Post Link-Time Optimization on the Intel IA-32 Architecture*

Benjamin William Schwarz

Abstract

Post link-time optimization of executables has been investigated by several projects in recent years. These optimization systems have targeted RISC architectures like the Compaq Alpha, and have shown that there is considerable room for improvement in compiler-generated code. Classical compiler optimizations like constant propagation, function inlining, and dead code elimination have been shown to be relatively effective when applied at link-time. In addition, other optimizations—such as value specialization, load/store forwarding, and code layout—that are not typically carried out at compile-time can also be used effectively. Unfortunately, many of the analyses introduced by other systems are insufficient when carried out on a CISC machine (e.g. the x86). We describe PLTO, a link-time optimizer for the Intel IA-32 architecture, that addresses the inherent difficulties in static analysis of binaries compiled for a CISC architecture. Many of the challenging issues stem from intrinsic characteristics of the architecture, such as the small register set which lends way to a heavy reliance on using the runtime stack. This paper discusses many analyses and optimizations used by PLTO, and we show the performance gains our system is able to achieve over compiler-generated, heavily-optimized executables.

*This research was supported by the National Science Foundation through grants ACR-9720738 and CCR-0113633.

Contents

1 Introduction

Modern compilers are good at code optimization; however, the analyses and optimizations they carry out are generally restricted to individual functions. Even those which perform interprocedural optimizations do not have access to pre-compiled library routines. As a result, there is considerable room for improvement in the performance of code generated by conventional compilers, even with a high degree of compile-time optimizations. One solution is to perform an additional phase of optimizations after the object modules have been linked. At this stage, the whole program (now available as machine code) and all statically-linked library code are available for inspection and modification. Although much semantic information is lost during compilation, there are still many opportunities to generate more efficient code than what the compiler has produced.

One specific set of applications which has great potential to benefit from link-time optimization is scientific distributed-memory applications. It is common for scientific applications to run on large Beowulf clusters (Pentiums running Linux). These machines are cheap and easy to set up for distributed-memory computations, making them a favorite among people who run simulations. Often these programs make heavy use of pre-compiled libraries, such as MPI (the Message Passing Interface), to handle the delicate details of communication across the network. These message-passing libraries are written with the goal of being applicable to a wide range of programs, ranging from gravitational N-body simulations to adaptive grid computations. As such, they are very general in their functionality and implementation, and allow for a good deal of flexibility. This flexibility, however, often comes at the cost of inefficient code. Unfortunately, traditional link-time optimization systems have targeted RISC (Reduced Instruction Set Computer) architectures like the Compaq (formerly DEC) Alpha and the Sun SPARC. Little work has been done for CISC (Complex Instruction Set Computer) machines such as the x86 (e.g., Pentium II, Pentium III, Pentium Pro) in the post link-time optimization arena.

Our contribution is a post link-time optimizer that operates on x86 executables compiled in the Executable and Linkable Format (ELF32), which is the binary file format used by the Linux operating system. Many of the analyses and optimizations carried out by our system are designed to deal with inherent difficulties of a CISC architecture. The dearth of registers and the strong reliance—by both compilers and the ISA (Instruction Set Architecture)—on using memory lend way to analyses that are very different from those designed for RISC machines. In particular, one cannot expect to see much improvement in performance by carrying out analyses across only registers. On a RISC architecture, however, it is perfectly feasible to ignore memory since the large register set enables most important operands to be stored in registers. Other characteristics of a CISC architecture—such as variable length instructions—complicate disassembly and re-assembly of the machine code instructions. The problem is made more challenging by the presence of data or jump tables embedded in sections of the executable that are typically reserved for code. Much of the remainder of this paper discusses our approach to overcoming the difficulties associated with the x86.

The rest of the paper is organized as follows: Section 2 describes PLTO, our Pentium Link-Time Optimization system and discusses some requirements and assumptions made about the input. Section 3 examines the preprocessing required to deal with abnormalities in ELF binaries, and then explains the novel approach PLTO uses for disassembling machine code. Analyses used by PLTO are detailed in Section 4. Section 5 explains how the analyses are used to guide optimizations throughout the system. We take a look at what optimizations result in the most performance improvements, and conclude with performance results in Section 6. Section 7 contains information about work related to binary rewriting, link-time optimization. Section 8 discusses future work and open problems, and our conclusions are presented in Section 9.

2 PLTO : A Pentium Link-Time Optimizer

PLTO (Pentium Link-Time Optimizer) is a post link-time optimization system designed to modify IA-32 (i.e., x86) executables compiled under the Linux operating system. PLTO is a binary-rewriting tool—both its input and output are machine code. It is closely related to `alto`, a link-time optimizer for the Compaq Alpha [21]. The goal is to carry out aggressive whole-program optimization while still producing code that is functionally equivalent to the original. As such, all transformations performed by PLTO are conservative to ensure that correctness is retained.

Like many highly-optimizing compilers and link-time optimizers, PLTO gathers execution profiles from training input before carrying out any optimizations. Currently, edge profiles are gathered to discover how many times control flows along each edge in the interprocedural control flow graph (ICFG). Basic block and instruction weights are derived from the edge profiles. Section 8 discusses some future directions in profiling of multiprocessor applications and context-sensitive profiling.

2.1 Requirements and Assumptions

During the course of optimization and instrumentation, existing instructions inside the executable often change locations. In addition, new instructions are inserted, and some original ones may be removed. PLTO is a binary rewriting tool, so both its input and output must be executable programs. The executables need to work without being run through a linker; as a result, PLTO must be able to resolve relocations and insure that all program addresses are patched correctly before the binary is rewritten. A consequence of this requirement is that PLTO needs to know which byte-sequences in the program are addresses (and thus relocatable), and which are simply encodings of an instruction. The requirement is the same imposed by a linker, and exists for the same reason.

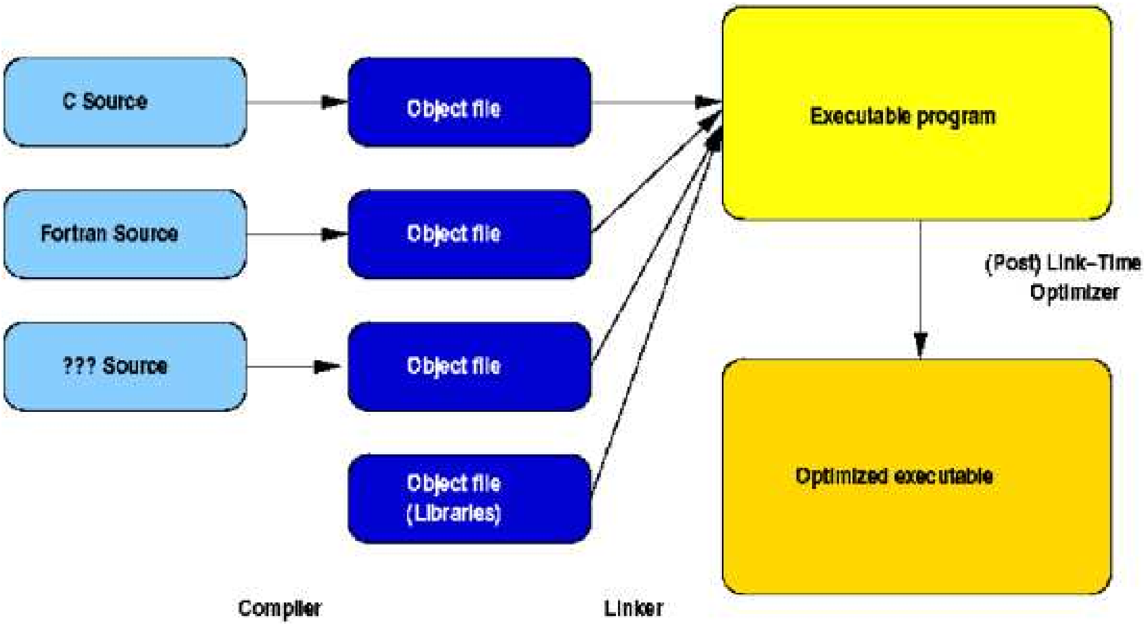

Figure 1: Compilation Model with Post Link-Time Optimization

We prepare binaries and run PLTO on RedHat Linux 7.2 workstations, although there is nothing intrinsic in the operating system that would prevent it from being used on a different distribution. The only machine-specific requirement is that a certain library (discussed below) be available. A typical invocation of the compiler is *gcc -O3 -Wl,-r program.c*. The *-Wl,-r* flag instructs the compiler to pass the *-r* flag to the linker. The *-r* flag tells the linker to not discard the relocation and symbol information it uses. The output from this command is an object file which serves as input to PLTO.[1] PLTO does not require a symbol table, although having one makes for easier debugging and allows the user to get a better handle on what is going on. The flags described above also result in the linker retaining symbol information. We do not feel this imposes an excessive burden upon the user. It seems likely that a person concerned enough about performance to use a link-time optimizer would be willing to invoke the compiler with the additional flags, most likely even at the expense of code growth resulting from libraries being statically-linked. Figure 1 shows a high level overview of how post link-time optimization fits into the standard compilation model.

To enhance portability we use the GNU Binary File Descriptor Library (libbfd) for reading and writing ELF executables. The library supports many common file formats such as COFF (Common Object File Format), the format used in Windows executables. In addition, the BFD library handles hairy details, such as updating the section header table to reflect any changes. Our disassembler is based loosely on the GNU disassembler for x86 executables that is available in the *binutils* software distribution.

PLTO makes certain assumptions about the input executable. We assume that all addresses that are relocatable are marked as such. If the linker does not correctly retain the relocation information, then addresses may not be updated appropriately, or we may mistakingly update bit-sequences that are not addresses. Secondly, code sequences

[1]The object file is not executable due to intricacies with the GNU C Compiler and the native linker, which refuses to patch relocations and also to retain them. Upon reading the binary, PLTO stores internal representations of the relocations, then invokes the native linker to make the object file an executable.

which perform arithmetic on addresses that point into the *.text* section of the executable can result in PLTO producing incorrect programs. An example (in generic assembly code) follows:

```
  Address          Instruction            Comment
-------------   -------------------     ----------------
   ...                 ...
  0x8048100       mov r1, 0x80481fc     Load an address into r1
  0x8048104       add r1, r1, 4         Add 4 bytes to the address
  0x8048108       call *(r1)            Function call to the address
   ...                 ...               contained in r1
  0x80481fc            ...
  0x8048200            ...              Start of some function
   ...                 ...
```

In this piece of code an assumption is made about the address of the function being called. Specifically, the code assumes that the function falls 4 bytes after the address *0x80481fc* in the executable. If the two instructions at the addresses *0x80481fc* and *0x8048200* are separated—even by something as simple as adding a NOP—the functional behavior of the program changes and becomes incorrect. Fortunately, such code fragments rarely exist in practice. On the IA-32 we have seen several instances of code that performs arithmetic on addresses in the *.text* segment. These examples arise in code that has been compiled to be *position-independent*, which is explained in detail in section 3.3.1. In position-independent code, jump tables are often embedded inside the *.text* section of the function containing the jump through the table. An instruction sequence is used to load the address of an instruction inside the function, and later a displacement is added to it. The resulting address is the start of the jump table. PLTO tries to detect situations where *.text* address arithmetic is performed and treat them accordingly; this usually involves marking the functions as being problematic, and not disassembling or carrying out optimizations on them. In all of the SPECint95 and SPECint2000 benchmarks there are only a handful of such occurrences, so the impact on disassembly is small.

3 Disassembly and Control Flow Analysis

3.1 Phase Flowchart

Figure 2 depicts the important pre-optimization stages in PLTO. They are explained in further detail in the following sections.

3.2 Pre-processing for ELF Executables

ELF executables can have multiple sections that contain code. In particular, most linkers include the standard sections *.init, .plt, and .fini*, which are used for program initialization, procedure linking, and for program termination. PLTO contains a preprocessing pass in which it combines all sections containing instructions into a single *.text* section—we term the process a "normalization" pass. This normalization stage is performed before anything else is done, and alleviates some of the work needed to be done by optimizations and analyses to handle intra-section anomalies. The process is fairly straightforward since virtual addresses are preserved through the addition of NOP instructions. Minor bookkeeping is needed to adjust relocations, which are typically provided as (section, offset) pairs. When a section is merged with another, any relocations residing in the merged section are rewritten and associated with the section into which they are merged.

3.3 Disassembly

Precise disassembly is a fundamental requirement for any system which aims to statically analyzes binary code. Unfortunately, correct disassembly is a hard problem, especially with the variable-length instruction encodings and the presence of data in sections of the executable which are typically reserved for code. Although compilers do not usually generate such code,[2] binary rewriting systems must be able to deal with hand-coded assembly routines. These routines are often designed to be as efficient as possible, and are sometimes non-conforming with regards to the standards used by the compiler. In particular, a programmer wanting to align a loop may choose to use invalid opcodes instead of valid NOP instructions, if it can be guaranteed that the bytes are never executed. A disassembler, however, does not have all the knowledge that the programmer had, and as a result is often unable to determine that a sequence of bytes

[2] An exception is position-independent code, which is explained in detail later.

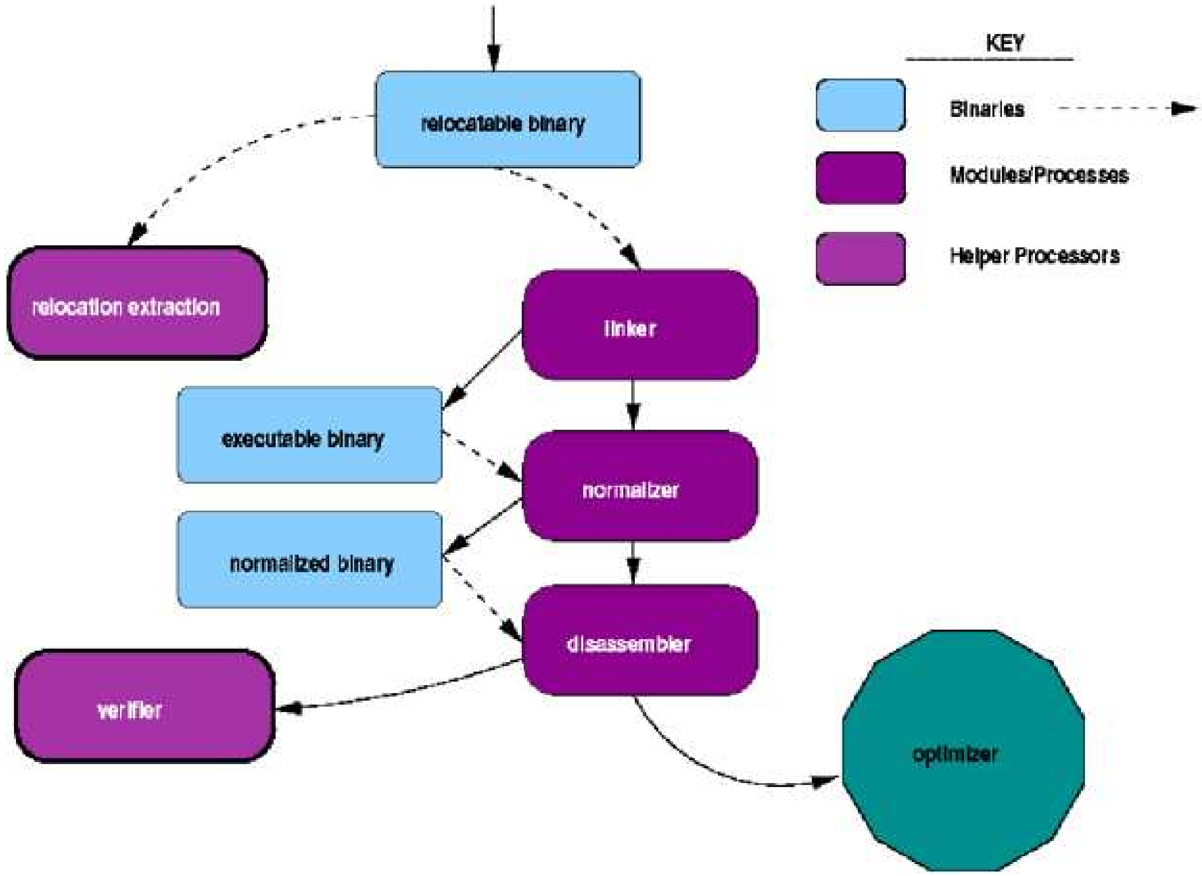

Figure 2: Pre-optimization Stages in PLTO

cannot be executed. Recent work has shown that the problem of disassembly is equivalent to the Halting Problem, and thus it is not always possible to correctly disassemble all code [25]. Moreover, it is not always possible to determine when a code fragment has been incorrectly disassembled! PLTO uses a novel approach to recovering the instructions from the byte stream that is a large improvement over other schemes used by binary rewriting systems, and over others discussed in the literature. It is able to detect position-independent code, jump tables embedded in the instruction stream, and data inserted for alignment purposes.

3.3.1 Position-Independent Code

Many compilers can be instructed to emit code that does not rely on being bound to any particular position in the program's address space. These code sequences are often referred to as *position-independent code* (PIC). In particular, PIC sequences do not contain any relocatable addresses embedded in the instructions. This property enables the code to work regardless of its memory location at runtime. Furthermore, PIC does not need to be patched by the loader, enabling it to be mapped as read-only data—which is useful for shared code such as dynamically linked libraries [17].

