Compiler-guaranteed Safety in Code-copying Virtual Machines

Gregory B. Prokopski, Clark Verbrugge
Sable Research Group
School of Computer Science, McGill University
Montreal, Quebec, Canada
{gproko,clump}@sable.mcgill.ca

Abstract. [1]Virtual Machine authors face a difficult choice between low performance, cheap interpreters, or specialized and costly compilers. A method able to bridge this wide gap is the existing *code-copying* technique that reuses chunks of the VM's binary code to create a simple JIT. This technique is not reliable without a compiler guaranteeing that copied chunks are still functionally equivalent despite aggressive optimizations. We present a proof-of-concept, minimal-impact modification of a highly optimizing compiler, GCC. A VM programmer marks chunks of VM source code as *copyable*. The chunks of native code resulting from compilation of the marked source become addressable and self-contained. Chunks can be safely copied at VM runtime, concatenated and executed together. This allows code-copying VMs to safely achieve speedup up to 3 times, 1.67 on average, over the *direct* interpretation. This maintainable enhancement makes the code-copying technique reliable and thus practically usable.

1 Introduction

Virtual Machines (VMs) are used as a target compilation architecture by many languages. The most widely known example is Java, but the same is true of a host of languages with dynamic properties, including Python, PHP, Perl6, Forth and many others. The choice of the operations represented by the virtual assembly (bytecodes) and the construction of a Virtual Machine differ for each language but they all require a virtual machine, and thus also a translation mechanism involving either the use of a cheap but slower *interpreter* or the use of a more dynamic just-in-time or ahead-of-time compiler that generates better optimized code, at greater cost. For many environments efficiency remains important, but the development and maintenance costs of an optimizing compiler are outweighed by the simplicity and rapid development time of an interpreter-based VM.

Code-copying has been proposed as a VM interpreter implementation technique that improves performance, reducing the gap between interpreters and

[1] This work is to apper in LNCS: International Conference on Compiler Construction 2008 proceedings.

compilers [5, 10]. In this work we address the main safety, practical implementation and maintenance problems inherent in such a technique that were left mostly unsolved by the previous works. Our design builds on the well-known GCC compiler to ensure semantic guarantees appropriate for code-copying in VM designs. This allows dynamic code construction and interpretation with good efficiency versus maintenance tradeoffs. Supporting language enhancements in a continually evolving, optimizing compiler such as GCC can be complex; we thus further show how changes to the basic VM compiler itself can be minimally intrusive, requiring changes dependent mainly on core, stable internal compiler structures. Low maintenance and easily isolated changes are important practical requirements for a feasible system.

An attractive feature of supporting advanced interpreter execution designs is that a static compiler such as GCC can become an effective back-end for multiple VM architectures. This provides optimized execution at low cost for a number of interpreted languages. We provide experimental data from an implementation based on the SableVM Java Virtual Machine [5]. Our results show that our automatic and verified safe design is able to match, and sometimes exceed that of previous, labour-intensive, hand-done and unverified attempts. This demonstrates the viability of our approach in terms of performance and portability.

We make the following specific contributions:

- We develop safe and practical code-copying techniques appropriate for a high-performance interpreter using GCC as a back-end. This also allows us to provide previously elusive safety guarantees for the code-copying technique.
- Our approach ensures a maintainable design within the context of GCC itself and should also be applicable to other compilers. Ensuring safety in code-copying could be performed by large, invasive efforts at nearly all levels of compilation; instead our technique minimizes the impact on general GCC development to insertion of few well-separated phases: initial code alterations and insertion of copyable code areas markers (early phases), restoration of basic block order and other properties of copyable code areas (after most of optimizations), and final verification (after all optimizations).
- Our work provides an attractive, single-compiler solution with potential for use in a variety of different programming languages and virtual machines. This takes advantage of the ubiquity and continuous development of a major compiler framework such as GCC.

In the next section we give related work on code-copying and other interpreter optimization techniques. Section 3 then gives background on code-copying techniques and requirements. Our design and GCC modifications are detailed in Section 4, and Section 5 provides some experimental results from our implementation.

2 Related Work

In our work we are concerned with optimizing interpreter-based VMs by enabling them to practically and safely use the *code-copying* technique. This technique originates from *direct-threaded* interpretation and was first described by Rossi and Sivalingam [10]. Compilers used at that time did not use too many optimizations that would make code-copying impossible, but their solution also did not give safety guarantees.

