The UQ Retargetable Binary Translator

Principal Investigators
Cristina Cifuentes
Centre for Software Maintenance
Department of Computer Science and Electrical Engineering
The University of Queensland

Norman Ramsey
Department of Computer Science
University of Virginia

Senior research assistant
Mike van Emmerik
CSEE - UQ

PhD student
David Ung
CSEE - UQ

Honours students
1997: Shane Sendall
CSEE - UQ

Research assistants
Term 1 99: Doug Simon, Trent Waddington
Summer 98-99: Ian Walters, Shawn Coyle, Trent Waddington
Summer 97-98: Shane Sendall, Doug Simon
CSEE - UQ

January 29, 1999

Abstract

Binary translation is a relatively new field of research. Existing binary translators rely on machine-dependent analyses that are written by hand. Such analyses are required to identify procedures, to find the code attached to those procedures, to find the targets of indirect branches, and to identify call sites and parameters. Redevelopment and hand implementation of such analyses makes it difficult and time-consuming to develop binary translators for new platforms.

In contrast, we propose to circumvent these problems by developing a machine-independent framework for analyzing binary codes. The framework will include register transfer lists (RTLs)—a machine-independent, machine-level representation of the effects of instructions—together with machine-independent analyses of code in RTL form.

We are also interested in applying these analyses to the translation of real binary programs on a variety of hardware platforms, including SPARC, x86, and Alpha. We have developed formal descriptions of the syntax and semantics of instructions on the first two platforms, and we plan to use the descriptions to derive the machine-dependent components used to translate between binary codes and RTLs. These automatically generated components will be useful not just in binary translators, but also in other binary manipulation tools, hence making it easier to port them to other architectures.

In the long term, we expect this framework to enable the development of new analyses to help improve the quality of automatically translated code.

This report attempts to document the design and implementation issues dealing with **uqbt**, the University of Queensland's retargetable binary translator for multiplatform operating systems.

We will *try* to keep this document up to date!

Acknowledgments

This work has been possible thanks to funding from the Australian Research Council (ARC Grant No.A49702762), Sun Microsystems Laboratories (Sponsored by Neil Wilhelm), and The University of Queensland (Grant No.97/UQNSRG009G).

Contents

List of Figures

Chapter 1

Call Signature Recovery

Design and documentation: Cristina [27 Jan 99], Implementation: Doug

The calling convention for a particular operating system is provided by the Application Binary Interface (ABI). Compiler writers need to adhere to the ABI calling convention if their code is to be compatible with that generated by other compilers. This does not preclude compiler writers from creating their own calling conventions and using them in code generated by their compiler—so long as the code is not to be shared with other object codes. In this work we are initially concerned with specifying the calling convention as described by the ABI and to recover the call signature for each user-code procedure of a program.

The aim of this work is to recover high-level CALL and RET (return) intermediate instructions rather than relying on low-level RTLs to describe their semantics. This allows us to abstract from machine dependencies; for example, in the way that parameters are passed to a procedure and how they are returned. The signature of the two instructions to be recovered are:

- CALL <addr> [<return type>] <list of parameters>
  Where the actual parameters have a type only, and the formal parameters have a type and a value (i.e. an expression).
- RET [<return value>]

The type of any parameter or return value is a low-level one; the types considered so far are:

- integer (possibly with a size if found useful)
- address (pointer to instruction or to data)
- float (with a size: 1 (single), 2 (double), 4 (quad))

This chapter does not deal with the recovery of these low-level types; you need only assume they exist for now.

A complete specification for SPARC and Intel can be found in Appendix ??.

1.1 Calling Convention Specification

The Unix System V ABI [1] describes the application binary interface rules to be followed on different machines when implementing Unix System V. The SPARC [3] and Intel [2] processor supplements describe machine-specific rules to be followed as part of implementing the ABI interface. Such rules include how to pass parameters to a procedure and how to return value(s) from a function. It also describes the stack frame and how values on it change during a call. For SPARC, machine dependencies such as the register window mechanism are also explained.

When a compiler generates code for a function call, it first needs to determine where to pass the parameters (on registers or the stack most commonly) and place them on the appropriate locations, it then emits a call to the destination address for the function, the called function sets up its stack frame and allocates enough space for local variables and any other space that the ABI may require it to allocate, then the code for the rest of the function is emitted. On function return, a returned value needs to be placed on certain location(s) and the stack frame needs to be restored to its original form prior to returning. Once returned, the caller determines whether to move the returned value to a register or a variable as needed.

The above description can be summarized in terms of the caller's and callee's prologue and epilogue. The caller's prologue places the actual arguments and invokes (calls) the function. The callee's prologue sets up the stack frame. The callee's epilogue places the return value, restores the stack frame and returns. The caller's epilogue decides what to do with the returned value.

