Compiler Techniques for Code Compaction

SAUMYA K. DEBRAY and WILLIAM EVANS
The University of Arizona
ROBERT MUTH
Compaq Computer Corp.
and
BJORN DE SUTTER
University of Ghent

In recent years there has been an increasing trend toward the incorporation of computers into a variety of devices where the amount of memory available is limited. This makes it desirable to try to reduce the size of applications where possible. This article explores the use of compiler techniques to accomplish code compaction to yield smaller executables. The main contribution of this article is to show that careful, aggressive, interprocedural optimization, together with procedural abstraction of repeated code fragments, can yield significantly better reductions in code size than previous approaches, which have generally focused on abstraction of repeated instruction sequences. We also show how "equivalent" code fragments can be detected and factored out using conventional compiler techniques, and without having to resort to purely linear treatments of code sequences as in suffix-tree-based approaches, thereby setting up a framework for code compaction that can be more flexible in its treatment of what code fragments are considered equivalent. Our ideas have been implemented in the form of a binary-rewriting tool that reduces the size of executables by about 30% on the average.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*; E.4 [**Coding and Information Theory**]: Data Compaction and Compression—*program representation*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Code compaction, code compression, code size reduction

---

The work of Saumya Debray and Robert Muth was supported in part by the National Science Foundation under grants CCR-9711166, CDA-9500991, and ASC-9720738. The work of Bjorn De Sutter was supported in part by the Fund for Scientific Research—Flanders under grant 3G001998.
Authors' addresses: S. Debray and W. Evans, Department of Computer Science, University of Arizona, Tucson, AZ 85721; email: {debray, will}@cs.arizona.edu; R. Muth, Alpha Development Group, Compaq Computer Corporation, Shrewsbury, MA 01749; email: Robert.Muth@compaq.com; B. De Sutter, Department of Electronics and Information Systems, University of Ghent, B-9000 Gent, Belgium; email: brdsutte@elis.rug.ac.be.

1. INTRODUCTION

In recent years there has been an increasing trend towards the incorporation of computers into a wide variety of devices, such as palm-tops, telephones, embedded controllers, etc. In many of these devices, the amount of memory available is limited due to considerations such as space, weight, power consumption, or price. At the same time, there is an increasing desire to use more and more sophisticated software in such devices, such as encryption software in telephones, or speech or image processing software in laptops and palm-tops. Unfortunately, an application that requires more memory than is available on a particular device will not be able to run on that device. This makes it desirable to try to reduce the size of applications where possible. This article explores the use of compiler techniques to accomplish this code compaction.

Previous work in reducing program size has explored the compressiblity of a wide range of program representations: source languages, intermediate representations, machine codes, etc. [van de Wiel 2000]. The resulting compressed form either must be decompressed (and perhaps compiled) before execution [Ernst et al. 1997; Franz 1997; Franz and Kistler 1997], or it can be executed (or interpreted [Fraser and Proebsting 1995; Proebsting 1995]) without decompression [Cooper and McIntosh 1999; Fraser et al. 1984]. The first method results in a smaller compressed representation than the second, but requires the overhead of decompression before execution. Decompression time may be negligible and, in fact, may be compensated for by the savings in transmission or retrieval time [Franz and Kistler 1997]. A more severe problem is the space required to place the decompressed code. This also has been somewhat mitigated by techniques of partial decompression or decompression-on-the-fly [Beneš et al. 1998; Ernst et al. 1997], but these techniques require altering the run-time operation or the hardware of the computer. In this article, we explore "compaction," i.e., compression to an executable form. The resulting form is larger than the smallest compressed representation of the program, but we do not pay any decompression overhead or require more space in order to execute.

Much of the earlier work on code compaction to yield smaller executables treated an executable program as a simple linear sequence of instructions, and used procedural abstraction to eliminated repeated code fragments. Early work by Fraser et al. [1984] used a suffix tree construction to identify repeated sequences within a sequence of assembly instructions, which were then abstracted out into functions. Applied to a range of Unix utilities on a Vax processor, this technique managed to reduce code size by about 7% on the average. A shortcoming of this approach is that since it relies on a purely textual interpretation of a program, it is sensitive to superficial differences between code fragments, e.g., due to differences in register names, that may not actually have any effect on the behavior of the code. This shortcoming was addressed by Baker [1993] using parameterized suffix trees, by Cooper and McIntosh [1999] using register renaming (Baker and Manber [1998] discuss a similar approach), and by Zastre [1993] using parameterized procedural abstractions. The main idea is to rewrite instructions so that instead of using hard-coded register names, the (register) operands of an instruction are expressed, if possible, in terms of a previous reference (within the same basic block) to that register. Further, branch instructions are rewritten, where possible, in PC-relative form. These transformations allow the suffix tree construction to detect the repetition of similar but not lexically identical instruction sequences. Cooper and McIntosh obtain a code size reduction of about 5% on the average using these techniques on classically optimized code (in their implementation, classical optimizations achieve a code size reduction of about 18% compared to unoptimized code). These approaches nevertheless suffer from two weaknesses. The first is that by focusing solely on eliminating repeated instruction sequences, they ignore other, potentially more profitable, sources of code size reduction. The second is that any approach that treats a program as a simple linear sequence of instructions, as in the suffix-tree-based approaches mentioned above, will suffer from the disadvantage of having to work with a particular ordering of instructions. The problem is that two "equivalent" computations may map to different instruction sequences in different parts of a program, due to differences in register usage and branch labels, instruction scheduling, and profile-directed code layout to improve instruction cache utilization [Pettis and Hansen 1990].

This article describes a somewhat different approach to code compaction, based on a "whole-system" approach to the problem. Its main contribution is to show that by using aggressive interprocedural optimization together with procedural abstraction of repeated code fragments, it is possible to obtain significantly greater reductions in code size than have been achieved to date. For the identification and abstraction of repeated code fragments, moreover, it shows how "equivalent" code fragments can be detected and factored out without having to resort to purely linear treatments of code sequences as in suffix-tree-based approaches. Thus, instead of treating a program as a simple linear sequence of instructions, we work with its (interprocedural) control flow graph. Instead of using a suffix tree construction to identify repeated instruction sequences, we use a fingerprinting scheme to identify "similar" basic blocks. This sets up a framework for code compaction that can be more flexible in its treatment of what code fragments are considered "equivalent." We use the notions of dominators and postdominators to detect identical subgraphs of the control flow graph, larger than a single basic block, that can be abstracted out into a procedure. Finally, we identify and take advantage of architecture-specific code idioms, e.g., for saving and restoring specific sets of registers at the entry to and return from functions. Among the benefits of such an approach is that it simplifies the development of code compaction systems by using information already available in most compilers, such as the control flow graph and dominator/postdominator trees, thereby making it unnecessary to resort to extraneous structures such as suffix trees.

Our ideas have been implemented in the form of a binary-rewriting tool based on *alto*, a post-link-time code optimizer [Muth et al. 1998]. The resulting system, called *squeeze*, is able to achieve significantly better compaction than previous approaches, reducing the size of classically optimized code by about 30%. Our ideas can be incorporated fairly easily into compilers capable of interprocedural code transformations. The code size reductions we achieve come from two sources: aggressive interprocedural application of classical compiler analyses and optimizations; and *code factoring*, which refers to a variety of techniques to identify and "factor out" repeated instruction sequences. Section 2 discusses those classical optimizations, and their supporting analyses, that are useful for reducing code size.

This is followed, in Section 3, by a discussion of the code factoring techniques used within *squeeze*. In Section 4, we discuss interactions between classical optimizations and factoring transformations. Section 5 contains our experimental results.

A prototype of our system is available at `www.cs.arizona.edu/alto/squeeze`.

2. CLASSICAL ANALYSES AND OPTIMIZATIONS FOR CODE COMPACTION

In the context of code compaction via binary rewriting, it makes little sense to allow the compiler to inflate the size of the program, via transformations such as procedure inlining or loop unrolling, or to keep obviously unnecessary code by failing to perform, for example, common-subexpression elimination and register allocation. We assume therefore that before code compaction is carried out at link time, the compiler has already been invoked with the appropriate options to generate reasonably compact code. Nevertheless, many opportunities exist for link-time code transformations to reduce program size. This section discusses classical program analyses and optimizations that are most useful for code size reduction. In general, the optimizations implemented within *squeeze* have been engineered so as to avoid increases in code size. For example, procedure inlining is limited to those procedures that have a single call site, and no alignment `no-ops` are inserted during instruction scheduling and instruction cache optimization.

2.1 Optimizations for Code Compaction

Classical optimizations that are effective in reducing code size include the elimination of redundant, unreachable, and dead code, as well as certain kinds of strength reduction.

2.1.1 *Redundant-Code Elimination.* A computation in a program is redundant at a program point if it has been computed previously and its result is guaranteed to be available at that point. If such computations can be identified, they can obviously be eliminated without affecting the behavior of the program.

A large portion of code size reductions at link time in *squeeze* comes from the application of this optimization to computations of a hardware register called the *global pointer* (`gp`) register which points to a collection of 64-bit constants called a *global address table*. The Alpha processor, on which *squeeze* is implemented, is a 64-bit architecture with 32-bit instructions. When a 64-bit constant must be loaded into a register, the appropriate global address table is accessed via the `gp` register, together with a 16-bit displacement.[1] Accessing a global object, i.e., loading from or storing to a global variable, or jumping to a procedure, therefore involves two steps: loading the address of the object from the global address table, and then accessing the object via the loaded address. Each procedure in an executable program has an associated global address table, though different procedures may share the same table. Since different procedures—which are generally compiled

[1]On a typical 32-bit architecture, with 32-bit instruction words and 32-bit registers, a (32-bit) constant is loaded into a register via two instructions, one to load the high 16 bits of the register and one for the low 16 bits; in each of these instructions, the 16 bits to be loaded are encoded as part of the instruction word. However, since the Alpha has 32-bit instructions but 64-bit registers, this mechanism is not adequate for loading a 64-bit constant (e.g., the address of a procedure or a global variable) into a register.

independently—may need different global pointer values, the value of the gp register is computed whenever a function is entered, as well as whenever control returns after a call to another function. At link time, it is possible to determine whether a set of functions has the same gp value, and therefore whether the recomputation of gp is necessary. It turns out that most functions in a program are able to use the same value of gp, making the recomputation of gp redundant in most cases. Each such computation of gp involves just one or two register operations, with no significant latency. On a superscalar processor such as the Alpha, the corresponding instructions can generally be issued simultaneously with those for other computations, and hence do not incur a significant performance penalty. Because of this, the elimination of gp computations generally does not lead to any significant improvements in speed. However, because there are so many recomputations of gp in a program, the elimination of redundant gp computations can yield significant reductions in size.

2.1.2 *Unreachable-Code Elimination.* A code fragment is unreachable if there is no control flow path to it from the rest of the program. Code that is unreachable can never be executed, and can therefore be eliminated without affecting the behavior of the program.

At link time, unreachable code arises primarily from the propagation of information across procedure boundaries. In particular, the propagation of the values of actual parameters in a function call into the body of the called function can make it possible to statically resolve the outcomes of conditional branches in the callee. Thus, if we find, as a result of interprocedural constant propagation, that a conditional branch within a function will always be taken, and there is no other control flow path to the code in the branch that is not taken, then the latter code becomes unreachable and can be eliminated.

Unreachable code analysis involves a straightforward depth-first traversal of the control flow graph, and is performed as soon as the control flow graph of the program has been computed. Initially, all basic blocks are marked as unreachable, except for the entry block for the whole program, and a dummy block called $\mathsf{B}_{unknown}$, which has an edge to each basic block whose predecessors are not all known (see Section 2.2.1). The analysis then traverses the interprocedural control flow graph and identifies reachable blocks: a basic block is marked reachable if it can be reached from another block that is reachable. Function calls and the corresponding return blocks are handled in a context-sensitive manner: the basic block that follows a function call is marked reachable only if the corresponding call site is reachable.

2.1.3 *Dead-Code Elimination.* Dead code refers to computations whose results are never used. The notion of "results not used" must be considered broadly. For example, if it is possible for a computation to generate exceptions or raise signals whose handling can affect the behavior of the rest of the program, then we cannot consider that computation to be dead. Code that is dead can be eliminated without affecting the behavior of the program.

Link-time opportunities for dead-code elimination arise primarily as a result of unreachable-code elimination that transforms partially dead computations (computations whose results are used along some execution paths from a program point but not others) into fully dead ones.

2.1.4 *Strength Reduction.* Strength reduction refers to the replacement of a sequence of instructions by an equivalent but cheaper (typically, faster) sequence. In general, the cheaper instruction sequence may not be shorter than the original sequence (e.g., multiplication or division operations where one of the operands is a known constant can be replaced by a cheaper but longer sequence of bit-manipulation operations such as shifts and adds). The benefits for code compaction come from situations where the replacement sequence happens to be shorter than the original sequence.

