Stack Analysis of x86 Executables *

Cullen Linn[1], Saumya Debray[1], Gregory Andrews[1], and Benjamin Schwarz[2]

[1] Department of Computer Science
University of Arizona
Tucson, AZ 85721
{linnc, debray, greg}@cs.arizona.edu

[2] Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
bschwarz@eecs.berkeley.edu

Abstract. Binary rewriting is becoming increasingly popular for a variety of low-level code manipulation purposes. One of the difficulties encountered in this context is that machine-language programs typically have much less semantic information compared to source code, which makes it harder to reason about the program's runtime behavior. This problem is especially acute in the widely used Intel x86 architecture, where the paucity of registers often makes it necessary to store values on the runtime stack. The use of memory in this manner affects many analyses and optimizations because of the possibility of indirect memory references, which are difficult to reason about. This paper describes a simple analysis of some basic aspects of the way in which programs manipulate the runtime stack. The information so obtained can be very helpful in enhancing and improving a variety of other dataflow analyses that reason about and manipulate values stored on the runtime stack. Experiments indicate that the analyses are efficient and useful for improving optimizations that need to reason about the runtime stack.

1 Introduction

Binary rewriting is being increasingly used for a variety of low-level code manipulation purposes, including instrumentation [5, 6, 12, 16], code optimization [2, 9, 17, 18], code compression [3, 4], and software security [7, 11, 19]. Among the advantages of binary rewriting, compared to traditional compile-time code manipulation, are that the availability of source code is not necessary, making it possible to process proprietary and third-party software; there is no need to rely on any particular compiler (and, therefore, any specific programming language supported by such a compiler); and the entire program, potentially including all library routines, is available for analysis and optimization. However, binary rewriting has its own problems. For example, much of the semantic information present in source code is lost by the time it has been transformed

* The work of B. Schwarz was carried out while the author was at the University of Arizona, Tucson. This work was supported in part by the National Science Foundation under grants EIA-0080123 and CCR-0113633.

to machine code, making it much more difficult to discover control flow or data flow information. Moreover, machine code is often rife with features that make analysis difficult, such as nontrivial pointer arithmetic and non-standard control flow behaviors, e.g., (conditional or unconditional) branches that go from the middle of one function into the middle of another, instead of the usual call/return mechanism for inter-procedural control flow (this is common in many library routines).

A result of such loss of semantic information at the machine code level is that good program analyses become even more important for the manipulation of programs. Memory references pose a significant problem in this regard, due to the issues of pointer aliasing and indirect memory references (it is known, for example, that alias analysis in the presence of multi-level pointers is complete for deterministic exponential time [8]). The problem is especially acute for the widely used Intel x86 architecture, because of a dearth of machine registers—there are six general-purpose registers available for use by the compiler—which forces the compiler to store values in memory when there are no registers available.

As a simple example, suppose we have a value that is in a register r in a RISC processor with many registers. If we wish to know whether this value is overwritten due to a call to a function f, and therefore has to be recomputed, it suffices to examine the registers overwritten by f and all functions called by f via a straightforward linear time analysis. On a register-poor architecture, however, the value is stored in memory (typically in the runtime stack), and in this case determining whether or not it has to be recomputed involves reasoning about the memory behavior of f and the functions it calls, which is a significantly more complex problem. For example, if f calls g and g writes to the stack, then we need to know whether such writes might affect the values within f's stack frame, which in turn requires knowing how large g's stack frame is and how far down in the stack g's `store` operations might reach.

A second problem that arises is that—unlike in RISC processors, where function arguments are typically passed in registers—on the x86 architecture, parameter passing is done via the stack. This makes tracking values across function boundaries significantly more difficult. The problem can be illustrated by the following simple example:

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

At the machine code level, the code for these functions has the following form:

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

```
   ...

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

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

1. ℓ is the same as that referenced by the instruction '`movl 8(%ebp), %eax`' in `g()`, in order to propagate the value of the argument into the body of `g()`.
2. ℓ is not overwritten by any prior store operations within `g()`.
3. ℓ becomes dead once all references to it in the body of `g()` have been replaced by the constant value of the argument.