Gagnon was the first to use the code-copying technique in a Java interpreter [5]. While this implementation solved some important problems specific to the interpretation of Java bytecode, its code-copying engine required manual tuning that could not give guarantees of safe execution and therefore could not be regarded as a production-ready solution. Interestingly, experiments done with a simple, non-optimizing portable JIT for SableVM (SableJIT [1]) showed that such a JIT was only barely able to achieve speeds comparable to the code-copying engine. This demonstrated once again that code-copying is a very attractive solution, save only for its lack of safety.

One of the important reasons why code-copying is significantly faster than other interpretation techniques is its positive influence on the success rate of branch predictors commonly used in today's hardware containing branch target buffers (BTB). As Ertl showed in his work on indirect branch prediction in interpreters [4] other solutions that improve branch prediction, like bytecode duplication, can also give significant performance improvement. Speedup due to branch prediction improvements much outweighs other negative effects such as increased instruction-cache misses.

A solution similar to a code copying engine is a JIT using code generated by a C compiler, as developed by Ertl [2]. In this solution, however, the pieces of code were actually modified (patched) on the fly, so as to contain immediate values and remove the need for the instruction counter. Due to the patching architecture-specific code was necessary. Ertl's solution did include automated tests to detect code chunks that were definitely not copyable, but it was not guaranteed to find all such chunks (see Figure 6 in [9] for an example) and thus did not ensure safety. Our solution can not only detect non-copyable code but actually change a formerly non-copyable code chunk into a copyable one.

Other solutions include systems like DyC [6] which dynamically recompiles programs during their execution to benefit from run-time values allowing for optimizations based on partial evaluation. There also exist portable JITs like GNU Lightning, but these often come with support for limited number of platforms and their own limited set of code primitives.

Specialized interpreters are another route to optimized performance. In Vmgen the VM system can be trained on a set of programs to detect the most often occurring small sequences of bytecodes and then modify the source of the interpreter to combine these sequences into superinstructions, optimized the next time the interpreter is recompiled [3]. While the speed benefits of this solution are indisputable, it still requires non-automated training, selection of the set of training programs and interpreter recompilation.

Another optimization based on exploitation of frequently occurring bytecode sequences were shown by Stephenson under the name of *multicode substitution* [11]. He showed that to limit the total number of instructions (including those created by the optimization itself) such an approach must be combined with careful selection of sequences based on how well a sequence of bytecodes can be optimized.

A completely different approach to execution of bytecode was taken by GCJ and LLVM. GCJ is a GCC-based Ahead-Of-Time compiler, including also a direct-threaded interpreter for dynamically loaded code. GCJ takes as its input either Java source or Java bytecode (class files) and compiles them to an architecture-specific executable. LLVM is a compilation framework created for lifelong program analysis that features its own code representation, own compiler and other tools that make it very extendable and reusable.

3 VM Execution and Code-Copying

Figure 3 shows a rough taxonomy of the different kinds of execution engines used by Virtual Machines; in general this is through an interpreter or compiler, though mixed designs are also possible [12]. On the right side of Figure 3 compiler approaches translate streams of bytecodes into native machine code, either Ahead-Of-Time, where the compiled code is stored and made ready for multiple, repeated execution, or Just-in-Time, compiling the code just prior to execution and (typically) discarding the result after the program is completed. Compilation is desirable for performance, but implies a very non-trivial resource commitment not always available to VM designers.

Interpreters have the advantage of simplicity, although improved performance is possible with different design approaches. We illustrate the main designs on the left side of Figure 3 to situate the code-copying approach; these include a basic *switch-threaded* interpreter, and a *direct-threaded* model.

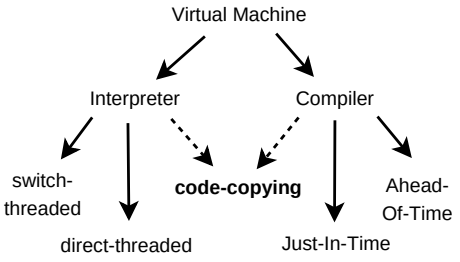

Fig. 1. The taxonomy of Virtual Machines execution engines.

A *switch-threaded* interpreter simulates a basic fetch, decode, execute cycle, reading the next bytecode to execute and using a large *switch-case* statement to branch to the actual VM code appropriate for that bytecode. This process is straightforward but if, such as in Java, bytecodes often encode only small operations the overhead of fetching and decoding an instruction is proportionally high, making the overall design quite inefficient.

