`alto` : A Link-Time Optimizer for the DEC 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

Technical Report 98-14

December 9, 1998

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 DEC 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 and CCR-9711166. 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. This is addressed to some extent in procedure-call-intensive programming languages, such as Prolog and Scheme, that rely greatly on inter-procedural analyses and optimizations [11, 24, 32, 36]; recent years have also seen a great deal of interest in inter-procedural analyses for languages like C (see, for example, [13, 14, 25, 29, 38]). However, even here the scope of the possible analyses and optimizations 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, cannot be effectively optimized. Other optimizations, e.g., to reduce the cost of address computations [35] requires information not available at link 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 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. 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 such an 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 DEC 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 [34] supplied by DEC.

The remainder of the paper is organized as follows: Section 2 describes the overall organization of `alto`. Section 3 discusses how control flow analysis is carried out. Section 4 describes the analyses carried out by `alto`, Section 5 describes the optimizations that are performed, and Section 6 gives performance results. Section 7 summarizes work related to ours. Finally, Section 8 concludes.

[1] `Alto` can use either basic block profiles, generated using the *pixie* tool, or basic block and edge profiles that it can itself generate; we are currently extending the system to also generate value profiles [5] at specific points of interest.

2 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, 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 [27], instruction scheduling, and insertion of no-ops for alignment purposes, after which the code is written out.

3 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. 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 4.1) 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.

4 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.

4.1 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 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

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

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.[2]

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 5.1), 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. This information can, nevertheless, be used to advantage in many cases, e.g., by replacing register operands by immediate operands.

As shown in Table 2, this analysis has a profound impact on the performance of the generated code. For example, the SPEC-95 benchmarks *li*, *m88ksim*, *perl*, and *vortex* suffer slowdowns of 15–20% when this analysis is turned off. 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,[3] which can significantly hamper optimizations. Finally, knowledge of constant addresses is useful for optimizations such as the removal of unnecessary memory references (Section 5.3) and strength reduction in constant computations (Section 5.2).

4.2 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

[2] 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.

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

Program	*Execution Time* (sec)		*Improvement*
	without analysis	with analysis	(%)
compress	280.21	259.46	0.926
gcc	248.72	229.74	0.924
go	341.19	304.35	0.892
ijpeg	333.29	328.90	0.987
li	293.15	248.62	0.848
m88ksim	267.89	210.98	0.788
perl	216.64	181.77	0.839
vortex	391.47	312.55	0.798

Table 2: Performance impact of interprocedural constant propagation

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 inter-procedural control flow, and a procedure does not have to be re-analyzed for its various call-sites [2, 3, 13, 29]. 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 [14, 25, 38]. 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 3. 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: details of the dataflow equations are given in the Appendix.

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. $\mathit{mayUse}[f]$, 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. $\mathit{mayUse}(f)$ 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. $\mathit{byPass}[f]$. The set of registers which, if live at n_r, will also be live at n_c.

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.

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

Table 3: Effect of Liveness Analysis on Load Instructions Executed

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. [4] 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 3 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 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. 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 6.

[4] 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 4: Experimental Results: Unreachable Code Elimination

5.1 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 block 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.

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 dead, and then basic blocks are marked reachable if they can be reached by another block that is reachable. The entry point of the program is always reachable. This analysis also makes use of context information as a basic block that follows a function call will be marked reachable if the corresponding call site is reachable, rather than the function that is called as the function could already be reachable due to a call from another call site.

The amount of unreachable code detected in our benchmarks is shown in Table 4. These numbers do not include no-ops inserted into reachable basic blocks for alignment and instruction scheduling purposes. It can be

Program	*Execution Time* (sec)		*Improvement*
	without optimization	with optimization	(%)
compress	269.54	259.46	0.963
gcc	232.24	229.74	0.989
go	304.77	304.35	0.999
ijpeg	329.44	328.90	0.998
li	260.74	248.62	0.954
m88ksim	233.38	210.98	0.904
perl	192.72	181.77	0.943
vortex	338.63	312.55	0.923

Table 5: Performance impact of constant computation optimization

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% [33].

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.

