Computer Science and Engineering

- Executive summary
  - “In nearly every part of modern life, the hardware and software of computer technology enable the delivery of services and products of higher quality to more people in less time than would otherwise be possible.”
  - “… the organization of (electronic) components into useful computer hardware and the ability to write the software required to exploit this hardware are primarily the fruits of CS&E.
  - “Further advances in computer power and usability will also depend in large part on pushing back the frontiers of CS&E and will be motivated by a myriad of applications that can take advantage of these advances.”

Locality and Caching Theory

Why Cache?

- Memory hierarchy
  - nothing travels faster than light
  - the faster the access, the smaller the data capacity
  - small fast memory
  - large slow memory/storage

- Cache for programmability
  - automatic management
  - transparent to the user
  - modular, efficient, portable
  - examples
    - CPU cache, virtual memory, DRAM/disk/IO buffers
    - memcached
    - used in Facebook/Twitter, supported in Amazon/Google cloud

- Fast, large and cheap memory, if we can get it to work

Cache is Everywhere

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Floating Point Capability</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops / clock</td>
</tr>
<tr>
<td>Single Precision Floating Point Capability</td>
<td>128 MAD ops / clock</td>
<td>240 MAD ops / clock</td>
<td>512 MAD ops / clock</td>
</tr>
<tr>
<td>Special Function Units (SFUs) / SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>Configurable 48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>Configurable 16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrency Kernels</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
Autonomous Cache Management

- Which data to store in cache?
  - the data needed most
- how to infer usage?
  - LRU replacement
- replace the least recently accessed datum
- LRU stack distance at every access
  - a miss iff the distance is greater than cache size

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Hardware/System View

Software View

The Cold Fact of Hot Data

Cache Performance for SPEC CPU2000 Benchmarks

Version 3.0

May 2003

Jason F. Cantin
Department of Electrical and Computer Engineering
1415 Engineering Drive
University of Wisconsin-Madison
Madison, WI 53706-1691
jcantin@ece.wisc.edu
http://www.jfred.org

Mark D. Hill
Department of Computer Science
1210 West Dayton Street
University of Wisconsin-Madison
Madison, WI 53706-1685
markhill@cs.wisc.edu
http://www.cs.wisc.edu/~markhill


Abstract

The SPEC CPU2000 benchmark suite (http://www.spec.org/osg/cpu2000) is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer’s CPU, memory system, and compilers. The benchmarks in this suite were chosen to represent real-world applications, and thus exhibit a wide range of runtime behaviors. On this webpage, we present functional cache miss ratios and related statistics for the SPEC CPU2000 suite. In particular, L1 instruction, L1 data, and L1 unified caches ranging from 1KB to 1MB with 64B blocks and associativities of 1, 2, 4, 8 and full. Prefetch operations were always executed, but results are posted both with and without them counted in the hit ratios. Most of this data was collected at the University of Wisconsin-Madison with the aid of the Simplescalar toolset (http://www.simplescalar.org).
**Locality**

• Started as an empirical observation
  • "During any interval of execution, a program favors a subset of its pages, and this set of favored pages changes slowly" – Peter Denning
  • The 80-20 rule
• Data in vs not in cache (hits vs misses)
  • Matthew Hertz’s beer analogy
  • Trishul Chilimbi’s walking-the-dog analogy
  • My platform analogy
• Two examples
  
<table>
<thead>
<tr>
<th>abcdaefghija...</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbbacccadda...</td>
</tr>
</tbody>
</table>
  • the amount of intervening data vs time

**Mark Wegman Problem**

• Mark Wegman
  • (co-)invented SSA with Ron Cytron, Jeanne Ferrante, Barry K. Rosen and F. Kenneth Zadeck
  • SSA used in Open64, GCC, LLVM, IBM/Oracle JVMs, Mono, Mozilla/Chromium JavaScript engine, PyPy, Android’s Dalvik, Standard ML compiler MLton, LuajIT, PHP compiler, Single-assignment C (SaC)
  • taught in CSC 255/455 Program Analysis and Improvement
  • First ACM Workshop on Memory System Performance and Correctness (MSPC) in 2002
  • "The first load takes a long time, the next 10 do not matter!"
  • Performance depends much more on miss count than instruction count

**Program Locality**

**Reuse Distance**

(i.e. LRU stack distance [Mattson et al. IBM 1970])

**Measuring Reuse Distance**

| time: 1 2 3 4 5 6 7 8 9 10 11 12 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| access: d a c b c c g e f a f b |
| distance: 5 distinct accesses |

(a) An example access sequence the reuse distance between two b’s is 5

• Naïve counting, O(N) time per access, O(N) space
  • N is the number of memory accesses
  • M is the number of distinct data elements
  • Too costly
  • N is up to 120 billion, M 25 million

**Precise Methods**

| time: 1 2 3 4 5 6 7 8 9 10 11 12 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| access: d a b e b c c g e f a f b |
| distance: 5 last accesses |

(b) Store and count only the last access of each data.