In *squeeze*, code size improvements from strength reduction come primarily from its application to function calls. Like many processors, the Alpha has two different function call instructions: the `bsr` ("branch subroutine") instruction, which uses PC-relative addressing and is able to access targets within a fixed displacement of the current location; and the `jsr` ("jump subroutine") instruction, which branches indirectly through a register and can target any address. The compiler typically processes programs a function at a time and generates code for function calls without knowledge of how far away in memory the callee is. Because of this, function calls are translated to `jsr` instructions. This, in turn, requires that the 64-bit address of the callee be loaded into a register prior to the `jsr`. As discussed in Section 2.1.1, this is done by loading the address of the callee from a global address table. The code generated for a function call consists therefore of a load instruction followed by a `jsr` instruction. If this can be strength-reduced to a `bsr` instruction, we obtain a savings in code size as well as an improvement in execution speed.

2.2 Program Analyses for Code Compaction

Three program analyses turn out to be of fundamental importance for the transformations discussed above, and are discussed in this section.

2.2.1 *Control Flow Analysis.* Control flow analysis is essential for all of the optimizations discussed in Section 2.1. It is necessary for redundant-code elimination, since, in order to identify a computation as redundant at a program point, we have to verify that it has been computed along every execution path up to that point. It is necessary for unreachable-code elimination as well as dead-code elimination because the classification of code as unreachable or dead relies fundamentally on knowing the control flow behavior of the program. Finally, the strength reduction transformation for function calls discussed in Section 2.1.4 relies on the knowledge of the targets of such calls.

Traditional compilers generally construct control flow graphs for individual functions, based on some intermediate representation of the program, in a straightforward way [Aho et al. 1985]. Things are somewhat more complex at link time because machine code is harder to decompile. In *squeeze*, we construct the interprocedural control flow graph for a program as follows:

(1) The start address of the program appears at a fixed location within the header of the file (this location may be different for different file formats). Using this as a starting point, we use the "standard" algorithm [Aho et al. 1985] to identify leaders and basic blocks, as well as function entry blocks. We use the relocation information of the executable to identify additional leaders, such as jump table targets, which might otherwise not be detected, and we mark these basic blocks

as relocatable. At this stage, we make two assumptions: (1) that each function has a single entry block and (2) that all of the basic blocks of a function are laid out contiguously. If the first assumption turns out to be incorrect, we "repair" the flow graph at a later stage. If the second assumption does not hold, the constructed control flow graph may contain (safe) imprecisions which may cause less effective (size) optimizations.

(2) We add edges to the flow graph. If the exact target of a control transfer instruction cannot be resolved, we assume that the transfer is to a special block $\mathsf{B}_{unknown}$ (in the case of indirect jumps) or function $\mathsf{F}_{unknown}$ (in the case of indirect function calls). We conservatively assume that $\mathsf{B}_{unknown}$ and $\mathsf{F}_{unknown}$ define and use all registers, etc. Any basic block whose start address is marked as relocatable may be the target of any unresolved indirect jump. Thus, we add an edge from $\mathsf{B}_{unknown}$ to each such block. Any function whose entry point is marked as relocatable may be the target of any unresolved indirect function call. Thus, we add a call edge to it from $\mathsf{F}_{unknown}$. (This is safe, but overly conservative. We discuss, below, how this can be improved.)

(3) We carry out interprocedural constant propagation on the resulting control flow graph, as described in Section 2.2.2. We use the results to determine addresses that are loaded into registers. This information is used, in turn, to resolve the targets of indirect jumps and function calls. If we can resolve such targets unambiguously, we replace the edge to $\mathsf{F}_{unknown}$ or $\mathsf{B}_{unknown}$ by an edge to the appropriate target.

(4) Thus far, we have assumed that a function call returns to its caller at the instruction immediately after the call instruction. At the level of executable code, this assumption can be violated in two ways.[2] The first involves *escaping branches*—ordinary (i.e., non-function-call) jumps from one function into another—that arise either due to tail call optimization, or because of code sharing in hand-written assembly code (such as is found in, for example, some numerical libraries). The second involves nonlocal control transfers via functions such as `setjmp` and `longjmp`. Both these cases are handled by the insertion of additional control flow edges, which we call *compensation edges*, into the control flow graph. In the former case, escaping branches from a function f to a function g result in a single compensation edge from the exit node of g to the exit node of f. In the latter case, a function containing a `setjmp` has an edge from $\mathsf{F}_{unknown}$ to its exit node, while a function containing a `longjmp` has a compensation edge from its exit node to $\mathsf{F}_{unknown}$. The effect of these compensation edges is to force the various dataflow analyses to approximate safely the control flow effects of these constructs.

(5) Finally, *squeeze* attempts to resolve indirect jumps through jump tables, which arise from `case` or `switch` statements. The essential idea is to use constant propagation to identify the start address of the jump table, and the bounds

[2]In some architectures, the callee may explicitly manipulate the return address under some circumstances. For example, the SPARC calling convention allows an extra word to follow a call instruction. In such a case, the callee increments the return address to skip over this word. (We are grateful to an anonymous referee for pointing this out to us.) Such situations do not arise in the Alpha architecture, and are not handled by *squeeze*.

check instruction(s) to determine the extent of the jump table. The edge from the indirect jump to $\mathsf{B}_{unknown}$ is then replaced by a set of edges, one for each entry in the jump table. If all of the indirect jumps within a function can be resolved in this way, any remaining edges from $\mathsf{B}_{unknown}$ to basic blocks within that function are deleted.

Potentially, any procedure whose entry-point address is stored in a data section can have this (relocatable) address used somewhere in the program as the target of an indirect function call. Because of this, as mentioned in step (2) above, such procedures must be assumed to be reachable via indirect calls as long as the program contains any call whose target is unknown. While this is safe, it is overly conservative. As discussed in Section 2.1.4, the code generated by the compiler for a function call typically consists of a load from a global address table followed by an indirect call. (A compiler can, in principle, optimize this to a direct call when the caller and callee are within the same module, but such a scheme is still necessary for inter-module calls.) This means that any procedure that is accessible from outside its own module has its relocatable address stored in the global address table (which is in a data section) and hence will be considered to be called from $\mathsf{F}_{unknown}$. As an indication of how conservative this simple technique is, we note that for the programs in the SPECint-95 benchmark suite, about 65% of all functions, on the average, are considered to be called from $\mathsf{F}_{unknown}$.

Alpha executables contain function relocation information that we use to improve the precision of our control flow analysis. The compiler uses special relocation entries, referred to as *literal relocations*, to tag every instruction that loads a function address from a global address table, and every instruction that uses this loaded address. (These relocation entries play a purely informational role, in that they can be ignored by the linker without affecting program behavior.) If every load of a function's address is used simply to jump to that address, we remove the edge from $\mathsf{F}_{unknown}$ to the function, and replace it with call edges from the basic blocks that contain the jump instructions. If a load of a function address is not followed by a jump, the address may be stored and, thus, may equal any unresolved target. In this case, we preserve the edge from $\mathsf{F}_{unknown}$ to the function. For the SPECint-95 benchmarks, this results in fewer than 14% of the procedures having a call from $\mathsf{F}_{unknown}$. The resulting improvement in control flow information has a very significant effect on the amount of code that can be eliminated as unreachable, and leads to a significant improvement in the amount of code compaction that can be realized.

2.2.2 *Interprocedural Constant Propagation.* As mentioned above, we assume that standard compiler analyses and optimizations—including constant propagation—have already been carried out prior to link-time code compaction. Where do opportunities for link-time constant propagation then arise? It turns out, not surprisingly, that constant values that are propagated at compile time are those that are present in source-level compilation units, while those propagated at link time are either values that are not available at compile time, e.g., addresses of global names, or those that the compiler is unable to propagate across compilation unit boundaries, e.g., from a caller to a callee. Link-time constant propagation opportunities also arise from architecture-specific computations that are not visible at the intermediate code representation level typically used by compilers for most optimizations. An example of this is the computation of the `gp` register on the Alpha processor.

The analysis we use in *squeeze* is essentially standard iterative constant propagation, limited to registers but carried out across the control flow graph of the entire program. This has the effect of communicating information about constant arguments from a calling procedure to the callee. To improve precision, *squeeze* attempts to determine the registers saved on entry to a function and restored at the exit from it. If a register r that is saved and restored by a function in this manner contains a constant c just before the function is called, r is inferred to contain the value c on return from the call.

Constant propagation turns out to be of fundamental importance for the rest of the system, since many control and data flow analyses rely on the knowledge of constant addresses computed in the program. For example, the code generated by the compiler for a function call typically first loads the address of the called function into a register, then uses a `jsr` instruction to jump indirectly through that register. If constant propagation determines that the address being loaded is a fixed value and the callee is not too far away, the indirect function call can be replaced by a direct call using a `bsr` instruction, as discussed in Section 2.1.4. This is not only cheaper, but also vital for improving the precision of the interprocedural control flow graph of the program, since it lets us replace a pair of call/return edges to $\mathsf{F}_{unknown}$ with a pair of such edges to the (known) callee. Another example of the use of constant address information involves the identification of possible targets of indirect jumps through jump tables. Unless this can be done, we must assume that the indirect jump is capable of jumping to any basic block of a function,[3] and this can significantly hamper optimizations. Finally, knowledge of constant addresses is useful for optimizations such as the removal of unnecessary memory references. We find that on the average, link-time constant propagation is able to determine the values of the arguments and results for about 18% of the instructions of a program. (This does not mean that these "evaluated" instructions can all be removed, since very often they represent address computations for indexing into arrays or structures or for calling functions.)

2.2.3 *Interprocedural Register Liveness Analysis.* Code factoring, discussed in Section 3, involves abstracting repeated instruction sequences into procedures. To call such procedures it is necessary to find a register that can be used to hold the return address. *Squeeze* implements a relatively straightforward interprocedural liveness analysis, restricted to registers, to determine which registers are live at any given program point. The analysis is *context-sensitive* in that it maintains information about which return edges correspond to which call sites, and propagates information only along realizable call/return paths. The "standard" dataflow equations for liveness analysis are extended to deal with idiosyncracies of the Alpha instruction set. For example, the `call_pal` instruction, which acts as the interface with the host operating system, has to be handled specially, since the registers that may be used by this instruction are not visible as explicit operands of the instruc-

[3] More precisely, any basic block that is marked as "relocatable," as discussed in Section 2.2.1.

tion. Our implementation currently uses the node $\mathsf{B}_{unknown}$ as the target for such calls. The conditional move instruction also requires special attention, since the destination register must also be considered as a source register.

In order to propagate dataflow information only along realizable call/return paths, *squeeze* computes summary information for each function, and models the effect of function calls using these summaries. Given the site of a call to a function f, consisting of a call node n_c and a return node n_r, the effects of the function call on liveness information are summarized via two pieces of information:

(1) $mayUse[f]$ is the set of registers that may be used by f. A register r may be used by f if there is a realizable path from the entry node of f to a use of r without an intervening definition of r. Hence $mayUse[f]$ describes the set of registers that are live at the entry to f independent of the calling context, and which are therefore necessarily live at the call node n_c.

(2) $byPass[f]$ is the set of registers whose liveness depends on the calling context for f. This consists of those registers r such that, if r is live at n_r, then r is also live at n_c.

The analysis proceeds in three phases. The first two phases compute summary information for functions, i.e., their *mayUse* and *byPass* sets. The third phase then uses this information to do the actual liveness computation.

It turns out that even context-sensitive liveness analyses may be overly conservative if they are not careful in handling register saves and restores at function call boundaries. Consider a function that saves the contents of a register, then restores the register before returning. A register r that is saved in this manner will appear as an operand of a `store` instruction, and therefore appear to be used by the function. In the subsequent restore operation, register r will appear as the destination of a `load` instruction, and therefore appear to be defined by the function. A straightforward analysis will infer that r is used by the function before it is defined, and this will cause r to be inferred as live at every call site for f. To handle this problem, *squeeze* attempts to determine, for each function, the set of registers it saves and restores.[4] If the set of callee-saved registers of function f can be determined, we can use it to improve the precision of the analysis by removing this set from $mayUse[f]$ and adding it to $byPass[f]$ whenever those values are updated during the fixpoint computation.

3. CODE FACTORING

Code factoring involves (1) finding a multiply-occurring sequence of instructions, (2) making one representative sequence that can be used in place of all occurrences, and (3) arranging, for each occurrence, that the program executes the representative instead of the occurrence. The third step can be achieved by explicit control transfer (via a call or jump), or by moving the representative of several occurrences to a point that dominates every occurrence. We first exploit the latter form of code factoring, since it involves no added control transfer instructions.

[4]We do not assume that a program will necessarily respect the calling conventions with regard to callee-saved registers, since such conventions are not always respected in libraries containing hand-written assembly code. This approach is safe, though sometimes overly conservative.

