By Bruno De Bus, Daniel Kästner, Dominique Chanet, Ludo Van Put, and Bjorn De Sutter

POST-PASS COMPACTION TECHNIQUES

Seeking to resolve many of the problems related to code size in traditional program development environments.

In the Java world, libraries are usually available in a bytecode representation that provides high-level semantic information, such as type information. This information is exploited by Java application extractors, as discussed in the article "Extracting Library-based Java Applications" in this section. By contrast, reusable code in the C/C++ world often is only distributed in the form of native machine code in object archives. Therefore, we cannot rely on the same techniques to avoid code reuse overhead in C/C++ programs. Code compaction techniques that operate on native assembly or machine code are the topic here; these techniques can be applied in the context of traditional development environments consisting of compilers and linkers.

In most C/C++ development tool chains, the compiler is the only tool that really optimizes code for speed or size [9]. The effect of the compiler optimizations depends on the amount of code available for inspection; techniques like register allocation and procedure inlining need a view of the entire program in order to achieve maximum effect. However, the scope of traditional compilers is often limited to one source code module at a time. Module boundaries are also optimization boundaries. The effectiveness of optimizations can be improved by extending their scope, that is, by increasing the size of the source code modules.

Typical linkers [8] also leave room for improvement since they perform few optimizations. Most linkers just examine the object files constituting a program to find out which external entities are referenced, search for these entities in libraries, and link the necessary library members into the program. Once all references are resolved, the linker assigns addresses to all the object files and library members, and patches references to these newly assigned addresses.

For most linkers, object file sections are the atomic building blocks of an executable. Hence, to assure that only referenced library code[1] is linked with the final program, an object file section should contain only one "linkable" entity,

[1] Unreferenced code is not limited to libraries. It is often found in large applications of which the code base has been maintained and adapted for several years by several different generations of programmers.

that is, one function or data element. Otherwise spurious entities can end up in the program that in turn require additional entities to be linked in or retained. There clearly is a goal conflict between compiler and linker: the compiler needs large compilation units generating large, optimized object file sections, while the linker needs fine-grained object files (having one program entity per section) to prevent unnecessary code bloat.

Partial smart linking solutions to this problem have existed for a long time. One quite common solution is to let the compiler generate multiple separately linkable sections from each source code file. This is only a partial solution, as the requirement of separate linkage by itself severely constrains the compiler optimization. Compiling several source code modules together is also only a partial solution, as it requires all code to be available in the source code format handled by the compiler. For closed-source third-party libraries and program parts written in assembly, it is inapplicable. Finally, embedded tool-chain builders often spend a lot of time fine-tuning their libraries to get good code size. While this tuning effort may minimize code size on average, it does not minimize the code size of any single application.

To further complicate the situation, modern software engineering advocates maximum source code reuse. A developer writing reusable code anticipates the contexts in which the code could be reused in the future. This involves generalizing the functionality, adding extra checks on parameters and adding specialized code to handle corner cases efficiently. In any single application, part of this extra code may be unnecessary, but the compiler cannot remove it, as its execution context is not known to the compiler.

It is clear from this discussion that traditional programming tool chains and modern software engineering result in programs containing lots of unnecessary code. In order to avoid this overhead, we need more advanced techniques that go beyond the smart linking of separately optimized pieces of code.

Post-Pass Whole-Program Optimization

Post-pass whole-program optimizers to a large extent solve this problem by applying an extra optimization pass on assembly or object code. Because this pass is applied after the regular optimization passes made by the compiler, the post-pass optimization usually has a larger scope: it can handle libraries, mixed-language code applications, and handwritten assembly.

Here, we present the most important techniques that post-pass compaction tools commonly use to produce more compact programs. To illustrate the potential of these techniques, three existing post-pass optimizers developed by the authors—the assembly optimizer aiPop and the link-time optimizers Squeeze++ and Diablo—are evaluated in three sidebars appearing at the end of this article.

The first broad class of post-pass compaction techniques consists of whole-program optimizations. Most of these optimizations are in fact local (in the sense that they transform only one small program fragment at a time) but they are called whole-program optimizations because they rely on the information collected by whole-program analyses.

Post-pass value analyses statically determine register values that, independent of a program's input, are either constant or can take values only from some restricted set. Value analyses can be used to remove computations whose result is statically known or to remove parts of the program that can never be executed, given the set of the values that are produced. Prominent examples of value analyses are constant propagation and interval analysis [9].

