The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation

Ian Piumarta

Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie,
4, place Jussieu, 75252 Paris Cedex 05, France

ian.piumarta@inria.fr

Abstract

Tools supporting dynamic code generation tend too be low-level (leaving much work to the client application) or too intimately related with the language/system in which they are used (making them unsuitable for casual reuse). Applications or virtual machines wanting to benefit from runtime code generation are therefore forced to implement much of the compilation chain for themselves even when they make use of the available tools. The VPU is an fast, high-level code generation utility that performs most of the complex tasks related to code generation, including register allocation, and which produces good-quality C ABI-compliant native code. In the simplest cases, adding VPU-based runtime code generation to an application requires just a few lines of additional code—and for a typical virtual machine, VPU-based just-in-time compilation requires only a few lines of code per virtual instruction.

1 Introduction

Dynamic compilation is the transformation of some abstract representation of computation, determined or discovered *during* application execution, into native code implementing that computation. It is an attractive solution for many problem domains in which high-level scripts or highly-compact representations of algorithms must be used. Examples include: protocol instantiation in active networks, packet filtering in firewalls, scriptable adaptors/interfaces in software object busses, policy functions in flexible caches and live evolution of high-availability software systems. In other domains it is an essential technique. In particular, virtual machines that interpret bytecoded languages can achieve good performance by translating bytecodes into native code "on demand" at runtime.

A dynamic code generator is the part of a dynamic compiler that converts the abstract operations of the source representation into executable native code. It represents a large part (if not the bulk) of any dynamic compiler—and certainly the majority of its complexity. However, very few utilities exist to ease the task of creating a dynamic code generator and those that are available are ill-suited to a simple, "plug-and-play" style of use.

1.1 Related work

Several utilities have been developed to help with the implementation of static code generators. Examples include C-- [1], MLRISC [3] and VPO [2] (part of the Zephyr compiler infrastructure). However, very few tools have been developed to help with the implementation of dynamic code generators. Those that do exist are concerned with the lowest (instruction) level of code generation.

`ccg` [9, 12] is a collection of runtime assemblers implemented entirely in C macros and a preprocessor that converts symbolic assembly language (for a particular CPU) embedded in C or C++ programs into macros calls that generate the corresponding binary instructions in memory. It is useful tool for building the final stage of a dynamic code generator, but constitutes only a small part of a dynamic compiler, and deals exclusively in the concrete instructions of a particular architecture.

`vcode` [6] and GNU Lightning [13] are attempts to create virtual assembly languages in which a set of "typical" (but fictitious) instructions are converted into native code for the local target CPU.[1] Both of these systems present clients with a register-based abstract model. Register allocation (one of the most

[1] GNU Lightning is based on `ccg` and is little more than a collection of wrappers around it.

difficult code generation problems to solve) is left entirely to the client, and both suffer from problems when faced with register-starved, stack-based architectures such as Intel.[2]

Beyond these systems code generators rapidly become intimately tied to the source language or system with which they are designed to operate. Commercial Smalltalk and Java compilers, for example, use sets of "template" macros (or some functionally equivalent mechanism) to generate native code, where the macros correspond closely to the semantics of the bytecode set being compiled. Adapting them for use in a different application (or virtual machine) requires significant work. Tasks such as register allocation also tend to be adapted for the specifics of the source language. (One notable exception is the Self compiler [4], for which a fairly language-neutral model of stack frames and registers was developed. Nevertheless, the rest of the Self compiler is so complex that nobody has managed to extract and reuse it in a completely different context.)

1.2 The VPU

The VPU fills the gap between trivial "virtual assembly languages" and full-blown dynamic compilers intimately tied to their source language and its semantics. It is a complete "plug-and-play" dynamic code generator that can be integrated into any application in a matter of minutes, or used as the backend for a dynamically-compiled language or "just-in-time" compiler. It presents the client with a simple, architecture-neutral model of computation and generates high-quality, C-compatible native code with a minimum of time and space overheads. It assumes full responsibility for many of the difficult tasks involved in dynamic code generation, including register allocation and the details of local calling conventions. Applications and language implementations using the VPU are portable (with no source code changes) to all the platforms supported by the VPU; currently PowerPC, Sparc and Pentium.

A useful analogy might be to consider languages that are "compiled" into C source code which is then passed to an existing C compiler for conversion into the final executable. The VPU could be used in a similar fashion: its input "language" is semantically equivalent to C and its output is C-compatible native code. The difference is that the VPU is integrated into the application and performs its compilation at runtime, sufficiently fast that the application should never notice pauses due to dynamic code generation.

[2]GNU Lightning solves the "Pentium problem" by supporting just 6 registers. `vcode` solves the problem by mapping "excess" registers within its model onto memory locations, with all the performance penalties that this implies.

```
#include <stdio.h>
#include <stdlib.h>

typedef int (*pifi)(int);

pifi rpnCompile(char *expr);

int main()
{
  int i;
  pifi c2f= rpnCompile("9*5/32+");
  pifi f2c= rpnCompile("32-5*9/");
  printf("\nC:");
  for (i = 0;  i <= 100;  i += 10)
    printf("%3d ", i);
  printf("\nF:");
  for (i = 0;  i <= 100;  i += 10)
    printf("%3d ", c2f(i));
  printf("\n\nF:");
  for (i = 32;  i <= 212;  i += 10)
    printf("%3d ", i);
  printf("\nC:");
  for (i = 32;  i <= 212;  i += 10)
    printf("%3d ", f2c(i));
  printf("\n");
  return 0;
}
```

Figure 1: Temperature conversion table generator. This program relies on a procedure `rpnCompile()` to create a native code functions converting degrees Farenheit to Celsius and vice-versa.

The rest of this paper is organised as follows: Section 2 describes the feature set and execution model of the VPU from the client's point of view. Section 3 then describes in some detail the implementation of the VPU, from the API through to the generation of native code (and most of the important algorithms in between). Section 4 presents a few performance measurements, and finally Section 5 offers conclusions and perspectives.

