IR: An Instrumentable, Architecture-Neutral Intermediate Representation For Valgrind

Julian Seward, jseward@acm.org

Revision 3

Revision history

18 Sept 04 (rev 1): Document created.

20 Sept 04 (rev 2): Fix some typos. Clarify semantic effects of dirty helper calls.

19 Apr 05 (rev 3): Track changes to the code base over the past few months. Note, description of condition-code thunk fields for amd64/x86 is still out of sync with the code.

1 Scope and Audience

This report describes an instrumentable, architecture-neutral intermediate representation intended to form the basis of a CPU porting framework for Valgrind. What this report contains is:

- A description of the problem and requirements
- An outline of the existing UCode-based framework, and its limitations
- A complete definition of the new IR representation
- Discussion of possible IR-level optimisations
- How to convert IR back into machine instructions
- How to add support for a new architecture
- Outline of how the IR can provide support for AMD64/x86

This report is required reading for anyone wishing to understand how the core of portable Valgrind works. It's also required reading for anyone thinking of porting Valgrind to a new CPU architecture. Familiarity with the existing UCode scheme is an advantage but not required. Some basic level of familiarity with usage and abilities with the existing Valgrind is assumed. Familiarity with CPU architecture, machine code and compiler technology basics is assumed.

2 Background and Rationale

2.1 Basic structure of Valgrind

Reduced to its bare essentials, Valgrind is a fast, instrumentable instruction-set simulator aimed at running user-mode code only. At run-time, Valgrind translates basic blocks of machine code, as they are required by program flow, producing replacement blocks of code which also contain instrumentation code created by the debugging or profiling tool in use. These replacement blocks of code are cached and executed. None of the original program code is executed, only translations of it are.

Valgrind has a carefully designed plugin architecture, with a core which does almost all of the hard work of running the simulation, and small, modular tools, which can add instrumentation code suitable for the debugging/profiling task in hand. This division of work makes the tools simple(r), in that they do not have to be concerned with the myriad details of dealing with the x86 instruction set, coordinating and caching translations, signals, system calls, threads, error management, memory management, startup and shutdown, and dealing with users -- the core does all that. Conversely, tools can add instrumentation code of their own which the core has no understanding of, and need not be concerned with.

Once familiar with the structure, a new simple tool (eg one which counts the number of floating-point instructions executed) can be written in an hour. Complex tools (eg Memcheck, Cachegrind, Helgrind) can be prototyped in a couple of weeks. This top-level structuring of Valgrind, and many other aspects, are described in the overview paper [Nethercote & Seward 03] and further in [Seward 02].

2.2 The existing UCode framework

This document is concerned with the core dynamic translation and instrumentation mechanism. A key feature of current Valgrind is the use of UCode as an intermediate representation of x86 code. Here, we shall not describe UCode in detail, as that is covered in [Seward 02] and [Nethercote & Seward 03] -- instead this section summarises the important aspects of UCode.

The x86 instruction set is large, complex, inconsistent, has many instructions with strange semantics, and has multiple encodings of the same instruction (there are at least 8 different forms of integer addition, for example).

Valgrind's tools need to be able to analyse in detail the meaning of sequences of x86 instructions, so they can create suitable instrumentation. Doing this directly on x86 code would be very difficult. Instead, Valgrind translates x86 code into a simpler form, a RISC-like code called UCode. Tools analyse UCode and create extra instrumentation code in UCode form. Valgrind's core therefore does the hard work of translating from x86 code to UCode (before instrumentation) and from UCode back to x86 (after instrumentation). The tools only need to understand UCode and are insulated from almost all x86-specific details.

As an example of the value of this, almost any interesting tool (memory debugger, thread debugger, cache profiler) is going to want to observe and specially handle, accesses to memory. On x86, most instructions may (optionally) reference memory, and so building such a tool directly analysing x86 code means it would have to handle literally hundreds of different instructions. In UCode, only a handful of UCode instructions access memory, so building the tool on UCode makes the problem two orders of magnitude easier.

UCode simplifies the tool-writer's life in other ways:

- All variants of the same primitive operation are merged into one. For example, the innumerable ways of encoding an add on x86 are all mapped to the single UCode ADD instruction.
- UCode abstracts away from guest registers. The registers of the machine being simulated (the guest) are stored in memory on the machine doing the simulation (the host), and are accessed from the UCode instruction stream using special GET and PUT uInstructions.
- UCode abstracts away from host registers. As is standard in compiler technology, UCode provides the illusion of an infinite supply of 'virtual' or 'pseudo' registers. Tools can generate instrumentation code (as UCode) using as many of these virtual registers as they like, without concern for how they are mapped back into real registers on the host machine. Doing that mapping (register allocation) is the responsibility of the Valgrind core.

UCode is one of the key ideas which makes Valgrind viable. In short, UCode exposes to tools what it is the code fragments compute, whilst hiding most x86-specific details. The use of a simple intermediate program representation to aid analysis and transformation is a basic concept in compiler technology, dating back three decades and probably more.

Interestingly, if irrelevantly, by the mid-90's, even the hardware design community had concluded that dealing with x86's complexity head-on was no longer viable. Almost all modern x86 CPUs (Pentium Pro, II, III, 4, M, AMD K6, Athlon, Opteron, all Transmetas) consist of a fast out-of-order superscalar core executing a simple RISC instruction set, and wrapped around that, a layer of hardware to translate x86 instructions into this RISC instruction set, so as to preserve the illusion that the hardware executes x86 instructions directly.

2.3 Limitations of UCode

UCode was conceived as 'the simplest thing that works' in order to facilitate building the current x86-only Valgrind. With some minor tuning since then it has served well. Nevertheless, it has significant shortcomings, which we seek to remedy in IR's design:

- Although UCode hides most x86-specific details, it is still strongly tied to x86. The UInstrs are x86-specific, and there are x86-specific kludges, particularly the handling of condition codes, floating point instructions, and MMX/SSE/SSE2 support. Translating any other instruction set, eg PPC, into UCode as it stands is not viable. To do so would require adding at least some eg PPC-specific extensions to UCode.
- As a result of the previous point, any attempt to support multiple architectures via an extended version of UCode will mean that all the tools are exposed to architecture-specific details. This is bad, since it leads to an N x M (architectures x tools) maintenance headache. Adding a new architecture would entail updating all tools. Conversely, adding a new tool would require making it aware of all supported architectures. This kind of non-modularity is strongly to be avoided. What is needed is an intermediate representation which is truly independent of the host and guest architectures.
- UCode provides insufficient description of SIMD operations, hence Memcheck cannot properly instrument them, and as a result may falsely report errors in MMX/SSE/SSE2 code. This will become more of a problem as compilers more routinely generate SIMD code (and yes, even gcc is getting there [Naishlos 04]).
- UCode is microarchitecturally naive. For technical reasons, UCode frequently and unavoidably swaps the %eflags (x86 condition code register) state between the real and simulated CPUs. What was discovered too late in the design cycle is that this is an expensive operation on PII/III/4, Athlon -- costing between 10 and 20 cycles. The same problem also afflicts the FPU simulation.
- UCode is a flat RISC-like code, not entirely unlike SPARC code. This has the advantage of simplicity. Unfortunately, it forces use of a naive, macro-expanding instruction selector (UCode -> x86 conversion) which often generates poor code. Instruction selectors based on tree pattern matching ideas from the late 80s / early 90s [FHP 92, Fraser & Hanson 95] are easy to implement and produce better code, but UCode precludes their use.
- UCode hardwires the assumption that the guest (machine being simulated) and host (machine doing the simulation) CPU architectures are the same. It therefore rules out any future possibility of cross-architecture debugging, something which is important in the embedded arena.
- UCode offers no support for optimisation across multiple basic blocks (BBs) - it deals with each BB in isolation. Optimising across multiple BBs offers significant potential for improving performance.
- A problem with the implementation, rather than the concept, of UCode: the UCode machinery is deeply wired into the rest of Valgrind. This is a mistake: it makes it impossible to debug and profile the translator/instrumentors using a simple test driver. Any new implementation should be supplied as a standalone library.

IR is not merely a fixed-up version of UCode. It is a new design which addresses all the above issues. Those familiar with UCode will see how and where IR draws inspiration from UCode.

Before looking further at IR, we must make a digression to explore some wider aspects of the design space.

2.4 Copy-Annotate versus Disassemble-Synthesise

Valgrind takes basic blocks of machine code and produces code which achieves the same as the original blocks, but also does some extra "shadow" computation as required by the tool in use. Stepping back from the specifics of UCode and IR, there are fundamentally two ways to do this:

- Disassemble-then-(re)Synthesise. As described above in relation to UCode, the incoming code is first unpicked (disassembled) into an intermediate representation which makes it clearer what is being computed. Instrumentation is added to the intermediate representation, which is then re-synthesised into final machine code.
- Copy-and-Annotate is an alternative way to think about the problem. The incoming code is copied almost verbatim to the output. Along the way, each instruction is tagged with an annotation describing what it does in enough detail that tools can generate instrumentation code. The original instructions and instrumentation code are suitably interleaved, generating the final machine code.

Although we will come down firmly in favour of the disassemble-synthesise approach, an examination the merits of both schemes is instructive.

One critical problem with disassemble-synthesise is achieving complete coverage of all needed instructions. A typical intermediate representation will provide some way to describe basic operations such as add, subtract, etc, memory accesses, and control flow. Most instructions can indeed be described adequately in these terms. However, there will always be instructions whose behaviour is sufficiently complex, obscure or underdocumented, that it is impossible to describe what they do using the chosen intermediate representation. For example, many of the Intel x86 floating point instructions have quite obscure semantics.

Copy-annotate sidesteps this, by copying instructions verbatim, and not using an intermediate representation. We don't care how obscure instructions are since we're only copying them. Copy-annotate must also tag each instruction with some form of description, since instrumentation requires that, but it doesn't matter if that description is partial or even missing completely -- the resulting code will still work. The partial description merely degrades usefulness of instrumentation code. With disassemble-synthesise, if any instruction cannot be completely described, no final code can be generated.

Another apparent advantage of copy-annotate is it appears simpler to implement. What could be simpler than copying code and interleaving in a few instrumentation instructions? By contrast, disassemble-synthesise seems heavyweight: it requires an intermediate representation, translators to and from it, register allocation, instruction selection -- much of the back end of a compiler, in short, and certainly thousands of lines of source code to implement.

On the face of it, then, copy-annotate is simpler and more robust. Curiously, though, a more detailed investigation reveals the opposite.

What, in detail, does disassemble-synthesise need?

- An intermediate representation.
- A translator from guest code into this representation.
- Tools which analyse, and generate instrumentation code, in this representation.
- A translation from the representation back to host machine code. In practice that means an instruction selector, a register allocator and an assembler.

Complex, but manageable. What in detail does copy-annotate need?

- An effects-description notation, for creating annotations, sufficient to drive the instrumenters.
- Machinery to analyse instructions and produce these annotations.
- Tools which take the annotations and produce instrumentation code.
- A way to interleave instrumentation code with the original code.

The interleaver might sound trivial, but is not. It will have to arrange to share resources -- particularly registers -- that the original code "believes" it has complete control of, in such a way that the instrumentation code has adequate registers. This means in the interleaver has to rename and spill registers in the original code in order to free up registers for the instrumentation code. Such "register stealing" might work for flat, regular register files, but on something like the x87 FP register stack it sounds very difficult.