5.2 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)`'.[5] 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 [35]; 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 most be taken to assure the constants involved are not addresses with the code sections of the executable. Since `alto` changes the code section, addresses therein are almost certain to chnage: such constants are therefore excluded from this optimization.

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:

[5] 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.

(a) original code	(b) initial optimized code	(c) final optimized code
(1) `ldq $r1, 16($29)`	(1) `ldq $r1, 16($29)`	(1) `ldq $r1, 16($29)`
(2) `ldq $r2, 96($29)`	(2′) `lda $r2, 8(r1)`	
(3) `ldq $r3, 32($29)`	(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)`

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 whenver 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 Table 5. The programs that benefit the most from this optimization are *compress*, *li*, *m88ksim*, *perl*, and *vortex*, with improvements ranging from 3.7% to 9.6%.

5.3 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 `loads` and `stores` 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 [37]; 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`, three 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.

   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 man-

Program	*Execution Time* (sec)		Ratio
	without optimization	with optimization	
compress	259.10	259.46	1.001
gcc	236.16	229.74	0.973
go	300.69	304.35	1.012
ijpeg	327.19	328.90	1.005
li	253.90	248.62	0.979
m88ksim	210.38	210.98	1.003
perl	184.35	181.77	0.986
vortex	313.78	312.55	0.996

Table 6: Performance impact of memory operation elimination

ner 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* [4] 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 [4], 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 Table 6. The programs that benefit the most from this optimization are *gcc* and *li*, with improvements in the neighborhood of 2–2.5%.

5.4 Partial Dead Code Removal

The code generated for a program may contain instructions that are partially dead, i.e., whose results may not be needed along some execution paths. Alto attempts to remove partially dead code where feasible. The algorithm used is as follows. Let B be a basic block ending in a conditional branch, such that the successor of B if the branch is taken is B_T and the successor if it is not taken is B_F. Let B contain an instruction I that has no side effects (e.g., a system call or a store to memory), and does not have a dependence to any later instruction in B, where an instruction i has a dependence to an instruction j if one of the following holds:

(1) i reads a register or memory location that may be defined by j; or

(2) i defines a register or memory location that may be read or defined by j.

In this case, if the register defined by instruction I is not live at entry to one of the successors of B, it is moved into the other successor. That is, if I is not live on entry to B_T, it is moved from B into B_F, and vice versa.

This transformation can later be undone by the instruction scheduler, which works on extended basic blocks and may decide to move the instruction back into B if this results in a better schedule.

The performance impact of this optimization is shown in Table 7. The greatest impact of this optimization is on *perl*, whose speed improves by 3%.

Program	*Execution Time* (sec)		Ratio
	without optimization	with optimization	
compress	261.16	259.46	0.993
gcc	229.20	229.74	1.002
go	310.07	304.35	0.982
ijpeg	327.91	328.90	1.003
li	248.78	248.62	0.999
m88ksim	213.60	210.98	0.988
perl	187.45	181.77	0.970
vortex	313.14	312.55	0.998

Table 7: Performance impact of partial dead code elimination

5.5 Function 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 5.3). The final reason is to improve branch prediction and instruction cache behavior using profile-directed code layout (cf. Section 5.6). 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 longer than its body;

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

(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, 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 [26], but these have not been implemented within `alto` at this time.

Program	*Number of Instructions*		Ratio
	without optimization	with optimization	
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 8: Code growth due to inlining

Inlining through indirect function calls, e.g., via function pointers in C or due to higher order functions in languages such as Scheme, is generally considered problematic. Traditional C compilers usually do not inline functions that are called indirectly at a call site. Some Scheme compilers deal with higher order functions using sophisticated control flow analyses [22] that are, we believe, too expensive to be practical at the level of machine code. Instead, we use a simple profile-guided inlining technique we call *guarded inlining* to achieve similar results. Suppose we have an indirect function call whose target we are unable to resolve. We use value profiling [5] to identify the most frequent target address at the call: suppose that this is an address $addr_0$, corresponding to a function f. 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 [22].

