Link-time Improvement of Scheme Programs *

Saumya Debray Robert Muth Scott Watterson
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
{debray, muth, saw}@cs.arizona.edu

Technical Report 98-16
December 1998

Abstract

Optimizing compilers typically limit the scope of their analyses and optimizations to individual modules. This has two drawbacks: first, library code cannot be optimized together with their callers, which implies that reusing code through libraries incurs a penalty; and second, the results of analysis and optimization cannot be propagated from an application module written in one language to a module written in another. A possible solution is to carry out (additional) program optimization at link time. This paper describes our experiences with such optimization using two different optimizing Scheme compilers, and several benchmark programs, via `alto`, a link-time optimizer we have developed for the DEC Alpha architecture. Experiments indicate that significant performance improvements are possible via link-time optimization even when the input programs have already been subjected to high levels of compile-time optimization.

* This work was supported in part by the National Science Foundation under grants CCR-9502826 and CCR 9711166.

1 Introduction

The traditional model of compilation usually limits the scope of analysis and optimization to individual procedures, or possibly to modules. This model for code optimization does not take things as far as they could be taken, in two respects. The first is that code involving calls to library routines, or to functions defined in separately compiled modules, cannot be effectively optimized; this is unfortunate, because one expects programmers to rely more and more on code reuse through libraries as the complexity of software systems grows, (there has been some work recently on cross-module code optimization [4, 13]: this works for separately compiled user modules but not for libraries). The second problem is that a compiler can only analyze and optimize code written in the language it is designed to compile. Consider an application that investigates the synthesis of chemical compounds using a top-level Scheme program to direct a heuristic search of a space of various reaction sequences, and Fortran routines to compute reaction rates and yields for individual reactions.[1] With the traditional compiler model, analyses and optimizations will not be able to cross the barrier between program modules written in different languages. For example, it seems unlikely that current compiler technology would allow Fortran routines in such an application to be inlined into the Scheme code, or allow context information to be propagated across language boundaries during interprocedural analysis of a Scheme function calling a Fortran routine.

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 an example, even for simple first-order programs, determining the extent of a jump table in an executable file, and hence the possible targets of the code derived from a `case` or `switch` statement, can be difficult when dealing with executables; 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 source language constructs, disregarding "nasty" features such as pointer arithmetic and out-of-bounds array accesses. At the level of executable code, on the other hand, 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: this is nothing but an out-of-bounds array reference.

- Executable programs tend to be significantly larger than the source programs they were derived from (e.g., see Figure 2). 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 exorbitant time or space requirements.

This paper describes our experiences with such optimization on a number of Scheme benchmark programs, using a link-time optimizer, called `alto` ("a link-time optimizer"), that we have built for the DEC Alpha architecture. Apart from a variety of more or less "conventional" optimizations, `alto` implements several optimizations, or variations on optimizations, that are geared specifically towards

[1] This is a slightly paraphrased account of a project in the Chemistry department at the University of Arizona a few years ago: the project in question used Prolog rather than Scheme, but that does not change the essential point here.

programs that are rich in function calls—in particular, recursion—and indirect jumps (resulting both from higher order constructs and tail call optimization). Experiments indicate that significant performance improvements are possible using link-time optimization, even for code generated using powerful optimizing compilers.

2 Scheme vs. C: Low Level Execution Characteristics

Programs in languages such as Scheme differ in many respects from those written in imperative languages such as C, e.g., in their use of higher order functions and recursion, control flow optimizations such as tail call optimization, and in their relatively high frequency of function calls. However, it is not *a priori* clear that, at the level of executable code, the dynamic characteristics of Scheme programs are still significantly different from C code. To this end, we examined the runtime distributions of different classes of operations for a number of Scheme programs (those considered in Section 7) and compared the results with the corresponding figures for the eight SPEC-95 integer benchmark programs. The results, shown in Figure 1, show some interesting contrasts:

