Whole-program optimization of binary executables

Bjorn De Sutter, Koen De Bosschere, Bruno De Bus, Bart Demoen, Peter Keyngnaert

Abstract— **Compiler optimization research has a long history and very complex algorithms were developed to optimize code. Unfortunately these algorithms, however complex they may be, suffer from the modular design of software. Compilers compile and optimize one source code module at a time. While they do so, they have no or few knowledge about the context in which the code will be executed. Even if summary information about all involved source code files is collected or if programs consist of one source code file only (which is not trivial for mixed-language programs) all kinds of code libraries will be linked with the programmer's code to produce the final executable binary. These libraries are often only available in object code and therefor no candidates for global analyses and optimizations by the compiler.**

A solution to this problem is to relegate part of the global optimizations to the linker, which has -by its nature- a global overview of the application. Unfortunately, the linker does not have access to the wealth of (semantic) information about the program that is available in the compiler. Therefor, a linker has to painstakingly reconstruct part of that information by analyzing object files. Nevertheless, it turns out to be possible to do this in a reasonable time, and to produce speedups of 19% for code that was already highly optimized by the compiler.

In the paper, we describe Alto, a link time optimizer for the alpha architecture. We describe Alto's framework, the analyses performed, the advantages and disadvantages of optimizing programs at link time, and an analysis of the speedups attained for the Spec benchmarks. We also describe additional program transformations applied by Squeeze, an Alto-derivative aimed at link-time code compression. Squeeze obtains additional link-time code compression of 23% on already compact compiler generated binaries.

Keywords— **linker, whole-program optimization, code compression,**

I. Introduction

High-level language programmers depend on compiler optimizations to produce efficient code. What efficient means depends on the programmer's objectives. It could mean fast execution, parallel execution, having a low power consumption or small memory footprint, etc.

Compiler optimization research has a long history and very complex algorithms were developed to optimize code. While most of the research has for a long time focused on execution speed[1] and parallelization, the increasing popularity of mobile and embedded systems is driving researchers more and more towards optimizations for code size, execution memory footprint and power consumption [2].

Algorithms targeting local optimizations have been extended to global optimizations. Unfortunately these algorithms, however complex they may be, suffer from the modular design of software. Compilers compile and optimize one source code module at a time. While they do so, they have no knowledge about the context in which the code will be executed. Even if summary information about all involved source code files is collected or if programs consist of one source code file only (which is not trivial for mixed-language programs) all kinds of code libraries will be linked with the programmer's code to produce the final executable binary. These libraries are often only available in object code and therefor no candidates for global analyses and optimizations by the compiler.

The linker combines the separately generated object files and libraries to working executable programs. It is only when a program has been linked, that a whole-program overview is available. Optimizing code at this level is however hampered by a lack of semantic information (alias analysis, stack behavior) that was gathered from the source code by the compiler but thrown away after dumping the object files.

In this paper we describe how Alto, A Link-Time Optimizer prototype for the 64-bit Alpha architecture, applies code optimizations on a very low level (that of already highly optimized linked binaries) and how this effectively optimizes these binaries beyond what compilers achieve. The opportunities for link-time optimization arise from

- the fact that static data structures and procedures have been placed in the program address space, making their addresses available for optimization,
- the fact that the whole program, including libraries, is available for inspection,
- architecture-specific computations become available for optimizations, which is not the case when intermediate code representations are used as in compilers.

In section II we describes Alto's framework, followed by a short description in section III of several analyses implemented to support the different optimizations Alto applies. In section IV some of the most important optimizations that Alto performs are described, together with some additional program transformations applied by Squeeze, an Alto-derivative aimed at code compression. The results that are obtained by these optimizations are presented and discussed in section V. Related work is shortly touched in section VI and final conclusions are drawn at the end of the paper.

II. Alto's framework

Alto is in fact a post link-time optimizer. It reads a binary executable, profiling information that is generated by an instrumented version of the binary and relocation information. After optimization, it dumps the new and

B. De Sutter, K. De Bosschere and B. De Bus are with the Electronics and Information Systems Department, Ghent University, Flanders. Koen De Bosschere is also a research associate with the Fund for Scientific Research – Flanders. E-mail: kdb@elis.rug.ac.be.

B. Demoen and P. Keyngnaert are with the Department of Computer Science, Katholieke Universiteit Leuven, Flanders.

The work described in this paper is funded by the Fund for Scientific Research - Flanders under grant 3G001998.

faster executable.

