Implementing a Stack Cache

Alex Hemsath
Robert Morton
Jan Sjodin

4/23/2002

Stack Cache: a Quick Reminder

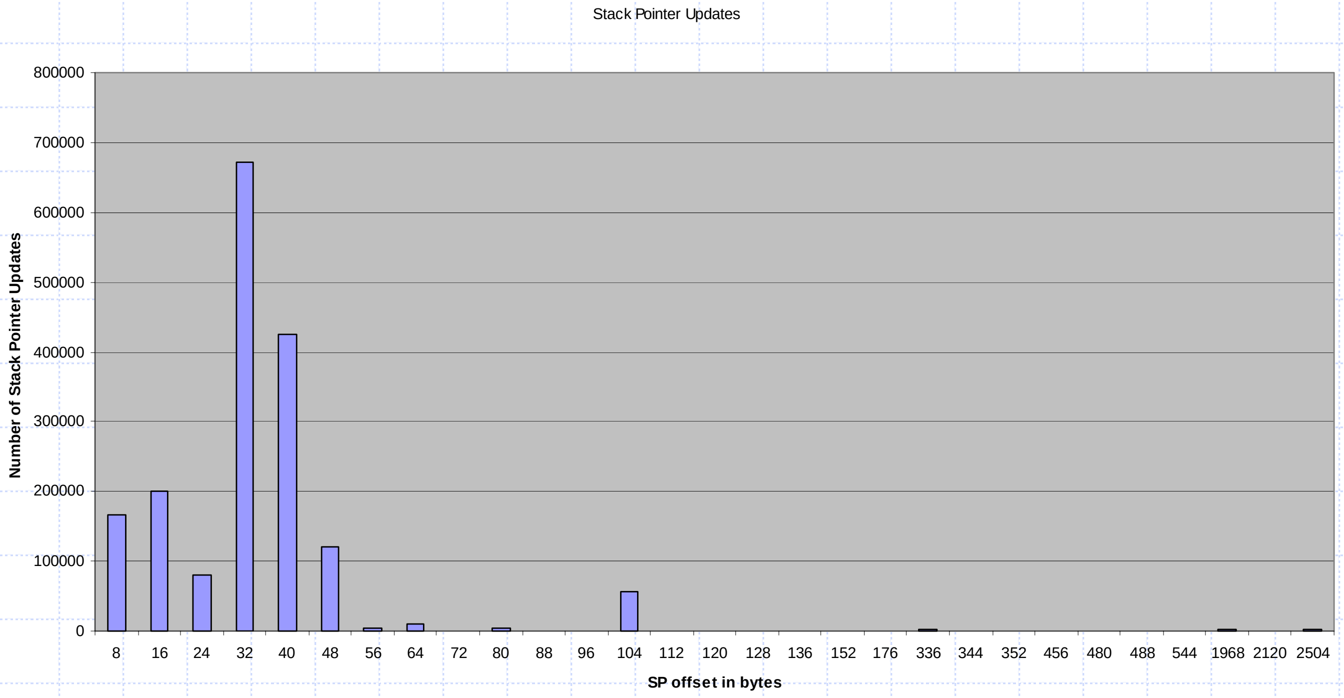
- A stack cache is intended to maintain a separate cache from the L1 for data that is accessed in a predictable, stack-like fashion.
- A stack cache handles all memory references with respect to the Stack Pointer (SP).

Our Hypotheses

- The stack cache will have a faster access than modern L1 caches.
- The stack cache will prevent short-lived stack data from evicting heap data from the L1.
- Prefilling and prespilling of the stack cache could effectively hide much of the L2 latency for stack cache misses.

Pre-implementation Analysis

- We used sim-cache to estimate the best size of a stack cache (equake):



Our Implementation

- We gathered statistics using sim-cache.
- We built the SC into sim-outorder.
- ~500 lines of code, plus datalogging.

- For ease of implementation, we forced all stack accesses to go through the SC instead of the L1 data cache.

More Implementation Details

- We monitor updates to the SP every cycle.
- Every cycle we examine the utility of the SC – as well as available L2 access bandwidth – to determine when to prefill/spill.
- Memory accesses that did not hit in the SC would be diverted to the L2.