When a compiler is emitting position-independent code it typically creates jump tables that are also position-independent. These tables are usually embedded in the text segment of the executable and consist of a sequence of offsets rather than virtual addresses. A jump that uses the offset table first loads a nearby address,[3] then uses this to index into the table and retrieve an offset. The offset is added to the address that was previously loaded and then used in an indirect jump to reach the desired destination. The problems posed by position-independent jump tables are three-fold: (*i*) the offset tables, which are really no different than data, appear in the instruction stream; (*ii*) the code

[3] On the Intel x86 this is done using a "`call 0`" instruction followed by a "`pop %eax`" instruction, which has the effect of storing the latter instruction's address into register `%eax`.

```
Location       Memory Contents        Disassembly Results

               ...
0x809ef45:     eb 3c                  jmp 0x809ef83
0x809ef47:     00 00                  add %al, (%eax)
0x809ef49:     00                     add %al,
0x809ef4a:     83 ee 04 83 ee              0xee8304ee(%ebx)
0x809ef4f:     04 83                  add $0x83, %al
               ...
0x809efaa:     73 9e                  jae 0x809ef4a
               ...
```

Figure 3: Code Fragment from the C Library Routine `strrchr`

sequences that perform the indirect jumps are often complicated and may not adhere to a single pattern that is easily recognizable; and (*iii*) it is entirely possible that an offset table does not contain relocation entries. Taken together, these properties make the task of disassembling PIC sequences involving jump tables more difficult than standard code.

3.3.2 Disassembly with Linear Sweep

The most straightforward approach to disassembly is to start at the first byte in the *.text* section and disassemble the instruction there. The pointer into the section is then advanced by the length (in bytes) of the instruction disassembled, and the next instruction is recovered. This process continues until the end of the instruction stream is encountered. We term it a *linear sweep*, as no backtracking is ever involved and we are always making forward progress. This scheme is employed by programs such as GNU's *objdump* utility [13], OM [28], alto [21], and spike [9]. The shortcoming of this approach is that it disassembles data or alignment bytes if they appear in the instruction stream. This can in turn lead to erroneous disassembly of valid instructions if the end of the data does not also end what is disassembled as an instruction encoding. Furthermore, disassembling data can produce bad programs even if we stay "in sync" with what are real instructions; for example, a liveness analysis may produce incorrect results if it determines a register is not live (because it is overwritten by an instruction decoded from the data) when in fact it may be. The problem of data embedded in the *.text* segment is illustrated by the code fragment in Figure 3.

The code comes from the library routine *strrchr* found inside the standard C library (libc). The highlighted boxes show three 0x00 (NULL) bytes that were inserted by the programmer, presumably to push the header of the loop at `0x809e4fa` forward to an address with a more desirable alignment. The code produced in Figure 3 comes from the *objdump* utility, and we can see that it interprets the alignment bytes to be `add` instructions. The problem is that it begins decoding the second add instruction, and consumes a number of bytes that were meant to form a valid instruction at `0x809e4fa`. Subsequent disassembly is then incorrect, but later gets back "on track". By `0x809efaa`, the end of the loop is encountered and the instruction disassembled is a conditional jump back up to the loop header. This immediately looks suspicious, as the target of the conditional jump is not the beginning of an instruction, but rather the middle of the `add` instruction that was decoded. The instruction sequence is clearly invalid, but the problem remains that we cannot always detect such situations. In particular, if the disassembly was still out of sync when the final instruction in the loop was being decoded then we would never see the conditional jump back up to the loop header, and thus would not become suspicious. Normally a compiler uses 1-byte NOP instructions for alignment; this particular routine, however, was written in hand-coded assembly and the programmer was aware that the data would never be executed, so used a NULL byte instead. The code is perfectly valid, but the linear sweep disassembly scheme becomes confused.

3.3.3 Disassembly with Recursive Traversal

A second approach—one that is perhaps more intuitive—is to decode instructions in a similar manner to how the processor does, namely by following the execution of the program. The problem with the linear sweep is that it does not take into account the control-flow behavior of the program. Instead we consider starting at the program entry point—an address supplied as part of the program header. Instructions are then decoded linearly until branches, jumps, or function calls are encountered. Upon discovering an instruction in which the program counter can be changed to something other than the next instruction, the algorithm recursively visits all possible control-flow successors; we term the algorithm a "recursive traversal". For example, in the case of a conditional jump there are two possible successors: the next instruction, and the target of the jump that is visited when the condition is true. Function calls have similar behavior, and unconditional jumps have only one possible successor. Using this scheme the algorithm will visit only locations in the executable that are actually reachable from the starting point of the program. A high level sketch of the algorithm follows:

```
proc Disassemble(Addr, instrList)
{
  if (Addr has already been visited)
    return;
  do {
    instr = DecodeInstr(Addr);
    Addr.visited = true;
    add instr to instrList;
    if (instr can alter the program counter) {
      T = set of possible control flow successors of instr;
      for each (target ∈ T) {
        Disassemble(target, instrList);
      }
    }
    else Addr += instr.length;   /* addr of next instruction */
  } while Addr is a valid instruction address;
}
```

Re-examining the problem in Figure 3, it becomes evident that the recursive traversal does not try to disassemble the NULL bytes inserted before the loop header. The unconditional jump at `0x809ef45` before the alignment bytes, in conjunction with the conditional jump at `0x809efaa`, result in the algorithm disassembling around the padding as desired.

The key assumption made by this algorithm is that the set of all possible control flow successors for an instruction can be found. This is a bold supposition at the machine-code level, where dynamic function calls and obscure indirect jumps are often a reality. Specifically, the `switch` in C often results in compilers generating indirect jumps through jump tables. If static analysis cannot precisely determine the bounds and starting address of all tables with 100% accuracy, we risk disassembling data or not disassembling all instructions, either of which is a fatal error. Moreover, we cannot be sure that the same compiler (or even a compiler at all!) was used to generate the machine code being analyzed, so simply trying to detect normal forms is not feasible. Another technique that have been proposed is performing constant propagation during disassembly (which requires a partial CFG to be constructed), which does not seems straightforward [30]. In addition, it suffers from another flaw which is discussed below.

Presumably any target of an indirect jump is going to be encoded somewhere in the binary as a relocatable address, otherwise there would be no way to load the address into a register and jump indirectly through it.[4] Since we are equipped with relocation information, one feasible solution would be to start disassembling at all relocatable addresses. The augmentation to the recursive traversal algorithm is the following procedure which is called at the top level in place of *Disassemble*:

```
proc Disassemble'(Program)
{
  instrList = NULL;
  for each relocation, r, ∈ Program.Relocations {
    Addr = ReadContentsAtAddress(r.address);
    Disassemble(Addr, instrList);
  }
  Disassemble(Program.startAddress, instrList);
}
```

[4] Address arithmetic could be used, and in practice this can pose serious problems. We discuss it in detail later.

```
Location      Memory Contents            Disassembly Results

              ...
0x80b1d8b:    8d 84 c0 95 1d 0b 08       lea  0x80b1d95(%eax,%eax,8),%eax
0x80b1d92:    ff e0                      jmp  *%eax
0x80b1d94:    8d                         lea
0x80b1d95:    74 26 00                        0x0(%esi,1),%esi
0x80b1d98:    8b 06                      mov  (%esi),%eax
0x80b1d9a:    13 02                      adc  (%edx),%eax
0x80b1d9c:    89 07                      mov  %eax,(%edi)
              ...
```

Figure 4: Code Fragment from the C Library Routine `__mpn_add_n`

The algorithm seems reasonable in that it will start disassembling at all locations in the program which can have their addresses taken—which should cover all targets of indirect jumps and function calls. The only problem is that it may disassemble starting at some address that it not a possible target. For instance, Figure 4 shows a routine, `__mpn_add_n`, extracted from the C library on our Redhat 7.2 systems. The colored address in the figure corresponds to the bytes that are relocatable in the instruction encoding: `0x80b1d8d` to `0x80b1d91`. The contents at that address is the program address `0x80b1d95`. Using the modified algorithm, we treat `0x80b1d95` as an address at which to begin disassembly. The problem is that `0x80b1d95` actually points into the middle of an instruction! The `lea` instruction which loads `0x80b1d95` into a register performs address arithmetic with it. On the IA-32 the lea instruction "`lea` *baseAddress*(r_0, r_1, m), r_2" does:

$$r_2 \leftarrow baseAddress + \text{contentsOf}(r_0) + \text{contentsOf}(r_1) \times m$$

The function of the `lea` is to load an address located in the middle of a loop, which begins at address `0x80b1d98`. The programmer who wrote the function was aware that the register %eax never could contain the value 0, allowing control to be transferred to the address `0x80b1d95`—which was where we started disassembling. Unfortunately, this is specialized knowledge about the program that is not conveyed at the machine code level.[5] In general, it is impossible to determine via static analysis what values may appear in a register during the course of execution. Strange address arithmetic like this prevents us from being able to rely solely on the recursive algorithm.

3.3.4 Extending Linear Sweep

The first linear algorithm discussed in Section 3.3.2 was fairly primitive, in that it did not use all the information about relocations that is available at link-time. It is foolish to disassemble starting at addresses that are marked as containing relocations (e.g., *.text* -embedded jump tables), since an instruction encoding starts with an opcode which cannot be part of any relocation. Using this observation we can improve the algorithm by disassembling around blocks of addresses marked as relocations. Given some contiguous sequence of k relocations, the goal is to identify which ones correspond to a jump table, if any. For instance, when $k = 1$ it is likely that the address is simply part of an instruction encoding (e.g., the `lea` instruction from Figure 4 contains such an address). However, to simply assume such a fact is erroneous; although a compiler may not generate 1-entry jump tables, a devious programmer could. The same reasoning applies when $k = 2$. Two contiguous addresses may be embedded as part of an instruction encoding, or they could be a small jump table. An additional observation is required, which is that architectures have finite lengths for instruction encodings, and can contain at most m relocatable addresses within these bit-sequences. On the IA-32 $m = 2$—that is, 2 addresses can appear adjacent to each other within the machine code for one instruction. Applying this observation, we can conclude that in sequences of k relocations, $k \geq 2$, the last $k - 2$ relocations are data. If they were not data, our observation about having at most 2 adjacent addresses would be invalidated; but we know that to be fact. The question remains whether or not the first two addresses in the sequence are also part of the jump table. It is entirely possible that the instruction immediately before the jump table is one that ends in 2 addresses, and the jump table consists of the last $k - 2$. PLTO uses the following algorithm to decide:

[5] And not at the "source code" level either, except in a comment above the code fragment.

1. [*Phase 1: Conservative Marking.*] Find k-sequences of relocations and mark the last $k-2$ as data so the linear sweep algorithm disassembles around them.

2. [*Phase 2: Linear Sweep.*] Begin the linear sweep algorithm as described in Section 3.3.2. Before trying to decode each instruction, check to see if the current program counter is positioned at a location that was marked in Phase 1, or a location that contains a relocatable address. If either is true, check the last instruction disassembled to see how many relocatable addresses it contained at the end. Assuming the instruction had n addresses, $n \leq 2$, we now know the jump table following the instruction contains $k-n$ addresses. Advance the current pointer to the end of the table and resume disassembly there.

The algorithm is capable of handling data, such as jump tables, embedded in the *.text* section, but still does not produce correct results when the data or alignment bytes do not have relocations associated with them. For example, the improved algorithm produces the same result when applied to the code fragment in Figure 3.

3.3.5 Combining Linear Sweep and Recursive Traversal: A Hybrid Algorithm

Both linear sweep and recursive traversal have merits as well as pitfalls. Unfortunately, either of them applied alone can always produce undetectable disassembly errors: the improved linear sweep fails when unmarked data resides in the sections of the executable reserved for instructions; the recursive algorithm fails when a relocatable target is not the start of an instruction. Both of these failures are fatal and compromise the correctness of the program. The solution employed in PLTO is to make use of both algorithms. First the improved linear sweep is used across the entire program. The results of the linear sweep are then verified with the recursive algorithm.[6] When the recursive scheme disassembles an instruction residing at an address a_i, it checks to see that the linear algorithm also disassembled an instruction there. If it did not, the function containing the address a_i is marked as problematic,[7] and all the instructions between its start and end that were disassembled by the linear scheme are removed. PLTO then stores the machine code for that function so that later when the final binary is being generated, it can emit the exact same code. Since the function and the instructions it contains do not appear in the ICFG, there is no risk of performing transformations that affect the functionality of the procedure. Worthy of note is that switching the order of the two algorithms does not produce the same results, due to the recursive algorithm avoiding unreachable code. In particular, the linear algorithm is likely to find many instructions that the recursive routine does not.[8] For this reason, if the linear sweep were used to verify the results of the recursive traversal, it would issue many queries to see if an address it a_i was disassembled; in many cases the answer would be "no", simply because the instruction was not reachable. However, this does not imply that disassembly went wrong. For this reason we use the recursive traversal to verify the results of the linear sweep.

A few challenging implementation issues arise when procedures are not disassembled. Such procedures may contain relocations which point elsewhere into the executable, and these need to be patched appropriately. A load from a global *.data* section is such an example, as the section's address changes between pre- and post-optimization. Fortunately, we can use the relocation information to determine which bit-sequences in the chunk of machine code need to be updated. Another issue is one of escaping PC-relative branches— those in which the target function is not the same as the function containing the branch. When the machine code for the function is retained and later emitted, there is an implicit assumption that PC-relative branches do not need to be updated because their relative offsets to other instructions in the function are the same. This is true for branches in which the target instruction is also inside the same procedure, but not true when the branch is interprocedural. Furthermore, function calls on the IA-32 are little more than interprocedural branches, as they take PC-relative offsets to the target procedure. This poses a problem: the function has not been disassembled so we do not know where the interprocedural jumps and function calls are located, yet we need to update their displacements—which are not marked as being relocatable because they are PC-relative. An additional degree of complexity arises from the fact that the process of updating the displacement for a `call` or interprocedural branch may result in the size of the instruction encoding changing! For example, a function call which was previously located 120 bytes away from its target may be 300 bytes away after optimization, and to encode 300 as the offset for the `call` requires an extra byte than what was needed to encode 120. To make matters worse, extending the instruction by an extra byte would invalidate every other branch instruction in the non-disassembled procedure,

[6] Only the results from the first run need to be stored in memory. The second phase which verifies the disassembly does not need to create an actual representation of an instruction.

[7] Section 8 discusses improvements to this approach. For instance, one could attempt to discover which algorithm produced the correct results.

[8] About 10% more; see Section 5.5.

as the assumption was made that the offsets on intra-procedural branches would remain the same. A PC-relative branch that was previously *n* bytes away from its target may become *n + 1* bytes away, if the `call` that is being extended happens to fall between the branch and its target. The entire situation is depressing and full of challenging details. Currently, PLTO makes use of an extra relocation type for relative offsets that is available in ELF32 binaries. Unfortunately, the relocation is only available in compiler-generated routines, and hand-written assembled procedures (where the disassembly errors often occur) do not usually contain them. In these cases PLTO uses unsafe heuristics to try and discover the `call` instructions inside the procedure even though it cannot be disassembled. Section 8 discusses a *trampolining* method that we plan to employ to deal with the problem of procedure calls changing length when updating their offsets.

3.3.6 Experimental Results

There are two "performance" issues at hand: the actual speed at which the disassembly takes place, and the precision of the hybrid algorithm. As expected, we see that using both algorithms in conjunction requires about twice as much processing time. Appendix B shows the disassembly times for the statically-linked benchmark for the integer subsets of the SPEC95 and SPEC2000 benchmark suits. We believe that with some careful tinkering, the execution time for the hybrid algorithm could be improved so that it runs in approximately the same time as the linear sweep. The purpose of the tables is to show that the disassembly time is not increased drastically by applying our algorithm; moreover, the total time spent in disassembly is only a small fraction compared to the time spent during optimization. Table 1 shows how well the hybrid algorithm performs for the same benchmarks. There is little to gauge its precision against, however, since the other approaches (those in Sections 3.3.2 and 3.3.3 we have discussed are not correct. Taken alone, however, the hybrid algorithm is quite successful. On average over 99.6% of functions, accounting for over 99.8% of all bytes in the programs can be correctly disassembled.

3.4 Issues in Control Flow Analysis

Following disassembly, PLTO constructs an interprocedural control flow graph (ICFG) over the entire program. Two interesting issues arise in the analysis of a program's control flow: jump tables and unknown control flow.

