Appears in Proccedings of 2005 Congress on Evolutionary Computation

Fast Evolution of Custom Machine Representations

Lorenz Huelsbergen
Bell Labs, Lucent Technologies
Murray Hill, NJ 07974
`lorenz@research.bell-labs.com`

Abstract- Described are new approaches for evaluating computer program representations for use in automated search methodologies such as the evolutionary design of software. Previously, program representations have been either evaluated directly on raw hardware, providing high speed but little control and flexibility; or, programs were interpreted by a software interpreter which can incorporate much flexibility into a program's evaluation, but does so at a large cost in time due to interpretation overheads.

Here we bridge this gap by providing intermediate *compilation techniques for machine representations* **that approach the speed of running raw bits directly on hardware, but that have all the flexibility and control of custom instruction sets. In particular, we describe two compilation techniques: the first uses** *just-in-time compilation* **to convert a custom instruction sequence to machine code; the second compiles an instruction set specification into a** *specialized interpreter* **which incurs only small overheads for instruction decoding. We show that both techniques can provide manyfold speedups over direct interpretation while retaining the expressiveness of custom representations.**

1 Introduction

The automatic search for interesting devices—be they physical, computational, or otherwise—requires the repeated evaluation of candidate individuals to ascertain their fitness. Often this evaluation is performed by a computer program simulating the target device's environment. Computer simulation of the environment is, however, a tradeoff. It allows modeling of environments expensive to build or difficult to control in real time, but typically incurs very large temporal simulation overheads. Since fitness evaluation is central to evolutionary search and other search methodologies, it is advantageous to remove as much of simulation's overheads as possible, while preserving most of its advantages.

Specifically, if computer programs are being evolved, then a computer may be used directly for evaluating solutions. Alternately, a computer may be used indirectly to simulate another computer, perhaps through multiple levels of intermediate interpretation. When search is used to find computer programs—as in the evolutionary computation areas of genetic programming and machine-language induction—both ends of this simulation spectrum have been used by its practitioners [3, 4, 2, 11, 10, 6]. The evaluation directly of the bits specifying the program on a hardware processor is on one end of this spectrum [11, 10]. Its primary advantage is its extremely fast execution speed. On the other end of this spectrum is the pure interpretation of the bit string as instructions by a software interpreter [6, 2]. Interpretation gives great flexibility to the design of a custom instruction set architecture (ISA) and to the subsequent evaluation of programs written in it. For example, restricting evaluation to a maximum fixed number of instructions—a central issue in the presence of loops—is easy with an interpreter but tricky when raw bits are run natively on a general purpose machine.

Speed, flexibility, and control are all desirable in the evaluation of a program representation since they contribute to the overall efficacy of the evolutionary paradigm: fast evolution of bits on a microprocessor helps find solutions quickly, custom ISA's allow targeting of a particular region—and often a smaller region—of the solution space, and custom instruction sets can avoid finding programs that might disrupt the search environment, ones that might reset the machine for example. This paper shows that the middle part of the "raw to interpreted" spectrum contains the best of both worlds. Custom ISA's—as previously used for many program evolution projects in many frameworks (*e.g.*, [3, 4, 2, 6])—can achieve speeds much closer to raw hardware processing of the instruction bits than pure interpretation has allowed, all the while completely retaining the safety, control and flexibility of custom instructions.

This paper contributes two techniques for speeding the evaluation of custom machine-language ISA's: *just-in-time compilation* (JIT or JITC) of freshly created individuals and *specialized interpreter generation* (SIG). We believe that these approaches are also valuable in domains other than machine-language induction. For example, one can envisage JITC or SIG being used to speed evaluation of the Lisp expressions typically used in Koza-style genetic programming [9] and being applied to other types of program representations as well. Note however that JITC in particular is most useful for evolvable representations that contain loop constructs since some runtime cost is incurred when new individuals are created and this cost must be offset by frequent use of the compiled instructions.

JITC does an on-the-fly translation of a program's custom instructions to the underlying hardware's machine-language. JITC can be fast; only a single translation pass over the program is required.[1] Typically, a custom instruc-

[1]Forward branches to points in the program not yet scanned require an additional small amount of work to rewrite the forward address once it

```
while (pc < prog_sz && n_steps--) {
  int instr = prog[pc];
  int op = GET_OPCODE(instr);
  int arity = arity_map[op];
  int kind = kind_map[op];
  int reg_ops[MAX_ARITY];

  pc++;
  if (kind == BRANCH_KIND) {
    int offset = GET_OFFSET_SZ(instr) *
     (GET_OFFSET_DIR(instr) ? 1 : -1);
    if (op != J_OPCODE) {
      int reg = GET_ARG1(instr);
      switch (op) {
        case JZ_OPCODE:
          offset = !regs[reg] ? offset : 0;
          break;
        /* other branch cases here */ } }
    pc = BOUND_PC(pc+offset,local_prog_sz);
  } else {
    while (arity--)
      reg_ops[arity] = GET_ARG_I(instr,arity);
    switch (op) {
      case ADD_OPCODE:
        regs[reg_ops[1]] += regs[reg_ops[0]];
        break;
      /* other instruction cases here */
```

