# Bidirectional liveness analysis, or how less than half of the Alpha's registers are used

B. De Sutter *, B. De Bus, K. De Bosschere

*Ghent University, Electronics and Information Systems Department, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium*

**Abstract**

Interprocedural data flow analyses of executable programs suffer from the conservative assumptions that need to be made because no precise control flow graph is available and because registers are spilled onto the stack. This paper discusses the exploitation of calling-conventions in executable code data flow analyses to avoid the propagation of the conservative assumptions throughout a program.

Based on this exploitation, the existing backward liveness analyses are improved, and a complementary forward liveness analysis is proposed. For the SPECint2000 programs compiled for the Alpha architecture, the combined forward and improved backward analysis on average finds 62% more free registers than the existing state-of-the-art liveness analysis.

With the improved analysis, we are able to show that on average less than half of the registers of the RISC Alpha architecture are used. This contradicts the common wisdom that compilers can exploit large, uniform register files as found on RISC architectures.

*Key words:* data flow analysis, interprocedural, bidirectional, binary code, architecture, register use

## 1 Introduction

RISC architectures are often considered better suited targets for compiler code generation than CISC architectures. A major contribution of the RISC paradigm is the relatively large number of uniform, general-purpose registers

* Corresponding author.
*Email addresses:* brdsutte@elis.ugent.be (B. De Sutter), bdebus@elis.ugent.be (B. De Bus), kdb@elis.ugent.be (K. De Bosschere).

such architectures offer to the compiler. For example, the Alpha architecture, which is one of the most pure RISC designs ever, offers 31 general-purpose integer registers, and 31 general-purpose floating point registers. In order to optimize the register use, compilers employ data flow analyses [1] and register allocation techniques.

Unfortunately, the register files offered by a RISC architecture are often not as uniform as we would like. Because compilers usually compile source code modules separately, they need to generate code that adheres to calling conventions and application binary interfaces. These describe which registers are callee-saved or caller-saved, which registers need to be used for parameter passing, for storing the return address or the return value, etc. Because of these conventions, compilers are limited in the way they can actually use the registers.

In this paper, we present an analysis of the effectiveness with which separate compilation exploits uniform register files of RISC architectures. This analysis is based on link-time data flow analysis of whole programs. Binary rewriting tools such as link-time optimizers [2–5] or instrumentation tools [6,7], usually use similar data flow analyses as compilers. The most important advantage of applying those analyses at link time is the whole-program overview that is available when a program is being linked. The major drawback is that the analyses need to be applied on executable code that contains much less high-level semantic information than source code files. Instead of analyzing the liveness of variables for example, liveness analysis at link time deals with registers. This follows from the fact that their is no notion of variables in executable code.

On the one hand, analyzing registers instead of variables offers the advantage that no aliasing occurs between them, since no pointers point to registers. On the other hand, data flow analysis of executable code is often hampered by the lack of a precise control flow graph. Moreover, data flow analyses applied on executable code needs to deal with register spills that are difficult to analyze because of aliasing between pointers to memory [8]. As a consequence, conservative assumptions have to be made.

On architectures with a sufficient number of general-purpose registers, such as the ARM, Alpha or PowerPC architectures, most of the register spills in a program originate from maintaining the calling conventions. While it seems obvious that calling convention information is useful to avoid the need for making overly conservative assumptions regarding register spills, we have noticed that existing executable code data flow analyses [3–5,9] fail to exploit most if not all of the available information. This contrasts with the fact that previous work has clearly demonstrated the benefits of having more accurate whole-program data flow analyses. For example, De Sutter et al. [10] have

shown that up to 16% of the achievable link-time code compaction can get lost if no adequate context-sensitive interprocedural liveness analysis is used. Most of this loss does not result from finding fewer useless instructions that can be eliminated. Instead, most of the loss results from not finding enough free registers to apply other optimizations. Similarly, Muth at al. [4] showed how load/store avoidance can decrease the number of executed load instructions with another 3% on average if context-sensitive liveness analysis is used instead of an context-insensitive one.

To remedy the current failure of exploiting calling conventions, this paper describes the circumstances under which calling-convention adherence of executable code can be assumed, and how this information can be exploited in a link-time data flow analysis of executable code. To that extent, we first analyze how separate compilation enforces adherence to calling conventions, and how knowledge thereof is available for link-time data flow analyses. Then we discusses how to improve the state-of-the-art (backward) link-time liveness analysis by exploiting separate compilation information, and we propose a new forward liveness analysis that complements the backward analysis. The major contribution of this article is to show that link-time liveness analysis of executable code is in fact a bidirectional data flow problem. [1]

The new, bidirectional analysis is then evaluated on all 12 programs of the SPECint2000 benchmark suite compiled for the Alpha architecture. With this evaluation, we show that relatively few registers are actually used in the separately compiled code, and that the actual use relates directly to the calling conventions.

This remainder of this paper is structured as follows. Section 2 discusses calling conventions and the register spilling information that is available at link time for data flow analyses. Section 3 discusses how this information can be exploited to extend the state-of-the-art liveness analysis of executable code, and in Section 4 we propose a forward liveness analysis proposed that complements the backward analysis. The proposed analyses and the register use on the Alpha are evaluated in Section 5 and related work is discussed in Section 6, after which conclusions are drawn in Section 7.

## 2 Calling Conventions and Separate Compilation

Calling conventions state amongst others which registers are used to pass arguments to callees, which registers are callee-saved, which are caller-saved,

[1] We, like Muchnick [1], use the term bidirectional to denote that information flows in both the direction of program execution and in the opposite direction.

and which registers are used to return result values of procedures. When source code modules are compiled separately, and data has to be passed from one module to another, a compiler has to rely on such conventions for passing data. This follows from the fact that at any one point during the compilation only one of the two (or more) code fragments involved in the intermodular [2] data passing is known. As a consequence, knowing where the compiler had to assume that intermodular control flow transfers can occur, implies knowing where calling conventions are respected, and where registers are spilled. This observation will be the basis of our exploitation of calling conventions for link-time data flow analysis. At link time, we hence need to be able to determine those program points where intermodular control flow transfers had to be assumed by a compiler.

Before discussing where compilers need to assume intermodular calls however, we will first discuss which calling conventions need to be maintained at those program points. For the sake of clarity, we will focus on intermodular procedure calls in the remainder of this paper. Other intermodular control flow transfers, such as interprocedural jumps that are inserted in the code because of tail-call optimization, can be treated in a similar fashion.

