Templates-based portable Just-In-Time compiler

Alex Iliasov, KRSU, Bishkek.
e-mail: alexili@soros.kg

Abstract

Usage of the platform-neutral bytecode interpreters is often limited by their restricted performance. Just-in-time compilers effectively solve this problem. However they are hard to develop and retarget. This paper demonstrates that dynamic code generation from the templates created by a C compiler can be used to build a simple and highly-portable JIT compiler.

Keywords: just-in-time compilation, bytecode interpretation, templates, dynamic code generation, code in-lining.

1 Introduction

Design of the existing JIT compilers generally follows the design of the common static compilers. Major differences come from the fact that JIT has to be quite fast since time spent for compilation is a part of the total program's running time. Some JIT compilers do selective compilation of the most speed-critical sections while using interpreter for the rest of the program. This gives additional time for JIT compiler to make more aggressive optimization that could be as good as optimization of a quality static compiler. JIT compilers are often developed as a companion of a bytecode interpreter. Many interpreting languages were designed to be platform-independent. Indeed classic bytecode interpreters are very portable. Existing JIT compilers require substantial changes to be ported from one platform to another. Porting JIT compiler requires in-depth knowledge of the target platform architecture. Hard to port and complicated JIT compilers are often out of reach for small development teams especially in research projects.

Suggested technique can be used to considerably increase the performance of an already existing switching, threaded or direct threaded bytecode interpreters. Existing compiler from source language to bytecodes instructions may be reused with the new JIT compiler. Carefully designed instruction set of the bytecode interpreter can be used as a foundation for the JIT compiler. Retargeting the JIT compiler for a new platform takes little effort and does not require any assembly language programming. Experience project for Lemick interpreter demonstrates 2 to 50 times speed up over pure switching interpreter.

2 Portable JIT compiler from bytecodes to the inlined machine code

In the suggested scheme a JIT compiler transforms bytecodes into a machine code using compiled code blocks or templates. During the compilation the JIT compiler substitutes a bytecode instruction with a corresponding template. Besides just copying, the JIT compiler can replace parts of the template with the operands of the bytecode instruction. Due to its simplicity this kind of JIT compiler should be very fast. Compilation using templates totally eliminates instruction dispatching overhead that it' alone gives significant performance boost. Inlining as a way of reducing dispatching overhead was first suggested in [1]. One important consequence of zero dispatching overhead is that previously interpreted languages could use very fine-grained instruction set as they migrate from interpreting to JIT compilation. Fine-grained instruction set with a good optimizing compiler to bytecodes could be a way to achieve the performance comparable to those of optimizing C compilers

Most complicated parts of implementing such JIT compiler are building proper code templates and discovering argument offsets inside the templates.

2. 1 Code templates

Code templates are created with any available C compiler for the target platform. GNU C first-class labels are needed to mark templates code boundaries; however it is possible to emulate this GNU C specific feature with just few lines of asm.

There are some restrictions on the templates of machine code. A machine code template must be relocatable to be inlined. However it may be used even if it can not be inlined. The JIT compiler compiles non-relocatable templates as an absolute jump to a template. In the extreme case when all the templates are non-relocatable the JIT compiler will produce some kind of the direct threaded code*.

To be relocatable a template must comply with the following restrictions:

- All the function calls have to use the absolute address of the called routine. Relative addresses are invalidated when the code template is copied to a new location. On many platforms C compilers use hidden function calls to implement some operations. A special care should be taken to detect such cases.

- If a bytecode instruction has immediate arguments then corresponding constant values in the code template must be compiled inline. For C compilers it is a common practice to put a value in the constant pool and then refer to it by an address.

- Code templates should not contain relative jumps that cross a border of a code template. JUMP-like bytecode instructions are the only exception. For them the JIT compiler will write the proper displacement value as an instruction operand.

- A function which contains code templates is not intended to do something meaningful. Sometimes C compiler optimizations may break templates consistency. Any part of a code template and a code template as whole should appear as a reachable code to the compiler to avoid "dead code" optimization. The result of the every calculation made inside the code template should appear to be useful to the compiler.

Code templates which can not be made relocatable are marked as non-relocatable. Non-relocatble templates the JIT compiler handles in a special way. Instead of the template code, the JIT compiler will write a jump to the template code. A relocatable template of an absolute jump instruction should be present in the templates set. In multi-threaded environment non-relocatable templates should not have any operands. Non-relocatable (NR) templates should pass control back to the next instruction in the compiled machine code. A realistic and thread-safe way to do this is to push return address on the stack just before the jump to the NR template.

