Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Towards GCC as a compiler for multiple VMs

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

Abstract

Virtual Machine authors face a difficult choice: to settle for low performance, cheap interpreter, or to write a specialized and costly compiler. A solution is to reuse a robust compiler like GCC. We present a proof-of-concept, minimal-impact implementation. VM programmer marks specific chunks of VM source code as *copyable*. The native code resulting from compilation of these chunks becomes an addressable, self-contained, copyable code chunk. Chunks can be safely copied at VM runtime, concatenated and executed together. With minimal impact on GCC maintenance we guarantee the necessary properties of chunks and verify their integrity. With this technique a VM achieved performance improvement up to 200% over *direct* interpretation, while having runtime safety ensured thanks to chunks integrity verification. This maintainable enhancement shows a new way GCC can be used as a compiler for multiple VMs.

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 Python, PHP, Perl6, Forth and many others. Each of these languages uses a virtual assembly, usually called bytecode, to encode mostly simple operations performed on a Virtual Machine. The choice of the operations represented by the bytecodes and the construction of a Virtual Machine differ for each language. For example, Java uses a virtual stack-based machine, while Perl6 uses a virtual register-based machine. Despite the differences between bytecodes of different programming languages 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 costly compiler that generates better optimized code. 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 an interpreter implementation technique that improves performance, reducing the gap between interpreters and compilers [7, 11]. In this work we address the main safety and practical implementation problems inherent in such a technique. Previous attempts at code-copying did not fully solve these problems and had significant maintenance concerns. Our design builds on the well-known GCC compiler to ensure semantic guarantees appropriate for code-copying in VM designs. Supporting language enhancements in a continually evolving, optimizing compiler such as GCC can be complex; we thus further show how support changes can be minimally intrusive, requiring changes dependent mainly on core, stable internal compiler structures. Low maintenance and easily isolated changes are important practical requirements.

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 interpretted languages. We provide experimental data from an implementation based on the SableVM Java Virtual Machine [7]. 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.

Contributions

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.

- Our approach ensures a maintainable design within the context of GCC itself. Ensuring safety in code-copying could be performed by large, invasive efforts at nearly all levels of compilation; our technique minimizes the impact on general GCC development to small changes at the beginning, some data recovery, and a verification phase.
- Our work provide an attractive, single-compiler solution for 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 making *code-copying* technique practically usable. This technique originates from *direct-threaded* interpretation and was first described by Piumarta and Riccardi in their work on, what they called, *selective inlining* [11]. 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 [7, 8]. 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 [3, 6] other solution that improve branch prediction, like bytecode duplication, can also give significant performance improvement. Speedup due to branch prediction improvements much outweights other negative effects such as increased i-cache misses.

A solution similar to code copying engine is a JIT using code generated by a C compiler developped by Ertl [4]. 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.

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 [5]. 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* [12]. He showed that to limit the total number of instructions (including those created by the optimization itself) such 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 [2] and LLVM [9]. 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

Our optimized design for *code-copying* is within the context of a VM interpreter. Figure 1 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 [10]. On the right side of Figure 1 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 1 to situate the code-copying approach; these include a basic *switch-threaded* interpreter, and a *direct-threaded* model.

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*. Normally a C[1] 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 variable. Later this variable can be used as an argument to 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

[1]The C language (and its close derivatives) is the most popular language in which operating systems and their related tools, including compilers, are written. Many virtual machines are also written in C; our work thus focuses on virtual machines written mainly in the C language.

