Link-Time Binary Rewriting Techniques for Program Compaction

BJORN DE SUTTER, BRUNO DE BUS, and KOEN DE BOSSCHERE
Ghent University

Small program size is an important requirement for embedded systems with limited amounts of memory. We describe how link-time compaction through binary rewriting can achieve code size reductions of up to 62% for statically bound languages such as C, C++, and Fortran, without compromising on performance. We demonstrate how the limited amount of information about a program at link time can be exploited to overcome overhead resulting from separate compilation. This is done with scalable, cost-effective, whole-program analyses, optimizations, and duplicate code and data elimination techniques. The discussed techniques are evaluated and their cost-effectiveness is quantified with SQUEEZE++, a prototype link-time compactor.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation; compilers; optimization*; E.4 [**Coding and Information Theory**]: *Data compaction and compression*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Program representation, compaction, interprocedural analysis, code abstraction, whole-program optimization, linker, binary rewriting

1. INTRODUCTION

Die area is one of the most important factors contributing to the production cost of embedded, mass-produced, consumer electronics systems. As on-chip memories occupy large fractions of the die area, keeping these memories small reduces production cost. In addition, smaller memories (on or off chip) consume less power, an important consideration for devices that rely on batteries. So even though computer memory has become increasingly cheap over recent decades, embedded devices continue to require limited amounts of it. Because

B. De Sutter is supported by the Fund for Scientific Research—Flanders (Belgium) as a postdoctoral research fellow. B. De Bus is supported by a grant from the Flemish Institute for the Promotion of the Scientific Technological Research in the Industry (IWT). Part of the research reported in this article was supported by Ghent University.
Ghent University is a member of HiPEAC.
Authors' address: Electronics and Information Systems Dept., Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium; email: {brdsutte, bdebus, kdb}@elis.UGent.be.

memories can only be made as small as the programs that need to be stored in them, smaller programs imply cheaper, smaller, lighter, and more autonomous devices.

Our goal is to produce the most compact applications while retaining the same or similar levels of functionality, performance, and other key criteria. Developing compact applications is not simple, however. Object-orientation, component-based programming, and other modern software engineering techniques increase programmer productivity, improve software reliability, and shorten time-to-market by hiding lower-level issues from the programmer and by enabling code reuse. Unfortunately, this often comes at the expense of program size. Reusable code libraries, for example, are written with general applicability in mind and provide more functionality than is typically needed by any single application. Unless the unused functionality can be eliminated, an application will be larger than necessary. Moreover, program optimizations performed at compilation time, either on application code or on reusable library code, are limited because the whole program is not available for optimization. Again, the result is increased program size.

This article discusses link-time binary rewriting techniques to overcome the discrepancy between modern software engineering practices and the need for compact programs. The discussed techniques are applicable on programs written in statically bound languages such as Fortran, C, or C++. Their goal is to eliminate unnecessary computations and duplicated code and data from a program.

Link-time compaction offers several potential advantages over compile-time optimization. First, all code is available for inspection and compaction at link time, even for mixed-language programs. This includes library code that is statically linked with a program, even if this library code is distributed in a machine code format only. Link-time rewriting therefore requires no change to the often-used business models under which software is distributed in a machine code format. Second, at link time, machine-specific optimizations are possible because the link-time techniques are applied on assembly code. Finally, link-time rewriting for compaction only requires modifying the linker, while all other tools in program development chains, such as compilers, need not be modified.

1.1 Contributions of this Article

In order to enable a link-time binary rewriting approach for program compaction, two major issues have to be addressed. First, no information other than that required by the linker can be assumed to be available to the link-time rewriter. Type information, for example, is usually not available in the binary code that is passed to the linker. An important contribution of this article shows that the limited amount of information available at link time is sufficient (1) to correctly compact a program, and (2) to detect compiler-generated code that may violate the application binary interface (ABI) or calling conventions. This detection is based on the fact that wherever a compiler generates code that violates these conventions or that needs special treatment by the linker,

the compiler must notify the linker. Therefore, any link-time rewriter is also notified of possible violations, and thus of adherence to conventions where no violations occur. This knowledge enables more aggressive program analysis and compaction.

The second major issue addressed by this article is that link-time whole program analyses and transformations are efficient, effective, and scalable. This article focuses on scalability. In particular, the techniques discussed in this article address scalability issues by explicitely targeting separately compiled code, rather than all possible exotic code that an assembly programmer can write. While the latter has to be handled correctly, most of the code compaction opportunities are found in compiler-generated code, so that is where we have focused our effort.

1.2 Structure of the Article

This article is structured as follows. Section 2 provides background information on how compilers and linkers interact, with special emphasis on the program overhead that separate compilation introduces. Section 2 also discusses information about a program that is available at link time, and how this information relates to the overhead. Section 3 then describes how link-time information can be used to build an internal program representation that is suited for effective program compaction. Whole-program optimization techniques, aimed at eliminating superfluous code and data from a program, are discussed in Section 4, while techniques to avoid unnecessary duplication of code and data are the topic of Section 5. How all the techniques are combined into an effective and efficient tool is discussed in Section 6. The assumptions made throughout this article are elaborated in Section 7, after which our prototype link-time compactor is evaluated in Section 8. Finally, related work is discussed in Section 9 and conclusions are drawn in Section 10.

2. COMPILERS AND LINKERS

This section provides background information on the interaction between compilers and linkers. In particular, it discusses the overhead resulting from separate compilation and the related information that is available to a link-time optimizer.

2.1 The Tool Chain

Figure 1 depicts a traditional programming tool chain. On the left-hand side, source code files (src1.c, src2.f, and src3.C) in different programming languages are compiled into *object files* (obj[1–4].o) by the language-specific compilers. The object files generated contain the generated machine code (hereafter called *object code*), the statically allocated data (hereafter called *object data*), and additional information that tells the linker how to combine the separately generated object files into a single working program (a.out).

Although there is no difference between the compilation of libraries and programs, there is a difference between how the generated object files are used. Instead of being linked into a program directly, all of a library's generated

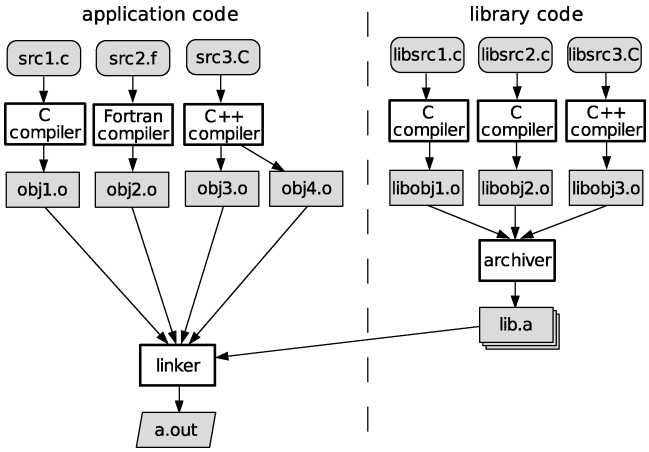

Fig. 1. Compilers, archivers, and linkers transform the source code files shown on top into the executable program shown at the bottom.

object files are first archived into a library file (lib.a on the right-hand side of Figure 1). When a program is statically linked with the archived library, the linker extracts only the library's object files that are needed by the program and links the library's needed object files with the program's object files.

Typically, library code is compiled separately from application code. In addition, the source code files of an application may also be compiled separately. During the separate compilation of one source code file or module,[1] the compiler has no knowledge of the other modules. As a consequence, compiler optimizations are conservative, which results in significant amounts of overhead in the object files.

The compaction techniques in this article try to remove as much of that overhead as possible by applying an additional program rewriting step at link time. It is at this point that all code is available for inspection and compaction. Throughout this article, we will indeed assume that all code constituting a program is available at link time, or in other words, that the programs are statically bound. This assumption is not valid for more dynamic languages such as Java, where, because of reflection, the complete program may never be available. Dynamic shared libraries, of which the code is only available when a program is loaded into memory [Franz 1997b; Levine 2000], also violate this assumption.

2.2 Overhead Resulting from Separate Compilation

There are four fundamental reasons for overhead in separately compiled programs.

2.2.1 *Limited Optimization and Analysis Scope.* With separate compilation, compile-time program analyses are limited to single modules. As a result,

[1]Whereas *file* refers to an entity stored on a disc, *module* denotes the corresponding entity in an intermediate program representation in the compiler.

the compile-time optimizations need to be very conservative. Even when a compiler analyzes all source code of a program at once, this does not include precompiled library code. This is particularly problematic because libraries are written with general applicability in mind. As such they contain much more functionality than is needed by any single program. Moreover, the simple library object file extraction scheme that linkers implement is not fine-grained enough to ensure that no unneeded functionality is linked into specific programs.

2.2.2 *Intermodular Data Passing.* During the compilation of one module, the machine code generated for other modules is unknown. As a result, *calling conventions* need to be used to pass data at intermodular control flow transfers. These conventions partition the available registers into argument registers, return value registers, callee-saved and caller-saved registers, etc. When a compiler generates machine code that passes data between modules, the compiler has to assume that all of the prescribed conventions need to be obeyed. Often, this assumption is overly conservative, but besides whole-program optimization, no workaround exists.

A typical example is the restoring of callee-saved registers prior to an intermodular procedure return. In this case, the compiler must assume that all callee-saved registers hold contents that will be used by the callee's (unknown) callers after returning to them. In practice, it often occurs that some callee-saved registers do not hold useful contents in a callee's callers. Hence their contents do not need to be restored by the callee.

2.2.3 *Unknown Addresses.* Each object file is partitioned into a number of different *object sections*. Each such section contains a specific type of code or data.[2] The `.text` section, for example, contains the object code; the `.rdata` section contains read-only object data (such as initialization values or literal strings), the `.data` section contains the mutable, that is, overwritable, object data, and the `.zero` section contains the zero-initialized data. During the linking process all object sections of the same type are combined into *program sections*, as illustrated in Figure 2.

Since the linker determines the final memory locations of the thus combined code and data, their final addresses are not known by the compiler. Therefore the compiler has to generates code and data at temporary addresses. Such code and data is called *relocatable*. Once the final locations of all code and data are determined, the linker replaces all temporary addresses stored in the code and data by the corresponding final addresses. This process is called *relocation*.

Unfortunately, relocatable code is often far from optimal. Since the relocation process is nothing more than the replacement of temporary addresses by final ones, all placeholders for temporary addresses (absolute or relative) have to be at least as wide as the widest final address that might need to be stored in them.

[2]Although the techniques discussed in this article are evaluated on the Alpha-Tru64Unix platform, the discussion of separate compilation issues and object formats is not based on the symbolic object format that exists on that platform. Unless explicitly stated otherwise, all background provided in this section and all assumptions made hold for all RISC and CISC platforms and object file formats the authors are aware of, including ELF, COFF and PE [Levine 2000].

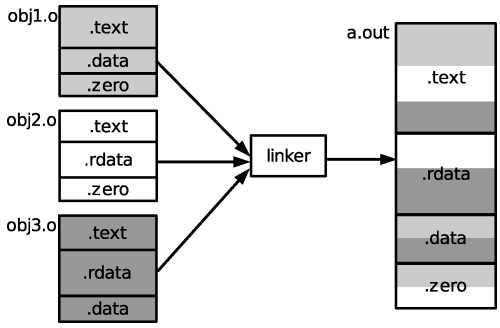

Fig. 2. The object sections in three object files shown on the left are combined by the linker into corresponding program sections in the executable program shown on the right.

In a 32-bit architecture, for example, all placeholders must be 32 bits wide. On most 32-bit RISC architectures (as well as 16-bit and 64-bit architectures) immediate operands of instructions cannot be 32 bits wide. This is because some of the bits in a 32-bit instruction are needed to encode the instruction's functionality. Absolute addresses therefore cannot be encoded as immediate operands of single instructions. On most RISC architectures, absolute addresses cannot even be encoded in the immediate operands of two instructions. The most efficient solution to produce such unknown but wide enough addresses then is to load them from memory. Of course, a similar problem now arises: where do we load the addresses from?

A common solution to this problem is the use of *Global Offset Tables* or GOTs. These tables contain the addresses of all global code and data and are stored in a section called `.got`.[3] To access global code or data, its address is first loaded from a GOT by indexing the *Global Pointer* or GP. The GP is a designated register that always points to the GOT of the code being executed.

The inefficiencies of GOTs have long been recognized [Srivastava and Wall 1994]. Besides the overhead needed to ensure that the GP always points to the part of the GOT associated with the code being executed, the indirection through the GOT is itself often not necessary. Without a whole-program overview, generating more efficient code is not possible, however.[4]

2.2.4 *Duplicated Code.* The fourth problem of separate compilation is the compiler's unawareness, during the compilation of one module, of the computations that are also implemented in other modules. The resulting code duplication obviously constitutes overhead.

[3]Unlike global code and data, local data is most often stored on the stack and accessed by indexing the stack pointer.

[4]On architectures with an explicit program counter, such as the ARM architecture, the GOT need not be a contiguous table. Instead, the GOT can be distributed in between the code and accessed through the program counter. This way, no register needs to function as a global pointer. CISC architectures that allow encoding absolute addresses as immediate operands can be seen as a special case of distributed GOTs; each absolute address that needs to be loaded at some point is encoded in the instruction that will use the loaded address. Fundamentally, the use of distributed GOTs or immediate operands is not different from the use of a contiguous GOT.

This problem becomes particularly important when code is automatically generated by code generators or precompilers that generate code from macros or templates. This is often the case in rapid application development environments that generate source code templates which the programmer has to complete. If the same code template is used repeatedly, one can expect that, unless measures are taken, a great deal of duplicated code will end up in the final program.

2.3 A Separate Compilation Example

Figure 3 illustrates how two source code files (A.c and B.c) are compiled separately into object files (A.o and B.o) and linked into a single executable (a.out). For the sake of generality, we have disassembled all machine code instructions in the object files and the executable into simple C statements. The target architecture is a 32-bit RISC architecture with five registers: `r0` to pass arguments and return values, `r1` and `r2` to store temporary values, `sp` to store the stack pointer, and `gp` to store the global pointer. `r1` is callee-saved, while `r2` is caller-saved.

In A.c, a global array `R` is defined, together with two global variables: `a` and `b`. The `main()` procedure calls procedure `f()` in B.c, and `f()` returns a value that depends on the array and variables defined in A.c.

Both the object files consist of three parts: relocatable code and data sections, symbol information, and relocation information. The latter two are used the replace temporary addresses, and are discussed later. The executable a.out contains only the combined code and data sections at their final addresses.

In both object files A.o and B.o, the code sections start at the temporary address 0x00. The code section in A.o contains the code generated for the `main()` procedure. After the GP has been set to point to the `.got` section, the address of procedure `f()` is loaded from the GOT, using the GP. The argument of the call is stored in register `r0`, and the call is executed. The GOT or `.got` section in A.o consists of one placeholder to store the address of `f()`, while the `.data` section in A.o holds the array `R` and the variables `a` and `b`.

After the GP has been set in procedure `f()` in B.o, `f()` first pushes the callee-saved register `r1` onto the stack and, just prior to returning, f() restores the callee-saved register by popping it off of the stack. This way, the calling conventions are maintained. The addresses of the three values that need to be added are all loaded in `r2` from the GOT through the GP, and the result of the consecutive additions is stored in register `r0`. The `.got` section in B.o provides placeholders for the addresses of `R`, `a` and `b`, since all three variables are accessed in `f()`. No global variables are defined in B.c, however, so there is no `.data` section in B.o.

Relocations indicate which addresses in the code and data sections are temporary. These addresses are marked in gray in Figure 3. Relocations `RA0` and `RA1` indicate that the instructions at temporary addresses `.text+0x00` need to set the GP to point to the GOT in the final program. Since multiple GOTs from object files will be combined in the final program, the indexes used to load addresses from the GOT through the GP also change in the final program. The required changes to the indexes are marked by relocations `RA2`, `RB4`, `RB5`, and `RB6`. Finally, the addresses of global code and data need to be put into the

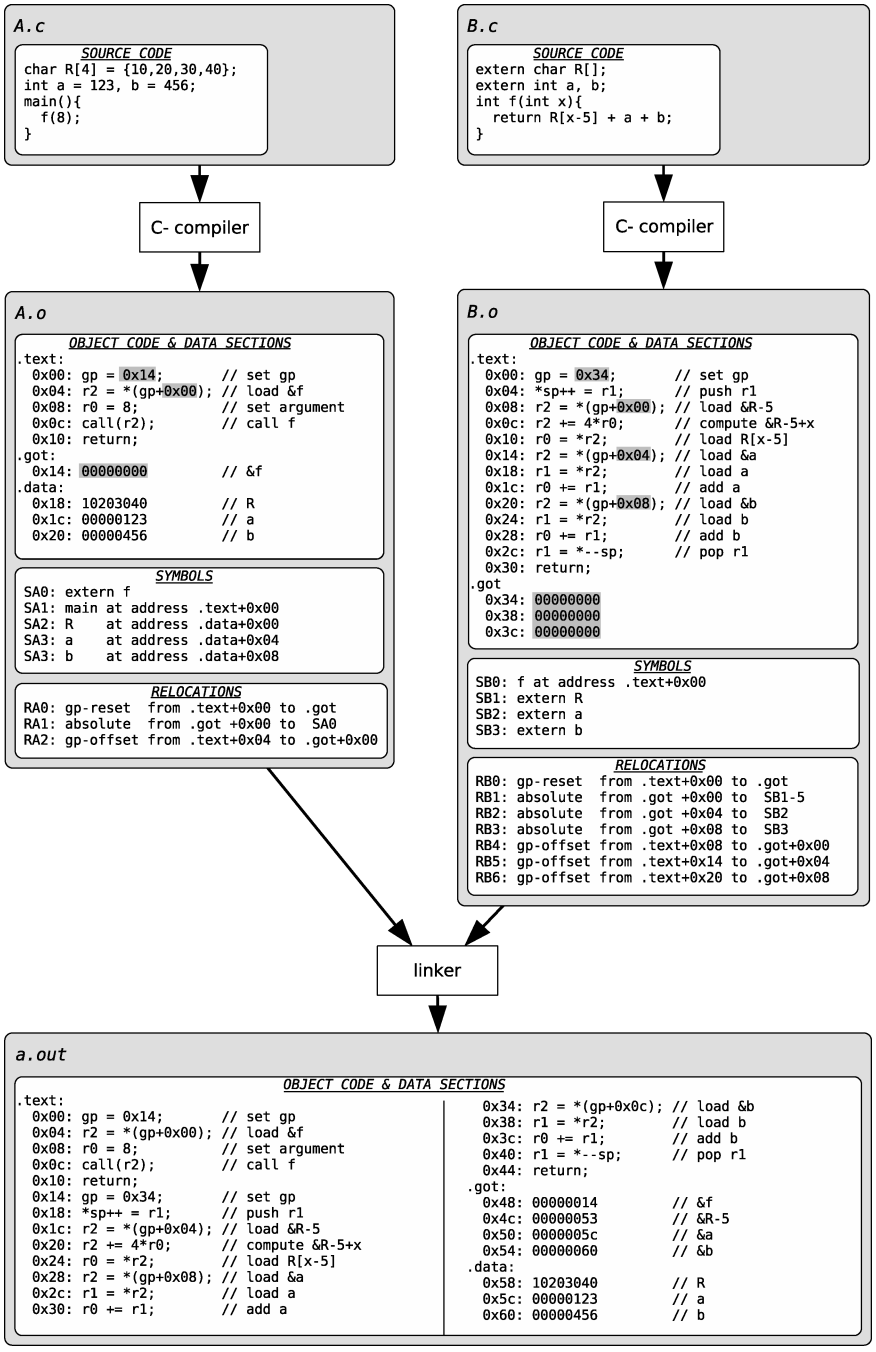

Fig. 3. The two C source code files on top (A.c and B.c) are compiled into two object files (A.o and B.o) and linked into an executable (a.out).

GOTs. For each such address, the required relocation is described via *symbol information*.

Each global code or data element defined in a program has one corresponding *defining symbol*. For example, symbol `SB0` in B.o defines procedure `f()` as the code starting at temporary address `B.text+0x00`. Similarly, symbol `SA2` in A.o defines array `R` as the data beginning at address `A.data+0x00`.

