Post-Compiler
Code Transformation

David W. Wall

DEC Western Research Laboratory

"Real Men hack a.outs."
— James R. Larus

Norman Ramsey

What is it? Why do it?

The building blocks

The general technique

System overviews

Pitfalls

Applications

The basic idea

Read the program, make changes, write a new version

Easier than doing it to a source program:
- no big parsing problem
- everything's in the same ''language''
- can include libraries if desirable

Harder than doing it to a source program:
- have to understand grungy details
- addresses of things matter

Why do it?

Optimization
- peephole optimization
- cache line allocation
- intermodule register allocation
- procedure integration

Instrumentation
- basic-block counting
- edge counting
- address tracing

Translation

Who does it?

Mahler [Wall91a]: Register allocation, pipeline scheduling, instrumentation

Moxie [Himelstein+87]: Compiled simulation

Pixie [MIPS89]: Block-counting, address tracing

Qp [LarusBall92]: Edge-counting, address tracing

MTOOL [GoldbergHennessy92]: Performance instrumentation

VEST [Digital92]: Architectural translation

Postloading [Johnson90]: Lots of stuff

Purify [HastingsJoyce92]: Memory use error detection

Outline

What is it? Why do it?

The building blocks

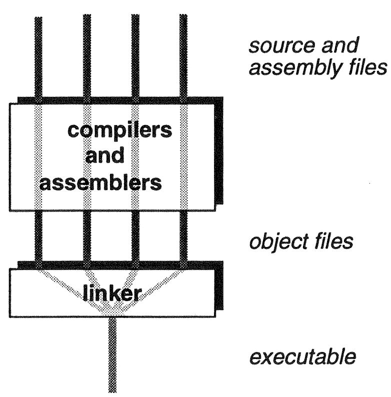
The general technique

System overviews

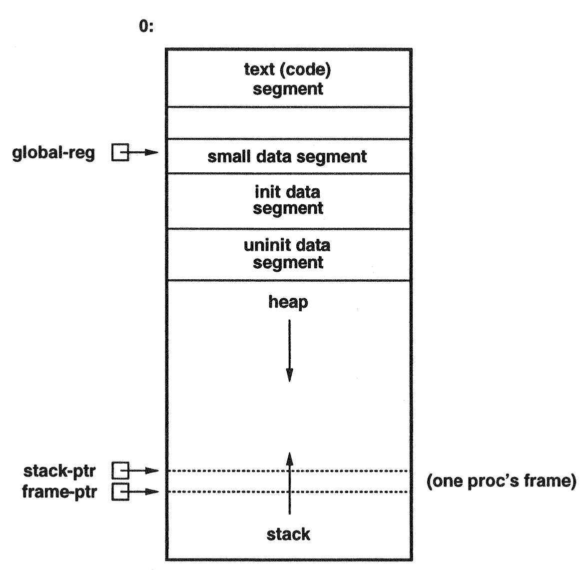
Pitfalls

Applications