A calling convention language, CCL [4], was developed to specify calling conventions for different languages and compilers. The language is used as part of a retargetable compiler and as such makes use of knowledge known at compile time, such as the number of arguments passed to a function and the types of such arguments.

At the machine-code level, the types of arguments are unknown until analysis is performed on them. Using a specification based on types does not aid in determining their types. For example, the FSA in Figure 1.1 states that for a particular machine, the first integer argument is passed on register `%o0` and the second integer argument on register `%o1`, or alternatively, a double floating point argument can be passed in both `%o0` and `%o1`. Although this makes sense from a compiler's point of view, in machine code we do not see a difference between storing integers, addresses or double floats in registers `%o0` and `%o1`. The only useful information to us is the fact that the registers `%o0` and `%o1` are used to pass arguments. Later analysis on the usage of the passed arguments may determine the difference between integer and double float arguments.

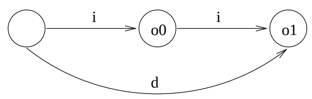

Figure 1.1: Sample FSA to Place Integer and Double Floating Point Arguments on Registers

The following sections describe what needs to be specified and gives examples for the SPARC and Intel architecture. *Note that the notation is still in change. The following sections will be rewritten once the notation is stable.*

SPARC

The standard stack frame for SPARC procedures includes a 16-word area to save the register window, one word to store the return address of a struct/union, 6 words for the callee to save the first 6 arguments to the stack, n words for input arguments 7 and above, and an area to store locals. Figure 1.2 shows the standard stack frame for SPARC.

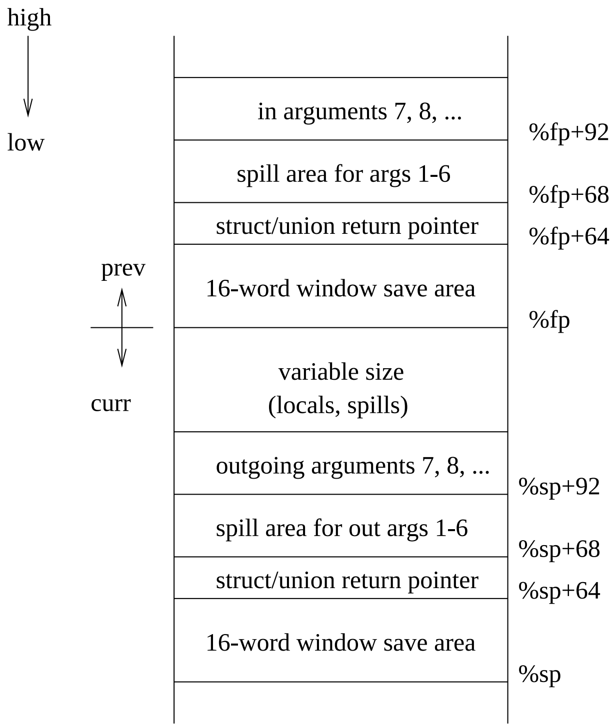

Figure 1.2: Standard Stack Frame for SPARC Code

1.1.1 Specification of the Caller and Callee Prologue and Epilogue

On the SPARC, there are two different ways of invoking a function based on the return value of the function. A `call` instruction is always used, however, if the return value is a structure, union or a quad floating point value, the instruction following the delayed instruction of the `call` needs to be a `unimp` instruction. The immediate 22 bits of the `unimp` are used to check the size of the returned value.

```
constructors
    caller_prologue1 addr     is  call__ (addr)

    caller_prologue2 addr     is  call__ (addr);
                                  _;             # any instruction
                                  UNIMP (imm22)
```

The callee only has one type of prologue, if used at all, alternatively, it may be skipped.

```
    callee_prologue1 simm13   is  SAVE ("%sp", imode(simm13), "%sp")
```

The callee has more than one way to return based on whether the return value is already in the relevant register or needs to be calculated or moved from another register. If the function uses tail optimization, the return instruction uses the `%o7` register instead of the `%i7` register. And, if the function returns a structure, union or quad, two instructions after the returned `pc` value need to be skipped (instead of only one).

```
    callee_epilogue1          is  ret;
                                  restore_

    callee_epilogue2 rs       is  ret;
                                  RESTORE (rs, rmode("%g0"), "%o0")

    callee_epilogue3          is  retl

    callee_epilogue4          is  JMPL (dispA ("%i7",12), "%g0);
                                  restore_

    callee_epilogue5 rs       is  JMPL (dispA ("%i7", 12), "%g0);
                                  RESTORE (rs, rmode ("%g0"), "%g0)

    callee_epilogue6          is  JMPL (dispA ("%o7", 12), "%g0)
```

Once the callee returns, any return values are in the right place and the stack has been restored, so there is no caller epilogue.