2 Plug-and-Play code generation

A simple (but complete) example illustrates the use of the VPU. Figure 1 shows a program that prints temperature conversion tables. It relies on a small runtime compiler, `rpnCompile()`, that converts an input expression (a string containing an integer function in reverse-polish notation) into executable native code. The program first compiles

```
#cpu pentium

pifi rpnCompile(char *expr)
{
  insn *codePtr= (insn *)malloc(1024);
  #[   .org    codePtr
       pushl   %ebp
       movl    %esp, %ebp
       movl    8(%ebp), %eax
  ]#
  while (*expr) {
    char buf[32];
    int n;
    if (sscanf(expr, "%[0-9]%n", buf, &n)) #[
!       expr += n - 1;
       pushl   %eax
       movl    $(atoi(buf)), %eax
    ]#
    else if (*expr == '+') #[
       popl    %ecx
       addl    %ecx, %eax
    ]#
    else if (*expr == '-') #[
       movl    %eax, %ecx
       popl    %eax
       subl    %ecx, %eax
    ]#
    else if (*expr == '*') #[
       popl    %ecx
       imull   %ecx, %eax
    ]#
    else if (*expr == '/') #[
       movl    %eax, %ecx
       popl    %eax
       cltd
       idivl   %ecx, %eax
    ]#
    else
      abort();
    ++expr;
  }
  #[   leave
       ret     ]#;
  return (pifi)codePtr;
}
```

Figure 2: Low-level code generation based on macro expansion. The input string is scanned for literals and arithmetic operators, and Intel native code generated accordingly. The sections delimited by '`#[...]#`' describe the code to be generated (and also "group" their contents like C curly braces). A preprocessor converts these sections into macro calls that produce the binary instructions in memory.

---

two temperature conversion functions `c2f` and `f2c`, and then uses them to produce a conversion table. The implementation of the "dynamic compiler" is encapsulated entirely within the `rpnCompile()` procedure.

Figure 2 shows one possible implementation of the `rpnCompile()` procedure. It uses `ccg` to create Intel

```
#include "VPU.h"

pifi rpnCompile(char *expr)
{
  VPU *vpu= new VPU;
  vpu                           ->Ienter()->Iarg()
                                ->LdArg(0);
  while (*expr) {
    char buf[32];
    int n;
    if (sscanf(expr, "%[0-9]%n", buf, &n)) {
      expr += n - 1;
      vpu                       ->Ld(atoi(buf));
    }
    else if (*expr == '+')  vpu->Add();
    else if (*expr == '-')  vpu->Sub();
    else if (*expr == '*')  vpu->Mul();
    else if (*expr == '/')  vpu->Div();
    else
      abort();
    ++expr;
  }
  vpu                           ->Ret();
  void *entry= vpu              ->compile();
  delete vpu;
  return (pifi)entry;
}
```

Figure 4: VPU-based code generation.

---

native code for a function implementing the input expression. (The program fragments shown in Figures 1 and 2 combine to form a complete program, whose output is shown in Figure 3.)

Figure 4 shows an alternative implementation of `rpnCompile()` that uses the VPU to produce the same native code in a platform-independent manner. The VPU appears to clients as a single C++ class. A client constructs an instance of this class on which it invokes methods corresponding to the operations of an abstract stack machine. Once a complete function has been described the client asks the object to compile it. The result is the address of the native code implementing the function which can be subsequently be called from both statically- and dynamically-compiled code, just like a "normal" 'pointer to function'. When the function is no longer needed the client can return the memory to the heap using the standard C library function `free()`. Figure 5 shows the code generated by the VPU-based `rpnCompile()` procedure when run on the PowerPC.

Note that the only client-side state is the `vpu` instance itself; this would be the case no matter how complex the function being compiled. All instruction arguments (such as the constant argument for the `Ld` instruction) can be computed at runtime.

```
C:  0  10  20  30  40  50  60  70  80  90 100
F: 32  50  68  86 104 122 140 158 176 194 212

F: 32  42  52  62  72  82  92 102 112 122 132 142 152 162 172 182 192 202 212
C:  0   5  11  16  22  27  33  38  44  50  55  61  66  72  77  83  88  94 100
```

Figure 3: Output generated by the temperature conversion program.

```
------------------------- code gen
003b14f0  mr      r4,r3
003b14f4  li      r5,9
003b14f8  mullw   r4,r4,r5
003b14fc  li      r5,5
003b1500  divw    r4,r4,r5
003b1504  addi    r3,r4,32
003b1508  blr
------------------------- 28 bytes
------------------------- code gen
003b1560  mr      r4,r3
003b1564  addi    r4,r4,-32
003b1568  li      r5,5
003b156c  mullw   r4,r4,r5
003b1570  li      r5,9
003b1574  divw    r3,r4,r5
003b1578  blr
------------------------- 28 bytes
```

Figure 5: Generated code for PowerPC.

2.1 The VPU's model of computation

Clients are presented with a stack-based module of computation that includes a complete set of C operations: arithmetic and shift, type coercion, bitwise logical, comparison, memory dereference (load/store), reference (address-of argument, temporary or local label), and function call (both static and computed). Three additional operators are provided for manipulating the emulated stack: `Drop()`, `Dup(`$n = 0$`)` (which duplicates buried values when $n > 0$) and `Put(`n`)` (which stores into a buried location in the stack). These last two operations allow non-LIFO access to data on the stack.

Control flow is provided by unconditional `Br(`***n***`)` and conditional branches, `Bt(`***n***`)` and `Bf(`***n***`)`. The interpretation of 'truth' is the same as in C: zero is 'false', anything else is 'true'. (The VPU treats the condition codes register as an implicit value on the stack that is reified *only* when a logical result is used as an integer quantity. For example, there is no reification of the result of a comparison when the next operation is a conditional branch.)

The argument 'n' in the above branch instructions refers to a local label. Labels are maintained on a parallel stack, with *control scopes* being pushed and popped in a strictly LIFO fashion. Local labels are created and defined via three instructions:

- `Begin(`***n***`)` pushes ***n*** undefined local labels onto the label stack;
- `Define(`***n***`)` associates the current program counter value with the ***n***th entry in the label on the stack; and
- `End(`***n***`)` pops the topmost ***n*** entries off the label stack.

