Lecture 18: Code optimization and Garbage Collection

Using solvers for code generation
Garbage collection in managed runtimes
Announcements

• HW4
  – Due tonight at 11pm (no late days)

• 50-min final quiz this Thursday in section
  – 2 pages of single-sided hand written notes
  – practice exams on course website
  – please attend your assigned section
Outline for today

• Techniques for code generation (continued)
  – Classical and solver-based techniques

• Managed runtimes
  – Garbage collection
Code optimization
(complete slides including from last lecture)
Scope of optimizations

Scope of study for optimizations:

• **peephole**: look at adjacent instructions

• **local**: look at straight-line sequence of statements

• **global**(intraprocedural): look at whole procedure

• **interprocedural**: look across procedures

Larger scope ⇒ better optimization,

but more cost & complexity
How is the program improved

- **naïve**: no optimization after code generation
- **rewrite rules**: used in peephole optimization
- **instruction selection**: tree covering
- **deductive**: derive equivalent programs
- **superoptimization** and synthesis: search for a correct program
Naïve code generation
Naïve code generation

For each AST node, generate a sequence of instructions.

The same as bytecode generation (see previous lectures).

Pros: simple

Cons: suboptimal code
Peephole optimization
Peephole optimizations

Replace a sequence of adjacent instructions with a more optimal sequence

\[
\begin{align*}
&\text{sw } \$8, 12(\$fp) \\
&\text{lw } \$12, 12(\$fp) \\
&\Rightarrow \\
&\text{sub sp, 4, sp} \\
&\text{mov r1, 0(sp)} \\
&\Rightarrow \\
&\text{sw } \$8, 12(\$fp) \\
&\text{mv } \$12, \$8 \\
&\text{mov r1, -(sp)}
\end{align*}
\]
Instruction selection via tree coverage
Better code-gen rules

Rather than translating one AST node to an instruction sequence, we map multiple nodes to a sequence.
Tree covering as parsing

```
+----+----+
|    |    |
|    |    |
|   +---+
|   |    |
|   |    |
|   |----|
|   |    |
|   |    |
|   |----|
|   |    |
|   |    |
|   |----|
|   |    |
|   |    |
|   |----|
|   |    |
|   |    |
|   |----|
|   |    |
|   |    |
```

reg

reg

Int(c)

reg

reg

reg

Int(a)

Int(b)
Deductive optimizers
Denali: synthesis with axioms and E-graphs

∀ \( n \). \( 2^n = 2**n \)

∀ \( k,n \). \( k \times 2^n = k \ll n \)

∀ \( k,n \): \( k \times 4 + n = \text{s4addl}(k,n) \)

\[
\text{reg6} \times 4 + 1 \quad \text{specification} \quad \text{\textbf{reg6}} \quad \text{s4addl}(\text{reg6},1) \quad \text{synthesized program}
\]
Two kinds of axioms

**Instruction semantics:** defines the language

∀ n. \(2^n = 2^{**n}\)

∀ k, n. \(k \times 2^n = k<<n\)

**Algebraic properties:** associativity of add64, memory modeling, ...

\(\forall k, n:: k \times 4 + n = s4addl(k, n)\)

\((\forall x, y :: \text{add64}(x, y) = \text{add64}(y, x))\)

\((\forall x, y, z :: \text{add64}(x, \text{add64}(y, z)) = \text{add64}(\text{add64}(x, y), z))\)

\((\forall x :: \text{add64}(x, 0) = x)\)

\((\forall a, i, j, x :: i = j\)

\(\forall \text{select}(\text{store}(a, i, x), j) = \text{select}(a, j))\)
Two kinds of axioms

**Instruction semantics:** defines the language

\[ \forall k, n . \ k \ast 2^n = k \ll n \]

\[ \forall k, n :: k \ast 4 + n = s4add1(k, n) \]

**Algebraic properties:** associativity of add64, memory modeling, ...

\[ \forall n . \ 2^n = 2^{\star\star n} \]

\[ (\forall x, y :: \text{add64}(x, y) = \text{add64}(y, x)) \]

\[ (\forall x, y, z :: \text{add64}(x, \text{add64}(y, z)) = \text{add64}(\text{add64}(x, y), z)) \]

\[ (\forall x :: \text{add64}(x, 0) = x) \]

\[ (\forall a, i, j, x :: i = j \]

\[ \vee \text{select}((\text{store}(a, i, x), j)) = \text{select}(a, j) \]
Properties of deductive synthesizers