Testing Procedure

- We came up with a base case and several variables to test in combination.
- Base case:
  - 16Kb L1 D$, 16Kb L1 I$, 3 cycle, 4-way
  - 512Kb shared L2, 12 cycle, 8-way
  - Stack cache disabled

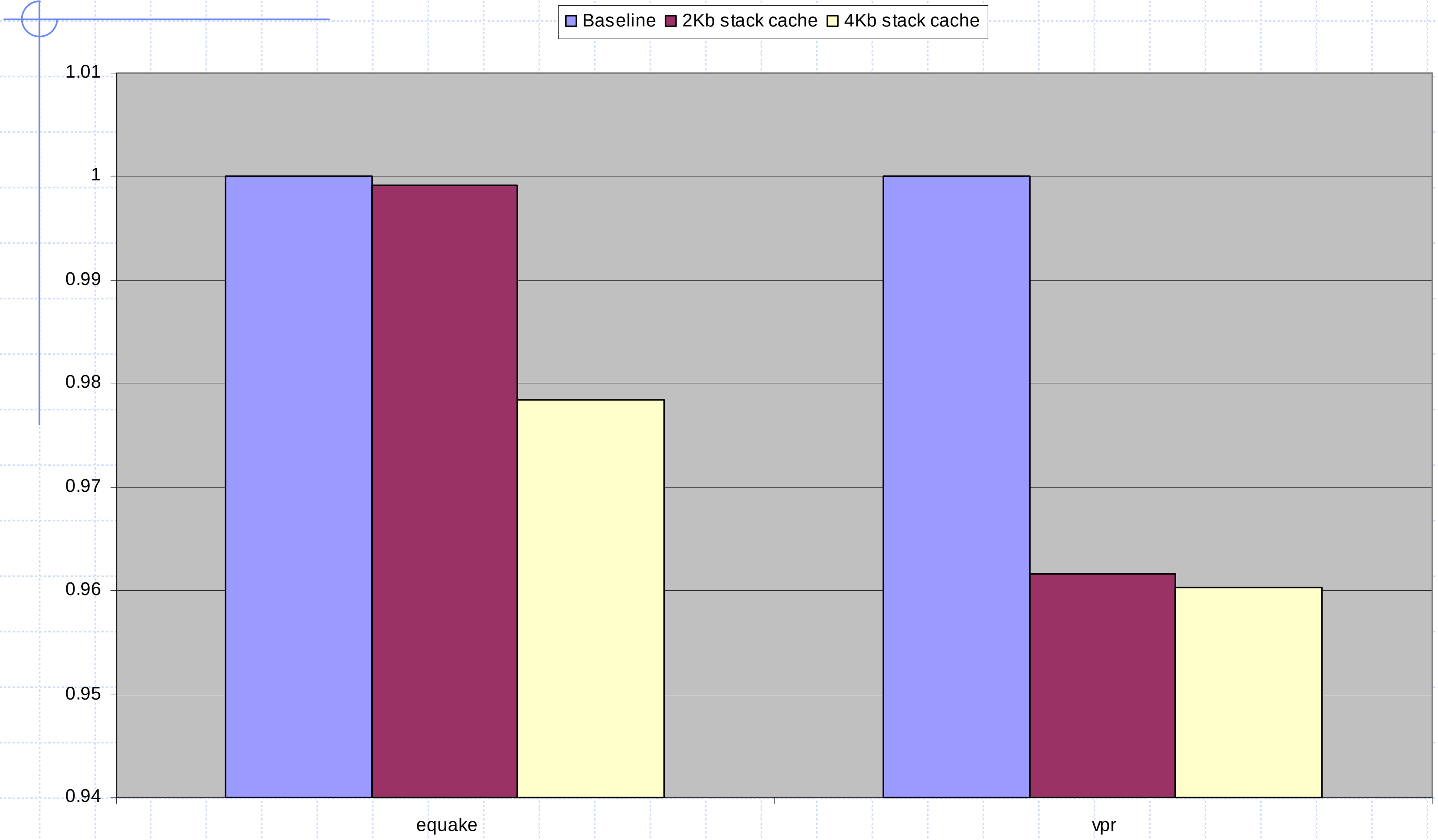
Testing Procedure (cont'd)

- Stack cache size:
  - 2Kb, 4Kb
- L1 D$ associativity:
  - 4-way, 2-way
- L1 D$ latency:
  - 3 cycles, 1 cycle
- L1 D$ size
  - 16Kb, 4Kb, 2Kb (must be powers of two)