During program execution constant values are produced by literal operands, by loads from read-only data, and by ALU-operations on known data. A major source of constants, not available to the compiler, are the code and data addresses that are determined only when the linker lays out the code and data in the final program. The computation of these addresses and their use in the program can be optimized by the post-pass optimizers just like any other computation.

Whole-program liveness analysis [9], another important analysis, determines for each program point which registers contain live values, that is, values that may be needed later during the execution of the program. Its results can be used during post-pass optimization to remove unnecessary parameter-passing code and redundant register saving/restoring code that typically results from overly conservative adherence to calling conventions. Additionally, post-pass optimizers reapply many standard compiler optimizations, such as peephole optimization, copy propagation, useless code elimination, and strength reduction [9]. There are two reasons to do so.

First, most compile-time optimizations are performed on the intermediate representation of the program. By contrast, post-pass optimizers handle the machine code instructions of the program individually. As such they can perform more fine-grained optimizations, including architecture-dependent optimizations tailored to individual properties of some target processor. Examples are addressing mode and memory-access optimizations.

Another reason to reapply the compiler optimizations is that the typical post-pass whole-program analyses and optimizations result in new opportunities for

typical compiler optimizations. This might be because more free registers have become available or because the elimination of some execution paths has made computations (partially) redundant.

Eliminating Duplicated Code

The second broad class of post-pass compaction techniques seeks to eliminate duplicate code fragments in programs. Code duplication can originate, for example, from the use of C++ templates. Templates allow a programmer to write generic code once and then specialize it at compile time for multiple execution contexts, which are often based on type information. Unless care is taken, a program containing multiple different instantiations of a template method will contain a lot of duplicated code.

A number of techniques have been developed to avoid linking identical template instantiations with a program several times [8]; most use type information to compare instantiations, which is very unsatisfactory. Code that seems different at the source code level (because a pointer to a Shape object, for example, has a different type from a pointer to an Employee object) can be identical or very similar at the assembly level where all pointers are simple addresses. Type-based techniques will not detect such duplicates.

Other techniques directly compare the assembly code of template method instantiations. Most of these techniques are very coarse-grained, as they only avoid the duplication of whole identical method instantiations. They do not at all avoid duplicated code in almost identical or very similar instantiations.

To get rid of such code duplicates as well, post-pass procedural abstraction is very well suited. With procedural abstraction (also called outlining), multiple occurring identical assembly code fragments are abstracted into a new procedure. All original occurrences are replaced by a call to this procedure. Going further, nonidentical, but similar or functionally equivalent code can first be made identical by reordering instructions, renaming registers or parameterization, after which they can be abstracted as well [2].

A technique similar to procedural abstraction is tail merging. The main difference is that no procedures containing common code sequences are built; instead, they are encapsulated in code entered by normal jump instructions.

While the code originating from templates proves to be an ideal candidate for procedural abstraction and tail merging, these techniques can also be used on other code. For example, compilers tend to generate identical or similar code at the start and end of procedures to save registers on the stack and for other aspects of calling conventions. These procedure prologues and epilogues are ideal candidates for abstraction, as they frequently occur at well-known program points, thus easing their detection [1].

Other code abstraction and tail-merging opportunities originate from the frequent use of copy and paste by programmers, a technique that not only makes programs more difficult to maintain but also makes them unnecessarily large.

Diablo

Diablo (see www.elis.ugent.be/diablo) is a retargetable framework for link-time optimization. Binaries are transformed into a representation that encodes both target-independent and target-dependent attributes, thus enabling target-independent and target-dependent optimizations. Diablo back-ends for the ARM, MIPS, IA32, IA64, SuperH, and PowerPC architectures are available.

As opposed to Squeeze++, Diablo is still in an early development stage, and a lot of optimization and compaction techniques still need to be implemented. More important, however, Diablo was developed to operate in highly competitive embedded software development tool chains, such as the ARM Developer Suite, a widely used tool chain that is highly regarded for the small binaries it generates.

Benchmark	Relative code size after compaction	Benchmark	Relative code size after compaction
Adpcm	87.2%	G721	85.2%
Bitcount	87.6%	HelloWorld	87.8%
Djpeg	95.9%	Qsort	93.1%
Epic	81.1%	Susan	95.1%

DIABLO: SIZE OF THE BINARIES AFTER COMPACTION.

