On the PPC7400 the speedup is between 1.43 (cross) and 2.42 (fib), median 1.94.

The speedup on the Athlon over gforth-super is between 1.06 (matrix) and 1.49 (tscp), median 1.32; on the PPC7400 the speedup is between 1.29 (cross) and 1.87 (fib), median 1.52. These speedups may appear relatively small compared to some of the other speed differences shown here, but if we put it in relation to the relatively small effort required to achieve it, we find that this is a very worthwhile improvement.

The differences between the machines can be explained in the following way: Branch mispredictions are more expensive on the Athlon, therefore gforth-plain loses more compared to gforth-super and gforth-native on this machine. The remaining indirect branches in gforth-super are more expensive on the PPC7400 (which does not predict indirect branches) than on the Athlon (which predicts nearly all of them correctly), so replacing these indirect branches by direct branches helps the PPC7400 more.

The speedup of bigForth over gforth-native is rather inconsistent, between 0.12 (CD16sim) and 2.09 (fib), median 1.19. The extreme slowdown of bigForth on CD16sim is caused by placing code close to frequently written data, resulting in cache consistency overhead.

[6] We did not use pentomino on the PPC7400, because this benchmark requires conditional branches over more then 32KB, and we have not implemented this on the PPC port yet.

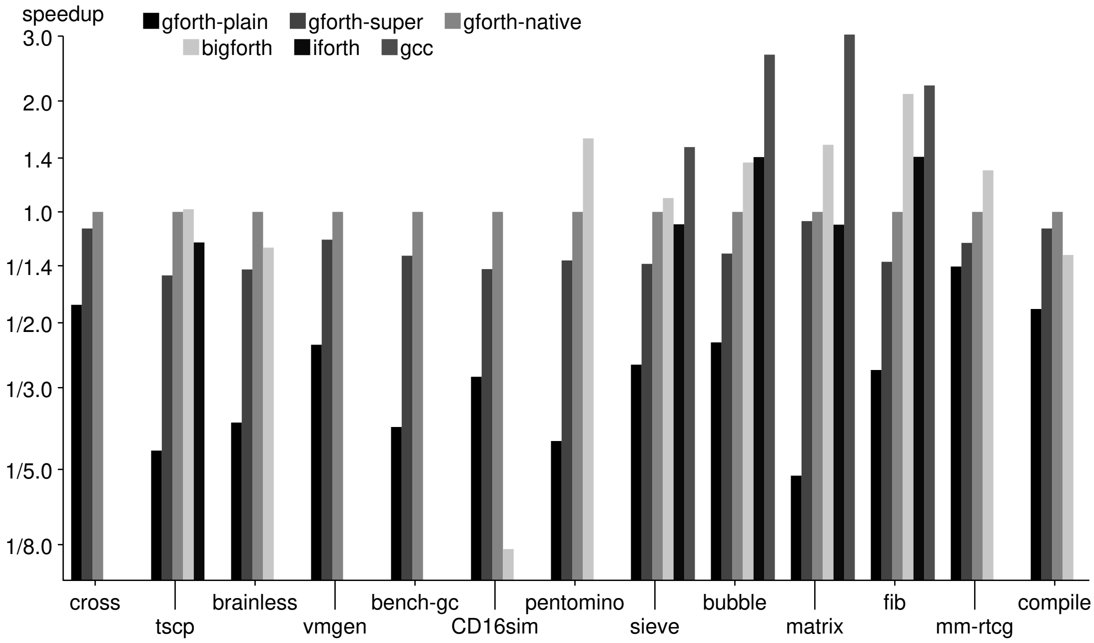

Figure 10. Speedup of various systems relative to gforth-native on a 1200MHz Athlon

