Verified Just-In-Time Compiler on x86

Magnus O. Myreen
Computer Laboratory, University of Cambridge
magnus.myreen@cl.cam.ac.uk

Abstract

This paper presents a method for creating formally correct just-in-time (JIT) compilers. The tractability of our approach is demonstrated through, what we believe is the first, verification of a JIT compiler with respect to a realistic semantics of self-modifying x86 machine code. Our semantics includes a model of the instruction cache. Two versions of the verified JIT compiler are presented: one generates all of the machine code at once, the other one is incremental i.e. produces code on-demand. All proofs have been performed inside the HOL4 theorem prover.

General Terms software verification, formal methods

Keywords self-modifying code, compiler verification, just in time

1. Introduction

Just-in-time (JIT) compilation is an effective technique for boosting the speed of program interpreters. The idea of JIT compilation, i.e. to dynamically translate input programs into native machine code, then execute only native code, is an old invention which dates back to 1960 [15]. However, the concept is still relevant today as JIT compilation is considered vital for competitive interpreter-based implementations of modern languages, Java, C# and ML etc.

To date, there seems to be no publications on verification of a full JIT compiler. The reasons for this is likely to lie in the fact that verification of JIT compilers poses a number of challenges that are generally considered hard:

1. **Compiler verification.** A JIT compiler needs to correctly map its input programs down to concrete machine code. In this case the target must be real machine code (numbers), not assembly code or intermediate code which most other verified compilers seem to output.
2. **Non-static code.** Conventional static compilers can treat generated code purely as data, since execution of generated code is not done during compilation. However, JIT compilers switch between execution of static code (the JIT compiler) and dynamically generated code; hence some data needs to be treated as code.
3. **Self-modifying code.** The dynamically generated code might also cause self-modification: the generated code can, in incremental JIT compilers, invoke the code generator which may alter the code that called it (Section 5.1 provides an example). Code that can modify itself poses verification challenges that only few have tackled [8, 6].
4. **Pointer arithmetic and code pointers.** Pointer arithmetic and particularly code pointers, both natural parts of JIT compilation and execution of the generated code, have been considered hard to deal with in a formal context.

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01…$10.00

This paper presents a method for creating formally correct JIT compilers. The tractability of our approach is demonstrated through, what we believe is the first, verification of a JIT compiler with respect to a semantics of self-modifying x86 code. The contributions of this paper are:

a) a semantics suitable for verification of self-modifying x86 code, i.e. an operational semantics which takes into account the hazards of a possibly out-of-date instruction cache.
b) a Hoare logic that fits on top of (a) and can reason about self-modifying code and code pointers by treating code as data and the program counter as a normal register,
c) a workflow, using (b), with which we verified two JIT compilers: one generates all of the machine code at once, the other one is incremental i.e. produces code only on-demand.

The input language of our verified JIT compiler is a simple stack-based bytecode (Section 3) with support for a few branch instructions and stack operations: pop, push, subtract and swap.

Our definitions and proofs have been developed inside the HOL4 theorem prover [26], which greatly helped keeping track of all the details involved. Our HOL4 proof scripts are available online [1] together with the verified JIT compilers. A very basic benchmark of how efficient the verified JIT compilers are when executed on real x86 hardware is presented in Section 4.6.

2. Main ideas

This section presents a high-level summary of the main ideas that make verification of JIT compilers possible. Subsequent sections explain the technical details of our proofs (Sections 3, 4, 5) and verification framework (Sections 6, 7).

2.1 Operational semantics

The basis of this work is a detailed and extensively tested semantics of x86 machine code (Section 6). This semantics is carefully designed to be suitable for verification of self-modifying x86 code, i.e. it includes an instruction cache that exposes the intricacies of a possibly out-of-date instruction cache. The memory is modelled as a function m and the instruction cache modelled as a function i:

1. instruction-fetches read instruction cache i;
2. data-reads and -writes access only memory m, and
3. occasionally i is updated with values from m.

Cache i can be out-of-date: reading from i might not return values accurate with respect to what is stored in m.

The model covers a number of 32-bit user-mode x86 instructions together with their precise bit-level fetch and decode.

2.2 Machine-code Hoare logic

We next set up a Hoare logic (informally described here, but formally defined in Section 7) to ease reasoning about the detailed x86 semantics. We define a Hoare triple $\{p\}\ c\ \{q\}$, in Section 7.4, as an abbreviating statement above the x86 semantics, and then prove each individual Hoare-triple statement as a theorem from the x86 semantics. This approach, from [9], contrasts with the original Hoare logic [11] where certain Hoare triples were axioms.

The three key features of this machine-code Hoare logic are:

1. the program counter is treated as a normal register,
2. code is simply safe-to-execute (instruction-cache accurate) data
3. updates to the cache 'are made local' using a cache abstraction.

A few examples will explain these features. The treatment of the program counter as a normal register can be observed in the following Hoare triple. This Hoare-triple theorem describes an x86 instruction `xchg eax,ebx`, encoded as `93`, which swaps the value v of register eax with the value w of register ebx, and simultaneously adds 1 (the byte length of the instruction) to the value p of program counter pc. For now, read $*$ informally as 'and'. This separating conjunction $*$ is defined and explained in Section 7.1.

$$\begin{array}{l}\{\mathsf{eax}\ v * \mathsf{ebx}\ w * \mathsf{pc}\ p\}\\ p : \mathtt{93}\\ \{\mathsf{eax}\ w * \mathsf{ebx}\ v * \mathsf{pc}\ (p+1)\}\end{array}$$

Direct assignments to the program counter are performed by jumps to code pointers, e.g. x86 instruction `jmp eax`, encoded as `FFE0`, assigns the value of eax to the program counter pc:

$$\begin{array}{l}\{\mathsf{eax}\ v * \mathsf{pc}\ p\}\\ p : \mathtt{FFE0}\\ \{\mathsf{eax}\ v * \mathsf{pc}\ v\}\end{array}$$

This treatment of the program counter significantly eases the effort involved in reasoning about code pointers.

The fact that code is simply safe-to-execute data can be observed from the following four theorems. First, code need *not* be in the middle of a Hoare triple:

$$\{p\}\ c\ \{q\} \quad = \quad \{p * \mathsf{code}\ c\}\ \emptyset\ \{q * \mathsf{code}\ c\}$$

Second, code can always be weakened to data in the postcondition:

$$\{p\}\ c\ \{q * \mathsf{code}\ \lfloor d \rfloor\} \implies \{p\}\ c\ \{q * \mathsf{datax}\ d\}$$

Third, data in the precondition can be strengthened into code:

$$\{p * \mathsf{datax}\ d\}\ c\ \{q\} \implies \{p * \mathsf{code}\ \lfloor d \rfloor\}\ c\ \{q\}$$

Fourth, arbitrary data can be turned into code (i.e. data can be made safe to execute) as a side-effect of certain jump instructions, e.g. `jmp eax`, which is encoded as `FFE0` and we already saw in a theorem above, also fits the theorem:

$$\begin{array}{l}\{\mathsf{eax}\ v * \mathsf{pc}\ p * \mathsf{datax}\ d\}\\ p : \mathtt{FFE0}\\ \{\mathsf{eax}\ v * \mathsf{pc}\ v * \mathsf{code}\ \lfloor d \rfloor\}\end{array}$$

These Hoare triples $\{p\}\ c\ \{q\}$ satisfies unconventional properties. Their formal definition is given in Section 7, but for now informally read them as: for any state which satisfies $p * \mathsf{code}\ c$, execution of the x86 semantics will always reach a state for which $q * \mathsf{code}\ c$ holds, and furthermore, no other part of the state was modified, i.e. our Hoare triple satisfies the frame rule from separation logic [24],

$$\{p\}\ c\ \{q\} \implies \forall r.\ \{p * r\}\ c\ \{q * r\}$$

which is essential for local reasoning.

In order to support the frame rule, we had to introduce a lightweight cache abstraction (Section 7.3) which makes the non-local updates to the instruction cache seem local.

2.3 Verification of JIT compilers

We have used our Hoare logic for self-modifying x86 to construct two JIT compilers based on the following work flow.

1. We start by defining the syntax and operational semantics of the input bytecode. Let $\xrightarrow{\mathsf{next}}$ be a relation which describes one step of the execution and let $\xrightarrow{\mathsf{exec}}$ be sequence of $\xrightarrow{\mathsf{next}}$ that end with execution of a special stop instruction. This semantics represents states as tuples (xs, l, p, cs) that consist of a stack xs, a natural number l which keeps track of available stack space, a bytecode program cs and a program counter p.
2. A coupling invariant, jit_inv, is then defined which allows us to relate states in the bytecode semantics to states in the x86 code, given a base address a:

   $$\forall s\ t\ a.\ \ s \xrightarrow{\mathsf{next}} t \implies \{\mathsf{jit\_inv}\ s\ a\}\ \emptyset\ \{\mathsf{jit\_inv}\ t\ a\}$$

   Informally, this jit_inv assertion requires, for bytecode state (xs, l, p, cs) and base address a, that a stack is located in x86 memory containing xs and free space l, that x86 code equivalent to bytecode program cs is stored in memory from address a onwards and that the x86 program counter contains an x86 address equivalent to bytecode program counter p.

   The invariant for the incremental version of our JIT compiler further requires that a code generator is present and that a mapping is maintained which keeps track of what and where bytecode instructions have been translated into x86 code.
3. It then follows that each successful execution $\xrightarrow{\mathsf{exec}}$ in the bytecode semantics is mimicked by the x86 implementation. The x86 code for the special stop instruction exits by jumping to an address w given in register edx. Here stack is the stack assertion from inside jit_inv.

   $$\begin{array}{l}(xs, l, p, cs) \xrightarrow{\mathsf{exec}} (xs_2, l_2, p_2, cs_2) \implies\\ \{\mathsf{jit\_inv}\ (xs, l, p, cs)\ a * \mathsf{edx}\ w\}\\ \emptyset\\ \{\mathsf{stack}\ (xs_2, l_2) * \mathsf{pc}\ w * \mathsf{edx}\ w * \mathsf{T}\}\end{array}$$

   This theorem guarantees that any x86 state which satisfies jit_inv will correctly perform evaluations according to $\xrightarrow{\mathsf{exec}}$.
4. Finally, it remains to produce verified x86 machine code that can establish an appropriate state which satisfies jit_inv. For this we needed verified x86 code which implements code generation i.e. the translation from bytecode to x86 code. By exploiting backwards compatibility to previously proved synthesis tools [22], we were able to easily create this x86 code from verified functional descriptions.

