6.818 P5: Performance Optimization

Team RockETH

December 2021

Compilation

We first parse the MITScript code using the same parser of the previous phases. Then, we iterate over the resulting AST tree to produce an intermediate representation of the program having the following design choices:

- The program consist of a collection of functions. Each of them is divided into a sequence of blocks. These stores a list of successors and predecessors. We can hence say that our code is actually stored as a control flow graph.
- Each also block stores instructions in static single assignment form. This means that we use virtual registers to store intermediate values, each of these registers can be assigned only once. Moreover, every block also stores phi nodes. These are needed for if and while statements.
- The variable resolution works similarly to the previous phase. One minor modification was to assign to each global variables an index, rather then using strings to identify them.
- We decided to use only one vector of constants/field names for the whole program instead of using one vector of constants/field names for each function scope. Also, all constants/field name in these vector are guaranteed to be unique to ensure space efficiency.
- The program also contains at the end of each loop block and before each return in a function an instruction named `GC`. Every time we encounter these instruction when executing the program, we invoke the garbage collector which will be described later on.
- We also decided to include asserts statements in our intermediate representation that check the type of the values contained in virtual registers. We use this assert statements to check the type of the operands for all binary/unary operations except for `ADD` (the behaviour of the operations changes here depending on whether we operate on integers or strings). We decided for this design choice since in many cases as we will see it is possible to remove these asserts statements. Moreover, the resulting binary/unary operations of the intermediate representation are closer to the actual corresponding machine instruction. This also simplifies the generation of the machine code instructions.

All instruction used in the program can be found in the project proposal. We will not describe their semantics here as they have a rather obvious behaviour which can be inferred from their names. The compiler code can be found in `compiler.cpp`. As in the previous project, variable resolution is done using the visitor classes `Assigns.h`, `FreeVariables.h` and `Globals.h`.

Optimization

The SSA form of our intermediate representation allowed us to implement without too much effort the following optimizations:

- **Constant propagation**: This optimization can be enabled by setting `--opt=constant-prop` as flag. The code can be found in `const_propagator.cpp`. This optimization allows us to:
  - For operations where both operands have to belong to a certain type, the compiler generates `ASSERTS` instruction to make sure that the operation is invoked with appropriate arguments. However, in many cases we perform addition, subtraction or comparisons in cases where one of the arguments is a constant whose value is already known.
  - Consider the case (which will occur quite often) where we initialize a value to some constant value before a loop, and then we overwrite the variable in the loop (e.g. `i = 0; while(i < n) {... i = i + 1;}`). In this case we first assign to a virtual register the constant value, then we use the virtual register in the phi node of the header of the loop. The optimizer will push the constant directly to the phi node of the loop, thus eliminating a virtual register. Of course this allows us to use less move operation. However, this elimination provides us an even more helpful feature: when allocating machine registers we will safe a register possibly avoiding pushing some other values to the stack.
  - An additional feature that is supported by the constant propagator (as the name suggests) is the propagation of constants through operations. That is, when both arguments of a binary operation are known (or the only argument of a unary operation is known) then we can evaluate the operation at compile time. We are aware that this optimization is will not provide any speed up since code in this form is very rare to find. Still, we decided to include this feature to simply extend the capabilities of our compiler.
- **Dead code elimination**: This optimization can be enabled by setting `--opt=dead-code-rm` as flag. The code can be found in `dead_code_remover.cpp`. At the beginning of each scope of a function all assigned variables hve to be assigned to none. However, in many cases this none values is never used since the virtual register holding that value is never used. This motivated us to finding virtual register whole value is never used and to eliminate them. As before, this ensures us not just to eliminate move instructions, but also to safe registers in our machine.
- **Type inference**: This optimization can be enabled by setting `--opt=type-inference`. The code can be found in `type_inferer.cpp`. We try to infer the type of variables and function. This type inference works for integers, strings and boolean values. Once a variables type is known we can perform the following optimizations:

  - We can eliminate unnecessary asserts of values. If an assert turns out not to hold, we cannot simply throw an exception. For instance, consider the case where we have function whose body contains the instruction `N = "NP" - "P"`. If the rest of the code does not contain any additional error, running the code would not result in an exception since the function would be never executed.
  - When we perform an `ADD` operation on two operands whose are known to be integers we can replace it with an `ADD_INT` operation. This operation has the advantage that it can be translated directly to a machine code integer add operation, without having to check whether one of the two operands is a string to perform string addition.