A. The Interprocedural Control Flow Graph

After reading the binary and disassembling the code, an interprocedural control flow graph is built (ICFG). This is one big graph with as vertices the basic blocks of the program and as edges the intraprocedural control flow transfer paths of the program and special call and return edges. Basic blocks consist of instructions. The intermediate code representation is a representation very close to the assembler level, but not static single-assignment.

The basic blocks and procedures are detected using a standard leaders detection algorithm[3]. Additional leaders are found using relocation information. All relocatable addresses that are stored in the data sections, can possibly be targets of indirect control flow transfers, since their address can be loaded into a register. At these indirect control flow transfer points, the targets are then not directly extractable from the code.

Therefor a so called *hell* node is added to the ICFG. This node behaves conservatively for all possible analyses and optimizations. Unresolved (indirect) control flow transfers are modeled using an edge to the hell node. Likewise, all leaders that could be the target of an unresolved control flow transfer are made reachable from the hell node by an additional edge in the ICFG.

Other control flow anomalies, such as with the `setjump` and `longjump` C-procedures or direct interprocedural branches, are modeled by adding compensating edges in the ICFG in such a way that all analyses and optimizations behave conservatively.

The use of a hell node and compensation edges is a very elegant way to solve problems arising from unknown or special control flow behavior. Instead of having to adapt all analyses and optimizations to deal with control flow anomalies for conservativeness and correctness, the conservativeness is incorporated in the behavior of the hell node and the compensating edges. Some properties of the hell node are that (1) it is always reachable, (2) all registers are live at its entry point, (3) it defines all registers with unknown contents (i.e. not constants).

Without going into too many details about the algorithms that we use to analyze these properties for the other program points, it is important to note that for (1) reachability analysis, (2) liveness analysis and (3) constant propagation, that are all fix-point calculation algorithms, it suffices to initialize the properties of the hell node before the fix-point algorithm is applied to guarantee conservativeness. From that moment on, the fix-point algorithms are not aware of the existence of the hell node.

B. Optimizing the binary

The actual optimization of the binary takes place in six phases:

1. *Simplifications* — Some early simplifications are performed, such as the removal of unreachable code and the removal of no-ops from the code to avoid clutter.
2. *Easy Optimizations* — Easy optimizations are performed, such as unreachable code elimination, copy propagation, constant propagation, dead code elimination, etc. These are all performed several times, because there is a phase ordering problem between them.
3. *Hard Optimizations* — Harder optimizations and optimizations that only need to be performed once, such as inlining functions with one call-site only, are performed only once.
4. *Easy Optimizations* — Again the easier optimization are performed for which extra possibilities could have been created by the harder optimizations.
5. *Code layout* — Basic blocks are reordered to optimize cache behavior.
6. *Code Scheduling* — Because the code is transformed severely, it needs to be rescheduled.

III. Analyses

To support the optimizations, several analyses are to be performed on the program. In this section, we'll describe the most important analyses and pay specific attention to the problems that arise when applying them on binary programs instead of on source or intermediary code the way compilers do.

A. Interprocedural Constant Propagation

Constant propagation is the propagation of constants produced in a program to the points where where they are consumed [1]. Compilers propagate constant values of variables from statements to statements, taking into account possible aliases.

There is no notion of variables in binaries. Binary programs work with registers and memory addresses. Statements in binaries are load/store operations or register operations. Due to the lack of variable-address bindings, retrieving alias or points-to information is a hard job. The alias information we have at our disposal is very limited. Because of that, we have for the time being found no conservative way to transform or optimize the data sections of binaries. There are however read-only sections of which the contents are known and available for constant propagation.

Taking these constraints into account, we have to limit the constant propagation to register contents only. It is clear that there is no aliasing between registers. Data memory is seen as a black-box except for read-only memory and stack save/restore pairs: loading data from constant addresses in read-only sections yields constant register contents, and registers that have their contents saved/restored using the stack (callee/caller saved) hold their (constant) values over the call/return edges.

Several flavors of constant propagation are implemented, reaching from simple local, i.e. intrablock, to call-site-specific conditional constant propagation. Experimentation has learned that the simplest flavors suffice to optimize the 64-bit address calculations that are addressed in section IV-A, while the most complex flavor is able to detect much more non-address constants, resulting in faster and smaller binaries [4].