Then there is the question of in what form instrumentation code is produced in copy-annotate. If we intend to support more than one architecture, and simultaneously insulate tools from architecture-specifics, it implies the tools need to produce instrumentation code in some architecture-neutral way. And that in turn implies a mechanism to translate that neutral instrumentation code into real machine code -- an instruction selector, register allocator and assembler.

So in reality, copy-annotate needs the following:

- An effects-description notation, for creating annotations, sufficient to drive the instrumenters.
- Machinery to analyse instructions and produce these annotations.
- An intermediate representation in which to generate instrumentation code.
- Tools which take the annotations and produce instrumentation code.
- Instruction selector, register allocator and assembler.
- A way to interleave instrumentation code with the original code.

Which is just as complex, if not more so, then disassemble-synthesise.

The two approaches have roughly the same complexity. Disassemble-synthesise, however, has two significant advantages: verifiability and cross-architecture support.

In disassemble-synthesise, both the final machine code and the instrumentation code are derived from the initial translation into the intermediate representation. If that translation is wrong, the resulting program simply won't work, and the error will be obvious. In other words, if the final code works, the instrumentation is likely to be correct.

In copy-annotate, the instrumentation code is derived from the annotation tacked onto each instruction. If the annotation is incorrect, the instrumentation will be incorrect, but the program will still work -- since the baseline instructions do not depend on the annotation. That means that incorrect instrumentation code can potentially go undetected. This is a real danger: existing Valgrind uses copy-annotate for SSE instructions. In July 2004 it was discovered that annotations on SSE loads and stores often confused loads with stores. This bug had been undetected for a whole year.

A second disadvantage of copy-annotate is that it necessarily forces the host and guest architectures to be the same. For use of Valgrind in desktop, server and cluster scenarios, this is not an issue. However, developers of embedded systems routinely use cross-simulation during development (eg, debugging an ARM application on an x86 host), so adoption of copy-annotate would permanently rule out availability of Valgrind to large parts of the embedded software development community.

On balance, disassemble-synthesise looks like the better choice. Copy-annotate's sole advantage, its ability to handle strange instructions, is something that can, with a little care, be worked into the disassemble-synthesise framework anyway.

UCode uses an hybrid strategy: disassemble-synthesise for integer x86 instructions, and a crude implementation of copy-annotate for everything else (x87 floating point, MMX, SSE, SSE2). UCode was designed and frozen before the above ideas were properly thought out, and is as a result, messy and inelegant.

The new proposed representation, "IR", is an almost-pure disassemble-resynthesise scheme, with just barely enough of the copy-annotate ideas to get it out of trouble.

3 A new intermediate representation: IR

3.1 An informal overview

We start off with a summary and some examples. Details are shown later. The representation is in reality very simple and has been carefully designed to be flexible, to allow good optimisation and code generation opportunities, and of course to support analyses and transformations needed to add instrumentation code.

Basic blocks

The fundamental unit of translation in IR is the extended basic block -- really a superblock. A superblock contains a single initial entry point and multiple possible exit points. Entering a superblock part-way through is not permitted. We will, with some abuse of standard terminology, refer to these simply as IR Basic Blocks (IRBBs).

An IRBB is a sequence of IR statements (IRStmts), and a where-next address expression. The IRStmts are executed top-to-bottom, and then control passes to the (translation of the) where-next address.

One kind of statement is a conditional early exit from the block. Arbitrary numbers of such early exits are allowed, which is how an IRBB can represent a superblock of guest code (single entry, multiple exits). The common idiom of a basic block terminated by a conditional jump is represented by an IRBB, the last statement of which is a conditional exit to one destination, and the next-expression address points to the other destination.

Guest state

Because IR is architecture neutral, using it for dynamic binary translation or cross-architecture debugging/profiling is feasible. Therefore take care to distinguish the guest instruction set (the machine being simulated) from the host instruction set (the machine running the simulation).

As in UCode, IR models the guest registers as living in the "guest state", a place which is neither memory, nor host registers, just an abstract place. Guest registers are read using Get (expressions) and Put (statements). The meaning of this will become clear shortly.

IR temporaries (IRTemps)

An IR temporary is a scalar-typed name to which a value can be assigned and later referred to. You can think of them as virtual registers, although there is a difference in that the static-single-assignment (SSA) property is enforced: each temporary may only have one definition point. For the examples in this section, temporaries will be written t1, t2, t3, etc.

Expressions (IRExprs)

UCode is a stylised flat machine code. IR is different -- it allows arbitrarily nested expression trees, with a wide variety of operators at the nodes. Allowable expression forms are: literals, temporaries, unary, binary and ternary operator applications, loads from memory, reads (Gets) from the guest state, and calls to side-effect-free helper functions. Here are some simple example expressions:

```
42             -- integer literal

t4             -- use of a temporary

Get(12):I32    -- fetch of guest state (probably an int register)

Add32(t1, Shl32(t2,3))    -- expression denoting t1 + (t2 << 3)

LDle(Sub32(t1,t2)):I16    -- 16-bit little-endian load from memory
                             at address t1 - t2.
```

Expressions denote pure values, and in particular are side-effect free. That is, evaluating the expression any number of times in the context of any given guest state and memory always returns the same value. By implication, evaluation of an expression cannot change the state of the guest machine. As a result, IR-level optimisation may legitimately change the number of times any expression is evaluated without changing the behaviour of the program.

Statements (IRStmts)

Since expressions cannot change the guest's state, they are not by themselves useful. Statements, by contrast, are executed precisely because they change the guest state. An IR basic block is a sequence of statements.

The allowable statement forms are:

- assign a value to a temporary

- non-indexed write (Put) part of the guest state
- indexed write (PutI) part of the guest state
- store a value to memory.
- conditional early exit from the basic block
- call to a complex ("dirty") helper function
- no-op statement, usually resulting from IR optimisation
- memory-fence statement
- instruction-mark statement, which does not change the state of the guest, but conveys information about the size/lengths of guest instructions through the compilation pipeline, for the benefit of instrumenters which need to see the boundaries of original instructions

There are no other statement forms. Here are some example statements:

```
t3 = Add32(t1,Shl32(t2,3))    -- assign value (t1 + (t2<<3)) to
                                 temporary t3

PUT(36) = Mul16(t5,334)       -- write a 16-bit slice of the guest
                                 state with value t5 * 334.  The offset
                                 of the guest state to write is 36, eg
                                 locations 36 and 37 will be written

PUTI(64:8xF64)[t27,-2] = t29  -- write t29 (F64-typed) to a guest state
                                 array whose base is at offset 64, with
                                 8 elements of type F64.  The index of
                                 the array element to write is
                                 (t27 - 2) % 8

STle(Add32(t8,-24)) = Sub32(t5,1)  -- 32-bit little endian write
                                      of (t5-1) to address (t8-24)

if (CmpEQ32(t1,0)) goto 0x12345678

                              -- jump to host address 0x12345678
                                 if t1 == 0 (conditional early exit)

DIRTY 1:I1 NeedsBBP
   MoFX-gst(0,4) WrFX-gst(12,4) WrFX-gst(4,4) WrFX-gst(8,4)
   ::: "x86g_dirtyhelper_CPUID_sse1"{0xB00701AC}()

                              -- unconditional call to dirty helper
                              -- function

IR-NoOp                       -- no-op

IR-MFence                     -- memory fence

IMark(0x8048BC3, 5)           -- instruction mark.  The IR that follows
                                 this mark, until the next mark or the
                                 end of the block, is from a guest insn
                                 at  0x8048BC3 of length 5 bytes.
```

That, perhaps surprisingly, summarises most important aspects of the new IR. Some details are difficult to understand from these examples -- types, indexed Gets and Puts, and calls to dirty helper functions -- but we will return to those later.

3.2 Some example x86 to IR translations

This section shows some translations of simple x86 basic blocks into IR basic blocks, to give a picture of how the IR is used. These examples omit any description of how the x86's condition code (%eflags) register is simulated. Suffice it to say there are at least two different ways to represent %eflags updates, both of which can be described in the IR.

Let the first 16 bytes of guest state hold the guest's %eax, %ecx, %edx and %ebx registers respectively. That is, the expression Get(0):I32 denotes a fetch of %eax, Put(4,I32)=... is a write to %ecx, etc. Consider the following x86 basic block (AT&T syntax, destinations on the right):

```
addl %eax, %ebx
shll $1,   %eax
jmp 0x12345678
```

Let t1, t2, etc be 32-bit typed temporaries. A simple translation into IR can be made by translating each instruction in isolation. One such translation of the block is:

```
t1 = Get(0):I32          -- read %eax
t2 = Get(12):I32         -- read %ebx
Put(12) = Add32(t1,t2)   -- write back %ebx
t3 = Get(0):I32          -- read %eax
Put(0) = Shl32(t3, 1)    -- write back %eax

(next-address field is set to the constant value 0x12345678)
```

Since expressions can be nested arbitrarily, an equally valid and more concise version can omit all the temporaries:

```
Put(12) = Add32(Get(0):I32, Get(12):I32)
Put(0)  = Shl32(Get(0):I32, 1)
(next-address = 0x12345678)
```

IR allows flexibility between building up deeply nested expressions and flattening everything out by assigning all intermediate values to temporaries. This is one of its major strengths over UCode. The more concise version of our running example is not necessarily preferable, since a basic IR-level optimisation is to observe that, in the first rendition, t1 and t3 have the same value and therefore the BB can be rewritten:

```
t1 = Get(0):I32          -- read %eax
t2 = Get(12):I32         -- read %ebx
Put(12) = Add32(t1,t2)   -- write back %ebx
Put(0) = Shl32(t1, 1)    -- write back %eax
(next-address = 0x12345678)
```

Any competent translation of IR back to target machine code will attempt to map all temporaries to host machine registers. So, our little IR optimisation has the effect of keeping a guest register (%eax) in a host register across more than one guest instruction, a potentially powerful optimisation.

Here's another example, showing how the IR represents a complex x86 instruction with awkward semantics. The instruction is

```
rep movsl
```

and the meaning of it is

```
while (%ecx != 0) {
   %ecx --;
   memory at %edi = memory at %esi
   %edi += (%dir << 2);
   %esi += (%dir << 2);
}
```

So this one instruction causes an x86 processor to implement a memory-to-memory block move operation, which moves words from source address specified in %esi to destination address specified in %edi. The number of words to be moved is put into %ecx. Depending on the direction flag %dir, which can (for this example) be regarding as having value 1 or -1, the pointers are stepped forwards or backwards after each word moved.

To make the example clearer, guest-state offsets in Get(..) and Put(..) in the following are replaced with the names of the guest registers in question. Suppose the original instruction was at address 0x10000. Then the resulting IR is:

```
t18 = Get(%ECX)                    -- get trip count
if (CmpEQ32(t18,0)) goto 0x10002  -- if done, move to next insn
Put(%ECX) = Sub32(t18, 1)          -- decrement counter
t17 = Shl32(Get(%DIR), 2)          -- fetch and scale step value
t19 = Get(%ESI)                    -- fetch src pointer
t20 = Get(%EDI)                    -- fetch dst pointer
STle(t19) = LDle(t20):I32          -- *dst = *src
Put(%ESI) = Add32(t19,t17)         -- move src along, and store
Put(%EDI) = Add32(t20,t17)         -- move dst along, and store
(next-address = 0x10000)           -- go round again
```

The second statement (if .. goto ..) is an example of a conditional side exit from a basic block.

