PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture*

Benjamin Schwarz Saumya Debray Gregory Andrews Matthew Legendre
Department of Computer Science
University of Arizona
Tucson, AZ 85721

{bschwarz, debray, greg, legendre}@cs.arizona.edu

ABSTRACT

This paper describes PLTO, a link-time instrumentation and optimization tool we have developed for the Intel IA-32 architecture. A number of characteristics of this architecture complicate the task of link-time optimization. These include a large number of op-codes and addressing modes, which increases the complexity of program analysis; variable-length instructions, which complicates disassembly of machine code; a paucity of available registers, which limits the extent of some optimizations; and a reliance on using memory locations for holding values and for parameter passing, which complicates program analysis and optimization. We describe how PLTO addresses these problems and the resulting performance improvements it is able to achieve.

1. INTRODUCTION

Results from a number of recent projects indicate that post-link-time code optimization of executable programs can yield significant improvements in performance, even for programs that have been subjected to extensive compile-time code optimization [6, 14, 16, 17]. Much of this work has been carried out in the context of processors with RISC architectures, which typically have a relatively small set of op-codes and addressing modes and a large number of general-purpose registers. By contrast, CISC architectures such as the widely-used Intel IA-32 have characteristics that make the task of link-time code modification and optimization considerably more challenging. Examples of these include a large number of op-codes and addressing modes, which increases the complexity of program analysis; variable-length instructions, which complicates disassembly of machine code; a paucity of available registers, which limits the extent of some optimizations; and a reliance on using memory locations for holding values and for parameter passing, which complicates program analysis and optimization.

This paper describes PLTO, a link-time instrumentation and optimization tool we have developed for the Intel IA-32 architecture. We describe how PLTO addresses the problems mentioned above and the resulting performance improvements it is able to achieve. We focus purely on static optimization; dynamic optimization, where code transformations are carried out at runtime, is not (yet) considered. Our goal is to carry out aggressive whole-program optimization, possibly including statically linked libraries [1]. For this reason, we cannot rule out the presence of hand-optimized assembly code that might not adhere to the application binary interface (ABI) for the system or that might not follow familiar conventions such as a single entry point per function. Consequently, we attempt to make as few assumptions as possible about the input executable code. For example, we do not assume that the code uses any specific code idioms that a particular compiler may generate or that it adheres to standard calling conventions. The main contribution of this paper is to describe the analyses and code transformations we use for link-time optimization where such optimization is non-trivial, both because of architectural features that make it harder to reason about the structure and behavior of executable files, and because of our desire to handle a wide variety of executables, including those containing statically linked libraries and hand-coded assembly routines.

*This work was supported in part by the National Science Foundation under grants ASC-9720738, CCR-0073394, EIA-0080123, and CCR-0113633.

2. SYSTEM OVERVIEW

The PLTO system consists of a front end for reading in executables, modules for code transformations, and a back end for emitting machine code. At present PLTO optimizes x86 executables, in the Executable and Linkable Format (ELF), under RedHat Linux. To enhance portability, we use the GNU Binary File Descriptor Library (`libbfd`); supporting other executable formats, e.g., the COFF (Common Object File Format) format used in Windows executables, would require making only minor changes to the front end. PLTO requires that the input executable contain relocation information (this currently implies statically linked executables, since the linker `ld` refuses to retain relocation information for non-statically-linked executables). Relocation information is used to distinguish between addresses and non-address constants; this distinction is essential because code addresses must be updated to reflect the results of optimizations, while the values of constants cannot be changed. Most linkers, if not all, can be instructed to leave relocation information inside the executable.

An executable file is processed in two stages to produce an optimized version of the program. In the first stage, described in Section 4, PLTO inserts instrumentation code into the binary to gather execution profiles. The user then executes the resulting instrumented binary on a training input set to produce representative weights for the edges in the interprocedural control flow graph (ICFG) of the program. In the second stage, PLTO uses these edge profiles to analyze and transform the program as follows:

ICFG Construction and Simplification. PLTO first disassembles all segments containing code, creates a single instruction stream, and constructs an interprocedural control flow graph for the entire program. It uses relocation information and knowledge about instruction semantics to guide these steps. PLTO then examines the initial ICFG to eliminate unreachable code; often up to 10% of the instructions in the original program can be removed. We do this optimization now (and again later) because it reduces the number of calling contexts for many functions and thus improves later analyses.

Code Optimization. PLTO then begins trying to simplify code by performing a round of constant propagation through registers and the runtime stack. Constant operands of arithmetic instructions are then replaced by immediate constants, while conditional branches with constant operands are simplified away by deleting the control flow edge that will not be taken. This reduces the number of register and memory references and usually results in more dead and unreachable code that can later be eliminated. Next, we inline functions that meet certain criteria (see Section 5.1 for details). This reduces the number of calling contexts and opens the way for a second round of constant propagation. This is followed by load/store forwarding to reduce the number of memory loads (see Section 5.2). Finally, PLTO does liveness analyses of registers, the runtime stack, and the program status word (PSW), as discussed in Section 3.5. This liveness information is used to eliminate instructions that store either to dead registers or to dead locations on the stack.

