JITB: A Back End for Just-In-Time Compilers

by

Mathew J. Hostetter

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1997

Author .....
Department of Electrical Engineering and Computer Science
February 11, 1997

Certified by .....
Stephen A. Ward
Professor
Thesis Supervisor

Accepted by .....
F.R. Morgenthaler
Chairman, Department Committee on Graduate Theses

JITB: A Back End for Just-In-Time Compilers

by

Mathew J. Hostetter

Submitted to the Department of Electrical Engineering and Computer Science
on February 11, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

JITB (Just-In-Time Backend) is a C library for generating executable code at runtime. JITB takes as input a function written in a machine-independent assembly language with an unlimited number of registers, translates it to executable native code, and returns a C-callable function pointer. Optional optimizations allow the user to exchange compilation speed for code quality. With all optimizations enabled on a Pentium Pro, JITB typically consumes between 600 and 2400 cycles per generated host instruction.

Thesis Supervisor: Stephen A. Ward
Title: Professor

Contents

Chapter 1

Background

1.1 Rationale

Runtime code generation, defined as "dynamically adding code to the instruction stream of an executing program" [8], is becoming increasingly popular. Interest in the efficient execution of mobile programs [1] and Java byte code in particular [7] has generated recent interest in "Just-In-Time Compilers". Other compelling advantages of runtime code generation are presented in [8].

Many programming language interpreters dynamically compile programming languages to an efficient interpreted form. Examples include GNU Emacs, Perl, and Scheme48. Unfortunately, few programs dynamically compile directly to native code, and those that do tend to be ad hoc and difficult to retarget to different architectures [5]. JITB's goal is to solve this problem by making it easy to generate high-quality native code at runtime.

1.2 Related work

Programmers who wish to augment their programs with runtime code generation have few choices. One choice is to leverage the runtime code generation effort which has already been done for virtual machines, such as the Java Virtual Machine [7] and OmniVM [1]. The drawback is that using virtual machines imposes substantial semantic constraints and other baggage unwanted for many projects.

DCG [5] avoids this baggage by providing a package that translates lcc intermediate representation trees [6] to native code. By keeping a simple intermediate form, DCG is able to quickly generate native code at a cost of approximately 350 instructions per generated instruction. Unfortunately, DCG does not perform global register allocation and performs few optimizations.

VCODE [4] avoids the tree interface overhead of DCG by providing only raw instruction emitters. VCODE provides a RISC-like virtual instruction set implemented as macros that emit the raw opcode bits with the same semantics on the host processor. Because it maintains no intermediate representation for the code, VCODE is very fast, requiring only 6-10 host instructions per generated instruction. VCODE targets the extremely fast, completely unoptimized end of the spectrum; with no intermediate form, there is no framework for doing optimizations. Many optimizations could be done by the program using VCODE, but some (such as instruction scheduling) depend on the actual host instructions VCODE selects. VCODE does not perform true register allocation, placing an awkward burden on each program that uses VCODE; however, this problem is being rectified with a VCODE wrapper that does register allocation.

Work is also being done on extensions to the C language to support runtime code generation.[2] The idea behind this system is to statically compile code "templates" which get specialized at runtime by fixing the values of certain operands. This systems is extremely useful for quickly specializing known functions in predetermined ways, but not useful for general-purpose runtime compilation of arbitrary programs.

Chapter 2

Overview

2.1 The Big Picture

JITB's approach is different from that of other runtime code generators. The user emits a sequence of assembly-like instructions by making calls to the JITB library. This assembly language provides an unlimited number of registers, which JITB eventually maps to the hardware registers available on the host processor.

JITB records these instructions in an opaque intermediate representation (IR). After emitting a complete function, the user invokes JITB to compile it into executable code. Once JITB creates the executable code, the IR is discarded. The user controls the compilation speed vs. code quality tradeoff by specifying which optimizations JITB should perform.

JITB is customizable to suit different projects. JITB's external interface, the rules for producing native code for each processor, and even its IR are all easily configurable. Because much of JITB's C code is machine-generated from its configuration files, JITB can be customized to suit different purposes without touching the core engine code.

Currently JITB's optimizer can perform instruction scheduling, global register allocation, and peephole optimizations. Adding more optimizations to future versions should be straightforward.

2.2 Architecture

The JITB source tree is composed of four distinct pieces:

- Scheme-like [12] configuration files, which describe the host processor, the interface to JITB itself, and other information.
- Scheme programs which process these configuration files and generate C code.
- A C "runtime engine" that handles resource allocation, coordinates the optimization and code generation passes, and performs miscellaneous bookkeeping.
- Host-specific C files that implement functionality that varies by host architecture. For example, JITB's 80x86 back end provides C support routines to emit individual 80x86 instructions, as well as C code to perform 80x86 instruction scheduling.

The first phase of the JITB build process involves running several Scheme programs over JITB's configuration files. These programs create C code to perform a variety of tasks, such as host instruction selection and dataflow analysis. One of these programs also constructs the header file that defines JITB's external interface.

The machine-generated C code and JITB's core engine C code are then compiled into a C library. To avoid namespace pollution, all external library symbols are prefixed with `jitb_`. No Scheme code remains in the final library.

2.3 Assembly Languages

JITB deals with three distinct and potentially disjoint assembly languages, each of which is described with a Scheme-like configuration file.

2.3.1 The Assembly Language of the Host Processor

Because JITB's compiler produces and returns a block executable code, it must of course know the instruction set of the host processor. JITB is unaware of the semantics of the host

processor's instruction set; it merely uses rules specified in a configuration file to create appropriate host instructions.

2.3.2 JITB's External Interface

The external interface is the assembly language of the "virtual CPU" that JITB users target. It has an infinite number of registers. Different JITB clients, such as CPU emulators and Java compilers, might desire different JITB interfaces tailored to their needs. However, any particular interface would remain unchanged across different host architectures, in order to insulate JITB clients from details about the underlying hardware.

The external interface is independent of the intermediate representation.

2.3.3 The Intermediate Representation (IR)

IR instructions, called "insns", are the fundamental building block of JITB's intermediate representation. Insns are what JITB optimizes and then translates to executable code.

The set of possible insns can vary per host processor, but need not do so. The only constraint on insn semantics is that it must be straightforward to generate native code from them after the register allocator has replaced their pseudo registers with host registers. One reasonable choice for insns is a language similar to the host processor's instruction set but with an infinite number of registers.

A separate configuration file describes how to map the external interface to insns.

2.4 Advantages

JITB's overall approach has several advantages:

2.4.1 A Consistent Interface to a Range of Compilation Speed/Code Quality Tradeoff Points

Most programs spend most of their time in a relatively small number of functions, so it only makes sense to spend time optimizing those functions. It can be difficult or impossible to

statically predict which functions will be the performance bottlenecks, so a reasonable strategy is to quickly compile all functions unoptimized and later recompile the "popular" functions optimized.

JITB makes it easy to explore different compilation speed vs. code quality tradeoffs. From the user's perspective, the only difference between compiling a function optimized and compiling it unoptimized is the set of optimizations requested when JITB is asked to compile the function. Most other runtime code generators do not provide this option.

One interesting strategy would be to maintain a heap of unoptimized functions, sorted by total execution time consumed (or some other heuristic). A low-priority background thread would always be optimizing the function at the top of the heap. Maximally optimized functions would be permanently removed from the heap. So while the user stared at his screen or grabbed a cup of coffee, his program would keep getting faster and faster.

JITB is reentrant; all state for each function being compiled is stored in a separate, dynamically allocated C struct. Consequently, multithreaded simultaneous compiles are not problematic.

2.4.2 An IR Easily Amenable to Optimization

JITB's IR is a directed graph of basic blocks, each consisting of a doubly-linked list of assembly-like instructions. This form lends itself naturally to standard compiler optimizations, such as code motion and instruction scheduling before register allocation. Most available general-purpose runtime code generation packages either have no IR at all [4] or a tree-based IR [5]. Unlike most runtime code generators, JITB can remember information about the final host instructions selected and use that to perform instruction scheduling on the native code.