### *2.1 Calling Conventions*

The first case in which calling conventions are needed to implement direct procedure calls concerns procedures of which the compiler does not know all the callers at compile time. These are the so called *global procedures* that are exported from their own module. At compile time, the compiler has to assume that each such procedure may have one or more callers that were generated under the assumption that their callee maintains the calling conventions. Hence global procedures always maintain the calling conventions from the perspective of their callers, i.e., as a callee. Apart from the stack pointer and other special-purpose registers, such global procedures will only consume the values passed to them in argument registers (or on the stack). Furthermore, such procedures will always restore the contents of the callee-saved registers just before control returns to a caller. Finally, such procedures will only pass return values to their callers through the registers defined as return value registers.

The second case concerns the *unresolved* procedure calls: calls of which the compiler does not know all the potential callees at compile time. Since any procedure that can be called at an unresolved call-site has at least one unknown caller, all the callees of unresolved calls are global procedures. Hence the code at an unresolved call-site has to be generated under the assumption

[2] From this point on, we use the term *intermodular* to denote a control flow transfer between two separately compiled modules.

that at least one of its unknown callees will do everything it is allowed to do under the calling conventions. In practice, this means that the code at the call-site is generated under the assumption that its callee will overwrite all registers that are not callee-saved. This implies that, apart from some special-purpose registers, only the argument registers and the callee-saved registers will be live just prior to an unresolved call. Immediately after the call, i.e., at the continuation point in the caller, only the special-purpose registers, the callee-saved registers and the return value registers will be live.

With respect to indirect calls, we assume that compilers always generate procedure calls through function pointers as unresolved calls. Furthermore, all procedures whose address is taken somewhere in the program will be assumed to be generated like global procedures. As far as we know, these are reasonable assumptions with the current state of the art in compiler technology. Would these assumptions be invalid, however, it is trivial to abandon them.

### *2.2 Traditional Separate Compilation*

With traditional compilers, all source code files are compiled into separate object files independently of each other. As a consequence, all calls from one object file to another are unresolved intermodular calls to global procedures. Since it is the linker that combines the separate object files into a program, a link-time optimizer can easily distinguish all such inter-object calls from intra-object calls. Moreover, symbol information [11] tells the linker which procedures are exported from their object files, and hence which procedures are global. Furthermore, relocation information [11] allows the link-time optimizer to detect which procedures have their address taken in the program. In all these cases, adherence to the calling conventions as described in Section 2.1 can be assumed.

By contrast, no assumptions can be made about intra-object calls to local procedures, since both the caller and the callee are known at compile time.

That leaves us with only one case to discuss: what can we assume about intra-object calls to global procedures? To answer this question, we will study a small example program, of which the source code is depicted in Figure 1(a). In the `main()` procedure the results of two externally defined, global procedures `a()` and `b()` are printed. Furthermore, the two definitions of procedure `b()` return the number of the file (*file1.c* or *file2.c*) they belong to. Finally, in *file2.c*, procedure `a()` returns the result of the called procedure `b()`.

In Figure 1(b), we have dumped the commands we used to compile these files separately (lines 1-3), to link the resulting object files into two different shared libraries (lines 4-5), and to link `main.o` to two combinations of the libraries,

*main.c*

```
#include <stdio.h>
extern char a();
extern char b();

int main(){
 printf("%c %c\n",a(),b());
}
```

*file1.c*

```
char b(){
 return '1';
}
```

*file2.c*

```
char b(){
 return '2';
}

char a(){
 return b();
}
```

(a) Three example source code files

```
1  % gcc -c main.c -o main.o
2  % gcc -c file1.c -o file1.o
3  % gcc -c file2.c -o file2.o
4  % gcc -shared file1.o -o lib1.so
5  % gcc -shared file2.o -o lib2.so
6  % gcc main.o -L. -l2 -o a.out
7  % gcc main.o -L. -l1 -l2 -o b.out
8  % ./a.out
9  2 2
10 % ./b.out
11 1 1
```

(b) Compilation and execution of the example code on a Linux command-line.

Fig. 1. Compilation and execution an example program.

producing two programs (`a.out` on line 6 and `b.out` on line 7). The output generated by both programs is also depicted (lines 9 and 11).

In `a.out` only the code generated for *main.c* and *file2.c* is linked into the program. The print statement indeed results in two 2s being printed. In `a.out` the call from `a()` to `b()` is therefore clearly an intra-object call, as both procedures originate from the same source file. In `b.out` the linker first included procedure `main()`, then procedure `b()` from `file1.o`, and then procedures `b()` and `a()` from `file2.o`. As can be seen in the output of `b.out`, procedure

`a()` now calls the version of procedure `b()` from *file1.c*: the library containing `file1.o` is fed to the linker first, so its version of procedure `b()` overrides the version in `file2.o`. As a result, the call from `a()` to `b()` has become an inter-object call.

When `file2.c` was compiled, the compiler apparently could not know whether or not the call from `a()` to `b()` would be an inter-object or an intra-object call. Conservatively, the compiler therefore had to assume that the callee of the call would be overridden, and hence the call was treated as an unresolved call. In general, any seemingly intra-object call to a global procedure may become an intermodular call when the callee is overridden by an object file or library that is passed to the linker first. So in the end, any such call will be generated as an unresolved call.

### *2.3 Partially Separate Compilation*

Unlike with traditional separate compilation, state-of-the-art programming tool chains can compile code from multiple source code files together, and still store the results in separate object files.

In practice, a lot of programs are not generated with separate invocations of a compiler and linker (as in Figure 1), but with one invocation of the compiler, which is then fed with all the source code files to produce an executable program. In such case, the compiler itself invokes the linker, and hence it knows precisely which application code constitutes the program, and which global procedures override each other. This knowledge can then be exploited to optimize the program beyond the calling conventions. In particular, all resolved calls to procedures that use very few registers may be optimized with the knowledge that more registers remain saved in their callee than is prescribed by the calling conventions.

However, since it is the linker that decides which libraries will be linked against the thus generated code, this reasoning does not hold for calls from the application code to the library code: such calls are still unresolved. Moreover, the fact that direct calls within the application-specific code are now all resolved does not imply that the callers of the global procedures are all known at compile time as well. Indeed, since the global procedures being compiled may still override library procedures, they potentially can have callers in the libraries that still only become known at link time. Compilers performing whole-program optimizations therefore still need to generate global procedures that adhere to the calling conventions as callees.

