Instruction Scheduling

Reorder instructions to better fit target machine's pipeline

- fill control transfer delay slots
- avoid using result of multi-cycle operations too early
  - loads, floating point operations, ...
- schedule code for VLIW, superscalar machines
  - coordinate multiple instructions to fit available machine resources

Techniques:

- **list scheduling**, in a basic block
- **trace scheduling**, across conditional branches
- **software pipelining**, across loop iterations

Loop unrolling often can help scheduling

Register allocation can hurt scheduling

List scheduling

[Gibbons & Muchnick 86]

Schedule a basic block...

- obeying data dependences
- avoiding interlocks

Previous work: exponential, $O(n^4)$ algorithms

This work: $O(n^2)$ algorithm, simple

Pipeline model

Hazards considered:

- load followed by use of target of load
- store followed by a load
- load followed by
  ALU op or load/store with address calculation

```
r2 := r1 + 1
sp := sp - 12
*A := r0
r3 := *(sp+4)
r4 := *(sp+8)
sp := sp - 8
*sp := r2
r5 := *A
r4 := r0 + 1
```

Step 1: construct data dependence graph

Convert linear basic block into a DAG representing data dependences

Loads & stores assumed to alias,
except that different offsets from common base reg (e.g. `sp`) do not alias

```
① r2 := r1 + 1
② sp := sp - 12
③ *A := r0
④ r3 := *(sp+4)
⑤ r4 := *(sp+8)
⑥ sp := sp - 8
⑦ *sp := r2
⑧ r5 := *A
⑨ r4 := r0 + 1
```

Step 2: traverse dependence graph, emitting code

Maintain set of **candidate nodes**
whose data dependence predecessors have been emitted

```
candidates := roots of DAG
while |candidates| > 0 do
   select best available candidate node
   emit it
   remove it from the DAG
   add any new root nodes to candidate set
```

Best node:

1. doesn't interlock with previous instruction, or
2. does interlock with an immediate successor node, or
3. has the most immediate successor nodes, or
4. is along the longest path to the leaves of the DAG

[Previous work used lookahead in DAG to guide choice
⇒ complex and slow (worst case)]

Results

["Effectiveness of a Machine-Level Global Optimizer",
Johnson & Miller, PLDI '86]

Compiling small benchmark programs: 7% improvement

Automatic Garbage Collection

Automatically free dead objects

- no **dangling pointers**, no **storage leaks** (maybe)
- can have faster allocation, better memory locality

General styles:

- reference counting
- tracing
  - mark/sweep, mark/compact
  - copying

Adjectives:

- generational
- conservative
- incremental
- parallel
- distributed

Reference counting

For each heap-allocated object,
maintain count of # of pointers to object

- when create object, ref count = 0
- when create new ref to object, increment ref count
- when remove ref to object, decrement ref count
- if ref count goes to zero, then delete object

```
proc foo() {
  a := new Cons;
  b := new Blob;
  c := bar(a, b);
  return c;
}

proc bar(x, y) {
  l := x;
  l.head := y;
  t := l.tail;
  return t;
}
```

Evaluation of reference counting

+ local, incremental work
+ little/no language support required
+ local ⇒ feasible for distributed systems

– cannot reclaim cyclic structures
– uses malloc/free back-end ⇒ heap gets fragmented
– high run-time overhead (10-20%)
  - can delay processing of ptrs from stack
    (deferred reference counting [Deutsch & Bobrow 76])
– space cost
– no bound on time to reclaim

Tracing collectors

Start with a set of **root** pointers
- global vars
- contents of stack & registers

Traverse objects transitively from roots
- visits **reachable** objects
- all unvisited objects are garbage

Issues:
- how to identify pointers?
- in what order to visit objects?
- how to know an object is visited?
- how to free unvisited objects?
- how to allocate new objects?
- how to synchronize collector and program (**mutator**)?

Identifying pointers

“**Accurate**”: always know unambiguously where pointers are
Use some subset of the following to do this:
- static type info & compiler support
- run-time tagging scheme
- run-time conventions about where pointers can be

Conservative [Bartlett 88, Boehm & Weiser 88]:
assume anything that looks like a pointer might a pointer,
& mark target object reachable
+ supports GC of C, C++, etc.

What “looks” like a pointer?
- most optimistic: just aligned pointers to beginning of objects
- what about interior pointers?
  off-the-end pointers?
  unaligned pointers?

Miss encoded pointers (e.g. xor’d ptrs), ptrs in files, ...