1.1.2 Specification of the Caller-Callee Interface

Only certain caller and callee prologues match some callee epilogues. The caller-callee interface describes which combinations are allowed.

```
caller-callee interface
    interface1 addr, simm13, rs   is
        caller_prologue1 addr  +  callee_prologue1 simm13
                               + (callee_epilogue1 |
                                  callee_epilogue2 rs)

    interface2 addr, rs  is
        caller_prologue1 addr  +  callee_epilogue3

    interface3 addr, simm13, rs   is
        caller_prologue2 addr  +  callee_prologue1 simm13
                               + (callee_epilogue4 |
                                  callee_epilogue5 rs)
```

```
    interface4 addr, rs  is
        caller_prologue2 addr  +  callee_epilogue6
```

These interfaces do not describe all code generated by compilers, however, new constructors can be created for the callee to overcome this problem. A common callee prologue and epilogue is when the called function is tail optimized and hence does not use a new register window but only allocated (and deallocates) space on the stack:

```
constructors
    callee_prologue2 simm13    is  ADD ("%sp", imode(simm13), "%sp")

    callee_epilogue7 simm13    is  retl;
                                   SUB ("%sp", immode(simm13), "%sp")

caller-callee interface
    inteface5 addr, simm13  is
        caller_prologue1 addr  +  callee_prologue2 simm13
                               +  callee_epilogue7 simm13
```

1.1.3 Matching of the Caller-Callee Interface

This section probably doesn't go here but I just want to tell you know.

Once a caller-callee interface has been matched, the high-level CALL and RET instructions are created instead of the caller's prologue and the callee's epilogue RTLs. The callee's prologue provides some extra information to the procedure object (PROC) in relation to the size of local variables in that procedure.

1.1.4 Specification of Locations used for Parameter Passing

As discussed in §1.1, at the machine-code level we cannot distinguish the variables and types that were used when placing parameters on appropriate parameter-passing locations. On SPARC, all parameters are copied by instances of one word, hence, a double floating point value is copied as two words, in much the same way as two individual integers or even two addresses are copied. Low-level type information can be retrieved from usage at the called site.

The ABI specifies which locations are suitable for passing parameters, and the order of usage of those locations. We can describe such locations by means of a simple grammar.

```
caller parameters
    params      -> params_regs params_stk
    params_regs -> "%o0" "%o1" "%o2" "%o3" "%o4" "%o5"
    params_stk  -> ["%sp"+92+n]  {n = 4 * _}
```

The `params` rule states that you first need to place parameters on registers and then use stack locations. The `params_regs` rule states which registers can be used and in which order. The `params_stk` rule states that offsets from the stack location `%sp+92` are used in multiples of 4 bytes. *The offsets should be limited to positive offets only.*

Note that the above rules do not follow EBNF rules completely.

1.1.5 Specification of Return Locations

Return values need to be placed in specific registers depending on the type of the value to be returned.

```
callee return
    integer -> "%i0"
    address -> "%i0"
    float   -> "%f0"
    double  -> "%f0"."%f1"
```

Note that double refer to a float of size 2 according to the introduction in this chapter; those sizes are wrong but you get the idea.

Note also that structures, unions and quads are returned as an address which is stored in `%fp+64`. The code ensures the right information has been stored at the relevant variable, hence we do not need to treat these types as special ones; all we see is an address to data being returned. However, the way the ABI describes the return of these types involves the use of a "hidden" parameter (that in `%sp+64`). Therefore, for interfaces 3 and 4, we make this parameter explicit as the first one, and the view change (see §1.1.6) will ensure that the right argument is copied (i.e. `%fp+64`).

In the following SPARC assembly code, the right-hand side shows the transformed code stored in our intermediate representation. The hidden parameter is made explicit as the first one and its low-level type is address.. *We will still need to know that this is a special parameter as for library functions that return a structure, union or quad, we cannnot pass this extra parameter but just need to place it in the right stack offset.*

```
add %fp, -32, %o0           add %fp, -32, %o0
st %o0, [%sp+64]            st %o0, [%sp+64]
call getInfo                getInfo ([%sp+64], ...)
nop                         sethi %hi(.LLC4), %o1
unimp 16                    ...
sethi %hi(.LLC4), %o1
...
```

1.1.6 Specification of the View Change

Can anyone think of a better name for this? This one comes from Bailey's work.

The view change indicates that some locations appear to have moved. Examples include change of offsets from the stack pointer once the stack frame is allocated/deallocated, and SPARC's register windows. This information is stored in the callee's prologue and epilogue.

