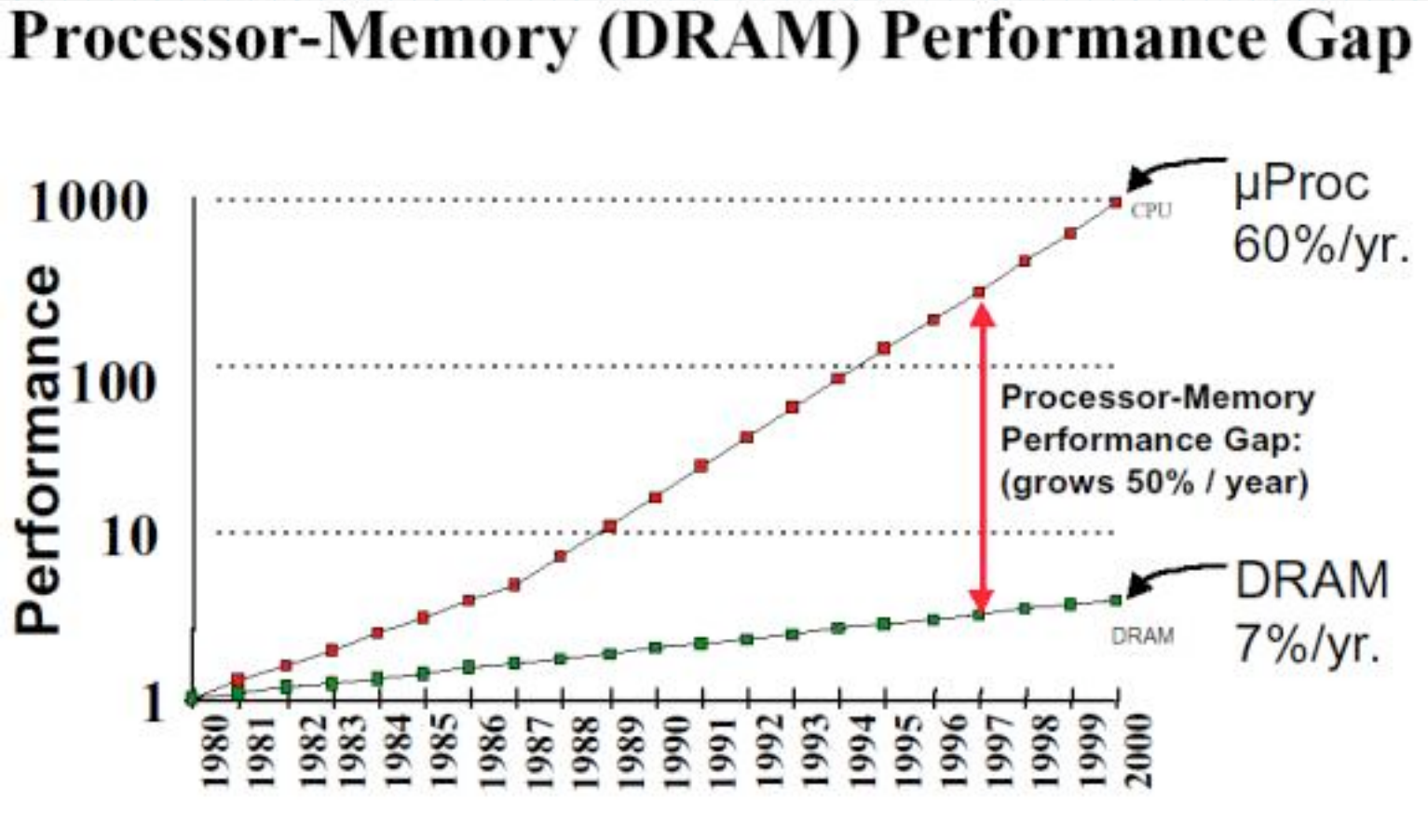
Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality

Trishul Chilimbi

15-745 Optimizing Compilers
Spring 2006

Sean McLaughlin

Memory optimizations are important!



Why Cache-Friendly Code is Important

Cache type	Size of item (bytes)	Latency (cpu cycles)
Registers	4 bytes	0
L1 Cache	32 bytes	1
L2 Cache	32 bytes	10
Main Memory	4-KB pages	100
Disk		millions

Outline

* Background
* Defining locality
* Measuring locality
* Exploiting locality

Defining locality (textbook)

* temporal locality - programs reference data items that were recently referenced themselves
* spacial locality - programs reference data items that are close to recently referenced items
* Note: definitions give no metric

improving locality

- **Clustering**: Put items frequently accessed together on the same page in memory
- **Clustering II**: Align items accessed together so they land in different cache lines
- **Pre-fetching**: Load data from a lower memory layer to a higher if its use is expected in the near future

The good news

* Control flow graphs and program paths (Larus) capture dynamic control flow, allowing for good instruction cache behavior.
* Aggregate load/store analysis can yield decent page-level clustering.