Testing Procedure (cont’d)

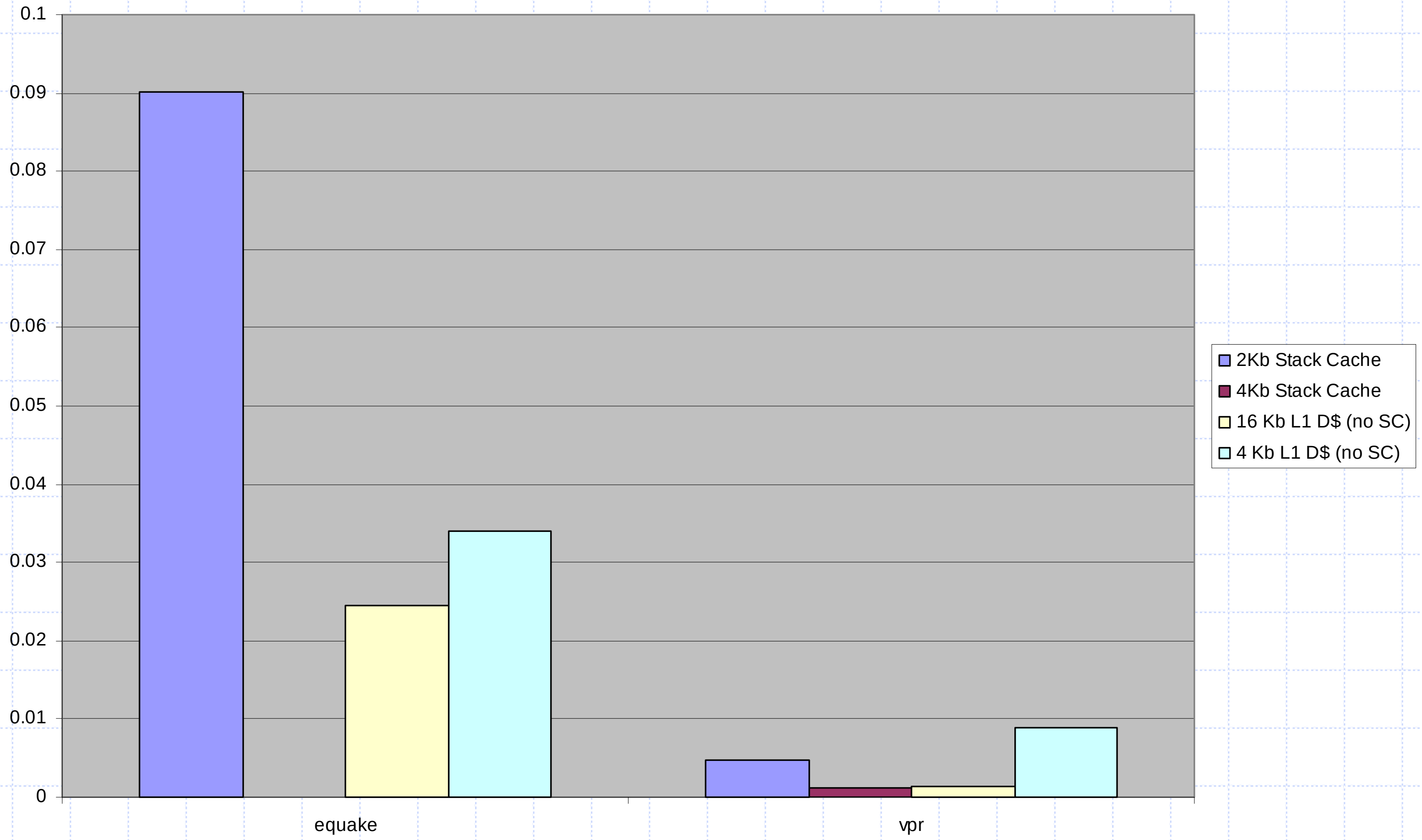
- We used the following benchmarks:
  - equake (SPECFP)
  - ammp (SPECFP)
  - parser (SPECINT)
  - vpr (SPECINT)
- We currently only have data available for *equake* and *vpr*.

