Memory Management Thread for Heap Allocation Intensive Sequential Applications

Devesh Tiwari, Sanghoon Lee, James Tuck and Yan Solihin
ARPERS Research Group
Electrical and Computer Engineering
NC State University

Motivation

- Dynamic memory operations expensive and ubiquitous
  - Factorization algorithm, object oriented robotics package
  - Language processing, dataflow constraint solvers
  - Minimum spanning tree problems

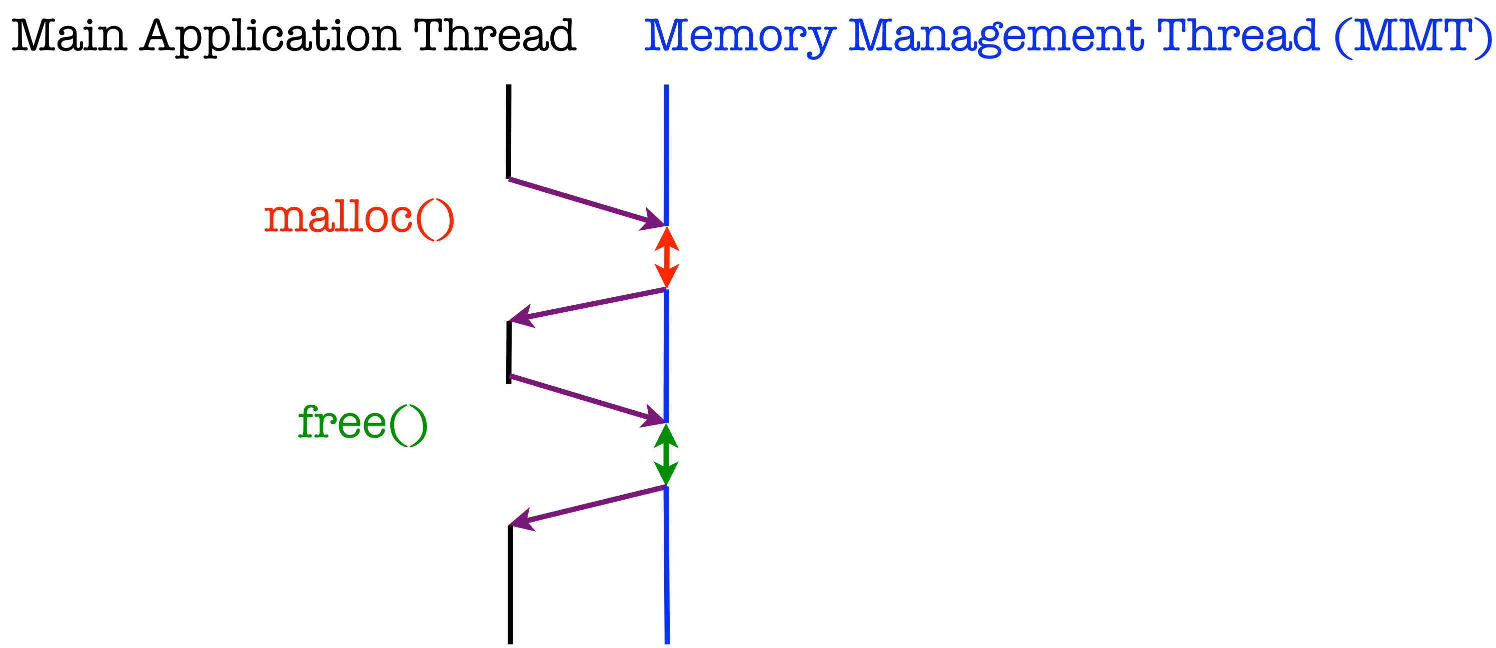
- Object-Oriented Programming Languages (C++)
  - More re-usable, extensible and modular
  - Syntactic and Semantic both constructs
    - new[] delete[] constructor() destructor()
  - Historical reasons (C Vs C++)

- Exploiting Multi Core Parallelism for Heap Intensive Sequential Applications

Outline

- Motivation
- Memory Management Thread : MMT Approach
  - Overview
  - Challenges and Contributions
- MMT Design and Implementation
- Evaluation
- Conclusion and Future Work