The bad news

* Caches are too small for simple page clustering to be effective.
* Aggregate data access information is not sufficient for cache-level layout.
* Static analysis too complex on modern architectures, so use a trace
* Access traces are too large to analyze quickly.
* Need for sequences, rather than individual accesses, prevent statistical sampling.

Problem

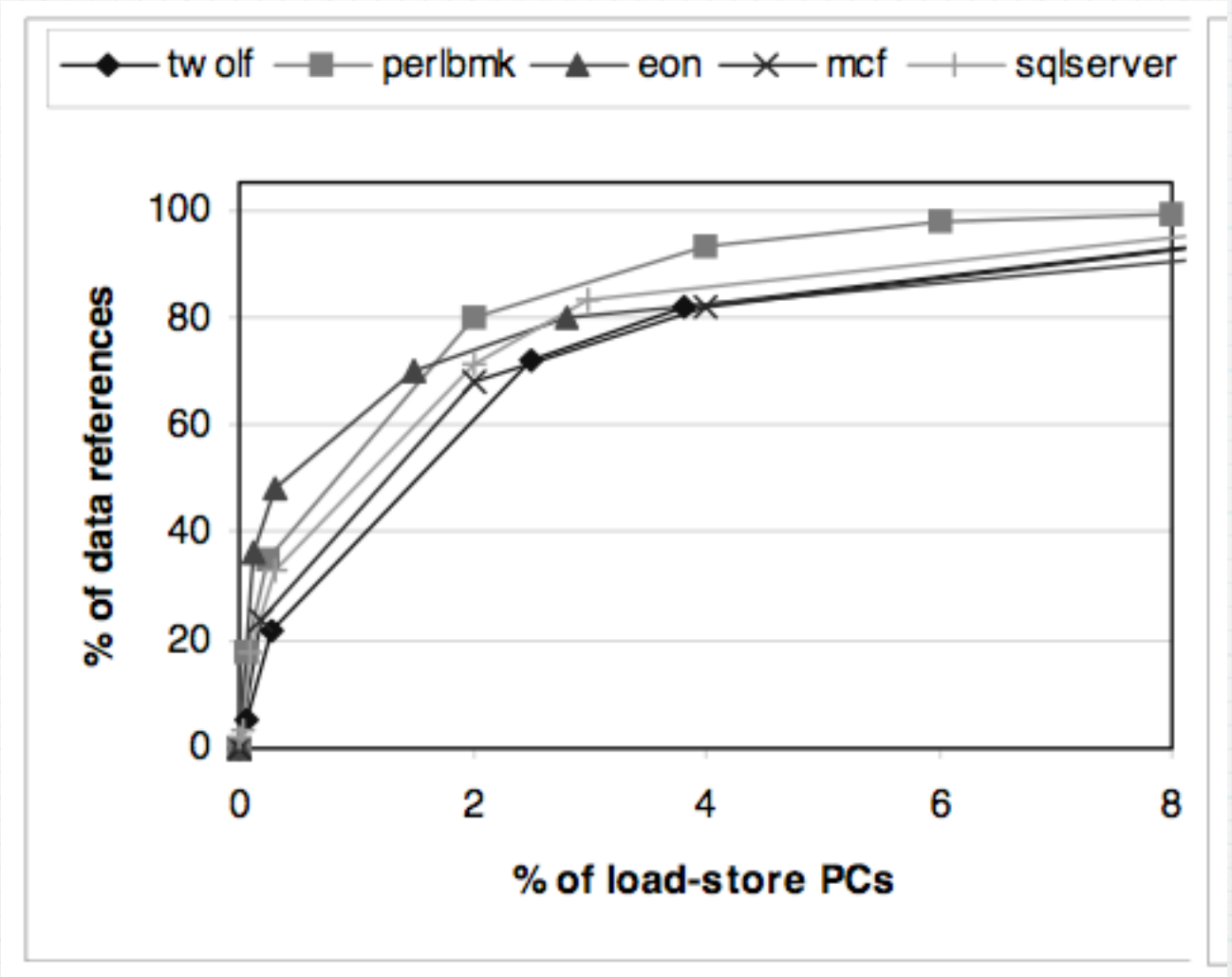
* need data reference abstractions to identify and measure locality (analogous to hot program paths)
* need efficient data reference representation (analogous to Whole Program Paths)