When global code or data is referenced in an object file, but not defined, a *referencing symbol* is used. Symbols `SA0`, `SB1`, `SB2`, and `SB3` are examples of such referencing symbols. The linker will match each referencing symbol with the corresponding defining symbol. After this *symbol resolution*, all relocations with referencing symbols are applied using the corresponding defining symbol. For example, `RA1` in A.o refers to `SA0` in A.o. By resolving `SA0` with `SB0` in B.o, the linker knows that, in the final program, the final address of `B.text+0x00` needs to be stored at the final address of `A.got+0x00`. In other words, the address 0x14 is stored at location 0x48 in the final program.

Note that the linker also relies on symbol resolution to decide which library object files need to be linked into a program. Library object files are iteratively added until all referencing symbols are resolved to a corresponding defining symbol.

Now let us examine some inefficiencies in the generated code. First of all, register `r1` is in fact unnecessarily pushed and popped onto and of the stack in `f()`, since `r1` holds no useful data at the only call-site of `f()` in `main()`. Of course, the compiler could not know this when B.c was compiled.

Second, two of the three indirections through the GOT could have been avoided in the code of `f()` if the compiler would have known that `R`, `a`, and `b` are located next to each other in memory. In that case, then generated code would access all three variables from the same base address, instead of loading three different base addresses. In fact, in the final program all three variables are so close to the GOT that they can be accessed directly from the GP. For example, a's value can be loaded directly with `r1 = *(gp+0x18)`. The compiler did not know this, however, since it didn't know the sizes of the object sections in other object files.

Finally, resetting the GP upon entry to `f()` was unnecessary because one GP value suffices to index the whole GOT of this small program.

2.4 Object File Information Available at Link Time

From our introduction to the linking process, it follows that all object files contain at least the generated object code and data, partitioned over a number of sections, and relocation and symbol information [Levine 2000]. Together with the knowledge of calling conventions, this is the only information our link-time rewriting step has at its disposal. While other information, such as additional information for debugging purposes, might be included in the object files, we will not use this information. It is not needed for simple linking, and hence depending on its availability would limit the applicability of our techniques.

2.4.1 *Relocation Information.* All temporary absolute addresses in the object files are annotated with relocations. Therefore relocation information

allows us to detect absolute addresses in the program, to distinguish them from ordinary numerical constants, and to adapt them once a program is compacted. All *intersection* relative addresses, that is, relative addresses corresponding to a displacement between two different object sections, are annotated by relocations as well, since no final values of such displacements are known at compile time.

No relocations need to be provided for *intrasection* relative addresses, however. Since instructions or data within an object section are not reordered by the linker, the intrasection displacements are never changed by the linker.

For relative intrasection data addresses, it is in general undecidable which data will be accessed with them. As a consequence, we need to treat such addresses very carefully. So far, the only conservative way we are aware of is to prohibit any link-time change to intrasection displacements. In other words, no motion or elimination of data within object sections is allowed. Instead, as far as data sections are concerned, we can only move or eliminate whole object sections.

For relative code addresses (such as computed addresses used in switch statements), we will simply assume that they are all relocatable. (The validity of this assumption is discussed in Section 7, when we reflect on all the assumptions we make throughout this article.) Assuming that all code addresses are annotated with relocations, we know the set of all code addresses that might be loaded or computed during the execution of a program. As this set is, by definition, a superset of the possible targets of indirect control flow transfers (i.e., transfers to an instruction whose address is computed or loaded), the relocation information on code addresses enables us to estimate all the targets of indirect control flow transfers. This allows us to build a conservative representation of the control flow of a program to be compacted, as is discussed in Section 3.

One important property, following from the fact that all absolute addresses and all intersection relative addresses are relocatable, is that an object (code or data) section can only be accessed through a relocatable address. In other words, for data in a data object section to be accessible, at least one relocatable address in another object section in the final program has to point to it, and that address must actually be used during the execution of the program. If all addresses pointing to a code or data object section are not used anywhere in the program, then that data object section can be removed.

2.4.2 *Symbol Information.* For symbol resolution, the only required symbol information is the relation between global code or data names and their addresses.[5]

For link-time compaction, global symbols defining procedures provide useful information. They indicate the entry-points of global procedures. Note, however, that such symbols need not be present for procedures that are not global, such as `static` procedures in C. In fact there can never be defining global symbols

[5]In programming languages where objects such as procedures can be overloaded, the symbol resolution relies on type information as well. Name mangling is then used, in which the names of the involved symbols (such as overloaded procedures) are extended with cryptic type descriptions.

for them. If such symbols existed, name clashing would result in faulty symbol resolution.

This important observation can be turned upside-down to our advantage as follows. If a procedure is global, that is, defined by a global symbol, it is exported from its module, and therefore the compiler is forced to assume that the procedure might be called from within other modules. Consequently, the compiler will have generated code that respects the calling conventions. In short, global procedures respect the calling conventions!

3. INTERNAL PROGRAM REPRESENTATION

The internal program representation used to facilitates our program transformations is the *Control Flow Graph* (CFG). This graph contains basic blocks as nodes, and potential control flow paths as edges. The basic blocks themselves consist of a list of instructions. As we will demonstrate, the CFG meets two important requirements: it can be built using the information we have available at link time and it enables the efficient implementation of the transformations that we want to apply.

This section describes the CFG components and the construction of the CFG. The actual transformations applied to the CFG are discussed in later sections.

3.1 Instructions

After the symbol resolution process is finished and all necessary object files are collected, the CFG construction stars with disassembling all code sections. This first step involves detecting the bits representing instructions in the code sections of the program, and decoding all instructions from their binary representation into a representation that is more easily manipulated.

For RISC platforms where no data is mixed between the code, disassembling the code sections is straightforward. All instructions occupy the same number of bits, and all bits in the code sections are in fact code. Although many articles have been written about the problem of differentiating mixed data and CISC code (see, for example, Schwarz et al. [2002]), we believe this to be a nonissue. If it is useful to separate code and data easily, compilers should generate tags (or symbols) in the object files that mark code and data. It required, for example, a patch of less than 10 lines to have the GNU assembler do this. With such tags, disassembling code is also straightforward for mixed data and CISC code.

For the disassembled code, we need to choose either a more concrete or a more abstract internal instruction representation. Do we choose a low-level assembler representation, or do we decompile the instructions to a higher-level RTL language or abstract syntax trees? Do the disassembled instructions operate on architectural registers or on static-single assignment (SSA) [Cytron et al. 1991] code where the operands are symbolic registers?

While many factors influence this choice, we will focus on the arguments for not choosing an SSA representation. The main attraction of an SSA representation is the elegance and efficiency it offers for the the implementation of a great deal of analyses and transformations. Since all control and data dependencies are explicitly encoded in the SSA representation, the analyses and transformations applied to it more or less automatically respect all data and

control dependency constraints. Of course this assumes that all other correctness constraints can be dealt with when real machine code is generated from the SSA representation.

One such constraint is the number of registers containing live values at each program point. This does not matter in the SSA representation, as the number of symbolic registers is virtually infinite. It does matter for real assembly code, however, because the number of architectural registers is always limited. To solve this problem, compilers allocate architectural registers when assembly code is generated. During the register allocation, the necessary register spills are inserted in the code. In reentrant code, spilled registers are spilled onto the stack.

At link time, things are not that simple. We have not yet found a satisfactory link-time method to determine which procedures are reentrant. Therefore, almost all spill code that we would need to introduce, after having transformed a program at link time, would have to spill onto the stack. Unfortunately, we have not yet found a link-time analysis that reveals, in enough detail, how the stack is used by a program to enable the insertion of additional stack space for all but the most trivial register spills.

The resulting inability to insert arbitrary register spills in a program at link time is our main reason not to use a SSA program representation. Any analysis or transformation on such a SSA representation would still need to take into account the constraints on the number of live registers. The implementation of such an analysis would be neither more elegant nor more efficient than our current code that operates on what is basically an assembly code representation. The operands of this representation are real architectural registers that offer the additional benefit, on most architectures, of not being aliased. On some architectures, such as the SPARC and IA64 architectures, some aliasing between registers exists because of software-controlled register renaming of register windows. But since all renaming is explicit on such architectures, the aliasing can be handled easily by inserting dummy instructions that mimic register copy operations.

Contrary to internal compiler representations, there is no notion of variables in a link-time intermediate representation. This follows from the fact that symbol information in the object files only defines labels, not variables. Furthermore, link-time points-to and alias analysis are very imprecise [Debray et al. 1998]. Consequently, we treat memory to a large extent as a black box, through which no information is propagated during data flow analysis. Instead the data flow analyses are limited to information propagated from registers to registers. Two exceptions to this rule are instructions loading data from a known, constant address in a read-only data section (for which we know what data they load), and easily analyzable register spills onto the stack (see Section 4.1).

3.2 Basic Blocks

Basic blocks are the nodes in the CFG. They consist of a sequence of instructions. Once the code is disassembled, basic blocks are easily detected with the *leader algorithm* [Aho et al. 1986] and the relocation information. All targets of

direct control flow transfers are entry points (so-called *leaders*) of basic blocks, as are all instructions directly following direct or indirect transfers. Since all code addresses annotated with relocations are initially assumed to be possible targets of all indirect control flow transfers, the instructions at those addresses are considered entry points of blocks as well. As such, each basic block consists of the instructions between (and including) its leader instruction and (excluding) the leader of the next block.

3.3 Edges

Detecting the edges for the CFG is as easy as detecting basic blocks. We can get them directly from the disassembled code for direct control flow transfers that encode their target in the instruction itself. For indirect transfers, we assume that all the instructions at relocatable addresses can be the targets. It suffices, therefore, to connect all indirect control flow transfer instructions with all possible (relocatable) targets. Finally, fall-through edges are added to all basic blocks ending with an instruction that does not unconditionally transfer control.

By initially assuming that all relocatable code addresses can be potential targets of all indirect control flow transfers, we build a very conservative CFG that contains many unrealizable execution paths. Section 4 will show that the information gathered by the analyses on this representation can be used to compact the program, as well as to remove unrealizable paths from the graph.

3.4 Procedures

At the source code level, the relation between procedures can usually be modeled by a call graph that encodes which procedures get called at which call-sites. This is not the case at link time. First, compile-time optimizations such as tail-call optimization or partial inlining result in interprocedural control flow transfers other than calls and returns. In addition, manually written assembly code can also contain all kinds of interprocedural control flow. When this occurs, it is not always easy to determine which procedure a basic block belongs to. A block may, in fact, belong to more than one procedure. Still, it is important to include the notion of procedures in the program representation, because this will often allow one to analyze a procedure call (and the execution of the callee) as an atomic operation, once procedures are properly identified.

For these reasons, we have not chosen a hierarchical program representation consisting of a call graph of procedure CFGs. We have opted instead for one *Interprocedural CFG* (ICFG) in which the nodes are the basic blocks, and edges model all possible execution paths, including all interprocedural control flow. Procedures in this graph are nothing more than a partition of basic blocks. To determine which basic blocks belong to which procedure, one can use several heuristics. The simplest heuristic considers targets of direct or indirect procedure calls as procedure entry points, together with the procedure entry addresses mentioned in the symbol information of the object files. It is then assumed that each procedure consists of the sequence of instructions from its entry point up to the instruction preceding the entry point of the next procedure.

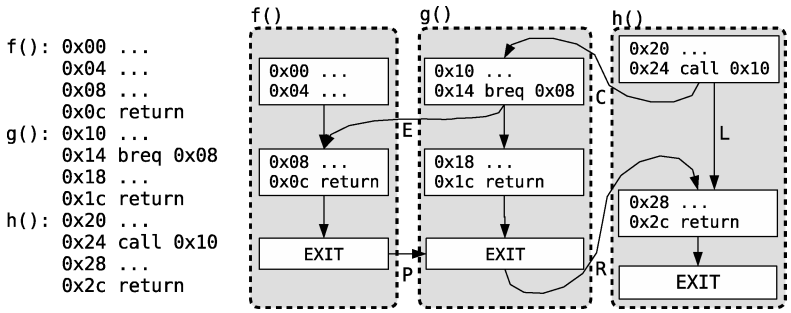

Fig. 4. On the left, the relevant control flow transfers in a fragment of assembler code are shown. This fragment consists of three procedures: `f()`, `g()`, and `h()`. On the right, the corresponding ICFG is shown, including three exit blocks, a call edge C, its corresponding return edge R, an escaping edge E, and its compensating edge P.

To ease the conservative, yet efficient, effective, and elegant modeling of all possible interprocedural execution paths in an ICFG, we add both virtual edges and virtual nodes. First, every procedure gets an *exit node*: an empty basic block that serves as a unique sink node of its procedure. All blocks ending with a return instruction are connected to the exit node of their procedure, as in the three procedures depicted in Figure 4. Procedure calls are then modeled with a call edge (C) connecting the call-site with the entry-point of the callee, and with a return edge (R) connecting the exit block of the callee with the return point in the caller.

Link edges (L) are added to connect call-sites with the corresponding return points following the call instructions. These link edges are added to the ICFG because they ease the modeling of a procedure call as an atomic operation during the analysis of a caller. Another frequently used term for such edges is *summary edges*.

Finally, *compensating edges* (P) are added to the ICFG to model execution paths that involve so-called *escaping edges* (E). Escaping edges model interprocedural control flow transfers other than calls and returns. In the program code depicted in the left-hand side of Figure 4, one possible execution path consists of the instructions at the addresses 0x24, 0x28, 0x10, 0x14, 0x08, 0x0c, 0x28, and 0x2c. By adding the escaping edge E and the compensating edge P to the ICFG, this execution path is modeled in the ICFG in Figure 4. Compared to adding an edge from the exit block of `f()` to the exit block of `h()` to model the execution path, the use of a compensating edge offers the advantage that there is a one-to-one correspondence between escaping edges and compensating edges, just like one return edge always corresponds to one call edge in the ICFG. This elegant, one-to-one correspondence for all interprocedural control flow simplifies the implementation of interprocedural analyses significantly.

It is important to note that, with the appropriate compensating edges, correctness is not affected by the assignment of basic blocks to procedures. In particular, it does not matter whether or not the partitioning of basic blocks into procedures actually reflects the original procedures from which the code was generated. All analyses and transformations will be conservative

independent of the partitioning. In fact, when partitioning basic blocks into procedures, more advanced heuristics do not try to approximate the procedures in a program. They do, however, try to minimize the number of compensating edges. Most often, but not always, this partitioning reflects the actual procedures in the program.

One problem that we have not addressed as yet is the detection of "return" instructions. On architectures with a program counter that can be accessed as an ordinary register (such as the ARM architecture), returns are implemented by copying the return address into the program counter. This is the same manner as any indirect procedure call is implemented by coping a procedure pointer into the program counter. The problem then is to correctly differentiate returns from other indirect control flow transfers.

We have found that on such architectures, heuristics suffice to detect most, if not all, returns correctly. Most modern processors use a return address predictor that requires easily recognizable return statements. Not to mislead to predictor on the architectures in question, compilers generate code that uses fixed locations to store return addresses, even if this is not required by the calling conventions or for program correctness. When this location is copied into the program counter, one can almost always be certain that the copy operation is a procedure return, and apply heuristics to further analyze the code. When one cannot be certain, a potential return can be modeled conservatively by an interprocedural edge to the *unknown* node and a corresponding compensating edge. This is discussed below.

3.5 The Unknown Node and the Unknown Procedure

Connecting all indirect control flow transfers with all instructions at relocatable code addresses results in a huge number of additional edges in the ICFG. Besides being inelegant, this makes the whole-program analyses inefficient.

To avoid this, we include a so-called *unknown node* in the ICFG that constitutes the *unknown procedure*. This node and procedure model unknown code, and resemble the so-called *extern node* [Chang et al. 1992] that models (unknown) external library code. Consider part of an ICFG as depicted in Figure 5. The callees of indirect procedure calls like the one in `caller()` are unknown, and therefore the unknown procedure is the callee of such indirect call-sites. And because procedures with relocatable entry points, such as `callee()`, might have unknown callers, they become callees of the unknown node.

The most important advantage of using an unknown node is that it reduces the quadratic number of indirect control flow edges to a linear number of edges.

Moreover, the unknown node makes an elegant and efficient implementation possible for dealing with unknown code in all kinds of analyses and program transformations. After having added the necessary edges to and from the unknown node, as in Figure 5, it suffices let the unknown node model the worst-case scenario for all possible analyses. For liveness analysis, for example, it suffices to initially mark enough registers as live on entry to the unknown node, after which the fix-point computations can be applied on the unknown node just like on any regular node.

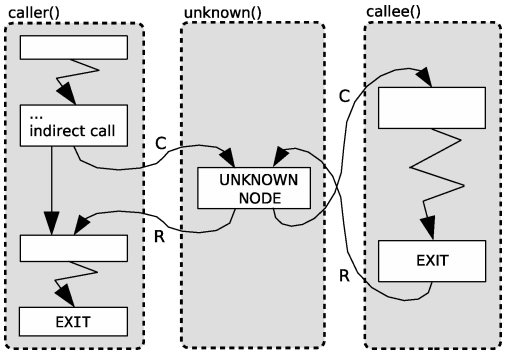

Fig. 5. Part of an example ICFG to illustrate the use of the unknown node and procedure.

As we will later see in Sections 4.1 and 4.2, the unknown node also allows the easy exploitation of calling convention information that we can get from the symbol information discussed in Section 2.4.2.

Please note that each outgoing call edge of the unknown node corresponds to a relocatable address stored or computed in the program. If we can, at some point during the analyses of the program, determine precisely how a relocatable address is used in the program, we may be able to replace its corresponding edge with less conservative edges in the ICFG. To enable this, we need to associate each relocatable address with both the entity that produces the relocatable address (that is, the instruction that computes the address, or the memory location where the address is stored), and with one "unknown edge."

4. WHOLE-PROGRAM ANALYSIS AND OPTIMIZATION

Once an initial conservative ICFG is constructed, we can start compacting the program. The compaction techniques that we apply at link time can be divided in two groups. On the one hand, *whole-program optimization techniques* eliminate superfluous code and data that resulted from the separate compilation issues discussed in Sections 2.2.1 through 2.2.3. *Duplication elimination techniques*, on the other hand, eliminate duplicated code and data (see Section 2.2.4). The latter techniques are discussed in Section 5. This section focuses on the optimization techniques and in particular on the underlying whole-program analyses.

We will not go through all the details of the analyses and optimizations we have implemented and evaluated. They are extensively discussed by Debray et al. [2000] and De Sutter et al. [2001]. In this section, we focus on the differences with compile-time analyses and on how the information discussed in Sections 2 and 3 can be exploited effectively. We also discuss how the ICFG can be refined using the information collected by the analyses. At the end of this section, we point out some effective ways to speed up the whole-program analyses. This is important for scalability and practical applicability of link-time compaction.

To overcome the limited scope of program analyses and optimizations at compile time (see Section 2.2.1), we perform roughly the same analyses at link time.

We have the advantage of performing these analyses on the whole program, albeit on a low-level internal representation. These whole-program analyses include liveness analysis to determine which registers hold useful contents, constant propagation to determine which registers hold constant values, and unreachable code detection [Aho et al. 1986] to detect code that can never be executed.

4.1 Liveness Analysis

Liveness analysis is used, first of all, to overcome the overhead resulting from overly conservative calling conventions. By detecting which registers hold live contents (i.e., contents that might be used later on during program execution) at intermodular control flow transfers, we can detect which parts of the calling conventions that the compiler had to assume necessary are in fact unnecessary, and optimize the code accordingly. Liveness analysis is also used to detect and eliminate instructions that have become useless after other transformations have been applied. Finally, liveness analysis is used to find free registers (i.e., registers that hold no useful contents) to store temporary values in order to enable other program transformations.

To help us understand the requirements of a whole-program liveness analysis, we make the following observations about procedure `f()` in Figure 3:

—Each caller of `f()` needs to store the argument to `f()` in register `r0`. In other words, `r0` is live at each call-site of `f()`.

—Register `r1` seems to be live on entry to `f()`, as `r1` is consumed by the second instruction in `f()`. Therefore, we can infer that `r1` is live at each call-site of `f()`.