In the actual low-level realization of this optimization, we take advantage of the facts that (i) the DEC Alpha uses the global register `gp` to compute global addresses; and (ii) at link time, the actual value of the `gp` register at any point in the program is known, which means that `alto` can determine the offset δ between the value of `gp` at the indirect function call and the desired address $addr_0$. Further, in the code prior to this optimization, the return address register `$ra` cannot be live at the point immediately before the call (since the `jsr` instruction will overwrite it), which means we can load $addr_0$ into this register and carry out the comparison, resulting in the following instruction sequence: [6]

```
ldq r0, memloc
lda $ra, δ(gp)          # $ra := gp + δ ≡ addr0
subq $ra, r0, $ra       # $ra := addr0 − r0
bne $ra, ...
⟨ inlined body of function f ⟩
```

The overhead incurred in testing for the common case is, therefore, 3–4 (relatively cheap) instructions. Moreover, in order for an indirect function call to be eligible for inlining, it will have to have a sufficiently high execution count (see above): often, this will be a result of the indirect call being within a loop. In this case, if there are any callee-saved registers available, the amortized cost of guarded inlining can be reduced further by loading the address $addr_0$ into a callee-saved register outside the loop, resulting in an overhead of 2 instructions per loop iteration. Finally, if the indirect function call is within a loop and we can determine that the target address is

[6] If $\delta \geq 2^{16}$, an additional `ldah` instruction is needed, since the `lda` instruction shown can set only the low 16 bits of a register.

Program	*Execution Time* (sec)		Ratio
	without optimization	with optimization	
compress	267.87	259.46	0.969
gcc	230.27	229.74	0.998
go	306.32	304.35	0.994
ijpeg	327.29	328.90	1.005
li	254.10	248.62	0.978
m88ksim	220.43	210.98	0.957
perl	176.12	181.77	1.032
vortex	314.01	312.55	0.995

Table 9: Performance improvements due to inlining

invariant within the loop, we create two versions of the loop: one containing a direct function call to the most frequent target (which subsequently gets inlined at this call site), the other the original indirect function call, and choose among the versions by testing the target address, as discussed above, outside the loop.

The notion of guarded inlining is conceptually very similar to a technique for optimizing dynamically dispatched function calls in object-oriented languages called "receiver class prediction" [20]. The transformation we describe is somewhat more general, for two reasons. First, it does not rely on specific language features such as an inheritance hierarchy, and so is applicable to any language. More importantly, it can be adapted to any indirect jump, not just indirect function calls. For example, in systems that support tail call optimization, it can be used to carry out "inlining" at function return points—recall that in such systems the target of a function return may not be obvious due to tail call optimization, since the callee may not return to its caller—and inline the function being returned to into the body of the function that is returning.

The extent of code growth due to inlining is shown in Table 9. Inlining causes only a modest increase in code size, in most cases in th neighborhood of 1%, and in a few cases leads to small decreases in code size.

The performance improvements resulting from inlining are shown in Table 8. The greatest benefits are observed for *compress*, *li*, and *m88ksim*, with improvements ranging from 2.2% to 4.3%. On the other hand, inlining leads to a slowdown of 3% for *perl*; presumably due to cache effects.

5.6 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

Program	*Execution Time* (sec)		Ratio
	without profiling information	with profiling information	
compress	261.48	259.46	0.992
gcc	276.82	229.74	0.830
go	329.98	304.35	0.922
ijpeg	329.17	328.90	0.999
li	257.71	248.62	0.965
m88ksim	254.59	210.98	0.829
perl	212.72	181.77	0.854
vortex	329.14	312.55	0.950

Table 10: Performance improvements due to profile guided basic block layout

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 [27].

Alto implements two code layout schemes, one that exploits profiling information while the other does not. 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. If profiling information is available, our primary goal is to reduce cache conflicts as far as possible. Given a value ϕ in the interval (0,1], we determine the largest execution frequency threshold N such that, the *hot* basic blocks in a program are defined to be the smallest set of blocks that 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), and contains at least as many instructions as will fit into the 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 code layout algorithm proceeds by grouping the basic blocks in a program into three sets: The *hot set* consists of the hot blocks in the program ($\phi = 0.66$); 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 seperately for each

Program	*Execution Time* (sec)		Ratio
	without optimization	with optimization	
compress	262.41	259.46	0.989
gcc	239.27	229.74	0.960
go	309.74	304.35	0.983
ijpeg	330.31	328.90	0.996
li	262.71	248.62	0.946
m88ksim	230.78	210.98	0.914
perl	190.43	181.77	0.955
vortex	334.42	312.55	0.935

Table 11: Performance improvements due to scheduling

set and concatenate the three resulting layouts to obtain the overall program layout. Our layout algorithm follows the (bottom-up positioning) approach of Pettis and Hansen [27], with minor modifications to address the problems identified by Calder and Grunwald [10]. Currently, `alto` does not carry out procedure placement.

