Advanced Compiler Construction Project

Just-in-Time Compiler

Marco Schmalz & Sébastien Cevey

24th June 2006

1 Introduction

We chose the Just-in-time compiler mini-project because we were interested in working on a modern runtime problem. The GNU lightning library dramatically reduced the work needed to produce native code, while providing an interface to make it compatible with several platforms.

We decided to work on both phases (ahead-of-time and hotspot versions), but early on we planned to realize phase 1 as a step leading to phase 2, to avoid extra-work in the short timeframe we had been given. The phase 1 is available by running the `minivm` with the flag `--jit ahead`, while phase 2 can be enabled using `--jit hotspot`.

The consequence of this is that the result of phase 1 is not as optimal as it could have been had we focused on it separately. For instance, we would not have needed to generate the v-code[1] instructions; we could directly have produced the whole program as n-code[2].

In short, it is important to keep that we developped this mini-project iteratively. Nevertheless, we think the result is very satisfying and meets our expectations.

2 Ahead-of-Time JIT Compilation

The goal of the first phase of the project was to dynamically translate *all* the virtual machine code to native machine code, before starting the execution of the generated code. N-code is generated for each virtual machine instruction by linearly iterating through the existing v-code.

The translation for most instructions is straightforward with the help of the GNU lightning library. Before executing the desired instruction (e.g. `ADD`), each argument value is loaded from the virtual register into a physical

[1] Virtual machine code.

[2] Native code, as generated by GNU lightning.

register and afterwards the result is stored back to the virtual `minivm` register in memory.

For the I/O instructions (`RINT, PINT, RCHR, PCHR`) and the `ALOC` instruction, function calls are generated to call for example the `printf` function.

Labels and references to labels need to be treated specially. Before compilation, a *lable table* containing all the labels in the program is created. An entry of this label table stores the label's v-code and n-code address and a list of pointers to n-code instructions that use the current label, called the use-list.

`LINT` instructions are the only instructions that load code pointers as immediate values. When translating a `LINT` instruction, where the value to load is actually a pointer to location in the v-code referenced by a lable, the label's n-code address is looked up and inserted as the immediate value. If the n-code address is not yet known, as this is the case for forward jumps, a reference to the currently generated instruction is stored in the label's use-list.

When a label is encountered during the translation, the current n-code address is stored and the instructions referenced in the use-list are patched with the label's n-code address.

`CMOV` instructions with the program counter (`R31`) as destination register are translated as branch instructions that load the branch target from the specified `minivm` register.

Problems Encountered

- The GNU lightning code generation library is implementad as a huge set of C macros. Because no typechecking is performed when using macros, type errors might not be detected during compilation. For example the use of an immediate value instead of a register will not yield a compilation warning. Therefore a programmer should be very careful when using this library to prevent small errors that are difficult to detect afterwards.

- Branch instruction to register targets are not supported in GNU lightning. This problem can be circumvented by using a combination of branch instruction and a jump instruction to a register target.

- One bug that took us quite some time to solve was that we forgot that there might two consecutive labels at one single code location, as it appeared only in the `bignums.scm` program. Once the problem source was located the solution was of course trivial.

- We know from another group that undetected code buffer overflows introduce strange bugs when they remain undetected. To avoid any

trouble we test the code buffer boundaries after every generated instruction.

3 Hotspot JIT Compilation

The second phase of the project was to implement a more realistic JIT and only compile parts of the code that are *hot* (i.e. frequently executed). We decided to use functions as the granularity of JIT compilation. A function starts and ends at a label whose name starts with "lambda". A function is considered to be *hot* after it has been executed a given number of times. A counter assigned to every function is incremented at every execution until it has reached the threshold value. At that moment, the function is compiled to n-code and from this point on, only the compiled version of the function is executed. A major problem was to handle jumps correctly, especially in the case when we jump from n-code to either n-code or v-code. Our first implementation is quite simplistic and treated all jumps uniformly. It introduces unnecessary overhead when jumping from n-code to n-code, that is removed in a second implementation.

The threshold value can be changed via the `-jit-threshold` command line argument.

Simple Implementation

In the first implementation all label adresses, whether in v-code or in n-code, are replaced by indexes to the label table. When a `CMOV` is interpreted, the target label is looked up in the label table. If the code has not yet been compiled, the counter for this label is incremented and if the counter reaches the threshold, the code is compiled and the n-code address of the compiled code stored in the label table. Depending on whether the code has been compiled the execution will continue in interpreted mode or will jump to the compiled native code.

