Lecture 23: Code optimization tips, JIT compilation

David Hovemeyer

December 1, 2021

601.428/628 Compilers and Interpreters

Today

- Implementing local value numbering
- Copy propagation
- Implementing register allocation
- Bytecode compilers, interpreters, and JIT compilation

Implementing local value numbering

Local value numbering

Relatively easy way to find redundant computations

A *value number* is an integer representing a runtime value; if two locations are known to have the same value number, then at runtime they are guaranteed to have the same value

Optimization: replace recomputations with use of available values

Fundamental data structure: map of "LVN keys" to value numbers

LVN key

A LVN key identifies a computed value:

- Value numbers of operand(s)
  - For commutative operations, canonicalize the order (e.g., left hand operation must have lower value number)
- Operation being performed (add, subtract, etc.)
- Whether the computed value is a compile-time constant

Data to keep track of

As the analysis progresses, keep track of:

- map of constant values to their value numbers (just for constant values)
- map of value numbers to constant values (just for constant values)
- map of virtual registers to value numbers (i.e., find out what value number is in each virtual register)
- map of value numbers to sets of virtual registers known to contain the value number
- map of LVNKey to value number
- next value number to be assigned

Modeling instructions

- see an unknown vreg: assign a new value number
- see a load from memory, or a read: assign a new value number
- computed value: find value number of value being computed
  1. find value numbers of operands
  2. create an LVNKey from opcode and operand value numbers
     - canonicalize order of operands if operation is commutative
     - determine if value is a compile-time constant
  3. check map of LVNKey to value number; if not found, assign new value number (and update the map)

For each def (assignment to vreg), goal is to know the value number of value being assigned to the vreg

Effect of defs

If a def assigns a value number to a vreg that is different than the one it previously contained, then we must update all data structures appropriately.

Including: removing it from the set of vregs known to contain its previous value number.

Transformation

To the extent possible, every def of the form

vreg $\leftarrow$ *some value*

is replaced with

vreg $\leftarrow$ *known value*

"known value" could be a compile-time constant (best case), or a vreg known to store the same value as *some value*

What LVN achieves

Value numbering doesn't eliminate any instructions: it just makes redundancies more explicit.

Subsequent copy propagation and elimination of stores to dead vregs passes will remove instructions that are no longer needed.

Copy propagation

Result of LVN, copy propagation

LVN will generate instructions of the form

$$vreg_n \leftarrow vreg_m$$

where vreg_m is a virtual register containing a previously computed value

Subsequent uses of vreg_n can be replaced with vreg_m. This transformation is *copy propagation*.

Copy propagation example

Consider the code:

```
/* original code */
addi vr2, vr0, vr1
mov vr4, vr0
addi vr5, vr4, vr3
```

Copy propagation example

After copy propagation:

```
/* original code */          /* transformed code */
addi vr2, vr0, vr1           addi vr2, vr0, vr1
mov vr4, vr0                 mov vr4, vr0
addi vr5, vr4, vr3           addi vr5, vr0, vr3
```

If vr4 became dead at the point of the assignment to it, it can be eliminated

Implementing local register allocation

Local register allocation

Goal of register allocator is to assign (temporarily!) machine registers to virtual registers

- Special case: virtual registers that are alive at the end of the basic block shouldn't have a temporary register assignment
- Could leave these allocated in memory, or use "long term" register assignment (i.e., callee-save registers)

Problem: there are a limited number of machine registers

- If we run out, steal a machine register that is currently in use, first spilling its value to memory
- Bottom-up register allocation: when stealing, choose the virtual register whose next def is the furthest in the future

Register allocator state

Information to keep track of as allocator progresses through instructions in basic block:

- Collection of available machine registers (stack or queue)
- Map of virtual register numbers to assigned machine register
- Collection of available spill locations (stack or queue)
- Map of virtual register numbers to spill locations

Recording register assignments

The register allocator will need to communicate register assignments to the low-level code generator.

One way to do this: add a field to Operand, if set to a non-negative value, it's the assigned machine register.

Making allocations and assignments

For each virtual register used in an instruction[1]:

- If its value is currently spilled, allocate a machine register and restore it
- If there is a current assignment to a machine register, record the assignment
- If there is no assignment, allocate a register and record the assignment

[1] Except for vregs excluded from being assigned a temporary register.