Control is permitted to jump forward out of a local scope, implicitly truncating the emulation stack at the destination label. Conversely, a balanced stack is enforced for backward branches (loops). A local label can remain undefined providing it is never referred to within the function body. Attempting to jump to a label that is never `Define`d will raise an error at (dynamic) compile time.

Formal arguments are declared immediately after the prologue. The type of the argument is explicit in the declaring instruction (`Iarg()` or `Darg()`). Arguments are referred to within the function by position relative to the first (corresponding to the 'leftmost' argument in the equivalent C declaration); 0 is the first actual argument.A single `LdArg(`n`)` is provided to push an actual argument onto the stack; the type of the value pushed onto the stack is implicit and corresponds to the explicit type given in the instruction that declared the argument.

Temporaries are similar to arguments, but can be declared and destroyed at any point within the body of the function. For example, `Itmp` creates a new local `int` variable on the temporary stack. Subsequent `LdTmp(`n`)` and `StTmp(`n`)` instructions read and write temporaries, with $n = 0$ referring to the topmost (most recently declared) temporary. The `DropTmp(`n`)` instruction pops the topmost n temporaries off the stack.

```
int main(int argc, char **argv)
{
  VPU *vpu= new VPU();
  Label fn;
  vpu->Define(fn)->Ienter()->Iarg()
    ->Itmp()
    ->Ld(0)->StTmp(0)->Drop()
    ->Begin(1)->Define(0)
      ->LdTmp(0)->Ld("%d\n")
        ->Icall(2, (void *)printf)->Drop()
      ->LdTmp(0)->Ld(atoi(argv[1]))->Add()
        ->StTmp(0)
        ->LdArg(0)->Le()->Bt(0)
    ->End(1)
    ->DropTmp()
    ->Ld(0)->Ret()
    ->compile();
  fn(atoi(argv[2]));
  free(fn);
  return 0;
}
```

Figure 6: Dynamically constructing a function that uses local variables, labels and conditional branches. The program takes two command-line arguments: a loop 'step' and 'limit'. (The program compiles the first into the dynamic code as a constant and passes the second to it as an argument.) A temporary variable is created and initialised to 0. A local label is then created and defined. The value of the temporary is then printed, it is stepped and compared to the limit ('`Le`' is a 'less-or-equal' comparison); if the limit has not been reached the loop continues. The local label is then popped of the label stack and the local variable destroyed before returning 0. Running this program with the arguments '`3 10`' prints '`0 3 6 9`' on `stdout`. The indentation reflects the depth of the label and emulation stacks within the function body. Note that the `Define` instruction is overloaded to accept integer arguments (local labels on the label stack) and `Label` objects (global labels whose values remain available to the client application after compilation). `Label` is a convenience class provided by the VPU that includes overloaded definitions of `operator()` to simplify the invocation of dynamically-compiled code (eliminating the need to cast its address to a pointer-to-function type).

The `compile()` method requires that both the emulation and temporary stacks be empty after the final `Ret()` instruction in the function.

Figure 6 shows a rather more complete example that uses temporary variables, local labels and conditional branches.

3 Implementation of the VPU

Figure 7 shows the four phases of compilation within the VPU. The function described by the client is converted into an internal abstract representation.

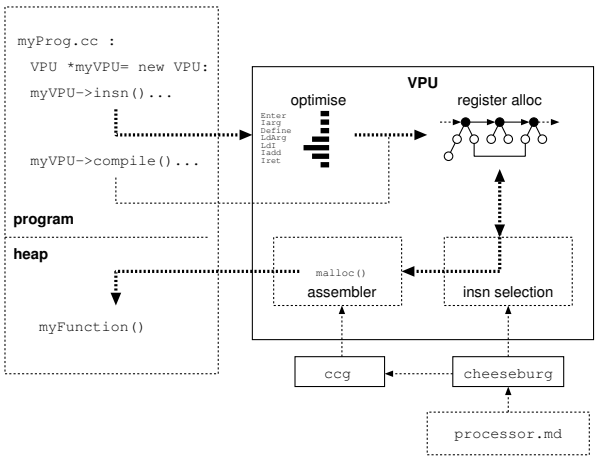

Figure 7: The VPU's architecture. An internal representation of the function and emulation stack is constructed in response to the client invoking methods on a vpu. Compilation involves performing analyses and optimisations on the internal representation, followed by instruction selection and register allocation (each of which can affect the other) to associate concrete instructions, physical registers and runtime stack locations with each abstract instruction. After sizing the final code space is allocated by calling `malloc()` into which a runtime assembler generates executable native code. Target-specific parts of the instruction selection and assembly phases are generated automatically from a processor description file by the program `cheeseburg`, similar in spirit to the `iburg` and `lburg` family of code generator generators.

Various optimisations are performed followed by concrete instruction selection and register allocation. Native code is then generated in memory, with a little help from `ccg`.

3.1 Creation and analysis of abstract code

This phase has the following goals:

- create of an abstract representation of the input function;
- verify the internal consistency of the function;
- perform architecture-neutral optimisations on the representation;
- resolve ambiguities in the stack arising from nonlinear control flow;
- eliminate dead instructions (unreachable or which compute unused values);

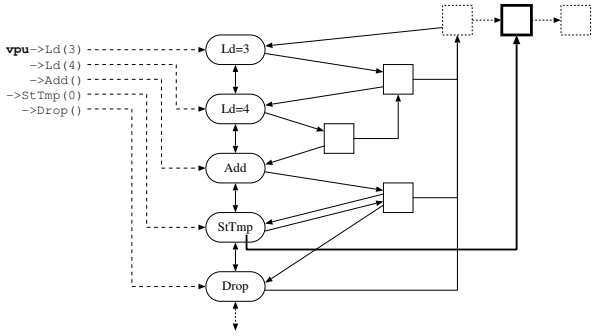

Figure 8: Internal representation of the abstract instructions and their input and output stacks. In this example the stack initially contains just temporaries and arguments. The two `Ld` instructions push new locations to the head of the stack; the tails of their respective output stacks point to the first location of their input stacks. The `Add` instruction consumes two input stack elements and pushes a new location for the result; the tail of its output stack therefore points to the third location of its input stack. The `StTmp` instruction has no stack effect and hence its input and output stacks are identical (and refer to the same location). The `Drop` instruction pops the first element off the stack; its output stack is therefore just the tail of its input stack.

