Anelise Newman, Kimberli Zhong, Stef Ren
5/16/2018

6.s081 Final Report

Introduction

For assignment 5, we successfully implemented code generation in assembly (our unoptimized code can be found on our `a5-milestone` branch). We then sped up our code by adding register allocation (`a5-submission`).

Choosing Optimizations

In our original assembly implementation, all of our local variables, including temporaries, were stored on the stack at a prescribed offset from `%rbp`. Loading and storing temporaries involved an expensive process of calculating an offset, storing the offset in a register, and then loading the appropriate object from memory into the register. This overhead was incurred for almost every operation. Thus, it was important that we implement a working register allocator to speed up our compiler.

Another optimization we included was storing our assembly functions once we compiled them down to an optimized representation. In our milestone, we compiled the function every time we were going to run it, which was useful for debugging but ended up being expensive and redundant. This was an almost trivial change to make and led to a big speedup.

We also noticed that by far the largest part of our assembly instructions were spent entering and returning from calls to C++ helpers. To limit the amount of function calls we had, we attempted to implement tagged pointers, where integers, booleans, and short strings were implemented as 64-bit numbers with a 2-bit tag in the highest bits for easy type-checking and unboxing. We successfully converted to the tagged pointer data type, but we were not able to it working with our assembly code in time. The version of our code that uses tagged pointers can be found at `tagptrs_vm` and can be run without the`--op=all` flag to execute in our VM.

Implementation

We decided to move from bytecode → IR → assembly. We found that having multiple intermediate representations was easier than trying to jump directly from BC → assembly or our AST → IR and was easier to debug. Our bytecode compiler lets us handle problems like how to flatten out our function structure and statically resolve the scope of variables. Our IR gives us a way to deal with memory in a more fine-grained way by introducing the idea of temporary variables that can be assigned to a variety of positions in registers or in the stack. Our IR→ assembly compiler deals with the details of resolving where different variables are stored and implementing a rigorous calling convention.

IR representation

The basic unit of our IR is an `IrFunc`, which is pretty similar to a Bytecode function in that it contains a list of `Constant` objects, a list of bytecode `Functions`, and some other metadata about the function like the number of arguments and local variables. The `IrFunc` code is specified as a list of IR instructions and a list of `Temp` objects corresponding to the temporary variables referenced in the function.

An `IrInstruction` has the following format:

- An operation `op`, taken from an enumeration of `IrOps`
- An optional integer `operand0` (for instance, to specify the index of a `Constant` for `LoadConstant`)
- An optional string `name` (for instance, to specify a record field for `FieldLoad`)
- A list of `Temps` that specify the operands and outputs of the operation

- A `Temp` is an object used to represent a temporary variable and its metadata. Initially, a `Temp` was little more than a unique index, but as we made optimizations we included other information like the `Temp`'s live interval and suggestions for where to store it. The important fields of a `Temp` are:
    - A unique index
    - An optional `x64asm` register, if this `Temp` should live in a register
    - An optional `stackOffset`, indicating an index into the stack, if the `Temp` should be stored in memory

This representation is flexible enough to accommodate all of our desired IR instructions.

Some of our IR instructions are very similar to bytecode instructions, but we did make some changes to our instructions. We added explicit instructions for type checking, boxing, and unboxing (for example, `AssertInteger`, `NewInteger`, and `UnboxInteger`). We added an explicit `GarbageCollect` instruction that would invoke our garbage collection using a C++ helper function. We also found that as we transitioned from a stack-based to a temp-based memory model, some of the bytecode instructions became useless--for instance, `LoadLocal` corresponded to a meaningless transfer between two temps or essentially a no-op, so we ended up removing it. We also changed `PushRef` to `PushFreeRef` to indicate a distinction between how we treat local ref variables and free variables.

Our IR representation is documented in our `ir.h` file.