  The type inference optimization works as follows:

  1. We iterate multiple times of the code. While doing this we keep the set of virtual registers whose type is known. We can start inferring the types in several ways. For example, we known the output type of several integer and boolean operations. Moreover, if we check the type of a virtual register once with an assert we know that the virtual register will have the asserted type after the assert instruction.
  2. While iterating, every time we encounter an assert that operates on a virtual register whose type is known we eliminate it. Also, when possible we use the `ADD_INT` instruction.

- **Record shape analysis**: This optimization can be enabled by setting `--opt=shape-analysis` as flag. The code can be found in `shape_analysis.cpp`. After implementing the first three optimizations it was possible to notice that more than half of the time in the benchmark tests (in some tests reaching the peak of 80%) was spend by invoking methods used in `std::unordered_map`. This overhead was especially significant in tests were the records are used as standard C++ structures. That is, they are initialized with a fixed number of fields and throughout the program no other field is created/accessed. This motivated us to store in the record along with the `std::unordered_map` also a fixed set of fields (corresponding to the fields that a record has when it is initialized). This set of fields can be accessed as a common array. We also introduced the following new instructions:

  - `REC_STORE_STATIC`: this instruction stores in one of these fixed fields a value. To access it quickly we provide to the instruction the index of the fixed field we are accessing.
  - `REC_LOAD_STATIC`:this instruction loads one of these fixed fields to a virtual register. Again, we provide to the instruction the index of the fixed field we are accessing.

  Note, by introducing this change to the design of the record we also need to modify the behaviour of other instructions for records (`REC_LOAD_NAME, REC_LOAD_INDX, REC_STORE_NAME, REC_STORE_INDX`). In fact, before accessing the `std::unordered_map` we have to check whether the accessed field belongs to the set of fixed fields. If this is the case we write/read from the position in the array were the field belongs to. To understand when it is possible to perform a static access instead of a dynamic access we have to again perform type inferences. Once we know that a variable is a record with a specific structure we can use the newly introduced operations to access it. Interestingly, just by using the new structure of the records (without doing any type inference) we already achieve a remarkable speedup. This shows how in some test cases the field accesses always happen to the fields that are set when initializing the record. By storing these fields on a fast access array we ensure that we almost never access a map in these test cases. When the optimization is active, the type of the records is inferred as follows:

1. We begin by determining at compile time for each record declaration how many fields are initialized. Moreover, for each of these records we keep track of their names, such that if at different points of the program we declare records with the same set of starting fields we are able to match them and assign them to the same type.
2. We initially mark the type of the virtual records that hold the records when they are initialized. We iterate over the program as long as we are not able anymore to infer the record type of any other virtual register. This inference is performed by using different rules for virtual registers, local reference variables, free variables, function return types and global variables (to ensure correctness we infer their type only if they are not accessed in a scope of some function). We decided to leave out the inference of types of function arguments since it is not always possible to determine which function is invoked on a call instruction (and thus we have no guarantees for their arguments). Finally, every time we perform an access of a field of a record where we know the structure we use the corresponding static access operation.

Register Allocation

The register allocation algorithm is based on the standard second-chance bin packing algorithm used in many JIT-compiled languages.