Indirect jumps through a table of addresses are often generated by a compiler for multi-way branches, such as those arising from the *switch* construct in C. Several techniques have been proposed in the literature and implemented in systems for binary analysis. One method, which is used by PLTO, involves tracing backward through the instruction stream from the location of an indirect jump. In IA-32 assembly, a jump through a table takes on a fairly normal form:

```
 Address       Instruction                Comment
----------  -------------------------   ---------------------------------
 80481e3:   mov    0x8(%ebp),%edx        Load index into %edx
 80481e6:   cmp    $0x9,%edx             Bounds check against '9'
 80481e9:   mov    $0x6,%eax
 80481ee:   ja     8048241 <foo+0x61>    Jump to default case
 80481f0:   jmp    *0x808dbf0(,%edx,4)   Scale %edx by 4, add base address of
 80481f7:   nop                            table, and perform the indirect jump
 80481f8:   mov    $0x2,%eax             First case (0) in table
 ...        ...
 8048230:   mov    $0x9,%eax             Last case (9) in table
```

The idea is to try and recover the base address and size of the jump table. Typically the size of the table can be discovered from the bounds check that is performed against the index of the entry being jumped through. The index is scaled by the address size on the architecture, and added to the base address of the table. Cifuentes and Van Emerick propose a technique for detecting several normal forms that is similar to the scheme employed in PLTO [8]. Theiling suggests a different approach, in which constant propagation and construction of the CFG is done at the same time [30]. The hope is that the necessary constants propagate themselves down to location in which in the indirect jump is issued. One can then recover the possible targets by using all the information which found its way to the jump. We have found that detecting a few normal forms for indirect jumps enables PLTO to almost always correctly find the targets of these jumps. Only in a select few hand-coded assembly routines does the algorithm become stumped.

Program	No. of Functions			No. of Text Bytes		
	N_f	P_f	P_f/N_f (%)	N_b	P_b	P_b/N_b (%)
compress	570	4	0.70	291552	792	0.27
gcc	2418	3	0.12	1146304	736	0.06
go	919	4	0.44	485472	792	0.16
ijpeg	968	4	0.41	403664	800	0.20
li	928	4	0.43	334992	800	0.24
m88ksim	832	4	0.48	394656	800	0.20
perl	887	4	0.45	502768	800	0.16
vortex	1506	4	0.27	671936	792	0.12
GEOMETRIC MEAN:			0.38			0.16

(a) SPECint-95

Program	No. of Functions			No. of Text Bytes		
	N_f	P_f	P_f/N_f (%)	N_b	P_b	P_b/N_b (%)
bzip2	634	3	0.47	339216	736	0.22
crafty	673	4	0.59	449632	792	0.18
eon	2288	4	0.17	810256	800	0.10
gcc	2607	3	0.12	1384176	736	0.05
gzip	663	3	0.45	344464	736	0.21
mcf	572	4	0.70	294880	792	0.27
parser	884	4	0.45	385280	792	0.21
twolf	751	4	0.53	457184	792	0.17
vortex	1506	4	0.27	671936	792	0.12
vpr	832	4	0.48	391440	800	0.20
GEOMETRIC MEAN:			0.38			0.16

(b) SPECint-2000

Key:
N_f: Total no. of functions
P_f: No. of functions inferred to be "problematic" and not disassembled
N_b: Total no. of bytes in the *.text* segment
P_b: No. of bytes in "problematic" functions

Table 1: Precision of Disassembly

Given that we cannot always glean exact control flow information from the machine code, a means by which unknown control flow can be modeled is desirable. A special pseudo-node in the ICFG, B_{HELL} , is used to represent all types of unknown control-flow. For instance, sometimes it is not possible to recover the targets of indirect jumps or virtual functions calls.[9] When a basic block ends with an instruction whose entire set of successors can not be determined, an edge from that block is added to B_{HELL} . Similarly, when a block is a possible target of a indirect jump or function call[10] we add an edge from B_{HELL} to that block. B_{HELL} is contained in its own function, F_{HELL} , and in all of the analyses it is treated conservatively. For instance, B_{HELL} is assumed to use all registers upon its entry and then to define all registers. This ensures that liveness information is computed correctly in the presence of unknown control-flow, and that constants are not propagated across edges leading in or out of B_{HELL} . Other analyses and optimizations treat B_{HELL} and F_{HELL} specially; dominator computations, for example, are not performed when a function has blocks with incoming B_{HELL} edges.

4 Analyses

4.1 Stack Analysis

The scarcity of general purpose registers on the IA-32 results in a large reliance by the compiler on using memory. In particular, function arguments are placed onto the runtime stack by the caller and retrieved inside the body of the callee. The `alto` project showed that constant propagation across function boundaries can result in a significant performance improvement [21]. However, these observations were made on an architecture in which function arguments are almost always passed through registers. As the gap between memory speeds and CPU speeds increases, eliminating loads and stores to memory becomes increasingly important. For this reason, we would like to be able to reason about the relationships among stack frames of functions. A simple example follows, which illustrates the potential for optimization that we would like to exploit:

```
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: ...                                 g: push %ebp           # save old frame ptr
   push $456      # push arg 2            movl %esp, %ebp     # update frame ptr
   push $123      # push arg 1            subl $32, %esp      # allocate stack frame
   call g                                 ...
   addl $8, %esp  # pop args              movl 8(%ebp), %eax  # load y
   ...                                    testl %eax, %eax    # y != 0 ?
                                          jne ...
                                          ...
                                          leave               # deallocate frame
                                          ret
```

Somehow we would like to take advantage of the fact that the arguments to `g` are constants, whose values can be found in the body of `f`. Assuming `g` has only one call site, the test inside its body could be eliminated if the value of y was known. The problem is tricky, however, as `f` pushes the argument into a location on the stack that is determined by the value of the stack pointer (*%esp*), and `g` explicitly pulls the argument off the stack via a load from its frame pointer (*%ebp*). It would be nice to know the relationship between these two registers. Specifically, a particularly useful piece of knowledge would be the height of a stack frame at any given program point—the value *%ebp* − *%esp*. Given this bit of information for each function, one could discover that the `push` instruction inside the body of `f` is writing to the same location used by the "`movl 8(`*%ebp*`), `*%eax*" instruction inside `g`. Since the value of this location is known to be 456, the load of `y` inside `g` can be replaced with a simpler instruction: "`mov 456, `*%eax*". It is likely that the optimization would proceed in a transitive manner and replace some subsequent occurrences of *%eax* with the value 456. Given our knowledge about the contents of *%eax* prior to `test` instruction, we can also determine the outcome of

[9] In fact, the targets of such calls may not even be defined until runtime.

[10] We can find this information by perusing addresses sitting in the data and read-only data section

the test and determine which way the conditional branch `jne` goes. The savings in this particular example are at least one less memory reference and two less instructions from being able to eliminate the `test` and `jne`.[11] Moreover, in this particular example the propagation also opens the door for dead code elimination. After replacing the load of `y` inside `g`, the `push` instruction in `f` writes to a location that is not ever used by subsequent instructions. Prior to the transformation, the location was used as a placeholder for `y` and was thus necessary. Following the transformation, however, the location is dead and the `push` serves no purpose. Thus, a round of dead code elimination is performed every time constant propagation is carried out. The result is the elimination of even more memory references. The optimized code follows:

```
f: ...
   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 456, %eax      # load y
   ...
   leave               # deallocate frame
   ret
```

4.1.1 Algorithm and Equations

The analysis is fairly intuitive. Each instruction in the program contributes some number of bytes to the height of the stack (most instructions contribute nothing). The contribution made by a basic block is simply the sum of all the instructions it contains. Given the effect that each block has on the stack, we can iteratively propagate these contributions around the CFG for a function. The equations for dataflow through a basic block, *B*, that is contained in a function *f* follow:

$$HeightOut(B) = HeightIn(B) + |B|$$

$$HeightIn(B) = \begin{cases} \perp & \text{if HeightOut(p)} \neq \text{HeightOut(p'), for some p and p'} \in \text{Predecessors(B) ;} \\ c & \text{if } \forall \text{ p} \in \text{Predecessors(B), HeightOut(p) = c} \end{cases}$$

subject to the initial conditions:

$$HeightIn(B) = \begin{cases} 0 & \text{if B} \notin \text{Successors(}B_{HELL}\text{) and B} \in \text{Successors(Entry(f)) ;} \\ \top & \text{if B} \notin \text{Successors(}B_{HELL}\text{) and B} \notin \text{Successors(Entry(f)) ;} \\ \perp & \text{if B} \in \text{Successors(}B_{HELL}\text{)} \end{cases}$$

The meet operator is defined analogously as it in constant propagation—a block with two incoming edges of different contributions results in a production of $\perp$. Such scenarios occur rarely in code we have inspected, but they do exist. For instance, a function call may be executed conditionally and the compiler may choose to not deallocate the arguments it pushes on the stack. This results in two execution paths with different contributions. Usually these scenarios occur toward the end of functions when no existing references to the stack are made, so their presence does not prevent the stack analysis from being effective.

4.1.2 Interprocedural Considerations

Hand-coded assembly routines may not adhere to conventions that we take for granted from the compiler. Interprocedural jumps and non-returning function calls are a reality. In some frequently-called routines that are written to be as fast as possible,[12] control often jumps between two functions so that the cost of procedure calls is not incurred. There are also circumstances where functions do not deallocate the stack frame they have set up, but instead leave the job to the calling function. The code can be considerably faster when only one function has to clean up instead of both. Furthermore, a smart compiler may also choose to carry out a transformation that generates such code. For this reason we cannot always be sure that the height of the stack (the value in %esp) is the same after a function call as it was

[11] This assumes that the PSW bits set by the test instruction are not used by any subsequent instructions. If they are in fact used, the `cmp` must remain in the function to preserve correct behavior.

[12] We see such situations in *malloc()*, the *setjmp()* family, and some math routines.

before. Formulated in terms of our analysis, we cannot be sure the contribution of a `call` instruction is 0 bytes. This adds an interesting twist to the problem. To carry out an analysis of the stack inside a function we need to know about the contribution of any functions that it may call. This requires knowing the height of the stack at the exit points for these functions, which poses a chicken-and-egg problem. To compute the stack frame size inside a function we need to have computed the stack frame sizes for all the functions that are reachable from it. This is not always possible; e.g., suppose *f* calls *g* and *g* also calls *f*.

PLTO uses a notion of function *well-behavedness* to address this problem. A function *f* is said to be well-behaved if and only if we can guarantee it leaves the stack as it found it. An initial set of well-behaved functions is constructed by performing a simple local analysis for each function. One condition that guarantees well-behavedness is the existence of compiler-generated function prologues and epilogues, which typically push the frame pointer onto the stack upon procedure entry and pop it off before returning.[13] The stack size remains unchanged regardless of any non-zero contributions that exist before this instruction is executed. Unfortunately not all functions will have a prologue and epilogue, as these may be optimized away or not generated at all. For the remaining functions in which nothing is known about their behavior, PLTO makes the optimistic assumption that they are well-behaved. This is later refined if we discover that the assumption was not true.

The implementation of the stack analysis involves a queue of functions, which initially contains all the functions in the program. As stack sizes are computed, a function may be found to be not well-behaved. When this occurs, all predecessors of this function in the call graph are re-enqueued, and their stack sizes are re-computed with the knowledge that one of their procedure calls is no longer well-behaved. The entire analysis is thus broken down into two phases:

1. [*Local analysis.*] Functions with prologues/epilogues that guarantee stack restoration are marked as being well-behaved.

2. [*Iterative propagation.*] Stack sizes at every program point are computed using information about well-behaved functions. Functions that are found to be not well-behaved inform their predecessors, and the information propagates back up the call graph.

We find that between 85% and 95% of all functions contain appropriate prologue and epilogue code, and hence are well-behaved. About half of the remaining functions are found to be well-behaved due to the net contributions being 0 at all exit points. PLTO is forced to deal conservatively with the other remaining functions and assume they are not well-behaved.

The stack analysis is used in almost all optimizations throughout the system. Function inlining uses the information when merging procedures to eliminate instructions that set up a new stack frame inside the callee. It adjusts the callee's references to the stack and writes them in terms of the stack for the calling function. Constant propagation uses the information to propagate constant values put on the stack by `push` instructions into the body of the callee. Table 2 shows the effect of using the stack analysis in constant propagation on the SPECint95 benchmark suite. On average, it enables about 20% more register or memory operands to be replaced by constant values.

Load/Store Forwarding also uses the stack analysis in a similar manner, except it does so to propagate registers in place of stack locations as opposed to constants in place of registers.[14] Without the stack analysis the optimization would be severely crippled. Our approach to interprocedural stack analysis is novel. Related work includes the Java bytecode verifier, which performs a similar local analysis to guarantee that functions do not affect the stack size [23].

4.2 Use and Kill-Depth Analyses

Performing analyses across interprocedural boundaries can be extremely time-consuming. But treating interprocedural edges too conservatively (e.g., assuming that a `call` instruction defines all registers and stack locations) can severely impact the effectiveness of an optimization. Often times a balance between the two extremes can achieve acceptable results and still run in a reasonable amount of time, with the added benefit of being easier to implement. PLTO carries out most optimizations across the ICFG with interprocedural considerations. In addition, it can also perform optimizations local to each function, using summary information about other functions to handle interprocedural aspects. *Use*- and *kill*-depth information falls into this category—it is used in constant propagation and liveness

[13] On the IA-32 this is usually accomplished with the `leave` instruction, which has the effect of restoring both the frame pointer and the stack pointer to their previous states.

[14] We later refer to this optimization as *register propagation*, as it is carried out concurrently with constant propagation

Program	*No. of operands replaced*		Percent Increase
	Before	After	
compress	318	389	22.33
gcc	427	593	38.87
go	423	478	13.00
ijpeg	372	439	18.01
li	330	397	20.30
m88ksim	347	433	24.78
perl	553	640	15.73
vortex	517	701	35.59
Geometric Mean:			22.11

Table 2: Effect of Stack Analysis on Constant Propagation

analyses to provide an improvement when the optimizations are not carried out across procedure boundaries. Finally, the information is also used in the interprocedural analyses as a solution to some issues concerning representation of the runtime stack. Section 5.1.3 discusses the stack representation in greater deal and serves as motivation for using these analyses even in an interprocedural optimizer.

A function's kill-depth describes the amount of space below its own stack frame that it may write to. The value is either positive or ∞, meaning some instruction inside the function may write to any location on the stack. Such instructions are generally stores into the heap, but PLTO is usually unable to infer this information due to the loss of semantic information accompanied with machine code, where indirect stores may be going to the stack, statically allocated data regions, or the heap. In intraprocedural constant propagation the kill-depth of a function is used when a `call` to some function is seen. Instead of assuming the worst-case—that the function may destroy the entire contents of the stack—we use kill-depth to limit the extent of the damage. Use-depth is the dual of kill-depth, used in intraprocedural liveness analyses. PLTO does not have to assume that the function being called reads all locations on the stack, instead use-depth can be used as an upper bound. The pseudo-function F_{HELL} is assumed to have a kill-depth and use-depth of ∞. Consequently, any function through which F_{HELL} is reachable must also have values of ∞, since execution of these functions could result in F_{HELL} being reached. The intuition is that if we do not know where control ends up (and thus are in the presence of F_{HELL} or B_{HELL}), the worst-case assumptions need to be made. The remainder of this section describes how kill-depth is computed; the case for use-depth is analogous.

The analysis is broken down into two phases: a local computation for each function, and iterative propagation of the local computations. Iterative propagation is necessary as kill-depths may span multiple stack frames and affect the kill-depths of other functions; F_{HELL} is such a case. In the local analysis, each instruction in the function is analyzed to determine the extent of the location to which it may store. Indirect stores through registers are treated conservatively and assumed to have a kill-depth of ∞. The "deepest" store to the stack inside a function is then set to be the function's kill-depth. The second phase of backwards iterative propagation along the call graph proceeds as follows: consider some function f with a current kill-depth of m. f statically has procedure calls, C_1, C_2,...,C_n to some other set of functions in the program, f_1, f_2, ... f_k with kill-depths g_1, g_2, ..., g_n respectively. From the stack analysis described in Section 4.1 we know the height of the stack, p_i at each call site C_i, $1 \leq i \leq n$. Let d_i denote the new kill-depth of f after propagation from g_i. d_i is computed according to the following rules:

- If the stack height at the call site C_i is unknown, that is $p_i = \perp$, we cannot determine the size of the stack frame at the call site. In the worst case the stack has not grown any (e.g., $p_i = 0$). Consequently, when f_i writes g_i bytes below it stack frame, the write will also be g_i bytes below f's stack frame since the call site was at height 0 in f's stack. In this case, $d_i = \max(m, g_i)$.
- If the stack height at the call site is a known value, that is $p_i \neq \perp$, d_i is computed as $\max(m, \max(0, g_i - p_i))$. The intuition is that a store to the stack g_i bytes below f_i's stack frame will write to a location $g_i - p_i$ below f's stack frame, and thus affect its kill-depth if this value is larger than f's current kill-depth.

This proceeds until a fixpoint is achieved, so that extremely "deep" stores (e.g., those that are ∞) can progagate

Program	*Eliminatable Instructions*			Percent From Stack
	Stack	Registers	Total	
compress	17	879	896	1.89
gcc	9	3057	3066	0.29
go	21	1561	1582	1.33
ijpeg	17	1197	1214	1.40
li	18	1123	1141	1.58
m88ksim	18	1115	1133	1.59
perl	17	1437	1454	1.17
vortex	51	2157	2208	2.31
Geometric Mean:				1.27

Table 3: Effect of Stack Liveness Analysis on Dead Code Elimination

upward to all places from which they are reachable. Unfortunately, when used in the intraprocedural analyses the use-depth and kill-depth are unnecessarily large at times; in fact, many functions receive ∞ because they have reachable paths to F_{HELL} or indirect stores. In practice the analyses are ineffective at improving the precision of constant propagation and liveness analyses. However, they are required in the interprocedural analyses as a means to fix a problem with representation of the stack. Since the stack is represented as a large array with a finite upper bound, a store extremely deep into the stack to a location that is larger than the upper bound on the stack size cannot be modeled in the interprocedural optimizations. In these cases, the kill-depth and use-depth are used to be sure that the correct information propagates backwards.