—However, whether or not the contents of `r1` really is live at a call-site of `f()` does not depend on `f()`, since no computation in `f()` depends on the value of `r1`. Procedure `f()` merely spills `r1` to maintain calling conventions. Consequently, register `r1` is live at a call-site of `f()` if, and only if, `r1` is live at the corresponding return point.

Several fix-point solutions for liveness analysis have been developed. These reflect the third observation by propagating liveness information of callee-saved registers back over link edges instead of over call edges. The version described by Muth [1999] is the most advanced, and it is this context-sensitive backward data flow analysis that is implemented in our prototype link-time compactor.

The accuracy of this liveness analysis depends on the accuracy with which we can determine which registers are spilled onto the stack by each procedure. One way to determine these registers is an analysis that inspects the code of each procedure for stack saves and restores. This analysis is feasible but difficult, because clever compilers try to move the register stores and restores away from the procedure entry and exit points into the procedure bodies.

Fortunately, the need for a very accurate analysis by code inspection can be eliminated, to a large extent, by looking at the unknown node and the available symbol information. In Section 2.4.2 we noted that a compiler-generated global procedure adheres to the calling conventions. These conventions define, among

others, the callee-saved registers, of which a callee must assume that they contain useful contents (for a caller), and of which it must guarantee that its contents are unchanged. In the initial ICFG, we can therefore safely assume that the callee-saved registers are unused and unchanged by global procedures.

Since getting rid of the adherence to overly conservative calling conventions is one of the main goals of whole-program optimization, this adherence to calling conventions may no longer hold after we have applied some optimizations. Can we, in such cases, still rely on the symbol information? If we make sure that no program transformation invalidates this property, the answer is yes. One option would be take this restriction into account during the implementation of every transformation. Because this would be very cumbersome and error-prone, this option is not viable.

A much simpler solution involves the use of the unknown node. The conservative properties of the unknown node by themselves guarantee that none of the unknown node's callees will ever be transformed from a calling-convention maintaining procedure into a calling-convention disregarding procedure. We can therefore safely conclude that for procedures that (1) are defined by a symbol and (2) are callees of the unknown node, all callee-saved registers will be unused and unchanged. For liveness analysis, such registers can be propagated over link edges instead of over call edges.

4.2 Constant Propagation

We rely on constant propagation to solve the problem of unknown addresses, as discussed in Section 2.2.3. In compilers, constant propagation is a forward data-flow analysis to detect expressions that evaluate to constant numerical values and detect and optimize expressions that depend on those values.

At link time, the constant values propagated over the ICFG of the program during constant propagation are not limited to numerical constants. Instead, constant (data) addresses are also propagated, and the instructions consuming and producing the constant addresses are optimized.

4.2.1 *Context-Sensitiveness.* Just like link-time liveness analysis, link-time constant propagation is performed on registers instead of on variables. No information is propagated through memory, and therefore the same problems with callee-saved registers arise as in link-time liveness analysis. Looking at the machine code of procedure `f()` in Figure 3 again, we observe that, even though the a value is loaded into register `r1` just before the return instruction, the content of `r1` after the return from a call to `f()` does not depend on `f()`. Instead its new content is identical to its content prior to the call to `f()`. As we described for liveness analyses, we can also use the symbol information and the unknown node to determine which registers remain unchanged by a callee at some call-site. And just as we propagated the liveness information for those registers backward over the link edge instead of over the call edge, we propagate constant register contents of callee-saved registers forward over the link edges instead of over the return edges.

The net result is that the constant propagation is context-sensitive with respect to the callee-saved registers, even though it is otherwise completely

context-insensitive. This is very beneficial, since in practice it results in an analysis that is almost as precise as a more generally context-sensitive propagation without requiring the computation time or space of the latter.

4.2.2 *Differences with Compile-Time Constant Propagation.* While link-time constant propagation is conceptually similar to compile-time constant propagation, there are some significant differences. Most importantly, at compile time, constant values are propagated from producers to consumers. Therefore the propagated values are by definition live.

At link time, we like to know all constant values stored in registers, whether they are live at some point or not. An example of the usefulness of propagating dead values can be seen in the code generated for procedure `f()` in the executable a.out in Figure 3. As a first step in the compaction of procedure `f()`, we want to replace the load of `b`'s address from the GOT by an addition that computes `b`'s address out of `a`'s address. In order to enable this optimization, constant propagation needs to detect that `a`'s address is available in `r2` at the point where `b`'s address is originally loaded. This detection is implemented by propagating `a`'s address in `r2` to the instruction that loads `b`'s address, even though `r2` is clearly dead just prior to that instruction.

Another primary difference with compile-time constant propagation is that constant code and data addresses can also be propagated at link time. This potential offers both an interesting phase-ordering problem, as well as some important compaction opportunities. Starting with the compaction opportunities, propagating constant addresses allows (1) the optimization of the generation and use of addresses, just like the generation and use of other numerical constants can be optimized and (2) the detection of which statically allocated data can be accessed in the program.

4.2.3 *Detection of Inaccessible Data.* As indicated at the end of Section 2.4.1, an object data section is accessible if, and only if, a relocatable address pointing to the section may be used in the program. Since relocatable addresses are constant in the linked program, constant propagation of constant addresses can be used to detect which relocatable addresses are used in the program. Hence constant propagation can be used to detect which statically allocated data can be accessed by a program. When an address is used in a load instruction, the data at that address is accessible, and when an address is used in a store instruction, the data at that address is nonconstant or mutable.

Where constant propagation cannot keep track of how an address is used, worst-case assumptions need to be made. This is, for example, the case when an address is stored in memory, since constants are not propagated through memory. Another case occurs when a register can hold more than one constant address, as those addresses are not propagated separately.

Fortunately, because the compiler cannot have generated code that accesses data in multiple data sections from one constant base data address (unless the relocation information tells us differently), all worst-case assumptions that need to be made are always restricted to one object section. If constant propagation cannot keep track of a relocatable, constant address, it suffices to

consider the whole object section to which the base address points accessible and mutable. In practice, this granularity of object sections on which worst-case assumptions need to be made has proven adequate to detect significant amounts of inaccessible data.

Without going into further details (refer to De Sutter et al. [2001]), it is important to note that knowing which absolute address is used at some point is by itself not sufficient to know which object section will be accessed with the address. Consider the final program a.out in Figure 3 again. The absolute address used to access array `R` in procedure `f()` is not `R`'s starting address `&R`, but `&R-5` = 0x57. It is this 0x57 that is used in the subtraction instruction at final address 0x20. Since an unknown value is at that point subtracted from the constant address 0x57, constant propagation loses track of the use of this address, and worst-case assumptions need to be made about its object section. This cannot be accomplished by looking at the value of the address being used however: 0x57 is not in A.o's `.data` section in the final program, which ranges from 0x5c to 0x67. Instead, the relocation associated with that 0x57 needs to be used to identify the object section that will be accessed with it. In the example, this is relocation RB1. Through symbol `SB1`, this relocation indeed points to a.o's `.data` section.

It is important that the relocations, rather than the corresponding constant addresses themselves, need to be used. This implies that this analysis is applicable at link time only, and not post link time. Even if a linked executable program is still relocatable because the linker provided relocation information in it, and even if a so called *link-map* is generated to distinguish the different object data sections that were combined into the program data sections, there is no way to tell which of the original object data sections will be accessed with which addresses. That information is only available in the original object files.

4.2.4 *A Phase-Ordering Problem.* The phase-ordering problem we mentioned at the end of Section 4.2.2 relates to the propagation of addresses. On the one hand, the propagation of constant addresses during constant propagation obviously requires that the code and data addresses that appear in the program be constant. On the other hand, using the results of the constant propagation to eliminate instructions or data to compact the program requires the remaining instructions or data to be moved, thus changing their addresses. In short, we want to change the addresses of code and data using the results of an analysis that requires them to be constant.

While this problem may at first sight seem similar to the compile-time phase-ordering problem of span-dependent branches, it is not at all similar. Whereas the optimization of span-dependent branches by Szymanski [1978] involves only part of the instructions of a program (i.e., the branches), address computation optimizations at link time can result in free registers, because of which many other optimization opportunities may be created for the surrounding code. Vice versa, other optimizations may result in free registers from which address computation optimizations can profit. Hence, it is impossible to separate the address computation optimizations from the other program optimizations in such

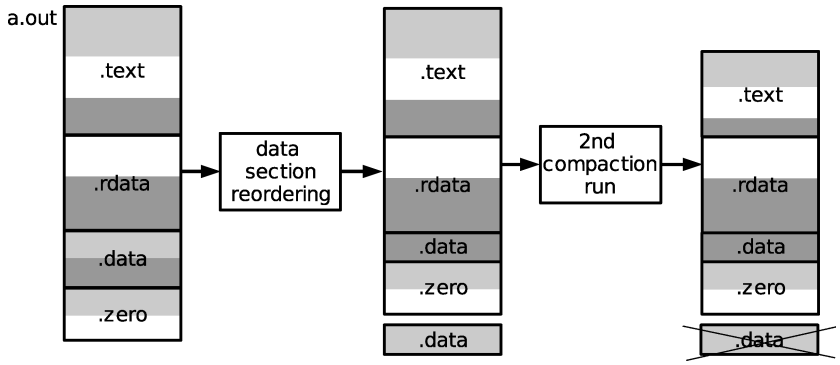

Fig. 6. Object data sections are reordered to separate inaccessible and accessible data before the second pass of the compaction begins.

a way that an optimization similar to Szymanksi's technique for optimizing span-dependent branches would be useful.

For code addresses, our solution to this phase-ordering problem is simple: we do not modify any instructions that compute code addresses. Since we assume that all code address computations in the original program are relocatable, the original relocations will tell us how code address computations need to be adapted after the final compacted program has been assembled again. This theoretically conservative solution of not optimizing any code address computation does not limit our compaction in practice, as programs rarely contain code address computations.

For data addresses, whose computations are one of the most important optimization targets in a link-time compactor, the same approach is not feasible. Instead, we have chosen to compact a program with two runs of our link-time compactor.

During the first run, no object data sections are eliminated from the program. Consequently, all code analyses and optimizations in that first run can exploit the fact that all data addresses are constant. After the first run, a temporary file is produced that describes which object data sections are inaccessible. Note that this is not limited to data sections that were inaccessible in the original program. Object data sections that were accessed in the original program might have become inaccessible after the first compaction run. This happens, for example, with parts of the GOT that are no longer accessed after the optimization of indirect data accesses.

Before the second compaction run, the compactor first reads the temporary file to reorder the data sections in the program. The new layout is such that all data determined inaccessible after the first run is positioned after the data determined accessible after the first run. An example is depicted in Figure 6. If we suppose that the first compaction run was executed on the program linked in Figure 2, and that is was determined that the `.data` section of `obj1.o` was inaccessible, the compactor will relayout the data sections at the beginning of the second run as depicted in the middle of Figure 6.

As a result, eliminating the inaccessible data from the program during the second compaction run no longer requires moving the remaining data. During the second run, data addresses can be treated as constants and it suffices not to include the inaccessible data in the binary written out after the second compaction run.[6]

We should point out that, in theory, there is no need for two full compaction runs. A first alternative would be a compactor with two phases: one that considers data addresses as not final, and one that considers data addresses as final. Obviously the implementation of program analyses and transformations that need to be run in both phases then becomes more complex.

A second alternative is to treat only the intrasection relative addresses as constant values, both during a first and a second compaction phase. During the first phase, all compaction techniques treat the intersection relative addresses as unknown values. Inaccessible object sections are eliminated and the remaining ones are placed next to each other. After the first phase, all remaining object sections are combined into one data section with fixed layout. Hence all intersection relative addresses from the first phase have become intrasection relative addresses in the second phase. The exact same compaction techniques can be applied to these as in the first phase. This two-phases scheme offers the advantage, as does our two-runs scheme, that the program analyses and transformations are identical in both phases. But this two-phases scheme cannot easily eliminate redundant addresses from the GOT, as these will remain accessible throughout the first phase.

4.3 Unreachable Code Detection

Unreachable code detection detects code that will never be executed, because no execution path leads to it. Basic unreachable code elimination is very simple [Aho et al. 1986]. Mark the entry point of a program, and iteratively mark all successors of marked program points. After this iterative process has converged, all unmarked blocks can be eliminated from the program, together with their predecessor and successor edges.

Wegman and Zadeck [1991] demonstrated that combining unreachable code detection and constant propagation as a single optimization performs better than separate constant propagation and unreachable code elimination optimizations at compilation time. We found this to be true at link time as well. So the constant propagator we implemented is a conditional constant propagator. If the condition of a conditional branch evaluates to some constant during constant propagation (i.e., we know the outcome of the conditional branch), we only propagate information over the corresponding edge, the branch-taken edge, or the fall-through edge. Once the fix-point computations of constant propagation have converged, conditional branches that were only evaluated in one direction are converted to unconditional control flow, and the never-taken edge is removed

[6]Note that, on most platforms, code and data are not stored immediately after each other in the memory space. Therefore changing the code addresses during the second compaction pass does not result in changed data addresses.

from the ICFG. If this results in basic blocks with no incoming edges, these will be eliminated during a subsequent, simple unreachable code elimination.

A similar reasoning as for never-taken conditional edges holds for edges from the unknown node to other basic blocks. Remember from Section 3.5 that such edges were initially added to the ICFG for relocatable code addresses stored in the statically allocated data or encoded in the immediate operands of instructions in the program, and that there is a one-to-one correspondence between the relocatable addresses and these edges. As long as the locations at which such code addresses are stored have not become accessible (in case of addresses stored in the data) or reachable (for instructions computing the code addresses) during constant propagation, there is no need to propagate over the corresponding edges from the unknown node.

And just as never-taken edges following conditional branches are removed from the ICFG, so too are edges from the unknown node over which no data ever needed to be propagated. As such, the detection of inaccessible data is not only useful to eliminate it from the program (during the second compaction run), but also to eliminate unreachable code of which we otherwise had to assume it was reachable through indirect control flow transfers. This form of unreachable code elimination proved to be particularly important for object-oriented programs, in which polymorphic method calls are implemented as indirect procedure calls.

4.4 Whole-Program Optimization

The information gathered by the liveness analysis, constant propagation, and unreachable code detection is consumed by several optimizations.

Among the optimizations that proved useful at link time are [Aho et al. 1986]: useless code elimination, unreachable code elimination, branch forwarding, constant folding, load/store avoidance, strength-reduction, peephole optimizations, copy propagation, copy elimination, inlining (of small procedures and of procedures with one call-site only), code hoisting and sinking (for example, when duplicated code can be hoisted or sunk to a common predecessor or successor block in the ICFG).

It is important to note that the whole-program overview of a link-time compactor is not the only reason why these optimizations are useful. Often these optimizations build on information that was available at compile time, but was not exploitable because of a lack of resources such as free registers. At link time, liveness analysis not only detects more free registers, but other optimizations also create more free registers. A typical use of constant folding, for example, is to encode small constant operands as immediate operands in instructions, thus avoiding the need to store them in a register. When more registers become available as a result, this often creates new opportunities for load/store avoidance, for copy propagation, and for other optimizations.

4.5 ICFG Refinement

Based on the results of conditional constant propagation, inaccessible data detection, and unreachable code detection, unrealizable edges are removed from

```
typedef struct
{
   unsigned int        Flags, AuxFlags; // Bitvector
   unsigned char       Type, Alignment; // Bitvector
   unsigned long       LiveRegistersOut; // Bitvector
   unsigned long       Code; // binary representation
   unsigned int        HashLink; // ID of next block in same hash table bucket
   unsigned int        InsFirst,InsLast,Next,Prev,Pred,Succ,Function;
                       // IDs of related CFG elements
   unsigned int        Master,PreDom,PostDom,LoopHeader,LoopNext,LoopFirst;
                       // IDs of dominator-related CFG elements
   struct VALUE_SPEC   ConstantsIn; // incoming constant values
} STRUCT_BBL;
```

Fig. 7. Original C-structure used to collect information about a basic block.

the ICFG, thus making it less conservative. As a result, unreachable code may be eliminated, and subsequent analyses may collect more accurate information.

Using the constant propagation results, we can to some extent also get rid of indirect calls that are modeled by an edge to the unknown node. If it turns out that an indirect procedure call has a known callee (i.e., there is a constant target address), it suffices to replace the indirect call instruction by a direct one and to replace the corresponding call and return edges to and from the unknown node by call and return edges to and from the actual callee.

Indirect jumps other than calls are also initially modeled with an edge to the unknown node. Most such indirect jumps in compiler generated code originate from switch-like statements in the source code. These statements are usually implemented with more or less fixed instruction sequences. A typical sequence involves normalization of the index of the switch-statement, a bounds check, the load of the actual target code address from some target address table using the normalized index, and the actual jump. Of course, this sequence might be scheduled around and in between other instructions, including procedure calls. By pattern matching different versions of this scheme with the program slice of the indirect jump (i.e., the instructions affecting the outcome of the jump), it is often possible to extract the location of the target address table and the number of elements in that table. At that point, the edge to the unknown node can be replaced by edges to all the possible targets of the jump. This is discussed in more detail by De Sutter et al. [2000] and Kästner and Wilhelm [2002].

4.6 Implementing Efficient Analyses

The ICFGs on which the whole-program analyses and optimizations are applied often contain several hundred thousand nodes, whose combined data structures do not fit in the caches of a memory hierarchy. It is therefore no surprise that data locality is an important implementation issue affecting compaction time.

One obvious way to optimize spatial data locality is to restructure the data collected about a program. A particularly interesting technique proves to be data remapping [Palem et al. 2002]. Take, for example, the information we collect about basic blocks. All the information collected about a block was originally stored in one structure like the one depicted in Figure 7, and we used one huge array of these structures to gather the data about all basic blocks.

While such a simple array enables simple and fast indexing to access the collected data, thus easing our programming task, its structure nonetheless

```
/* some blocks are initially marked */
while ((block = getMarkedBlock()) != -1) {
  ...  /* do something with block */
  if (...)
    markBlock(relatedBlock);
}
```

Fig. 8. Naive implementation of a fix-point computation.

```
/* some blocks and their procedures are initially marked */
while ((proc = getMarkedProc()) != -1) {
  while ((block = procGetMarkedBlock(proc)) != -1) {
    ...  /* do something with the block */
    if (...)
      markBlock(relatedBlock);
      markProc(relatedProc);
  }
}
```

Fig. 9. Implementation of a fix-point computation exploiting temporal locality.

proves to be very bad for performance. Each subsequent analysis, going over the hundreds of thousands of basic blocks, only accesses a small number of members from the depicted C-structure. However, when one member of the C-structure is loaded into the cache, so are its neighboring members, whether they are needed by the analysis or not. The result is a very low cache line utilization, and because each subsequent analysis accesses different members, reordering the members of the structures can never completely avoid the resulting cache pollution.

The solution to this problem is the replacement of an array of structures by a structure of arrays, of which each array stores one original structure member for all basic blocks. As a result, only the members needed by an analysis are loaded into the cache. This data remapping of the basic block data and of the edge data structures results in a speedup by a factor of up to 2.5.

Besides restructuring the data to avoid underutilization of cache lines, temporal locality should also be exploited as much as possible. Since most of the analyses are iterative fix-point computations, increasing the temporal locality is achieved by changing the order of the iterations. At first, our iterative analyses mostly worked as depicted in Figure 8. In each iteration, all basic blocks in the program for which recomputation is necessary are visited.

This loop obviously has little temporal locality. For most analyses, huge speedups were achieved by simply converting the single iteration loop into two nested loops, as depicted in Figure 9. The outer loop iterates over procedures that need recomputation. In the inner loop, local convergence is first achieved within one procedure. Because of the nested loops, basic blocks are revisited much more quickly, when chances are higher that their data is still cached. For context-sensitive liveness analysis, for example, we noticed a speedup by a factor of 20 for larger test programs.

With respect to the iteration order in the inner loop, our experience was that the optimal theoretical order is most often not optimal in practice. Instead, we experienced that most iterative analysis performed best when a stack was used to push and pop elements for which recomputation was needed. The LIFO

order of the stack improved temporal locality, and, more importantly in light of the already high stress on the caches, a minimal amount of memory was required to store the set of elements that needed recomputation. Overall, it proved beneficial to minimize the execution time per iteration by using a stack, instead of optimizing the number of iterations.