Repair and Layout. During the course of its optimizations, PLTO is likely to ruin any attempt the linker or compiler made to schedule instructions intelligently. In addition, no-ops inserted by the compiler for alignment purposes have been eliminated. Consequently, PLTO redoes instruction scheduling and alignment in order to improve decoding and execution efficiency. Basic blocks are also positioned to improve instruction cache utilization and reduce the frequency with which jumps are taken [15]. Alignment no-ops are inserted where appropriate to improve instruction-fetch hit rates. Finally, relocation information is used to update any addresses that are referenced in the executable, and the binary is written out.

Sections 3 and 5 describe the most interesting analyses and optimizations that PLTO performs.

3. ANALYSES

PLTO uses a number of different analyses to support the optimizations it carries out. This section describes the most important of these analyses.

3.1 Disassembly and Control Flow Analysis

Disassembly begins at the entry point of the program, as specified in the header of the executable file. Instructions are disassembled in sequence, as they are encountered (see [3]). One potential problem that arises is that of data or alignment bytes appearing in code segments, which can complicate the disassembling of machine code. As an example, in one of the string libraries on our system (RedHat Linux), we found that the programmer had inserted a NULL byte (`0x00`) at one point, presumably for alignment. Unfortunately, during disassembly this looks like a valid `add` instruction; because of the IA-32 architecture's variable-length instructions, this causes the rest of the disassembly from that point on to be erroneous. To deal with such problems, PLTO currently looks for targets of jumps and relocations that may point to the middle of another instruction. If there is a jump into the middle of an already disassembled instruction, then an error was made in constructing the instruction stream. In this case PLTO repairs the error and restarts the disassembly process from the target of the jump or relocation.

Once disassembly is complete, PLTO constructs an interprocedural control flow graph (ICFG) for the program. Several issues complicate the construction of the ICFG: indirect calls, indirect jumps through tables, and data appearing in segments, such as `.text`, that are typically reserved for instructions. Indirect calls through registers are modeled using a special pseudo-node in the ICFG, B_{HELL}. This node belongs to a special pseudo-function, F_{HELL}. Both B_{HELL} and F_{HELL} are used by many of the analyses and optimizations to represent worst-case scenarios. For example, B_{HELL} is assumed to use all registers, define all registers, and possibly write to all possible (writable) memory locations, possibly overwriting data in the stack frames of any callers. This ensures that all analyses performed by PLTO are conservative.

Indirect jumps through a table of addresses are often generated by a compiler for multi-way branches, such as `switch` constructs in C. PLTO attempts to recover the possible targets of these indirect jumps by tracing backwards through the instruction stream to find the size and base address of the table, using the bounds check to infer the table size [5]. Jumps whose targets cannot be resolved are modeled using the special node B_{HELL}.

3.2 Stack Analysis

One of our optimization goals for PLTO is to allow function inlining and the subsequent propagation of the actual parameters from the call site into the (cloned) body of the inlined function.[1] In particular, we want to be able to optimize away conditional branches in the inlined code based on constant arguments in the caller. Since function arguments are passed on the stack in the IA-32 architecture, this requires the ability to reason about the caller's stack frame and its relationship to the callee's stack frame. This is done via stack analysis. To the best of our knowledge, this analysis is novel: other comparable binary rewriting systems do not implement a similar analysis.

The idea can be illustrated by the following source code fragments:

```
int f(...)                void g(int x, int y)
{                         {
  ...                       ...
  g(123, 456);              if (y != 0) ...
}                         }
```

At the machine code level, the code for these functions resembles the following:

```
f: ...
   push $456      # push arg 2
   push $123      # push arg 1
   call g
   addl $8, %esp  # pop args
   ...

g: push %ebp            # save old frame ptr
   movl %esp, %ebp      # update frame ptr
   subl $32, %esp       # allocate stack frame
   ...
   movl 8(%ebp), %eax   # load y
   testl %eax, %eax     # y != 0 ?
   jne ...
   ...
   leave                # deallocate frame
   ret
```

We would like to inline `g()` into the body of `f()`, propagate the value of the (constant) second argument into the inlined body, and thereby eliminate the test and conditional branch corresponding to the statement '`if (y != 0) ...`,' as well as the `push` operation(s) at the call site for parameter passing. To do this, we have to be able to infer the following about the location ℓ written to by the instruction '`push $456`' in `f()`:

1. ℓ is the same as that referenced by the instruction '`movl 8(%ebp), %eax`' in `g()`, in order to propagate the value of the argument into the body of `g()`.