4.3 Liveness Analyses

Liveness analysis is performed to eliminate instructions that store to locations that are not subsequently used. Constant propagation and function inlining create dead code, so the dead code elimination is run after these are carried out. The IA-32 has only eight general purpose registers, and as such relies heavily on the runtime stack. Eliminating writes to stack locations is potentially more fruitful than eliminating writes to registers, since memory operations are expensive. For this reason PLTO carries out both a register and a stack liveness analysis. In addition a PSW analysis is used by both these analyses. In order for an instruction to be eliminated, the PSW bits it defines much also be dead.

4.3.1 Stack Liveness Analysis

PLTO carries out a context-insensitive, intraprocedural stack liveness analysis complemented with the use-depth information presented in Section 4.2. Table 3 shows the number of eliminatable instructions from using this analysis, along with their percentage as the number of total eliminatable instructions. Surprisingly, the stack liveness analysis is relatively ineffective, accounting for only a small percentage of total eliminatable stores in most cases. Indirect loads play a major factor in hindering the effectiveness of stack liveness analysis; in particular, at any program point that has a reachable path to an indirect load, all stack locations are live. Section 8 discusses some future work we have in mind to combat this problem.

4.3.2 Register Liveness Analysis

Register liveness analysis in PLTO is interprocedural and comes in both the context-insensitive and context-sensitive flavors. The context-insensitive analysis is a straightforward implementation of what has been described in many papers [19, 14, 21, 28]. The context-sensitive analysis considers only realizable paths in the ICFG, and has also been discussed in some deal in recent work [19, 14]. The basic idea is illustrated in Figure 5, which shows how information can legally propagate in an ICFG through unrealizable paths by using call and return edges.

The blue path leading up the graph is the path on which the use of x propagates back to the call site from which it is not reachable. The context-sensitive analysis in PLTO is based on the dataflow equations by Muth [19] and Goodwin [14]. The context-sensitive analysis computes summary information for each function, which holds only if control enters from the first block in that function. Some care must be taken in the presence of control flow irregularities like

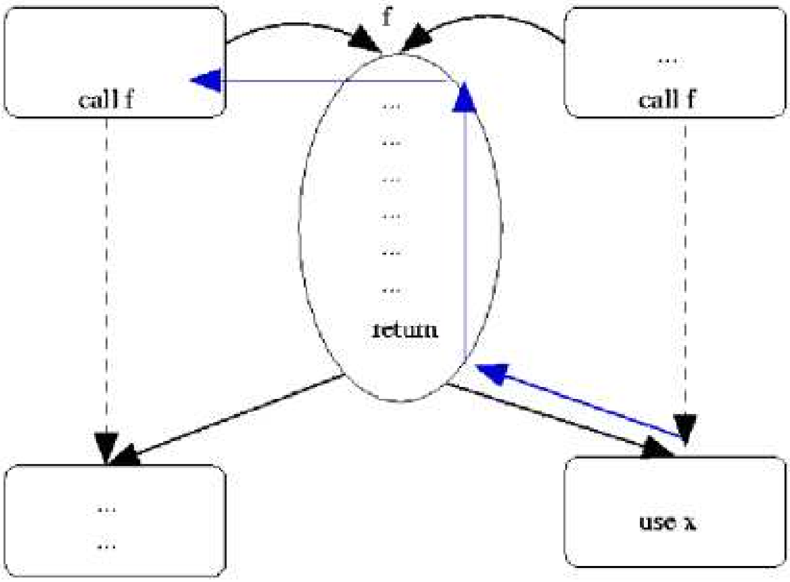

Figure 5: Backward Propagation Along an Unrealizable Path

interprocedural jumps and function calls to the middle of procedures. Muth found that on the Compaq Alpha one could expect about a 70% increase in the number of available registers using the context-sensitive approach.

On the IA-32 we find that the context-sensitive analysis performs better, but only by about 7% on the average; we see an increase from 1.67 dead registers up to 1.8. Table 4 shows the effectiveness of both approaches with regards to the number of free registers they find. Section 8 discusses some potential uses for these registers.

Program	*Free Registers*		Percent Increase
	Insensitive	Sensitive	
compress	1.735	1.847	6.45
gcc	1.494	1.647	10.24
go	1.886	2.143	13.62
ijpeg	1.642	1.738	5.85
li	1.636	1.753	7.15
m88ksim	1.718	1.825	6.32
perl	1.659	1.799	8.44
vortex	1.580	1.644	4.05
Geometric Mean:			7.31

Table 4: Effectiveness of Context-Sensitive Register Liveness Analysis

The effect that callee-saving register sequences in a function prologues have on liveness is worthy of note. Typically a function will save the set of callee-saved registers[15] upon entry by storing them somewhere in its stack frame. Since the act of storing them to the stack (usually done with a `push`) constitutes a use of the register being saved, liveness information propagates backwards and the calling function believes these registers are later used. This is usually incorrect, as convention (and lack of registers) dictates that function arguments be passed through the stack, meaning these registers should only be live if later used within the calling function before being defined. Ideally we would like to discover that although these registers are used, they are later restored in the function epilogue (usually via

[15]These are typically %ebx, %esi, and %edi on the IA-32

Program	*Static Eliminatable Instructions*		Percent Increase
	Standard Analysis	Non-conservative Analysis	
compress	896	997	11.27
gcc	3066	4335	41.38
go	1582	1871	18.26
ijpeg	1214	1377	13.42
li	1140	1328	16.49
m88ksim	1133	1292	14.03
perl	1454	1852	27.37
vortex	2208	2797	26.68
Geometric Mean:			19.31

Table 5: Potential Benefit from Knowing Callee-Saved Registers Are Not Live

`pop` instructions) and not subsequently used, meaning their contents are not live. Unfortunately, like in other analyses throughout PLTO, indirect loads must be treated conservatively—we assume they can come from the locations on the stack to which the callee-saved registers are stored. Although it is highly unlikely that a function will read from these locations, it can be a reality in hand-written assembly routines. The potential payoff for knowing that these registers are not live is rather large. Table 5 shows how the effectiveness of dead-code elimination would improve if we could assume that the callee-saved registers are not live upon function entry. Section 8 discusses an analysis we are currently investigating that would limit the range of indirect loads and stores; we hope it will eventually lead to improvements close to what are seen in Table 5. Although assuming that callee-saved registers are not live is technically incorrect, the programs in SPECint95 ran correctly for us after testing out them out with that assumption. These unsafe assumptions have been made in several systems: Goodwin suggests an approach in which indirect calls to unknown targets obey the standard calling conventions defined for the architecture [14]. This was implemented in the context of the Spike optimization system [9]. Muth describes a similar approach in which a *saved*-register set is constructed for each function, which contains a representation of all the callee-saved registers in that procedure [19]. It is not clear what sort of analysis, if any, is used to determine that the locations to which the registers are stored are not read from. The author also describes a slight modification to computing liveness in the presence of indirect function calls whose targets are not known, in which calling conventions are assumed to hold [19].

4.4 Analysis of the Optimization Potential of Code Fragments

Many transformations are carried out for the express purpose of creating better opportunities for other optimizations. Other transformations, such as function inlining, have their own expected payoffs but also enable other optimizations to perform better. In general, transformations that have their own payoffs do not consider the potential gains they may enable other optimizations to take advantage of. A novel approach to optimization is to consider both the immediate payoff and also any future payoff, when deciding where and when to carry out transformations such inlining, specialization, and cloning.

Value specialization, function inlining, and cloning are three optimizations that are carried out with other optimizations in mind. Value specialization relies on constant propagation and dead code elimination to create more efficient code fragments [20, 31]. The transformation involves cloning a region of code to be specialized for a value for a particular register or stack location, v. Candidates for specialization are gathered from profiling registers (and perhaps stack locations) in the program and determining the distribution of their values. A region of code is then duplicated, and a test for equality to v is inserted before control can enter one region or the other. The constant propagator is then able to infer that inside one region of code the register provably contains the constant v. The hope is that the constant propagator can make the cloned region of code much more efficient by knowing this fact.

Function inlining is another transformation that is carried out for two reasons: to eliminate the `call` and `return` overhead associated with procedure calls, and to provide better opportunities for constant propagation by reducing the number of calling contexts for the inlined procedure. A function f taking one argument may be called from five different places in the executable. In every spot the argument may be a different constant, so f cannot be optimized with respect to any one value. If, however, f is inlined into a call site, the number of calling contexts for f is effectively reduced to one, and the inlined procedure can be optimized with respect to one of the values. Finally, cloning is a transformation much like specialization, but makes use of existing transfers in control-flow rather than introducing new tests.

The general problem is the following: We would like to compute the expected future payoff of performing transformations like specialization and inlining without actually having to carry out the optimizations themselves. The problem has previously been addressed in compilers [3] and in the `alto` link-time optimization system [20], but characteristics of the IA-32 (and CISC architectures in general) warrant that the problem be revisited. Computations were carried out in a compiler on the procedure level in work by Ball [3]. Strong and weak dependence sets were constructed, where the set members were function parameters. Strong sets indicated a variable's value could be fully computed from the members of the set; weak sets indicated a variable's value was influenced by the members of its set, but they were insufficient to completely determine the value. The sets are then used when a function call with some constants parameters is seen, and an estimate on the reduction in code size and execution time is made based on the strong sets. Although this method works well to predict savings from constant propagation and dead code elimination after function inlining is carried out, the analysis does not completely meet our requirements. We would like to be able to carry out this analysis at any arbitrary program point for any program variable (in our case, register or stack location). It is not clear that extending this analysis would be easy or even work well for our needs. The authors of the `alto` system showed how to extend such computations for whole-program analysis at the cost of large memory requirements [20]. In particular, use-definition chains were constructed on top of the ICFG, and the chains were traversed in an intelligent manner to determine which instructions would become eliminatable from performing specialization.

The IA-32 poses a new challenge of resource usage. In particular, one cannot expect to gain a realistic estimate on the expected savings by considering only register usage as was done in work for the `alto` system [20]. Many instructions use operands located on the runtime stack and in statically allocated data. Modeling the stack in a similar manner to what is done for constant propagation and dead code elimination (Section 5 provides extensive details) is cumbersome and space intensive, due to the size of use-def chains. We have developed an on-the-fly approach in PLTO that requires no additional resources over what are already allocated, but at the expense of sacrificing some precision.

4.4.1 On-Demand Computation

One method to discover the eliminatable instructions is to iteratively propagate information around the CFG until a fixpoint is achieved. The basic idea is to mark an initial set of operands that are known to take on constant values. For value specialization one would mark the single operand being considered in order to estimate how profitable specialization is. In function inlining one would mark the stack locations which contain constant values from "push" instructions—they are the constant arguments that will enable constant propagation to simplify code. After marking the necessary registers or stack locations as being known, a phase of iterative propagation takes place. The two values that are propagated are $\perp$—to represent that a location is unknown—and k, or any other arbitrarily chosen symbol—to represent that the location contains a known value; the exact value is irrelevant. Instructions in which all the source operands are known are marked as being eliminatable, and their destination operands (if they exist) are also marked as being known. The meet operation for locations (register, stack slot) is defined in the standard fashion, producing $\perp$ when any incoming edge has $\perp$.

Some unmarking of eliminatable instructions is required when the meet operation produces $\perp$ for a location, and some incoming branch contains k for that location. Figure 6 illustrates the problem.

The left side of the figure shows a hypothetical CFG of 3 blocks in which the potential for knowing that the variable x contains a known value is being computed. In *B1* the value of x is known, but the variable y does not contain a known value. As such, z cannot be fully computed and the instruction is not eliminatable. In *B2* however, x is added to the constant 2 and stored in z. Since both x and 2 (trivially) contain known values, the value of z can be computed and the instruction is perhaps eliminatable. On the right side of the graph is the result after propagating this information to *B3*. The red variables indicate ones that were known, and the large red dot indicates the instruction was marked as being eliminatable. It should be noted that when the meet operation is applied to z, it finds that z is not known since along the *B1* branch it could not be computed. Since z is not known at the start of *B3*, the use of z cannot be replaced with the constant it contains when constant propagation is carried out. Consequently, we must execute both the instruction in *B1* and the instruction in *B2* so z contains the correct value at the entry to *B3*. The initial analysis concluded otherwise, marking the instruction in *B2* as one that goes away after subsequent optimizations are performed. Thus, we need to

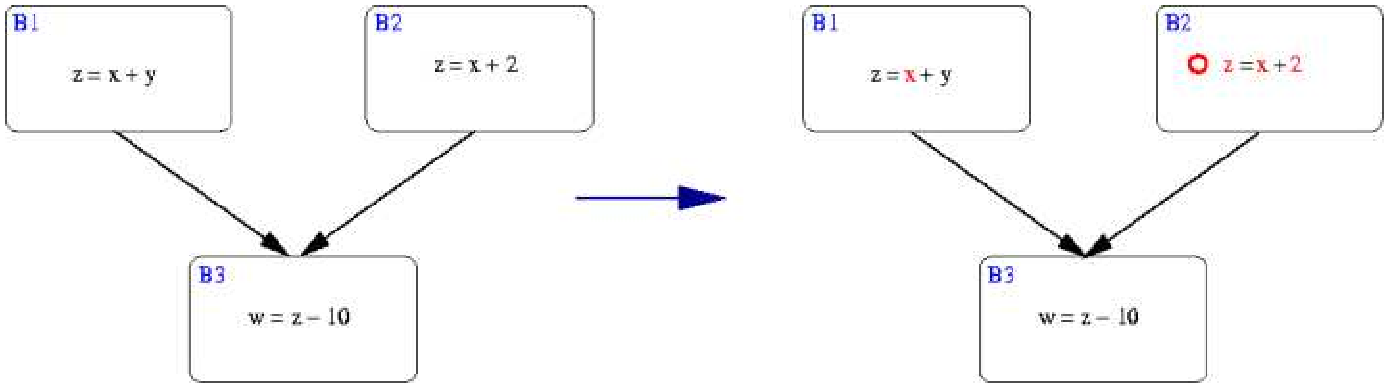

Figure 6: Incorrect Marking of Eliminatable Instructions

carry out some form of instruction unmarking when the meet operation produces $\perp$ and some branch contains k.

Instruction unmarking need not proceed transitively back up the CFG, and iterative propagation is not necessary. Consider a register r_i that is $\perp$ coming in along one branch and k along the other. If we walk back up the k branch and continue until we find the instruction that defines r_i, all properties that we think are true at this program point are in reality true. For all registers r_j that contain either a known or an unknown value, they were assigned $\perp$ or k based on information that propagated down the graph in a legal manner. If we unmark the instruction that defines r_i this cannot cause other instructions that have been marked to be candidates for unmarking. In particular, consider both an instruction before the one we unmarked, I_n, and an instruction after the one we unmarked, I_m. If I_n uses or defines r_i, it is of no consequence. The properties about the registers which propagated down to the I_n still hold, and anything that is marked should remain marked. The reasoning is similar for I_m. Suppose I_m uses r_i, and we had marked I_m as eliminatable because all the source operands were known. The fact remains that I_m is still eliminatable, despite the unmarking of the instruction defining r_i. This is because the fact remains that we still know the value of r_i inside the block even though it is unmarked. As a result, any uses of r_i can be replaced during constant propagation by the actual value of the register. Given that we do not need backward propagation, the complete algorithm is only one phase and is fairly simple to implement. After a round of propagation produces no changes, a new set of operands are marked, and a number of instructions are marked as evaluatable. The weights of the eliminatable instructions can be summed to get an estimate on how beneficial knowing the value of one or more locations may be.

PLTO implements a slight variation on this scheme in which a list-scheduling algorithm is employed in place of iterative propagation. Specifically, a block is not evaluated unless all its predecessors have been evaluated. We plan to integrate the fixpoint approach in the near future. The resource requirements of these algorithms are fairly minimal. No use-def chains need to be constructed, which saves a great deal of initial overhead time and memory usage. However, each query may take longer to execute since traversing use-def chains can proceed faster than our algorithm. We find that in practice the overall time requirements are reduced by using an on-demand approach, and we require no extra storage that has not already been allocated for optimizations like constant propagation and dead code elimination.

4.5 Instruction-Cache Analysis

Many transformations carried out by compilers and link-time optimizations have effects on the size of the resulting code. For instance, function inlining creates a duplicate copy of a procedure and merges it in place of a function call. The code size grows if, after inlining, the function still has other call sites and thus cannot be removed. Profile-guided value specialization is another optimization that can grow the size the program. Both inlining and value specialization, as implemented in PLTO and other optimizers like `alto`, perform cost-benefit analyses to determine where these optimizations should be carried out. For instance, the value-specialization analysis attempts to determine which program points are good candidates for specialization. A "good" candidate may be one where the expected payoff, in terms of the number of cycles saved, is positive. That is, if we look at the cost associated with inserting the test for the specialized value (typically a compare and a branch instruction), we find that it is less than the number of cycles that we expect to save in the specialized piece of code. Although this may provide us with a general feel for which locations are good, it is not a completely accurate model. One would like to account for the side effects that code growth may have on the execution behavior of the program.

