`alto` : A Link-Time Optimizer for the Compaq Alpha*

Robert Muth   Saumya Debray   Scott Watterson

Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{muth, debray, saw}@cs.arizona.edu

Koen De Bosschere

Vakgroep Elektronica en Informatiesystemen
Universiteit Gent
B-9000 Gent, Belgium
kdb@elis.rug.ac.be

Keywords : compilers, code optimization, executable editing

November 2, 1999

Abstract

Traditional optimizing compilers are limited in the scope of their optimizations by the fact that only a single function, or possibly a single module, is available for analysis and optimization. In particular, this means that library routines cannot be optimized to specific calling contexts. Other optimization opportunities, exploiting information not available before linktime such as addresses of variables and the final code layout, are often ignored because linkers are traditionally unsophisticated. A possible solution is to carry out whole-program optimization at link time. This paper describes `alto`, a link-time optimizer for the Compaq Alpha architecture. It is able to realize significant performance improvements even for programs compiled with a good optimizing compiler with a high level of optimization. The resulting code is considerably faster that that obtained using the OM link-time optimizer, even when the latter is used in conjunction with profile-guided and inter-file compile-time optimizations.

* The work of Robert Muth, Saumya Debray and Scott Watterson was supported in part by the National Science Foundation under grant numbers CCR-9502826, CCR-9711166, and CDA-9500991. Koen De Bosschere is a research associate with the Fund for Scientific Research – Flanders.

1 Introduction

Optimizing compilers for traditional imperative languages often limit their program analyses and optimizations to individual procedures [1]. This has the disadvantage that some possible optimizations may be missed because they depend on propagating information across procedure boundaries. However, even if a compiler implements interprocedural analyses (see, for example, [16, 17, 26, 32, 39]), the analyses and optimizations possible are limited to code that is available for examination at compile time. This means that code involving calls to library routines, to procedures defined in separately compiled modules, and to dynamically dispatched "virtual functions" in object-oriented languages (in the case where the virtual function is never overridden), cannot be effectively optimized. Other optimizations, e.g., to reduce the cost of address computations [37] require information not available at compile time.

A possible solution is to carry out program optimization when the *entire* program—library calls and all—is available for inspection: that is, at link time. While this makes it possible to address the shortcomings of the traditional compilation model, it gives rise to its own problems, for example:

- Machine code usually has much less semantic information than source code, which makes it much more difficult to discover control flow or data flow information. As a simple example, even for simple first-order programs (i.e., where functions are not treated as data and passed around, e.g., in the form of closures in languages such as Scheme or ML, procedure parameters in languages such as Pascal, or using function pointers as in C), control flow analysis of executable files can be difficult because determining the extent of jump tables, and hence the possible targets of the code derived from `case` or `switch` statements, can be difficult; at the source level, by contrast, the corresponding problem is straightforward.

- Compiler analyses are typically carried out on representations of source programs in terms of high-level source language constructs. Here, "nasty" features are either infrequent, or result in non-standard-conforming programs whose observable behavior are not guaranteed to be preserved under optimizations. For example, explicit nontrivial pointer arithmetic—i.e., beyond simple increment or decrement operations on pointers—are usually not frequently encountered, while out-of-bounds array accesses typically result in non-standard-conforming programs. Because of this, compiler analyses can either handle them very conservatively—essentially, giving up when such features are encountered—or adhere strictly to the language semantics, e.g., by assuming that an array access `a[i]` addresses only elements of the array `a`, thereby simply ignoring potentially non-standard-conformant constructs such as out-of-bounds accesses. Neither alternative is difficult to implement or has a significant adverse impact on the extent of optimization achieved for most programs.

  At the level of executable code, by contrast, all we have are the nasty features. Nontrivial pointer arithmetic is ubiquitous, both for ordinary address computations and for manipulating tagged pointers. If the number of arguments to a function is large enough, some of the arguments may have to be passed on the stack. In such a case, the arguments passed on the stack will typically reside at the top of the caller's stack frame, and the callee will "reach into" the caller's frame to access them: since the stack frame is typically accessed as an array of words indexed by the stack (or frame) pointer, this is nothing but an out-of-bounds array reference. Unfortunately, source-level approaches to handling such features are no longer adequate at the level of executable code: treating nontrivial pointer arithmetic conservatively by giving up on them has a significant adverse impact on optimization, while ignoring the effects of out-of-bounds array accesses can cause incorrect optimization to be carried out.

- Executable programs tend to be significantly larger than the source programs they were derived from. Coupled with the lack of semantic information present in these programs, this means that sophisticated analyses that are practical at the source level may be overly expensive at the level of executable code because of their time or space requirements.

This paper describes a link-time optimizer that we have built for the Alpha architecture. Our system, which we call `alto` ("a link-time optimizer"), reads in an executable file produced by the linker (we currently support

Digital UNIX ECOFF binaries; a version for Elf binaries under Linux has been developed and is currently being tested), as well as execution profile information (optional),[1] carries out various analyses and optimizations, and produces another executable file. Experiments indicate that even though it currently implements only relatively simple analyses—for example, checks for pointer aliasing are only implemented in the most rudimentary and conservative way—the performance of the code generated by the system is considerably better than that generated by the `om` link-time optimizer [36] supplied by the vendor.

The remainder of the paper is organized as follows: Section 2 gives a brief overview of the Alpha processor. Section 3 describes the overall organization of `alto`. Section 4 discusses how control flow analysis is carried out. Section 5 describes the analyses carried out by `alto`, Section 6 describes the optimizations that are performed, and Section 7 gives performance results. Section 9 summarizes work related to ours. Finally, Section 10 concludes.

`Alto` can be downloaded free of charge from `http://www.cs.arizona.edu/alto`.

2 The Alpha Architecture: an Overview

The Alpha is a conventional superscalar RISC processor with 64-bit words and 32-bit instructions. It has thirty-two 64-bit integer registers (registers `$0` ... `$31`) and thirty-two floating-point registers (registers `$32` ... `$63`). Of these, register `$31` is hard-wired to the integer value 0, while `$63` is hard-wired to the floating-point value 0.0. Additionally, "standard usage" of these registers is as follows:

Integer Registers	*Floating-Point Registers*	*Usage*
`$0`	`$32`, `$33`	return values of functions
`$1`–`$8`, `$22`–`$25`, `$27`–`$28`	`$42`–`$47`, `$54`–`$62`	scratch registers
`$9`–`$15`	`$34`–`41`	callee-saved registers
`$16`–`21`	`$48`–`$53`	argument registers for function calls
`$26` (`ra`)		return address register for function calls
`$29` (`gp`)		"global pointer" register
`$30` (`sp`)		stack pointer

Of these, the use of the global pointer register gp (`$29`) deserves some explanation. On a typical 32-bit architecture, with 32-bit instruction words and 32-bit registers, a (32-bit) constant is loaded into a register via two instructions, one to load the high 16 bits of the register and one for the low 16 bits; in each of these instructions, the 16 bits to be loaded are encoded as part of the instruction word. However, since the Alpha has 32-bit instructions but 64-bit registers, this mechanism is not adequate for loading a 64-bit constant (e.g., the address of a procedure or a global variable) into a register. Instead, such constants are collected into one or more *global address tables*, one for each separately compiled module. The generated code accesses this table via the `gp` register, together with a 16-bit displacement. Accessing a global object involves two steps: first, the address of the object is loaded from the global address table; this is then used to access the object referred to, e.g., to load from or store to a global variable, or jump to a procedure.

3 System Organization

The execution of `alto` can be divided into five phases. In the first phase, an executable file (containing relocation information for its objects) is read in, and an initial, somewhat conservative, inter-procedural control flow graph is constructed. In the second phase, a suite of analyses and optimizations is then applied iteratively to the program. The activities during this phase can be broadly divided into three categories:

Simplification : Program code is simplified in three ways: dead and unreachable code is eliminated; operations are normalized, so that different ways of expressing the same operation (e.g., clearing a register) are rewritten, where possible, to use the same operation; and no-ops, typically inserted for scheduling and alignment purposes, are eliminated to reduce clutter.

[1] `Alto` can use either basic block profiles, generated using the vendor-supplied *pixie* tool, or basic block and edge profiles that it can itself generate; we are currently extending the system to also generate value profiles [9] at specific points of interest.

Analysis : A number of analyses are carried out during this phase, including register liveness analysis, constant propagation, stack usage patterns, and jump table analysis.

Optimization : Optimizations carried out during this phase include standard compiler optimizations such as peephole optimization, branch forwarding, copy propagation, and invariant code motion out of loops; machine-level optimizations such as elimination of unnecessary register saves and restores at function call boundaries; architecture-specific optimizations such as the use of conditional move instructions to simplify control flow; as well as improvements to the control flow graph based on the results of jump table analysis.

This is followed by a function inlining phase. The fourth phase repeats the optimizations carried out in the second phase to the code resulting from inlining. The final phase carries out profile-directed code layout [30], instruction scheduling, and insertion of no-ops for alignment purposes, after which the code is written out.

`Alto` carries out inlining because there may be opportunities for inlining at link time, e.g., across module and library boundaries, that may not have been present at compile time. The reason the simplification and optimization phases are performed twice, before and after inlining, is that they influence inlining and are influenced by it. For example, whether or not a function call will be inlined depends, in part, on the size of the callee, which is affected by dead and unreachable code elimination prior to inlining. These, in turn, are affected by optimizations such as copy propagation and constant folding. For example, inter-procedural constant propagation and constant folding prior to inlining can propagate the value of a constant argument into a library routine; this can then allow the outcome of a conditional branch in that routine to be statically determined, and the subsequent removal of unreachable code can reduce the size of that routine to the point where it gets inlined into one or more call sites. The inlining phase, in turn, can give rise to further opportunities for optimizations. For example, most of the optimizations within `alto` are conservative in their treatment of function calls in that they assume that the callee may read or write to any memory location; inlining exposes the memory access behavior of the inlined routine and can thereby enhance the effects of many of these optimizations.

4 Control Flow Analysis

Traditional compilers generally construct control flow graphs for individual functions, based on some intermediate representation of the program. The determination of intra-procedural control flow is not too difficult; and since an intermediate representation is used, there is no need to deal with machine-level idioms for control transfer. As a result, the construction of a control flow graph is a fairly straightforward process [1].

Things are somewhat more complex at link time because machine code is harder to decompile. The algorithm used by `alto` to construct a control flow graph for an input program is as follows:

1. The start address of the program appears at a fixed location within the header of the file (this location may be different for different file formats). Using this as a starting point, the "standard" algorithm [1] is used to identify leaders and basic blocks, as well as function entry blocks. The relocation information of the executable is used to identify additional leaders which would otherwise not be detected (eg. jump table targets) and those basic blocks are marked relocatable. At this stage `alto` makes two assumptions: (*i*) that each function has a single entry block; and (*ii*) that all of the basic blocks of a function are laid out contiguously. If the first assumption turns out to be incorrect, the flow graph is "repaired" at a later stage; if the second assumption does not hold, the control flow graph constructed by `alto` may contain (safe) imprecisions, and as a result its optimizations may not be as effective as they could have been.

2. Edges are added to the flow graph. Whenever an exact determination of the target of a control transfer is not possible, `alto` estimates the set of possible targets conservatively, using a special node $\mathsf{B}_{unknown}$ and a special function $\mathsf{F}_{unknown}$ that are associated with the worst case data flow assumptions (i.e., that they use all registers, define all registers, etc.). Any basic block whose start address is marked as relocatable is considered to be a potential target for a jump instruction with unresolved target, and has an edge to it from $\mathsf{B}_{unknown}$; any function whose entry point is marked as relocatable is considered to be potentially a target of

an indirect function call, and has a call edge to it from $\mathsf{F}_{unknown}$. Any indirect function call (i.e., using the `jsr` instruction) is considered to call $\mathsf{F}_{unknown}$ while other indirect jumps are considered to jump to $\mathsf{B}_{unknown}$.