A *direct-threaded* interpreter is a more advanced interpreter that minimizes decoding overhead. This kind of interpreter requires an extension offered by some compilers known as *labels-as-values.* Many operating systems, their tools

and VMs are written in C or its close derivatives. Normally a C program can contain *goto*s only to labels. With the labels-as-values extension it is possible to take an address of a label and store it in a pointer type variable. Later this variable can be used as an argument of a goto. In a direct-threaded interpreter a stream of bytecodes is thus replaced by a stream of addresses of labels. The labels themselves are placed at the start of the code responsible for the execution of operations encoded by each bytecode. With this mechanism the interpreter can immediately execute a direct *goto* to the right chunk of code. Optimization is implied by reducing the repeated decoding of instructions, trading repeated test-and-branch sequences for a one-time preparatory action where a stream of bytecodes is translated into a stream of addresses.

It is important to notice that the speed advantage of a direct-threaded interpreter over a switch-threaded interpreter already comes with the requirement of additional, specialized support from the compiler used to compile the interpreter.

3.1 Code-copying technique

In some sense, and as indicated in Figure 3, code-copying[2] bridges interpreter and compiler-based VM implementation approaches. Code-copying is a further optimization to interpreter design, but one which makes relatively strong assumptions about compiler code generation. The basic idea of code-copying is to make use of the compiler applied to the VM to generate binary code for matching bytecodes. Parts or *chunks* of the VM code are used to implement the behaviour of each bytecode. Those chunks of code are marked with labels at their beginning and end. At runtime, the interpreter copies the binary chunks corresponding to an input stream of bytecodes and concatenates them into a new place in memory, as shown in Figure 2. Such a set of concatenated instructions is called a *superinstruction* and it can execute at a much greater speed than using any of the other two formerly described techniques. Depending on the application and other factors the code-copying technique can give from 1.2 to 3 times performance gain [5] over the direct-threaded technique.

3.2 Safety

As numerous studies have shown the performance gains from using code-copying technique are clear [4, 5, 10]. However one of the biggest problems the implementators of code-copying VMs face is ensuring that the fragments of the code chunks copied to construct superinstructions are still fully functional in their new locations and as parts of superinstructions.

Unfortunately, the C standard does not contain any semantics that would allow us to express and impose the necessary restrictions on selected parts of code. For instance the bracketing labels placed before and after source code of

[2] Note that in the literature what we call code-copying is sometimes referred to as *inlining* or *inline-threading* [5]; these latter terms, however, we find, mislead most compiler developers and researchers to think of inlining of functions or methods.

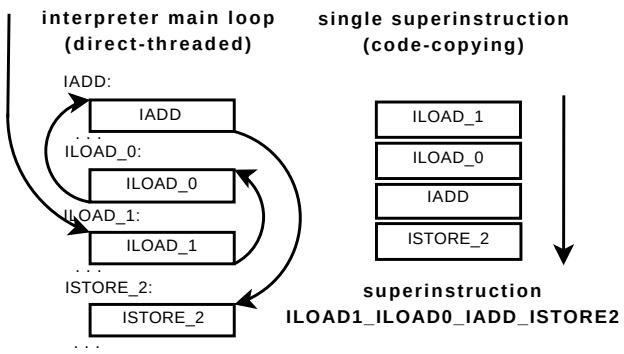

Fig. 2. A simplified comparison of direct-threaded and code-copying engines.

chunks and used to address them do not guarantee contiguity of the resulting binary code chunks, nor do they place restrictions on the use of relative addressing. Without ensured contiguity compiler optimizations will often relocate basic blocks of a chunk outside of the bracketing labels. At VM runtime this will result in incomplete copies of such a code chunk. The use of relative addressing of jump or call targets outside of a code chunk will make the copies of such chunk contain jumps or calls to invalid addresses. These and other related serious issues have to be handled, otherwise virtual machine crashes or undefined behavior are to be expected. To the best of our knowledge there is no production-quality solution that would ensure creation of code chunks by an optimizing C compiler that can be safely copied and executed.

Without guaranteed safety of code-copying an interpreter cannot practically, reliably make use of this powerful technique. Previous results used hand-done examination, trial-and-error, and manual porting combined with specialized test suites [9] in attempt to ensure safety. The large effort required, and the lack of a fully verified result motivates our design in the next section.