[1] We expect the compiler to have already carried out any compile-time inlining it is able to. Our focus is on inlining across module and library boundaries.

2. ℓ is not overwritten by any prior store operations within `g()`.

3. ℓ becomes dead once all references to it in the body of `g()` have been replaced by the constant value of the argument.

To make these inferences, we have to be able to determine the position of the location ℓ addressed by the instruction '`push $456`' relative to both the "old" frame pointer in `f()` as well as the "new" frame pointer in `g()` and to reason about the liveness of specific memory locations within the stack frame of `f()` after the call to `g()` has been inlined. We do this using a stack analysis that allows us to model the stack frame of a function as an array of words; subsequent analyses then reason about the contents, liveness, etc., of locations within this array.

To determine the size of a function's stack frame, we examine the basic blocks of the function and compute the maximum value of the difference between the frame pointer register `%ebp` and the top-of-stack pointer register `%esp`. The essential intuition is to keep track of operations that update the stack and frame pointers. When we come to a function call, we cannot in general assume that the stack will have the same height on return from the callee as it did on entry to it. Hence, to determine the size of the stack frame when control returns from the callee, we have to take into account the behavior of the callee. To this end, we first carry out a well-behavedness analysis to identify functions that leave the stack at the same height as it had when the function was entered.

A function f is said to be *well-behaved* if there is no net change in the height of the runtime stack due to the execution of f, for all possible execution paths through f. Well-behavedness analysis is done in two phases. First, we mark as well-behaved all those functions that (*i*) push the frame pointer `%ebp` on entry; and (*ii*) execute the `leave` instruction immediately before returning to the caller. The effect of this combination is to restore the stack and frame pointers to the values they had just before entry to f. Second, as described below, we propagate information about changes in the height of the runtime stack due to the execution of each basic block in the program and use this to identify functions where the runtime stack is at the same height on return as it was on entry.

Given information about well-behavedness of functions, we analyze each function to determine the (maximum) size of its stack frame (including the space for actual parameters, which is shared with the caller). This is done as one would expect. We first determine, for each basic block in the function, the change in the stack size due to the execution of that block. This is then propagated through the control flow graph of the function. If a basic block has more than one predecessor, and different incoming edges have different values for the change in stack size along that execution path, then the stack frame height at the entry to that block is set to *unknown*. This procedure is iterated, in a manner very similar to that for constant propagation, until the stack height at the entry to, and exit from, each basic block has stabilized. This allows us to determine the change in stack size at the entry to each block relative to that at entry to the function, and thence the maximum size of its stack frame.

3.3 Use-Depth and Kill-Depth Analysis

The relatively small number of compiler-visible general purpose registers in the IA-32 architecture often causes values to be placed in (or spilled to) a function's stack frame. In the absence of any other information, program analyses must make worst-case assumptions about the effects of function calls on such values. For example, constant propagation (Section 3.4) must assume that a function call can destroy all such values, because a function might write to any memory location, while stack liveness analysis (Section 3.5) must assume that stack locations are live because they may be accessed by a function call. Such worst-case assumptions can affect the precision of our analyses quite significantly. To address this, we use *use depth* and *kill depth* analyses to estimate the effect of function calls on the runtime stack.

The *use depth* of a function is either a non-negative integer or the value ∞; it represents an upper bound on the depth in the stack, relative to the top of stack when the function is called, from which the function may read a value. The psuedo-function F_{HELL}, which is used to model indirect function calls, is assumed to have a use depth of ∞. The use depth of the other functions in the program are computed in two phases:

1. [*Local analysis.*] The instructions in each function are examined to determine from how far down the stack they may load a value. Indirect loads are assumed to be able to load from any location, and result in a use depth of ∞. This forms an initial approximation to the use depth of each function.

2. [*Iterative propagation.*] Use depth information is iteratively propagated along the call graph of the program from callee to caller. In a given iteration, consider a function f whose use depth is currently set to m. Suppose that f calls functions $g_1,\ldots,g_k$ from call sites $C_1,\ldots,C_k$ respectively, and that the use depths of the functions $g_1,\ldots,g_k$ are set to $n_1,\ldots,n_k$ respectively. Moreover, suppose that the height of f's stack frame, determined from the stack analysis described in Section 3.2, at the call site C_i is p_i, $1 \le i \le k$. Let d_i be the maximum depth in the stack that can be accessed by the call to g_i, $1 \le i \le k$, relative to the stack top at the time f was called. We compute d_i as follows:

   – If $p_i = unknown$, we do not know how large f's stack frame is at that call site. In this case, the deepest location in the stack that can be accessed by a load operation in the callee g_i cannot be deeper than n_i relative to the top of the stack at the call site C_i in f. It follows that this location cannot be deeper that n_i relative to the top of the stack when f was called (since f's stack frame cannot have negative size). So we set $d_i = n_i$.