Outline

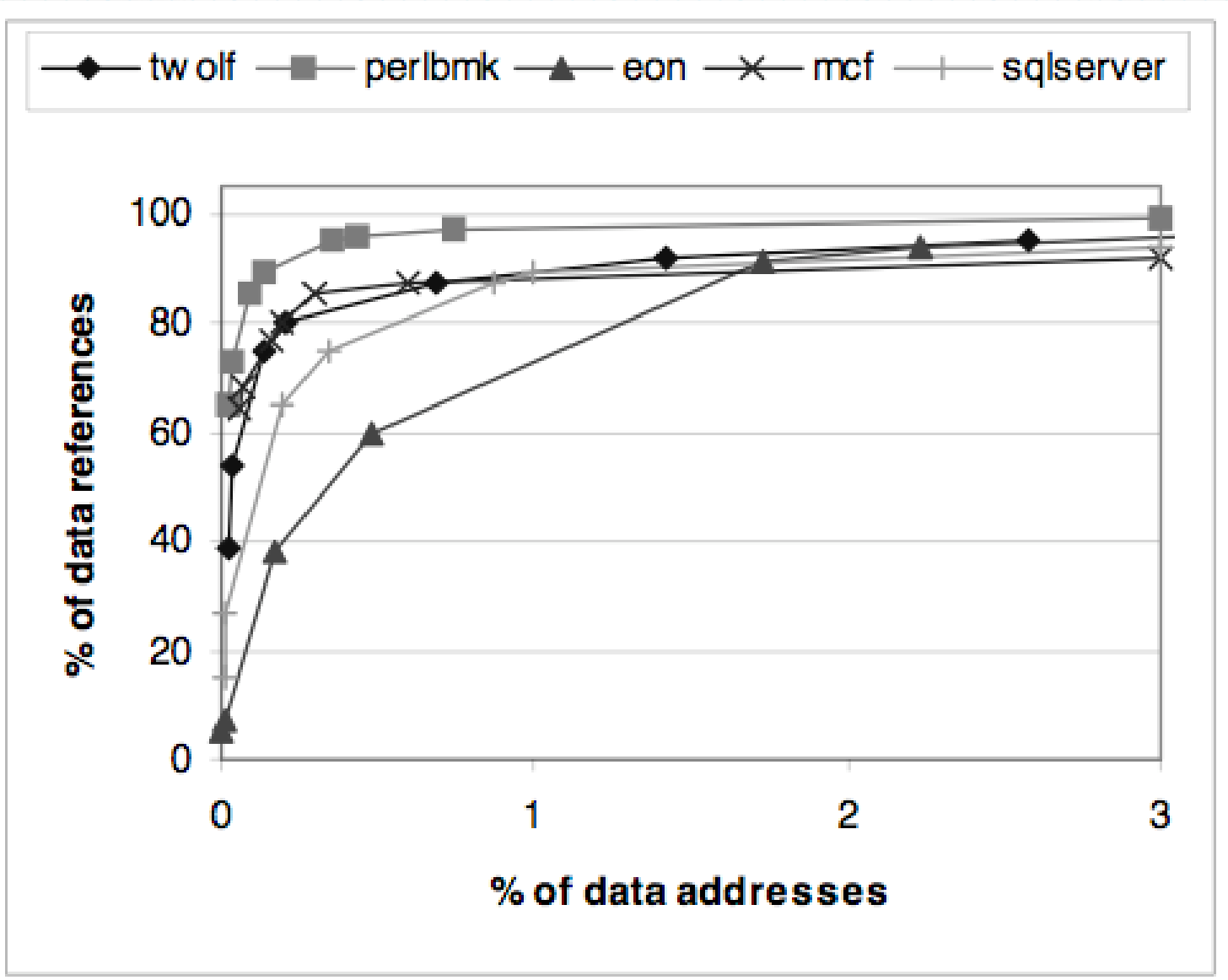
* Background
* Defining locality
* Measuring locality
* Exploiting locality

Defining locality (informally)

* the most recently used data is likely to be accessed again in the near future
* Good locality implies a large skew in the reference distribution.
* 90/10 rule for data



Locality in terms of hottest load/store instructions



Locality in terms of data addresses

Defining locality (formally)

* To be exploitable by cache opts, data references must
  * exhibit reference locality
  * exhibit regularity
* regular + ref locality = exploitable locality