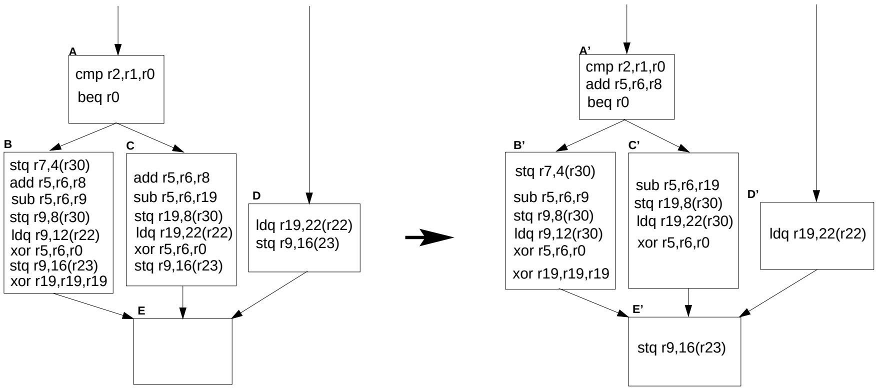


Fig. 1. Local code factoring.

3.1 Local Factoring Transformations

Inspired by an idea of Knoop et al. [1994], we try to merge identical code fragments by moving them to a point that pre- or postdominates all the occurrences of the fragments. We have implemented a local variant of this scheme which we describe using the example depicted in Figure 1. The left hand side of the figure shows an assembly code flowchart with a conditional branch (`beq r0`) in block A. Blocks B and C contain the same instruction `add r5,r6,r8`. Since these instructions do not have backward dependencies with any other instruction in B or C, we can safely move them into block A just before the `beq` instruction, as shown in the right-hand side of Figure 1. Similarly, blocks B, C, and D share the same store instruction `stq r9,r16(r23)`, and since these instructions do not have forward dependencies with any other instruction in B, C, and D, they can be safely moved into block E. In this case, it is not possible to move the store instruction from B and C into A because, due to the lack of aliasing information, there are backward dependencies to the load instructions (`ldq`) in B and C. In general, however, it might be possible to move an instruction either up or down. In this case, we prefer to move it down, since moving it up, over a two-way branch, will eliminate one copy while moving it down to a block that has many predecessors might eliminate several copies.

Our scheme uses register reallocation to make this transformation more effective. For example, the `sub` instructions in B and C write to different registers (`r9` and `r19`). We can, however, rename `r9` to `r19` in B, thereby making the instructions identical. Another opportunity rests with the `xor` instructions in B and C. Even though they are identical, we cannot move them into A because they write register `r0` which is used by the conditional branch. Reallocating `r0` in A to another register which is dead at the end of A will make the transformation possible.

3.2 Procedural Abstraction

Given a single-entry, single-exit code fragment C, procedural abstraction of C involves (1) creating a procedure f_C whose body is a copy of C and (2) replacing the appropriate occurrences of C in the program text by a function call to f_C. While the first step is not very difficult, the second step, at the level of assembly or machine code, involves a little work.

In order to create a function call using some form of "jump-and-link" instruction that transfers control to the callee and at the same time puts the return address into a register, it is necessary to find a free register for that purpose. A simple method is to calculate, for each register r, the number of occurrences of code fragment C that could use r as a return register. A register with the highest such figure of merit is chosen as the return register for f_C. If a single instance of f_C, using a particular return register, is not enough to abstract out all of the occurrences of C in the program, we may create multiple instances of f_C that use different return registers. We use a more complicated scheme when abstracting function prologs (see Section 3.5.1) and regions of multiple basic blocks (see Section 3.4).

3.3 Procedural Abstraction for Individual Basic Blocks

Central to our approach is the ability to apply procedural abstraction to individual basic blocks. In this section, we discuss how candidate basic blocks for procedural abstraction are identified.

3.3.1 *Fingerprinting.* To reduce the cost of comparing basic blocks to determine whether they are identical (or similar), we use a *fingerprint function* to compute a fingerprint for each basic block, such that two blocks with different fingerprints are guaranteed to be different. In general, such fingerprint functions are defined with respect to the notion of "equality" between basic blocks. For example, in our current implementation, two blocks are considered to be equal if the instruction sequences in them are the same. Thus, the fingerprint function of a block is based on the sequence of instructions in the block. On the other hand, if a code compaction scheme defines equality of basic blocks with respect to definition-use chains then a fingerprint based on the number of occurrences of each type of opcode may be used.

In our current implementation, a fingerprint is a 64-bit value formed by concatenating 4-bit encodings of the opcodes of the first 16 instructions in the block. Since most "systems" applications tend to have short basic blocks, characterizing the first 16 instructions seems enough for most basic blocks. This means that two blocks that are different, but which have the same sequence of opcodes for their first 16 instructions, will have the same fingerprint: we will discover them to be different later, when we actually compare them instruction by instruction.

With 4 bits per instruction, we can encode 15 different opcodes and reserve one code for "other." We decide which 15 will be explicitly represented by considering a static instruction count of the program. The 15 most frequently occurring opcodes are given distinct 4-bit patterns. The remaining pattern, `0000`, represents opcodes that are not in the top 15 in frequency.

To reduce the number of pairwise comparisons of fingerprints that must be carried out, we use a hashing scheme such that basic blocks in different hash buckets are guaranteed to have different fingerprints, and so need not be compared.

3.3.2 *Register Renaming within Basic Blocks.* When we find two basic blocks that are "similar," i.e., have the same fingerprint and the same number of instructions, but which are not identical, we attempt to rename the registers in one of them so as to make the two identical. The basic idea is very simple: we rename

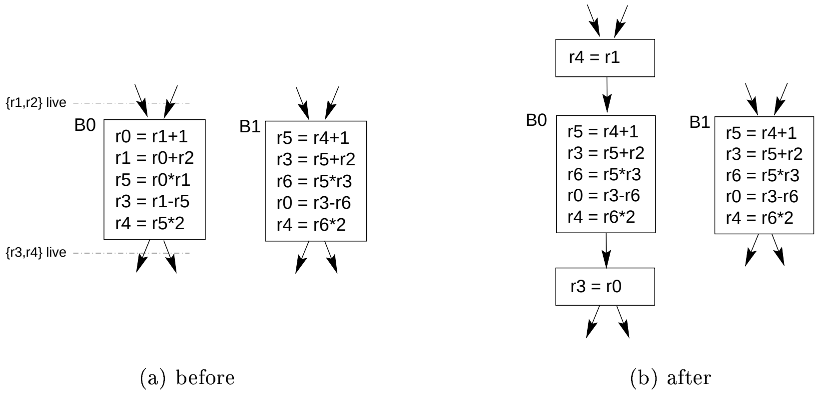


Fig. 2. Example of basic-block-level register renaming.

registers "locally," i.e., within the basic block; and if necessary, we insert register-to-register moves, in new basic blocks inserted immediately before and after the block being renamed, so as to preserve program behavior. An example of this is shown in Figure 2, where block B0 is renamed to be the same as block B1.

For soundness, we have to ensure that the renaming does not alter any use-definition relationships. We do this by keeping track of the set of registers that are live at each point in the basic block, as well as the set of registers that have already been subjected to renaming. These sets are then used to detect and disallow renamings that could alter the program's behavior. The pseudocode for our renaming algorithm is given in Appendix A.

The renaming algorithm keeps track of the number of explicit register-to-register moves that have to be inserted before and after a basic block that is being renamed. The renaming is undone if, at the end of the renaming process, the cost of renaming, i.e., the number of register moves required together with a function call instruction, exceeds the savings from the renaming, i.e., the number of instructions in the block.

Cooper and McIntosh [1999] describe a different approach to register renaming. They carry out register renaming at the level of entire live ranges. That is, when renaming a register r_0 to a different register r_1, the renaming is applied to an entire live range for r_0. This has the advantage of not requiring additional register moves before and after a renamed block, as our approach does. However, it has the problem that register renaming to allow the abstraction of a particular pair of basic blocks may interfere with the abstraction of a different pair of blocks. This is illustrated in Figure 3, where solid double arrows indicate identical basic blocks, while dashed double arrows indicate blocks that are not identical but which can be made identical via register renaming. Blocks B0, B1, and B2 comprise a live range for register `r0`, while B3 and B5 comprise a live range for `r1`. We can rename `r0` to `r5` in this live range, so as to make blocks B1 and B3 identical, but this will cause blocks B2 and B4 to not be identical and therefore not abstractable into a function. We can also rename `r5` to `r0` in block B3 so as to make it identical to B1, but this will interfere with the abstraction of blocks B5 and B6. Because of such interference effects, it is not clear whether live-range-level renaming produces results that are necessarily superior to basic-block-level renaming. Notice that the

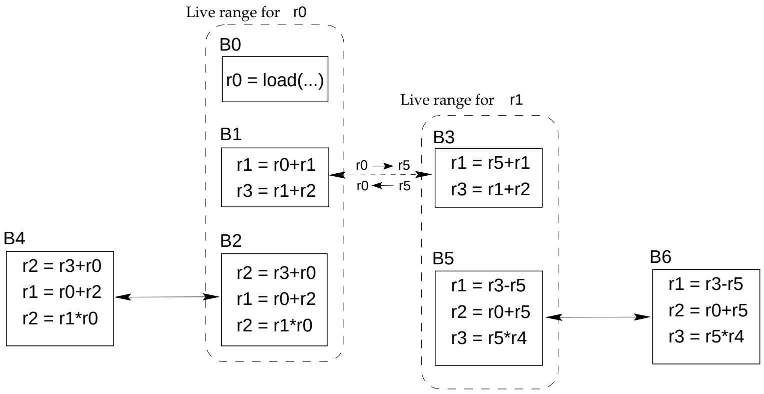


Fig. 3. Interference effects in live-range-level register renaming.

problem could be addressed by judiciously splitting the live ranges. Indeed, the local renaming we use can be seen as the limiting case of live-range-level renaming if splitting is applied until no live range spans more than one basic block.

3.3.3 *Control Flow Separation.* The approach described above will typically not be able to abstract two basic blocks that are identical except for an explicit control transfer instruction at the end. The reason for this is that if the control transfers are to different targets, the blocks will be considered to be different and so will not be abstracted. Moreover, if the control transfer instruction is a conditional branch, procedural abstraction becomes complicated by the fact that two possible return addresses have to be communicated.

To avoid such problems, basic blocks that end in an explicit control transfer instruction are split into two blocks: one block containing all the instructions in the block except for the control transfer, and another block that contains only the control transfer instruction. The first of this pair of blocks can then be subjected to renaming and/or procedural abstraction in the usual way.

The next section describes how code fragments larger than a single basic block can be subjected to procedural abstraction.

3.4 Single-Entry/Single-Exit Regions

The discussion thus far has focused on the procedural abstraction of individual basic blocks. In general, however, we may be able to find multiple occurrences of a code fragment consisting of more than one basic block. In order to apply procedural abstraction to such a region R, at every occurrence of R in the program, we must be able to identify a single program point at which control enters R, and a single program point at which control leaves R. It isn't hard to see that any set of basic blocks R with a single entry point and a single exit point corresponds to a pair of points (d, p) such that d dominates every block in R and p postdominates every block in R. Conversely, a pair of program points (d, p), where d dominates p and p postdominates d, uniquely identifies a set of basic blocks with a single entry point and single exit point. Two such single-entry, single-exit regions R and R' are considered to be identical if it is possible to set up a 1-1 correspondence $\simeq$ between their members such that $B_1 \simeq B_1'$ if and only if (1) B_1 is identical to B_1', and (2) if B_2 is a (immediate) successor of B_1 under some condition C, and B_2' is a (immediate) successor of B_1' under the same condition C, then $B_2 \simeq B_2'$. The algorithm to determine whether two regions are identical works by recursively traversing the two regions, starting at the entry node, and verifying that corresponding blocks are identical.

In *squeeze*, after we apply procedural abstraction to individual basic blocks, we identify pairs of basic blocks (d, p) such that d dominates p and p postdominates d. Each such pair defines a single-entry, single-exit set of basic blocks. We then partition these sets of basic blocks into groups of identical regions, which then become candidates for further procedural abstraction.

As in the case of basic blocks, we compute a fingerprint for each region so that regions with different fingerprints will necessarily be different. These fingerprints are, again, 64-bit values. There are 8 bits for the number of basic blocks in the region and 8 bits for the total number of instructions, with the bit pattern `11...1` being used to represent values larger than 256. The remaining 48 bits are used to encode the first (according to a particular preorder traversal of the region) 8 basic blocks in the region, with each block encoded using 6 bits: two bits for the type of the block,[5] and four bits for the number of instructions in the block. Again, as in the case of basic blocks, the number of pairwise comparisons of fingerprints is reduced by distributing the regions over a hash table.