2.4.3 Easily Customizable and Extensible

JITB does not care much about the semantics of the IR; for the most part JITB optimizes the IR and generates code through blind pattern matching. One way to think about JITB is as a configurable runtime compilation engine. By adjusting various configuration files it is

easy to extend JITB's external interface, modify the IR, change how code is generated, etc.

This is nice because JITB's engine can be thoroughly tested, optimized, and debugged and then used for a variety of projects which have dissimilar needs. For example, the JITB interface most useful for a PowerPC emulator might be very different from the interface appropriate for a Java bytecode compiler.

Chapter 3

JITB Library Interface

3.1 Creating a function

JITB compiles one function at a time. The semantics of each function are specified by feeding JITB a sequence of assembly-like instructions with an unlimited number of registers (the "external interface"). Once the sequence is complete, the function can be compiled. The specific instruction set for the external interface is determined by a configuration file that is processed when the JITB library is built. This thesis considers JITB's default external interface, a simple three-address load/store assembly language.

Here is how to create a function:

```
jitb_compilation_t my_function;
my_function = jitb_new_compilation(0, 0);
```

`my_function` will be used to keep track of information about the function as it is created. It will serve as the first argument to almost every JITB function.

Instructions can now be appended to `my_function` by calling JITB functions. The specific set of instructions in the external interface is described in a configuration file, but each of them is emitted by calling the corresponding `jitb_emit_...` C routine. For example, here is how to append an instruction that increments a register:

```
jitb_emit_addc(my_function, my_reg, my_reg, 1);
```

JITB uses functions to append instructions rather than a parser for some representation of those instructions for three reasons. First, a parser-based interface would be unwieldy for many applications, such as compilers for programming languages. Second, constructing a representation of an instruction only to have it deconstructed by a parser imposes needless overhead. Third, if a parser is desired one can easily be layered on top of JITB's function-based interface.

3.2 Operands

JITB knows about three classes of register and constant operands: word, long, and floating point. There are three disjoint sets of pseudo registers, one to hold values of each type.

A "word" is an integer value of the natural size for the host processor. For most CPUs of interest a "word" is a 32-bit integer, but on some (such as the 64-bit DEC Alpha), a "word" is 64-bit. In contrast, a "long" is always a 64-bit integer value.

Another operand type is a "memory zone set." This is a bitmask that describes which of several disjoint memory regions might be touched by each instruction that accesses memory. For example, one memory zone might refer to space on the stack, and another might refer to heap-allocated memory. Notifying JITB which specific areas of memory each instruction might touch helps JITB recognize when it can safely perform certain optimizations.

For example, in many languages code which modifies array bounds can be distinguished from code that modifies array contents. By using separate memory zones for each, it becomes much easier for JITB to detect when it can load the array bounds only once outside of a loop that writes to the array.[1] Another example is the stack slots which hold spilled registers. Stack slots get their own "memory zone", which gives JITB much more freedom to schedule register loads and spills around other memory references than it would otherwise have.

The types of the operands for each instruction are determined by the opcode. For example, `sub` takes three word registers as arguments, but `subl` takes three long registers

[1] JITB does not yet perform this optimization.

as arguments. A type error such as using a word register where a long register is expected results in undefined behavior.

Pseudo registers are created by calling a JITB routine that doles them out one at a time. For example, the following code creates and returns a register that can hold a word-sized value:

```
my_reg = jitb_new_reg(my_function, JITB_WORD_REG);
```

There is no fixed limit on the number of registers that can be created this way, although creating a very large number of registers will consume more memory and slow down compilation.

Incoming function parameters are also accessed via pseudo registers. These registers are allocated the same way as any other, but the register type is different. `JITB_WORD_PARAMETER_REG`, `JITB_LONG_PARAMETER_REG`, and `JITB_FP_PARAMETER_REG` indicate word, long, and floating point parameters, respectively. The first parameter register created is assumed to be the first argument to the procedure, and so on. Note that although pseudo registers are used to access function parameters, it is irrelevant whether the calling convention on the host processor is actually register based. JITB hides such details.

After emitting a complete function, the user calls a JITB routine to compile that function into executable code. One of the parameters to the compilation routine is a bit mask indicating which optimizations are desired. Currently the set of supported optimizations is small: global register allocation, instruction scheduling, and peephole optimizations. As one would expect, enabling optimizations produces better code but takes longer.

Here is an example that creates a function that moves a constant into a register, doubles its value, and returns it. This example demonstrates both how to create code and the associated housekeeping. Note that most of the routines called here need to be called only once per function generated.

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

```
#include "jitb.h"


int
main (int argc, char *argv[])
{
  jitb_compilation_t comp;
  jitb_word_t (*func)();
  jitb_word_t result;
  int reg;

  /* Initialize JITB.  Call this only once. */
  jitb_init ();

  /* Start a new compilation. */
  comp = jitb_new_compilation (0, 0);

  /* Allocate a new pseudo register to hold a "word". */
  reg = jitb_new_reg (comp, JITB_WORD_REG);

  /* Move the constant 123 into that register. */
  jitb_emit_movc (comp, reg, 123);

  /* Add the register to itself. */
  jitb_emit_add (comp, reg, reg, reg);

  /* Return that value to the caller. */
  jitb_emit_ret (comp, reg);

  /* Compile the function. */
  func = jitb_compile_word_func (comp, JITB_ALL_OPTIMIZATIONS,
                                 0, 0);

  /* Free up the memory assocated with the compilation. */
  jitb_destroy_compilation (comp);

  /* Call the function we just created. */
  result = func ();

  /* Free up the function we compiled. */
  jitb_free_compiled_code (func, 0);

  /* Print out the result. */
  printf ("Function returned: %ld\n", (long) result);
```

```
  return EXIT_SUCCESS;
}
```

Chapter 4

Configuration Files

JITB uses four Scheme-like configuration files to specify a complete configuration. These files are processed by various Scheme programs and used to construct C code. These files describe JITB's "external interface", its intermediate representation (IR), the mapping between the external interface and the IR, and the mapping between the IR and the host processor's instruction set. JITB creates implicit default configuration file entries when appropriate, to help minimize the size and complexity of some of these files.

4.1 External Interface

The file `default-api.scm` defines the assembly-like language that JITB clients will target. This "external interface" is independent of what processor JITB is actually running on, so programs that use JITB need not worry about the underlying hardware.

For example, the default external interface specifies a `mov` instruction which moves one register to another:

```
(define-api (mov (dst-reg outregw) (src-reg inregw)))
```

Note how the `mov` opcode is followed by a list of (operand name, operand type) pairs. Each name can be an arbitrary identifier, but each type must be one of a specific list of types JITB knows about. An `inregw` is a word-sized input register, and an `outregw` is a

word-sized output register. Several other operand types exist. A `mov` instruction would be appended to the current function with a C call like:

```
jitb_emit_mov(current_function, dest_reg, source_reg);
```

Here's another example, this time for `addc`, which adds a constant to a word register and stores the result in a word register:

```
(define-api (addc (dst-reg outregw) (src-reg inregw)
                  (num constw)))
```

4.2 Intermediate Representation "Insns"

"Insns", described in section 5.3, are the assembly-like instructions of the IR. The specific set of insns available in the IR typically vary by host processor; the processor-specific file `host-insn.scm` defines any insns that are needed beyond those created by default from the external interface. The specific syntax of `host-insn.scm` entries is unimportant so it is not described here; this section merely provides some examples to give a feel for what they look like.