Figure 1: Fragment of a register-machine interpreter written in C and used in program evolution experiments. Overheads due to instruction decoding and program-counter maintenance and termination detection are evident.

tion will translate into a small number (3-5) of machine instructions, which is much less than the number of machine instructions required for its interpretation. Bookkeeping code for terminating execution in the presence of long or infinite loops, for example, adds significantly to the complexities of interpretation. JIT compilation produces machine code to perform these tasks quickly as well.

To illustrate the benefits of JITC and SIG, Figure 1 contains a small portion of an interpreter written in C used for fully interpreted machine-language induction experiments [6]. Only the code for a couple of instructions is shown: a branch (`JZ_OPCODE`) and addition (`ADD_OPCODE`). The full interpreter is much larger—a few hundred lines of C. (The operational semantics of the full interpreter are given later in Figure 2.) The point is that there are significant overheads to interpretation due to the task of instruction decoding. In the interpreter code one can see that the instruction kind must be determined as well as the arguments to the instruction (`reg_ops`). Furthermore, the program counter (`pc`) must be maintained. A good C compiler can eliminate blatant overheads in this code, but the essential task of decoding a custom instruction remains, as well as the control of the interpretation.

Specialized interpreter generation (SIG) for a custom ISA is this paper's second technique for speeding machine representations. It is well known in the programming language and compiler communities that for small instruction sets, say for instructions where opcodes and operands are contained in 16 bits, one can trade space for time and generate a (potentially large) custom interpreter that essentially makes every possible instruction and operand combination a special case. The SIG approach is simpler than a JIT compilation system and is also largely machine and OS independent, but does still incur runtime overhead due to its instruction dispatch loop; whereas JITC creates a true program that does not require interpretation. SIG can be substantially faster than pure interpretation since all instruction decoding operations are removed. As shown in Figure 1, SIG removes, for example, the tests for determining the kind of instruction being decoded and for parsing its register operands.

This paper's work on fast evaluation of custom instruction sets is part of the & (AMP–automatic machine programming) project currently being designed and implemented to experiment with modular evolution of programs, widely distributed parallel evolution, and fast evolution techniques.

The paper continues with related work in the next section (§2). Section 3 describes a simple, but complete, interpreter that has been used in evolving machine-language programs, which is used in the subsequent sections describing JITC and SIG. Section 4 describes JIT compilation of the interpreter by giving a full translation of the interpreter's instructions and a description of the runtime data structures required for compilation and for implementing evolutionary operators such as crossover. Another technique for improving upon pure interpretation is by specializing the interpreter for all possible instructions; SIG is described in Section 5. We summarize this work in Section 6 with a comparison of JITC and SIG to the known methods of running raw bits directly on hardware and of interpreting instructions in a straightforward manner.

2 Related Work

The general ideas of just-in-time compilation and interpreter specialization are from the programming languages and compiler literature; [1] provides a thorough background in the machinery necessary for implementing such features. The author is unaware of the use of JITC or SIG in evolving program representations. Therefore, this paper supplies the details of how such techniques can function in an evolutionary setting with operators like crossover, time constraints on evaluations, *etc.*, while providing large latitude in instruction set customization.

Nordin attempted to evolve programs directly on microprocessor instruction sets [11]. The "brittleness" of such representations—changing a single bit can create a program that crashes the machine—led Kühling, Wolff and Nordin [10] to design methods for containing the evolution of machine code by essentially using the operating system to trap exceptions (invalid memory addresses, for example). The substantial OS support their scheme requires curtails portability across systems and OS's, making its implementation inaccessible to most practitioners. We believe JITC and especially SIG to be relatively easy to implement.

becomes known, as explained in Section 4.

A critique of [11, 10] more relevant to evolutionary computation concerns is that the search space of their representation is enormous since it encompasses the ISA of the underlying native machine—in the CISC[2] processors the authors work with, opcodes alone can be 32 bits wide, not including operands. This makes solutions difficult and (and perhaps virtually impossible) to come by since the search space is "polluted" by instructions that cannot contribute to the sought after solution. Furthermore, many instruction codings may now result in an operating system trap—this can increase the cost of executing such an instruction enormously. The JITC and SIG proposals of this paper yield execution almost as fast as that of their system, yet admit small ISA's while providing extreme flexibility to the EC experimenter. Researchers using raw bits on hardware [11, 10] have not demonstrated evolved programs using complex control flow (*i.e.*, loops or recursion) that provably compute their intended functions exactly; we surmise that even their fast evaluation method only in part compensates for the extreme size of the search landscapes implied by a CISC processor's instruction set size.