Finally, we should mention an important use of the exit nodes of procedures and of the mapping between incoming interprocedural edges and outgoing interprocedural edges. Recall that each call edge has a corresponding return edge in the ICFG, and that each escaping edge has a corresponding compensating edge. To implement context-sensitive analysis efficiently, this one-to-one mapping must be stored directly, because one most often propagates data-flow information from the head of the incoming edge to the tail of the outgoing edge.

To iterate over the outgoing interprocedural edges of a procedure, it suffices to iterate over the successor edges of the procedure's exit node: those edges are the return and compensating edges exiting a procedure. More importantly, we can iterate over the incoming edges of a procedure by iterating over the successor edges of the exit node, and taking their corresponding edges. Thus, we only need to interate over one block's predecessor edges.

In the presence of escaping edges and procedures with multiple entry points that would otherwise complicate the implementation of interprocedural analyses, this unusual use of the exit node has proven to be both elegant and efficient. Apart from storing the corresponding edge for each edge, which is necessary anyway for efficiently implementing context-sensitive analyses, there is no need to store and maintain additional information (such as a list of incoming edges per procedure, or whether or not an edge is interprocedural).

5. DUPLICATE CODE AND DATA ELIMINATION TECHNIQUES

As briefly discussed in Section 2.2.4, separately compiled programs may contain duplicate code fragments. Furthermore, if some global code or data is accessed by multiple source files, then each of their corresponding object files will contain a placeholder for that code or data. As a result, a lot of addresses end up multiple times in the data of a final program, and in particular in the GOT. Other data that typically occurs multiple times in an executable are the numeric constants that appear in the source code and that are cheaper to load than to compute, such as floating-point values or wide bit masks. Such constants are stored in the read-only object data of all modules in which code uses them. This section discusses techniques to eliminate duplicate code and duplicate data.

5.1 Duplicate Code Elimination

To remove duplicate code at link time, the technique commonly referred to as *code abstraction* is used. Multiple occurring code fragments are replaced by one procedure implementing the code sequence, and the original occurrences are replaced by calls to this procedure. Often code abstraction techniques are not limited to identical multiple occurring code fragments. Some techniques are able to abstract functionally equivalent but different code fragments, while

other techniques are able to detect similar fragments that can be abstracted only after some abstraction-enabling code transformations are applied.

This section will not delve into the technical details of code abstraction techniques. Readers interested in specific link-time techniques are invited to read Debray et al.'s [2000] or De Sutter's et al.'s [2002] work. This section instead focuses on how to engineer scalable techniques. Considering that the programs we want to compact consist of up to hundreds of thousands of basic blocks and millions of instructions, any link-time code abstraction technique should scale very well.

Conceptually code abstraction techniques consist of three steps: the detection of identical or similar fragments, the application of abstraction-enabling transformations, and the actual abstraction. Since the last two steps are transformations on relatively small code fragments, the difficulties with respect to scalability lie primarily in the detection of identical or similar fragments.

A very natural approach to detect identical or similar code fragments is the bottom-up approach: starting with pairs or groups of identical code fragments consisting of one instruction only, we iteratively detect larger fragments by growing the existing identical (or similar) fragments. In the software engineering community, several very powerful bottom-up techniques for duplicated code detection have been developed. They are used to detect code that has been duplicated by programmers with copy&paste(&edit), which frequently occurs during the development of large software projects, but is considered bad practice because of its detrimental impact on software maintainability.

The best known bottom-up approaches [Krinke 2001; Komondoor and Horwitz 2001] build on program dependency graphs and program slices. Unfortunately, these bottom-up approaches do not scale well at all. Krinke [2001] reported detection times ranging from 0.6 s to more than 48 h for programs ranging from 2,402 to 24,950 lines of source code. Komondoor and Horwitz [2001] similarly reported detection times ranging from 40 s for a program of 1,569 lines of source code, to 93 min for a program of 11,520 source code lines. Clearly such detection times are not feasible in compiler tool chains. To the best of our understanding, this nonscalability is a fundamental problem of bottom-up approaches, as they are specifically designed to explore extremely large search spaces, which typically are much larger than what is useful for automated code abstraction.

We strongly believe that the key point of making code abstraction at link time scalable lies in limiting the search space. This is feasible because many of the potentially duplicated code fragments are too irregular to be abstracted into procedures, thus making it useless to detect them. Also, many potentially duplicated code fragments occur very infrequently, making it not worthwhile to try to detect them. Finally, as is common for problems with large search spaces, divide-and-conquer approaches are useful. We have found it particularly useful to engineer different abstraction techniques for different types of program fragments, such as procedures, basic blocks, instruction sequences, etc. Not only does this divide the search space, but it also allows us to conquer each subspace with specific techniques.

5.1.1 *Duplicate Procedure Elimination.* Duplicate procedures, either fully or partially identical, mainly result from two programming techniques: copy&paste by programmers, and the use of templates to reuse code. The latter is frequently done in C++ programs, but is not limited to that or similar languages. Any rapid prototype programming environment, for example, generates large amounts of code of which the skeleton is based on templates.

In C++ programs, it often occurs that different template instantiations at the source code level result in identical instantiations at the object code level. At the source code level, pointers to different types are themselves different types, but at the assembly level, all pointers are simple addresses. And even if the generated code is not identical, for example, because pointer arithmetic in the instantiations depends on the size of the types the template is instantiated for, the generated code will still have the same structure and show only local differences. A more extensive discussion of such local differences is given by De Sutter et al. [2002]. Code generated from code skeletons, and copied&pasted code that was later edited, also often show similar structure with local differences only.

Furthermore, procedures are the typical scope with which compilers optimize and generate code, and compilers usually do so very deterministically. So there is no reason to expect big differences in code schedules, register allocations, or code layout for procedures that have the same structure and only very local differences.

Therefore we can limit the detection of similar or identical procedures to procedures with identical code structures, or, in other words, with identical CFGs. Since the compactor is deterministic in its CFG creation, we can use a direct comparison of the procedure CFGs, not requiring a complex detection of different but isomorphic graphs. In order to do this efficiently and to reduce the number of pairwise comparisons between procedures we need to perform, we use a fingerprinting scheme to prepartition all procedures. For each procedure, a deterministic depth-first traversal is used to build a string of characters that identifies the basic blocks in the procedure and their types: 'C' stands for a block ending with a call, 'B' for a block ending with a branch, etc. Procedures with identical fingerprints are put in the same partition.

Inside each partition, a more accurate pairwise comparison of procedures is then performed. It is important to note that this pairwise comparison is very simple. Because of the deterministic way in which compilers generate code schedules and register allocations for the procedures we target, the pairwise comparisons do not need to handle differences in code schedules or register allocations. Instead counting the number of identical pairs of corresponding instructions in two procedures suffices to measure how similar two procedures (of which we already know that they have the same structure) are.

If multiple *identical* procedures are discovered this way, we can simply remove all but one of them, and convert all their call-sites to call the one remaining copy. If two or more *similar* procedures are detected, the differences will be local only. In this case the identical code in the procedures can be eliminated by creating a new procedure that merges all code from the original procedures as follows. All identical code from the original procedures occurs once in the new procedure, as does all code that was not identical and appeared in only some of

the original procedures. Wherever there was a local difference between the original procedures, the differing fragments in the merged procedure are preceded by a conditional branch (or a tree of branches). These branches control which of the original different fragments will be executed, and they do so by testing an additional parameter of the merged procedure that identifies the call-site from which it was called. The original similar procedures are simply replaced by code that sets the parameter and calls the merged procedure. De Sutter et al. [2002] discussed in detail how one can add an additional parameter to the merged procedure at link time.

5.1.2 *Duplicate Code Region Elimination.* A code region is a group of basic blocks with a single entry point and a single exit point. They seem at first sight to be good candidates for code abstraction. However, detecting identical or similar code regions is not as simple as detecting identical or similar procedures.

First of all, the search space is larger: there are far more code regions in a program than procedures. Moreover, the code schedules and register allocations inside code regions that implement identical or similar source code will likely show much more variation than the variation seen in identical or very similar procedures, since these properties are typically influenced by the nonsimilar code surrounding the regions. Fewer functionally equivalent regions will be identical or mostly identical. So on top of having to explore a larger search space, we will also need a more complex comparison to detect whether or not some regions are functionally equivalent.

Furthermore, the abstraction of code regions is complicated by the fact that a place has to be found to store the return address of the call to the abstracted code. Unlike whole-procedures, there is no one-to-one correspondence between stack frames and code regions, making it difficult to find such space on the stack. Finding a free register to store the return address is not trivial either. First, if the functionally equivalent regions contain procedure calls, it has to be checked whether the abstracted procedure would be reentrant. If it would, no register can be used. If it would not be reentrant, a free register actually has to be found. While this is not complex, it is relatively time-consuming. For example, for each of the equivalent regions, the sets of live and defined registers have to be computed. This would not be problematic, were it not that the chances for finding a free register are relatively small because of the rather large variation in register allocation in functionally equivalent regions. Our experience to this date is that the relatively time-consuming search with relatively small success rate has not proven worthwhile.

For these reasons, we have not yet found a scalable technique to detect abstractable, nontrivial code regions. From examining compiler-generated code fragments, we do know that there definitely are code regions that can be factored out, but finding a cost-effective technique remains work for the future.

5.1.3 *Duplicate Basic Block Elimination.* Like code regions, basic blocks implementing the same computations can be expected to show some variation in code schedule and the registers used. And obviously there are even more basic blocks than code regions. But because basic blocks involve no control flow, the

detection of functional equivalence of two blocks is much simpler.[7] Moreover, because liveness information is already available per basic block, finding a free register to store the return address of an abstract basic block is much less time-consuming than finding a free register for an abstracted region. For these reasons, basic blocks have proven to be good abstraction candidates.

To detect whether or not two blocks perform the same computations, we compare their dependence graphs. These directed acyclic graphs (DAGs) consist of instructions and edges that connect producers with consumers. Comparing two dependence DAGs is straightforward, if the order of computations is assumed identical. A single iteration over all the instructions suffices. Furthermore, if we only try to detect functionally equivalent blocks in which all computations are performed in the same order, it is trivial to generate fingerprints to prepartition all the basic blocks as we did with procedures. In the case of blocks, the fingerprint consists of the number of instructions in a block and a concatenation of the opcodes of the instructions (excluding register operands).

In practice, we have experienced that limiting the detection of functionally equivalent blocks to blocks with identical instruction schedules does not significantly impact the number of abstracted blocks. Less than 0.5% of the abstracted blocks are no longer abstracted because of this simplification. This result corresponds with our experience that compilers most often generate identical schedules for identical DAGs. It must be said, however, that this may not be the case with more ambitious compiler schedulers. When global schedulers move code between basic blocks aggressively, basic blocks that originally were identical may become polluted with different code from different neighbors.

To test whether we can abstract basic blocks with the same fingerprints and the same DAG, but possibly different register operands, we try to rename the register operands in the blocks to each other. Again, a single iteration over the instructions of the blocks suffices. To rename one block to another, we simply insert copy operations before and after the block to be renamed. This was discussed in detail by Debray et al. [2000] and by De Sutter et al. [2002]. Renaming is considered successful if (1) it is possible under the existing register pressure, and (2) fewer register copies have to be inserted than what can be gained from abstracting the code.

Note that by inserting copy operations just before and after a renamed basic block, this renaming is local. Therefore, renaming one block does not influence possible renamings of other blocks in its neighborhood. Cooper and McIntosh [1999] proposed global register renaming, not requiring the addition of copy operations. We believe our approach to be at least as effective. First, more renaming will be performed in our approach, as it sometimes is applicable where no global register renaming is possible. Second, in cases where global renaming would perform better, a post-pass copy elimination step can eliminate the inserted copy operations. The complexity of doing this is no greater than that

[7]As abstracted basic blocks need to end with a return instruction, we limit abstraction to basic blocks that do not end with control flow transfers. In order to be able to abstract all basic blocks, it suffices to split all blocks ending with a control flow transfer into two separate blocks, of which the first contains all the computations.

of doing global register renaming, as copy elimination and register renaming are based on almost identical data flow analysis.

After renaming, identical basic blocks are abstracted. One copy forms the body of a new procedure, and calls thereto replace the original occurrences of the block.

5.1.4 *Duplicate Subblock Instruction Sequence Elimination.* Since the number of instruction sequences in a program is much larger than the number of basic blocks, the search space for equivalent subblock instruction sequences again is larger. And, as the code fragments become smaller, there will be less to gain from each group of abstracted subblock instruction sequences. As a result, one is justified in making compromises in the detection of equivalent instruction sequences.

We propose a compromise that needs very little implementation effort. We only try to abstract identical sequences. To detect them, a fingerprinting scheme like that for basic blocks is used, but now the fingerprints include the register operands. Using the fingerprints, we greedily select identical sequences, starting with the largest ones. The blocks in which they are found are split to generate identical basic blocks. Once the blocks have been split, we simply reapply the basic block reuse techniques. The number of identical subblock instruction sequences that can be found this way is relatively small, but so is the implementation effort.

5.1.5 *Duplicate Procedure Epilogue and Prologue Elimination.* Unlike general subblock instruction sequences, which are too small and of which there are too many to allow complex detection techniques, two kinds of instruction sequences occur so frequently, and in such fixed locations, that a special treatment is worthwhile. These sequences are the procedure prologues and epilogues that allocate and deallocate the stack frames, and store and restore the callee-saved registers. Because these registers are determined by calling conventions, all epilogues and prologues are very much alike, making them very good abstraction candidates. On the other hand, as these sequences mainly consist of loads and stores, compilers will most often try to hide their latency by scheduling them in between the instructions of the procedure body. Because of this, it has proven worthwhile to isolate the prologues and epilogues by rescheduling the entry blocks and the return blocks of procedures before trying to abstract them.

While the isolated sequences can then be abstracted just like basic blocks, one can often find more efficient ways of doing so, such as by combining the abstraction of procedure epilogues with tail-call optimization. Some clever schemes to do so were discussed by Debray et al. [2000].

5.1.6 *Cost of Duplicate Code Elimination.* Unlike most of the code optimizations we mentioned in Section 4, code abstraction techniques do not speed up a program, but instead slow it down because additional instructions will be executed. These additional executed instructions constitute the glue code needed to implement the code abstraction: the calls to the abstracted procedures, the return instructions, the register copy operations introduced because of register renaming, and the setting and testing of the additional parameters introduced when merging similar procedures.

Moreover, while abstraction is performed to reduce the size of a whole program, it does not imply that the instruction cache behavior improves. Reducing the size of the whole program indeed does not imply that the working set size is reduced. In practice, the working set size even increases [De Sutter et al. 2003]. This follows from the 90/10 rule that states that 90% of the executed instructions corresponds to 10% of the static instructions. Because of this rule, a hot (i.e., frequently executed) code fragment is much more likely abstracted together with a cold fragment than with another hot fragment. Whenever one hot fragment is abstracted with one or more cold fragments, the introduced overhead only adds to the hot working set, thus putting additional pressure on the instruction cache.

Fortunately, the performance degrading side effects of code abstraction can be avoided easily by using profile information and by limiting code abstraction to infrequently executed code. Because of the 90/10 rule, excluding the frequently executed code from abstraction is not detrimental for code size. Moreover, as programmers typically apply more optimization (and therefore more code specialization) to the frequently executed code, we are less likely to find duplicated code in it anyway.

5.2 Duplicate Data Elimination

As indicated at the beginning Section 5, data is frequently duplicated in programs as well. In the case of read-only data, this data can be eliminated by converting loads that load the same constant data from different locations into loads that load them from the same location. If this conversion results in whole object sections becoming inaccessible, they can be eliminated from the program. This is particularly interesting for compacting a contiguous GOT. As all loads from such a GOT are manifest direct loads using the global pointer, each entry in the GOT can be seen as a separate object section. Therefore, the elimination of inaccessible data from such a GOT can be performed at the granularity of single entries.[8]

Moreover, the elimination of duplicate data may also create new opportunities for code abstraction. Besides local differences resulting from different pointer arithmetic, different instantiations of the same template also differ locally because they use the same constant data, but load it from different locations. Such data includes initialization values, and procedure addresses that are loaded for polymorphic (indirect) procedure calls. Converting these loads from different locations to loads that load from identical locations reduces the number of local differences between procedures, resulting in more duplicate code elimination opportunities.

6. A LINK-TIME COMPACTION STRATEGY

This section presents an ordering in which to apply the analyses and transformations discussed in isolation in the previous sections. This ordering or strategy is implemented in our prototype link-time compactor SQUEEZE++.

[8]Early experiments [De Bus et al. 2004] indicated that this is also the case on architectures with distributed GOTs, such as the ARM architecture.

6.1 Dependencies Between the Analysis and Transformations

There are a number of obvious dependencies between the analyses and transformations discussed in the previous sections. First, program transformations rely on information gathered by the program analyses. Moreover, applying certain transformations may result in some analyses becoming more accurate, while other analyses will become less accurate. Finally, applying one transformation may create new opportunities for other transformations. A short discussion of the most important dependencies between the various analysis and transformations follows.

All program analyses benefit from the ICFG being refined and becoming less conservative. And as a result of the more accurate analyses, more aggressive program transformations can be applied. Since the ICFG refinement is built on top of the analyses, however, there is a clear circularity in the dependencies between refinement and analyses.

Of all analyses, liveness analysis is required for the largest number of transformations. Besides being necessary to find free registers to store the return addresses of abstracted procedures, transformations such as copy elimination also rely on liveness information. Furthermore, liveness analysis is useful for the detection of inaccessible data during constant propagation. As discussed in Section 4.2.3, an important part of this detection consists of making the necessary worst-case assumptions when the use of constant addresses can no longer be tracked by the constant propagator. Obviously, when some register becomes dead, there is no need to make worst-case assumptions about how the address in the register will be used: it won't be used. Using liveness information to exploit this observation speeds up the detection of inaccessible data, and improves its accuracy.

The two most important control flow transformations are inlining and duplicate code elimination. On the one hand, inlining is an effective way to overcome the limitations of the partially context-sensitive constant propagation. Therefore, inlining may have a positive influence on constant propagation. On the other hand, inlined procedures complicate the analysis of the stack behavior of a program, and thus may also negatively influence the data flow analyses. Furthermore, inlining complicates the elimination of large duplicated procedures, since inlined procedures are no longer separate procedures that are easily detected and compared.

Finally, code abstraction may result in less accurate partially context-sensitive analyses such as constant propagation, because code abstraction results in longer call chains. Moreover, constant propagation may suffer from the elimination of duplicated procedures because, after this elimination, different constants that were first propagated separately through separate equivalent procedures, are now propagated through one procedure, in which they are no longer a single constant.

6.2 Squeeze++

To exploit the positive dependencies maximally and to avoid negative dependencies between analyses and transformations, our link-time compactor

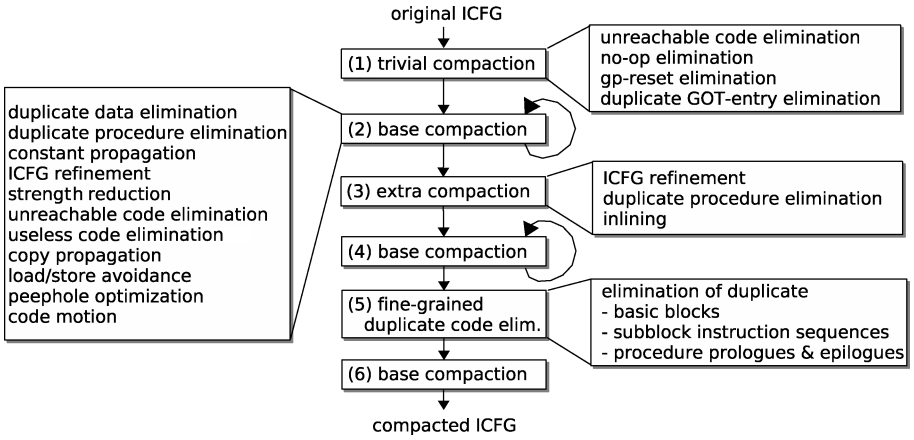

Fig. 10. The phases in the compaction strategy of SQUEEZE++, our link-time compaction prototype.

SQUEEZE++[9] applies the program transformations and the underlying analyses in six phases (see Figure 10). These six phases are applied in both the first and second runs of our compactor, after the code has been disassembled, and the initial ICFG is constructed.