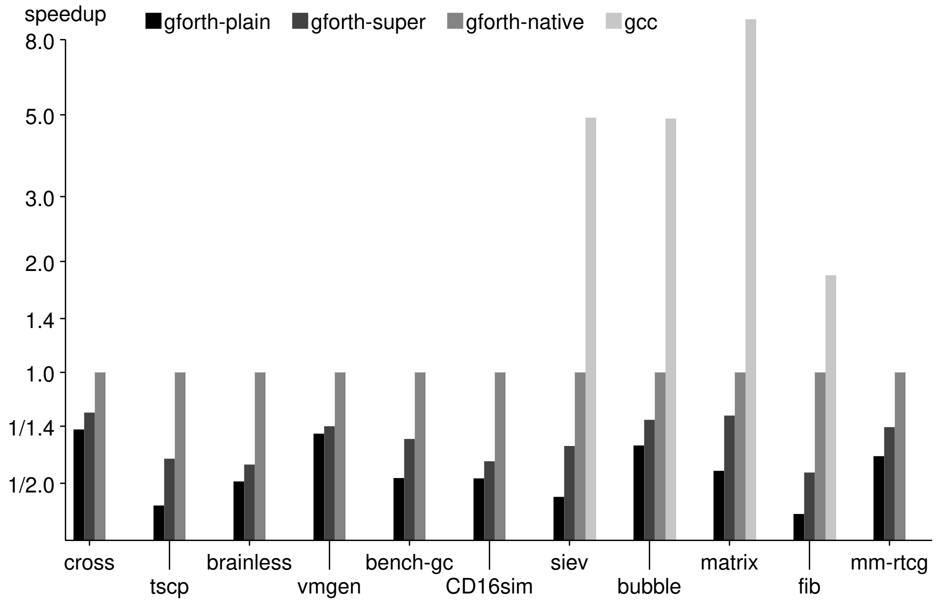

Figure 11. Speedup of various systems relative to gforth-native on a 450MHz PPC7400

The speedup of iForth over gforth-native is between 0.83 (tscp) and 1.41 (fib), median 0.93. We conclude that gforth-native is competetive in performance with simple hand-written native-code compilers.

The speedup of gcc-2.95.1 on the Athlon over gforth-native is between 1.50 (sieve) and 3.03 (matrix), median 2.44. On the PPC7400 the speedup of gcc-3.3.2 over gforth-native is between 1.84 (fib) and 9.09 (matrix), median 4.90. This is not a fair comparison (gcc is not an interactive system, it compiles much slower, and the compile time is not included in the gcc timings), but it provides a kind of upper bound for the speedup that we can still achieve. There is still quite a bit of work left to do, especially on the PPC7400. We explain the slowdown of gforth-native over gcc at least partially with the stack-in-memory accesses; we have seen a good speedup (especially on the PPC 7400) from eliminating a part of this overhead with static stack caching [EG04].

Concerning compilation speed, the *compile* benchmark shows that gforth-native compiles faster than gforth-plain or gforth-super; the reason is that a good part of the compile time is spent in the front end, which is written in Forth and gets a speedup from the faster execution in gforth-native. The small additional compile time in the back-end of the compiler more than pays for itself. The gforth-native compiler is also faster than bigForths compiler.

5 Related Work

Rossi and Sivalingam [RS96] proposed dynamically generating native code by copying fragments from an interpreter.

Piumarta and Riccardi [PR98] expanded on that and implemented this idea in a full-scale interpreter. The main difference between their work and ours is that their generated code still uses the interpreted code for immediate arguments and for control flow. Therefore they do not patch literals and branch targets into the native code.

They deal with non-relocatable code by falling back to ordinary threaded code, which is not possible in our IP-less approach (see Section 3.3 for our solution). They rely on programmer-supplied tables for recognizing relocatability, in contrast to our automatic way (Section 3.4).

Other recent work [EG03] looked at various ways to reduce the dispatch branch cost in interpreters. The present work investigates eliminating interpreter dispatch altogether and everything related to it, resulting in even more performance.

Tempo [NHCL98] is a run-time specializer that works by producing C code for fragments and concatenating and patching them at run-time. The difference from the present work is that Tempo specializes specific functions for specific inputs, whereas we implement a general compiler; in particular, Tempo does not have to deal with arbitrary sequences of fragments (see Section 3.2).

Another difference is that we have a fall-back strategy for dealing with non-relocatable code (Section 3.3), while Tempo has none, making it harder to retarget Tempo (relocation has to work in all cases).

Finally, Tempo uses the meta-information from the object file to produce the patch information, whereas we compare two native-code fragments with different embedded constants (see Section 3.5). One disadvantage of Tempo's approach is that it is hard to port to many 64-bit systems, where large constants like addresses are not stored in the code, but in separate constant tables. Another disadvantage is that it always uses the full-size encoding of constants, and does not allow for more efficient encodings.

Vitale and Abdelrahman [VA04] explore turning an interpreter into a JIT compiler by concatenating and patching (specializing) machine code derived from an interpreter, just like we do. The main difference from our work is that they use somewhat different techniques for deriving the fragments and patch information than we do, and their methods require postprocessing the assembly code produced by GCC. One interesting difference is that they rematerialize the instruction pointer in places where it is necessary, which may also be a useful idea for our approach (there were a few places in Gforth that we changed so that we would not need an instruction pointer). They implemented their approach for Tcl on SPARC, and observed a lot of I-cache misses, in contrast to us (that is probably mainly due to the differences between the Tcl and the Forth VMs and benchmarks).