- The proportion of memory operations in Scheme code is 2.5 times larger than that in C code; we believe that this is due to a combination of two factors: the use of dynamic data structures such as lists that are harder to keep in registers, and the presence of garbage collection.
- The proportion of conditional branches is significantly higher (by a factor of almost 1.8) in Scheme code than in C code. This is due at least in part to runtime dispatch operations on tag bits encoding type information.
- The proportion of indirect jumps in Scheme code is close to three times as high as that in C code. This is due, in great part, to the way tail call optimization is handled.
- The proportion of (direct and indirect) function calls is somewhat smaller in the Scheme programs than in the C code. To a great extent, this is because the Scheme compilers try hard to eliminate function calls wherever possible.

Code generated for programs in dynamically typed languages usually also carries out pointer arithmetic to manipulate tagged pointers. We didn't measure the proportion of instructions devoted to tag manipulation (there didn't seem to be a simple and reliable way to do this in the context of our implementation), but we note that Steenkiste's studies indicate that Lisp programs spend between 11% and 24% of their time on tag checking [17]. We expect that the overall conclusion—that programs spend a significant amount of time in tag manipulation—holds for Scheme programs as well.

Most of the prior work on link-time optimization has focused on imperative languages [7, 12, 15, 16, 19]. The differences in runtime characteristics between Scheme and C programs, as discussed above, can have a significant effect on the extent to which systems designed for executables resulting from (human-written) C programs will be effective on code generated from Scheme programs. The reasons for this are the following:

1. The pointer arithmetic resulting from tag manipulation tends to defeat most alias analysis algorithms developed for languages such as C (see, for example, [20]).
2. The higher proportion of memory operations in Scheme programs can inhibit optimizations because, in the absence of accurate alias information, they greatly limit the optimizer's ability to move code around. The problem is compounded by the fact that pointer arithmetic resulting from tag manipulation adversely affects the quality of alias information available.

Operation	*Scheme* (%)	*C* (%)	*Scheme/C*
integer ops	44.81	35.70	1.25
memory ops	34.02	13.63	2.50
floating point ops	0.58	0.76	0.76
conditional branches	10.44	5.89	1.77
indirect jumps	2.89	0.99	2.92
direct calls	0.29	0.44	0.66
indirect calls	1.51	1.72	0.88

Figure 1: Dynamic distributions for classes of common operations

3. The higher proportion of indirect branches in Scheme code can interfere with low-level control flow analysis and inhibit optimizations such as profile-directed code layout to improve instruction cache utilization [11].

Our experiments, described in Section 7, show that `alto` is able to achieve significant speed improvements, even for Scheme programs that have been heavily optimized at compile-time; in this regard it consistently outperforms the OM link-time optimizer [15] from DEC.

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.

Analysis : A number of analyses are carried out during this phase, including register liveness analysis, constant propagation, 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. Finally, the final phase carries out profile-directed code layout [11], instruction scheduling, and insertion of no-ops for alignment purposes, after which the code is written out.

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]. Matters 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. 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. 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 identify 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. When this happens, the edge from the indirect jump to $\mathsf{B}_{unknown}$ is 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, then any remaining edges from $\mathsf{B}_{unknown}$ to basic blocks within that function are deleted.

5 Data Flow Analysis

`Alto` carries out a variety of inter-procedural data flow analyses, including reachability analysis, constant propagation, register liveness analysis, side effect analysis, etc. The most important of these is inter-procedural constant propagation, which plays a central role in the construction of the control flow graph of the program. A discussion of these analyses is omitted due to space constraints, except for the observation that we find that for Scheme programs, `alto` is able to determine, on the average, the operands and results for about 29% of the instructions in programs. This is considerably higher than for C and Fortran programs: e.g., for the programs in the SPEC-95 benchmark suite, it is able to evaluate about 17% of the instructions on the average: we are currently looking into the reason for this difference. Note that this does not mean that a third of the instructions of a program can be removed by `alto`, since in most cases the these represent address computations. This information can, nevertheless, be used to good advantage elsewhere, as discussed above: experiments indicate that for the Scheme benchmarks considered, disabling constant propagation leads to a performance loss of 5%–12% (compared to when all analyses and optimizations are enabled).