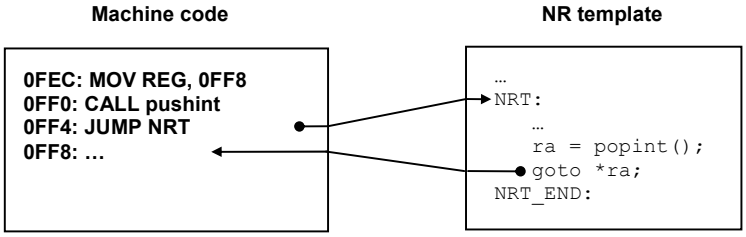

Figure 1: Compilation of non-relocatable templates

Notice that machine code on the figure 1 is also created with the help of templates. First two lines are from the template that calls function *pushint* with one immediate argument. Third line could be produced by a relative jump template.

In C labels can not be placed outside functions thus all the templates are put inside a single function. If the function containing code templates has local variables then the compiled machine code should begin with the function prologue. The function prologue code allocates space on the stack for the local variables. If the control passed to the compiled

* An absolute jump instruction should be relocatable to implement this variation of the direct threaded interpreter.

machine code using CALL-style instruction then the compiled machine code should end with the function epilogue. The function epilogue code does the proper return from the function call.

```
typedef void* (*compiled_start_t)(void*);
…
compiled_start_t compiled_function;
void *function_args;
…
compiled_function  = compile_bytecode(bytecodes_array, templates_set1);
compiled_function(function_args);
```

Figure 2: Function-style call

Code templates can be written so that they use only global or static variables. In this case compiled machine code can be executed with an absolute jump, like *goto *address*. To return back to the caller the compiled machine code should end up with a jump to some address defined by the caller.

```
void *compiled_code, *return_address;
        …
        compiled_code  = compile_bytecode(bytecode_array, templates_set2);
        *return_address = &&L1;
        goto *compiled_code;
        L1:
```

Figure 3: Goto-style call

Function-style call is easier to use, especially in multi-threaded environments. Below is an example of the tiny yet complete code templates function. Two instructions, MOVAI and MOVBI take arguments – integer numbers. Since it is not guarantees that variables *a_int* and *b_int* will be really allocated in the registers these templates can be only used with a function-style call.

```
void* codemine(void* args) {
        register int a_int, b_int;
        void* retpos = &&LFOOTER;
        LHEADER:
[here should be code to build the instructions table]
                …
                sMOVAI:
                        a_int = UNIQUE_4B(111);
                sMOVBI:
                        b_int = UNIQUE_4B(111);
                sADDI:
                        a_int = a_int + b_int;
                sMULI:
                        a_int = a_int * b_int;
                sEXIT:
                        goto *retpos;
                eEXIT:
                …
                dummy(a_int, b_int);
        LFOOTER:
        return 0;}
```

Figure 4: templates function

The JIT compiler needs templates addresses and their operands offsets. Here is the code to build the templates table. Macros defenitions can be placed anywhere. Table construction code should be placed inside the templates function.

```
#define ADD_INST(c, s, e) memcpy(cwp, s, e - s); \
                          cwp += e - s; it[c].args = 0;
                          it[c].s = s; it[c].e = e; \
#define ADD_INST_1ARG(c, s, e, o1) memcpy(cwp, s, e - s); \
                          cwp += e - s; \
                          it[c].s = s; it[c].e = e; \
                          it[c].args = 1; it[c].argsof = o;

ADD_INST_1ARG(IN_MOVAI, &&sMOVAI, &&sMOVBI, OFS_MOVAI)
ADD_INST_1ARG(IN_MOVBI, &&sMOVBI, &&sADDI, OFS_MOVBI)
ADD_INST(IN_ADDI, &&sADDI, &&sMULI)
ADD_INST(IN_MULI, &&sMULI, &&sEXIT)
ADD_INST(IN_EXIT, &&sEXIT, &&eEXIT)
ADD_INST(IN_ENTER,codemine, &&LHEADER)
```

Figure 5: templates table construction

Now it is obvious how to compile this bytecode into the machine code:

```
ENTER    ; function prologue
MOVAI 1  ; load 1 (integer) into register A
MOVBI 2  ; load 2 (integer) into register B
ADDI  A  ; add A and B and store result in A
MULI  A  ; multiply A and B and store result in A
EXIT     ; leave function. A = 6; B = 2
```

2.2 Templates portability

Immediate values that can not be handled within a single instruction, C compiler breaks into the several parts. For i86x these parts are machine words, in case with i86x-32 their size is 32 bits. Example below shows that 64 bits double constant is broken into two 32 bits parts. Thus ODR should look for two 32 bits fields when dealing with double type.

Long double type that is 12 bytes long on i86x-32 is broken into three 32 bits parts. If C compiler supports long long type it is represented as two words, the same case is with double type.
Sometimes C compiler will use a reference to the value in the constants pool instead of an immediate value. However there is a portable way to make C compiler output relocatable code for constants assignment.