Here is the insn the 80x86 uses for left shifts. This insn specifies that the shift count register must be stored in `%ecx`. This is a constraint imposed by the 80x86 architecture. Forcing the register allocator to choose `%ecx` for the shift count register tremendously simplifies the task of the 80x86 code generator (which maps insns to 80x86 instructions).

```
(define-insn (sl (dst-reg outregw) (src-reg-1 inregw)
                 (shift-count-reg inregw))
 ((hard-reg-shift-count-reg "REG_ECX")))
```

Here's another example, this time for the 80x86 `idiv` instruction. `idiv` requires certain fixed registers, and also has two output registers. Defining a specific insn for `idiv` tells the register allocator what it needs to know to allocate registers properly.

```
(define-insn (i386-idiv-insn (quotient-reg outregw)
                             (remainder-reg outregw)
                             (src-reg-1 inregw)
                             (src-reg-2 inregw))
   ((hard-reg-quotient-reg "REG_EAX")
    (hard-reg-remainder-reg "REG_EDX")
    (hard-reg-src-reg-1 "REG_EAX")
    ; Use any register but %edx for the second operand.
    ; The back end will need to temporarily use %edx to
    ; hold the sign-extension of %eax.
    (hard-reg-src-reg-2 "REG_EAX" "REG_EBX" "REG_ECX"
                        "REG_ESI" "REG_EDI")))
```

4.3 External Interface → IR Mapping

The assembly languages for JITB's external interface and JITB's IR need not be the same. When they differ, JITB needs to know how to map from the external interface (which users of JITB see) to the IR manipulated by JITB's optimization and code generation passes (which users never see). The file `host-apihandler.scm` specifies this mapping.

JITB uses this specification to construct C functions whose names and arguments meet the external interface, but whose end result is to create the corresponding IR and append it to the function currently being accrued.

For example, the `divs` handler emits an `i386-idiv-insn` (described in the previous section). If this handler were not present, JITB would default to creating a `divs` insn type and emitting a `divs` insn whenever `jitb_emit_divs` were called. That default is unacceptable because it would not convey enough information to the register allocator.

```
(define-apihandler (divs dst-reg dividend-reg divisor-reg)
  (emit i386-idiv-insn dst-reg "MAGIC_CLOBBER_REG"
        dividend-reg divisor-reg))
```

Another example is the `callwv` handler (`callwv` calls a procedure returning "void" whose address is in a word register). The standard 80x86 calling convention specifies that operands be pushed on the stack. This handler calls a subroutine that emits a number of

i386-push insns, then an i386-call insn, then an addc insn to clean up the stack. Here one external interface instruction gets mapped to many insns.

```
(define-apihandler (callwv func-reg num-args args)
  (begin (declare-and-set "size_t" "stack_arg_size"
                          (call "jitb_i386_push_all_func_args"
                                "_comp" num-args args))
         (call "_jitb_emit_insn_i386_callw_insn" "_comp"
               "MAGIC_CLOBBER_REG" func-reg num-args args
               "MAGIC_CLOBBER_REG" "MAGIC_CLOBBER_REG")
         (if "stack_arg_size"
         (call "jitb_emit_addc" "_comp" "REG_ESP" "REG_ESP"
               "stack_arg_size"))))
```

4.4 Back End

The file host-backend.scm specifies patterns that map sequences of insns to executable code for a particular processor. Here are the patterns from the 80x86 back end for orc. orc does a bitwise OR of a constant and a register and stores the result in another register.

The syntax of these patterns is explained in chapter 6.2.1.

```
; Bitwise OR of -1 and a register always yields -1
(define-pattern ((orc dr reg -1))
  #t
  (movl-const-reg dr -1))

; ORing 0 to a register is a NO-OP
(define-pattern ((orc dr dr 0))
  #t
  )

; ORing 0 to one register and storing it in another is a move
(define-pattern ((orc dr sr 0))
  #t
  (i386-movl-reg-reg dr sr))

; ORing a constant to a register is one 80x86 instruction
(define-pattern ((orc dr dr num))
```

```
  #t
  (i386-orl-const-reg dr num))

; If src and dest registers differ we need two instructions
(define-pattern ((orc dr sr num))
  #t
  (i386-movl-reg-reg dr sr) (i386-orl-const-reg dr num))
```

Chapter 5

Intermediate Representation

5.1 Functions

Functions are created one at a time. All information about each function is stored in a `jitb_compilation_t` struct, which serves as the first argument to nearly every JITB routine. This struct records several pieces of information, such as the number and type of all pseudo registers created so far. It also maintains a pool of temporary memory used to hold dynamically created data structures associated with the current function. All temporary memory will be freed at once when compilation is complete. Dynamically created data structures are allocated from this pool both to increase speed and to avoid the possibility of memory leaks.

The intermediate representation (IR) for a function's code is a directed graph of basic blocks, each containing a doubly-linked list of "insns". An insn typically represents a simple instruction like `add r5,r645,r12`, which adds register 645 to register 12 and stores the result in register 5.

5.2 Basic Blocks

Basic blocks represent intraprocedural [1] linear control flow sequences of insns. The only entry point for a basic block is at the beginning, and the only place a branch can appear is as the last insn in a block.

Blocks can have 0, 1, or 2 "child" blocks (blocks to which control flow might be transferred after this block). An example of a block with zero children is a block ending with a "return from function" instruction. Blocks which "fall through" to the next block and blocks which end in an unconditional branch have one child. Only blocks ending in conditional branches have two children.

Each block can have an arbitrary number of parent blocks, except the first block in the function, which always has zero parents. This guarantees JITB's compilation engine a place to insert code that gets executed only once per function. This magic first block is "hidden" to the user, so the zero-parent constraint isn't visible. This hidden block "falls through" to the first block visible to the user, which can have as many parent blocks as it likes.

Each block's insn list has special "sentinel" insns at the beginning and end. This representation simplifies certain operations, such as splicing in new insns. The sentinel insns are actually fields in the `jitb_block_t` struct itself, making it easy to immediately find either the first or last insn in the block. Both forward and backward traversals of the insn list are easy.

Blocks also record several other pieces of information, such as loop nesting depth, the "liveness" of each register on entry, and which host registers contain which pseudo registers on entry and on exit. For this reason, the memory JITB requires to compile a function is proportional to the product of the number of blocks and the number of pseudo registers.

[1] I say "intraprocedural" because subroutine call insns can actually appear in the middle of a basic block. Although they temporarily transfer control elsewhere, they can appear in the middle of a block because they do not affect control flow in any problematic way. They are not a problem because JITB makes conservative assumptions about the side effects of a subroutine call; it assumes subroutines clobber all memory and any registers not preserved by the calling convention.

5.3 Insns

An insn represents a single instruction such as adding two registers together and storing the result in a third register. Each insn is recorded in a `jitb_insn_t` C struct containing the opcode, its operands, and liveness information for each register operand. This liveness information is computed by the dataflow pass and used by the register allocation pass.

The `jitb_insn_t` struct type is visible only inside the JITB library. Clients of the library have no access to it. The only thing a client can do with insns is call JITB's external interface routines, which append insns to the current function.

Like JITB's external interface, the set of possible IR insns is also described with a configuration file (see section 4.2). This file lists the opcodes, operand types, and special properties of the insns used in the IR. The insns in the IR can vary by host architecture; for example, the Intel 80x86 JITB configuration files describe an `i386-idiv-insn` which clobbers certain 80x86 registers and computes both quotient and remainder, just as the 80x86 `idiv` instruction does. `i386-idiv-insn`, like all insns, is invisible outside JITB, but is useful internally because it provides valuable information to the register allocator.

