

Program-Context Specific Buffer Caching

***Feng Zhou** (zf@cs), Rob von Behren (jrvb@cs),
Eric Brewer (brewer@cs)*

System Lunch, Berkeley CS, 11/1/04

Background

- Current OS's use LRU or its close variants to manage disk buffer cache
 - Works well for traditional workload with temporal locality
- Modern applications often have access patterns LRU handles badly, e.g. one-time scans and loops
 - Multi-media apps accessing large audio/video files
 - Information retrieval (e.g. server/desktop search)
 - Databases

Outperforming LRU

- LRU works poorly for one-time scans and loops
 - Large (>cache size) one-time scan evicts all blocks
 - Large (>cache size) loops get zero hits
- Renewed interests in buffer caching
 - Adaptation: ARC (FAST'03), CAR (FAST'04)
 - Detection: UBM (OSDI'00), DEAR (Usenix'99), PCC (OSDI'04)
- State-of-art
 - Scan-resistant
 - Detection of stable application/file-specific patterns

Our work

- *Program-context specific buffer caching:*
- A detection-based scheme that classifies disk accesses by *program contexts*
 - much more consistent and stable patterns compared to previous approaches
- A robust scheme for detecting looping patterns
 - more robust against local reorders or small spurious loops
- A low-overhead scheme for managing many cache partitions

Results

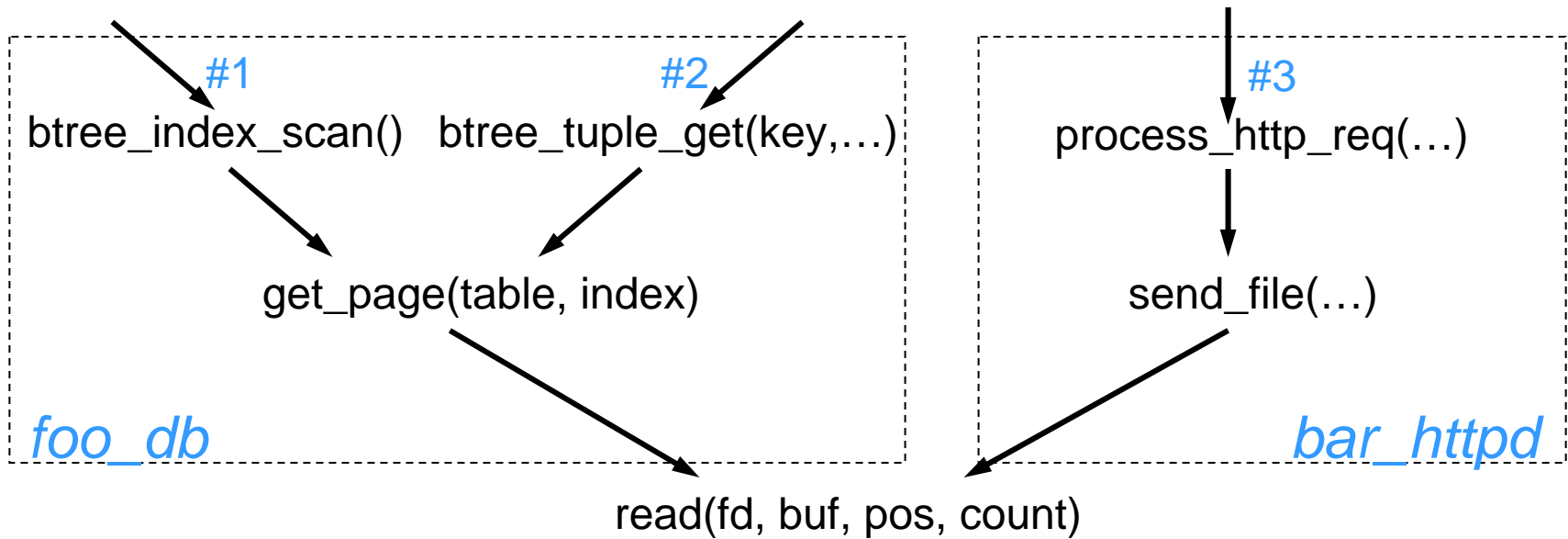
- Simulation shows significant improvements over ARC (the best alternative to LRU)
 - >80% hit rate compared to <10% of ARC for trace *gnuplot*
 - up to 60% less cache misses for DBT3 database benchmark (a TPC-H implementation)
- Prototype Linux implementation
 - shortens indexing time of Linux source tree using *glimpse* by 13%
 - shortens DBT3 execution time by 9.6%

Outline

- Introduction
- *Basic approach*
- Looping pattern detection
- Partitioned cache management
- Simulation/measurement results
- Conclusion