Overview

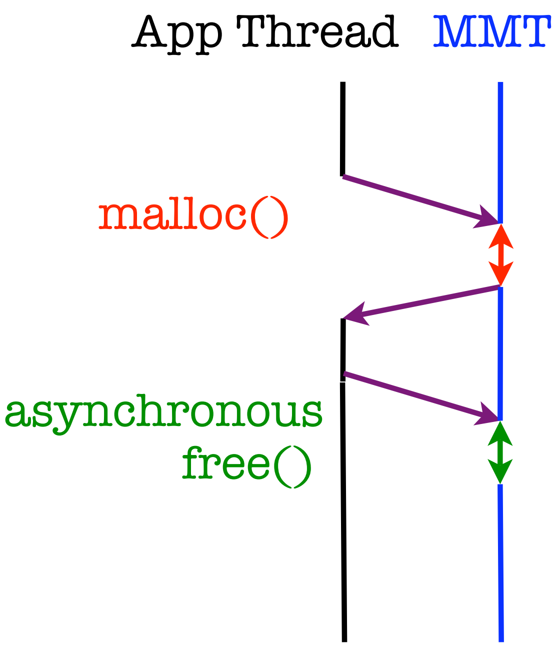


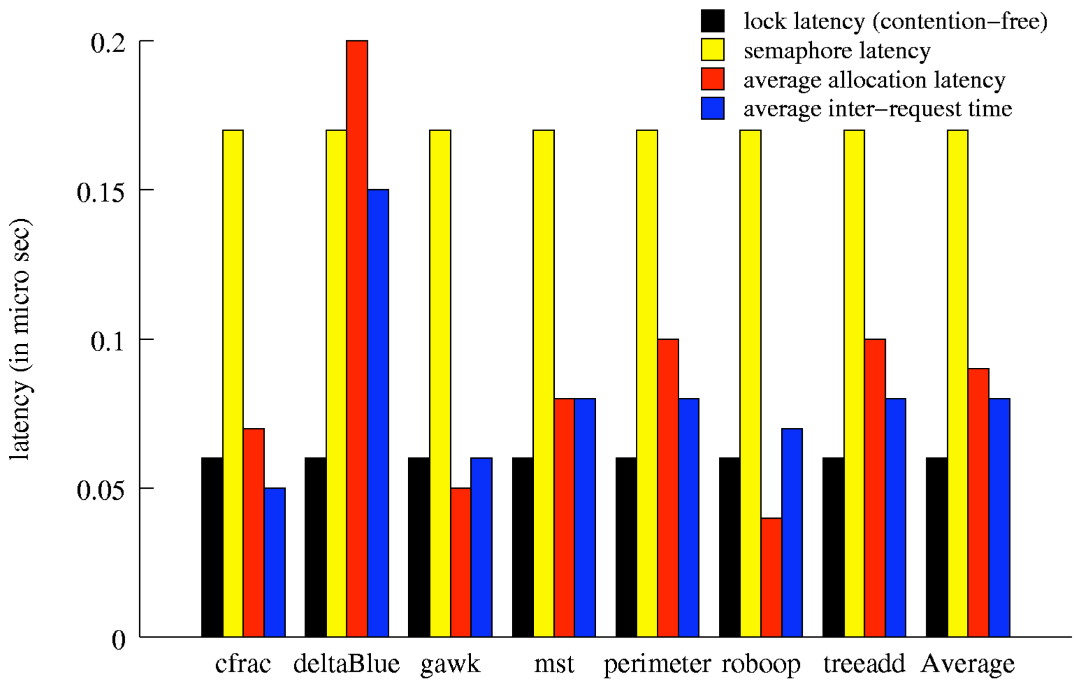
Decoupling dynamic memory management from main application routines

Investigating the approach of designing and implementing
a dedicated memory management thread (MMT) for sequential applications

Challenge

Why is it so challenging to speed up applications using a dedicated MMT?





- Need to exploit "enough" parallelism between application thread and MMT
- Need to reduce offloading latency for such fine grain tasks

Contributions

Memory Management Thread (MMT) Approach for Speeding up Heap Allocation Intensive Sequential Applications

- Exploiting the parallelism between main application thread and MMT
- Reducing the communication cost between main application and MMT
- Being agnostic to underlying memory management library algorithm

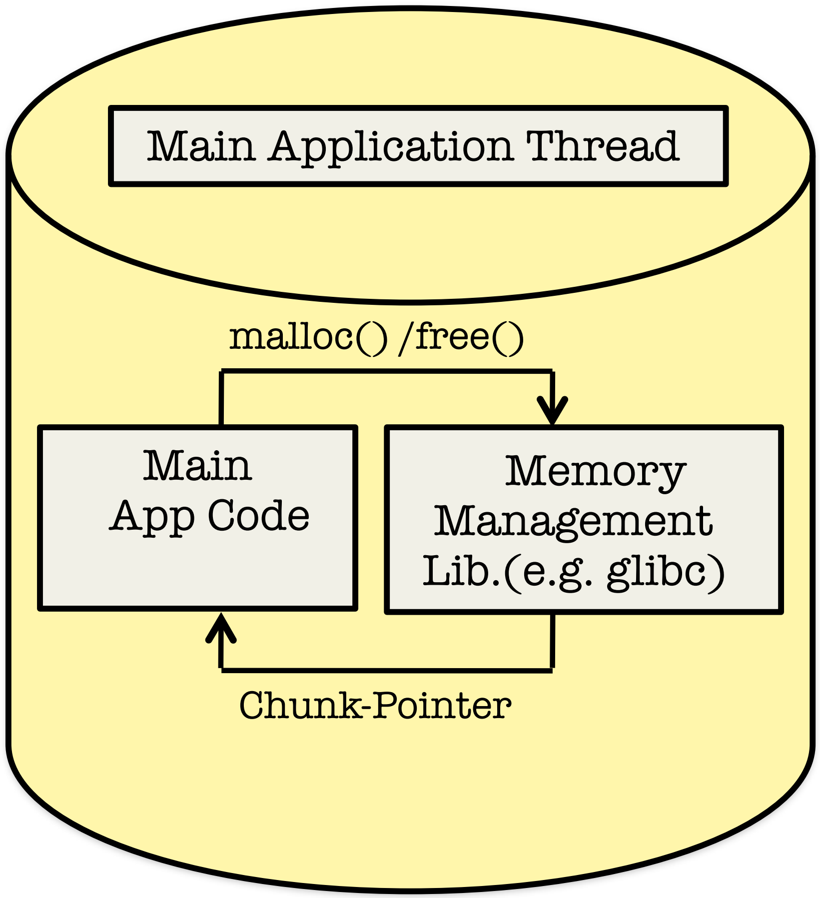
- ✓ Not exploiting implementation details of underlying memory allocator
- ✓ No hardware or compiler support
- ✓ No source code modification
- ✓ No application specific tuning