Results – Stack Cache size



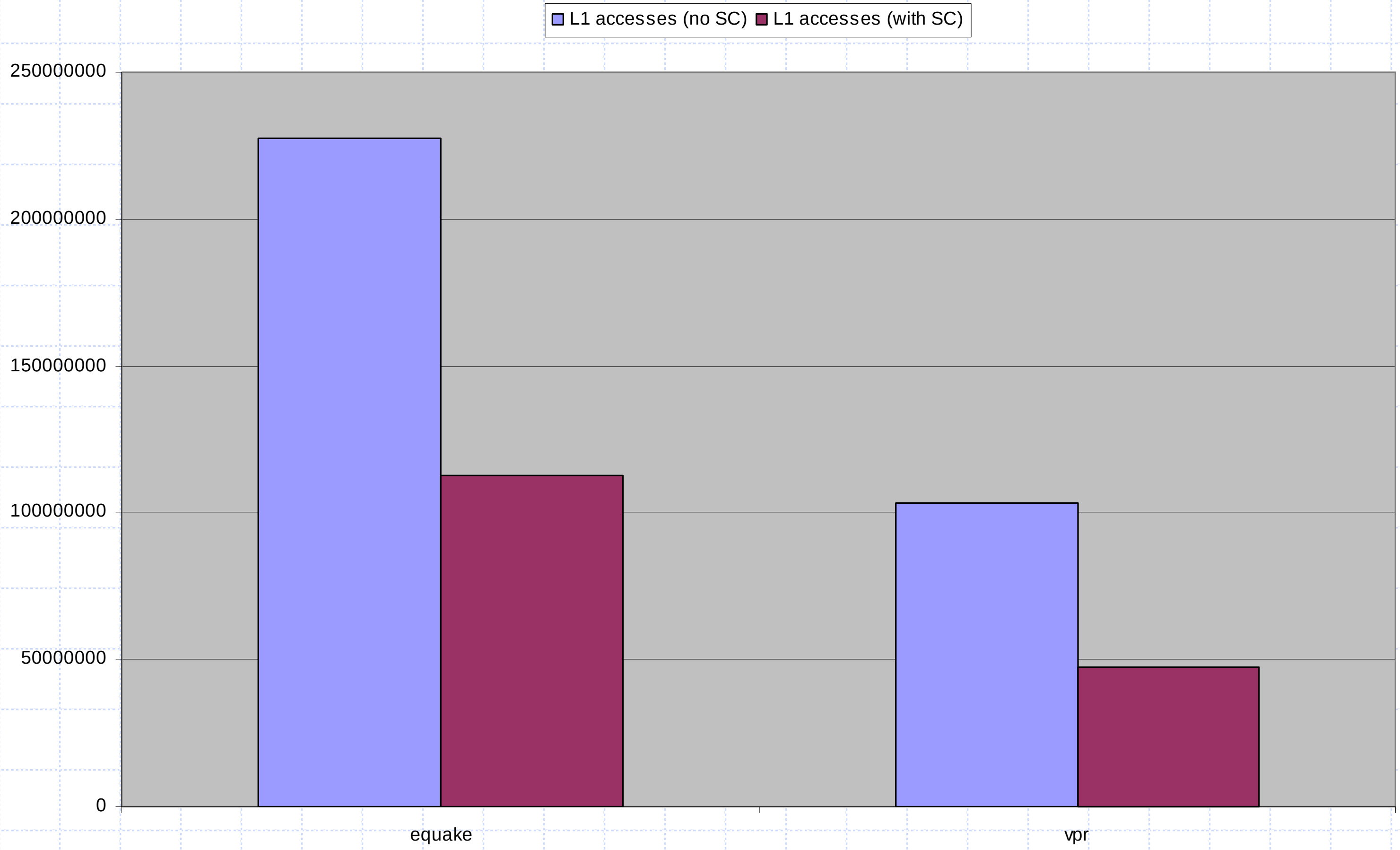
Normalized cycle count (baseline = 1.0, lower is better)