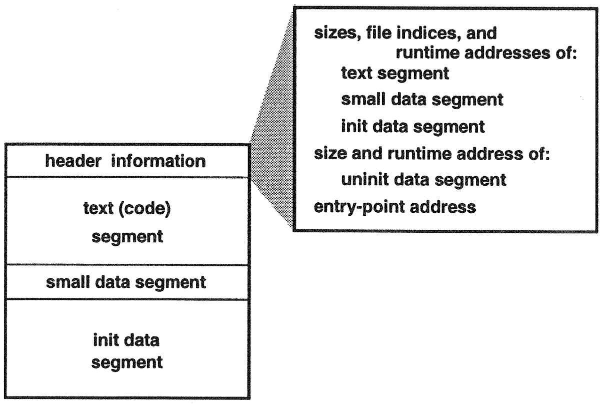
Compiling and linking



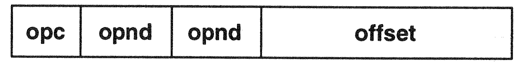
Typical memory layout



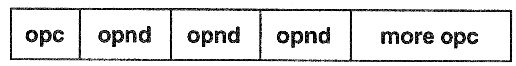
Typical executable format



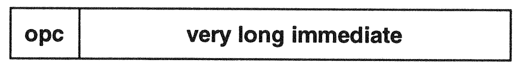
Instruction formats



offset may be word, byte, or immediate



three-operand register operations

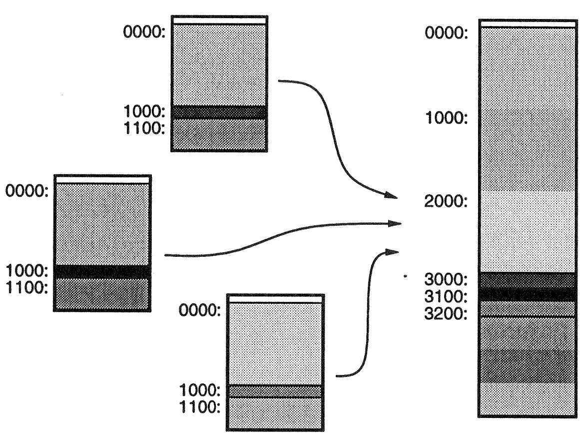


mainly used for long jumps and calls

What a linker does

Resolves external references in modules

Combines corresponding segments of objects into segments of executable



Object files are marked-up executables

```
extern Radix (double);
extern double delta;
static int counts;
Prefetch () {
   Query(counts);
}
Query (int i); {
   Radix (delta);
}
```

```
extsym Prefetch:
   0x104: (start of Prefetch)
            . . .
   0x168: load  r1,20(gp)   [displ sdata]
   0x16c: jal   0x320       [jdest text]
            . . .
extsym Query:
   0x320: (start of Query)
            . . .
   0x440: load  r1,0(gp)    [displ ext:delta]
   0x444: load  r2,4(gp)    [displ ext:delta]
   0x448: jal   0x0         [jdest ext:Radix]
```

Linking algorithm

Step through objects noting segment sizes

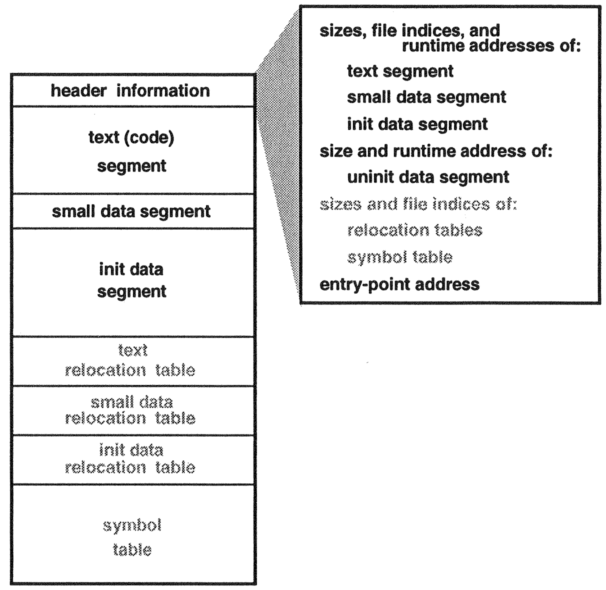
Plan where each object segment goes in final executable

Note final value of each external symbol

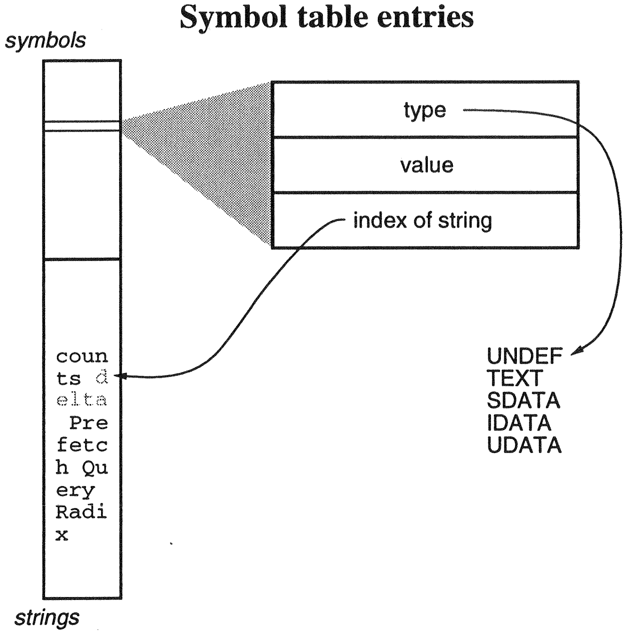
Step through objects again, putting pieces in place