Program contexts

- Program context: the current PC + all return addresses on the call stack



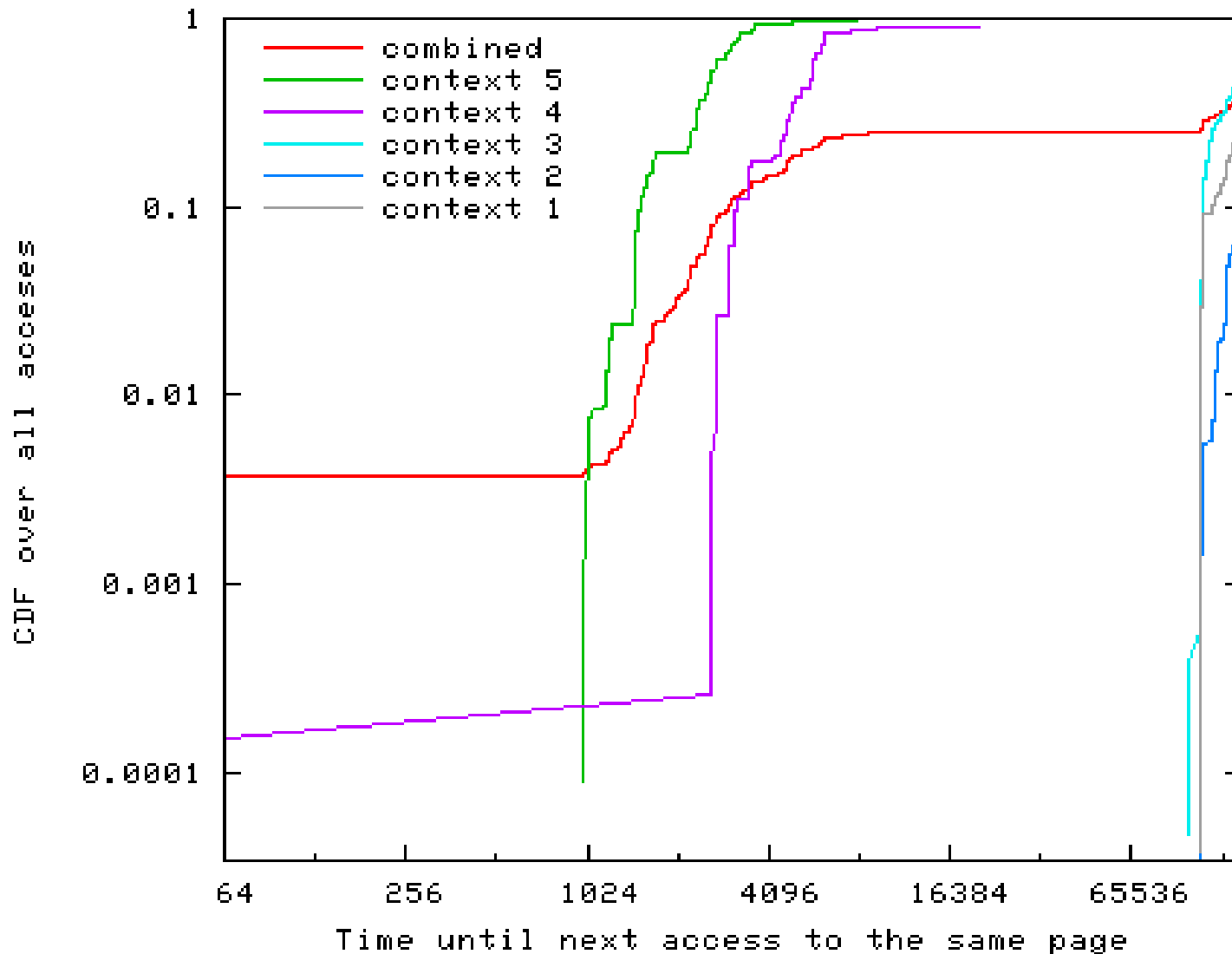
Ideal policies

#1: MRU

#2: LFU

#3: LRU

Correlation between contexts and I/O patterns (trace *glimpse*)