	Non-relocatable constant assigment	Relocatable constant assigment

C code	`double a;`
	`a = imm;`
		`double a;`
		`register int *b;`
		`…`
		`b = &a;`
		`b[0] = bits(imm, 0, 31);`
		`b[1] = bits(imm, 32, 63);`

Sparc assembler	`.LLC0:`
	`.uaword imm`
	`…`
	`sethi %hi(.LLC0), %o1`
	`or %o1, %lo(.LLC0), %o0`
	`ld [%o0], %o1`
	`st %o1, [%fp-32]`
	`sethi %hi(.LLC0), %o1`
	`or %o1, %lo(.LLC0), %o0`
	`ld [%o0], %o1`
	`st %o1, [%fp-28]`
		`sethi %hi(bits(imm,10,31)), %o1`
		`or %o1, bits(imm,0, 9), %o0`
		`st %o0, [%fp-32]`
		`sethi %hi(bits(imm,42,63)), %o1`
		`or %o1, bits(imm,32,41), %o0`
		`st %o0, [%fp-28]`

imm here is a 64 bit double constant. **bits(value, start, end)** is a pseudo-function that cuts *end – start* bits begining from *start* bit and shifts the resulting value *start* bits right.

sethi instruction sets higher 22 bits of a register and clears all the remaining bits. **or** is an inclusive-or operation.

2.3 Instruction operands in the JIT compiler

Many bytecode instructions require arguments. Even if a bytecode interpreter is a pure stack machine there are still some instructions which take immediate arguments not from the stack. To simplify the JIT compiler and make it faster it is a good idea to pass only immediate values as arguments. All the rest like registers and operands types is better coded in an opcode of an instruction. So instead of a single instruction that accepts several different sets of arguments you should better make several instructions, each with a fixed argument set. Types that are longer then machine word are always represented as integer number of words. Thus JIT compiler has to accept only word-size immediates. Immediate values longer then word could be treated as several word-size arguments. This simplifies the JIT compiler and makes the compilation process faster.

A label as an argument of a bytecode instruction is a special case for the JIT compiler. Bytecode compiler can not calculate jump offset values as it does not know anything about the corresponding code templates. Thus it inserts some bytecode instructions to mark a label position and puts label code as an argument for various JUMP instructions. There are two essentially different strategies for the JIT compiler to replace label codes with the proper jump offsets. First strategy is to compile in two passes. In the first pass the addresses of all the labels are calculated. In the second the machine code is written.
Another strategy is to compile in a single pass. Address of the very label met so far is used to compile all but the forward jumps. When the compiler meets a jump to an unknown label (forward jump) it records the position of the instruction argument in the compiled machine code and later writes there a proper value.

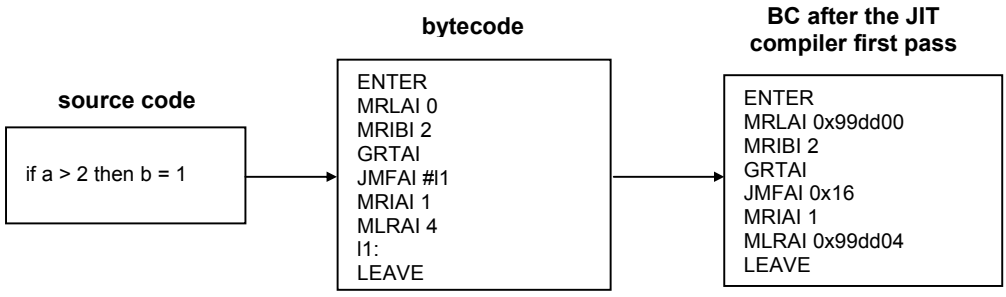

Figure 5: Labels as instruction operands

Figure 5 shows what transformation with the initial source code should occur before the compilation to the machine code.

3 Experience

Suggested JIT compiler design was first developed to build a JIT compiler for the existing Lemick bytecode interpreter. Lemick is research project on building a environment-neutral language for easy and secure distributed programming. More information can be obtained on http://lb.to.kg.
Beside other things the JIT compiler in Lemick is used to resolve opcodes overloading. Overloading is used when some operations have several meaning depending upon the execution context. Lemick allows remote execution of the program threads over the network. Remote threads should use specific implemetation of many operations, such as global variables access, IO and other. This can be very effectively achieved with a JIT compilation.

Lemick VM loosely follows load/store architecture. The registers are used to do all the calculations. Ten registers of various sizes are available. Lemick VM operates on five data types with 2 registers for every type.