Relocate internal and external address references

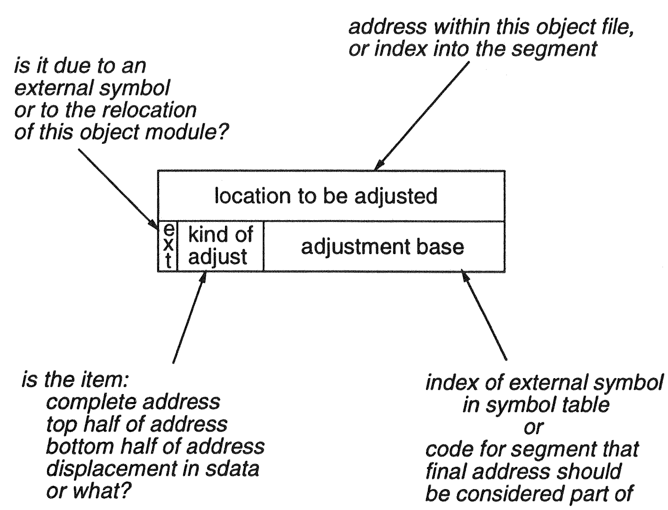
Typical object format



Symbol table entries



Relocation records



Relocation algorithm

location to be adjusted		
e x t	kind of adjust	adjustment base

Find the word at the specified location

Use kind to determine field of word to relocate

If relocation is external, then:

> Add value of adjustment symbol to relocated field

Else:

> Interpret field value as address in adjustment segment of object, and reinterpret it as address in executable

digital Western Research Laboratory

Outline

What is it? Why do it?

The building blocks

The general technique

System overviews

Pitfalls

Applications

digital Western Research Laboratory

What's hard?

Inserting or deleting code changes addresses

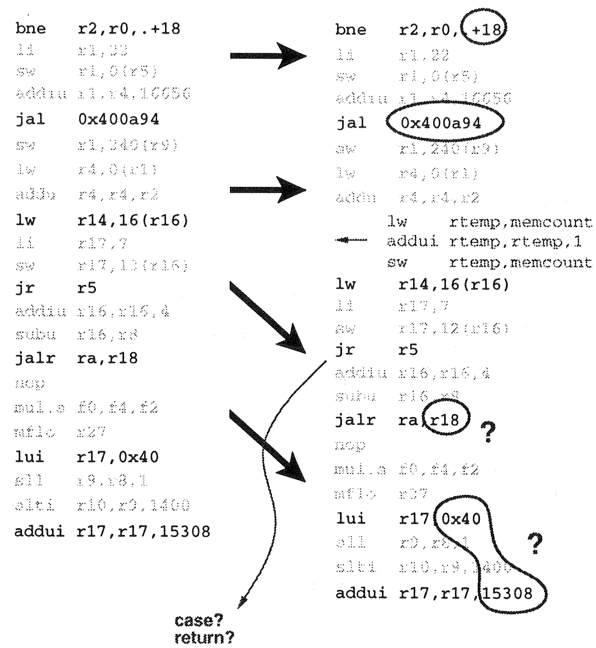
How do we get new resources?

How do we deduce the control structure?

Architectural considerations:

- branch/jump ranges
- multi-instruction operations
  - bgt(r8,r9,L) into slt(r1,r9,r8);bnz(r1,L)
  - la(r4,0x10007d30) into lui(r4,0x1000);add(r4,r4,0x7d30)
- pipeline constraints
- delayed branches

digital Western Research Laboratory



digital Western Research Laboratory

Mapping old addresses to new

Build table mapping old addresses to new

- keep track of insertions and deletions
- each old address corresponds to a new address

Insertion *after* or *before*?

```
sw    r1,240(r9)
lw    r4,0(r1)
addu  r4,r4,r2
lw    r14,16(r16)
li    r17,7
sw    r17,12(r16)
addiu r16,r16.4
subu  r16,r8
bne   r2,r0,.-5
```

```
sw    r1,240(r9)
lw    r4,0(r1)
addu  r4,r4,r2
      lw    rtemp,memcount
      addui rtemp,rtemp,1
      sw    rtemp,memcount
lw    r14,16(r16)
li    r17,7
sw    r17,12(r16)
addiu r16,r16.4
subu  r16,r8
bne   r2,r0,.-??
```

?

-5? or -8?

Western Research Laboratory

Two kinds of address translation

Are relocation tables still available?

Static address translation:

- use relocation tables to identify addresses
- translate them during code transformation

Dynamic address translation:

- build table literally into modified program
- translate addresses when used during modified execution