   The final correctness theorem follows as a result of composing the code for establishing jit_inv with the code for executing the bytecode program.

The resulting correctness theorem states that the JIT compiler correctly implements $\xrightarrow{\mathsf{exec}}$: given a stack, a string representing the encoding of a bytecode program, and enough space to fit the generated x86 code, the JIT compiler will always terminate in a state where the stack has been updated according to $\xrightarrow{\mathsf{exec}}$.

3. Input language

To make this paper readable we chose a simple input language for our JIT compiler.

Syntax. The input language is a stack-based bytecode, supporting the following instructions. Here i is a 7-bit immediate constant.

pop	pop top element off the stack
sub	subtract
swap	swap top two stack elements
push i	push value i onto stack
jump i	jump to instruction i
jeq i	jump to i, if top two equal
jlt i	jump to i, if top two less
stop	halt execution

Bytecode programs are lists of the above instructions.

Concrete encoding. The abstract syntax restricts constants to 7-bit values in order to make the concrete encoding (almost) readable as a string. Constants are encoded as a character `"0"` $+\ i$. Here ord maps characters to natural numbers, chr is the inverse of ord.

$$\mathsf{imm}\ i \ = \ \mathsf{chr}\ (\mathsf{ord}\ \texttt{"0"} + i)$$

This encoding of immediate constants makes small constants 1, 2, 3 readable strings `"1"`, `"2"`, `"3"`, while larger constants are less intuitive, e.g. 41, 42, 43 are encoded as `"X"`, `"Z"`, `"["`.

The concrete encoding of individual instructions is defined as the function enc. Here ++ concatenates strings.

$$\begin{array}{lcl}
\mathsf{enc}\ (\mathsf{pop}) & = & \texttt{"p"} \\
\mathsf{enc}\ (\mathsf{sub}) & = & \texttt{"-"} \\
\mathsf{enc}\ (\mathsf{swap}) & = & \texttt{"s"} \\
\mathsf{enc}\ (\mathsf{push}\ i) & = & \texttt{"c"} \mathbin{++} \mathsf{imm}\ i \\
\mathsf{enc}\ (\mathsf{jump}\ i) & = & \texttt{"j"} \mathbin{++} \mathsf{imm}\ i \\
\mathsf{enc}\ (\mathsf{jeq}\ i) & = & \texttt{"="} \mathbin{++} \mathsf{imm}\ i \\
\mathsf{enc}\ (\mathsf{jlt}\ i) & = & \texttt{"<"} \mathbin{++} \mathsf{imm}\ i \\
\mathsf{enc}\ (\mathsf{stop}) & = & \texttt{"."}
\end{array}$$

Bytecode programs are encoded as the concatenation of the individual instruction encodings:

$$\begin{array}{lcl}
\mathsf{encode}\ [] & = & \texttt{""} \\
\mathsf{encode}\ (c :: cs) & = & \mathsf{enc}\ c \mathbin{++} \mathsf{encode}\ cs
\end{array}$$

This concrete encoding makes small bytecode programs almost readable, e.g. the string `"=6<4-j0sj0."` is the encoding of a bytecode program which calculates the greatest common divisor:

$$\begin{array}{rcl}
0 & : & \mathsf{jeq}\ 6 \\
1 & : & \mathsf{jlt}\ 4 \\
2 & : & \mathsf{sub} \\
3 & : & \mathsf{jump}\ 0 \\
4 & : & \mathsf{swap} \\
5 & : & \mathsf{jump}\ 0 \\
6 & : & \mathsf{stop}
\end{array}$$

Semantics. The effect of executing a bytecode program is defined next. First, let fetch be a function which looks up the next instruction to be executed from a list of instructions:

$$\begin{array}{lcll}
\mathsf{fetch}\ n\ [] & = & \mathsf{none} & \\
\mathsf{fetch}\ n\ (c :: cs) & = & \mathsf{some}\ c & \text{if } n = 0 \\
\mathsf{fetch}\ n\ (c :: cs) & = & \mathsf{fetch}\ (n-1)\ cs & \text{if } n > 0
\end{array}$$

We define the relation $\xrightarrow{\mathsf{next}}$ to describe the effect of executing the next instruction. Here states are tuples (xs, l, p, cs), where xs is the data stack (a list of 32-bit words), l is a natural number which keeps track of available stack space, p is the bytecode program counter and cs is the bytecode program.

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some\ pop}}{(x :: y :: xs, l, p, cs) \xrightarrow{\mathsf{next}} (y :: xs, l{+}1, p{+}1, cs)}$$

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some\ sub}}{(x :: y :: xs, l, p, cs) \xrightarrow{\mathsf{next}} ((x-y) :: y :: xs, l, p{+}1, cs)}$$

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some\ swap}}{(x :: y :: xs, l, p, cs) \xrightarrow{\mathsf{next}} (y :: x :: xs, l, p{+}1, cs)}$$

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some}\ (\mathsf{push}\ i)}{(xs, l{+}1, p, cs) \xrightarrow{\mathsf{next}} (i :: xs, l, p{+}1, cs)}$$

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some}\ (\mathsf{jump}\ i)}{(xs, l, p, cs) \xrightarrow{\mathsf{next}} (xs, l, i, cs)}$$

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some}\ (\mathsf{jeq}\ i) \qquad x = y}{(x :: y :: xs, l, p, cs) \xrightarrow{\mathsf{next}} (x :: y :: xs, l, i, cs)}$$

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some}\ (\mathsf{jeq}\ i) \qquad x \neq y}{(x :: y :: xs, l, p, cs) \xrightarrow{\mathsf{next}} (x :: y :: xs, l, p{+}1, cs)}$$

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some}\ (\mathsf{jlt}\ i) \qquad x < y}{(x :: y :: xs, l, p, cs) \xrightarrow{\mathsf{next}} (x :: y :: xs, l, i, cs)}$$

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some}\ (\mathsf{jlt}\ i) \qquad y \leq x}{(x :: y :: xs, l, p, cs) \xrightarrow{\mathsf{next}} (x :: y :: xs, l, p{+}1, cs)}$$

We use $\xrightarrow{\mathsf{next}}$ to define an inductive relation $\xrightarrow{\mathsf{exec}}$ that describes the effect of successfully executing a bytecode program. Each successful execution must reach the stop instruction.

$$\frac{\mathsf{fetch}\ p\ cs \ = \ \mathsf{some\ stop}}{(xs, l, p, cs) \xrightarrow{\mathsf{exec}} (xs, l, p, cs)}$$

$$\frac{s \xrightarrow{\mathsf{next}} t \qquad t \xrightarrow{\mathsf{exec}} u}{s \xrightarrow{\mathsf{exec}} u}$$

4. Verified JIT compiler – version 1

Given the above definition of an input language, we can construct our verified JIT compilers. As mentioned earlier, two JIT compilers will be presented: one JIT compiler generates all of the target x86 code as an initialisation step, the other one produces code incrementally on-demand. This section describes the first version; the second version is presented in Section 5.

4.1 Informal intuition

The most basic JIT compilers perform the following steps, they:

1. generate native machine code from bytecode, and then
2. let the generated code run on bare metal.

The first version our JIT compiler implements this separation between code generation and code execution.

The code generation, we consider, is also very simple: each bytecode instruction is always translated to the same x86 instructions. Before describing the exact mapping from bytecode instructions to x86 instructions, we start with an informal description of how the stack xs in the state of a bytecode program, from the previous section, is represented on x86:

- register `eax` holds the value of the top of the stack xs,
- register `edi` points to the rest of the stack; we write `[edi]` for the location of the first element of the rest of the stack.

- register `edx` holds the address to which stop is to jump.

The choice of general purpose registers `eax`, `edi`, `edx` is arbitrary, and does not depend on any special x86 features.

The mapping from bytecode instructions is informally the following. Push and pop, translate to some pointer arithmetic and move instruction `mov`, subtraction translates to subtraction on x86, the swap instruction translates to x86 instruction `xchg` (exchange), and jump instructions are translated into jump instructions on x86, conditional jumps require a compare instruction before the jump.

pop	$\rightsquigarrow$	`mov eax,[edi]; add edi,4`
sub	$\rightsquigarrow$	`sub eax,[edi]`
swap	$\rightsquigarrow$	`xchg [edi],eax`
push i	$\rightsquigarrow$	`sub edi,4; mov [edi],eax; mov eax,`i
jump i	$\rightsquigarrow$	`jmp` *offset*
jeq i	$\rightsquigarrow$	`cmp eax,[edi]; je` *offset*
jlt i	$\rightsquigarrow$	`cmp eax,[edi]; jb` *offset*
stop	$\rightsquigarrow$	`jmp edx`

The tedious part is to get the jump offsets correct as x86 instructions vary in length. Fortunately, our proof methodology helped us find and remove our off-by-one bugs early in the design.

4.2 Invariant maintained by the generated x86 code

The previous section described informally a coupling invariant which the generated by x86 code maintains to the bytecode. We start the construction of our JIT compiler by first formalising an invariant jit_inv and proving that execution of each bytecode instruction respects this invariant:

$$\forall s\; t\; a.\;\; s \xrightarrow{\mathsf{next}} t \implies \{\mathsf{jit\_inv}\; s\; a\}\; \emptyset\; \{\mathsf{jit\_inv}\; t\; a\}$$

Invariant jit_inv maintains a stack, makes sure that appropriate x86 code is in memory, and also requires that the value of x86 program counter is set correctly. We will formalise each part of the invariant separately.