To make these inferences, we have to be able to determine the position of the location ℓ addressed by the instruction '`push $456`' relative to both the "old" frame pointer in `f()` as well as the "new" frame pointer in `g()` and to reason about the liveness of specific memory locations within the stack frame of `f()` after inlining the call to `g()`.

As this discussion suggests, in order to reason about the low level behavior of programs on the x86 architecture, it is important to be able to determine how the runtime stack is used: which stack locations may be overwritten (or can be guaranteed to not be overwritten) by a function call; which stack locations may be live at a given program point; how stack references at one point in a program correspond to stack references elsewhere; and so on. Without such information, many analyses and optimizations are forced to treat stack-allocated variables conservatively, potentially reducing their impact considerably. This paper describes analyses we use within the PLTO post-link-time optimizer [13] to obtain basic information about the way in which programs manipulate the stack. The information so obtained can be very helpful in enhancing and improving a variety of other dataflow analyses that reason about and manipulate values stored on the runtime stack. Experiments indicate that the analyses are efficient and useful for improving such analyses and optimizations.

2 System Overview

The PLTO binary rewriting system consists of a front end for reading in executables, modules for code transformations, and a back end for emitting machine code. At present PLTO optimizes x86 executables, in the Executable and Linkable Format (ELF), under RedHat Linux.

PLTO begins processing an executable by disassembling each executable section of the binary [1, 14]. Once disassembly is complete, PLTO constructs an interprocedural control flow graph (ICFG) for the program. Several issues complicate the construction of the ICFG: indirect calls, indirect jumps through tables, and data appearing in segments, such as `.text`, that are typically reserved for instructions. The targets of indirect jumps through jump tables are identified using specific usage patterns involving relocation entries [14]. Control transfers whose targets cannot be resolved, namely, indirect function calls as well as indirect jumps that cannot be resolved as above, are modeled using special pseudo-nodes in the ICFG: $\mathtt{B}_\perp$, a pseudo-block belonging to the pseudo-function $\mathtt{F}_\perp$. These pseudo-nodes are used to represent worst-case scenarios, e.g., use all registers, define all registers, and possibly write to all possible (writable) memory locations, possibly overwriting data in the stack frames of any callers. Their use ensures that all analyses and optimizations performed by PLTO are conservative.

The construction of the ICFG is followed by various program analyses and code optimizations, e.g., dead and unreachable code elimination, constant folding, and load forwarding. After this, instruction scheduling and profile-guided code layout [10] are carried out. Finally, relocation information is used to update addresses appropriately, and the binary is written out.

3 Frame Size Analysis

In order to reason about the stack behavior of a function, we have to be able to model the stack frame of that function. One straightforward way to do this is as an array of words; subsequent analyses then reason about the contents, liveness, etc., of locations within this array. For such a model to be feasible, however, we have to first determine the (maximum) size of a function's stack frame.

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

A function f is said to be *well-behaved* if there is no net change in the height of the runtime stack due to the execution of f (expect for popping off the return address that the caller pushed on the stack), for all possible execution paths through f. Well-behavedness analysis is done in two phases. First, we mark as well-behaved all those functions that have standard function prologue and epilogue combinations that ensure that the height of the stack at function exit is the same as that at its entry; this involves a simple comparison against a small set of known instruction sequences for function prologues and epilogues. Second, as described below, we propagate information about changes in the height of the runtime stack due to the execution of each basic block in the program. This allows us to additionally identify other functions that are well-behaved.

Given information about well-behavedness of functions, we analyze each function to determine the (maximum) size of its stack frame (including the space for actual parameters, which is shared with the caller). The stack frame size of a function f is defined to be the maximum height of the stack, over all points in all basic blocks in the function, relative to that at the entry to f. To determine this, we first compute, for each basic block in the function, the change in the stack size due to the execution of that block. This is then propagated iteratively through the control flow graph of the function until a fixpoint is attained.