Early experiments by Friedberg *et al.* [3, 4] attempted to evolve very simple machine-language programs, including branches, using an interpretive approach. Their early experiments were not successful when compared to random search, but they elucidated the MLI ideas being explored today. Cramer [2] describes an experiment that interpreted a representation similar to a machine language, but endowed with high-level iteration operators. Experiments by us (see [6] for reference chain) use interpreted register machines to evolve exact solutions for many functions that require complex control flow. JITC and SIG are intended to push the capabilities of machine-language induction farther.

The JITC and SIG concepts of this paper are applicable too to evolving other program representations of interest. For example, the Lisp-based genetic programming work of Koza [9] and the large body of work based on it, stands to benefit through run-time compilation of Lisp expressions or the creation of custom interpreters for the same.

3 Virtual Register-Machine Interpreter

This section provides a sample custom ISA that has been used for evolving machine-language programs requiring evolution of loop structures (*e.g.*, [6]); it is used in the subsequent sections. The reader may wish to skip ahead to sections 4 and 5 and refer to this section's description only as necessary. The important thing to note is that the VRM consists of external state in the form of a set of virtual registers, internal state in the form of a program counter, and instructions.

The operational semantics of the VRM are given in Figure 2. It includes instructions similar to those of contemporary processors that perform arithmetic, move data, and twiddle bits. Branch instructions allow synthesis of arbitrary control flow. This unrestricted control flow is important because it enables solution of non-trivial problems by synthesizing complex control structures such as loops and recursion.

[2]Complex instruction set computer.

3.1 External State: Registers

We define the *register state* as a vector

$$\vec{R} \equiv \langle R_0, \ldots, R_{m-1} \rangle$$

of m signed integers. The register state constitutes the machine's mutable memory. The precision of a register is inherited from the underlying implementation.[3] Many program instructions (*e.g.*, `ADD`) modify registers directly.

3.2 Internal State: PC

In addition to the external register state, VRM maintains a piece of internal state: a program counter (*PC*). The *PC* is an integer, $0 \le PC < n$, that selects which instruction to fetch and execute. Branch instructions modify the *PC* by adding a signed offset to it; all other instructions always increment the *PC* by one. The *PC* is initially zero. In Figure 2, the function

$$Bnd\,(\mathit{off}, PC, n) = \min\left(\max\left(0, PC + \mathit{off}\right), n\right)$$

computes the new *PC*.

3.3 Instruction Set

A *program* is a vector of n instructions

$$\vec{I} \equiv \langle I_0, \ldots, I_{n-1} \rangle$$

The program counter corresponds to an index of $\vec{I}$. A program *terminates* when $PC = n$, that is, when evaluation steps past the end of the program. Note that an interpreter must explicitly maintain the *PC*. One advantage of this is that it is easy to limit the maximum number of instruction evaluated; the disadvantage is that it must expend many cycles for its maintenance.

The VRM's ISA consists of a register move instruction `MOV`, an unconditional branch `J`, branches conditional on a register's value relative to the zero value (`JZ`, `JNZ`, `JLZ`, `JGZ`), instructions that initialize registers (`SET` and `CLR`), instructions to increment (`INC`) and decrement (`DEC`) a given register, and a nullary instruction `NOP` which does nothing. The arithmetic instructions (`ADD`, `SUB`, `MUL`, `DIV`, `MOD`) perform the respective two's complement operation on source and destination, leaving the result in the destination register. The arithmetic `NEG` instruction negates the value in its argument register. The arithmetic instructions mimic C's behavior and "wrap around" on the exceptional conditions of integer overflow (or underflow) instead of trapping. Arithmetic operations that can generate traps in C are `DIV` and `MOD` which are susceptible to divide-by-zero. Our VRM evaluator checks for zero divisors, a condition which we have (arbitrarily) defined to place zero in the destination register. Note that we must choose how JITC and SIG will handle such exceptions.