Outline

- ◦Motivation
- ◦Memory Management Thread : MMT Approach
- ◦MMT Design and Implementation
  - ◦Basic MMT Design
  - ◦Speculative Memory Management
  - ◦Bulk Memory Management
  - ◦Understanding the Interaction between Speculative and Bulk Memory Management
- ◦Evaluation
- ◦Conclusion and Future Work

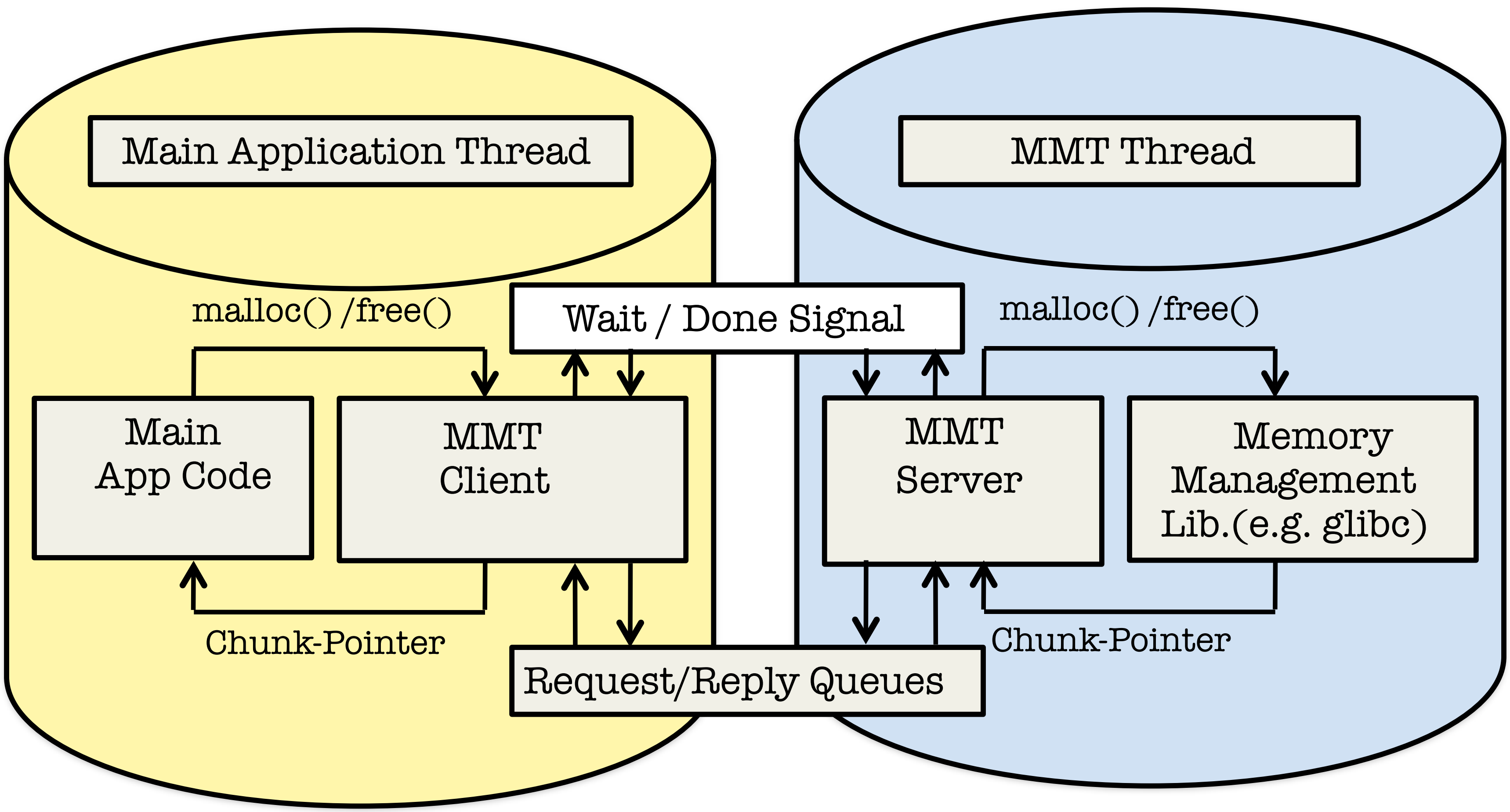
Basic MMT Design

Traditional Memory Management

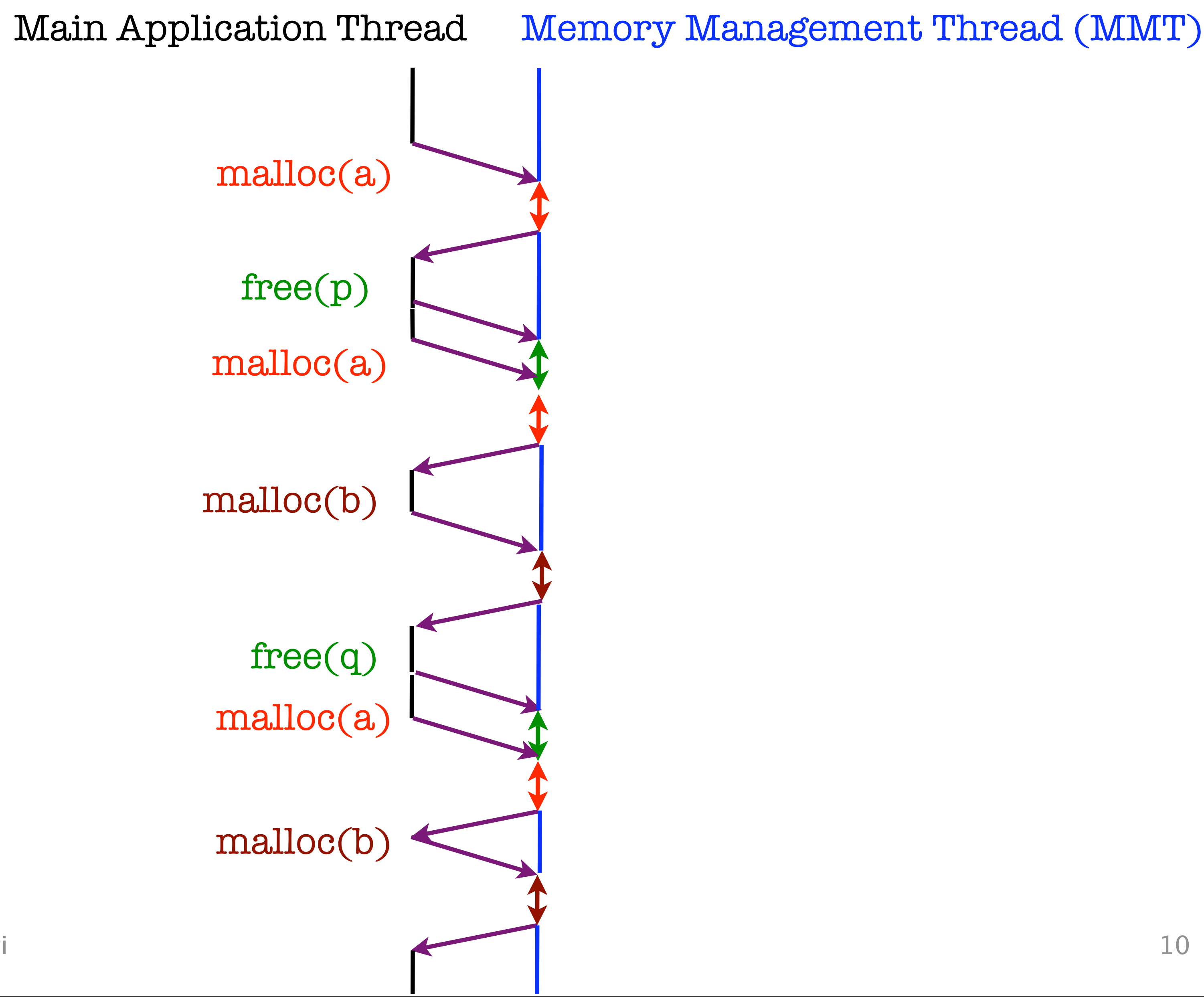