4 Design

For VM designers our approach requires the additional use of simple identifiers bracketing copyable code. We make use of the well-known *#pragma* operator to surround and thus help identify copyable chunks. The bulk of our design effort is in ensuring safety for code copying, a result guaranteed by a small set of well-specified additional passes within GCC. Below we first detail requirements for code to be *relocatable* and thus suitable for code-copying, followed by a description of the GCC modifications, including the final verification phase.

4.1 Generation of safely copyable code

There are specific requirements that a chunk of code has to meet so it could be copied to another location in memory, concatenated with other chunks and

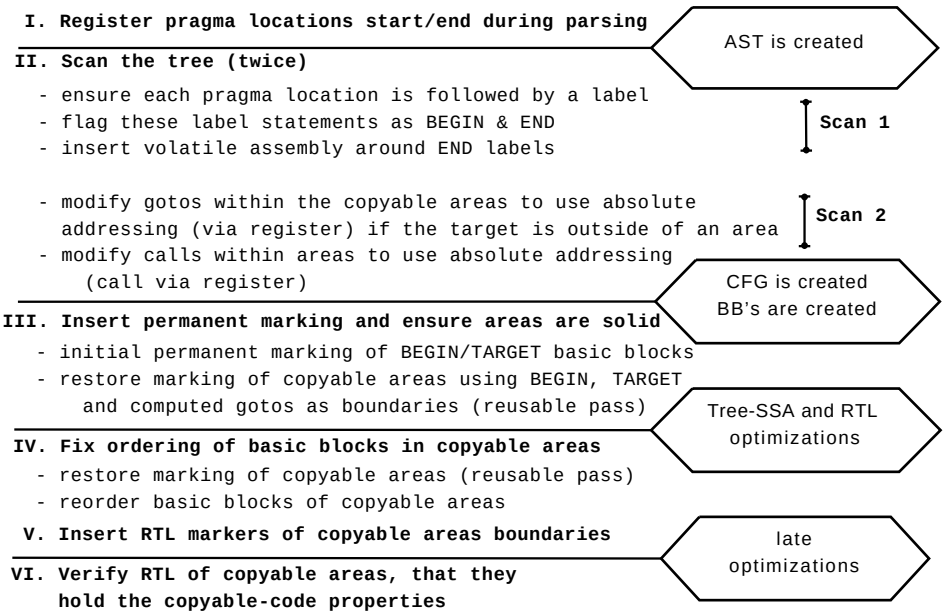

Fig. 3. To produce copyable code with minimal changes to the internal structure of the compiler we inserted several well isolated passes.

safely executed. A code chunk can only be safely copied if its copy is *functionally equivalent*, i.e. chunk of code $C_{baseaddr\alpha} \equiv C_{baseaddr\beta}$ where $\alpha \neq \beta$.

We thus define a chunk of code C to be *copyable* if all of the following conditions ensuring functional equivalence are true:

- C occupies a single contiguous space in memory that starts and ends with two distinct code labels specified by a programmer.
- Natural control flow enters C only at its "top" and exits only at its "bottom."
- Any jump from inside of C to code outside of C (e.g. to an exception handler) uses an absolute target address.
- Any jump from the inside of C to another place inside C uses a relative target address.
- Any function call from inside of C uses an absolute target address.
- At C boundaries registers must be used consistently with other code chunks boundaries (this is already ensured by GCC's computed goto extension).

4.2 GCC modifications

Our goal was to modify a highly optimizing C compiler, such as GNU C Compiler 4.2, to selectively generate code that meets these requirements therefore ensuring functional equivalence of selected code chunks.

To compile a single function GCC executes several dozens of optimization passes. These passes modify the code in ways that are usually supposed to improve the speed of the resulting code, or its other parameters. It is not feasible to

modify and maintain all of these passes to selectively generate code conforming to our requirements. Instead we modify the compiler to:

- preserve the information about which parts of the code have to be treated specially—from the moment the source code is parsed to the moment the final assembly is generated,
- allow (almost) all of the optimizations to execute without modifications and then at certain selected points of the compilation process use additional passes that modify the code in a manner that makes selected code chunks copyable.