Moving from Bytecode to IR

In order to move from the stack-based model of memory that we had in our bytecode to the temp-based model of our IR, we use a symbolic temp stack to mirror the execution of the bytecode program. Every time we would have pushed something to the bytecode stack, we create a temp with a unique integer ID (we generated an ID simply by incrementing a counter) and push that temp to our symbolic stack. When we need to pass arguments to an operation, we do so by popping elements from the stack. This gives us an easy way of tracking the lifetime of a value and where the operands to functions are stored.

We also made sure to explicitly encode important assumptions and checks on types by generating the appropriate IR instructions—for example, including `AssertInteger` for operands before performing an arithmetic operation.

The files involved in translating bytecode to IR are in the `ir/` folder, particularly in the file `bc_to_ir.cpp`.

Bytecode Changes

While translating our IR to assembly, we found that some actions could be more easily achieved by modifying our bytecode. For instance, in assembly we found it more convenient to use labels and jumps to conduct while loops and if/else statements than using offsets, so we added a `Label` instruction to our bytecode, as well as a `labels_` map mapping label indices to instruction indices in the resulting bytecode function. We also added no-op operations `StartWhile` and `EndWhile` which served as convenient annotations for determining live ranges for register allocation.

Moving from IR to Assembly

For our milestone, our assembly stack had a dedicated slot for every local variable, a pointer to an array of free variables, and a slot for every temp variable. In our optimized code, we keep the pointer to the array of free variables, but we use the convention of representing our local variables also as temps, with our *n* locals being stored as the first *n* temps. This lets us have a dedicated space for each local variable while also letting us use the same location metadata we used for our temps for our locals as well.

In our unoptimized code, we accessed temps and locals by loading the variable from its stack slot into a register, performing an operation on it, and returning it to the stack. In our optimized code, each temp contains a hint of either a register or a stack index where it should be stored (only variables that do not fit in registers go on the stack—see the Register Allocation section). When accessing a variable, we first check its `Temp` metadata to find its location. If it is already present in a register, we can operate on it directly. Otherwise, depending on the operation we wish to perform, we use an assembly instruction that can take an item in memory as an operand or we load the variable into a “scratch” register that we free by pushing a register to the stack temporarily and restoring it when we are done using it.

One of the more challenging aspects of our assembly generation was the calling convention and the transitions between running code in our VM and running compiled assembly. In our bytecode interpreter, we have a function `call` that takes in a list of arguments and a closure that can branch to either run bytecode in our Assignment 4 interpreter or optimized assembly code. If the `shouldCallAsm` flag is set in our interpreter, our IR compiler, IR optimizations, and assembly compilers run to generate an `x64asm` function. We use the provided `MachineCodeFunction` class to perform the pre-call procedure to break into assembly. In our IR → assembly compiler, we define `prolog` and `epilog` functions that implement storing/restoring the callee-save registers and moving arguments/return values to their appropriate places based on our temp hints. We also have a `callHelper` function that handles pre-call and post-call procedures for helpers implemented in C++; specifically, it handles storing/restoring caller-save registers and moving arguments to the correct registers. When we create our assembly compiler, we pass it a pointer to our VM interpreter, which can then be passed into helper functions for use in allocating new objects, accessing global variables, or transferring control flow back to the VM to dispatch to another MITScript function.

Our `asm/` folder contains our code for translating from IR → assembly. The files `asm/ir_to_asm.cpp` and `asm/asm_helpers.cpp` contain our assembly compiler, while `asm/helpers.cpp` contains helper functions implemented in C++ that we call from assembly.

Register Allocation

We implemented the linear scan algorithm from [this paper](#) (Linear Scan Register Allocation). We analyzed the live ranges of locals and temps by marking the defs and uses of each variable while stepping through the IR to generate the assembly. Each variable and temps stores a startInterval and endInterval for its live range.