A traditional approach for simulating such a complex instruction would be to call a specially written helper function. Whilst IR does allow this, expressing the activity in-line has a big benefit: it exposes to instrumentation tools the precise meaning of the instruction. With a helper function, there is no easy way to know how the helper function alters the guest state, and so no easy way to generate suitable instrumentation code.

3.3 Types in the IR

Unlike UCode, IR is strongly typed. Such typing brings several benefits:

- it facilitates a clean, well defined semantics for subword operations (16/8 bit)
- it clearly differentiates integer, floating point and SIMD activity, which is important for good host code generation.
- it makes it easier for the IR to be independent of the host and guest word sizes. Bear in mind that the IR must work equally well in both 32- and 64-bit environments.
- it facilitates enhanced automatic sanity checking of the IR at various stages in the compilation pipeline, ultimately enhancing reliability of the implementation.

The types supported by IR are low-level machine types:

- 8, 16, 32, 64, 128 bit integers (I8, I16, I32, I64, I128)
- 32 and 64-bit IEEE754 compliant floating point (F32, F64)
- 1-bit, boolean values (Bit)
- 128-bit SIMD values (V128)

Note that integer types do not indicate signfulness; that is a property of operations (right shifts, etc) rather than of the types themselves.

Types are pervasive in IR. All expressions must have a distinct type. Since a temporary can be an expression, this means all temporaries must have types too. Each IR basic block therefore carries a table giving a type for each temporary mentioned in the block. Gets from guest state and reads from memory must specify the type of the returned value. Dually, Put and store statements do not specify the type of the transferred value, as that can be inferred from the expression to be stored/Put'd.

Values of type Bit cannot be stored in memory or in the guest state. Such values are unusual and generally the result of comparison operations.

All unary and binary expression operators have specific types. For example, operator Add64 takes two I64s and produces an I64. It is an error to apply Add64 to arguments of any other type.

Pointer-typed values are represented as either I32 or I64-typed values, depending on the word size of the guest.

IR supplies a rich collection of unary and binary operators, shown in Appendix C. Amongst those are casting/conversion operators which enable all necessary conversions. Under no circumstances whatsoever is any silent or implicit casting between types allowed. All type conversions are explicit.

3.4 Indexed Gets and Puts

The Get (expression) and Put (statement) mechanism provides a convenient way to describe accesses to the guest state (primarily the guest registers). However, the offsets on Put and Get must be constants, known at translation time, which in effect means the identities of all referenced guest registers must be known at translation time.

This is too inflexible. There are situations where the real identity of referenced registers is not known until run-time, for example:

- The identity of registers in the Intel x87 FPU register stack. The meaning of %st(0), %st (1) etc depends on the setting of the FPU register stack pointer.
- On architectures with register windows (SPARC, Itanium), the real identity of integer registers that engage with the window mechanism is not known statically.

IR's solution to this is to provide variants of Get and Put, called GetI and PutI, in which the guest-state offset is not fixed, but an expression computed at run-time. These so-called indexed Gets/Puts allow expression of such run-time dependencies. They are differentiated from normal Puts/Gets because the run-time nature of the offsets involved complicates IR-level optimisations involving redundant-Get and redundant-Put elimination. So it seems simpler to have them as distinct entities.

GetI/PutI therefore provide for circular indexing into parts of the guest state. The part of the guest state to be treated as a circular array is described in an IRArray structure attached to the GetI/PutI. IRArray holds the offset of the first element in the array, the type of each element, and the number of elements.

The array index is indicated rather indirectly, in a way which makes optimisation easy: as the sum of variable part (the 'ix' field) and a constant offset (the 'bias' field). Since the indexing is circular, the actual array index to use is computed as (ix + bias) % number-of-elements-in-the-array.

Here's an example. The description

```
(96:8xF64)[t39,-7]
```

describes an array of 8 F64-typed values, the guest-state-offset of the first being 96. This array is being indexed at (t39 - 7) % 8.

t is important to get the array size/type exactly correct since IR optimisation looks closely at such info in order to establish aliasing/non-aliasing between seperate GetI and PutI events, which is used to establish when they can be reordered, etc. Putting incorrect info in will lead to obscure IR optimisation bugs.

3.5 Helper functions

Calling helper functions from generated code is a standard technique in dynamic code generation systems. Typically, helpers are called to perform tasks which are too complex to handle in-line in generated code.

The big difficulty with helpers in an instrumentable IR system is how to describe, to tools, what the helper function does, in such a way that the tool can generate suitable instrumentation. To make it more difficult still, the description has to be sufficiently comprehensive to cater to the needs of all current and envisaged tools.

IR allows two different types of helper calls: "clean" and "dirty".

Calls to clean helper functions form part of the expression (IRExpr) type. Clean helpers are heavily restricted in what they may do, for two reasons: (1) so as to fit in with the pure-value semantics of expressions, and (2) so as to ensure tools can generate at least some minimal instrumentation pertaining to the call.

A clean helper function must return a value which is dependent purely on the arguments supplied to it, and not on any stored state. It may not read, write or modify either guest memory or the guest state. Repeated calls to a helper with the same arguments must always return the same result.

Note that these restrictions are not automatically checkable, so the onus for ensuring they are observed falls heavily on the programmer. Failing to observe them may cause generation of incorrect instrumentation code, and incorrect behaviour of the program, especially in the presence of aggressive IR-level optimisation.

A good example of the use of a clean helper is to examine the bitfields in an IEEE754 double-precision number, to decide whether it is an Infinity, a NaN, a normal number, etc. Doing this with in-line IR is possible, but tedious; calling a helper is simpler.

Dirty helper functions are more flexible and much more dangerous. Calls to dirty helper functions form part of the statement (IRStmt) family. Dirty helpers:

- take zero or more arguments, which are IR expressions (IRExprs)
- may return either no result, or one result, which is assigned to a nominated temporary
- may read, write or modify at most one contiguous area of guest memory, the address and size of which must be stated on the call
- may read, write or modify arbitrarily many contiguous pieces of the guest state, all of which must be stated on the call
- may store private state and therefore return different results and/or behave differently when called repeatedly with the same arguments

Clearly dirty helpers have side effects, which is why they are classified as statements: their sequencing with respect to the effects of other statements is important.

Calls to dirty helpers must carry a declaration of all the guest state components they access, and a declaration of any guest memory accessed, in order that tools can generate minimal instrumentation code accordingly.

An example of use of a dirty helper is to implement the x86 CPUID instruction. This reads guest %eax and writes guest %eax, %ebx, %ecx and %edx, in complex system-dependent ways. The helper call takes no arguments and produces no result. It is annotated to say it modifies guest %eax and writes guest %ebx, %ecx and %edx, but does not reference guest memory. The helper function itself can then be written to directly access with the guest state.

Dirty helpers severely inhibit IR-level optimisation and should only be used sparingly, for non-performance-critical purposes. At the very least, dirty calls force the optimiser to flush to the guest state any values cached from points in the guest state stated as referenced by the helper, and similarly values cached from memory reads are also invalidated. In all likelyhood a more conservative optimiser will invalidate most IRTemp-held values across a dirty call, forcing code after the call to regenerate those values.

The worst-case behaviour of a dirty helper is that it writes both guest state and guest memory. Hence, when considering the validity of moving code across a dirty helper call, it helps to reason about safety as if the helper is both a store statement and a Put statement.

3.6 A more formal definition of the IR

Extended basic blocks

The fundamental unit of translation is the extended basic block (IRBB). An IRBB consists of:

- a sequence of zero or more statements (IRStmt), executed top-to-bottom
- an expression (IRExpr), of type equal to the guest word size (I32 or I64), indicating the next guest address to execute if control reaches the end of the block
- a control-transfer hint pertaining to the end-of-block jump. This is indicates to the basic block dispatcher whether special actions need to be taken before the next BB is run. The available hints are: Boring (vanilla jump), Call (guest is doing a call), Ret (guest is doing a return), ClientReq (do a client request before continuing), Syscall (do a syscall before continuing), Yield (client is yielding to the thread scheduler), EmWarn (post an emulation-warning before continuing), NoDecode (the next instruction cannot be decoded), MapFail (guest-supplied address-space mapping failed), TInval (invalidate some translations before continuing).
- a table which gives a type (IRType) for each temporary (IRTemp) mentioned in the block.

The following constraints are required, and should be mechanically checked by implementations:

- Each IRTemp must be assigned exactly once.
- No IRTemp may be used prior to its definition point.
- All statements must be type-correct.
- All IRTemps mentioned in the block must have a corresponding type listed in the BB's type-table.

IRTemps in each block have static and dynamic scope confined to that block. All values to be carried from one block to another must be stored either in guest memory or guest state, and so values cached in IRTemps must be flushed prior to any conditional or unconditional exit from the block.

Statements

Statements are executed for their side effect; they do not denote values. A statement may be any of the following forms:

```
IR-NoOp
```

Do nothing.

```
IMark(literal guest address, length)
```

Semantically a no-op. However, indicates that the IR statements which follow it originally came from a guest instruction of the stated length at the stated guest address. This information is needed by some kinds of profiling tools.

```
Put(constant offset) = expr1
```

Evaluate expr1, giving value v1. Write value v1 to the guest state at the given offset.

```
PutI(array-descriptor)[expr1, constant] = expr2
```

Evaluate expr1 and expr2 in unspecified order, giving values v1 and v2. Write the guest state array described by array-descriptor at the offset (v1 + constant) % array-size with the value v2. expr1 must have type I32. expr2 may have any type except Bit. The type of expr2 must match the type stated in array-descriptor.

`tmp = expr`

Assignment to an IR temporary. Evaluate expr and store the value in tmp. tmp must be declared to have the same type as expr. No other statement in this block may assign to tmp. tmp may not be read by any statement prior to this one.

`STle(expr1) = expr2`

Evaluate expr1 and expr2 in unspecified order, giving values v1 and v2. Store v2 in guest memory at address v1, in little-endian format. expr1 must have type equal to the guest word size (I32 or I64). expr2 may have any type except Bit. Big-endian stores are also provided.

`A call to a dirty helper function`

Semantics as described above.

`if (expr1) goto literal-value`

This is a conditional exit from the block. expr1 must have type Bit (boolean). Evaluate expr1. If the result is 1 (True), exit the basic block immediately and continue execution at the guest address indicated by literal-value. literal-value must have type equal to the guest word size (I32 or I64). Optionally, the exit may carry a control-transfer hint, as described above in "Extended Basic Blocks".

`IR-MFence`

Memory fence. All guest loads/stores to/from memory prior to the fence must complete before execution continues after the fence.

There are no other statement forms.

Types

These have been discussed in detail above, and we merely re-state that the available types are:

- 8, 16, 32, 64, 128 bit integers (I8, I16, I32, I64, I128)
- 32 and 64-bit IEEE754 compliant floating point (F32, F64)
- 1-bit, boolean values (Bit)
- 128-bit SIMD values (V128). 64-bit SIMD is done using the I64 type; there is no 64-bit SIMD type.

