CS184b: Computer Architecture (Abstractions and Optimizations)

Day 8: April 23, 2003

Binary Translation

Caching Introduction

Today

- Binary Translation
  - Competitive/online translation
  - Some numbers
- Memory System
  - Issue
  - Structure
  - Idea
  - Cache Basics

Previously: BT Idea

- Treat ABI as a source language
  - the specification
- Cross compile (translate) old ISA to new architecture (ISA?)
- Do it below the model level
  - user doesn't need to be cognizant of translation
- Run on simpler/cheaper/faster/newer hardware

Finding the Code

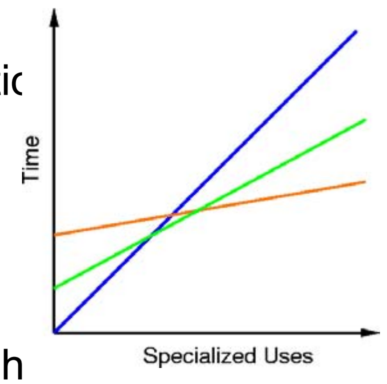
- **Problem:** can't always identify statically
- **Solution:** wait until "execution" finds it
  - delayed binding
  - when branch to a segment of code,
    - certainly know where it is
    - and need to run it
  - translate code when branch to it
    - first time
    - nth-time?

Common Prospect

- Translating code is large fixed cost
  - but has low incremental cost on each use
  - hopefully comparable to or less than running original on old machine
- Interpreting/Emulating code may be faster than "compiling" it
  - if the code is run once
- Which should we do?

Optimization Prospects

- Translation vs. Emulatic
  - $T_{trun} = T_{trans} + nT_{op}$
  - $T_{trns} > T_{em\_op} > T_{op}$
- If compute long enough
  - $nT_{op} >> T_{trans}$
  - $\rightarrow$ amortize out load



"Competitive" Approach

- Run program emulated
- When a block is run "enough", translate
- Consider
  - $N_{thresh}\ T_{emop} = T_{translate}$
- Always w/in factor of two of optimal
  - if $N<N_{thresh}$ optimal
  - if $N=N_{thresh}$ paid extra $T_{translate}$ =2×optimal
  - as $N>>N_{thresh}$ extra time amortized out with translation overhead
    - think $T_{translate}$ ~=2$T_{translate}$

On-the-fly Translation Flow

- Emulate operations
- Watch frequency of use on basic blocks
- When run enough,
  - translate code
  - save translation
- In future, run translated code for basic block

Translation “Cache”

- When branch
  - translate branch target to new address
  - if hit, there is a translation,
    - run translation
  - if miss, no translation
    - run in emulation (update run statistics)

Alternately/Additionally

- Rewrite branch targets so address translated code sequence
  - when emulator finds branch from translated sequence to translated sequence
  - update the target address of the branching instruction to point to the translated code

Self-Modifying Code

- Mark pages holding a translated branch as read only
- Take write fault when code tries to write to translated code
- In fault-handler, flush old page translation

Precise Exceptions

- Again, want exception visibility relative to simple, sequential model
  - …and now old instruction set model
- Imposing ordering/state preservation is expensive

Precise Exceptions

- Modern BT technique [hardware support]
  - "backup register" file
  - commit/rollback of register file
  - commit on memories
  - on rollback, recompute preserving precise state
    - drop back to emulation?
- …active work on software-only solutions
  - e.g. IBM/WBT'00

Remarkable Convergence?

- Aries: HP PA-RISC→IA-64
  - new architecture
- IBM: PowerPC→BOA
  - ultra-high clock rate architecture? (2GHz)
    - IBM claims 50% improvement over scaling?
    - 700ps = 1.4GHz in 0.18μm
- Transmeta: x86 →Crusoe
  - efficient architecture, avoid x86 baggage

Remarkable Convergence?

- All doing dynamic translation
  - frequency based
- To EPIC/VLIW architectures

Academic Static Binary Translation

	Translated Code		Native Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) sec	27.7	28.5	28.6	25.9
bytes	16,512	7,292	16,144	16,152
Sieve(3000) sec	17.8	17.4	18.9	18.6
bytes	16,244	6,548	15,964	15,944
Mbanner(500K) sec	42.5	n/a	80.5	44.8
bytes	22,240		21,524	25,436

Static SPARC to Pentium Translation

[Cifuentes et. al., Binary Translation Workshop 1999]

