Compiler Backend Design

Yunjie Zhang, Minh Ho, Kevin Lou

October 2025

Backend Design

High-Level

Our backend implements a Tiger-IR to MIPS32 compiler that emits SPIM-executable assembly. The pipeline parses Tiger-IR, performs instruction selection, and applies one of two register allocation strategies—naïve per-instruction allocation or an intra-block greedy allocator—before serializing the instruction stream to `.s` for SPIM. Arrays are stack-allocated by default; array parameters are passed as pointers. Labels are qualified by function for uniqueness. System call intrinsics (`geti`, `puti`, `getc`, `putc`, `getf`, `putf`) are mapped to SPIM syscall codes.

Low-Level

Implementation Language: C++17.

Module overview with key data structures and rationale:

- `ir_reader.cpp` (*Tiger-IR parser*): Parses the textual IR into an object model. Uses `std::vector` to keep ordered lines, `std::regex` for token/type validation, and an `unordered_map<string, IRVariableOperand>` as a symbol table per function to ensure single definition and fast lookup. Errors are reported via an `IRException` for robust diagnostics. The parser builds typed operands (constants, variables, labels, functions) and checks type rules (array vs. scalar, int vs. float).
- `ir_core.cpp` / `ir_types.cpp` (*IR data model*): Defines `IRProgram`, `IRFunction`, `IRInstruction`, and singleton types `IRIntType`/`IRFloatType`. Array types `IRArrayType` are *interned* via an `unordered_map<type,size>` cache to avoid duplicate allocations and enable fast pointer-comparison of types. Includes an `IRPrinter` for debugging IR dumps.
- `frame_builder.cpp` (*stack layout*): Computes a compact frame for each function. Uses an `unordered_set<string>` to recognize array parameters (pointer-only slots) and an `unordered_map<string,int>` *varOffset* from variable name to byte offset. A running offset is 8-byte aligned (reserve 0($fp)=ra, 4($fp)=fp) to simplify address arithmetic. Local arrays receive contiguous space; scalar locals get 4 bytes.
- `register_manager.cpp` (*register pool & stack slots*): Manages a pool of physical caller-saved registers ($t0–$t9) with `availableRegs`/`usedRegs` (`std::vector`). Maps IR variable names

to currently assigned registers using `unordered_map<string, Register>`. Provides virtual register generation for temporaries and a monotonically growing negative stack offset allocator for spills. We intentionally avoid $s* to prevent SPIM from reading uninitialized callee-saved registers.

- `emit_helpers.cpp` (*emission utilities*): Small helpers to standardize `lw/sw` for scalars, `l.s/s.s` for floats, and array address computation. Centralizing this logic reduces code duplication and keeps address rules (param vs. local arrays) consistent.
- `alloc_naive.cpp` (*baseline allocator*): For each IR instruction, loads inputs into temporaries ($t0–$t4), executes the operation, then stores results immediately to the stack. The design is stateless and uses only local temporaries, guaranteeing correctness while sacrificing memory efficiency—useful as a reference implementation.
- `alloc_greedy.cpp` (*intra-block greedy allocator*): Partitions a function into basic blocks (bounded by labels/branches/calls/returns). Builds a backward **next-use** map `vector<unordered_map<string,int>>` per instruction index. Maintains a small array of register **slots** ($t5–$t9) with metadata {`var`, `occupied`, `dirty`}. A fast `unordered_map<string,int>` `varToSlot` resolves current residency. On demand, it chooses a victim by furthest next-use (or no-future-use) and spills if dirty. At block exits, all dirty slots are flushed. This structure strikes a balance between speed and reduced memory traffic.
- `instruction_selector.cpp` (*driver & program skeleton*): Emits a minimal program prologue (`jal main`; `li $v0,10`; `syscall`) and dispatches each function to the chosen allocator. Provides assembly serialization (`.text` header + linear walk) and file I/O. Keeps policy for label qualification and argument passing in the first four $a registers.
- `mips_instructions.cpp` (*assembly printing*): Maps the internal `MIPSOp` enum to strings and pretty-prints each instruction as `[label:]  op arg1, arg2, ...`. This keeps emission logic structured and easy to diff.
- `ir_to_mips.cpp` (*CLI*): Thin entry point handling `--naive`/`--greedy`. Loads IR, selects the allocator, assembles, and writes `.s`. All errors propagate as exceptions with user-friendly messages on failure.