Western Research Laboratory

Static address translation

```
bne   r2,r0,.+18
```

inst.disp := xl[oldaddr+inst.disp] - newaddr

```
jal   0x400a94  [jdest  text]
```

inst.dest := xl[inst.dest]

```
jal   0x0  [jdest  ext:P3]
```

leave it alone, but leave it relocated

```
lui   r17,0x40  [hi16  text]
addui r17,r17,15308  [lo16  text]
```

dest := inst1.disp << 16 + inst2.disp
dest := xl[dest]
inst2.disp := (dest << 16) >> 16
inst1.disp := (dest - inst2.disp) >> 16

```
(data) 0x403bcc  [word text]
```

change to xl[self]

```
jr    r5
jalr  ra,r18
```

leave them alone; code to generate address used was translated statically

Western Research Laboratory

Dynamic address translation

No relocation, so must be conservative

Translate branches and direct jumps statically

Must include translation table in transformed program

Precede all indirect jumps by code to compute new address

Do *all* address computation in old space

Western Research Laboratory

Dynamic translation of procedure call/return

```
0x100:  (start of proc)
    . . .
0x188: jr   r31

    . . .

0x428: jal  r31,0x100
0x42c:
```

```
0x140:  (start of proc)
    . . .
0x22c: lw   r23,xl[r31]
0x230: jr   r23
    . . .

0x5c0: la   r31,0x42c
0x5c4: jump 0x140
0x5c8:
```

xl maps old addresses to new:

```
0x100 → 0x140
0x188 → 0x22c
0x428 → 0x5c0
0x42c → 0x5c8
```

digital Western Research Laboratory

Don't forget data addresses

Making code bigger may change data addresses

- with relocation tables, no problem
- leaving lots of room in code segment in the first place helps
- leave entire image at original address and put translation elsewhere
- worst case: dynamic translation at loads and stores

digital Western Research Laboratory

How do we get new resources?

Registers

- save/restore them, or
- emulate them, or
- allocate them

Memory

- allocate on top of stack at program start
- copy program arguments to new location

digital Western Research Laboratory

Deducing the control structure

Finding basic blocks

- see next slide

Finding control paths

- recognize code generation patterns
- is symbol table useful?
- can code generator help?
- what about assembly code?

digital Western Research Laboratory

Finding basic block boundaries

Find:

- branch destinations
- jump destinations
- load-addresses of text address
- text addresses in data
- (text addresses in symbol table)

Relocation tables guarantee answers

Without them, must be conservative

Recognizing code patterns

Main goal is to understand indirect jumps

- allows flow graph construction
- allows more static address translation

Where do indirect jumps come from?

- subroutine return
- case statement
- calling a procedure variable
- FORTRAN assigned goto (yecch)

Subroutine return

Machine may have explicit RETURN instruction

Machine may *encourage* use of one return register

Symbol table may declare proc's return register

MIPS codegen always uses r31 as return register

Case statement

Canonical code for case jump:

```
bge    t6,CaseCount,L  --if not in range, skip
sll    t6,t6,2         --conv. to byte offset
lw     t6,Table(t6)    --ld destination addr
jr     t6              --jump to selected case
```

Tells us address and size of table

More validity checks:

- table is in read-only memory
- addresses in table are in current procedure
- table not referenced elsewhere in code

FORTRAN assigned goto

```
    assign 99 to X
    ...
    goto X,(99,100,4,29)
    ...
99
```

Destination list invisible at machine level

Compiles to a jr with little context

Can symbol table tell you if you're looking at FORTRAN code?

Can compiler tell you where these occur?

Fortunately, nearly obsolete

digital Western Research Laboratory

Calling a procedure variable

Main tipoff: jump is jalr instruction

But jr might be used for *tail-call*

Putting it together

If exactly one pattern matches, great

Treat any unmatched jr as a tail-call

Treat any address taken as an entry point

digital Western Research Laboratory

There's still assembly code

Mainly in standard libraries

Mainly follows the usual conventions

Things you can do
- treat (bad) assembler routines as black boxes
- rewrite to adopt standard conventions
- recognize problem library routines personally
- augment assembler to document violations
- raise assembler to exclude violations

digital Western Research Laboratory

Patterns help address translation

If we can split off a set of
- (guaranteed) address specifications
- jumps

such that
- the jumps go only to those addresses
- no other jumps go to those addresses

Then we can translate these statically

Examples:
- (trivially) branches and direct jumps
- jump via address table for case statement
- procedure call/return for non-addressed procedures

digital Western Research Laboratory

Typical phases

Break code into basic blocks

Categorize the jumps

Link blocks into flow graph

Generate preface code

Build new version of code [and code relocation table], one block at a time

Compute mapping from old addresses to new

Adjust destinations of branches and jumps where needed

[Adjust code addresses marked for relocation in code or data]

Adjust addresses in loader symbol table

Outline

What is it? Why do it?