### *2.4 None-compiled Code*

So far, we have silently assumed that all code in a program is generated by a compiler. In practice, this is not true. There are in general two types of code that are not compiled (separately), and for which the aforementioned assumptions hence not hold.

The most obvious case in which the assumptions may be invalid relates to hand-written assembly code. If an assembly programmer has written a procedure and all of its callers by hand, then he can neglect the calling conventions, even if not all the callers reside in the same source code file. Fortunately, it is trivial to detect such cases. For example, we needed to add less then 20 lines of source code to the GNU gcc compiler and binary utilities to enforce them to produce special symbols that tag inline-assembly code in compiled source code. Furthermore, most object file formats provide place in object files to store the source code file name from which an object file was generated. Conventionally, files ending with .S or .s extensions are assembler files. Consequently, our link-time analysis can differentiate between compiled code and manually-written assembler code by simply looking for inline-assembly tags and assembler file name extensions. In the remainder of this paper, we can hence safely neglect unconventional assembler code, and focus on compiled code instead.

The second case relates to so-called *built-in* routines. These are used to implement often recurring programming constructs for which no hardware implementation exists on the target architecture, and for which a call to a true library function is considered too expensive. Since these routines are generated by the compiler that also generates their callers, they do not need to maintain the calling conventions. Although built-in routines occur frequently in practice, we will simply neglect them in this paper. As any link-time binary rewriter that is integrated in a compiler tool chain can be notified by the compiler/assembler when a built-in routine is used [3], the rewriter will be notified too, and hence the rewriter can drop the assumptions about calling conventions where necessary.

## 3 Interprocedural Liveness Analysis

This section discusses how calling convention information can be used to extend the state-of-the-art liveness analysis of executable code that is described in detail by Muth [9], and is used in the link-time optimizers Squeeze [2],

[3] The GCC tool-chain, for example, offers this possibility.

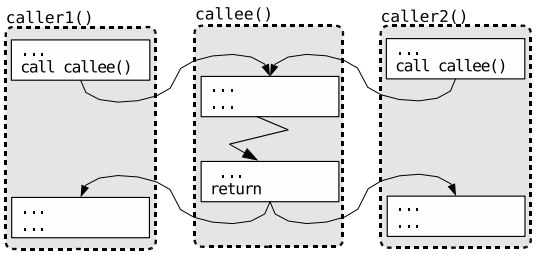



Fig. 2. Interprocedural edges in the ICFG.

| | |
|---|---|
| $r(c)$ | is the corresponding return point of call-site $c$, |
| $ca(c)$ | is the procedure (callee) called at call-site $c$, |
| $cs(r)$ | is the corresponding call-site of the return point $r$, |
| $p(n)$ | is the procedure of basic block $n$, |
| $use(n)$ | is the set of registers used in node $n$ before being defined, |
| $def(n)$ | is the set of registers defined in node $n$, |
| $succ(n)$ | is the set of successor nodes of node $n$, |
| $pred(n)$ | is the set of predecessor nodes of node $n$, |
| $i(p)$ | is the init node of procedure $p$, |
| $x(p)$ | is the exit node of procedure $p$, |
| $N$ | is the set of all nodes, |
| $C$ | is the set of all call nodes, |
| $R$ | is the set of all return nodes, |
| $I$ | is the set of all init nodes, |
| $\mathbf{R}$ | is the set of all registers, |
| $\mathbf{R}_a$ | is the set of registers used for argument passing according to the calling conventions, |
| $\mathbf{R}_r$ | similarly is the set of registers used for passing return values, |
| $\mathbf{R}_s$ | similarly is the set of callee-saved registers, |
| $\mathbf{R}_l$ | similarly is the set of special-purpose registers that is always live at the entry and exit of a procedure. This set includes, e.g., the stack pointer and the global pointer. |

Table 1
Notation used throughout the paper

Squeeze++ [10,12], and Alto [4]. For this discussion, we will use the notation introduced in Table 3. Furthermore, it needs to be understood that all analyses we will describe are assumed to operate on an interprocedural control flow graph (ICFG), in which the call-sites are connected to the entry nodes of their callees. Nodes ending with a return statement are connected to the continuation points of the calls that call the procedure containing the return statement. This is illustrated in Figure 2. Note that such an ICFG is obtained by disassembling an executable program, and by partitioning the assembled instructions in basic blocks and procedures. This disassembler and ICFG-construction algorithm is discussed in detail by De Sutter et al. [10].

To avoid the imprecision resulting from considering unrealizable paths in a program's ICFG, a context-sensitive interprocedural analyses first computes summary information about procedures, in terms of which the data flow equations at call-sites are then stated. Two pieces of summary information are usually collected.

- $mayUse(p)$ is the set of registers whose values upon entry to procedure $p$ may be used in $p$ (or any of its callees) before being defined. This set is hence always live on entry to $p$. It typically contains the arguments of a procedure, the stack pointer, and the return address.
- $byPass(p)$ is the set of registers that will always hold the same value after returning from $p$ as they held upon entry to $p$. In other words, the registers in $byPass(p)$ are live at some call-site $c$ calling $p$ if they are live at the corresponding return node $r(c)$. Typically these are the callee-saved registers and the registers which are not used at all by $p$ or any of its callees.

With these sets, the context-sensitive liveness analysis can be described with the following data flow equations:

$$\forall n \in N : live_{in}(n) = use(n) \cup \Big(live_{out}(n) - def(n)\Big), \tag{1}$$

$$\forall n \in N \setminus C : live_{out}(n) = \bigcup_{s \,\in\, succ(n)} live_{in}(s), \tag{2}$$

$$\begin{aligned} \forall c \in C : live_{out}(c) = {} & mayUse\Big(ca(c)\Big) \\ & \cup \Big(byPass\Big(ca(c)\Big) \cap live_{in}\Big(r(c)\Big)\Big). \end{aligned} \tag{3}$$

This set of equations is most often solved with an iterative fix-point computation subject to the initial values

$$\forall n \in N : live_{in}(n) := live_{out}(n) := \emptyset. \tag{4}$$

Over the years several different solutions have been proposed for computing the *mayUse* and *byPass* sets [3,5,9]. Of these solutions, Muth's solution [9] is as precise as the others and it is the most elegant one, as it computes the *mayUse* and *byPass* sets with the same equations (albeit on different sets) as the context-sensitive liveness analysis itself.

```
proc f():
    store r2 -> 0(sp)
    move 0x123 -> r2
    mul r0,r2 -> r0
    load  r2 <- 0(sp)
    return
```

Fig. 3. Example pseudo assembler code

### *3.1.1 Spilled Registers*

To illustrate the problem of spilled registers that this analysis suffers from, consider the assembly code of procedure `f()` depicted in Figure 3. Clearly whether or not register `r2`, which is temporarily stored on the stack, contains a live value at a call-site calling `f()` depends on whether or not `r2` holds a live value after `f()` has finished executing. In order to model this less conservatively in our context-sensitive liveness analysis, such registers must be placed in the *byPass* sets instead of in the *mayUse* sets, even though in the strictest sense procedure f() uses and defines `r2`.