(1) *Trivial compaction*. In the first phase, some trivial program optimizations are performed, such as the elimination of unreachable code, no-ops, and unnecessary GP computation code. If the GOT of a program is small enough to be indexed with one GP-value only, the GP will have the same value throughout the program, and all conservatively inserted GP-computations may be removed. Note that this could just as well be done during constant propagation, but treating the GP-computations as a special case is more efficient. This was also done by Srivastava and Wall [1994]. Finally, during the initial compaction phase, duplicate entries in the GOT are eliminated.

(2) *Base compaction—round 1*. Following the trivial optimizations of the first phase, a number of the base whole-program analyses and program optimizations are applied iteratively to exploit the circular dependencies between analyses, transformations, and ICFG refinement. In each iteration over the base optimizations, we also try to eliminate duplicate identical procedures. The rationale for doing so in this second phase is that we do not want to miss early opportunities for duplicate code eliminations of two procedures that initially are identical. If we apply too much program transformations before eliminating identical procedures, we risk that two procedures that initially were identical will become different as they are optimized in their different calling contexts.

(3) *Extra compaction*. In the third phase we eliminate nearly identical duplicate procedures. The reason for postponing this elimination until this third phase is that, unlike the duplication elimination of whole identical procedures, the merging of nearly identical duplicate procedures introduces

[9] `http://www.elis.UGent.be/squeeze++`.

overhead in the program. By postponing this merging until the ICFG has been refined and most of the unreachable code has been eliminated from the program, we avoid merging two procedures for which it later, after some more ICFG refinement, could have turned out that one of them was in fact unreachable.

Once the merging of nearly identical procedures is completed, larger procedures with one call-site are inlined. Thus, we avoid having inlining obfuscate the original, mergeable procedures in a program.

(4) *Base compaction—round 2*. Because this inlining results in additional analysis accuracy and compaction possibilities, we reiterate over the base analyses and transformations after the inlining. In each iteration, we still perform the identical procedure elimination, because it can happen that procedures that are at first not identical are made identical after some of the other transformations, such as the elimination of duplicate data. In order not to miss those opportunities, we keep applying duplication elimination for identical procedures during all iterations of the base compaction techniques.

(5) *Duplicate code elimination*. Unlike inlining, the finer-grained code duplication techniques do have a negative effect on the precision of the data flow analysis. Therefore these techniques are first applied in phase 5.

(6) *Base compaction—round 3*. After the finer-grained code duplication techniques, one final run of all base analyses and transformations is performed. This run eliminates as much of the overhead introduced during the previous phase as possible. Copy propagation, for example, eliminates some of the register copy operations that have been inserted during register renaming.

After these six phases, the basic blocks in the compacted ICFG are layed out, using profile information and a Pettis-Hansen [Pettis and Hansen 1990] style code layout algorithm. Finally, as our transformations have changed the program radically, we reschedule the code using a fairly simple list scheduler, and we assemble the code into an executable program.

7. VALIDITY OF UNDERLYING ASSUMPTIONS AND CORRECTNESS

Having brought together all discussed techniques in the prototype SQUEEZE++ in Section 6, this section reflects on the validity of the underlying assumptions made throughout the previous sections. First, the use of calling convention information and our assumptions about address computations are put in perspective. Later, our prototype's handling of more complex software constructs such as self-modifying code, volatile memory, and exception handling are discussed.

7.1 Calling Conventions

From Section 2.4.2 on, we assumed that calling conventions are maintained by global procedures, which can be detected by looking at the global symbol information. In some situations, however, this assumption may be incorrect.

First, manually written assembly code does not need to maintain calling conventions. While SQUEEZE++ currently does not include any tests for hand-written

assembly, its use of calling convention information can be turned off easily. In practice, however, we have not experienced a single case in which this was necessary. Apparently none of the system libraries that were linked into our testing applications contained problematic hand-written assembly code. If problematic code did show up in a program, the only solution (apart from abandoning the calling convention information) would be to have the compiler annotate the hand-written code and to take the annotations into account in the link-time analyses. While depending on such annotations would limit the applicability of a link-time optimizer to tool chains that can provide them, adding such capabilities to a tool chain is trivial. Some tool chains, such as the GCC compilers, already offer this option.

Furthermore, whole-program compilers that compile all source code together (including all library code) can disregard calling conventions completely. This is not a problem for link-time compaction, however, since link-time compaction would be useless in the presence of such whole-program compilers anyway.

Finally, there exists a limited group of exported, compiler-generated routines that do not maintain calling conventions. These routines typically implement frequently occurring source-level programming language computations that have no direct translation into a short instruction sequence. On the Alpha architecture, for example, this is the case for integer divide and remainder computations. Since there are no single machine instructions to perform these computations, they are implemented by calls to routines that perform the necessary computations. Such routines are called the *builtin* routines because they are built into the compiler. SQUEEZE++ is aware of the builtin functions as well. Their names are simply hard-coded in SQUEEZE++. We feel this is acceptable because, if the compiler knows that the builtin functions require special treatment, why wouldn't a link-time compactor know it too?

A final argument we want to make is the following: when followed by a link-time compactor that can remove much if not all of the overhead relating to calling conventions, a compiler or assembly programmer has no reason not to adhere to the conventions. We conjecture that, even in cases where the link-time optimizer can not get rid of all the overhead related to calling conventions, this is compensated by the additional program size reductions and program optimizations that are achieved because of this assumption. We argue that adherence to calling conventions of global procedures is a reasonable and acceptable *sine qua non* of link-time compaction.

7.2 Address Computations

The detection of inaccessible data during constant propagation relies on the fact that all intersection relative addresses used in a program are annotated as relocatable. In theory, compiler-generated or manually written assembly code can violate this assumption when detailed knowledge about the linking process is exploited during the code generation.

In practice, however, we believe our assumption is still valid. First of all, we never experienced any problems with this assumption while we were developing

SQUEEZE++. More importantly, there is simply no incentive to generate code breaking the assumption. Since a link-time program optimizer or compactor is ideally suited to optimize the data address computations in a program, there is, in our opinion, absolutely no incentive to optimize intersection address computations in such a way that the assumption is broken. And even if a reason should arise to optimize the intersection address computations at compilation time, providing the appropriate relocation information would be a trivial exercise.

With respect to code addresses, the detection of the potential targets of indirect control flow transfers relies on the assumption that either no code address computations are performed, or that they are annotated with relocation information. In practice, some of these annotations that we rely on need not be present. For example, the linker does not require relocation information for intramodular code displacements in position-independent code.

Again, we claim that we are allowed to make this assumption, as it is a trivial matter for a compiler or assembler to provide the necessary relocation information, even though it is redundant for simple linking. We know from discussions with compiler developers at major embedded computing firms that some of their compilers already annotate all address computations with relocations. They do so precisely to enable link-time program optimization and compaction. This is also the case on the Alpha Tru64Unix platform, for which we developed SQUEEZE++.

7.3 Self-Modifying Code

There are two basic ways to implement self-modifying code. The first one is to have some code reside in the writable sections of a program, perhaps even on the heap. This poses no problem, as the unknown node models this variable code. Another possibility is to modify code in the read-only text section of a program. In such cases, our compaction techniques will not work, as they assume that the text sections contains fixed code. With most operating systems, however, changing code in the text section requires system calls to get write access for the read-only memory pages in the text section. It suffices to detect these system calls and back off to treat this kind of self-modifying code correctly.

7.4 Volatile Memory

For the moment, the load/store avoidance optimizations in SQUEEZE++ cannot handle volatile memory. We simply assume no volatile memory at all. We believe this is not a fundamental problem. It would be simple to have a compiler provide information to SQUEEZE++ about memory locations that are to be considered volatile. Alternatively, we could exclude the load/store avoidance optimization since we have experienced that load/store avoidance on average contributes very little to the reductions in code size obtained with SQUEEZE++.

7.5 Exception Handling

Robust C++ program optimizers used in real production environments should be able to deal with exception handling. As a research prototype, SQUEEZE++ does

not. When exceptions are thrown in a program compacted with SQUEEZE++, the behavior is undefined since the thrown exception will not be caught correctly.[10]

The problems with exception handling relate to stack unwinding and exception handler resolution. We conjecture that these do not pose fundamental problems for link-time compaction. Discussions with industrial compiler developers strengthen our belief that exception handling is an implementation issue. In practice, it only poses problems for researchers that are not in control of the object file formats and the exception handling mechanism of their target platform, and have to live with vendor-supplied formats and conventions.

When exception handling is needed by a program, object files contain *code region descriptors*. In the context of this section, a *code region* is a range of instructions between two code addresses. Code region descriptors describe for each code region which exceptions are caught and which exceptions handlers should be called. At first sight, code regions seem problematic for program size when code abstraction is applied, since each abstracted procedure seems to need a separate descriptor. However, while most current object file formats and code region descriptor formats require that all procedures (or even parts of procedures) have their own descriptors, this is not a fundamental requirement. The existing formats can easily be adapted to enable the use of a single descriptor for multiple procedures. With such an extended format, and when abstracted code is grouped in the program layout (which is possible without influencing performance, since we only abstract cold code when we care about performance anyway) and all abstracted code handles exceptions in the same way, one descriptor will suffice for all abstracted code, and the influence on the whole program size will be insignificant.

In practice, a very small number of descriptors will be necessary to cover the abstracted procedures, rather than just one. For each abstracted region, it suffices to let the exception handling consist of a rethrow of the exception. As a result, the exception will be caught by the caller of the abstracted code. This rethrow requires stack unwinding, though, and it is because of this unwinding that multiple (rather than a single) descriptors are needed. In order to implement the stack unwinding, a descriptor encodes where the return address can be found in the described code region and what the stack frame looks like. Abstracted procedures for which the return address is stored in different places therefore require different descriptors. The number of the descriptors needed will remain small, however. There are only a liminted number of places to store the return address: in a register or in the top stack frame. Therefore the effect on the total program size will still be negligible.

Note that if two identical or similar code fragments each consist of more than one code region with different corresponding exception handlers, simply rethrowing an exception in the abstracted code is not an option. Handling this

[10]Note that programs compacted with SQUEEZE++ have no problem with handling signals. Signal handlers, because their address is passed to the OS when they are installed, are treated conservatively by the unreachable code detection. The system call is modeled by a call to the unknown node, and therefore worst-case assumptions are made about the signal handlers as soon as their address is propagated to the unknown node.

situation in abstracted code would require more extensive changes to existing exception handling mechanisms. Our experience with C++ code is that this situation rarely occurs in the types of code fragments that our duplicate code elimination targets. Therefore backing off in this situation, but still abstracting the contained regions separately, will not result in significantly lower code size reductions.

8. EXPERIMENTAL EVALUATION

This section evaluates the compaction techniques incorporated in SQUEEZE++.

8.1 Benchmark Programs

Our benchmark program suite consists of four groups of benchmarks:

(1) five embedded C applications from the *MediaBench*[11] suite;
(2) the full *SPECint2000*[12] benchmark suite, whose 11 C and one C++ programs are larger than the MediaBench programs;
(3) two scientific Fortran programs from the *SPECfp2000*[12] suite;
(4) a set of six C++ programs, including typical PDA applications such as the arcade game `xkobo`, the WYSIWYG word processor `LyX`, and the lightweight window manager `blackbox`.

All programs were compiled for the Alpha EV67 / Tru64 Unix (V5.1) platform with the vendor-supplied Compaq CC C (V6.3-025), C++ (V6.3-002), and Fortran (X5.3 ECO2) compilers (hereafter commonly described as *CC compilers*), and with the GNU GCC 3.3.2 compilers. The compiler flags we used were "-O1 -arch ev67" for CC compilers, and "-Os" for the GCC compilers, which instructed the compiler to optimize for size rather than for execution speed, without applying whole-program optimizations. All programs were statically linked with the vendor-supplied linker against the vendor-supplied system libraries.[13]

Since our target platform and the vendor-supplied tool chain were not oriented at code size, the generated binaries contained some overhead that an embedded compiler would not have generated. Since we consider it unfair to include the elimination of that overhead in the compaction results obtained with SQUEEZE++, we first eliminated that overhead by applying a base version of SQUEEZE++ on the linked binaries.[14] This base SQUEEZE++ version first eliminated duplicate entries from the GOT. This could be done without even disassembling the code. Instead the relocation information sufficed. Furthermore, the base version applied a simple unreachable code elimination because the thus eliminated code would not have been present if the system libraries had been engineered and structured with code size rather than execution speed

[11] `http://www.cs.ucla.edu/~leec/mediabench/`.
[12] `http://www.spec.org`.
[13] There exists no GNU linker or assembler for the Alpha/Tru64Unix platform.
[14] Note that SQUEEZE++ itself does not link a program. It operates on the linked binary and relies on the vendor-supplied linker to provide a link map and relocation information to extract all the necessary information from the original object files.

Table I. Original Sizes of the Code Sections, the Data Sections, and the Executable Files of the Base Benchmarks

	Compaq CC V6.3, Fortran X5.3			GCC 3.3.2		
Benchmark	Code Size	Data Size	Binary Size	Code Size	Data Size	Binary Size
MediaBench C Programs						
adpcm	135 KiB	66 KiB	208 KiB	135 KiB	65 KiB	208 KiB
epic	217 KiB	88 KiB	312 KiB	219 KiB	88 KiB	320 KiB
gsm	175 KiB	84 KiB	264 KiB	179 KiB	83 KiB	272 KiB
mpeg2decode	201 KiB	99 KiB	312 KiB	204 KiB	99 KiB	312 KiB
mpeg2encode	266 KiB	108 KiB	384 KiB	273 KiB	107 KiB	392 KiB
SPECint2000 C Programs						
164.gzip	172 KiB	77 KiB	256 KiB	189 KiB	77 KiB	272 KiB
175.vpr	327 KiB	135 KiB	472 KiB	327 KiB	131 KiB	472 KiB
176.gcc	1,453 KiB	405 KiB	1,864 KiB	1,553 KiB	380 KiB	1,944 KiB
181.mcf	203 KiB	79 KiB	296 KiB	204 KiB	79 KiB	288 KiB
186.crafty	368 KiB	135 KiB	512 KiB	363 KiB	128 KiB	504 KiB
197.parser	287 KiB	106 KiB	400 KiB	298 KiB	101 KiB	408 KiB
253.perlbmk	724 KiB	223 KiB	952 KiB	742 KiB	213 KiB	960 KiB
254.gap	638 KiB	139 KiB	784 KiB	631 KiB	117 KiB	760 KiB
255.vortex	633 KiB	237 KiB	880 KiB	663 KiB	224 KiB	896 KiB
256.bzip2	162 KiB	73 KiB	248 KiB	166 KiB	73 KiB	248 KiB
300.twolf	399 KiB	124 KiB	528 KiB	394 KiB	112 KiB	520 KiB
SPECint2000 Fortran Programs						
168.wupwise	697 KiB	181 KiB	888 KiB	245 KiB	100 KiB	360 KiB
178.galgel	868 KiB	193 KiB	1,072 KiB			
C++ Programs						
252.eon	525 KiB	222 KiB	760 KiB	784 KiB	332 KiB	1,128 KiB
blackbox	1,086 KiB	262 KiB	1,360 KiB	1,101 KiB	266 KiB	1,376 KiB
bochs	1,248 KiB	375 KiB	1,632 KiB	1,283 KiB	877 KiB	2,168 KiB
gtl	631 KiB	461 KiB	1,104 KiB	623 KiB	259 KiB	888 KiB
lcom	387 KiB	208 KiB	608 KiB	539 KiB	296 KiB	840 KiB
tyx	5,148 KiB	2,988 KiB	8,152 KiB	4,053 KiB	1,318 KiB	5,384 KiB
xkobo	987 KiB	338 KiB	1,328 KiB	1,010 KiB	467 KiB	1,480 KiB

in mind. Next, the base version removed no-ops from the linked binaries; a code size conscious compiler would not have inserted them in the first place. Finally, we used the profile-based code layout and code scheduling algorithms in SQUEEZE++ for the generation of the base versions of the benchmarks. This allowed us to evaluate the side effects of code compaction on execution speed, instead of comparing the quality of the compiler and SQUEEZE++ code scheduling back-ends.

The absolute sizes of all base program versions are given in Table I. Any size reductions presented in this article were achieved on these base binaries. For completeness, the white bars in Figure 11 indicate the fraction of the linked programs that was removed by the base SQUEEZE++ version.

Note that, for most benchmarks, both base versions (CC and GCC) were comparable in size. This is not surprising. Since the GNU C library was not supported on our evaluation platform, all C-library code linked into both program versions originated from the same library. The only significant differences in base size appeared in the Fortran program `168.wupwise` and in some of the C++ programs. This was due to the use of different libraries, such as the bigger

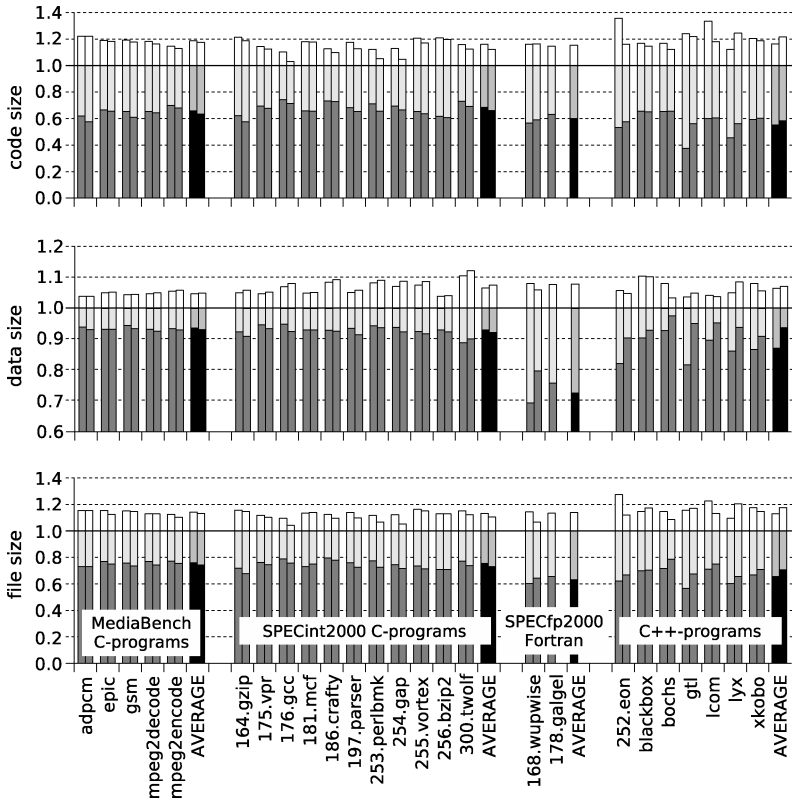

Fig. 11. Normalized static sizes of the compacted programs. All sizes are normalized to the size of the base program versions. For each benchmark, the left bar indicates the relative sizes of the CC-compiled program versions, while the right bar indicates the sizes of the GCC-compiled versions. For each program version, the dark bottom bar indicates the relative size of the program after compaction with SQUEEZE++. The white bars on top are added for completeness. They indicate the size of the original programs, as linked with the standard linker, compared to the size of the base programs. No average is given for the GCC versions of the Fortran programs, as only one Fortran program (`168.wupwise`) was compiled with the GCC compiler. `178.galgel` is a Fortran90 program, for which no GCC compiler exists.

Fortran math library that was linked into the program when the CC compiler was used, and the use of the GNU Standard C++ library by the GCC compilers. Furthermore, the instantiation of templates differs from compiler to compiler. The vendor-supplied C++ compiler used a repository [Levine 2000] to store template instantiations. As all instantiations in the repository were unique, only one copy of each instantiation was linked into the program. The GNU C++ compiler instead instantiated one copy of each required template class per compile command. In order to avoid duplication of instantiations, all source code was compiled with one single execution of the compiler front end. Note that this did not change the fact that each source code file was compiled and optimized separately. As can be seen in Table I, the GNU C++ compiler sometimes generated the smallest binaries, while the CC compiler did a better job for other programs.

Note that zero-initialized data sections consumed no space in the executable files stored on disk or in ROM. Those sections are therefore not included in the static data sizes presented in Table I. The data sections containing exception handling information (see Section 7.5) are included in the presented sizes, however. All standard and adapted versions of SQUEEZE++ used for this article left those sections unchanged.

