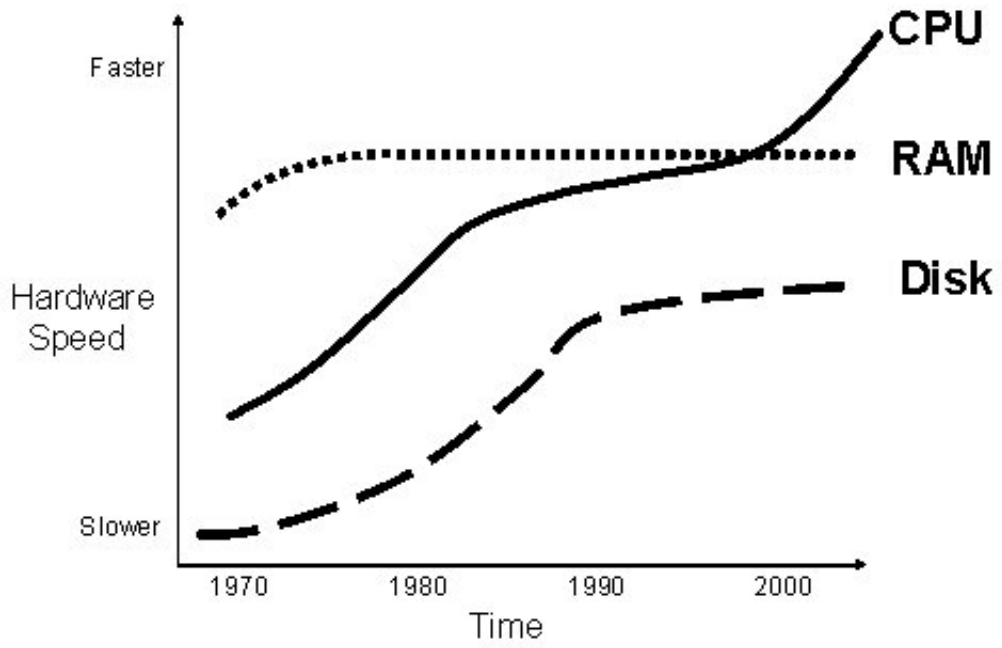
Reuse Signature Analysis and Optimization

Tom Cauchois <tcauchois@gmail.com>
15-745 S07

Motivations



Storage latencies can't keep up with processor utilization; SRAM too expensive

Solution: memory hierarchy and caching

Feeding the Cache

Taking advantage of spatial locality:

- *Loop reordering, loop body alignment, struct and spill allocation on the stack*

Taking advantage of temporal locality:

- *More loop reordering, code motion, register allocation*

But we need to know what memory access patterns will be...

Determining Locality

Easy at the basic block level; use compile-time information about variable accesses and array accesses in loops

What about indirection/inter-procedural locality?

- *Not enough control flow data to do it statically*
- *Profiling results only give you access patterns for the tested input*