Squeeze [2], Squeeze++ [10,12], and Alto [4] use a simple stack analysis by code inspection to detect such stack saves and restores, thus improving the precision of the context-sensitive liveness analysis without requiring any changes to the data flow equations. Unfortunately however, this code inspection still yields far from precise results, because optimizing compilers try to mix the saves and restores with the code of the procedure body.

### *3.1.2 Unknown Control Flow*

In the introduction, we mentioned that executable code liveness analysis can suffer from the lack of a precise ICFG. This happens, e.g., for indirect control flow transfers of which a link-time rewriter cannot detect all possible targets, and for code fragments whose address is loaded or computed somewhere, but of which it is not known how those addresses will be used in the program. In the former case the successor nodes of the indirect transfer in the ICFG are unknown, and in the latter case, the possible predecessors of the code fragment are unknown.

The standard solution to this problem is to model the unknown successors and unknown predecessors with a virtual *unknown node* $n_u$ that behaves conservatively for all analyses and transformations applied on the ICFG. This node is the only block of a virtual *unknown procedure* $p_u$ in the program. The unknown node becomes a successor of all nodes ending with unknown control

flow, and a predecessor of all nodes of which we cannot detect all predecessors. For the correctness of liveness analysis in the presence of unknown control flow, it suffices to initialize the properties of the unknown node conservatively:

$$live_{in}(n_u) := live_{out}(n_u) := \mathbf{R}, \tag{5}$$
$$mayUse(p_u) := byPass(p_u) := \mathbf{R}. \tag{6}$$

### *3.2 Exploiting the Calling Conventions*

This section discusses when and how calling convention adherence in a program can be exploited effectively in the state-of-the-art analysis.

#### *3.2.1 Unknown Callees*

First of all, we can model the unknown callees of indirect procedure calls with a special *unknown-callee* procedure $p_{ucee}$ instead of with the general unknown procedure $p_u$. This follows from our assumption that indirect calls in the assembly code were unresolved by the compiler. The following properties hold for $p_{ucee}$:

$$mayUse(p_{ucee}) := \mathbf{R}_a \cup \mathbf{R}_l, \tag{7}$$
$$byPass(p_{ucee}) := \mathbf{R}_s. \tag{8}$$

N ote that on some platforms this node and procedure can also be used to model the unknown code called by system calls. Usually, however, the calling conventions for system calls differ from those for normal procedure calls. If that it the case, a similar but different node and procedure must be used to model the system calls.

#### *3.2.2 Unknown Callers*

Furthermore, we can model unknown callers of procedures with a special *unknown-caller* procedure $p_{ucer}$ and two unknown-caller nodes: $n_{uceri}$, the node that calls the procedure with unknown callers, and $n_{ucerx}$, the continuation point of $n_{uceri}$ in $p_{ucer}$. This follows from our assumption that procedures whose address is taken in the program are treated as global procedures by the compiler. The node $p_{ucerx}$ then models the fact that upon returning to an unknown caller, only a limited number of registers can be live. Formally

$$live_{in}(n_{ucerx}) := \mathbf{R}_s \cup \mathbf{R}_r \cup \mathbf{R}_l. \tag{9}$$

### *3.2.3 Global Procedures*

As mentioned in Section 3.1.1, the computation of the *mayUse* and *byPass* sets of a procedure can benefit from analyzing the registers spills. In practice, however, such an analysis still lacks the necessary precision.

Fortunately, there exists a shortcut for global procedures, as we can place an upper-bound on the *mayUse* sets. More precisely, we know that the following conditions should hold for any global procedure $p$:

$$mayUse(p) \subset mayUse(p_{ucee}). \tag{10}$$

This follows from the fact that such a procedure $p$ adheres to the calling conventions as a potential callee of (at compile time) unknown library code.

For the *byPass* set, a similar reasoning applies, but here we have to consider two cases. In the case of traditional separate compilation, we know that it is safe to compute the $byPass(p)$ set of a global procedure $p$ as follows:

$$byPass(p) = byPass(p_{ucee}), \tag{11}$$

since no caller, not even one in the same source code file, of the global procedure $p$ will have made any assumptions about the (potentially overridden) implementation of $p$.

In the case of partially separate compilation however, we cannot guarantee this. There might be callers that exploit the fact that more registers remain unchanged than the number prescribed by the calling-conventions. Yet, since a global procedure always maintains the calling conventions as a callee, we can still place a lower-bound on the *byPass* sets of such procedures. Formally,

$$byPass(p) \supset byPass(p_{ucee}). \tag{12}$$

In both the cases of fully or partially separated compilation, these new bounds can be used to move registers from the computed *mayUse* set of a global procedure to the *byPass* set, thus improving the precision of the liveness analysis.

Please note that these bounds should not be used independently of each other. Consider the code in Figure 3 again, and suppose that we do not have a stack analysis that detects that `r2` is in fact spilled to the stack. Since we fail to detect that the store instruction is a part of a stack spill, `r2` initially ends up in $mayUse(f)$. And since we fail to detect that the load instruction is part of a stack spill, `r2` initially is not an element of $byPass(f)$. When we exploit the fact that `f()` is global, and if we assume that `r2` $\in \mathbf{R}_s$, we can remove `r2` from $mayUse(f)$ and add it to $byPass(f)$. Obviously removing `r2` from $mayUse(f)$

```
proc g():
    ...
    call h(): // intermodular call
    sub r0,0x1 -> r2
    jump r3  // to unknown target
    ...
```

Fig. 4. Example pseudo assembler code of a separately compiled procedure.

without adding it to $byPass(f)$ would be incorrect, as the incorrect result would then be that `r2` is considered dead at each call-site of `f()`.