- create optimal conditions for the following register allocation phase.

3.1.1 The abstract representation

A function is represented as a doubly-linked list of abstract instructions. Each instruction contains pointers to its *input stack* and *output stack*. The stack representation is similar to that described in [7] and consists of a linked list of stack *locations*. The "tail" of the stack is shared between successive instructions (rather than recreating a complete list of stack locations for each instruction), as illustrated in Figure 8.

This representation not only reduces memory requirements but also guarantees properties that are critical to the VPU's compilation process:

- the lifetime of any value on the stack is explicit: it begins with the instruction that creates the corresponding location and ends with the instruction that removes the location from its output stack;
- any two elements on the stack that represent the same logical value will also share identity (a single stack location object represents the value for all instructions that might reference it): a single location object represents a given value for the entirety of its lifetime.

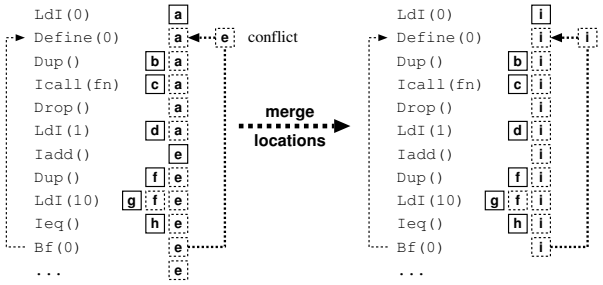

Figure 9: Contradictions in the stack caused by control flow. When two or more flows of control merge the VPU ensures that stack location identity is preserved between the branch and the destination. Whenever conflicts are detected (in this case because a loop iteration variable is kept on the stack) a new location is allocated and replaces the two original conflicting locations.

3.1.2 Control flow analysis

A control flow graph is constructed for the abstract representation. This graph is a set of $\{source \rightarrow destination\}$ tuples formed from the union of explicit control flow and that implied by linear flow into a label (represented by a `Define` instruction):

$$\{branch \rightarrow label\} \cup \{insn_{i-1} \rightarrow label_i\}$$

The graph is used primarily to detect and correct location ambiguities introduced by loops. This situation occurs, for example, when a loop uses a temporary location on top of the stack as an iteration variable. The output stack (at the point of the backwards branch) will not represent the same set of locations as the input stack (at the head of the loop). Figure 9 illustrates the problem. The VPU resolves this situation by replacing the two "colliding" locations with a newly-allocated location, which restores location identity over the lifetime of the loop.

3.1.3 Type analysis

Simple analysis is performed on the program to determine the input types of each instruction. This is necessary, for example, to unambiguously associate a concrete arithmetic instruction with a virtual instruction that has only one form (which might represent either an integer or floating point operation) and also to ensure that instructions having restricted types (shift instructions, memory ref-

erences, etc., which are only defined for integer operands) are being used correctly.

For each instruction the types of the input arguments (if any) are verified to ensure that they are consistent with each other and with the output type of the instruction; if the output type is not known then it is inferred from the input types. The output type is stored in the instruction's output location for use in checking the input types of subsequent instructions.

The type checking phase also determines the "class" (void, integer, float, condition code, etc.) of the instruction's result. This class is stored in the output location for use during instruction selection. The combination of input classes and output class for a given instruction will be referred to below as the *mode* of the instruction.

3.1.4 Optimisations on the abstract representation

The VPU performs relatively few optimisations on the abstract program. The goal is to generate high-quality (but not necessarily optimal) code as quickly as possible. For a particular optimisation to be considered it must satisfy the following conditions:

- The VPU must be able to detect the opportunity for, and then implement, each optimisation *in parallel with* some other *essential* operation that would be necessary even for "unoptimised" compilation. In other words, all optimisations must be "piggy-backed" onto some other, required, traversal or manipulation of the abstract representation.
- Only optimisations that have a significant effect-to-cost ratio are considered.
- Global optimisations are not considered. (Their cost is always significant, requiring additional traversals of the abstract representation and/or the creation and maintenance of additional data structures.)
- Peephole optimisations, that transform particular sequences of instructions into a more efficient sequence, are also not considered. The costs associated with "pattern matching", and the need for an additional pass over the code, are not justified by the relatively small resulting gain in code quality [5].

Optimisations that do meet these criteria are constant folding, jump chain elimination and the removal of dead code. They can be performed (for example) in parallel with control flow or type analysis.

Constant folding is trivial. For a given instruction of arity n, if the topmost n elements of its input stack are never written and are associated with `Ld` instructions then:

- the n `Ld` instructions are deleted from the program;
- the constant result r of the operation is calculated;
- the original operation is transformed into `Ld(`r`)`.

Dead code occurs when a basic block is unreachable or when a side effect-free sequence of instructions computes a result that is never used. Any instruction that has a side effect (such as a store) sets an attribute s on its input and output location. For other operations of arity n, s is propagated from the output location to the n topmost locations on its input stack. During some *subsequent* traversal of the program, if s is unset for the input location of a `Drop` then both the `Drop` instruction and the instruction that generated the dropped location can be deleted from the program.[3] (Since locations are shared between all instructions that refer to them, a `Drop` occurring along one control path will never delete an instruction generating a value that is used along an alternate path through the program.)

Elimination of unreachable code consists of finding the transitive closure of reachable blocks starting from the entry point of the function. Any block not marked as reachable can safely be deleted. The algorithm is trivial and can be performed in parallel with the construction of the flow graph.