Branches (`J`, `JZ`, `JNZ`, `JLZ`, `JGZ`) are always relative to the program counter. Negative offsets describe a backward branch. Note that the operational semantics rewrites a branch to an address < 0 as a branch to I_0 (*i.e.* $PC \leftarrow 0$)

[3]A typical implementation inherits 32-bit signed integers via C's `int` type on a 32-bit machine.

$$\texttt{NOP} \equiv \left(PC \leftarrow PC + 1 \right.$$

$$\texttt{MOV}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow Rsrc \end{array} \right.$$

$$\texttt{SET}(Ra) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ Ra \leftarrow 1 \end{array} \right.$$

$$\texttt{CLR}(Ra) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ Ra \leftarrow 0 \end{array} \right.$$

$$\texttt{INC}(Ra) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ Ra \leftarrow Ra + 1 \end{array} \right.$$

$$\texttt{DEC}(Ra) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ Ra \leftarrow Ra - 1 \end{array} \right.$$

$$\texttt{NEG}(Ra) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ Ra \leftarrow 0 - Ra \end{array} \right.$$

$$\texttt{ADD}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} + Rsrc \end{array} \right.$$

$$\texttt{SUB}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} - Rsrc \end{array} \right.$$

$$\texttt{J}(off) \equiv \left(PC \leftarrow Bnd(off, PC, n) \right.$$

$$\texttt{JZ}(Ra, off) \equiv \left(PC \leftarrow \begin{cases} Bnd(off, PC, n) & \text{if } Ra = 0 \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

$$\texttt{JLZ}(Ra, off) \equiv \left(PC \leftarrow \begin{cases} Bnd(off, PC, n) & \text{if } Ra < 0 \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

$$\texttt{MUL}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} \cdot Rsrc \end{array} \right.$$

$$\texttt{DIV}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ \langle R_{dst} \leftarrow R_{dst} \,/\, Rsrc \rangle \end{array} \right.$$

$$\texttt{MOD}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ \langle R_{dst} \leftarrow R_{dst} \,\%\, Rsrc \rangle \end{array} \right.$$

$$\texttt{AND}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} \,\&\, Rsrc \end{array} \right.$$

$$\texttt{OR}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} \mid Rsrc \end{array} \right.$$

$$\texttt{XOR}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} \,\hat{}\, Rsrc \end{array} \right.$$

$$\texttt{NOT}(Ra) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ Ra \leftarrow \sim Ra \end{array} \right.$$

$$\texttt{SHL}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ \langle R_{dst} \leftarrow R_{dst} << Rsrc \rangle \end{array} \right.$$

$$\texttt{SHR}(R_{dst}, Rsrc) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ \langle R_{dst} \leftarrow R_{dst} >> Rsrc \rangle \end{array} \right.$$

$$\texttt{JNZ}(Ra, off) \equiv \left(PC \leftarrow \begin{cases} Bnd(off, PC, n) & \text{if } Ra \neq 0 \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

$$\texttt{JGZ}(Ra, off) \equiv \left(PC \leftarrow \begin{cases} Bnd(off, PC, n) & \text{if } Ra > 0 \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

Figure 2: Operational semantics for the virtual register machine VRM. An arithmetic or logical operator on the right-hand side mostly inherits the C language's semantics of that operator. Expressions enclosed in ⟨brackets⟩ yield zero on exceptional cases (*e.g.*, divide-by-zero, invalid shift amounts).

and a branch past the end of the program ($n - 1$) as termination (*i.e.* $PC \leftarrow n$). A jump instruction I_j can therefore branch to any one of $n + 1$ distinct addresses. The conditional branches are parameterized by a register and the jump displacement. `JZ` branches if the register is zero. `JNZ` branches on any value but zero. `JLZ` branches on a negative, and `JGZ` on a positive, register value.

The six bit-wise logical instructions found in the right-hand column of Figure 2 perform their namesake's operation and are defined in terms of the respective C operators [8]. A departure from this semantics is the interpretation of the shift operators as NOPs if the shift amount is either negative or exceeds the number of bits comprising an implementation register. Since VRM registers are signed, a right shift (`SHR`) of a negative quantity will effectively "reset" the sign bit.

4 Just-in-Time Compilation (JIT)

This section shows how a simple, but fairly complete VRM can be translated by JIT compilation to native machine instructions. Auxiliary machinery necessary to evaluate the resulting translated program is then described. We also give two extensions to the translation for custom representations that:

1. require more registers than available on the native machine (as may well be the case for Intel x86 [7] architectures), and
2. memory operations with load/store instructions

The translation assumes without loss of generality that the underlying machine is a reduced instruction set processor (RISC) and uses in particular the MIPS instruction set architecture [5] as the target native machine. This choice of MIPS as the ISA is in some sense arbitrary; however, the MIPS ISA is simple enough that, with the explanation in this text, its semantics should be clear and substitution of other common ISA's is straightforward.

4.1 Translation

The translation of the VRM of Figure 2 is given in Figure 3. After describing the strategy for maintaining internal VRM state, we proceed to describe the translation itself for a representative sample of instructions.

The result of JIT translation of a VRM program P is a linear sequence of native machine instructions comprising a native machine program P'. Since P' is no longer governed by an interpreter loop, it must manage its own state, both internal (program counter) and external (registers). This is accomplished by mapping this state to the hardware's register set.