• stack algorithm [Mattson IBM 70]
  • O(M) time per access, O(M) space
• vector tree [Bennett&Kruskal IBM 75]
  • O(log N) time per access, O(N) space
• search tree [Olken LBL 81, Sugumar&Abraham UM 93]
  • O(log M) time per access, O(M) space
• space cost remains a major problem
Approximation

- Basic idea
  - measure only the first few digits of a long distance
  - use non-unit size tree nodes
  - tree size = M / average node size
  - bound the error by tree node size
  - Guaranteed relative accuracy
    - \( a \leq \frac{\text{measured}}{\text{actual distance}} \leq 1 \)
    - e.g. \( a = 99\% \)
  - Hashable cost
  - space problem solved by Bennett-Kruskal in 1975
  - not considered in the discussion

Tree node

(time, weight)

7, 7

4, 3

10, 3

1, 1

6, 1

8, 1

11, 1

Calculate the number of last accesses since time 4 till time 12
\( \text{rd} = 6 \)

(b) Store and count only the last access of each data.
(b) Store and count only the last access of each data.

Tree node

(time, weight, capacity, size)

Search for last access of b, whose access time is 4.

(time, weight, capacity, size)

Set \( d = 1 \) first. The error in distance is at most 3.

Add node weight: \( d = 1 + 2 + 1 \). Measured distance is 4, 67% of the actual distance.

The three tree nodes have capacities 1, 2, and 6. It guarantees 33% accuracy.

Complexity

- Tree size at full occupancy (\( a \) is the accuracy, \( 1 > a > 0 \))
  - node \( i (i > 1) \) capacity and size = \( \left\lfloor \frac{1-a}{a^i} \right\rfloor \)
  - number of tree nodes = \( 2 \log_2 M + 1 \)
- Dynamic tree compression
  - compresses when below 25% occupancy
  - always increases occupancy to 50% or more
  - \( O(\log M) \) space, \( O(\log \log M) \) time per access
- Observations
  - can use any balanced tree
  - accuracy can be arbitrarily close to 1
  - \( \log \log M \) is almost constant
Locality Statistics III

Footprint

[PPOPP 2008/11, PACT 11, CCGrid 12, ASPLOS 13, CCGrid 15]
Composable Analysis of Locality

- Program/thread composition
- p threads or programs
- 2p - 1 groupings
- Data composition
- [MICRO 2014] PORPLE: Data placement optimization on GPU
  - Xipeng Shen at NCSU
- Composable analysis
  - measurement
  - footprint in component traces
  - prediction by composition
  - compute using the Footprint

Linear-time Measurement* [PACT 2011]

\[
fp(w) = m - \sum_{t=w}^{\infty} (t-w)P(rt=t)
\]

<table>
<thead>
<tr>
<th>traces</th>
<th>metrics</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>≥4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_a_a_a_...</td>
<td>P(rt=t)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1/3</td>
<td>0</td>
</tr>
<tr>
<td>... (m = 1)</td>
<td>fp(w)</td>
<td>0</td>
<td>1/3</td>
<td>2/3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>abcabcabc</td>
<td>P(rt=t)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>... (m = 3)</td>
<td>fp(w)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

* simplified for the case of infinite-length traces

Data Locality Composition

<table>
<thead>
<tr>
<th>footprint fp(w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>traces</td>
</tr>
<tr>
<td>individual</td>
</tr>
<tr>
<td>locality</td>
</tr>
<tr>
<td>measured</td>
</tr>
<tr>
<td>a_a_a_...</td>
</tr>
<tr>
<td>b_b_b_...</td>
</tr>
<tr>
<td>c_c_c_...</td>
</tr>
<tr>
<td>composed</td>
</tr>
<tr>
<td>locality</td>
</tr>
<tr>
<td>(computed)</td>
</tr>
<tr>
<td>ab_ab_ab_...</td>
</tr>
<tr>
<td>bc_bc_bc_...</td>
</tr>
<tr>
<td>a_c_a_c_c_...</td>
</tr>
<tr>
<td>abcabcabc</td>
</tr>
</tbody>
</table>