6 Program Optimization

The optimizations carried out by `alto` are typically guided by execution profile information and the availability of machine resources. Space constraints preclude a detailed discussion of these optimizations: here we discuss only the most important ones.

6.1 Inlining

Traditionally, Scheme compilers carry out inlining at, or close to, the level of the source program [2, 4, 10, 18]. At this level, the primary benefits of inlining come from specializing and simplifying the inlined function, e.g., by evaluating conditionals and pruning away code that becomes unreachable. Code growth during inlining is usually controlled via syntax-driven techniques, ranging from simple syntax-directed estimates of the size of the callee [2, 4, 10] to more refined estimates based on the residual size of the callee after specializing it to the call site under consideration [18]. At link time, by contrast, it is reasonable to expect that considerable amounts of inlining have already been carried out by the Scheme compiler being used (and then possibly some more by the C compiler, if the compilation is via translation to C). This means that, while some code simplification might occur due to the propagation of constant arguments into library routines, it seems unlikely that link-time inlining will give rise to large amounts of code simplification and pruning. On the other hand, more accurate information is available about object code size, making it easier to consider the effects of inlining on the instruction cache utilization of a program.

The motivations for carrying out inlining within `alto` are three-fold: to reduce the function call/return overhead; to simplify reasoning about aliasing between the caller's code and the callee's code, since after inlining they typically refer to the same stack frame rather than two different frames (see Section 6.2); and to improve branch prediction and instruction cache behavior using profile-directed code layout [11]. In `alto`, code growth due to inlining is controlled by ensuring that (`alto`'s estimate of) the cache footprint of the resulting code does not exceed the size of the instruction cache: in particular, if the call site being considered for inlining lies within any loop, the total size of the "hot" execution paths through the loop is not allowed to exceed the size of the primary instruction cache.

Inlining in the presence of higher order functions has typically been accomplished using sophisticated control flow analyses [10]. We believe that such analyses are too expensive to be practical at the level of machine code. Instead, we use a simple profile-guided inlining technique we call *guarded inlining*—which is conceptually similar to, though somewhat more general than, a technique for optimizing dynamically dispatched function calls in object-oriented languages referred to as "receiver class prediction" [5, 8]—to achieve similar results. Suppose we have an indirect function call whose target we are unable to resolve. We use profiling to identify the most frequent targets at each such indirect call. Suppose that the most frequent target is a function f at address $addr_0$. With guarded inlining, we test whether the target address is $addr_0$: if this is the case, execution drops through into the inlined code for f; otherwise, an indirect function call occurs, as before. It's not too difficult to see, in fact, that in general the transformation can be adapted to any indirect branch. This mechanism allows us to get the benefits of inlining even for call sites that can, in fact, have multiple possible targets, in contrast to schemes that require control flow analysis to identify a unique target for a call site before inlining can take place [10].

6.2 Memory Access Optimizations

We use an intra-basic-block transformation we call *register forwarding* to reduce the number of unnecessary loads from memory. The opportunity for this optimization arises because, in the course of other optimizations such as the elimination of unreachable code, register reassignment and elimination of unnecessary register saves and restores at function boundaries, etc., `alto` is able to free up registers that can then be reused for other purposes. In the simplest case, a register $\mathtt{r}_a$ is stored to a memory location $addr_0$, and a register $\mathtt{r}_b$ subsequently loaded from that address, with no redefinition of $\mathtt{r}_a$ in between. In this case, assuming that we can verify that the contents of location $addr_0$ have also not been modified, register forwarding replaces the load operation by a register-to-register move from $\mathtt{r}_a$:

store $\mathtt{r}_a$, $addr_0$		store $\mathtt{r}_a$, $addr_0$
...	$\Rightarrow$	...
load $\mathtt{r}_b$, $addr_0$		move $\mathtt{r}_a$, $\mathtt{r}_b$

In general, register $\mathtt{r}_a$ may be modified after it has been stored to location $addr_0$ but before $\mathtt{r}_b$ is loaded from that location. In this case (again assuming that location $addr_0$ can be guaranteed to not have been modified), if there is a free register $\mathtt{r}_{tmp}$, it can be used to save the original value of $\mathtt{r}_a$ before $\mathtt{r}_a$ is modified, and eventually moved over to $\mathtt{r}_b$.