The VRM's program counter's function is subsumed by the PC of the native machine. However, we need to retain the flexibility of the interpreted VRM in that it should be possible to select the exact number of translated VRM instructions to be executed. This can be accomplished by

$$\texttt{NOP} \equiv \texttt{termchk\_macro} \equiv \left(\begin{array}{l} \texttt{addi } r_t,\ r_t,\ \texttt{-1} \\ \texttt{bltz } r_t,\ \texttt{l\_term} \end{array} \right.$$

$$\texttt{MOV}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{add } r_{dst},\ r_0,\ r_{src} \end{array} \right.$$

$$\texttt{SET}(R_a) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{addi } r_a,\ r_0,\ \texttt{1} \end{array} \right.$$

$$\texttt{CLR}(R_a) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{add } r_a,\ r_0,\ r_0 \end{array} \right.$$

$$\texttt{INC}(R_a) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{addi } r_a,\ r_a,\ \texttt{1} \end{array} \right.$$

$$\texttt{DEC}(R_a) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{addi } r_a,\ r_a,\ \texttt{-1} \end{array} \right.$$

$$\texttt{NEG}(R_a) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{sub } r_a,\ r_0,\ r_a \end{array} \right.$$

$$\texttt{ADD}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{add } r_{dst},\ r_{dst},\ r_{src} \end{array} \right.$$

$$\texttt{SUB}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{sub } r_{dst},\ r_{dst},\ r_{src} \end{array} \right.$$

$$\texttt{MUL}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{mult } r_{dst},\ r_{src} \\ \texttt{mflo } r_{dst} \end{array} \right.$$

$$\texttt{DIV}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{div } r_{dst},\ r_{src} \\ \texttt{mflo } r_{dst} \end{array} \right.$$

$$\texttt{MOD}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{div } r_{dst},\ r_{src} \\ \texttt{mflo } r_{dst} \end{array} \right.$$

$$\texttt{AND}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{and } r_{dst},\ r_{dst},\ r_{src} \end{array} \right.$$

$$\texttt{OR}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{or } r_{dst},\ r_{dst},\ r_{src} \end{array} \right.$$

$$\texttt{XOR}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{xor } r_{dst},\ r_{dst},\ r_{src} \end{array} \right.$$

$$\texttt{NOT}(R_a) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{addi } r_{tmp},\ r_0,\ \texttt{-1} \\ \texttt{xor } r_a,\ r_{tmp},\ r_a \end{array} \right.$$

$$\texttt{SHL}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{sllv } r_{dst},\ r_{dst},\ \texttt{src} \end{array} \right.$$

$$\texttt{SHR}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{slrv } r_{dst},\ r_{dst},\ \texttt{src} \end{array} \right.$$

$$\texttt{J}(\mathit{off}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{beq } r_0,\ r_0,\ \texttt{fix\_offset(offset,this\_instr\_loc)} \end{array} \right.$$

$$\texttt{JZ}(R_a, \mathit{off}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{beq } r_0,\ r_a,\ \texttt{fix\_offset(offset,this\_instr\_loc)} \end{array} \right.$$

$$\texttt{JNZ}(R_a, \mathit{off}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{bneq } r_0,\ r_a,\ \texttt{fix\_offset(offset,this\_instr\_loc)} \end{array} \right.$$

$$\texttt{JLZ}(R_a, \mathit{off}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{bltz } r_a,\ \texttt{fix\_offset(offset,this\_instr\_loc)} \end{array} \right.$$

$$\texttt{JGZ}(R_a, \mathit{off}) \equiv \left(\begin{array}{l} \texttt{termchk\_macro} \\ \texttt{bgtz } r_a,\ \texttt{fix\_offset(offset,this\_instr\_loc)} \end{array} \right.$$

Figure 3: JIT equivalents for the VRM of Figure 2 using MIPS instruction set. The function `fix_offset` computes the absolute address of the branch target at compile time. Note the use of the termination register to limit the number of instructions executed and the use of a temporary register to hold intermediate values. See text for details.

dedicating a machine register, here denoted $\mathtt{r_t}$, to holding the number of VRM instructions remaining to be evaluated. To terminate the program after N VRM instructions, $\mathtt{r_t}$ is loaded with N at the start of execution of P'. JITC inserts before the translation of every VRM instruction a check to see if $\mathtt{r_t}$ has reached zero and, if so, to terminate the program by branching to an absolute label. This label, denoted `l_term` in the figure, is where control should flow after the program terminates. A logical point is at the end of P' since "falling off the end" of P' signifies termination as well. It may however be placed anywhere convenient; if it is not placed at the end of P', an unconditional branch instruction to `l_term` must be placed after the last translated instruction in P'.

The two-instruction macro, `termchk_macro`, defined and used in Figure 3, performs the task of decrementing the termination register and checking whether it has reached zero. Note that the VRM's `NOP` instruction translates into this macro. Also note that all VRM instruction translations begin with `termchk_macro`.