Efficient and provably correct
- thanks to semantics-preserving rules
- only correct programs are explored

Similar systems were built for axiomatizable domains
- expression equivalence (Denali)
- linear filters (FFTW, Spiral)
- linear algebra (FLAME)
- statistical calculations (AutoBayes)
- data structures as relational DBs (P2; Hawkins et al.)
Downsides of deductive optimizers

**Completeness** hinges on sufficient axioms
some domains hard to axiomatize (e.g., sparse matrices)

**Control** over the “shape” of the synthesized program
we often want predictable, human-readable programs

Solver-based Inductive synthesis achieves these
see next section
Superoptimization
Massalin’s superoptimization (1987)

Search nearly exhaustively for an optimal program.

```
signum(x)
int     x;
{
    if(x > 0)    return 1;
    else if(x < 0)    return -1;
    else
        return 0;
}
```

```
add.l  d0,d0  \textit{add d0 to itself}
subx.l d1,d1  \textit{subtract (d1 + Carry) from d1}
negx.l d0     \textit{put (0 - d0 - Carry) into d0}
addx.l d1,d1  \textit{add (d1 + Carry) to d1}
```

[Alexia Henry Massalin, Superoptimizer: a look at the smallest program, ASPLOS 1987]
The scope of superoptimization alone is limited.

**Lesson:** think of it as a tactical tool.
Synthesis with partial programs
see example of SIMD matrix transpose from previous lecture
Preparing your language for synthesis

Extend the language with two constructs

\[ \phi(x, y) : y = \text{foo}(x) \]

\[ \text{?? substituted with an int constant satisfying } \phi \]

Instead of \texttt{implements}, assertions over safety properties can be used
Synthesis as search over candidate programs

Partial program (sketch) defines a candidate space
we search this space for a program that satisfies $\phi$

Usually can’t search this space by enumeration
space is too large ($\gg 10^{10}$)

Describe the space symbolically
solution to constraints encoded in a logical formula gives
values of holes, indirectly identifying a correct program
Synthesis from partial programs

spec sketch

program-to-formula compiler

\( \phi \)

solver
“synthesis engine”

\( h \mapsto 1 \)

code generator

\( P[h] \)

\( P[1] \)
CounterExample - Guided Inductive Synthesis (CEGIS)

Inductive Synthesizer
compute a candidate implementation from concrete inputs.

verifier/checker
Your verifier/checker goes here

search for completion
succeed
candidate implementation
fail
unrealizable
ok
fail
add a (bounded) counterexample input
Garbage Collection

Slides courtesy of Profs. Alex Aiken and George Necula
Lecture Outline

• Why Automatic Memory Management?

• Garbage Collection

• Three Techniques
  – Mark and Sweep
  – Stop and Copy
  – Reference Counting
Why Automatic Memory Management?

• Storage management is still a hard problem in modern programming

• C and C++ programs have many storage bugs
  – forgetting to free unused memory
  – dereferencing a dangling pointer
  – overwriting parts of a data structure by accident
  – and so on...

• Storage bugs are hard to find
  – a bug can lead to a visible effect far away in time and program text from the source
Type Safety and Memory Management

• Some storage bugs can be prevented in a strongly typed language
  – e.g., you cannot overrun the array limits

• Can types prevent errors in programs with manual allocation and deallocation of memory?
  – some fancy type systems (linear types) were designed for this purpose but they complicate programming significantly

• If you want type safety then you must use automatic memory management
Automatic Memory Management

• This is an old problem:
  – studied since the 1950s for LISP

• There are several well-known techniques for performing completely automatic memory management

• Until recently they were unpopular outside the Lisp family of languages
  – just like type safety used to be unpopular
The Basic Idea

- When an object that takes memory space is created, unused space is automatically allocated
  - In 401, new objects are created by new X
- JS memory manager keeps track of all allocated objects and amount unused heap space
- After a while there is no more unused space
- Some space is occupied by objects that will never be used again
- This space can be freed to be reused later
The Basic Idea (Cont.)