Each temp is stored in only one place at all times in the assembly, so each has a optional reg and optional stack field. Only one of these should be populated. Whenever the assembly code is accessing a specific temp or variable, it looks at the hint provided by the assembly to check whether it is in a register or a specific stack location. We left one register, R10, always unassigned so that we could use it for scratch space. We allocated 13 other registers as possible (rdi, rsi, rax, rbx, rcx, rdx, r8, r9, r11, r12, r13, r14, r15).

We assigned registers by the following algorithm (from the above linked paper). We consider the ranges in sorted order by startInterval. We free registers if we've moved past the end of currently assigned ranges, and assign registers to the new ranges as long as there are free registers. If there are no more free registers, then we either choose to spill the newest range or the second newest range (that has just been assigned on the previous round). This heuristic minimizes the number of registers spilled, but doesn't consider number of times used or any other heuristics. Then in the assembly code, we can allocate the exact number of temp slots on the stack that are needed.

We tried to implement different sets of ranges for locals, so that they could be stored in different places throughout a program. This is currently close to working, but needs to have defs and uses separated more clearly -- the work-in-progress is on the "better_reg_alloc" branch.

Extras

Testing scripts:
We have three different kinds of automated tests:

- Correctness tests in `tests/vm/` that run against both staff-provided tests and tests we wrote ourselves
- Memory usage tests in `tests/gc/` (does not work properly with generated assembly, though)
- Performance tests in tests/perf/ that run against performance-intensive tests with user-provided inputs (to minimize the impact of branch prediction)

Debugging techniques: The asm and reg layouts in gdb were very useful for stepping through our generated assembly code and checking register values. For instance, while debugging our calling convention, we could check the value of a register before and after the call to ensure that it had not changed. Another useful technique was setting breakpoints inside our C++ helpers to check the value of variables at certain points in the program. We also made use of assertions inside our assembly compiler to check that we were maintaining certain invariants (for instance, the stack should always be the same size before and after a single IR instruction is run).

- **Tagged pointers data type:** Although we were not able to integrate it with our assembly, we were able to implement booleans, integers, and short strings as tagged pointers. We stored our values as 64-bit integers, where the highest two bits identified the object as an integer, a boolean, a short string, or a pointer to another object (like a `Record`, `Function`, or `ValWrapper`). This data type and helpers for manipulating it can be found on the branch `tagptrs_vm` in the `op`/ folder.

Bugs/Challenges

Here is an overview of some of the more challenging bugs we encountered:

- **Division:** because the division assembly instruction requires specific registers, it took a significant amount of debugging to get this to work with temps in arbitrary positions (either the stack or a reg). Because this also specifically required 32-bit registers instead of full 64-bit registers, we had to add a lot of infrastructure to handle moving around 32-bit registers and temps.
- **Getting and releasing scratch registers**: for some IR instructions, we needed register space even if all temps involved were stored in memory. We addressed this by temporarily stashing a register to use for scratch space and then popping it when done. We had a problem implementing the Call IR instruction where we would push arguments to the stack using a scratch register as a trampoline, and then we tried to release the temp register we ended up popping one of the arguments off the stack by accident, which lead to garbage in our registers.
- **Debugging different temp locations:** tracking down bugs in our register allocation implementation was difficult because bugs could appear based on which register a variable

was assigned to. We tested our implementation with different subsets of registers invalidated to check that it worked with different assignments of temps to registers/stack locations.

- **Live ranges for while loops:** We had to add bytecode instructions `StartWhile` and `EndWhile` in order to indicate the ranges of loops. This was because the live range of a variable could be after its last use, if it was inside a while loop.

Code Base Layout

See the `README.md` in our base directory for an overview of the different folders.

Contributions

In terms of optimizations, Stef worked mainly on implementing the register allocation algorithm and generating register suggestions, Anelise worked on changing the assembly compiler to use register suggestions, and Kim implemented tagged pointers. We all contributed heavily to debugging the code (both during the milestone and optimization phases).