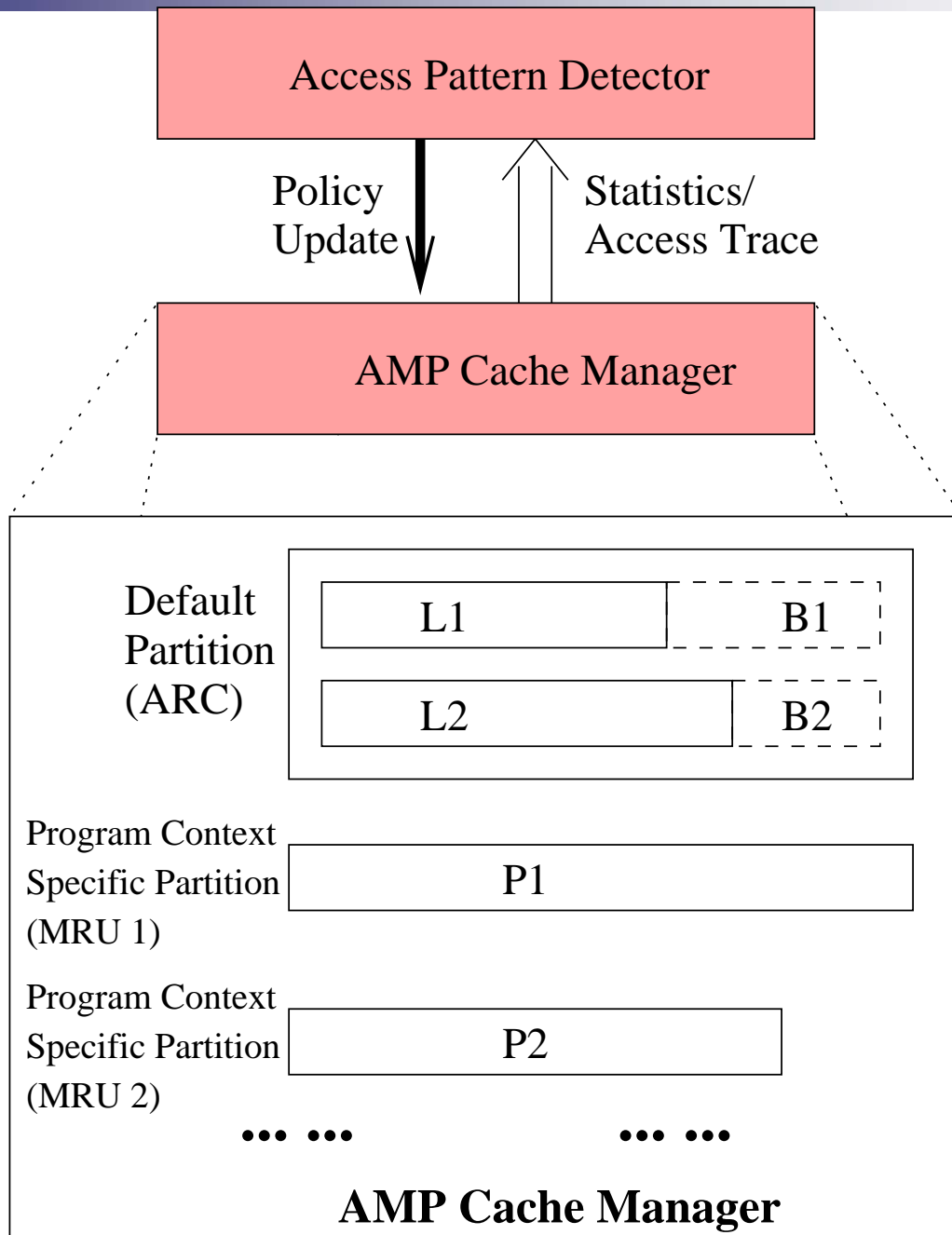


Alternative schemes

- Other possible criteria to associate access pattern with:
 - **File** (UBM). Problem case: DB has very different ways of accessing a single table.
 - **Process/thread** (DEAR). Problem case: a single thread can exhibit very different access pattern over time.
 - **Program Counter**. Problem case: many programs have wrapper functions around the I/O system call.
 - **Let the programmer dictate the access pattern** (TIP, DBMIN). Problem: some patterns hard to see, brittle over program evolution, portability issues.

Adaptive Multi-Policy (AMP) caching

- Group I/O syscalls by program context IDs
 - Context ID: a hash of PC and all return addresses, obtained by walking the user-level stack
- Periodically detect access patterns of each context
 - *oneshot, looping, temporally clustered and others*
 - expect patterns to be stable
 - detection results persist over process boundaries
- Adaptive partitioned caching
 - Oneshot blocks dropped immediately
 - One MRU partition for each major looping context
 - Other accesses go to the default ARC/LRU partition



Outline

- Introduction
- Basic approach
- *Looping pattern detection*
- Partitioned cache management
- Simulation/measurement results
- Conclusion

Access pattern detection

- Easy: detecting one-shot contexts
- Not so easy: detecting loops with irregularities
 - 123456 12**4**356 12**4**56 12**3**756
- Previous work
 - UBM: counting physically sequential accesses in a file
 - PCC: counting cache hits and new blocks for each context
 - DEAR: sorting accesses according to last-access times of the blocks they access
 - For loops, more **recently**-accessed blocks are accessed again **farther** in the future
 - The contrary for temporally clustered
 - Group nearby accesses to cancel small irregularities

Loop detection scheme

- Intuition: measure the *average recency* of the blocks accessed
- For the i -th access
 - L_i : list of all previously accessed blocks, ordered from the oldest to the most recent by their last access time.
 - $|L_i|$: length of L_i
 - p_i : position in L_i of the block accessed (0 to $|L_i|-1$)
- Define the *recency* of the access as,

$$R_i = \begin{cases} p_i / (|L_i| - 1), & |L_i| > 1 \\ 0.5, & |L_i| = 1 \\ \perp, & \text{undefined for first access} \end{cases}$$

Loop detection scheme cont.