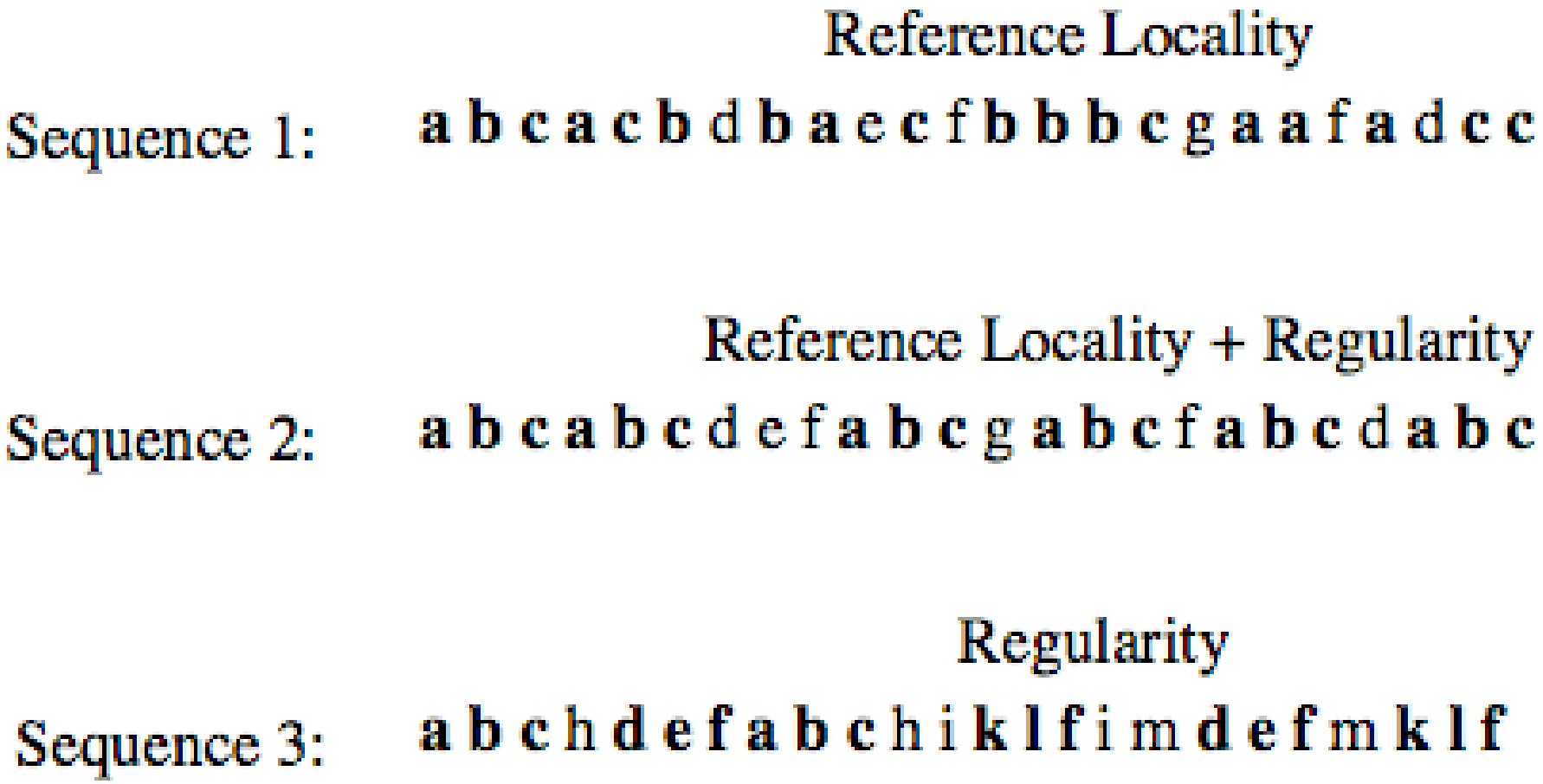


Figure 2. Data reference sequence characteristics.