More formally, the analysis can be specified as follows. Given a basic block B, let $\mathsf{IN}(B)$ and $\mathsf{OUT}(B)$ denote the height of the runtime stack at the entry to, and exit from, the basic block B, relative to that at the entry to the function containing B, and *addrsz* denote the size of an address. We can write the dataflow equations for this analysis as follows:

- To compute $\mathsf{IN}(B)$, we have the following cases:
  1. If B is the entry block of the function, then $\mathsf{IN}(B) = 0$.
  2. Otherwise, if B is a return block, i.e., a block to which control returns from a function call block B', where the callee is a function f, then

     $$\mathsf{IN}(B) = \begin{cases} \mathsf{OUT}(B') - \mathit{addrsz} & \text{if } f \text{ is well-behaved;} \\ \\ \perp & \text{otherwise.} \end{cases}$$

     The reason for subtracting *addrsz* here is that the return address, which had been pushed on the stack by the `call` instruction to f, gets popped off the stack by the '`ret`' instruction in the callee.
  3. Otherwise, if B is neither the entry block nor a return block, then:

     $$\mathsf{IN}(B) = \bigwedge\{\mathsf{OUT}(B') \mid B' \text{ is a predecessor of } B\};$$

     where $\bigwedge$ is the meet operator over the flat lattice of integers, as in the case of constant propagation:

     $$x \wedge y = \begin{cases} x & \text{if } x = y; \\ \\ \perp & \text{otherwise.} \end{cases}$$

– To compute $\mathsf{OUT}(B)$, the most interesting case is when B is an exit block of the function containing a standard epilogue that matches the prologue. In this case, the effect of executing B is to restore the stack and frame pointers to their values at entry to the function, and then pop the return address off the stack while transferring control back to the caller (via a '`ret`' instruction). Thus, the net height of the stack at the end of the block, relative to that at entry to the function, is $-addrsz$, because the return address, which was on top of the stack at the function entry, now gets popped off. In general, we have the following cases:
  1. If B is an exit block, then:

$$\mathsf{OUT}(B) = \begin{cases} -addrsz & \text{if } B \text{ contains a standard epilogue that matches the} \\ & \text{prologue in the entry block of the function;} \\ \perp & \text{otherwise.} \end{cases}$$

  2. Otherwise, $\mathsf{OUT}(B) = \mathsf{IN}(B) + \delta_B$, where δ_B denotes the net change in stack height due to the instructions in B, and the addition is strict, i.e., $\perp + x = x + \perp = \perp$.

The analysis can be illustrated using the example shown in Figure 1. This is the control flow graph for the function `xor()` in the SPECint-95 benchmark program *li*, a Lisp interpreter; it was generated using the *gcc* compiler at optimization level `-O3`. Notice that over half the instructions—16 out of 31—use the stack, either by pushing or popping values, or by accessing a value in the runtime stack as an operand. Notice also that the stack pointer register `%esp`, which points to the top of the stack, is changed in several places, both explicitly (e.g., via `add` and `sub` operations, e.g., at instructions 15, 19, and 22) and implicitly (via `push`, and `pop` operations). This makes the problem of keeping track of the top of the stack, relative to the frame pointer `%ebp`, nontrivial.

The frame size analysis proceeds as follows:

1. The functions in the program are examined to see which can be identified as well-behaved. Assume that the functions `xlsave()` and `xlevarg()` are identified as well-behaved; the function `xor`, shown in Figure 1, itself has one of the standard prologue/epilogue combinations that PLTO recognizes, and is marked as well-behaved.
2. Each basic block is analyzed to identify the net change in stack height due to its instructions. For example, in block `B0`, we find six `pushl` instructions, each of which pushes 4 bytes on the stack; an explicit allocation of 20 bytes on the stack (instruction 6),[1] and a `call` instruction, which pushes the return address (4 bytes) on the stack; the total change in stack height is thus 48 bytes. At the end of this phase, we have the following net changes to stack height inferred for the various basic blocks:

[1] The runtime stack grows downwards, from high addresses towards low addresses. For this reason, `sub` instructions (e.g., instructions 6, 19) allocate space on the stack, while `add` instructions (e.g., instructions 15, 22) deallocate space.

Basic block	*effect on stack* (bytes)
B0	+48
B1	−16
B2	0
B3	+20
B4	−16
B5	?