The actual `jitb_insn_t` struct type is machine-generated when the JITB library is built. Since the insns available can vary by host architecture, so must the `jitb_insn_t` struct. The `jitb_insn_t` struct consists of fields shared by all insns followed by a union that holds the operands appropriate for each opcode. For example, it might look like this:

```
typedef struct _jitb_insn_t
{
  struct _jitb_insn_t *prev, *next;
  jitb_opcode_t opcode;

  union
  {
    _jitb_insn_union_0 u0;
    _jitb_insn_union_1 u1;
    _jitb_insn_union_2 u2;
    _jitb_insn_union_3 u3;
    _jitb_insn_union_4 u4;
  } u;
} jitb_insn_t;
```

Each element of union u describes a unique set of operand types. For example, add and sub insns have exactly the same set of operands; in each case operand 0 is an output register and operands 1 and 2 are input registers. JITB automatically creates one union type to hold the operands for both add and sub:

```
typedef struct __jitb_insn_union_0
{
  jitb_word_t _a0;           /* reg for operand 0        */
  jitb_word_t _a0_liveness;  /* "liveness" for operand 0 */
  jitb_word_t _a1;           /* reg for operand 1        */
  jitb_word_t _a1_liveness;  /* "liveness" for operand 1 */
  jitb_word_t _a2;           /* reg for operand 2        */
  jitb_word_t _a2_liveness;  /* "liveness" for operand 2 */
} _jitb_insn_union_0;
```

Sharing works fairly well; on the 80x86 JITB creates only 25 separate union entries to describe the 25 distinct sets of operand types required by 109 different insns.

Sharing the same union type for add and sub (not to mention and, or, xor, etc.) frequently allows JITB to use the same pieces of code to process both insns. For example, registers are allocated for both add and sub insns with precisely the same block of code (machine-generated code that "knows" that insn->u.u0._a0 holds an output register and insn->u.u0._a1 and insn->u.u0_a2 hold input registers). Sharing code in this way helps keep library size down and improve instruction cache locality.

Since the order in which the fields in each _jitb_insn_union_N struct (e.g. _a0, _a1) are listed is semantically irrelevant, JITB sorts the struct fields by their size. So, for example, all 8-byte operands (e.g. those of type double) are listed first, followed by all 4-byte operands. This avoid needless "struct hole" padding bytes inserted by the C compiler to ensure that N-byte struct fields are aligned modulo N. Thus struct size is minimized.

The fact that the jitb_insn_t struct type is machine-generated suggests a question: how does JITB create and manipulate insns when the details of their C types are forever subject to change? The answer is that every line of JITB's code that accesses insn operands is machine-generated. When configuration files change, all the code which creates and manipulates insns is regenerated, keeping it always "in sync" with the current jitb_insn_t

datatype. Details of this process are presented in the next chapter.

Chapter 6

Machine-Generated C Code

6.1 Philosophy

Several years ago I wrote Syn68k, a dynamically compiling 68LC040 emulator [11]. Syn68k and JITB face several of the same problems; for example, how to encode rules for processing a wide variety of instructions. For the most part Syn68k used lookup tables to encode a wide variety of information about how to process instructions. In retrospect, this was a mistake; I now believe that such information should have been specified using code, not data. I did not make this mistake with JITB.

JITB abolishes machine-generated lookup tables and uses machine-generated C code instead. Scheme-like expressions in configuration files are processed, optimized, and turned into C code when the JITB library is built. Since these expressions need to evaluate to something that can be statically translated to C code, they are really just macros. Even so, macros are a powerful way to express a wide range of functionality. JITB hackers can of course define their own macros and use them however they like in JITB's configuration files.

For example, if a hypothetical configuration file wanted to specify that an "increment" instruction should be emitted if the constant one is being added to a register, it might include code that looked like this:

```
(if (= value 1)
```

```
(emit increment reg)          ; "if" case
(emit addc reg reg value))    ; "else" case
```

(In this particular case, JITB's pattern matcher generator might be better suited to the job; this is described in section 6.2).

Using code instead of lookup tables incurs a space penalty since lookup tables are more concise than code. However, I believe the penalty to be small for what JITB does, especially considering that many sequences of "hard-coded" C code can be "shared" by different cases requiring the same functionality (for example, in JITB `add` and `sub` wind up sharing the same register allocation code). Furthermore, the hard-coded approach is usually faster than interpreting lookup tables.

One place where machine-generated C code is found is in JITB's "emit" functions, which append insns to the function being accrued. If the configuration files specify JITB's external interface provide an `add` instruction, JITB will automatically create a `jitb_emit_add` function.

The default machine-generated `jitb_emit_add` allocates a new `jitb_insn_t` struct exactly large enough to hold an `add` insn, fills in its operands from the parameters to `jitb_emit_add`, and appends the newly created insn to the current function.

It turns out that the default `jitb_emit_sub` is almost exactly the same as `jitb_emit_add`. Each function creates and appends an insn with two input register operands and one output register operand. The only difference is the opcode field of the created `jitb_insn_t` struct: it is `JITB_OPC_ADD` in one case and `JITB_OPC_SUB` in the other. JITB handles this nicely by creating one magic hidden function to emit either insn. This special function takes an extra argument that provides the opcode. JITB creates C macros for `jitb_emit_add` and `jitb_emit_sub` that call the shared routine with the extra opcode argument. Sharing code in this way keeps library size down and improves instruction cache locality.

The rest of JITB's machine-generated code is created by the pattern matcher generator described in the next section.

6.2 The Pattern Matcher Generator

At the core of the C code generator used to build JITB is the pattern matcher generator (PMG). It is a Scheme [12] program that takes as input an ordered list of {pattern, outcome} pairs and produces C code that executes the outcome of the first matching pattern listed.

PMG is the engine that outputs C code to perform register allocation, dataflow analysis, and disassembly of the intermediate form. It also creates the code which maps sequences of insns to host instructions.

6.2.1 Syntax

A pattern is an arbitrarily long sequence of insn descriptions and a constraint on the values of their operands. The constraint is an arbitrary Boolean expression which must be true if the pattern is to match. Each pattern has an associated "outcome" that is executed when the pattern matches. For example:

```
(define-pattern ((addc dest-reg source-reg constant-number))
  (= constant-number 0)          ; operand constraint
  (move dest-reg source-reg))    ; outcome
```

This pattern says "if adding the constant zero to a register, just move that register to the destination register."

The pattern syntax allows some shorthand. If the same identifier names two different operands, they must be equal. If an operand is not named and instead just listed as a constant, that operand must equal that constant. So this:

```
(define-pattern ((addc dest-reg dest-reg 0))
  #t     ; "true"
  )      ; Adding 0 to a register is a NO-OP. Do nothing
```

is exactly equivalent to this:

```
(define-pattern ((addc dest-reg source-reg constant-number))
  (and (= dest-reg source-reg) (= constant-number 0))
  ) ; Adding 0 to a register is a NO-OP. Do nothing
```

Multi-insn patterns are also useful. Here is the pattern from the 80x86 back end that recognizes the common "decrement-and-branch-if-nonzero" sequence and generates an 80x86 decrement instruction followed by an 80x86 conditional branch instruction:

```
(define-pattern
  ((subc dr dr num)      ; subtract num from register dr
   (jnec t dr 0))        ; jump if register dr != 0
  (!= num 0)
  (addl-const-reg dr (- 0 num))
  (i386-jne))
```

Without this pattern, the default behavior would be to match the `subc` and the `jnec` independently. This would generate an `add` instruction, then a compare of the result against zero, then a conditional branch — three 80x86 instructions. However, the compare is unnecessary because the condition code bits set by the `add` already say whether or not the result was zero. By pattern matching more than one insn at a time the back end can potentially omit the compare, generating two 80x86 instructions instead of three.

6.2.2 How the Pattern Matcher Generator Works

PMG creates a sequence of C switch statements and binary searches to see which pattern matched. It switches on insn opcodes and does binary searches by testing "atomic" predicates, such as seeing whether an operand is zero, seeing if one operand equals another, and so on. These predicates are "atomic" because they are the simplest thing JITB can test in an `if` statement. A pattern constraint can combine an unlimited number of atomic predicates in an arbitrary way with `and`, `or`, and `not`.