Jump chain elimination is performed in parallel with the construction of the control flow graph. The transformations are as follows:

$Br^+ \rightarrow Bx(L) \Rightarrow Bx(L)$ pulls any destination branch forward into a referent unconditional branch; and

[3] This is best done during a backwards traversal of the program, for example while assigning fixed and "constrained" registers to instruction output locations as described below.

$Bx \rightarrow Br^{+}(L) \Rightarrow Bx(L)$ pulls an unconditional branch forward into any referent branch.

These transformations are applied repeatedly for each branch instruction in order to find the destination of the chain.

3.1.5 Summary

At this point in the compilation we have:

- a linear program consisting of abstract instructions;
- an input stack and output stack attached to each instruction;
- a mode (type) associated with each instruction;
- a location in the emulation stack associated with each value used in the program (but not yet associated with any physical machine location);
- certain guaranteed conditions that simplify the subsequent phases of compilation, most importantly: no conflicts (contradictions in location identity) between the input stack at each label definition and the output stacks at each instruction that feeds control into the label.

3.2 Allocation of physical resources

This phase has the following goals:

- associate each sequence of one or more abstract instructions with a sequence of zero or more concrete machine instructions;
- determine the architectural characteristics and constraints that might affect register allocation (e.g, incoming/outgoing argument locations or register selections imposed by particular machine instructions);
- allocate machine resources (registers, physical stack locations) to each reified value in the emulation stack while: respecting architectural constraints, avoiding move chains and minimising the number of concrete instructions generated for each abstract instruction.

3.2.1 Instruction selection

Instruction selection determines which concrete machine instructions should be emitted to implement a given abstract instruction in the program.

Instruction selection and register allocation are intimately related. The selection of instructions determines when (and possibly which) registers should be allocated. Register allocation in turn can cause spill and reload code to be inserted into the program which in turn will use registers (which have to be allocated). In any case, register selection cannot begin until an initial instruction selection has determined:

- whether a given combination of operation and input/output modes is supported directly by the hardware or whether a sequence of machine instructions is required to synthesise the required operation;
- whether or not a given literal value can appear as an immediate operand (according to its size and whether the hardware supports an immediate in the corresponding operand position);
- whether an operation must reify an integer value in a register (for example, returning a logical value as the result of a function call).

Several approaches to instruction selection are possible. The simplest is to implement an exhaustive set of instruction emitters that covers all possible combinations of *operation* × *operand* mode(s). This approach has severe drawbacks:

- the number of emitters undergoes combinatorial explosion (from the number of possible permutations of operand modes);
- exhaustive case analysis is required to determine the correct emitter for a given combination of operation and operand(s) (`switch`es inside `switch`es inside...);
- the case analysis code is difficult (or even impossible) to generate automatically. It must be written (and maintained) manually;
- the resulting code generator is relative simple, but large (because of a high degree of repetition) and slow (because of the many conditional branches and indirect jumps in the case analysis).

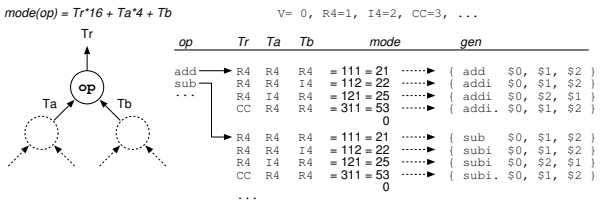

Figure 10: Table-driven instruction selection in the VPU. The output and input operand mode(s) are encoded as a numeric signature. A table associated with each operation maps mode signatures onto emitter functions each of which deals with one particular combination of modes. (In this diagram the function pointers are elided and instead the assembler template within the emitter is show in its place.)

At the other extreme is the `iburg` approach which transforms a formal, bottom-up rewrite grammar into a program that finds minimal cost covers of arbitrary subtrees in the intermediate representation. Each cover represents a (sequence of) machine instruction(s) to be emitted. In general, by maximising the number of tree nodes consumed by each cover the code generator minimises the number of instructions generated. This approach suffers from search complexity and the need for backtracking in the algorithm, both of which slow down code generation noticeably and progressively as more highly-optimised covers are added to the grammar to deal with obscure cases. (This is especially significant when all other phases of compilation are designed to minimise compilation time.)

The VPU takes an intermediate approach. It uses a table-driven, non-backtracking algorithm and a few simple heuristics to determine an optimal instruction sequence for a given combination of operation and input/output modes (in effect, a minimal cost cover for a "subtree" of depth one in the intermediate representation).

Figure 10 illustrates the table used to select machine instructions for an abstract instruction. (The nodes of the 'tree' in the VPU's abstract program are the locations in the simulation stack rather than the operations themselves.) These tables are generated trivially from a processor description file, part of which is shown in Figure 11.

For a given operation, the input and output mode(s) are combined into a numeric signature. Instruction selection searches the table associated with the operation to find an emitter function matching the signature (which it stores in the instruction for use during final assembly). If no emitter is found (which means the mode is illegal) then the first input operand that is neither a register nor a constant is forced into a register and the search repeated. If no match is found with only register and literal inputs then the first non-register operand is converted to a register and the search repeats. If there is still no match when all operands are in registers then the table (which must provide register-register modes for all instructions) is necessarily incomplete, indicating an error in the machine description file itself.

```
RI4: Add(RI4, LI4) { #[ addi r($0), r($1), $2    ]# }
RI4: Add(RI4, RI4) { #[ add  r($0), r($1), r($2) ]# }
CCR: Cmp(RI4, LI4) { #[ cmpi r($1), $2           ]# }
RI4: Cmp(RI4, LI4) { #[ cmpi r($1), $2           ]#
                     setcc($0, insn->op)           }
```

Figure 11: Part of the processor description for PowerPC. Each line describes the instruction(s) to generate for a given combination of input and output modes (such as `RI4`, representing to a 4-byte integer register). The assembler between '`#[`' and '`]#`' delimiters is converted into code that generates the corresponding binary instructions (by the `ccg` preprocessor, see Section 3.3). The positional arguments refer to the modes in each "rule" and are replaced with expressions representing the physical location (register, stack offset, constant) corresponding to that mode. (`setcc` is a function that emits code to place `0` or `1` in a register depending on the state of the condition codes and a given logical relation.)