The stack consists of an array-like list of 32-bit words xs, that grows from address a upwards. Here M a x asserts that 32-bit word x is located at address a, while emp is just the unit of $*$.

$$\begin{array}{lcl} \mathsf{list}\; a\; [] & = & \mathsf{emp} \\ \mathsf{list}\; a\; (x :: xs) & = & \mathsf{M}\; a\; x * \mathsf{list}\; (a{+}4)\; xs \end{array}$$

We also have to be precise about what space is reserved as free stack space. Let space a n state that there is space for n instances of M elements below address a.

$$\begin{array}{lcl} \mathsf{space}\; a\; 0 & = & \mathsf{emp} \\ \mathsf{space}\; a\; (n{+}1) & = & \exists x.\; \mathsf{M}\; (a{-}4)\; x * \mathsf{space}\; (a{-}4)\; n \end{array}$$

The stack maintained by jit_inv has the following specification: the stack must not be empty [], the top of the stack is located in eax while the rest is placed in memory from some address a upwards, address a is stored in register edi. We also add an assertion which requires that a is 32-bit word aligned, i.e. $a \;\&\; 3 = 0$.

$$\begin{array}{lcl} \mathsf{stack}\; ([], l) & = & \mathsf{F} \\ \mathsf{stack}\; (x :: xs, l) & = & \exists a.\; \mathsf{eax}\; x * \mathsf{edi}\; a * \langle a \;\&\; 3 = 0\rangle * \\ & & \quad \mathsf{list}\; a\; xs * \mathsf{space}\; a\; l \end{array}$$

To formalise that appropriate x86 code is in memory, we need to define the exact x86 encoding into which each bytecode should be translated. A 32-bit immediate constant w is represented as follows in an instruction as a list of 4 bytes. Here w2w translates 32-bit words to 8-bit words, and $\gg$ is logical-shift right.

$$\begin{array}{l} \mathsf{ximm}\; w\; = \\ \quad [\mathsf{w2w}\; w, \mathsf{w2w}\; (w \gg 8), \mathsf{w2w}\; (w \gg 16), \mathsf{w2w}\; (w \gg 24)] \end{array}$$

The encoding of each instruction is now defined as xenc where t is a function which given the address of a bytecode instruction returns the 32-bit address for the corresponding x86 instruction; w2n converts an unsigned n-bit word into a natural number, and in this case w2w converts 7-bit words to 8-bit words.

$$\begin{array}{lcl} \mathsf{xenc}\; t\; (\mathsf{pop}) & = & [\mathtt{8B}, \mathtt{07}, \mathtt{83}, \mathtt{C7}, \mathtt{04}] \\ \mathsf{xenc}\; t\; (\mathsf{sub}) & = & [\mathtt{2B}, \mathtt{07}] \\ \mathsf{xenc}\; t\; (\mathsf{swap}) & = & [\mathtt{87}, \mathtt{07}] \\ \mathsf{xenc}\; t\; (\mathsf{stop}) & = & [\mathtt{FF}, \mathtt{E2}] \\ \mathsf{xenc}\; t\; (\mathsf{push}\; i) & = & [\mathtt{83}, \mathtt{EF}, \mathtt{04}, \mathtt{89}, \mathtt{07}, \mathtt{B8}, \mathsf{w2w}\; i, \mathtt{00}, \mathtt{00}, \mathtt{00}] \\ \mathsf{xenc}\; t\; (\mathsf{jump}\; i) & = & [\mathtt{E9}] +\!\!+ \mathsf{ximm}\; (t\, (\mathsf{w2n}\; i) - 5) \\ \mathsf{xenc}\; t\; (\mathsf{jeq}\; i) & = & [\mathtt{3B}, \mathtt{07}, \mathtt{0F}, \mathtt{84}] +\!\!+ \mathsf{ximm}\; (t\, (\mathsf{w2n}\; i) - 5) \\ \mathsf{xenc}\; t\; (\mathsf{jlt}\; i) & = & [\mathtt{3B}, \mathtt{07}, \mathtt{0F}, \mathtt{82}] +\!\!+ \mathsf{ximm}\; (t\, (\mathsf{w2n}\; i) - 5) \end{array}$$

Using this encoding function we define the length of each encoding. Here n2w converts a natural number into a 32-bit word.

$$\mathsf{xenc\_length}\; c \;=\; \mathsf{n2w}\; (\mathsf{length}\; (\mathsf{xenc}\; (\lambda x.\, 0)\; c))$$

Now we can define a mapping from address p in the bytecode program cs to addresses in the corresponding x86 code, given that the x86 code starts at address a:

$$\begin{array}{lcl} \mathsf{addr}\; cs\; a\; 0 & = & a \\ \mathsf{addr}\; []\; a\; p & = & a \\ \mathsf{addr}\; (c :: cs)\; a\; (p{+}1) & = & \mathsf{addr}\; cs\; (a + \mathsf{xenc\_length}\; c)\; p \end{array}$$

The code assertion, code $\lfloor m \rfloor$, which jit_inv will use, asserts, for each $a \in$ domain m, that the memory of the x86 state contains code byte $m(a)$ at address a. Let memb a m bs make sure that bytes bs appear in m from address a onwards.

$$\begin{array}{lcl} \mathsf{memb}\; a\; m\; [] & = & \mathsf{T} \\ \mathsf{memb}\; a\; m\; (b :: bs) & = & (m(a) = b) \wedge \mathsf{memb}\; (a{+}1)\; m\; bs \end{array}$$

We can now define an assertion which states that m contains code for an entire bytecode program cs, from address a onwards:

$$\begin{array}{l} \mathsf{code\_in\_mem}\; cs\; a\; m \;\; = \\ \quad \forall p\; c.\;\; \mathsf{fetch}\; p\; cs = \mathsf{some}\; c \implies \\ \qquad\quad \mathsf{let}\; branch = (\lambda i.\; \mathsf{addr}\; cs\; a\; i - \mathsf{addr}\; cs\; a\; p)\; \mathsf{in} \\ \qquad\quad\;\; \mathsf{memb}\; (\mathsf{addr}\; cs\; a\; p)\; m\; (\mathsf{xenc}\; branch\; c) \end{array}$$

The value of the x86 program counter should always point at the addr cs a p, i.e. contain the address where instruction with index p is stored.

Using these definitions we define jit_inv as maintaining a stack, program counter, and appropriate x86 code. (Here s hides the value of the eflags that are irrelevant between instructions.)

$$\begin{array}{l} \mathsf{jit\_inv}\; (xs, l, p, cs)\; a \;\; = \\ \quad \mathsf{stack}\; (xs, l) * \mathsf{pc}\; (\mathsf{addr}\; cs\; a\; p) * \mathsf{s} * \\ \quad \exists m.\; \mathsf{code}\; \lfloor m \rfloor * \langle \mathsf{code\_in\_mem}\; cs\; a\; m \rangle \end{array}$$

Proving that each case of $\xrightarrow{\mathsf{next}}$ satisfies jit_inv is a simple exercise of applying the proof rules from Section 7.5 to Hoare triple theorems for each of the instructions that xenc can generate. For example, proving the case for swap starts from theorem:

$$\begin{array}{l} \{\mathsf{eax}\; x * \mathsf{edi}\; w * \mathsf{M}\; w\; y * \mathsf{pc}\; p\} \\ \;p : 8707 \\ \{\mathsf{eax}\; y * \mathsf{edi}\; w * \mathsf{M}\; w\; x * \mathsf{pc}\; (p{+}2)\} \end{array}$$

which then implies the following,

$$\begin{array}{l} (m(p) = 87) \wedge (m(p{+}1) = 07) \implies \\ \{\mathsf{eax}\; x * \mathsf{edi}\; w * \mathsf{M}\; w\; y * \mathsf{pc}\; p\} \\ \;\lfloor m \rfloor \\ \{\mathsf{eax}\; y * \mathsf{edi}\; w * \mathsf{M}\; w\; x * \mathsf{pc}\; (p{+}2)\} \end{array}$$

which in turn leads the following, etc.

$$\begin{array}{l}\mathsf{code\_in\_mem}\ cs\ a\ m \implies \\ \{\mathsf{stack}\ (x :: y :: xs, l) * \mathsf{pc}\ (\mathsf{addr}\ cs\ a\ p) * \mathsf{code}\ \lfloor m \rfloor\} \\ \emptyset \\ \{\mathsf{stack}\ (y :: x :: xs, l) * \mathsf{pc}\ (\mathsf{addr}\ cs\ a\ (p+1)) * \mathsf{code}\ \lfloor m \rfloor\}\end{array}$$

4.3 Bytecode programs are executed by x86 code

The previous section showed how to prove that jit_inv is maintained for each transition of $\xrightarrow{\mathsf{next}}$:

$$\forall s\ t\ a.\ \ s \xrightarrow{\mathsf{next}} t \implies \{\mathsf{jit\_inv}\ s\ a\}\ \emptyset\ \{\mathsf{jit\_inv}\ t\ a\}$$

To prove that all successful executions $\xrightarrow{\mathsf{exec}}$ are handled correctly, we also proved, in much the same manner, the following result for the stop instruction:

$$\begin{array}{l}\mathsf{fetch}\ p\ cs = \mathsf{some\ stop} \implies \\ \{\mathsf{jit\_inv}\ (xs, l, p, cs)\ a * \mathsf{edx}\ w\} \\ \emptyset \\ \{\mathsf{stack}\ (xs, l) * \mathsf{pc}\ w * \mathsf{edx}\ w * \mathsf{T}\}\end{array}$$

The two theorems above are sufficient for proving that any successful execution of a bytecode program, i.e. $\xrightarrow{\mathsf{exec}}$, will always be computed by the x86 code inside jit_inv:

$$\begin{array}{l}(xs, l, p, cs) \xrightarrow{\mathsf{exec}} (xs_2, l_2, p_2, cs_2) \implies \\ \{\mathsf{jit\_inv}\ (xs, l, p, cs)\ a * \mathsf{edx}\ w\} \\ \emptyset \\ \{\mathsf{stack}\ (xs_2, l_2) * \mathsf{pc}\ w * \mathsf{edx}\ w * \mathsf{T}\}\end{array}$$

This theorem follows by a simple induction on the relation $\xrightarrow{\mathsf{exec}}$.

4.4 Implementing code generation

The remaining part of the construction of the JIT compiler is to produce verified x86 machine code that can establish an appropriate state which satisfies jit_inv in the precondition of the theorem above. For this we needed verified x86 code which implements code generation i.e. translation from bytecode to x86 code.