## 4 Forward Liveness Analysis

In this section, we propose a forward liveness analysis that complements the backward analysis. To illustrate the necessity of this forward analysis, we will use the example code fragment depicted in Figure 4, of which we assume that procedures `g()` to `h()` were compiled separately. Furthermore, we assume a target architecture with five registers `r0`-`r4`, of which $\mathbf{R}_a = \{\texttt{r0}, \texttt{r1}\}$, $\mathbf{R}_r = \{\texttt{r0}\}$ and $\mathbf{R}_s = \{\texttt{r2}, \texttt{r3}\}$. With the backward liveness analysis only, all registers will be considered live before the indirect jump because this jump is modeled with an edge to the unknown node $n_u$, and because $live_{in}(n_u) = \{\texttt{r0}, ..., \texttt{r4}\}$.

We should make the following observation however. When the unresolved call `g()` to `h()` was compiled, the compiler had to assume conservatively that procedure `h()` would overwrite registers $\{\texttt{r0}, \texttt{r1}\}$, as this is allowed under the calling conventions. Of those two registers, only `r0` can hold a meaningful value when control returns from `h()` to `g()`: the return value of `h()`. By contrast, `r1` will never be used to pass data from `h()` to `g()`, and hence `r1` cannot be live when control returns from `h()` to `g()`. And if `r1` is not live then, it cannot be live just prior to the indirect jump in `g()`. Obviously the backward data flow analysis can never model this behavior.

Instead a forward analysis is necessary. Informally, this analysis needs to determine which registers may contain meaningful data. These are registers that are defined at some point or, when calling conventions can be assumed, that may be used for passing data according to the calling conventions. Since a compiler will never have generated code that consumes meaningless values, [4]

[4] Note that some compilers do generate instructions that consume meaningless values. For example, an instruction such as `xor r1,r1,r1` can be used to set the

$$\forall n \in N : mean_{out}(n) = mean_{in}(n) \ \cup \ def(n), \tag{13}$$

$$\forall i \in I : mean_{in}(i) = \Big( \bigcup_{p \ \in \ pred_D(i)} mean_{out}(p) \ \cap \ \Big( mayUse\big(p(i)\big) \ \cup \ byPass\big(p(i)\big) \Big) \Big) \cup \ \Big( \bigcup_{p \ \in \ pred_I(i)} mean_{out}(p) \ \cap \ \Big( mayUse\big(p(i)\big) \ \cup \ \mathbf{R}_s \Big) \Big), \tag{14}$$

$$\forall r \in R_G : mean_{in}(r) = \Big( \mathbf{R}_r \ \cap \ mean_{out}\Big( x\Big( ca\big( cs(r) \big) \Big) \Big) \Big) \cup \ \Big( byPass\Big( ca\big( cs(r) \big) \Big) \ \cap \ mean_{out}\big( cs(r) \big) \Big), \tag{15}$$

$$\forall r \in R \setminus R_G : mean_{in}(r) = mean_{out}\Big( x\Big( ca\big( cs(r) \big) \Big) \Big) \cup \ \Big( byPass\Big( ca\big( cs(r) \big) \Big) \ \cap \ mean_{out}\big( cs(r) \big) \Big), \tag{16}$$

$$\forall n \in N \setminus (R \cup I) : mean_{in}(n) = \bigcup_{p \ \in \ pred(n)} mean_{out}(p), \tag{17}$$

$$mean_{out}(n_u) := \mathbf{R}, \tag{18}$$

$$mean_{out}(n_{uceri}) := \mathbf{R}_a \cup \mathbf{R}_s, \tag{19}$$

$$\forall n \in N \setminus \{n_u, n_{uceri}\} : mean_{out}(n) := \emptyset, \tag{20}$$

$$\forall n \in N : mean_{in}(n) := \emptyset. \tag{21}$$

Fig. 5. Data flow equations of the forward liveness analysis.

registers that do not hold meaningful contents at some program point are dead, even if this cannot be derived by the traditional backward liveness analysis. More formally, the data flow problem we want to solve can be summarized as follows: compute the set $mean(p)$ sets of registers that satisfy the equations shown in Figure 5. [5]

contents of register `r1` to zero, regardless of the original contents of `r1`. In such cases `r1`, albeit live in the strict sense of the word, does not have a contents whose value matters before the instruction. Hence `r1` can be treated as dead in practice.

[5] Note to overload the equations, we have omitted the set $\mathbf{R}_l$. Adding it is trivial.

The meaning of equation (13) is obvious: any register having a meaningful value upon entry to a node, has one upon exit too. Moreover, all defined registers are assumed to be defined with meaningful values.

In equation (14) $pred_D$ and $pred_I$ denote the sets of all direct and indirect call-sites resp. This equation states that the set of meaningful registers upon entry to a callee is a subset of all registers with meaningful values at the callee's call-sites. This first of all includes the arguments in the *mayUse* set of the callee. For a direct call, the caller might know all the registers in the *byPass* set of the callee and the caller therefore can use these registers to pass values over the procedure call, thus storing a meaningful value in them. In the case of an indirect call, the caller does not know the *byPass* set, and therefore it can only pass registers from $\mathbf{R}_s$ over the procedure call.

$R_G$ is the set of all return nodes corresponding with calls to global procedures. This includes return nodes corresponding with calls to the unknown-callee node. Equation (15) states that the registers with meaningful contents after the return from a global callee are the registers that are meaningful at the exit of the callee and that can contain return values, plus the registers that are passed over the callee and have meaningful contents at the corresponding call-site.

Equation (16) is similar to equation (15). The only difference is that a local procedure can use other registers than the ones in $\mathbf{R}_r$ for returning values. For all other nodes, equation (17) is straightforward.

The iterative fix-point solution of this problem is subject to the initialization given in equations (18)–(21).

The results of this analysis can be used very easily to limit the number of live registers at each program point, since the set of live registers at program point $p$ consists of $live(p) \cap mean(p)$. One can easily verify that the backward and forward analysis combined in this manner determine the smallest possible set of live registers at the `jump` statement in the example of Figure 4.