The reason we can't compute a net change to the stack for block B5 is that instruction 24 sets the value of the stack top pointer `%esp` to the value $val(\%\texttt{ebp}) - 12$, where $val(\%\texttt{ebp})$ denotes the value of the frame pointer register `%ebp`, which means that the net change in the height of the stack due to block B5 depends on the value of register `%ebp` at the entry to B5.

3. We now propagate the stack height changes to determine, for each basic block, the height of the stack at its entry and exit, relative to that at entry to the function. First, OUT(B0) is computed as +48. Given that `xlsave()` is well-behaved, and therefore has no net effect on the stack except to pop off the return address, we therefore have $\mathsf{IN}(B1) = \mathsf{OUT}(B0) = 48 - 4 = 44$. Since block B1 effects a net change of -16 in stack height, we get $\mathsf{OUT}(\mathrm{B1}) = 44 - 16 = 28$. Proceeding in this way, we get the following:

Basic block	*Relative Stack Height* (bytes)	
(B)	IN(B)	OUT(B)
B0	0	+48
B1	+44	+28
B2	+28	+28
B3	+28	+48
B4	+44	+28
B5	+28	−4

Two of these values are of particular interest. The value of IN(B2) is computed twice: first when only OUT(B1) has been determined, and again when the OUT sets of both of its predecessors—namely, B1 and B4—have been determined, to ensure that it is the same for both predecessors (which, in this case, it is). The value of OUT(B5) is computed to be -4, even though the actual change in stack height due to B5 is unspecified (see above), because B5 ends with a standard epilogue, which means that the stack height after all its instructions have been executed is 4 bytes below that at entry to the function, as discussed above.

4. Finally, using the values of IN(B) and OUT(B) for each block B, we determine the maximum stack height, relative to that at entry to the function, at each point within each block. In this case, this value is computed as +48.

Thus, we conclude that for this function, the stack frame size is 48 bytes. Notice that this is quite different from the 20 bytes of storage explicitly allocated on entry to the function in block B0 (instruction 6), because it also takes into account space allocated on the stack via other instructions elsewhere in the function.

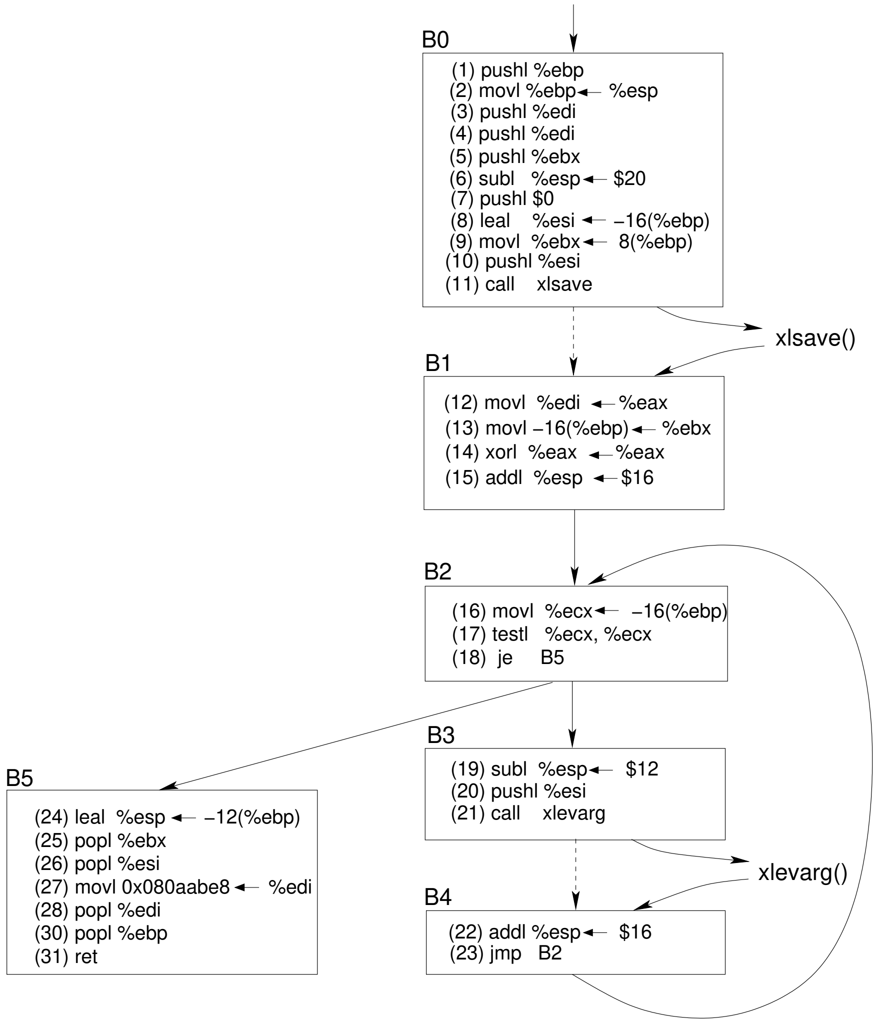

Fig. 1. An example control flow graph (function `xor()`, from the SPEC-95 benchmark *li*)

4 Use-Depth and Kill-Depth Analysis

The relatively small number of compiler-visible general purpose registers in the x86 architecture often causes values to be placed in (or spilled to) a function's stack frame. In the absence of any other information, program analyses must make worst-case assumptions about the effects of function calls on values kept in the stack. For example, constant propagation must assume that a function call can destroy all such values, because a function might write to any memory location, while stack liveness analysis must assume that stack locations are live because they may be accessed by a function call. Such worst-case assumptions can affect the precision of our analyses quite significantly.