8.2 Static Program Size Reductions

The relative code, data, and executable file sizes after compaction with SQUEEZE++ are depicted with the dark gray bars in Figure 11. From the base CC binaries, SQUEEZE++ was able to eliminate between 26–62% of the code. Depending on the type of benchmark programs, the average code size reductions ranged from 31% to 45%. Data size reductions ranged from a meager 5% to 31%, with averages between 6% and 27%, depending on the category of benchmarks.

For Fortran, the lack of pointers in the Fortran programs helped our detection of inaccessible data, as relatively few worst-case assumptions about the use of data addresses had to be made. Fortran programs also contain far fewer indirect control flow transfers, and thus allowed us to build a very precise ICFG, on which the whole-program optimizations performed very well. This explains the higher than average results for the Fortran programs.

C++ programs, on the other hand, are written in an object-oriented programming language that strongly supports code reuse. Thus, the programs rely more on library code, on which whole-program optimization in general, and unreachable code elimination in particular, perform very well. Furthermore, the duplicate code elimination techniques perform very well on the C++ programs containing a large number of templates, such as `252.eon`, `gtl` and `LyX`.

The combined code and data compaction results in the total executable file compaction depicted in the bottom chart of Figure 11. The reductions obtained on the full executable files were along the lines of the reductions obtained on the code alone, albeit somewhat smaller. The executable file reductions ranged from 20 to 43%. The reason that they followed the code size reductions was that (1) there was very little overhead in the executable file (such as headers describing the sections in the executable), and (2) compared to the code sections in an executable, the data sections were relatively small as well, thus playing a minor role in total compaction.

The reductions obtained on the GCC binaries were mostly comparable to those obtained on CC binaries. For the C programs, the average code size reductions were even a couple of percentage points higher, ranging from 27% to 42%. The reason for the higher compaction was code quality: the code produced by the GCC compilers was less optimized. As a result, more inaccessible data was detected and many more edges coming from the unknown node were eliminated. The code sequences generated by the GCC back-end also showed less variation, resulting in more code being abstracted.

For the C++ programs, link-time compaction also performed significantly differently on the programs generated with GCC. To some degree, this resulted from the fact that a different standard C++ library implementation was used by the two C++ compilers. With respect to code size, the big differences in

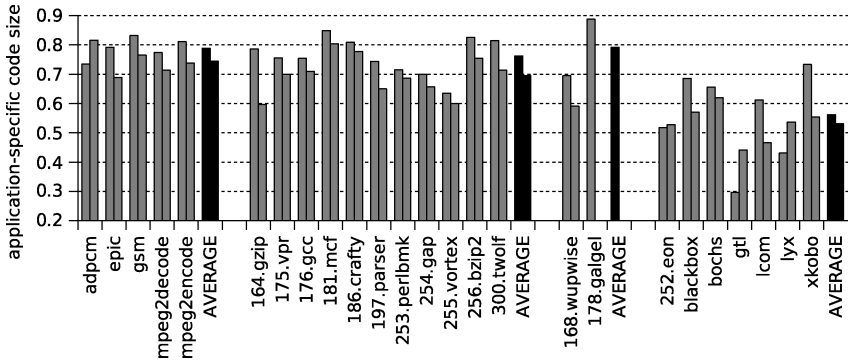

Fig. 12. Relative static size of the application-specific parts of the code after compaction with SQUEEZE++. The left bar for each benchmark indicates the relative size of the CC version, the right bar of the GCC version.

reduction between the CC and the GCC versions observed for `gtl`, `252.eon`, and `LyX` were due to the weaker performance of the duplicate procedure elimination in the case of GCC. Many fewer identical duplicated procedures were discovered and eliminated in the programs generated with the GCC compiler. This resulted from the different template instantiation mechanism, as discussed in Section 8.1. This also explains, to some extent, why much less inaccessible data was eliminated in the programs generated with the GNU C++ compiler. With the CC compiler and its use of a repository for template instantiations, each instantiation was a separate object file in the repository, and each instantiation therefore came with its own object data sections. When the GNU C++ compiler was used to generate code, template instantiations were generated in the object files that used them, together with the nontemplate code. This resulted in a link-time inaccessible data detection that worked on fewer, and more coarse-grained object data sections, and that was hence less accurate.

Finally, it should be noted that SQUEEZE++ could not bridge the large size gap between the two base versions of `168.wupwise`. While the base CC version was about twice as big as the base GCC versions, the relative code and data reductions obtained on both versions were within 10% of each other. This clearly shows that although large program size reductions can be obtained with link-time compaction, it is not a silver bullet for compiler tool chain builders. Even with link-time compaction, they still need to engineer their libraries very carefully.

To exclude the effects of the commonly used system libraries on our measurements for both compilers, and to get an indication of the code size reductions that can be achieved with link-time compaction on programs that use dynamically linked libraries (such as shared libraries), we evaluated SQUEEZE++ on the application-specific code of the programs. In this experiment, an adapted SQUEEZE++ treated all library code as a blackbox of unknown code.

The resulting relative sizes of the compacted application-specific code are depicted in Figure 12. Typically, the code size reductions obtained on the application-specific part of the code were less than that obtained on the whole

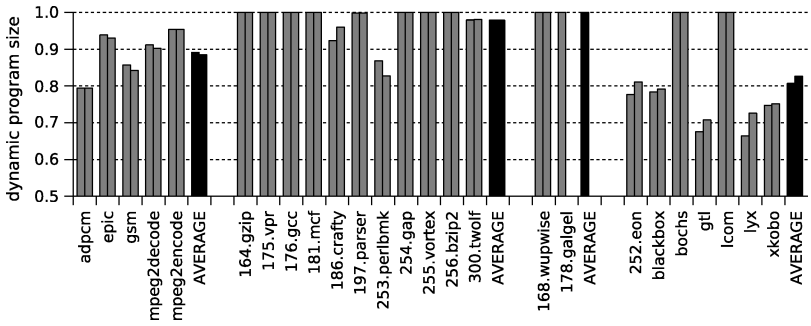

Fig. 13. Relative dynamic sizes of the programs after compaction with SQUEEZE++. The left bar for each benchmark corresponds to the CC version, the right bar to the GCC version.

program. The reason is that libraries are the pieces of code where separate compilation leaves the most optimization opportunities for a link-time compactor. Furthermore, it is clear that the lower quality of code generated by the GCC compiler left significantly more compaction possibilities for link-time compactors. As can be seen in Figure 12, SQUEEZE++ performed significantly better on the GCC-compiled (application-specific) C code.

For the C++ programs, the code size reductions obtained were still very large, at least for the programs consisting for a large part of template instantiations, such as `gtl` and `LyX`. For those programs, duplicate code elimination performed exceptionally well on the CC programs, and as a result the application-specific part of the program was compacted more than the whole program. For C++, the average code size reduction remained at 43%, coming from 45%, for the programs compiled with CC. Because the template instantiation method in GCC generates less duplicated code, SQUEEZE++ performed significantly worse on the GCC versions of `gtl` and `LyX`. On most of the other C++ benchmarks, however, SQUEEZE++ performed better on the GCC versions. As for C-programs, the GCC compiler-generated code of a lower quality and with less variation, which was more easily analyzed and abstracted.

8.3 Dynamic Program Size Reductions

The results presented thus far are static results: they present the relative sizes of the code, the data, and the executable files as they are stored in ROM, a flash memory, or on disk. Figure 13 presents the dynamic memory size reductions achieved with SQUEEZE++, that is, the reductions on the amount of RAM that the OS needs to allocate for executing a benchmark.

For some programs, such as the MediaBench programs and most of the C++ programs, the dynamic size reduction closely follows the static reduction on the size of the executable file presented in the bottom chart of Figure 11. The reason is, of course, that these programs allocate little if any memory on the heap, and that their zero-initialized data sections (which are allocated when the programs are loaded by the OS) are not very large compared to the sizes of their code and other data sections. On this type of benchmark, the achieved dynamic memory reductions range from 4% to 34%. Not surprisingly, the

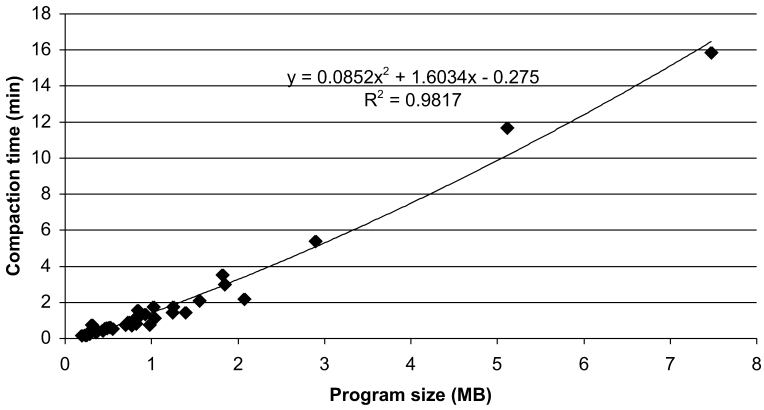

Fig. 14. Compaction time in minutes for the benchmark programs.

MediaBench benchmarks fall in this category. The MediaBench programs are typical examples of embedded applications that need to run on embedded systems with limited amounts of memory. `blackbox`, the lightweight windows manager designed to consume as little memory as possible, is no surprise in this category either. Other programs such as `LyX` and `xkobo` pleasantly surprised us. Apparently these interactive programs also require very little dynamically allocated memory.

For other benchmarks that allocate large amounts of memory dynamically, such as most of the SPEC benchmarks, the static size savings are completely outweighed by the size of the dynamically allocated memory. Since SQUEEZE++ never changes the amount of memory allocated on the heap (its optimizations are much lower-level), the reductions obtained on the dynamic memory footprint of these benchmarks approximates zero.

8.4 Cost-Effectiveness of Whole-Program Analyses

Common wisdom about whole-program analyses says that they are often too slow to be practically viable. In Figure 14, we have depicted the execution time of one run of SQUEEZE++ as a function of the input program size.[15] The equation of the quadratic curve fitted to the results shows that the compaction time scales quite well with program size, albeit not linearly. Memory consumption varies linearly with program size, and for our largest benchmarks, we needed about 1 GiB of memory.

While we have optimized SQUEEZE++ to some degree to ease our lives during its development, there is still room for further optimizations. Nonetheless we believe it is meaningful to measure the cost-effectiveness of some key analyses in SQUEEZE++. To that extent, we have disabled or simplified some of those analyses. The resulting changes in achieved compaction and in required compaction time are presented in the remainder of this section.

[15]All experiments reported in this article were executed on a lightly loaded dual 667-MHz Alpha 21264 EV67. The four-way superscalar processors each have a split four-way associative L1 data and instruction cache of 64 KiB and a unified L2 cache of 2 MiB. The main memory is 2.5 GiB large.

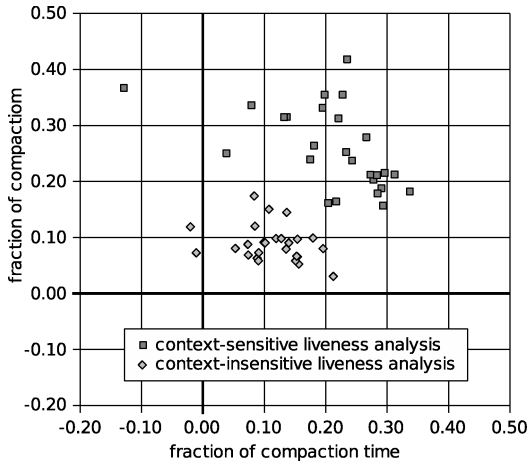

Fig. 15. Fraction of the compaction time that can be saved and fraction of the code compaction that is lost when disabling liveness analysis or when using a simpler version.

We strongly favor this method for measuring cost-efficiency over measuring the execution time that is spent in any single analysis in isolation. In our opinion, the latter would not result in meaningful numbers, since disabling or simplifying one particular analysis always influences the effectiveness and efficiency of other analyses. Measuring the number of instructions eliminated by a specific program transformation is not useful either. Useless code elimination, for example, eliminates both instructions that are useless in the original program, and instructions that have become useless because other transformations made them useless. Unfortunately, it is not possible to differentiate between the two categories in SQUEEZE++, because program transformations and ICFG refinement are applied together.

8.4.1 *Liveness Analysis.* For liveness analyses, we first compare SQUEEZE++'s standard liveness analysis to a nonanalysis that conservatively assumes all registers are live after all basic blocks. The results of this comparison for CC-compiled binaries are depicted in Figure 15. We have omitted the results for GCC-compiled binaries, because they are along similar lines and offer no additional insights.

On the horizontal axis, the dark gray dots indicate the fraction of the compaction time that can be saved by disabling liveness analysis. In other words, on the horizontal axis the dark gray dots indicate (1) the fraction of the compaction time spent in the context-sensitive liveness analysis itself and (2) what additional/lesser time is spent in other analyses that take longer/shorter when they consume the accurate liveness information. Figure 15 indicates that up to 34% of the compaction time in SQUEEZE++ is due to liveness analysis. However, there are also programs on which liveness analysis consumes very little time. On one program, `255.vortex`, SQUEEZE++ with liveness analysis enabled runs about 12% faster than SQUEEZE++ without liveness analysis. On that benchmark, the availability of accurate liveness information speeds up the detection

of inaccessible data during constant propagation by a factor of 4, thus saving more time than is spent in the liveness analysis itself. Furthermore, more code is eliminated when liveness information is available. More useless code is detected, more free registers are found for program transformations, such as procedural abstraction, and more inaccessible data (including inaccessible procedure pointers) is eliminated. So when accurate liveness analysis is available, the other analyses and transformations in SQUEEZE++ are applied on a smaller program. Hence they run more quickly.

The fraction of the total code compaction that directly or indirectly results from liveness analysis is indicated by the dark gray dots on the vertical axis in Figure 15. This amounts to up to 42% of the total compaction. Obviously liveness analysis is very important for link-time compaction.

To evaluate whether or not the interprocedural liveness analysis needs to be context-sensitive, we also implemented a context-insensitive liveness analysis in SQUEEZE++. The results of this comparison are depicted in Figure 15 by means of the light gray dots. On the horizontal axis, they indicate the fraction of the compaction time that can be saved by replacing the context-sensitive analysis by a context-insensitive one. This ranges from −3% to 22%. The reason why using a simpler analysis can result in a slowdown of 3% is similar to the aforementioned reasons for the 12% speedup caused by enabling the context-sensitive analysis.

Obviously opting for a simpler analysis has an associated cost. As the light dots in Figure 15 indicate on the vertical axis, between 4% and 16% of the maximal compaction gets lost. Still, opting for a context-insensitive liveness analysis usually saves relatively more time than it costs in code size reduction. So whereas disabling the liveness analysis altogether is clearly not cost-efficient, the context-insensitive analysis may be considered more cost-efficient than a context-sensitive one.

8.4.2 *Constant Propagation.* For constant propagation, we performed a similar experiment as for liveness analysis. First, we disabled constant propagation altogether. The results are depicted in Figure 16, by means of the dark gray dots.

On average constant propagation is an expensive analysis: even though we only apply it context-insensitively, not employing it saved us between 30% and 50% of the compaction time. However, the code compaction for many benchmarks drops by more than 30% if no constant propagation is employed. Furthermore, without constant propagation, no inaccessible data is detected or eliminated. So while constant propagation is time-consuming, we believe it to be worthwhile.

To evaluate whether or not the detection of inaccessible data during constant propagation is cost-efficient for code compaction, we replaced the standard constant propagation in SQUEEZE++ with a version that does propagate constants, but that does not detect inaccessible data.

Compared to the standard constant propagation, using the simpler version saves between −1% and 30% of the compaction time, as is indicated by the light gray dots on the horizontal axes of Figure 16. The cost of opting for the

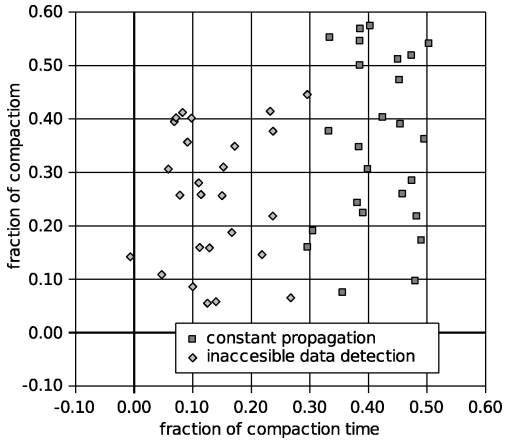

Fig. 16. Fraction of the compaction time that can be saved and fraction of the code compaction that is lost when disabling constant propagation or when using a simpler version.

simpler constant propagation is high however. Besides not being able to eliminate inaccessible data with the simpler version, between 5% to 45% fewer code is eliminated from the benchmarks. Clearly adding the detection of inaccessible data to the constant propagation proves to be very cost-efficient.

The reason that both constant propagation and inaccessible data detection are so important for compaction is threefold: (1) the optimization of address computations largely depends of the detection of the addresses being computed by constant propagation, (2) the refinement of the ICFG during the compaction process depends largely on the detection of constant targets of indirect control flow transfers, and (3) the detection of code pointers in inaccessible data section is important in detecting additional unreachable code.

8.4.3 *Duplicate Code Elimination.* The cost-efficiency of the duplicate code elimination is depicted in Figure 17. On the horizontal axis, the fraction of the compaction time spent in the duplicate code elimination techniques (and the underlying analyses) is indicated. This varies between −3% and 44%. On the vertical axes, the fraction of the total compaction that is achieved through duplicate code elimination is depicted. This amounts to 46% for the C++ programs. For the C and Fortran programs, between 9% and 20% of the total code size reduction originates from duplicate code elimination.

It should not come as a surprise that the higher dots on the chart are the C++ programs. For `gtl`, duplicate code elimination was so effective during the first iterations of base optimizations, that its execution time was compensated for the fact that the other analyses were performed on a much smaller program.

Unlike for C++ programs, duplicate code elimination is rather time-consuming for the small C-programs, and hence not very cost-efficient. For such small C programs, we can conclude that whole-program optimization is typically more important than duplicate code elimination, even though the latter still results in significant code size reductions.

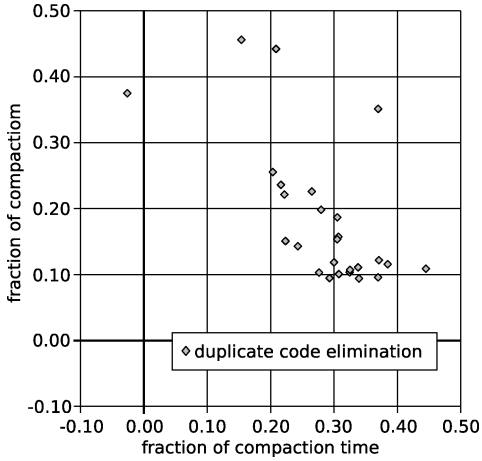

Fig. 17. Fraction of the compaction time that can be saved and fraction of the code compaction that is lost when disabling duplicate code elimination.

8.5 Other Performance Metrics

So far, we have quantified code size reductions and compaction times. In what follows we provide insights on other important aspects of the compacted programs.

8.5.1 *Execution Speed.* Most compilers have the capability of enabling or disabling optimizations that involve a speed versus size tradeoff, such as inlining and loop unrolling. This provided tradeoff supports the common belief that smaller programs are most often slower. As for the compaction techniques discussed in this article, two direct effects on execution speed can be observed:

—The whole-program optimizations most often will have a positive effect on execution speed, as they result in fewer instructions being executed.

—Some duplicate code elimination techniques will result in a slowdown of the compacted programs, as they introduce run-time overhead. Code abstraction introduces procedure calls and returns, and the merging of similar whole procedures involves the addition of conditional branches and the setting and testing of a parameter. Register renaming involves the insertion of copy operations.

To quantify these effects, we measured the execution times of the SPECint2000 benchmarks using the reference input sets.[16] Besides the base versions of the programs, we measured three versions that were compacted with SQUEEZE++:

—*Full duplicate code elimination*. For this version, all compaction techniques in SQUEEZE++ were applied to generate the smallest possible binaries.

[16]All profile information (i.e., basic block counts) used in this article were collected using the much smaller and different standard training inputs.