3. Inter-procedural constant propagation is carried out on the resulting control flow graph, and the results used to determine addresses being loaded into registers. This information, in turn, is used to resolve the targets of indirect jumps and function calls: where such targets can be resolved unambiguously, the edge to $\mathsf{F}_{unknown}$ or $\mathsf{B}_{unknown}$ is replaced by an edge to the appropriate target.

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

5. Finally, `alto` attempts to resolve indirect jumps through jump tables, which arise from `case` or `switch` statements. This is done as part of the optimizations mentioned at the beginning of this section. These optimizations can simplify the control and/or data flow enough to allow the extent of the jump table to be determined. The essential idea is to use constant propagation (Section 5.1) to identify the start address of the jump table, and the bounds check instruction(s) to determine the extent of the jump table. The edge from the indirect jump to $\mathsf{B}_{unknown}$ is then replaced by a set of edges, one for each entry in the jump table. If all of the indirect jumps within a function can be resolved in this way, any remaining edges from $\mathsf{B}_{unknown}$ to basic blocks within that function are deleted.

5 Program Analysis

Once the flow graph has been constructed for a program, it is subjected to various dataflow analyses, the most important of which are described here.

5.1 Interprocedural Constant Propagation

There are generally more opportunities for interprocedural constant propagation at link time than at compile time. There are two reasons for this: first, the entire program, including all the library routines, is available for inspection; and second, at link time it is possible to detect and deal with architecture-specific computations that are not visible at the intermediate code representation level typically used by compilers for most optimizations. An example of the latter case is the computation of the gp register on the Alpha processor: the value of this register is generally recomputed at the entry to each function as well as on return from every function call, but in many cases the recomputation is unnecessary and can be eliminated by propagating the value of the register through a program. It should be noted that this optimization cannot be carried out at compile time since the value of gp is only determined at link time.

The analysis used in `alto` is essentially a standard iterative constant propagation, limited to registers but carried out across the control flow graph of the entire program. This has the effect of communicating information about

[2]In some architectures, the callee may explicitly manipulate the return address under some circumstances, e.g., part of the SPARC calling convention is that in some cases there is an extra word immediately following the call instruction, and in these cases, the callee increments the return address to skip over this word (we are grateful to an anonymous referee for pointing this out to us). Such situations do not arise in the Alpha architecture, and are not handled by `alto`.

Program	*No. of instructions*		Evaluated/Total
	Total	Evaluated	
compress	20707	3140	0.152
gcc	353002	67352	0.191
go	83929	14661	0.175
ijpeg	62639	7470	0.119
li	40832	7464	0.183
m88ksim	53498	10576	0.198
perl	107229	20920	0.195
vortex	155030	39204	0.253
Geometric Mean:			0.180

Table 1: Efficacy of Interprocedural Constant Propagation

constant arguments from a calling procedure to the callee. To improve precision, `alto` attempts to determine the registers saved on entry to a function and restored at the exit from it: if a register r that is saved and restored by a function in this manner contains a constant c just before the function is called, then r is inferred to contain the value c on return from the call.[3]

The results of constant propagation, after all optimizations have been carried out, are shown in Table 1. The column labelled "Total" gives the (static counts for) the total number of instructions in each program (after unreachable code elimination—see Section 6.1), while the column labelled "Evaluated" gives the number of instructions whose operands and result could be determined at link time. It can be seen that, on the average, it is possible to evaluate about 18% of the instructions of a program at link time. However, this does not mean that these 18% of the instructions in a program can be removed by `alto`, since very often the instructions whose outcome can be evaluated ahead of time represent address computations for accessing arrays or records, or for function calls. This information can, nevertheless, be used to advantage in many cases, e.g., by replacing indirect function calls with direct calls, or register operands by immediate operands.

As shown in Figure 1, this analysis has a profound impact on the performance of the generated code. Turning off this analysis results in an overall slowdown of over 10% on the SPEC-95 benchmarks, with some programs, such as *m88ksim*, *perl*, and *vortex* suffering slowdowns of 15–20%. The reason for this impact, in great part, is that many control and data flow analyses rely on the knowledge of constant addresses computed in the program. For example, the code generated by the compiler for a function call typically first loads the address of the called function into a register, then uses a `jsr` instruction to jump indirectly through that register. If constant propagation can be used to determine that the address being loaded is a fixed value, and the callee is not too far away, the indirect function call can be replaced by a direct call using a `bsr` instruction: this is not only cheaper, but also vital for the construction of the inter-procedural control flow graph of the program and for other optimizations such as inlining. Another example of the use of constant address information involves the identification of possible targets of indirect jumps through jump tables: unless this can be done, an indirect jump must be assumed as being capable of jumping to any basic block of a function,[4] which can significantly hamper optimizations. Finally, knowledge of constant addresses is useful for optimizations such as the removal of unnecessary memory references (Section 6.3) and strength reduction in constant computations (Section 6.2).

[3]Unfortunately, we cannot rely on the calling conventions being observed: hand-written assembly code in libraries does not always obey such conventions, and compilers may ignore them when doing interprocedural register allocation.

[4]More precisely, any basic block that is marked as "relocatable."

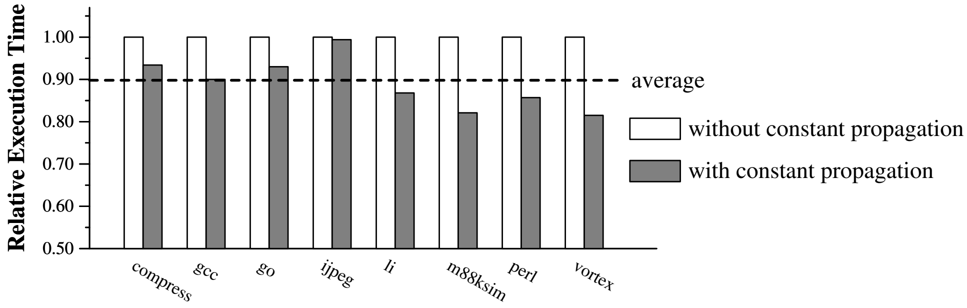

Figure 1: Performance impact of interprocedural constant propagation

5.2 Interprocedural Liveness Analysis

Interprocedural dataflow analyses can be either *context-insensitive* or *context-sensitive*. Context-insensitive analyses simply combine the control flow graphs for individual procedures into a single large graph and analyze this using standard intra-procedural techniques, without keeping track of which return edges correspond to which call edges. This has the advantages of simplicity and efficiency: nothing special needs to be done to handle interprocedural control flow, and a procedure does not have to be re-analyzed for its various call-sites [6, 7, 16, 32]. The problem is that such analyses can suffer from a loss of precision because they can explore execution paths containing call/return pairs that do not correspond to each other and therefore cannot occur in any execution of the program. Context-sensitive analyses, by contrast, avoid this problem by maintaining information about which return edges correspond to which call sites, and propagating information only along realizable call/return paths [17, 26, 39]. The price paid for this improvement in precision is an increase in the cost of analysis.

`Alto` implements a relatively straightforward interprocedural liveness analyses [1], restricted to registers, and extended to deal with idiosyncracies of the Alpha instruction set. For example, the `call_pal` instruction, which acts as the interface with the host operating system, has to be handled specially since the registers that may be used as through this instruction are not visible as explicit operands of the instruction: our implementation currently implements this using the node $\mathsf{B}_{unknown}$ mentioned in Section 4. The conditional move instruction also requires special attention as the destination register has to be considered as a source register as well. The remainder of this section gives a high-level overview of our liveness analysis.

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

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

2. $byPass[f]$. The set of registers which, if live at n_r, will also be live at n_c.

There is some flexibility in the choice for $byPass[f]$. Srivastava and Wall choose $byPass[f]$ to be the complement of the set of registers that are guaranteed to be dead at entry to f [36]. The problem with this is that it introduces a mutual dependency between the *byPass* and *mayUse* sets, which complicates the flow equations. Goodwin chooses $byPass[f]$ to be $\overline{mustDef}[f]$, the complement of the set of registers that will necessarily be defined by f: this avoids the mutual dependency problem mentioned [21]. In general, however, it is not hard to see that any set which lies between $\overline{mustDef}[f]$ and $\overline{mustDef}[f] \cup mayUse[f]$ is a valid candidate for $byPass[f]$. Our choice for $byPass[f]$ is

Program	*Load Instructions Executed* ($\times 10^6$)			Triv/C-Ins	Triv/C-Sens
	Trivial (Triv)	*Context-insensitive* (C-Ins)	*Context-sensitive* (C-Sens)		
compress	12.069	12.069	11.706	1.000	0.970
gcc	11.750	11.464	11.160	0.976	0.950
go	19.706	18.850	17.897	0.957	0.908
ijpeg	20.116	20.000	19.955	0.994	0.991
li	18.102	17.948	17.628	0.991	0.974
m88ksim	15.506	15.028	14.469	0.967	0.933
perl	12.616	12.267	11.930	0.972	0.946
vortex	24.504	24.048	23.326	0.981	0.952
Geometric Mean				0.980	0.953

Table 2: Effect of Liveness Analysis on Load Instructions Executed

a superset of Goodwin's, and results in more uniform dataflow equations that are somewhat simpler to implement [29].

Our analysis proceeds in three phases. The first two phases compute summary information for functions, i.e., their *mayUse* and *byPass* sets; the third phase then uses this information to do the actual liveness computation. While the first two phases can be carried out in parallel, doing them sequentially reduces the amount of space used, though possibly at the cost of increased execution time. Our implementation carries out the phases sequentially in order to conserve space.

It turns out that even context-sensitive liveness analyses may nevertheless be overly conservative if they are not careful in handling register saves and restores at function call boundaries. Consider a function that saves the contents of a register, then restores the register before returning. A register r that is saved in this manner will appear as an operand of a `store` instruction, and therefore appear to be used by the function; in the subsequent restore operation, register r will appear as the destination of a `load` instruction, and therefore appear to be defined by the function. A straightforward analysis will therefore infer that r is used by the function before it is defined, and this will cause r to be inferred as live at every call site for f. To handle this problem, `alto` attempts to determine, for each function, the set of registers it saves and restores.[5] If the set of callee save registers of function f, $save[f]$, can be determined we can use it to make the analysis somewhat less conservative by removing this set from $mayUse[f]$ and adding it to $byPass[f]$ whenever those values are updated during the fixpoint computation.

Ultimately, the utility of various analyses should be measured by the extent to which they enable optimizations to be carried out. In particular, analyses that attain improved precision at the cost of increased complexity should be justified by the additional code optimizations that become possible as a result of the improvement in precision. Table 2 compares context-insensitive and context-sensitive versions of our interprocedural register liveness analyses with respect to the reduction in the number of load and store instructions executed; the column marked *Trivial* corresponds to the base case, i.e., where no liveness information is available. It can be seen that our liveness analysis leads to a reduction in the number of loads from memory by about 2.5–5%, with the *go* program achieving a reduction of over 9%. Compared to a simple context-insensitive analysis, the context-sensitive liveness analysis yields an additional improvement of about 2.5–3%.

[5] We do not make *a priori* assumptions that a program will necessarily respect the calling conventions with regard to callee-saved registers: this is safe, though possibly conservative.