In order to guarantee that the memory location $addr_0$ is not modified between the initial store of $\mathtt{r}_a$ to it and the subsequent load into $\mathtt{r}_b$, we verify that any intervening stores to memory write to locations other than $addr_0$. For this, we use a slight generalization of a technique called *instruction inspection*, commonly used in compile-time instruction schedulers. we first carry out interprocedural constant propagation to identify references to global addresses. The memory disambiguation analysis then proceeds as follows: two memory reference instructions i_1 and i_2 in a basic block can be guaranteed to not refer to the same memory location if one of the following holds:

1. one of the instructions uses a register known to point to the stack and the other uses a register known to point to a global address; or

Program	Source lines	Bigloo			Gambit-C		
		functions	blocks	instructions	functions	blocks	instructions
boyer	568	2061	24358	114007	1050	39004	188178
conform	432	2080	24689	115809	1036	39388	190257
dynamic	2318	2202	27633	132576	1050	43716	220461
earley	651	2069	24608	115928	1050	39319	191091
graphs	602	2079	24538	115885	1050	39200	189977
lattice	219	2061	24331	113994	1050	39016	188451
matrix	763	2091	24746	116729	1050	39734	192569
nucleic	3478	2162	27131	126612	1050	40257	199192
scheme	1078	2301	26333	123465	1050	41479	202127

Figure 2: The benchmark programs used

2. i_1 and i_2 use address expressions $k_1(r_1)$ and $k_2(r_2)$ respectively, and there are two (possibly empty) chains of instructions whose effects are to compute the value $c_1 + \mathit{contents\_of}(r_0)$ into register r_1 and $c_2 + \mathit{contents\_of}(r_0)$ into r_2, for some register r_0, such that the two chains do not use different definitions of r_0 in the basic block under consideration, and $c_1 + k_1 \neq c_2 + k_2$.

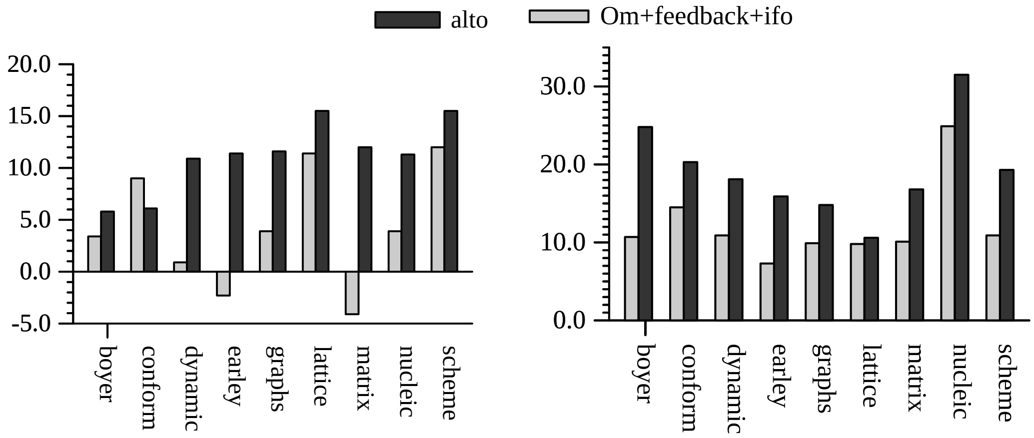
Apart from this transformation, *shrink-wrapping* [6] is used to reduce register save/restore operations at function call boundaries.