B. Interprocedural Liveness Analysis

Liveness analysis detects which of the values produced by instructions in a program are used by other instructions. If the produced value is not used, i.e. it is dead, the producer can be eliminated.

Liveness analysis in Alto is implemented for both context-insensitive and context-sensitive analysis.

Context-insensitive analysis treats the ICFG as if it is nothing but a large CFG. Context-sensitive analysis treats call and return edges differently. Live registers over return edges are then only propagated over the corresponding call edges, not over all call edges to the callee. In other words, context-insensitive takes the meet-over-all-paths while context-sensitive takes the meet-over-all-valid-paths. See [5] for more details on these analyses.

C. Alias Analysis

Alias analysis determines whether or not two address expressions point to the same memory location. This information is necessary to be able to a.o. reorder load/store instructions for obtaining efficient instruction schedules.

There is an extensive body of work on alias analysis of various kinds. Most of the literature however is limited to high level analyses working on representations of source programs in terms of source language constructs. They typically disregard "nasty" features such as casting, pointer arithmetic and out-of-bounds array accesses. At the assembly level, we have to deal with all those nasty features, and our alias analysis has to be generic enough to deal with all possible assembly implementations of all higher-level language constructs of all possible languages, including hand-written assembly where the programmer does not adhere to standard implementation techniques.

Most techniques map load/store statements to a set of memory addresses. Although space-efficient methods exist to do so, this is not useful for a whole-program optimization for two reasons: the memory requirements would still be too high and for most of the load/stores, we have absolutely no idea about the addresses they access.

Therefor we limit our alias analysis to three simple cases, that have nevertheless proven to be quite efficient:

- two instructions that load/store data to/from constant and different addresses do not alias,
- if one of the instructions points to the stack while the other points to the global data area, there is no aliasing,
- two instructions i_1 and i_2 loading/storing data at addresses $d_1(r_1)$ and $d_2(r_2)$ that are found in the same basic block for which the base registers r_1 and r_2 (d_1 and d_2 are displacements) are computed by two (possibly empty) sequences of instructions such that $r_1 = c_1 + contents\_of(r_0)$ and $r_2 = c_2 + contents\_of(r_0)$ are non-aliasing if the data accessed at $c_1 + d_1$ and $c_2 + d_2$ do not overlap. To detect the register r_0 a simple backwards data flow algorithm is used.

All other load/store pairs are considered to be aliasing.

A more formal description of the alias analysis can be found in [6].

IV. Optimizations

This section discusses the most important optimizations applied by alto.

A. 64-bit Address Calculations

To access data or for indirect control flow transfers, addresses have to be produced. General-purpose computing is evolving towards 64-bit computing. Producing code or data addresses on a 64-bit architecture differs significantly from doing so on a 32-bit architecture. While for the latter architectures, two instructions using immediate operands suffice to fill a register (one for the lower 16 bits and one for the higher 16 bits), a comparable scheme would be too expensive for 64-bit (RISC) architectures where the instructions still are only 32-bits wide, limiting the width of the immediate operands.

While not many programs use the whole program space of a 64-bit architecture (32 bits already yields a 4GB address space), the compiler, on compiling one module, must assume that the final program will use the whole 64-bit address space.

This increases the inefficiency of modular programming for two reasons:

- *control flow* — the displacement between source and target of a control flow transfer is only known by the compiler for intramodular transfers. It is only for these transfers that the compiler can use efficient implementations using small displacements. For intermodular calls, a 64-bit address will have to be generated.
- *data access* — Since the compiler does not know how large the data segment of the final program will be, it must use full 64-bit addresses to access data.

A common solution for this is the use of a *global address table* (GAT). This table holds the addresses of all statically declared objects, data and procedures. To access an object (or to transfer control to it), its address is first loaded from the GAT into a register and from that register the data is accessed. For accessing the GAT, a *global pointer* is used. This is one special-purpose register that always points to the GAT. For most object files generated by the compiler, one GAT and one GP suffice, since the displacement that is possible from the GP in one (or two) instructions is wide enough to reach to the extends of the GAT.

On linking, the GATs from different object files are combined. The compiler can however not assume that one GAT and one GP will suffice for the whole program after linking. Therefor, after all intermodular control flow transfer targets, the GP has to be reset to point to the GAT corresponding to the object file where the target came from. This reset typically takes two instructions.