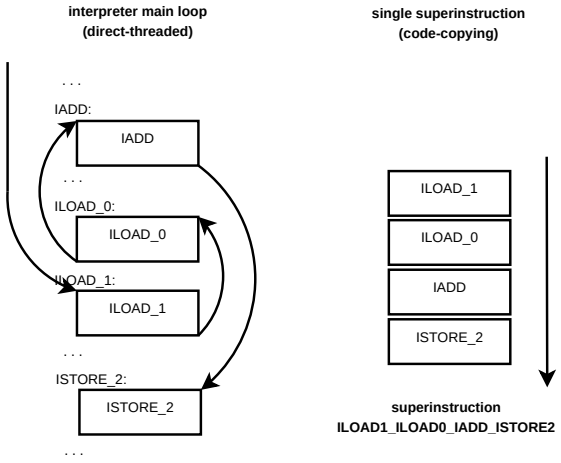

Figure 2: A simplified comparison of direct-threaded and code-copying engines

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

Code-copying[2] 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 begin 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.

[2]Note that in the literature what we call code-copying is sometimes referred to as *inlining* or *inline-threading* [7]; these latter terms, however, we find suggest method or function inlining to most compiler developers and researchers.

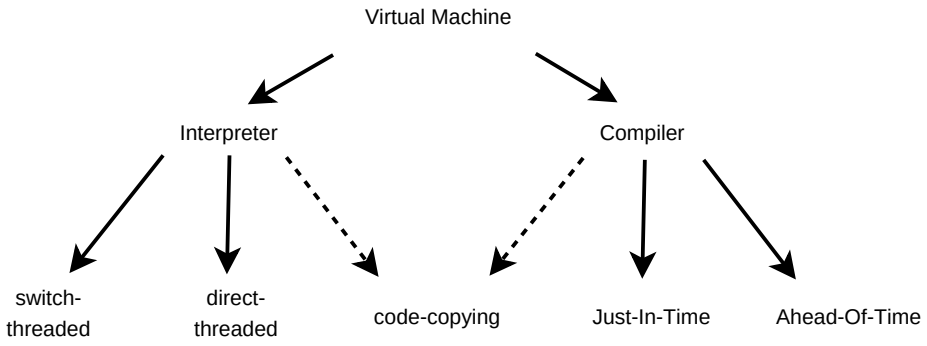

Figure 1: The taxonomy of Virtual Machines execution engines

Depending on an application and other factors the code-copying technique can give from 20% to 200% performance gain over the direct-threaded technique. There are two main reasons for this large improvement:

- Reduction of the number of dispatches. With the code-copying technique there is only one dispatch per superinstruction instead of one dispatch per instruction. This usually removes about 70% of the dispatches in superinstructions [7].
- Improvement in branch prediction. In a code-copying interpreter there are multiple copies of each instruction, each copy being a part of one of many superinstructions. In direct-threaded or switch-threaded interpreters there usually is only one copy of each instruction. Because of that the branches to the next instruction have a highly variable target, making branch misprediction rates extremely high [3].

In this way the code-copying technique removes a vast amount of dispatches and mispredicted dispatches that are especially costly on modern, highly pipelined processors.

3.2 Safety

As numerous studies have shown the performance gains from using code-copying technique are clear [6, 3, 8, 7, 11]. However one of the biggest problems the implementators of code-copying interpreter engines face is ensuring that the fragments of the code chunks copied to construct superinstructions are still fully functional in their new locations and as a part of a superinstruction. In particular, to be fully functional a code chunk must not contain relative jumps or calls to targets that would be outside of the chunk, and its control flow must start at the *top* and exit at the *bottom*. Chunks which do not possess these properties cannot guarantee safety at runtime.

Unfortunately, the C standard does not contain any semantics that would allow us to express and impose such restrictions on selected parts of code. The labels placed before and after the code chunks do not guarantee contiguity of the resulting binary code chunks, nor do they place restrictions on the use of relative addressing. Even with the sub-optimal property of disabling optimizations selectively for code chunks (let alone the entire VM) to our best knowledge there is no production-quality solution that would ensure creation of code chunks that can be safely copied and executed.

Without guaranteed safety in code-copying an interpreter cannot practically, reliably make use of this useful technique. Previous results used hand-done examination, trial-and-error [7], and manual porting combined with specialized test suites[3] to 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 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 passes within GCC. Below we first detail requirements

[3] Based on unpublished research within the SableVM framework.