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 Table 10. Many programs can be seen to benefit significantly from profile-directed code layout: the greatest benefits are obtained for *gcc*, *m88ksim*, and *perl*, with improvements of around 17%.

5.7 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 upto 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.

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, this achieves an effect very similar to trace scheduling [17].

The performance impact of instruction scheduling is shown in Table 11. Several programs show performance improvements exceeding 4%, with *m88ksim* showing the largest gain of over 8%.

Program	*Execution Time* (sec)				T_{om}/T_{base}	T_{ifo}/T_{base}	T_{alto}/T_{base}
	Base (T_{base})	Om (T_{om})	Ifo+FB+Om (T_{ifo})	`alto` (T_{alto})			
compress	282.49	276.14	272.49	259.46	0.978	0.965	0.918
gcc	270.12	232.96	229.52	229.74	0.862	0.850	0.851
go	340.32	301.38	300.70	304.35	0.886	0.884	0.894
ijpeg	337.63	329.95	333.57	328.90	0.977	0.988	0.974
li	315.08	292.44	289.36	248.62	0.928	0.918	0.789
m88ksim	325.78	255.44	231.63	210.98	0.784	0.711	0.648
perl	246.64	209.54	203.94	181.77	0.850	0.827	0.737
vortex	469.56	394.38	396.14	312.55	0.840	0.844	0.666
Geometric Mean:					0.886	0.869	0.802

Table 12: Performance results: C Programs

6 Performance Results

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 [34].

Because most of the development and testing of `alto` was carried out using C benchmarks, we wanted to evaluate its effect on code generated from source programs in very different languages. To this end, we also tested it on a set of Prolog and Scheme programs, whose low-level dynamic characteristics are considerably different from those of C programs [12]. These are both dynamically typed languages, which means that, unlike C, there is extensive low-level pointer arithmetic to add and remove “tag bits” that are used to attach type information to pointers. Further, the presence of garbage collection induces a greater proportion of memory references in these languages. The control flow characteristics of Scheme and Prolog programs can also be expected to be different than those of C programs, due to the use of higher-order and/or `call-cc` constructs in Scheme and backtracking in Prolog.

Our results are shown in Tables 12, 13 and 14. The timings were obtained on a lightly loaded DEC 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 is the smallest time of 15 runs.

6.1 C Programs

The benchmarks we used to test the effect of `alto` on C programs were the eight programs in the SPEC-95 integer benchmark suite; characteristics of these programs are given in Appendix B.1. The execution times reported are for the SPEC reference inputs.

For processing by `alto`, the programs were compiled with the DEC C compiler V5.2-036 invoked as `cc -O4`, with linker options to retain relocation information and to produce statically linked executables. The execution times for these executables is given under the column labelled “Base” (T_{base}). These executables were instrumented using `pixie` and executed on the SPEC training inputs to obtain an execution profile that was provided to `alto`, which was invoked with default switches together with a flag to use this profile information. The execution times of the executables produced by `alto` are reported in the column labelled `alto` (T_{alto}). The column labelled T_{base}/T_{alto} gives the improvement obtained from using `alto`.