The prologue changes the register window the procedure is to use, by sharing the 8 output registers as input registers. The new window provides for new 8 local and 8 output registers. The effect of the `SAVE` instruction is to also allocate the size of the stack frame.

```
view change
    callee_prologue1 simm13  ->  %o0 -> %i0
                                 %o1 -> %i1
                                 %o2 -> %i2
```

```
%o3 -> %i3
%o4 -> %i4
%o5 -> %i5
%sp -> %fp
%o7 -> %i7
new %l0..%l7
new %o0..%o7
%sp -> %sp + simm13
```

The epilogue normally restores the register window (if a new one was used) and returns a value (if it needs to be moved from another register to the default one, or if some computation needs to be performed on it). By restoring the stack, the `%sp` pointer is changed to point to the caller's stack frame.

```
view change
    callee_epilogue1
    callee_epilogue4  ->  %i0 -> %o0
                          %i1 -> %o1
                          %i2 -> %o2
                          %i3 -> %o3
                          %i4 -> %o4
                          %i5 -> %o5
                          %fp -> %sp
                          %i7 -> %o7

    callee_epilogue2 rs
    callee_epilogue5 rs  -> %i0 = rs
                            %i0 -> %o0
                            %i1 -> %o1
                            %i2 -> %o2
                            %i3 -> %o3
                            %i4 -> %o4
                            %i5 -> %o5
                            %fp -> %sp
                            %i7 -> %o7
```

As can be seen, not all epilogues create a view change, and different epilogues may create the same view change.

Intel

The standard stack frame of a procedure includes space for arguments, the return address of the caller, the frame pointer value of the caller (`%ebp`), and enough space for local variables and spilled values (including register that need to be preserved across procedure calls). Figure 1.3 shows the standard stack frame for Intel code.

The following specifications for Intel are not accurate in the sense that they do not explicitly refer to the SLED names in the NJMC specs. This is because there are many different patterns that match

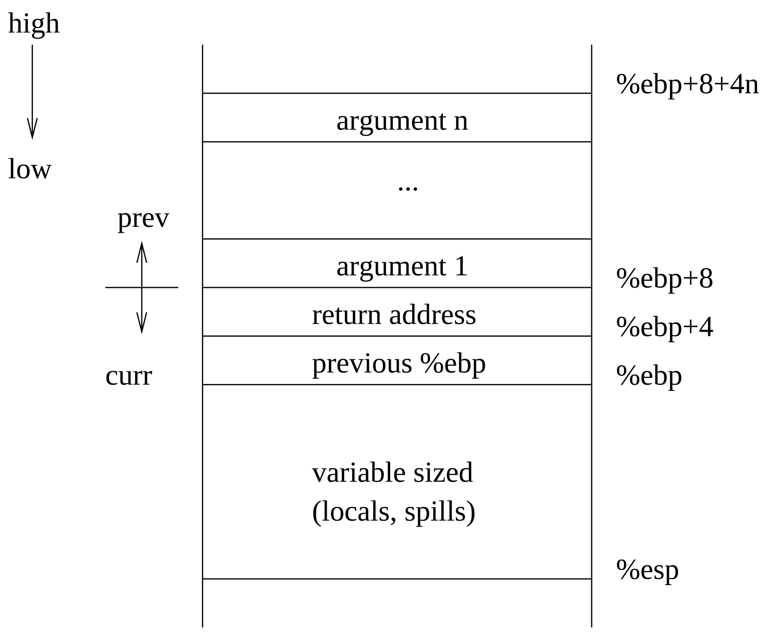

Figure 1.3: Standard Stack Frame for Intel Code

one particular assembly instruction. I am normally interested in the assembly instruction, and hence all the underlying patterns that match it. However, since the Unix System V is a 32-bit interface, not all patterns will be used, only those relevant to 32-bit addressing modes and registers. In the specs presented below, I have used `i16` for "immediate", however, I doubt it is a 16-bit quantity; this needs to be checked.

1.1.7 Specification of the Caller and Callee Prologue and Epilogue

On Intel x86, there is only one way to invoke a procedure, even if the procedure is to return a value or a structure. The `call` instruction is used, and although this assembly instruction maps to one of five different machine instructions, only one is used for direct calls; the intra-segment direct call (CALL.JVOD reloc32). *I think this is the one; this needs to be checked.* For indirect calls (i.e. via a register), the intra-segment indirect call is used (CALL.EVOD modrm). Indirect calls require extra analysis to determine the target address of the call; this is addressed in a different document. For now assume all calls are direct.

```
constructors
    caller_prologue1 addr     is  CALL addr
```