Program	Original (no. of instrs)	Unreachable (no. of instrs)	Unreachable/Original
compress	25097	4391	0.175
gcc	367760	14759	0.040
go	89346	5418	0.061
ijpeg	74307	11669	0.157
li	46117	5286	0.115
m88ksim	59656	6159	0.103
perl	114782	7554	0.066
vortex	186655	31626	0.169
Geometric Mean:			0.098

Table 3: Experimental Results: Unreachable Code Elimination

6 Optimizations

This section describes some of the more important optimizations implemented within `alto`. To maintain continuity, with each such optimization we discuss its performance impact; our experimental methodology is described in Section 7, while the raw data regarding execution times are presented in Table 8 in Appendix A. The performance impact of a particular optimization is measured by comparing the execution speeds attained when all optimizations are turned on against that attained when only that optimization is turned off. The details of the methodology used for these experiments, including the benchmarks, compiler options, and hardware processor used, are given in Section 7. It should be noted that because of interactions between different optimizations, the overall performance improvement for a program is not usually the same as the sum of the improvements for individual optimizations.

6.1 Unreachable Code Elimination

In compilers, unreachable code—i.e., code that will never be executed—typically arises due to user constructs (such as debugging statements that are turned off by setting a flag) or as a result of other optimizations, and is usually detected and eliminated using intra-procedural analysis. By contrast, unreachable code that is detected at link time usually has very different origins: most of it is due to the inclusion of irrelevant library routines, together with some code that can be identified as unreachable due to the propagation of actual parameter values into a function. In either case, link-time identification of unreachable code is fundamentally interprocedural in nature.

Even though unreachable code can never be executed, its elimination is desirable for a number of reasons:

1. It reduces the amount of code that the link-time optimizer needs to process, and can lead to significant improvements in the amount of time and memory used.

2. It can enable optimizations that otherwise might not have been enabled, such as bringing two basic blocks closer together, allowing for more efficient control transfer instructions to be used, or allowing for a more precise liveness analysis which might trigger several other optimizations.

3. The elimination of unreachable code can reduce the amount of "cache pollution" by unreachable code that is loaded into the cache when nearby reachable code is executed. This, in turn, can improve the overall cache behavior of the program.

4. The elimination of unreachable code simplifies the processing of extended basic blocks (i.e., a sequence of instructions where incoming control flow edges are allowed only at the top, but where there may be outgoing control flow edges at intermediate points in the sequence), since it makes it unnecessary to check for certain situations, such as an unreachable cycle of basic blocks, that could otherwise prove to be problematic.

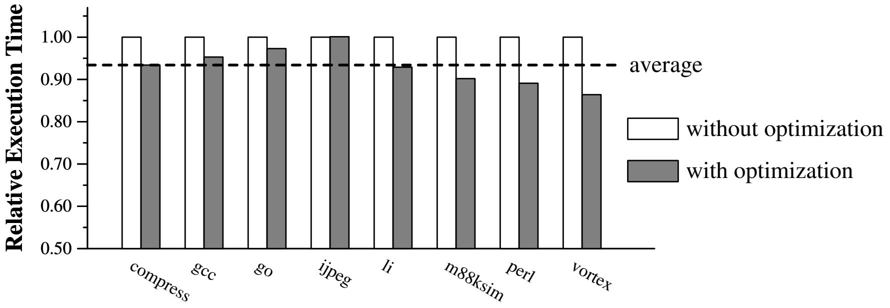

Figure 2: Performance impact of constant computation optimization

Unreachable code analysis involves a straightforward depth-first traversal of the control flow graph, and is performed as soon as the control flow graph of the program has been computed. Initially, all basic blocks are marked as unreachable, except for the entry block for the whole program, and $B_{unknown}$, which has an edge to each basic block that has unknown predecessors (see Section 4). The analysis then traverses the inter-procedural control flow graph and identifies reachable blocks: a basic block is marked reachable if it can be reached from another block that is reachable. Function calls and the corresponding return blocks are handled in a context-sensitive manner: the basic block that follows a function call is marked reachable only if the corresponding call site is reachable.

The amount of unreachable code detected in our benchmarks is shown in Table 3. These numbers do not include no-ops inserted into reachable basic blocks for alignment and instruction scheduling purposes. It can be seen that the amount of unreachable code is quite significant: in many programs, it exceeds 10%, and in one case, the *vortex* program, it is almost 17%. On the average, about 10% of the instructions in our benchmarks were found to be unreachable. This is somewhat higher than the results of Srivastava, whose estimate of the amount of unreachable code in C and Fortran programs was about 4%–6% [35].

For our benchmarks, the primary impact of unreachable code elimination is on code size: the measured impact of this optimization on execution speed is small.

6.2 Optimization of Constant Value Computations

If it is possible to determine, from constant propagation/folding, that a value being computed or loaded into a register is a constant, `alto` attempts to find a cheaper instruction to compute the constant into that register. (This optimization could be generalized to cheap instruction sequences to replace high latency operations, such as multiplication.) The simplest case of this optimization involves computing the values of constants using specific registers whose values are known at each program point, namely, register `$31`, whose value is always 0, and the global pointer register `gp`, whose value at any program point is known at link time. If the (signed) constant k can be represented with 16 bits, the instruction to compute that constant into a register r is replaced by the instruction '`lda` r, k`($31)`' (an instruction `lda` r_a, m`(`r_b`)` computes into register r_a the result of adding m to the contents of r_b, where m is a signed 16-bit value). Similarly, if the difference between the constant k and the value of the `gp` register is representable as a signed 16 bit integer, we can do the same thing using `gp` as the base register. The basic optimization is described by Srivastava and Wall [37]; in `alto` it is generalized so that a constant can be computed from a known value in any register, not just `$31` or `gp`.

Care must be taken to ensure that the constants involved are not addresses within the code sections of the executable. Since `alto` changes the code section, addresses therein are almost certain to change: such constants are therefore excluded from this optimization. Data addresses are not a problem, by contrast, since the transformations implemented within `alto` do not cause data addresses to change.

As an example of this optimization, consider the following C statement, where `a`, `b` and `c` are global variables of type `long`, with addresses 0x1400021558, 0x1400021560, and 0x1400021568 respectively:

```
a = b + c;
```

The code generated for this would typically be as follows:

(a) original code	(b) initial optimized code	(c) final optimized code
(1) `ldq $r1, 16(gp)`	(1) `ldq $r1, 16(gp)`	(1) `ldq $r1, 16(gp)`
(2) `ldq $r2, 96(gp)`	$(2')$ `lda $r2, 8(r1)`	
(3) `ldq $r3, 32(gp)`	$(3')$ `lda $r3, 16(r1)`	
(4) `ldq $r4, 0($r1)`	(4) `ldq $r4, 0($r1)`	(4) `ldq $r4, 0($r1)`
(5) `ldq $r5, 0($r2)`	$(5')$ `ldq $r5, 8($r1)`	$(5')$ `ldq $r5, 8($r1)`
(6) `addq $r4, $r5, $r6`	(6) `addq $r4, $r5, $r6`	(6) `addq $r4, $r5, $r6`
(7) `stq $r6, 0($r3)`	$(7')$ `stq $r6, 16($r1)`	$(7')$ `stq $r6, 16($r1)`

Here, an `ldq` r_a, $k(r_b)$ loads into register r_a the contents of the quadword (i.e., 8 bytes) at the address computed by adding k to the contents of register r_b; the `stq` instruction stores a quadword analogously. In the original code, instructions $(1) - (3)$ load the addresses of the variables from the global address table, using the global pointer register gp to index into this table. Instructions $(4) - (7)$ implement the actual addition. `Alto` is able to determine the addresses loaded into registers `r1`, `r2` and `r3`, since it it is able to determine the contents of gp, and the global address table is a read only area of memory. This allows constant value optimization of instructions (2) and (3), which replaces the address loads with cheaper `lda` instructions. Instructions (5) and 7) are also modified, to use `r1` as the base register. The resulting code is shown in the column labelled "initial optimized code." Note that registers `r2` and `r3` are no longer used in this code: assuming that they are now dead at the end of this code fragment, instructions $(2')$ and $(3')$ will subsequently be deleted, resulting in the final optimized code sequence shown.

`Alto` also tries to optimize the use of constants. Some Alpha instructions allow the use of a small immediate value in place of the second operand register. `Alto` attempts to exploit this feature whenever possible. If only the first operand register is determined to be constant, `alto` will try to swap the operands of the instruction. This is trivial if the instruction is commutative in its operands, but requires more serious analysis and modifications if it is not.

The performance impact of this optimization is illustrated in Figure 2. The programs that benefit the most from this optimization are *m88ksim*, *perl*, and *vortex*, with improvements of around 10% to 13%; overall, the SPEC-95 benchmarks experience a performance improvement of about 6.4% due to this optimization.

6.3 Elimination of Unnecessary Memory Operations

It is sometimes possible to identify `load` (and, less frequently, `store`) operations as unnecessary at link time, and eliminate such operations. Unnecessary `load`s and `store`s can arise for a variety of reasons: a variable may not have been kept in a register by the compiler because it is a global, or because the compiler was unable to resolve aliasing adequately, or because there were not enough free registers available to the compiler. At link time, accesses to globals from different modules become evident, making it possible to keep them in registers [38]; inlining across module boundaries, and of library routines, may make it possible to resolve aliasing beyond what can be done at compile time; and a link time optimizer may be able to scavenge registers that can be used to hold values that were spilled to memory by the compiler. In `alto`, two distinct optimizations are used to eliminate unnecessary memory operations:

1. Suppose that an instruction I_1 stores a register r_1 to memory location l (or loads r_1 from memory location l), and is followed soon after by an instruction I_2 that loads from location l into register r_2. If it can be shown that that location l is not modified between these two instructions, then *load forwarding* attempts to delete instruction I_2 and replace it with a register move from r_1 to r_2. It may happen that register r_1 is overwritten between instructions I_1 and I_2: in this case, `alto` tries to find a free register r_3 (which may or may not be the same as r_2) that can be used to hold the value in r_1.

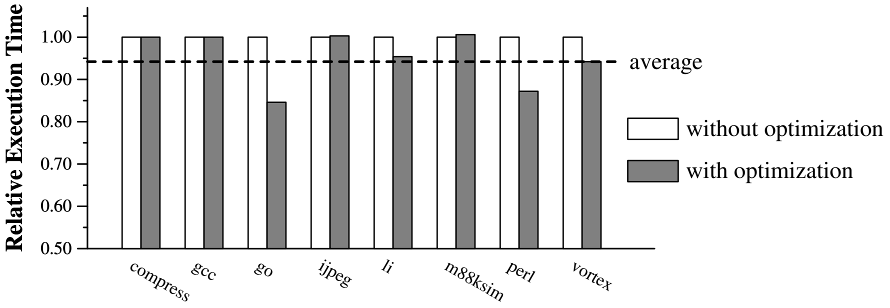

Figure 3: Performance impact of memory operation elimination

If the instruction I_1 can now be shown to be dead, it can be deleted. In our current implementation, this happens less frequently for `store` than for `load` operations because liveness analysis for memory locations is very limited.

2. Memory accesses can result from the saving and restoring of callee-save registers at function boundaries. Some of these accesses may be unnecessary, either because the registers saved and restored in this manner are not touched along all execution paths through a function, or because the code that used those registers became unreachable, e.g., because the outcome of a conditional branch could be predicted as a result of inlining or interprocedural constant propagation, and therefore was deleted. To reduce the number of such unnecessary memory accesses, `alto` uses a variation on *shrink-wrapping* [8] to move register save/restore actions away from execution paths that don't need them. The difference between our implementation of shrink-wrapping, and that originally proposed by Chow [8], is that we don't allow any execution path through a function to contain more than one each of save and restore actions. Apart from this, if a function saves and subsequently restores a callee-save register r but does not change r, the instructions to save and restore r are eliminated.