Summarizing, the compiler is forced to implement data accesses and intermodular control flow transfers indirectly via the GAT and GP and the GP-reset code has to be included in the object code after every possible intermodular control flow transfer.

Often however, programs are not that large. For the smaller (i.e. most) programs, one global pointer suffices. The executable's header indicates how many different

global pointers there are in the program. If there is only one, Alto removes all the global pointer resets in the code, since they do nothing else than recalculating the same value.

Accessing an object indirectly through the GAT is also often redundant. If the displacement between an object's address and the GAT is small enough (less or equal than 32-bits), it suffices to index the global pointer to access the data directly. This way Alto optimizes all data accesses to known (constant) addresses.

This also holds for procedure calls. Since the compiler does not know the displacement between a call-site and the callee for intermodular calls, these calls are implemented with an address load from the GAT and an indirect procedure call instruction. If Alto is able to determine the target of an indirect call, it is transformed to a direct procedure call if the displacement is small enough.

B. Unreachable code elimination

In compilers, unreachable code (code that will never be executed) usually arises from the fact that the programmer disables debugging code or as a result of other intraprocedural optimizations.

Unreachable code at link-time differs from that because its origins are fundamentally interprocedural: most of it is due to the inclusion of irrelevant library routines and to the propagation of interprocedural constants.

While unreachable code is never executed, removing it from the program has some advantages:
- it decreases the amount of code that the link-time optimizer has to process and therefor its time and space requirements,
- the elimination of non-executable paths can enable other optimizations,
- it can reduce the amount of cache pollution in cases where unreachable code fragments would have been loaded into the cache on loading nearby code that is to be executed,
- the latter holds even more for the virtual memory system, where unreachable code elimination can reduce the number of page faults.

Unreachable code elimination is performed using a fixpoint algorithm that marks basic blocks as executable if they have executable predecessors.

C. Optimization of Constant Value Computations

Constant value computations are detected by constant propagation and are simplified where possible by the following optimizations:
- *Immediate operands* — Small values can often be used as immediate operands to instructions, possibly resulting in the producers of the small values becoming dead. Commutativity of instructions is exploited where this is beneficial and for small negative constants, negation of the constant combined with negation of the instruction (e.g. subtraction becomes addition) is applied if possible.
- *Strength-reduction* — Conditional branches are strength-reduced to unconditional branches if the condition register holds a constant value or if, for call-site specific propagation, the branch is evaluated in the same direction for the data coming from all call-sites.
- *Copy Propagation* — If data is to be accessed at two memory addresses that are close enough to each other, it suffices to generate only one address pointing to their neighborhood. The displacement between the two can be encoded in the load/store instructions.
- *Idempotent Code Elimination* — The constant propagation algorithms mark idempotent instructions, i.e. instructions that replace a constant value in a register with the same constant. Such instructions are useless and can be killed. Whether an idempotent instruction should be killed depends on the liveness of the other producers of the constant value. Eliminating the instruction lengthens the liveness range of the values produced by the other producers, possibly making otherwise dead instructions (that have not yet been eliminated) live again.

More details about optimizing the constant value computations can be found in [4]

D. Load/store Avoidance

Next to the load instructions loading addresses from the GAT, a number of other loads (and sometimes stores) can often be eliminated at link-time:
- global variables of which the linker now knows the address can be stored in registers,
- after inlining, more detailed aliasing information may reveal that there is no aliasing between a pair of pointers, allowing the accessed data to be kept in registers,
- having a whole-program overview, overhead created by calling-conventions at procedure calls that actually is redundant can be detected, e.g., if there is callee-saved register that is dead at the call-site anyway.
- as a result of other optimizations, registers can be freed and then used to eliminate spill code.

If Alto detects a store followed by a load accessing the same address, it replaces the load by a register-register move. If the store subsequently becomes dead, it is removed (this rarely happens because of the limited memory liveness analysis). If the register where the stored value is found in is overwritten between the store and the load, Alto tries to find a third (free) register, to temporarily store the value.

Register save/restores at procedure boundaries are eliminated if the register stored on the stack is not changed during the execution of the procedure. Using a variation of shrink-wrapping, the save/restore instruction pair is moved away from to executable paths of the procedure that don't need the save/restore.

E. Inlining

There are two main reasons to inline procedures at link-time which are basically the same as for compilers (that lack the whole-program overview however):
- reducing the overhead of the procedure call/return (i.e. call instruction, return instruction, stack frame allocation, possibly register saves/restores),

- and to specialize procedures for specific call-sites.