Code growth can have particularly devastating effects on the behavior of a program with regards to the instruction cache. The gap between memory access speeds and CPU speeds is large enough that additional misses in the i-cache (which ultimately result in memory accesses to read the appropriate block) can offset any gains that one expects from having specialized code. As the memory/processor gap widens, this problem becomes even more important to address. Ideally, we would like an oracle to tell us the exact number of extra i-cache misses the program will incur as a result of the transformation. One could then estimate the penalty for having to read from memory, and compute the number of extra cycles the i-cache misses are responsible for. This penalty could then be used in conjunction with the cost-benefit analysis; specifically, it could be added to the cost associated with performing the transformation. Thus, any decision about inlining or specialization becomes cache-conscious.

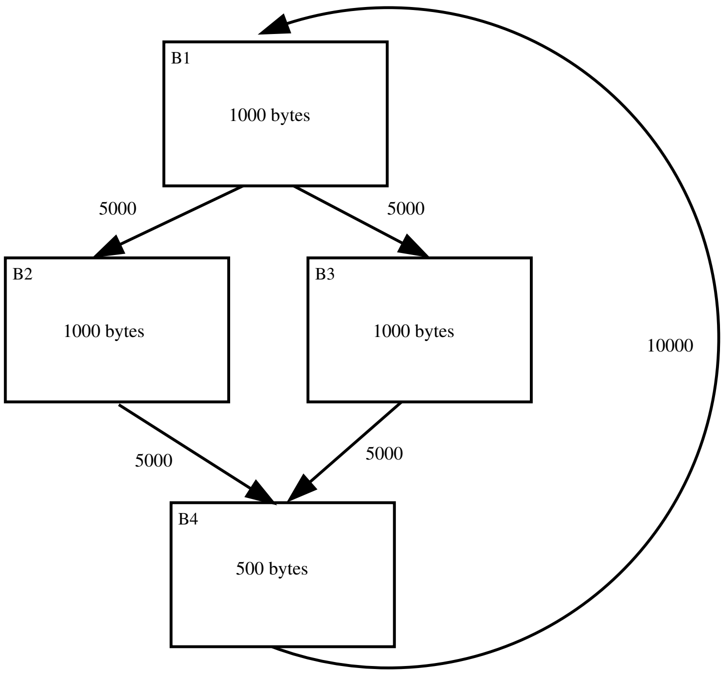

Figure 7: A simple 4-block loop with block sizes in bytes

Unfortunately, we do not have perfect knowledge about the program. Edge profiling gives no indication of the temporal relationship between the issuing of instructions. For example, Figure 7 shows a loop that will have an i-cache footprint of 2500 bytes if the false edge of B1 is taken 5000 times and then the true edge is taken 5000 times. The code for B2 need not be stored in the i-cache when B1, B3, and B4 are executing in the loop. Similarly, the code for B3 need not stored in the i-cache when B1, B2, and B4 are executing. The same loop would have a footprint of 3500 bytes if the branch instruction at the end of B1 alternated each time. B1, B2, B3, and B4 would all need to be resident in the cache at the same time. The edge weights do not help us in determining which of the two situations we might be dealing with.

4.5.1 A Non-conservative model

McFarling, in a paper on instruction cache considerations when merging procedures, proposes a novel technique for computing the sizes of loops in a program [18]. A probabilistic approach is used to compute an expected cache-footprint for each loop in the program. Inlining decisions are then made using a heuristic-based cost-benefit analysis. The problem of choosing the order in which to inline functions is shown to be equivalent to the knapsack-problem, and thus is NP-hard. The gist of the approach is to compute a frequency for each instruction; this frequency represents the number of times one expects it to be executed during one iteration of the loop. The frequency is capped at 1.0, since an instruction executing twice or three times during a loop only finds its way into the cache one time. However, for an instruction with a frequency of .1, we can roughly say it takes up only 10% of its total size in the cache. In the example illustrated in Figure 7, the probabilistic model works well when one path is followed many times, then the other path is followed. It performs poorly when the paths alternate, and both are resident inside the cache, because the expected loop size is half of the real size. This property of McFarling's algorithm leads to estimates that are not conservative; in particular, they can lead to inlining decisions that result in poor i-cache behavior.

The interprocedural considerations of the algorithm makes the problem more interesting. For a given instruction *I* not in the same function *F* as the header of the loop, the frequency of *I* can be computed in several ways. The most intuitive method is to compute the frequency of *I* in the normal manner (divide *I*'s execution count by the count of the loop header) and multiply by a scaling factor *s*. *s* is computed by examining the percentage of total calls to *F* that come from the loop. For example, if the function containing *I* has 10000 call sites, 1000 of which come from within the loop, then the scaling factor is .1. The nature of the algorithm leads to a simple two-pass recursive implementation. In the first pass, frequencies are assigned to each basic block:

```
proc AssignFrequency(BasicBlock, ScalingFactor)
{
  BasicBlock.frequency = max(1, (B.weight / LoopHeader.weight) * scalingFactor);

  foreach intraprocedural, non-visited successor, s, do
    AssignFrequency(s, scalingFactor);
  end

  foreach interprocedural, non-visited successor, s, do
    AssignFrequency(s, scalingFactor * (B.weight / s.weight));
  end

}
```

Minor bookkeeping is involved to prevent blocks from being visited more than once. Once frequencies are assigned, a second phase—similar to the first, and not shown here—computes the i-cache footprint of a loop by summing the products of each block's frequency and size. The interprocedural aspect of the algorithm is not well-described in McFarling's work, but this algorithm is a logical extension. A more clever scheme—which would require context-sensitive profiling or path profiling—would not use the probabilistic assumptions. For example, when a procedure call to a function is encountered, one could try to discover the real execution paths taken inside that function given the context of the call site. Currently, our algorithm decides that if the call site accounts for 10% of all calls to the function, then the function's total i-cache footprint should be scaled by .1. In practice this is not likely to be the case, as functions tend to have many paths through them that are often determined by the calling context. This same problem arises when trying to re-assign edge weights inside functions that have been cloned for inlining.

4.5.2 A Conservative Model

The model described in Section 4.5.1 works best when the pattern of control-flow inside the loop is somewhat regular. In Figure 7 the loop size is accurately estimated when control flows along one path for a large number of iterations, then along the other path. The footprint is underestimated when control alternates among the two paths frequently. PLTO implements, in addition to the model proposed by McFarling, a more conservative algorithm for computing loop sizes. The basis of the approach stems from the observation that in situations like in Figure 7, it may be the case that both B2 and B3 are competing for space in the instruction cache. This happens when control switches from the path B1 $\rightarrow$ B2 $\rightarrow$ B4 to B1 $\rightarrow$ B3 $\rightarrow$ B4 and back, and some of the code in either B2 or B3 is evicted. The conservative algorithm in PLTO is a trivial extension to the algorithm described above—all blocks with a weight greater than some threshold, w, receive a frequency of 1. Simply put, we assume they are always competing for space in the cache. Experiments involving the threshold w suggest that its value does not matter much, but should be between .001% and .1% of the weight of the loop header. The threshold is used to exclude blocks with very small execution counts, which are probably not resident inside the cache for any meaningful duration.

4.5.3 Comparison

We have compared both models discussed in Sections 4.5.1 and 4.5.2 across the integer subset of the SPEC95 benchmark suite. Appendix A contains graphs for each benchmark; these show the speedup obtained as a function of how much inlining was carried out with each model. The x-axis, labeled "Degree of Inlining", corresponds to how profitable a call site needed to be in order for inlining to be carried out. As the number rises, the amount of inlining decreases. Each graph contains three lines: the blue corresponds to PLTO running with the inlining optimization turned off; it is independent of the "Degree of Inlining", so the lines are horizontal. The green line corresponds to the model discussed in Section 4.5.1, and the red line is the conservative model from Section 4.5.2. In most cases the two algorithms achieve similar results, and the curves are very close. We have concluded that either model can be used to make smart inlining decisions, and usually one can expect results better than those obtained when not carrying out inlining at all. Section 8 discusses future plans for path profiling and context-sensitive profiling, which would hopefully enable us to develop a more accurate model.

5 Optimizations

PLTO carries out numerous optimizations that make use of the analyses described in Section 4. Figure 8 is a high level flowchart of the optimization process. Unreachable code elimination is performed first, so that subsequent optimizations do not waste time improving code that cannot be executed. Constant propagation is then performed, followed by function inlining. If value profiling is being carried out it is done after inlining[16]; if value profiles are available, then value specialization is performed. Both value specialization and inlining create many opportunities for constant propagation that were missed during the initial phase, therefore an iterative phase of propagation, dead code elimination, and unreachable code elimination is carried out. Constant propagation generates dead code, which can create situations where some branches of execution are optimized away, which in turn affects the precision of constant propagation as there are fewer paths of execution leading into blocks. For this reason the optimizations are carried out a few times until one round of them produces no changes. Finally, a profile-guided code layout algorithm is used and the instructions are run through a scheduler. As depicted in Figure 8, peephole optimizations are invoked from many places in the optimizer.

The remainder of this section describes how the optimizations are carried out, and the effects that they have on performance. This section is organized as follows: Section 5.1 describes constant propagation as carried out in PLTO. Function inlining in PLTO is explained in Section 5.2. Section 5.3 explains a special form of specialization—intended for indirect jumps through tables of addresses—that is used in PLTO. Section 5.4 describes code layout, and Section 5.5 contains a few remarks about unreachable code elimination. Section 5.6 gives a glimpse into the peephole optimizations that are carried out.

5.1 Constant Propagation

Interprocedural constant propagation is perhaps the best motivating factor for doing optimization at link-time. Many opportunities exist—in both application and library code—to forward constant arguments across procedure and module boundaries. In the `alto` system it was shown that interprocedural constant propagation was an important source for performance improvements; on average the benchmarks were 10% faster as a result of using this optimization [21]. PLTO performs constant propagation on the general purpose registers and the runtime stack across the entire program. Propagation of PSW bits is also performed so that conditional jumps may be eliminated when the results of previous comparisons are known. In the `alto` system, constant propagation was performed by actually executing the instructions on the processor and observing the changes to the execution environment. PLTO adopts a different approach, and emulates the semantics of the machine code instructions in software much like a virtual machine would do. The IA-32 instruction set consists of over 300 instructions, but we have found that relatively few of these are used in practice, and it is sufficient to have the constant propagator know the semantics of about 40 frequently used instructions.[17] The few instructions that are not recognized by PLTO are treated conservatively. Our experiments show that being able to evaluate instructions that are not currently supported is probably not worth the implementation effort. Table 6 shows the effect of constant propagation in PLTO.

[16] We are investigating carrying out value profiling/specialization at other points.

[17] Static and dynamic distributions were gathered to discover which instructions were "important" enough to warrant writing evaluation functions.

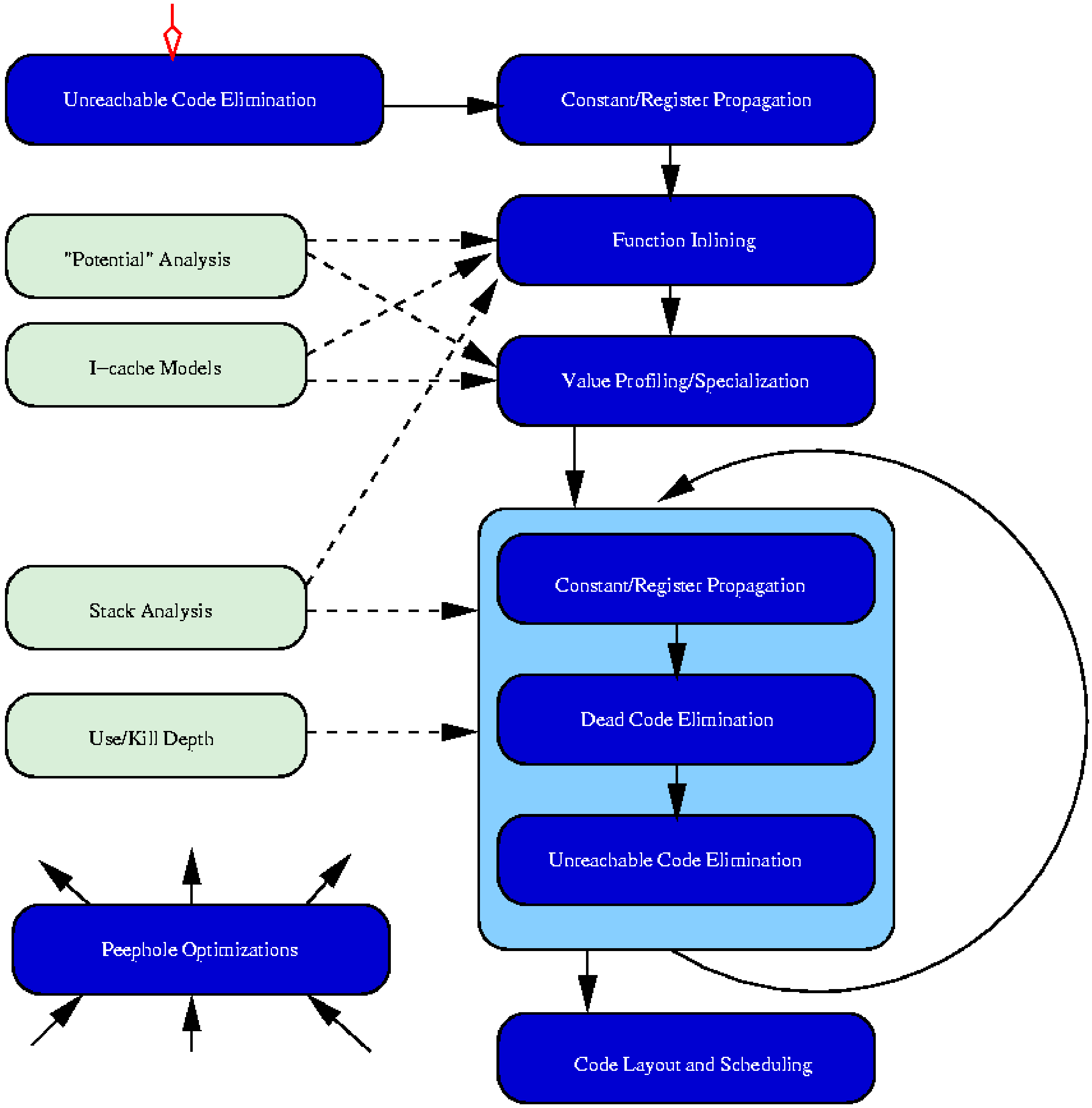

Figure 8: Flowchart for Major Optimizations in PLTO

5.1.1 Register Propagation

Load/store forwarding is an optimization that attempts to find multiple loads from the same location and replace them by register-to-register moves. The idea is to find two instructions, *I* and *J*, that load into registers from some memory location. One then attempts to prove that the memory location was unchanged between the execution of *I* and *J*, and that the register loaded by *I* still contains the value when control reaches *J*. For memory disambiguation rules, indirect memory references are assumed to overlap all other memory regions, and absolute memory references are assumed to overlap everything but the stack. The act of "propagating" the registers in place of memory locations is analogous

Program	*Execution Times*		Percent Speedup
	plto w/o propagation	*plto*	
compress	144.93	126.12	12.97
gcc	92.91	85.94	7.50
go	149.28	145.36	2.62
ijpeg	158.26	143.46	9.35
li	113.33	110.85	2.18
m88ksim	106.25	104.92	1.25
perl	83.01	80.14	3.45
vortex	171.33	149.64	12.65
Arithmetic Mean:			6.49

Table 6: Speedup: Constant Propagation

to propagating constants in place of registers or memory locations during constant propagation; so close, in fact, that PLTO carries out load/store forwarding concurrently with constant propagation. The idea is to keep track—inside the snapshots, using flags—of which stack locations and registers are aliasing each other at each program point. In effect, we propagate registers (hence the name "register propagation") around the CFG using the standard meet/join rules for constant propagation. The propagation automatically handles all the intricate details that load/store forwarding has to worry about, which are described in several papers [26, 21].

5.1.2 Analysis of Precision

An interesting question to ask is how much do indirect stores damage the effectiveness of our analyses? To answer the question, we have run experiments in which indirect stores are not treated as potentially overlapping any memory region. Table 12 in Appendix C shows the number of additional constants and registers that are propagated throughout the program under these assumptions. Other analyses and optimizations experience similar improvements. Dead code elimination, for instance, benefits a great deal when liveness analysis is more precise.