6.3 Profile-Directed Code Layout

In order to reduce the performance penalty associated with control flow changes and instruction cache misses, `alto` uses profile information to direct the layout of the code (modern processors typically use dynamic branch prediction, so the effect of code layout on branch misprediction is not considered). The algorithm used here follows that of Pettis and Hansen [11], with a few minor modifications. The code layout algorithm proceeds by grouping the basic blocks in a program into three sets: The *hot set* consists of the set of basic blocks, considered in decreasing order of execution frequency, which account for 2/3 of the total number of instructions executed by the program at runtime; the *zero set* contains all the basic blocks that were never executed; and The *cold set* contains the remaining basic blocks. We then compute the layout separately for each set using a greedy algorithm to construct chains of basic blocks, and concatenate the three resulting layouts to obtain the overall program layout.

7 Experimental Results

We evaluated our link-time optimizer using two optimizing Scheme compilers: Bigloo version 1.8, by M. Serrano [14], and Gambit-C version 3.0 by Marc Feeley. Our experiments were run using nine commonly used Scheme benchmarks: *boyer*, a term-rewriting theorem prover; *conform* is a type checker, written by J. Miller; *dynamic* is an implementation of a tagging optimization algorithm for Scheme [9], applied to itself; *earley* is an implementation of Earley's parsing algorithm, by Marc Feeley; *graphs*, a program that counts the number of directed graphs with a distinguished root and k vertices each having out-degree at most 2; *lattice* enumerates the lattice of maps between two lattices; *matrix* tests whether a given random matrix is maximal among all matrices of the same dimension obtainable via a set of simple transformations of the original matrix; *nucleic* is a floating-point intensive program to determine nucleic acid structure; and *scheme* is a Scheme interpreter by Marc Feeley. The size of each of these



(a) Gambit-C

(b) Bigloo

Figure 3: Speed improvements (%)

benchmarks is reported in Figure 2.[2] We considered only compiled systems, and restricted ourselves to compilers that translated Scheme programs to C code because `alto` requires relocation information to reconstruct the control flow graph from an executable program, which means that the linker needs to be invoked with the appropriate flags that instruct it to not discard the relocation information; systems that compiled to C seemed to offer the simplest way to communicate the appropriate flags to the linker.

The Bigloo compiler was invoked with options `-O4 -farithmetic -unsafe -cgen`, except for the *nucleic* program, for which the options used were `-O3 -unsafesv -cgen`. The Gambit-C compiler was invoked without any additional compiler options, but the resulting C code had the switch `-D___SINGLE_HOST` passed to the C compiler to generate faster code The resulting C code was compiled with the DEC C compiler V5.2-036 (the highly optimizing GEM compiler system [3], which we found generates faster code than current versions of `gcc`) invoked as `cc -O4`, with additional flags to retain relocation information and produce statically linked executables. The profiling inputs used were the same as that used for the actual benchmarking. The timings were obtained on a lightly loaded DEC Alpha workstation with a 300 MHz Alpha 21164 processor with a split primary 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 smallest time of 15 runs is considered. Measurements of the number of different kinds of operations executed, cache misses, etc., were obtained using hardware counters on the processor, using the best number out of 5 runs.

Figure 3 shows the execution time improvements due to `alto`, compared to what is achievable otherwise using aggressive compile-time optimization (at level `-O4`), together with profile-guided and inter-file optimization as well as link-time optimization using the Om link-time optimizer [15]. There are two main points to note from this figure. First, note that in almost all cases—the sole exception is *conform* under Gambit-C—`alto` produces code that is significantly faster than that produced using Om.

[2] The numbers reported here are for the programs available with the Gambit-C 2.7 distribution (`http://www.iro.umontreal.ca/~gambit`), measured for the "core program", i.e., without system-specific definitions, using the `wc` utility. Of course, "lines of code" is not really an appropriate measure of size for link-time optimization, and these numbers are shown only to provide some intuition: a more appropriate measure, for our purposes, is the number of instructions in the final executable, as reported in this table.