The callee sets up the frame base pointer and the stack pointer, as well as optionally allocating space on the stack for locals and spilled values. Further, the contents of registers `%edi`, `%esi` and `%ebx` need to be preserved across procedure calls. If these registers are to be used by the callee, they need to be spilled to the stack as part of the prologue. In the case where structures or unions are to be returned, their return address is passed as the first (hidden) argument of the call. It need also be returned in `%eax`, hence it needs to be taken from the stack and stored in a register (`%eax` in the below `callee_prologue3` specification; we need to check if that's always the case or whether it gets stored in other registers in the mean time). Note that in the following spec, `i` stands for the immediate number of bytes optionally pushed onto the stack (i.e. 4 times the number of optional pushes). This value is needed in order to determine the actual change of size on the stack.

```
callee_prologue1 i16, i   is  PUSH ebp;
                              MOV ebp, esp;
                              SUB esp, i16;
                              [ PUSH edi |
                                PUSH esi |
                                PUSH ebx ]*

callee_prologue2 i        is  PUSH ebp;
                              MOV ebp, esp;
                              [ PUSH edi |
                                PUSH esi |
                                PUSH ebx ]*

callee_prologue3 i16, i   is  POP eax;
                              XCHG eax, esp;
                             (callee_prologue1 i16,i | callee_prologue2 i);
                              MOV [ebp-4], eax
```

The callee uses more than one way to return based on whether a structure/union will be returned or not. The standard epilogue restores any of the registers that need to be preserved across a procedure call (i.e. `%ebx`, `%esi` or `%edi`), restores the stack pointer and the frame pointer, and returns to the caller's return address. Restoring registers to be preserved across procedure calls can be done in one of two ways; by popping them from the stack (`callee_epilogue1`), or by indexing into the stack directly (`callee_epilogue2`). It has also been noticed that some compilers generate a `LEA` (load effective address) at the start of the epilogue, to ensure the stack pointer is pointing to the right address, even if this instruction is redundant (as has been seen in gcc -O2 code).

Procedures that return structures or unions require to return the address of the struct/union in register `%eax`. This can also be done in two ways; by popping the value from the stack, or by indexing into the stack.

```
callee_epilogue1          is  [ LEA esp, ebp-n ] ;
                              [ POP ebx |
                                POP esi |
                                POP edi ]* ;
                              LEAVE;
                              RET    { n = 4 | 8 | 12 }

callee_epilogue2          is  [ LEA esp, ebp-n ] ;
                              [ MOV ebx, ebp-n |
                                MOV esi, ebp-n |
                                MOV edi, ebp-n ]* ;
                              LEAVE;
                              RET    {n = 4 * _}
```

```
callee_epilogue3         is  [ POP ebx | POP esi | POP edi ]* ;
                             POP eax ;
                             LEAVE ;
                             RET .

callee_epilogue4         is  MOV eax, [ebp-4] ;
                             callee_epilogue1 | callee_epilogue2 .
```

On callee return, the stack needs to be restored by the caller in order to remove the parameters that were passed on the stack. Restoring of the stack is done by modifying the value of the stack pointer by a certain number of bytes, or by popping values from the stack a certain number of times (4 bytes at a time). Either way will tell us how many bytes are restored from the stack.

```
caller_epilogue1 i16     is  ADD esp, i16

caller_epilogue2         is  (POP _)*
```

1.1.8 Specification of the Caller-Callee Interface

The following caller-callee interface describes the combinations that are allowed between callers and callees. Note that the callee epilogue is optional in all cases as optimizing compilers may delay the restoring of the stack for several calls. This means that once a caller epilogue is met, the number of bytes restored are for a number of parameters to procedures rather than just for the callee procedure.

```
caller-callee interface
    interface1 addr, i16, i   is
        caller_prologue1 addr  +  callee_prologue1 i16, i
                               + (callee_epilogue1 |
                                  callee_epilogue2 )  +
        [ caller_epilogue1 i16 |
          caller_epilogue2 ]

    interface2 addr, i         is
        caller_prologue1 addr  +  callee_prologue2 i
                               +  ( callee_epilogue1 |
                                    callee_epilogue2 ) +
        [ caller_epilogue1 i16 |
          caller_epilogue2 ]

    interface3 addr, i16, i    is
        caller_prologue1 addr  +  callee_prologue3 i16, i
                               + ( callee_epilogue3 |
                                   callee_epilogue4 ) +
        [ caller_epilogue1 i16 |
          caller_epilogue2 ]
```

1.1.9 Specification of Locations used for Parameter Passing

On Intel x86, all parameters are to be passed on the stack (according to the ABI). We know that some compilers pass parameters on registers; this code is not ABI-compliant though and has not been specified at present time [it will in the future].

Integers and addresses are passed on the stack as one word, floats are passed as 1, 2 or 3 words depending on whether they are using single, double or extended precision, and structs/unions are passed as a series of words, rounded to the nearest multiple of 4.