To address this, we use *use depth* and *kill depth* analyses to estimate the effect of function calls on the runtime stack. The *use depth* of a function is either a non-negative integer or the value ∞; it represents an upper bound on the depth in the stack, relative to the top of stack when the function is called, from which the function may read a value. The *kill depth* of a function is analogous to that of use depth: it is either a non-negative integer or the value ∞, and represents an upper bound on the depth in the stack, relative to the top of the stack when the function is called, to which that function may write a value.

4.1 The Basic Analyses

The psuedo-function $\mathsf{F}_\perp$, which is used to model indirect function calls, is assumed to have a use depth of ∞. The use depth of the other functions in the program are computed in two phases:

1. [*Local analysis.*] The instructions in each function are examined to determine from how far down the stack they may load a value. Indirect loads are assumed to be able to load from any location, and result in a use depth of ∞. This forms an initial approximation to the use depth of each function.
2. [*Iterative propagation.*] Use depth information is iteratively propagated along the call graph of the program from callee to caller. In a given iteration, consider a function f whose use depth is currently set to m. Suppose that f calls functions $g_1, \ldots, g_k$ from call sites $C_1, \ldots, C_k$ respectively, and that the use depths of the functions $g_1, \ldots, g_k$ are set to $n_1, \ldots, n_k$ respectively. Moreover, suppose that the height of f's stack frame, determined from the stack analysis described in Section 3, at the call site C_i is p_i, $1 \leq i \leq k$. Let d_i be the maximum depth in the stack that can be accessed by the call to g_i, $1 \leq i \leq k$, relative to the stack top at the time f was called. We compute d_i as follows:
   - If $p_i = \perp$, we do not know how large f's stack frame is at that call site. In this case, the deepest location in the stack that can be accessed by a load operation in the callee g_i cannot be deeper than n_i relative to the top of the stack at the call site C_i in f. It follows that this location cannot be deeper that n_i relative to the top of the stack when f was called (since f's stack frame cannot have negative size). So we set $d_i = n_i$.
   - If $p_i \neq \perp$, we have two possibilities. If $p_i \geq n_i$, then load operations within the callee cannot access any stack location outside f's stack frame. On the other hand, if $p_i < n_i$ then the deepest location accessed by a load operation within g_i, relative to the stack top at the point when f was called, is at most $n_i - p_i$. In this case, therefore, we have $d_i = max(0, n_i - p_i)$.

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

The computation of kill depths is exactly analogous to that of use depths.

4.2 Improving the Treatment of Indirect Memory References

The basic analysis described above is very conservative in its treatment of indirect memory references. This can be problematic, because at the binary level, even simple source-level code constructs—such as accessing an array element—involve pointer arithmetic off a base address followed by an indirect memory reference. Our implementation therefore extends the basic analysis described above with a straightforward and efficient region-based pointer analysis that aims to determine which area of memory (stack, heap, global memory, etc.) a pointer can be pointing at.