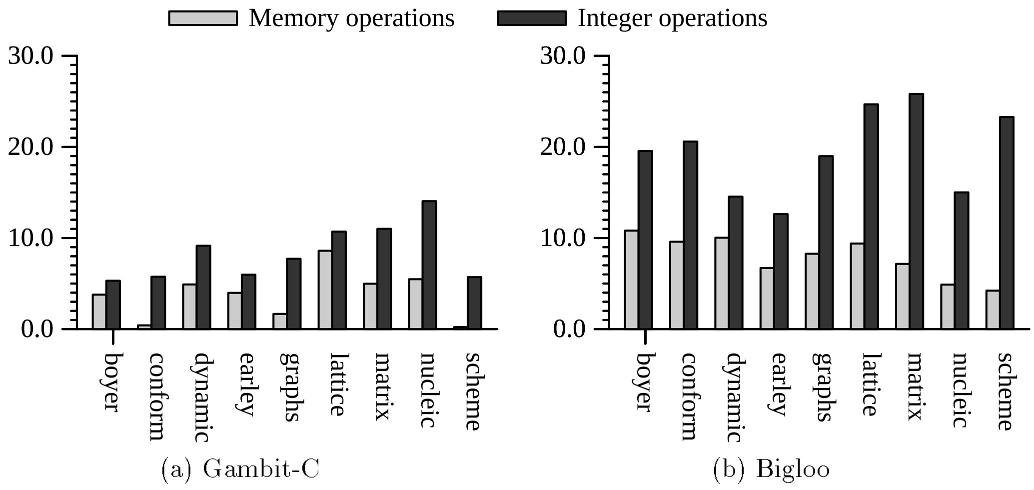


Figure 4: Reduction in operations executed (%)

The second point is that, even though the programs were subjected to a high degree of optimization by both the Scheme and the C compilers, `alto` nevertheless succeeds in achieving significant further improvements in performance. The improvements for Bigloo range from about 10% to over 30%, with an average improvement of 19.3%. The improvements are smaller for Gambit-C, ranging from about 6% to about 15.5%, with an average improvement of about 10.2%.[3]

Figure 1 indicates that for the programs considered, there are three main classes of operations executed: memory operations, integer operations, and branch operations. Figure 4 shows the effect of `alto` on the number of memory and integer operations executed; while `alto` is able, in some cases, to reduce number of branch operations executed, the reductions are generally not large enough to be significant. It can be seen that for Gambit-C, `alto` is able to effect a reduction of around 4% in the number of memory operations and 5–10% in the number of integer operations; for Bigloo the improvements are more dramatic, with 5–10% reductions in the number of memory operations and 15–25% reductions in the number of integer operations executed. The vast majority of the memory operations eliminated turn out to be load operations, which are typically more expensive than store operations.

Table 1 shows the effect of `alto` on the instruction cache behavior of the programs. The group of columns marked 'ORIGINAL' refers to the original program, while those grouped under 'OPTIMIZED' refer to the output of `alto`; the column marked 'Δ Accesses' refers to the percentage improvement in the number of i-cache accesses from the original to the optimized program, relative to the original program. We see, from the ΔAccesses column, that the number of i-cache accesses, i.e., the number of instruction accesses, generally decreases after optimization: the sole exception is the *scheme* benchmark under Gambit-C, which experiences an increase of over 9% in the number of i-cache accesses. We are currently investigating the reason for this anomaly. In all cases, however, both the number of i-cache misses, and the i-cache miss rate, decrease dramatically, primarily due to profile-guided code layout.

[3] These averages were computed as follows: for each of the systems considered, we determined the ratio of the execution times after optimization to the original execution time (i.e., Opt./Orig. in Table 3), computed the geometric mean, and subtracted this from 1.00.