The overall set of modifications is divided into separate passes that collectively track or restore information throughout the whole compilation process; a general description is shown in Figure 3. Depending on the representation of the code at each stage of compilation this information is tracked in a different form. In the source code it exists as *#pragma* lines, then as special flags of selected AST elements, later we attach it to basic blocks and *computed goto's,* and eventually it is inserted in a form of *notes* into the assembly. Tracking this information turned out to be the most difficult part of our work. It is because of all the aggressive optimizations that might duplicate, remove, and move parts of the code in which we are interested that ensuring copyable code is non-trivial. We ensure that this information is not lost, misplaced or mangled by separating it from structures accessed by optimization passes where possible and by employing multiple sanity checks in each of our passes that use this information.

Phase I: Code parser pragma hook Figure 5 illustrates a fragment of interpreter source code for a single code chunk. The code of an instruction (bytecode) is surrounded by the special *copyable #pragma* statements that mark the beginning and end of the copyable chunk.

Phase II: Scan the tree (1) To ensure chunks are properly identified and separated an initial pass is performed to check starting and ending conditions. Each location of *#pragma copyable begin* and *end* registered during parsing is checked to ensure it is followed by a label. These *start* and *end* labels have then their special *start* and *end* flags set accordingly. Finally the code is modified by artificially inserting into the stream of statements two empty *volatile assembly* instructions around the *end* label.

The volatile assembly code acts as a barrier to code movement, and is used to ensure the basic blocks directly following areas, the *target* blocks, are preserved and act as the sole and unique exits of the natural control flow from a copyable area. Our tests showed that otherwise some optimizations would attempt to remove or merge *target* blocks.

Phase II: Scan the tree (2) In most architectures control flow jumps can be *relative* or *absolute.* Relative jumps have the advantage of being (usually) smaller

instructions, but having a machine-specific limitations on the distance for which they are useful. Absolute jumps are often longer instruction sequences since the complete target address must be encoded, not just the relative displacement. As mentioned in Section 4.1 for control flow that goes outside of the copyable area *absolute* jumps are required to ensure the code behaves the same once copied. Similarly, jumps within a copyable region must use relative addressing to guarantee a copy will behave in an equivalent fashion.

Our second phase thus includes a pass to convert control flow statements that go outside of a copyable area (and not to the *target block*) to use absolute addresses for their targets. There are two cases of such control flow: a *goto* and a function call, both complicated by the fact that GCC itself does *not* produce the final binary code, rather it uses an external, platform-specific assembler program. It is in fact the assembler's role to choose the addressing mode for each call or jump; typically the shortest addressing mode to reach the target is chosen, but there is no general and relatively platform-agnostic way to specify in the assembler input that a jump or a call is to use absolute addressing. Below we describe how we ensure absolute jumps are used through the use of *computed goto*s, and then how we process the code chunk to ensure control flow is safe for copying.

Original code within a copyable area:

```
goto NullPointerException; /* label outside of the copyable area */
```

Is replaced with:

```
{ void *address = &&NullPointerException;
  /* this assembly prevents constant propagation   */
  __asm__ __volatile__ ("" : "=r" (address) : "0" (address) : "memory");
  goto *address; /* computed goto uses absolute addressing */ }
```

Fig. 4. To ensure absolute addressing a *goto* to outside of a copyable area is replaced with a specially crafted *computed goto*.

To force selected jumps and calls to use absolute addressing we modify the code of these instructions to make jumps and calls via a register. As shown in Figure 4, in C these instructions are represented respectively by a *computed goto* and a function call using a *function pointer*. A *computed goto* is a special feature of the *labels-as-values* extension of GCC used by direct-threaded engine. It is a *goto* whose argument is not a label but a variable containing the address of a label (or any other address). Using a register to hold the destination address may have a negative impact on the performance that will vary from platform to platform, or even CPU type. Here the benefits of maintainability and safety are paramount, and as we will show in Section 5 our solution is efficient in practice.

Nevertheless, more portable ways of expressing absolute addressing could slightly improve performance.

Our current system assumes that code chunks are small enough that the compiler will use optimal, relative jumps within the code of instructions found in a region. While it does not attempt to ensure intra-area jumps are relative, an appropriate pass could easily be added. Violations to this assumption, however, will still be detected in our final verification phase.

Phase III: Mark and ensure areas are solid Rather than modifying a large part of GCC to ensure properties of copyable code regions are preserved at all subsequent compilation stages, by all compilation passes, we instead inserted two additional passes. The first pass modifies the code in a way that ensures the minimal information about copyable code regions is always preserved. The second (reusable) pass uses this information and is capable of finding all the basic blocks belonging to copyable areas after arbitrary optimizations. Both passes include sanity checks mentioned earlier ensuring the additional information on code chunks is not lost or mangled.