The rationale for this analysis comes from the manner in which the different sections of an executable file are generated. The object module generated by a compiler from a source module typically consists of several code and data sections, e.g., the code section, the constant data section, the zero-initialized data section, etc. The linker combines a number of such object modules into an executable program: in the process, it puts all the sections in their final order and location. The sections of the same type coming from different object modules are typically combined into a single region of that type in the final executable. In general, when generating an object module from a source module, a compiler has no information about other object modules, e.g., their number, size, or the order in which they will be linked together, so it cannot make any assumptions about the eventual locations of these regions in the final executable. As a result, because the distance between the two regions of memory is not known at compile time, the code generated by a compiler for address computations cannot use a pointer to a particular region of memory to obtain an address pointing to some other region of memory. In other words, an address obtained by doing address arithmetic starting with a pointer to a particular region of memory can be safely assumed to fall within that same region of memory. This observation forms the basis of this analysis.

When faced with an instruction I that is an indirect memory reference off one or more registers, we attempt to discover whether or not the register(s) used by I could be pointing into the stack. We do this by tracing back from I in an attempt to determine the origin of the initial value of the registers in question. If we can determine that the base address being used for the pointer arithmetic is in global memory or the heap, then, from the reasoning above, we can conclude that the indirect memory reference cannot affect the stack, and may therefore be ignored for the purposes of stack analysis. A more detailed formulation of this region analysis as a dataflow analysis is given elsewhere [15]; we omit it here due to space constraints.

5 Experimental Results

We evaluated our analyses using ten programs from the SPECint-2000 benchmark suite: *bzip2*, *crafty*, *gap*, *gcc*, *gzip*, *mcf*, *parser*, *twolf*, *vortex*, and *vpr*.[2] The programs were compiled using the *gcc* compiler version 3.2.2, at optimization level `-O3`, with addi-

[2] We were unable to build two of the benchmarks from the SPECint-2000 suite: *eon* and *perlbmk*.

tional flags to instruct the linker to produce statically linked executables containing relocation information (PLTO requires relocation information to identify and update addresses). We ran our experiments on a 2.4 GHz Pentium 4 workstation with 1 GB of main memory.

PROGRAM	TOTAL FUNCTIONS (*Total*)	KNOWN FRAME SIZE (*Known*)	*Known/Total* (%)
bzip2	592	534	90.2
crafty	663	593	89.4
gap	1387	1306	94.2
gcc	2375	1875	78.9
gzip	620	562	90.6
mcf	555	505	91.0
parser	741	674	91.0
twolf	721	660	91.5
vortex	1192	1120	94.0
vpr	808	713	88.2
GEOMETRIC MEAN:			89.8

KNOWN FRAME SIZE: Number of functions for which we are able to determine a nontrivial stack frame size.

Table 1. Precision of Frame Size Analysis

5.1 Frame Size Analysis

Table 1 shows the precision of our stack frame analysis. It can be seen that the analysis is able to infer a nontrivial stack frame size (i.e., a value other than $\perp$) for most functions in the programs: ranging from about 79% for the *gcc* benchmark, to 94% for *gap*, with an average of just under 90%. The main reason the remaining 10% of the functions are not inferred to have a balanced stack is that of "escaping edges"—that is, inter-procedural control flow edges that do not adhere to the normal function call/return mechanism. Such edges are commonly encountered in hand-written assembly code found in some libraries, as well as in code resulting from space-saving optimizations [3]. In our benchmarks, we find that roughly 10%–13% of all functions hav either incoming or outgoing escaping edges; for one program, *vpr*, this figure is as high as 16.9%. To a great extent, this high incidence of escaping edges stems from the fact that we consider statically linked executables; if library code were noet considered, then the proportion of functions without escaping edges—and, concomitantly, the proportion of functions with defined stack frame sizes—would be considerably higher.