We also compared the performance improvements obtained using `alto` with those obtained using the OM link-time optimizer from DEC [34]. 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

```
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. The execution times obtained with the resulting executables are reported in the column labelled "Om" (T_{om}). The column labelled T_{base}/T_{om} gives the improvement obtained from using OM.

Finally, we measured the performance achievable using 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 [34]. 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.

Program	*Execution Time* (sec)				T_{om}/T_{base}	T_{ifo}/T_{base}	T_{alto}/T_{base}
	Base (T_{base})	Om (T_{om})	Ifo+FB+Om (T_{ifo})	`alto` (T_{alto})			
boyer	10.33	10.02	9.97	9.73	0.970	0.966	0.942
conform	8.44	7.67	7.68	7.92	0.909	0.910	0.939
dynamic	17.39	16.42	17.23	15.49	0.944	0.991	0.891
earley	11.86	11.97	12.13	10.51	1.009	1.023	0.886
graphs	4.55	4.37	4.38	4.03	0.959	0.961	0.884
lattice	27.11	24.19	24.01	22.92	0.892	0.886	0.845
matrix	22.96	23.67	23.91	20.22	1.031	1.041	0.880
nucleic	4.34	4.16	4.17	3.85	0.960	0.961	0.887
scheme	33.92	29.70	29.86	28.66	0.875	0.880	0.845
Geometric Mean:					0.949	0.956	0.888

(a) Gambit-C

Program	*Execution Time* (sec)				T_{om}/T_{base}	T_{ifo}/T_{base}	T_{alto}/T_{base}
	Base (T_{base})	Om (T_{om})	Ifo+FB+Om (T_{ifo})	`alto` (T_{alto})			
boyer	10.04	8.89	8.96	7.55	0.885	0.893	0.752
conform	3.81	3.24	3.26	3.04	0.850	0.855	0.797
dynamic	3.95	3.56	3.52	3.24	0.900	0.891	0.819
earley	8.52	8.00	7.90	7.17	0.939	0.927	0.841
graphs	11.98	10.74	10.79	10.21	0.896	0.901	0.852
lattice	18.50	16.06	16.69	16.55	0.868	0.902	0.894
matrix	21.51	19.27	19.35	17.89	0.896	0.899	0.832
nucleic	18.52	13.91	13.90	12.69	0.751	0.751	0.685
scheme	21.92	19.35	19.53	17.72	0.882	0.891	0.808
Geometric Mean:					0.873	0.877	0.807

(b) Bigloo

Table 13: Performance results: Scheme Programs

The execution times of the resulting executables are given in the column labelled "Ifo+FB+Om"[7] (T_{ifo}).

It can be seen, from Table 12, that for most of the programs tested, the executable obtained using `alto` is faster than those obtained using OM and Ifo+FB+Om. In several cases, the difference in the improvements is quite significant: for example, *li* gets an 8% improvement with both OM and Ifo+FB+Om, compared to a 21% improvement with `alto`. Interestingly, we find that—with the exception of *m88ksim* and *perl*—the use of profile-guided inter-file optimization within the compiler has very little 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 13%; by contrast, link-time optimization using `alto` produces an average improvement of just under 20%.

6.2 Scheme Programs

To evaluate `alto` on Scheme programs, we used two different optimizing Scheme compilers: Bigloo version 1.8, by Serrano [31], and Gambit-C version 3.0 by Feeley [15]. Our experiments were run using nine commonly used Scheme benchmarks, whose characteristics are reported in Appendix B.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 the options `-O4 -unsafe -farithmetic -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. The resulting C programs were compiled as described in Section 6.1, with the Gambit-C benchmarks additionally having the switch `-D____SINGLE_HOST` passed to the C compiler to generate faster code. The profiling inputs used, for each of `alto`, Om, and Ifo+FB+Om, were the same as that used for the actual benchmarking.

It can be seen, from Table 13, that the OM link-time optimizer achieves an improvement of about 5% on the average for Gambit-C and about 12% for Bigloo programs. As for the C benchmarks, these numbers do not change much when inter-file and profile-directed optimization is used in conjunction with OM. On the other hand, `alto` produces significant improvements for all of the benchmarks used, and yields executables that are uniformly faster than those obtained using Om and Ifo+FB+Om. On average, `alto` produces an overall improvement of about 11% for Gambit-C and about 20% for Bigloo.

6.3 Prolog Programs

To evaluate `alto` on Prolog programs we used `wamcc` version 2.21, by Codognet and Diaz [6]. This system relies on extensions to C implemented within *gcc*, which precludes a comparison against the effects of `alto` with those of profile-directed inter-file and link-time optimization within `cc`. The programs were compiled with the option `-fast_math`, and the resulting C code compiled using *gcc* version 2.7.2.2 at optimization level `-O2`, with additional flags to produce a statically linked executable and retain relocation information. The resulting executables were profiled using the same inputs as were used for the actual benchmarking. The benchmarks we used are part of the `wamcc` distribution: their characteristics are reported in Appendix B.3.

As can be seen from Table 14, `alto` produces significant performance improvements. The overall improvement is about 35% on the average, with one program (*sendmore*) experiencing an improvements of 46%.

[7] Here, Ifo stands for inter-file optimization; FB for feedback; and Om for the *om* link-time optimizer.