The VRM's external state of registers is mapped into the native hardware's general purpose register (GPR) set. To allow this, it is necessary to save all GPR's to spill memory before executing P' and to restore these registers after `l_term` is reached. Here we assume that the number of VRM registers is less than the number of free GPR's; below, we describe how additional registers may be virtualized using some additional memory.

Note that in the MIPS architecture register $\mathtt{r_0}$ is tied to zero. Therefore, the mapping of VRM registers to native GPR's starts with register $\mathtt{r_1}$. In addition to $\mathtt{r_t}$, another GPR here denoted $\mathtt{r_{tmp}}$ is reserved for use as temporary storage. Depending on the representation being translated, more (or perhaps fewer) dedicated registers may be required.

The actual translation of VRM instructions is straightforward. This is due to the VRM being very close to contemporary machine languages. Note, however, that the MIPS instruction set uses the arithmetic instruction for addition `add` for many purposes, including register-to-register transfer. The `MOV` instruction, for example, becomes an addition of the source register to the zero register with the result placed in the destination register. Other instructions such as `SET`, `CLR`, `INC`, `DEC`, and of course `ADD`, can be defined in terms of MIPS' `add` or `addi` instructions. The MIPS `sub` instruction is used similarly for `NEG` and `SUB`.

Note the arithmetic functions of multiplication and division translate to multiple MIPS instructions (see [5] for details). Also, MIPS does not define an exception condition for division by zero or arithmetic overflow. (In the case of divide-by-zero the result is undefined [5].) If it is necessary for the VRM to identify such conditions, additional instructions must be inserted by the translation to, for example, check if the $\mathtt{r_{src}}$ register is zero. If so, a conditional branch instruction can transfer control to `l_term` or elsewhere.

The translation of VRM's branch instructions is also quite direct. Most instructions have a direct MIPS analog, but special treatment is required for the relative branch offset. The correct offset is computed by JITC as it processes a branch instruction by the `fix_offset` function, which takes the VRM offset and the instruction number in the program and computes the target instruction in the resulting translation. Such translation is necessary since in the VRM it was possible to branch past both the start and beginning instruction of the VRM program. Also, since VRM instructions now translate into multiple native (MIPS) instructions and this number is variable from instruction to instruction, `fix_offset` must compute the appropriate native relative offset. Because the address of forward VRM branch instructions are not known during the translation pass, the location of the incomplete native branch offset must be retained and resolved once all VRM instructions have been processed. Auxiliary data structures are necessary for the JIT compiler to resolve such issues.

4.2 Auxiliary Data Structures

The following data structures support JIT translation. First, a block of executable memory B large enough to hold the translated program P' is needed. As we remarked earlier, this block may require prologue code to spill registers that are in use and epilogue code to restore them. Prologue code can also be used to initialize registers and epilogue code can return the program's output from registers.

Second, a vector V of native start addresses for VRM instructions is used in resolving branches (both forward and backward). Shortly we describe how V also aids in implementing operators like crossover. Note that one can dispense with maintaining V entirely by making all VRM instructions translate into the same number of native instructions; one can pad translations shorter than the longest instruction with MIPS `nop`'s. If this is done, the instruction number suffices to determine the start of the instruction's translation in B. This simplifies JIT compilation and evolutionary operator implementation, but significantly slows the evaluation.

Another data structure useful for implementing the evolutionary operators is a type vector J. This vector denotes whether an instruction is a branch or not, and if so, what its relative offset is in the corresponding VRM program. This information is necessary to process branch instructions after relocation by an evolutionary operator (see below) since the compiled offsets computed by `fix_offset` may need to be recomputed when an evolutionary operator moves an instruction, for example.

4.3 Evolutionary Operators

When applied to a program representation, evolutionary operators such as crossover, point-wise mutation, or macro-mutation, shuffle program instructions or replace existing instructions with new ones.

To implement crossover between two parents X and Y to produce offspring Z, for instance, the system allocates a new executable memory block B_Z and copies the desired translated instructions from the parent blocks B_X and B_Y successively into B_Z. The instruction-address vectors V_X

and V_Y defined above aid this copying. As defined in Figure 3, all instructions except for the branch instructions are relocatable; that is, they may be moved to a new address without change. Branch instructions, however, may need to have their offsets recomputed by `fix_offset` since their location within the program may have changed. For example, an instruction i that in the VRM program branched past the end of the program may no longer do so after a crossover operation since i may have been moved to the beginning of the resulting offspring. The type vectors J_X and J_Y are used to find the branch instructions in Z inherited from X and Y and the address vectors V_X and V_Y are used in reapplying `fix_offset`. Note that new address and type vectors V_Z and J_Z are formed for Z during this process.