Mark/sweep collection

[McCarthy 60]: stop-the-world tracing collector

Stop the application when heap fills

Trace reachable objects
- set mark bit in each object
- tracing control:
  - depth-first, recursively using separate stack
  - depth-first, using pointer reversal

Sweep through all of memory
- add unmarked objects to free list
- clear marks of marked objects

Restart mutator
- allocate new objects using free list

Evaluation of mark/sweep collection

- \+ collects cyclic structures
- \+ simple to implement

- – “embarrassing pause” problem
- – poor memory locality
  - when tracing, sweeping
  - when allocating, dereferencing due to heap fragmentation
- – not suitable for distributed systems

Some improvements

Mark/**compact** collection:
when sweeping through memory, compact rather than free

- all free memory in one block at end of memory space;
  no free lists
- \+ reduces fragmentation
- \+ fast allocation
- – slower to sweep
- – changes pointers
  ⇒ requires accurate info about pointers

Generational mark/∗
Incremental and/or **parallel** mark/∗

- \+ (greatly) reduce embarrassing pause problem
- \+ may be suitable for real-time collection
- – more complex

Copying collection

[Cheney 70]

Divide heap into two equal-sized **semi-spaces**

- mutator allocates in **from-space**
- **to-space** is empty

When from-space fills, do a GC:

- visit objects referenced by roots
- when visit object:
  - copy to to-space
  - leave forwarding pointer in from-space version
  - if visit object again, just redirect pointer to to-space copy
- scan to-space linearly to visit reachable objects
  - to-space acts like breadth-first-search work list
- when done scanning to-space:
  - empty from-space
  - **flip**: swap roles of to-space and from-space
- restart mutator

Evaluation of copying collection

- \+ collects cyclic structures
- \+ supports compaction, fast allocation automatically
- \+ no separate traversal stack required
- \+ only visits reachable objects, not all objects

- – requires twice the (virtual) memory,
  physical memory sloshes back and forth
  - could benefit from OS support
- – “embarrassing pause” problem still
- – copying can be slow
- – changes pointers

An improvement

Add small **nursery** semi-space [Ungar 84]

- nursery fits in main memory (or cache)
- mutator allocates in nursery
- GC when nursery fills
  - copy nursery + from-space to to-space
  - flip: empty both nursery and from-space

+ reduces cache misses, page faults
  - most heap memory references satisfied in nursery?

– nursery + from-space can overflow to-space

– more complex

Another improvement

Add semi-space for large objects [Caudill & Wirfs-Brock 86]

- big objects slow to copy, so allocate them in separate space
- use mark/sweep in large object space

+ no copying of big objects

– more complex

Generational GC

Observation:
most objects die soon after allocation

- e.g. closures, cons cells, stack frames, numbers, ...

Idea:
concentrate GC effort on young objects

- divide up heap into 2 or more generations
- GC each generation with different frequencies, algorithms

Original idea: Peter Deutsch
Generational mark/sweep: [Lieberman & Hewitt 83]
Generational copying GC: [Ungar 84]

Generation scavenging

A generational copying GC [Ungar 84]

2 generations: **new-space** and **old-space**

- new-space managed as a 3-space copying collector
- old-space managed using mark/sweep
- new-space much smaller than old-space

Apply copy collection (**scavenging**) to new-space frequently

If object survives many scavenges, then copy it to old-space

- **tenuring** (a.k.a. **promotion**)
- need some representation of object’s age

If old-space (nearly) full, do a full GC

Roots for generational GC

Must include pointers from old-space to new-space as roots when scavenging new-space

How to find these?

Option 1: scan old-space at each scavenge

Option 2: track pointers from old-space to new-space

Tracking old→new pointers

How to remember pointers?

- individual words containing pointers [Hosking & Moss 92]
- **remembered set** of objects possibly containing pointers [Ungar 84]
- **card marking** [Wilson 89]

How to update table?

- functional languages: easy!
- imperative languages: need a **write barrier**
  - specialized hardware
  - standard page protection hardware
  - in software, inserting extra checking code at stores

Evaluation of generation scavenging

+ scavenges are short: fraction of a second
+ low run-time overhead
  - 2-3% in Smalltalk interpreter
  - 5-15% in optimized Self code
+ less VM space than pure copying
+ better memory locality than pure mark/sweep

– requires write barrier
– still have infrequent full GC’s
– need space for age fields
  - some solutions in later work

Extensions

Multiple generations

- e.g. Ephemeral GC: 8 generations [Moon 84]
- many generations obviates need for age fields

Feedback-mediated tenuring policy [Ungar & Jackson 88]

Large object space

Incremental & parallel GC

Avoid long pause times by running collector & mutator in parallel
- physical or simulated parallelism

Main issue: how to synchronize collector & mutator?
- read barrier [Baker 78, Moon 84]
- write barrier [Dijkstra 78; Appel, Ellis & Li 88]

Implementing Functional Languages

e.g. Lisp, Scheme, ML, Haskell, Miranda

Lisp and Scheme: dynamic typing
⇒ uniform “boxed” representation of all data objects, tagged pointers to encode some types (e.g. ints) cheaper

ML, Haskell, Miranda: polymorphic static typing
⇒ uniform “boxed” representation....
- “unboxing”: choose better data layout where possible

First-class, lexically-nested functions
- static scoping of nested functions
  ⇒ closures to represent function values
- functions can outlive defining scope
  ⇒ heap-allocated environments
- calls of computed expressions
  ⇒ (fancier) call graph analysis

Heavy use of recursion instead of iteration
⇒ tail call, tail recursion elimination

