Techniques for Efficiently Allocating Persistent Storage

Arun Iyengar, Jim Challenger

IBM T. J. Watson Research Center

Hawthorne, New York

Shudong Jin

Department of Computer Science

Boston University

Overview

- Motivation For This Work
- Disk Management
- Memory Management: PMFLF
- Measurements
- Summary

Motivation

- We have a need for efficient persistent memory to replace or augment main memory
  - Manage very large data structures (large)
  - Low startup and shutdown time (persistent)
  - Move and replicate structure (mobility)
  - Low overhead (efficient)

- Simple lightweight interfaces:
  - Low level API: malloc() and free()
  - Java collection classes: Hashtable, Btree, Queue

Motivation

Sydney 2000 use:
•Object Dependence Graph (1.3 GB, 750,000 objects)
•Persistent fragment cache (4 GB, 750,00 objects)
•Persistent expiration table (1,000,000 entries)

Other Proposed Uses
•Routing tables
•Proxy Caches
•Persistent Worlds

Related Work

DBM, NDBM, GDBM, JDBM, Perl DBM etc.

- Object size restrictions
- Copyright problems
- Bad fragmentation characteristics
- Clumsy API
- Scalability problems

Rogue-Wave Btree and Hashtable on Disk

- Undesirable fragmentation characteristics
- Inefficient allocation
- Scalability problems

Disk Management Highlights

- In-Memory freelists for fast allocation and deallocation
- Most allocations and deallocations require single seek/write
- Periodic checkpoint of disk structures amortizes cost
- Deferred coalescing
- Checkpoint freelists on disk on clean shutdown for fast start
- Reconstruct freelists from disk scan after system failure

Disk Management I

Representation on Disk:

Allocation Status	Size	Data	Allocation Status	Size	Data	Allocation Status	Size	Data
+	32		-	512		-	64	

Allocation:

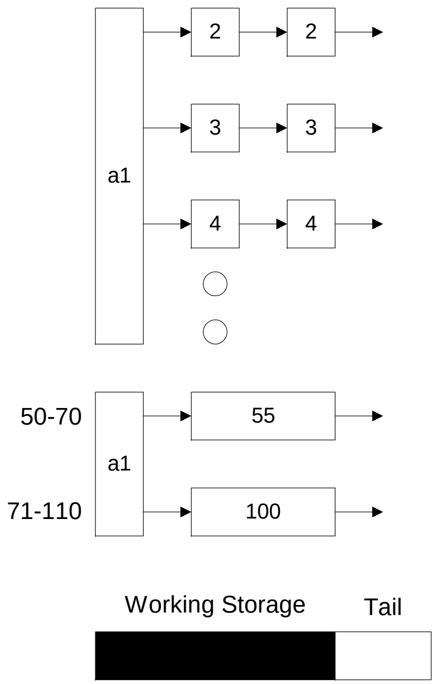
1 seek if block is not split and not tail allocation

Deallocation:

1 seek if coalescing not required

Memory Management - PMFLF

•Based on Multiple Free List Fit I (MFLF I)
•Multiple quick lists reduce splits
•Multiple misc lists for large blocks
•Acceptable waste permits “close” fit on misc list to reduce splits
•Defer tail pointer updates
•Defer coalesce
•Adaptive behavior
•Works well with Disk Management I



PMFLF Experience

Did not coalesce:

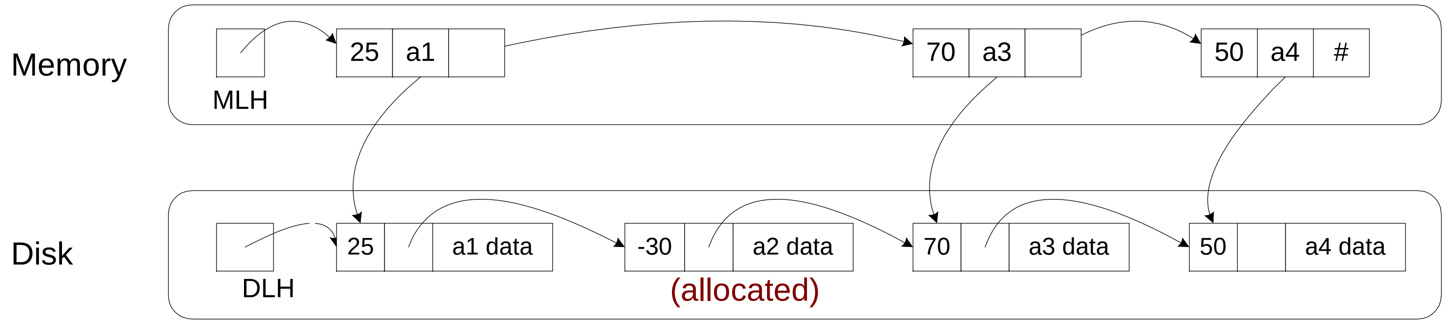
- No fragmentation for ODG, expiry table and one of the fragment repositories.
- Some fragmentation in other fragment repository. Better tuning of quicklists to data would have solved this.

Fragment caches grew to total of 4GB.

No performance problems.

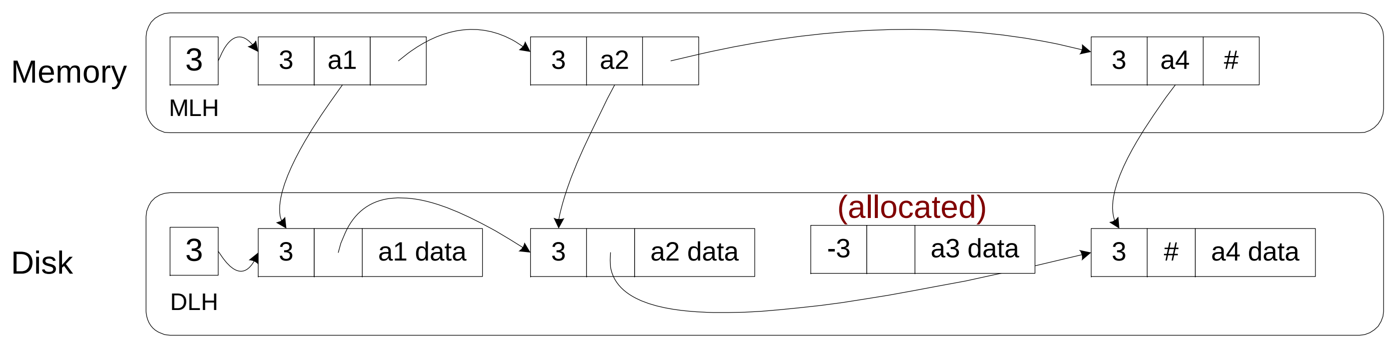
Disk Management II

- Maintain lists on disk as well as in memory
- Lists on disk contain both allocated and unallocated blocks
- Allocation / deallocation updates single bit on disk
- Can defer update of Disk List Head (DLH) when new blocks are added
- Some blocks “lost” from lists, if failure between checkpoints
- Data is never lost to applications
- Lost blocks can be recovered by header scan and coalesce
- Amortized cost of DLH updates is low



Disk Management III

- Maintain lists on disk as well as in memory
- Lists on disk contain only unallocated blocks
- Allocation / deallocation updates single bit on disk
- Can defer removal of allocated blocks from disk lists
- Can defer update of DLH when new blocks are added
- Some blocks “lost” from lists if failure between checkpoints.
- Data is never lost to applications.
- Lost blocks can be recovered by coalesce operations
- Amortized cost of DLH updates is low



Measurements

Java2 Hashmap implemented over:

- •File System (one file per object)
- •Database System
- •Single file managed by PMFLF, Disk Management I