5.2 Use-Depth and Kill-Depth Analysis

Table 2 shows the precision of our Use-Depth and Kill-Depth analyses. The precision of these analyses are considerably lower than for Stack Frame Size. We find that the proportion of functions inferred to have a known value of use-depth is about 7.4% on average, with values ranging from 4.9% for *vortex* to 8.9% for *vpr*. For kill-depth analysis the numbers are slightly better: on average some 16.8% of functions are found to have nontrivial kill-depths, with values ranging from 10.7% for *vortex* to 20.2% for *crafty*. The reason for the relatively low values for these two analysis is a combination of the small number of available registers (which causes values to frequently be spilled to the stack, and which necessitates stack analysis in the first place), and our conservative handling of indirect memory operations. The problem is that loads and stores from arrays or records involve pointer arithmetic off a base address, followed by an indirect memory access. The basic algorithm treats all such indirect accesses conservatively, and assumes that it can address any location in memory. Our simple extension to incorporate information about whether the base address is a global or in the function's own stack frame—in essence, associating a very rudimentary form of "region information" with pointers—improves the precision of the analysis somewhat. However, in our current implementation this region information is associated only with registers, not with memory locations. As a result, if a pointer in a register is ever spilled to memory (because of the small number of registers, this is not infrequent), and then subsequently loaded back into a register, all information about the pointer's memory region is lost. A possible solution would be to associate region information with stack slots as well, but we have not yet implemented this.

PROGRAM	TOTAL FUNCTIONS (*Tot*)	USE-DEPTH $\neq \infty$ (*Use*)	*Use/Tot* (%)	KILL-DEPTH $\neq \infty$ (*Kill*)	*Kill/Tot* (%)
bzip2	592	48	8.1	108	18.2
crafty	663	57	8.6	134	20.2
gap	1387	88	6.3	205	14.8
gcc	2375	170	7.2	374	15.7
gzip	620	50	8.1	115	18.5
mcf	555	46	8.3	101	18.2
parser	741	53	7.2	137	18.5
twolf	721	51	7.1	137	19.0
vortex	1192	59	4.9	128	10.7
vpr	808	72	8.9	137	17.0
GEOMETRIC MEAN:			7.4		16.8

USE-DEPTH $\neq \infty$: Number of functions whose use-depth is a known value, i.e., not ∞.
KILL-DEPTH $\neq \infty$: Number of functions whose kill-depth is a known value, i.e., not ∞.

Table 2. Precision of Use-Depth and Kill-Depth Analysis

5.3 Analysis Time

The time taken for each of these analysis is only a relatively small fraction of PLTO's overall execution time. For the benchmarks we used, frame size analysis took, on average, 0.4% of the total processing time, with individual benchmarks taking from 0.08% to 0.8% of the total time. Use-depth and Kill-depth analyses each took 0.2% of the total processing time on average, with individual programs taking from 0.03% to 0.3% of the total time. Interestingly, the proportion of time spent in these analyses was smallest for the largest programs: for *gcc*, for example, frame size analysis took 0.08% of the total time, use-depth analysis took 0.03%, and kill-depth analysis took 0.04%; for *vortex*, frame size analysis took 0.26%, use-depth took 0.12%, and kill-depth took 0.13%. The reason for this is that for the larger programs, I/O and disassembly dominate PLTO's execution time.

5.4 Optimization Effects of Stack Analyses

We evaluated the effect of our stack analyses on other aspects of the system using two optimizations: *load forwarding* and *dead stack store elimination*:

Load forwarding: This optimization attempts to eliminate unnecessary load operations from memory. Suppose we have a pair of instructions I and J such that:
  (i) I is a load instruction $r_0 \leftarrow \texttt{load}(\ell)$;
  (ii) J loads a register r_1 from, or stores r_1 to, the location ℓ;
  (iii) J dominates I; and
  (iv) the contents of memory location ℓ do not change between J and I.
In this case, provided that some additional conditions are satisfied, we can replace the load operation I by a register-to-register move from r_1 to r_0 (or, if $r_0 = r_1$, simply eliminate I). The optimization can be thought of as a special case of common subexpression elimination; it also has the effect of promoting variables into registers that may have been freed up as a result of other optimizations.
This optimization uses kill-depth analysis to check condition (iv) above.