Basic MMT Design

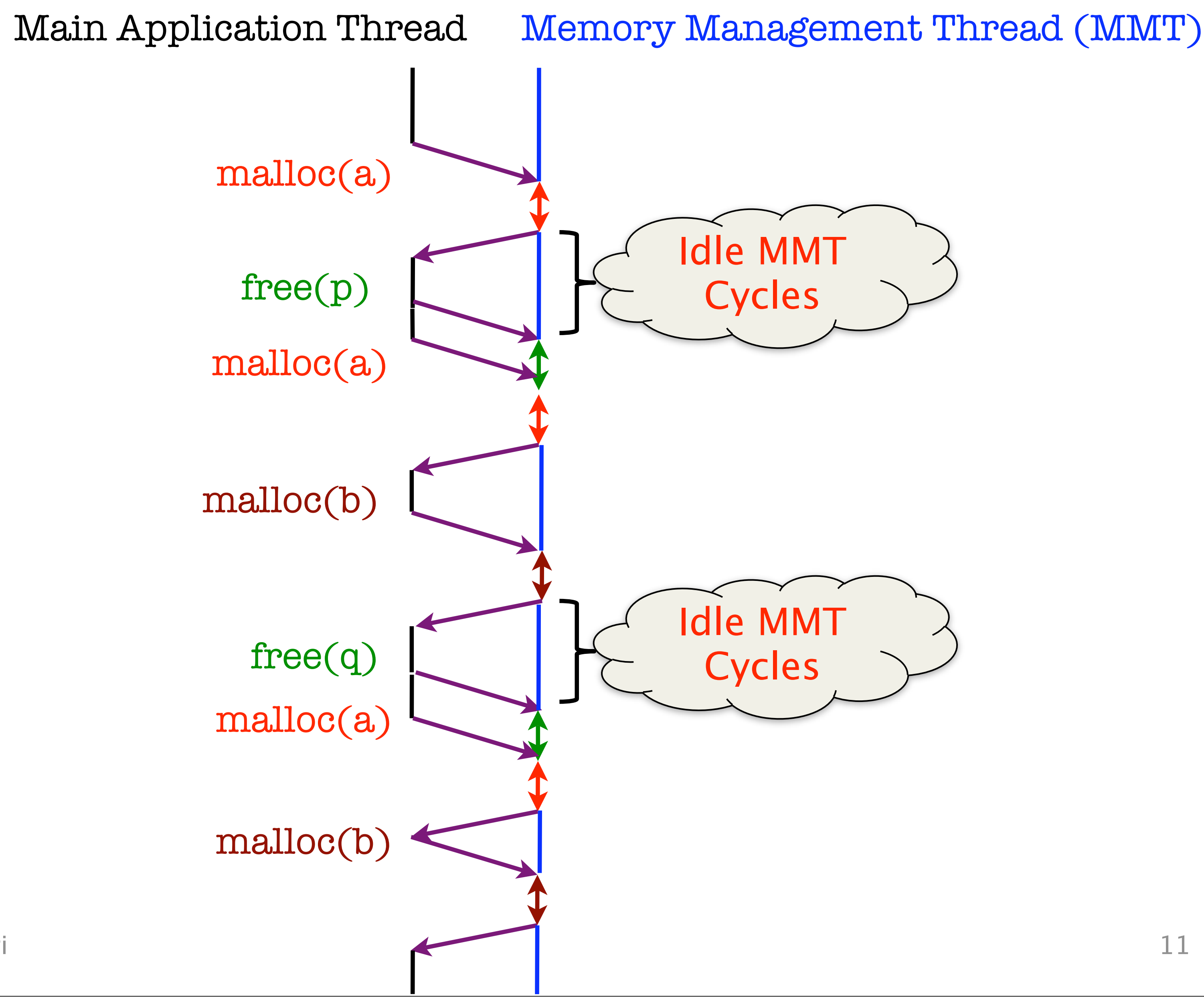
Memory Management Thread Approach



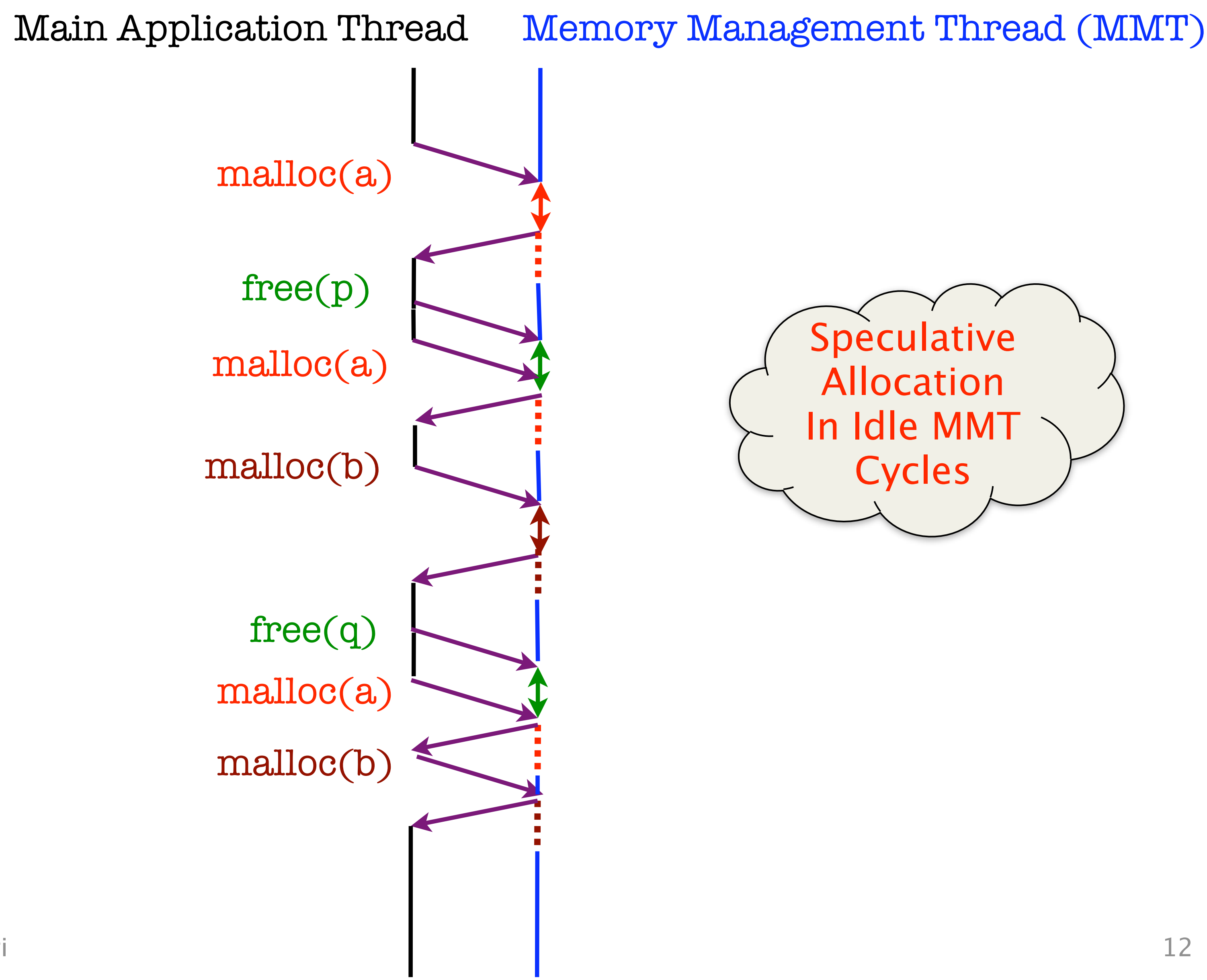