- *Average recency* \bar{R} of the whole sequence is the average of all defined R_i ($0 \leq \bar{R} \leq 1$)
- Detection result:
 - *loop*, if $\bar{R} < T_{loop}$ (e.g. 0.4)
 - *temporally clustered*, if $\bar{R} > T_{tc}$ (e.g. 0.6)
 - *others*, o.w. (near 0.5)

Loop detection example 1

- Block sequence [1 2 3 1 2 3]

i	block	L_i	p_i	R_i
1	1	empty	\perp	\perp
2	2	1	\perp	\perp
3	3	1 2	\perp	\perp
4	1	1 2 3	0	0
5	2	2 3 1	0	0
6	3	3 1 2	0	0

- $\bar{R} = 0$, detected pattern is *loop*

Loop detection example 2

- Block sequence [1 2 3 4 4 3 4 5 6 5 6]

i	block	L_i	p_i	R_i
1	1	empty	\perp	\perp
2	2	1	\perp	\perp
3	3	1 2	\perp	\perp
4	4	1 2 3	\perp	\perp
5	4	1 2 3 4	3	1
6	3	1 2 3 4	2	0.667
7	4	1 2 4 3	2	0.667
8	5	1 2 3 4	\perp	\perp
9	6	1 2 3 4 5	\perp	\perp
10	5	1 2 3 4 5 6	4	0.8
11	6	1 2 3 4 6 5	0	0.8

Loop detection example 2 cont.

- $\bar{R} = 0.79$, detected pattern is *temporally clustered*
- Comments
 - The average recency metric is robust against small re-orderings in the sequence
 - Robust against small localized loops in clustered sequences
 - $O(m*n)$, m: number of unique blocks, n: length of sequence
 - Cost can be reduced by sampling blocks (not sampling accesses!)

Detection of synthesized sequences

AMP \overline{R}	tc 0.755	loop 0.001	loop 0.347	tc 0.617	loop 0.008	loop 0.010	other 0.513
DEAR	other	loop	<i>other</i>	other	loop	<i>other</i>	other
PCC	<i>loop</i>	loop	loop	<i>loop</i>	loop	<i>other</i>	<i>loop</i>

“tc”=temporally clustered

Colored detection results are wrong

Classifying *tc* as *other* is deemed correct.

Outline

- Introduction
- Basic approach
- Looping pattern detection
- *Partitioned cache management*
- Simulation/measurement results
- Conclusion

Partition size adaptation

- How to decide the size of each cache partition?
- Marginal gain (MG):
 - the expected number of extra hits over unit time if we allocate another block to a cache partition
 - used in previous work (UBM, CMU's TIP)
- Achieving local optimum using MG
 - MG is a function of current partition size
 - assuming monotonously decreasing
 - locally optimal when every partition has the same MG

Partition size adaptation cont.

- Let each partition grow at a speed proportional to MG (by taking blocks from a random partition)
- Estimate MG
 - ARC: ghost buffer (a small number of inexistent cache blocks holding just-evicted blocks)
 - Loops with MRU: $1/loop_time$
- Adaptation
 - Expand the ARC partition by one if ghost buffer hit
 - Expand an MRU partition by one every $loop_size/ghost_buffer_size$ accesses to the partition

Correctness of partition size adaptation

- During time period t , number of expansion of ARC is:

$$m = t(|B_1| + |B_2|) \cdot MG_{ARC} \quad (|B_1| + |B_2| = ghost_buffer_size)$$

- The number of expansions of an MRU partition i is

$$\begin{aligned} m' &= \frac{t \cdot loop_size}{loop_time} \cdot \frac{|B_1| + |B_2|}{loop_size} \\ &= \frac{t(|B_1| + |B_2|)}{loop_time} \\ &= t(|B_1| + |B_2|) \cdot MG_i \end{aligned}$$