Program	*Execution Time* (sec)		T_{alto}/T_{base}
	Base (T_{base})	`alto` (T_{alto})	
boyer	23.27	15.09	0.649
chat	21.28	14.48	0.680
nand	20.88	12.70	0.608
poly10	20.98	12.87	0.614
reducer	19.27	12.83	0.666
sendmore	19.11	10.32	0.540
tak	20.97	19.17	0.914
zebra	22.26	13.88	0.624
Geometric Mean:			0.655

Table 14: Performance results: Prolog Programs

7 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 [30] and Wall [37]. The Zuse Translation System [9] and the `mld` link-time optimizer [16] are aimed at reducing the cost of abstraction in object-oriented languages. These works rely on specially engineered compilers that produce either object files containing special annotations to assist the link-time optimizer [37], 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 [9, 16, 30]. One implication of this is that 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 [23], but unlike `alto`, this system does not carry out interprocedural analysis and optimizations.

The systems that are the closest to ours are the OM [34, 35], Spike [8], and Etch [28] 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 DEC Alpha and Intel x86 processors respectively. Spike carries out three different optimizations [8]: hot-cold optimization [7], 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 6) indicate that the code produced by `alto` is considerably faster than that produced by OM. Unfortunately, because Spike and Etch run on a different operating system than `alto`, it was not possible to compare `alto` with these two systems.

8 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 DEC 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 [34] supplied by DEC.

Acknowledgements

We are grateful to Craig Neth for his help with the use of OM and feedback-directed optimization; and to Jeffrey Siskind and Marc Feeley for their help with benchmarking the Scheme programs.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] P. Codognet and D. Diaz, "`wamcc`: Compiling Prolog to C", *Proc. Twelfth International Conference on Logic Programming*, June 1995, pp. 317–332. MIT Press.

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

[8] 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.

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

[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] K. De Bosschere, S. K. Debray, D. Gudeman, and S. Kannan, "Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages", *Proc. 21st. ACM Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994, pp. 409–420.

[12] S. K. Debray, R. Muth, and S. Watterson, "Link-Time Improvement of Scheme Programs", *Proc. 8th. International Conference on Compiler Construction (CC'99)*, March 1999 (to appear).

[13] 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.

[14] 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.

[15] M. Feeley, *Gambit-C, version 2.8c: a portable implementation of Scheme*, Edition 2.8c, Dept. of Computer Science and Operations Research, University of Montreal, Feb. 1998.

[16] 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.

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

[18] 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.

[19] R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.

[20] 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.

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

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

[23] 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.

[24] D. Krantz, *ORBIT: An Optimizing Compiler for Scheme*, Ph.D. Dissertation, Yale University, 1988. (Also available as Technical Report YALEU/DCS/RR-632, Dept. of Computer Science, Yale University, Feb. 1988.)

[25] 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.

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

[27] 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.

[28] 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).

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

[30] 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

[31] 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.

[32] O. Shivers, "Control Flow Analysis in Scheme", *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988, pp. 164–174.

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

[34] 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.

[35] 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.

[36] P. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

[37] D. W. Wall, “Global Register Allocation at Link Time”, *Proc. SIGPLAN ’86 Symposium on Compiler Construction*, July 1986, pp. 264–275.

[38] 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.

A Dataflow Equations for Interprocedural Register Liveness Analysis

Our analysis proceeds in three phases, as described below. The first two phases compute summary information for functions; the third phase then uses the summary information to do the actual liveness computation. The basic dataflow equations we use are:

$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \begin{cases} \cup\{in[s] \mid s \text{ a successor of } n\} & \text{if } n \text{ is not a call node} \\ mayUse[f] \cup (in[n'] \cap byPass[f]) & \text{if } n \text{ is a call node to a function } f \text{ and} \\ & n' \text{ the corresponding return node} \end{cases}$$

Throughout the analysis, we use the following equations for $\mathsf{B}_{unknown}$ and $\mathsf{F}_{unknown}$:

$$\begin{array}{llll} in[\mathsf{B}_{unknown}] & = \text{all registers} & mayUse[\mathsf{F}_{unknown}] & = \text{all registers} \\ out[\mathsf{B}_{unknown}] & = \text{all registers} & byPass[\mathsf{F}_{unknown}] & = \text{all registers} \end{array}$$

Phase 1 : *Computation of* $byPass[f]$: iteratively compute the least fixpoint of the basic dataflow equations listed above, augmented with the equation

$$byPass[f] = in[entry(f)]$$

and with the modification that information is not propagated into the exit node of any function, i.e., $out[n]$ is always taken to be the set of all registers for an exit node n. The initial values used are the following:

$$in[n] \ = \ use[n]$$

$$out[n] = \begin{cases} \text{all registers} & \text{if } n \text{ is the exit node of a function} \\ \emptyset & \text{otherwise} \end{cases}$$

$$byPass[f] = mayUse[f] = \emptyset \text{ for all functions } f \ (f \neq \mathsf{F}_{unknown}).$$

Once the fixpoint has been computed, $byPass[f]$ is determined as $in[entry(f)]$ for each function f.

The principal difference between these dataflow equations and those used by Goodwin [18] lies in the equation for $ByPassIn[n]$: we have added (unioned) $use[n]$ on the right hand side. Since $use[n] \subseteq MayUse[n]$ our equation for $ByPassIn[n]$ still lies within the bounds given above. The major virtue of this change is that it makes the equations of this and the following phases so similar that one fixpoint computation algorithm can be used for all 3 phases.

Phase 2 : *Computation of* $mayUse[f]$: iteratively compute the least fixpoint of the basic dataflow equations listed above, augmented with the equation

$$mayUse[f] = in[entry(f)]$$

and with the modification, as in Phase 1, that information is not propagated into the exit node of any function. The initial values used are the following:

$$\begin{array}{ll} \left.\begin{array}{ll} in[n] & = use[n] \\ out[n] & = \emptyset \end{array}\right\} & \text{for all nodes } n \ (n \neq \mathsf{B}_{unknown}) \\ mayUse[f] = \emptyset & \text{for all functions } f \ (f \neq \mathsf{F}_{unknown}). \end{array}$$

Once the fixpoint has been computed, $mayUse[f]$ is determined as $in[entry(f)]$ for each function f.

Phase 3 : *Computation of liveness information*: iteratively compute the least fixpoint of the basic set of dataflow equations listed above, starting out with no registers being live at the exit nodes but allowing propagation of liveness information into the exit nodes. The values of $byPass[f]$ and $mayUse[f]$ are those determined in Phases 1 and 2. The initial values used are the following:

$$\left.\begin{array}{ll} in[n] & = use[n] \\ out[n] & = \emptyset \end{array}\right\} \quad \text{for all nodes } n \ (n \neq \mathsf{B}_{unknown})$$

$$mayUse[f] = \emptyset \qquad \text{for all functions } f \ (f \neq \mathsf{F}_{unknown}).$$

It turns out that the *in* and *out* sets computed in this phase must contain, or be equal to, the sets computed in Phase 2. The last fixpoint iteration therefore does not to start from scratch, but can start with the results from the previous run.

Since the dataflow equations of the three phases are very uniform it is not hard to see that the interative fixpoint computation will converge and could even be done in parallel. However, if the three phases described above are executed sequentially the space used to hold $ByPassOut[n]$ and $ByPassIn[n]$ in Phase 1 can be used to hold $MayUseOut[n]$ and $MayUseIn[n]$ in Phase 2 which in turn can be reused in Phase 3 to hold $LiveIn[n]$ and $LiveOut[n]$. So not only can we use an almost identical algorithm for all three phases, the algorithm also uses identical memory locations. Furthermore, it is safe to initialize $LiveIn[n] := MayUseIn[n]$ and $LiveOut[n] := MayUseOut[n]$ thereby accelerating Phase 3 which does not need to start the fixpoint iteration from scratch.

Next we describe how to improve Phase 3 more drastically exploiting the following observation. For a register r at node n of function f, we have

$$r \in LiveOut[n] \Rightarrow r \in MayUseOut[n] \ \vee \ r \in ByPassOut[n] \tag{1}$$

Conversely,

$$r \in MayUseIn[n] \Rightarrow r \in LiveIn[n] \tag{2}$$

But $r \in ByPassIn[n] \not\Rightarrow r \in LiveIn[n]$. The latter does not hold because our initial values for *ByPassOut* of the exit nodes was pessimistic; we essentially assumed that all registers could be live. During Phase 3 it might turn out that not all registers are live at some exit nodes. The correct condition is

$$r \in ByPassOut[n] \ \wedge \ r \in LiveOut[ExitNode[f]] \Rightarrow r \in LiveOut[n] \tag{3}$$

This suggests the following alternative approach for Phase 3 which has the virtue that it only iterates over the call graph rather than the much bigger supergraph.