5.1.3 Representation Issues

The runtime stack introduces many extra challenges for optimizations. Conceptually, one can envision the stack as a very large register set, with 4-byte stack slots taking the place of general purpose registers. In PLTO, each basic block contains a "snapshot" of what the execution environment for the program looks like before control enters the block. In particular, it contains storage for the eight general purpose registers, the PSW and the stack among other things. When constant propagation is carried out for a block, the snapshots are cloned, the instructions in the block are evaluated, and the cloned snapshot is updated to reflect the changes in the state of the program. The cloned snapshots are then propagated to the successors of the block and later destroyed when they are no longer needed. Thus, a block first constructs its snapshot by merging the snapshots from all its predecessors using the standard meet/join rules for constant propagation. In PLTO, snapshots of the registers are simply an 8-slot array of 32-bit integers. For implementation ease, and to make the environment homogenous, the stack is also modeled using an array with a large upper bound. We chose the upper bound to be 10000 bytes, which allows for 99% of all stack frames in the integer subset of SPEC95 to be modeled. A simple analysis is carried out in each function to determine the maximum size of the stack at any program point within the function. Each basic block is then allocated an array large enough so that it can represent its own stack frame plus the stack frame of the largest predecessor in the call graph. It is important that a caller's stack frame be modeled inside the callee so that interprocedural constant propagation can take place. Given this representation, the instruction evaluator can treat stack locations analogously to registers when storing dataflow values in them. One down side is that stack references that exceed the upper bound cannot be evaluated. Large stack frames may be formed when a hot path of functions is inlined consecutively into one giant function, or when a procedure has large local arrays that result in a big stack frame allocation. Fortunately, we find that this limitation is

Program	*Execution Times*		Percent Speedup
	PLTO w/o inlining	PLTO	
compress	124.38	126.12	-1.39
gcc	86.39	85.94	0.52
go	144.34	145.36	-.70
ijpeg	143.38	143.46	-.06
li	112.78	110.85	1.71
m88ksim	109.21	104.92	3.92
perl	81.23	80.14	1.34
vortex	147.78	149.64	-1.26
Arithmetic Mean:			0.51

Table 7: Speedup: Function Inlining

not that restrictive in practice, and relatively few opportunities for propagation are missed.[18]

A superior—albeit harder to implement—representation would be to condense the array representation into a list with intervals. For instance, an array of 8 values: [$\perp$, $\perp$, c_1, c_2, $\perp$, $\perp$, $\perp$, $\perp$] could be condensed into a list with 4 elements: $\{$ ([0-1] : $\perp$), ([2] : c_1), ([3] : c_2), ([4-7] : $\perp$)$\}$. When procedures have large local arrays, and hence large stacks frames, the interval approach would be more storage efficient. Indirect stores into arrays can rarely be disambiguated, so slots in the stack frame that are allocated for an array often (if not always, in PLTO) contain $\perp$. The compact interval representation is superior in that it allows for unbounded stack frames at a fraction of the storage cost that arrays use. The down side is that implementation is tricky. Merging and updating lists is not a trivial matter. Eventually we would like to convert to a list-based scheme.

5.2 Function Inlining

Function inlining is an optimization in which a procedure call is replaced by an actual instance of the function. The goal is to eliminate the overhead associated with issuing call and return instructions and setting up a new stack frame, and to enable opportunities for constant propagation by reducing the number of calling contexts to one for the inlined procedure. However, not all the effects of inlining are good. The transformation is usually accompanied by code growth, and consequently can cause an increase in i-cache misses and page faults. A delicate balance between the two must be achieved in order to reap the full benefits of inlining. In `alto`, procedure merging was responsible for only a small improvement in overall program performance. We have had similar experiences in PLTO, but have seen many situations where inlining has been terribly detrimental to performance (e.g., 10-15% slowdowns). The sophisticated i-cache model described in Section 4.5, the analysis to compute optimization potential described in Section 4.4, and a large number of heuristics help to guide the optimization and to make smart inlining decisions. Appendix A shows how various degrees of inlining can affect program performance. The blue line corresponds to PLTO optimizing programs without inlining turned on, and the other two lines correspond to two different i-cache models that were used to guide inlining decisions. In most cases there is a significant difference—larger than that seen in a comparable system like `alto` —between the peaks for inlining and the non-inlining curve. This suggests there are potentially more payoffs if one can make intelligent inlining choices. Table 7 shows the performance gains (and degradations) seen in PLTO as a result of using inlining compared to not using it. For reference, Table 13 in Appendix D contains more information about how much inlining was carried out.

The transformation of inlining functions is implemented to produce much more efficient code. First the procedure is cloned and the `call` and `ret` instructions are eliminated from the caller and callee respectively. Instructions to set up the stack frame inside the body of the callee are also eliminated, and all references to the stack frame inside the function are written in terms of the caller's stack frame. The stack analysis provides the necessary pieces of information to make this work. If the height of the stack at the call site is h_i, then h_i can be added to all the stack frame references inside the callee. When the frame pointer is used directly (e.g., `mov %eax` $\leftarrow$ `%ebp`), PLTO inserts an instruction before it that restores the original value, and an instruction after the use that reverses the restoration. We

[18]Experiments show that the static number of stores and loads that use large offsets into the stack is small.

find that this happens very infrequently in practice, so it is unlikely to have any significant impact on performance. All procedure calls are initially candidates to be inlined; those that actually get picked fall into one of three categories:

1. [*Functions with one call site.*] Inlining a function that is statically called from only one place in the program can never result in code growth since the original function can be removed from the program. In addition, no extra i-cache misses can be incurred since the procedure was not shared by multiple functions or loops. Thus, procedures that only have one call site in the program are automatically inlined regardless of how profitable or unprofitable they may be.

2. [*"Small" functions.*] A small function, as defined in PLTO, is one with five or less instructions. Since inlining can usually eliminate three or four instructions, there is probably not any code growth associated with merging the procedure. Small functions are also automatically inlined, regardless of their execution frequency.

3. [*Profitable functions.*] A profitable candidate for inlining is a procedure where the expected payoff, in terms of the number of machine cycles saved, is greater than some user-defined threshold. The expected benefit is the sum of two numbers: (1) the direct savings from eliminating the `call`, `ret`, and the setup for the stack frame, which is computed by scaling the weight of the basic block containing the `call` by 3.0 – 5.0; (2) the indirect savings from enabling other optimization opportunities, as described in Section 4.4. We have experimented with different thresholds, and have found that requiring inlining to be able to eliminate the equivalent of 10–15 *hot* instructions is a good heuristic. We define a *hot* instructions to be ones that account for the top 80% of all instructions executed.[19] The term *hot instruction* really refers to the weight of the instruction with the smallest count that is still considered *hot*. If this weight is w_i, and 10 *hot* instructions are needed to be eliminatable for inlining to occur, the total expected savings must exceeded $10 \times w_i$. Appendix A presents graphs for various inlining models in which the threshold for the "number of eliminatable instructions needed in order to inline" was varied. Although some benchmarks benefit from excessive inlining (e.g., when the number of *hot* instructions is very low), the best overall performance occurs between 10 and 15.

5.3 Jump Table Specialization

Indirect jumps through tables of address, such as those arising from `switch` statements in C, are expensive at the machine code level. They are typically created by compilers to implement multiway branches and to avoid performing an excessive number of runtime tests. Unfortunately this flexibility comes at a cost. Branch prediction for such jumps is difficult due to the large set of possible control flow successors, and mispredicted branches cause the instruction pipeline to be flushed and hence result in long stalls. Jump table specialization is an optimization in which a test for the most common index into the jump table is inserted. A conditional jump to the target associated with the index is also added. The idea is that most of the time the high overhead associated with indirect jumps through tables can be avoided at the smaller expense of a test and conditional jump. Such a test can be inserted before the bounds check and index scaling for the jump table is done; since the bounds check alone requires a test and a conditional jump, the extra test inserted by the optimization is—in a sense—"free" when the common case is encountered. The down side is that all the non-common cases incur the extra cost of the test and conditional branch. The potential benefit for having the cheaper test is large when the common case is executed very frequently, such as 80%–90% of the time. Our experiences with this optimization on the IA-32 have been somewhat disappointing, however. We find that there are relatively few jump tables in the integer subset of the SPEC95 benchmarks in which the common case has a high frequency and thus is worth specializing.[20] In those which PLTO does specialize, there has been no noticeable performance improvement. We are currently investigating if these jumps are executed enough to make a difference, and trying to estimate what payoff we should expect to see.

5.4 Profile-Guided Code Layout

Pettis and Hansen describe an algorithm for the placement of basic blocks within an executable with the goal of reducing (1) the number of taken branches, (2) the number of misses in the instruction cache through code locality, and (3) the number of page faults, also through locality [22]. The algorithm has been used in a number of optimization systems [21, 29] and is also implemented in PLTO. The `alto` system found that a slight modification to the algorithm produced better results. The idea is to partition the set of basic blocks into three disjoint subsets: a *hot* set, a *cold*

[19] The 80% threshold is user-defined, but our experiments indicate it is a reasonable number to use.

[20] There are, on average, 3–4 per benchmark.

Program	*Execution Times*		Percent Speedup
	Base	PLTO w/ only layout	
compress	130.18	125.55	3.55
gcc	96.97	93.12	3.97
go	146.29	147.90	-1.10
ijpeg	144.73	158.77	-9.70
li	113.88	118.01	-3.62
m88ksim	128.76	116.51	9.51
perl	87.08	82.75	4.97
vortex	155.40	164.99	-6.17
Arithmetic Mean:			0.22

Table 8: Speedup: Code Layout

set, and the *zero* set. The *hot* set is for all blocks with an execution count greater than some user-defined threshold. Generally, a threshold that allows for 50-95% of the blocks to be considered *hot* is desirable. The *zero* set is reserved for blocks which were not executed at all during the program's execution with the training input. The other blocks which are neither *hot* nor *zero* end up in the *cold* set. The Pettis-Hansen algorithm is then applied to each set individually, disallowing control flow edges in which the source block is not in the same set as the destination block. The *hot* set is then laid out first, followed by the *cold* set, and finally the *zero* set at the end of the *.text* section. The motivation for doing this comes from not wanting *zero* or *cold* blocks to be in the same region as the frequently executed coded.

In PLTO we have experimented with several algorithms, including the two described above. The best performance improvements come from the straightfoward Pettis-Hansen greedy algorithm applied at the function level. Interprocedural edges are not considered, as they often cause the algorithm to create inefficient code fragments. For instance, a procedure `call` can be laid out directly before the function it calls, but the `call` cannot be eliminated due to its side effects. In addition, the *return-block* which previously followed the block containing the `call` can no longer be laid out directly after it. We have found that allowing interprocedural edges results in a high degree of procedure intermixing, which is somewhat undesirable from a performance standpoint. Table 8 shows the improvements that code layout has on speed. An interesting note is that the programs experience a staggering increase in the number of branch target buffer (BTB) misses. We have investigated the matter and have found that branches only find their way into the BTB after being taken for the first time. Since the Pettis-Hansen algorithm aims to reduce the number of taken branches, it often takes longer for them to appear in the BTB and there is an increase in misses. In the "worst" case, a branch is laid out so that it is never taken, and thus never appears in the BTB—causing a miss every time it is executed! We suspect that for this reason, the payoff from doing code layout is not as large on the IA-32 as what is seen in systems like `alto` . The fallout of missing in the BTB is having to make a static prediction, which is generally less accurate and based on simple heuristics. It is not clear from the IA-32 documentation that is available exactly how expensive the BTB misses are, especially when the branch is statically predicted correctly. Nevertheless, when static prediction fails and there is a mispredicted branch, there is a large cost associated with flushing the instruction pipeline. As a side experiment, we tested a variant of Pettis-Hansen that chooses the lowest edge weights first; these programs experienced a large decrease in the number of BTB misses, but overall they ran slower. In the future we would like to revisit the problem and perhaps develop a better solution that incorporates these effects in a cost-benefit model.

5.5 Unreachable Code Elimination

Unreachable code elimination has a number of desirable effects: it (1) improves the precision of dataflow analyses by reducing the number of execution paths leading into basic blocks, (2) creates a smaller executables, which can lead to better paging performance, and (3) reduces the time PLTO spends on performing analyses and optimizations, as there is less code to worry about. Unreachable code elimination is straightforward, and carried out in a similar manner as done in the `alto` system [21]. We notice that in most benchmarks about 10% of the code is unreachable. However, in a few—such as vortex and li—the improvements are about twice that. This is consistent with the work done in `alto`, but it is more than what was estimated in work done by Srivastava [27].

Program	*Execution Times*		Percent Speedup
	PLTO w/o Peephole	PLTO	
compress	119.42	126.12	-5.21
gcc	86.34	85.94	0.47
go	146.06	145.36	0.48
ijpeg	143.24	143.46	-0.15
li	110.76	110.85	-0.08
m88ksim	109.51	104.92	4.37
perl	80.95	80.14	1.01
vortex	151.70	149.64	1.36
Arithmetic Mean:			0.28

Table 9: Speedup: Peephole Optimizations

5.6 Peephole Optimizations

A number of peephole optimizations are performed to exploit some opportunities created by PLTO's optimizations. Table 9 shows the performance improvements seen from carrying out peephole optimizations. The transformations that achieve the best results are:

1. *Branch Trampolining.* If a conditional or unconditional jump leads directly to another unconditional jump, e.g., $j_i \rightarrow j_k \rightarrow j_m$, the middle jump can be eliminated by simply re-routing the target of the first jump to be the third: $j_i \rightarrow j_m$.

2. *Coalescing Math Operations.* PLTO assumes that the compiler did its job with regards to simplifying math expressions as much as possible, but some optimizations such as inlining introduce code sequences where consecutive instructions add or subtract from the same register. For example, two consecutive `add`'s to the stack frame often occur after inlining, since the inlined function has instructions to deallocate the stack frame, and the caller also deallocates the arguments right after. PLTO tries to coalesce adjacent `add` or `sub` instructions that store to the same register, e.g., (1) `add %eax` $\leftarrow$ `10` and (2) `add %eax` $\leftarrow$ `16` combine to form (1) `add %eax` $\leftarrow$ `26`.

3. *Effectless Instruction Elision.* Some instructions are not technically "dead" as defined by a liveness analysis, but they have no effect on the program's execution. An example is move from a register back to itself: `mov %eax` $\leftarrow$ `%eax`. Regardless of whether %eax is subsequently used, the instruction can be eliminated.

4. *Conditional Move (CMOV).* The IA-32 contains a conditional move instruction, `cmov`*cc* $r_i \leftarrow l$, which conditionally moves (based on the condition code *cc*, which may be something like $\leq$, $\neq$, etc.) either a register or a stack location to another register. If the condition is true, the move is executed. If not, the instruction has no effect. PLTO looks for situation where the effect of a branch is to jump over a move instruction, and converts three instructions—the comparison, the branch, and the move—to one conditional move. This is an architecture specific optimization that gcc did not carry out. It is quite effective on the *m88ksim* benchmark from SPEC95.

6 Experimental Results

6.1 SPECint-95 and SPECint-2000

The total speedup PLTO is able to achieve on the integer subsets of SPEC95 and SPEC2000 can be seen in Table 10. On average, we observe a speedup of 6.11% on the SPECint95 suite and 2.89% on the SPECint2000 suite. The tests were run on an otherwise unloaded Pentium III 550 megahertz SMP machine that was running Redhat Linux 7.2. Each benchmark was run five times; the highest and the lowest runs were discarded, and the remaining times were averaged to produce the numbers seen in the tables. We also used the *rabbit* tool to monitor the low level execution behavior of the program. Worthy of note is a reduction in the number of (1) memory operations by about 5%, (2) taken branches by about 74%, (3) mispredicted branches by around 12%, and (4) instruction fetches by about 5.5%.

Program	*Execution Times*		Percent Speedup
	Base	Optimized	
compress	130.18	126.12	3.11
gcc	96.97	85.94	11.37
go	146.29	145.36	0.64
ijpeg	144.73	143.46	0.88
li	113.88	110.85	2.66
m88ksim	128.76	104.92	18.51
perl	87.08	80.14	7.97
vortex	155.40	149.64	3.71
Arithmetic Mean:			6.11

(a) SPECint-95

Program	*Execution Times*		Percent Speedup
	Base	Optimized	
bzip2	1033.97	1061.25	-2.64
crafty	520.78	471.46	9.46
eon	1050.49	961.29	8.49
gcc	574.47	529.88	7.76
gzip	827.19	820.44	0.82
mcf	1820.75	1833.73	-0.71
parser	1271.90	1261.93	0.78
twolf	1897.29	1920.32	-1.21
vortex	915.94	864.08	5.66
vpr	959.80	954.84	0.52
Arithmetic Mean:			2.89

(b) SPECint-2000

Table 10: Total Speedup: SPECint95 and SPECint2000

The integer benchmarks contain many branches, and often many procedure calls as well. There are very few floating point instructions. The applications are representative of many non-scientific "real-world" applications. *compress* is an in-memory file compression program, *gcc* is the GNU C compiler (it emits SPARC assembly), *go* is a program for playing the game of "go", *ijpeg* is an in-memory image compression/decompression program, *li* is a lisp interpreter, *m88ksim* is a simulator for the Motorola 88100 processor, *perl* is a perl interpreter, and *vortex* is an object oriented database. The programs range in static executable size from less than 100,000 instructions (compress) to over 300,000 instructions (gcc). The mean basic block size is around 4 instructions for these programs.

6.2 Floating Point Benchmarks

We have tested PLTO on the floating point subset of the SPEC95 suite as well, but there is no speedup that is worthy of note. In particular, the difference between execution times for the original program and our optimized program was no more than 1% in the best case. These benchmarks have extremely large sequences of floating point instructions, and it is not uncommon for basic blocks to contain thousands of instructions. In one benchmark, a single basic block has over 10,000 floating point instructions! Since PLTO does not have any optimizations that are specific to floating point calculations, it is ineffective at improving the performance for these applications. In the future we would like to look at scheduling of floating point instructions to hide latencies and reduce the number of stalls in the floating point unit (FPU).