The performance impact of this optimization is illustrated in Figure 3. The programs that benefit the most from this optimization are *go* and *perl*, with improvements in the neighborhood of 12–15%; overall, the SPEC-95 benchmarks experience an improvement of around 5.7% due to this optimization.

6.4 Inlining

The motivations for carrying out inlining within `alto` are three-fold. The first is to eliminate the function call/return overhead. Usually, inlining a function call gets rid of 2–6 instructions (the call and return instructions, load and store instructions for saving and restoring the return address at the callee, and allocating and deallocating the callee's stack frame; a leaf function, i.e., one that does not call any other functions, will not need to save and restore its return address, and may not have to allocate a stack frame). Additionally, register reassignment can be used to reduce the overhead of saving and restoring registers across call boundaries. The second is to exploit callsite-specific information in the callee: for example, aliasing relationships between the caller's code and the callee's code may become easier to determine after inlining, when they would refer to the same stack frame rather than two different frames (see Section 6.3). The final reason is to improve branch prediction and instruction cache behavior using profile-directed code layout (cf. Section 6.5). Code growth due to inlining is controlled in `alto` as follows: a function is inlined into a call site only if at least one of the following hold:

(i) the callee is "small enough" that the calling and return sequences are together longer than its body;

(ii) the call site under consideration is the only call site for that function; or

Program	Number of Instructions		N_{no_inl}/N_{inl}
	no inlining (N_{no_inl})	with inlining (N_{inl})	
compress	21408	21632	1.010
gcc	317648	318784	1.004
go	78112	77760	0.995
ijpeg	60016	59936	0.999
li	37856	37952	1.003
m88ksim	50720	50912	1.004
perl	98864	100560	1.017
vortex	130032	129840	0.999

Table 4: Code growth due to inlining

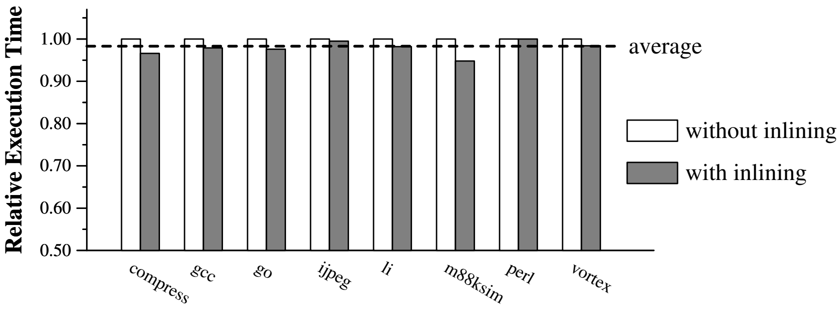

Figure 4: Performance impact of inlining

(*iii*) the call site is "hot," i.e., has a sufficiently high execution count, and (`alto`'s estimate of) the cache footprint of the resulting code does not exceed the size of the instruction cache.

The reason for the last condition is that inlining without attention to cache behavior can have a significant negative effect on program performance. To address this problem, a hot call site C to a function f is considered for inlining by `alto` if it satisfies the following criteria (here, a *critical subgraph* of a control flow graph refers to a subgraph consisting of the hot basic blocks, together with enough other blocks and edges to permit a path, within this subgraph, from the entry node to each hot block and thence to the exit node):

1. for each loop L enclosing the call site C, the number of instructions in the critical basic blocks of L, together with the instructions in the critical subgraph of the callee f, should not exceed the capacity of the level-1 instruction cache (in our case, using the Alpha 21164 processor, this is 8 Kbytes, i.e., 2048 instructions); and

2. if C is not within any loop, then the total number of instructions in the critical subgraphs of the caller and the callee should not exceed the capacity of the level-1 instruction cache.

More sophisticated strategies are possible [28], but these have not been implemented within `alto` at this time.

The extent of code growth due to inlining is shown in Table 4. Inlining causes only a modest increase in code size, in most cases in the neighborhood of 1%, and in a few cases leads to small decreases in code size.

The performance improvements resulting from inlining are shown in Figure 4. The greatest benefits are observed for *m88ksim*, with an improvement of a little over 5%. In general, however, the effect of inlining is small: for the SPEC-95 benchmarks overall, the performance improvement due to inlining is less than 2%. We believe

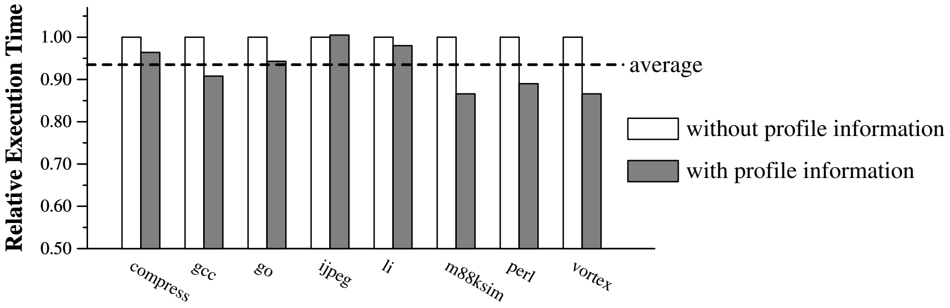

Figure 5: Performance impact of profile-directed code layout

there are three reasons for this: first, the input executables have already been subjected to inlining by the compiler; second, our inter-procedural constant propagation and register liveness analyses are precise enough that they do not benefit significantly from inlining; and third, profile-directed code layout is able to mitigate much of the locality effects of inlining.

6.5 Code Layout

When `alto` creates the interprocedural control flow graph for a program, all unconditional branches are eliminated. The responsibity of the code layout phase is to arrange the basic blocks in the program into a linear sequence, reintroducing unconditional branches where necessary. There are three important issues that should be considered when determining the linear arrangement of basic blocks:

1. ***Branch mispredict penalties*** : During the execution of a conditional branch, instructions are fetched from memory before the branch target has been determined in order to keep the instruction pipeline full and hide memory latencies. In order to do this, the CPU "predicts"—i.e., guesses—the target of the branch. If the guess is wrong, the instructions in the pipeline fetched from the incorrectly predicted target have to be discarded, and instructions from the actual target have to be fetched. The execution cost associated with an incorrect prediction is referred to as a branch mispredict penalty.

   Older processors often use static branch prediction schemes, e.g., where backward branches are predicted as taken and forward branches as not taken. For such processors the benefit of a careful basic block layout is obvious. More modern CPUs, such as the Alpha 21164 used in our experiments, use history-based dynamic branch prediction schemes in the hardware, and result in code where branch misprediction penalties are much less sensitive to code layout. For this reason, `alto` does not consider this issue in determining code layout.

2. ***Control flow change penalty*** : Since instruction fetching precedes instruction decoding in the instruction pipeline, a change in control flow causes the fetch performed while decoding the instruction causing the control flow change to be wasted, thereby incurring a small performance penalty. Note that this is different from the branch mispredict penalty discussed above, since this penalty is incurred even for an unconditional branch, which can always be correctly predicted. A change in control flow also increases the possibility of a miss in the instruction cache.

   This suggests the following guidelines for code layout: unconditional branches should be avoided where possible, and conditional branches should be oriented so that the fall-through path is more likely than the branch-taken path.

3. ***Instruction cache conflicts*** : Because modern CPUs are significantly faster than memory, delivering instructions to them is a major bottle neck. A high hit-rate of the instruction cache is therefore essential. Primary

instruction caches typically are relatively small in size and have low associativity, in order to improve speed. This makes it advantageous to lay out the basic blocks in a program in such a way that frequently executed blocks are positioned close to each other, since this is less likely to lead to cache conflicts [30].

Alto implements two code layout schemes, one that exploits profiling information while the other does not. If profiling information is available, our primary goal is to reduce cache conflicts as far as possible. This is done using a variant of the (bottom-up positioning) approach of Pettis and Hansen [30], with minor modifications to address the problems identified by Calder and Grunwald [10]. This attempts to lay out the basic blocks in a way that minimizes the number of branches that will be taken at runtime; this has the effect that blocks that are executed close to each other temporally tend to be placed close to each other spatially. Currently, `alto` does not carry out procedure placement, i.e., the positioning of the code for different procedures in a program guided by call frequency information; this is mitigated, to some extent, by the fact that the profile-guided placement of basic blocks is carried out in an inter-procedural manner, so that, for example, the block containing a frequently executed function call can be placed close to the entry block of the callee. If no execution profile is available, `alto` attempts to minimize the number of uncoditional branches while maintaining the original code layout in the input program as closely as possible. Here we describe the layout algorithm used when profile information is available.

When profile information is available, the code layout algorithm proceeds by grouping the basic blocks in a program into three sets: The *hot set* consists of the "frequently executed" (according to some threshold, as discussed below) blocks in the program; the *zero set* contains all the basic blocks that were never executed; and The *cold set* contains the remaining basic blocks. The basic block layout for each of these sets is determined separately, and the resulting code sequences concatenated to obtain the overall program layout.

Central to this discussion is the determination of the hot set, i.e., of blocks that are executed "sufficiently frequently." Given a value ϕ in the interval (0,1], we determine the largest execution frequency threshold N such that the set of basic blocks that have execution frequencies exceeding N together account for at least the fraction ϕ of the total number of instructions executed by the program (as indicated by its basic block execution profile). The *hot* basic blocks in a program are defined to be the smallest set of blocks that (i) contain all blocks with execution frequencies exceeding N; and (ii) together contain at least as many instructions as will fit into the primary instruction cache. For example, given $\phi = 0.95$, the hot basic blocks of a program consist of those that allow us to account for at least 95% of the instructions executed at runtime. If those basic blocks fill up the instruction cache we have found N otherwise we will go beyond the 95% until we are able to fill the instruction cache. The value of N, and therefore the hot set, obviously depends on the threshold ϕ: we determine the value of ϕ via empirical tuning, though in principle it could also be specified by the user. Our layout algorithm currently uses $\phi = 0.66$; however, our experiments with a range of values for ϕ indicates that, as long as the *zero set* is separated from the frequently executed code, performance is not very sensitive to the actual value of ϕ.

The performance impact of profile-directed code layout, compared to code layout without the use of profile data (which adheres closely to the layout of the original code), is shown in Figure 5. Many programs can be seen to benefit significantly from profile-directed code layout: the greatest benefits are obtained for *m88ksim*, *perl*, and *vortex*, with improvements of 11–13%. On average, the performance of the SPEC-95 benchmarks improves by about 6.5% due to this optimization.

6.6 Instruction Scheduling

Since the various optimizations effected by `alto` can significantly alter the instruction sequence executed by the processor, an instruction rescheduling phase before regenerating the executable is desirable. This is especially true since the Alpha 21164 processor can issue up to four instructions per cycle, provided that appropriate constraints are met (e.g., not more than one instruction in such a group should try to access memory, access the same functional unit, etc.). Because of this, it is possible that a plausible link-time code transformation, such as the deletion of a `no-op` instruction, can alter the instruction sequence in such a way that opportunities for multiple instruction issues are reduced dramatically, with a corresponding loss in performance. For these reasons, `alto` carries out instruction scheduling after its optimizations have been carried out and the layout of code determined based on execution profiles.