This algorithm is much faster than BURG-style instruction selection and yet results in a similar quality of generated code on RISC processors.

After instruction selection we know precisely:

- the final class (constant, integer/floating point register, void, etc.) of each location in the emulation stack (and hence the required mode for every abstract operation);
- the locations for which machine registers must be allocated;
- the emitter function corresponding to each operation for its particular mode (cached in the instruction for use during final code generation, as described below).

3.2.2 Register allocation

Before final register allocation can begin, the code generator must satisfy any constraints imposed by

the architecture on the choice of registers. Three kinds of constraints must be dealt with:

- input constraints: for example, on the Pentium we have no choice but to place dividends in register `eax`;
- output constraints: for example, on the Pentium we have no choice but to retrieve the quotient and remainder from the register pair `eax:edx`;
- volatile registers: for example, on the PowerPC registers `r3` through `r12` are clobbered across function calls.

(The need to pass outgoing arguments, and find incoming arguments, in particular registers is just a particular combination of the above constraints.)

A separate pass is made through the program to preallocate constrained registers in their associated emulation stack locations. The bulk of the algorithm is as follows. For each instruction, in *reverse* order (from the last to the first):

- if the input is required in a particular register and this register is not flagged as clobbered in the instruction, then
  - assign the register to the instruction's output location
- if the instruction clobbers one or more registers, then
  - iterate over the instruction's output stack adding the register(s) to the set of clobbered registers for each emulation stack location (final register allocation will avoid allocating a register to a given location if it is marked clobbered in that location); and
  - remove any preallocated register for the location if it coincides with one the clobbered register(s).

Final register allocation can now be performed. The allocator creates a bit mask for each register class (integer, float, etc.) representing the set of available registers in that class and then iterates (forwards) over the program. For each instruction:

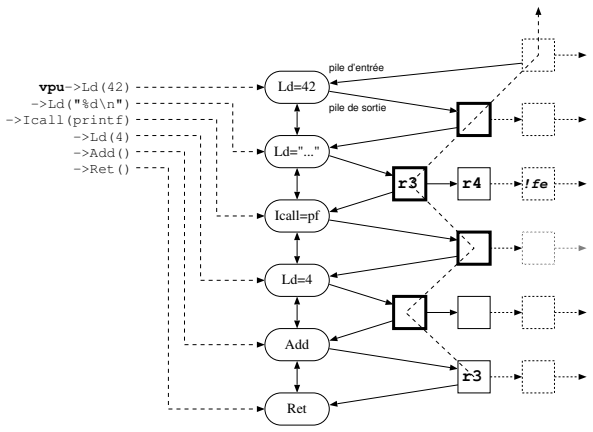

Figure 12: Allocation of constrained registers. An initial backwards pass is made through the program. The final `Ret` instruction requires its input in `r3` (on the PowerPC). The earlier call instruction requires its two arguments in `r3` and `r4` and also clobbers registers `5` through `10` in all locations beneath them on the stack (represented here by the mask '`!fe`').

- if the instruction consumes inputs then add any registers associated with its input locations to the appropriate mask;
- if the instruction generates a value in a register and the output location has not yet been allocated a register, then remove a register from the appropriate mask and assign it to the output location;
- if the instruction occurs at a basic block boundary (branch, label definition or call) then rebuild the register masks by
  - resetting them to their initial state and
  - iterating over the instruction's output stack, removing all registers encountered from the mask.

This process is illustrated in Figures 12 through 14.

3.2.3 Register spill and reload

If the register allocator runs out of available registers then it must choose a register to free up for allocation, by spilling it into the stack and then reloading it later (sometime before the instruction that uses its value). It is difficult to determine the optimal choice of register to spill without employing expensive algorithms, however a good choice (and frequently optimal) is to spill the register that is

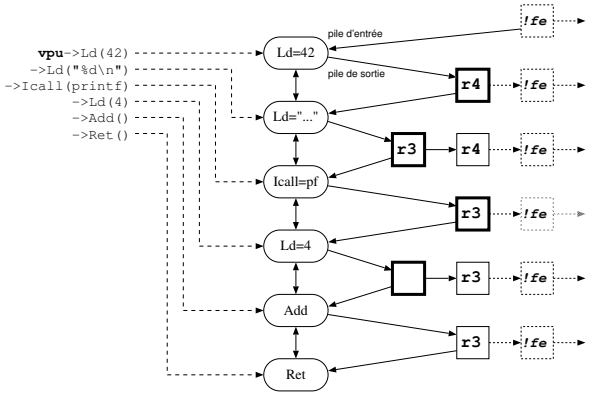

Figure 13: The situation before final allocation begins. Since locations are shared, any registers constraints and clobbered register sets are instantaneously propagated forwards and backwards to all instructions that might be affected.

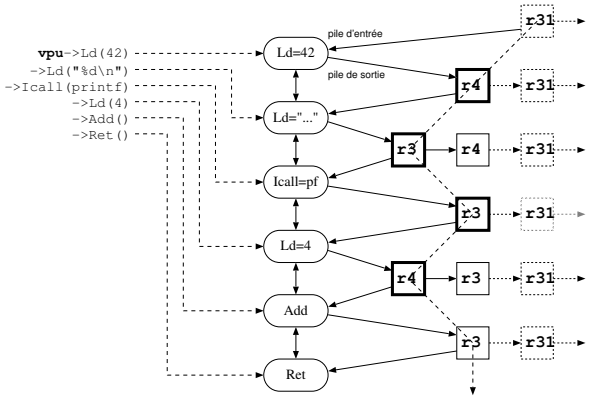

Figure 14: Final register allocation. A forward pass is made through the program to complete register allocation. Registers allocation respects the clobbered register masks stored in each location. The presence of the call instruction sets this mask to `!fe` for the lowest (shown) location in the stack thereby preventing a call-clobbered register being allocated to it; instead it is allocated the call-saved register `r31`.

most distantly used (MDU). An excellent approximation to the MDU register is available immediately to the allocator in the emulation stack attached to each instruction. The deepest location containing a register of the required class typically corresponds to the MDU. The allocator therefore scans the stack (below the location for which it is trying to allocate) to find this register and then inserts code just before the current instruction to spill it into a stack location. This is illustrated in Figure 15.