Expressions

Expressions denote pure values. The type of any expression can be determined by examining it, in conjunction with the temporary-type-map supplied by the basic block containing the expression.

The following expression forms are available:

`Get(constant offset, ty)`

Fetch a value from the given offset of the guest state. The resulting value has type ty. ty may be any type except Bit.

```
GetI(array-descriptor)[expr1, constant]
```

Evaluate expr1, giving value v1. The denoted value is then supplied by reading the guest guest state array described by array-descriptor at the offset (v1 + constant) % array-size The resulting value has the same type as the array element type, which is stated in array-descriptor.

```
LDle:ty(expr1)
```

Evaluate expr1, giving value v1. The resulting value is then supplied by doing a little-endian load from guest memory at address v1. The resulting value has type ty. ty may be any type except Bit. expr1 must have type equal to the guest word size (I32 or I64). Big-endian loads are also provided.

```
tmp
```

Denotes the value stored in this IRTemp. The type of it is determined by looking up tmp in the basic block's IRTemp-type-map.

```
binop(expr1,expr2)
```

Evaluate expr1 and expr2 in unspecified order, giving values v1 and v2. The binary operator binop is then applied to v1 and v2, giving the result value. This expression can have any type, however the result type and types of expr1 and expr2 must match exactly the argument/result types defined for the operator. binop is a primitive operator (IROp) of arity 2. Appendix C contains a complete listing of all IROps available, along with their meanings and types.

```
unop(expr1)
```

Evaluate expr1, giving value v1. The unary operator unop is then applied to v1, giving the result value. This expression can have any type, however the result type and the type of expr1 must match exactly the argument/result types defined for the operator, as specified in Appendix C. unop is a primitive operator (IROp) of arity 1.

```
Mux0X(exprC,expr0,exprX)
```

Informally, Mux0X is like C's ?-: ternary operator, with the then and else clauses swapped, and with one other critical difference: both expr0 and exprX are evaluated, and then the result is chosen. This makes code generation simpler. It also means expr0 and exprX should be as small as possible. Formally: exprC must have type I8. expr0 and exprX can have any type, but they must have the same type. Evaluate expr0 and exprX in unspecified order, giving v0 and vX. Then evaluate exprC, giving vC. If vC is zero, the result value is v0. If vC is non-zero, the result value is vX.

```
CCall helper-fn-name(arg1 .. argN):ty
```

Call to a clean helper function, the name of which is supplied as a string. The resulting type is ty. arg1 .. argN are evaluated in unspecified order and passed to the helper function. The helper function -- presumably written in C -- must have argument and return types which match this call. There is no easy way for implementations to check that, so the burden of correctness falls on the programmer.

`Constants`

Literal values of any type except I128 are allowable. Enough information is stored with the literal that its type can be determined exactly. There is no way to express a literal of type I128. Literals of type V128 can only be expressed in a limited way, by supplying a 16-bit integer, each bit of which denotes the value of each byte lane (0x00 or 0xFF) in the V128 value.

There are no other expression forms.

Aside from the definition of the operators (IROps), that completes the formal definition of the IR notation. The operators are detailed in Appendix C.

4 IR-level transformations

In this proposed new framework, guest instructions are translated by the front end into IR. Each guest instruction is translated in isolation, with no information about the instructions which precede or follow it. This scheme makes the front end simple (if tedious) to write and verify correct, at the expense of generating poor-quality IR for basic blocks as a whole. There are two main reasons for this:

- Each individual translation has to Get and Put all the guest state it refers to, creating a lot of traffic to and from the guest state area. Guest state is not cached in temporaries over multiple guest instructions.
- Many of the translations benefit from constant folding and (IR-level) dead code elimination, based on propagating and folding constant values exposed by earlier instructions.

The IR-level optimiser solves both problems, often dramatically improving the quality of the IR as a result. A second motivation for doing IR optimisation immediately after the initial translation from guest code, but prior to instrumentation, is that it reduces the amount of code the instrumenter(s) have to deal with, hence indirectly improving the quality of the instrumentation code.

The expected transformation pipeline for a production-quality system based on IR is as follows:

- translate guest instructions individually into IR
- generic IR optimisations
- analyse IR and add suitable instrumentation code (tool-specific)
- do tool-specific IR-level improvements
- generic IR optimisations
- build maximal IR trees

At this point, IR's work is done. For completeness, the rest of the pipeline is:

- translate IR trees to host machine code, with virtual registers
- register-allocate host machine code, removing all virtual registers

- schedule host code, if absolutely unavoidable (viz, if not doing so produces unacceptable performance loss. Currently only contemplated for Itanium.)
- assemble/link host code, producing executable bytes in memory

Generic IR optimisations, the subject of this section, improve IR regardless of the source. Tool-specific IR optimisations are useful when the instrumentation added by a tool can be improved, but requires specialised understanding of that instrumentation, which is not appropriate for inclusion in a generic optimiser. The current (UCode-based) Memcheck implementation works like this.

4.1 Flattening

This removes all nested subexpression trees by assigning all interior nodes to fresh IR temporaries, thereby generating traditional flat SSA-style code. This is not by itself an improvement -- in fact it makes the final code much worse. The value is that it makes the rest of the optimiser simpler, by removing the need to deal with nested expression trees.

For example, the statement

```
t4 = Shl32(t2,And8(Sub8(t5,0x1:I8),0x1F:I8))
```

flattens out into:

```
t14 = Sub8(t5,0x1:I8)
t13 = And8(t14,0x1F:I8)
t4  = Shl32(t2,t13)
```

where t13 and t14 are new temporaries with suitable types.

4.2 Redundant-Get removal

The first real improvement phase is a limited form of common-subexpression elimination, in which second and subsequent Gets of the same piece of guest state are replaced by the temporary in which the first Get stores its value:

```
t9 = GET(12,I32)
... intervening statements ...
t21 = GET(12,I32)
```

becomes

```
t9 = GET(12,I32)
... intervening statements ...
t21 = t9
```

The optimiser needs to be careful to ensure that the intervening statements do not fully or partially overwrite (Put, PutI) the guest state fetched by the first Get, as this would invalidate the transformation.

4.3 Redundant-Put removal

This is a form of dead-store elimination. It removes writes (Puts) to guest state when it can be shown that a later Put to the same area of the guest state overwrites the values stored by the first Put:

```
PUT(40) = t19
... intervening statements ...
PUT(40) = t8
```

becomes

```
(first Put is deleted)
... intervening statements ...
PUT(40) = t8
```

In order to show this is valid, the optimiser must first prove that (1) the second Put postdominates the first Put, which for our purposes means the intervening statements do not contain any conditional or unconditional exits, and (2) the intervening statements do not Get any part of the guest state written by the first Put (since otherwise the first Put does not write a dead value).

Used together, redundant-Get removal followed by redundant-Put removal are strikingly effective at removing redundant traffic to/from the guest state storage area, keeping guest register values in IR temporaries over the entire basic block in most circumstances. By itself, this significantly improves code quality. Equally important, it improves the prospects for constant folding.

4.4 Copy propagation, constant propagation, and constant folding

These passes together are often referred to simply as "constant folding" in standard compiler texts.

They are best illustrated with the following running example:

```
t2 = 0x10
t3 = 0x1F
t4 = t2
t5 = t3
t6 = And32(t4,t5)
```

Copy propagation removes trivial temp-to-temp copies by renaming uses of temporaries which are known to be copies of earlier-defined temporaries. Here, t4 and t5 are renamed at their use points, giving:

```
t2 = 0x10
t3 = 0x1F
t4 = t2
t5 = t3
t6 = Add32(t2,t3)
```

Similarly, constant propagation replaces uses of temporaries which are known to be bound to literals, with the literal values themselves, effectively moving all literals to their use sites. Our example becomes:

```
t2 = 0x10
t3 = 0x1F
t4 = 0x10
t5 = 0x1F
t6 = Add32(0x10,0x1F)
```

Finally, constant folding evaluates operators applied to literal arguments, when possible. Our example becomes:

```
t2 = 0x10
t3 = 0x1F
t4 = 0x10
t5 = 0x1F
t6 = 0x2F
```

Operator identities can also be folded out at this point, for example Add32(0,x) --> x for any x at all, even non-literal x, and Mul32(0,x) becomes zero. Folding out the ternary Mux0X operator when the guarding condition is known is also worthwhile.

Doing redundant Get/Put removal prior to constant folding helps because that makes literal values available across the entire (extended) basic block, rather than just in the individual guest instruction translation in which they originally appeared.

4.5 Dead code removal

The above transformations often remove all uses of a given temporary. The assignment to that temporary is then, by definition, useless and can itself be deleted. Deleting an assignment often gives rise to further dead temporaries earlier on. A single backward sweep over the statements can remove all dead assignments in one pass. In our running example, supposing t2, t3, t4 and t5 are not used further, they are simply deleted, leaving:

```
t6 = 0x2F
```

Because expressions are guaranteed side-effect-free, the optimiser's only proof obligation prior to deleting such a binding is to ensure that the temporary is not used later on.

4.6 Partial evaluation of clean helper functions

Clean helper functions are inserted into the IR by front-ends, and so the IR optimiser, which is generic, has no idea of what these functions do, and cannot optimise around them without outside assistance. The preceding set of optimisations often reduce arguments to helper functions to literal values, and it can be useful to enquire whether, given those literal values, the helper function's behaviour can be replaced by something simpler -- ideally by an in-line piece of code.

Here is a contrived example. Consider the following helper function:

```
int maybe_fooble ( int a, int b, int c )
{
    if (a == 0)
       return b * (c+1);
    else
       return /* really complex expression involving a,b and c */
}
```

Now suppose the IR optimiser encounters a call to this helper function:

```
t1 = "maybe_fooble"( 0, t2, t3 )
```

By itself the IR optimiser can do nothing with this. However, it can call back to the front end, saying in effect "I am seeing a helper function call that you put in, with arguments literal zero, t2 and t3. Can you replace this by anything simpler?"

And the front end -- which is the only party which "understands" the behaviour of the helper -- replies "yes, that is equivalent to Mul32(t2, Add32(t3,1))". So the IR optimiser replaces the statement with

```
t1 = Mul32(t2, Add32(t3,1))
```

which is a big improvement, perhaps becoming two host instructions rather than at a minimum 15-20 host instructions to call out to a helper.

Note of course that if the call's first argument had been non-zero, the front end would say "no, there is no simple replacement", and so the call would have to be left in place in the IR.

Condition code handling (%eflags) for x86 and AMD64 guests will likely depend heavily on calling helper functions, so this transformation is potentially important.