Software Challenges

1. Interpreter safety: SPIM flagged reads of uninitialized callee-saved registers. We restricted allocation to caller-saved `$t*` registers and avoided `$s*`.
2. Array addressing: Parameter arrays vs. local arrays required separate base handling; unified via `FrameInfo`.
3. Block structure & liveness: Correct block boundaries (labels, branches, calls, returns) were essential for next-use computation.
4. Stack layout: Mixed scalars/arrays needed consistent offsets and alignment; standardized in `frame_builder.cpp`.

Unresolved Bugs/Deficiencies

- Greedy allocation is intra-block only (no global liveness).
- Floating-point variables beyond syscalls are not fully supported.
- Callee-saved `$s*` preservation omitted (safe for our SPIM tests).
- No heap array allocation via `sbrk`; arrays reside on the stack.

Build/Usage Instructions

Build (from repo root): `./build.sh`
Run: `./run.sh <input.ir> <output.s> [--naive | --greedy]`
Example: `./run.sh public_test_cases/prime/prime.ir prime.s --greedy`
Execute in SPIM with stats: `spim -keepstats -f prime.s`

Results (Prime Only)

The table below reports dynamic statistics from SPIM for nine prime test inputs (0–8). The "Reduction vs. Naïve" column reports percentage decrease in memory *reads* for `--greedy` relative to `--naive`.

Test Case	Allocator	#Instr	#Reads	#Writes	#Branches	#Other
prime 0	Naive	48	6	10	4	28
prime 0	Greedy	50	5	10	4	31
prime 1	Naive	153	39	32	18	64
prime 1	Greedy	159	26	29	18	86
prime 2	Naive	257	71	55	32	99
prime 2	Greedy	264	45	50	32	137
prime 3	Naive	361	103	78	46	134
prime 3	Greedy	369	64	71	46	188
prime 4	Naive	235	63	49	31	92
prime 4	Greedy	241	40	45	31	125
prime 5	Naive	502	145	108	67	182
prime 5	Greedy	511	89	99	67	256
prime 6	Naive	1713	519	377	228	589
prime 6	Greedy	1734	311	344	228	851
prime 7	Naive	235	63	49	31	92
prime 7	Greedy	241	40	45	31	125
prime 8	Naive	6393	1959	1412	858	2164
prime 8	Greedy	6459	1166	1289	858	3146
prime 9	Naive	12633	3879	2792	1698	4264
prime 9	Greedy	12759	2306	2549	1698	6206

Performance Analysis. The comparison between the naive and greedy allocators across the `prime0`–prime9 test cases shows a consistent reduction in the number of memory reads (#Reads) achieved by the greedy approach. For smaller inputs, the improvement was moderate: from 6 to 5 reads in `prime0` (16.7%), 39 to 26 in `prime1` (33.3%), and 71 to 45 in `prime2` (36.6%). As input size increased, the effect became more pronounced: `prime3` reduced from 103 to 64 (37.9%), `prime5` from 145 to 89 (38.6%), and the large-scale `prime8` and `prime9` tests saw decreases from 1959 to 1166 (40.5%) and from 3879 to 2306 (40.6%), respectively. Averaged across all benchmarks, the greedy allocator reduced #Reads by roughly 35–40%. This aligns with theoretical expectations for intra–block liveness analysis: by retaining frequently used variables in physical registers and avoiding redundant memory reloads, it significantly lowers memory traffic and improves runtime efficiency while maintaining correctness.