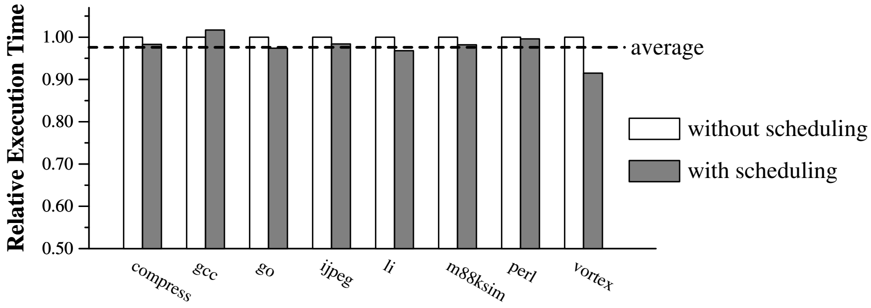

Figure 6: Performance impact of instruction scheduling

The instruction scheduler works on extended basic blocks—that is, a sequence of basic blocks that can be entered only at the beginning, but where control may leave at intermediate points in the sequence—subject to the restriction that the basic blocks constituting the extended basic block must be consecutive in the code layout. Increasing the scope of the scheduler to handle extended basic blocks has two benefits:

1. The scheduler might choose to move instructions over basic blocks boundaries if this improves the schedule. This is especially useful for no-ops which have been introduced for basic block alignment purposes.
2. Basic blocks are not scheduled in isolation: inter-block dependencies are taken into account.

Since profile-directed code layout is carried out prior to scheduling, our use of extended basic blocks achieves an effect very similar to trace scheduling [20].

The performance impact of instruction scheduling is shown in Figure 6. Most programs show performance improvements in the neighborhood of 2%, with *vortex* showing the largest gain of about 9.5%.

7 Performance Results

7.1 Background

Previous sections have discussed the effects of specific analyses and optimizations implemented in `alto`. This section presents the overall performance improvements attained using `alto`, and compares this with the performance obtained using inter-file and profile-directed optimizations within the compiler together with link-time optimization using the *om* link-time optimizer [36]. The benchmarks we used to test the effect of `alto` on C programs were the eight programs in the SPEC-95 integer benchmark suite: *compress* is a file compression program; *gcc* is a commonly used C compiler; *go* is a game-playing program; *ijpeg* is an image compression program; *li* is a Lisp interpreter; *m88ksim* is a simulator for the Motorola 88100 microprocessor; *perl* is a Perl language interpreter; and *vortex* is a single-user object-oriented database transaction benchmark. The size of each program, at both the source and object code levels, is shown in Table 5: the number of source lines reported were measured using the command `wc -l *.c`.

For processing by `alto`, the programs were compiled with the vendor-supplied C compiler V5.2-036 invoked as `cc -O4`, with linker options to retain relocation information and to produce statically linked executables. These executables were instrumented using the vendor-supplied `pixie` and executed on the SPEC training inputs to obtain an execution profile that was provided to `alto`, which was invoked with default switches. We also compared the performance improvements obtained using `alto` with those obtained using the OM link-time optimizer supplied by the vendor [36]. For this, we obtained an execution profile for the base program using `pixie`, as described above, and then used the resulting profile to recompile each program, this time specifying that the compiler should invoke OM, using the command

Program	Source lines	functions	blocks	instructions
compress	1420	316	5092	20707
gcc	193752	2465	77839	353002
go	28457	945	16035	83929
ijpeg	17848	788	11682	62639
li	6916	722	9213	40832
m88ksim	17251	638	11582	53498
perl	23678	722	22765	97079
vortex	52624	1446	28884	155030

Table 5: Static characteristics of our benchmark programs

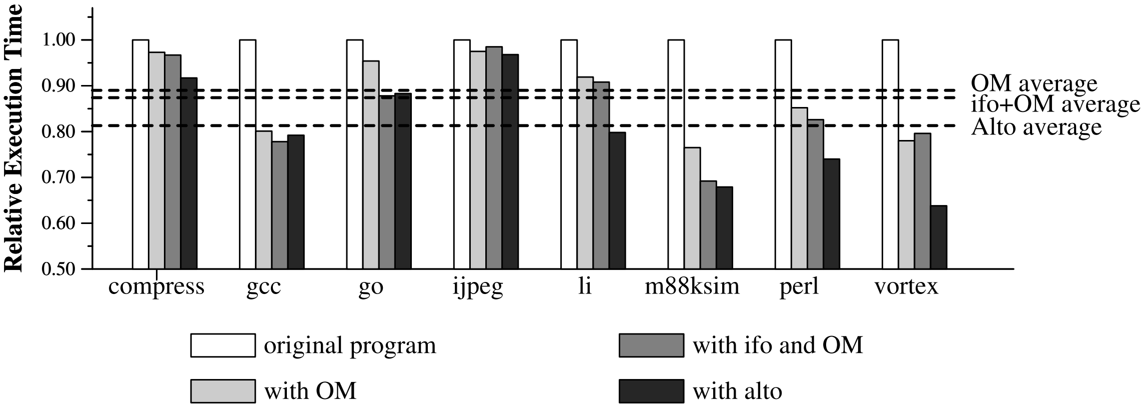

Figure 7: Performance Results: C Programs

```
cc -O4 -om -WL,-om_compress_lita -WL,-om_ireorg_feedback,profile-input
   -WL,-om_dead_code $(CFILES) -non_shared -o om.out -lm
```

where CFILES is a list of all the C source files for the program. Finally, we measured the performance achievable using all of the existing capabilities for static optimization available under Digital Unix. For this, we compiled the programs at the same optimization level as before, but additionally with profile-directed inter-file optimization and link-time optimization using OM [36], as described in Appendix B.

7.2 Performance of Optimized Code

The relative execution times of the different executables obtained as discussed in the previous section, based on the SPEC reference inputs, are shown in Figure 7. The timings were obtained on a Compaq Alpha workstation with a 300 MHz Alpha 21164 processor with a split primary direct mapped cache (8 Kbytes each of instruction and data cache), 96 Kbytes of on-chip secondary cache, 2 Mbytes of off-chip backup cache, and 512 Mbytes of main memory, running Digital Unix 4.0. In each case, the execution time reported was obtained as follows: the run times for each of 7 runs of the executable, run in single-user mode, were recorded; the smallest and largest of these execution times were discarded; and the average of the remaining 5 times reported. The raw data regarding execution times are presented in Table 8 in Appendix A

It can be seen, from Figure 7, that for most of the programs tested, the executable obtained using `alto` is considerably faster than those obtained using OM, both by itself as well as in conjunction with profile-guided

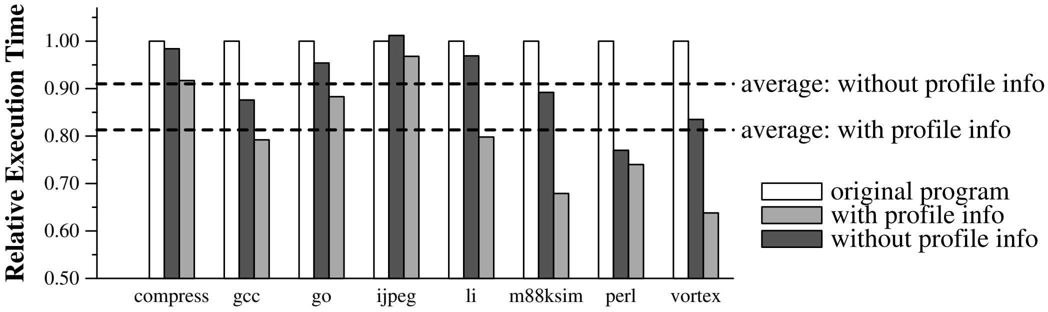

Figure 8: Performance impact of profile information

inter-file optimization. In several cases, the difference in the improvements is quite significant: for example, *li* gets an 8% improvement with OM (9% when profile-guided inter-file optimization is also carried out), compared to a 20% improvement with `alto`. Interestingly, we find that—with the exception of *go* and *m88ksim*—the use of profile-guided inter-file optimization within the compiler does not have significant additional effect on performance beyond what is achieved using just OM; indeed, for two programs, namely, *ijpeg* and *vortex*, the executables obtained with Ifo+FB+Om are slightly slower than those obtained using just OM. Overall, link-time optimization using OM produces an average improvement of around 11%, and the use of profile-guided inter-file optimizations within the compiler in addition to link-time optimization using OM yields an average improvement of about 12.5%; by contrast, link-time optimization using `alto` produces an average improvement of 18.7%.

7.3 Effect of Profile Information

Several of the optimizations performed by `alto`, such as profile-guided code layout, inlining, instruction scheduling, etc., rely on the availability of profile information. In general, it may happen, however, that profile information is unavailable for a program, or is inapplicable because the program's execution is highly input dependent, making it difficult to find "representative" profiling inputs. It is therefore interesting to examine the performance achieved by the code optimized by `alto` when no profile information is available.

The relative execution times when execution profiles are unavailable, compared to the original execution times as well as those when profiles are available, are shown in Figure 8. Two things are evident from this. First, it can be seen that execution profiles have a significant performance impact: on the average, the availability of profiles yields an additional reduction in execution time of about 10%. The second is that, even if execution profiles are not available, however, `alto` is still able to achieve a reduction in execution time of around 9% on the average. Interestingly, when we compare the performance of `alto` without profile information (Figure 8) with that of OM using profile feedback as well as that of OM with profile feedback that is combined with profile-guided inter-file optimization within the compiler (Figure 7), we find that the average performance improvement of 9% achieved using `alto` without profiles is not significantly worse than the 11% improvement for OM and the 12.5% improvement for OM together with profile-guided inter-file optimization, even though the latter two use profile information for their optimizations.

7.4 Static Linking: Impact of Libraries

As mentioned in Section 7.1, our experimental results were obtained using statically linked executables, i.e., where the code for the library routines is linked into the executable statically by the linker. This is due partly to the fact that, as mentioned in Section 1, one of our research objectives in building `alto` was to investigate the effect of analyses and optimizations that had access to the entire program, including library routines. However, the primary reason for the requirement for statically linked executables is that `alto` relies on the presence of relocation

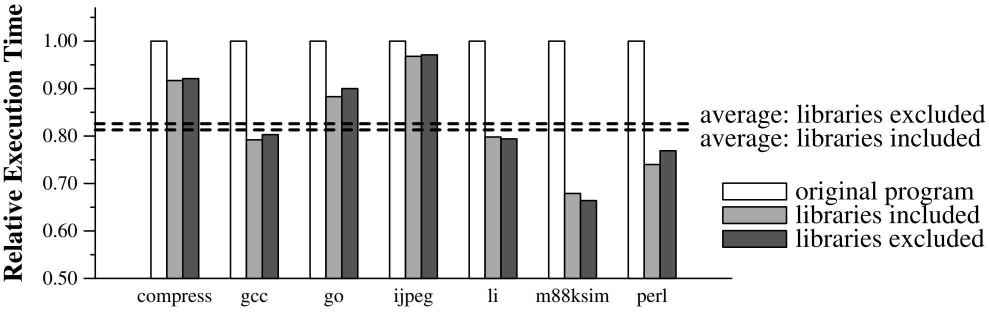

Figure 9: Performance impact of library code

information for its control flow analysis (see Section 4); the Digital Unix linker `ld` refuses to retain relocation information for non-statically-linked executables. (It should be noted that, like `alto`, both the OM link-time optimizer and the related ATOM low-level instrumentation tool require statically linked executables.)

This immediately raises the question of the extent to which our results would hold if the static linking requirement were absent. As indicated above, the static linking requirement is fairly fundamental to `alto`'s operation, making it impractical to actually try to run `alto` on non-statically-linked executables. Instead, we modified `alto` to ignore all library code when carrying out its analyses and optimizations. Thus, inter-procedural liveness analysis and constant propagation, confronted by a call to a library routine, conclude only that the calling conventions will be respected, i.e., that the contents of callee-save registers will be preserved. Inlining of library routines into user code is disallowed, as are all optimizations of library code. Profile-guided code layout, as discussed in Section 6.5, places all library code in the *zero set*, away from user code. We believe this provides a reasonable approximation to the performance that could be attained using link-time optimization on programs that do not have library code linked in statically.