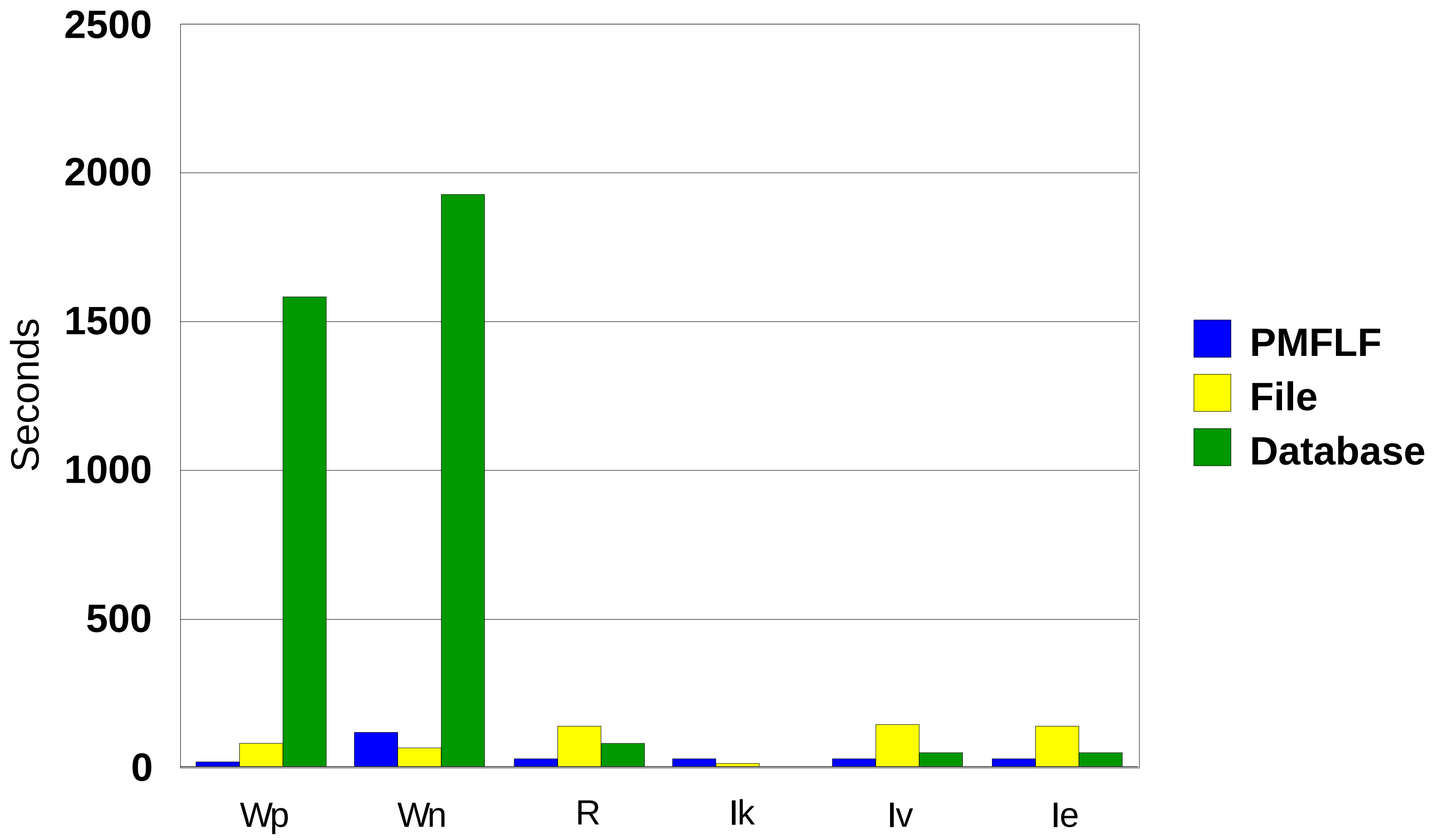
The Operating System

- •Linux RedHat 6.0 Kernel 2.2.13 (Basic Tests)
- •NT 4.0 for throughput tests