7 Related Work

Link-time optimization, binary rewriting, binary instrumentation, and whole-program optimization have been explored by a number of people. PLTO is tied closely to `alto`, a link-time optimizer for the Compaq Alpha that was very successful in demonstrating the room for improvement in the quality of compiler-generated code [21]. On average, `alto` was able to achieve an 18% performance improvement over the SPECint95 benchmarks, with some benchmarks experiencing a speedup of over 50%. Like PLTO, `alto` was fairly ineffective at improving the performance for the floating point subset of SPEC95—for the same reasons discussed in Section 6. OM [28] and Spike [9], two earlier optimization systems also targeting the Compaq Alpha, were able to achieve moderate speedup on the same benchmarks.

Etch is a system that focuses on instrumentation of IA-32 executables for gathering data about the program [24]. For instance, during execution of a program it can gather data about the behavior of that program in the instruction cache. It performs a profile-guided code layout transformation as its only optimization. It is not clear from this work if the optimization results in any performance improvements. Other such systems include NT-Atom and HiProf, which are aimed at instrumentation and analysis of IA-32 executables as well.

Also related to PLTO is UQBT (the University of Queensland Binary Translator), which is able to statically translate executables across different architectures. It targets, among other things, the IA-32, and has faced some of the same challenges that we describe in this work. The optimizations it performs are less aggressive than those carried out by PLTO, and it requires that machine code adhere to certain idioms. UQBT is unable to optimize or translate non-conforming routines that have been written in assembly by a programmer. It is also not clear how much performance improvement the system is able to achieve on standard benchmark suites. UQDBT is a related system that shifts the focus from static to dynamic translation. Again, many of the issues we deal with in PLTO have also been considered by this work.

There also has been much related work in function inlining and procedure cloning—an optimization we would like to explore in the future as an alternative to function inlining. Cooper *et al* discuss how procedure cloning can be used effectively [10]. As mentioned in Section 4.5, McFarling proposes a scheme for modeling the instruction cache behavior of a program in order to make intelligent inlining decisions [18]. Davidson and Holler address the issue of inlining creating bloated code which can be detrimental to a demand-paging system [11]. This work also discussed how inlining can cause degradations in performance, as we have discussed in Section 4.5. Ayers *et al* suggest an approach for aggressive inlining of functions; in some benchmarks over 1000 procedures are inlined [2]!

8 Future Work and Open Problems

8.1 Uses for Free Registers

Context-sensitive register liveness analysis finds between 1.5 and 2.0 free general purpose registers per basic block.[21] An obvious use of these registers is for the passing of function arguments. Arguments are usually pushed onto the stack by convention, but at link-time convention can be thrown out the window. If free registers are available before a procedure call is issued, there is no reason why the arguments cannot be put in registers instead of being pushed onto the stack. The loading of arguments into registers is a trivial transformation; more difficult is trying to ensure that subsequent loads inside the body of the callee are replaced by register-to-register moves. Indirect loads, which are assumed to come from any memory region including the stack, are an obstacle much like they are in a stack liveness analysis. Realistically we do not expect a compiler to produce indirect loads for function arguments that come from anywhere except the frame pointer. A clever programmer, however, could easily alias the frame pointer with another register and load from that. For this reason, one must ensure that all indirect loads from general purpose registers do not come from the stack before this optimization is carried out. We expect that this optimization would work best accompanied with the memory disambiguation analysis described in Section 8.2.

8.2 Memory Disambiguation: Insight into Indirect Loads and Stores

As discussed several times, indirect loads and stores make many analyses and optimizations less precise than we would like them to be. One solution we have considered is a memory region analysis, in which we try to prove that registers point into the stack, heap, or statically allocated data. Supplied with this information, optimizations like constant

[21] The stack and frame pointer are excluded from the definition of "general purpose" here. We consider only %eax, %ebx, %ecx, %edx, %esi, and %edi.

propagation would only have to set the entire stack to $\perp$ when indirect stores through registers that are known to point somewhere into the stack are found. If the register contains an address that resides in the heap or some other data region, we would correctly assume that it does not overlap with the stack. We envision a dataflow analysis in which information about regions is propagated across the ICFG through our stack and register snapshots. A logical extension to the current snapshots would be a means by which to model the heap. The best—and most precise—solution for discovering when registers are loaded with heap addresses would be to special case the return value from the `malloc` family of routines, or to require the user to specify which functions return pointers into the heap. We suspect that the majority of indirect loads and stores come from, and go to, either the heap or global arrays residing in data sections of the executable. The memory disambiguation analysis would limit the extent of "damage" they do, and produce more precise optimizations. Related work has been done by Debray *et al*, who have described a *mod-k* alias analysis used in the `alto` system [12]. The analysis is effective at narrowing the scope of loads and stores to smaller sets of targets.

8.3 Profiling

Currently PLTO carries out edge profiling, from which basic block weights are easily extracted. Some optimizations that carry out transformations on the ICFG often need to re-weight edges and blocks to reflect the changes they make. For instance, in inlining when a procedure is cloned and substituted in place of a function call, weights must be assigned to the cloned procedure. With only edge weights available, the logical approach is to retain the same edge and blocks weights that were present in the original function, and scale them by a factor s_f, where s_f corresponds to the percentage of total calls contributed to the callee by the calling procedure. Value specialization faces a similar problem when regions of code are duplicated. The re-weighting process is entirely probabilistic, however, and has no real justification other than that it is the best we can do given only edge weights. One can imagine a situation in which a function is called from two places an equal number of times, and has a conditional branch taken 100% of the time when called from one place and never taken when called from another. When inlined into either of the two call sites, the edge weights coming out of the basic block that contains the branch will be equal—which is incorrect. One possible solution would be to gather path profiles in addition to (or in place of) edge profiles [4, 5]. A path profile may enable one to re-weight the edges in a more intelligent manner, as they often carry information about calling contexts. A second approach would be to use context-sensitive profiling, in which edge profiles for each function are gathered for each calling context it has. For example, a function with 10 call sites would have 10 sets of edge profiles (one for each caller). The problem with context-sensitive profiling is that things quickly grow to be large even when profiling only 1 context; multiple contexts would probably be infeasible. In the future we would like to explore some of these options to find a better solution than what currently exists.

Profiling of parallel and distributed-memory applications is another area of future work. Currently, PLTO will instrument an executable which can then be run on concurrently on multiple processors to generate a set of execution profiles. There is no means by which these profiles can all be used as input, however. One approach would be to sum the edge weights in each profile to generate one large profile that may be representative of each program. Interesting issues in load-balancing arise from this solution, which are beyond the scope of this paper. An alternative would be to invoke PLTO n different times if there are n edge profiles available. This would result in an executable being generated for each processor. PLTO was designed to be used to optimize distributed-memory scientific applications, and we intend to pursue the challenging issues that these applications bring to the table.

8.4 Disassembly Revisited

As discussed in Section 3.3.5, the hybrid disassembly algorithm implemented in PLTO fails when either the linear sweep or the recursive traversal disagree about the results. One could imagine extending our approach to try and determine which algorithm was correct. For instance, if the function contains no indirect jumps and the algorithms produce different results, we can probably determine that the recursive traversal was correct. The only situations in which it fails are in those functions containing indirect jumps; Section 3.3.3 provides the details.

One could also imagine extending our algorithm to incorporate additional verification stages. Since it is not always possible to determine when a function has been incorrectly disassembled, additional heuristic-based algorithms could be implemented as extra verification steps. The idea would remain the same: if any of the algorithms suspects a problem, the function is problematic.

Section 3.3.5 mentions an interesting problem; that of updating the offsets in PC-relative functions that have not been disassembled. We raise the point that the length of these instructions may change as a result of updating the offsets, and if this happens then other PC-relative jumps in the function may no longer be correct. A solution to this problem is *branch-trampolining*, or in this case trampolining of procedure calls. If a function call in the executable, prior to being run through PLTO, had a displacement of k bytes, then we can be assured that the instruction length need not be increased if, after running through PLTO, the displacement changes to a number less than k. It may require less bytes, but nothing stops us from using an inefficient representation with the same number of bytes as the original. Knowing this, it is safe—in that the instruction length for the procedure call will not change—to redirect a `call`, I_c, to a new target instruction, I_t—called the trampoline—residing directly after the procedure that was not disassembled.[22] The original displacement was k bytes, and we know that the instruction being targeted resides right after the procedure, so this displacement is some number less than or equal to k. The instruction, I_t, which we insert is an unconditional jump to the original target of the function call, I_f. Thus, the flow of control changes from $I_c \rightarrow I_f$ to $I_c \rightarrow I_t \rightarrow I_f$. The purpose of the trampoline, I_t, is to allow the function call, I_c, to target any location in the executable without having the length of its encoding change. It should be noted that an additional *return-trampoline* is also necessary, to redirect control back to the instruction following I_c; the idea is analogous and not described here.

Trampolining lends way to another problem. Multiple trampolines sitting at the end of a procedure may cause some procedure call, I_c, to require a displacement that is greater than k, its original offset. We propose a solution that uses a one master trampoline, I_M, that jumps to a second set of trampolines that performs the functions described in the previous paragraph. Using this implementation, we can be sure that all procedure calls will still use offsets of less than k bytes. Should we discover cases where our the length of encoding for procedure calls does change, we will likely implement this solution.

9 Conclusions

Post link-time optimization of executable programs can be a useful process to undergo when one wishes to squeeze the most performance of a program. Unfortunately, not all programs will experience mind-blowing speedup. Large applications—such as *vortex*, *gcc*, and *eon* from the SPEC95 and SPEC2000 suites—tend to see the most speedup, since optimizations like function inlining and code layout are effective for bigger programs. Floating-point intensive benchmarks are on the opposite end of the spectrum. They are very hard to optimize due to relatively few dynamic branches and procedure calls. In addition, much of the time is spent issuing floating point instructions.

Our system is not able to achieve results that are comparable to the `alto` system, which is closely related to PLTO . The IA-32, and CISC architectures in general, offer up many challenges—not addressed by systems for RISC architectures—that must be tackled. The reliance on using of memory is a particularly difficult thing to deal with, and often lends way to complicated analyses and optimizations. The potential payoffs, however, are more fruitful as the processor-memory speed gap widens and these payoffs are amplified even more. Every analysis in PLTO is conservative, in that they expect nasty features from the machine code. Consequently, the results of these analyses are almost always less precise than we would like them to be. There is much future work to be done in this arena, particularly with improving the precision of optimizations like constant propagation, register propagation, and liveness analysis.

10 Acknowledgements

This research was supported by the National Science Foundation through grants ACR-9720738 and CCR-0113633. Special thanks to Professor Saumya Debray and Professor Gregory Andrews, who were especially helpful in assisting with the writing of this paper and the research behind it.

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 1997 Conference on Programming Language Design and Implementation (PLDI)*, pp. 134–145, June 1997.

[22] It may need to be inserted before the procedure if the offset for the procedure call is negative—that is, it calls something at a lower address in the executable.

[3] J. E. Ball, “Predicting the Effects of Optimization on a Procedure Body” *Proc. SIGPLAN Symposium on Compiler Construction*, ACM SIGPLAN Notices 14(8), pp. 214–220, August 1979.

[4] T. Ball and J. Larus, “Efficient Path Profiling” *Proc. of the 29th International Symposium on Microarchitecture*, pp. 46–57, Dec. 1996.

[5] T. Ball, P. Mataga, and S. Sagiv, “Edge Profiling versus Path Profiling: The Showdown”, *Proc. of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 134–148, Jan 1998.

[6] C. Cifuentes and K. J. Gough, “Decompilation of Binary Programs”, *Software—Practice and Experience* 25(9), 1995.

[7] 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.

[8] C. Cifuentes and M. Van Emerick. “Recovery of Jump Table Case Statements from Binary Code,” *Science of Computer Programming*, to appear.

[9] 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.

[10] K. Cooper, M. Hall, and M. Kennedy, “Procedure Cloning”, *Proc. 1992 IEEE Conference on Computer Languages*, pp. 96–105, April 1992.

[11] J. Davidson and A. Holler, “Subprogram Inlining: A study of its effects on program execution time”, *IEEE Transactions on Software Engineering*, vol. 18, pp. 89–101, Feb. 1992.

[12] S. Debray, R. Muth, and M. Weippert, “Alias Analysis of Executable Code”, *Proc. 1998 ACM Symposium on Principles of Programming Languages (POPL-98)*, Jan. 1998, pp.12-24.

[13] GNU Project – Free Software Foundation, `objdump`, *GNU Manuals Online*, `http://www.gnu.org/manual/binutils-2.10.1/html_chapter/binutils_4.html`.

[14] 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.

[15] 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/index.html`

[16] J. Larus and E. Schnarr, “EEL: Machine-Independent Executable Editing”, *Proc. SIGPLAN ’95 Conference on Programming Language Design and Implementation (PLDI)*, pp. 291–300, June 1995.

[17] J. R. Levine, *Linkers and Loaders*, Morgan Kaufman, 2000.

[18] S. McFarling “Procedure Merging with Instruction Caches”, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM SIGPLAN Notices 26(6), pp. 71–91.

[19] 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`.

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

[21] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, “`alto` : A Link-Time Optimizer for the Compaq Alpha”, *Software Practice and Experience* **31**:67–101, Jan. 2001.

[22] 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.

[23] J. Powers, “Java Language Security”, `http://www.cs.may.ie/~jpower/Courses/se209/jvm/jvm-security.pdf`

[24] 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).

[25] B. Schwarz, S. Debray, and G. Andrews, “Disassembly of Executable Code Revisited”, (in submission: 2002 Workshop on Reverse Engineering).

[26] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, “PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture” *Proc. 2001 Workshop on Binary Translation*.

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

[28] 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.

[29] A. Tamches and B. Miller, “Dynamic Kernel I-Cache Optimization” *Proc. 2001 Workshop on Binary Translation*.

[30] H. Theiling, “Extracting Safe and Precise Control Flow from Binaries”,

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

[32] 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.

Appendix A Graphs for Inlining Models in SPECint95

Following are eight graphs, one for each integer benchmark in the SPEC95 suite. The graphs show the performance of McFarling's i-cache model and that of a more conservative model, both described in Section 4.5. In most cases the two models exhibit similar performance characteristics. The red line is the conservative model, the green line is McFarling's i-cache model, and the blue line shows the speedup when not carrying out any inlining at all. The x-axis is labeled the "Degree of Inlining", which refers to how profitable an inlining opportunity must be in order for procedure merging to be carried out. As the degree of inlining increases, the number of functions being inlined decreases.