Academic/Static BT

	Translated Code		Native Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) sec	23.0	24.3	41.0	23.0
bytes	24,916	6,680	24,628	24,564
Sieve(3000) sec	26.9	23.9	29.3	24.5
bytes	24,776	6,312	24,552	24,452
Mbanner(500K) sec	53.3	36.9	63.7	26.6
bytes	34,188	21,448	30,652	30,268

Static Pentium to SPARC Translation

[Cifuentes et. al., Binary Translation Workshop 1999]

Academic/Dynamic BT

			Without Hot Paths		With Hot Paths			
Test programs	Startup time	Translation time	Execution time without reg caching	Execution time with reg caching	Optimisation time	Execution time without reg caching	Execution time with reg caching	Native gcc compiled
Sieve3000	0.54	0.14	98.25	73.14	0.16	80.98	66.29	29.22
Fibonacci	0.54	0.10	186.17	154.97	0.11	147.56	133.69	41.18
mbanner	0.52	0.34	219.01	146.28	0.37	146.22	126.28	22.85

Table 1: Pentium to SPARC translation (second)

[Ung+Cifuentes, Binary Translation Workshop 2000]

Caching

Memory and Processors

- Memory used to compactly store
  - state of computation
  - description of computation (instructions)
- Memory access latency impacts performance
  - timing on load, store
  - timing on instruction fetch

Issues

- Need big memories:
  - hold large programs (many instructions)
  - hold large amounts of state
- Big memories are slow
- Memory takes up areas
  - want dense memories
  - densest memories not fast
    - fast memories not dense
- Memory capacity needed not fit on die
  - inter-die communication is slow

Problem

- Desire to contain problem
  - implies large memory
- Large memory
  - implies slow memory access
- Programs need frequent memory access
  - e.g. 20% load operations
  - fetch required for every instruction
- Memory is the performance bottleneck?
  - Programs run slow?

Opportunity

- Architecture mantra:
  - exploit structure in typical problems
- What structure exists?

Memory Locality

- What percentage of accesses to unique addresses
  - addresses distinct from the last N unique addresses

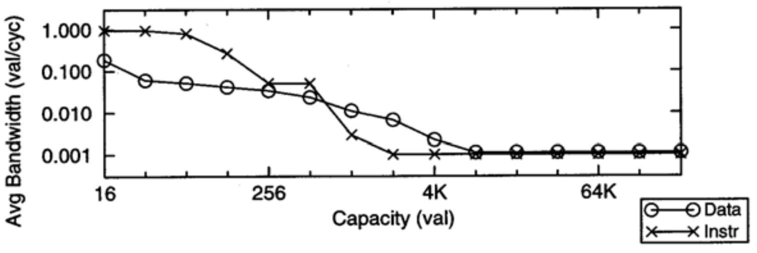


Figure 3-1 Bandwidth spectrums for *cjpeg*.

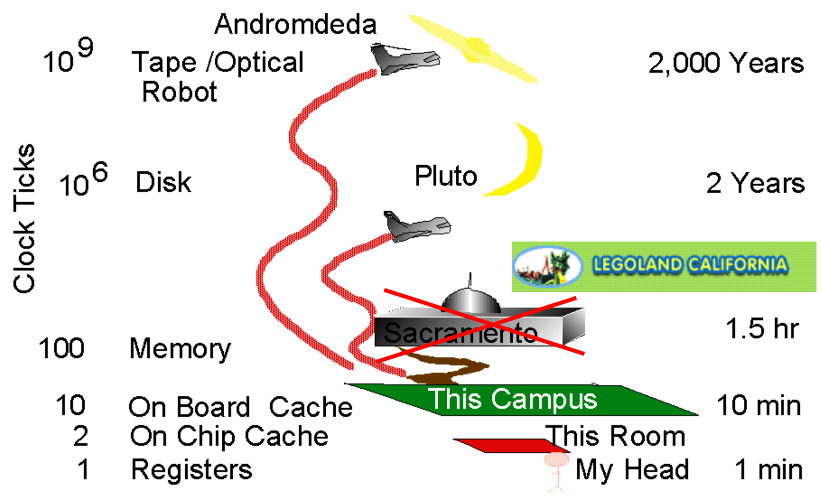
[Huang+Shen, Intrinsic BW, ASPLOS 7]

[from CS184a]

Hierarchy/Structure Summary

- "Memory Hierarchy" arises from area/bandwidth tradeoffs
  - Smaller/cheaper to store words/blocks
    - (saves routing and control)
  - Smaller/cheaper to handle long retiming in larger arrays (reduce interconnect)
  - High bandwidth out of registers/shallow memories

	DRAM	SRAM	RF bit	FF/RF	RF×1	XC	In FF	net FF	FF/LUT
λ^2	100	1200	2K	5K	40K	40K	75K	200K	800K
bw/cap.	$1/10^7$	$1/10^5$–10^3		1/100	1/100	1/16	1/4	1/1	1/1



From: AlphaSort: A Cache-Sensitive Parallel External Sort
ACM SIGMOD'94 Proceedings/VLDB Journal 4(4): 603-627 (1995).