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 safely executed. If a chunk of copied code does not mimic the functionality of the original it cannot be safely copied. We thus define a chunk of code *C* to be *copyable* if all of the following conditions 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.

If any of the above requirements is not met then a particular chunk of code is not copyable. Our goal was to modify a highly optimizing C compiler, such as GNU C Compiler 4.0, so it could process input chunk requests and selectively generate code that meets these requirements.

4.2 GCC modifications

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 specially crafted 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.

Phase I: Code parser pragma hook

The information about copyable areas originates from the soure code, so it is necessary to start tracking this information from the moment the source code is parsed. We plug our *#pragma* handler into the standard GCC mechanism for parsing pragmas to register the locations of *copyable pragma*s in the source code. Figure 4 illustrates a fragment of interpreter source code for a single code chunk. The first part of the code performs the initialization necessary for the code-copying engine. The second part is the actual chunk or body of a bytecode instruction. The code is surrounded by the special *copyable #pragma* statements that mark the begining and end of the copyable chunk.

GCC contains generic code for handling pragmas, so we only had to add to GCC a function that is called when this *#pragma* is encountered. This function records the position of *#pragma*s in the source code which are the beginning and ending positions that encompass each *copyable area.* At this stage the compiler also performs sanity checks and warns about doubly started or open copyable areas.

In GCC inlining of functions is done very early, soon after the parsing is completed. The result of parsing is a