   – If $p_i \neq unknown$, we have two possibilities. If $p_i \ge n_i$, then load operations within the callee cannot access any stack location outside f's stack frame. On the other hand, if $p_i < n_i$ then the deepest location accessed by a load operation within g_i, relative to the stack top at the point when f was called, is at most $n_i - p_i$. In this case, therefore, we have $d_i = max(0, n_i - p_i)$.

   The use depth of f is then updated to $max(m, d_1, \ldots, d_k)$. This is repeated until a fixpoint is reached and the use depth of every function stabilizes.

The *kill depth* of a function is analogous to that of use depth: it is either a non-negative integer or the value ∞, and represents an upper bound on the depth in the stack, relative to the top of the stack when the function is called, to which that function may write a value. The computation of kill depths is exactly analogous to that of use depths.

3.4 Constant Propagation

Our experience with the *alto* system [14] showed that constant propagation—which at link time propagates addresses and constant arguments across function and module boundaries—can be an important source of performance improvements resulting from link-time optimization. To carry out constant propagation on machine code, however, it is necessary to specify, in some form, the semantics of the instructions of the underlying architecture. The

IA-32 instruction set contains a large number (over 300) of different instruction classes,[2] making this a tedious and time-consuming proposition. It turns out, however, that a relatively small number of different instruction classes account for the vast majority of all instructions actually encountered in executable files in practice. PLTO therefore uses semantic knowledge about only a small subset of all possible IA-32 instruction classes, based on an examination of the static and dynamic distribution of instructions in the code for all of the SPEC-95 benchmark programs; the remaining instruction classes are treated conservatively and their results taken to be unknown. This allows us to handle the vast majority of instructions encountered in practice without requiring an exorbitant implementation effort. This results in some 31 different instruction classes that are considered by the constant propagator; however, with knowledge about the semantics of these 31 instruction classes it is able to process all of our benchmarks without significant loss of information.

The constant propagation algorithm PLTO uses is straightforward conditional constant propagation [19]; processor status word (PSW) bits are treated as one-bit registers. The notion of kill depth, discussed in Section 3.3, is used to estimate the effect of function calls on the runtime stack. This allows us to limit the amount of stack whose contents have to be invalidated during constant propagation when a function call is encountered. For example, if the callee has a kill depth of 8, then only the top 8 bytes of the stack can be affected, and the values of deeper locations need not be discarded by the constant propagator.

3.5 Liveness Analysis

The IA-32 has eight general purpose registers—six are generally available to a compiler; two are reserved for the stack and frame pointer. This small set of usable registers results in the compiler generating code that operates directly on memory, rather than explicitly loading, modifying and storing values. For this reason, it is insufficient to restrict liveness analyses to registers. To this end, PLTO models and analyzes the runtime stack in addition to the set of registers. For liveness analysis purposes, moreover, PSW bits are treated as one-bit registers.

PLTO uses a standard context sensitive approach to computing register liveness information [9, 12]. Muth found that on a RISC architecture, the Compaq Alpha, the use of a context sensitive liveness analysis increased the number of dead registers available per basic block in the SPECint-95 benchmark suite, on the average, from about 3.0 to 5.2, compared to a context insensitive approach—an increase of about 70% [12]. By contrast, in PLTO a context-sensitive analysis finds an average of 1.8 dead registers per basic block in the SPECint-95 suite, up from 1.67 dead registers per block obtained with a context-insensitive analysis: an improvement of only 7.8%. This incremental improvement increase is much less than Muth's using `alto`, but this is not entirely surprising given that there are so few registers to work with.

It turns out that when a function stores callee-saved registers to the stack (via `push` instructions upon entry), the use of these registers propagates back to the caller. This results in the callee-saved registers being live in blocks prior to the function call. A straightforward liveness analysis does not recognize that callee-saved registers are restored (via `pop` instructions on function exit) prior to returning control to the caller. One would like to know that although these registers are used, their values are not examined except for the save on entry and restore on exit. To solve this problem, PLTO performs a local analysis for each function to determine if a callee-saved register is used. It is used if the contents of the register are used (ignoring the action of saving the register) before being defined, if the location to which it was stored on the stack is used, or if the register is restored and then used. The results from the local analysis are then iteratively propagated along the call graph until a fixpoint is reached. Indirect loads through registers are treated conservatively—we assume that they could come from anywhere, including slots on the stack. Our context sensitive analysis is tuned to subtract the registers that PLTO finds as being callee-saved from the set of registers used by that function.