Dead stack store elimination: This is simply the process of doing liveness analysis of stack locations, followed by the elimination of stores to stack locations that are dead.
This analysis uses use-depth analysis as part of the stack liveness analysis.

Table 3 shows the effects of our stack analyses on some of the optimizations implemented within PLTO. The columns labelled 'Analysis Off' give the number of times the optimization could be used when the stack analyses were turned off; the columns labelled 'Analysis On' give the corresponding numbers when the stack analyses were turned on. The difference between these two columns for a particular optimization gives the effect of our stack analyses on the efficacy of that optimization.

It can be seen, from Table 3, that use-depth information is quite effective in improving dead stack store elimination. For most of the benchmarks examined, only a very

PROGRAM	LOAD FORWARDING			DEAD STACK STORE ELIM.		
	Analysis Off	Analysis On	Change (%)	Analysis Off	Analysis On	Change (%)
bzip2	590	596	1.02	2	18	800.0
crafty	903	909	0.67	2	37	1750.0
gap	980	987	0.71	2	20	900.0
gcc	1486	1505	1.28	2	23	1050.0
gzip	584	591	1.20	2	19	850.0
mcf	557	563	1.08	2	18	800.0
parser	649	655	0.92	2	18	800.0
twolf	746	753	0.94	2	18	800.0
vortex	603	650	1.72	44	62	40.9
vpr	603	611	1.33	2	18	800.0

Table 3. Optimization Effects of Stack Analysis

small number of stores into the stack can be eliminated as dead without any information about the liveness of stack locations, and use-depth information is crucial for reasoning about this. By contrast, the effect of kill-depth information on load forwarding is relatively much smaller, though noticeable, even though we are typically able to determine nontrivial kill-depth information for over twice as many functions as for use-depth. The reason kill-depth information has proportionally less impact on load forwarding is that in the case of load forwarding, kill-depth information makes a difference when there is a function call separating the two instructions involved in the forwarding, but because of the small number of registers available, it is very often not possible to find a free register to hold the forwarded value between the two instructions.

6 Conclusion

Binary rewriting is becoming increasingly popular for implementing a variety of low-level code manipulations, ranging from traditional performance-oriented optimizations, to code compression, to security-oriented applications. Because binaries typically have a lot less semantic information than source programs, they are correspondingly harder to analyze. This problem is especially acute for register-poor architectures such as the widely-used Intel x86, where the paucity of registers very often makes it necessary to store values on the stack. This use of the stack, in turn, means that analyses and optimizations need a significant amount of information about how a program uses its stack. This paper describes several simple analyses that aim to obtain such information. Experiments show that the analyses are efficient and can be effective in improving the results of optimizations.

References

1. C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software—Practice and Experience*, 25(7):811–829, July 1995.
2. R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997.
3. S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.
4. S. K. Debray and W. Evans. Profile-guided code compression. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, pages 95–105, June 2002.
5. J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software—Practice and Experience*, 24(2):197–218, February 1994.
6. M. Legendre, G. R. Andrews, and S. K. Debray. BIT: A binary instrumentation toolkit for the Intel IA-32 architecture. Technical report, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, November 2003.
7. C. Linn and S.K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, pages 290–299, October 2003.
8. R. Muth and S. K. Debray. On the complexity of flow-sensitive dataflow analyses. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 67–80, January 2000.
9. 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, January 2001.
10. K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.
11. M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proc. USENIX Technical Conference*, June 2003.
12. J. F. Reiser and J. P. Skudlarek. Program profiling problems, and a solution via machine language rewriting. *ACM SIGPLAN Notices*, 29(1):37–45, January 1994.
13. B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.
14. B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, pages 45–54, October 2002.
15. N. Snavely, S. K. Debray, and G. R. Andrews. Unspeculation. In *Proc. 18th IEEE International Conference on Automated Software Engineering (ASE-2003)*, pages 205–214, October 2003.
16. A. Srivastava and A. Eustace. ATOM—A system for building customized program analysis tools. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.
17. A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.
18. A. Srivastava and D. W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 49–60, June 1994.
19. D. D. Zovi. Security applications of dynamic binary translation, December 2002. Bachelor of Science Thesis, Dept. of Computer Science, University of New Mexico.