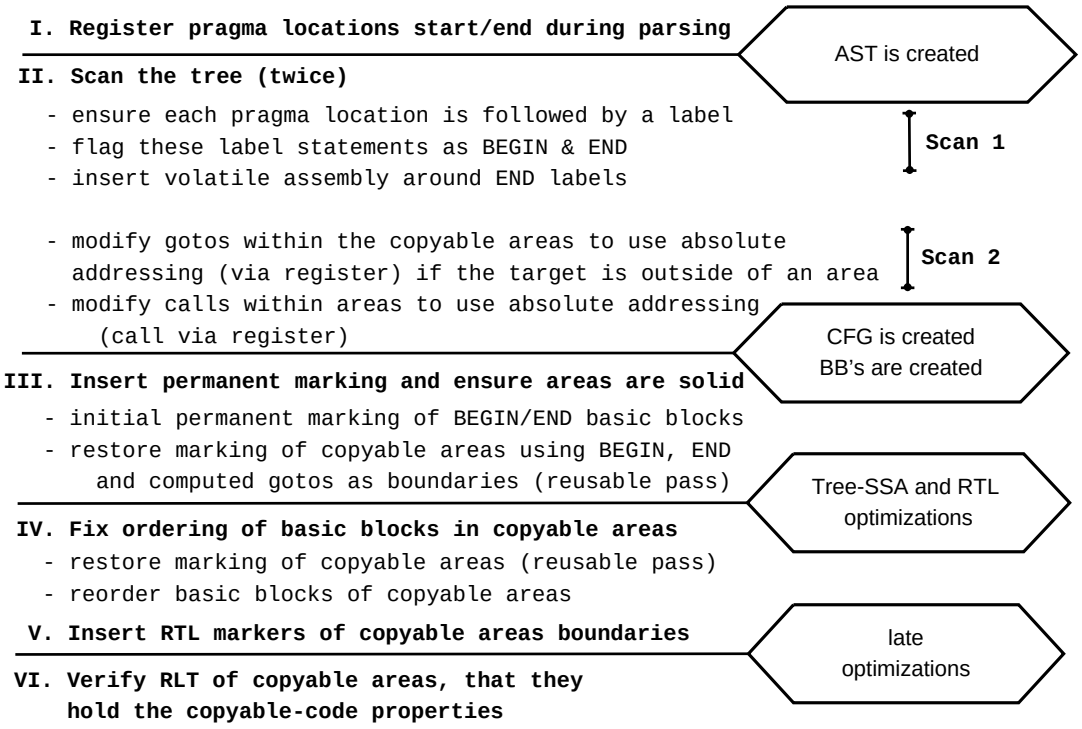

Figure 3: To produce copyable code with minimal changes to the internal structure of the compiler we inserted several well isolated special passes.

```
  case SVM_INSTRUCTION_LCMP:
      { /* instruction initialization */
        vm->instructions[instr].param_count = 0;
        vm->instructions[instr].copyable_code = &&COPYABLE_START_LCMP;
        env->vm->instructions[instr].copyable_size =
          ((char *) &&END_LCMP) - ((char *) &&COPYABLE_START_LCMP);
        break;
      }

#pragma copyable begin
      COPYABLE_START_LCMP:
        { /* instruction body */
          jlong value1 = *((jlong *) (void *) &stack[stack_size - 4]);
          jlong value2 = *((jlong *) (void *) &stack[stack_size - 2]);
          stack[(stack_size -= 3) - 1].jint =
            (value1 > value2) - (value1 < value2);
        }
#pragma copyable end
        END_LCMP:
```

Figure 4: Pragma directives are placed around the code that will be used by code-copying engine at runtime.

```
Original source code:

#pragma copyable end
        END_LCMP:

Is changed into:

__volatile__ __asm__ ("":::"memory");
        END_LCMP:
__volatile__ __asm__ ("":::"memory");
```

Figure 5: Volatile statements are inserted around the *end label* to ensure that the *target* basic block will remain intact throughout optimizations.

stream of statements describing the parsed function and does not contain information about *#pragma*s. However each statement has attached information about the source code location from which it was created. Integrating our previously gathered information on the location of *#pragma*s allows us to identify the code of copyable areas within the stream of statements.

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 emphend 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, as shown in Figure 5.

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. In principle a similar concern applies to the first basic block of a copyable area, the *start*ing block. However in our tests the compiler would never try to remove or duplicate this block. We did not investigate it further, but if it did it could always be handled the same way as in *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 a similar 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.

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 6, 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 improve performance further.

Our pass scans each copyable area for *goto*s having as

```
Original code within a copyable area:

  goto labelX; /* where labelX is outside of the copyable area */

Is replaced with:

  {
    void *address = &labelX;

    /* this assembly claims to read and modify address
     * and in this way prevents constant propagation   */
    __asm__ __volatile__ ("" : "=a" (address) : "memory");

    goto *address; /* computed goto uses absolute addressing */
  }
```

Figure 6: To ensure absolute addressing a *goto* to outside of a copyable area is replaced with a specially crafted *computed goto*.

their targets a label outside that area; a similar process is applied to function calls. As show in the Figure 6, each such *goto* is replaced with a stream of statements that force the compiler to use a *computed goto,* and therefore an absolute addressing mode. First a declaration of a variable is inserted into the stream of statements. At runtime this variable will contain the address of the target. Then an empty volatile assembly statement is inserted. This statement claims to use and modify the value of the variable. Then a new *goto* is created to replace the existing one. The target of the new *goto* is the address stored in the variable. We do not concern ourselves with pre-existing *computed goto*s as these already use absolute addresses.

Our current system assumes that instructions are small enough that the compiler will use optimal, relative jumps within the code of instructions found in a region, and so does not attempt to ensure intra-area jumps are not absolute. Violations to this assumption, however, will 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.

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. 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.

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. Note that the initial marking, as shown in Figure 8 might not be preserved by the optimizations performed later which might split, join, duplicate, and delete basic blocks. It is therefore necessary to have a method of restoring the marking after the optimizations, if we are to be able to to detect which basic blocks belong to a copyable area.