4.7 Common subexpression elimination (CSE)

Another textbook compiler technique. This removes duplicated evaluations of identical expressions. Let Op stand for any binary operator (Add32, Mul8, etc). Then the fragment

```
t3 = Op(t1,t2)
... intervening statements ...
t4 = Op(t1,t2)
```

becomes

```
t3 = Op(t1,t2)
... intervening statements ...
t4 = t3
```

Applied repeatedly, CSE removes entire duplicate expression trees. This is particularly useful for cleaning up the offset calculations in GetI and PutI (used for simulating stacks and windows of guest registers). For example, the Intel x87 "fsqrt" instruction translates (note: simplified!) into this:

```
PUTI(Add32(Shl32(And32(GET(52,I32),0x7:I32),0x3:I8),0x38:I32))
= SqrtF64(
     GETI(Add32(Shl32(And32(GET(52,I32),0x7:I32),0x3:I8),0x38:I32))
```

Most of the computation here is to do with simulating the FP register stack. The big subterm is duplicated, and so this CSEs to:

```
t1 = Add32(Shl32(And32(GET(52,I32),0x7:I32),0x3:I8),0x38:I32)
PUTI(t1) = SqrtF64(GETI(t1))
```

This makes it clearer that t1 is the guest-state offset of the %st(0) floating point register, as calculated at run-time. Die-hards may decipher this further by noting that GET(52,I32) yields the simulated FPU's register stack pointer, and that the first floating point register has offset 0x38 in the guest state.

4.8 Reconstructing expression trees

This is the opposite transformation to flattening. Tree-building is the last IR transformation step prior to instruction selection (translation out of IR into host machine code). The aim is to remove statements which assign a value to a temporary that is only used once. In such cases, the statement is removed and the right-hand-side of the binding is substituted at the temporary's one-and-only use site:

```
t2 = Shl32(t1, 2)
t3 = Add32(t2, 42)
t4 = LDle(t3):I8
t5 = 8Uto32(t4)
```

becomes, providing t2, t3 and t4 are not used anywhere else:

```
t5 = 8Uto32(LDle(Add32(Shl32(t1,2),42)):I8)
```

This transformation is significant because of the opportunities it presents to tree-matching instruction selectors. A simple instruction selector that handles each statement in isolation is forced to emit at least four instructions for the original sequence. The transformed version offers the possibility of doing better, if the instruction selector can match the entire pattern, or large parts of it, to a suitable host instruction. This example does an address computation, an 8-bit load from memory and an unsigned widening to 32 bits, and therefore an x86 instruction selector can produce a single instruction of the form 'movzbl 42(,%reg1,4), %reg2', which is a big improvement over four instructions.

Tree-building, in collaboration with a tree-matching instruction selector, is a powerful aid to good host code generation. It is also potentially dangerous and needs care to ensure correctness. The danger arises because tree-building in effect allows the optimiser to arbitrarily reorder chunks of computation. Both the sequences

```
t1 = LDle(t2):I32
t3 = LDLe(t4):I32
t5 = Add32(t1,t3)
```

and

```
t3 = LDLe(t4):I32
t1 = LDle(t2):I32
t5 = Add32(t1,t3)
```

build into

```
t5 = Add3(LDle(t2):I32), LDle(t4):I32)
```

and so for at least one of the original sequences, the order of the loads in the final host code will be different from that in the IR.

The rules for safe tree-building are:

- No expression may be floated forwards past a statement which invalidates or potentially invalidates the expression. Specifically, a statement containing a load may not be floated past a following store statement or dirty helper call, unless alias analysis or other mechanisms guarantee the addresses are non-aliasing. Similarly, a statement containing a Get or GetI may not be floated past a Put, PutI or dirty helper call, again unless non-aliasing of guest state can be proven.
- A binding 't = E' may only be floated forwards and substituted at t's use point if t is used just once. If t is used more than once, the transformation is still safe, but substituting E at all the use points duplicates E and hence duplicates work in the final host code.

Making tree-building both fast and reliable is surprisingly difficult, but the resulting host code quality improvement it facilitates is worth the effort.

4.9 Putting it all together

None of these transformations in isolation is very effective. Yet each creates opportunities for the others, and when applied together the results can be impressive. As a final example, consider the guest x86 instruction sequence

```
addl %eax, %ebx
shll $16, %ebx
cmpl $0x12345678, %ebx
jle  0x8048308
```

Without justification here, we can say that the initial translation into IR, including simulation of %eflags, is as follows:

```
0x80482F9:  addl %eax,%ebx

   t2 = GET(12,I32)
   t1 = GET(0,I32)
   t0 = Add32(t2,t1)
   PUT(32) = 0x3:I32
   PUT(36) = t1
   PUT(40) = t2
   PUT(12) = t0

0x80482FB:  shll $16, %ebx

   t3 = GET(12,I32)
   t8 = And8(0x10:I8,0x1F:I8)
   t5 = t3
   t6 = Shl32(t5,t8)
   t7 = Shl32(t5,And8(Sub8(t8,0x1:I8),0x1F:I8))
   PUT(32) = Mux0X(t8,GET(32,I32),0x18:I32)
   PUT(36) = Mux0X(t8,GET(36,I32),t7)
   PUT(40) = Mux0X(t8,GET(40,I32),t6)
   t4 = t6
   PUT(12) = t4
0x80482FE:  cmpl $0x12345678, %ebx

   t11 = GET(12,I32)
   t10 = 0x12345678:I32
   t9 = Sub32(t11,t10)
   PUT(32) = 0x9:I32
   PUT(36) = t10
   PUT(40) = t11

0x8048304:  jle-8 0x8048308

   if (32to1(calculate_condition(0xE:I32,GET(32,I32),
                  GET(36,I32),GET(40,I32)):I32)) goto 0x8048308:I32
   goto 0x8048306:I32
```

A couple of things to note. On x86, a shift instruction (shll) sets the condition codes, except if the shift amount is zero. Hence the 3 uses of Mux0X in that instruction's IR. Secondly, the sequence ends in a conditional jump, and a helper function 'calculate_condition' has to called to inspect the stored condition codes to make a go/no-go decision.

The optimised (and tree-built) result is a big improvement:

```
t17 = Shl32(Add32(GET(12,I32),GET(0,I32)),0x10:I8)
PUT(12) = t17
PUT(32) = 0x9:I32
PUT(36) = 0x12345678:I32
PUT(40) = t17
if (32to1(1Uto32(CmpLE32S(t17,0x12345678:I32)))) goto 0x8048308:I32
goto {Boring} 0x8048306:I32
```

t17 makes it a lot clearer what the original really computes. The shift amount is known to be non-zero, and so the Mux0Xs have been folded out. No less than seven redundant Puts have been removed. Finally, in collaboration with the x86->IR front end, the call to calculate_condition has been replaced by a 32-bit signed less-or-equal comparison. In effect the optimiser has managed to recover the original comparison which the compiler of this code fragment (gcc) translated into the "cmpl; jle" sequence.

5 Instrumenting the Intermediate Representation

Once guest basic block(s) have been translated into an IR block, and an initial IR optimisation pass done, the resulting IR is passed to the selected tool for instrumentation. The tool analyses the IR and generates instrumentation code, also in IR form, creating a new, instrumented IR block.

Tools vary in the amount of instrumentation they add, and the demands they place on the IR representation. The most demanding and complex standard tool is Memcheck, so demonstrating that the IR supports Memcheck is sufficient for our current goals.

Memcheck really does two different tasks:

- **V-bit tracking:** for each byte of guest state, Memcheck supplies a "shadow" byte which tracks whether or not that guest state byte is initialised. For efficiency of code generation, a shadow bit value of zero indicates the corresponding guest state bit is initialised, and one indicates not-initialised. In previous literature on Valgrind, these shadow values are called V-bits (V == valid).

  Memcheck also shadows all the guest's memory with V bits. Each byte of guest memory has a shadow V byte. When data moves between the guest state (registers etc) and guest memory, Memcheck inserts corresponding moves between the shadow guest state and shadow guest memory.

  Whenever the original program performs an operation which could lead to "visibly" incorrect behaviour in the presence of uninitialised values, Memcheck inserts a check to ensure the corresponding shadow values show the used data to be initialised. If not, an error logging routine is called, eventually leading to the user seeing an error message.

  "Visibly" incorrect behaviour approximately means becoming at risk of an exception that could not otherwise happen. For example, if any component of an address calculation is uninitialised, then the program might read or write data at some random address, and hence take a page fault, so Memcheck generates a check at that point. Similarly, if a conditional jump is reached, and the condition codes are uninitialised, an error is logged.

- **A-bit tracking:** for each byte of guest memory address space, Memcheck maintains a single "A" bit, which indicates whether or not the program can validly access that location. Using this mechanism, Memcheck can detect invalid-addressing errors, such as block overruns, use of freed memory, and accessing below the stack pointer.

The first of these, V-bit tracking, is more complex. The demands it places on the IR are:

- There must be a way to explain to Memcheck the layout of the guest state, in order that Memcheck can lay out suitable shadow state. In current x86-only Valgrind, this is not an issue since only one guest architecture (x86) is supported, so there is only one guest state layout, and Memcheck has knowledge of this layout hardwired into it.

  This is easily enough achieved by passing to Memcheck a table indicating the size and offset of all guest state fields, and also indicating which areas of the guest state are indexable using GetI/PutI. It is believed this supplies enough information for Memcheck to make sense of all Get/GetI/Put/PutI references in the IR it must instrument.

- Once shadow state layout is established, the V bit computations themselves are straightforward and can be expressed in IR. Memcheck needs to compute the definedness of every subexpression in the incoming IR. This is most easily done if that incoming code is in flat form, as achieved by the first stage of IR optimisation.

  For example, consider the statement:

  ```
  t1 = Add32(t2,t3)
  ```

  Let q1, q2 and q3 be the 32-bit temporaries shadowing t1, t2 and t3 respectively. Memcheck must generate a binding for q1 based on q2 and q3. The relevant code is:

  ```
  qtmp = Or32(q2,q3)
  q1   = Or32(qtmp, Neg32(qtmp))
  ```

  The second term, qtmp | -qtmp, propagates any undefinedness leftwards in the word, simulating the worst-case propagation of undefinedness through the carries in the original addition.

  A number of other such operations are needed. It is believed that all of them can be expressed using the integer arithmetic primops in (Appendix C), and also that they can be efficiently translated to host machine code.

- As the original code does Put/PutI/Get/GetI to/from guest state, Memcheck generates corresponding Put/PutI/Get/GetI to move corresponding shadow values to/from shadow guest state.

  As the original code reads and writes guest memory, Memcheck must arrange shadow transfers from/to shadow guest memory.

  A read of a shadow memory location can be implemented using a clean helper call. The semantics of clean helpers disallow them from accessing guest memory, but accessing shadow guest memory is acceptable: such calls are still referentially transparent, exception-free, and do not need further instrumentation. The read helper function(s) not only retrieve the relevant V bits, they also check the A bits corresponding to the accessed address(es) and emit a warning for invalid accesses.

  Writes are more complex. Since writes must not be reordered with respect to other writes or reads, the appropriate vehicle is a dirty helper call. The helper is supplied the guest address and the V bits to store, and returns no value. The helper is stated to access neither guest memory nor guest state (both of these claims are true). The helper stores the V bits appropriately and also performs an A-bit check, to detect writes to invalid locations.

- As the guest stack pointer moves up and down, Memcheck must emit calls to a similar dirty helper. This adjusts the A bits to maintain the property that guest memory at and above the stack pointer is accessible, but that below it is not.

- The most difficult part is dealing with helper calls in the incoming IR. Since Memcheck has no idea what these helpers do it can only produce an approximately correct instrumentation sequence.

  For clean helpers, the issue is how to calculate the V bits of the result given the V bits of the arguments. Two strategies come to mind:

    - "Strict": assume the result is defined. Check the arguments and signal errors if any are undefined. This strategy disallows uninitialised values from "flowing through" clean helpers.