It turns out that applying procedural abstraction to a set of basic blocks is not as straightforward as for a single basic block, especially in a binary rewriting implementation such as ours. The reason is that, in general, when the procedure corresponding to such a single-entry, single-exit region is called, the return address will be put into a register whose value cannot be guaranteed to be preserved through that entire procedure, e.g., because the region may contain function calls, or because the region may contain paths along which that register is overwritten. This means that the return address register has to be saved somewhere, e.g., on the stack. However, allocating an extra word on the stack, to hold the return address, can cause problems unless we are careful. Allocating this space at the top of the stack frame can cause changes in the displacements of other variables in the stack frame, relative to the top-of-stack pointer, while allocating it at the bottom of the stack frame can change the displacements of any arguments that have been passed on the stack. If there is any address arithmetic involving the stack pointer, e.g., for address computations for local arrays, such computations may be affected by changes in displacements within the stack frame. These problems are somewhat easier to handle if the procedural abstraction is being carried out before code generation, e.g., at the level of abstract syntax trees [Franz 1997]. At the level of assembly code [Cooper and McIntosh 1999; Fraser et al. 1984] or machine code (as in our work), it becomes considerably more complicated. There are, however, some simple cases where it is possible to avoid the complications associated with having to save and restore the return address when introducing procedural abstractions. Here, we identify two such situations. In both cases, let (d_0, p_0) and (d_1, p_1) define two

[5]In essence, the *type* of a block describes its control flow behavior, i.e., whether it contains a procedure call, a conditional branch, an indirect jump through a jump table, etc.

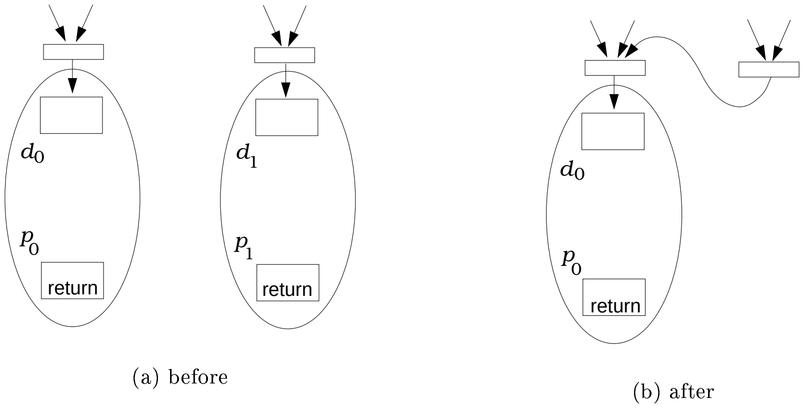


Fig. 4. Merging regions ending in `returns` via cross-jumping.

identical regions.

The first case involves situations where p_0 and p_1 are return blocks, i.e., blocks from which control returns to the caller. In this case there is no need to use procedural abstraction to create a separate function for the two regions. Instead, we can use a transformation known as *cross-jumping* [Muchnick 1997], where the code in the region (d_1, p_1) is simply replaced by a branch to d_0. The transformation is illustrated in Figure 4.

In the second case, suppose that it is possible to find a register r that (1) is not live at entry to either region, and (2) whose value can be guaranteed to be preserved up to the end of the regions (r may be a general-purpose register that is not defined within either region, or a callee-saved register that is already saved and restored by the functions in which the regions occur). In this case, when abstracting these regions into a procedure p, it is not necessary to add any code to explicitly save and restore the return address for p. The instruction to call p can simply put the return address in r, and the return instruction(s) within p can simply jump indirectly through r to return to the caller.

If neither of these conditions is satisfied, *squeeze* tries to determine whether the return address register can be safely saved on the stack at entry to p, and restored at the end. For this, it uses a conservative analysis to determine whether a function may have arguments passed on the stack, and which, if any, registers may be pointers into the stack frame. Given a set of candidate regions to be abstracted into a representative procedure, it checks the following:

(1) for each function that contains a candidate region, it must be safe, with respect to the problems mentioned above, to allocate a word on the stack frame of the function;

(2) a register r_0 must be free at entry to each of the regions under consideration;

(3) a register r_1 must be free at the end of each of the regions under consideration; and

(4) there should not be any calls to `setjmp()`-like functions that can be affected by a change in the structure of the stack frame.

If these conditions are satisfied then, on entry, p allocates an additional word on the stack and saves the return address (passed via r_0) into this location; and, on exit, loads the return address from this location (using r_1) and restores the stack frame. The current implementation of the safety check described above is quite conservative in its treatment of function calls within a region. In principle, if we find that space can be allocated on the stack but have no free registers for the return address at entry or exit from the abstracted function, it should be possible to allocate an extra word on the stack in order to free up a register, but we have not implemented this.

3.5 Architecture-Specific Idioms

Apart from the general-purpose techniques described earlier for detecting and abstracting out repeated code fragments, there are machine-specific idioms that can be profitably exploited. In particular, the instructions to save and restore registers (the return address and callee-saved registers) in the prolog and epilog of each function generally have a predictable structure and are saved at predictable locations within the stack frame. For example, the standard calling convention for the Compaq Alpha AXP architecture under Tru64 Unix[6] treats register `r26` as the return address register (`ra`) and registers `r9` through `r15` as callee-saved registers. These are saved at locations `0x0(sp)`, `0x8(sp)`, `0x10(sp)`, and so on. Abstracting out such instructions can yield considerable savings in code size. Such architecture-specific save/restore sequences are recognized and handled specially by *squeeze*, for two reasons: first, these instructions often do not form a contiguous sequence in the code stream; and second, handling them specially allows us to abstract them out of basic blocks that may not be identical to each other.

3.5.1 *Abstracting Register Saves.* In order to abstract out the register save instructions in the prolog of a function f into a separate function g, it is necessary to identify a register that can be used to hold the return address for the call from f to g. For each register r, we first compute the savings that would be obtained if r were to be used for the return address for such calls. This is done by totaling up, for each function f where r is free at entry to f, the number of registers saved in f's prolog. We then choose a register r with maximum savings (which must exceed 0), and generate a family of functions $Save^r_{15}, \ldots, Save^r_9, Save^r_{ra}$ that save the callee-saved registers and the return address register, and then return via register r. The idea is that function $Save^r_i$ saves register i and then falls through to function $Save^r_{i-1}$.

As an example, suppose we have two functions `f0()` and `f1()`, such that `f0()` saves registers `r9`, …, `r14`, and `f1()` saves only register `r9`. Assume that register `r0` is free at entry to both these functions and is chosen as the return address register. The code resulting from the transformation described above is shown in Figure 5.

It may turn out that the functions subjected to this transformation do not use all of the callee-saved registers. For example, in Figure 5, suppose that none of the functions using return address register `r0` save register `r15`. In this case, the code for the function $Save^0_{15}$ becomes unreachable and is subsequently eliminated.

[6]Tru64 Unix was formerly known as Digital Unix.

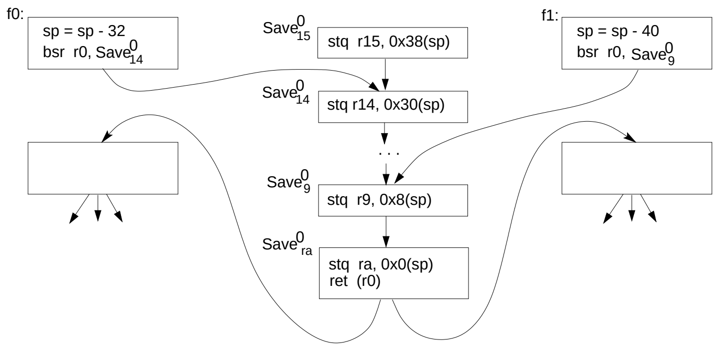


Fig. 5. Example code from abstraction of register save actions from function prologs.

A particular choice of return address register, as described above, may not account for all of the functions in a program. The process is therefore repeated, using other choices of return address registers, until either no further benefit can be obtained, or all functions are accounted for.

3.5.2 *Abstracting Register Restores.* The code for abstracting out register restore sequences in function epilogs is conceptually analogous to that described above, but with a few differences. If we were simply to do the opposite of what was done for register saves in function prologs, the code resulting from procedural abstraction at each return block for a function might have the following structure, with three instructions to manage the control transfers and stack pointer update:

```
...
bsr r1, Restore   /* call function that restores registers */
sp = sp + k       /* deallocate stack frame */
ret (ra)          /* return */
```

If we could somehow move the instruction for deallocating the stack frame into the function that restores saved registers, there would be no need to return to the function f whose epilog we are abstracting: control could return directly to f's caller (in effect realizing tail call optimization). The problem is that the code to restore saved registers is used by many different functions, which in general have stack frames of different sizes, and hence need to adjust the stack pointer by different amounts. The solution to this problem is to pass, as an argument to the function that restores registers, the amount by which the stack pointer must be adjusted. Since the return address register `ra` is guaranteed to be free at this point—it is about to be overwritten with f's return address prior to returning control to f's caller—it can be used to pass this argument.[7] Since there is now no need for control to return to f after the registers have been restored—it can return directly to f's caller—we can simply jump from function f to the function that restores registers, instead of using a function call. The resulting code requires two instructions instead of three in each function return block:

[7]In practice not all functions can be guaranteed to follow the standard calling convention, so it is necessary to verify that register `ra` is, in fact, being used as the return address register by f.

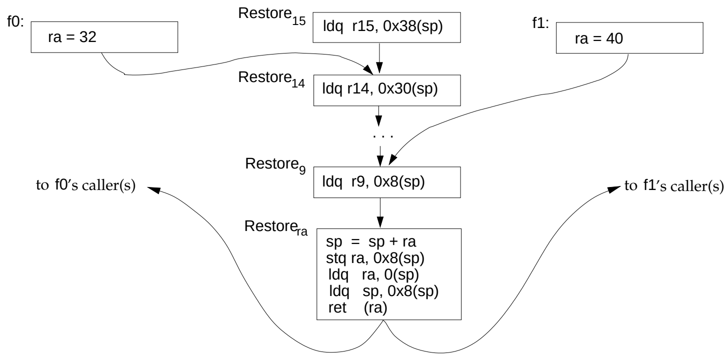


Fig. 6. Example code from abstraction of register restore actions from function epilogs.

```
ra = k              /* sp needs to be adjusted by k */
br Restore          /* jump to function that restores registers */
```

The code in the function that restores registers is pretty much what one would expect. Unlike the situation for register save sequences discussed in Section 3.5.1, we need only one function for restoring registers. The reason for this is that there is no need to call this function: control can jump into it directly, as discussed above. This means that we do not have to generate different versions of the function with different return address registers. The overall structure of the code is analogous to that for saving registers: there is a chain of basic blocks, each of which restores a callee-saved register, with control falling through into the next block, which saves the next (lower-numbered) callee-saved register, and so on. The last member of this chain adjusts the stack pointer appropriately, loads the return address into a register, and returns. There is, however, one minor twist at the end. The amount by which the stack pointer must be adjusted is passed in register `ra`, so this register cannot be overwritten until after it has been used to adjust the stack pointer. On the other hand, since the memory location from which f's memory address is to be restored is in f's stack frame, we cannot adjust the stack pointer until after the return address has been loaded into `ra`. At first glance, it seems that the problem can be addressed using something like the following instruction sequence:

```
sp = sp + ra       /* sp = sp + ra ≡ new sp */
ra = sp - ra       /* ra = sp - ra ≡ old sp */
ra = load 0(ra)    /* ra = return address */
ret (ra)
```

This code is incorrect, however, because the stack pointer is updated—i.e., the stack frame is deallocated—before the return address is loaded from the stack frame. As a result, if an interrupt occurs between the end of the first instruction and the beginning of the third instruction, the return address may be overwritten, resulting in incorrect behavior. To avoid this, we have to ensure that the stack pointer update is the last instruction before the `ret` instruction. We do this by first computing the new value of the stack pointer and storing it in the stack frame (in the slot where the first callee-saved register, was originally stored), then updating the return address register, and finally loading the new value of the stack pointer from memory:[8]

```
ra = sp + ra          /* ra = sp + ra ≡ new sp */
8(sp) = store ra      /* new sp saved at location 8(sp) */
ra = load 0(sp)       /* ra = return address */
sp = load 8(sp)       /* sp = new sp */
ret (ra)
```

The resulting code for restoring saved registers, for the functions considered in the example illustrated in Figure 5, is shown in Figure 6.