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 not anymore be placed between the *start* and *target* basic blocks of this area and might not carry the initial marking. An example is shown in Figure 7 where the last basic block (**BB6**), even though placed far after the *start* and *target* basic blocks, clearly

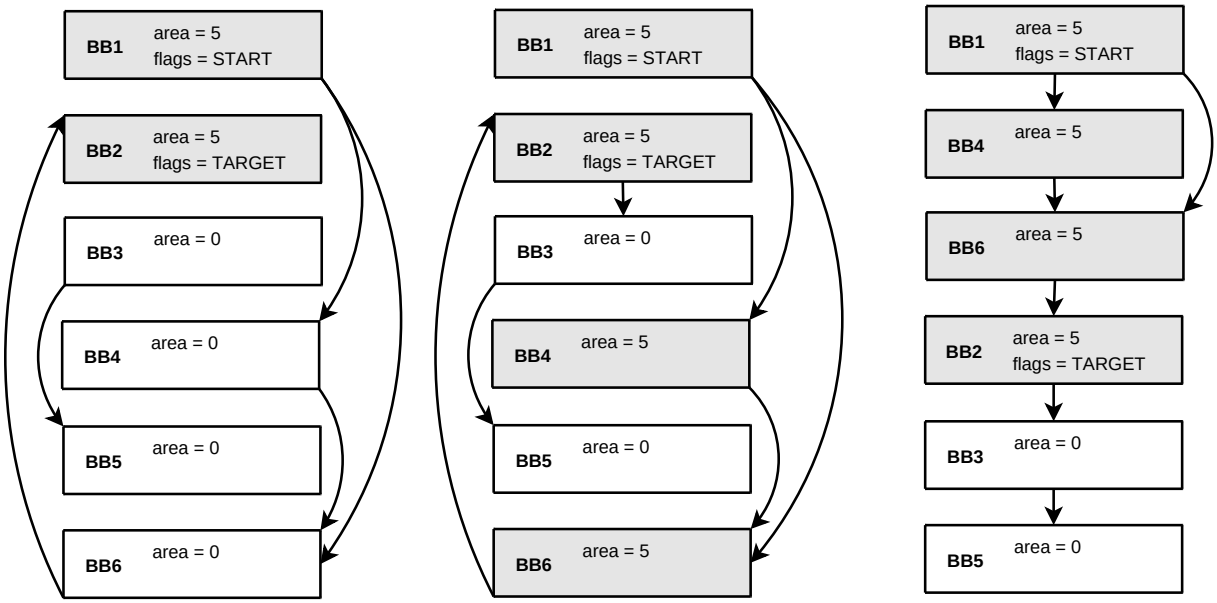

Figure 7: From the marking of only two basic blocks, *start* and *target*, the complete marking can be restored by following the edges of the control flow graph. Once the marking is restored it is possible to rearrange the basic blocks of a marked copyable area.

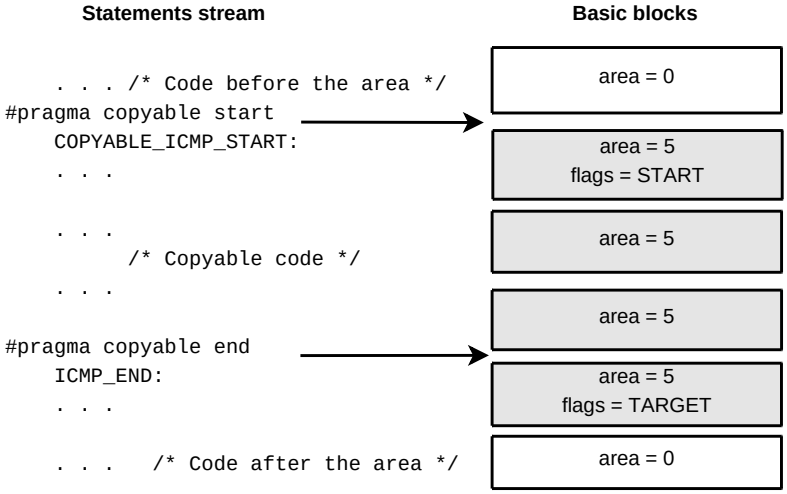

Figure 8: Initial marking of basic blocks right after parsing

belongs to the control flow of this copyable area.