An important objective for any pattern matcher generator is to minimize the expected time taken to match a pattern. JITB makes this goal explicit by associating with each atomic predicate a likelihood and a "delay cost" to test it. This provides a quantitative way to determine the "expected time" of a decision tree, and therefore the efficiency of two

decision trees can be compared. The fastest tree (i.e., the one with the smallest expected time) is the best.

Although PMG uses these likelihood and delay weights internally, currently JITB provides no syntax to specify them. All atomic predicates are assumed to have a likelihood of $\frac{1}{2}$ and an evaluation cost of one. This generates nicely balanced search trees, but they can't take into account how likely certain predicates are. One day JITB should gather predicate likelihood information at runtime and then feed that information back into JITB to create more efficient decision trees. However, profiling shows that JITB spends a very small amount of time matching patterns, so this is a low priority. It may become a higher priority once the very fast but non-optimizing end of the code generation spectrum is explored.

The first thing PMG does is generate a `switch` on the opcode of the insn currently being matched. It must do this because the insn's operands are stored in a `union`; knowing the opcode is a prerequisite for knowing which union element holds those operands. PMG notices when the case statements for two different opcodes turn out to be exactly the same and merges them together into one case.

Once the opcode is known, PMG filters out all but the patterns which match that opcode and examines their predicates. Typically multiple patterns will share some of the same "atomic predicates." For example, both of these inclusive-or patterns from the 80x86 back end check to see if the third operand is zero:

```
(define-pattern ((orc dest-reg dest-reg 0)) #t)
(define-pattern ((orc dest-reg source-reg 0)) #t
  (i386-movl-reg-reg dest-reg source-reg))
```

PMG begins constructing a decision tree by making a list of all the atomic predicates that any matchable pattern cares about. PMG then evaluates decision trees that test those atomic predicates in different orders. Each decision tree is scored by computing the expected time to match the pattern. The fastest tree is what eventually gets generated as C code.

PMG works recursively. It creates an `if` statement that tests an atomic predicate, then figures out which patterns can still match if that predicate is true and which can match if

it is false. It recursively creates two subtrees with the filtered pattern lists. PMG notices when any two subtrees are identical and shares them with a `goto` statement. This has the effect of making an `or` test in a pattern constraint efficient.

Furthermore, PMG knows that some atomic predicates imply others. For example, suppose two patterns test `(> x 15)` and `(> x 5)`, respectively. PMG knows that if `(> x 15)` is true, then `(> x 5)` must also be true and there is no need to test it. It also knows the implications of the converse. PMG currently only knows how to draw inferences between pairs of simple relational operators.

Although some will find this horrifying, PMG currently works by brute force: it exhaustively creates and scores all valid decision trees. This algorithm obviously takes exponential time, but in practice the number of distinct atomic predicates has been so small that it is fast enough. Building the pattern matcher for the entire 80x86 back end takes less than two minutes, which is not bad considering that PMG is written in Scheme.

If the exponential growth becomes a problem, there are a variety of simple heuristics that could be applied. For example, at each level of the recursion PMG could try testing each "unknown" atomic predicate and counting how many patterns remain matchable if it is true and how many remain if it is false. It would then select the atomic predicate which minimizes the sum of the squares of those two numbers and recurse.

When a pattern lists multiple insns, PMG recursively generates switch statements to test the opcodes of subsequent insns. Inside each of those switches are a sequence of decision trees, which can themselves contain switches. PMG uses its normal "expected pattern match cost" function to determine where in the decision tree such switches should be placed.

6.2.3 When PMG Gets Used

As one would expect, PMG is used when a configuration file lists an explicit set of patterns to be matched. What is not so obvious is that PMG is also used internally by JITB whenever insns with different opcodes need to be handled in different ways. As an example, consider JITB's dataflow pass.

One phase of the dataflow pass marches through the insns in a block from last to first, keeping track of which registers are live [1] and which are dead at each point in the block. The code that computes this information needs to "know" for the insn it is currently processing which operands are input registers and which are output registers.

The body of this dataflow loop is machine-generated by a Scheme program. This program creates exactly one constraintless pattern for each possible insn opcode and feeds the pattern list through PMG. The outcome associated with each pattern is the code that updates register liveness information in the way appropriate for that opcode. For example, one such pattern might look like:

```
(define-pattern ((add out in1 in2))
  #t
  (mark-reg-dead-on-entry out)
  (mark-reg-live-on-entry in1)
  (mark-reg-live-on-entry in2))
```

Here PMG is basically used to generate a big switch statement and to deal with the details of accessing the operand values for different types of insns. PMG also does some handy optimizations; in the case of the dataflow loop, PMG notices that the opcodes `add`, `sub`, `and`, `or`, and `xor` are handled identically and shares the code to process them.

The C code created by PMG is typically used within a loop over all the insns in a block. For example, this is the aforementioned dataflow loop:

```
/* Start out at b->last_insn.prev to skip the sentinel. */
for (i0 = b->last_insn.prev; ; ++insn_index, i0 = i0->prev)
{
#include "mgen-propagate.h"
}
sentinel_hit:
```

`mgen-propagate.h` is the file created by PMG. It basically consists of a large `switch` statement with one entry for each opcode. The handler for the "sentinel" insn

[1] A "live" register is one whose value is needed by a subsequent insn.

(recall that a sentinel insn starts and ends each block) does a `goto sentinel_hit`, thus properly terminating the loop without any special loop tests.[2]

PMG optionally speeds up these pattern matching loops with a GNU C Compiler (gcc) extension to the C language. gcc lets programs take the address of a C label with a unary `&&` operator, and `goto` an arbitrary pointer. Instead of creating a `switch` statement, which must "bounds check" its argument, PMG generates a direct table dispatch (e.g. `goto *dispatch_table[insn->opcode]`). Instead of `case` statements, PMG generates labels whose addresses are listed in the dispatch table. Furthermore, each "pattern outcome" *ends* with a `goto`, jumping directly to the decision tree for the next insn. This eliminates the branch to the top of the loop.

[2]The sentinel insn handler can't just `break` because it may be inside a switch statement.

Chapter 7

Compilation Algorithms

Most compiler research has focused on aggressive static compilation. Sacrificing compilation speed for improved performance in the resulting binary is common. Comparatively little research has been done on very fast approximations to standard compiler techniques. This chapter partially fills that need by explaining JITB's fast heuristic compilation algorithms.

7.1 Block Sort

The order in which JITB processes basic blocks affects the quality of its generated code. The register allocator in particular is sensitive to processing order; it will tend to do a better job allocating registers for a basic block if its parent blocks have already been processed. It is therefore important to choose a good block ordering. Since everything is being done at runtime, the ordering must be chosen quickly.

JITB uses the following algorithm to order the basic blocks. Once a block is listed, it will not be considered by any later steps in this algorithm.

1. Create an array with one slot for each block.

2. Store the function's entry block in the first array slot.

3. Identify each block whose children are all already listed and store that block in the

last empty array slot. Recurse on each parent of that block, in case that parent now has all children listed.

4. Identify each block whose parents are all already listed and store that block in the first empty array slot. Recurse on each child of that block, in case that child now has all parents listed.

5. If some blocks remain unlisted, store the unlisted block with the lowest "address" in the first empty array slot and go to step (4). A block has address N if it was the Nth block appended to the function by the user.

This ordering has some interesting properties:

- All blocks not in any loop are grouped together at the beginning or at the end of the list.

- If block X is an ancestor of block Y, and block Y can be executed no more than once, block X will be listed before block Y.

- Ancestors of a block will usually be listed before that block, except of course in the case of blocks in loops where no such ordering is possible.

- Blocks tend to be listed in "dominance order", although they are not always. Block X dominates block Y if and only if any path from the function entry block to block Y must go through block X.

- Blocks in a loop will tend to be listed contiguously between the loop entry block and the branching block at the end of the loop.