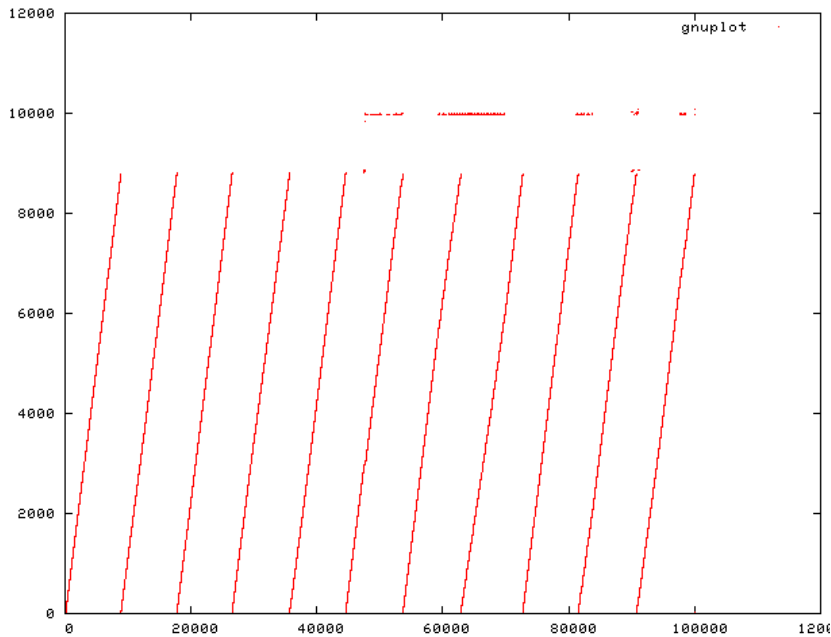
- They are both proportional to their respective MG with the same constant

Outline

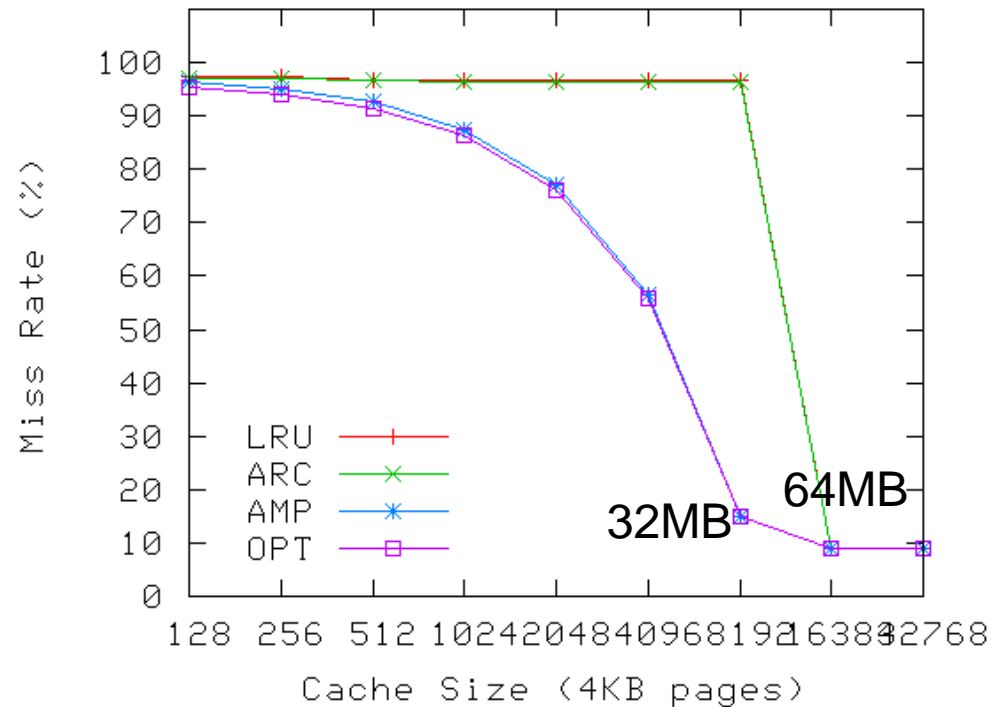
- Introduction
- Basic approach
- Looping pattern detection
- Partitioned cache management
- *Simulation/measurement results*
- Conclusion