Program	Original			Optimized			
	Accesses ($\times 10^6$)	Misses ($\times 10^6$)	Miss Rate (%)	Accesses ($\times 10^6$)	Misses ($\times 10^6$)	Miss Rate (%)	ΔAccesses (%)
boyer	1252.26	1.27	0.10	907.34	0.58	0.06	27.54
conform	453.70	10.10	2.22	398.00	1.19	0.30	12.27
dynamic	395.96	6.43	1.62	338.82	2.14	0.63	14.43
earley	974.06	2.68	0.27	920.58	0.78	0.08	5.49
graphs	1495.59	32.62	2.18	1320.81	6.02	0.45	11.68
lattice	2491.41	3.85	0.15	2443.31	1.55	0.06	1.93
matrix	2637.69	23.64	0.89	2262.69	11.46	0.50	14.21
nucleic	1793.26	164.36	9.16	1447.69	30.37	2.09	19.27
scheme	2767.45	30.47	1.10	2216.75	7.15	0.32	19.89

(a) Bigloo

Program	Original			Optimized			
	Accesses ($\times 10^6$)	Misses ($\times 10^6$)	Miss Rate (%)	Accesses ($\times 10^6$)	Misses ($\times 10^6$)	Miss Rate (%)	ΔAccesses (%)
boyer	1230.11	9.78	0.79	1084.09	1.06	0.09	11.87
conform	913.96	27.24	2.98	866.91	16.95	1.95	5.14
dynamic	1735.13	42.31	2.43	1518.46	14.46	0.95	12.48
earley	1274.87	23.57	1.84	1185.54	0.96	0.08	7.00
graphs	433.58	21.51	4.96	432.86	3.91	0.90	0.16
lattice	2976.57	77.80	2.61	2729.05	1.93	0.07	8.31
matrix	2470.24	71.63	2.89	2213.54	28.21	1.27	10.39
nucleic	377.09	14.76	3.91	358.74	7.97	2.22	4.86
scheme	3126.19	318.07	10.17	3413.83	17.90	0.52	–9.20

(b) Gambit

Table 1: Instruction Cache behavior

Table 2 shows the effect of `alto` on the data cache behavior of programs tested. While `alto` does not do anything to change data layouts, it can be seen that for most programs there are noticeable reductions in the number of data cache accesses: we believe that this is likely to be due to the elimination of load operations by `alto`. Surprisingly, in a few programs the reduction in data cache accesses is accompanied by an increase in data cache misses: we conjecture that this may be because some load operations, which would have caused nearby memory words to be brought into the cache, were eliminated by `alto`, and that this resulted in cache misses when those nearby words were accessed.

The amount of code growth due to inlining ranges from about 0.5% for the Gambit-C system, where very little inlining takes place, to about 1.5% for Bigloo. For either implementation, the amount of inlining does not seem to vary significantly for the different benchmarks, suggesting that most of the inlining involves library routines; this is not surprising, since the programs used were single-module programs and one would expect the Scheme and C compilers to have inlined most of the obvious candidates at compile time.

Program	Original			Optimized			ΔAccesses (%)
	Accesses ($\times 10^6$)	Misses ($\times 10^6$)	Miss Rate (%)	Accesses ($\times 10^6$)	Misses ($\times 10^6$)	Miss Rate (%)	
boyer	578.28	49.90	8.00	515.76	47.92	9.00	10.81
conform	284.95	18.35	6.00	257.55	21.28	8.00	9.61
dynamic	207.61	26.06	12.00	186.84	25.01	13.00	10.00
earley	649.46	59.11	9.00	605.88	57.47	9.00	6.71
graphs	730.72	128.23	17.00	670.17	96.32	14.00	8.28
lattice	1844.44	46.35	2.00	1671.10	92.60	5.00	9.39
matrix	1463.61	154.46	10.00	1358.69	146.29	10.00	7.16
nucleic	1105.26	79.70	7.00	1051.39	73.54	6.00	4.87
scheme	1331.42	167.60	12.00	1275.19	162.43	12.00	4.22