Opportunity

- Small memories are fast
- Access to memory is not random
  - temporal locality
  - short and long retiming distances
- Put commonly/frequently used data (instructions) in small memory

Memory System Idea

- Don’t build single, flat memory
- Build a hierarchy of speeds/sizes/densities
  - commonly accessed data in fast/small memory
  - infrequently used data in large/dense/cheap memory
- Goal
  - achieve speed of small memory
  - with density of large memory

Hierarchy Management

- Two approaches:
  - explicit data movement
    - register file
    - overlays
  - transparent/automatic movement
    - invisible to model

Opportunity: Model

- Model is simple:
  - read data and operate upon
  - timing not visible
- Can vary timing
  - common case fast (in small memory)
  - all cases correct
    - can answered from larger/slower memory

Cache Basics

- Small memory (cache) holds commonly used data
- Read goes to cache first
- If cache holds data
  - return value
- Else
  - get value from bulk (slow) memory
- Stall execution to hide latency
  - full pipeline, scoreboarding

Cache Questions

- How manage contents?
  - decide what goes (is kept) in cache?
- How know what we have in cache?
- How make sure consistent ?
  - between cache and bulk memory

Cache contents

- **Ideal:** cache should hold the N items that maximize the fraction of memory references which are satisfied in the cache
- **Problem:**
  - don't know future
  - don't know what values will be needed in the future
    - partially limitation of model
    - partially data dependent
    - halting problem
      - (can't say if will execute piece of code)

Cache Contents

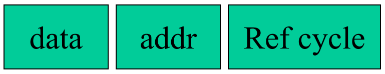
- Look for heuristics which keep most likely set of data in cache
- **Structure:** temporal locality
  - high probability that recent data will be accessed again
- **Heuristic goal:**
  - keep the last N references in cache

Temporal Locality Heuristic

- Move data into cache on access (load, store)
- Remove "old" data from cache to make space

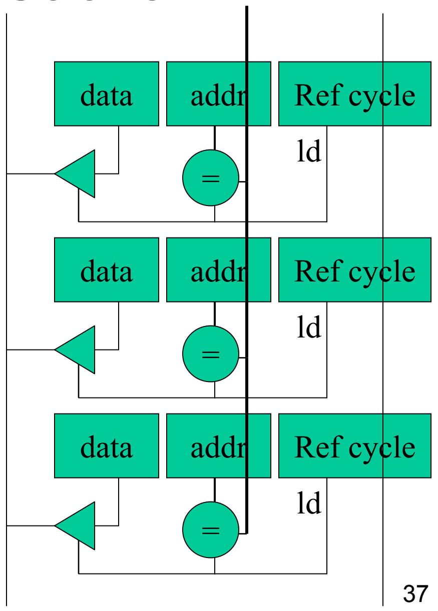
"Ideal" Locality Cache

- Stores N most recent things
  - store any N things
  - know which N things accessed
  - know when last used



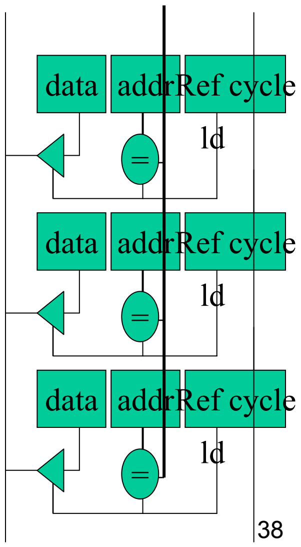
"Ideal" Locality Cache

- Match address
- If matched,
    - update cycle
- Else
    - drop oldest
    - read from memory
    - store in newly free slot



Problems with "Ideal" Locality?