The time-complexity of the forward analysis is obviously the same as the complexity of the backward analysis: similar information is propagated over roughly the same edges in the ICFG (intraprocedural edges, summary edges (that connect call-sites to their corresponding return points) and call edges instead of return edges). Moreover, if the forward analysis is used as an extension of the backward analysis, the *mayUse* and *byPass* sets can be reused for free in the forward analysis. Consequently, the complexities of both the forward analysis and the backward analysis equal that of a traditional liveness analysis that is applied four times: two times to compute the *mayUse* and *byPass* sets [9], once for the actual backward analysis, and once for the forward analysis.

Provided that the forward analysis is used as an extension of the backward analysis, the forward analysis doesn't require any additional memory either. In order to implement the backward liveness analysis efficiently, three register sets need to be stored for each node $n$ in the ICFG: $def(n)$, $use(n)$ and $live_{out}(n)$. Since we do no longer require the *use* sets in the forward analysis, we can reuse that space to store the $mean_{in}$ sets. In fact, assuming that no compiler will ever generate code that uses meaningless register contents in some node, we can initialize the $mean_{in}$ sets with the *use* sets.

Furthermore, an efficient implementation of the forward analysis is very similar to that of the backward analysis. In particular, we found it most efficient to work with a work list of procedures. For each procedure to be visited, the basic blocks are iteratively visited in code layout order (i.e., along increasing addresses) until convergence is obtained inside a procedure. By using the code layout order, which corresponds to the disassembly order and hence the memory allocation order, little overhead needs to be spent on computing and, more importantly, storing theoretically optimal, but more complex, orderings, thus saving precious cache space. This saving is important because we want to compute liveness information on hundreds of thousands of blocks for large programs. For such large programs, obtaining good cache behavior has proven to be difficult, but not impossible. Not waisting space on complex ordering information is one way to do so, thus improving the spatial locality of the data accesses. The second way is to improve the temporal locality, by making the equations converge on a procedure-per-procedure basis, using the procedure work list. For a more detailed discussion on such an efficient implementation, we refer to De Sutter et al. [10], which includes an extensive performance evaluation of several whole-program data flow analyses.

We can conclude that the combined backward and forward analyses are no more complex than the backward analysis by itself.

## 5 Experimental Evaluation

To evaluate the way registers are used with separate compilation, and to evaluate the exploitation of calling convention information in the liveness analyses we have proposed in the previous sections, we have implemented them in Diablo [13,14], a link-time rewriting framework that can be used for program optimization, compaction, instrumentation and other link-time transformations.

We compiled all 12 SPECint2000 (http://www.spec.org) programs with a single invocation of the vendor-supplied Compaq C and C++ compilers (version 6.3) for the Alpha 21264 EV67 Tru64 Unix 5.1 platform and used Diablo

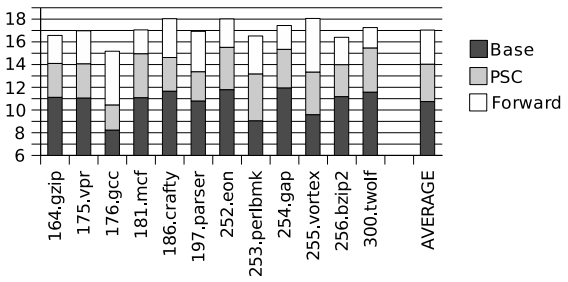



Fig. 6. The average number of dead registers at the end of all basic blocks.

to analyze the thus generated, statically linked binaries. The context of our evaluation is therefore that of partially separate compilation, as discussed in Section 2.3. The three versions of liveness analysis we implemented in Diablo are:

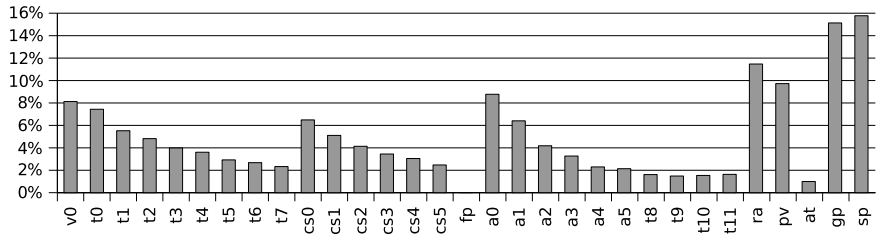
(1) *Base* — The base version of our experiments is the liveness analysis used in Alto [4] and Squeeze [2] and described in more detail by Muth in his Phd. thesis [9]. This analysis solves equations (1)–(9).
(2) *PSC* — In addition to the equations used in the base version, this version also uses equations (10)–(12). This backward analysis therefore exploits the calling conventions to the limit, as discussed for the case of Partially Separate Compilation (PSC).
(3) *Forward* — This version combines the backward PSC analysis and the forward analysis described by the equations (13)–(21).

To validate our extensions, we inserted instructions in the program that write random numbers in all registers that are detected as dead at the entry points of all basic blocks. This "instrumentation" did not alter the functionality of any of the benchmark programs, thus validating the correctness of our analyses and the results we present.

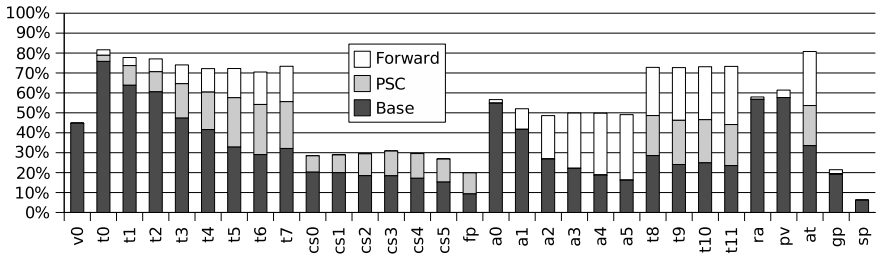
### *5.1 Precision of the Analyses*

The number of dead registers found by the three analysis versions is depicted in Figure 6. On average, the use of calling convention information to compute *mayUse* and *byPass* sets results in 3 additional registers detected as dead in the PSC version of the backward analysis. For the individual benchmarks, the number of additional registers marked dead ranges from 2 to 4. When the forward analysis is applied as well, another 3 registers are detected as dead on average, ranging from 2 to 4.5.

The average number of registers considered dead by our extensions therefore in total increases from 10.5 to 17. This is a 62% increase, of which slightly less than half results from the forward analysis. This clearly demonstrates that




(a) Percentage of the instructions that use or define a register.