Gnuplot

- Plotting (35MB) a large data file containing multiple data series



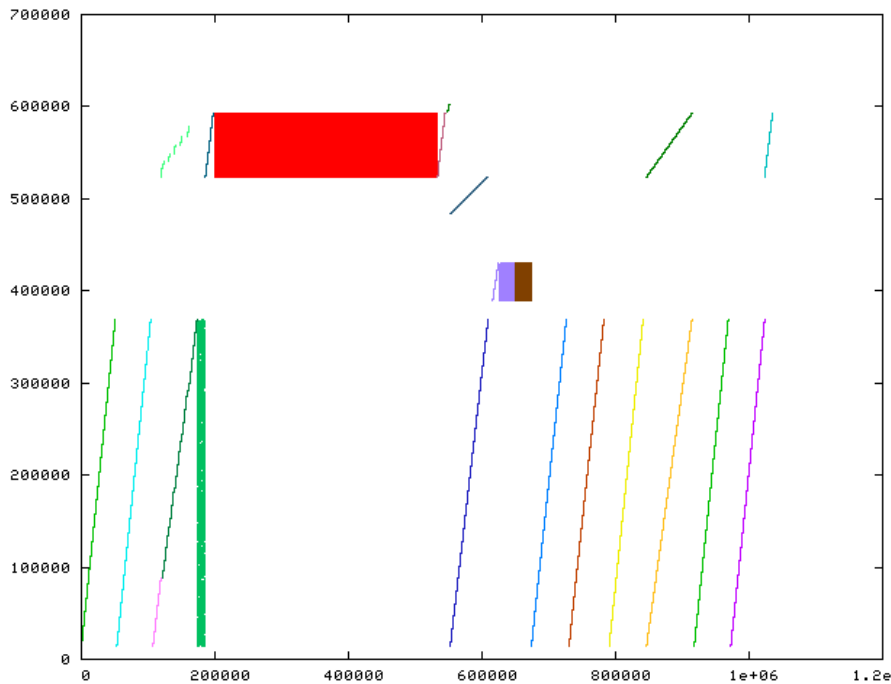
Access sequence



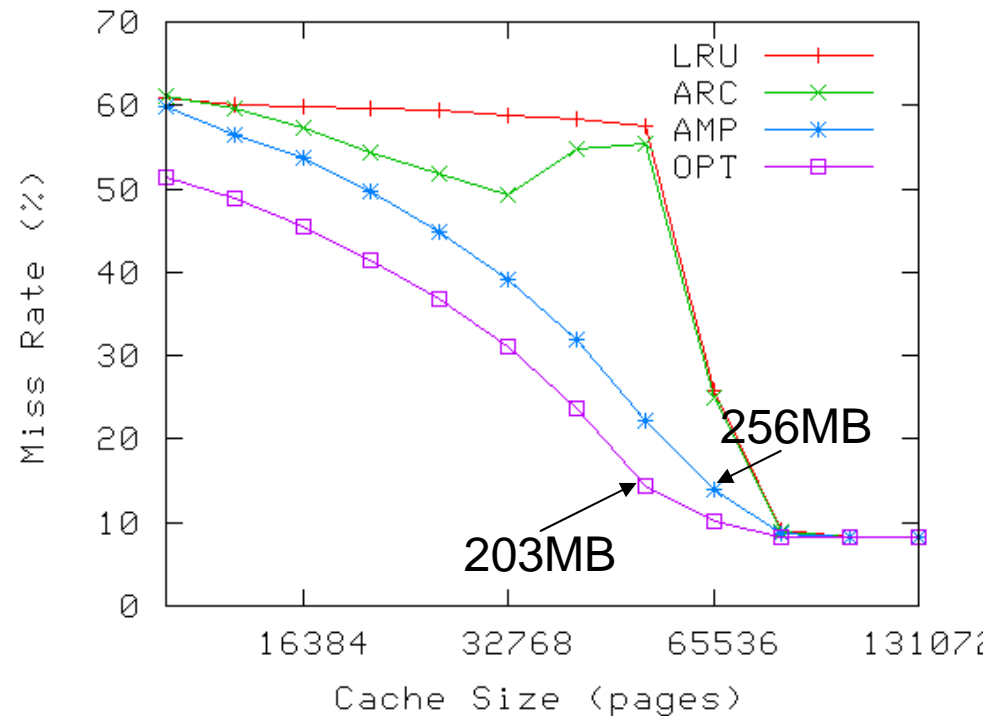
Miss rates vs. cache sizes

DBT3

- Implementation of TPC-H db benchmark
- Complex queries over a 5GB database (trace sampled by 1/7)