Global Locality Analysis

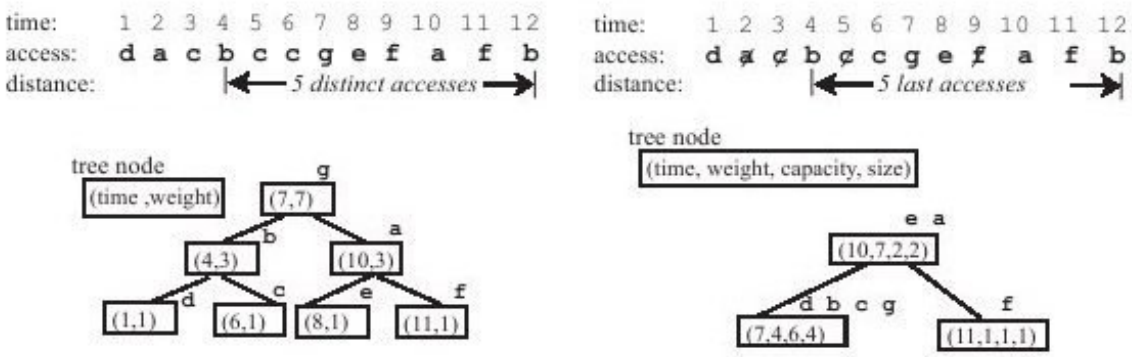
New concept: data reuse distance

- *The number of intervening memory accesses between two accesses of the same variable*
- *We can develop a model of reuse distance distribution of a given variable/instruction*
- *This lets us predict L2 miss rate*
- *Which opens the door to a variety of optimizations*

Global Locality Analysis

To compute reuse distance, we need a compact way to store the last time a memory transaction happened.

Lots of literature; Ding proposes an approximate tree-based datastructure that coalesces nodes.



Global Locality Analysis

Instrument the code; for each memory transaction, record the reuse distance in a histogram and update the last-use time

Gives us a statistical distribution of reuse distance over a particular program input, but how do we generalize?

Global Locality Analysis

Transpose the histogram; find average reuse distance of the top k% of variables, and define:

- $d_{n,i}$ = *avg reuse distance of ith percentile on iteration n*
- s_n = *program data size on iteration n*

Need to do a regression on

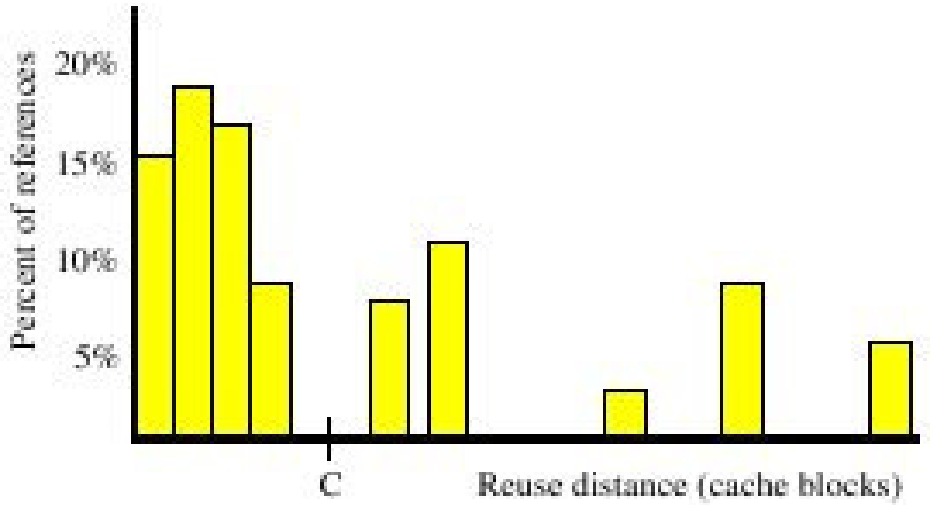
$$d_{n,i} = c_i + e_i * f_i(s_n)$$

Since reuse distance is bounded by program size, f_i is linear or sub-linear; one of x^1, $x^{0.5}$, or x^0 will usually work.

Global Locality Analysis

A fully associative cache of size C kicks a line out after C other lines have been accessed.

For a given instruction, if we can predict the percentage of reuses with distance less than C, that gives us the L2 cache hit rate.



Applications of this data

Reuse analysis gives the instructions with the highest probability of generating an L2 miss

Since most programs are memory bound, and the CPU only talks to memory on an L2 miss, these instructions are critical performance bottlenecks

Miss-rate prediction is still relatively accurate for set-associative caches; can also model L1 hit/miss

Applications of this data

If we know an instruction will cache miss, there are a variety of ways to deal with it:

- *hardware and software prefetching; for example, IA-64's speculative execution mechanism*
- *feedback-directed memory reorganization*
- *smarter scheduling*

Questions?