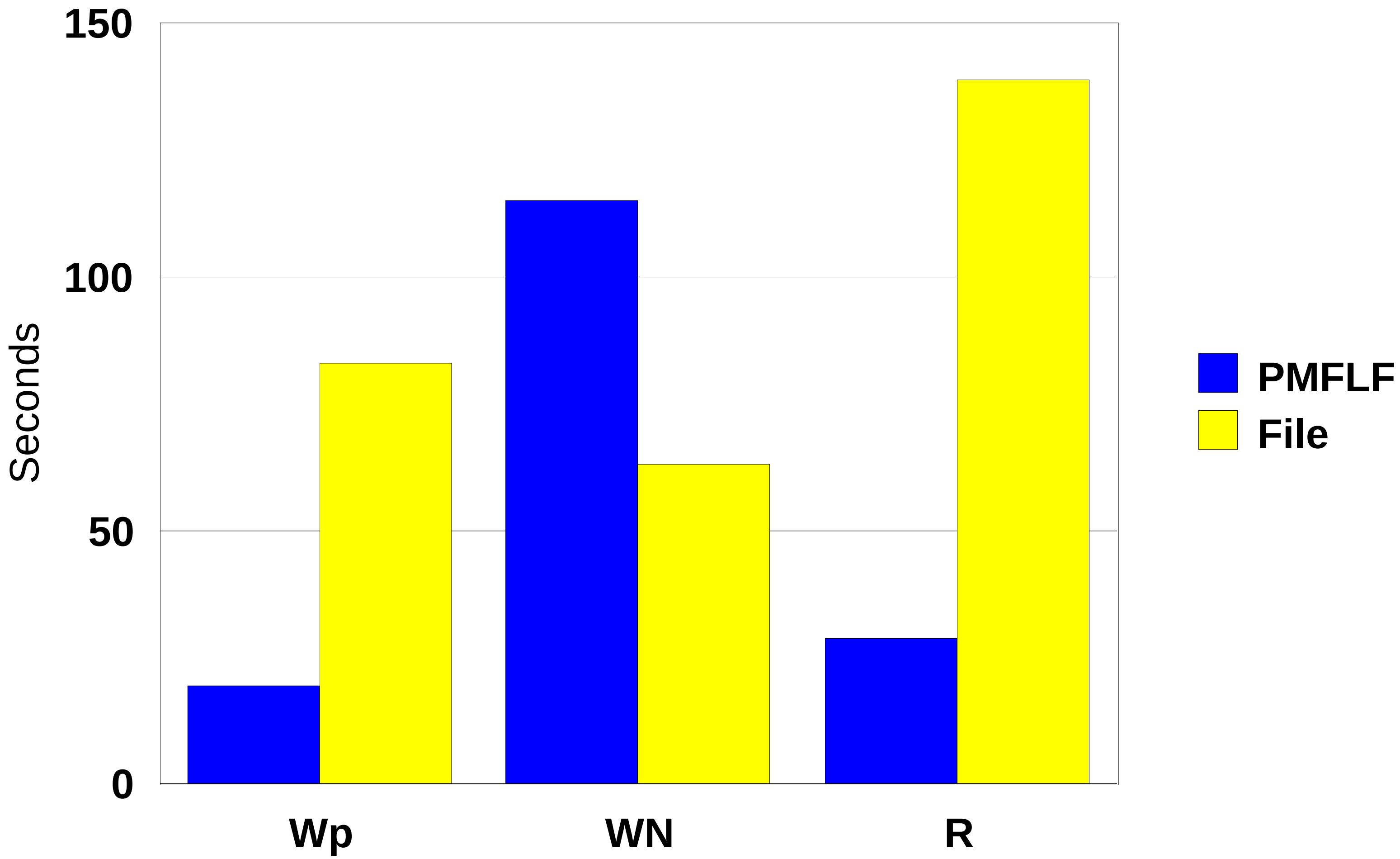
The Data

- •Object Dependence Graph (March, 2000) with 27,800 objects for basic tests
- •ODG and Dec Proxy Log for throughput tests