A call to n-code will return the index in the label table of the next jump target. This means that every `CMOV` in the compiled code will return the index of the target label to the interpreter. The interpreter will take care of the jump and return to native code if possible or alternatively continue in interpreted mode. Not only the translation of the `CMOV` instruction is relatively simple, but also the handling of labels is straightforward, as we just need to store the n-code address in the corresponding entry of the label table. Finally the translation `LINT` instructions boils down to loading the value of the target label index into a `minivm` register.

Optimized Implementation

Our first implementation introduces an overhead when jumping from native code to native code. Instead of returning the index of the next jump to the interpreter and letting the interpreter do the dispatching, an optimal solution would jump directly to the target n-code address.

The key idea is to distinguish between indexes of the label table and pointers to compiled native code. As we know that label indexes are always smaller then the total number of labels in the program, we can assume that all values that are greater than the total number of labels must be code pointers. Under Linux for example the first 128 MiB of the virtual address space are never used which means that we can use at least 128 millions labels before pointing to a possibly valid address in the heap. Even if this is not the case, it is extremely unlikely to run into trouble, because the label table is allocated before the native code buffer. In any case a simple test that compares the the pointer value of the begining of the n-code buffer with the total number of labels assures that our assumption holds.

When translating a `LINT` instruction, a patchable `movi` instruction[3] is generated and patched with the label index of target label. A reference to the generated instruction is stored in the target label's use-list. If later on during execution the code containing the target label is compiled, the immediate value of the `movi` instruction is re-patched with the correct n-code address. If the target n-code address existed already when translating the `LINT` instruction, it is of course used directly.

The compiled version of the `CMOV` instruction will test if the target value is an index to the label table, in which case it will return the value to the interpreter, or alternatively it will jump directly to the specified n-code address. This method reduces the overhead introduced for jumps within native code to one single conditional branch instruction.

Problems Encountered

- A problem encountered was, that code pointers to native code were casted to function pointers and used as such. As the called code was not always the beginning of a function, and thus did not execute the function prolog[4], a return instruction directly returned to the main function. To avoid this problem we have created a small function at the beginning of the native code buffer that takes a pointer as an argument and then jumps to the desired location.

[3] move an immediate value to a register.

[4] setting up the stack, saving the return register, stack frame pointer etc.

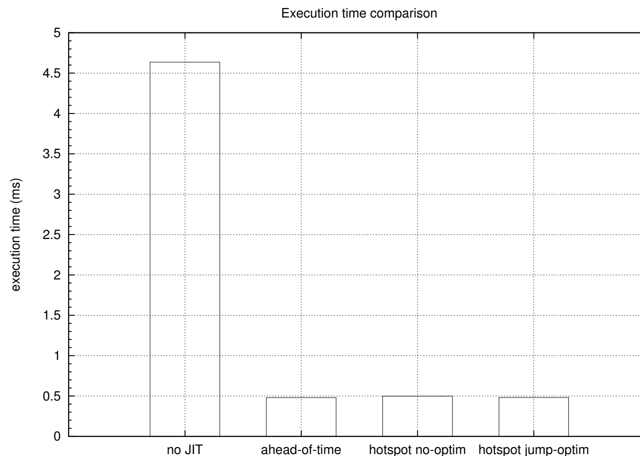

Figure 1: Execution time comparison

4 Results

We ran some benchmarks on the different versions of the virtual machine. In each case, the `bignums.scm` programs was used to compute the factorial of 300. The debugging output was disabled and we enabled the mark-and-sweep garbage collector. The virtual machine was compiled with `gcc-3.4.5` without code optimization (`-O0`) so as to be fair with the generated n-code which could not be optimized. The resulting value is the average of execution time spent in userspace[5] over 20 iterations.

The results are shown on Figure 1. We see that the Just-in-time tests are about 9 times faster than the intepreted execution.

The best performance is (unsurprisingly) achieved by the ahead-of-time JIT. However, this is not always an option in the case of a large program, which one might not want to fully compile to native code on each execution. For very small programs, we also noticed that the overhead of the compilation exceeded the time needed to simply interpret the program. This solution is therefore a mostly theoretical one, not very adapted to real-life usage and especially optimal for medium-size programs.

One positive result, however, was the performance of the hotspot JIT (with jump optimizations), as it comes very close to that of the ahead-of-time version in spite of the overhead of online compilation (with the overhead of context switches, additional runtime tests, etc). However, as noted in the introduction, the ahead-of-time JIT could still be optimized and maybe it would then perform better than the hotspot version.