Mutations and macro-mutations are implemented similarly through copying in general. Since different VRM instructions can translate into differing numbers of native instructions, full copying may be necessary. However, it may be possible to perform a mutation in place if the number of native instructions comprising the VRM instruction being mutated is larger than the number of native instructions comprising the mutation. Again, the vector V can be used to determine this.

4.4 Translating Memory Operands

An important class of instructions absent from the sample VRM (Figure 2) is memory load/store instructions. In the VRM, all memory is contained in the register state. However, experimenters may wish to evolve programs with instructions that have memory operands.

A simple way to translate memory instructions is as follows. A runtime block of memory M is allocated as "the memory" and initialized by the prologue code if necessary. A VRM instruction that then indexes into this memory, say ADD($\mathtt{r}_{\mathtt{dst}}$,M[$\mathtt{r}_{\mathtt{index}}$]), would insert native instructions to test if the value of $\mathtt{r}_{\mathtt{index}}$ is in range; that is, a *bounds check*, $0 \leq \mathtt{r}_{\mathtt{index}} < |M|$, would occur to restrict access to memory outside of the allocated block M. Instructions writing to memory would similarly be bounds checked.

4.5 Simulating Additional Registers

In the above it was assumed that the number of registers specified by the VRM fits into the registers on the processor of the target machine. For some contemporary processors (*e.g.*, Intel's x86 [7]) this may not be the case. Here we describe how to translate a set of virtual registers that cannot be mapped directly into available machine registers. Standard compiler techniques accomplish this (*cf.*, [1]).

A bank of m registers is allocated as a block of m memory words (where a word can hold a single VRM register, typically 32 or 64 bits). Denote this block as `vrb`. When a VRM register is referenced by a VRM instruction, the translation uses native temporary registers to load the virtual register from its location in `vrb`. The specified operation is performed on the temporary registers and the result is written back to the appropriate places in the `vrb` as necessary.

Consider translation of ADD($\mathtt{r}_9$,$\mathtt{r}_{15}$) as an example:

```
 st  spill0, tmpreg0  #free tmp regs
 st  spill1, tmpreg1
 ld  tmpreg0, vrb[15] #get operands
 ld  tmpreg1, vrb[9]
 add tmpreg0, tmpreg0, tmpreg1  #do add
 st  vrb[9], tmpreg0  #update result reg
 ld  tmpreg0, spill0  #restore tmp regs
 ld  tmpreg1, spill1
```

Here, $\mathtt{r}_9$ and $\mathtt{r}_{15}$ reside at offsets 9 and 15 of the `vrb` respectively. If the temporary native registers are in use (or it is not known if they are), they must be saved in a spill area and restored from this area when the addition operation completes. Also note that the result of the addition (in `tmpreg0`) is written back to the virtual destination register at `vrb[9]`.

4.6 Optimization

The translation described in this section can be further improved by using compiler optimization techniques as cataloged in standard compile text such as [1]. Additional translation effort can be used to do *peep hole optimization* (PHO) within a small window of generated native instructions. PHO and other more costly optimizations can have a dramatic effect on the runtime of the translated program. However, the cost of the optimization must also be taken into account—one must be sure to recoup the time spent optimizing through time saved evaluating the resulting program.

5 Specialized Interpreter Generation (SIG)

Another technique to speed interpreter evaluation is by specializing the interpreter for all possible instruction/operand combinations that may occur. Though not as speedy as programs produced by JIT compilation, SIG can remove the instruction decoding overheads—which are not minor and cost many tens of machine instructions—from the interpreter's loop. A primary advantage of SIG is that it is very simple to implement and that it is portable across compilers and OS's.

A prerequisite for using SIG is that the total number of opcode/operand combinations be manageable—by this we mean that this number be small enough that a *jump table* can be constructed in memory for dispatching every such combination. (For counting the number of such combinations in a VRM see, for example, [6].) More precisely, for a given VRM, SIG will produce a multi-way "switch" statement where every "case" (or "label") is one of the possible combinations. This table must be small enough to fit in memory and it should furthermore be possible to compile the resulting switch statement with a C or Java compiler. We have verified that opcodes/operands encoded in 16 bits and hence resulting in 2^{16} cases are readily compiled by conventional C compilers. In practice, 20 or 24 bit opcodes should be possible, but many interesting VRM's can already be defined with 16 bits.