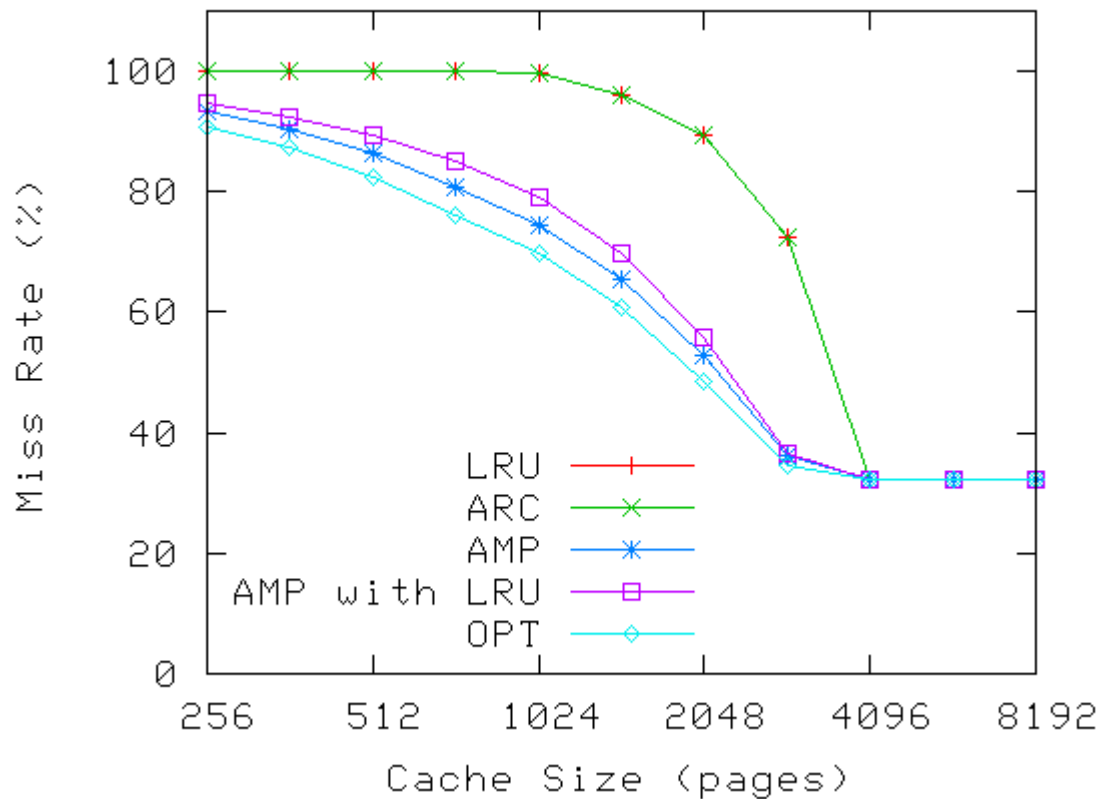
Access sequence



Miss rates vs. cache sizes

OSDB

- Another database benchmark
- 40MB database



Linux Implementation

■ Kernel part for Linux 2.6.8.1

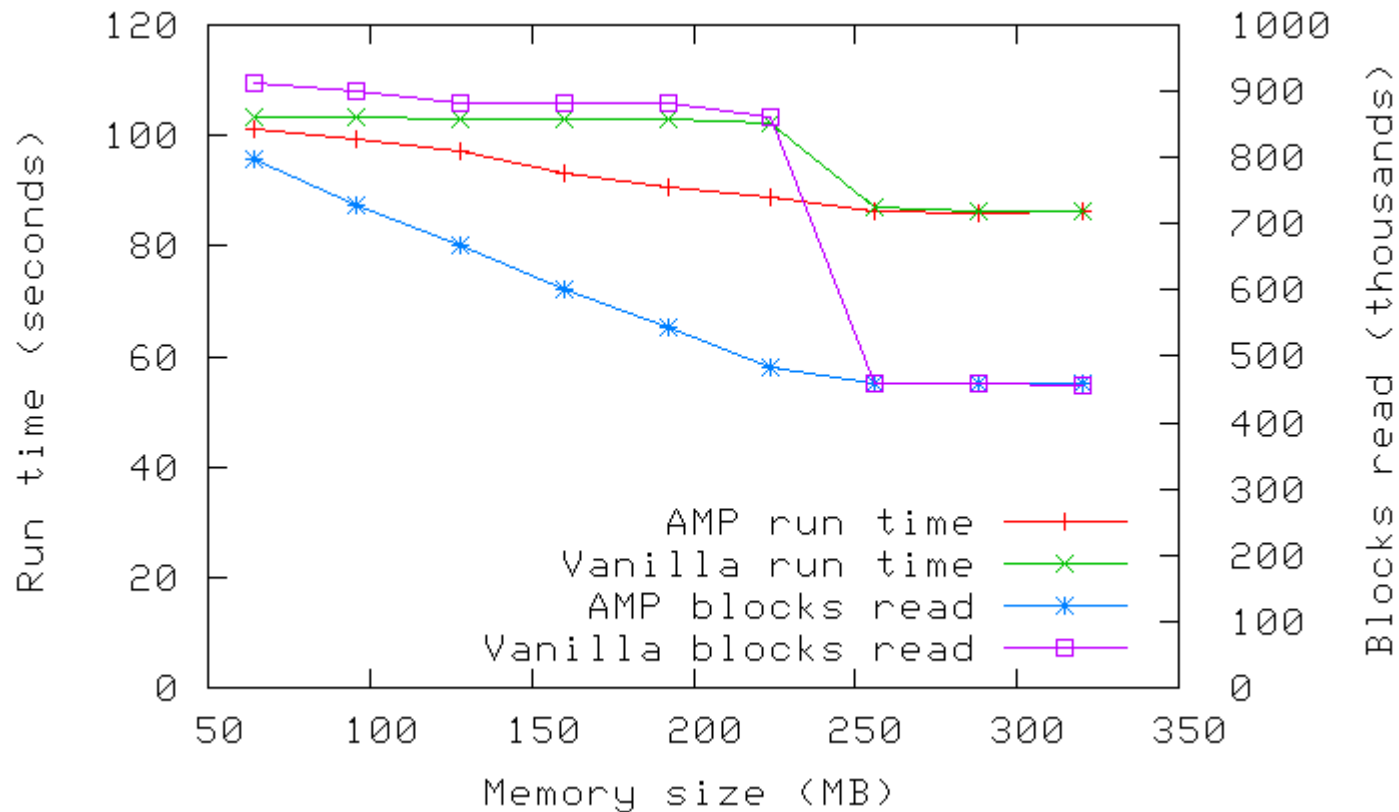
- Partitions the Linux page cache
- Default partition still uses Linux's two-buffer CLOCK policy
- Other partitions uses MRU
- Collects access trace