```
FOREACH n ∈ Nodes DO
   LiveOut[n]:= MayUseOut[n]
   LiveIn[n] := MayUseIn[n]
REPEAT
   change := false
   FOREACH f ∈ Functions DO
      new_out := ⋃ s∈Succ[ExitNode[f]] :  LiveIn[s]
      IF new_out ≠ LiveOut[ExitNode[f]] THEN
         change := true
         liveOut[exit[f]] := new_out
         FOREACH n ∈ Nodes[f] DO
            LiveOut[n]:= MayUseOut[n]∪(ByPassOut[n] ∩ new_out)
            LiveIn[n] := MayUseIn[n] ∪(ByPassIn[n] ∩ new_out)
UNTIL ¬ change
```

We begin by setting the start values for the fixpoint iterations using the improvement mentioned above (Lines 1 through 3). Then, we recompute the liveness information at the exit nodes for all functions until there is no change (Lines 4-14). If the liveness information at an exit node has changed we propagate this change according to Equation 3 to all nodes of this function (Lines 11 through 13). [8]

LiveOut and *MayUseOut* (resp. *LiveIn* and *MayUseIn*) need not be kept in separate location they can be merged into one, ie. all occurences of *LiveOut* (resp. *LiveIn*) can be replaced by *MayUseOut* (resp. *MayUseIn*) which will then contain the liveness information upon completion of the fixpoint iteration.

Since the last phase is usually the costliest of the three, this enhancement typically cuts down execution time drastically. (See the following section for experimental results). The drawback is that space usage almost doubles because both ByPass and MayUse information have to be kept around for each node.

[8] It would be sufficient to propagate this information to return nodes only.

B Characteristics of Benchmark Programs

B.1 C Programs

The C programs used were the eight SPEC-95 integer benchmarks: *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 below: the number of source lines reported were measured using the command `wc -l *.c`.

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

B.2 Scheme Programs

The Scheme benchmarks we used are taken from Gambit-C 2.7 distribution, available at `www.iro.umontreal.ca/~gambit`. They consist of the following programs: *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 [21], 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 source sizes given are for the "core program", i.e., without system-specific definitions, measured using the `wc` utility.

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

B.3 Prolog Programs

The Prolog benchmarks we used are available with the `wamcc` 2.21 distribution, which can be obtained from `ftp://ftp.inria.fr/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/wamcc/`. The eight programs we used were the following: *boyer*, a term-rewriting theorem prover; *chat*, a small database with a natural language front end; *nand*, a logic synthesis program; *poly10*, a program for symbolic manipulation of polynomials; *reducer*, a combinator graph reduction program; *sendmore*, a cryptarithmetic puzzle; *tak*, a small, heavily recursive, program; and *zebra*, a logical puzzle based on constraints. Their characteristics are as follows:

Program	Source lines	functions	blocks	instructions
boyer	377	2932	22833	85812
chat	1138	4640	29940	109274
nand	493	3389	25520	93438
poly10	86	2495	19758	74651
reducer	301	2939	22146	82103
sendmore	43	2433	19316	73020
tak	15	2336	18788	71541
zebra	36	2378	19092	72734