    - "Deferred": regard the result as being defined iff all the argument components are defined. Otherwise regard the result as undefined. This is the strategy used in current UCode-Memcheck.

  For dirty helpers, the same strategies are valid. The only issue is identifying the inputs and outputs to be considered. This is possible because a dirty helper must state which parts of guest memory, and which parts of guest state, it accesses.

  Since a dirty helper can access guest memory, instrumentation of any such helper must also include an A-bit check for the accessed address.

- The generic IR optimiser is effective enough to clean up after instrumentation (remove unneeded instrumentation artefacts.) This is a change from UCode-world, in which Memcheck required a specialised post-instrumentation cleanup pass.

That concludes the implementation summary for Memcheck in this IR framework. Other tools, such as Cachegrind, Addrcheck and Helgrind, are simpler, as they only care about guest memory accesses, and do not track value computations.

With very minor modifications it is believed IR will also support the experimental Annelid bounds-checking tool described in [Nethercote 04]. Annelid's demands on the IR framework as similar to those made by Memcheck.

6 Appendix A: Supporting a new architecture in the framework

As should be clear by now, a distinction is made between supporting an architecture as a guest (so as to debug/profile programs for that architecture) and as a host (the real machine on which Valgrind runs).

The simple case is when guest and host are the same, and in that case the new code to be written is as follows:

- First, a top-level plan of how to simulate the guest architectural state has to be made. From that, the guest state storage requirements can be defined. For example, x86 and AMD64 represent condition codes in delayed form, so three words are allocated in the guest state for the delay thunk.
- The most onerous step is to write a translator from guest instructions into IR, with the generated IR updating the defined guest state to reflect the instruction semantics. Each instruction can be translated individually. The precise formulation of each translation is not particularly critical, since IR optimisation will remove many redundancies and small differences.
- Type definitions, constructors and auxiliary functions for host instructions must be defined. The auxiliary functions are:
    - print a host instruction (for debugging)
    - assemble a host instruction (emit machine code)
    - describe host instruction register uses (for register allocation)
    - map register uses in a host instruction, under the supervision of the register allocator
    - indicate whether or not a host instruction is a reg-reg move
- A definition of host registers that the register allocator may use must be provided. This is simple. Multiple register classes are supported.
- An instruction selector, which translates IR into virtual registerised host machine code, must be written. As discussed in Appendix B, this can be done by any means, but a top-down tree-matching scheme works well.

That is all. In particular, there should be no need to update any of IR-only transformations, including the tools' instrumentation systems, since the IR exists expressly to insulate them from machine dependencies. Indeed, consideration of the above 5 points shows them all to be connections in- and out- of IR-world.

7 Appendix B: Converting IR back to host machine code

One modern idiom for writing simple, portable instruction selectors which generate good code is to do pattern-matching on expression trees. The general approach is explained well in [FHP 92] and [Fraser & Hanson 95]. There are a variety of approaches, giving varying tradeoffs between complexity, quality of generated code, and speed of code generation. A popular choice is the "lburg" scheme, which is a two-pass algorithm that gives code sequences that are optimal with respect to any non-negative cost function on host instructions.

Of course none of this mandates any specific IR-to-host translation scheme. However, the following appears promising:

- Do instruction selection by top-down greedy traversal of expression trees ("maximal munch"). This is simple, effective, not guaranteed-optimal, but works well in practice.
- Do linear scan register allocation, as per the algorithm of [THS 98]. Current UCode-Valgrind uses this algorithm and it has proven fast and effective.

Instruction selectors are somewhat simplified if the allocator is capable of coalescing register-to-register moves. This gives instruction selectors freedom to emit redundant reg-reg moves in the knowledge that they will be coalesced away. That freedom particularly simplifies instruction selection for 2-address host architectures such as x86 and AMD64.

8 Appendix C: list of binary and unary operators in the IR

Here is a list of all the IR binary and unary operations so far defined, as required for simulating an x86 guest. It will require minor extension for 64-bit targets. Operators are binary unless stated otherwise.

INTEGER ARITHMETIC/LOGICAL/COMPARISON

```
Iop_Add8,  Iop_Add16,  Iop_Add32,  Iop_Add64,
   8/16/32/64-bit two's complement integer addition.
   types are (I8,I8) -> I8,  (I16,I16) -> I16, (I32,I32) -> I32,
   (I64,I64) -> I64 respectively.

Iop_Sub8,  Iop_Sub16,  Iop_Sub32,  Iop_Sub64,
   8/16/32/64-bit two's complement integer subtraction.

Iop_Mul8,  Iop_Mul16,  Iop_Mul32,  Iop_Mul64,
   8/16/32/64-bit signless, non-widening integer multiplication.

Iop_And8,  Iop_And16,  Iop_And32,  Iop_And64,
Iop_Or8,   Iop_Or16,   Iop_Or32,   Iop_Or64,
Iop_Xor8,  Iop_Xor16,  Iop_Xor32,  Iop_Xor64,
   8/16/32/64-bit and/or/xor.

Iop_Shl8,  Iop_Shl16,  Iop_Shl32,  Iop_Shl64,
   8/16/32/64-bit shift left.  Note that the shift amount -- the second
   argument -- is a value of type I8, regardless of the type of the shifted
   argument.  Note also, only shift amounts of zero .. word_size - 1 are
   defined; these operations are undefined for shifts equal to or larger
   than the word size.  You must ensure they are never given out-of-range
   shift values.

Iop_Shr8,  Iop_Shr16,  Iop_Shr32,  Iop_Shr64,
   8/16/32/64-bit zero-extending shift right.  Same constraints on shift#
   amount as for Iop_Shl{8,16,32,64}.

Iop_Sar8,  Iop_Sar16,  Iop_Sar32,  Iop_Sar64,
   8/16/32/64-bit sign-extending shift right.  Same constraints on shift
   amount as for Iop_Shl{8,16,32,64}.

Iop_CmpEQ8,  Iop_CmpEQ16,  Iop_CmpEQ32,  Iop_CmpEQ64,
Iop_CmpNE8,  Iop_CmpNE16,  Iop_CmpNE32,  Iop_CmpNE64,
   Integer equality/non-equality comparisons.  These take two values of
   type I8/I16/I32/I64 respectively and produce a boolean (type Bit).

Iop_CmpLT32S,
Iop_CmpLE32S,
Iop_CmpLT32U,
Iop_CmpLE32U,
   32-bit signed/unsigned comparisons "less-than" or "less-than-or-equal".
   All with type (I32,I32) -> Bit.

Iop_Not8,  Iop_Not16,  Iop_Not32,  Iop_Not64,
   8/16/32/64-bit bitwise negation.  Unary op.

Iop_Neg8,  Iop_Neg16,  Iop_Neg32,  Iop_Neg64,
   8/16/32/64-bit two's complement negation.  Unary op.

Iop_MullS8, Iop_MullS16, Iop_MullS32,
Iop_MullU8, Iop_MullU16, Iop_MullU32,
   Signed and unsigned widening integer multiplies.
      Iop_MullS8/U8 have type (I8,I8) -> I16
      Iop_MullS16/U16 have type (I16,I16) -> I32
      Iop_MullS32/U32 have type (I32,I32) -> I64

Iop_Clz32, Iop_Ctz32
   Count leading / trailing zeroes in a 32-bit word.  Unary ops with type
   I32 -> I32.  Result is undefined when given an  argument of zero -- you
   must ensure that never happens.

Iop_DivModU64to32
   Unsigned integer division and remainder.  Divides an unsigned  64-bit
   integer by an unsigned 32-bit integer.  Has type (I64,I32) -> I64.
   The top 32 bits of the result is the remainder and the bottom half is
   the divided value.

Iop_DivModU64to32
   Signed version of Iop_DivModU64to32.  TODO: clarify semantics re
   rounding, negative arguments.
```

INTEGER CONVERSION

```
Iop_8Uto16, Iop_8Uto32, Iop_16Uto32, Iop_32Uto64
   Unary operators, which do unsigned (zero-extend) widening.
   Types are I8 -> I16, I8 -> I32, I16 -> I32, I32 -> I64 respectively.

Iop_8Sto16, Iop_8Sto32, Iop_16Sto32, Iop_32Sto64
   Sign-extending versions of the above.

Iop_32to8
   Unary op, converts a value of I32 to one of type I8, by throwing away
   the top 24 bits.

Iop_16to8
   Unary, type I16 -> I8.  Returns low half of argument.

Iop_16HIto8
   Unary, type I16 -> I8.  Returns high half of argument.

Iop_8HLto16
   Binary, type (I8,I8) -> I16.  Construct a 16-bit word from two 8-bit
   pieces.

Iop_32to16, Iop_32HIto16, Iop_16HLto32
   As Iop_16to8, Iop_16HIto8, Iop_8HLto16, but converting between 16- and
   32-bit sizes.

Iop_64to32, Iop_64HIto32, Iop_32HLto64
   As Iop_16to8, Iop_16HIto8, Iop_8HLto16, but converting between 32- and
   64-bit sizes.

Iop_32to1
   Unary, type I32 -> Bit, returns the lowest bit.

Iop_1Uto8
   Unary, type Bit -> I8, unsigned widen to I8.

Iop_1Uto32
   Unary, type Bit -> I32, unsigned widen to I32.
```

FLOATING POINT ARITHMETIC/COMPARISON

```
Iop_AddF64, Iop_SubF64, Iop_MulF64, Iop_DivF64
   Binary operations (add, sub, mul, div) mandated by IEEE754-1985.
   All with type (F64, F64) -> F64.

Iop_NegF64, Iop_SqrtF64
   Unary operations (negation, square root) mandated by IEEE754-1985.
   With type F64 -> F64.

Iop_AtanF64,      /* FPATAN,  arctan(arg1/arg2)       */
Iop_Yl2xF64,      /* FYL2X,   arg1 * log2(arg2)       */
Iop_Yl2xp1F64,    /* FYL2XP1, arg1 * log2(arg2+1.0)   */
Iop_PRemF64,      /* FPREM,   remainder(arg1/arg2)    */
Iop_ScaleF64,     /* FSCALE,  arg1 * (2^RoundTowardsZero(arg2)) */
Iop_PRemC3210F64, /* C3210 flags resulting from FPREM, :: I32 */
```

```
      Binary operations supported by x86 but not mandated by IEEE754.
      All have type (F64, F64) -> F64, except
      Iop_PRemC3210F64, which has type (F64, F64) -> I32.

   Iop_AbsF64,    /* FABS */
   Iop_SinF64,    /* FSIN */
   Iop_CosF64,    /* FCOS */
   Iop_2xm1F64,   /* (2^arg - 1.0) */
      Unary operations supported by x86 but not mandated by IEEE754.
      All have type F64 -> F64.

   Iop_Cmp64
      Binary, floating point comparison.  Has type (F64, F64) -> I32.
      The returned integer is one of the four values:
         0x45 Unordered (will be if either arg is a NaN)
         0x01 first arg less
         0x00 first arg greater
         0x40 equal
      These return values may be rationalised in later revisions.
```