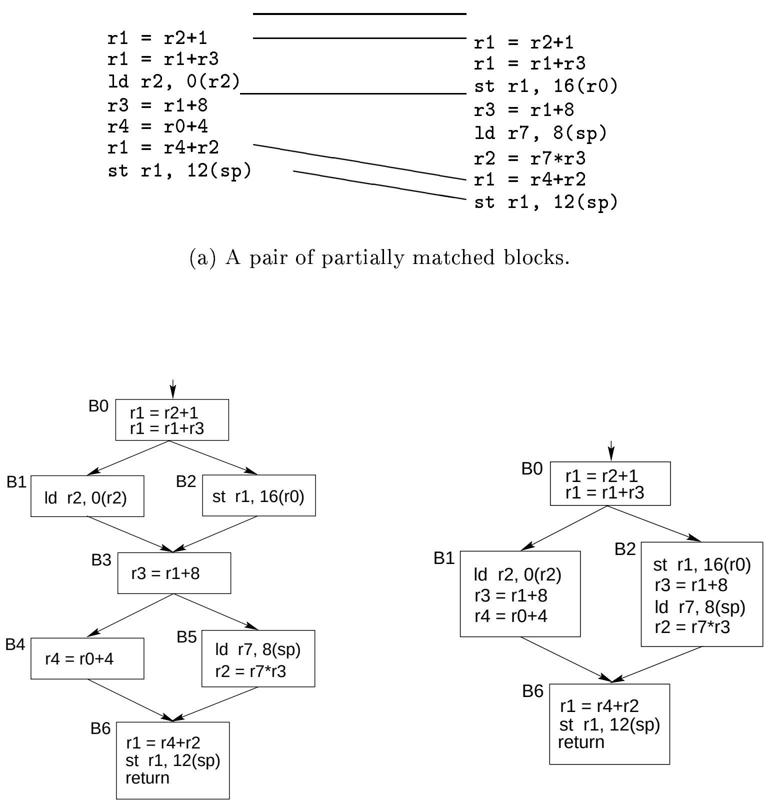
We go through these contortions in order to minimize the number of registers used. If we could find another register that is free at the end of every function, we could load the return address into this register, resulting in somewhat simpler code. However, in general it is not easy to find a register that is free at the end of every function. The reason we go to such lengths to eliminate a single instruction from each return block is that there are a lot of return blocks in the input programs, typically amounting to about 3%–7% of the basic blocks in a program, excluding return blocks for leaf routines that do not allocate/deallocate a stack frame (there is usually at least one—and, very often, more than one—such block for each function). The elimination of one instruction from each such block translates to a code size reduction of about 1%–2% overall. (This may seem small, but to put it in perspective, consider that Cooper and McIntosh report an overall code size reduction of about 5% using suffix-tree-based techniques.)

3.6 Abstracting Partially Matched Blocks

As discussed in the preceding sections, the smallest code unit considered for procedural abstraction by *squeeze* is the basic block. In other words, *squeeze* will not attempt to carry out any form of procedural abstraction on two blocks that are not the same, even though there may be a significant amount of "partial match" between them, i.e., the blocks may share common subsequences of instructions. This is illustrated by the pair of basic blocks shown in Figure 7(a), with matched instructions indicated by lines drawn between them. Our experiments, described in this section, indicate that abstraction of partially matched blocks is computationally quite expensive but adds very little additional savings in code size. For this reason we have chosen not to include partial matching within *squeeze*.

There are two issues that have to be addressed when considering procedural abstraction of partially matched blocks: first, how to identify partially matched blocks to abstract; and second, how to transform the code to effect this abstraction. In our experiments, abstraction of partially matched blocks was carried out after procedural abstraction of "fully matched" blocks, discussed in Section 3.3. In general, a particular basic block B_0 may be partially matched against many different blocks, which may match different subsequences of its instructions. The savings obtained from procedural abstraction in this case depends on the block B_1 that is chosen as a match. Once a block B_1 is partially matched with B_0 and subjected to procedural abstraction, B_1 is not available for partial matching against other basic blocks. This

[8] We are indebted to Anders Lindgren for pointing out the problem in our original code, as well as suggesting the solution shown.



(a) A pair of partially matched blocks.

(b) Procedure obtained from the maximal matching

(c) Procedure obtained after unmatching unprofitable instructions

Fig. 7. Procedural abstraction of partially matched blocks.

means that even though, from B_0's perspective, B_1 may yield the largest savings when procedural abstraction is carried out, this may not be the best choice globally, since we may have obtained greater savings by matching B_1 with some other block. The problem of computing a globally optimal set of partial matches for a set of basic blocks, i.e., one that maximizes the savings obtained from their procedural abstraction, is computationally difficult (the related *longest common subsequence* problem is NP-complete [Garey and Johnson 1979]). We therefore take a greedy approach, processing basic blocks in decreasing order of size. When processing a block B_0, we compare it against all other blocks and choose a block B_1 that yields maximal savings (computed as discussed below) when procedural abstraction is carried out based on partial matching of B_0 and B_1: B_1 is then put into a partition associated with B_0. When all blocks have been processed in this manner, all of the blocks in the same partition are abstracted into a single procedure.

The benefit obtained from the procedural abstraction of two partially matched blocks B_0 and B_1 is determined as follows. First, we use dynamic programming to determine the minimum edit distance between the two blocks, and thus the best match between them. Now consider the second issue mentioned above, namely, carrying out the program transformation. Since we have a partial match between these blocks, there will have to be multiple execution paths through the resulting procedure, such that the call from B_0 will take one path while that from B_1 will take another. We can do this by passing an argument to the abstracted procedure indicating, for any call, which call site it originated from, and therefore which instructions it should execute. When scanning down blocks B_0 and B_1, whenever we find a mismatched sequence of instructions in either block, we generate code in the abstracted procedure to test this argument and execute the appropriate instruction sequence based on the outcome. Figure 7(b) shows the control flow graph of the resulting procedure. In addition to the instructions shown, we also have to manage control flow. For this, we need a conditional branch at the end of blocks B0 and B3 (in general, if there are more than two blocks in the partition being abstracted, we may need explicit comparison operations to determine which of a set of alternatives to execute), and an unconditional branch for each of the pairs of blocks {B1, B2} and {B4, B5}, for a total of 15 instructions. Notice that by designating the instruction in block B3 as a "match" between the two original blocks, and thereby having B3 be common to the execution paths for both of the call sites of the procedure, we save a single copy of this instruction, but pay a penalty of two branch instructions for managing control flow around it. In this case, it turns out to be better, when determining the original partial match, to ignore the fact that the two `r3 = r1+8` instructions can be matched. This would yield the code shown in Figure 7(c), with a total of 14 instructions. On the other hand, if instead of the single matched instruction in B3 we had a sequence of, say, 10 matched instructions, the savings incurred from combining them into a single block within the abstracted procedure would outweigh the cost of the additional instructions needed to manage control flow. As this example illustrates, the minimal edit distance between the two blocks does not necessarily yield the greatest savings: sometimes we can do better by ignoring some matches. It is not obvious that the dynamic programming algorithm for computing minimum edit distance can be modified in a straightforward way to accommodate this. Instead we use a postprocessing phase to "unmatch" instructions that incur too great a control flow penalty.

Even with the improvement of unmatching instructions where a match is not deemed profitable, the cost of control flow management significantly lowers the overall benefits of procedural abstraction based on partial matches. In the example shown in Figure 7, for example, at each call site for the resulting procedure we would need two additional instructions—one to set the argument register identifying the call site, another to carry out the control transfer—for an overall total of 18 instructions. By contrast, the two original basic blocks shown in Figure 7(a) contain a total of 15 instructions. Thus, despite the significant partial match between these two blocks, it is not profitable in this case to abstract them out into a procedure. In general, we found that procedural abstraction based on partial matches incurs a large computational cost, but yields overall code size savings of around 0.4–0.6%. We obtained similar results with a number of other variations on this scheme, such as factoring out only common suffixes or prefixes of blocks. Because of the high computational cost of this transformation and the low benefit it produces, we decided not to include it within *squeeze*.

4. INTERACTIONS BETWEEN CLASSICAL OPTIMIZATIONS AND CODE FACTORING

There is considerable evidence that (appropriately controlled) optimization can yield significant reductions in code size. Compiler "folklore" has it that some amount of peephole optimization can speed up the overall compilation process because of the resulting reduction in the number of instructions that have to be processed by later phases.[9] Cooper and McIntosh [1999] observe code size reductions of about 18% due to compiler optimizations, while our own experiments, discussed in Section 5, indicate that enabling optimizations that do not increase code size yield a code size reduction of about 20% on the average.

However, since classical compiler optimizations are aimed primarily at increasing execution speed, the reductions in size they produce are, in many cases, the happy but coincidental outcome of transformations whose primary goal is a reduction in execution time. Examples of transformations that can, in some situations, lead to an increase in code size include machine-independent optimizations such as partial-redundancy elimination, procedure inlining, and shrink wrapping, as well as machine-dependent optimizations such as instruction scheduling and instruction cache optimization, both of which can result in the insertion of `no-ops` for alignment purposes. Even for transformations that lead to code size reductions, using execution speed improvement as the primary goal of optimization can yield smaller size reductions than might be possible otherwise. For example, in the local factoring transformation discussed in Section 3.1, if an instruction can be hoisted either upward or downward, it is preferable to hoist it downward, since this can yield greater size reductions. However, if our primary goal is increasing execution speed, we would prefer to hoist it upward instead, so as to hide latencies.

This discussion does not take into account interactions between classical optimizations, whose primary goal is a reduction in execution time, and code-factoring transformations, whose primary goal is a reduction in code size. As a simple example, consider the code sequences in the following two basic blocks:

Block B_1	*Block B_2*	
`load r1, 8(sp)`	`load r1, 8(sp)`	
`add r1, r2, r3`	`add r1, r2, r3`	
	`load r1, 12(sp)`	(*)
`add r4, r5, r6`	`add r4, r5, r6`	
	`add r1, r4, r1`	(*)
`mul r3, r6, r3`	`mul r3, r6, r3`	
`add r3, r5, r3`	`add r3, r5, r3`	
`store r3, 16(sp)`	`store r3, 16(sp)`	

As presented, these two blocks are different, and cannot be subjected to procedural abstraction into the same procedure. If the compiler determines that the two instructions in block B_2 marked as `(*)` are dead (e.g., due to code-eliminating optimizations elsewhere that cause `r1` to become dead at the end of block B_2), and eliminates them, the two blocks then become identical and can be factored out into

[9] We believe this observation is due to W. A. Wulf.

a procedure. However, if the compiler does an even better job of optimization, and is able to find a free register in block B_1 that allows it to eliminate the load instruction in that block, the two blocks again become different and cannot be abstracted into a procedure. Notice that in the latter case, the compiler's decision to eliminate the load instruction is a locally good decision—it reduces code size by one instruction and is likely to improve speed—but, from the standpoint of code compaction, not such a good decision globally.

Interactions such as these give rise to a phase-ordering problem between size-oriented and speed-oriented transformations. One possible way to deal with this would be to iterate the transformations to a fixpoint. However, this is not a satisfactory solution, because transformations such as code factoring require a lot of code sequence comparisons to identify repeated instruction sequences that can be factored out, and therefore are quite expensive; iterating over them is likely to be so expensive as to be impractical. We currently do not do perform such iteration.

5. EXPERIMENTAL RESULTS

To evaluate our ideas, we used the eight SPEC-95 integer benchmarks, as well as six embedded applications, *adpcm*, *epic*, *gsm*, *mpeg2dec*, *mpeg2enc*, and *rasta*, obtained from the MediaBench benchmark suite from UCLA (`http://www.cs.ucla.edu/~leec/mediabench`). We evaluated *squeeze* on code obtained from two different C compilers: the vendor-supplied C compiler *cc* V5.2-036, invoked as `cc -O1`, and the GNU C compiler *gcc* version 2.7.2.2, at optimization level `-O2`. The programs were compiled with additional flags instructing the linker to retain relocation information and to produce statically linked executables.[10] The optimization level chosen for each compiler was selected to allow "standard" optimizations except for those, such as procedure inlining and loop unrolling, that can increase code size. At optimization level `-O1`, the vendor-supplied compiler *cc* carries out local optimizations and recognition of common subexpressions; global optimizations including code motion, strength reduction, and test replacement; split lifetime analysis; and code scheduling; but not size-increasing optimizations such as inlining; integer multiplication and division expansion using shifts; loop unrolling; and code replication to eliminate branches. Similarly, at the `-O2` level of optimization, the *gcc* compiler carries out most supported optimizations that do not involve a space-speed trade-off. In particular, loop unrolling and function inlining are not carried out.

The baseline for our measurements is code optimized by the compiler as discussed above, but with unreachable code and no-ops removed and profile-guided code layout—which can improve performance significantly, but is not carried out by either of the compilers we used for our experiments—carried out. This eliminates library routines that are not referenced by the program but which get linked into the program because of references to other routines in the library, and excludes size reductions that could be trivially obtained by a traditional compiler. We include profile-directed code layout in the baseline to allow a fair comparison:

[10]The requirement for statically linked executables is a result of the fact that *squeeze* relies on the presence of relocation information for its control flow analysis. The Tru64 Unix linker `ld` refuses to retain relocation information for executables that are not statically linked.