Type name	Size in bytecode (bits)	Size in VM (bits)	description
Integer	32	32 or 64	fastest available integer

Long	64	32, 64 or 128	largest available integer
Object	32	32 or 64	pointer type
Float	32	32	32 bit float
Single	64	64	64 bit float
Double	128	64, 96 or 128	largest available float

Table 1: Lemick VM data types

There is a growing stack to save temprorary values and pass arguments to functions. Local and global variables are stored in the separate memory regions. Distinct commands are used to access local and global memory regions. Lemick VM and the JIT compiler was initially implemented for i86x-32 and ported to Sparc and MIPS. All the instructions were successfully implemented as relocatable.

3.1 Benchmarks

Tests were made on Pentium III 600Mhz running Linux 2.4.18 with GNU C 3.2. GNU *time* command was use to measure running time. Values in the table are the summary time in seconds spent by the program in kernel and user modes. First benchmark tests expressions evaluation and loops. Each row represents results for the same test but with the different variables types as denoted in the leftmost column. *LJIT* column shows the running time for the code compiled from Lemick Basic source. *LJIT hw* are running time for the same program but hand written bytecodes. *Kaffe* is a open source JIT-based Java VM, version 1.0.7. *C* column shows running time for non-optimized C code, *C -O3* – for C code compiled with the most aggressive optimization. To let reader feel how performance of compilers relates to the performance of interpreters there is a running time for the corresponding program in *Perl*. Numbers in braces shows many times were faster *LJIT* (the first value) and *LJIT hw* (the second value) in the test.

Type	LJIT	LJIThw	Kaffe	C	C –O3	Perl
double	2.746	0.814	12.469(4.54/15.30)	1.100(0.40/1.35)	0.556(0.20/0.68)	51.120(18.61/62.89)
integer	0.776	0.452	0.659(0.84/01.45)	0.664(0.85/1.42)	0.278(0.35/0.61)	
float	1.337	0.682	6.235(4.66/10.02)	1.098(0.82/1.76)	0.554(0.41/0.81)	
long	1.199	0.781	3.250(2.71/04.16)	0.864(0.73/1.10)	0.460(0.38/0.58)	

Table 2: Arithmetic test task with different types of the variable for each test (leftmost column)

Difference between performance of the optimized (hand-written) and non-optimized bytecode is very high. Hand-written bytecode delivers very high performance, up to the 80% of the optimized C code. Non-optimized bytecode runs runs approximately 2 – 3 times slower. There is no wonder in this since neither bytecodes compiler not JIT compiler do any optimizations. This paticular test is rather synthetic; it was chosen to demonstrate the best possible results for LJIT. On other tests even hand optimized bytecode will suffer from too little avaible integer-type registers.

Three other benchmarks test LJIT performance on more realistic compuational problems. No manual optimizations were made in these tests. Meaning of the numbers is the same as in the test above.

Test Task	LJIT	Kaffe	C	C –O3
Prime numbers (loops, integer arith.)	2.912	2.047(0.70)	1.642(0.56)	1.766(0.60)
Seive (arrays, integer arith.)	2.405	0.622(0.25)	1.073(0.44)	0.764(0.31)
Pi value approx (double arith.)	1.038	4.463(4.29)	0.758(0.73)	0.674(0.64)

Table 3: running time for various test tasks

LJIT is considerably slower in integer arithmetic. It suffers from the lack of any bytecode-level optimization, especially register allocation. In tests where register allocation can not give much speed up LJIT shows much better results.

4 Limitations

Templates-based JIT compilation uses considerably more memory then interpreter. Generally templates-compiled machine code will be 5-20 times larger then the bytecode.
For large programs compilation time may be unacceptable. This subject was extensively researched in connection with Java JIT compilers. Such techniques as compilation on request and background compilation can be easily applied to flatten the problem.

Templates-based JIT compiler implies certain limitations on the instruction set. For every instruction there should be a set of templates that correspond to all possible combinations of non-immediate arguments. In some cases number of templates needed to implement an instruction could be very large. For example RISC-style instruction

```
opcode reg, reg, reg
```

requires 32^3=32768 templates when reg is one of 32 registers. Run-time code generation systems [3] can solve this problem however at the cost of simplicity and portability.

5 Conclusion

Proposed JIT compiler may be used to bring the performance of the interpreted languages to the level comparable with the performance of the traditional static compilers. The most attractive features of the suggested technique are portability and implementation simplicity. Experience project shows that this approach can be potetially as fast as the current state-of-the-art JIT compilers.

6 References

[1]Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. SIGPLAN '98 Conference on Programming Language design and implementation. ACM Press, June 1998.

[2]Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. Proceedings of the ACM SIGPLAN '96 conference on Programming Language design and implementation. ACM Press, 1996.

[3]Francois Noel, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In Proceedings of the IEEE Computer Society International Conference on Computer Languages 1998. IEEE Computer Society Press, April 1998.