7.2 Loop Nesting Depth

Knowing how frequently each basic block will be executed is extremely useful. If a block will be executed a great many times it makes sense to push as much work as possible outside of that loop, minimize register spills in that loop, and perhaps even devote extra time to optimizing that loop (e.g. use a better but slower instruction scheduler).

One way to compute frequency information is to gather runtime profiling data about how often each block is executed and feed that information back into a second compilation of the same function. JITB does not yet support this. Instead, JITB does what static compilers have been doing for years: it guesses based on loop nesting depth.

The "loop nesting depth" of a basic block is zero for all blocks not in any loop. The depth is one for blocks inside a single loop, two for blocks inside a loop inside a loop, and so on. The motivation for computing this number is that the more deeply a block is nested, the more frequently it will probably be executed.

JITB uses an extremely fast yet effective algorithm to guess the loop nesting depth of basic blocks. This algorithm can be fooled but it gets exactly the right answer for most functions. [1]

The algorithm is as follows:

1. Set a loop nesting depth counter (LNDC) to zero.

2. Process the blocks in the reverse of the order computed in the "block sort" pass described in section 7.1. For the current block being considered:

   (a) For each child block not already processed, increment LNDC.

   (b) Record the current value of LNDC as this block's loop nesting depth.

   (c) For each parent block already processed, decrement LNDC.

Intuitively this algorithm loops through the list of blocks keeping track of how many "backward branches" are active at each point. That number approximates the loop nesting depth. It relies on the fact that if a block listed after block X by the "blocksort" heuristic branches to a block listed before block X, block X is probably part of a loop involving both those blocks.

[1] Unfortunately, getting the wrong answer can be costly. In one benchmark JITB's generated code lost more than a factor of two in performance because it misidentified certain basic blocks as not being in the innermost loop of a triply-nested loop. For this reason, and because computing loop nesting depths takes almost no time with the current algorithm, it probably makes sense to replace this algorithm with something slower but more robust.

7.3 Instruction Scheduler

Currently JITB does not have a generic instruction scheduler that could be used for several different CPU types. Instead it has a scheduler for the 80x86. The scheduling algorithm is generally applicable to any superscalar processor so it should be straightforward to generalize it to other CPUs. The algorithm does not generate an optimal schedule but it does identify most easy wins. This section describes the 80x86 scheduler.

Since every JITB memory reference insn is annotated with a bit mask describing which of several disjoint memory zones it touches, JITB has enough information to reorder many loads and stores with respect to each other. For example, JITB "knows" that an access to the stack slot for a spilled register and an access to heap memory cannot overlap, so it is free to schedule these instructions with respect to each other.

JITB's 80x86 instruction scheduler is a simple greedy list scheduler that processes one basic block at a time. The Pentium is an extremely difficult chip to schedule for optimally, so JITB doesn't even try. Instead, JITB simply tries to avoid placing an instruction which computes a value next to an instruction which uses that value. Since the Pentium is only two-way superscalar this does a pretty good job in practice.

On other processors in the 80x86 family, the benefits of instruction scheduling are less significant. It doesn't help much on the 80486, but it can avoid certain pipeline stalls. The Pentium Pro is quite forgiving of poorly scheduled code, so JITB could reasonably not bother to schedule when it detects that the host processor is a Pentium Pro. JITB does not yet do this.

JITB's compilation pass maps the insn sequence in each block to a sequence of annotated 80x86 instructions. JITB annotates each 80x86 instruction with a bit mask describing which values it inputs and a bit mask describing which values it modifies. The bits in these masks correspond to a variety of things: the registers in the 80x86 register set, the 80x86 condition code bits, various disjoint "memory zones", and a "no reorder" bit which means "don't reorder any instructions with respect to this one."

"no reorder" is a special magic value which is all instructions claim to input but is output only by non-reorderable instructions, so the "don't reorder" effect happens automatically

as a side effect of meeting normal dataflow dependency constraints.

JITB schedules with a simple greedy algorithm. It starts by emitting the first instruction in the block and starts looking for the next instruction to emit. After it finds and emits the best one it looks for the next one to emit, and so on until all instructions have been emitted.

In this loop JITB scans the next few instructions and considers only those which dependency constraints allow to be emitted now. JITB emits the first of those instructions which inputs no values that were output by the most recently emitted instruction. In other words, it avoids putting "producers" and "consumers" back-to-back. If no such instruction exists, JITB emits the first listed instruction which has not yet been emitted.

This algorithm takes linear time with a low constant factor, because analyzing dependency constraints requires only a few bitwise operations per instruction. This algorithm does miss scheduling opportunities, but in practice it picks up most of the easy wins. A more powerful (but slower) scheduling algorithm would be a nice option to offer the user.

7.4 Dataflow

Effective register allocation requires knowledge about how each register is used. JITB's dataflow pass computes this information and annotates each register operand for each insn with an integer indicating "how live" it is.

7.4.1 "Liveness"

A register is "live" at a certain point in the code if its value may be used by some subsequent insn. A register is "dead" if its value can never be used again. The classic live/dead distinction divides registers by *whether* they are ever used again. JITB further divides live registers based on *where* they will next be used. This information determines the "liveness" of each register.

A liveness of zero means that register's value is completely dead and it can be clobbered with impunity. Very large livenesses are reserved for registers that will be used again later in the same basic block. The sooner the register will be used again within the block, the "more live" it is.

The liveness of any other register is simply the loop nesting of the block where it is next used. So a register next used inside a tight loop is considered "more live" than a register that isn't. The intent is to tend to allocate registers in a way that will not require any extra spills inside loops. If a register next might be used in either of two places, the deeper loop nesting determines its liveness.

7.4.2 Computing Liveness

Conceptually, JITB begins by determining which registers are used and which registers are preserved by each block, ignoring all others. This "local use" information is then used as a building block to recursive searches that determine the "global liveness" of each register on entry to each block. Global liveness takes into account both the usage patterns of the current block and the usage patterns of all descendant blocks. This is the really useful information; local use information is useful only to speed up these global liveness computations.

In practice, JITB often doesn't bother computing "local use" information for particular blocks. If all children of some block have already had their "global liveness" information computed, then JITB can directly compute the "global liveness" for that block and skip the local use computation.

Local use information is computed by a straightforward loop through the insns in a block. Global liveness information is computed with recursive searches that check how registers get used by any possible path through the code. Once the global information for all child blocks is known, JITB can loop through the insns in the block and annotate each one with the liveness of each register. Both loops described in this paragraph are machine-generated as described in section 6.2.3.

These passes are simple, but one neat trick is worth mentioning. The "block entry livenesses" computed by these passes are small integers indicating the loop nesting depth where each register will next be used. JITB records these loop nesting depths as an array of one byte unary numbers, with one array element per register. This representation is handy because the MAX operation in unary is simply bitwise OR. By ORing a word's worth of bytes together at a time, JITB efficiently performs multiple MAX computations in

parallel. This trick lets JITB quickly merge the liveness arrays for the two descendants of a conditional branch.

7.5 Register Allocation

JITB uses a heuristic-based register allocation algorithm. Registers are "lazily" allocated one basic block at a time, and then extra "patchup" instructions are inserted between blocks. These patchup instructions assure that registers are allocated in an acceptable way on entry to the destination block. JITB uses various heuristics to minimize the number of patchup instructions created, to minimize the number of overall register spills, and to try to keep register spills outside loops. The intent of this heuristic approach is to do a decent job allocating registers without tackling any NP-complete graph-coloring problems.

7.5.1 The "Regstate"

JITB maintains two "regstate" structs for each block: one for block entry, and one for exit. Each regstate records mappings between pseudo registers and host registers. A regstate also notes whether or not each host register holds a "dirty" value (one whose value has been modified). Registers which do not hold "dirty" values can be overwritten without spilling the old value to a stack slot.