Table I. Code Size Improvements Due To Different Transformations

Transformation	*Savings* (%)
redundant computation elimination	34.14
Basic block and region abstraction	27.42
Useless code elimination	22.43
Register save/restore abstraction	9.95
Other inter-procedural optimizations	6.06

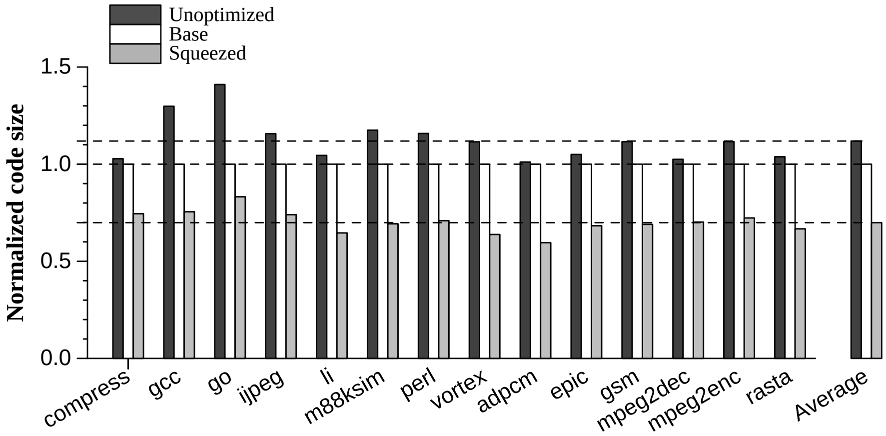
squeeze carries out this optimization, and we do not want the resulting performance improvements to unduly inflate the execution speed of the resulting executables.

To obtain instruction counts, we first disassemble the executable files and discard unreachable code and `no-op` instructions. This eliminates library routines that are linked in but are not actually called, as well as any `no-op` instructions that may have been inserted by the compiler for instruction scheduling or alignment purposes. To identify unreachable code, we construct a control flow graph for the entire program and then carry out a reachability analysis. In the course of constructing the control flow graph, we discard unconditional branches. We reinsert those that are necessary after all the code transformations have been carried out: during code layout, just before the transformed code is written out. To get accurate counts, therefore, we generate the final code layout in each case (i.e., with and without compaction) and count the total number of instructions.

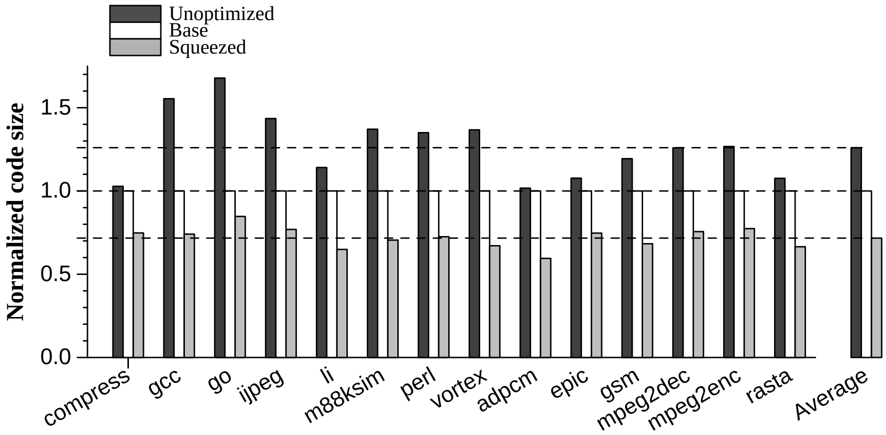
5.1 Code Size

The overall code size reductions achieved using our techniques are summarized in Figure 8. The corresponding raw data are given in Debray et al. [2000]. Figure 8(a) shows the effects of *squeeze* on code compiled using the vendor-supplied C compiler *cc*, while Figure 8(b) shows the effects of *squeeze* on code compiled using the GNU C compiler *gcc*. The columns labeled "Unoptimized" refer to programs compiled at optimization level `-O0`, where no optimization is carried out, and serve as a reference point to indicate how much code size reduction is realized using only optimizations carried out by the compiler, while the columns labeled "Base" refer to code optimized at the appropriate level, as discussed above, with unreachable code and `no-ops` removed. It can be seen from Figure 8 that by using classical compiler optimizations, each of these compilers is able to achieve significant improvements in code size compared to the unoptimized code: *cc* obtains a size reduction of just over 10% on the average, while *gcc* is able to achieve an average size reduction of about 20%. More importantly, however, it can be seen that, even when given the already optimized executables as input, *squeeze* is able to achieve significant further reductions in size. For the *cc*-compiled programs it achieves an average size reduction of just over 30%, while for the *gcc*-compiled programs the average size reduction is a little over 28%. The greatest reduction in size is about 40% for the *adpcm* program, while the smallest is about 15–17% for the *go* program.

Table I gives a breakdown of the average contribution of different kinds of code transformations toward the code size reductions we achieve. Four classes of transformations account for most of these savings. About a third of the savings comes from



(a) Compiler: *cc*



(b) Compiler: *gcc*

Fig. 8. Effects of compaction on code size (normalized).

the elimination of redundant computations of the global pointer register `gp`; about 27% comes from "ordinary" procedural abstraction; architecture-specific abstraction of register save/restore sequences accounts for another 10%; and useless-code elimination accounts for about 22% of the savings. (Recall that our baseline programs have already had unreachable code and `no-ops` removed. The figure given here refers to code that subsequently becomes useless, due to interprocedural optimization, as discussed in Section 2.1.) The remainder of the savings arise due to a variety of interprocedural optimizations.

We also measured the extent to which basic blocks of different sizes contribute to the overall savings due to procedural abstraction. For small basic blocks, the savings per block abstracted tend to be small, but the likelihood of finding other similar blocks, and thereby increasing the total resulting savings, is large. The

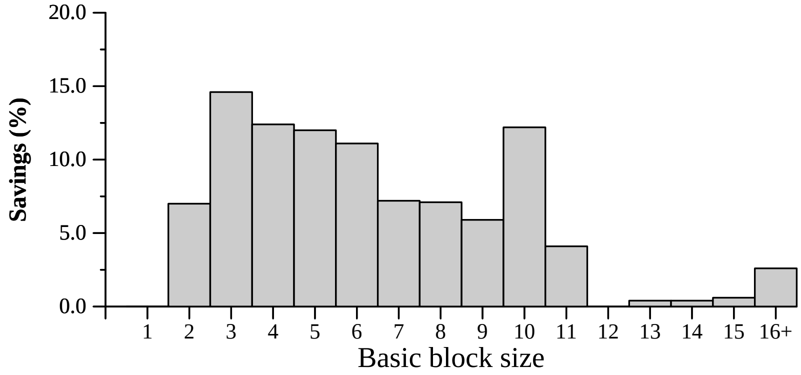


Fig. 9. Contribution to procedural abstraction savings for basic blocks of different sizes.

opposite is true for large blocks: each basic block that is abstracted accrues a significant savings, but the likelihood of finding similar or identical blocks that can be abstracted is not as high. The distribution of the average savings we observed for our benchmarks is shown in Figure 9. It can be seen that small blocks account for a significant amount of the savings: about 7% of the savings comes from blocks containing just two instructions, while close to 15% comes from blocks containing three instructions. Beyond this the savings generally drop off as the number of instructions increases, except for a large bump at basic blocks of size 10. The reason for this, it turns out, is that very often there is a large number of return blocks that restore all the callee-saved registers and the return address register from memory, deallocate the stack frame, and then return from the function. These actions require 10 instructions on the processor we used. The contribution of large basic blocks—those exceeding 12 instructions in length—is, on the average, quite small, even though occasionally we are able to abstract blocks that are quite long. (In the *gcc* and *vortex* benchmarks, basic blocks of up to 25 instructions are abstracted. In the *rasta* benchmark, such blocks can be up to 44 instructions long.)

As mentioned earlier, our experiments use statically linked executables, where the code for the library routines is linked into the executable by the linker prior to compaction. It is possible that library code is more (or less) compressible than user code. This could happen, for example, if the libraries are compiled using different compilers or compiler optimization levels. It is desirable to identify, therefore, the extent to which the presence of library code influences our results. For example, if it turns out that library code is highly compressible while user code is not, then our results would not be readily applicable to executables that are not statically linked. To this end, we instrumented *squeeze* to record, for each addition or deletion of code during its run, the function(s) with which the size change should be associated. For the classical optimizations implemented within *squeeze*, this is straightforward. For procedural abstraction, we use the following approach. Suppose that n different instances of a particular code fragment were abstracted into a procedure, resulting in a net savings in code size of m, then the function containing each of these instances is credited with a savings of m/n instructions (not necessarily an integral quantity). We then use a list of functions in the user code, obtained using a modi-

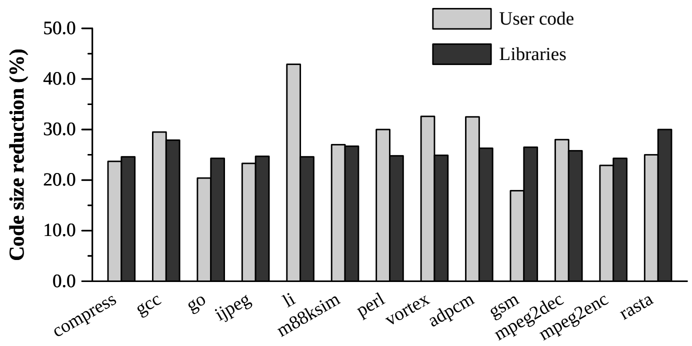


Fig. 10. Contributions to code size reduction: User code versus libraries.

fied version of the *lcc* compiler [Fraser and Hanson 1995], to estimate the total size of user code and the code savings attributable to it. These measurements do not account for indirect effects of having the library code available for inspection, such as improved precision of dataflow analyses, which may give rise to additional opportunities for optimization. Nevertheless, this information is useful for obtaining qualitative estimates of the influence of library code on our overall numbers. Our results are shown in Figure 10. The bars labeled “User code” represent the fraction of instructions in user code, relative to the total number of user code instructions, that were deleted in the process of code compaction, while those labeled “Libraries” give the corresponding figures for library code. For both the user code and libraries, the amount of reduction in code size typically ranges from around 25% to around 30%, with an average reduction of about 27% for user code and about 26% for library code.[11] There are a few programs (*li*, *perl*, *vortex*, *adpcm*) where the user code is noticeably more compressible than the libraries, and a few others (*go*, *gsm*, *rasta*) where the libraries are more compressible. In general, however, the user and library code are more or less comparable in their contribution to the overall code size reduction measured.

5.2 Code Speed

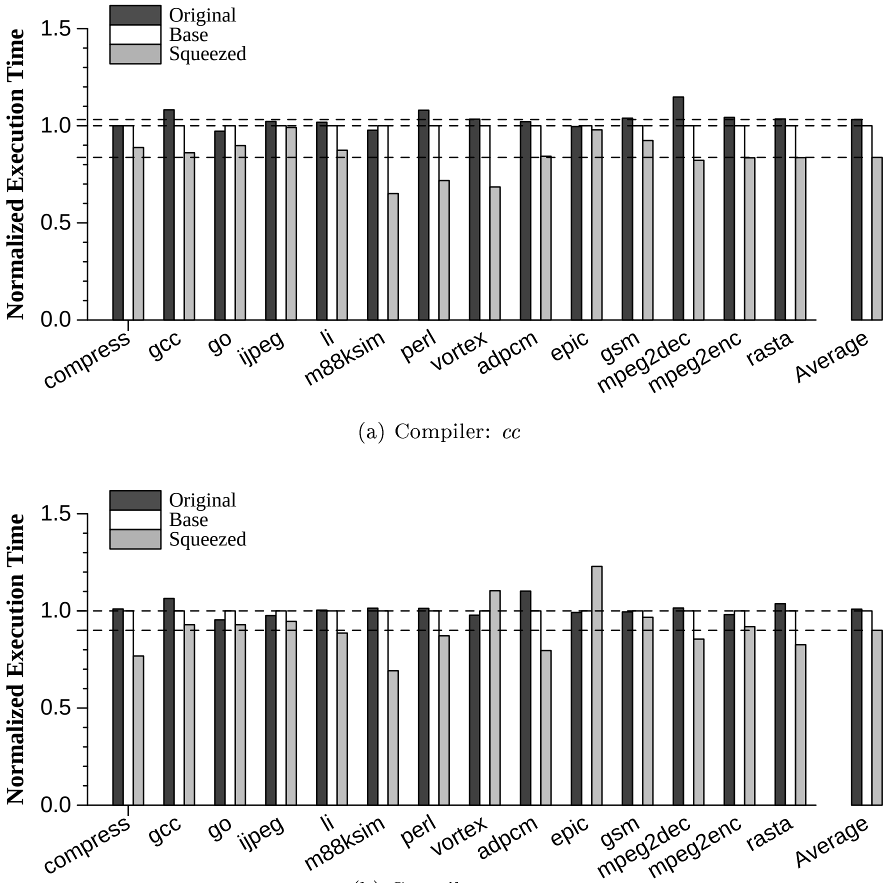
One intuitively expects the programs resulting from the code compaction techniques described here to be slower than the original code, primarily because of the additional function calls resulting from the procedural abstraction that occurs. A more careful consideration indicates that the situation may be murkier than this simple analysis suggests, for a number of reasons. First, much of the code size reduction is due to aggressive interprocedural optimizations that also improve execution speed. Second, transformations such as profile-directed code layout, which need not have a large effect on code size, can nevertheless have a significant positive effect on speed. On the other hand, on a superscalar processor such as the Alpha 21164, slow-downs can occur in the compressed code for reasons other than procedural abstraction, e.g., due to the elimination of `no-ops` inserted by the instruction scheduler in order

[11] These numbers refer to the control flow graph prior to code layout, i.e., before unconditional branches are added while linearizing the graph.

to align the instructions so as to increase the number of instructions issued per cycle.