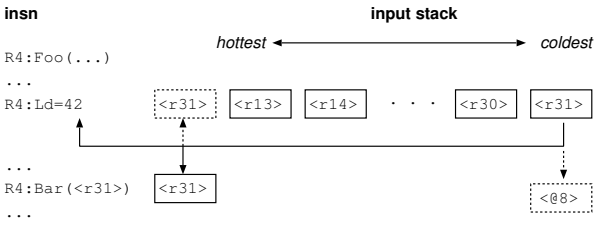

Figure 15: Spilling a register. The instruction `Bar` requires its input (a constant) in a register but none are available. The allocator scans the input stack to find the deepest register of the required class, in this case `r31`. The corresponding location is changed to refer to a location in the runtime stack (frame offset `8` in this example) and the register reused.

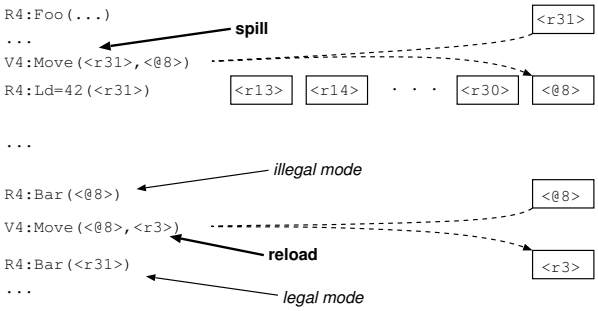

Figure 16: Reloading a register. The spilled location no longer represents a legal mode for its subsequent use in the `Bar` instruction. The instruction selection algorithm reconverts the location into a register (such that `Bar`'s mode is legal), reallocates a register to hold the reloaded value and inserts a `Move` pseudo-instruction into the program to effect the reload.

Register reload is performed implicitly by rerunning instruction selection for instructions that consume spilled locations. If the new mode is legal then the instruction can consume the value directly from the spilled location and no further action is needed. Otherwise the selection algorithm will convert the spilled input argument(s) to register modes (as described earlier), insert a 'move' pseudo-instruction into the program to mark the reload, and reallocate registers for the newly-inserted instruction. Figure 16 illustrates this process.

3.3 Final assembly

All that remains is to iterate over the code and call the emitter function attached to each abstract instruction in the program. (The emitter function ap-

CPU	*OS*	*libvpu.a*	*emit.o*
PowerPC	Darwin	158,725	52,212
PowerPC	GNU/Linux	175,612	52,088
Intel 386	GNU/Linux	124,791	31,374

Table 1: Compiled size of the VPU for 10,600 lines of C++ source code. Approximately one third of the compiled size, but less than 5% of the source code, is accounted for by the instruction emitters (inlined calls to `ccg` macros). With a little work the size of these emitters could be reduced significantly (by replacing the inline-expanded macros with function calls) at the cost of slightly slower final instruction assembly.

propriate for a given operation and operand modes is found during instruction selection and stored in the abstract instruction to avoid having to rescan the tables.) The code generator performs this iteration twice. The first iteration generates code but does not write any instructions into memory. Instead the program counter (PC) is initialised to zero and the emitter called for each instruction in turn; for each `Define` instruction, the value of the PC is stored in the associated local label. At the end of this first iteration the PC will contain the size of the final code and each local label will contain the offset of the label relative to the first instruction in the generated code. Memory is then allocated for the final code (by calling `malloc()` or some other, application-defined memory allocator) at some address M. Each local label then has M added to its (relative) value to convert it to its final, absolute address. Finally, the PC is set to M and a second iteration made over the program to generate binary instructions in memory at their final locations.

The assembler templates for binary instructions are written using a runtime assembler generator called `ccg`. A full description of it is beyond the scope of this paper, but it should be noted that the cost of assembling binary instructions using `ccg` is very low: within the emitter functions, an average of 3.5 instructions are executed for each instruction generated in memory.

4 Evaluation

At least five metrics are important when evaluating a dynamic code generator: the size of the code generator itself, its compilation speed, the memory requirements during compilation, the size of the generated native code and the execution speed of that code.

CPU	*clock*	*v-insns/sec*	*binary/sec*
PowerPC G3	400 MHz	288,400	1.1 MB
PowerPC G4	1 GHz	610,000	2.5 MB
Intel P3	1 GHz	656,108	1.8 MB
Intel P4	3.6 GHz	1,611,111	4.1 MB

Table 2: Compilation speed. The third column shows the number of virtual instructions compiled per second, and the final column the amount of native code generated per second. These figures were calculated by compiling 20 different input functions (of between 7 and 80 virtual instructions) 10,000 times in a loop. A total of 4,350,000 virtual instructions were compiled into 17.5 MBytes (PowerPC) or 11.7 MBytes (Intel) of native code, taking between 15 seconds (on the slowest machine) and 2.7 seconds (on the fastest).

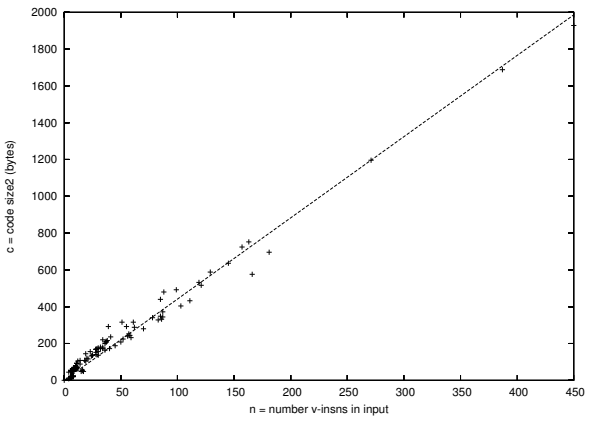

Figure 17: Compiled native code size c (bytes) per n virtual instructions, for each function of a medium-sized program (a Lisp-like dynamic language and its runtime system). The dotted line is a linear fit of the data points: $c = 4.4n$.