The performance effects of ignoring library routines is shown in Figure 9. It can be seen that while there is some performance benefit to statically linking in library routines, the effect is small: the overall performance improvement drops from 18.7% for "standard" `alto` to 17.4% when analysis and optimization of library routines is disabled, a change of only 1.3%. For two of the benchmarks, *li* and *m88ksim*, the version obtained by ignoring libraries is actually slightly faster than that obtained using "standard" `alto`: we believe this is due to instruction cache effects arising from differences in profile-directed code layout.

These results came as something of a surprise to us, since we had expected that analysis and optimization of library code would have a larger effect on the overall performance of a program. A detailed examination of the benchmarks indicates that the reason for this is that the most frequently executed code fragments typically do not contain calls to library routines. We conjecture that this may be due at least partly to the fact that users believe function calls to be expensive and therefore tend to avoid calls to library routines in hot spots in their programs.

7.5 Alto Resource Usage

Since `alto` is a research prototype whose primary design goal was the evaluation of a variety of link-time optimizations, speed was not a primary design concern and leaves a lot of room for improvement. For example, liveness information is always recomputed before any optimization that uses this information, even though this is unnecessary if no "liveness-altering" transformations have occured since the last liveness computation.

Table 6 lists the optimization times for `alto` and compares it to the compilation time for those programs using `cc -O4`, as well as the time taken to compile them using profile-guided inter-file optimization as well as link-time optimization using OM (`cc`+Ifo+OM). It can be seen that in general, `alto` is slower than `cc`+Ifo+OM by a factor ranging from 2.7 to 5.9; the exception is *compress*, for which `alto` is slower by a factor of 13.6. On average, `alto` is slower by a factor of about 5.

Program	Processing time (secs)			Ratio
	`cc (-O4)`	cc+Ifo+OM	`alto`	`alto`/cc+Ifo+OM
compress	1.61	1.76	23.95	13.61
gcc	162.67	226.01	1265.87	5.60
go	22.75	30.66	130.24	4.24
ijpeg	28.55	32.74	88.58	2.70
li	9.55	11.87	69.71	5.87
m88ksim	26.04	29.10	83.73	2.88
perl	38.44	45.68	225.08	4.93
vortex	67.24	85.02	394.31	4.64
Geometric Mean:				4.91

Table 6: Processing times for compile-time and link-time optimization

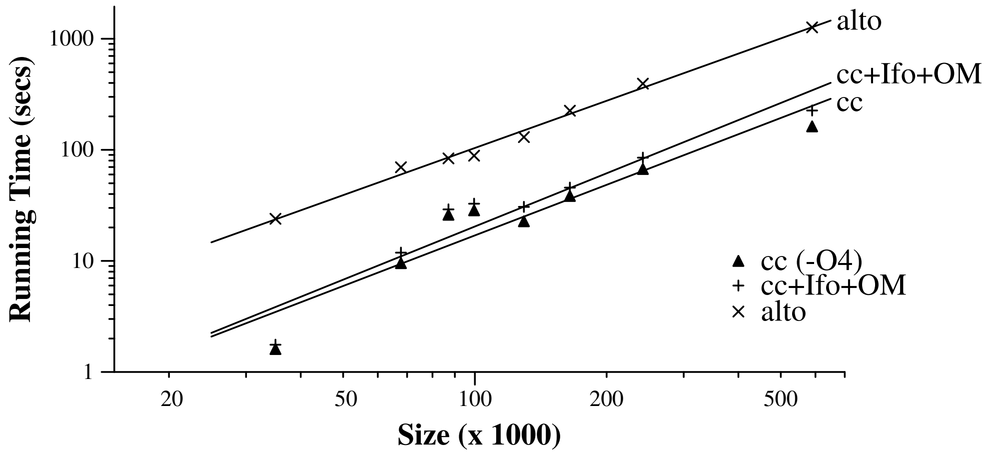

Figure 10: Processing Times: `cc`, cc+Ifo+OM, and `alto`

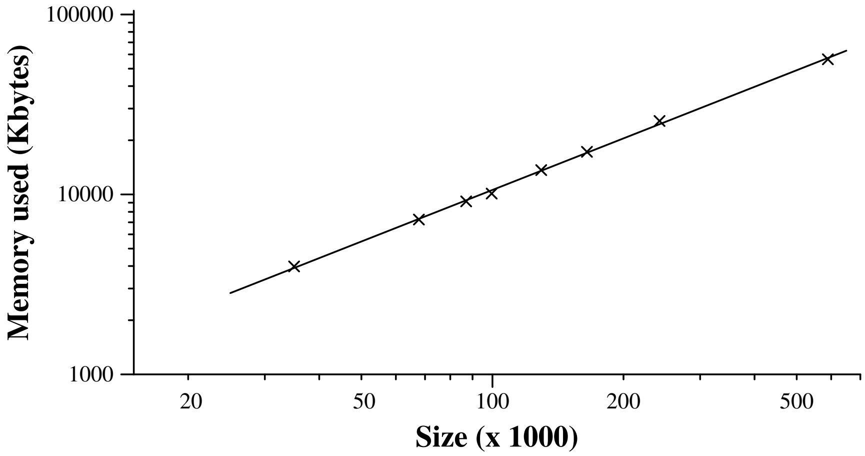

Figure 11: alto memory usage

Apart from the absolute execution times, another interesting and important issue is that of the rate at which this grows as the input size increases. To discuss this meaningfully, we have to specify what is meant by the "size" of a program. For some of the analyses and optimizations within alto, such as constant propagation, the time taken depends on the number of instructions; for others, such as liveness analysis, it depends on the number of basic blocks; yet other inter-procedural analyses may depend on the number of functions in the program. To accommodate these, we chose the "size" of a program to be the sum of the total number of functions, basic blocks, and instructions in the program. Figure 10 plots the running time of alto against the input size, according to this measure, for each of our benchmarks. The line shown for alto, obtained using a least square fit, indicates that the running time of alto is $O(n^{1.411})$. We also plotted the running times for (*i*) compilation using cc -O4 and (*ii*) using inter-file optimization and the OM link-time optimizer (it is not clear that our notion of size, or any similar notion defined entirely in terms of low-level aspects such as the number of instructions, appropriately measures *input* sizes in this case, but it nevertheless gives us some indication of how processing time increases as the input programs get larger). Least squares curve fitting indicates that the growth rate of execution time with program size is $O(n^{1.513})$ in the first case and $O(n^{1.591})$ in the second case.

Perhaps more important than execution time is the amount of memory used: profligate memory usage can have an adverse impact on execution time due to excessive paging, and in extreme cases can cause the program to crash. Figure 11 shows how the memory actually used by alto for its data structures varies with the size of the input program (where the "size" of a program is as described above). Least squares curve fitting indicates that the growth rate of alto's memory usage with input size is $O(n^{0.952})$.[6]

8 Discussion

8.1 Correctness

Since there is generally less high-level semantic information available at link time, alto is at a disadvantage compared to a traditional compiler. For example, input programs can contain arbitrary machine code that need not necessarily correspond to source language programs or conform to assumptions satisfied by code generated by the compiler. Examples of this include numerical library routines where control jumps from the middle of

[6]For simplicity of implementation, the current implementation of alto uses statically allocated arrays for the data structures that hold instructions, basic blocks, control flow edges, etc.: Figure 11 shows how much of these arrays is actually used, overall, for a given input size. If dynamically allocated memory were used instead, memory usage would increase by a small constant factor if we used 64-bit pointers, and not at all if we required the use of 32-bit pointers via the -taso compiler flag; in either case, the sublinear asymptotic growth rate would be unaffected.

one function into the middle of another without going through the usual function call/return interface, and where standard calling conventions about argument register usage may be violated. `Alto` makes a number of assumptions about the behavior of the input programs: if these assumptions are violated, the output generated by `alto` may not be correct.

There are three fundamental assumptions that `alto` makes about the input program:

1. It assumes that the input programs they do not carry out address arithmetic with text segment addresses (address arithmetic involving data addresses is not a problem). The reason for this is that the optimizing transformations carried out by `alto` almost inevitably result in changes to code addresses, and this can cause the program to behave incorrectly if it carries out nontrivial arithmetic involving such addresses. Given this assumption, simple relocation information indicating which words in the executable denote text segment addresses is sufficient to solve the address translation problem statically. Relocation information also allows us to determine which functions are potential candidates for indirect calls through function pointers and callbacks, namely, any function whose address is taken. It should be noted that this assumption is not particular to `alto`, but is fundamental to most tools that rewrite executable files, e.g., instrumentation tools such as *pixie* and *atom*. Because of this assumption, it turns out that the current version of `alto` is unable to handle executables generated by some functional language implementations, such as Objective Caml [27].

2. It assumes that the top-of-stack pointer resides in a particular register and behaves as expected, i.e., always points to the current top of stack.

3. It assumes that the text segment is not modified in the course of execution (see below).

Since `alto` does not currently support dynamically linked libaries, there are no unanalyzed modules in the program. As mentioned in Section 7.4, the static linking requirement is not really fundamental to the way `alto` works, but rather is a byproduct of the requirement for relocation information in the input programs. As discussed in Section 4, function calls whose targets cannot be resolved are handled using an artificial function $\mathsf{F}_{unknown}$ with worst-case dataflow assumptions; similarly, any function whose entry point is marked as relocatable, and which is therefore potentially a target of an unresolved indirect call, is considered to be called from $\mathsf{F}_{unknown}$. Because of the worst-case assumptions made about $\mathsf{F}_{unknown}$, this is conservative and therefore sufficient for correctness. This also suffices for correctly (but conservatively) handling other situations involving statically unpredictable runtime control flow, e.g., where the address of an exception handler is passed by the program to the operating system. If the issue of relocation information were to be resolved, calls to dynamically linked libraries could be handled correctly using a similar approach (actually we could do slightly better, since it can be assumed that such calls conform to calling conventions, e.g., in their treatment of argument registers and callee-saved registers). The problem of operating system calls is solved similarly, with the difference that it is not necessary to make worst case assumptions about the call, since the interface and behavior of system calls is well documented.

Another problem that can arise is that of dynamic code generation, where code is generated and executed at runtime and is therefore not available for inspection prior to execution. There are two possibilities here:

1. If the dynamically generated code is written into the data segment, as in most systems for dynamic code generation (e.g., Tempo [14], DCG [18] and DyC [22]), `alto`'s treatment of the program is conservative and safe, since in this case code that `alto` believes to be static is in fact static and cannot be altered at runtime, while control transfers to or from dynamically generated code are handled conservatively. Calls from static code to the dynamically generated code are represented within `alto` as calls to $\mathsf{F}_{unknown}$, while branches from static code to dynamic code are modelled as branches to $\mathsf{B}_{unknown}$ (see Section 4); since `alto` makes worst-case assumptions about $\mathsf{F}_{unknown}$ and $\mathsf{B}_{unknown}$, such code is therefore treated conservatively. Calls or branches from the dynamically generated code to static code require that the addresses of the static code targets be taken and passed to the dynamic code. This, in turn, leads to these addresses being marked as relocatable, so `alto` inserts, in the control flow graph for the static code, edges from $\mathsf{F}_{unknown}$ and/or $\mathsf{B}_{unknown}$, as appropriate. The result is that the control transfers from the dynamically to the static code are also treated conservatively.

2. If the dynamically generated code is written to the text segment, and can actually modify code that `alto` believes to be static, `alto` may fail (in the sense that the `alto`-optimized code may not be semantically equivalent to the original program). One way around this problem would be to bail out if the program contains any instructions to flush the i-cache. This has not been implemented yet.