FLOATING POINT CONVERSION

```
   Iop_I32toF64
      Unary, type I32 -> F64.  Convert a signed integer to an IEEE754
      double, without loss of accuracy.

   Iop_I64toF64
      Unary, type I64 -> F64.  Convert a signed integer to an IEEE754
      double.  Hmm.  Surely this needs to take account of the required
      rounding mode, as the conversion cannot always be exact?  TODO: Fix.

   Iop_F64toI64, Iop_F64toI32, Iop_F64toI16
      Unary, type (I32,F64) -> I64/I32/I16.  Convert an IEEE754 double into
      a signed integer, rounded as indicated by the least significant two
      bits of the first argument:
         00b  to nearest
         01b  to -infinity
         10b  to +infinity
         11b  to zero
      If one of these conversions gets an out-of-range condition, or a
      NaN, as an argument, the result is host-defined (blargh).  On x86
      the "integer indefinite" value 0x80..00 is produced.  On PPC it is
      either 0x80..00 or 0x7F..FF depending on the sign of the argument.

   Iop_RoundF64
      Rounds an IEEE754 double to the nearest integral value, but does
      not convert it to an integer.  Also takes an initial I32 argument
      indicating the rounding mode to use.  Unary, type is (I32,F64) -> F64.

   Iop_F32toF64
      Convert an IEEE754 single-precision value to double precision.
      Unary, type is F32 -> F64.

   Iop_F64toF32
      Convert an IEEE754 double-precision value to single precision.
      Unary, type is F64 -> F32.  Surely needs to take into account the
      required rounding mode?  TODO: Fix.
```

64-bit SIMD integer

```
      /* MISC (vector integer cmp != 0) */
      Iop_CmpNEZ8x8, Iop_CmpNEZ16x4, Iop_CmpNEZ32x2,

      /* ADDITION (normal / unsigned sat / signed sat) */
      Iop_Add8x8,   Iop_Add16x4,   Iop_Add32x2,
      Iop_QAdd8Ux8, Iop_QAdd16Ux4,
      Iop_QAdd8Sx8, Iop_QAdd16Sx4,

      /* SUBTRACTION (normal / unsigned sat / signed sat) */
      Iop_Sub8x8,   Iop_Sub16x4,   Iop_Sub32x2,
      Iop_QSub8Ux8, Iop_QSub16Ux4,
      Iop_QSub8Sx8, Iop_QSub16Sx4,

      /* MULTIPLICATION (normal / high half of signed/unsigned) */
      Iop_Mul16x4,
      Iop_MulHi16Ux4,
      Iop_MulHi16Sx4,

      /* AVERAGING: note: (arg1 + arg2 + 1) >>u 1 */
      Iop_Avg8Ux8,
      Iop_Avg16Ux4,

      /* MIN/MAX */
      Iop_Max16Sx4,
      Iop_Max8Ux8,
      Iop_Min16Sx4,
      Iop_Min8Ux8,

      /* COMPARISON */
      Iop_CmpEQ8x8,  Iop_CmpEQ16x4,  Iop_CmpEQ32x2,
      Iop_CmpGT8Sx8, Iop_CmpGT16Sx4, Iop_CmpGT32Sx2,

      /* VECTOR x SCALAR SHIFT (shift amt :: Ity_I8) */
      Iop_ShlN16x4, Iop_ShlN32x2,
      Iop_ShrN16x4, Iop_ShrN32x2,
      Iop_SarN16x4, Iop_SarN32x2,

      /* NARROWING -- narrow 2xI64 into 1xI64, hi half from left arg */
      Iop_QNarrow16Ux4,
      Iop_QNarrow16Sx4,
      Iop_QNarrow32Sx2,

      /* INTERLEAVING -- interleave lanes from low or high halves of
         operands.  Most-significant result lane is from the left
         arg. */
      Iop_InterleaveHI8x8, Iop_InterleaveHI16x4, Iop_InterleaveHI32x2,
      Iop_InterleaveLO8x8, Iop_InterleaveLO16x4, Iop_InterleaveLO32x2,
```

128-bit SIMD FP

```
/* --- 32x4 vector FP --- */

/* binary */
Iop_Add32Fx4, Iop_Sub32Fx4, Iop_Mul32Fx4, Iop_Div32Fx4,
Iop_Max32Fx4, Iop_Min32Fx4,
Iop_CmpEQ32Fx4, Iop_CmpLT32Fx4, Iop_CmpLE32Fx4, Iop_CmpUN32Fx4,

/* unary */
Iop_Recip32Fx4, Iop_Sqrt32Fx4, Iop_RSqrt32Fx4,

/* --- 32x4 lowest-lane-only scalar FP --- */

/* In binary cases, upper 3/4 is copied from first operand.  In
   unary cases, upper 3/4 is copied from the operand. */

/* binary */
Iop_Add32F0x4, Iop_Sub32F0x4, Iop_Mul32F0x4, Iop_Div32F0x4,
Iop_Max32F0x4, Iop_Min32F0x4,
Iop_CmpEQ32F0x4, Iop_CmpLT32F0x4, Iop_CmpLE32F0x4, Iop_CmpUN32F0x4,

/* unary */
Iop_Recip32F0x4, Iop_Sqrt32F0x4, Iop_RSqrt32F0x4,

/* --- 64x2 vector FP --- */

/* binary */
Iop_Add64Fx2, Iop_Sub64Fx2, Iop_Mul64Fx2, Iop_Div64Fx2,
Iop_Max64Fx2, Iop_Min64Fx2,
Iop_CmpEQ64Fx2, Iop_CmpLT64Fx2, Iop_CmpLE64Fx2, Iop_CmpUN64Fx2,

/* unary */
Iop_Recip64Fx2, Iop_Sqrt64Fx2, Iop_RSqrt64Fx2,

/* --- 64x2 lowest-lane-only scalar FP --- */

/* In binary cases, upper half is copied from first operand.  In
   unary cases, upper half is copied from the operand. */

/* binary */
Iop_Add64F0x2, Iop_Sub64F0x2, Iop_Mul64F0x2, Iop_Div64F0x2,
Iop_Max64F0x2, Iop_Min64F0x2,
Iop_CmpEQ64F0x2, Iop_CmpLT64F0x2, Iop_CmpLE64F0x2, Iop_CmpUN64F0x2,

/* unary */
Iop_Recip64F0x2, Iop_Sqrt64F0x2, Iop_RSqrt64F0x2,

/* --- pack / unpack --- */

/* 64 <-> 128 bit vector */
Iop_V128to64,     // :: V128 -> I64, low half
Iop_V128HIto64,   // :: V128 -> I64, high half
Iop_64HLtoV128,   // :: (I64,I64) -> V128

Iop_64UtoV128,
Iop_SetV128lo64,
/* 32 <-> 128 bit vector */
Iop_32UtoV128,
Iop_V128to32,     // :: V128 -> I32, lowest lane
Iop_SetV128lo32,  // :: (V128,I32) -> V128
```

128-bit SIMD integer

```
/* BITWISE OPS */
Iop_NotV128,
Iop_AndV128, Iop_OrV128, Iop_XorV128,

/* MISC (vector integer cmp != 0) */
Iop_CmpNEZ8x16, Iop_CmpNEZ16x8, Iop_CmpNEZ32x4, Iop_CmpNEZ64x2,

/* ADDITION (normal / unsigned sat / signed sat) */
Iop_Add8x16,   Iop_Add16x8,   Iop_Add32x4,  Iop_Add64x2,
Iop_QAdd8Ux16, Iop_QAdd16Ux8,
Iop_QAdd8Sx16, Iop_QAdd16Sx8,

/* SUBTRACTION (normal / unsigned sat / signed sat) */
Iop_Sub8x16,   Iop_Sub16x8,   Iop_Sub32x4,  Iop_Sub64x2,
Iop_QSub8Ux16, Iop_QSub16Ux8,
Iop_QSub8Sx16, Iop_QSub16Sx8,

/* MULTIPLICATION (normal / high half of signed/unsigned) */
Iop_Mul16x8,
Iop_MulHi16Ux8,
Iop_MulHi16Sx8,

/* AVERAGING: note: (arg1 + arg2 + 1) >>u 1 */
Iop_Avg8Ux16,
Iop_Avg16Ux8,

/* MIN/MAX */
Iop_Max16Sx8,
Iop_Max8Ux16,
Iop_Min16Sx8,
Iop_Min8Ux16,

/* COMPARISON */
Iop_CmpEQ8x16,  Iop_CmpEQ16x8,  Iop_CmpEQ32x4,
Iop_CmpGT8Sx16, Iop_CmpGT16Sx8, Iop_CmpGT32Sx4,

/* VECTOR x SCALAR SHIFT (shift amt :: Ity_I8) */
Iop_ShlN16x8, Iop_ShlN32x4, Iop_ShlN64x2,
Iop_ShrN16x8, Iop_ShrN32x4, Iop_ShrN64x2,
Iop_SarN16x8, Iop_SarN32x4,

/* NARROWING -- narrow 2xV128 into 1xV128, hi half from left arg */
Iop_QNarrow16Ux8,
Iop_QNarrow16Sx8,
Iop_QNarrow32Sx4,

/* INTERLEAVING -- interleave lanes from low or high halves of
   operands.  Most-significant result lane is from the left
   arg. */
```

```
Iop_InterleaveHI8x16, Iop_InterleaveHI16x8,
Iop_InterleaveHI32x4, Iop_InterleaveHI64x2,
Iop_InterleaveLO8x16, Iop_InterleaveLO16x8,
Iop_InterleaveLO32x4, Iop_InterleaveLO64x2
```

9 Appendix D: Implementing AMD64 support in the IR framework

The AMD64 instruction set is a somewhat cleaned-up version of x86, extended to 64 bits and with larger register sets. Consequently, showing that x86 code can be viably translated to IR also demonstrates beyond reasonable doubt that AMD64 can be supported too. This section therefore describes the x86 to IR translation, but applies equally to AMD64 to IR conversion, unless otherwise stated.

Translation of x86 is for the most part straightforward. There are two difficult points: condition codes, and the x87 FPU register stack.

9.1 Integer x86/AMD64 instructions

Ignoring condition codes, x86 translations are simple enough. For example, instruction

```
subl %edi,-56(%ebp)
```

subtracts %edi from the value in memory at %ebp-56. Here is one IR rendition (there are of course many valid translations) which makes explicit the address calculation (t5), the load (t4), the operation (t2), and the final store:

```
t5 = Add32(GET(20,I32),0xFFFFFFC8:I32)
t4 = LDle:I32(t5)
t3 = GET(28,I32)
t2 = Sub32(t4,t3)
STle(t5) = t2
```

"rep"-prefixed string operations are translated into an IR sequence which performs one iteration of the instruction and jumps back to itself. This is one place where conditional exits from IR blocks are important.

Condition codes present the only real challenge. x86 has six condition codes: O (overflow), S (sign), Z (zero), A (auxiliary carry), C (carry) and P (parity). The difficulty is that (1) most integer instructions (add/sub/mul/and/or/xor, and more) set all the condition codes, and (2) computing them all is prohibitively expensive, taking around 20 host instructions.