PLTO's liveness analysis of the runtime stack is similar to that of registers, with 4-byte slots on the stack taking their place. Indirect loads from registers are treated conservatively—they may load from anywhere, including the stack. Indirect stores through registers are also treated conservatively; in most cases we can safely say that they define nothing, i.e., any live stack slot remains live after an indirect store. The analysis is interprocedural, and uses the use depth of a function, discussed in Section 3.3, to estimate the may-use behavior for function calls.

[2]An instruction class corresponds, roughly, to an operation, e.g., *add* or *load*; within a particular instruction class there may be several different op-codes, specifying different kinds or sizes of operands.

4. INSTRUMENTATION

Many of the optimizations carried out by PLTO rely on low-level profile information. Since Red Hat Linux does not currently come with instrumentation tools for gathering such low-level profiles, we have built this functionality into PLTO.

A command-line option can be used to instruct PLTO to add instrumentation code for gathering edge profiles. This causes the insertion of profiling blocks along each edge in the interprocedural control flow graph of the original program. Each such block contains code to update a (64-bit) counter that records the number of times the corresponding edge has been traversed. The data section of the program is expanded to accommodate these counters. PLTO also adds a procedure to the code that writes the edge profiles to a file, and inserts a call to this procedure just before the program exit point. The profiles gathered by executing this instrumented binary on representative training input are subsequently read in by PLTO and used to guide optimization decisions.

PLTO also supports the gathering and use of value profiles [13]. A value profile for a variable x at a program point p is a partial probability distribution of the values taken on by x at p. If this distribution is found to be sufficiently skewed to a particular value a, we can generate specialized code when the value of x is a at program point p. In particular, when control reaches p we test whether x has value a and, if so, branch to the specialized code. Whether such specialization is worthwhile is determined using a low-level cost-benefit analysis [13]. We use goal-directed value profiling to reduce the cost of gathering value profiles [18]. The results described in this paper, however, did not rely on value profiles.

5. OPTIMIZATIONS

PLTO performs numerous code optimizations, as summarized in Section 2. Below we describe some of the more important of these optimizations.

5.1 Inlining

Inlining is a well-known optimization where a call to a function f is replaced by a copy of the body of the callee f [2, 8]. When inlining is carried out on executable programs after linking, the goal is to inline across module and library boundaries. The main benefits of inlining—apart from locality effects arising from bringing the code for the caller and callee closer together in memory—are three-fold. The first is to eliminate the function call/return overhead. Usually, inlining a function call gets rid of 2–5 instructions: the call and return instructions, and—if the callee is not a leaf function—instructions to allocate a stack frame on entry and deallocate it on return. The second benefit is to reduce or eliminate the overhead of argument passing by eliminating `push` operations at the call site and memory loads within the callee. The third benefit is to exploit callsite-specific information in the callee. For example, constant argument values can be exploited to eliminate conditional branches within the callee that use those values; and memory 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. The main potential disadvantage to inlining is code growth; doing inlining without attending to its effects on cache behavior can have a significant negative effect on program performance (this is dramatically evident in, for example, the SPECint-95 benchmark program *go*).

The criteria used for inlining within PLTO are as follows. If a function has a single call site, or if its body contains at most 5 instructions, it is always inlined, since this cannot result in any code growth. Otherwise, a function f is inlined into a call site C provided that each of the following hold:

1. The inlining can be expected to yield a reasonable performance improvement. We consider two possibilities. The first is that C is a "hot" call site, i.e., it is executed sufficiently often that the benefit of eliminating the overheads of parameter passing and control transfer is likely to be significant. The second is that C passes constant arguments to f, and f uses its arguments or passes them to other functions that use them, such that optimizing f's code using information about constant arguments at C can yield performance benefits.

2. Inlining f into C will not cause excessive code growth or interfere with other optimizations within PLTO. Under this criterion we check, for example, that C is not a recursive call and that the inlining will not adversely affect stack analysis (Section 3.2). The cache model we use to estimate the effect of inlining on the instruction cache utilization of the program can be thought of as a simplified version of McFarling's [11].

We are currently investigating techniques to improve our estimation of the benefits of knowing constant arguments to a function.

5.2 Load/Store Forwarding

Load/store forwarding is an optimization that attempts to eliminate unnecessary load operations from memory. The idea is to find a pair of instructions I and J such that: (i) I is a load instruction $r_0 \leftarrow \mathtt{load}(\ell)$; (ii) J loads a register r_1 from, or stores r_1 to, the location ℓ; (iii) J dominates I; and (iv) we can guarantee that the contents of memory location ℓ do not change between J and I. In this case, provided that some additional conditions are satisfied, we can replace the load operation I by a register-to-register move from r_1 to r_0 (or, if $r_0 = r_1$, simply eliminate I). The optimization can be thought of as a special case of common subexpression elimination.