Finally, signal handling and volatile variables may pose correctness problems. This was brought home to us while experimenting with Scheme programs compiled with the Bigloo v1.8 Scheme compiler [34], whose runtime system used version 4.7 of the Boehm-Demers-Weiser conservative garbage collector [4]. The garbage collector contained code of the following form:

```
[file: os_dep.c]
  GC_find_limit()
  {
     static volatile char *result;
     ...
     GC_setup_temporary_fault_handler();
     ...
     for(;;) {
       if (...) result += MIN_PAGE_SIZE;
       else result -= MIN_PAGE_SIZE;
       GC_noop(*result);
     }
     ...
  }
```

```
[file: mark.c]
  void GC_noop()
  {
    /* do nothing */
  }
```

In this code, an apparently nonterminating `for` loop repeatedly changes the value of the pointer variable `result` until it becomes an illegal address, so that dereferencing it prior to the call to `GC_noop()` generates an exception. This exception is fielded by a handler set up prior to the `for` loop by the call to `GC_setup_temporary_fault_handler()`: this allows control to leave the `for` loop. When processing (the machine code resulting from) this code, `alto` inlined the call to `GC_noop()`, then eliminated the dereference operation `*result` after inferring that it was unnecessary since it was not used. This, of course, got rid of the exception raised by dereferencing an illegal address, and produced a nonterminating program. We got around the problem by rewriting the code slightly to force `GC_noop()` to use its argument; the problem was noticed independently, and fixed (in a somewhat different way) in subsequent releases of the garbage collector [5]. The problem in this case arises from dead code elimination; one can imagine analogous problems with memory operation elimination (Section 6.3) applied to variables declared to be `volatile`. It is possible to disable these particular optimizations, via command-line switches, when invoking `alto`: a straightforward solution to these problems, albeit one that is not entirely satisfactory esthetically, would be to have the user disable these or other optimizations manually on programs that contain such constructs. A functionally equivalent solution that may be preferable for users would be to provide a command-line option specifying a "conservative" mode of operation where the only optimizations performed are the conversion of indirect function calls to direct function calls using the results of constant propagation (Section 5.1) and profile-directed code layout (Section 6.5): this would in many cases still give nontrivial performance improvements.

8.2 Efficacy of Optimizations

As Figure 7 shows, the link-time optimizations performed by `alto` can lead to significant improvements in program execution speeds, even on programs that have been subjected to a high degree of compile-time optimization. While `alto` implements a large suite of classical intra- and inter-procedural compiler optimizations, it turns out that a relatively small number of these account for most of the performance benefits due to link-time optimization:

– Conversion of indirect function calls (via `jsr` instructions) to direct calls (via `bsr` instructions) produces a speedup of about 10% on average. This optimization relies on constant propagation to determine call targets.

– Constant computation optimization, whose primarily beneficiaries are instructions loading constant addresses, i.e., addresses of global variables and functions, from the read-only data segment into registers. This allows the elimination of associated *load* instructions and gives an average speedup of 6.4%.

– Memory operation elimination, which uses the results of liveness analysis (to identify free registers) and alias analysis (to disambiguate memory references) to eliminate unnecessary *load* instructions. This yields a speedup of 5.7% on the average.

– Profile-directed code layout, which uses execution profile information to lay out the code in such a way as to improve instruction cache utilization. This produces an average speed improvement of about 6.5%.

Of course, the optimizations are not all independent, so the speedup figures are not additive. There were several aspects of these results that we found interesting:

1. Much of the performance benefits result from information that is unavailable at compile time, namely, addresses of globals and functions. This information plays a crucial role in the first two optimizations mentioned above, i.e., optimization of indirect function calls and constant computations, and is also helpful in the alias analysis that supports memory operation optimization. While this may not be entirely unexpected, in retrospect, it suggests that link-time optimization is likely to be useful for improving the performance of programs regardless of the extent of compile-time optimization carried out.

2. Having the entire program available for examination and optimization is useful, but not as much as we had expected:

   (a) Given that the input programs were compiled with a high degree of optimization (-O4), the compiler had already done a good job of register allocation. There were, nevertheless, more opportunities for elimination of memory operations than we had expected. Most of these came about from interprocedural propagation of constant addresses and inter-procedural liveness analysis.

   (b) Profile-directed code layout, applied to the entire program without regard to procedure boundaries, was also very useful for performance improvement. Of course, the observation that profile-directed code layout can yield significant performance benefits has been made by numerous authors, and is hardly new: the point here is that having the entire program available for manipulation allows us to optimize code layout for inter-procedural execution paths as well.

   (c) The availability of library routines for analysis and optimization had a surprisingly small effect on performance (about 1.3% on average).

   (d) The ability to carry out procedure inlining across module/file boundaries had less of a performance impact than we had anticipated (under 2%). This may be due partly to the fact that some inlining had already been carried out by the compiler.

9 Related Work

Link-time code optimization has been considered by a number of other researchers. Link-time register allocation, aimed at allowing global variables to be kept in registers and reducing register saves and restores at inter-module calls, is discussed by Santhanam and Odnert [33] and Wall [38]. The Zuse Translation System [13] and the `mld` link-time optimizer [19] are aimed at reducing the cost of abstraction in object-oriented languages. Ayers *et al.* describe a production-quality link-time optimizer for Hewlett-Packard systems running HP-UX [2], which is distinguished by its ability to perform whole-program optimizations on very large programs, by virtue of the careful attention paid to memory management issues. These works rely on specially engineered compilers that produce either object files containing special annotations to assist the link-time optimizer [38], or an intermediate representation of the program (together with semantic information about it) that is subsequently optimized and translated to executable code by the linker [2, 13, 19, 33]. One implication of this is that performance-critical modules written in hand-coded assembly language, third-party software such as libraries for which source code is not available, or code that is not in the source language supported by the compiler, is not amenable to optimization by these tools. Machine-level global optimization is discussed also by Johnson and Miller [25], but unlike `alto`, this system does not carry out interprocedural analysis and optimizations.

Several authors have investigated whole program optimization at compile time: examples include the Fortran-D compiler and its successors, developed at Rice University [23], which targets parallel and distributed scientific programs; and the Vortex compiler for object-oriented languages [15], which targets a number of object-oriented languages. There are three primary differences between this work and ours. The first is that, as compilers, they target a particular language or family of languages; by contrast, `alto` is able to process code generated from a variety of languages, regardless of the source language the code was generated from, as long as the code respects the assumptions discussed in Section 8.1. The second is that the specific set of analyses and optimizations implemented is different for each of the systems, since these depend on characteristics of the specific classes of applications the language being compiled tends to be used for, e.g., dependence analysis in the Fortran-D compiler, receiver class prediction in Vortex. The third difference arises from the fact that, as discussed in Section 8.2, the entities visible at compile time tend to be different from those visible at link time, and as a result the sources of performance improvement in a compiler that carries out whole-program optimization will be different from those for a link-time optimizer; our experiences indicate that, precisely for this reason, link-time optimization can be useful for improving program performance even if the compiler carries out whole-program optimization.

The systems that are the closest to ours are the OM [36, 37], Spike [12], and Etch [31] link-time optimizers. The actions carried out by these systems are conceptually very similar to ours (as they must be), though they differ in details. Spike and Etch are intended for executables running under Windows, on Compaq Alpha and Intel x86 processors respectively. Spike carries out three different optimizations [12]: hot-cold optimization [11], register allocation, and profile-directed code layout; of these, `alto` does not currently implement hot-cold optimization, but implements the other two optimizations, as well as others described earlier. Because they are targeted to different operating systems, a direct comparison of `alto` against these systems was not feasible. Our comparisons with OM (see Section 7) indicate that the code produced by `alto` is considerably faster than that produced by OM. This is due at least partly to the fact that OM implements relatively few optimizations, which are primarily intra-procedural in nature and do not have the benefit of alias analysis or register liveness analysis; in particular, optimizations that need scratch registers are not carried out.

The Dynamo system takes a very different approach to global optimization: it optimizes native executables dynamically, as they execute [3]. This system is able to carry out optimizations across procedure and module boundaries, and has the advantage of being able to handle either statically or dynamically linked libraries. The main disadvantage is that dynamic optimization necessarily incurs some runtime overhead, and in some cases this overhead can overwhelm the optimization benefits and yield a net loss in performance. A related problem is that the desire to keep the overhead of dynamic optimization low, so as to avoid such problems, makes it difficult to implement sophisticated but potentially expensive analyses or optimizations.

10 Conclusions

Traditional compile-time analyses and optimizations are limited by the scope of the compilation unit: analyses and optimizations are usually limited to individual procedures (even interprocedural optimizations are generally limited to individual modules, and library routines are not available for either analysis or optimization). Since the entire program is available for inspection after linking, link-time optimization can overcome some of these deficiencies. This paper describes `alto`, a link-time optimizer that we have implemented for the Compaq Alpha. Experiments indicate that even though it currently implements only relatively simple analyses—for example, checks for pointer aliasing are only implemented in the most rudimentary and conservative way—the performance of the code generated by the system is, on the average, significantly better than that generated by the OM link-time optimizer [36] supplied by the vendor.

Acknowledgements

We are grateful to Robert Cohn of Compaq Computer Corp. for information about the OM link-time optimizer, and to Craig Neth of Compaq Computer Corp. for his help with the use of OM and feedback-directed optimization. Comments by the anonymous referees helped improve both the content and the presentation of the paper significantly.

References

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

[2] A. Ayers, S. de Jong, J. Peyton, and R. Schooler, "Scalable Cross-Module Optimization", *Proc. SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998, pp. 301–312.

[3] V. Bala, E. Duesterwald, and S. Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo", Technical Report *HPL-1999-78*, Hewlett-Packard Laboratories, Cambridge, Mass., June 1999.

[4] H.-J. Boehm, "Space-efficient Conservative Garbage Collection", *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, 1993, pp. 197–206.

[5] H.-J. Boehm, personal communication, March 1998.

[6] A. L. Chow and A. Rudnick, "The Design of a Data Flow Analyzer", *Proc. SIGPLAN '82 Conference on Compiler Construction*, June 1982, pp. 106-119.

[7] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of Pointers and Structures", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 296–310.

[8] F. C. Chow, "Minimizing Register Usage Penalty at Procedure Calls", *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988, pp. 85–94.

[9] B. Calder, P. Feller, and A. Eustace, "Value Profiling", *Proc. MICRO-30*, Dec. 1997.

[10] B. Calder and D. Grunwald, "Reducing Branch Costs via Branch Alignment", *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994, pp. 242–251.

[11] R. Cohn and P. G. Lowney, "Hot Cold Optimization of Large Windows/NT Applications", *Proc. MICRO29*, Dec. 1996.

[12] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Optimizing Alpha Executables on Windows NT with Spike", *Digital Technical Journal* vol. 9 no. 4, 1997, pp. 3–20.

[13] C. S. Collberg, *Flexible Encapsulation*, Ph.D. Thesis, Lund University, 1992.

[14] C. Consel and F. Noël, "A General Approach for Run-time Specialization and its Application to C", *Proc.* 23^{rd} *Annual ACM Symposium on Principles of Programming Languages*, Jan. 1996, pp. 145–156.

[15] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers, "Vortex: An Optimizing Compiler for Object-Oriented Languages", *Proc. OOPSLA'96: Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1996, pp. 83–100.

[16] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond *k*-Limiting", *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 230–241.

[17] M. Emami, R. Ghiya, and L. J. Hendren, "Context-Sensitive Interprocedural Analysis in the Presence of Function Pointers", *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 242–256.

[18] D. R. Engler and T. A. Proebsting, "DCG: An Efficient, Retargetable Dynamic Code Generation System", *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 263-271.