Immutable update-by-copy data structures
⇒ version arrays, compile-time reference counting

Miranda & Haskell: lazy evaluation
⇒ strictness analysis

Implementing higher-order functions

Functions are first-class data values
- passed as arguments, returned from fns, stored in data structures
- potentially anonymous
- lexically-scoped

Example:

```
(define mul-by (lambda lst n)
  (map (lambda (x) (* x n)) lst))
```

2 components of a function value (a **closure**):
- code pointer
- lexically-enclosing environment pointer

Steps in deciding how to implement a closure:
- **strategy analysis**: where to allocate closure
- **representation analysis**: how to lay out data structure

Strategy analysis

Option 1: heap allocation
- \+ most general option
- \+ simple decision to make
- – expensive to create, invoke, and reclaim closure
- – may require heap-allocation of lexically-enclosing env

Supports “upward funargs”

Example:

```
(define add (lambda x) (lambda (y) (+ x y)))
(define inc (add 1))
(define dec (add -1))
(print (inc (dec 3)))
```

Stack allocation

Option 2: stack allocation

If closure's dynamic extent is contained within the extent of its lexically-enclosing activation record, then can allocate closure as part of a.r.'s stack frame (a LIFO closure)

+ faster allocation, free reclamation
+ enclosing environment can be stack-allocated

– invocation still slow
– restricted applicability

Inlining calls to closures

Option 3: represent closure in-line

If invoking a known closure, inline-expand body

If all uses of known closure inlined away, don't create closure

- closure's environment turns into local variables

+ free allocation, fast invocation, free reclamation

– limited applicability

Enables closure-based user-defined control structures

Escape analysis

Determine if closure (or any data structure) has LIFO extent, i.e. does not **escape** stack frame

+ use stack allocation for non-escaping data structures

Track flow of value, see where it goes

Has LIFO extent (i.e., doesn't escape):

- when created
- when assigned to local variable
- when invoked

A hard case:

- passed as argument to function
  - if intraprocedural analysis: escapes
  - if interprocedural analysis: may or may not escape

Harder cases:

- returned
- stored in global/non-local variable or (escaping) data structure

Assume escapes

Interprocedural escape analysis

Compute for each formal parameter whether that parameter **escapes**

Construct program's call graph

Initialize all formals to "does not escape"

Initialize worklist to empty set

Process each function:
    if formal parameter labeled "does not escape"
        escapes locally within this function,
    change formal to "escapes" and put all callers on worklist

While worklist non-empty:
    remove function from worklist, reprocess

- at call site, actual argument escapes if corresponding formal escapes

Representation analysis

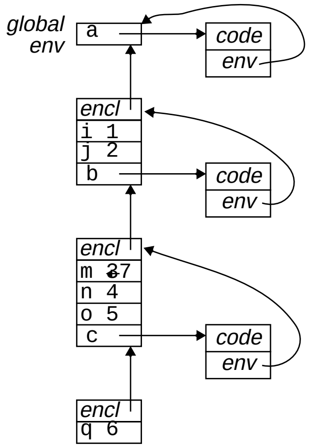
How to represent closure's lexical environment?

Option 1: **deep binding**

- represent environment as pointer to shared env record

Example of deep binding

```
(define a (lambda (i j)
  (define b (lambda m n o)
    (define c (lambda q)
      (+ q m i)) ;; here
    (set! m 7)
    (c 6))
  (b 3 4 5))
(a 1 2)
```



Representation analysis, cont

Option 2: **shallow binding**

- copy needed values into environment when created

Option 3: **very shallow binding**

- copy needed values into closure itself

Cannot copy values of mutable variables
⇒ do **assignment conversion** first

- replace mutable variable with pointer to heap-allocated reference cell

\+ can copy the pointer freely

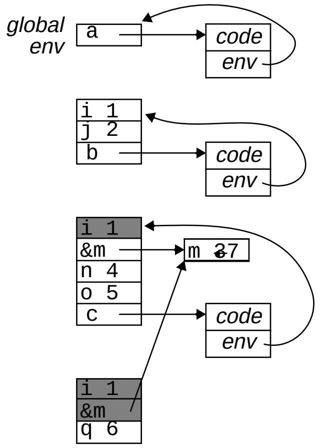
– space overhead

– extra indirection

⇒ best for mostly functional code

Example of shallow binding

```
(define a (lambda (i j)
  (define b (lambda m n o)
    (define c (lambda q)
      (+ q m i)) ;; here
    (set! m 7)
    (c 6))
  (b 3 4 5))
(a 1 2)
```



Comparison

Deep binding:

- \+ simple
- \+ space-efficient
- \+ fast to create closure
- – slow to access lexically enclosing vars

Shallow binding:

- \+ fast access to lexically enclosing vars
- \+ may not need to heap-allocate enclosing environment
- – slower closure creation
- – more space consuming, if >1 var needed
- – requires assignment conversion

Very shallow binding: like shallow binding, but:

- \+ even faster access to enclosing vars
- – even slower closure creation, if >1 var needed
- – even more space consuming, if >1 var or >1 closure needed