By exploiting backwards compatibility to previously developed synthesis tools [22], we can create x86 code from functional descriptions in HOL4. For example, x86 code for calculating addr, given above, is constructed by first defining, in HOL4, a function x86_addr which we think might calculate addr.

```
x86_addr (r1,r5,r6,g) =
  if r1 = 0w then (r5,g) else
    let r1 = r1 - 1w in
      if (g r6 = n2w (ORD #"-")) ∨ (g r6 = n2w (ORD #"s")) ∨
         (g r6 = n2w (ORD #".")) then
        let r6 = r6 + 1w in let r5 = r5 + 2w in
          x86_addr (r1,r5,r6,g) else
      if g r6 = n2w (ORD #"p") then
        let r6 = r6 + 1w in let r5 = r5 + 5w in
          x86_addr (r1,r5,r6,g) else
      if g r6 = n2w (ORD #"c") then
        let r6 = r6 + 2w in let r5 = r5 + 10w in
          x86_addr (r1,r5,r6,g) else
      if g r6 = n2w (ORD #"j") then
        let r6 = r6 + 2w in let r5 = r5 + 5w in
          x86_addr (r1,r5,r6,g) else
      if (g r6 = n2w (ORD #"=")) ∨ (g r6 = n2w (ORD #"<")) then
        let r6 = r6 + 2w in let r5 = r5 + 8w in
          x86_addr (r1,r5,r6,g) else
          (r5,g)
```

Our synthesis tool can from such functions generate x86 code and prove that the generated x86 code executes the input function. In this case, the tool returns a Hoare-triple theorem which states that x86_addr is executed by the generated code:

$$\begin{array}{l}\mathsf{x86\_addr\_pre}\ (eax, ecx, ebx, g) \implies \\ \{\mathsf{eax}\ eax * \mathsf{ebx}\ ebx * \mathsf{ecx}\ ecx * \mathsf{data}\ g * \mathsf{pc}\ p * \mathsf{s}\} \\ \ p : \mathsf{83F8007449}\ldots \\ \{\mathsf{let}\ (ecx, g) = \mathsf{x86\_addr}\ (eax, ecx, ebx, g)\ \mathsf{in} \\ \ \ \mathsf{eax}\ \_ * \mathsf{ebx}\ \_ * \mathsf{ecx}\ ecx * \mathsf{data}\ g * \mathsf{pc}\ (p{+}78) * \mathsf{s}\}\end{array}$$

By then manually proving that x86_addr calculates addr correctly,

$$\begin{array}{l}\mathsf{string\_in\_memory}(a, \mathsf{encode}\ cs, g) \wedge i < 256 \implies \\ \mathsf{x86\_addr\_pre}\ (\mathsf{n2w}\ i, w, a, g)\ \wedge \\ \mathsf{x86\_addr}\ (\mathsf{n2w}\ i, w, a, g) = (\mathsf{addr}\ cs\ w\ i, g)\end{array}$$

we can produce a formal guarantee that the x86 code produced by our synthesis tool indeed calculates addr.

We produced the full code generator that establishes the invariant jit_inv by synthesising x86 code from functions which were manually proved to correctly write x86 instructions into memory with respect to jit_inv.

4.5 Final correctness theorem

The resulting overall correctness theorem states that the JIT compiler correctly implements $\xrightarrow{\mathsf{exec}}$: given a stack (xs, l), a string representing the encoding of a bytecode program cs, and enough space to fit the generated x86 code, this JIT compiler will always terminate in a state where the stack has been updated according to $\xrightarrow{\mathsf{exec}}$.

$$\begin{array}{l}(xs, l, p, cs) \xrightarrow{\mathsf{exec}} (xs_2, l_2, p_2, cs_2)\ \wedge \\ \mathsf{string\_in\_memory}(a, \mathsf{encode}\ cs, g) \implies \\ \{\mathsf{stack}\ (xs, l) * \mathsf{data}\ g * \mathsf{edx}\ a * \mathsf{enough\_space\_for}\ cs\} \\ \ p : \mathsf{89D789C60F}\ldots \\ \{\mathsf{stack}\ (xs_2, l_2) * \mathsf{pc}\ (p{+}410) * \mathsf{T}\}\end{array}$$

4.6 Basic benchmark

Finally, we conducted experiments to discover how fast or slow this JIT compiler is on real x86 hardware (a 2.6 GHz Intel processor). In order to run our verified machine code, we wrote a small C wrapper which essentially just jumps to the verified machine code after making the necessary calls to the operating system for allocating heap space with execute permissions enabled. (The verified machine code is supplied to the GNU C compiler gcc as an assembly file consisting of only `.byte` directives.)

Our JIT compiler calculates the greatest common divisor (GCD) using the bytecode program `"=6<4-j0sj0."` (Section 3) in very fast times for small inputs, e.g. 135 and 345:

```
$ time ./jit "=6<4-j0sj0." 135 345
Top of stack on exit: 5

real 0m0.003s
user 0m0.001s
sys  0m0.002s
```

Only for large inputs, e.g. the pair 2 and 200,000,000, do we get more reliable measurements:

```
$ time ./jit "=6<4-j0sj0." 2 200000000
Top of stack on exit: 2

real 0m0.131s
user 0m0.128s
sys  0m0.003s
```

Incidentally, the time, 0.128 seconds, matches the execution time of a C program which calculates GCD using a loop:

```
while (x != y)
  { if (x < y) { y = y - x; } else { x = x - y; } }
```

Our C program `gcd.c` is compiled using gcc version 4.0.1 and optimisation flag `-O3`:

```
$ gcc -O3 gcd.c -o gcd
$ time ./gcd 200000000 2
GCD: 2

real 0m0.129s
user 0m0.127s
sys  0m0.002s
```

Our JIT compiler makes no optimisations and thus the comparison can easily be made in favour of gcc by simply providing our JIT compiler with less optimal bytecode, e.g. `"=8<4sj0s-sj0."`. However, the point which we want to make is that simple JIT compilers produced as described above are *not necessarily slow*; by providing the JIT compilers with optimised bytecode we can get competitive performance.

5. Verified JIT compiler – version 2 – incremental

The previous section presented a JIT compiler which generates all of the native code at once and then jumps to the generated code. This section describes the construction of a verified JIT compiler which only generates code on-demand, incrementally.

The construction and proof of this incremental JIT compiler follows very closely the workflow of the simpler non-incremental JIT compiler. Therefore, this section will mainly concentrate on describing the intuition of how this incremental JIT compiler works, and will point out how the invariant used in the proof had to be modified.

5.1 Informal intuition

The basic intuition for our incremental JIT compiler is that each non-branching instruction (sub, swap, pop, push,stop) is generated as before, but branch instructions (jump, jlt, jeq) will be generated with calls to the code generator, e.g. bytecode instruction jeq i at location p will always initially result in the following x86 code:[1]

```
    cmp eax,[edi]
    je L
    xor ecx, p+1
    call ebx
L:  xor ecx, i
    call ebx
```

When this code is run one of the `call` instructions will call the code generator, whose address is in register `ebx`. The value in register `ecx` will tell the code generator for which bytecode instruction it should generate x86 code.

Suppose for this example that the x86 code calls the code generator using the first call, i.e. with $p + 1$ in `ecx`, then the code generator will perform a look-up in a table it maintains of where and what bytecode instructions have been translated to x86 code. If it finds that $p + 1$ has not yet been generated then it translates the longest sequence of non-jump instructions around $p + 1$ into native code and replaces the `xor` and `call` instructions with a jump directly to desired location in the new code, i.e. the code becomes:

```
    cmp eax,[edi]
    je L
    jmp G
L:  xor ecx, i
    call ebx
    ...
    (new code starts here)
G:  (code for instruction p + 1 in bytecode)
    (new code ends here)
```

[1] Here `xor ecx,`i has the effect of `ecx`$:=i$ since, whenever such `xor` instructions are encountered, `ecx` contains zero. We chose to use `xor` instead of `mov` because `xor` allows us to use a shorter instruction encoding.

In case the code generator had already generated code for instruction $p + 1$, then no new code would have been generated, instead only the new jump instruction would have been inserted.

The interesting aspect of this JIT compiler is that it can generate slightly different x86 code depending on the input, i.e. depending on which branches are taken in which sequence. Branches that are never taken will never be replaced by `jmp` instructions.

5.2 Invariant

This incremental JIT compiler maintains an invariant which, instead of having static code corresponding to the bytecode program, states that a code generator is present and x86 code for part of the bytecode exists.

The interesting part of the invariant describes the state maintained in between calls to the code generator. Its state consists of a partial mapping from bytecode instruction indexes (natural numbers) to locations (32-bit addresses) where that particular bytecode instruction is represented as x86 code. We represent this partial map as a total function to an option type; the type of this mapping, for which we will use variable j, is:

$$\mathbb{N} \rightarrow \mathsf{word32\ option}$$

Defining how code is represented in memory is slightly more complicated now. In particular, each jump can potentially be represented in two ways, either as the combination of `xor` and `call`, or as just a direct `jmp`. The encoding of a jump to bytecode location p, in the generated x86 at address a, is either present as `xor ecx,` p; `call ebx` (encoded as 83F1[i]FFD3) or, if $j\ i = \mathsf{some}\ w$, simply as an unconditional jump `jmp` $(w{-}a{-}5)$ (encoded as E9[$w{-}a{-}5$]). We reuse the definition of ximm from the Section 4.2.

$$\begin{array}{l}\mathsf{enc\_jmp}\ a\ p\ j\ bs\ =\\ \quad (bs = [\mathtt{83}, \mathtt{F1}, \mathsf{n2w}\ p, \mathtt{FF}, \mathtt{D3}]) \lor\\ \quad \exists w.\ (j\ p = \mathsf{some}\ w) \land (bs = [\mathtt{E9}]\ +\!\!+\ \mathsf{ximm}\ (w{-}a{-}5))\end{array}$$

The relation between bytecode instructions and x86 code is defined as $\mathsf{xenc\_inc}\ c\ a\ p\ j\ bs$: a relation which states that byte list bs, at machine address a, is a valid x86 code corresponding to bytecode instruction c, at location p. The first 5 cases are identical to the encoding xenc for the non-incremental version.