Specializing procedures for specific call-sites can be beneficial in many ways: the cache behavior and branch prediction can be improved by better code-layout, aliasing relationships might enable faster schedules, constant arguments at the call-site that are not constant at other call-sites can be propagated into the callee, etc.

With inlining, possible code explosion has to avoided. Therefor Alto limits inlining to those callees for which at least one of the following hold:
- the body of the callee is smaller than the call/return overhead,
- or the call-site under consideration is the only call-site for the callee,
- or the call-site is hot (i.e. very frequently executed) and the memory footprint of hot code in the combined caller and callee does not exceed the instruction cache size.

F. Code Layout

Code layout is a very important issue. Especially for DSP based systems with heterogeneous memory systems (faster on-chip and slower off-chip memory, often split in several banks) code layout is important for execution speed and power consumption.

While the compiler can only decide on the relative position between pieces of code from the same module, the linker creates the global layout. Having the global layout and profiling information at its disposal, a link-time optimizer for general-purpose architectures can influence code layout for optimizing:
- *Branch prediction efficiency* — For older processors having only static branch prediction mechanism, code layout played an important role in optimizing branch prediction efficiency. Since modern processors, such as the Alpha 21164, use history-based dynamic branch prediction schemes, code layout (or the ordering of basic blocks) has fewer or even no influence on branch prediction efficiency anymore. Therefor this is not considered in Alto.
- *Control flow transfer penalty* — On pipelined architectures, even modern ones, control flow changes cost cycles, since the fetch of the next instruction to be executed takes place, while the control flow transfer instruction is being decoded. In the case of a transfer, the fetch will have to be redone. This is even the case for unconditional branches. Therefor Alto tries to minimize the number of unconditional branches and tries to orient conditional branches in such a way that the fall-through path is more likely followed than the branch-taken path. For indirect jumps, that are a.o. used for implementing `switch`-statements using address tables, the most frequently executed successor directly follows the jump in the code layout.
- *Instruction cache behavior* — The number of conflict misses (two instructions being mapped on the same cache location) can be reduced by carefully placing the hottest instructions on memory locations where will not interfere when loaded in the cache. Therefor, the order in which basic block sequences are located in memory is determined using an algorithm derived from the Hansen and Pettis algorithm [7]. The program is first split in hot and cold sections, i.e. code pieces that have high and low execution counts. The hot sections are then layouted close to each other to lower the chance that they cause conflicts in the instruction cache. This implies that a procedure's code is not necessarily layouted consecutively anymore.

G. Scheduling

On in-order superscalar architectures such as the Alpha 21164, the performance difference between good scheduled code and poorly scheduled code can be quite impressive.

Since the quality schedules generated by the compiler are severely "damaged" by the optimizations in Alto, new schedules have to be generated. Alto schedules code per extended basic block. An extended basic block is a tree of basic blocks with only one entry point. The advantage of scheduling per extended block is that operations can be moved over basic block boundaries and that interblock latencies are taken into account.

Combining global code layout with extended basic block scheduling results in an approach very similar to trace scheduling [8], [9].

H. Value Profiling

It sometimes happens that important optimizations are not applicable because some (necessary or sufficient) conditions are not fulfilled. The reason might be that the conditions do not always hold, that the implemented analyses are not precise enough to detect that they in fact always hold or that it is undecidable whether they always hold.

This is e.g. the case for some code kernels that can be scheduled much more effectively if there is no aliasing between a load and a store or if code could be specialized for some constant values.

This problem can be solved by inserting condition checks before a code fragment and providing two pieces of the fragment: one that is specialized for the case that the condition holds and another generic one.

Alto implements a five-phase process to optimize such opportunities:
1. *Instrumentation* — The binary is instrumented by Alto for profiling execution counts.
2. *Execution count profiling* — The instrumented binary is executed and execution counts are gathered.
3. *Opportunity seeking and Instrumentation* — Using the profile information, Alto looks for hot code fragments on which it cannot apply certain optimizations and tries to find a sufficient condition that should hold for the optimization to become applicable. This might be the fact that a register's value is a specific constant or some relationship between two registers, such as the fact that they do not alias. The program is then instrumented to keep track of the fraction of executions of the code fragment for which the condition holds.
4. *Value profiling* — The instrumented binary is executed and the inserted conditions are profiled.
5. *Code specialization* — If the necessary condition holds frequently, the condition test is inserted in the program and

two versions of the code fragment are inserted: a specialized one and a generic one.