To determine the actual effect of our transformations on our benchmarks, we compared the execution times of the original optimized executables with those resulting from the application of *squeeze* to these executables. Execution profiles, in the form of basic block execution counts, were obtained for each program using `pixie`, and these were fed back to *squeeze* during code compaction. The SPEC benchmarks were profiled using the SPEC training inputs and subsequently timed on the SPEC reference inputs. For each of the remaining benchmarks, we used the same input for both profiling and subsequent timing. The timings were obtained on a lightly loaded Compaq Alpha workstation with a 300-MHz Alpha 21164 processor with a split primary direct mapped cache (8 KB each of instruction and data cache), 96 KB of on-chip secondary cache, 2 MB of off-chip secondary cache, and 512 Mbytes of main memory, running Tru64 Unix 4.0. Our results are shown in Figure 11. The corresponding raw data are given in Debray et al. [2000]. In each case, the execution time was measured as the smallest time of 10 runs. The columns labeled "Original" refer to the execution times of the inputs optimized at the appropriate level for each compiler, as discussed earlier, but without the elimination of unreachable code and `no-ops`. These are provided as a reference point. The columns labeled "Base" refer to executables obtained by removing unreachable code and `no-ops` from the original executables and then performing profile-directed code layout. The execution times of the executables produced by *squeeze* correspond to the columns labeled "Squeezed."

The results of our timing experiments indicate that it is by no means a foregone conclusion that squeezed code will be slower than original code. For many of our benchmarks, the squeezed code runs significantly faster than the original. For example, for the *compress* benchmark compiled using *cc*, the squeezed executable is about 11% faster than the base and original executables, and using *gcc*, it is about 23% faster than the base and original executables. For *m88ksim* compiled using *cc*, the squeezed executable is about 35% faster than the base and about 36% faster than the original, and using *gcc*, it is about 30% faster than both the base and original. For *perl* compiled using *cc*, it is about 28% faster than the base and about 22% faster than the original, and using *gcc*, it is about 13% faster than the base and original. Only two programs suffer slow-downs as a result of code compaction: *vortex* and *epic*, both under the *gcc* compiler. The former slows down by about 10%, the latter by about 23%. The reasons for these slow-downs are discussed in Section 5.3. Overall, for the set of benchmarks considered, the average speedup, compared to both the base and original programs, is about 16% for the *cc*-compiled executables and about 10% for the executables obtained using *gcc*. In other words, code compaction yields significant speed improvements overall, and the compressed code performs favorably even when the performance of the original code is enhanced via profile-guided code layout. The reasons for this, explored in Section 5.3, are generally that for most of our benchmarks, the squeezed code experiences significant decreases in the number of instruction cache misses and the average amount of instruction-level parallelism that can be sustained.



(a) Compiler: *cc*

(b) Compiler: *gcc*

Fig. 11. Effects of compaction on execution time (normalized).

5.3 Low-Level Dynamic Behavior

To better understand the dynamic behavior of programs subjected to code compaction, we examined various aspects of their low-level execution characteristics. Our results, which are summarized in Figure 12, were obtained using hardware counters on the processor, in each case using the smallest of three runs of the program.

5.3.1 *Total Instructions Executed.* Code size reductions during code compaction come from two sources: interprocedural optimization and code factoring. Some interprocedural optimizations reduce the number of instructions executed: for example, the elimination of unnecessary gp register computations, elimination of no-ops inserted for alignment and instruction scheduling, dead-code elimination, and inlining of procedures called from a single call site. Other optimizations, in particular the elimination of unreachable code, have no effect on the number of instructions executed. Code factoring, on the other hand, leads to the execution of additional

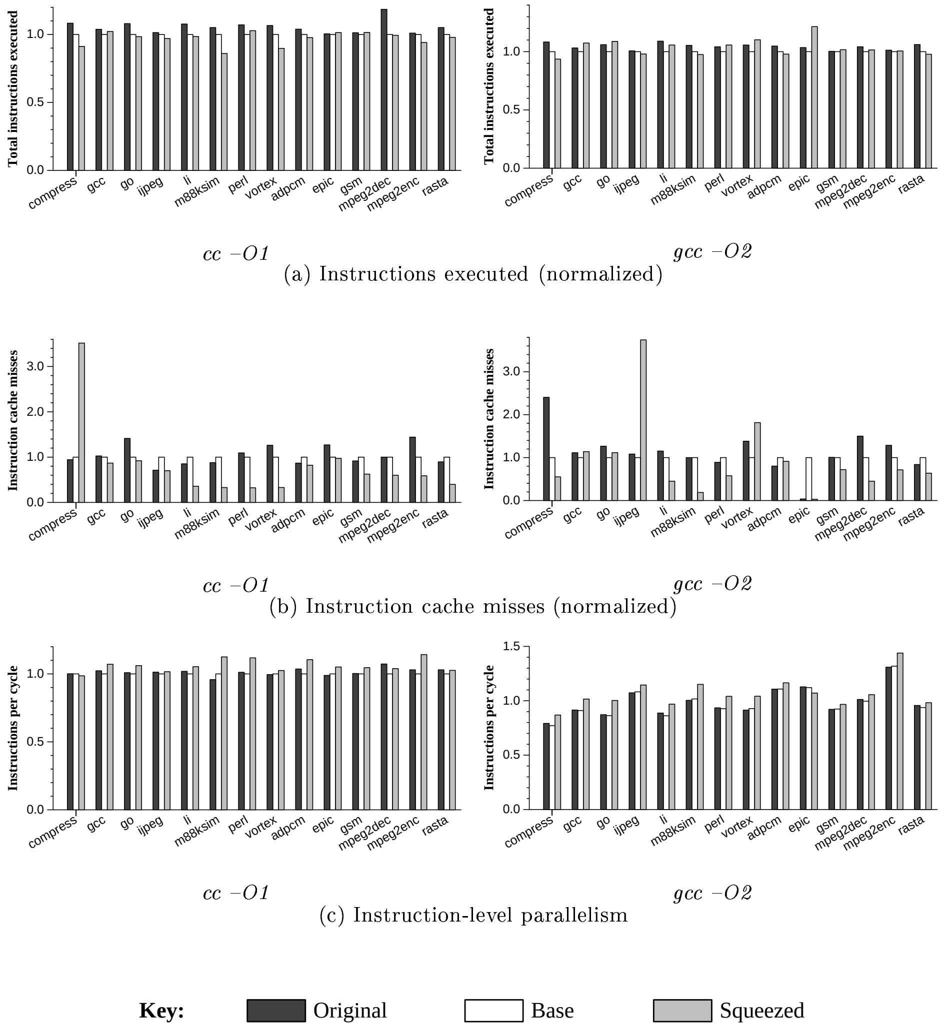


Fig. 12. Low-level dynamic behavior.

branch instructions for the procedure calls and returns, and so always results in an increase in the number of instructions executed.

Figure 12(a) shows the relative number of instructions executed by the original and the squeezed programs, compared to the base program. As one might expect, since the only difference between the original and base programs is that the base program has had unreachable code and no-ops eliminated, the base program always executes fewer instructions than the original. Moreover, the difference between these—due entirely to eliminated no-ops—is typically not large, ranging from about 1% to 9% and averaging about 4%. More interestingly, when we consider the code generated by *squeeze*, we find that for many programs, the squeezed version executes fewer instructions than the base programs. For these programs, the reduction in instructions executed resulting from optimizations by *squeeze* offset any dynamic increases due to factoring. For other programs, the effects of code factoring outweigh those due to optimizations, and result in a net increase in the number of instructions executed. Overall, we find that for the benchmarks considered, the squeezed versions of the code obtained for *cc* execute about 3% fewer instructions on the average than the base versions, while for the *gcc*-compiled binaries they execute a little over 3% more instructions, on the average.

5.3.2 *Instruction Cache Misses.* Since modern CPUs are significantly faster than memory, delivering instructions to them is a major bottleneck. A high instruction cache hit-rate is therefore essential for good performance. Primary instruction caches, in order to be fast, tend to be relatively small and have low associativity. This makes it advantageous to lay out the basic blocks in a program in such a way that frequently executed blocks are positioned close to each other, since this is less likely to lead to cache conflicts [Pettis and Hansen 1990]. However, code factoring can undo the effects of profile-directed code layout, by "pulling out" a code fragment into a procedure that cannot be positioned close to its call site. The problem arises when, for example, we have two instances of a repeated code fragment that are not close to each other but where both code fragments are frequently executed. If these code fragments are factored out into a procedure, there will be two frequently executed call sites for the resulting procedure, and it may not be possible to lay out the code in a way that positions the body of the procedure close to both of these call sites. This can lead to an increase in instruction cache misses.

Figure 12(b) shows the effect of code compaction on instruction cache misses. For the *cc*-compiled programs, the *compress* benchmark experiences a large increase in the number of instruction cache misses as a result of factoring. For the binaries obtained from *gcc*, two programs—*ijpeg* and *vortex*—suffer large increases in the number of cache misses, while two others—*gcc* and *go*—experience smaller but nevertheless noticeable increases. The number of instruction cache misses goes down for the remaining programs; in a few cases—notably, *compress*, *li*, *m88ksim*, *epic*, and *mpeg2dec*—quite dramatically. Overall, the squeezed programs incur 36% fewer instruction cache misses, on the average, for the *cc*-compiled binaries, and 40% fewer misses for the *gcc*-compiled binaries, than the corresponding base programs.

5.3.3 *Instruction-Level Parallelism.* The Alpha 21164 processor, on which our experiments were run, is a superscalar machine that can execute up to four instructions per cycle, provided that various scheduling constraints are satisfied. For example, at most two integer and two floating-point instructions can be issued in a cycle; and no more than one instruction in a group of simultaneously issued instructions should try to access memory or access the same functional unit. Instructions are fetched in groups of four, and each such group is then examined for opportunities for multiple issues by evaluating to what extent they satisfy these constraints. This means that it is possible for a plausible code transformation, such as the deletion of a `no-op` instruction, to alter the instruction sequence in such a way that opportunities for multiple instruction issues are reduced dramatically, with a corresponding loss in performance (conversely, the judicious insertion of `no-ops` can lead to an increase in the level of instruction-level parallelism that can be exploited).

To address this problem, *squeeze* carries out instruction scheduling after all other transformations have been applied and the final code layout has been determined.

Since *squeeze* eliminates `no-ops` inserted by the compiler for scheduling and alignment purposes, there is the potential for a significant loss in instruction-level parallelism in the code it produces. To evaluate whether this is the case, we measured the average number of instructions issued per cycle for the various executables. The results are shown in Figure 12(c). It can be seen that the elimination of `no-ops` incurs a price in the base program, where the average number of instructions issued per cycle is slightly smaller (by about 1% for *cc* and 0.5% for *gcc*) than the original program. However, the instruction scheduler in *squeeze* is able to overcome this problem and, for almost all of the programs tested, is able to attain a higher number of instructions per cycle. On the average, the instructions issued per cycle in the squeezed programs, compared to the base programs, improves by about 6% for the *cc*-compiled binaries and about 8% for the *gcc*-compiled binaries.

5.3.4 *Summary.* As Figure 11 shows, two of the 14 benchmarks we used, *vortex* and *epic* compiled under *gcc*, suffer a slowdown as a result of code compaction. Their low-level execution characteristics indicate the possible reasons for this. Like many of the other programs, code compaction causes an increase in the total number of instructions executed for both of these programs. While the other programs are generally able to compensate for this by improvements elsewhere, *vortex* suffers an increase in instruction cache misses, and *epic* suffers a reduction in the average number of instructions issued per cycle. Some of the other programs incur degradations in some dynamic execution characteristics but are able to compensate for this with improvements in other characteristics. For example, *compress* under *cc* and *ijpeg* under *gcc*, both of which suffer dramatic increases in the number of instruction cache misses, are nevertheless able to eke out overall improvements in speed due to a combination of a reduction in the total number of instructions executed and—for *ijpeg* compiled with *gcc*—an increase in the average number of instructions issued per cycle.

5.4 The Effects of Code Factoring

Figure 13 shows the effect of code factoring by itself on code size and execution time. The raw data are given in Debray et al. [2000]. The graphs compare *squeeze* performing all code transformations except for code factoring, against *squeeze* with code factoring enabled. It can be seen that factoring reduces the size of the programs by about 5–6%. An interesting aspect of this comparison is that the elimination of code due to various optimizations within *squeeze* has the effect of reducing the apparent efficacy of code factoring, since code that might otherwise have been factored is eliminated as useless or unreachable. The result of this is that the greater the code-shrinking effects of classical optimizations, the smaller we find the benefits due to factoring.