[5]as reported by the `user` entry of the command `time(1)`.

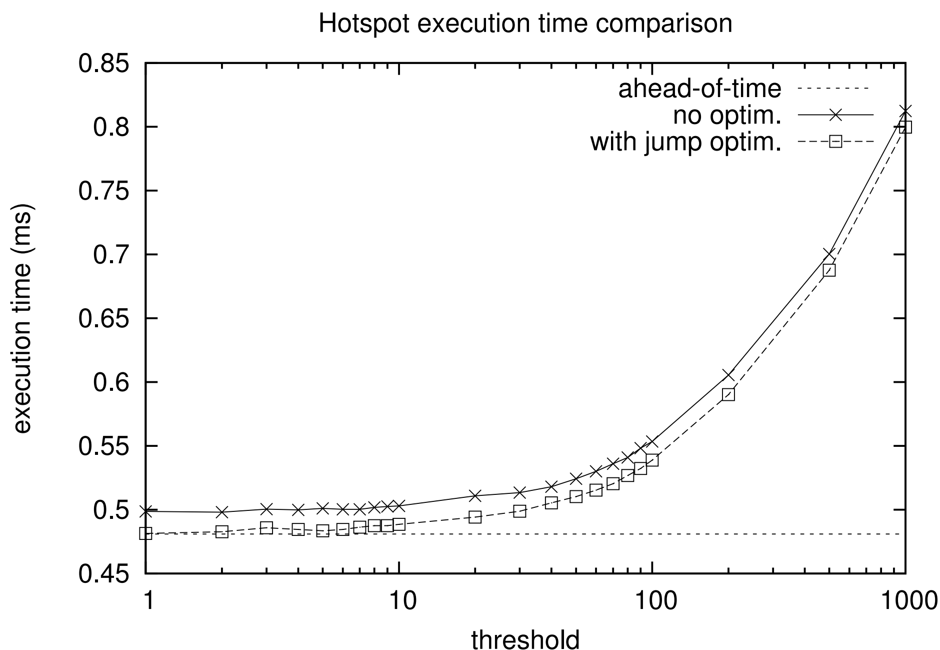

Figure 2: Hotspot execution time comparison

As can be seen on Figure 2, we tested a range of values for the hotspot threshold[6]. The results are more or less equivalent for a threshold between 1 and 10 (probably because the number of time each function is executed is much greater than 10 and thus the difference is neglictible). The execution time then increases as we raise the threshold, because less and less code is compiled (or later in the execution).

We can conclude that the optimal value for the threshold is towards the lower end (e.g. `1`). However, to assess this remark, more tests should be performed on programs of various sizes (both very large and very small), including programs where there is a great disparity of execution between different functions in the program. We did not spend too much time doing this because it was not in the scope of this mini-project.

The difference of execution time between the hotspot JIT with or without jump optimization is also quite visible on the figure (see both curves). The performance improvement is not neglictible as it makes the hotspot JIT come closer to the ahead-of-time JIT. However, please note that the scale of the Y-axis does not start at zero at the origin.

[6]The threshold represents the number of time a v-code function is visited before it is compiled to n-code.

5 Comments

Because of time constraints, we did not have time to implement all the optimizations we could think of. Here are two points which could be improved:

Bind virtual registers to processor registers. In our current JIT implementation, the content of virtual registers used by the bytecode has to be loaded from memory (the registers are stored on the stack of the interpreter) into one processor register whenever an instruction uses their value.

A more efficient solution would be to perform clever register allocation, by binding some virtual registers to processor registers, either on a global basis (by identifying which registers are most used), or on a local basis, e.g. per function.

Processor registers would still have to be saved to memory at the end of the function (in case we change context into v-code after returning), but the improvement of performance would still be significant (in terms of number of instructions and memory accesses).

Allocate blocks for n-code dynamically. Instead of allocating one big fixed block for all the n-code, of a size proportional to the size of the v-code, one could imagine a less greedy approach to memory allocation: allocate (small) fixed-size blocks of memory to store the n-code. When reaching the end of a block, simply allocate a new block and insert a jump to its start.

This requires to pay attention to properly identifying the end of blocks while writing instructions (and keep enough space for a jump), but it would help port the JIT on different architectures (RISC or CISC) where the v-code/n-code ratio could vary quite a lot depending on the available instruction set. Less memory would thus be lost by over-allocation, and the computing overhead would still be rather small.