The building blocks

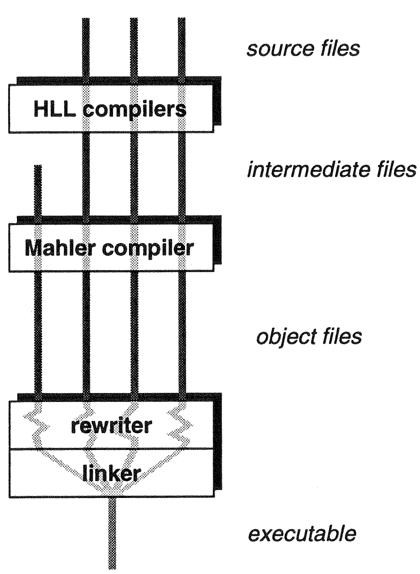
The general technique

System overviews

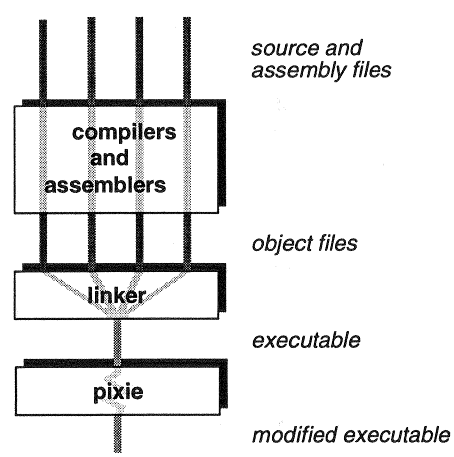
Pitfalls

Applications

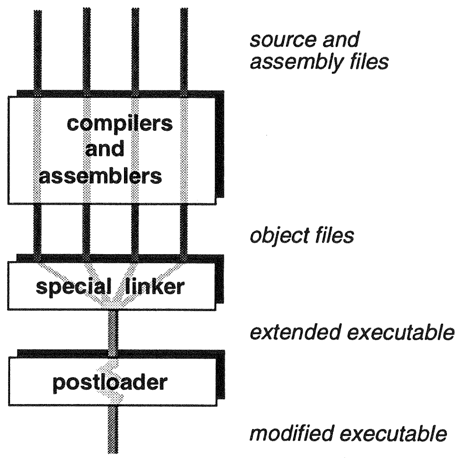
Mahler



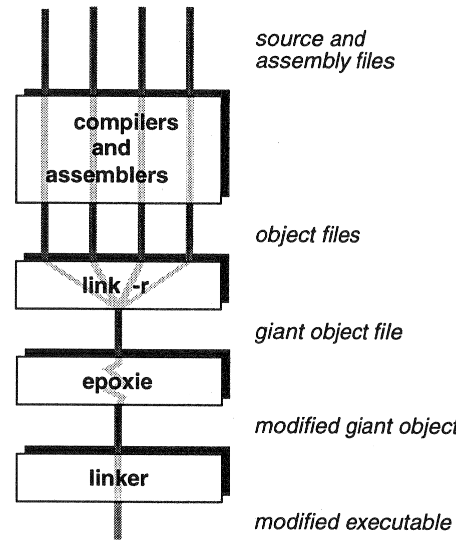
Pixie (and kin)



Johnson's postloader



Epoxie



Outline

What is it? Why do it?

The building blocks

The general technique

System overviews

Pitfalls

Applications

Pitfalls

Code in data segment

Data in code segment

Delayed branches

System calls

Signals

Code in data segment

How does it arise?

- constructed instructions
- ''compiled'' arguments
- *trampolines* to dynamically-linkable code

Getting there

- static address translation works fine
- dynamic translation dies
- include range test in dynamic translation

Modifying it

- if you can recognize it, maybe no problem
- but why is it in data at all?

Data in code segment

Literal constants and jump tables

Don't hurt it:

- don't ''correct'' it
- don't instrument it

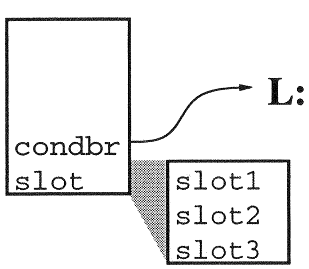
Find it:

- symbol table?
- reachability analysis?

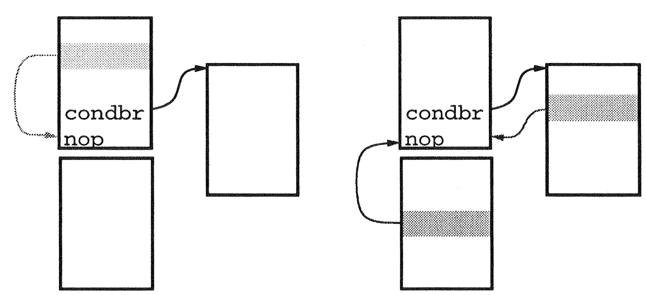
Make sure it's accessible!