To recover marking after optimizations we rely on the preservation of the *start* and *target* blocks. 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 either properly restore the list of blocks belonging to a copyable area or fail explicitly (which is reported as an internal compiler error).

During the marking restoration process the compiler performs several sanity checks. It ensures that both *start* and *target* blocks exist for all initially existing copyable areas. It detects an attempt to mark a basic block twice, from two different areas. Each of these problems is reported as an internal compiler error, as that kind of situation should never arise because the lists of basic blocks belonging to copyable areas can never intersect or overlap.

There exists one optimization performed by the compiler that has to be disabled for a function that uses copyable areas. This optimization, in GCC, is called *cross-jump*. It attempts to find parts of code within a function that are identical and then share a single copy of the code among all the places in a function where this code is used. This optimization clearly conflicts with the need of the code-copying engine to use self-contained code chunks and has therefore always been useless in this context. We trivially modified the pass controller in GCC so that this optimization is automatically disabled for functions that use copyable areas. This selective approach does not change the way all the rest of code of Virtual Machine is compiled.

Phase IV: Fix basic blocks ordering

Our initial marking pass ensured that the minimal information about the copyable areas is always preserved or recovered. However, physical ordering of blocks is not necessarily guaranteed—blocks for areas may not be all located within the *start* and *target* blocks after optimizations are complete. A further pass is thus applied to reorder basic blocks and ensure that code belonging to an area is contained within its *start* and *target* blocks.

The main reason for basic block reordering 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 function, 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.

It is of course possible to disable this optimization on a per-function basis. This was deemed unsatisfactory for two reasons. First, we perceive the fall-thru edges optimization as a welcomed attempt to improve the quality of the resulting code later used for code-copying. Secondly, we have to be aware that there are other optimization passes that can also relocate basic blocks. With or without block partitioning we had to create a solution that would be able to deal with any kind of relocation of basic blocks.

For a chunk of code to be copyable the compiler has to restore the order of basic blocks so that the 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, as illustrated in Figure 7. 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. Our copyable region data is therefore again out of date for these final passes. 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 strong 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 a special pass inserts into the program representation (*RTL* stream) special *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 the previous special 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[4]
- 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 a code compiles properly then the copyable chunks of code will be safe to copy and execute in the code-copying engine. In our experience we have not yet enountered a case where the verification pass would fail when all the former passes executed properly.

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

5 Experimental Results

To examine practicality of our design we modified a Java Virtual Machine, SableVM [8], to use our modified GCC and mark 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. The goal of our experiments was thus to demonstrate that our new approach allows the code-copying strategy to be realistically and more reliably used while maintaining or improving performance.

The results shown in Figure 9 have been gathered using a machine with Intel Pentium IV at 3GHz, 512MB RAM. The SPEC benchmarks were run with their default settings, and performance is shown normalized to the speed of the direct-threaded engine as a baseline for comparison.

The performance of SableVM version 1.13 modified to use the extension actually improved in most cases with our technique. Although the performance improvement was not our goal, 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 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. Overall, however, the effect is clear, our modifications efficiently enable code-copying as a safe technique for VM interpreter design.

Interestingly enough, in other series of experiments, where we attempted to enable code-copying for as many bytecodes as possible (mainly Java bytecodes of conditionals), we found that making many more bytecodes copyable actually decreased the overall performance of the code-copying engine. This may be due to lower i-cache hit rate caused by bigger number of superinstructions, or again due to increased use of barriers and indirect jumping. We intend to further investigate this as near future work.

