Advanced Compiler Construction: Advanced Project

Cédric Luthi
cedric.luthi@epfl.ch

Benoît Perroud
benoit.perroud@epfl.ch

June 26, 2006

Abstract

This report describes our work on the minischeme virtual machine and compiler. We have chosen to achieve JIT compilation and precise garbage collection for the advanced part of the project.

Contents

Chapter 1

JIT compilation

Just-in-time compilation (JIT compilation) is a technique used to improve the performance of bytecode-compiled programming systems, by translating bytecode into native machine code at run-time. In this project, JIT compilation has been performed on a whole program, i.e. a program is loaded into memory, then entirely translated to machine code and finally executed.

Here are the conventions used in this report:

- *vm* means virtual machine
- *native code* refers to code that has been dynamically generated for the underlying architecture
- *bytecode* refers to the minivm bytecode
- *opcode* refers to the minivm opcode scheme

GNU lightning[1], a multi-architecure library for dynamic code generation has been used as jit compiler.

1.1 Translating opcodes to native code

Translating most opcodes using GNU lightning was straightforward. For example the `ADD` opcode is easily translated as shown by this code:

[1] http://www.gnu.org/software/lightning/

```
case OP_ADD:
  // R[a] = R[b] + R[c];
  jit_load(JIT_R0, b)
  jit_load(JIT_R1, c)
  jit_addr_i(JIT_R2, JIT_R0, JIT_R1);
  jit_store(a, JIT_R2)
  break;
```

Some opcodes like `ALOC` and `DIV` for example required some more lines in order to call a function. GNU lightning provides an easy way to handle function calls, so implementing these opcodes was relatively easy too.

1.2 Handling labels

Special care is needed in handling labels. Labels are loaded as integers with the `LINT` opcode but unlike to integers, they represent a location in memory (a pointer). It is mandatory to distinguish between an integer and a pointer in order to adapt the `LINT` of labels. As we need this distinction for the precise garbage collection also, we have have implemented a tagging method, as explained in paragraph 2.1.

1.2.1 Jumps and the references table

As explained just above, labels represent an address in memory. Theses addresses are used to jump to different parts of the code. The problem to solve is to convert the addresses from *bytecode address space* to *native code address space*. The mapping between bytecode address space and native code address space is resolved during the jit compilation process. The `reference_t` struct is used to handle this mapping.

The last part of the problem is to patch the loading of an address. Obviously, the mapping can be resolved after an immediate move instruction has been compiled. GNU lightning offers the handy `jit_patch_movi` macro that changes the value of an immediate move after it has been compiled. So, each time we encounter a label address in a `LINT` opcode, we store a reference to it so that it can be patched at the end of jit compilation, when all mappings are resolved.

Once all references to labels have been patched, i.e. their value corresponds to native code address space, the `CMOV` instructions becomes trivial to implement: we just have to take care of compiling a jump instruction when `CMOV` destination register is R31.

1.3 Optimizations

Some small optimizations are possible to generate better native code. Two are described in this section.

1.3.1 R0 optimization

As the register R0 always contains the 0 value, it's useless to load the value from the R0 register. Instead, we generate an immediate move instruction with the value 0. For this purpose, the macro `jit_load_R0_opt` has been introduced

```
#define jit_load_std(r_dst, r_src)
            jit_ldi_i((r_dst), &R[(r_src)]);

#define jit_load_R0_opt(r_dst, r_src)           \
        if ((r_src) == 0) {                     \
                jit_movi_i((r_dst), 0);         \
        } else {                                \
                jit_load_std((r_dst), (r_src)) \
        }
```

Now, each time R0 is used, the generated instruction is

```
xorl %eax,%eax
```

that zeroes the %eax register. This is faster than the

```
movl 0x1000,%eax
```

instruction who accesses memory at R[0] (assuming &R[0] = 0x1000).