The implementation of load/store forwarding in PLTO works as follows. We consider situations of the form

$$
\begin{array}{lll}
\text{/* Block } B_1 \text{ */} & J : r_1 \leftarrow \mathtt{load}(\ell) \quad \text{or} & \ell \leftarrow \mathtt{store}(r_1) \\
 & \quad\ldots & \\
\text{/* Block } B_n \text{ */} & I : r_0 \leftarrow \mathtt{load}(\ell) &
\end{array}
$$

where the sequence of basic blocks $B_1 \ldots B_n$ forms an extended basic block. In other words, if we trace back from B_n up to B_1, each block along the way (except possibly B_1) has exactly one predecessor. To ensure that the contents of location ℓ are not modified between instructions J and I, we examine each store instruction between these instructions to see whether the target location may overlap ℓ. This is done using the following memory disambiguation rules:

1. an indirect memory reference can overlap any other memory reference;

2. an absolute memory location does not overlap a stack location; and

3. two memory references are non-overlapping if they use the same base and index registers, and the same scale factor, but have different displacements.

If $r_0 \neq r_1$, we have to insert a register move instruction to copy r_1 into r_0; this is inserted as late in the extended basic block $B_1 \ldots B_n$ as possible, while ensuring that at the insertion point r_1 still contains the value loaded from location ℓ and that register r_0 is not live at that point. As a pragmatic measure we allow only one such copy operation to be inserted per load operation being eliminated; this has the effect of allowing at most one of the registers r_0 or r_1 (none, if $r_0 = r_1$) to be modified between instructions J and I. We also ensure that the execution frequency of the program point where this copy operation is inserted is low enough, relative to that of instruction I, so that its runtime cost does not exceed the benefit of eliminating instruction I.

5.3 CMOV Introduction

The Intel P6 processor incorporates some instructions not found in older Pentium processors, including *conditional move* (CMOV) instructions for integer and floating point operands. The effect of a CMOV instruction is to copy its source operand to its destination if the condition specified holds. If PLTO encounters a branch instruction whose only effect is to conditionally jump over a move operation, it optimizes this to replace the conditional branch and move operation with an appropriate CMOV instruction. The effect of this optimization is to eliminate potentially unpredictable branches. Among the SPECint-95 benchmarks, the *m88ksim* program experiences a significant performance improvement due to this optimization.

6. EXPERIMENTAL RESULTS

We evaluated the performance improvements obtained from PLTO using the SPECint-95 benchmark suite. Our experiments were run on an otherwise unloaded 600 MHz Pentium III system with 128 MB of main memory running RedHat Linux 6.2. The programs were compiled with *gcc* version *egcs-2.91.66* at optimization level `-O3`. The programs were profiled using the SPEC training inputs, optimized by PLTO using these profiles, and then timed on the SPEC reference inputs.

The execution times for our programs, before and after optimization, are shown in Table 1. Each number was obtained by running the corresponding executable 7 times, discarding the highest and lowest times, and computing the arithmetic mean of the remaining 5 times. The last line of Table 1 shows the average speed improvement obtained, computed as the geometric mean of the speed ratios for the individual programs. The *m88ksim* benchmark obtains the greatest performance improvement of a little over 15%, while the *gcc*, *vortex*, and *perl* benchmarks obtain improvements of 8.6%, 7.4%, and 6.4% respectively. On average, we see an improvement of 6.2%.

We also measured the effects of PLTO on the low-level execution behavior of programs. These were measured using the Rabbit performance counters library [10], with each number obtained as the average of three runs on an otherwise unloaded machine. Some of the results are shown in Figure 1:

Memory Operations : Improvements in the number of memory operations are shown in Figure 1(a). For several programs, PLTO is able to achieve significant reductions in the number of memory operations, due primarily to the effects of inlining and load/store forwarding, with *li* achieving a reduction of over 9% and *vortex* a reduction of over 7%. On average, the number of memory operations is reduced by about 5.6%.

Program	*Execution Time* (secs)		*Ratio*
	Original (T_0)	Optimized (T_1)	T_1/T_0
compress	116.96	113.89	0.974
gcc	79.89	72.98	0.914
go	123.12	121.05	0.983
ijpeg	128.56	127.45	0.991
li	100.45	94.79	0.944
m88ksim	115.27	97.63	0.847
perl	77.30	72.38	0.936
vortex	132.75	122.95	0.926
GEOMETRIC MEAN:			0.938

Table 1: Speed improvements due to PLTO

Taken Branches : The number of taken branches, shown in Figure 1(b), is reduced dramatically, by about 64.5% on the average. This is due primarily to code layout. The *vortex* program experiences the greatest improvement, with a reduction of close to 74%.