To demonstrate Diablo's (very preliminary) compaction potential, we evaluated it on some small programs (less than 100,000 instructions) taken from the Mibench and Mediabench suites. These programs were compiled with the ARM Developer Suite (version 1.1). The table here shows the size of the binaries after compaction, relative to the size of the original binaries.

These results indicate link-time compaction offers significant code size reductions (10% on average), even in environments already geared toward producing compact code. Also, contrary to common belief about whole-program optimization techniques, these techniques need not be too slow to be practically viable. All presented benchmarks were compacted by Diablo in several seconds. C

Building an Internal Representation

The construction of an internal program representation is a problem we did not mention so far. Yet it is critical for the success of post-pass compaction tools, as these tools process flat lists of machine-code instructions without high-level control flow constructs like loops or switch statements.

The first step of any post-pass compaction is the construction of a control flow graph [1, 7]. The nodes of this graph represent basic blocks of instructions, and its edges represent all possible control flow between the nodes. Computing this graph is straightforward, except for indirect control flow instructions where the target address is contained in a register. Such indirect control flow transfers are pessimistically approximated by assuming that every computable code address (for example, a code label in assembly code) is a possible target. While the resulting graph contains a lot of unre-

aiPop

aiPop (see www.AbsInt.com/aipop and [5]) is a commercial assembly-based post-pass optimizer for the C16x/ST10 processor family that performs a wide range of code optimizations. Quick retargeting to other processors is supported by an underlying hardware specification mechanism.

To speed up the time-consuming identification of repeated code sequences, aiPop analyzes the entire application and builds a pattern database. It is not necessary to rebuild the database in each optimization run; instead, when there are only minor changes in the application, a previously created pattern database can be used.

aiPop leaves most of the symbolic debug information intact and annotates the generated code with information about the applied transformations, as depicted in the example code fragment here. Only the debug information for factorized program parts must be discarded. However, since the extracted program parts are usually only a few instructions long, source-level debugging is only slightly influenced. All transformations of aiPop are deterministic, that is, fully reproducible. This property, together with the possibility of validating the correctness of the applied transformations by studying the generated annotations, has proven to be a key aspect to the industry acceptance of this tool.

Command line options are available that disable optimizations potentially degrading performance, so that customers can individually determine the trade-off between code size and performance. aiPop has reduced entire customer applications to between 95.2% (for a small application featuring highly optimized C code) and 79.61% (for a large mobile-phone application) of their original size. The latter result means that 25% more code and functionality can be packed into a flash memory of the same size.

Optimization Annotations—Simple Example

```
; file.c  21 a = 15; b = -3; l = 8L;
       MOVB RL3,#0Fh
;      MOV R13,#0FFFDh       ; -aipop166: --opt-mc
       MOV R15,#0fffdh       ; +aipop166: --opt-mc
       MOV R1, #08h
       MOV R2, #00h
; file.c  22 cdb = b; xg = c;
;      MOV R15,R13           ; -aipop166: --opt-mc
;      NOP                   ;-aipop166: --opt-nop; +aipop166: --opt-mc

; Explanation:
; Assembly comments generated by the compiler in debug mode are preserved. The string
; 'aipop166: --opt-mc' indicates that a transformation has been caused by the 'mc' module
; which collapses redundant move chains (b:=a; c:=b -> c:=a). If an instruction is
; removed, the annotation is prefixed by '-', if an instruction is added, it is prefixed
; by '+'.

; MOV R13,#0FFFDh is replaced by MOV R15,#0fffdh. MOV R15,R13 is replaced by a NOP the
; NOP is removed by optimization module --opt-nop which eliminates NOP instructions that are
; not necessary for timing reasons.
```

alizable execution paths, it is a good starting point for the whole-program analyses we discussed earlier.