1.3.2 R29 optimization

As suggested, it would be beneficial to map some vm registers to native registers. This is the case already for the program counter which is now the real PC of the machine. Mapping the frame pointer (R29) to a real register would be beneficial for performance as this register is used a lot. GNU lightning exposes three registers that are preserved across function

call. `JIT_V2` is one of those and has been chosen to be the FP. The same idea as for the R0 optimization has been implemented:

```
#define jit_store_opt(r_dst, r_src)                \
        if ((r_dst) == 29) {                       \
                jit_movr_i(JIT_V2, (r_src));       \
        } else {                                   \
                jit_store_std((r_dst), (r_src))    \
        }

#define jit_load_opt(r_dst, r_src)                   \
        if ((r_src) == 29) {                         \
                jit_movr_i((r_dst), JIT_V2);         \
        } else {                                     \
                jit_load_R0_opt((r_dst), (r_src))    \
        }
```

While this technique worked in simple programs, it has failed with a *Memory access outside of heap segment* error on a more complex program that computes large factorials (`bignums.asm`). Unfortunately we have not been able to track where the error came from, thus these two optimized macros are defined but not used. The reasons are explained in detail in paragraph 1.4.

1.4 Difficulties

Achieving the JIT compilation was not an easy task. The main reason is that the generated jit code is hard to debug. GNU lightning jit_ macros are so obscure that it is impossible to guess what code is going to be generated. On small programs, it is possible to identify what native code corresponds to bytecode while on bigger programs, it becomes almost impossible.

The *R29 optimization* problem perfectly illustrates the inability to debug dynamically compiled code. To determine where the problem happened, we added some code to increment a counter after each execution of an opcode. It turned out that the problem occurred after the execution of more than 80'000 opcodes! Given the lack of function concept in the vm, there was no backtrace in the debugger, which resulted in a so hard and time consuming task that we preferred to give up on this optimization.

1.4.1 Mac OS X ABI

Another difficulty was to fix crashes that seemed rather random under Mac OS X running on an Intel chip. When calling functions, for example `exit_vm` from dynamically compiled code, the vm would crash with an `EXC_BAD_INSTRUCTION` exception on a `movdqa %xmm0,32(%esp)` instruction.

It turned out that the Mac OS X ABI Function Call Guide[2] for IA-32 specifies that **the stack must be 16-byte aligned at the point of function calls**, a convention that GNU lightning was not aware of. This has been corrected in GNU lightning i386 part and a patch has been submitted to Paolo Bonzini, author of GNU lightning. Another solution discovered later is simply to use the `-mstackrealign` switch of gcc that takes care of realigning the runtime stack. The former solution has been finally chosen as the latter solution is less efficient.

1.4.2 SPARC

While our virtual machine has been thoroughly tested on different x86 machines and on PowerPC, testing on SPARC has been reduced to the minimum. Launching a simple *Hello, world!* did not work on SPARC: it exited with a *Memory access outside of heap segment* error. Lacking of time and of SPARC assembly knowledge, we have not investigated this problem further and have decided to simply drop support for SPARC.

[2] http://developer.apple.com/documentation/DeveloperTools/Conceptual/LowLevelABI/index.html

1.5 JIT compilation performances

Once JIT compilation was achieved, some performance comparisons have been done on a program that does intensive calculation: computing the factorial of a large number.

The results are very impressive. Here is a table summarizing the execution times.

	standard	threaded	jit
factorial(1200) on x86 (3Ghz)	7.150s	7.250s	1.160s
factorial(600) on x86 (3Ghz)	1.580s	1.610s	0.260s
factorial(1200) on ppc (667 Mhz)	25.532s	25.736s	6.801s
factorial(600) on ppc (667 Mhz)	5.646s	5.692s	1.491s

We notice that jit compiled code runs a lot faster, from 4 times on PowerPC to 6 times on x86. We also observe that threaded mode is not as efficient as one would expect. On smaller program, it's not even worth jit compiling since most of the time is spent in compilation instead of running the program.

Chapter 2

Precise copying garbage collection

A copying garbage collector, called *gc* in this report, is another beautiful technique to manage dynamic memory deallocation in a *vm*. But to run such a *gc*, one needs to make it *precise*, which means to distiguish clearly integers from pointers. That's why this chapter is divided in two parts, one about how to make the *vm* precise and one about the implementation of the *gc* itself.

2.1 Differenciating integers from pointers

The usual way to distinguish clearly integers from pointers is to use a tagging scheme. The one we chose is to represent an integer n as 2*n+1 in the bytecode, and to align every pointer on 4 bytes[1]. Thus the least significant bit of pointers is always 0, and that of integers is always 1.