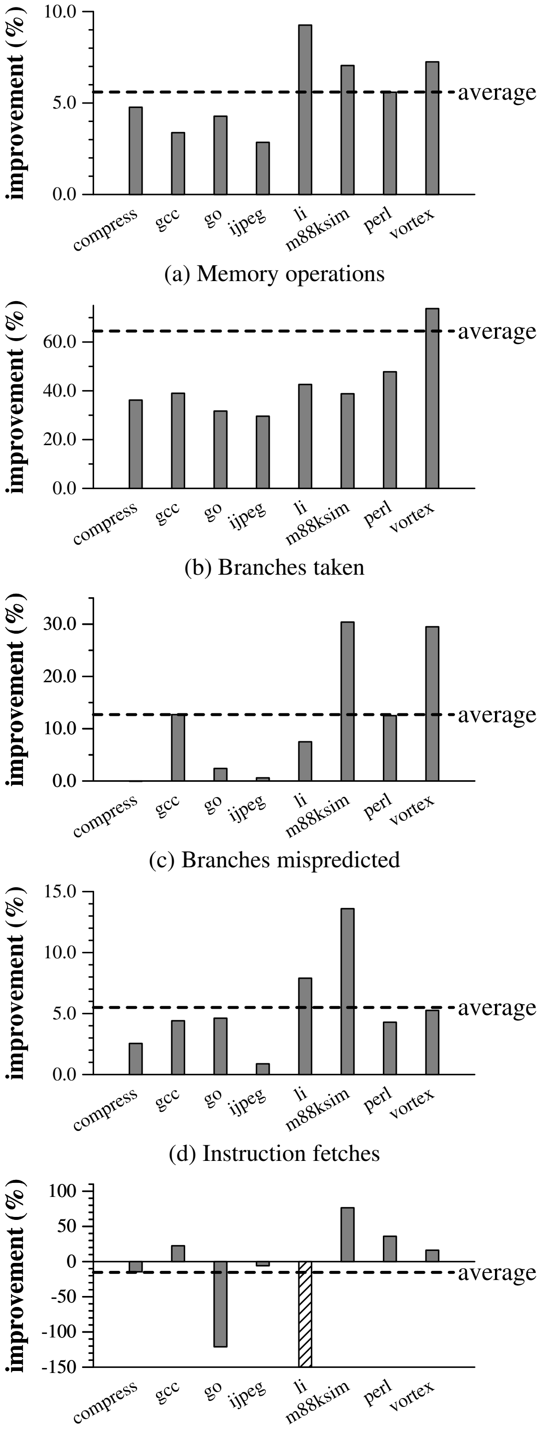
Mispredicted Branches : The improvement in the number of mispredicted branches is shown in Figure 1(c). Most of the benchmarks experience significant improvements in the number of branch mispredictions, with *m88ksim* and *vortex* having improvements of around 30%, while *gcc* and *perl* experience smaller—but still significant—improvements of over 12%. Overall, the programs experience a 12.7% improvement in the number of mispredicted branches.

Instruction Fetches : The number of instruction fetches decreases for all programs, as shown in Figure 1(d). The largest improvements are seen for *m88ksim*, with a 13.6% reduction in the number of instructions fetched, and *m88ksim*, with a 7.9% reduction. The average reduction in the number of instructions fetched is 5.5%.

Instruction Cache Misses : Changes in the number of instruction cache misses are shown in Figure 1(e). Unfortunately, these numbers are significantly worse than we would like: several programs experience significant increases in the number of i-cache misses, with *go* incurring a 121% increase, and *li* incurring a 12-fold(!) increase. On the other hand, some programs exhibit significant reductions in the number of i-cache misses, with *m88ksim* and *perl* experiencing improvements of 76.5% and 36% respectively. Overall, the programs suffer a 15.3% increase in the number of i-cache misses.

We are currently investigating the reasons for these increases, and anticipate further improvements in the overall speedups PLTO is able to achieve when this problem has been addressed.

An interesting aspect of our experiences with PLTO was in the interaction between code layout and the Branch Target Buffer (BTB) management algorithm. Our experiments indicated that profile-guided code layout using the Pettis-Hansen algorithm [15] led to a huge increase (by roughly two orders of magnitude) in the number of BTB misses. This initially raised the possibility that profile-guided code layout may have been inadvertantly introducing significant runtime overheads arising from these BTB misses. We eventually tracked the problem down to the fact that on the Pentium III processor, a conditional branch does not enter the BTB until it has been taken. Recall that the Pettis-Hansen code layout algorithm is set up so that conditional branches fall through—i.e., are not taken—wherever possible. Each such branch can, therefore, give rise to a BTB miss each time it is encountered until eventually it



See text for remarks on i-cache misses for *li*

(e) I-cache misses

Figure 1: Effects of PLTO **on Low Level Program Behavior**

enters the BTB (if it does). Fortunately, it turns out that in this particular case the BTB miss does not incur any performance penalty. However—apart from the fact that this behavior was initially quite disconcerting—it also means that the "unpenalized" BTB misses arising from profile-guided code layout can mask BTB misses elsewhere in the program that do incur a performance penalty, thereby making it harder to identify and rectify the latter.