Allocating a register

- If a machine register is available, allocate it (easy case)
- If no machine register is available (harder case):
  1. Choose a victim vreg
  2. Allocate a currently-unused spill location, otherwise, use a new spill location
  3. Emit a spill instruction (specifying the vreg, mreg, and spill location)
  4. Use the stolen mreg to satisfy the allocation

Restoring a spilled register

Assuming that an machine register has already been allocated:

1. Emit a restore instruction (specifying vreg, mreg, and spill location)
2. Return the (no longer used) spill location to the collection of available spill locations

Allocating storage for spill locations

Determine maximum number of spill locations used (over all basic blocks)

Place storage area for spills somewhere in the stack frame

Low-level code generator will need to determine an offset into the storage area for each spill and restore

Dealing with procedure calls

The compiler must assume that a call to a procedure could change the value of any caller-save register! (e.g., %rcx, %rdx, %r10, etc.)

Could emit code to push values of in-use caller-save registers on the stack prior to call, pop saved values after call returns

Bytecode compilers, interpreters, JIT compilation

Another approach for implementing interpreters

Recall from Assignment 2 (remember that!) that we implemented an interpreter using recursive traversal of an AST as the evaluation strategy

Another approach: *bytecode compilation and interpretation*

Bytecode

Bytecode: a "virtual machine language"

- Opcodes implement the semantic operations of the source program
- Body of each procedure is represented by an array of bytecode instructions
- Can be register-based or stack-based

Advantages: code representation is compact, contiguous in memory (good for cache utilization), execution is simpler (10x better than AST-based interpreter is possible)

Interpreter generates bytecode for each procedure (so, the interpreter is really a compiler!)

Interpreting bytecode

```
char *code = /* the bytecode array */
int pc = 0, done = 0;
while (!done) {
  int opcode = code[pc++];
  switch (opcode) {
    /* cases for the various opcodes */
  }
}
```

Additional bytes (past the opcode) could encode additional information (e.g., constant pool entry, register number(s), etc.)

Branches assign the target instruction offset to `pc`. A procedure return instruction would set `done` to 1.

Stack-based bytecode

In a *stack-based* bytecode interpreter, each instruction takes its operand values from the operand stack, and pushes its result onto the operand stack.

Java bytecode works this way: see Gosling, "Java Intermediate Bytecodes" (linked from course schedule page.) CIL (the DotNET bytecode language) is also stack-based.

Code representation is very compact because locations of intermediate values do not need to be represented explicitly.

Register-based bytecode

In a *register-based* bytecode interpreter, temporary values and local variables are represented by *registers*.

- Exactly the same idea as virtual registers in our high-level IR in Assignments 4 and 5

JIT compilation

Although bytecode interpreters can achieve reasonably good performance, they are still significantly slower than native machine code.

A *JIT* (just-in-time) compiler dynamically translates bytecode instructions into native machine instructions.

Modern JIT-based language runtimes (e.g., JVM with Hotspot) can be competitive with optimizing ahead-of-time compilers

- Sometimes they are superior!

The essential challenge of JIT compilation

An ahead-of-time compiler can liberally perform optimizations that are expensive in terms of runtime and memory use

A JIT compiler is more constrained, because its running time and memory use compete with the running program

Techniques used by JIT-enabled language runtimes:

- Use a variety of execution strategies
- Do JIT compilation selectively
- Do optimizations selectively

The fundamental principle of performance

The efficiency of code matters in proportion to how frequently it is executed

So, effort should be focused on improving the performance of the (typically small) fraction of code in the program that actually has an impact on performance

Execution strategies, profiling

A language runtime can use a variety of strategies to execute program code

Strategy	Advantage	Disadvantage
Interpretation	Easy, quick to start	Code execution is slow
Simple JIT compilation	Somewhat faster code execution	JIT compilation takes time, missed optimization opportunities
Optimized JIT compilation	Fastest code execution	JIT compilation could be expensive (time and memory)

Idea:

- Always start with interpretation
- Continually profile the running program to determine where time is spent
- As performance-critical procedures are found, use more expensive JIT compilation

Profiling

Language runtime must have a way to record where time is being spent

- Overhead of instrumentation for profiling must be kept low!
- Counting procedure calls is fairly cheap
- Counting basic block executions: much more expensive
  - Could do this infrequently

Simple JIT compilation

A "first stage" JIT can use simple and fast optimization and code generation techniques

- Might not produce the best code possible, but can do significantly better than interpretation

Again, see Gosling, "Java Intermediate Bytecodes"

Optimized JIT compilation

Once the language runtime has identified performance critical code (e.g., a core loop computation), it can apply more sophisticated optimization techniques

Compiler IR and optimization passes must be designed to be runtime and memory efficient

30+ years of research on this (hard to summarize)

Lots of difficult issues: multiple program threads, interaction with garbage collector, on-stack replacement, etc.