First, it is interesting to consider why the simplest linear scan algorithm is not suitable for compiling MITScript. The language is designed around very frequent function calls. The issue with these function calls is that they are mostly dynamic: the value of the object being called depends on runtime variables. Therefore, whenever a function is called, it is necessary to save *all* live registers, as it is not known which registers the called function will clobber (note that in the internal calling convention all registers are volatile). For this reason, it would in most cases be unreasonable to try and allocate a single register to a variable for its entire lifetime, as the linear scan algorithm does. Hence, it is necessary to employ strategies based on interval splitting.
Additional complications stem from architectural restrictions. For example, in x86-64, multiplications have to be performed using the RAX and RDX registers, which further restricts live ranges and introduces the need for spilling registers. The register allocator accommodates all such cases by allowing physical registers to be specified as "reserved" for certain intervals.

As input, the algorithm requires the set of live intervals for all virtual registers. Thanks to the usage of SSA form in the previous compilation steps, the task of finding these intervals can be completed in linear time, without any kind of data flow analysis. While allocating the physical registers, the algorithm also naturally decomposes all phi nodes introduced by the SSA form. The output is a control flow graph which can relatively easily be translated to machine code.

Some additional complications are introduced by the need to resolve parallel copy semantics: Between any two instructions, any live value may need to be moved between registers. In most cases there is a requirement on the ordering in which these moves have to be performed. For example, if RDX needs to be moved to RSI at the same time as RSI has to be moved to RDI, the second move obviously needs to be performed first. It might even be the case that a set of registers needs to be arbitrarily permuted, in which case x64 `xchg` instructions are used to perform register swaps.

Register allocation is implemented in `regalloc.cpp`.

Garbage Collection

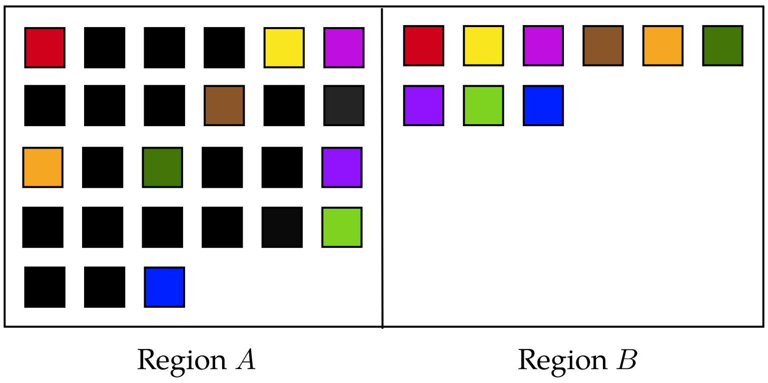
When analyzing the performance metrics for the `bignum` benchmark, we noticed that memory allocation and deallocation using malloc made up almost 30% of the total runtime. To make allocation and deallocation much master, we implemented a relatively simple semi-space copy collector. At the beginning of program execution, a large chunk of heap memory is allocated. Allocations are performed simply by incrementing the current address. Once garbage collection is initiated, all live values are traced and copied to the inactive segment of the allocation.

Additional care is required when handling immediate values: The value of immediates is embedded in the generated code. This means that immediates which contain a pointer (such as long strings) cannot be moved in memory. The solution is to allocate these objects separately and to set a bit marking the object as static in the header of the allocation. Then, during the collection phase of the algorithm, objects which have this bit set are simply not moved.

The generated machine code is in many ways designed with garbage collection in mind. For example, frame pointers are never omitted: Although they are not technically required for program execution itself, tracing the stack is much easier if the base of each stack frame is known. This is because not every value on the stack can be traced: While all stack slots used in execution contain valid values, the stack also contains the return addresses pushed during function calls. Because the stack location of the return address is fixed in relation to the frame pointer, skipping these values becomes much easier when the frame pointer is preserved.

Additionally, all stack slots are initialized to zero upon function entry. This is again required to prevent the tracing of invalid values, as garbage collection may occur at any point during function execution, and stack slots may contain invalid values from previous function calls.

Garbage Collection is implemented in `value.cpp`.