Results – Cache miss rates



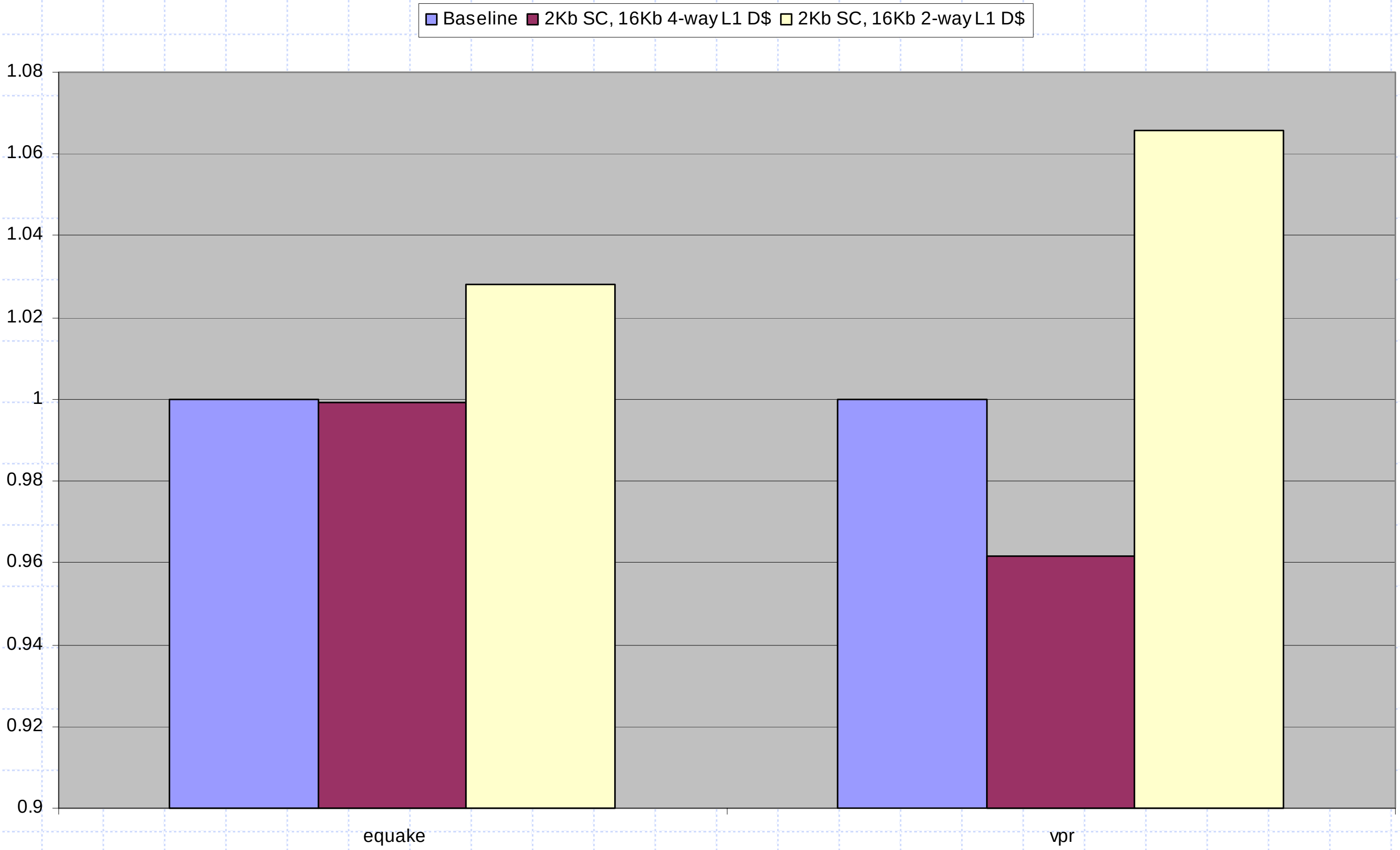
Miss rates for 2Kb SC, 4Kb SC, 16Kb L1 D$, 4Kb L1 D$