Our first implementation to differenciate pointers was all integrated in the *vm*. The changes of the operators were straighforward and because the parser of the *vm* has the knowledge of what is a label and what is an integer, we marked a bit of the load of label instruction ($LINT$)to recognize it at run time. Altough this implementation was faster than the final one, because it avoided lots of instructions provided to correct the 2n+1 arithmetic by the compiler, some drawbacks come into the light :

- We need to modify every operation of the *vm*, and this in each part : normal, threaded and jited.

[1]Alignment on 2 bytes would also be correct, but for compatibility with our old *mark & sweep gc* we maintain it on 4 bytes

- Optimisations which could be done in 2n+1 form aren't possible at run time

Thus we handled the tagging directly in the compiler and do just the minimal changes in the *vm*. These transformations are explained below.

2.1.1 Changes in the compiler

The main idea in the compiler is to represent every integer as an odd value, encoded with 2n+1 scheme. So in the vm, there will never be any even integer neither in the register nor in memory. But there are a few exceptions where even integers are produced by the compiler :

- The integer 2 used to represent any integers as 2n+1.
- The offset of the LOAD and STOR instructions.
- Some intermediate value of the transformation in 2n+1.
- The false value is 0 in the *vm* and 1 in the compiler.

The compiler is modified in the way to handle integers as 2n+1, by modifying the class IntegerConstant to print 2 * *value* + 1 instead of simply *value*, and then some changes for the exceptions listed above are made.

We introduced a new class *PreciseIntegerConstant*, which does not transform integers in 2n+1.

Instruction.scala

```
case class IntegerConstant(value: Int) extends Constant {
  override def toString(): String = (2 * value + 1).toString();
}
case class PreciseIntegerConstant(value: Int) extends Constant {
  override def toString(): String = value.toString();
}
```

LOAD and STOR instructions generation is modified to use *PreciseIntegerConstant* for their offsets

Code.scala

```
def emit(op: OpcodeRRC, r1: Register, r2: Register, i: Int): Unit =
  emit(new InstructionRRC(op, r1, r2, PreciseIntegerConstant(i)));
```

Last but not least, we generated correction code for the arithmetic and the logical operations on integers. The 2n+1 arithmetic is given below, as well as the modification done for the multiplication.

n and m are the two real integers, f and g are their representation in 2n+1 form. Thus if the program want to multiply n to m, it expect the result n*m, which encoded is 2*(n*m)+1. As the compiler uses f and g, it must substract 1 to f and g, multiply them, divide the result by 2 and finally add 1.

```
2 * ( n + m ) + 1 = f + g - 1
2 * ( n - m ) + 1 = f - g + 1
2 * ( n * m ) + 1 = (( f - 1) * (g - 1)) / 2 + 1
2 * ( n / m ) + 1 = (( f - 1) / (g - 1)) * 2 + 1
2 * ( n % m ) + 1 = (( f - 1) % (g - 1)) + 1
```

Every time PreciseIntegerConstant is used in order to perform operations on real 1 and 2 instead of 3 and 5. So we load the value 2 in a register, which can be potentially be seen as a pointer by the *vm*. So we load it in *targetReg* to be sure that this value is overwritten by a valid one. Finally we load 1 in *yReg*, to also be sure to have a valid value in it, because its previous value can also potentially be even.

```
Generator.scala
  code withFreshRegister { lintReg =>
    code.emit(new InstructionRC(LINT, lintReg,
        PreciseIntegerConstant(1)));
    code.emit(SUB, targetReg, targetReg, lintReg);
    code.emit(SUB, yReg, yReg, lintReg);
    code.emit(MUL, yReg, targetReg, yReg);
    code.emit(new InstructionRC(LINT, targetReg,
        PreciseIntegerConstant(2)));
    code.emit(DIV, targetReg, yReg, targetReg);
    code.emit(ADD, targetReg, targetReg, lintReg);
    code.emit(new InstructionRC(LINT, yReg,
        PreciseIntegerConstant(1)));
  }
```

The logical operations shouldn't be modified because they compare two tagged integers so the result would be correct. But the *if* logical structure should be modified because the result of the condition is tagged, but the CMOV instruction used to perform the jump compares the value with a real

0. What we do is the following : we compute the condition, which is 1 if it is false, 3 or greater otherwise and we compare it with a tagged 0. At this point we have the inverse of the right answer, 0 if true and 1 if false. We then generate another comparison with the register R0, which is always 0. The final result is then 0 or 1, and can be interpreted correctly by the CMOV.

```
Generator.scala
  code withFreshRegister { lintReg =>
    code.emit(new InstructionRC(LINT, lintReg,
        IntegerConstant(0)));
    code.emit(ISEQ, targetReg, targetReg, lintReg);
    code.emit(ISEQ, targetReg, targetReg, code.R0);
  }
```

