CS444 Assignment 5 & 6: Code Generation

Amrit Sehgal - a38sehga
Aryaman Dhingra - a23dhing
Jagvir Dehal - jdehal
Paranjay Agarwal - p38agarw

Building the compiler

Our compiler can be built with make, using either make build or just make. The system it's built on needs the following dependencies, all of which are available in the student linux environment: g++ 11.4.0, GNU Make 4.3, cmake 3.22.1, flex 2.6.4, and bison (GNU Bison) 3.8.2. After building, the executable joosc will be placed in the project root directory.

Design and Implementation

Our compiler is written in C++. The important components of our program that were added or modified in assignments 5 and 6 are detailed below.

IR Representation

Just as we represented our regular AST, we chose to represent our IR through C++ variants instead of standard object inheritance. We created classes for each IR node and gave all classes the necessary fields and methods. Then, we defined two variants - `StatementIR` and `ExpressionIR`, each of which wrapped their respective IR nodes. Then, we defined a top-level variant called `IR` that encompassed the `StatementIR` and `ExpressionIR`, the `FuncDeclIR` node and the `CompUnitIR` node.

Defining the intermediate representation like this allowed us to implement elegant base visitor classes that visited each of these nodes and performed some action on them, and it also allowed us to write inline visitors using `std::visit`.

Simulation + Java Simulation

To test our IR and Canonical IR, we built an IR simulator in C++, similar to the simulator that is provided by the course. This simulator was essential in helping us catch and fix many bugs that were present within our IR conversion. Once we had a basic version of this working, we created an automated test suite that ran the simulator on the regular IR and the canonical IR for all marmoset tests, which is how we verified the correctness of our IR. The script that runs the test suite is `make compiled-output-test` for all tests and `make compiled-output-test-single path=path/to/test/case`. We added many debug modes to our C++ simulator to help trace through simulator execution and catch bugs within the intermediate representation.

At one point, we started running into a lot of issues with our C++ IR, so we built a new IR visitor that essentially outputted the Java code that represented an IR tree similar to the example provided by the course administrators. With this Java code, we compiled it with the course-provided code and ran the Java simulator on this code. This simulation was done for the

regular IR as well as the Canonical IR, and this helped us debug our IR as well as our C++ simulator. It also gave us another metric we could use to verify the correctness of our IR conversion. We automated this through the commands `make interpret-java-test` and `make interpret-java-test-single path=path/to/test/case`.

Building these workflows proved extremely helpful when debugging the intermediate representation conversion.

IR Conversion

The intermediate representation graph was built as a standard AST traversal, just like the traversals in static analysis. This visitor essentially visited each type of expression or statement node, and then created the IR equivalent of that node and returned that to each parent. This is how the IR was constructed from our AST on a high level. There is a lot of intricate logic for constructing the IR for each AST node, so going into more detail is not a sensible discussion within this report. The header file for the IR builder can be found in `ast-to-ir.h` and the implementation of all the conversion functions as well as the visitors can be found in `ast-to-ir.cc`. After this conversion step, the IR is flattened into a canonical IR form to more closely represent assembly instructions, instead of being in a large tree-like structure.

Canonical IR Conversion

Our IR canonicalizer flattened the tree-like structure of the IR into something that more closely resembles assembly code. Since our IR was arranged in a tree of variants, this was done by writing simple inline visitors that visited each IR node, lowered and flattened them into a vector of Statements, and returned an instance of a new class called `LoweredExpression` or `LoweredStatement` which is just a vector of statements with an expression or just a vector of statements, respectively. This can be found in `ir-canonicalizer.h`.

Tiling

The tiling process within the compiler is a process that maps the canonical IR to a set of assembly instructions. This tiling operation checks each node in the IR and constructs a combination of tiles that represent the ASM of the tile. Each tile is selected using the decideIsCandidate function, ensuring that the overall assembly sequence is cost-effective. Detailed logic underpins the selection and optimization of tiles, but an exhaustive discussion of these mechanisms exceeds the scope of this report. The source code can be found in `tile.h` and `ir-tiling.h`.

Register Allocation

We created 2 main register allocators, `BrainlessRegisterAllocator` and `LinearScanningRegisterAllocator`, under the abstract class `RegisterAllocator`. The `RegisterAllocator` base class outlines the basic framework of the register allocation algorithm. The method `allocateRegisters` is to be implemented by the derived classes to handle allocating abstract registers to real registers or spilling them to the stack, and returns the

stack size required. `reg_offsets` stores the mapping of an abstract register to its corresponding stack offset while the `replaceAbstracts` method handles replacing the abstract registers in the instructions with real registers or stack offsets and ensuring the modified instructions are appended to the target list. Loading and storing of abstract registers is handled by `loadAbstractRegister` and `storeAbstractRegister` respectively. The allocator in general is run on one function at a time.

`BrainlessRegisterAlocator` implements the naive register allocation where we store all the abstract registers in the stack. The `allocateRegisters` method spills all the variables to stack while also storing the stack offsets for each abstract register used in the function body in `reg_offsets`. Then we call `replaceAbstracts` which adds the load/store instructions for the respective stack values and handles replacing the abstract registers with the corresponding machine registers. This proves to be very inefficient with a lot of redundant assembly instructions, `LinearScanningRegisterAllocator` proves to be a significant upgrade to this.