Results – L1 D$ access count



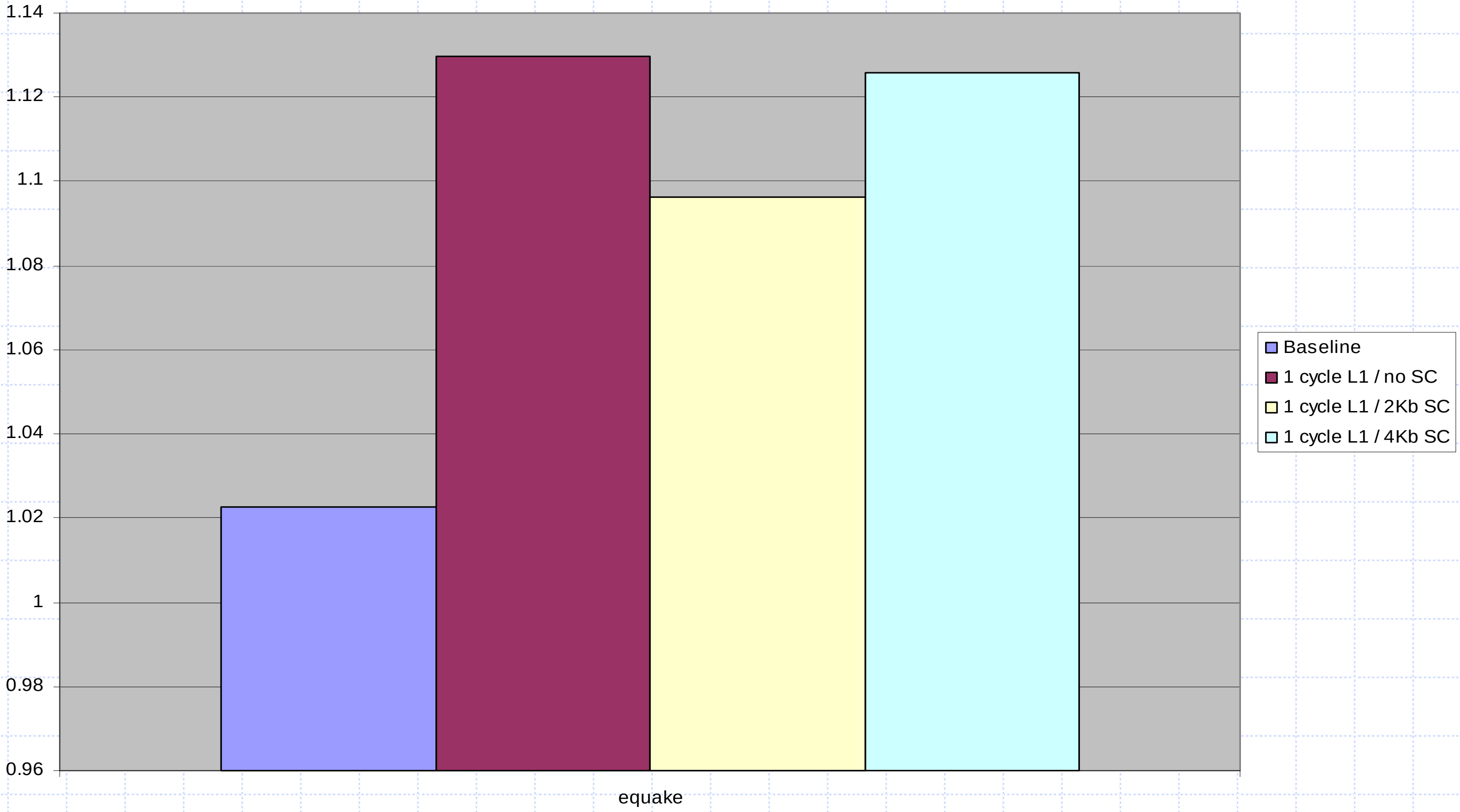
Access counts for L1 D$ without SC / with SC