(b) Percentage of basic blocks at the end of which a register is dead according to the three analysis versions.

Fig. 7. Liveness information for each general-purpose register, averaged over the whole SPECint2000 benchmark suite. On the Alpha architecture, $\mathbf{R}_a = \{a0, ..., a5\}$, $\mathbf{R}_s = \{cs0, ..., cs5, sp, ra\}$, $\mathbf{R}_r = \{v0\}$, and the scratch registers are $\{t1, ..., t11, at\}$.

liveness analysis of executable code should be considered as a bidirectional data flow problem.

To understand the analysis improvement resulting from the two liveness analysis extensions presented in this paper, Figure 7b shows the fraction of basic blocks at the end of which each general-purpose register of our target architecture is dead. To help interpreting these fractions, Figure 7a depicts how frequently each register is read or written on average.

In general one would expect that the fewer a register is used in a program, the more it contains dead values. Yet the opposite seems to occur when the base analysis is used. For the scratch registers, the argument registers and the callee-saved registers, we learn from Figures 7a and 7b that the more a register is actually used in the code, the more the base analysis detects that it is dead.

The reason is that when the base analysis is used, the conservative assumption about a register in the unknown node $n_u$ propagates through the program precisely until that register is defined somewhere. Thus, the fewer a register is defined in a program, the higher is the impact of the conservative assumptions about that register in the unknown node.

For the callee-saved registers, the use of calling convention information in the PSC analysis solves this problem to a large extent by avoiding that these

registers appear in the *mayUse* sets instead of in the *byPass* sets. For scratch registers, the PSC version also solves this problem to some extent because the scratch registers can be removed from the *mayUse* sets for global procedures.

To solve this problem for the argument registers, however, that are conservatively assumed to be all elements of $mayUse(n_{ucee})$ and that are not removed from any *mayUse* sets in the PSC backward analysis, the forward analysis is required. Additionally, the forward analysis proves to be useful for detecting dead scratch registers. This follows from the fact that in the case of partially separate compilation, we can only put a lower bound on the *byPass* sets of global procedures, not allowing us to remove the scratch registers from the *byPass* sets. As a result, the improved backward analysis still is overly conservative. This is effectively remedied by the forward analysis however.

With respect to the time-complexity of the extended analysis as discussed in section 4, we have verified that there is no additional complexity. For none of the benchmarks, the total analysis time increased by more than 35% when the forward analysis was applied on top of the backward analysis. The average slowdown due to the forward analysis was limited to 26%. As indicated in section 4, the extensions do not increase the memory footprint of the analyses.

### *5.2 Computer Architecture Design*

From a computer architecture point of view, the following observations can be made following the evaluation of our extensions to liveness analysis.

- Much to our surprise, the results indicate that on average 17 of the 31 general-purpose integer registers of the Alpha architecture are dead. We believe that this firmly contradicts the common wisdom that compilers can easily exploit large, uniform general-purpose register files.
- Clearly, not all registers are used equally. From Figure 7, it is clear that a register's function in the calling conventions strongly influences how effectively it can be used.
- Either a lot more optimization opportunity for global register allocation at link time exists than what is already known [15,5], or maybe 31 general-purpose registers is just too much.

### *5.3 Discussion*

First, it should be noted that the detection of additional dead registers with the backward analysis is only indirectly useful for link-time program optimization. As the additional dead registers this analysis detects are in fact not

used or defined, their detection cannot result in more useless (meaning that produce dead values) instructions being eliminated. Still the additional dead registers may be used by other transformations that need scratch registers, such as whole-program register allocation [15,5], code abstraction [12] and code instrumentation [6,7].

Secondly, it may be dangerous to draw conclusions from the experiments presented in this paper, which are limited to one archticture and one compiler. We do believe that broad conclusions can be drawn, however, for the following reasons:

- While we did not report numbers obtained with other compilers, we do now that the low number of used registers is not a symptom of poor register allocation by the used compiler. For example, De Sutter et al. [10] have shown that more link-time code compaction can be achieved on code generated with the open-source GCC compiler, that on executable code produced by the proprietary compiler we used for this paper. This difference was to a large extent the result of poorer register allocation in the case of GCC. This gives credibility to the fact that the results presented in this paper are fundamental results, rather than artifacts of a poor compiler.
- While we have not reported results for other architectures, we have experienced similar behavior on other architectures. For example, with the ARM backend of Diablo, manual inspection of the ICFGs generated by ARM's priorietary compiler as well as with GCC shows that many of its 16 registers remain largely unused as well. However, since the ARM calling conventions specify that the same four register that are used to pass arguments can also be used to pass (multiword) return values, meaning $\mathbf{R}_a = \mathbf{R}_r$, our forward analysis results in much fewer additional dead registers being found. We hope to overcome this by using compiler-generated type information to tell us the return types of procedures. If these are single word types, as with most procedures, we will be able to get more accurate estimations than $\mathbf{R}_r$. This is future work, however, since Diablo currently only uses information that is always available in object files, such as relocation information and symbol information [11]. Type information is not necessary for linking, and thus not necessarily available. Still many compilers and assemblers can provide type information to support debugging.

## 6 Related work

Our extended backward interprocedural liveness analysis of executable code is based on the analysis used in Alto [4] and Squeeze [2], and described in detail by Muth in [9]. Because of a lack of space, we have not described in detail how the *mayUse* and *byPass* sets can be computed by code inspection.

Muth [9] describes the most elegant of such computations. Other variations are described by Srivastava and Wall [5] for the OM link-time optimizer and Goodwin [3] for the Spike [15] link-time optimizer, in which global register allocation is performed.

It is not exactly clear to which degree the results reported by Srivastava, Wall and Goodwin depend on adherence to calling conventions. Knowing that these researchers belonged to research groups closely cooperating with the compiler groups of their target platforms, it seems reasonable that they could be less conservative than we can afford to be in our third-party link-time transformation framework.

Muth [9] certainly did not exploit all of the available calling convention information. Instead he writes "Unfortunately, we have no control over the enforcement of calling conventions in general, except for system calls. [...] It seems reasonable, however, to assume that calls to shared libraries and calls through function pointers respect the calling convention". In Section 2, we have clearly rejected the overly conservative first part of his statement. Moreover, as co-authors of Squeeze [2] and Alto [4], we know that none of the equations (10)–(21) were ever implemented in those prototype tools. Equations (10)-(12) were first introduced in Squeeze++ [10,12], while equations (13)–(21) were first implemented in Diablo [13,14], and are first presented in this paper.