HOTL Conversion [ASPLOS 2013]

HOTL theory: \(c = fp(w), mr(c) = fp(w+1) - fp(w)\)

<table>
<thead>
<tr>
<th>window length</th>
<th>w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>≥3</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-elem</td>
<td>footprint</td>
<td>fp(w)</td>
<td>0</td>
<td>1/3</td>
<td>2/3</td>
</tr>
<tr>
<td>traces</td>
<td>cache size</td>
<td>c</td>
<td>0</td>
<td>1/3</td>
<td>2/3</td>
</tr>
<tr>
<td>miss ratio</td>
<td>mr(c)</td>
<td>1/3</td>
<td>1/3</td>
<td>1/3</td>
<td>0</td>
</tr>
<tr>
<td>all-elem</td>
<td>footprint</td>
<td>fp(w)</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>trace</td>
<td>cache size</td>
<td>c</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>miss ratio</td>
<td>mr(c)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Optimal Footprint Symbiosis in Shared Cache

Abstract—On multicore processors, applications are run sharing the cache. This paper presents online optimization to co-locate applications to minimize cache interference to maximize performance.

The paper formulates the optimization problem and solution, presents a new sampling technique for locality analysis and evaluates them in an exhaustive test of over 150 tests. For locality analysis, previous sampling was too coarse of a magnitude faster than full trace analysis. The new sampling reduces the cost by another two orders of magnitude. This tool prior work improves on-run performance by 50%, on average. The new optimization improves it by another 20%. When sampling and optimization is applied to multiple programs, the new model reduces the overall run-time over 80% compared to the prior work.

The paper develops an online algorithm for program to obtain on-run performance that is within 1.2% of the best possible performance.

Composable Modeling of GPU Cache

Modern GPUs have a heterogeneous memory hierarchy with explicitly managed memory and multiple types of cache memory. The GPU performance may vary significantly depending on which program data is placed in which type of memory. It is inefficient and even impractical to test all possible layouts.

This paper presents compositional analysis, which profiles an execution once and then predicts the miss ratio of all array placements on all cache sizes. The analysis models the data access footprint, the cache behavior, and the throughput of each memory system. The analysis captures the interaction between data and cache behavior.

The analysis is quick derived. The exponential-to-linear reduction of the analysis shows that several times of performance could be brought from the data accesses. The analysis is 96% accurate for the top rank and 99.7% for the median rank.

The paper shows two views of the analysis in a recent tool for GPU program optimization.
Array regrouping and structure splitting using whole-program reference affinity

Authors: Xie Chen Dong

Published in: Proceedings of the 2004 ACM SIGPLAN conference on Programming language design and implementation

Bibliometrics
- Downloads: 1657
- Downloads (6 months): 109
- Downloads (cumulative): 109
- Citation Count: 2

ABC Optimizer: Affinity Based Code Layout Optimization
Rahman Lavaee
Chen Ding

The University of Rochester
Dept. of Electrical & Computer Engineering
Computer Science Department
Rochester, NY 14627

Technical Report TR-987
January 2014

Safe Parallel Programming
(Behavior-oriented Parallelization)

[PLDI 2007, OOPSLA 2011]

Parallel Programming by Hints

- Possible rather than definite parallelism
  - \( ppr(U) : V \)
  - PPR means possibly parallel routine
  - \( U \) may be parallel with \( V \)

- Safety
  - sequential equivalence
  - same result as sequential execution
  - a hint may be wrong
  - lose parallelism but not correctness
  - no non-determinism, no dead/live lock, no parallel debugging
Locality

- **Timesharing**
  - A machine supporting bursts of computation on demand
- **Virtual memory**
  - The ability of a process to refer to a larger range of memory addresses than may be present in primary memory at any one time
  - OSes that manage memory for many processes at once can do a better job overall than is possible when applications handle memory management individually
  - The LRU algorithm generally performs well... programs tend to access memory locations that are local to recently accessed locations, a characteristic referred to as locality of reference.
  - Progress benefitted from theoretical work, stimulated by practical ideas, responsive to changes in technologies and needs

Summary

- **Algorithms + Data Structures + Locality = Efficient Programs**
  - Principles of locality
  - The human brain is a "memory-prediction framework," Jeff Hawkins "On Intelligence"
- **Theory**
  - Formal relations between locality metrics
  - Fundamental properties, composable analysis
- **Optimization**
  - Data and code layout
- **Programming**
  - Safe parallel Ruby