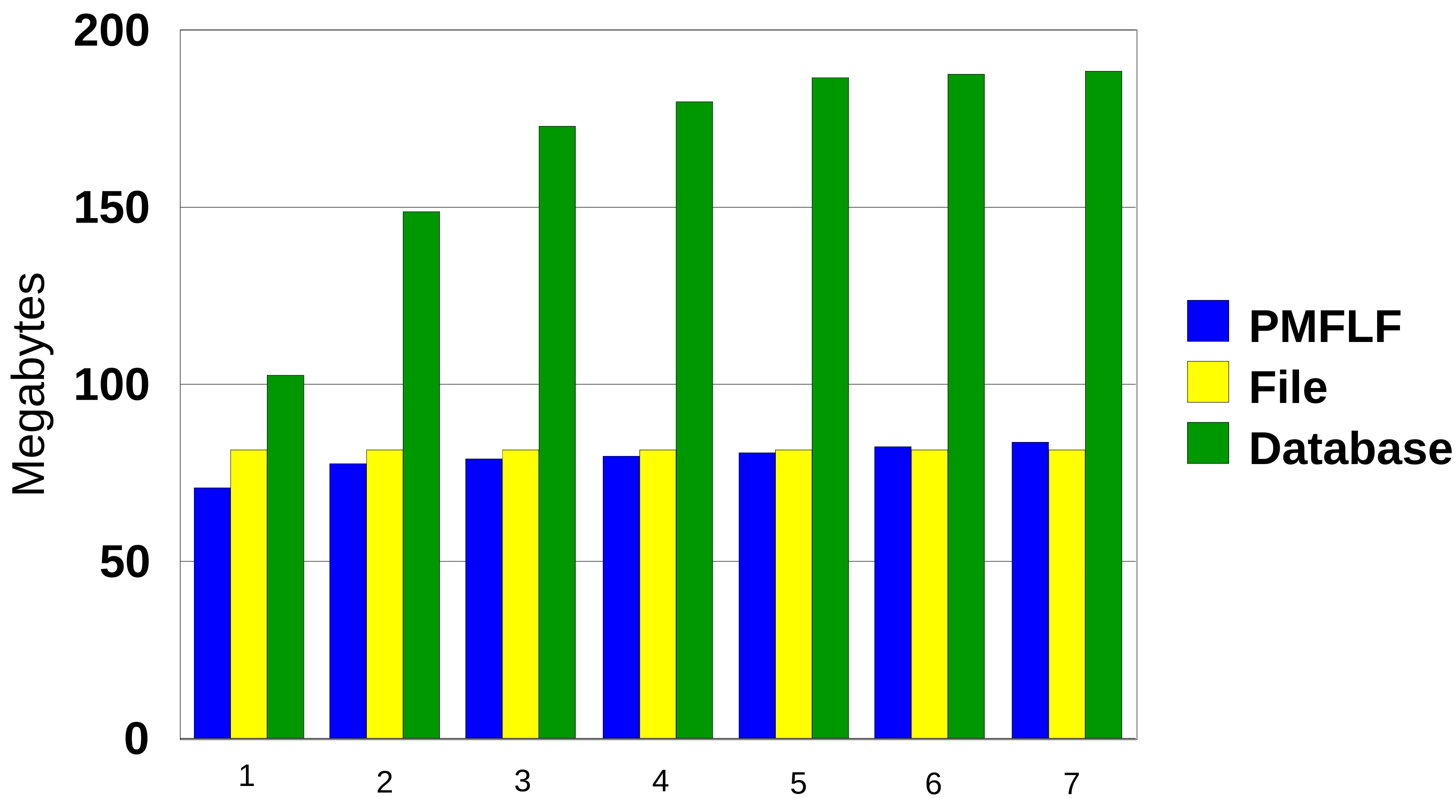
Timings, 27.8K Items



Timing Details, PMFLF vs File



Database Growth



Summary

- Algorithms minimize seek and write operations
- Disk Management I + PMFLF Used in Sydney Server (Java)
- Can be implemented over raw disk or within a file
- Extremely useful for Web workloads
    - Persistence of structure
    - Large structures don’t strain real memory
    - Simple programming model (hash table interface)
    - No restrictions on object size