[19] M. F. Fernández, "Simple and Effective Link-Time Optimization of Modula-3 Programs", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 103–115.

[20] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

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

[22] B. Grant, M. Philipose, M. Mock, C. Chambers, S.J. Eggers, "An Evaluation of Staged Run-time Optimizations in DyC", *Proc. SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999, pp. 293–304.

[23] M.W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng, "Interprocedural Compilation of FORTRAN D for MIMD Distributed Memory Machines", *Proc. Supercomputing '92*, November 1992.

[24] S. Jagannathan and A. Wright, "Flow-directed Inlining", *Proc. SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996, pp. 193–205.

[25] M. S. Johnson and T. C. Miller, "Effectiveness of a Machine-Level Global Optimizer", *Proc. SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 99–108.

[26] W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing", *Proc. SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992, pp. 235–248.

[27] X. Leroy, "The Effectiveness of Type-Based Unboxing", *Workshop on Types in Compilation '97*, Amsterdam, 1997.

[28] S. McFarling, "Procedure Merging with Instruction Caches", *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991, pp. 71–79.

[29] R. Muth, "Register Liveness Analysis of Executable Code", Manuscript, Dept. of Computer Science, The University of Arizona, Dec. 1998. Available as `http://www.cs.arizona.edu/alto/papers/liveness.ps`.

[30] K. Pettis and R. C. Hansen, "Profile-Guided Code Positioning", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[31] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen, "Instrumentation and Optimization of Win32/Intel Executables", 1997 USENIX Windows NT Workshop (to appear).

[32] E. Ruf, "Context-Insensitive Alias Analysis Revisited", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 13–22.

[33] V. Santhanam and D. Odnert, "Register Allocation across Procedure and Module Boundaries", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 28–39

[34] M. Serrano and P. Weis, "Bigloo: a portable and optimizing compiler for strict functional languages" *Proc. Static Analysis Symposium (SAS '95)*, 1995, pp. 366–381.

[35] A. Srivastava, "Unreachable Procedures in Object-Oriented Programming", *ACM Letters on Programming Languages and Systems* vol. 1 no. 4, Dec. 1992, pp. 355–364.

[36] 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.

[37] A. Srivastava and D. W. Wall, Link-time Optimization of Address Calculation on a 64-bit Architecture", *Proc. SIGPLAN '94 Conference Programming Language Design and Implementation*, June 1994, pp. 49–60.

[38] D. W. Wall, "Global Register Allocation at Link Time", *Proc. SIGPLAN '86 Symposium on Compiler Construction*, July 1986, pp. 264–275.

[39] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 1–12.

Program	*Execution Time* (sec) 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	283.33 s ±0.67%	275.54 s ±0.21%	273.92 s ±0.49%	259.73 s ±0.30%	0.973	0.967	0.917
gcc	291.01 s ±0.23%	233.11 s ±0.46%	226.40 s ±0.39%	230.56 s ±0.50%	0.801	0.778	0.792
go	340.49 s ±0.05%	324.73 s ±6.91%	299.05 s ±0.23%	300.69 s ±0.03%	0.954	0.878	0.883
ijpeg	337.84 s ±0.23%	329.55 s ±0.11%	332.65 s ±0.15%	326.93 s ±0.02%	0.975	0.985	0.968
li	318.81 s ±1.08%	293.01 s ±0.61%	289.49 s ±0.21%	254.36 s ±0.88%	0.919	0.908	0.798
m88ksim	333.22 s ±0.04%	254.88 s ±0.04%	230.71 s ±0.16%	226.21 s ±0.06%	0.765	0.692	0.679
perl	246.91 s ±0.18%	210.41 s ±1.06%	203.93 s ±0.26%	182.59 s ±0.16%	0.852	0.826	0.740
vortex	497.68 s ±0.23%	388.29 s ±0.27%	395.92 s ±2.01%	317.62 s ±0.80%	0.780	0.796	0.638
Geometric Mean:					0.890	0.874	0.813

Table 7: Performance results: `alto` compared to OM and OM+Ifo+Feedback

A Performance Impact of Alto Optimizations: Raw Data

This section gives the raw performance data for our experiments. The timings were obtained on a Compaq Alpha workstation with a 300 MHz Alpha 21164 processor with a split primary direct mapped cache (8 Kbytes each of instruction and data cache), 96 Kbytes of on-chip secondary cache, 2 Mbytes of off-chip backup cache, and 512 Mbytes of main memory, running Digital Unix 4.0. In each case, the execution time reported was obtained as follows: the run times for each of 7 runs of the executable, run in single-user mode, were recorded; the smallest and largest of these execution times were discarded; and the average of the remaining 5 times reported. In addition, the variation among the different timings is shown as $\pm x\%$, where x is the magnitude of the maximum deviation of any of the 5 timings considered from the mean, expressed as a percentage of the mean. Thus, given the set of timings 91, 95, 98, 100, 101, 102, 110, we would discard the lowest (91) and highest (110), and use the remaining 5 numbers to obtain the timing $99.2 \pm 4.23\%$, where 99.2 is the mean of the remaining 5 times, and the maximum deviation from the mean ($99.2 - 95 = 4.2$) is 4.23% of the mean. It can be seen, from Tables 7 and 8, that the timings obtained for any particular executable do not show much variation: in most cases, the maximum deviation from the mean is less than 1%.

Table 7 compares the performance improvements obtained with `alto`, compared to those obtained using OM as well as those resulting from OM coupled with profile-guided inter-file optimization. Table 8 shows performance data comparing the effects of different optimizations. For each benchmark, this table shows the performance obtained when various different optimizations are turned off. Each such performance number is presented in three rows: the top row shows the (mean) execution time; the middle row shows the maximum deviation from the mean; and the third row expresses the mean execution time as a fraction of the execution time of the original, i.e., input, program.

Program	Original	noCProp	noCOpt	noMOpt	noInline	noLayout	noSched	noProfile	AllOpts
compress	283.33 s ±0.67% (1.000)	278.17 s ±0.27% (0.982)	278.21 s ±0.01% (0.982)	259.61 s ±0.02% (0.916)	268.82 s ±0.06% (0.949)	269.29 s ±0.10% (0.950)	264.35 s ±0.11% (0.933)	263.97 s ±0.03% (0.932)	259.73 s ±0.30% (0.917)
gcc	291.01 s ±0.23% (1.000)	256.18 s ±1.56% (0.880)	241.92 s ±0.04% (0.831)	230.58 s ±0.02% (0.792)	235.45 s ±0.56% (0.809)	253.84 s ±0.10% (0.872)	226.66 s ±0.04% (0.779)	263.34 s ±0.02% (0.905)	230.56 s ±0.50% (0.792)
go	340.49 s ±0.05% (1.000)	323.48 s ±0.03% (0.950)	308.97 s ±0.01% (0.907)	355.39 s ±0.00% (1.044)	308.16 s ±1.70% (0.905)	318.71 s ±4.88% (0.936)	308.73 s ±0.00% (0.907)	315.25 s ±0.00% (0.926)	300.69 s ±0.03% (0.883)
ijpeg	337.84 s ±0.23% (1.000)	328.83 s ±0.07% (0.973)	326.74 s ±0.03% (0.967)	325.90 s ±0.01% (0.965)	328.54 s ±0.62% (0.972)	325.19 s ±0.22% (0.963)	332.29 s ±0.02% (0.984)	323.21 s ±0.01% (0.957)	326.93 s ±0.02% (0.968)
li	318.81 s ±1.08% (1.000)	293.20 s ±0.24% (0.920)	273.87 s ±0.00% (0.859)	266.59 s ±0.21% (0.836)	258.94 s ±0.03% (0.812)	259.68 s ±0.44% (0.815)	262.82 s ±0.02% (0.824)	262.58 s ±0.01% (0.824)	254.36 s ±0.88% (0.798)
m88ksim	333.22 s ±0.04% (1.000)	275.66 s ±0.13% (0.827)	250.86 s ±0.04% (0.753)	224.86 s ±0.08% (0.675)	238.50 s ±0.08% (0.716)	261.27 s ±0.13% (0.784)	230.33 s ±0.06% (0.691)	253.55 s ±0.04% (0.761)	226.21 s ±0.06% (0.679)
perl	246.91 s ±0.18% (1.000)	212.99 s ±0.12% (0.863)	204.91 s ±0.08% (0.830)	209.32 s ±0.07% (0.848)	182.51 s ±0.11% (0.739)	205.13 s ±0.81% (0.831)	183.37 s ±0.04% (0.743)	237.13 s ±0.03% (0.960)	182.59 s ±0.16% (0.740)
vortex	497.68 s ±0.23% (1.000)	389.57 s ±0.46% (0.783)	367.43 s ±0.42% (0.738)	337.22 s ±0.96% (0.678)	322.77 s ±0.22% (0.649)	366.87 s ±2.28% (0.737)	347.13 s ±0.68% (0.697)	380.38 s ±0.44% (0.764)	317.62 s ±0.80% (0.638)
Geom. mean	1.000	0.905	0.870	0.861	0.826	0.868	0.832	0.893	0.813

Key:

Original	:	Input program
noCProp	:	No constant propagation (Section 5.1)
noCOpt	:	No optimization of constant value computations (Section 6.2)
noMOpt	:	No memory access optimizations (Section 6.3)
noInline	:	No inlining (Section 6.4)
noLayout	:	No profile-guided code layout (Section 6.5)
noSched	:	No instruction scheduling (Section 6.6)
noProfile	:	No profile information (Section 7.3)
AllOpts	:	`alto` with all optimizations

Table 8: Performance impact of various optimizations

B Compiling Programs using Inter-File Optimization and OM

To use both the inter-file optimization capability of the vendor's C compiler as well as the OM link-time optimizer, we compiled the programs at the same optimization level as before, but additionally with profile-directed inter-file optimization and link-time optimization using OM [36]. For this, the programs were compiled as follows:

1. First, the programs were compiled as

   ```
   cc -O4 $(CFILES) -non_shared -o orig.out -lm
   ```

   where `CFILES` is a list of all the C source files for the program.

2. The resulting executable `orig.out` was instrumented with `pixie` and run on the SPEC training input for the benchmark to produce an execution profile. A feedback file was then generated from this profile using the command

   ```
   prof -pixie -feedback opt.out.fbo orig.out
   ```

3. The source files were recompiled with profile-guided and inter-file optimization turned on, using the feedback file generated in the previous step:

   ```
   cc -O4 -ifo -inline speed -feedback opt.out.fbo $(CFILES)
      -non_shared -o ifo_fb.out -lm
   ```

   The switch `-ifo` turns on inter-file optimization (this is the reason all the C files are specified together using `CFILES`), and `-inline speed` instructs the compiler to inline routines to enhance execution speed.

4. The resulting executable `ifo_fb.out` was again instrumented with `pixie`, using the SPEC training inputs.

5. The resulting execution profile was used to recompile the program a final time, this time with the OM link-time optimizer turned on as well:

   ```
   cc -O4 -ifo -inline speed -feedback opt.out.fbo
      -om -WL,-om_compress_lita -WL,-om_ireorg_feedback,ifo_fb.out
      -WL,-om_dead_code $(CFILES) -non_shared -o ifo_fb_om.out -lm
   ```

   The reason it is necessary to regenerate the profile information for OM is that the feedback-directed optimizations can change code addresses, rendering the original profile useless from the perspective of OM. Notice that in this step, two distinct sets of profiles are being used: the feedback file `opt.out.fbo`, generated from the original profile obtained in step 2; and the profile for `ifo_fb.out`, obtained for the executable resulting from feedback-directed inter-file optimization in step 4.