Besides work on the liveness analysis itself, and the use of information it computes, there has been some research on the analyses of spill code and other memory accesses. For example, Balakrishnan and Reps [16] present a method to analyse how memory locations (on the stack and in statically allocated memory) are used by an executable program to extract information on the local and global variables used in a program without (the often unavailable) symbol information. This information aids in inspecting executable programs, for example to verify whether a program is benign or whether it contains virus-like code. Unlike our conservative analysis, of which the results can be used for automatic program rewriting, their results are approximative only, and not always conservative. This poses no problem for their application of manual code inspection, but clearly their non-conservative analysis cannot be used to augment data-flow analyses in automatic binary rewriters.

Finally, Linn et all. [17] propose a method for analyzing the stack behavior of x86 executables. For that Intel CISC architecture, with very few registers, the analysis of stack saves and restores is much more important to obtain precise data-flow than on RISC architectures such as the Alpha. We believe that it still could be useful to apply their stack analysis as a pre-pass to our liveness analysis, as their analysis can complement our interpretation of calling convention information in two ways. Firstly, their analysis can reveal information about unconventional code fragments, such as local procedures and manually-written

assembly code. Secondly, their analysis can obtain additional information on register spills that do not relate to calling conventions, but rather originate from a total lack of free registers. However, given the few program points at which this happens, and the fact that almost all code in programs these days is compiler-generated, we conjecture that the contribution of their analysis to our liveness analysis would be minimal. A final observation we should make about their method is that they do not consider calling conventions at all. For example, a first step in Linn et al.'s method determines which procedures are always exited with the same stack height as when they are called. In their work, this step only relies on code inspection. Obviously, calling convention information as used in this paper would be very useful for this step too.

## 7 Conclusions

We have discussed which calling convention information is available to a link-time data flow analysis of executable code. We have presented a new extension to exploit this information in the state-of-the-art liveness analysis. With this extension, on average more than 3 additional dead registers are detected in the SPECint2000 benchmark suite compiled for the Alpha platform. Furthermore, we have proposed a forward analysis to complement the backward analysis, thus showing that link-time liveness analysis of executable code is in fact a bidirectional data flow problem. The addition of the backward analysis on average again results in 3 more registers being detected as dead.

In total, the two extensions we presented in this paper have increased the average number of dead registers detected in executable Alpha programs from an average of 10.5 to an average of 17. This is a 62% improvement.

We showed that on average not less than 17 registers out of the 31 general-purpose registers hold no useful contents on the Alpha architecture. At first sight, this contradicts the common wisdom that compilers can optimally exploit the uniform register files offered by RISC architectures. Fundamentally, the inefficient use of the available registers directly results from the fact that, due to calling conventions, the register file cannot really be considered uniform under the most often used program development practice of separate compilation.

## Acknowledgements

For this research, Bjorn De Sutter was supported by the Fund for Scientific Research - Flanders (Belgium) as a postdoctoral research fellow. Bruno De

Bus was supported by a grant from the 'Flemish Institute for the Promotion of the Scientific Technological Research in the Industry' (IWT). Part of the research reported in this paper was supported by Ghent University, which is a member of the HiPEAC network.

## References

[1] S. Muchnick, Advanced Compiler Design & Implementation, Morgan Kaufmann Publishers, 1997.

[2] S. Debray, W. Evans, R. Muth, B. De Sutter, Compiler techniques for code compaction, ACM Transactions on Programming Languages and Systems 22 (2) (2000) 378–415.

[3] D. W. Goodwin, Interprocedural dataflow analysis in an executable optimizer, in: Proceedings of the ACM SIGPLAN 1997 conference on Programming Language Design and Implementation (PLDI), 1997, pp. 122–133.

[4] R. Muth, S. Debray, S. Watterson, K. De Bosschere, Alto : A link-time optimizer for the Compaq Alpha, Software Practice and Experience 31 (1) (2001) 67–101.

[5] A. Srivastava, D. W. Wall, A practical system for intermodule code optimization at link-time, Journal of Programming Languages 1 (1) (1992) 1–18.

[6] B. De Bus, D. Chanet, B. De Sutter, L. Van Put, K. De Bosschere, The design and implementation of fit: a flexible instrumentation toolkit, in: PASTE '04: Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2004, pp. 29–34.

[7] A. Srivastava, A. Eustace, ATOM: a system for building customized program analysis tools, in: Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation (PLDI), 1994, pp. 196–205.

[8] S. Debray, R. Muth, M. Weippert, Alias analysis of executable code, in: Proceedings of the ACM 1998 Symposium on Principles of Programming Languages (POPL'98), 1998, pp. 12–24.

[9] R. Muth, Alto: A platform for object code modification, Ph.D. thesis, University Of Arizona (1999).

[10] B. De Sutter, B. De Bus, K. De Bosschere, Link-time binary rewriting techniques for program compaction, ACM Transactions on Programming Languages and Systems 27 (5) (2005) 882–945.

[11] J. Levine, Linkers & Loaders, Morgan Kaufmann Publishers, 2000.

[12] B. De Sutter, B. De Bus, K. De Bosschere, Sifting out the mud: low level C++ code reuse, in: Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002, pp. 275–291.

[13] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, K. De Bosschere, Link-time optimization of ARM binaries, in: Proceedings of the 2004 ACM SIGPLAN-SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'04), 2004, pp. 211–220.

[14] B. De Bus, D. Kästner, D. Chanet, L. Van Put, B. De Sutter, Post-pass compaction techniques, Communications of the ACM 46 (8) (2003) 41–46.

[15] R. Cohn, D. Goodwin, P. Lowney, N. Rubin, Spike: An optimizer for Alpha/NT executables, in: Proceedings of the USENIX Windows NT Workshop, 1997, pp. 17–24.

[16] G. Balakrishnan, T. Reps, Analyzing memory accesses in x86 executables, in: Proc. Int. Conf. on Compiler Construction, 2004, pp. 5–23.

[17] C. Linn, S. Debary, G. Andrews, B. Schwarz, Stack analysis of x86 executables, http://www.cs.arizona.edu/people/debray/papers/stack-analysis.ps.