- Need O(N) comparisons
- Must find oldest
    - (also O(N)?)
- Expensive

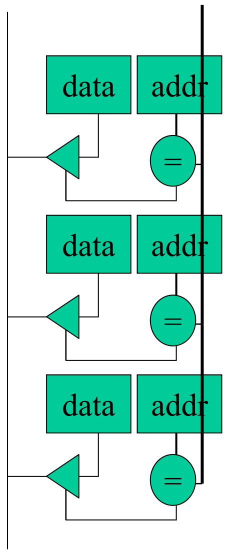


Relaxing “Ideal”

- Keeping usage (and comparing) expensive
- Relax:
  - Keep only a few bits on age
  - Don’t bother
    - pick victim randomly
    - things have expected lifetime in cache
    - old things more likely than new things
    - if evict wrong thing, will replace
    - very simple/cheap to implement

Fully Associative Memory

- Store both
  - address
  - data
- Can store any N addresses
- approaches ideal of “best” N things

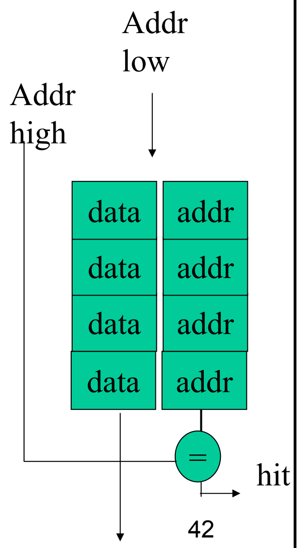


Relaxing "Ideal"

- Comparison for every address is expensive
- Reduce comparisons
  - deterministically map address to a small portion of memory
  - Only compare addresses against that portion

Direct Mapped

- Extreme is a "direct mapped" cache
- Memory slot is f(addr)
  - usually a few low bits of address
- Go directly to address
  - check if data want is there



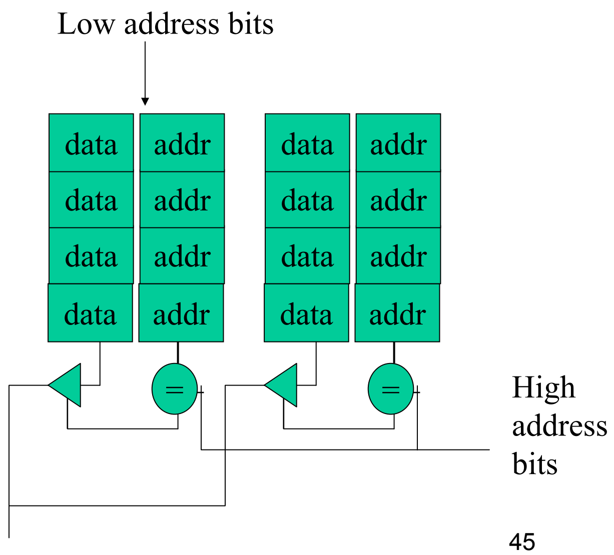
Direct Mapped Cache

- Benefit
  - simple
  - fast
- Cost
  - multiple addresses will need same slot
  - conflicts mean don't really have most recent N things
  - can have conflict between commonly used items

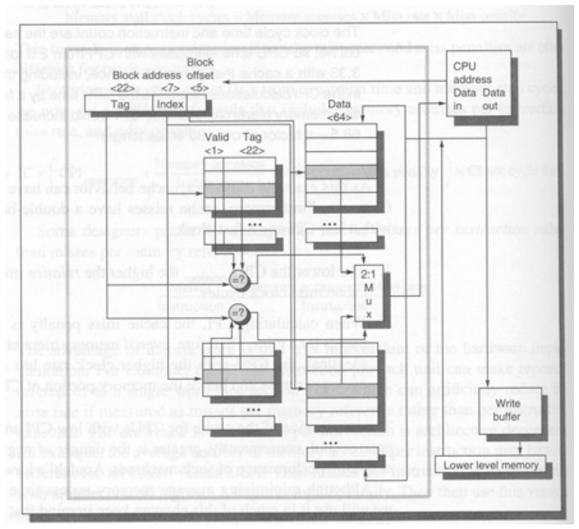
Set-Associative Cache

- Between extremes set-associative
- Think of M direct mapped caches
- One comparison for each cache
- Lookup in all M caches
- Compare and see if any have target data
- Can have M things which map to same address

Two-Way Set Associative



Two-way Set Associative



[Hennessy and Patterson 5.8e2]