■ User-level part in Java

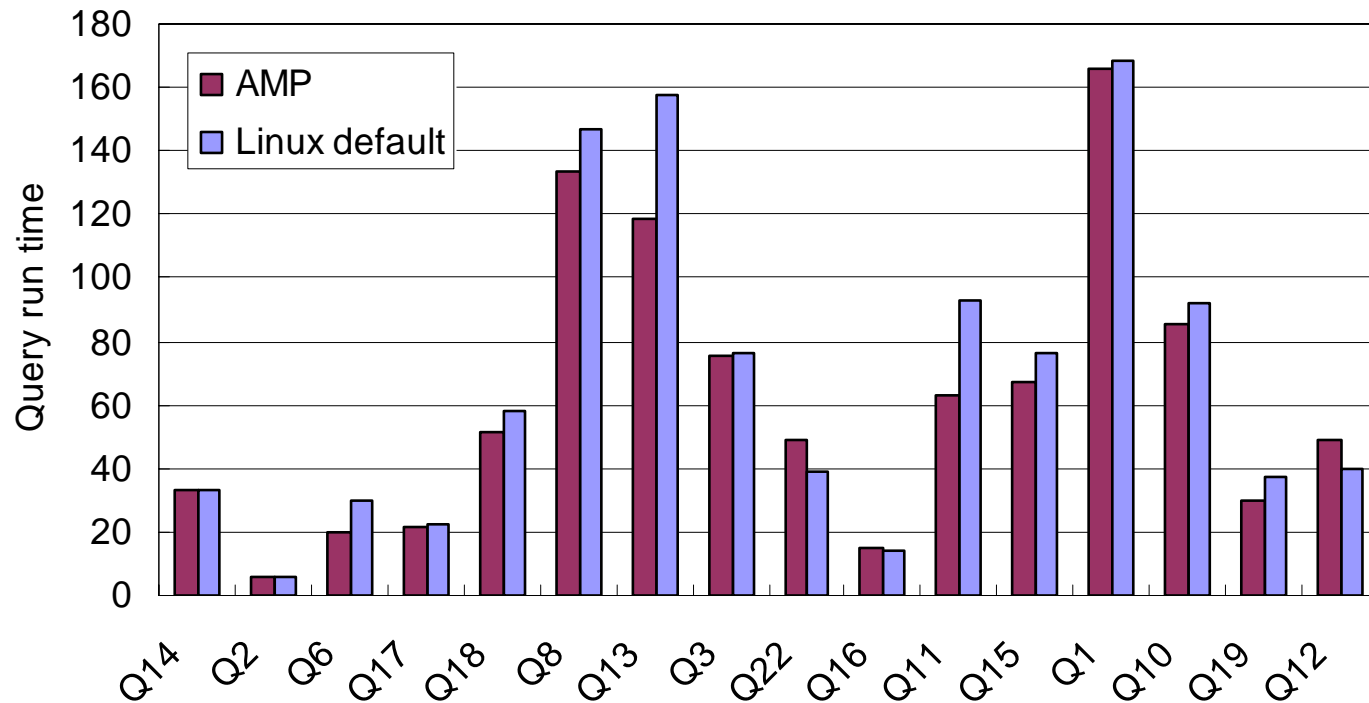
- Does access pattern detection by looking at the access trace
- The same code used for access pattern detection in simulation

Glimpse on Linux implementation

- Indexing the Linux kernel source tree (220MB) with *glimpseindex*



DBT3 on AMP implementation



- Overall execution time reduced by 9.6% (1091 secs → 986 secs)
- Disk reads reduced by 24.8% (15.4GB → 11.6GB), writes 6.5%

Conclusion

- AMP uses program context statistics to do fine-grained adaptation of caching policies among code locations inside the same application.
- We presented a simple but robust looping pattern detection algorithm.
- We presented a simple, low-overhead scheme for adapting sizes among LRU and MRU partitions.

Conclusion cont.

- AMP significantly out-performs the best current caching policies for application with large looping accesses, e.g. databases.
- Program context information proves to be a powerful tool in disambiguating mixed behaviors within a single application. We plan to apply it to other parts of OS as future work.

Thank You!