Note that struct/union is passed as a whole block of words, whereas on SPARC the address of the struct/union was passed as argument. This may imply that we need a low-level type for "block of memory", where this can be a struct, union or an array; but I'm not sure at present whether this is really needed.

```
caller parameters
    params      -> params_stk
    params_stk  -> ["%ebp"+8+n]  {n = 4 * _}
```

The `params` rule states that there is only one way to pass parameters. The `params_stk` rule states that parameters are passed as offsets from the frame pointer, starting at offset 8, and on increments of 4 bytes. *The offsets should be limited to positive offsets only.*

1.1.10 Specification of Return Locations

Values are returned on `%eax` if they are of type integer or an address, or `%st(0)` (and `%st(1)`) if they are of type float. Structs/unions are returned on `%eax`; this is the address of the returned struct/union.

```
callee return
    interface1
    interface2
    interface3 ret integer -> "%eax"
                   address -> "%eax"
                   float   -> "%st(0)"
```

1.1.11 Specification of the View Change

The `entry` and `exit` from a callee define the view change of the stack pointer. I have expressed the view change in terms of the interfaces as this provides us with more information than if callee prologues and epilogues were considered individually; state information can be stored therein.

Note: the SPARC view change will need to be upgraded to this type of spec if we choose this one.

```
view change
    interface1 addr, i16, i  view
        entry  %esp -> '%esp - (i16 + 8 + i)
               %ebp -> '%esp - 8
        exit   %esp -> '%esp + (i16 + 8 + i )
```

```
interface2 addr, i  view
    entry  %esp -> '%esp - 8 - i
           %ebp -> '%esp - 8
    exit   %esp -> '%esp + 8 + i

interface3 addr, i16, i  view
    entry  %esp -> '%esp - (i16 + 12 + i)
           %ebp -> '%esp - 8
    exit   %esp -> '%esp + (i16 + 12 + i)
```

Do we also need to specify the view change for ebp?

1.2 Call Signature Analysis

Call signature analysis is the recovery of parameters (formal and actual) and the return type (if any) for a CALL instruction. This analysis is to be performed on user functions; library functions will be assumed to have been processed by now, either through generation of call signatures from header files or through application of the following analysis to library code.

The analysis is based on interprocedural liveness analysis of uses and definitions of parameter and return locations. Throughout this section we will make use of the SPARC example in Figure 1.4. Issues specific to Intel are discussed in §1.2.3.

1.2.1 Recovery of Parameters

Actual parameters are placed by the caller in one or more of the locations specified by `caller parameters`. The callee will effect the `view change` applicable to its prologue, and will then use the passed parameters directly or place them on the parameter spill area. Either way, the parameters are used before definition within the callee, and this is what tells us that the information in that location was setup elsewhere in the program.

In the example of Figure 1.4, `main` calls `gcd`, using the most frequently used calling convention; `interface1`. At the `call` site, the parameter locations that are live are:

```
live = {%o0, %o1}
```

At the callee site, we apply the `view change` of `callee_prologue1` to the live parameter locations, the stack pointer, and the return address, leading to

```
%o0 -> %i0
%o1 -> %i1
%sp -> %fp
%o7 -> %i7
%sp -> %sp-112
```

For `gcd` we summarize the live-in information for the whole procedure based on the parameter locations (view changed). This gives us

```
gcd:
    save %sp,-112,%sp
    mov %i0,%l0
    cmp %l0,%i1
    bge .LL12
    mov %i1,%i0
    b .LL12
    mov %l0,%i0
.LL6:
    call .rem
    mov %i0,%o1
    cmp %o0,0
    bne,a .LL12
    add %i0,-1,%i0
    mov %i1,%o0
    call .rem
    mov %i0,%o1
    cmp %o0,0
    be .LL10
    nop
    add %i0,-1,%i0
.LL12:
    cmp %i0,1
    bg .LL6
    mov %l0,%o0
.LL10:
    ret
    restore
.LLC0:
    "gcd of %d, %d is %d\n"
```

```
main:
    save %sp,-112,%sp
    mov 10,%o0
    mov 5,%o1
    sethi %hi(.LLC0),%l0
    call gcd
    or %l0,%lo(.LLC0),%l0
    mov %o0,%o3
    mov %l0,%o0
    mov 10,%o1
    call printf
    mov 5,%o2
    ret
    restore
```

Figure 1.4: SPARC Assembly Code for GCD Program

```
liveIn(gcd) = {%i0, %i1}
```

This information tells us that there are 2 parameters, which match the two live parameter locations at the call site, hence the actual parameters to the call are `%o0` and `%o1`. The transformed CALL instruction looks like this:

```
CALL gcd [<ret type>] <(%o0,<type>), (%o1,<type>)>
```

Note that for parameter locations that are on the stack, the view change of these looks as follows:

```
%sp+92+n -> %fp+92+n
```

Hence, accesses to `%fp+92+n` at the callee site are accesses to parameter locations. It is also feasible for the callee site to access these locations using the stack pointer; this is needed for leaf routines but it can also be used in non-leaf routines:

```
%sp+(92+simm13)+n
```

We need to support both views of stack parameter locations.