After the source code is parsed into the stream of statements the compiler creates descriptions of basic blocks. Each such description contains pointers to the first and the last instruction that a basic block contains. We found that a basic block is a convenient unit to carry the additional information about the copyable code. It gives an easy access to smaller components of the code, like each particular instruction, while also being easily accessible via higher-level structures like the control flow graph. We extended the data structure describing a basic block to store the unique id of the copyable area a block belongs to and to store a field of utility flags. The initial marking of basic blocks is straightforward. We scan the stream of statements for labels earlier marked as *start* and *end*, and mark basic blocks with corresponding flags.

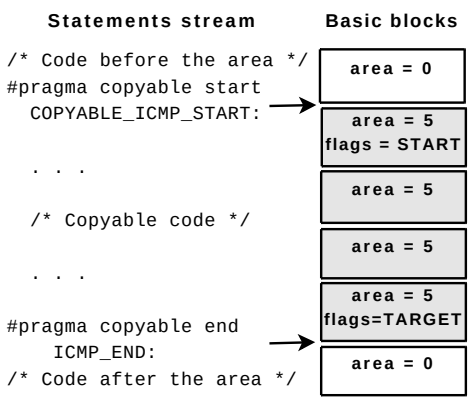

Fig. 5. Initial marking of basic blocks right after parsing.

In general optimizations can create new basic blocks, move or split existing ones. One of the possible results is that some basic blocks that functionally are part of a copyable area might no longer be placed between the *start* and *target* basic blocks of this area and might not carry the initial marking. To recover marking after optimizations we rely on the preservation of the *start* and *target* blocks, which in turn is ensured with sanity checks. Area marking restoration can then be done through simple propagation along the control flow graph, from the *start* block of each area until the *target* block and jumps via computed gotos.

It is critical that the compiler had earlier modified all the jumps to outside of copyable areas to use computed gotos. This way it is possible to always find the limits of copyable areas.

Importantly, our approach *does not use a heuristic* and is guaranteed to properly restore the list of blocks belonging to each copyable area. We still included extensive sanity checks that in practice should never be triggered. This is because, for instance, we earlier inserted volatile assembly around chunks end labels and disabled *cross-jump* optimization (see below). With these measures in place previously executed optimizations should not have inserted or deleted *start* or *target* blocks or cause the control flow graphs of different code chunks to interfere.

For functions containing copyable code we disabled *cross-jump* optimization which attempts to find identical code chunks within a function and share a single instance of the code. This clearly conflicts with with the need of the code-copying engine to use self-contained code chunks.

Phase IV: Fix basic blocks ordering The main reason for our basic block reordering pass is an optimization performed by GCC by default, *basic block partitioning*. This pass does two things. It divides the set of basic blocks of a function into those that are expected to be executed frequently (hot blocks) and those that are expected to be executed rarely (cold blocks). In the final assembly all the hot blocks of each function are located contiguously in the upper part of the code, and the cold blocks are located below the hot blocks. This optimization also reorders basic blocks to ensure that the fall-thru edges are used for the most often encountered control flow. These are heuristic techniques for improving instruction cache hit rate and simplifying control flow, and this optimization can in practice improve the performance of a virtual machine by several percent, therefore we want to allow for it.

For a chunk of code to be copyable the compiler has to restore the order of basic blocks so that the marked code is self-contained. In this case the goal is to move basic blocks to ensure that the *start* basic block of the copyable area is followed by all other blocks belonging to it, which are then followed by the *target* basic block of the same copyable area. After the marking of basic blocks belonging to all areas is restored (as described in the previous section) it is relatively easy to move all basic blocks belonging to an area into the wanted positions. Positions of other basic blocks, not belonging to copyable areas, are left unchanged.

4.3 Phase V and VI: RTL markers and final verification

The additional passes described above modify the structure of the code based on up to date information about the boundaries of basic blocks, construction of the control flow graph, and other data. During the last compilation passes the GCC compiler discards some of this information or does not keep it up to date. In our tests we found that these last optimization passes do not change the structure

of the code enough to invalidate the properties of copyable code. Nonetheless, this was not sufficient for the safety guarantees we required and another solution was needed. We therefore added two passes.