Detecting the opportunities for value profiling is a very complex task and has not yet reached full wisdom. We refer to [10] for more details on the current status of this work.

I. Code Factoring

While optimizing for execution speed is the main goal of Alto, we are also working on Squeeze, an Alto-derivative aimed at code compression. Code compression becomes more and more important with the rise of embedded and mobile systems.

Squeeze is in fact a version of Alto that does not apply code transformations that increase the code size. Besides that it also applies code factoring, i.e. finding multiple-occurring sequences of instructions for which then one representative sequence is generated that, combined with the necessary control flow transfers, replaces all the originally occurring sequences.

A local code factoring scheme in Squeeze [11] looks for identical sequences of instructions in basic blocks. Register renaming is applied to make sequences that apply the same computations on different registers identical. If identical code sequences are found, they are abstracted to procedures. The code that is factored in this way often includes stack saves/restores of callee-saved registers.

The simple local scheme has been extended to single-entry/single-exit program regions: using dominators and postdominators these regions are located. For the comparison of different code sequences, a fingerprinting system is used: it lowers the memory usage and allows fast comparison. It allows not only to detect identical linear sequences of instructions, but also identical functional sequences that are mixed with other (independent thereof) instructions, e.g. because of instruction scheduling reasons.

V. Results

In this section, the speedups (Alto) and code size reductions (Squeeze) that our optimizations are able to achieve are presented and discussed.

A. Alto

Alto focuses on optimizing execution speed. The results (timings and speedups) for the SPECint95 benchmark suite are presented in table I.

The binaries for these measurements are generated using the Digital C compiler (V5.2-036) for Digital Unix v4.0. The target architecture is an Alpha 21164 in-order 64-bit processor with 8KB level-1 I-cache and 8KB level-1 D-cache. The level-2 cache is 96KB large, the level-3 cache 2MB. The target machine has 512 MB of main memory.

All binaries are generated with compiler flags `-O4 -Wl,-r -Wl,-z -Wl,-d -non_shared`. This way highly optimized statically linked executables are generated that include the relocation information[1]. Profiling of the SPECint95 benchmarks is performed using the training data sets, execution times are measured for the reference input sets: 7 runs are timed, of which the fastest and slowest are discarded and the average is taken over the remaining 5. (The maximum deviation among the 7 runs is typically less then 1% of the average.)

The *Base* version of the binaries is produced using exactly the flags mentioned in the previous paragraph. For the *Om* version the binaries are additionally optimized by the Om link-time optimizer [12], [13]. This optimizer is provided by Digital as part of its programmer's development environment and is shortly described in section VI. The *Ifo+FB+Om* version is the Om version with additional profile-guided intermodular optimizations applied.

Comparing the speedups obtained by Alto to those obtained by Om and the profile-guided intermodular optimizations performed by the compiler, it is clear that Alto obtains far better speedups.

The importance of different analyses, available information and optimizations can be derived from table II. It shows the relative geometric mean execution times compared to the base binaries for several optimizations by Alto where part of the optimizations are turned off. It shows that constant propagation and profiling information provide most opportunities for Alto to optimize the binaries. Inlining and instruction scheduling are clearly less important.

B. Squeeze

For optimizations by Squeeze, we compiled the SPECint95 benchmark and five embedded applications, obtained from the MediaBench benchmark suite from UCLA (`http://www.cs.ucla.edu/ leec/mediabench`) with gcc (version 2.7.2.2) using the flags `-Wl,-r -Wl,-z -Wl,-d -static`.

The code sizes for different executables can be found in table III. Compiling with `gcc -O0` applies no optimizations. Compiling with `gcc -O2` includes most if not all compiler optimizations that do not incorporate space-speed tradeoffs. This means that the resulting binaries are the smallest that the compiler can generate. They are on average 22% smaller than the non-optimized ones. This is comparable to the figures provided in [14]. Squeeze is able to take another 23% of those already optimized binaries.

Roughly 35% of our code size reductions come from code factoring, while the other 65% comes from the application of compiler optimizations. These are fundamentally interprocedural in their origins. If we split the whole program in user code and library code, we would notice that on average, their compressibility is comparable, with a few exceptions where user or library code is more compressible.