The VPU is a little over 10,600 lines of C++ source code. Table 1 shows the size of the compiled library (without symbols): about 170 KBytes and 125 KBytes on on PowerPC and Intel architectures, respectively.

Table 2 shows the compilation speed, averaged over 16 different functions of various sizes, compiled (and then `free()`ed) 10,000 times in a loop. On three-year-old PowerPC hardware the VPU compiles 288,000 virtual instructions per second (generating a little over 1 MByte of code), and about 656,000 instructions per second (for a little over 1,8 MByte of native code) on Intel Pentium. On current hardware the figures are 610,000 instruc-

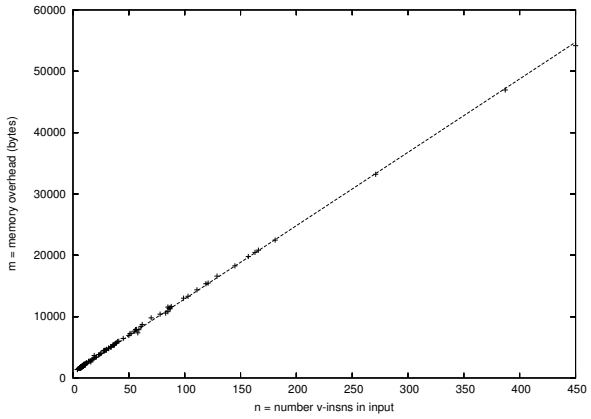

Figure 18: Memory overhead m (bytes) per n virtual instructions in each function of a medium-sized program. The dotted line is a linear fit of the data points: $m = 908 + 120n$.

tions/second (2.5 MBytes of native code) and 1.6 million instructions/second (over 4 MBytes of native code) on PowerPC and Pentium, respectively.

Figure 17 shows the generated code size plotted as a function of the number of virtual instructions per function for a much larger program (a dynamic compiler and runtime support for a complete, Lisp-like programming language). The native code size is approximately 4.4 bytes per virtual instruction on the PowerPC.

Figure 18 shows the memory requirements during compilation (for the same Lisp-like language and runtime system). Instantiating a VPU costs a little under 1 KByte of memory, with an additional 120 bytes required per virtual instruction added to that function. (All of this memory, other than that required to hold the native code, is released by the VPU once code generation is complete.)

The code produced by the VPU is typically between 10% and 20% larger than than that produced by '`gcc -O2`' for an equivalent program. Numerical benchmarks run at between 90% and 115% the speed of the equivalent programs compiled with '`gcc -O2`'.

The VPU has been used to implement many different kinds of language runtime support; for example, dynamic binding (for dispatch to virtual functions) with an inline cache. Dispatching through a VPU-generated inline cache costs approximately 1.66 times a statically-compiled C function call.

5 Conclusions

The VPU is a plug-and-play dynamic code generator that provides application support for runtime generation of C ABI-compatible native code. Integrating the VPU into any application is trivial, after which it can be used to generate arbitrary functions (from simple "partial evaluation" type optimisations through compiling scripting languages into native code for better performance). It is also an ideal component for the core of portable "just-in-time" compilers for dynamic languages, where a VPU-based dynamic code generator can be added with very little work.

Several projects are currently using the VPU aggressively. It is the execution engine for the YNVM [10], a dynamic, interactive, incremental compiler for a language with C-like semantics, a Lisp-like syntax (in other words, a C compiler in which programs and data are the same thing) and the same performance as statically-compiled, optimised C. The JNJVM [11] is a highly-reflexive Java virtual machine built entirely within the YNVM that uses the VPU directly to generate code for Java methods. Outside the domain of languages the YNVM (with a VPU inside) has been used to create C/SPAN [8], a self-modifying web cache that modifies its cache policies (by dynamically recompiling new ones) in response to fluctuations in web traffic and network conditions.

References

[1] `http://www.cminusminus.org`

[2] `http://www.cs.virginia.edu/zephyr/papers.html`

[3] `http://cs1.cs.nyu.edu/leunga/www/MLRISC/Doc/html/INTRO.html`

[4] `http://research.sun.com/research/self/papers/papers.html`

[5] M. Chen, K. Olukotun, *Targeting Dynamic Compilation for Embedded Environments*. 2nd USENIX Java Virtual Machine Research and Technology Symposium (Java VM'02), San Francisco, California, August 2002, pp. 151–164.

[6] D.R. Engler, *VCODE: a Retargetable, Extensible, Very Fast Dynamic Code Generation System*. SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1996, pp. 160–170.

[7] A. Krall, *Efficient JavaVM Just-in-Time Compilation*. International Conference on Parallel Architectures and Compilation Techniques, Paris, 1998, pp. 205-212.

[8] F. Ogel, S. Patarin, I. Piumarta and B. Folliot, *C/SPAN: a Self-Adapting Web Proxy Cache*. Workshop on Distributed Auto-adaptive and Reconfigurable Systems, ICDCS 2003, Providence, Rhode Island, May 2003.

[9] Ian Piumarta, *CCG: A Tool For Writing Dynamic Code Generators*. OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design, Denver Co, November 1999.

[10] I. Piumarta, *YNVM: dynamic compilation in support of software evolution*. OOPSLA'01 Workshop on Engineering Complex Object Oriented System for Evolution, Tampa Bay, Florida, October 2001.

[11] G. Thomas, F. Ogel, I. Piumarta and B. Folliot, *Dynamic Construction of Flexible Virtual Machines*, submitted to Interpreters, Virtual Machines and Emulators (IVME '03), San Diego, California, June 2003.

[12] `http://www-sor.inria.fr/projects/vvm/realizations/ccg/ccg.html`

[13] `http://www.gnu.org/software/lightning/lightning.html`