Abstraction: data streams

* A data stream is a subsequence that exhibits regularity
* A hot data stream also covers a large amount of the data references
* We formally define exploitable locality in terms of hot data streams

measuring locality

* We want to measure locality, as it can identify opt targets
* Standard "definitions" are vague

measuring locality

* Inherent exploitable spatial locality = weighted average of spatial regularity across hot data streams (weight=magnitude)
* Inherent exploitable temporal locality = average HDS temporal regularity
* Realized exploitable locality = cache block packing efficiency = min/actual cache blocks needed to store stream

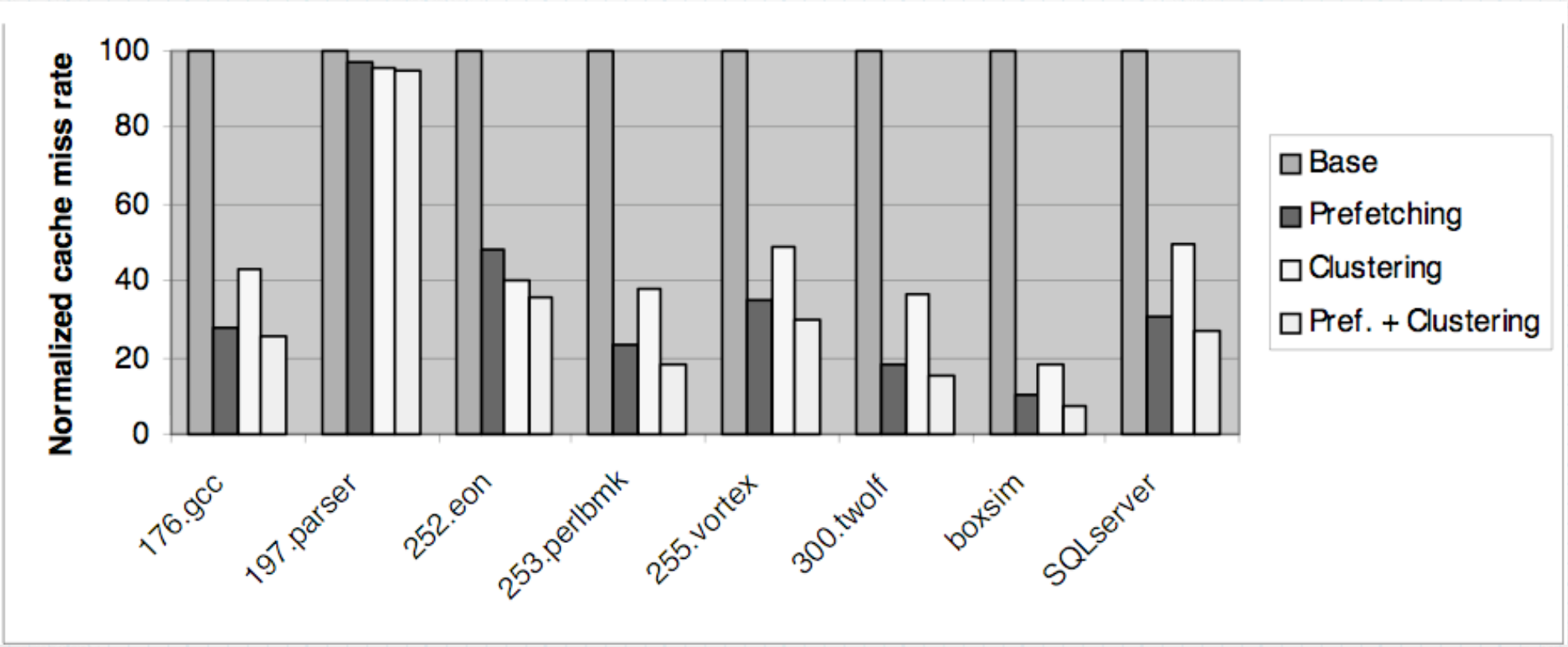
Exploiting locality

- Hot data streams + locality metric = improved data reference locality
  - identify suboptimal programs
  - focus opts on particular streams
  - identify salient optimizations

Exploiting locality

* measures can be used to determine what combination of clustering and prefetching will be most effective, eg.
  * hot streams with poor temporal locality are served by prefetching (not clustering)
  * streams with poor packing efficiency can be helped by clustering

Results


Normalized cache miss rate
100
80
60
40
20
0
176.gcc
197.parser
252.eon
253.perlbmk
255.vortex
300.twolf
boxsim
SQLserver
Base
Prefetching
Clustering
Pref. + Clustering

Questions

* How is it possible that optimizing a program with such a fine grain detail helps other runs?
* The *measurement* of locality is wrt a particular trace of a program, even for inherent locality. Can this be made more general?

Questions

* Problems with the scheme?
* Runtime improvements?
* How do these memory opts interact with the scalar opts of an aggressive compiler?
* What about programs with sensitive input behavior? (cf. generational GC, which often behaves well, but also works terribly in some instances)

The End