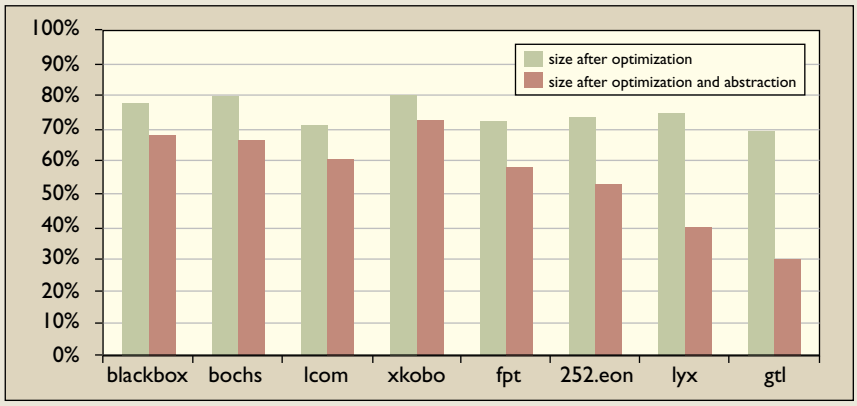
The results of those analyses can be used to eliminate some unrealizable execution paths from the graph, thus refining it into a more precise representation of the program. For example, when the target address of an indirect jump is found to be constant, we can replace many control flow edges by one. Because the resulting graph is more accurate, new opportunities arise for propagating constants, and thus call for even further refinement.

Sometimes such an iterative refinement, involving a new program analysis after each refinement, is computationally impractical. Often program slicing is more appropriate. A program slice consists of the program parts that potentially affect the values computed at some point of interest. Control flow refinement by program slicing determines statically known register values, but only by analyzing the code sequences responsible for computing the control flow targets [7].

With the refined graph, it is straightforward to detect code sequences that are never executed. It suffices to recursively mark all nodes of the control flow graph that are reachable from any entry point of the program. All nodes that are not marked reachable can be removed from the graph. Unreachable data can be removed in a comparable fashion. If none of the pointers to a data section of some object file are used throughout the program, that section can be removed from the program. Due to aliasing problems the analy-

Squeeze++

Squeeze++ (see www.elis.ugent.be/squeeze++ and [1—4]) is a link-time binary rewriter that uses only the information necessary to link a program. Source or assembly code need not be available; object files including relocation and symbol information suffice. Squeeze++ is a proof-of-concept research prototype that applies an entire range of whole-program optimizations and code-abstraction techniques. Its target architecture, the clean Alpha architecture, provided an excellent research platform to the researchers. The techniques implemented in Squeeze++ are not tied to the Alpha architecture, however.



SIZE OF THE BINARIES COMPACTED WITH SQUEEZE++ RELATIVE TO THEIR SIZE BEFORE COMPACTION.

Benchmark	Description	Code Size	Templates
blackbox	Fully functional lightweight window manager	183816	0%
bochs	Pentium processor simulator	407652	0%
lcom	"L" hardware description language compiler	122968	0%
xkobo	Arcade space shooter game	10112	0%
fpt	High-performance Fortran automatic parallelization tool	743256	7%
252.eon	Probabilistic ray tracer (from the SPECint2000 suite)	277896	10%
lyx	WYSIWYG word processor for scientific documents	3510188	23%
gtl	Test program from the Graph Template Library	388648	47%

DESCRIPTION OF A SET OF C++ BENCHMARK PROGRAMS. SIZES ARE IN BYTES; THE LAST COLUMN DEPICTS THE FRACTION OF THE CODE THAT ORIGINATES FROM TEMPLATES.

The compaction results for the set of C++ benchmarks listed in the table here are depicted in the figure in this sidebar. Some programs are reduced to about half or even one-third of their original size. This is the case for programs consisting largely of template code, and it can be seen that for those applications, code abstraction is vital in order to achieve good compaction. With respect to side effects on performance, we made the following observations. First, the compacted programs become 2%–30% faster, with the speedup averaging approximately 11%. The main reason for this speedup is that, with code abstraction limited to infrequently executed code only, the whole-program optimizations results in far less instructions being executed. We also counted up to 16% fewer instruction cache misses and up to 45% fewer data cache misses during simulations of relatively small caches (split level-1 instruction and data caches of 4KB).

The compaction times for these programs range from tens of seconds to about 15 minutes for the largest application. This is not extremely fast but is fast enough to be practically viable. C

ses needed to detect unreachable data are complex, yet they have proven to be quite effective [3].

Note that the detection of unreachable data can also affect code size; for code addresses stored in unreachable data, we do not have to make the pessimistic assumption that they can be the target of any indirect jump.