- with relocation: even data references to code segment get fixed
- without relocation: include unmodified code segment in new segment

Delayed branches



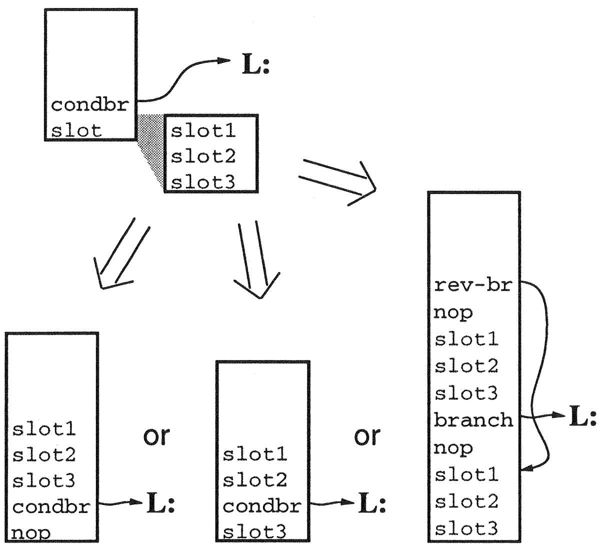
How do slots get filled?



slot filled from same basic block

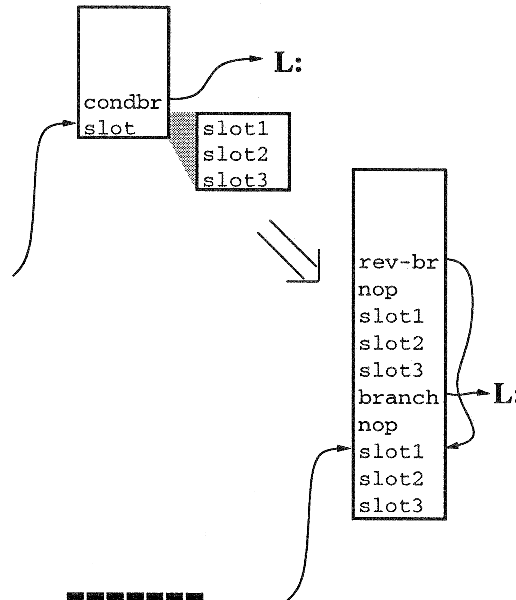
slot filled from a successor block, if safe

The solution



Western Research Laboratory

Branches to branch slots



Western Research Laboratory

System calls and signal handling

Identifying a system call is probably easy

```
loadimm   r2,NNN
syscall
```

Some registers may be officially damageable

Code addresses may be passed, e.g. for signal handling

- techniques using relocation work fine
- other techniques must recognize system call

Emulated registers not seen by kernel

May want to recognize *exit* system call

Western Research Laboratory

Outline

What is it? Why do it?

The building blocks

The general technique

System overviews

Pitfalls

Applications

Western Research Laboratory

Applications

Instruction-level instrumentation

Address tracing

Source-level profiling

Pipeline scheduling

Intermodule register allocation

Architecture translation

digital Western Research Laboratory

Pixie: Basic block counting

Break program into basic blocks

Precede each block with

```
load    rx,N(rz)
addimm  rx,rx,1
store   rx,N(rz)
```

where N is the index of this block

How do you count labeled branch slots?

digital Western Research Laboratory

Why do you want basic block counts?

Combine with static info about the blocks

Count loads, stores, or other instructions

And (symbol table permitting)

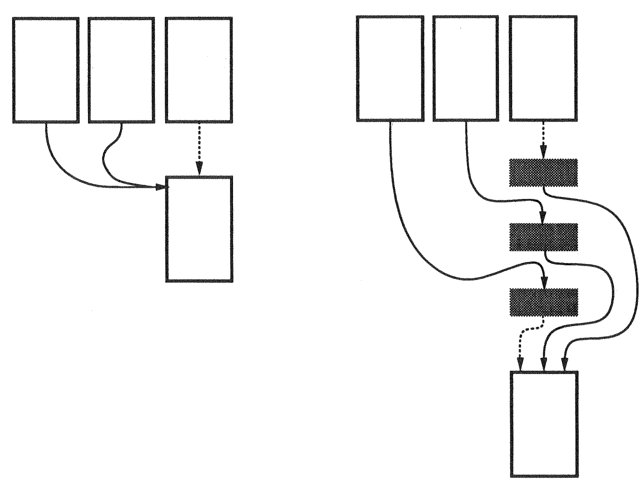
- count uses of variables or procedures
- count executions of source lines

digital Western Research Laboratory

Qp: Edge counting

Break program into basic blocks

Insert counting code for each *edge*.