Fixed vs Variable Number of Arguments

Most procedures take a fixed number of arguments. However, languages like C allow for variable number of arguments to be passed at any one time. The ABI [1] does not place any rules on variable number of parameters, but states that the calling convention rules need to be satisfied; that is, the first 6 word parameters go into registers and the next go onto the stack. Disassembled C code shows that the first thing the callee does is to move all the register parameters to the parameter spill area, and then use them all on the stack (as the stack parameters are contiguous to the spilled register parameters).

It is not clear that in all cases usage analysis of parameter locations at the callee will determine the number of parameters taken by the callee (think of `printf`, it can take any number and the code is bound to be a loop on a string). Also, different invocations of the procedure will take different number of arguments. *I propose we pass all the live parameter locations at the call site in the mean time; we will see from the implementation whether this will cause problems with stack parameter locations.*

1.2.2 Recovery of Return Value

The callee will place a return value on a valid `callee return` location. We note that if anything is placed on a return location, this location will be live-out of the callee. The caller will need to apply the inverse of the `view change` to live-out callee locations. If the caller is to make any use of a returned value, the location where it was stored will be used before being re-defined. Usage is commonly in the form of storing to a local variable or using it as a parameter to another procedure. Once this is determined, the callee's RET instruction is set to return the relevant return location.

For the example of Figure 1.4, the callee, `gcd`, has only one return location live-out, which matches its `callee return` for `interface1`:

```
live-out = {%i0}
```

The caller uses `%o0` before definition, hence this is our return location. The callee's RET instruction can now be transformed to:

```
RET %i0
```

The caller's CALL instruction can now be transformed into an ASSIGN (assignment) instruction as follows:

```
%o0 = CALL gcd <(%o0,<type>), (%o1,<type>)>
```

Should we introduce a HL ASSIGN or shall we use RTL assign instead?

Returned Values not Used

Returned function values are not always used by the caller. In such cases, different invocations of the same procedure will show different return location usage. In these cases, we go for the more general one (i.e. the procedure returns a value) and we annotate each actual call with whether the return value is used or not. In this way, the signature for the procedure is correct.

1.2.3 Issues Relating to Intel Call Signature Analysis

The nature of Intel passing parameters on the stack means that optimizing compilers may delay the restoring of the stack until after several calls to (different) procedures have been emitted. This means that the caller epilogue is optional, and where available, it may not necessarily match the number of bytes passed as parameters to the caller but may also include bytes used in a previous call. We can still make use of liveness analysis to determine which arguments are passed to each procedure nevertheless. For the purposes of illustration, we will make use of Figure 1.5, an optimized version of the GCD program for Intel.

```
gcd:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%ebx
    movl 12(%ebp),%edi
    movl %edi,%esi
    cmpl %edi,%ebx
    jge .L12
    movl %ebx,%edi
    jmp .L12
.L6:
    movl %ebx,%eax
    cltd
    idivl %edi
    testl %edx,%edx
    jne .L7
    movl %esi,%eax
    cltd
    idivl %edi
    testl %edx,%edx
    je .L13
.L7:
    decl %edi
.L12:
    cmpl $1,%edi
    jg .L6
.L13:
    movl %edi,%eax
    leal -12(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    leave
    ret
.LC0:
    "gcd of %d, %d is %d\n"
```

```
main:
    pushl %ebp
    movl %esp,%ebp
    pushl %esi
    pushl %ebx
    movl $10,%esi
    movl $5,%ebx
    pushl %ebx
    pushl %esi
    call gcd
    pushl %eax
    pushl %ebx
    pushl %esi
    pushl $.LC0
    call printf
    leal -8(%ebp),%esp
    popl %ebx
    popl %esi
    leave
    ret
```

Figure 1.5: Intel Optimized Assembly Code for GCD Program

Procedure `main` calls `gcd` using the calling convention specified in `interface2` with an immediate value of 12 bytes. The calling convention does not include a caller epilogue. At the `call` site, the *parameter locations* that are live are:

```
live = {[%esp], [%esp+4]}
```

Applying the `view change` for `interface2` to the live parameter locations we get:

```
['%esp]   -> [%esp+20]
['%esp+4] -> [%esp+24]
```
and
```
['%esp]   -> [%ebp+8]
['%esp+4] -> [%ebp+12]
```

Procedure `gcd` has the following set of parameter locations live on entry, and return locations live on exit:

```
liveIn = {[%ebp+8], [%ebp+12]}
liveOut = {%eax}
```