Since the smallest code unit we consider for procedural abstraction is the basic block, our approach does not pick out and abstract instruction sequences that are subparts of a block. By comparison, suffix-tree based approaches such as those of Cooper and McIntosh [1999] are able to abstract out repeated-instruction sequences that are subsequences of a block. Despite this limitation in our approach to code

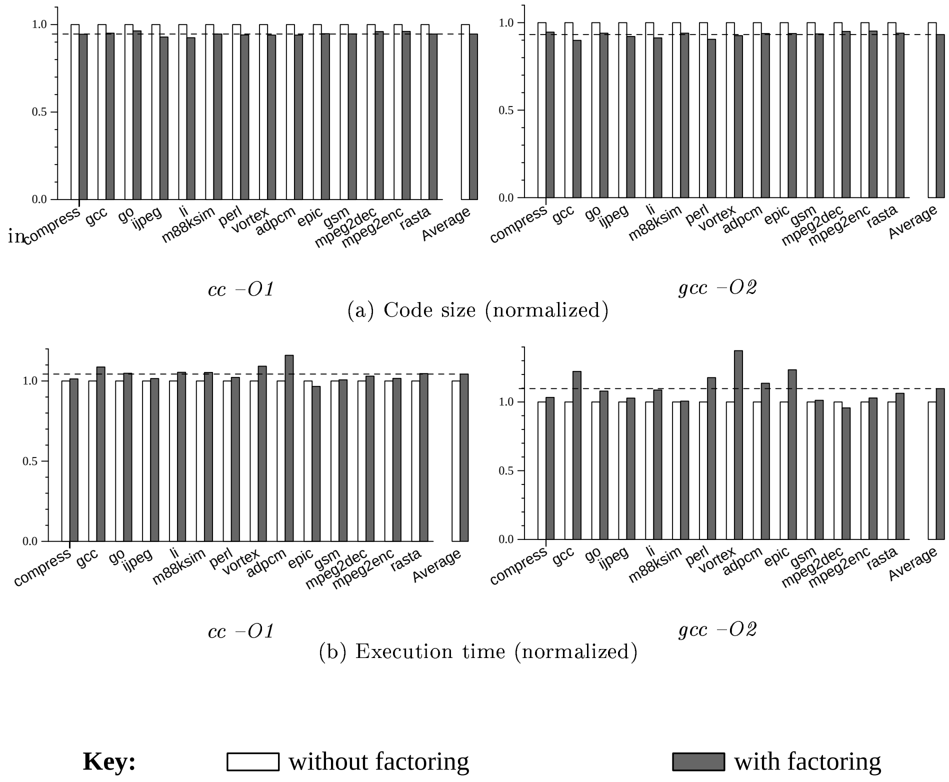


Fig. 13. Relative impact of code factoring on code size and execution time.

factoring, the relative size reductions we obtain via factoring are essentially the same as those of Cooper and McIntosh. A possible explanation for this is that the ability to abstract out subsequences within a basic block is likely to make a difference only for large basic blocks, and the proportion of such blocks generally tends to be small in most programs.

As one would expect, factoring causes an increase in the number of instructions executed. On the average, this results in an increase in execution time of about 4% for the *cc*-compiled binaries, and about 10% for the *gcc*-compiled binaries. Some *gcc*-compiled binaries experience significant slow-downs, with *vortex* slowing down by about 37%, *epic* by about 23%, and *perl* by about 18%.

6. CONCLUSIONS

This article focuses on the problem of code compaction to yield smaller executables. It describes a "whole-system" approach to the problem, where the use of aggressive interprocedural optimization, together with procedural abstraction of repeated-code fragments, yields significantly greater reductions in code size than have been achieved to date. For the identification and abstraction of repeated code fragments, it departs from classical suffix-tree-based approaches. Instead, it uses information already available in most compilers, such as the control flow graph and dominator/postdominator trees. Because it does not treat the program as a simple linear sequence of instructions, it can be more flexible in its treatment of what code fragments may be considered "equivalent." This simplifies the implementation and sets up a framework for code compaction that can be more flexible in its treatment of what code fragments are considered "equivalent." This results in a system that is able to obtain considerably greater compaction, even on optimized code, than previous approaches, without incurring significant performance penalties.

APPENDIX

A. THE LOCAL REGISTER-RENAMING ALGORITHM

Suppose we want to rename the registers in a basic block B_{from}, if possible, to make it identical to a block B_{to}. Pseudocode for the algorithm used by *squeeze* for this is shown in Figure 14. For simplicity of exposition, we assume that instructions are of the form $reg_3 = reg_1 \ op \ reg_2$. The ith operand of an instruction I is given by $I.\mathtt{Op}[i]$. We assume that operands 1 and 2 are the source operands, and operand 3 is the destination operand. In addition, each instruction I has fields $I.\mathtt{oldOp}[\mathtt{i}]$ that are used to keep track of the operand register before renaming. These fields are used to undo the renaming if necessary, and are all initialized to $\perp$. The algorithm maintains two global arrays, `InSubst` and `OutSubst`, that keep track of register moves that have to be inserted at the entry to and exit from the block, respectively, if the renaming is successful. Each element of these arrays is initialized to $\perp$.

The main routine that carries out the renaming is *RenameBlock*, illustrated in Figure 14. The basic idea is to work through each instruction in B_{from} and try to rename its operands to make it identical to the corresponding instruction in B_{to} without violating any semantic constraints. If this cannot be done, or if the total number of move instructions that must be inserted before and after the block exceeds the savings that would be obtained from procedural abstraction of the block, the renaming is abandoned. In this case, control is transferred to the label BAILOUT, where the renaming of each instruction in the block is undone.

The pseudocode for renaming individual operands is shown in Figure 15. The idea is to record the original value of the operand in the appropriate `oldOp` field of the instruction being renamed, rename the operand, and then propagate this renaming forward in the basic block until the register that is being renamed becomes redefined or the end of the block is reached.

ACKNOWLEDGEMENTS

We are grateful to Anders Lindgren and Johan Runeson (IAR Systems, Sweden) for pointing out some errors in an earlier version of this paper. Thanks are also due to Nathaniel McIntosh for helpful discussions, and for pointing us to the UCLA Mediabench benchmark programs. Comments by the anonymous reviewers were very helpful in improving the contents of the article.

```
function RenameBlock(B_from, B_to)
begin
    if NumInstr(B_from) ≠ NumInstr(B_to) return FAIL;

    n := NumInstr(B_from);
    LiveIn := {r | r is live at entry to B_from};
    LiveRegs := {r | r is live at entry to B_from};
    NumMoves := 0;
    SavedRegs := {r | r is a callee-saved register that is saved by the function containing B_from};
    Forbidden := LiveRegs ∪ {r | r is callee-saved and r ∉ SavedRegs};

    for i := 1 to n do
        ins_from := B_from[i] ≡ 'reg_3^from = reg_1^from op reg_2^from';
        ins_to := B_to[i] ≡ 'reg_3^to = reg_1^to op reg_2^to';
        if (ins_from ≠ ins_to) then
            for j ∈ {1, 2} do
                if reg_j^from ≠ reg_j^to and reg_j^from ∈ LiveIn then
                    if (InSubst[reg_j^from] ≠ ⊥) goto BAILOUT;
                    InSubst[reg_j^from] := reg_j^to;
                    NumMoves += 1;
                fi
                if (ReplaceOp(j, ins_from, ins_to, LiveIn) = FAIL) goto BAILOUT;
            od

            if the definition ins_from reaches the end of B_from then
                if the definition ins_to does not reach the end of B_to goto BAILOUT;
                OutSubst[reg_3^from] := reg_3^to;
                NumMoves += 1;
            fi
            if (ReplaceOp(3, ins_from, ins_to, Forbidden) = FAIL) goto BAILOUT;

            if (ins_from ≠ ins_to) goto BAILOUT;
            LiveIn := LiveIn − {reg_3^from};
            LiveRegs := (LiveRegs − {reg_3^from}) ∪ {reg_3^to};
        fi
    od

    if (NumMoves + 1 < n) then      /* the '+1' is for the bsr that will be added */
        InsertMoves(B_from, InSubst, OutSubst);
        return SUCCESS;
    fi

BAILOUT:
    for i := 1 to n do
        ins_from := B_from[i];
        if (ins_from.oldOp[1] ≠ ⊥) then ins_from.Op[1] := ins_from.oldOp[1];
        if (ins_from.oldOp[2] ≠ ⊥) then ins_from.Op[2] := ins_from.oldOp[2];
        if (ins_from.oldOp[3] ≠ ⊥) then ins_from.Op[3] := ins_from.oldOp[3];
    od
    return FAIL;
end
```

Fig. 14. Algorithm for local register renaming.

```
function ReplaceOp(k, ins_from, ins_to, Forbidden)
begin
    r_from := ins_from.Op[k];
    r_to := ins_to.Op[k];
    if (r_from = r_to) return SUCCESS;
    if (r_to ∈ Forbidden) return FAIL;

    ins_from.oldOp[k] := r_from;
    ins_from.Op[k] := r_to;

    for each instruction I after ins_from to the end of the block do
        for j ∈ {1, 2} do
            if (I.Op[j] = r_from) then
                if (I.oldOp[j] ≠ ⊥) return FAIL;
                I.oldOp[j] := r_from;
                I.Op[j] := r_to;
            fi
        od
        if (I.Op[3] = r_from) break;
    od

    return SUCCESS;
end

function InsertMoves(B_from, InSubst, OutSubst)
begin
    if ∃r : InSubst[r] ≠ ⊥ then
        if B_from has multiple predecessors then
            create a new basic block B' and redirect all edges entering B_from to enter B' instead;
            add an edge from B' to B_from;
        else
            B' := B_from;
        fi
        for each r' = InSubst[r] s.t. r' ≠ ⊥ do
            insert an instruction 'r' := r' in B';
        od
    fi
    if ∃r : OutSubst[r] ≠ ⊥ then
        if B_from has multiple successors then
            create a new basic block B'' and redirect all edges out of B_from to be out of B'' instead;
            add an edge from B_from to B'';
        else
            B'' := B_from;
        fi
        for each r' = OutSubst[r] s.t. r' ≠ ⊥ do
            insert an instruction 'r' := r' in B'';
        od
    fi
end
```

Fig. 15. Pseudocode for operand replacement and move insertion.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1985. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

Baker, B. S. 1993. A theory of parameterized pattern matching: Algorithms and applications (extended abstract). In *Proc. ACM Symposium on Theory of Computing.* ACM Press, New York, N.Y., 71–80.

Baker, B. S. and Manber, U. 1998. Deducing similarities in Java sources from bytecodes. In *Proc. USENIX Annual Technical Conference.* Usenix, Berkeley, CA, 179–190.

Beneš, M., Nowick, S. M., and Wolfe, A. 1998. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society, Washington, D.C.

Cooper, K. D. and McIntosh, N. 1999. Enhanced code compression for embedded RISC processors. In *ACM Conference on Programming Language Design and Implementation.* ACM Press, New York, N.Y., 139–149.

Debray, S., Evans, W., Muth, R., and De Sutter, B. 2000. Compiler techniques for code compaction. Tech. Rep. 00-04, Dept. of Computer Science, The University of Arizona. Mar.

Ernst, J., Evans, W., Fraser, C., Lucco, S., and Proebsting, T. 1997. Code compression. In *ACM Conference on Programming Language Design and Implementation.* ACM Press, New York, N.Y.

Franz, M. 1997. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. In *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds. Number 1222 in Springer Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 263–276. Tech. Report 97-04, Department of Information and Computer Science, University of California, Irvine.

Franz, M. and Kistler, T. 1997. Slim binaries. *Commun. ACM 40,* 12 (Dec.), 87–94.

Fraser, C. and Proebsting, T. 1995. Custom instruction sets for code compression. Unpublished manuscript. `http://research.microsoft.com/ toddpro/papers/pldi2.ps`.

Fraser, C., Myers, E., and Wendt, A. 1984. Analyzing and compressing assembly code. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction.* Vol. 19. ACM Press, New York, N.Y., 117–121.

Fraser, C. W. and Hanson, D. R. 1995. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, Reading, Mass.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, N.Y.

Knoop, J., Rüthing, O., and Steffen, B. 1994. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst. 16,* 4 (July), 1117–1155.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufman, San Francisco, CA.

Muth, R., Debray, S. K., Watterson, S., and Bosschere, K. D. 1998. `alto` : A link-time optimizer for the DEC Alpha. Tech. Rep. 98-14, Dept. of Computer Science, The University of Arizona. Dec. To appear in *Software Practice and Experience.*

Pettis, K. and Hansen, R. C. 1990. Profile-guided code positioning. In *ACM Conference on Programming Language Design and Implementation.* ACM Press, New York, N.Y., 16–27.

Proebsting, T. 1995. Optimizing an ANSI C interpreter with superoperators. In *Proc. Symp. on Principles of Programming Languages.* ACM Press, New York, N.Y., 322–332.

van de Wiel, R. 2000. The "Code Compaction" Bibliography. `http://www.win.tue.nl/cs/pa/rikvdw/bibl.html`.

Zastre, M. J. 1993. Compacting object code via parameterized procedural abstraction. M.S. thesis, Dept. of Computing Science, University of Victoria.