digital Western Research Laboratory

Advantages of edge counting

Block counts can be derived from edge counts

Edge counts tell you more
- count branches *taken*
- count pipeline stalls

Most program graphs are planar, so |E| ~ |V|

Permits an interesting optimization:
- estimate execution counts of edges
- find maximum spanning tree of edge graph
- don't count those edges

Address tracing

Useful to simulate cache/memory/IO hierarchies

Precede

```
load    r1,d(r2)
```

by code to compute

```
r2+d
```

and append it to a log

Same for stores

Instrument basic blocks for instruction addresses
- but log the old code address, not the new one

Periodically do something with the accumulated logs

Source-level profiling

Procedure entry counts are easy
- basic block counts
- procedure entry points from symbol table

Approximate procedure call edge counts are easy, too
- basic block counts
- identify calls and see where they come from and go to
- doesn't count calls via procedure variable

Instrument procedure entry with code that examines return address
- gprof uses such a traceback inserted by compiler
- easy to insert it post-compiler

Goldberg's bottleneck profiling

How can we identify performance losses due to cache thrashing, multiprocess synchronization, etc?
- instrument program to count basic blocks, and predict a run time
- instrument program to time basic blocks
- compare the two

Except you can't really time basic blocks

Analyze structure of program to find candidate chunks

Purify: Detecting memory use errors

Modify object modules to detect

- reads of uninitialized locations
- references to unallocated locations
- allocated but unreachable locations

Maintain a big table telling what bytes have been assigned and allocated

Insert code at stores, mallocs, and frees to change state of bytes

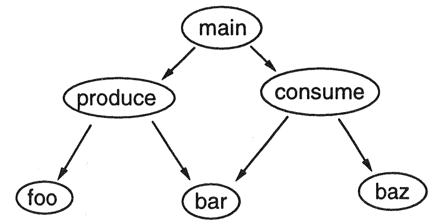
Insert code at loads and stores to check for illegal state

Insert dbx-callable procedure to do mark-and-sweep garbage detection

digital Western Research Laboratory

Intermodule register allocation

Call-return discipline can guide allocation of locals



Analyzing whole program tells which globals are safe

Remove loads and stores, and rename registers used

May require cooperation from compiler

digital Western Research Laboratory

Pipeline scheduling

Bunch of work on the problem:

- scheduling within a basic block [HennessyGross83,GibbonsMuchnick86]
- speculative scheduling across a branch [WallPowell87, e.g.]
- flow-based global scheduling [BernsteinRodeh91]
- scheduling vs register allocation [GoodmanHsu88,Bradlee+91]

Reasons to do it so late:

- interaction with very global register allocator
- scheduler itself may be very global

Don't do this using a technique with overhead

digital Western Research Laboratory

Architectural translation

Hardware bug workarounds

- alter code to avoid pipeline bugs

Fast simulation of new hardware

- moxie: from MIPS to VAX

Translation to new hardware

- VEST: from VAX to Alpha

Translation has to somehow be *complete*

digital Western Research Laboratory

The big finish

There's a lot you can do with late code modification

Help from compiler/loader makes it easier

Instrumentation < optimization < translation

It's amazing how some people will spend their time

Moxie: backward translation

Wanted to test MIPS compilers before hardware exists

Simulation is too slow for thorough testing

Moxie translates:

- MIPS code into VAX code
- MIPS Unix calls into VAX Unix calls
- MIPS loader format to VAX loader format

Like using VAX instruction set to microcode a very slow ''MIPS processor''

This approach led directly to pixie

VEST: forward translation

Translate VAX (user-mode) executables without sources to Alpha

Programs should never slow down

Problems:

- nobody planned for this fifteen years ago
- CISC architecture not conducive to translation
  - different length instructions
  - a lot of state (condition codes, stack top, etc.) that may not be relevant
- VAX executables not necessarily well-behaved
  - code and data interspersed (or overlap)
  - branches to middle of instructions (!)
  - self-modifying code

Vintage Environment Software Translator

Find entry points from header and symbol table (if present)

Follow threads of control (incl. idiomatic jumps)

Separate into basic blocks and build flow graph

Push context information around flow graph

Generate Alpha code for each basic block

Details and consequences

Original VAX image is included in translation

Full VAX instruction interpreter is also included

Jumps to unknown location are translated as calls to interpreter

Interpreter runs much slower, but usually needs to do only a few instructions

Interpreter provides feedback for a later translation

Profiling original VAX code also provides feedback

Recompiling to native makes it 2x or 3x faster, but...