Qemu is a 386 emulator using dynamic translation. It achieves portability by compiling pieces of C code and concatenating the resulting machine code fragments. For filling in constants, it uses linker relocation information, like Tempo.

Performing compilation based on an interpreter has been proposed before, in particular by specializing (partially evaluating) an interpreter for a specific application program [JGS93, GMP$^+$00]. With the exception of Tempo, those works ignore retargeting and leave it to the back-end of the specializer or (for source-to-source specializers) to a later compiler run. In contrast, the main focus of the present work is on creating an easily retargetable back-end. So, despite superficial similarities, we are attacking a different problem than the partial evaluation community.

Dynamic optimizers like DynamoRIO have a hard time with interpreters; however, calling the API of an enhanced dynamic optimizer in certain places in the interpreter, these problems can be eliminated and even a good amount of speedup can be gained [SBB$^+$03]. While these optimizations can achieve some of the benefits of our approach, they do not achieve them all (e.g., the instruction pointer is not eliminated). Moreover, these optimizations are limited to platforms that have a dynamic optimizer that supports this API (currently only 386). It is also not clear how the effort for enhancing the dynamic optimizer and the interpreter compares to the effort required for our approach.

Other approaches of generating compiler back-ends have been proposed; in particular, VCODE [Eng96] and GNU Lightning provide fast code generation interfaces for dynamic code generation. Unfortunately, they suffer from the usual problems of native-code compilers: only a few platforms are supported, with no fall-back option for other platforms. Moreover, you have to use specific primitives in code generation, whereas our approach allows defining your own primitives in C, letting the C compiler optimize that, and using these primitives for interpretation or compilation.

Collberg [Col02] proposed and implemented ADT, a system that creates machine descriptions for a code generator automatically by analysing the assembly code produced by a C compiler for various test programs. Differences from the approach proposed in Section 3.6 are that the code generators produced by ADT work through an assembler and linker (less compilation speed), and that ADT learns a lot about the actual instructions and registers, whereas our approach treats the fragments as mostly opaque and discovers only the patch information.

Debaere and Van Campenhout [DV90] have proposed instruction-path coprocessing as a way to speed up interpreters: The coprocessor would interpret the program and generate a sequence of instructions for the main processor. These instructions would only perform data computations, whereas the control flow of the interpreted program and the interpreter is performed and eliminated by the coprocessor, eliminating most of the interpreter overhead. Our approach also eliminates most of the interpreter overhead, but it keeps the control flow of the interpreted program; and of course, our approach does not require special hardware.

6 Conclusion

We present an approach that fills the gap between the portability of interpreters and the execution speed of native-code compilers. We start out with code fragments extracted from a variant of an interpreter, concatenating and patching them into native code for the program we want to compile. This approach completely eliminates all references to the interpreted code and the VM instruction pointer used for referencing the interpreted code.

There are a number of technical problems to solve for implementing this approach, and we present solutions for them: The fragments and their patch information have to be determined, we have to determine relocatability, and deal with non-relocatability.

This approach results in execution speed comparable to simple native-code compilers, in speedups of up to 5.2 over a fast threaded-code interpreter, and up to 1.87 over an interpreter with dynamic superinstructions and replication (the fastest interpretive technique). The cost of implementing this approach in an existing interpreter was about two person-weeks for the first target (386), and a day for the second target (PPC).

Acknowledgements

The referees of PLDI'03 and PACT'04 provided helpful comments on earlier drafts of this paper. Ulrich Neumerkel and Peter Sestoft provided insight into compiler construction with partial evaluation.

7 References

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

[Col02] Christian S. Collberg. Automatic derivation of compiler machine descriptions. *ACM Transactions on Programming Languages and Systems*, 24(4):369–408, July 2002.

[DV90] Eddy H. Debaere and Jan M. Van Campenhout. *Interpretation and Instruction Path Coprocessing*. The MIT Press, 1990.

[EG03] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *IVME '04 Proceedings*, pages 7–14, 2004.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[Eng96] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 160–170, 1996.

[GMP+00] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[HATvdW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, September 1999.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[NHCL98] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimantal study. In *IEEE International Conference on Computer Languages (ICCL'98)*, pages 123–142, 1998.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996.

[SBB+03] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 50–57, 2003.

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pages 42–50, 2004.