A block's entry regstate places a burden on other blocks. Any block which transfers control to this block must guarantee that pseudo registers are cached in host registers in a way compatible with the destination block's entry regstate. For example, a block might demand that, on entry to the block, pseudo register 19 must already be cached in host register `%eax` and pseudo register 330 be cached in host register `%ebx`. If a block does not specify any pseudo register to be cached in a particular host register, that host register is "clobberable" by that block and must not be holding any useful value when that block is entered. The entry regstate can also demand that registers not be dirty.

The exit regstate reflects where pseudo registers actually ended up at the end of the block. For example, it might indicate that `%eax` is holding pseudo register 37, `%ebx` is holding a "dirty" value for pseudo register 5, and `%ecx` is holding no useful value.

7.5.2 Processing Order

JITB's register allocator processes blocks in the order computed by the "blocksort" pass described in section 7.1. For each block, it usually begins by assuming registers are cached in the same the way they were upon exit from the parent block with the deepest loop nesting.[2] The idea is to minimize the number of "patchup" insns that need to be inserted between those two blocks. JITB then processes the insns in the block from first to last, allocating registers as it goes.

7.5.3 Machine-Generated Inner Loop

The main register allocation loop is machine-generated based on information in JITB's configuration files. Custom code is created to allocate registers for each type of insn (see chapter 6 for an overview of JITB's machine-generated C code). Generating custom code gracefully handles the wide variety of possible insns. For example:

- An `add` instruction needs to allocate three registers while a `mov` instruction allocates two.
- Some insns need to allocate floating point registers, most do not.
- Some insns require specific host registers be used to hold certain operands.
- Some insns clobber particular host registers (e.g. a "call" insn clobbers registers not preserved by the C calling convention).

7.5.4 Register Selection

JITB allocates registers one insn at a time, processing insns in order from first to last in each basic block. It remembers which pseudo registers are cached in which host registers as it goes, and how "live" each register is. For each insn it allocates all input registers

[2]The exception is when the parents are not nested as deeply as the current block; in that case, their output regstates are ignored until it is time to insert "patchup" instructions. This way the entry regstate for the loop head block only contains what it needs.

and then all output registers. JITB processes operands in order of how tightly constrained the register choices for that operand are. The mostly tightly constrained operands get their choice of registers first to make sure that some other operand doesn't "tie up" a particular host register they desperately need.

The exact heuristics JITB uses to choose which host register to use to hold a particular operand are somewhat complicated. Here is the current list of heuristics, in order of decreasing importance:

- JITB must obey constraints about which host registers are acceptable. For example, the 80x86 shift-by-register insns require that the shift count be stored in register `%ecx`. JITB must, first and foremost, obey these restrictions.
- JITB cannot ever allocate the same host register for two different input operand registers for the same insn.
- If a pseudo register is already in a host register, JITB will just leave it there when possible.
- JITB strongly prefers to grab a host register which is holding no useful value.
- If all host registers currently hold useful values, JITB prefers the register that seems to hold the least important value (i.e. has the smallest "liveness"). Liveness is described in section 7.4.1.
- JITB prefers "non-dirty" registers that don't need to be spilled back to memory.
- JITB prefers to allocate input registers in host registers that have not been touched so far in this basic block. This allows the load of that host register's value to be pushed back to the block entry regstate, and perhaps outside of a loop.
- JITB remembers a set of "preferred" host registers for each pseudo register, and prefers to allocate registers from that set. Typically this set contains only the host register most recently allocated for this pseudo register. So if register 37 is cached in register `%eax` in one place, it will tend to get cached in `%eax` in other places too.

The intent is to minimize "patchup" code between blocks by allocating registers in a consistent way.

- JITB remembers the last "killed" register (the one that most recently stopped holding a live value). JITB slightly favors reusing this register, for two reasons. First, it has the side effect of compiling instructions like "move r52,r37" into NO-OPs when the source register is no longer used after that block; r52 and r37 will simply be allocated to the same register and no move is needed. Second, reusing a source register as the destination register generates better code for two-address architectures.

- Failing everything else, JITB allocates registers "round robin". The intent is to facilitate instruction scheduling by using non-overlapping registers for adjacent instructions when possible. It is not clear if this heuristic is useful.

When JITB detects that a register needs to be loaded from or spilled to a stack slot, JITB doesn't simply insert an insn to do the load/spill (although that would work). Instead, JITB sees if it can change the "entry regstate" for the current block to request that the value be loaded/spilled before the block is even entered. This has the nice property of often moving loads/spills outside loops. JITB will eventually "push" these requests back from one block to another, to try to move them out of nested loops, but this feature is not yet implemented.

Although this heuristic allocator does a decent job even on the register-starved 80x86, it turns out to be much more complicated than I thought it would be. The initial idea sounded simple: lazily allocate registers within blocks and then tie those blocks together with patchup insns. Unfortunately, I found that a surprisingly large amount of bookkeeping and heuristics are required to make this approach effective. This slows down what at first sounded like a very fast algorithm.

For example, `jitb_alloc_word_reg`, which selects a register for a single operand, is 179 lines of C code. This is far more than I anticipated. Fortunately, most of these lines deal with specific cases and don't get executed each time the function is called. Even so, it still executes roughly 17 lines of bookkeeping code even when the pseudo register is already in a host register. Furthermore, the trickiness of handling all those uncommon cases correctly has been JITB's biggest source of subtle bugs.

Some of `jitb_alloc_word_reg`'s complexity can be blamed on its being flexible enough to support the sometimes irritating register constraints of the 80x86 instruction set. However, most of its problems seem to be inherent in the approach; the simplicity of lazily allocating registers "as it goes" is at least partially offset by the complexity of figuring out exactly when loads and spills can be "pushed" back to the block entry regstate.

Fortunately, it should be easy to experiment with different register allocation algorithms, and even allow the user to select between those algorithms. It may also be the case that the ideal runtime register allocation algorithm for RISC processors with many registers is substantially different from the ideal algorithm for the 80x86, which has only 6 free registers (which, to add insult to injury, are non-orthogonal). If this is true, then creating one algorithm general enough to deal with the idiosyncracies of the 80x86 may have been a mistake. This deserves further investigation.

Chapter 8

Security Issues

Some runtime compilation systems allow arbitrary code to be compiled and executed securely [7, 1]. With these systems, malicious code (perhaps downloaded from the Internet) either fails to compile or generates exceptions at runtime. This is a useful property, since it allows untrusted code to be executed without fear.

JITB makes no security guarantees. It remains neutral on the security issue by shifting the burden to the package using JITB. If things like runtime checks are desired, the package using JITB should create instructions to perform the checks and feed them to JITB where they will be compiled just like any other instructions.

Omitting security from JITB's responsibilities is not laziness. By leaving it out JITB remains useful for projects where no security checks are desired. One example is an emulator for trusted code. Another example is a system where security is handled through a completely different mechanism, such as one based on the "proof carrying code" concept [9], where untrusted programs bring with them an easily verifiable "proof" that they violate no security constraints. For experimental systems like that, security checks inside JITB would be unnecessary and unwelcome baggage.

Chapter 9

Benchmarks

There are two facets to JITB's performance: JITB's compilation speed and the speed of JITB's generated code. To measure these, I integrated JITB into the runtime compiler for the programming language Curl [3] and ran benchmarks.

9.1 Curl

Curl is an object-oriented language with many interesting features. Curl code can be compiled either dynamically or statically. Curl is a "safe" language: type mismatch errors are always caught and array references are bounds-checked.

Before JITB there were three ways to run Curl programs:

- Interpreter. The Curl interpreter, itself written in Curl, is slow but very portable.
- Curl's "simple" runtime native code generator. Even without JITB, Curl can generate native code at runtime. However, the simple code generator does not perform register allocation or any other optimizations.
- Curl→C translator. Curl's runtime compiler was extended to generate C code instead of native code. The C code is then compiled with a standard C compiler, such as the GNU C Compiler (gcc). Because the same runtime compiler creates both the C code and JITB's input, meaningful comparisons can be drawn between the code produced

by JITB and that produced by an optimizing C compiler. The drawback to using the Curl→C translator is that such code must be compiled statically for a particular processor and then linked into the Curl executable.