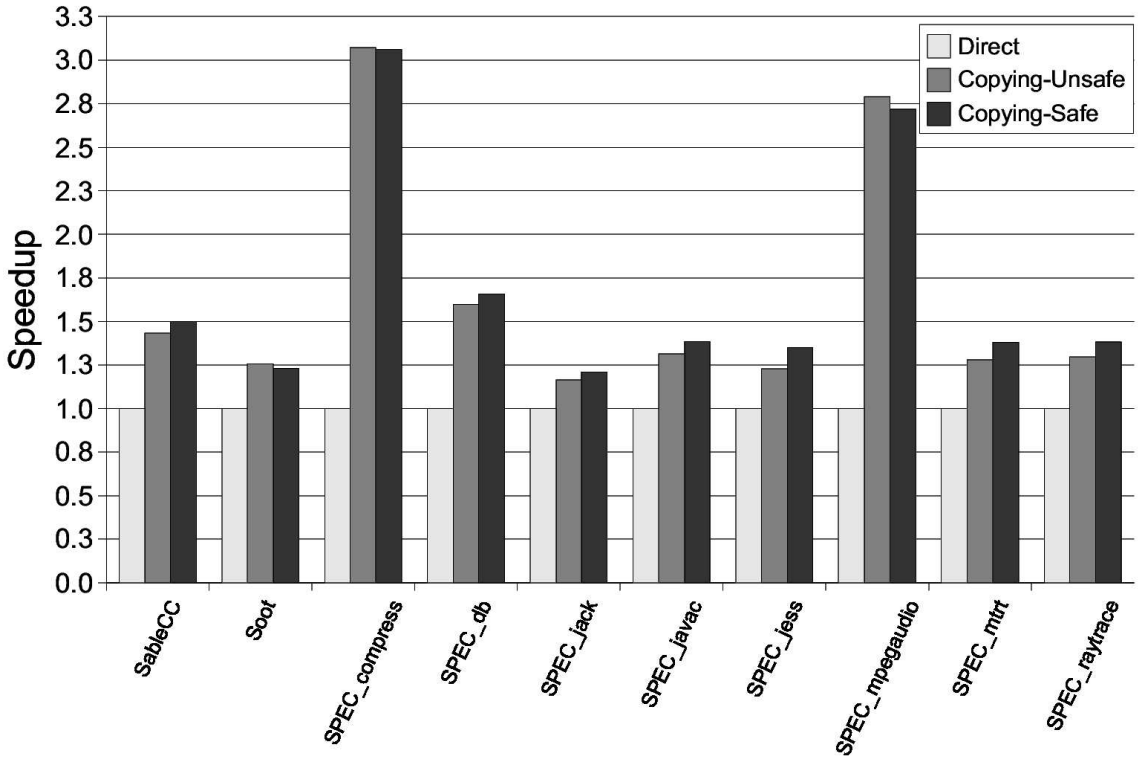

Figure 9: Comparison of the performance of SableVM with direct-threaded engine, unreliable code-copying engine and code-copying engine using the GCC copyable-code extension.

6 Conclusions and Future Work

Code-copying interpreter designs depend on a small, but important semantic understanding of how code is generated from source. Copyable code must behave functionally the same when copied, and while conceptually simple these strong guarantees are simply not provided by current compiler or C language extensions. Our work here shows how the safety properties essential to code-copying can be practically guaranteed in the standard GCC compiler. Our design, moreover, is relatively isolated and makes only a few assumptions about GCC behaviour. The choice of many key elements of our design is in fact driven by the need to ensure that maintainability of GCC development is well separated from our modifications.

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. Simplified use of code-copying could enable 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.

References

[1] David Bélanger. SableJIT: A retargetable just-in-time compiler. Master's thesis, McGill University, August 2004.

[2] Per Bothner. Compiling java with gcj. *Linux J.*, 2003(105):4, 2003.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Etienne Gagnon and Laurie Hendren. Sablevm: A research framework for the efficient execution of java bytecode. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[8] Etienne M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, 2002.

[9] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Geetha Manjunath and Venkatesh Krishnan. A small hybrid jit for embedded systems. *SIGPLAN Not.*, 35(4):44–50, 2000.

[11] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1998. ACM Press.

[12] 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.