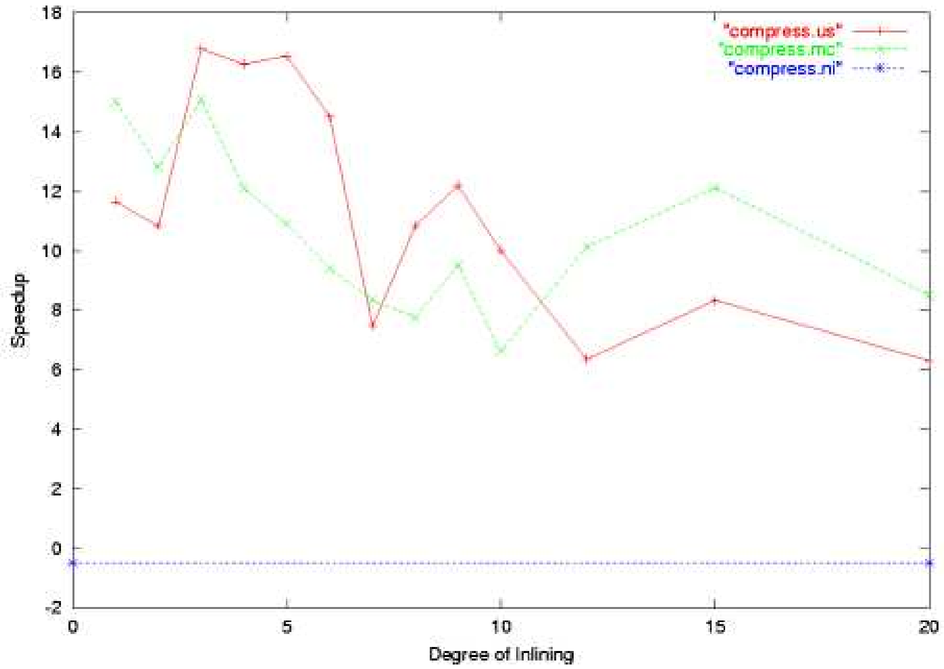

Figure 9: Inlining Models for compress95

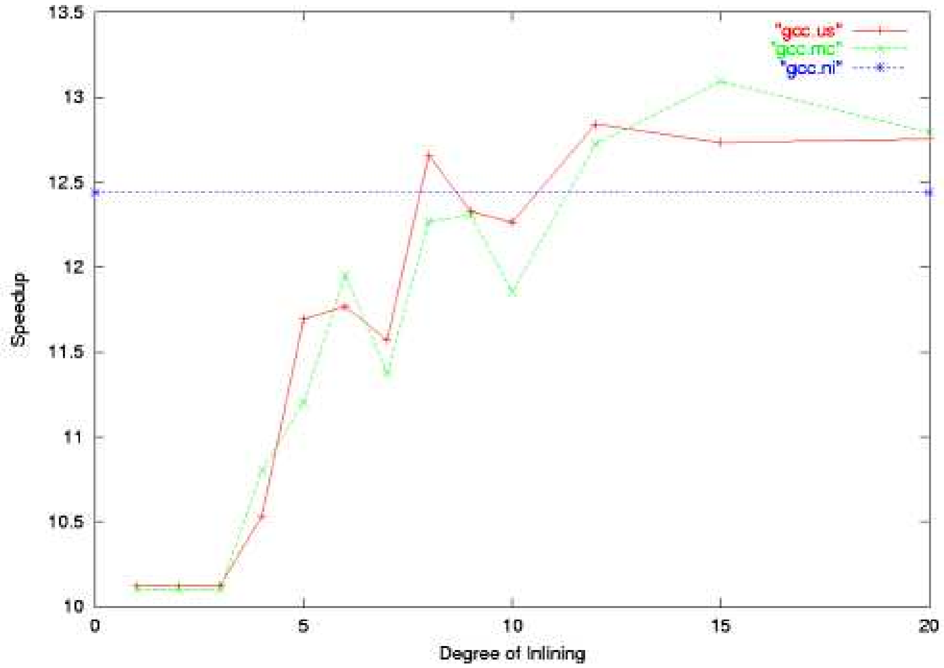

Figure 10: Inlining Models for gcc

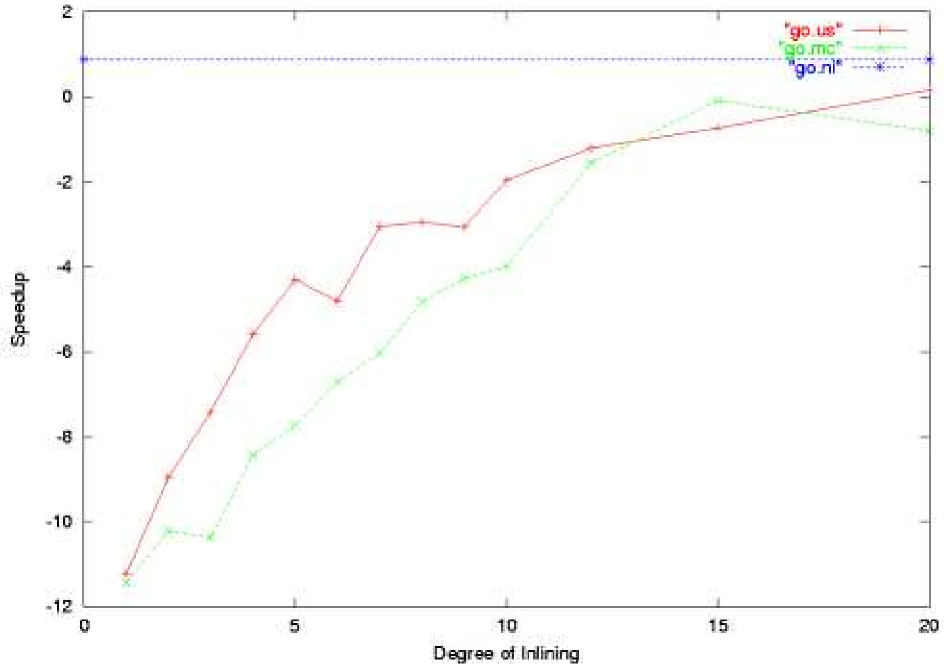

Figure 11: Inlining Models for go

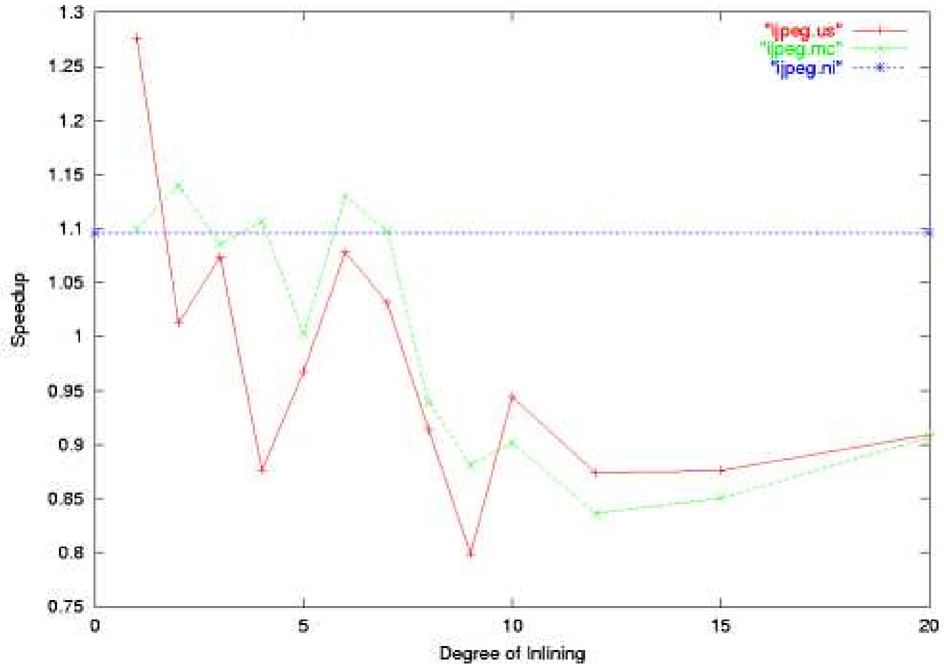

Figure 12: Inlining Models for ijpeg

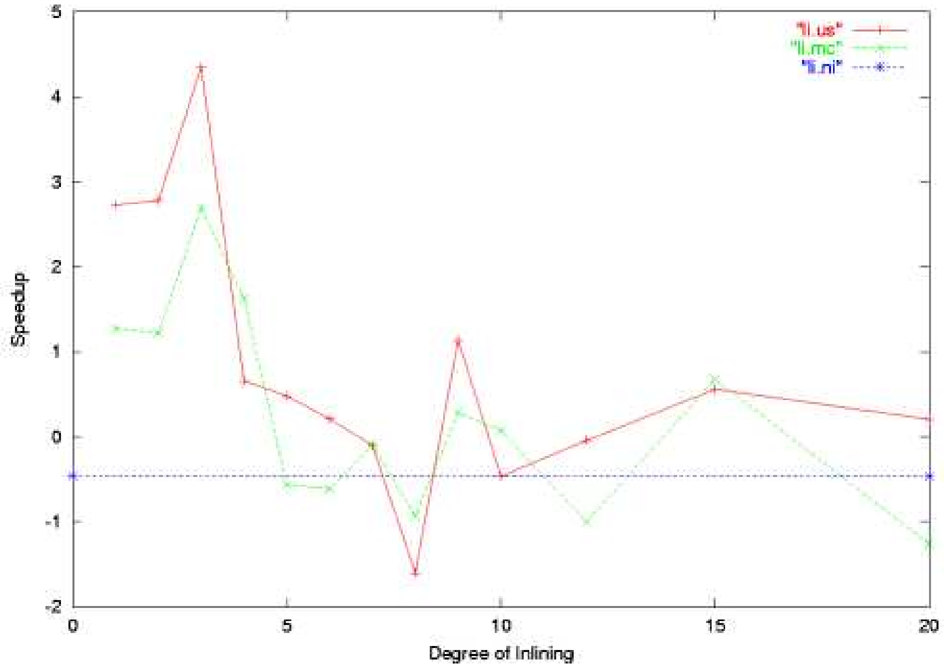

Figure 13: Inlining Models for li

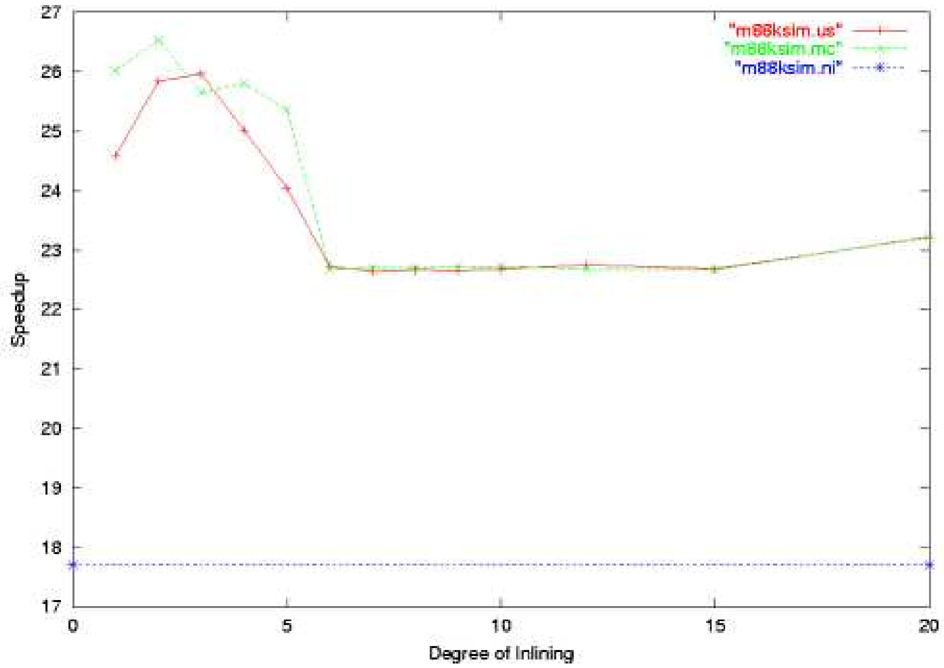

Figure 14: Inlining Models for m88ksim

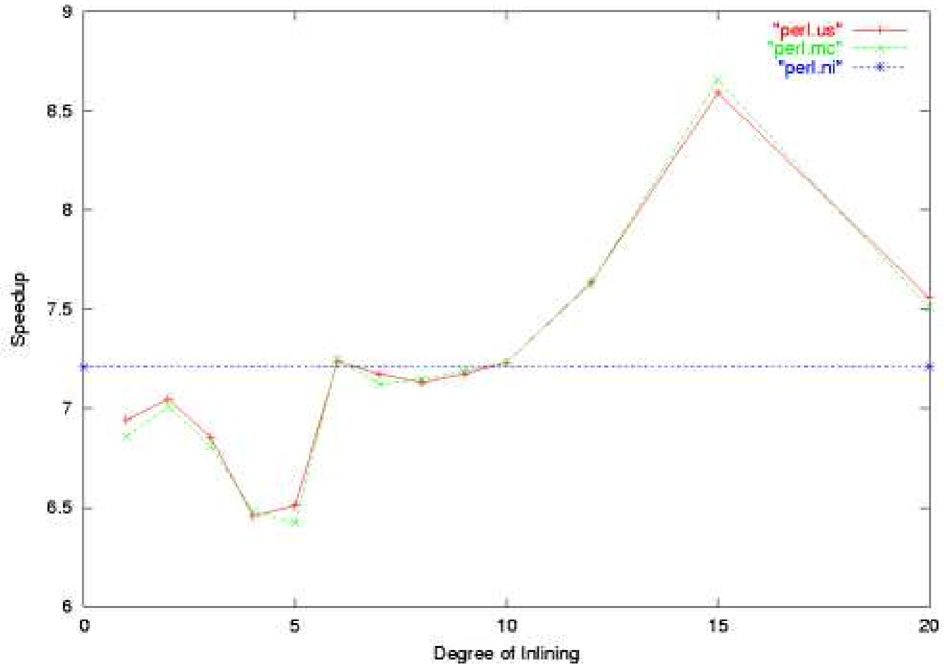

Figure 15: Inlining Models for perl

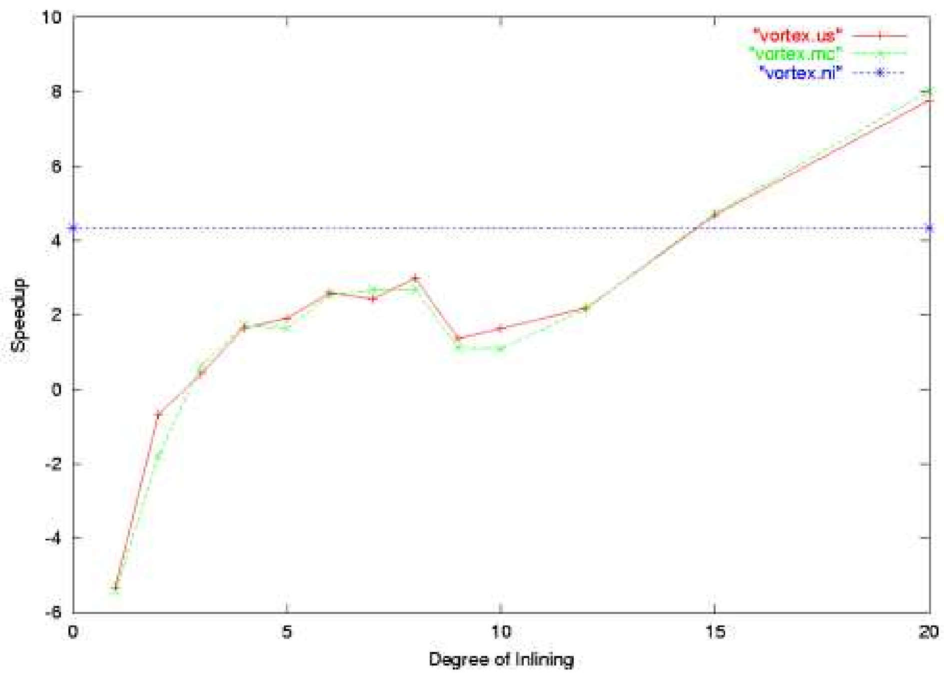

Figure 16: Inlining Models for vortex

B Disassembly Speeds with Linear, Recursive, and Hybrid

Program	*Disassembly Time* (sec)			T_{Hybrid}/T_{Linear}	$T_{Hybrid}/T_{Recursive}$
	T_{Linear}	$T_{Recursive}$	T_{Hybrid}		
compress	1.16	1.02	2.06	1.78	2.02
gcc	10.63	7.47	16.4	1.54	2.20
go	2.64	2.16	4.40	1.67	2.04
ijpeg	1.87	1.54	3.10	1.66	2.01
li	1.61	1.34	2.67	1.66	1.99
m88ksim	1.96	1.63	3.29	1.68	2.02
perl	2.84	2.32	4.73	1.66	2.04
vortex	4.40	3.24	7.07	1.61	2.18
GEOMETRIC MEAN:				1.66	2.06

(a) SPECint-95

Program	*Disassembly Time* (sec)			T_{Hybrid}/T_{Linear}	$T_{Hybrid}/T_{Recursive}$
	T_{Linear}	$T_{Recursive}$	T_{Hybrid}		
bzip2	1.44	1.18	2.45	1.70	2.08
crafty	2.32	1.88	3.82	1.65	2.03
eon	5.71	4.19	9.28	1.62	2.22
gcc	14.59	10.82	23.94	1.64	2.21
gzip	1.45	1.19	2.41	1.66	2.02
mcf	1.18	1.00	1.98	1.68	1.98
parser	1.71	1.38	2.83	1.66	2.05
twolf	2.10	1.73	3.52	1.68	2.04
vortex	3.91	2.87	6.28	1.61	2.19
vpr	1.72	1.46	2.91	1.69	1.99
GEOMETRIC MEAN:				1.66	2.08

(b) SPECint-2000

Key:
T_{Linear}: Disassembly time using the extended linear sweep algorithm
$T_{Recursive}$: Disassembly time using recursive traversal
T_{Hybrid}: Disassembly time using the hybrid algorithm

Table 11: Performance: Disassembly Speed

C Potential Benefit for Disambiguating Indirect Loads and Stores

Program	*Constants Propagated*		Percent Increase
	Conservative PLTO	Non-conservative PLTO	
compress	389	412	5.91
gcc	593	659	11.13
go	478	595	24.47
ijpeg	439	460	4.78
li	397	418	5.29
m88ksim	433	465	7.39
perl	640	660	3.13
vortex	701	752	7.27
Arithmetic Mean:			8.67

(a) Constant Propagation in SPECint95

Program	*Registers Propagated*		Percent Increase
	Conservative PLTO	Non-conservative PLTO	
compress	622	747	20.01
gcc	1604	1833	14.28
go	881	1031	17.03
ijpeg	720	862	19.72
li	650	781	20.15
m88ksim	786	929	18.19
perl	1132	1315	16.17
vortex	1039	1237	19.06
Arithmetic Mean:			18.08

(b) Register Propagation in SPECint95

Table 12: Effect of Conservative Treatment of Indirect Loads and Stores

D Inlining: Further Details

Program	*Function Calls*		Percent Reduction
	Before Inlining	After Inlining	
compress	922927	888865	3.69
gcc	17699155	16762064	5.29
go	6014400	4133272	31.27
ijpeg	38619336	36458229	5.59
li	5079246	4840814	4.69
m88ksim	352937599	153590549	56.48
perl	779525	682698	12.42
vortex	51675029	15767226	69.48
Arithmetic Mean:			23.61

Table 13: Function Inlining Statistics