The overhead created by the compression using code factoring is the slower execution of the program itself resulting from the insertion of more procedure calls during the factoring and the fact that no no-ops are inserted in the code to produce faster schedules. There is no decompression overhead. Knowing that 65% of the code compression results from compiler optimizations, it should be no

[1]It is not possible to generate dynamically linked executables that contain relocation information for Digital Unix v4.0.

Program	Base T_{base}	Om T_{om}	Ifo+FB+Om T_{ifo}	alto T_{alto}	T_{om}/T_{base}	T_{ifo}/T_{base}	T_{alto}/T_{base}
compress	283.3	275.5	273.9	259.7	0.97	0.97	0.92
gcc	291.0	233.1	226.4	230.6	0.80	0.78	0.79
go	340.5	324.7	299.1	300.7	0.95	0.88	0.88
ijpeg	337.8	329.6	332.7	326.9	0.98	0.99	0.97
li	318.8	293.0	289.5	254.4	0.92	0.91	0.80
m88ksim	333.2	254.9	230.7	226.2	0.77	0.69	0.68
perl	246.9	210.4	203.9	182.6	0.85	0.83	0.74
vortex	497.7	388.3	395.9	317.6	0.78	0.80	0.64
Geometric mean					0.89	0.87	0.81

TABLE I
EXECUTION TIMES (SECS) AND RATIOS

Optimizations performed	geom. mean
Base binary	1.00
Alto without constant propagation	0.91
Alto without profiling information	0.89
Alto without optimizing constant value computations	0.87
Alto without profile-guided layout	0.87
Alto without memory access optimizations	0.86
Alto without inlining	0.83
Alto without instruction scheduling	0.83
Alto full optimization	0.81

TABLE II
RELATIVE GEOMETRIC MEAN OF THE EXECUTION TIMES FOR DIFFERENT VERSIONS OF ALTO COMPARED TO THE ORIGINAL BINARIES.

Program	gcc -O0 N_{nopt}	gcc -O2 N_{opt}	squeeze N_{sqz}	N_{opt}/N_{nopt}	N_{sqz}/N_{opt}
compress	21956	20997	16611	0.96	0.79
gcc	528353	338064	251655	0.64	0.74
go	134353	79563	64764	0.59	0.81
ijpeg	80760	56179	44669	0.70	0.80
li	44356	38792	28582	0.88	0.74
m88ksim	72563	52829	40493	0.73	0.77
perl	138394	102271	76008	0.74	0.74
vortex	205670	150403	109540	0.73	0.73
adpcm	18664	18344	14303	0.98	0.78
gsm	36245	30312	24167	0.84	0.80
mpeg2dec	35371	28033	21609	0.79	0.77
mpeg2enc	52551	41438	32809	0.79	0.79
rasta	97326	90191	65330	0.93	0.72
Geometric mean				0.78	0.77

TABLE III
CODE SIZES (# INSTRUCTIONS) AND COMPRESSION RATIOS BY SQUEEZE

surprise however that the compressed binaries can execute faster than the uncompressed ones, especially since instruction scheduling and inlining, two optimizations that have to deal with space-speed tradeoffs have shown not to affect Alto's performance very much. Table IV confirms that a speedup is obtained for a number of binaries, while for others a significant slow-down is noticed. On average however, the compressed binaries are still 6% faster than the original binaries.

VI. Related work

Several optimizing linkers have seen the day-light in recent years.

The first was proposed by David Wall [15]. His idea was to have the compiler provide hints to the linker for global (interprocedural) register allocation. The linker, having an overview of the whole program, decided which registers should not be spilled and the hints provided him with information to transform the program to reflect the decision.

OM [12], [13] is a post link-time optimizer by Compaq for their Alpha/Unix systems. Started as a post-linking system for optimizing the 64-bit calculations also targeted by our constant propagator, OM also performs invariant code motion out of loops and interprocedural dead code elimination. Spike [16], [17] optimizes executables for the Alpha/NT environment. It concentrates on global register allocation and global code layout: using profile information a code-layout is produced using so-called hot-cold optimizations to improve cache behavior.

MLD [18] and Vortex [19] are two whole-program optimizers for object-oriented languages. They focus on reducing the overhead created by virtual method invocation. Looking at the whole class hierarchy of a program, some of the virtual method invocations can be replaced by direct ones. These systems also reduce the performance penalty due to polymorphism by using profile information to optimize the method calls for the most frequently appearing object types.