Figure 4 gives a small portion of a specialized interpreter corresponding to the interpreter fragment of Figure 1. Here the VRM is again the one of Figure 2 and it is instantiated to 16 registers. The encoding is into 14 bits as follows. For sin-

```
int r0=0, r1=0, ..., r15=0;
while (pc > 0 && pc < prog_sz && nsteps--) {
  switch(prog[pc++]) {
    case 0x0000:  /* NOP */
      break;
    ...
    case 0x0100:  /* MOV(r0,r0) */
      r0   = r0; break;
    case 0x0101:  /* MOV(r0,r1) */
      r0   = r1; break;
    case 0x0102:  /* MOV(r0,r2) */
      r0   = r2; break;
    ...
    case 0x015e:  /* MOV(r5,r14) */
      r5   = r14; break;
    ...
    case 0x08f7:  /* XOR(r15,r7) */
      r15 ^= r7; break;
    ...
    case 0x3a63:  /* JLZ(r6,-3) */
      if(r6<0)
        pc += -3;
      break;
    ...
```

Figure 4: Fragment from a specialized interpreter for the VRM of Figure 2. Note how every opcode/operand combination maps to a `case` label. Also, registers are stored directly as variables. See text for the instruction encoding.

gle register opcodes, the top bits 8-13 are zero, the opcode is encoded in bits 4-7, and bits 0-3 contain the register. For branch instructions, the top bit (bit 13) is one and bits 8-11 encode the branch opcode; bits 4-7 contain the register (for a conditional branch) and bits 0-3 contain the offset with the offset's sign encoded in bit 12. The remaining two register operand instructions are encoded with bits 12-13 zero, the opcode in bits 8-11, and the register pair in the lower byte.

It is important to note that a good compiler will produce code for the cases very similar to that of the JITC approach due to the fact that VRM registers have been mapped directly to variables and not to arrays as in the slow interpreter of Figure 1. This removes many memory references since indirection through an array address is no longer necessary to fetch/store VRM registers.

Since the encoding of the VRM can be made identical for standard interpretation (Figure 1) and for SIG, branch instructions that required special treatment for JITC can be processed as before. SIG retains the interpreter loop and can easily detect program counters that fall outside the program proper. Relatedly, implementation of the evolutionary operators for SIG is also simple—evolutionary operators can shuffle and modify the program array in an unrestricted manner since the interpreter loop again checks for valid program counter conditions. As with the raw execution of bits on native hardware [11, 10] and unlike JIT compilation, SIG can admit crossover (and other evolutionary operators) at bit boundaries if all possible opcode values are defined.

6 Summary

This paper extends just-in-time compilation and specialized interpreter generation techniques to program representations for use in evolutionary computation experiments. In particular, we show how custom representations may be efficiently translated so that their execution efficiency approaches that of native machine languages. In particular we find that for a typical virtual register machine, JIT compilation introduces an overhead of 2-3 additional instructions per VRM instruction—it therefore can run half to a third as fast as native code, but retains the ability to terminate evaluation after a fixed number of instruction executions and the flexibility to customize fully the instruction set, among other advantages. SIG, on the other hand, requires 12-20 instructions on average for the same VRM, but this is still an improvement over straight interpretation which can spend more than 100 cycles on instruction decode and execution.

	Raw	JITC	SIG	Int
speed	very high	high	moderate	slow
portability	very low	moderate	high	very high
program size	small	moderate	small	small
extensible	no	yes	yes	yes
bit operators	yes	difficult	yes	possible
self-modify	yes	no	yes	possible

Table 1: Comparison of the two techniques of this paper, JITC and SIG, to direct execution of bit strings on native hardware (Raw) and to fully interpreted register machines (Int).

The above table compares the two new and the two existing schemes along six dimensions: speed, portability between OS's, size of the candidate programs, extensibility of the ISA, availability of evolution operators at bit boundaries, and support for self modifying code.

As can be seen from the table entries for just-in-time compilation and specialized interpreter generation, EC experimenters now have at hand two new means for evolving complex software machines while retaining high degrees of expressiveness in their choice of machine representation.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the International Conference on Genetic Algorithms and their Applications*, pages 183–187. Texas Instruments, July 1985.

[3] R. M. Friedberg. A learning machine: Part I. *IBM Journal of Research and Development*, 2:2–13, 1958.

[4] R. M. Friedberg, B. Dunham, and J. H. North. A learning machine: Part II. *IBM Journal of Research and Development*, 3:282–287, 1959.

[5] J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, Inc., second edition, 1994.

[6] L. Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In *Proceedings of the Third Conference on Genetic Programming*, pages 158–166, July 1998.

[7] Intel Corporation. *Pentium processor user's manual*. Intel Corporation, 1993.

[8] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[9] J. Koza. *Genetic Programming: On the Programming of Computers by the Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[10] F. Kühling, K. Wolff, and P. Nordin. Brute-force approach to automatic induction of machine code on CISC architectures. In J. A. Foster *et al.*, editor, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 288–297. Springer-Verlag, April 2002.

[11] P. Nordin. A compiling genetic programming system that directly manipulates the machine-code. In K. Kinnear Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.