Not long before the information about basic blocks and control flow graph becomes unavailable an additional pass inserts into the program representation (*RTL* stream) special (untouchable by other passes) *notes* that mark the start and end of copyable areas, including the ID of an area. The second pass is then a simple verification pass that uses only a minimum of information. It is executed just before the final assembly is sent to the external assembler. With the *notes* inserted by our previous pass it is possible to verify all the necessary properties of copyable areas when the code is final. The verification algorithm takes each instruction from the instruction stream and ensures that:

- all copyable areas are present,
- copyable areas do not interleave with one another,
- jumps from a copyable area A are either to a target within A or to this area's target label (the label that begins the target basic block). Note that it is also necessary to ensure that all jumps within A are also within the allowable range of a relative jump[3],
- jumps to the outside of an area are made via register and not a symbol (thus are absolute),
- all calls from within areas are made via register and not a symbol (thus are absolute).

A verification error at this point is uncorrectable and is treated as an internal compiler error. This guarantees that if source code compiles properly then the copyable chunks of binary code will be safe to copy and execute in a code-copying VM. Sanity checks in all the passes ensure proper flow of the information on code chunks which allows the final verification to function reliably. In our experience we have not yet encountered a case where the verification pass would fail when all the former passes executed properly.

5 Experimental Results

To examine practicality of our design we modified a Java Virtual Machine, SableVM [5], to use our enhanced GCC. In SableVM source we marked code chunks with our *copyable #pragma*. Code-copying was already supported in SableVM, but required globally disabling block reordering in GCC and did not provide safety guarantees. During preparations we used our enhanced GCC to verify the unsafe code formerly used by SableVM's code-copying engine and found several cases where execution of a less likely control flow path in a byte-code would result in a VM crash due to a function call using relative addressing.

The goal of our main experiments was thus to demonstrate that our new approach allows the code-copying strategy to be realistically and more reliably

[3] This check has not been implemented in our current system.

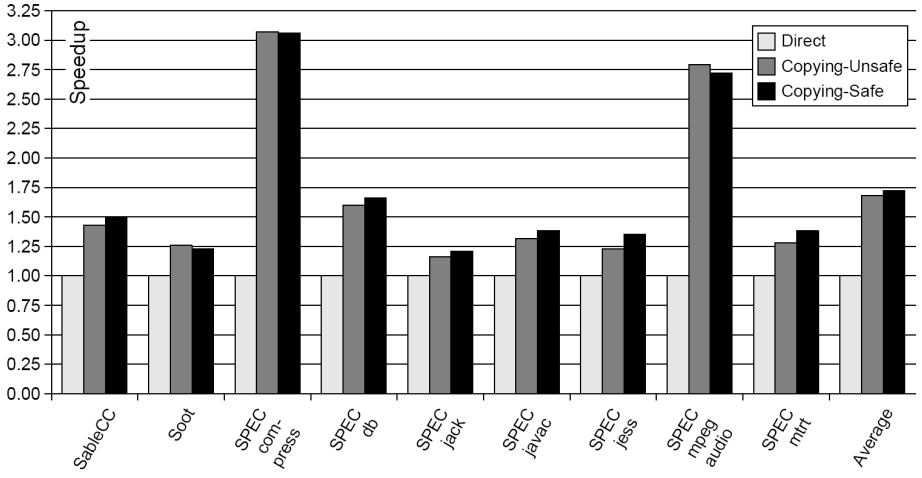

Fig. 6. Performance comparison of SableVM with standard direct-threaded engine, unsafe code-copying engine and safe code-copying engine using the GCC copyable-code enhancement.

used while maintaining the performance. The results shown in Figure 6 have been gathered using a machine with Intel Pentium IV at 3GHz, 512MB RAM. The SPEC benchmarks, averaged over 10 runs, were executed with their default settings (`-S 100`), and performance is shown normalized to the speed of the direct-threaded engine as a baseline for comparison. SableCC and Soot are large, object-oriented, in-house benchmarks.