• How can we tell whether an object will “never be used again”?
  – in general it is impossible to tell
  – we will have to use a heuristic to find many (not all) objects that will never be used again

• Observation: a program can use only the objects that it can find:
  
  lambda f () { def a = new A() }
  
  f()
  – After f() there is no way to access the newly allocated object
Garbage

• An object x is **reachable** if and only if:
  – an interpreter frame (sym table) contains a pointer to x, or
  – another reachable object y contains a pointer to x
• You can find all reachable objects by starting from interpreter frames and following all the pointers
• An unreachable object can never by referred by the program
  – these objects are called **garbage**
Reachability is an Approximation

- Consider the program:

```java
x = new A() // p1
y = new B() // p2
x = y
if (alwaysTrue) { x = new A() } // p3
else { x.foo() }
```

- After `x = y` (assuming y becomes dead there)
  - the object A @ p1 is not reachable anymore
  - the object B @ p2 is reachable (through x)
  - thus B @ p2 is not garbage and is not collected
  - but object B @ p2 is never going to be used
A Simple Example

- We start tracing from pointers from all frames
  - These are called roots
- C is not reachable from any frames
- Thus we can reuse its storage

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td>null</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sym</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Elements of Garbage Collection

• Every garbage collection scheme has the following steps
  1. Allocate space as needed for new objects
  2. When space runs out:
     a) Compute what objects might be used again
        (by tracing objects reachable from the “root”)
     b) Free the space used by objects not found in (a)
• Some strategies perform garbage collection before the space actually runs out
Algorithm 1: Mark and Sweep
Mark and Sweep

- When memory runs out, GC executes two phases
  - the mark phase: traces reachable objects
  - the sweep phase: collects garbage objects

- Every object has an extra bit: the **mark** bit
  - reserved for memory management
  - initially the mark bit is 0
  - set to 1 for the reachable objects in the mark phase
The Mark Phase

def todo = { roots }
while todo ≠ ∅ {
    pick v ∈ todo
    todo = todo - { v }
    if mark(v) == 0 { // v is unmarked yet
        mark(v) = 1
        v₁,…,vₙ = pointers that v points to
        todo = todo ∪ {v₁,…,vₙ}
    }
}
The Sweep Phase

• The sweep phase scans the heap looking for objects with mark bit 0
  – these objects have not been visited in the mark phase
  – they are garbage

• Any such object is added to the free list

• The objects with a mark bit 1 have their mark bit reset to 0
The Sweep Phase (Cont.)

for (obj : allocatedObjs) {
    if (mark(obj) == 1) {
        mark(obj) = 0
    } else {
        // free obj and add it back to unallocated heap
    }
}

• Memory manager keeps track of each object’s size
  – This can be done using types

• Memory manager typically maintains a “free list”
  – Removes an entry from free list when new T is called
Mark and Sweep Example

After mark:

After sweep:
Details

• While conceptually simple, this algorithm has a number of tricky details
  – this is typical of GC algorithms

• A serious problem with the mark phase
  – it is invoked when we are out of space
  – yet it needs space to construct the todo list
  – the size of the todo list is unbounded so we cannot reserve space for it a priori
Mark and Sweep: Details

• The todo list is used as an auxiliary data structure to perform the reachability analysis

• There is a trick that allows the auxiliary data to be stored in the objects themselves
  – pointer reversal: when a pointer is followed it is reversed to point to its parent

• Similarly, the free list is stored in the free objects themselves
Mark and Sweep. Evaluation

• Space for a new object is allocated from the new list
  – a block large enough is picked
  – an area of the necessary size is allocated from it
  – the left-over is put back in the free list
• Mark and sweep can fragment the memory
• Advantage: objects are not moved during GC
  – no need to update the pointers to objects
  – works for languages like C and C++
Algorithm 2: Stop and copy
Stop and Copy

- Memory is organized into two areas
  - old space: used for allocation
  - new space: used as a reserve for GC

- The heap pointer points to the next free word in the old space
  - allocation just advances the heap pointer
Stop and Copy Garbage Collection

• Starts when the old space is full
• Copies all reachable objects from old space into new space
  – garbage is left behind
  – after the copy phase the new space uses less space than the old one before the collection
• After the copy the roles of the old and new spaces are reversed and the program resumes
Stop and Copy Garbage Collection.

Example

Before collection:

After collection:
Implementation of Stop and Copy

- We need to find all the reachable objects, as for mark and sweep
- As we find a reachable object we copy it into the new space
  - And we have to fix ALL pointers pointing to it!
- As we copy an object we store in the old copy a forwarding pointer to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied
Implementation of Stop and Copy (Cont.)