$$\begin{array}{l}\mathsf{xenc\_inc}\ (\mathsf{pop})\ a\ p\ j\ bs = (bs = \mathsf{xenc}\ (\lambda x.\ 0)\ \mathsf{pop})\\ \mathsf{xenc\_inc}\ (\mathsf{sub})\ a\ p\ j\ bs = (bs = \mathsf{xenc}\ (\lambda x.\ 0)\ \mathsf{sub})\\ \mathsf{xenc\_inc}\ (\mathsf{swap})\ a\ p\ j\ bs = (bs = \mathsf{xenc}\ (\lambda x.\ 0)\ \mathsf{swap})\\ \mathsf{xenc\_inc}\ (\mathsf{stop})\ a\ p\ j\ bs = (bs = \mathsf{xenc}\ (\lambda x.\ 0)\ \mathsf{stop})\\ \mathsf{xenc\_inc}\ (\mathsf{push}\ i)\ a\ p\ j\ bs = (bs = \mathsf{xenc}\ (\lambda x.\ 0)\ (\mathsf{push}\ i))\\ \mathsf{xenc\_inc}\ (\mathsf{jump}\ i)\ a\ p\ j\ bs = \mathsf{enc\_jmp}\ a\ (\mathsf{w2n}\ i)\ j\ bs\\ \mathsf{xenc\_inc}\ (\mathsf{jeq}\ i)\ a\ p\ j\ bs =\\ \quad \exists bs_0\ bs_1\ bs_2.\ (bs = bs_0\ +\!\!+\ bs_1\ +\!\!+\ bs_2) \land\\ \quad\ (bs_0 = [\mathtt{3B}, \mathtt{07}, \mathtt{0F}, \mathtt{84}, \mathtt{05}, \mathtt{00}, \mathtt{00}, \mathtt{00}]) \land\\ \quad\ \mathsf{enc\_jmp}\ (a{+}8)\ (p{+}1)\ j\ bs_1 \land \mathsf{enc\_jmp}\ (a{+}13)\ (\mathsf{w2n}\ i)\ j\ bs_2\\ \mathsf{xenc\_inc}\ (\mathsf{jlt}\ i)\ a\ p\ j\ bs =\\ \quad \exists bs_0\ bs_1\ bs_2.\ (bs = bs_0\ +\!\!+\ bs_1\ +\!\!+\ bs_2) \land\\ \quad\ (bs_0 = [\mathtt{3B}, \mathtt{07}, \mathtt{0F}, \mathtt{82}, \mathtt{05}, \mathtt{00}, \mathtt{00}, \mathtt{00}]) \land\\ \quad\ \mathsf{enc\_jmp}\ (a{+}8)\ (p{+}1)\ j\ bs_1 \land \mathsf{enc\_jmp}\ (a{+}13)\ (\mathsf{w2n}\ i)\ j\ bs_2\end{array}$$

We can now define what it means for bytecode cs to be represented in memory m according to mapping j: whenever bytecode instruction c is according to j stored at an address w, then there exists some sequence of bytes bs, which represents c, stored in memory m at address w. We use memb defined in Section 4.2.

$$\begin{array}{l}\mathsf{code\_in\_mem}\ cs\ j\ m\ =\\ \quad \forall p\ c\ w.\ (\mathsf{fetch}\ p\ cs = \mathsf{some}\ c) \land (j\ p = \mathsf{some}\ w) \implies\\ \qquad\qquad \exists bs.\ \mathsf{memb}\ w\ m\ bs \land \mathsf{xenc\_inc}\ c\ w\ p\ j\ bs\end{array}$$

We separately assert a well-formedness requirement on mapping j: each x86 representation of a bytecode instruction which is not a jump must, if and only if present in the x86 code, be immediately followed by the x86 representation of the next bytecode instruction. Let ilength c be a function which return the length of each instruction encoding, e.g. ilength (jump i) = 5.

$$\mathsf{is\_jump}\ c = \exists i.\ c \in \{\,\mathsf{jump}\ i, \mathsf{jeq}\ i, \mathsf{jlt}\ i, \mathsf{stop}\,\}$$

```
wellformed $cs\ j$ =
    $\forall c\, p.$ (fetch $p\ cs$ = some $c$) $\wedge$ fetch $(p{+}1)\ cs \neq$ none $\wedge$
        $\neg$is_jump $c \implies$
        (($j\ p$ = none) $\iff$ ($j\ (p{+}1)$ = none)) $\wedge$
        $\forall w.$ ($j\ p$ = some $w$) $\implies$
            ($j\ (p{+}1)$ = some ($w$ + ilength $c$))
```

The incremental JIT compiler's main invariant, i.e. its version of jit_inv, now states that the stack is maintained as stack (xs, l), defined in Section 4.2; the program counter holds a value eip for which there exists some generated x86 code, i.e. $j\ p$ = some eip; the code cs is represented in memory m according to a wellformed mapping j; and in order to make code generator work, we also include that the code generator is present in memory, that ebx holds a pointer to the entry point in the code generator and register ecx is zero (which is necessary for the xor trick to work). Some inessential details are elided with '. . . '.

```
jit_inv $(xs, l, p, cs)\ a$ =
    $\exists m\ j\ eip\ w.$
        stack $(xs, l)$ * pc $eip$ * $\langle j\ p$ = some $eip\rangle$ * s *
        code $\lfloor m \rfloor$ * $\langle$code_in_mem $cs\ j\ m \wedge$ wellformed $cs\ j\rangle$ *
        code (codegen $w$) * ebx $w$ * ecx 0 * . . .
```

The verification proof can be found in our HOL4 proof scripts [1].

6. x86 semantics

We use an operational semantics for (user-mode) x86 machine code which builds on a previously presented semantics of x86 [25]. The following subsections will detail how the sequential instantiation of our previously developed x86 semantics is extended to include an instruction cache and read/write/execute memory permissions.

6.1 x86 state and memory model

The operational semantics represents x86 states as tuples which consist of a register file r, program counter/instruction pointer e, status bits/eflags s, memory m and instruction cache i:

$$(r, e, s, m, i)$$

The type definition makes use of the following data-types:

α option	::=	some α | none
regs	::=	EAX | EBX | ECX | EDX | EDI | ESI | EBP | ESP
eflags	::=	CF | PF | AF | ZF | SF
perm	::=	r | w | x

The type of each x86 state component is:

r	:	regs $\to$ word32
e	:	word32
s	:	eflags $\to$ bool option
m	:	word32 $\to$ (word8 * perm set) option
i	:	word32 $\to$ (word8 * perm set) option

The option type is used in places where the actual value may be missing, e.g. the value of an eflag in s may be any of three values:

some T	has value true (written T),
some F	has value false (written F), or
none	has undefined/unpredictable value.

Memory locations in m can either be absent (none), or present (some). Memory locations contain two components: the 8-bits of data that are stored at the address and a set of read/write/execute permissions describing how this data can be accessed.

All memory reads and writes are defined in the operational semantics using read_mem, write_mem and read_instr.

```
read_mem $a\ (r, e, s, m, i)$ =
    case $m\ a$ of
      none $\to$ none
    | some $(w, p) \to$ if {r} $\subseteq p$ then some $w$ else none
```

```
write_mem $a\ v\ (r, e, s, m, i)$ =
    case $m\ a$ of
      none $\to$ none
    | some $(w, p) \to$ if {w} $\subseteq p$ then
                            some $(r, e, s, m[a \mapsto$ some $(v, p)], i)$
                          else none
```

The function which fetches 8-bits of an instruction gives priority to the instruction cache, and requires the execute permission x.

```
read_instr $a\ (r, e, s, m, i)$ =
    case $(i\ a, m\ a)$ of
      (none, none) $\to$ none
    | (none, some $(w, p)$) $\to$ if {r, x} $\subseteq p$ then some $w$ else none
    | (some $(w, p)$, _) $\to$ if {r, x} $\subseteq p$ then some $w$ else none
```

Our user-mode semantics has no instructions for altering the permissions attached to memory locations; instead permissions remain static through out execution. In reality operating systems provide user-mode programs with procedures they can call to alter such read/write/execute permissions on a per-page granularity.

6.2 Instruction-cache update

Our semantics executes an instruction-cache update $\xrightarrow{\mathsf{icache}}$ before each instruction is performed. Each update deletes some set of old address from the cache and loads, from memory, a new set of addresses into the cache:

$$\begin{array}{l} \quad (r, e, s, m, i) \xrightarrow{\mathsf{icache}} (r_2, e_2, s_2, m_2, i_2) \\ = \\ \quad \exists new\ old. \\ \qquad r_2 = r \wedge e_2 = e \wedge s_2 = s \wedge m_2 = m \wedge \\ \qquad i_2 = \lambda addr.\ \mathsf{if}\ addr \in new\ \mathsf{then}\ m\ addr\ \mathsf{else} \\ \qquad\qquad\qquad \mathsf{if}\ addr \in old\ \mathsf{then\ none\ else}\ i\ addr \end{array}$$

This cache update transition over approximates the number of different updates a real instruction cache can perform, e.g. a real cache cannot load the entire memory. Over approximating possible cache updates is sufficient, since we will prove that no cache update can cause unwanted behaviour.

A noteworthy feature of this cache update transition is that it will never introduce new inaccuracies. We say that the instruction cache is accurate (not out-of-date) for address a if the instruction cache either does not contain an entry for address a or memory location a is correctly represented in the cache, i.e.

$$\mathsf{accurate}\ a\ (r, e, s, m, i) \ = \ i\ a = \mathsf{none} \ \vee\ i\ a = m\ a$$

Automatic cache updates will never introduce inaccurate entries:

$$\forall s\ t\ a.\ s \xrightarrow{\mathsf{icache}} t \ \wedge\ \mathsf{accurate}\ a\ s \ \Rightarrow\ \mathsf{accurate}\ a\ t$$

Similarly inaccuracies might be removed:

$$\forall a\ s.\ \neg(\mathsf{accurate}\ a\ s) \ \Rightarrow\ \exists t.\ s \xrightarrow{\mathsf{icache}} t \ \wedge\ \mathsf{accurate}\ a\ t$$

Address a becomes inaccurate whenever address a is represented in the instruction cache and a store instruction successfully modifies the byte stored at address a using write_mem a v, for some 8-bit data v.