2.1.2 Changes in the virtual machines

The more interessting changes performed in the *vm* are in the two functions

```
int is_heap_pointer(int addr);
int is_code_pointer(int addr);
```

These functions test the least significant bit of *addr* in order to determine precisely if it is a heap or code pointer. Their results can be fully trusted. This accuracy is also valid for the *mark & sweep gc*, in which the others tests we added to remove as much conservativeness as possible, are now obsolete.

The only instructions we need to adapt are OP_ALOC, OP_PINT and OP_PCHR where integers should be untagged, as well as OP_RINT and OP_RCHR where integers should be tagged. These few modifications were done in the normal execution mode as well as in the threaded and the jitted one.

2.2 Copying grabage collector

The copying *gc* has its memory space divided into two part, *from_space* and *to_space*. Moreover it has a variable *space_size* which stores the size of each space. The goal is to allocate linearly blocks in *from_space* until it is full, then copy every living block (*i.e.* reachable directly or indirectly from a register) to *to_space*, and finaly invert the role of *from_space* and *to_space*.

2.2.1 Allocation

The allocation is done in *from_space* in a linear way. The pointer *next_free* indicate the end of the allocated zone, and is increased from the space of the allocated block. This is very efficient.

A header containing the size of the allocated block is added at the begining of each allocated block.

2.2.2 Copying phase

When *from_space* is full, we run the *copy* procedure. It will iterate through every registers (except R0) to find every heap pointer. If a heap pointer is found, the pointed block will be copied to a new address in *to_space* and its new address will be returned. The *copy to new address* procedure is described below using pseudo-code.

- Check if the block has already been moved
- If it is the case, retrieve the forwarding pointer and return it.
- Otherwise copy the block to *to_space* + *next_free*
- Tag the size of the block with a mask to note it as already copied
- Overwrite the first 4 bytes after the headers of the block with the forwarding pointer[2]
- Return the new address.

After every block reachable from a register, the *copy* procedure will iterate through the copied block using Cheney's algorithm, in order to find indirect living blocks, copy them and update the pointers references. Cheney's algorithm is used because it adds very little overhead to visite every block. The *copy* procedure will end when the *scan* pointer will reach the *next_free* one, meaning that every block has been visited. The *copy* procedure will finally invert *from_space* and *to_space*, and the allocation can continue, except if the procedure didn't free enough place.

[2]Admitted because the overwritten value are never read again

2.3 Performance and others results

The performance improvment of the copying gc is quite impressive. This improvment comes from the linear allocation and the fact that it doesn't scan the whole heap memory like *mark & sweep gc*, but only the living objects. The following table shows computation times of a factorial program launched without *gc*, with *mark & sweep gc* and with *copying gc*. The heap size is the default one, except without gc, which requierd more than 100MB of heap size.

	none	ms	copy
factorial(1200)	7.128s	13.625s	7.220s
factorial(600)	1.624s	2.340s	1.720s

We also do some tests to see which amount of memory is freed every time the *gc* runs. The results are given with the default heap size and the computation of factorial(1200)

	range of % freed	average %
mark & sweep	49 - 89%	84%
copying	83 - 94%	91%

The difference in the results between the two *gc* comes mainly from the fragmentation generated by the free list, and the overhead of our implementation which sometimes allocate a bigger block than the one requested if the remaining free block is to small. This was done in order to avoid very small blocks of memory to be lost forever.

2.4 Drawback and further work

The main drawback of such precise *gc* is the reduction of the integer range, but it can be avoided using others techniques, like using pointers to integers.

We also remark that in minischeme, the closures are persistant objects stored in the global frame pointer, so they shouldn't be copied from left to right at every cleaning. This statement is the reason why *generational gc* was developed.

Chapter 3

Previous parts

Implementing the two advanced parts, we remarked that some bugs remained in the previous parts. We spent time to correct them in order to achieve a completely running project.

- Threaded code : we corrected the disassembling
- The *mark & sweep gc* became precise with the advanced part, so it is not wrong anymore to store marks in the headers. It has also be cleaned from its obsolete checks
- Closure contained a small bug which is now corrected