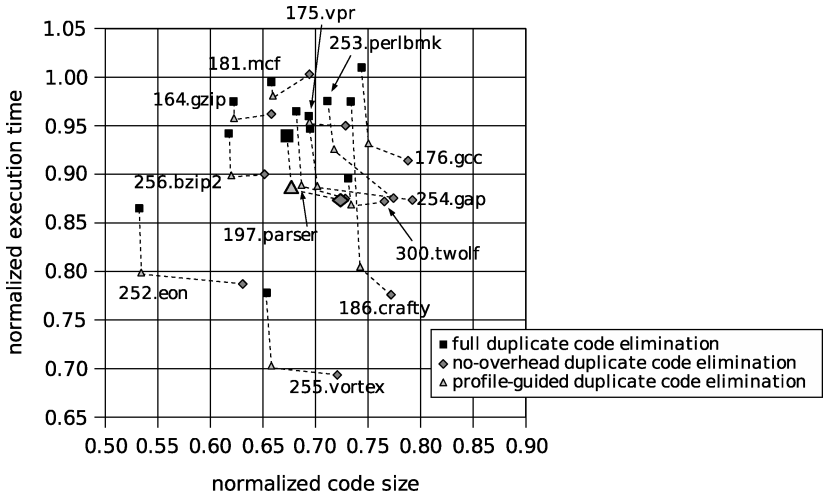

Fig. 18. Normalized code sizes versus normalized execution times for the SPECint2000 programs compacted with SQUEEZE++. All values are normalized to the base program version, and the enlarged symbols denote the averages.

—*No-overhead duplicate code elimination*. All optimizations were applied, together with those compaction techniques that do not introduce run-time overhead.

—*Profile-guided duplicate code elimination*. All compaction techniques were applied, but duplicate code elimination techniques that introduce run-time overhead were not applied on hot code. Hot code is code that is executed most frequently and, in our experiments, was responsible for 95% of the dynamic instruction count. This 95% was chosen out of a couple of possibilities (90%, 99%, etc.) because it resulted in our opinion in the best mix between execution speed and code size.

The chart in Figure 18 shows the normalized code sizes and execution times of the benchmarks generated with CC. On average, the full compaction version was about 32% smaller than the base version. It was also about 6% faster. On average, the speedup obtained with the whole-program optimizations was therefore larger than the slowdown caused by duplicate code elimination. If we didn't apply duplicate code elimination techniques that introduced overhead, the speedup obtained on average rose from 6% to about 13%. The associated drawback was that the code size reductions dropped from 32% to 27%. Fortunately, there is no need to make this compromise between execution speed and code size. By using profile information to guide the duplicate code elimination, we can, on average, get close to the best of both worlds. Less than 1% of the obtainable size reduction is lost by excluding the hot code from duplicate code elimination, while most of the possible speedup can still be obtained (11.5% on average).

Note that for some programs (such as `181.mcf`) the program version without duplicate code elimination seems to be slower than the program versions with profile-guided or full duplicate code elimination. The reason is that SQUEEZE++

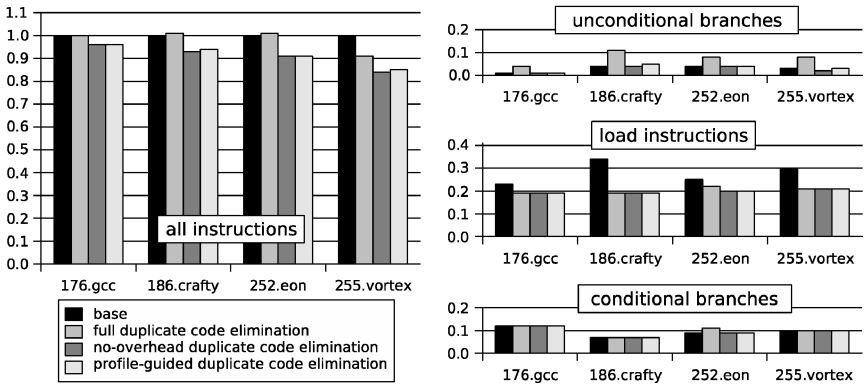

Fig. 19. Normalize dynamic instruction counts for several types of instructions, for four versions of four benchmarks generated with the CC compilers. All counts are normalized to the total dynamic instruction count of the base version of each benchmark.

does not insert no-ops to align hot basic block and procedure entry points for optimal execution times. Not adding no-ops adds noise to the execution times, because the quality of the generated schedules then depends on all the code in the program. We have tried to remove this noise by adding no-ops to hot code, but were unable to remove all noise without significantly increasing program size. Therefore we chose not to add no-ops at all.

8.5.2 *Instruction Counts.* The speedups/slowdowns are to a large extent caused by increasing/decreasing dynamic instruction counts. In Figure 19, we have depicted the dynamic instructions counts for several important categories of instructions for the same four versions of four benchmark programs. These counts were measured with the SimpleScalar simulation tool set [Burger and Austin 1997], using parts of the reference SPEC input data sets to avoid overly long simulation times. Still all simulations were more than 36 billion instructions long.

The numbers in Figure 19 indicate that link-time compaction can indeed significantly reduce the number of executed instructions, provided that duplicate code elimination is not applied on frequently executed code.

The number of load instructions drops significantly because of whole-program optimization, and in particular because of the elimination of most loads from the GOT. This normally is not affected by duplicate code elimination, except for programs such as `252.eon` on which a lot of procedure parameterization is applied. If this is applied on hot code, some additional loads are executed to load the newly added parameter from the stack. These loads vanish as soon as no hot duplicate code is parameterized.

As expected, the numbers of executed unconditional control flow transfers (including procedure calls) increased when hot code was abstracted.

Finally, we notice that the number of conditional branches executed did not decrease significantly. Instead, when parameterization was applied on hot code, as in `252.eon`, the number of executed conditional branches increased, because conditional branches followed the tests on the new parameters.

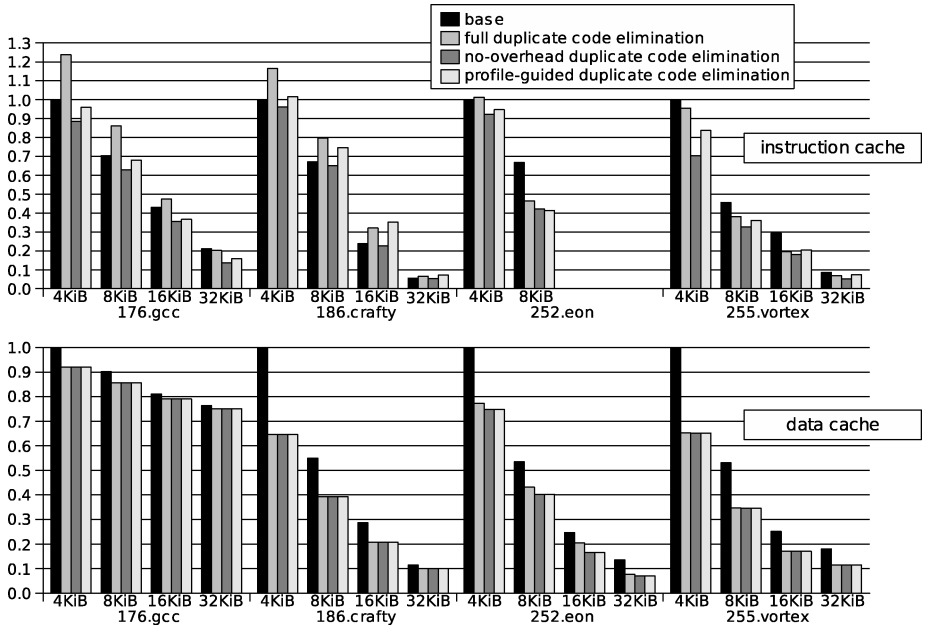

Fig. 20. Normalized number of instruction and data cache misses for four versions of four CC-compiled programs and four cache sizes.

8.5.3 *Cache Performance.* To study the side effects of program compaction on cache performance, we simulated the same benchmark programs with SimpleScalar again, measuring the number of level 1 caches misses for a variety of split instruction and data cache sizes and associativities. In Figure 20 we have depicted the normalized number of instruction and data cache misses for two-way set associative caches with cache lines of 64 bytes. The results for other cache line sizes and for direct mapped and four-way set associative caches were along similar lines, and are therefore not included. For `252.eon`, we have omitted the instruction cache misses for the two larger cache sizes, as they were too low to be significant.

It is no surprise that there were fewer data cache misses, since the number of data cache misses was highly correlated to the decrease in executed load operations depicted in Figure 19. We can see that substantial savings can be made, especially when very small caches are used.

With respect to the instruction cache misses, the results presented in the top chart of Figure 20 are in line with the results presented by Debray et al. [2000] and De Sutter et al. [2003]. Because of the whole-program optimizations, the working set size decreased, and therefore fewer instruction cache misses were observed when no duplicate code elimination was applied that introduced overhead. When duplicate code elimination is applied blindly, however, the working set size increases again, as discussed in Section 5.1.6. This effect can be limited by using profile information to guide the duplicate code elimination.

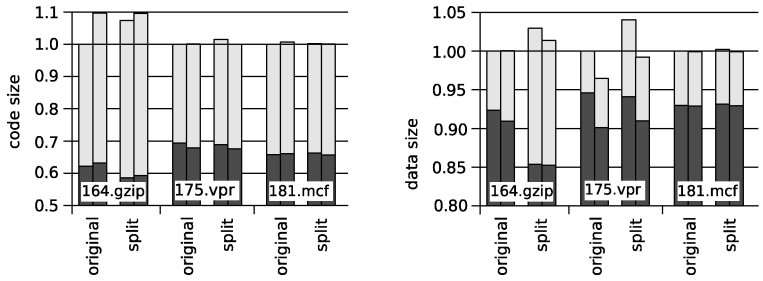

Fig. 21. Relative code and data sizes of three benchmark programs compiled with the CC and the GCC compilers, with original and split source code, before (in light gray) and after (in dark gray) compaction with SQUEEZE++. All code and data sizes are normalized to the sizes in the base program versions compiled with CC.

8.6 Effects of Separate Compilation and Compiler Optimization

Section 2 discussed the overhead resulting from the conservative optimization of separate compilation, together with the additional knowledge about the compiled code that results from separate compilation. This additional knowledge potentially leads to better link-time compaction. To evaluate whether or not additional knowledge and opportunities resulting from separate compilation and less aggressive optimization can compensate for the less efficient code generated by a compiler, we performed the following experiment.

We split the source code files of three of our benchmarks into multiple source code files that each defined one global procedure or one global variable. Then we compiled these split programs in the same manner as we compiled the original programs. As a consequence of this splitting, all object code sections contained exactly one procedure. Therefore all procedure calls were intermodular, and all compiled procedures adhered to the calling conventions. Furthermore, each global variable now was stored in its own object data section. For `164.gzip`, the 14 original source code files were split into 163 new source files. For `175.vpr` 61 original files were split into 346 new source files, and for `181.mcf` 11 files were split into 33 files. The vendor-supplied C-library code was not split, as we did not have its source code.

The results for code and data compaction on the "split" programs are depicted in Figure 21. The light gray bars in the left chart indicate that the code size of the base CC version of `164.gzip` increased about 7% when all procedures and global variables were compiled separately. This did not happen with GCC, where both versions were about 10% larger than the original CC version.

The bottom, dark gray bars indicate that the situation has reversed after compacting `164.gzip` with SQUEEZE++: the compacted code compiled from split source code files was smaller than the compacted code compiled from the original source code. Much to our surprise, separately compiling all procedures and global data resulted in an additional 4% code size reduction when SQUEEZE++ was used. This effect of more separate compilation also occurred, albeit to a much smaller extent, with the CC version of `175.vpr`. It did not occur with `181.mcf`, however.

Furthermore, the code compiled with GCC and compacted with SQUEEZE++ has become as small as the compacted code that was compiled with the CC compiler. This resulted from the fact that the GCC generated code is easier to analyze, and shows less variation, which results in more code abstraction.

The reason why SQUEEZE++ can sometimes compensate for the weaker compiler optimization in case of split source code files is twofold. First of all, the detection of inaccessible data is performed at a much finer granularity. Since each global variable now has its own object section, the worst-case assumptions that have to be made for the detection of inaccessible data during constant propagation are now limited to single variables instead of to object sections containing multiple variables. The finer granularity results in more data being eliminated from the program, and since the additional eliminated data also contains dead procedure pointers, the improved elimination of dead data also improves the elimination of unreachable code. The right chart in Figure 21 shows that all three programs showed increased data sizes for at least one version of the base binaries. However, the improved link-time elimination of inaccessible data was able to compensate for this. In the case of `175.vpr` compiled with CC, the resulting compacted data became a little bit smaller, and in the case of `164.gzip` the difference was substantial: the data size decreased by another 7%.

Second, and of minor importance, is the fact that some local procedures have been made global during the source splitting process, thus resulting in code which we can assume maintains calling conventions.

Please note that the effects of more separate compilation were most noticable on `164.gzip`, while they were nonexistent for `181.mcf`. For `164.gzip`, each original source code file was on average split into 11.6 new files, while each `181.mfc` source code file was only split into three new files.

The compaction results in Figure 21 suggest that future compilers should be adapted to reflect the presence of a link-time optimizer or compactor down in the tool chain. Today, by contrast, programming tool chain developers design their compilers under the assumption that no link-time optimizations will be applied.

On the one hand, compilers designed under this assumption only pass the information to the linker that is necessary to do the linking job, that is, code and data, relocation information, and symbol information [Levine 2000]. All other types of information collected during the compilation process are thrown away when the object files are written. This includes alias information, information about the stack frames of procedures, and all kinds of higher-level semantic information. Much of this information is very hard [Debray et al. 1998], if not impossible, to recollect from the object files. It would therefore be better if more of the information collected at compile time were passed to the optimizing binary rewriter.

On the other hand, compilers try to optimize the code as much as they can, since they are engineered to be the only optimization phase in the tool chain. When a binary rewriting tool follows, however, some of the optimizations performed by the compiler are useless or, even worse, counterproductive. Optimizations on address computations, for example, are better applied at link time, when the whole program layout is known. In general, compile-time

optimizations should be disabled in the presence of link-time rewriting if the compiler optimizations (1) can be performed by the binary rewriter as well, (2) when they obfuscate the program to such an extent that the binary rewriting phase cannot extract the necessary information from the object files, and (3) when they do not allow the binary rewriter to make some important assumptions, such as the ones mentioned in Section 7.

9. RELATED WORK

An survey of code-size reduction methods, covering a very large range of techniques, was recently presented by Beszédes et al. [2003].

9.1 Squeeze

SQUEEZE++ is an evolution of SQUEEZE, which in turn was derived from ALTO [Muth et al. 2001], a link-time optimizer for the Alpha platform optimizing for speed. SQUEEZE in fact was ALTO minus some code size increasing optimizations, plus code abstraction. SQUEEZE was first presented by Debray et al. [2000]. The major additions and developments after that article was published are the following:

—that SQUEEZE++ is a link-time compactor, requiring relocation and symbol information that is available in the object files only, and not in a relocatable executable, on which SQUEEZE operated;
—the optimization of the time-consuming analyses, as discussed in Section 4.6;
—the elimination of inaccessible data, as described in detail in by De Sutter et al. [2001];
—the two-phase compaction as discussed in Section 4.2.4;
—the use of the unknown node for the analysis of stack behavior, as discussed in Section 4.1;
—the elimination of duplicate identical and similar procedures, of subblock instruction sequences, and of duplicate read-only data as discussed in Section 5 and by De Sutter et al. [2002];
—the conditional constant propagation that extends simple constant propagation;
—a radical change in compaction strategy: the idea by Debray et al. [2000] was to first optimize a program as much as possible by iteratively applying the base and extra compaction techniques; afterwards code was abstracted at the lower granularity levels, and this was followed by another round of base compaction techniques to remove some overhead introduced by the code abstraction techniques; in its current state, SQUEEZE++ employs the strategy discussed in detail in Section 6.2.

To compare the performance of SQUEEZE++ to that of SQUEEZE as presented by Debray et al. [2000], we have applied an adapted "old" version of SQUEEZE++ on all our benchmark programs. In this adapted version, all new techniques are disabled. Running the original version of SQUEEZE on the benchmarks is simply not possible, because it requires too much memory to compact our largest benchmarks on our current hardware (with 2.5 GiB of memory).

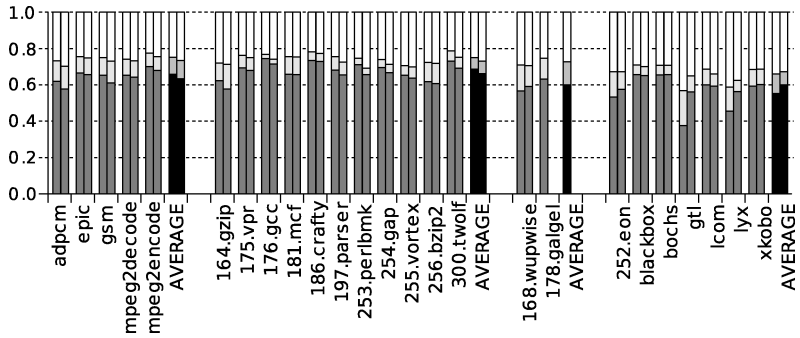

Fig. 22. Code compaction achieved with an adapted "old" version of SQUEEZE++ that resembles the state of SQUEEZE as presented in 2000 [Debray et al. 2000]. The middle bars indicate the progress that has been made since then.

The code size reductions obtained with the old version of SQUEEZE++ and with the standard version are depicted in Figure 22. The middle (light gray) bars indicate the fraction of the program eliminated by the standard version, but not by the old version. It is clear that the new techniques and the new strategy result in significant additional code size reductions. Besides the data size reductions, the new techniques in SQUEEZE++ eliminate between 3% and 19% more code. On average, the code size reductions have increase from 28% to 36%.

The speedups resulting from the compaction with the old version of SQUEEZE++ do not differ significantly from those achieved with the current version.

9.2 Code Compression

There is a considerable body of work on code compression. Much of this has focused on compressing executable files as much as possible in order to reduce storage or transmission costs. These approaches generally produce compressed representations that are smaller than those obtained using our approach, but they have the drawback that they must either be decompressed to their original size before they can be executed [Ernst et al. 1997; Franz 1997a; Franz and Kistler 1997; Fraser 1999; Pugh 1999]—which can be problematic for limited-memory devices—or require special hardware support for executing the compressed code directly [Lekatsas et al. 2003; Kemp et al. 1998; Kirovski et al. 1997; TriMedia Technologies Inc. 2000; Corliss et al. 2003].

Using another branch of the SQUEEZE code, Debray and Evans [2002] added code compression to already compacted binaries. They used profile data to identify infrequently executed code fragments which they compressed to a nonexecutable form. At run-time the fragments were decompressed into a buffer on demand and executed. They obtained additional code size reductions of 13.7% to 18.8% (including the decompressor and buffer). The influence on performance ranged from a slight speedup to a 28% slowdown.

Ros and Sutton [2003] studied the influence of compiler optimizations on the achieved compression ratio of compression schemes for the TI TMS320C6xxx VLIW DSP family. Their conclusions are in line with our observation in

Section 8.6 that more compaction is achieved, in particular through code abstraction, when following lower levels of compiler optimization.

Proebsting [1995] looked for repeated patterns in the intermediate program representation used by the compiler. So called super-operators were chosen, corresponding to the most frequently occurring patterns. These (application-specific) superoperators were used to extend a virtual instruction architecture, for which the program was compiled. At the same time, an interpreter able to interpret the extended instruction set was generated in C, from which it could be compiled to the original target architecture. They reported an average code size reduction of 50%, albeit with an undesirable large impact on execution speed.

Evans and Fraser [2001] similarly described a method for producing compact, bytecoded instruction sets and interpreters for them. Their system transforms a grammar for programs, creating an expanded grammar that represents the same language as the original grammar, but permits a shorter derivation of programs. Typically the program size reductions obtained were much larger than the increase in the size of the adapted interpreter.

Clausen et al. [2000] applied minor modifications to the Java Virtual Machine to allow it to decode macros that combine frequently recurring bytecode instruction sequences. They reported code size reductions of 15% on average. Our techniques do not rely on changing the underlying architecture on which a program is executed and, within the group of statically bound languages that are compiled directly into native machine code, are not source-language-dependent.