6.3 Next-state relation

The top level next-state relation $\xrightarrow{\text{x86}}$ is defined as a composition of an instruction cache update, then fetch-and-decode, followed by execution of the fetched instruction. The definition of execute can be found in our proof scripts [1]. The function fetch_and_decode is explained in Appendix A.

$$\begin{array}{l} \quad s \xrightarrow{\text{x86}} u \\ = \\ \quad \exists t\ instr\ len. \\ \quad (s \xrightarrow{\text{icache}} t) \wedge \\ \quad (\mathsf{fetch\_and\_decode}\ t = \mathsf{some}\ (instr, len)) \wedge \\ \quad (\mathsf{execute}\ instr\ len\ t = \mathsf{some}\ u) \end{array}$$

Both the decoder and the execute function have been tested extensively against real x86 hardware, as part of previous work [25]. This gives us confidence that our semantics is, if not completely correct, at least very nearly completely right for the instructions it covers:

```
ADC ADD AND CALL CMOVA CMOVB CMOVE CMOVNA CMOVNB
CMOVNE CMOVNS CMOVS CMP CMPXCHG DEC DIV INC JA
JB JE JMP JNA JNB JNE JNS JS LEA LOOP LOOPE
LOOPNE MOV MOVZX MUL NEG NOT OR POP POPAD PUSH
PUSHAD RET SAR SBB SHL SHR SUB TEST XADD XCHG XOR
```

6.4 Clearing the instruction cache

The x86 instruction set has no instruction for the sole purpose of clearing the instruction cache. However, the Intel Manual (March 2009) [12] states that it is safe to execute self-modifying code (in sequential programs) if the following steps are taken:

1. store modified code
2. jump to new code or intermediate code
3. execute new code

This ambiguous description makes it seem safe to assume that some kind of jump is enough to erase any instruction cache inaccuracies that might make the new code unsafe to execute.

In order to make as few assumptions as possible, we will only assume that one type of jump has the ability to clear the instruction cache. We make clearing the instruction cache a side-effect of execution of jump instructions of the form "`jmp r32`", i.e. jumps to a code pointer stored in a 32-bit register. Other jump instructions, such as branch-to-offset, procedure calls, and procedure returns, are not given this extra side-effect. The relevant part of the execute definition sets i to the empty cache, i.e. $\lambda a.$none:

$$\begin{array}{l} \mathsf{execute}\ (\mathsf{Xjmp}\ (\mathsf{Xreg}\ d))\ len\ (r, e, s, m, i) \ = \\ \quad \mathsf{some}\ (r, r(d), s, m, \lambda a.\mathsf{none}) \end{array}$$

All other cases of execute leave the instruction cache i untouched.

6.5 Execution sequences

In the next section we will quantify over all possible x86 execution sequences. For this purpose, we define a valid execution sequence x from initial state s, written x86_seq s x, to be an infinite sequence of x86 states (with type $\mathbb{N} \to$ x86_state) such that

$$x(0) = s$$

and for each natural number n:

$$\begin{array}{ll} x(n) \xrightarrow{\text{x86}} x(n+1) & \text{if } \exists y.\ x(n) \xrightarrow{\text{x86}} y \\ x(n+1) = x(n) & \text{otherwise} \end{array}$$

The above definition of x86_seq makes stuck states repeat forever. Repeating stuck states is reasonable as we will only consider judgements of total-correctness which asserts that each execution sequence x will contain a desirable final state $x(n)$ satisfying some postcondition $post$:

$$\forall x.\ \mathsf{x86\_seq}\ s\ x \implies \exists n.\ post\ (x(n))$$

7. Machine-code Hoare logic

This section presents a definition of a Hoare triple and associated Hoare proof rules which allow local reasoning in the presence of an instruction cache. The work presented in this section builds on previous experience [19, 21] in adapting separation logic [24] to machine languages.

7.1 Separating conjunction: $*$

Conventionally the separating conjunction $*$ is defined to split partial functions. However, for this work, and previous work on Hoare logic for machine languages, we have found that such a separating conjunction is ill suited for machine languages as processor states are essentially multiple different mappings (mappings from register names to register values, memory locations to memory values, status-bit names to bit values etc.).

We choose to use a set-based separating conjunction in order to treat all resources uniformly and hence make the frame rule (presented later) apply to all types of resources simultaneously. Our set-based separating conjunction $*$ splits a set (of state elements) into two sets: $p * q$ is true for set s if s can be split into two disjoint sets u and v such that p is true for u and q is true for v.

$$(p * q)\ s \ = \ \exists u\ v.\ p\ u \wedge q\ v \wedge (u \cup v = s) \wedge (u \cap v = \{\})$$

The separating conjunction $*$ is associative and commutative. Its unit is emp and angled brackets $\langle \ldots \rangle$ will be used for carrying pure boolean assertions ($\forall p\ c\ s.\ (p * \langle c \rangle)\ s = p\ s \wedge c$):

$$\begin{array}{rcl} \mathsf{emp}\ s & = & (s = \{\}) \\ \langle b \rangle\ s & = & (s = \{\}) \wedge b \end{array}$$

This separating conjunction requires states to be represented as sets of state components. Let the type of an x86 state component be defined by a data-type x86_el with the following constructors. A boolean is attached to each memory component to indicate whether of not that byte is accurately represented in the instruction cache.

$$\begin{array}{lll} \mathsf{Eip} & : & \mathsf{word32} \to \mathsf{x86\_el} \\ \mathsf{Reg} & : & \mathsf{regs} \to \mathsf{word32} \to \mathsf{x86\_el} \\ \mathsf{Status} & : & \mathsf{eflags} \to \mathsf{bool\ option} \to \mathsf{x86\_el} \\ \mathsf{Mem} & : & \mathsf{word32} \to (\mathsf{word8} * \mathsf{perm})\ \mathsf{option} \to \mathsf{bool} \to \mathsf{x86\_el} \end{array}$$

This data-type allows us to define a translation function x86set which maps states represented as tuples x86_state, as described in the previous section, to states represented as sets of x86 state elements. Here range $f = \{\ y \mid \exists x.\ f\ x = y\ \}$.

$$\begin{array}{l} \mathsf{x86set}\ (r, e, s, m, i) = \\ \quad \{\ \mathsf{Eip}\ e\ \} \cup \\ \quad \mathsf{range}\ (\lambda a.\ \mathsf{Reg}\ a\ (r\ a)) \cup \\ \quad \mathsf{range}\ (\lambda a.\ \mathsf{Status}\ a\ (s\ a)) \cup \\ \quad \mathsf{range}\ (\lambda a.\ \mathsf{Mem}\ a\ (m\ a)\ (\mathsf{accurate}\ a\ (r, e, s, m, i))) \end{array}$$

Let R r w assert that register r has value w, similarly let S a x assert that eflag a has value x, and let pc assert the value of the instruction pointer.

$$\begin{array}{rcl} (\mathsf{R}\ a\ x)\ s & = & (s = \{\mathsf{Reg}\ a\ x\}) \\ (\mathsf{S}\ a\ x)\ s & = & (s = \{\mathsf{Status}\ a\ x\}) \\ (\mathsf{pc}\ x)\ s & = & (s = \{\mathsf{Eip}\ x\}) \end{array}$$

The above assertions have their intended meaning when used together with the translation function x86set, e.g.

$$\begin{array}{rcl}(\mathsf{R}\;a\;x \ast p)\;(\mathsf{x86set}(r,e,s,m,i)) & \Longrightarrow & (r\;a = x)\\ (\mathsf{S}\;a\;x \ast p)\;(\mathsf{x86set}(r,e,s,m,i)) & \Longrightarrow & (s\;a = x)\\ (\mathsf{pc}\;x \ast p)\;(\mathsf{x86set}(r,e,s,m,i)) & \Longrightarrow & (e = x)\end{array}$$

but $\ast$ also separates between assertions of the same kind:

$$\begin{array}{rcl}(\mathsf{R}\;a\;x \ast \mathsf{R}\;b\;y \ast p)\;(\mathsf{x86set}(r,e,s,m,i)) & \Longrightarrow & a \neq b\\ (\mathsf{S}\;a\;x \ast \mathsf{S}\;b\;y \ast p)\;(\mathsf{x86set}(r,e,s,m,i)) & \Longrightarrow & a \neq b\\ (\mathsf{pc}\;x \ast \mathsf{pc}\;y \ast p)\;(\mathsf{x86set}(r,e,s,m,i)) & \Longrightarrow & \mathsf{F}\end{array}$$

We will often abbreviate R EAX w with just eax w, R EBX w with ebx w etc. Another abbreviating assertion is s which hides the values of the eflags:

$$\mathsf{s} \;=\; \exists c\;p\;a\;z\;s.\;\mathsf{S}\;\mathsf{CF}\;c \ast \mathsf{S}\;\mathsf{PF}\;p \ast \mathsf{S}\;\mathsf{AF}\;a \ast \mathsf{S}\;\mathsf{ZF}\;z \ast \mathsf{S}\;\mathsf{SF}\;s$$

Assertion s is used in theorems where we need to state that the eflags were modified but we want hide their actual values, which are frequently not of interest.

7.2 Memory assertions: M, code, data

The most basic memory assertion B $a\;x$ is defined to state that byte x, which can be read and written but not executed, is located in memory at address a, which might or might not be accurately represented in the instruction cache:

$$(\mathsf{B}\;a\;x)\;s \;=\; \exists acc.\;s = \{\mathsf{Mem}\;a\;(\mathsf{some}\;(x,\{\mathsf{r},\mathsf{w}\}))\;acc\}$$

A similar assertion M, which states that 32-bit data w is at address a, can be defined using four B assertions. Here $w[j{-}i]$ extracts bits i to j (inclusive) from 32-bit word w.

$$\begin{array}{rcl}\mathsf{M}\;a\;w & = & \mathsf{B}\;(a{+}0)\;(w[7{-}0]) \ast\\ & & \mathsf{B}\;(a{+}1)\;(w[15{-}8]) \ast\\ & & \mathsf{B}\;(a{+}2)\;(w[23{-}16]) \ast\\ & & \mathsf{B}\;(a{+}3)\;(w[31{-}24])\end{array}$$

Our machine-code Hoare triple represents code as set of code fragments: each element is a tuple (a, x, p) where a is a 32-bit address, x is 8-bits of an instruction and p is write permission w iff location a has write permissions set. The code assertion makes sure that code set c is stored in memory and accurately represented in the instruction cache, hence T below.

$$\begin{array}{l}(\mathsf{code}\;c)\;s \;=\\ \quad (s = \{\;\mathsf{Mem}\;a\;(\mathsf{some}\;(x,\{\mathsf{r},\mathsf{x},p\}))\;\mathsf{T} \mid (a,x,p) \in c\;\})\end{array}$$

The data assertion data m, which informally states that $m(a)$ is stored in memory at location a if $a \in$ domain m, is defined as using auxiliary function aux:

$$\begin{array}{l}\mathsf{aux}(m,p,acc) \;=\\ \;\{\;\mathsf{Mem}\;a\;(\mathsf{some}\;(m(a),\{\mathsf{r},\mathsf{w},p\}))\;(acc\;a) \mid a \in \mathsf{domain}\;m\;\}\end{array}$$

Now let data m and datax m, respectively, assert that m is non-executable or executable data, regardless of which addresses are accurately represented acc in the instruction cache:

$$\begin{array}{rcl}(\mathsf{data}\;m)\;s & = & \exists acc.\;s = \mathsf{aux}(m,\mathsf{r},acc)\\ (\mathsf{datax}\;m)\;s & = & \exists acc.\;s = \mathsf{aux}(m,\mathsf{x},acc)\end{array}$$

When turning datax m into a code assertion code $\lfloor m \rfloor$, the notation "$\lfloor m \rfloor$" stands for:

$$\{\;(a, m(a), \mathsf{w}) \mid a \in \mathsf{domain}\;m\;\}$$

7.3 Cache abstraction: $\preceq$

Separation logic, which we use as an inspiration for our machine-code Hoare logic, is based on the notion of local reasoning: each action can be described by a Hoare triple that only describes small local updates, such Hoare triples can separately be brought into a grander context using, what is known as, the frame rule, which will be described later.