Side Effects on Performance

Although in some situations code size is the only important constraint on a program, often performance (execution speed and power consumption) is even more important. It is clear that whole-program optimizations in general do not only optimize code size, but also optimize performance, just like most compiler optimizations do. Post-pass optimizers aiming at performance [6, 10] not surprisingly overlap to a large extent with post-pass compaction tools.

The two exceptions are procedural abstraction and tail merging. Besides the insertion of control flow transfers, any non-trivial abstraction or tail merging involves the insertion of additional glue code and therefore of runtime overhead. As more instructions have to be executed and fetched from memory, code abstraction and tail merging often result in a slowdown and increased power consumption.

In most applications 10%–20% of the code is responsible for 80%–90% of the execution time. One important observation worth mentioning is that minimizing the size of a whole program through code abstraction or tail merging does not imply a size reduction of the frequently executed code. As a result, applying these techniques to minimize the size of an entire program does not necessarily imply better instruction cache performance. A simple solution to this problem is to separate frequently and infrequently executed code during code abstraction. An even simpler solution is to apply code abstraction and tail merging only to infrequently executed code. This solution avoids all possible kinds of overhead [4].

Discussion

We've described how post-pass compaction tools can solve many of the code-size-related problems in today's program development environments. The added value of post-pass compaction results mainly from the global scope of their analyses and transformations and the application thereof to machine code, where all details are exposed.

The three tools discussed in the sidebars here—in particular aiPop—prove that the discussed techniques are practically viable and robust. Our experience with Diablo and Squeeze++ enforces our belief that these techniques will gain importance in the future, but also that a lot of research remains to be done in this area.

An additional advantage of existing post-pass tools is that they can easily be integrated into existing tool chains: they are just another shackle. This ease of integration (or should we call it lack of integration) is not sustainable. The main limitation of today's post-pass optimizers is they do not have access to the detailed (semantic) information a compiler has access to. Better integration in existing tool chains and better preservation along the chain of the information collected by compilers should allow post-pass tools to improve upon their current performance. C

References

1. Debray, S., Evans, W., Muth, R., and De Sutter, B. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems 22,* 2 (Mar. 2000), 378–415.
2. De Sutter, B., De Bus, B., and De Bosschere, K. Sifting out the mud: Low level C++ code reuse. In *Proceedings of the SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Nov. 2002), 275–291.
3. De Sutter, B., De Bus, B., Debray, S., and De Bosschere, K. Combining global code and data compaction. In *Proceedings of the SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (June 2001), 29–38.
4. De Sutter, B., Vandierendonck, H., De Bus, B., and De Bosschere, K. On the side-effects of code abstraction. In *Proceedings of the SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems* (June 2003).
5. Ferdinand, C. Post-pass code compaction at the assembly level for C16x. *CONTACT: Infineon Technologies Development Tool Partners Magazine* (2001).
6. Kästner, D. ILP-based approximations for retargetable code optimization. In *Proceedings of the 5th International Conference on Optimization: Techniques and Applications* (2001).
7. Kästner, D. and Wilhelm, S. Generic control flow reconstruction from assembly code. In *Proceedings of the SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems Embedded Systems* (June 2002).
8. Levine, J. *Linkers and Loaders.* Morgan Kaufmann, 2000.
9. Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
10. Muth, R., Debray, S.K., Watterson, S., and De Bosschere, K. Alto: A link-time optimizer for the DEC Alpha. *Software-Practice and Experience 31*, 1 (Jan. 2001), 67–101.

Bruno De Bus (bdebus@elis.ugent.be) is a Ph.D. student in the Electronics and Information Systems department (ELIS) at Ghent University in Belgium.
Daniel Kästner (kaestner@absint.com) is a post-doctoral research associate at Saarland University, Saarbrücken, Germany, and a co-founder of AbsInt GmbH.
Dominique Chanet (dchanet@elis.ugent.be) is a Ph.D. student in the ELIS department at Ghent University in Belgium.
Ludo Van Put (lvanput@elis.ugent.be) is a Ph.D. student in the ELIS department at Ghent University in Belgium.
Bjorn De Sutter (brdsutte@elis.ugent.be) is a post-doctoral research fellow of the Fund for Scientific Research–Flanders in Belgium.