7. RELATED WORK

Binary rewriting and link-time code optimization have been considered by a number of researchers. The work most closely related to ours are those focusing on static optimization of executable binaries. Most of the work in this context, including Spike [6], *alto* [14], and OM [17], has focused on RISC architectures such as the Compaq Alpha. Compared to such RISC architectures, the Intel IA-32 architecture targeted by PLTO offers very different challenges to link-time optimization.

The Etch system, like PLTO, is aimed at modifying IA-32 executables [16]. Its primary goal appears to be instrumentation rather than optimization. It implements a relatively small set of optimizations; the only one specifically mentioned by the authors is profile-guided code layout. Other systems aimed at instrumentation and analysis of IA-32 programs include NT-Atom and HiProf [7].

Also related is UQBT, a binary translation system that is able to process IA-32 executables [4]. The primary focus of this work is on the translation of executable programs across platforms rather than aggressive optimization within the context of a particular platform. This difference in focus between UQBT and PLTO leads to significant differences in their internal architectures as well as the assumptions they make about input binaries; in particular, UQBT does not attempt to optimize code that does not conform to high-level specifications, such as hand-optimized assembly code within libraries.

8. CONCLUSIONS

We have described PLTO, a link-time optimizer for the Intel IA-32 architecture. The goal of our system is to optimize executables that have been aggressively optimized by the compiler, and which may contain statically linked libraries and/or hand-optimized assembly code that need not adhere to higher level conventions. Our system implements several analyses that are, to the best of our knowledge, new: examples include the stack analysis, use depth and kill depth analyses, etc. We currently obtain reasonable performance improvements on medium-sized programs, as exemplified by the SPECint-95 benchmark suite, which experience a speed improvement of a little over 6% on the average.

9. REFERENCES

[1] G. R. Andrews, S. K. Debray, B. W. Schwarz, and M. P. Legendre, "Using Link-Time Optimization to Improve the Performance of MPI Programs", manuscript, Dept. of Computer Science, The University of Arizona, Tucson, April 2001.

[2] A. Ayers, R. Schooler, and R. Gottlieb, "Aggressive Inlining", *Proc. SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997, pp. 134–145.

[3] C. Cifuentes and K. J. Gough, "Decompilation of Binary Programs", *Software—Practice and Experience*, Vol. 25, No. 7, July 1995, pp. 811–829.

[4] C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon, and T. Washington, "Preliminary Experiences with the UQBT Binary Translation Framework", *Proc. Workshop on Binary Translation*, Oct. 1999, pp. 12–22.

[5] C. Cifuentes and M. Van Emmerik. "Recovery of Jump Table Case Statements from Binary Code," *Science of Computer Programming*, Vol. 40, Issue 2-3, July 2001, pp. 171–188.

[6] 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.

[7] Compaq Corp., *Release Notes for NT-Atom 1.53*, April 1999. `www.support.compaq.com/amt/perftools/153relnote.txt`.

[8] J. W. Davidson and A. M. Holler, "Subprogram Inlining: A Study of its Effects on Program Execution Time", *IEEE Transactions on Software Engineering* vol. 18 no. 2, Feb. 1992, pp. 89–102.

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

[10] D. Heller, "Rabbit—A Performance Counters Library for Intel/AMD Processors and Linux", Scalable Computing Laboratory, Ames Laboratory, U.S. D.O.E., Iowa State University. `http://www.scl.ameslab.gov/Projects/Rabbit`

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

[12] R. Muth, "Register Liveness Analysis of Executable Code", Manuscript, Dept. of Computer Science, The University of Arizona, Nov. 1997. Available at `www.cs.arizona.edu/alto/papers/liveness.ps`.

[13] R. Muth, S. Watterson, and S. K. Debray, "Code Specialization based on Value Profiles", *Proc. 7th. International Static Analysis Symposium* (SAS 2000), June 2000. Springer LNCS vol. 1824, pp. 340–359.

[14] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, "`alto` : A Link-Time Optimizer for the Compaq Alpha", *Software Practice and Experience*, Vol. 31, January 2001, pp. 67–101.

[15] 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.

[16] 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, August 1997, pp. 1–8.

[17] A. Srivastava and D. W. Wall, "A Practical System for Intermodule Code Optimization at Link-Time", *Journal of Programming Languages*, March 1993, pp. 1–18.

[18] S. A. Watterson and S. K. Debray, Goal-Directed Value Profiling. *Proc. 2001 International Conference on Compiler Construction* (CC 2001), April 2001, pp. 319–333.

[19] M. N. Wegman and F. K. Zadeck, "Constant Propagation with Conditional Branches", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 2, April 1991, pp. 181–210.