The benefits of code-copying are clear. We are able to achieve approximate parity with the unsafe code-copying approach. More surprising perhaps is that the performance of SableVM version 1.13 modified to use our GCC extensions actually improved over the manual code-copying design in most cases. We attribute the general improvement to the fact that previously SableVM had to globally disable basic block reordering for the code-copying engine to work at all. With the added GCC support for code-copying this useful optimization was enabled. We also note that the performance of two SPEC benchmarks that benefit the most from code-copying, as well as *Soot* slightly decreased, about 2-3%. We suspect that this effect is caused by the memory barriers inserted into the code in places where the special *#pragma* is used. These barriers might be inhibiting some of the optimizations. Previously Gu et al. [7, 8], however, showed that changes to the executable code placement

Metric	#
Data structures modified	4
Fields added to data structures	6
Data structures added	3
Functions added to existing files	4
Function calls/hooks inserted	8
Code lines added or modified	139
Code lines in new files	1500

Fig. 7. Metrics of code modified and added to GCC

without actual changes to the functioning of a VM can cause a tremendous variance (up to almost 10%) in the VM performance. More detailed analysis of performance gains and losses is thus warranted, but certainly the magnitude of correlation in Figure 6 is sufficient to demonstrate the general success of our compiler-facilitated approach. Overall, the effect is clear: our modifications efficiently enable code-copying as a safe technique for VM interpreter design.

One of our goals was to minimize the impact of our changes to GCC on GCC maintenance. Figure 7 shows the results of our impact measurements in terms of required changes to code and data structures. In a truly large project such as a GCC we see these numbers as indicators that our extension has minimal impact on the existing GCC code and its maintenance. We also report that a major upgrade of our enhanced GCC from 3.4 to 4.2 (about 2 years of GCC development) took only a few hours and consisted mostly of renaming and testing. We believe this validates our claim that a relatively simple compiler modification can help improve the performance of dynamic execution environments.

6 Conclusions and Future Work

For a variety of reasons, including simplicity and dynamic support, many modern languages are based on virtual machine (VM) designs. Efficiency and ease of design are key features for rapidly evolving languages and associated execution environments.

Code-copying interpreters offer a good trade-off between performance and maintenance, but were previously limited by the lack of critical safety guarantees, as well maintenance concerns with respect to the VM compiler itself. Copyable code must behave functionally the same when copied, and while conceptually trivial these guarantees are simply not provided by current compilers or C language extensions.

With our work we demonstrate that it is possible to make code-copying safe and practical. Our approach to GCC modifications demonstrates viability of our technique for ensuring the safety properties essential to code-copying. We show how this technique can be relatively easily integrated with a modern C compiler, while keeping the changes relatively isolated and making only limited assumptions about the inner workings of a compiler, thus ensuring long-term maintainability.

The implementation of a code-copying GCC extension on which we based this paper was focused on supporting the i386 architecture. On other architectures there might be additional issues with delay slots (e.g. MIPS), relative addressing of externs and globals (e.g. x86_64), or relative-jump span limitations (e.g. PowerPC). We are currently working on incorporating the necessary detection and correction mechanisms into our GCC extension leading to full support of more architectures.

As well as deeper performance analysis, further determining the source of our gains over hand-done code-copying, our immediate future work is in the application of our technique to other VM architectures and other hardware archi-

tectures. Simplified use of code-copying could improve performance for a variety of predominantly interpreted languages, and we hope to show greater generality of our design by replicating the code-copying technique in other environments.

This research was supported in part by NSERC and FQRNT. The authors would also like to thank Etienne M. Gagnon for his suggestion regarding compiler modification as a way of making code-copying practically usable. The source code of our modified GCC 4.2 is available at `http://www.prokopski.com` .

References

1. David Bélanger. SableJIT: A retargetable just-in-time compiler. Master's thesis, McGill University, August 2004.
2. M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004.
3. M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Softw. Pract. Exper.*, 32(3):265–294, 2002.
4. M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006. Journal papers from *.NET Technologies 2006* conference.
5. Etienne M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, 2002.
6. Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. A retrospective on: "an evaluation of staged run-time optimizations in DyC". *SIGPLAN Not.*, 39(4):656–669, 2004.
7. Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in JVM performance. In *Component And Middleware Performance workshop, OOPSLA 2004*, 2004.
8. Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 111–121. ACM Press, 2006.
9. Gregory B. Prokopski, Etienne M. Gagnon, and Christian Arcand. Bytecode testing framework for SableVM code-copying engine. Technical Report SABLE-TR-2007-9, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, September 2007.
10. Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki Univeristy of Technology, May 1996.
11. Ben Stephenson and Wade Holst. Multicodes: optimizing virtual machines using bytecode sequences. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 328–329, New York, NY, USA, 2003. ACM Press.
12. T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, 2000.