Applying ideas from separation logic naively to our x86 model does not work as the updates to the instruction cache are non-local and can occur at random; a basic set up that exposes instruction-cache updates would struggle to support the vital frame rule.

In order to regain local-reasoning, the definition of our Hoare triple will assert pre/postconditions p, not just as,

$$p\;(\mathsf{x86set(s)})$$

but instead using an instruction cache abstraction $p \preceq s$ which allows p to be true for some state t with a less accurate instruction cache but otherwise equivalent to s:

$$p \preceq s \;=\; \exists t.\; t \xrightarrow{\mathsf{icache}} s \;\wedge\; p\;(\mathsf{x86set}\;t)$$

Since all non-local cache updates only introduce new accuracies, a different state t can always be chosen in such a way that old inaccuracies (that p might potentially depend on) can be reintroduced before p is asserted.

7.4 Definition of Hoare triple: $\{p\}\;c\;\{q\}$

The previous subsections defined the necessary building blocks for our instruction-cache-aware Hoare triple, namely: $\preceq$, code and x86_seq. We first define $p \rightsquigarrow q$ to be a total-correctness Hoare "triple" without any code: if some part of the x86 state satisfies p then every possible execution sequence will reach a state which satisfies q. Here '$\ast\;r$' ensures that every resource that is modified is mentioned in precondition p.

$$\begin{array}{rcl}p \rightsquigarrow q & = & \forall s\;r.\;(p \ast r) \preceq s \Rightarrow\\ & & \quad \forall x.\;\mathsf{x86\_seq}\;s\;x \Rightarrow \exists n.\;(q \ast r) \preceq x(n)\end{array}$$

Our machine-code Hoare triple $\{p\}\;c\;\{q\}$ is an abbreviation which maintains code c as an invariant separate from pre- and postcondition p and q, respectively:

$$\{p\}\,c\,\{q\} \;=\; (p \ast \mathsf{code}\;c) \rightsquigarrow (q \ast \mathsf{code}\;c)$$

The relationship between the two judgements can be seen in the following theorems. Here $\emptyset$ is the empty set.

$$\begin{array}{rcl}\{p\}\,\emptyset\,\{q\} & = & p \rightsquigarrow q\\ \{p\}\,c\,\{q\} & = & \{p \ast \mathsf{code}\;c\}\,\emptyset\,\{q \ast \mathsf{code}\;c\}\end{array}$$

These arise from the fact that: $p \ast \mathsf{code}\;\emptyset = p \ast \mathsf{emp} = p$.

7.5 Proof rules

The machine-code Hoare triple, defined above, supports a few unusual proof rules, i.e. theorems proved from the definition of our machine-code Hoare triple $\{p\}\;c\;\{q\}$ and x86 semantics. Section 2.2 already presented the following proof rules for transforming data into code and vice versa:

$$\{p\}\;c\;\{q\} \;=\; \{p \ast \mathsf{code}\;c\}\;\emptyset\;\{q \ast \mathsf{code}\;c\}$$

$$\{p\}\;c\;\{q \ast \mathsf{code}\;\lfloor d \rfloor\} \;\Longrightarrow\; \{p\}\;c\;\{q \ast \mathsf{datax}\;d\}$$

$$\{p \ast \mathsf{datax}\;d\}\;c\;\{q\} \;\Longrightarrow\; \{p \ast \mathsf{code}\;\lfloor d \rfloor\}\;c\;\{q\}$$

Section 2.2 also mentioned that jumps to code pointers, i.e. instructions of the form `jmp r32`, can turn data into code, e.g. `jmp eax`, encoded as `FFE0`, turns data into code:

$$\begin{array}{l}\{\mathsf{eax}\;v \ast \mathsf{pc}\;p \ast \mathsf{datax}\;d\}\\ \;p : \mathtt{FFE0}\\ \{\mathsf{eax}\;v \ast \mathsf{pc}\;v \ast \mathsf{code}\;\lfloor d \rfloor\}\end{array}$$

Other code related rules include the rule for code extension:

$$\{p\}\;c\;\{q\} \;\Longrightarrow\; \forall e.\;\{p\}\;(c \cup e)\;\{q\}$$

which illustrates well that these Hoare triples state only that code c is sufficient to transform states satisfying p into states satisfying q. Thus any extension e to set c will also be sufficient for transforming states satisfying p into states satisfying q. The last code related rule is one which introduces $\lfloor m \rfloor$:

$$\begin{array}{l}\{p\}\ c\ \{q\} \implies \\ ((\forall a\ w.\ (a,w) \in c \implies m(a) = w) \implies \{p\}\ \lfloor m \rfloor\ \{q\})\end{array}$$

Other more conventional rules are:

$$\{p\}\ c\ \{q\} \implies \forall r.\ \{p * r\}\ c\ \{q * r\}$$

$$\{p\}\ c_1\ \{q\} \land \{q\}\ c_2\ \{r\} \implies \{p\}\ c_1 \cup c_2\ \{r\}$$

$$\{p\}\ c\ \{q\} \land (\forall s.\ q\ s \implies r\ s) \implies \{p\}\ c\ \{r\}$$

$$\{p\}\ c\ \{q\} \land (\forall s.\ r\ s \implies p\ s) \implies \{r\}\ c\ \{q\}$$

$$\{\exists x.\ p\ x\}\ c\ \{q\} = \forall x.\ \{p\ x\}\ c\ \{q\}$$

$$\{p * \langle b \rangle\}\ c\ \{q\} = (b \implies \{p\}\ c\ \{q\})$$

8. Quantitative data

Large parts of the basis for this work, the x86 semantics, machine-code Hoare logic and some proof automation, consist of only minor extensions to proof scripts/automation which were developed in previous work [19, 21, 22, 25]. The following table lists the number of lines of HOL4 code, i.e. proof scripts and automation, that were reused (column old) and that are new for this work (column new).

	old	new	total
x86 semantics	2319	150	2469
general Hoare logic	425	7	432
x86 instantiation of Hoare logic	587	430	1017
proof automation*	4851	21	4872
input language for JIT compiler	0	99	99
JIT compiler version 1	0	1231	1231
JIT compiler version 2	0	2550	2550
total	8182	4488	12670

* includes a proof-producing compiler [22] which maps functions in the logic of HOL4 to ARM, x86 and PowerPC machine code, used in Section 4.4. This compiler is based on a proof-producing decompiler [21] which is also included in the number 4851.

The verified JIT compilers can be run on real x86 hardware. A few tests, in Section 4.6, suggest that the JIT compiler is reasonably efficient: the JIT compiler evaluates the bytecode program for calculating the greatest common divisor (GCD), from Section 3, for the pair 2 and 200,000,000 in 0.13 seconds — a time which matches the execution time of compiled C code for calculating GCD. The C code to which we compare, also listed in Section 4.6, was compiled using the GNU C compiler gcc with optimisation turned on.

9. Related work

This paper has touched on a number of topics: JIT compilation, self-modifying code, verification of machine-code programs and compiler verification. Related work on each topic is briefly discussed below.

JIT compilation. The history of JIT compilation has been surveyed by Aycock [3] who traced back the origins of JIT compilation to McCarthy's LISP paper in 1960 [15]. But the term *just-in-time* (JIT) was only introduced in connection with attempts to speed up Java, which was slow before using JIT technology:

> " Java isn't just slow, it's *really* slow, *surprisingly* slow. "
> — from Tyma [28] and Aycock [3]

Discussions such as those of Tyma [28] and Aycock [3] make it clear that JIT compilation is vital for implementation of efficient interpreters for modern languages: Java, C#, ML etc.

Self-modifying code. There is very little published work [8, 6] on verification of self-modifying code. To the best of our knowledge, Gerth [8] was the first to propose a solution. His solution is to not have code but instead only data. He demonstrates that linear temporal logic can be used to prove functional correctness and termination of self-modifying code in a toy assembly language. Gerth presents only a few small examples, questions whether his approach will scale, and does not include an instruction cache in his model of the assembly language.

More recent work by Cai et al. [6] is somewhat more closely related to our work: they use a Hoare logic and target larger case studies in more realistic machine languages. Cai et al. developed an extension, called GCAP, to an assembly-level Hoare logic called CAP [29], and formalised GCAP in the Coq theorem prover. They present a long list of small verification examples for MIPS and 16-bit x86 code, some of which generate code dynamically, but none of which is a full JIT compiler.

One of the main differences to this paper is their approach for mutable code: they introduce the concept of possibly overlapping code blocks that might or might not be present at any given time, but one of which must be present in memory when starting execution of a block at that address. In comparison, our approach seems much simpler since for us code is just safe-to-execute data:

$$\{p\}\ c\ \{q\} = \{p * \mathsf{code}\ c\}\ \emptyset\ \{q * \mathsf{code}\ c\}$$

Another contrast to the work by Cai et al. is that our work builds on a semantics of x86 which takes into account hazards of an out-of-date instruction cache, while their work ignores the instruction cache. Also our framework provides total-correctness results, while GCAP produces partial-correctness results.

Verified machine code. Construction of the two versions of our JIT compiler required verified (functional correctness) implementations of code generation, i.e. x86 code that reads the input bytecode and produces equivalent x86 code.