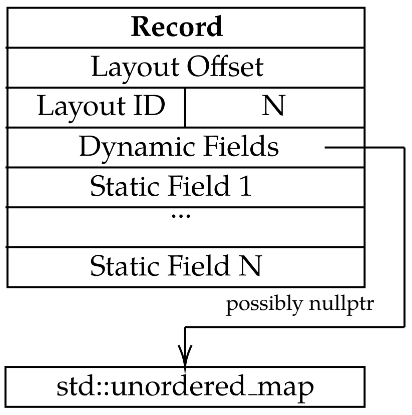
Value Representation

To be compatible with x64 general purpose registers, any possible internal value must be representable with 64 bits. Our garbage collector guarantees that all allocations will be aligned to at least eight bytes. Therefore, when handling values which must be dynamically allocated (such as records and closures), the lower three bits are *always* zero.

We use this fact to store a tag in these lowest three bits of any value, indicating the type that is represented. When the represented type is not a pointer type, the upper 61 bits can be used to represent none, booleans, integers and strings of seven bytes or less directly within one 64 bit value. The obvious advantage of this approach is that memory indirection is greatly reduced; most values can be processed directly within registers.

The usage of inlined short strings provides many intricate performance benefits. Obviously, the reduction of indirection is beneficial for performance. But there are more subtle advantages, too: computing the hash of a short string is a no-op, as it is already uniquely mapped to a 64 bit value. This means that accesses to the record-internal hash map are greatly accelerated for short field names. Further, any integer within the range $[0, 10^8)$ can be efficiently converted to an inline string. Combining this with the fact that computing the hash is a no-op greatly improves performance when using a record as an array.

As has already been alluded to in the "Optimization" section, the representation of records is changed significantly. Each record consists not only of a hash map, but additionally contains a set of fixed fields, which allows very fast accesses if the field index is known thanks to record shape analysis. However, even when the shape is not known, this representation provides considerable speedups, as checking the fixed fields is much faster than searching in the map. To further improve performance, the first four static fields are checked at once using `AVX2` vector instructions directly in the generated assembly. Only if the field is not found is it necessary to fall back to the C++ implementation.



As shown in the diagram, each record keeps track of a layout index. A layout is a list of strings, representing the static field names of a record. This prevents having to store field names in each record of the same structure. In addition, this simplifies the process of checking the first four field names from the generated assembly: The `vpcmpeqq` instruction which is used to check the first four field names takes the layout as a memory operand.

The functions operating on values, as well the functionality tied to the program context, is implemented in `value.cpp`. Much of the functionality is embedded in the generated code, which is emitted by functions defined in `codegen.cpp`

Machine Code Generation

The machine code generator translates the entire program to machine code in a single pass. Each instruction from the high-level intermediate representation is mapped to a sequence of machine code instructions. Registers R10 and R11 are used as temporary registers: Any values being operated on are first loaded into R10 and R11, modified, and finally written back to the destination. While it would in many cases be possible to operate directly in the destination register, implementing this would introduce a lot of additional complexity which we do not consider essential: register-to-register move instructions are very cheap and therefore do not introduce a lot of overhead during program execution.

During the execution of the compiled program, it is required to keep track of additional state, such as the global variables of the program, the memory allocation status, etc. To facilitate this, at the beginning of program compilation, a structure which holds this information is allocated. Because this structure is heap-allocated, pointers to it can be safely embedded into the generated machine code.

A benefit of compiling the entire program in one pass is that graceful handling of errors is facilitated. We define multiple labels, one for each kind of error that can occur during program execution. Then, if an error occurs (by a type check not passing, for instance), execution can simply jump to this label. The code at these labels is set up to load a return value into RAX to indicate the error type which occurred, restore the program state to be the same as at the beginning of execution, and return. This is enabled by the stack pointer being saved at the beginning of execution, in the aforementioned structure holding the program execution context.

Machine code generation is implemented in `codegen.cpp`.

Division of Labor

Jakob implemented the compiler, as well as all optimization and analysis steps. Georgijs implemented the register allocator, machine code generator and garbage collector.