Results – L1 D$ associativity



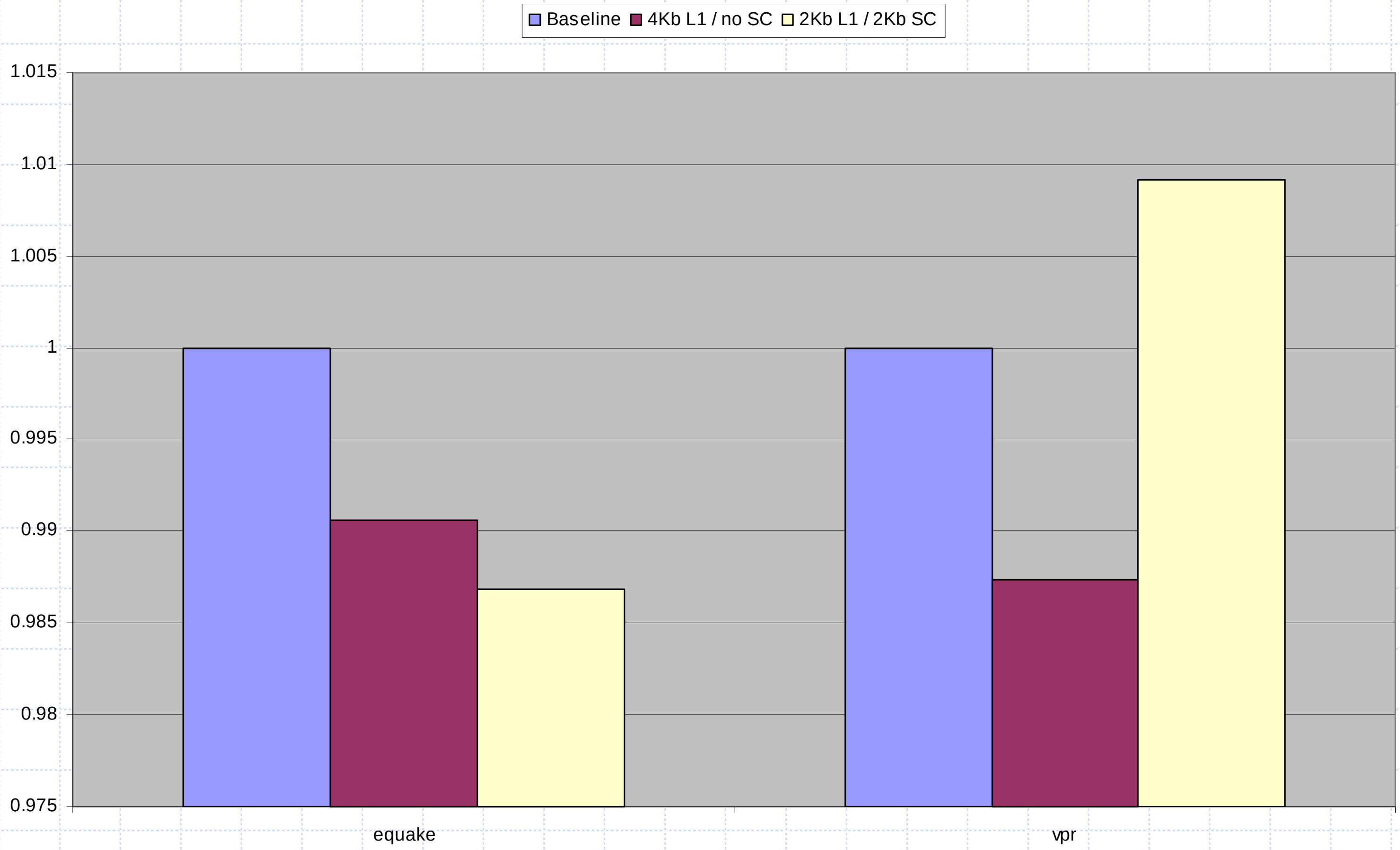
Baseline / 2Kb SC, 16Kb 4-way L1 / 2Kb SC, 2-way L1
(Normalized cycle count, lower is better)

Results – L1 D$ latency



IPC – higher is better

Results – Dividing hardware resources



Baseline / 4Kb L1 D$ / 2Kb L1 D$, 2Kb SC (normalized IPC, higher is better)

Successes

- We have shown that the stack cache is very space efficient.
  - It has a high hit rate and a low latency
  - It intercepts ~1/2 of the accesses to the L1 data cache
  - Performance improvements ranged from 2-4% with the addition of a 4Kb stack cache.
- This verifies our first two hypotheses.

Shortcomings

- One pitfall was having implemented an unaggressive prefill/spill unit
  - This meant our cache had to be twice as large to operate as expected.
  - This seems to verify our last hypothesis – a more aggressive prefill/spill unit can aid the stack cache tremendously.
- An additional problem was our limited data-gathering capabilities
  - The modified simulator ran extremely slowly.
  - The simulator had quirky parameter rules.

Conclusion

- The stack cache appears to be a viable option, provided:
  - It is large enough (the performance discrepancy between sizes is large)
  - It has an aggressive prefill/spill mechanism
  - It can truly be implemented with 1 cycle latency
- Modern architectures with heavy reliance on the stack, like the x86, could benefit enormously.