More information about Alto and Squeeze and source code is available at `http://www.cs.arizona.edu`.

VII. Conclusion

However complex compiler optimizations may be, they suffer from the modular design of software. An optimizing linker, having a whole-program overview, can further optimize binaries despite the lack of high-level (semantic) information about the program. The prototype optimizing linkers Alto and Squeeze, aiming at execution speedup and code compression improve execution speed by up to 19% or compress binaries by 23%. This compression on average still results in a smaller speedup of 6%.

Acknowledgments

We are very grateful to Saumya Debray, Robert Muth and Scott Watterson for their many fruitful discussions on Alto and Squeeze over the last years.

References

[1] S. S. Muchnick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997, ISBN: 1-55860-320-4.

[2] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology*, Kluwer Academic Publishers, 1998, ISBN: 0-7923-8288-9.

[3] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986.

[4] B. De Sutter, B. De Bus, K. De Bosschere, B. Demoen, and P. Keyngnaert, "Optimizing binaries using link-time constant propagation," Submitted to IEEE International Conference on Automated Software Engineering (ASE00).

[5] R. Muth, "Register liveness analysis of executable code," Tech. Rep., Department of Computer Science, University of Arizona, 1997, Available from `http://www.cs.arizona.edu/alto`.

[6] S. Debray, Muth. R., and M. Weippert, "Alias analysis of executable code," in *Proc. 1998 ACM Symposium on Principles of Programming Languages (POPL)*, January 1998, pp. 12–24.

[7] K. Pettis and R.C. Hansen, "Profile-guided code positioning," in *SIGPLAN '91 Conference on Propgramming Language Design and Implementation*, June 1995, pp. 16–27.

[8] J.A. Fisher, "Global code generation for instruction-level parallelism: Trace scheduling-2," Tech. Rep. HPL-93-43, Hewlett-Packard, June 1993.

[9] J.A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, July 1981.

[10] R. Muth, S. Watterson, and S. Debray, "Code specialization based on value profiles," in *Proc. 7th. International Static Analysis Symposium (SAS 2000)*, 2000, (to appear).

[11] S. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compression," *ACM Transactions on Programming Languages and Systems*, 2000, Accepted for publication.

[12] A. Srivastava and D.W. Wall, "A practical system for intermodule code optimization at link-time," *Journal of programming Languages*, pp. 1–18, March 1993, Also available as WRL Research Report 92/06.

[13] A. Srivastava and W. Wall, "Link-time optimization of address calculation on a 64-bit architecture," in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 49–60, Also available as WRL Research Report 94/1.

[14] K.D. Cooper and McIntosh N., "Enhanced code compression for embedded risc processors," in *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, 1999, pp. 139–149.

[15] D.W. Wall, "Global register allocation at link time," in *Proceedings of the ACM SIGPLAN '86 conference on Compiler Construction*, 1986, pp. 264–275.

[16] R. Cohn, D. Goodwin, P.G. Lowney, and N. Rubin, "Spike: An optimizer for alpha/nt executables," in *USENIX Windows NT Workshop*, August 1997.

[17] D.W. Goodwin, "Interprocedural dataflow analysis in an executable optimizer," in *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, June 1997, pp. 122–133.

[18] M.F. Fernández, *A Retargetable, Optimizing Linker*, Ph.D. thesis, Princeton University, January 1996.

[19] G. Chambers, J. Dean, and D. Grove, "Whole-program optimization of object-oriented languages," Tech. Rep. 96-06-02, Department of Computer Science and Engineering, University Of Washington, June 1996.

Program	gcc -O2 N_{opt}	squeeze N_{sqz}	N_{sqz}/N_{opt}
compress	373.4	311.5	0.83
gcc	284.3	306.9	1.08
go	390.2	356.6	0.91
ijpeg	395.2	362.2	0.92
li	363.5	338.5	0.93
m88ksim	398.6	332.4	0.83
perl	268.2	254.2	0.95
vortex	532.9	606.1	1.14
adpcm	15.5	15.4	0.99
gsm	8.2	7.5	0.91
mpeg2dec	9.6	8.7	0.90
mpeg2enc	15.4	14.4	0.94
rasta	6.5	6.1	0.94
Geometric mean			0.94

TABLE IV
EXECUTION TIME (SECS) AND SPEEDUP WITH SQUEEZE