The liveIn parameters match the ones that were live on entry, hence we can safely assume that 2 words (8 bytes) were passed as arguments. The call is transformed to the following HL instruction:

```
CALL gcd [<ret type>] <([%esp],<type>), ([%esp+4],<type>)>
```

which is further transformed into what was actually placed at those stack locations (i.e. this information needs to be stored previously):

```
CALL gcd [<ret type>] <(%esi,<type>), (%ebx,<type>)>
```

The liveOut information tells us that `%eax` is returned. Further, at the caller's site, the value of `%eax` is used prior to definition. Even if this value was not used, the ABI states that return locations should only be set to a value if they are intended to return a value, as there is no type checking on this interface. The return instruction in `gcd` is changed to

```
RET %eax
```

and the caller's site call is changed to

```
%eax = CALL gcd <(%esi,<type>), (%ebx,<type>)>
```

Although there is no caller epilogue to restore the stack, we have determined the right arguments to this call.

The next call that `main` performs is to the library procedure `printf`. In this case, if we had signatures for `printf` we could only be assured of one fixed parameter (an address) and maybe some more parameters, as this is a variable argument procedure. Also, the calling convention does not specify a caller epilogue in this case either, hence we cannot determine exactly how many bytes are passed on the stack to this call. The best that can be done is to pass *all* parameter locations that are live at the call site:

```
live = {[%esp], [%esp+4], [%esp+8], [%esp+12], [%esp+16], [%esp+20]}
```

When replacing this information into the HL call, we get:

```
CALL printf <(.LC0,<addr>), (%esi,<type>), (%ebx,<type>), (%eax,<type>),
             (%esi,<type>), (%ebx,<type>)>
```

Note that in this case, the last two arguments are still technically live as the stack wasn't restored. Although we are passing them to `printf`, the code within `printf` will not use them as they were not expected (by checking the string `.LC0`).

If however the stack was restored after the call to `printf`, the following code could have been emitted to restore both calls made by `main`:

```
addl $24, %esp
```

Which would account for 8 bytes that we already know `gcd` takes as arguments, and 16 bytes for `printf`. This type of arithmetics will allow us to determine the number of bytes passed to variable argument procedures in some cases (bearing in mind that each time a different number of arguments may be passed).

1.3 Type Recovery Analysis

Low-level type recovery is tightly coupled with the call signature recovery analysis. We perform type recovery for the purposes of determining low-level types for parameters and return values, so that these can be represented in an abstract way in our intermediate representation.

As mentioned in the introduction, we work with 3 base low-level types; integer, address, and float. For the purposes of analysis, the address type is divided into two types; address to instruction and address to data. Given that the ABI [1] states that floats and integers (of any size) are passed on integer registers, and the fact that addresses are also integer numbers, our default data type for any location is an integer (i.e. the bottom of the lattice). In a lattice representation, if a condition holds true, a type can be promoted to one higher up the lattice. In our case, we have a few types which can be represented in a very simple lattice as per Figure 1.6.

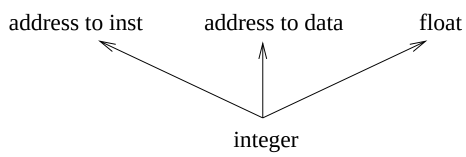

Figure 1.6: Lattice of Low-Level Types

Types are determined based on usage of a location across a live range. Given that a particular location can be re-used as different variables of different types by a compiler, the only safe assumption is that a live range for a given location will have *one* type.

A complete analysis next week; the next subsections give you the general idea.

1.3.1 Promotion to Float Type

A float type is determined when placing a location onto a floating point register. The size of a float is determined based on the usage of the floating point register(s). For example, when a double register is used, then its size is twice the word size (64 for SPARC). *I think we should represent the size in terms of the number of bits in the float; that is, 32, 64 and 128 for SPARC.*

1.3.2 Promotion to Address to Data

An address is an integral number that is a virtual address within the range of addresses of the data segment(s) of the program or of a symbolic name.

The data segments of Elf binaries are:

- Sections that contain read-only data: .rodata, .rodata1
- Sections that contain initialized read-write data: .data, .data1
- Section of uninitialized read-write data: .bss

The dynamic symbol table, present in all programs, even stripped programs, contains names of symbols and their virtual addresses.

1.3.3 Promotion to Address to Instruction

An address to an instruction is a virtual address that falls within the text section(s) of the program. For Elf binaries, these are:

- .text
- .plt

The dynamic symbol table also contains names of dynamic library functions.

Bibliography

[1] *Unix System V: Application Binary Interface.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[2] *Unix System V: Application Binary Interface, Intel Processor Supplement.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[3] *Unix System V: Application Binary Interface, SPARC Processor Supplement.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[4] M.W. Bailey and J.W. Davidson. A formal model and specification language for procedure calling conventions. In *ACM Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, CA, January 1995.