Speculative Memory Management



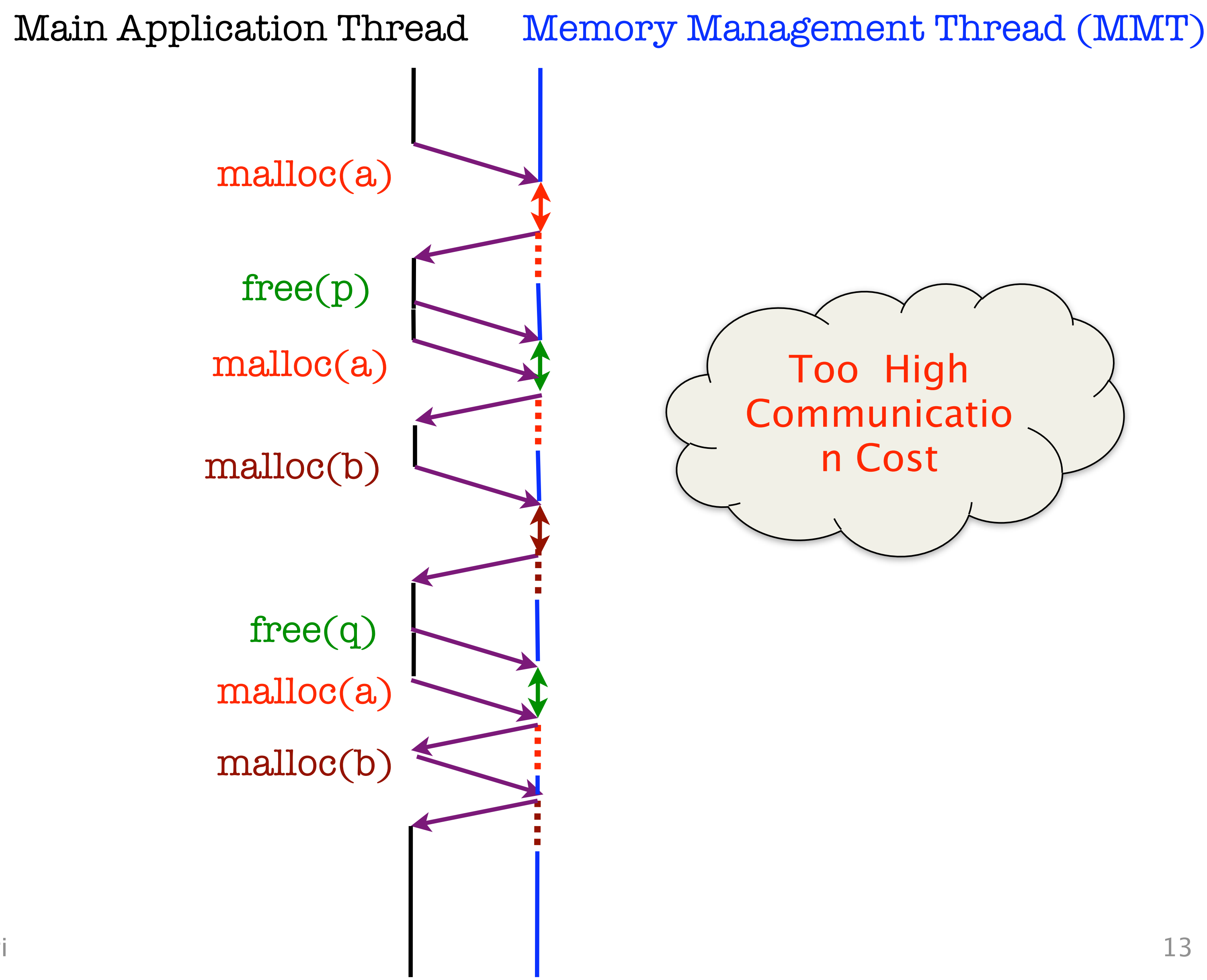
Speculative Memory Management



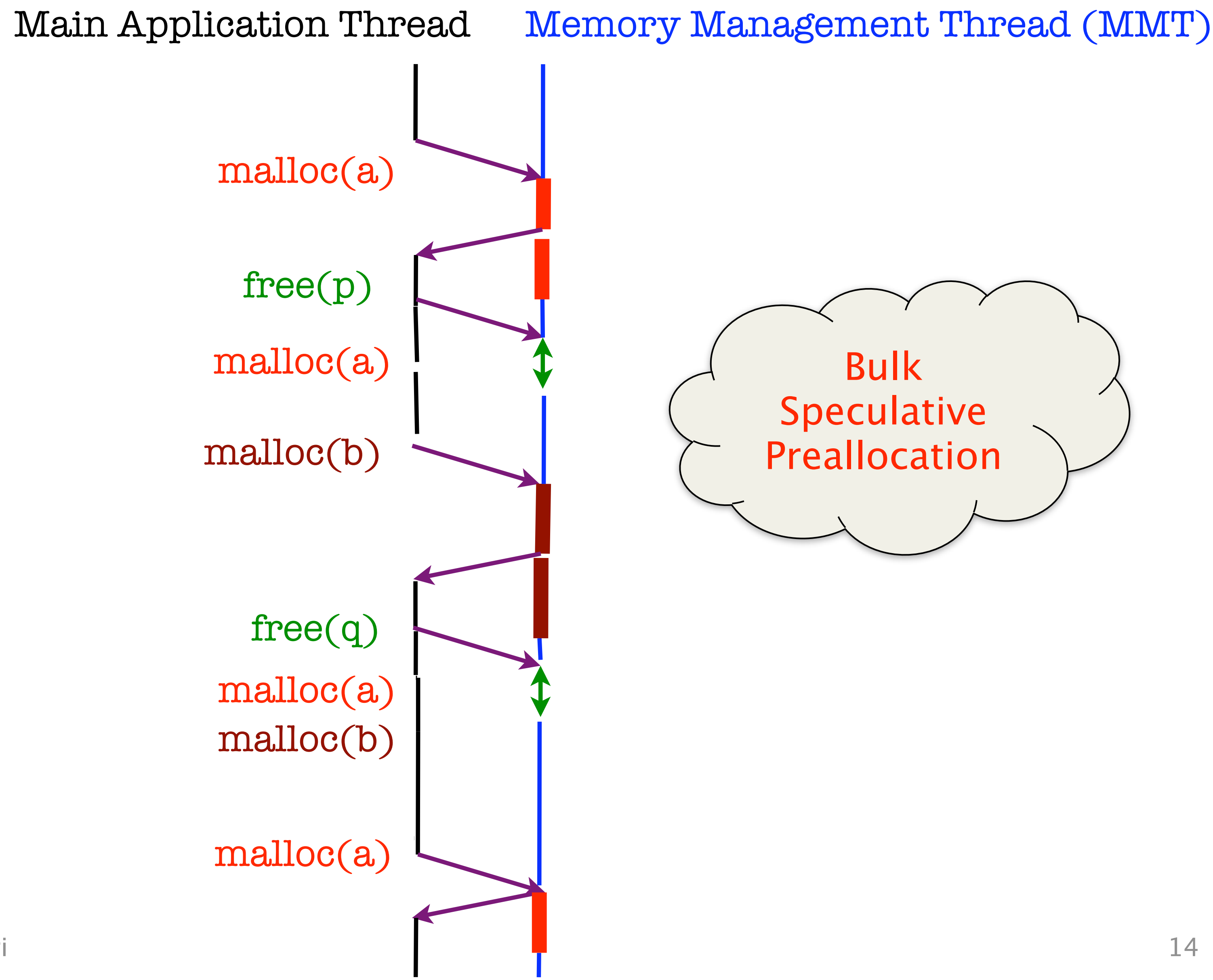
Speculative Memory Management