Set Associative

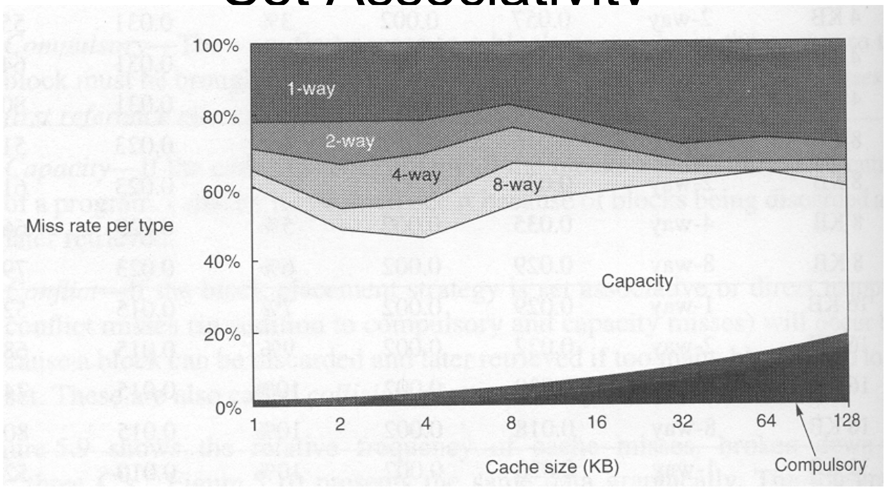
- More expensive that direct mapped
- Can decide expense
- Slower than direct mapped
  - have to mux in correct answer

- Can better approximate holding N most recently/frequently used things

Classify Misses

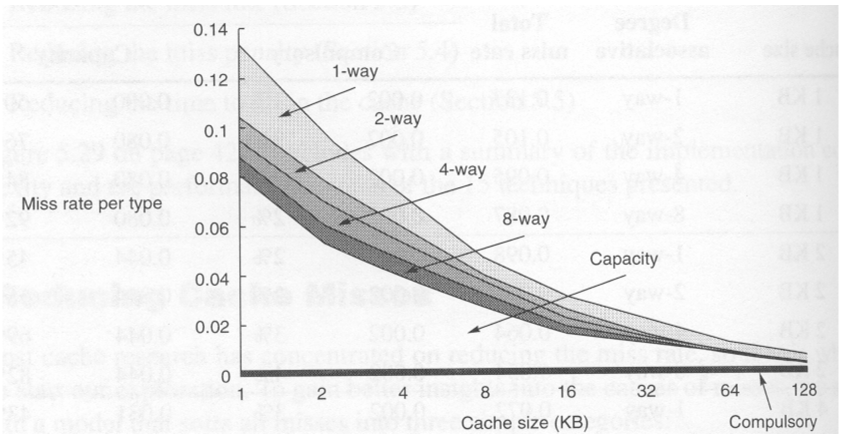
- Compulsory
  - first refernce
  - (any cache would have)
- Capacity
  - misses due to size
  - (fully associative would have)
- Conflict
  - miss because of limit places to put

Set Associativity



[Hennessy and Patterson 5.10e2]

Absolute Miss Rates



[Hennessy and Patterson 5.10e2]

Policy on Writes

- Keep memory consistent at all times?
  - Or cache+memory holds values?

- Write through:
  - all writes go to memory and cache
- Write back:
  - writes go to cache
  - update memory only on eviction

Write Policy

- Write through
  - easy to implement
  - eviction trivial
    - (just overwrite)
  - every write is slow (main memory time)
- Write back
  - fast (writes to cache)
  - eviction slow/complicate

Cache Equation...

- Assume hits satisfied in 1 cycle
- CPI = Base CPI + Refs/Instr (Miss Rate)(Miss Latency)

Cache Numbers

- CPI = Base CPI + Ref/Instr (Miss Rate)(Miss Latency)
- From ch2/experience
  - load-stores make up ~30% of operations
- Miss rates
  - …1-10%
- Main memory latencies
  - 50ns
- Cycle times
  - 300-500ps … shrinking

Cache Numbers

500ps
cycle

- No Cache
  - CPI=Base+0.3*100=Base+30
- Cache at CPU Cycle (10% miss)
  - CPI=Base+0.3*0.1*100=Base +3
- Cache at CPU Cycle (1% miss)
  - CPI=Base+0.3*0.01*100=Base +0.3

Wrapup

Big Ideas [Binary Trans]

- Well-defined model
  - High value for longevity
  - Preserve semantics of model
  - How implemented irrelevant
- Hoist work to earliest possible binding time
  - dependencies, parallelism, renaming
  - hoist ahead of execution
    - ahead of heavy use
  - reuse work across many uses
- Use feedback to discover common case

Big Ideas

- Structure
  - temporal locality
- Model
  - optimization preserving model
  - simple model
  - sophisticated implementation
  - details hidden

Big Ideas

- Balance competing factors
  - speed of cache vs. miss rate
- Getting best of both worlds
  - multi level
  - speed of small
  - capacity/density of large