The obvious way of creating this verified code would have been to, first, either handcraft or generate the machine code, then apply post hoc verification to the proposed machine code. Such an approach could have used:

- *symbolic simulation* which the ACL2 community has emphasised and successfully applied [4, 17], e.g. Boyer and Yu [5] verified of machine code for the Motorola MC68020;
- a *programming logic directly*, e.g. Cai et al. [6] chose this method, which works for small examples, but is very labour-intensive to apply to larger examples [16];
- *verification condition generation* (VCG) applied to machine code: Matthews et al. [14] presented a light-weight approach to trustworthy VCG for machine code – an approach that Hardin et al. have applied to AAMP7G machine code [10]; or
- *decompilation into logic* — a new method proposed by Myreen et al. [21] which maps, via proof, machine code into equivalent functions in logic (details and examples in Myreen [19]).

Instead of using any of the above methods for post hoc verification of handcrafted code, we chose to synthesis appropriate machine code using a proof-producing compiler [22]. Our compiler maps functions from the logic of HOL4 down to functionally equivalent x86 machine code and completely automatically proves a certificate theorem. In order to get the desired verified machine code we wrote functions with the appropriate behaviour, proved them correct, and then related the verification result to the automatically generated machine code using the certificate theorem produced by our proof-producing compiler [22].

Influential work on proof-carrying code (PCC) by Necula [23], typed-assembly language (TAL) by Morrisett et al. [18], and foundational PCC (FPCC) by Appel [2, 27] share the goal of creating trustworthy software, but are not directly applicable, since work in this direction has aimed to automatically ensure safety properties. In contrast, this paper targets much stronger properties of full functional correctness and termination.

Compiler verification. This paper touched on the topic of compiler verification: the code generator inside the JIT compilers was proved to correctly translate (without any optimisations) bytecode into x86 machine code. From the point of view of compiler verification, this is a very simple and slightly unusual implementation to prove correct. Most papers on compiler verification, of which Dave has made a survey [7], concentrate on proving a few optimising transformations correct. An impressive exception to this trend is Leroy's recent proof a full end-to-end implementation of an optimising C compiler [13].

Another contrast to previous work on compiler verification, is that our simple code generator is implemented in x86 machine code (numbers) and produce concrete x86 code as output (numbers); while majority of other work implement the compiler in small toy languages or functional languages and output either assembly code or code in an intermediate language used only inside compilers.

10. Future work

The input language of our verified JIT compiler is a simple stack-based bytecode. In the future, we plan to extend this input language to a bytecode language suitable for reimplementation of our verified x86, ARM and PowerPC implementations of a LISP interpreter [20], i.e. we aim to extend the language to operate over a garbage collected stack of s-expressions. Such an extension ought to be largely orthogonal to the developments here as that extension should boil down to replacing the stack assertion in jit_inv with a more complicated assertion sexp_stack, while code generation is, in principle, unchanged.

Acknowledgments

I would like to thank John Matthews for suggesting that I try to tackle the problem of verifying a JIT compiler. I am grateful for comments from Mike Gordon. This work was funded by the EPSRC, UK.

A. Instruction fetch and decode

Modelling fetch and decode for x86 is made interesting due to the complex instruction encodings that result in variable length instructions. Modelling x86 instruction fetch and decode was part of previous work [25] but did not get a thorough explanation there, so we present fetch-and-decode here, as the current author was responsible for modelling of fetch-and-decode for [25].

Fetch. First, how do we cleanly separate fetching from decoding? The problem is that fetching needs to know how many bytes to fetch, but the number of bytes (the length of the x86 instruction) is only known after decoding.

A close inspection of the x86 manual [12] shows that no 32-bit mode x86 instruction is longer than 20 bytes. Our solution is to first fetch 20 bytes using fetch 20 $a\ s$,

$$\begin{array}{lcl} \mathsf{fetch}\ 0\ a\ s & = & [] \\ \mathsf{fetch}\ (n{+}1)\ a\ s & = & \mathsf{read\_instr}\ a\ s :: \mathsf{fetch}\ n\ (a{+}1)\ s \end{array}$$

Then evaluate decode (outlined later) on a list in which errors, i.e. none elements, have been replaced by zero: decode is applied to clean (fetch 20 $a\ s$) where clean is:

$$\begin{array}{lcl} \mathsf{clean}\ [] & = & [] \\ \mathsf{clean}\ (\mathsf{none} :: cs) & = & 0 :: \mathsf{clean}\ cs \\ \mathsf{clean}\ (\mathsf{some}\ x :: cs) & = & x :: \mathsf{clean}\ cs \end{array}$$

A successful application of decode returns some (i, len), where i is the abstract syntax tree representation of the instruction and len is the length (number of bytes) of the encoding. Before returning the result, we check that the first len bytes are not none, using not_none len:

$$\begin{array}{lcl} \mathsf{not\_none}\ 0\ cs & = & \mathsf{T} \\ \mathsf{not\_none}\ (n{+}1)\ (\mathsf{none} :: cs) & = & \mathsf{F} \\ \mathsf{not\_none}\ (n{+}1)\ (\mathsf{some}\ x :: cs) & = & \mathsf{not\_none}\ n\ cs \end{array}$$

The definition of fetch_and_decode is hence:

```
fetch_and_decode $(r, e, s, m, i)$ =
  let $cs$ = fetch 20 $e$ $(r, e, s, m, i)$ in
   case decode (clean $cs$) of
    none → none
   | some $(i, len)$ → if not_none $len$ $cs$ then
                        some $(i, len)$ else none
```

Decode. The Intel Manual [12] defines instruction encodings using lists of the following style: each line has two parts, the first part (left of the bar) provides the encoding formats, the second part (right of the bar) provides the assembly instruction. Most assembly instructions have multiple different encodings. Here are some of the encodings for the move instruction `mov`:

```
" 89 /r      | MOV r/m32, r32    "
" 8B /r      | MOV r32, r/m32    "
" B8+rd id   | MOV r32, imm32    "
" C7 /0 id   | MOV r/m32, imm32  "
```

The Intel manual provides a description of what each encoding symbol `89`, `B8+rd`, `/r`, etc. means.

In order to minimise errors that can occur when trying to write the equivalent definitions in a theorem prover, we decided to copy across these lines from the Intel Manual into HOL4 and then write, in the HOL4 logic, an interpreter for this syntax based on Intel's description of what it means. The interpreter first splits each string into two at the bar character `|` and then breaks each side up into a list of tokens, e.g. the first line from above becomes the pair:

`["89","/r"]` and `["MOV","r/m32","r32"]`

When the interpreter receives a concrete input to decode, e.g. a list of bytes [0x89, 0x07, 0x86, ...], it will execute a match function which attempts to fit the encoding format, in this case `["89","/r"]`, onto the concrete byte list. If a match is found then a separate function constructs the data-type used for representing the assembly instruction in our operational semantics. The interpreter also returns the unused tail of the input. The example above produces:

some (Xmov (Xrm_r (Xm none (some EDI) 0) EAX),
  [0x86, ...])

This output from the interpreter means that decoding found x86 instruction `mov [edi], eax` and that input [0x86, ...] was left unused, and hence the instruction consisted of the first two bytes of the input. The top-level decode function returns the generated data-type together with the number of bytes consumed from the input list.

Decoder speed-up. Running the decoder inside the theorem prover logic, e.g. evaluating decode [0x89, 0x07, 0x86, . . .], to prove decoding results about specific machine instruction is very slow if done naively. It takes nearly 15 minutes for HOL4 to evaluate the above example without helping lemmas. (Evaluating decode outside the logic is not acceptable as we want a theorem certified by the logical core of HOL4.) However, a significant speed up can easily be achieved by partially evaluating the decode function for the list of x86 instruction encodings, i.e. the list of strings mentioned above. When the theorem produced by partial evaluation is supplied to the standard HOL4's evaluation engine, evaluation is performed inside the logic in less than 2 seconds for most instruction, some take up to a bearable 6 seconds.

References

[1] HOL4 proof scripts, verified x86 code and other supporting material: `http://www.cl.cam.ac.uk/~mom22/jit/`.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science (LICS)*. IEEE, 2001.

[3] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35:97–113, 2003.

[4] R. S. Boyer and J S. Moore. Proving theorems about pure LISP fucntions. *JACM*, 22(1):129–144, 1975.

[5] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.

[6] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Programming Language Design and Implementation (PLDI)*, pages 66–77. ACM, 2007.

[7] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

[8] R. Gerth. Formal verification of self modifying code. In *Int. Conf. for Young Computer Scientists*, pages 305–313. International Academic Publishers, China, 1991.

[9] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer, 1989.

[10] David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for highly secure applications. In Panagiotis Manolios and Matthew Wilding, editors, *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2006.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[12] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual*. Intel Corporation, March 2009.

[13] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*, pages 42–54. ACM Press, 2006.

[14] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *Logic Programming and Automated Reasoning (LPAR)*, volume 4246 of *LNCS*, pages 362–376. Springer, 2006.

[15] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 1960.

[16] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 468–479. ACM, 2007.

[17] J Strother Moore. Symbolic simulation: An ACL2 approach. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 334–350, 1998.

[18] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Principles of Programming Languages (POPL)*, pages 85–97. ACM Press, 1998.

[19] Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.

[20] Magnus O. Myreen and Michael J.C. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS. Springer, 2009.

[21] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In Alessandro Cimatti and Robert B. Jones, editors, *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2008.

[22] Magnus O. Myreen, Konrad Slind, and Michael J.C. Gordon. Extensible proof-producing compilation. In Michael I. Schwartzbach Oege de Moor, editor, *Compiler Construction (CC)*, LNCS. Springer, 2009.

[23] George C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*, pages 106–119. ACM, 1997.

[24] John Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of Logic in Computer Science (LICS)*. IEEE Computer Society, 2002.

[25] Susmit Sarkar, Pater Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Principles of Programming Languages (POPL)*. ACM, 2009.

[26] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 28–32. Springer, 2008.

[27] Gang Tan and Andrew W. Appel. A compositional logic for control flow. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of Verification, Model Checking and Abstract Interpretation (VMCAI)*, LNCS. Springer, 2006.

[28] Paul Tyma. Why are we using Java again? *Commun. ACM*, 41(6):38–42, 1998.

[29] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, 2004.