JITB provides a fourth way to compile Curl programs, and will eventually replace the "simple" runtime code generator.

9.2 Compilation Speed

I measured JITB's compilation speed with the 80x86 `rdtsc` instruction. This instruction returns the number of processor cycles elapsed as a 64-bit number. The difference between this number before and after JITB's main compilation routine indicates how long a function takes to compile.[1] Dividing the cycles elapsed by the number of host instructions created gives a normalized measure of compilation performance.

On a Pentium Pro with all optimizations enabled, JITB typically consumes between 600 and 2400 cycles per generated host instruction. This is unacceptable for programs that need to recompile extremely frequently, such as programs that create specialized functions "hard-coded" for particular inputs. However, this latency is perfectly acceptable for "lazily" compiling normal programs at runtime. To put it in perspective, the 180 MHz Pentium Pro on my desk can generate between 75000 and 300000 instructions per second with full optimization.

Unfortunately, the completely unoptimized portion of JITB's performance curve has not yet been explored. Currently JITB requires a global register allocation pass to produce correct code. This could be fixed without too much difficulty, although making JITB's "best case" compilation times extremely good will require a new approach to describing host architectures, to teach JITB how to emit raw instruction bits in only a few cycles.

The factor of four difference in compilation times is interesting. The complexity of the function being compiled determines how long it takes to compile; functions with nested

[1] Unfortunately, `rdtsc` gives pessimistic numbers on a multitasking system, since cycles taken by other processes count against JITB. However, I believe this effect is small since I ran benchmarks on an unloaded machine.

loops, many basic blocks, and very many pseudo registers take much longer to compile than small, simple functions. Reducing worst-case compilation times is a subject of ongoing investigation.

9.3 Code Quality

This table lists the number of seconds that various benchmarks take to execute (so smaller numbers are better). All benchmarks are written in Curl, and the times for the two run-time compilers (JITB and Simple) include runtime compilation time. gcc -O2 times were computed by compiling the output of the Curl→C translator with gcc.

	JITB	gcc -O2	Simple	Interpreter
bubblesort	8.44	8.70	39.35	3483.46
gcd	17.92	15.53	36.16	4820.27
fibonacci	29.02	25.94	26.60	3191.09
matrix multiply	8.69	7.91	37.42	2661.87

- "bubblesort" bubble sorts an array of 10000 ints. Each array reference is bounds-checked.
- "gcd" uses a "bitwise-arithmetic" algorithm (no divides or mods) to compute the sum of the greatest common divisors of nine million different pairs of numbers.
- "fibonacci" uses a doubly-recursive algorithm to compute all values of the Fibonacci function up to Fib(37). Its running time is entirely dominated by subroutine call overhead.
- "matrix multiply" does 90000 10x10 matrix multiplies using a simple matrix multiply algorithm. The dot product inner loop dominates the running time. Each array reference is bounds-checked.

JITB's code quality is comparable to gcc's on all of these benchmarks. This is encouraging. Admittedly there exist many other benchmarks on which gcc would perform far

better; for example, gcc would soundly beat JITB on any code where constant propagation, constant folding, or loop transformations resulted in a substantial improvement. There is no reason in principle why JITB could not also perform these optimizations; this is a topic for future research.

Chapter 10

Future Work

10.1 Support More Architectures

Currently only an 80x86 back end has been written. Porting to new architectures should be relatively simple; the bulk of the work will be writing the architecture-specific configuration files.

10.2 Cleaner Specification of Host Processor Instruction Sets

The 80x86 back end uses many ad hoc C functions to emit raw 80x86 instructions. This was useful because the specific sequence of bytes for an 80x86 instruction can vary in strange ways depending on which specific registers are used; these rules are encoded in a few shared subroutines. It would be possible to encode all the rules for generating 80x86 code directly in the JITB's host processor configuration file [11], but that approach would probably compile to larger code.

The New Jersey Machine-Code Toolkit [10] is a fairly general system for describing instruction sets. It may be possible to use the New Jersey Machine-Code Toolkit to simplify the process of creating JITB back ends.

10.3 Byte Code Back End

For some applications, it might make sense to compile infrequently executed code to a concise byte code and interpret it. Byte code that got executed often enough could get recompiled into faster native code. This would save memory.

10.4 CPU Emulator Support

The code generation needs of CPU emulators are somewhat different than those of normal programming languages.

A CPU emulator typically needs only a fixed number of pseudo registers (one per register of the machine being emulated plus a few more for miscellaneous purposes). Several of JITB's passes could be sped up if the number of pseudo registers were fixed.

A more flexible dataflow pass would be useful. For example, some CISC processors (such as the 80x86 and 680x0) have opcodes that modify the low byte of the destination register without touching the rest. Few if any RISC processors have opcodes like this. These 8-bit opcodes can be emulated on a RISC processor with clumsy byte-insert instruction sequences, but this is often unnecessary. Knowing the liveness of bytes *within* individual registers, instead of just the liveness of complete registers, would allow these 8-bit opcodes to be "widened" to 32-bit opcodes when the high 24 bits of the destination register were determined to be dead. Explicit support for condition code bits might also be handy.

Being able to compile raw sequences of insns that are not part of a function is vital for CPU emulation.

10.5 More Optimizations

- Simple dead code elimination should be trivial. It could be done as a side effect of the dataflow pass.
- Common subexpression elimination would frequently be useful.

- Loop code motion would be nice. JITB should actually be able to do a fairly good job of this because it knows which "memory zones" are touched by each memory reference.
- Constant folding and propagation would substantially help some code.

10.6 More Register Allocation Algorithms

Currently JITB always performs the dataflow and register allocation passes because it doesn't know how to generate code without host registers being assigned. JITB should be extended to optionally skip these passes and generate very bad code very quickly.

Functions with few pseudo registers might sometimes be able to have their registers trivially assigned: one particular host register for each pseudo register for the entire duration of the function.

10.7 Finish Support for 64-bit Arithmetic

The implementation of JITB's "long" opcodes is not yet complete. Since these require pairs of registers on 32-bit processors, this further complicates register allocation.

10.8 Support Archiving Dynamically Created Code to Disk

Some applications may find it useful to archive heavily optimized code.

Chapter 11

Summary

JITB shows that a fast compiler can produce very efficient code through careful application of approximations and heuristics. Furthermore, JITB shows that efficiency and flexibility are not mutually exclusive; JITB's approach of mapping configuration files to C code allows a wide range of functionality to be specified in a clean way without sacrificing speed.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. Technical Report CMU-CS-95-204, Carnegie Mellon University, October 1995.

[2] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershard. Fast, effective dynamic compilation. In *Conference on Programming Language Design and Implementation*, 1996.

[3] Curl home page. http://curl.lcs.mit.edu/curl.

[4] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the 23rd Annual ACM Conference on Programming Language Design and Implementation*, May 1996.

[5] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of ASPLOS-VI*, pages 263–272, October 1994.

[6] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. software - practice and experience. *Software - Practice and Experience*, 21(9):963–988, September 1991.

[7] Java white papers. http://java.sun.com/nav/read/whitepapers.html.

[8] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report Technical Report 91-11-04, University of Washington, 1991.

[9] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Submitted to the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[10] Norman Ramsey and Mary Fernández. The new jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference, New Orleans, LA*, pages 289–302, January 1995.

[11] Syn68k: ARDI's dynamically compiling 68LC040 emulator. ftp://ftp.ardi.com/pub/SynPaper.ps.

[12] William Clinger and Jonathan Rees, Editors. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers IV*, July–September 1991.