The techniques discussed in this article do not require any modifications to the compilers or the target architecture and produce programs that are faster, rather than slower. However, being late in the tool chain, SQUEEZE++'s implementation (not its concepts!) is highly target-architecture-dependent. This contrasts to some of the compression techniques [Franz and Kistler 1997; Proebsting 1995] that operate on a machine-independent intermediate program interpretation. As such, those techniques offer the additional benefit of being target-machine-independent.

9.3 Mixed-Width Instruction Sets

Mixed-width instruction sets combine 32-bit instruction sets with 16-bit instruction sets. The latter will typically require more instructions to encode some functionality, but as they are smaller, the overall code size is reduced. The best known examples of these mixed-width instruction sets are MIPS with MIPS16 [Kissell 1997] and ARM with Thumb [Turley 1995]. Recently, Krishnaswamy and Gupta [2002] proposed an automated profile-guided technique to select ARM/Thumb code. When combined with Thumb extensions [Krishnaswamy and Gupta 2005] that increase its performance, these techniques can avoid the need to compromise between performance and code size.

9.4 Code Abstraction

Most of the previous work on code abstraction to yield smaller executables has treated an executable program as a simple linear sequence of instructions

[Baker and Manber 1998; Cooper and McIntosh 1999; Fraser et al. 1984]. It used suffix trees to identify repeated instructions in the program and abstract them into procedures. The size reductions they reported were modest, averaging about 4–7%.

The use of predicated instructions for abstracting nonidentical but similar code fragments has been studied by Cheung et al. [2003]. On the ARM architecture, the introduction of predicated execution in abstracted basic blocks is able to improve on identical code abstraction by 28% or 37%, depending on the compiler used to generate the original code.

The use of so called *echo instructions* was first proposed by Lau et al. [2003]. With echo instructions, the single copy of the abstracted code can be left in its original place. The other copies are then replaced by an echo instruction, that tells the processor to execute a subset of the instructions from the single copy. An implementation of the necessary code transformations and the insertion of echo instructions in SQUEEZE revealed that an additional 10% code size reduction can be achieved with the use of echo instructions.

9.5 Binary Rewriting and Binary Translation

Multiple link-time binary rewriters that focus on speed optimization have been implemented for the clean Alpha architecture. They include OM [Srivastava and Wall 1994], Spike [Cohn et al. 1997; Flower et al. 2001], and Alto [Muth et al. 2001].

Techniques to build a control flow graph from disassembled binary code have been discussed in numerous articles, including Debray et al. [2000], De Sutter et al. [2000], Kästner and Wilhelm [2002], and Snavely et al. [2003].

We know of two static binary rewriting tools for embedded systems targeting code size. CodeCompressor (CC)[17] from Raisonance is a code compactor applying inlining, code abstraction (we have found no details on their techniques), and peephole optimizations on programs compiled for the 8051 architecture. The creators of this commercial tool expect program size reductions of up to 25%. aiPop [Ferdinand 2001] is a more sophisticated post link-time code compactor for the C16x architectures. It includes, for example, constant propagation, peephole optimizations, code abstraction, procedure tail merging, and dead code elimination. The reported code size reductions ranged from 4% to 20%. Besides the techniques discussed in this article, SQUEEZE++ implements a broad range of whole-program analyses and optimizations. These include peephole optimization, copy propagation, load/store avoidance, constant propagation, dead code elimination, unreachable code elimination, inaccessible data elimination, inlining, and code layout optimizations. We refer the reader to Debray et al. [2000] and De Sutter et al. [2001] for a more detailed discussions.

Recently, we have started developing a retargetable framework for link-time optimization, compaction, and instrumentation called Diablo.[18] Our experience is that large parts of the whole-program analysis and optimization code can

[17]"Getting the Best Code Density for 8051 with Code Compressor." Go to `http://www.raisonance.com`.

[18]`http://www.elis.ugent.be/diablo`.

be reused for multiple target architectures. However, even with a retargetable code link-time editing framework, each architecture will always require a separate back-end.

Although the optimization and compaction techniques implemented in Diablo are not as extensive as those implemented in SQUEEZE++, preliminary results indicate that significant amounts of compaction and optimization can be achieved for a wide range of architectures, including ARM [De Bus et al. 2004], MIPS [Madou et al. 2004], and IA64 [Anckaert et al. 2004]. Moreover, these preliminary results indicate that significant amounts of compaction can be obtained even in code-size-conscious programming environments that are targeted specifically for the embedded market. In the Arm Developer Suite environment, for example, code size reductions averaging around 14.6% were obtained, while speedups up to 27% and energy reductions up to 23% were achieved with Diablo [De Bus et al. 2004]. On MediaBench programs compiled for the MIPS architecture and linked against the very small uClibc[19] standard C-library implementation, code size reductions averaging around 23% were achieved [Madou et al. 2004].

There has been a great deal of interest in dynamic binary optimization (see, for example, Hookway and Herdeg [1997] and Bala et al. [2000]). In these approaches, however, the data structures necessary for run-time execution monitoring and optimization incur nontrivial additional memory overheads, and hence are not suited for the goal of this work, which is memory footprint reduction of applications. For this reason, we do not discuss them further.

9.6 Other Program Compaction Techniques

The elimination of unused data from a program has been considered in Srivastava and Wall [1994] and in Sweeney and Tip [1998]. Srivastava and Wall, describing a link-time optimization technique for improving the code for subroutine calls in Alpha executables, observed that the optimization allows the elimination of most of the global address table entries in the executables. However, their focus was primarily on improving execution speed, and they did not investigate the elimination of data areas other than the global address table. In our previous work [De Sutter et al. 2001], the same optimizations were applied, but in a more general way and not limited to the global address table.

Sweeney and Tip [1998] focused on the removal of dead data members from classes in C++ programs. They reported a run-time high watermark (i.e., the largest object space needed during the execution of the program) reduction of 4.4% on the average. This was the result of the elimination of 12% of the data members. Ananian and Rinard [2003] proposed an extended set of data member transformations that included the transformation of instance fields into class fields if instance fields of all instances always have the same value. For a set of Java programs, these techniques reduced the maximum live heap size by as much as 40%.

For object-oriented programming languages, several application extraction techniques have been proposed to extract precisely those parts from the

[19]`http://www.uclibc.org`.

libraries and/or run-time environments that are needed by a specific application. For the dynamically typed language Self [Agesen and Ungar 1994], such extraction tools obtain higher compaction levels than SQUEEZE++. The Self tools are to some extent programming-language-specific however, and their starting point is a program containing the whole Self run-time environment.

Tip et al. [2002] achieved results for Java programs very similar to our results. Although most of their techniques were based on language-independent algorithms, for example, for building a call graph of a program [Tip and Palsberg 2000], some of the applied optimizations were language-dependent, such as the compaction of the constant pools in Java programs. Besides that, their techniques exploited the type information that is available in the Java bytecode. Such information is not available in native binary programs. Srivastava [1992] has studied the removal of unreachable procedures in object-oriented programming environments as well.

9.7 Template Instantiation Mechanisms

If identical template instantiations occur in multiple modules from a program, several techniques [Levine 2000] can be used to avoid linking these instantiations with the program multiple times.

With incremental linking, the compiler initially generates no instantiations at all. The linker notices that some code and/or data cannot be retrieved because referencing symbols cannot be resolved and feeds this back to the compiler that generates the necessary instantiations. This technique can only avoid linking multiple identical instantiations at the source code level into a program, as it is based on the names of symbols the linker does not find.

Another approach is the use of a so-called *repository*, a database consisting of all the instantiated templates. As in relational databases, all records are unique, thus avoiding multiple identical instances of the same template. As the records in the repository are identified by names, these repositories have the same limitations as incremental linking.

A third technique is used by the GNU compilers. All sections in object files that are generated for instantiating templates have a special tag: .gnu.linkonce.d. The linker compares these sections (again using symbol names only) and thus avoids multiple occurrences of the same template instantiation in the final program. Some Microsoft linkers use a comparable scheme, and in addition they compare the code in those sections to remove duplicate procedures with different names [Levine 2000].

It is clear that these techniques do not address the occurrence of identical or similar procedures as SQUEEZE++ does.

Very different techniques to avoid code growth because of using template-like language features have been extensively researched in the past, especially for the Ada programming language and its so-called *generics*. Most of the proposed techniques use polymorphism [Bray 1984; Rosenberg 1983] to avoid the need for static specialization of the generics used. The consequence of those techniques is that no optimized specializations are generated. On the contrary, overhead is introduced to implement the polymorphism.

10. CONCLUSIONS

This article presented link-time whole-program analyses and optimizations that target the program size overhead resulting from separate compilation. Special attention was given to the program information that a link-time compactor can exploit to build a program representation on which to apply the analyses and optimizations. To eliminate the particular overhead of duplicate code and data, scalable duplicate elimination techniques were presented. The cost-effectiveness of various important analyses, such as liveness analysis and constant propagation, was evaluated by means of the prototype link-time compactor SQUEEZE++.

On a set of 25 benchmark programs, consisting of all 12 SPECint2000, two SPECfp2000, five MediaBench, and six additional C++ programs, average code size reductions between 27% and 45% were obtained, depending on the type of benchmark. The average statically allocated data size reductions ranged between 6% and 27%. The highest code size reductions (up to 62%) were obtained for C++ programs that use a lot of templates, because the presented duplicate code elimination techniques work especially well on C++-template code. Combined, the code and data size reductions resulted in executables that were between 20% and 43% smaller. Furthermore, these size reductions could be approximated without degrading performance by using profile information consisting of basic block execution counts.

Finally, this article has demonstrated that a better cooperation between compilers and a link-time compactor will result in even smaller programs.

ACKNOWLEDGMENTS

The authors would like to thank Will Evans, Saumya Debray, Robert Muth, Dominique Chanet, Scott Meyers, and the anonymous reviewers for their extensive remarks and comments on earlier versions of this article. Finally, we would like to thank Kristof Beyls for his aid in getting all our benchmarks compiled.

REFERENCES

AGESEN, O. AND UNGAR, D. 1994. Sifting out the gold: Delivering compact applications from an exploratory object-oriented environment. In *Proceedings of the 1994 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA'94). 355–370.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.

ANANIAN, C. AND RINARD, M. 2003. Data size optimizations for Java programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES'03). 59–68.

ANCKAERT, B., VANDEPUTTE, F., DE BUS, B., DE SUTTER, B., AND DE BOSSCHERE, K. 2004. Link-time optimization of IA64 binaries. In *Proceedings of the Euro-Par 2004 Conference*. 284–291.

BAKER, B. AND MANBER, U. 1998. Deducing similarities in Java sources from bytecodes. In *Proceedings of the USENIX Annual Technical Conference*. Usenix, Berkeley, CA, 179–190.

BALA, V., DUESTERWALD, E., AND BANEJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'00). 1–12.

Beszédes, A., Ferenc, R., Gyimóthy, T., Dolenc, A., and Karsisto, K. 2003. Survey of code-size reduction methods. *ACM Comput. Surv. 35*, 3, 223–267.

Bray, G. 1984. Sharing code among instances of Ada generics. In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction* (CC'84). 276–284.

Burger, D. and Austin, T. 1997. The simplescalar toolset, version 2.0. Tech. Rep. TR-97-1342, University of Wisconsin-Madison, Madison, WI.

Chang, P., Mahlke, S., Chen, W., and Hwu, W. 1992. Profile-guided automatic inline expansion for C programs. *Softw. Pract. Exp. 22*, 5, 349–269.

Cheung, W., Evans, W., and Moses, J. 2003. Predicated instructions for code compaction. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. Lecture Notes in Computer Science, vol. 2826. Springer, Berlin, Germany, 17–32.

Clausen, L., Schultz, U., Consel, C., and Muller, G. 2000. Java bytecode compression for low-end embedded systems. *ACM Trans. Prog. Lang. Syst. 22*, 3 (May), 471–489.

Cohn, R., Goodwin, D., Lowney, P., and Rubin, N. 1997. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*. 17–24.

Cooper, K. and McIntosh, N. 1999. Enhanced code compression for embedded RISC processors. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'99). 139–149.

Corliss, M., Lewis, E., and Roth, A. 2003. A dise implementation of dynamic code decompression. In *Proceedings of the ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES'03). 232–243.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4, 451–490.

De Bus, B., De Sutter, B., Van Put, L., Chanet, D., and De Bosschere, K. 2004. Link-time optimization of ARM binaries. In *Proceedings of the 2004 ACM SIGPLAN-SIGBED Conference on Languages, Compilers and Tools for Embedded Systems* (LCTES'04). 211–220.

De Sutter, B., De Bus, B., and De Bosschere, K. 2002. Sifting out the mud: Low level C++ code reuse. In *Proceedings of the of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA'02). 275–291.

De Sutter, B., De Bus, B., De Bosschere, K., and Debray, S. 2001. Combining global code and data compaction. In *Proceedings of the of the ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES'01). 29–38.

De Sutter, B., De Bus, B., De Bosschere, K., Keyngnaert, P., and Demoen, B. 2000. On the static analysis of indirect control transfers in binaries. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA). 1013–1019.

De Sutter, B., Vandierendonck, H., De Bus, B., and De Bosschere, K. 2003. On the side effects of code abstraction. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems* (LCTES'03). 245–253.

Debray, S. and Evans, W. 2002. Profile-guided code compression. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'02). 95–105.

Debray, S., Evans, W., Muth, R., and De Sutter, B. 2000. Compiler techniques for code compaction. *ACM Trans. Prog. Lang. Syst. 22*, 2 (Mar.), 378–415.

Debray, S., Muth, R., and Weippert, M. 1998. Alias analysis of executable code. In *Proceedings of the ACM 1998 Symposium on Principles of Programming Languages* (POPL'98). 12–24.

Ernst, J., Evans, W., Fraser, C., Lucco, S., and Proebsting, T. 1997. Code compression. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'97). 358–365.

Evans, W. and Fraser, C. 2001. Bytecode compression via profiled grammar rewriting. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'01).

Ferdinand, C. 2001. Post pass code compaction at the assembly level for c16x. *CONTACT, Infineon Technologies Development Tool Partners 3*, 9, 35–36.

Flower, R., Luk, C.-K., Muth, R., Patil, H., Shakshober, J., Cohn, R., and P. G., L. 2001. Kernel optimizations and prefetch with the spike executable optimizer. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization* ("FDDO-4").

FRANZ, M. 1997a. 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. Lecture Notes in Computer Science, vol. 1222. Springer, Berlin, Germany, 263–276.

FRANZ, M. 1997b. Dynamic linking of software components. *IEEE Comput. 30*, 3 (Mar.), 74–81.

FRANZ, M. AND KISTLER, T. 1997. Slim binaries. *Commun. ACM 40*, 12 (Dec.), 87–94.

FRASER, C. 1999. Automatic inference of models for statistical code compression. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (PLDI'99). 242–246.

FRASER, C., MYERS, E., AND WENDT, A. 1984. Analyzing and compressing assembly code. In *Proceedings of the 1984 ACM SIGPLAN Symposium on Compiler Construction* (CC'84). Vol. 19. 117–121.

HOOKWAY, R. AND HERDEG, M. 1997. Digital FX!32: Combining emulation and binary translation. *Dig. Tech. J. 9*, 1, 3–12.

KÄSTNER, D. AND WILHELM, S. 2002. Generic control flow reconstruction from assembly code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems* (LCTES): *Software and Compilers for Embedded Systems* (SCOPES). 46–55.

KEMP, T. M., MONTOYE, R. M., HARPER, J. D., PALMER, J. D., AND AUERBACH, D. J. 1998. A decompression core for PowerPC. *IBM J. Res. Devel. 42*, 6 (Nov.), 807–812.

KIROVSKI, D., KIN, J., AND MANGIONE-SMITH, W. H. 1997. Procedure based program compression. In *Proceedings of the 30th Annual International Symposium on Microarchitecture* (MICRO-30).

KISSELL, K. D. 1997. Mips16: High-density mips for the embedded market. In *Proceedings of Real Time Systems* (RTS'97).

KOMONDOOR, R. AND HORWITZ, S. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 8th Static Analysis Symposium* (SAS).

KRINKE, J. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*. 301–309.

KRISHNASWAMY, A. AND GUPTA, R. 2002. Profile guided selection of ARM and thumb instructions. In *Proceedings of the of the 2002 ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems* (LCTES'02). 55–63.

KRISHNASWAMY, A. AND GUPTA, R. 2005. Dynamic coalescing for 16-bit instructions. *ACM Trans. Embed. Comput. Syst. 4*, 1 (Feb.), 3–37.

LAU, J., SCHOENMACKERS, S., SHERWOOD, T., AND CALDER, B. 2003. Reducing code size with echo instructions. In *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (CASES). 84–94.

LEKATSAS, H., HENKEL, J., CHAKRADHAR, S., JAKKULA, V., AND SANKARADASS, M. 2003. Coco: A hardware/software platform for rapid prototyping of code compression technologies. In *Proceedings of the 40th Conference on Design Automation* (DAC). 306–311.

LEVINE, J. 2000. *Linkers & Loaders*. Morgan Kaufmann, San Francisco, CA.

MADOU, M., DE SUTTER, B., DE BUS, B., VAN PUT, L., AND DE BOSSCHERE, K. 2004. Link-time optimization of MIPS programs. In *Proceedings of the 2004 International Conference on Embedded Systems and Applications* (ESA'04). 70–75.

MUTH, R. 1999. Alto: A platform for object code modification. Ph.D. dissertation. University Of Arizona, Tuscon, AZ.

MUTH, R., DEBRAY, S., WATTERSON, S., AND DE BOSSCHERE, K. 2001. Alto: A link-time optimizer for the compaq alpha. *Softw. Prac. Exp. 31*, 1 (Jan.), 67–101.

PALEM, K. V., RABBAH, R. M., VINCENT, J., MOONEY, I., KORKMAZ, P., AND PUTTASWAMY, K. 2002. Design space optimization of embedded memory systems via data remapping. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*. 28–37.

PETTIS, K. AND HANSEN, R. 1990. Profile-guided code positioning. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'90). 16–27.

PROEBSTING, T. 1995. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the ACM SIGPLAN-SIGACT 1995 Symposium on Principles of Programming Languages* (POPL'95). 322–332.

PUGH, W. 1999. Compressing Java class files. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'99). 247–258.

ROS, M. AND SUTTON, P. 2003. Compiler optimization and ordering effects on VLIW code compression. In *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (CASES). 95–103.

ROSENBERG, J. 1983. Generating compact code for generic subprograms. Ph.D. dissertation. Carnegie-Mellon University, Pittsburgh, PA.

SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. 2002. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering* (WCRE 2002). 45–54.

SNAVELY, N., DEBRAY, S., AND ANDREWS, G. 2003. Unscheduling, unpredication, unspeculation: Reverse engineering Itanium executables. In *Proceedings of the 2003 Working Conference on Reverse Engineering*. 4–13.

SRIVASTAVA, A. 1992. Unreachable procedures in object-oriented programming. *ACM Lett. Programm. Lang. Syst. 1*, 4 (Dec.), 355–364.

SRIVASTAVA, A. AND WALL, W. 1994. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'94). 49–60.

SWEENEY, P. AND TIP, F. 1998. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (PLDI'98). 324–323.

SZYMANSKI, T. G. 1978. Assembling code for machines with span-dependent instructions. *Commun. ACM 21*, 4, 300–308.

TIP, F. AND PALSBERG, J. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA'00). 281–293.

TIP, F., SWEENEY, P. F., LAFFRA, C., EISMA, A., AND STREETER, D. 2002. Practical extraction techniques for Java. *ACM Trans. Prog. Lang. Syst. 24*, 6, 625–666.

TRIMEDIA TECHNOLOGIES INC. 2000. *TriMedia32 Architecture*. TriMedia Technologies Inc, Sunnyvale, CA.

TURLEY, J. 1995. Thumb squeezes ARM code size. *Microproc. Rep. 9*, 4 (Mar.), 1–5.

WEGMAN, M. AND ZADECK, F. 1991. Constant propagation with conditional branches. *ACM Trans. Prog. Lang. Syst. 13*, 2 (Apr.), 181–210.

Received October 2002; revised August 2003, May 2004; accepted September 2004