(a) Bigloo

Program	Original			Optimized			ΔAccesses (%)
	Accesses ($\times 10^6$)	Misses ($\times 10^6$)	Miss Rate (%)	Accesses ($\times 10^6$)	Misses ($\times 10^6$)	Miss Rate (%)	
boyer	728.49	108.08	14.00	700.84	100.10	14.00	3.79
conform	574.35	38.73	6.00	572.06	37.73	6.00	0.39
dynamic	1301.02	101.28	7.00	1237.12	82.82	6.00	4.91
earley	1105.85	50.79	4.00	1061.87	50.23	4.00	3.97
graphs	422.90	20.85	4.00	415.82	20.79	5.00	1.67
lattice	2165.83	76.89	3.00	1979.51	64.55	3.00	8.60
matrix	1816.00	131.53	7.00	1725.56	118.03	6.00	4.98
nucleic	301.20	43.53	14.00	284.68	47.74	16.00	5.48
scheme	2077.62	306.80	14.00	2072.70	302.66	14.00	0.23

(b) Gambit

Table 2: Data Cache behavior

We believe that the performance improvements reported here are conservative, because the benchmarks used don't really have the characteristics where link-time optimization can be expected to pay off significantly. Each benchmark consists of a single file; the use of libraries is limited to system primitives (i.e., there is very little code reuse at the user level); and the programs don't use more than one language. In the near future we intend to investigate larger, more realistic, multi-module benchmarks: we believe link-time optimization will offer even greater benefits for such applications.

8 Conclusions

The traditional model of compilation is unable to optimize code that is not available for inspection at compile time. This means that applications that make extensive use of library routines, or where different modules are written in different languages, may incur a performance penalty. One way to address this problem is to apply low level code optimizations at link time. However, the manipulation of machine code has challenges of its own, including increased program size and difficulty in extracting information about program behavior.

This paper describes `alto`, a link-time optimizer we have developed for the DEC Alpha architecture, and our experiences with its application to several Scheme benchmarks, using code generated by three different optimizing Scheme compilers. Even though the benchmarks lack the features that would show off the benefits of link-time optimization, and were compiled with high levels of compiler optimization (both at the Scheme and C level), we find that `alto` is able to achieve significant performance improvements.

Acknowledgements

We are grateful to Jeffrey Siskind and Marc Feeley for their invaluable help with benchmarking. Thanks are also due to Suresh Jagannathan for providing some of the benchmarks.

References

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

[2] J. M. Ashley, "The Effectiveness of Flow Analysis for Inlining", *Proc. 1997 SIGPLAN International Conference on Functional Programming*, June 1997, pp. 99–111.

[3] D. Blickstein *et al.*, "The GEM Optimizing Compiler System", *Digital Technical Journal*, 4(4):121–136.

[4] M. Blume and A. W. Appel, "Lambda-splitting: A Higher-Order Approach to Cross-Module Optimizations", *Proc. 1997 SIGPLAN International Conference on Functional Programming*, June 1997, pp. 112–124.

[5] B. Calder and D. Grunwald, "Reducing Indirect Function Call Overhead in C++ Programs", *Proc. 21st ACM Symposium on Principles of Programming Languages*, Jan. 1994, pp. 397–408.

[6] 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.

[7] 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.

[8] D. Grove, J. Dean, C. Garrett, and C. Chambers, "Profile-Guided Receiver Class Prediction", *Proc. Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '95), Oct. 1995, pp. 108–123.

[9] F. Henglein, "Global Tagging Optimization by Type Inference", *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pp. 205–215.

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

[11] 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.

[12] 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", *Proc. 1997 USENIX Windows NT Workshop*.

[13] 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

[14] 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.

[15] 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.

[16] 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.

[17] P. A. Steenkiste, "The Implementation of Tags and Run-Time Type Checking", in *Topics in Advanced Language Implementation*, ed. P. Lee, 1991. MIT Press.

[18] O. Waddell and R. K. Dybvig, "Fast and Effective Procedure Inlining", *Proc. 1997 Static Analysis Symposium* (SAS '97), Sept. 1997, pp. 35–52. Springer-Verlag LNCS vol. 1302.

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

[20] 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.