We create a new data structure called `Interval` in LinearScanningRegisterAllocator, which contains all the details about abstract registers. First, we check if all the temporary registers are initialized using the `checkAllTemporariesInitialized` method before we create the intervals within `constructIntervals`. As it goes through the function body, `constructIntervals` updates uncommitted intervals whenever they are used. These intervals are initialized for each register when they are used. On a write to a register, an open interval is first searched for by the method. If not found, then it commits the interval at the previous read point and creates a new one reflecting its current use. Adding reads increases the lifespan of this entity that covers all uses needed. Special registers such as the stack pointer are excluded. The remaining open intervals would be committed to and sorted according to their starting times so as they can be ready for an effective allocation of registers plus save on spilling.

After storing the Intervals, we iterate over each interval in that list of intervals. To begin with, we have to update our current active interval list at every iteration by determining and handling those, which have grown inactive when the register allocator advances using `finishInactiveIntervals` helper function. Then we let real registers take hold of themselves thus freeing and giving off other intervals currently utilizing them. After this is assigning abstract registers to real registers or stack offsets via `assignInterval` helper function.

The `assignInterval` helper function handles assignment of registers or stack slots to intervals based on availability and necessity. If a specific predetermined register is passed into this method along with the interval it will first try to assign it there while excluding that register from the available free registers set. The method looks for any free register to assign to the interval if no predetermined register is specified, then marking it as used by removing it from the

`free_registers` set immediately. If the registers have been exhausted or if they are not there, an attempt must be made to allocate stack space. The allocation of a new stack space comes last when previously used stack spaces are unavailable to avoid expanding the stack frame unnecessarily, this helps us maintain efficient memory usage and reduce runtime overhead.

We then proceed to do the actual instruction replacement.To begin with, we clear and prepare to repopulate `active_intervals` and make a fresh list for updated instructions. As we loop through all the source function body instructions, we update which intervals are active based on the current instruction index, remove intervals that have become inactive and add those that become active at their starting points. Once each instruction has been processed to replace abstract registers with their allocated concrete counterparts. A fresh list is created to gather the modified instructions which incorporate register allocation and stack accesses. The next step involves using `replaceAbstracts` to replace abstract registers with real ones and adding necessary load/store instructions for registers which are spilled on the stack. Finally, this new list replaces the original instruction set in the function body.

We also have a `NoopRegisterAllocator` which does nothing, so it just prints the instructions with the abstract registers. This does not produce working programs but is very useful for debugging.

Benchmarking

You can use `make bench` to generate a benchmark of optimized vs unoptimized code generation. The results will be in `benchmarks/results.csv`. This was done using a simple python script that compiled the program without optimized register allocation and then we timed the runtime of the executable, and then did the same thing but compiled the program using the optimized register allocation. Each program is run 100 times and the average is taken for each figure.

This is the result of our benchmarks on our programs:

optimization	benchmark_name	time in ms w/ opt	time in ms w/o opt	speedup
opt-reg-only	factorial_recursive.java	1.133115291595459	2.0787644386291504	1.8345568664087042
opt-reg-only	fibonacci.java	0.888512134552002	1.2122035026550293	1.3643072002232544
opt-reg-only	factorial_iterative.java	2.1836066246032715	2.3467612266540527	1.074717946086294

Testing and Known Issues

The primary way we have been testing our compiler until now is with integration tests and course-provided sample programs. For these two assignments, we added even more test suites that ran our IR simulator on our intermediate representation on both canonical and non-canonical forms of the IR. We also added a test suite that runs the IR on the Java simulator (we did this through a unique IR-to-Java code converter, which can be found in `IR-java-converter.h` and `IR-java-converter.cc`). We have a range of different test modes we use to debug and test our program. The most common test command is `make compiled-output-test`. This command essentially runs the IR and Canonical IR on the simulator, as well as the compiled ASM, and compares the output of all three. If the three outputs match, it is considered a successful test, otherwise it is considered a failure.

Another thing we added for this assignment is a fuzzer to generate random test cases to try and crash the compiler. This can be found in the `fuzz/` directory.

Work Split

Amrit

- Added support to our testing infrastructure for testing return codes of compiled programs
- Implemented canonicalizing of IR to canonical IR
- Implemented tiling and mostly focused on code generation at the assembly level
- Designed internal representation of assembly instructions as classes, for easier passes
- Built brainless register allocator and linear scanning register allocator
- Added additional information needed for codegen to environment building and type-checking

Aryaman

- Defined a third of the IR classes
- Built the C++ simulator and the Java simulation workflow for IR testing purposes
- Worked on string literal IR conversion
- Built functionality to allocate dispatch vector to memory, and assign functions in dispatch vector
- Wrote a colouring algorithm to help assign functions to positions within a dispatch vector
- Helped debug the IR for array creation and array access
- Added the ‘make bench’ functionality and workflow

Jagvir

- Implemented fuzzing system for automatically generating diverse tests for the compiler
- Implemented the bulk of the IR builder that converted the abstract syntax tree into the intermediate representation
- Created helper static functions for creating IR nodes
- Implemented the dispatch vector builder
- Worked on building array creation and array access
- Made minor adjustments to the simulator to help with static fields and functions

Paranjay

- Wrote the basic implementation of brainless register allocation.
- Worked on the linear scanning allocator.
- Added support to pick the allocator to use.
- Helped debug the allocators.
- Wrote the benchmark programs to test the compilers performance.

Thoughts

These two assignments were especially hard, as we had to debug an intermediate representation as well as assembly code across multiple files, so it was definitely a huge learning experience. We set up a somewhat decent testing infrastructure that made the debugging process a lot easier. The interpreter helped us a lot in debugging the intermediate representation. However, it became useless after a5 as we did not have the time to adapt our interpreter to OOP, and debugging pure ASM was much more challenging.

We still had a lot of fun, and learnt a lot!