RISC/CISC speed difference means no *loss* of performance

[BallLarus91] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *Nineteenth Annual ACM Symposium on Principles of Programming Languages,* pp. 59-70, January 1992.

[BenitezDavidson88] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pp. 329-338. Published as *SIGPLAN Notices 23* (7), July 1988.

[BernsteinRodeh91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation,* pp. 241-255. Published as *SIGPLAN Notices 26* (6), June 1991.

[Borg+89] Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall. Long address traces from RISC machines: Generation and analysis. *Seventeenth Annual International Symposium on Computer Architecture,* pp. 270-279, May 1990. A more detailed version is available as WRL Research Report 89/14, September 1989.

[Bradlee+91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. *Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems,* pp. 122-131, April 1991. Published as *Computer Architecture News 19* (2), *Operating Systems Review 25* (special issue), *SIGPLAN Notices 26* (4).

[Chaitin82] G. J. Chaitin. Register allocation & spilling via graph coloring. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction,* pp. 98-105. Published as *SIGPLAN Notices 17* (6), June 1982.

[Chaitin+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages 6,* pp. 47-57, 1981.

[Chow+86] F. Chow, M. Himelstein, E. Killian, and L. Weber. Engineering a RISC compiler system. *Digest of Papers: Compcon 86,* pp. 132-137, March 1986.

[Digital92] Digital Equipment Corporation. VEST User's Guide.

[McFarling89] Scott McFarling. Program optimization for instruction caches. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems,* pp. 183-191, April 1989. Published as *Computer Architecture News 17* (2), *Operating Systems Review 23* (special issue), *SIGPLAN Notices 24* (special issue).

[GibbonsMuchnick86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* pp. 11-16. Published as *SIGPLAN Notices 21* (7), July 1986.

[GoldbergHennessy91] Aaron Goldberg and John Hennessy. Performance debugging shared memory multiprocessor programs. *Proceedings of Supercomputing '91,* pp. 481-490, November 1991.

[GoldbergHennessy92] Aaron Goldberg and John Hennessy. MTOOL: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE TPDS,* to appear.

[GoodmanHsu88] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. *International Conference on Supercomputing,* pp. 442-452, July 1988.

[Graham+82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software — Practice and Experience 13* (8), pp. 120-126, August 1983.

[HastingsJoyce92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. *Proceedings of the Winter 1992 USENIX Conference,* pp. 125-136, January 1992.

[HennessyGross83] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems 5* (3), pp. 422-448, July 1983.

[Himelstein+87] Mark I. Himelstein, Fred C. Chow, and Kevin Enderby. Cross-module optimizations: Its implementation and benefits. *Proceedings of the Summer 1987 USENIX Conference,* pp. 347-356, June 1987.

[Johnson90] S. C. Johnson. Postloading for fun and profit. *Proceedings of the Winter 1990 USENIX Conference,* pp. 325-330, January 1990.

[Kane87] Gerry Kane. *MIPS R2000 Risc Architecture.* Prentice Hall, 1987.

[KeppelEtal91] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

[LarusBall92] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. University of Wisconsin Technical Report 1083, March 1992.

[MIPS90] MIPS Computer Systems. *MIPS Assembly Language Programmer's Guide. MIPS Computer Systems, Inc., 930 Arques Ave., Sunnyvale, California 94086. 1990.*

[MIPS89] MIPS Computer Systems. *RISCompiler and C Programmer's Guide.* MIPS Computer Systems, Inc., 930 Arques Ave., Sunnyvale, California 94086. 1989.

[Sarkar89] Vivek Sarkar. Determining average program execution times and their variance. *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* pp. 298-312. Published as *SIGPLAN Notices 24* (7), July 1989.

[Wall91a] David W. Wall. Experience with a software-defined machine architecture. *ACM Transactions on Programming Languages and Systems,* to appear. Also available as WRL Research Report 91/10, August 1991.

[Wall91b] David W. Wall. Systems for late code modification. *Proceedings of the CODE 91 Workshop on Code Generation,* Springer Workshops in Computer Science, to appear. Also available as WRL Research Report 92/3, May 1992.

[WallPowell87] David W. Wall and Michael L. Powell. The Mahler experience: Using an intermediate language as the machine description. *Second International Symposium on Architectural Support for Programming Languages and Operating Systems,* pp. 100-104. Published as *Computer Architecture News 15* (5), *Operating Systems Review 21* (4), *SIGPLAN Notices 22* (10), October 1987. A more detailed version is available as WRL Research Report 87/1.