• We still have the issue of how to implement the traversal without using extra space
• The following trick solves the problem:
  – partition the new space in three contiguous regions

start → copied and scanned → copied → empty

- copied objects whose pointer fields were followed
- copied objects whose pointer fields were NOT followed
Stop and Copy. Example (1)

- Before garbage collection
Stop and Copy. Example (3)

- Step 1: Copy the objects pointed by roots and set forwarding pointers
Stop and Copy. Example (3)

- Step 2: Follow the pointer in the next unscanned object (A)
  - copy the pointed objects (just C in this case)
  - fix the pointer in A
  - set forwarding pointer
Stop and Copy. Example (4)

- Follow the pointer in the next unscanned object (C)
  - copy the pointed objects (F in this case)
Stop and Copy. Example (5)

- Follow the pointer in the next unscanned object (F)
  - the pointed object (A) was already copied. Set the pointer same as the forwarding pointer
Stop and Copy. Example (6)

- Since scan caught up with alloc we are done
- Swap the role of the spaces and resume the program
The Stop and Copy Algorithm

while (scan != alloc) {
    O = the object at scan pointer
    for (each pointer p contained in O) {
        find O’ that p points to
        if (O’ is without a forwarding pointer) {
            copy O’ to new space (update alloc pointer)
            set old O’ to point to the new copy
            change p to point to the new copy of O’
        } else {
            set p in O equal to the forwarding pointer
        }
    }
    increment scan pointer to the next object
}
Stop and Copy. Details.

• As with mark and sweep, we must be able to tell how large is an object when we scan it
  – and we must also know where are the pointers inside the object

• We must also copy any objects pointed to by the stack and update pointers in the stack
  – this can be an expensive operation
Stop and Copy. Evaluation

• Stop and copy is generally believed to be the fastest GC technique

• Allocation is very cheap
  – just increment the heap pointer

• Collection is relatively cheap
  – especially if there is a lot of garbage
  – only touch reachable objects

• But some languages do not allow copying (C, C++)
Why Doesn’t C Allow Copying?

• Garbage collection relies on being able to find all reachable objects
  – and it needs to find all pointers in an object

• In C or C++ it is impossible to identify the contents of objects in memory
  – E.g., how can you tell that a sequence of two memory words is a list cell (with data and next fields) or a binary tree node (with a left and right fields)?
  – Thus we cannot tell where all the pointers are
Conservative Garbage Collection

• But it is Ok to be conservative:
  – if a memory word looks like a pointer it is considered a pointer
    • it must be aligned
    • it must point to a valid address in the data segment
  – all such pointers are followed and we overestimate the reachable objects

• But we still cannot move objects because we cannot update pointers to them
  – what if what we thought to be a pointer is actually an account number?
Algorithm 3: Reference Counting
Reference Counting

- Rather that wait for memory to be exhausted, try to collect an object when there are no more pointers to it
- Store in each object the number of pointers to that object
  - this is the reference count
- Each assignment operation has to manipulate the reference count

- C++: smart pointers (boost library), memory header (C++11)
  - Requires writing code to explicitly transfer object ownership
Implementation of Reference Counting

- new returns an object with a reference count of 1
- If x points to an object then let rc(x) point to its reference count
- Every assignment \( x = y \) must be changed:

```c
// increase ref count of obj pointed to by y
rc(y) = rc(y) + 1
// reduce ref count of obj pointed to previously by x
rc(x) = rc(x) - 1
if(rc(x) == 0) { mark x as free }
x = y   // perform actual assignment
```
Reference Counting Evaluation

• Advantages:
  – easy to implement
  – collects garbage incrementally without large pauses in the execution

• Disadvantages:
  – cannot collect circular structures
  – manipulating reference counts at each assignment is very slow
Garbage Collection Evaluation

- Automatic memory management avoids some serious storage bugs
- But it takes away control from the programmer
  - e.g., layout of data in memory
  - e.g., when is memory deallocated
- Most garbage collection implementation stop the execution during collection
  - not acceptable in real-time applications
Garbage Collection Evaluation

- Garbage collection is going to be around for a while
- Researchers are working on advanced garbage collection algorithms:
  - concurrent: allow the program to run while the collection is happening
  - generational: do not scan long-lived objects at every collection
  - parallel: several collectors working in parallel