One standard solution, as described in [Cmelik & Keppel 93, sec 6.3.4] and widely used, is to observe that the condition codes are almost always dead, that is, are overwritten prior to use. Therefore it is appropriate to use a deferred evaluation ("thunk") scheme. It turns out that storing three words of information after every condition code setting guest instruction provides enough information to compute the condition codes later, if needed. One of the three words is an indication of the operation (add/sub/logic/shift-left/shift-right/rotate-left/rotate-right/inc/dec and various kinds of multiply). The other two words are usually, but not always, the operand values.

The x86 to IR translation does not represent the %eflags register directly. Instead, these three words are allocated space in the guest state, and %eflags is synthesised on those rare occasions when it is needed. In the following example, the three words are stored at guest state offsets 32, 36 and 40. Here, then, is the full translation of a subtract instruction:

```
subl %edx,%eax
```

becomes

```
t5 = GET(0,I32)         -- get %eax
t4 = GET(8,I32)         -- get %edx
t3 = Sub32(t5,t4)       -- result
PUT(32) = 0x9:I32       -- store operation, 9 == 32-bit sub
PUT(36) = t4            -- store one operand
PUT(40) = t5            -- store the other
PUT(0) = t3             -- store %eax
```

Note how this interacts naturally with the redundant-Put optimisation. If a second condition code setting instruction in the same basic block follows this one, and there is no intervening use of the condition codes, redundant Put elimination will remove the Puts to 32, 36 and 40, in effect recognising that the condition codes are dead immediately after the subtract. Pleasingly, this x86/AMD64-specific transformation is achieved by the architecture-neutral IR-level optimisation machinery.

There are some awkwardnesses. A few x86 instructions, most notably inc and dec, set some but not all condition codes. Therefore it is first necessary to compute the previous value of the condition codes when building the 3-word thunk for those instructions, which is a drag on performance.

Most instructions that read the condition codes are conditional jumps. A conditional jump translates to an IR conditional exit whose condition is established by calling a clean helper function to evaluate the condition codes, based on the stored thunk:

```
jz-32 0x4204E694
```

becomes

```
if (32to1(calculate_condition( 0x4:I32,
                               GET(32,I32),
                               GET(36,I32),
                               GET(40,I32)):I32))
   goto 0x4204E694:I32
```

Crucially, the 3-word thunk is passed in pieces to the helper as args 2, 3 and 4. The first argument, 0x4, indicates which condition the helper should evaluate.

This again is designed to interact well with IR optimisation. Firstly, redundant-Get elimination will replace the Gets with the thunk values stored by some earlier flag-setting instruction, assuming it occurs in the same basic block, which is highly likely. Secondly, constant folding/propagation may turn the operation argument (the second arg) into a literal value. If it does, the IR optimiser's partial-evaluation mechanism for clean helpers may be able to replace the call by a simple condition, a big improvement. Consider these two instructions, shown with their initial IR:

```
testl %eax,%eax
```

```
      t18 = GET(0,I32)
      t17 = GET(0,I32)
      t16 = And32(t18,t17)
      PUT(32) = 0xF:I32
      PUT(36) = 0x0:I32
      PUT(40) = t16

   jz-32 0x4204E694

      if (32to1(calculate_condition(
                   0x4:I32,GET(32,I32),
                   GET(36,I32),GET(40,I32)):I32)) goto 0x4204E694:I32
```

"testl" ANDs its arguments, sets the condition codes and throws away the result. "jz" jumps if the zero flag is set. After IR optimisation this becomes:

```
   t38 = GET(0,I32)            -- t38 = %eax
   PUT(32) = 0xF:I32           -- thunk op = AND32
   PUT(36) = 0x0:I32           -- (not important)
   PUT(40) = t38               -- store thunk arg2
   if (32to1(1Uto32(CmpEQ32(t38,0x0:I32)))) goto 0x4204E694:I32
```

The conditional jump simply checks if guest %eax is zero and jumps if so. Note that although the expensive helper call to evaluate the flag thunk is removed, it's still necessary to build the thunk (Put 32, 36, 40) just in case some basic block executed after this one wishes to inspect the flags prior to setting them.

9.2 Dealing with x86 floating point

At the outset it is worth mentioning that a design goal for IR as a whole (and hence for future Valgrinds) is to provide floating point arithmetic support in accordance with the widely used IEEE754-1985 standard.

x86 legacy floating-point support is provided by Intel 8087-compatible architectural features, namely a stack of 8 80-bit floating point registers, and instructions to deal with them. This architecture is widely regarded as difficult to work with.

More modern x86 implementations include SSE2 instructions, which provide a cleaner implementation of scalar and vector floating point support. Support of SSE2-based floating point is not regarded as a significant challenge for the IR framework, and is not discussed further.

Dealing with the 8087 ("x87") floating point architecture is challenging for several reasons:

- The floating point registers are 80 bits long, with 64 bit mantissas and 15 bit exponents. This differs from almost all other IEEE754-1985 compliant implementations (except Itanium), which only offer the 754-mandated formats. In particular the largest widely supported format is 64-bit (52 mantissa bits, 11 exponent bits).

  This forces an awkward design choice on IR. Should IR include 80-bit floating point types and operations, in order that x87 floating point can be represented exactly? Or should it avoid acquiring x87-specifics, and instead only support 754-mandated formats and operations?

The current decision is the latter: only support IEEE754. This means 80-bit values on an x86 guest will be approximated by 64-bit values. There will be some loss of range and accuracy: 80-bit floating point values can represent numbers in the range 1e+/-4920, roughly, and with about 19 significant figures accuracy. 754-compliant 64-bit values can only represent values in the range 1e+/-308, and with 16 significant figures.

There will therefore be small differences in floating point results for programs run natively compared to running on IR-based Valgrind. However these are not believed to be significant:

- Any program which functions significantly differently with 80-bit vs 64-bit floating point arithmetic is inherently unportable, since only the x86 and IA64 architectures support 80-bit arithmetic. PowerPC, POWER5, Alpha, Sparc, etc, do not.
- Even on x86, 80-bit accuracy is not guaranteed. Specifically, if the compiler chooses to do floating point arithmetic via the SSE2 facilities, this will be at 64-bit accuracy, since SSE2 does not support 80-bit arithmetic.

In short, IR aims to provide an IEEE754-1985 compliant floating point implementation, and that should be adequate for all reasonable purposes.

- x87 supplies a stack (really, a circular array) of 8 registers. A 3-bit stack pointer indicates a notional current top-of-stack register. All instructions refer to registers relative to this stack pointer -- there is no direct naming of the underlying registers. This means that the Get and Put mechanism is useless for simulating register accesses since they only allow fixed guest state offsets. Instead, every register reference requires adding the current stack pointer value to the stated register number, and using that to circularly index into an array of 8 64-bit floating point values in the guest state.

  This is the initial motivation for the providing GetI/PutI, which do run-time indexing of the guest state. x87 floating point register accesses require scaling, masking and offsetting operations to fetch the actual value. Although this translates to short sequences of host instructions, this work has to be done for each and every FP register access, and so presents a significant performance overhead.

  One challenge therefore is to ensure that IR-level optimisations remove most of this overhead. It is believed that common-subexpression-elimination will be an effective measure.

  A second IR-optimisation challenge is to make redundant-Put/Get elimination work for the PutI/GetI variants. This is needed to enable floating point values to be stored in host registers across multiple guest instructions -- without it, FP values are flushed back the the guest state after every guest instruction. It is believed this is possible with moderate extra complexity in the IR optimiser.

- A further complication is that x87 defines each register to have two associated tag bits, which indicate (amongst things) whether that register is notionally empty or full. Reading an empty register produces a NaN; writing a full register causes a NaN to be written. This mechanism must also be simulated in IR, using the same GetI/PutI mechanism as for the FP registers themselves.

  The challenge is to ensure, once again, that most of this overhead can be optimised away at the IR level.

  The most irritating thing about the register tags is that all this work is totally pointless. What is achieves is to detect misuses of the FP stack mechanism. Yet any compiler which emits code with such misuses is emitting incorrect code, which will not get correct results. And so all compiler-generated x87 code correctly observes the stack discipline, rendering the tag simulation pointless.

One positive aspect of developing IR optimisations capable of removing this baggage is that similar techniques will be required to support rotating registers in the IA64 architecture. Solving such problems at this stage enhances the prospects of a viable port of Valgrind to IA64.

- Exceptions in floating point code. IEEE754 defines five types of exceptions which may arise:
    - Invalid operation (sqrt of negative number, etc)
    - Division by zero
    - Overflow
    - Underflow
    - Inexact (loss of precision)

  For each of the five kinds, two courses of action are possible.
    - A user-defined exception handler may be called when the exception happens.
    - Alternatively, 754 defines, for each exception, a default action, which "fixes things up" and allows the computation to proceed without intervention from a signal handler. For example, the default fixup action for Invalid operation is to write a NaN to the operation's destination.

  As planned, IR will only support the default fixup actions. It is believed that the default fixup actions are suitable in almost all circumstances. If this is not the case, and user-defined exception support is required, the design will have to be reviewed. It is not believed that serious problems will result; however, exception support may reduce performance of floating point code.

  In any case, and much to the disappointment of the 754 committee, proper programming language support for 754 has never really materialised, despite almost all hardware supporting it. As a result programmers are often unaware of, and/or unable to use, all the facilities that 754-compliant hardware offers. In particular there is no portable way to use the 754-defined exception mechanism from C, C++, Fortran (?).
- Rounding: IR does observe the 4 IEEE-mandated rounding modes (to nearest, to +infinity, to -infinity, to zero) for float to integer conversions, and for float-to-float rounding to integral values. Currently only round-to-nearest is supported for all other floating point operations. As with exceptions, this is not believed to be a significant problem, and if it does turn out to be so, the design can be revisited.

10 Bibliography

[Cmelik & Keppel 93]

Robert F. Cmelik and David Keppel. **Shade: A Fast Instruction-Set Simulator for Execution Profiling.** 1993. Technical Report UWCSE 93-06-06. University of Washington.

[FHP 92]

Christopher Fraser, David Hanson, Todd Proebsting. **Engineering a simple, efficient code-generator generator.** 1992. ACM Letters on Programming Languages and Systems 1(3), 213-226.

[Fraser & Hanson 95]

Christopher Fraser and David Hanson. **A Retargetable C Compiler: Design and Implementation.** 1995. Addison-Wesley Publishing Company, ISBN 0-8053-1670-1

[Naishlos 04]

Dorit Naishlos. **Autovectorisation in GCC.** 2004. Proceedings of the 2004 GCC Developers' Summit, June 2-4th, 2004, Ottawa Congress Centre, Ottawa, Canada.

[Nethercote & Seward 03]

Nicholas Nethercote and Julian Seward. **Valgrind: A Program Supervision Framework.** July 2003. Proceedings of the Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA.

[Nethercote 04]

Nicholas Nethercote. **Dynamic Binary Analysis and Instrumentation.** 2004. PhD thesis (in preparation), University of Cambridge Computer Laboratory.

[Seward 02]

Julian Seward. **The design and implementation of Valgrind.** March 2002. Technical documentation supplied with all source distributions of Valgrind. Currently (Sept 04) an on-line copy exists at http://developer.kde.org/~sewardj/docs-2.2.0/mc_techdocs.html

[THS 98]

Omri Traub, Glenn Holloway and Michael Smith. **Quality and Speed in Linear-scan Register Allocation.** 1998. ACM PLDI98, pp142-151.