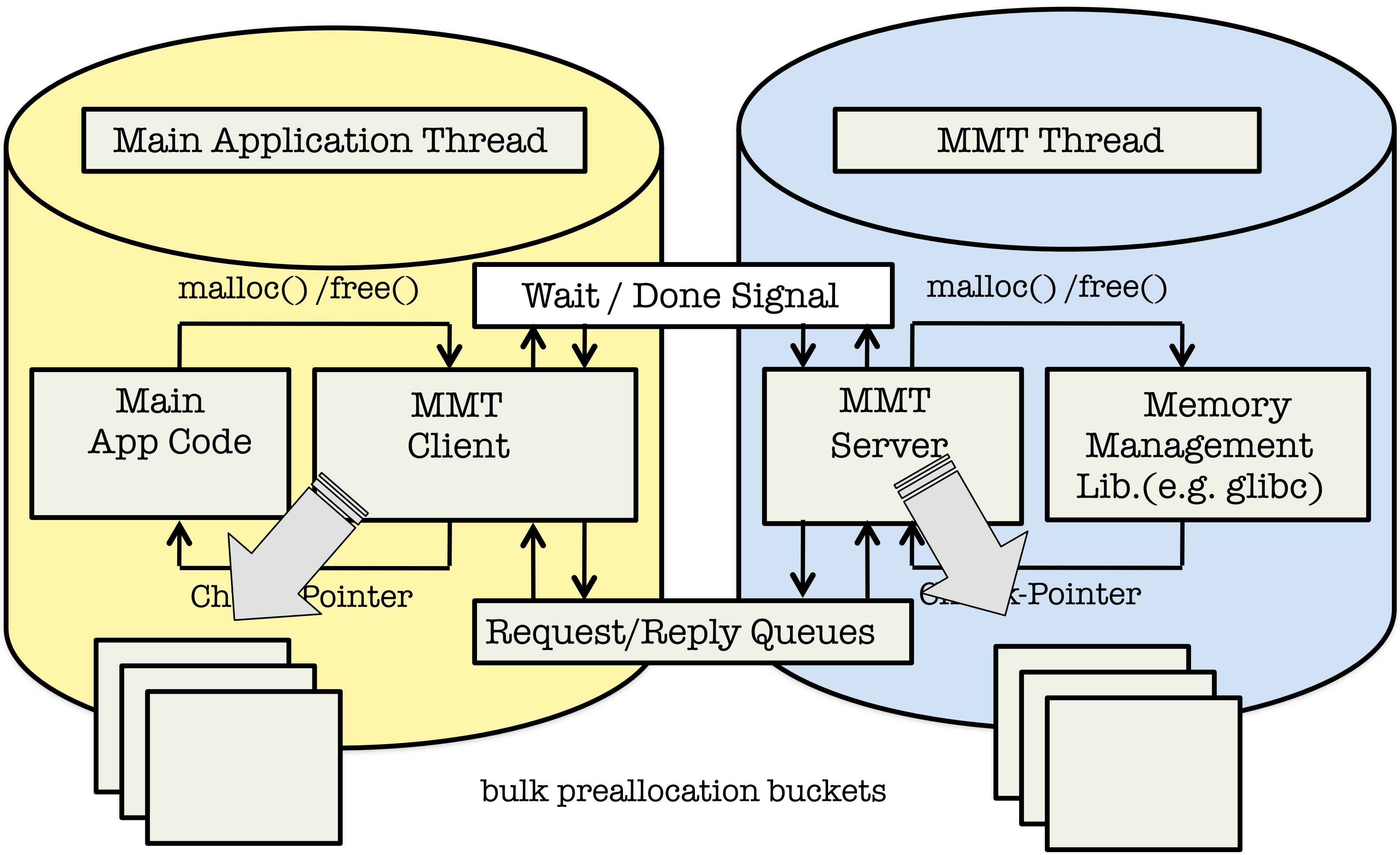
Bulk Memory Management



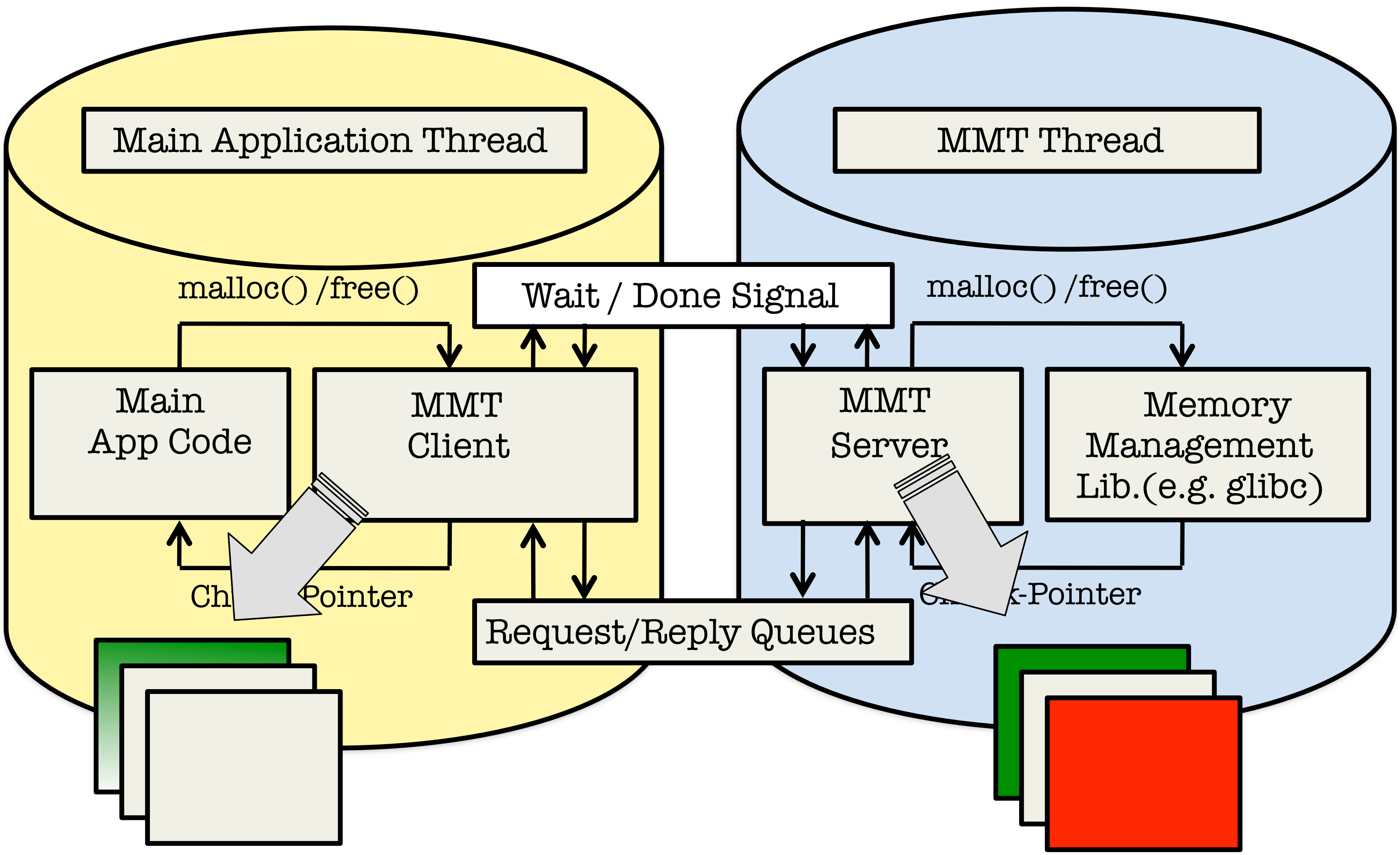
Bulk Memory Management



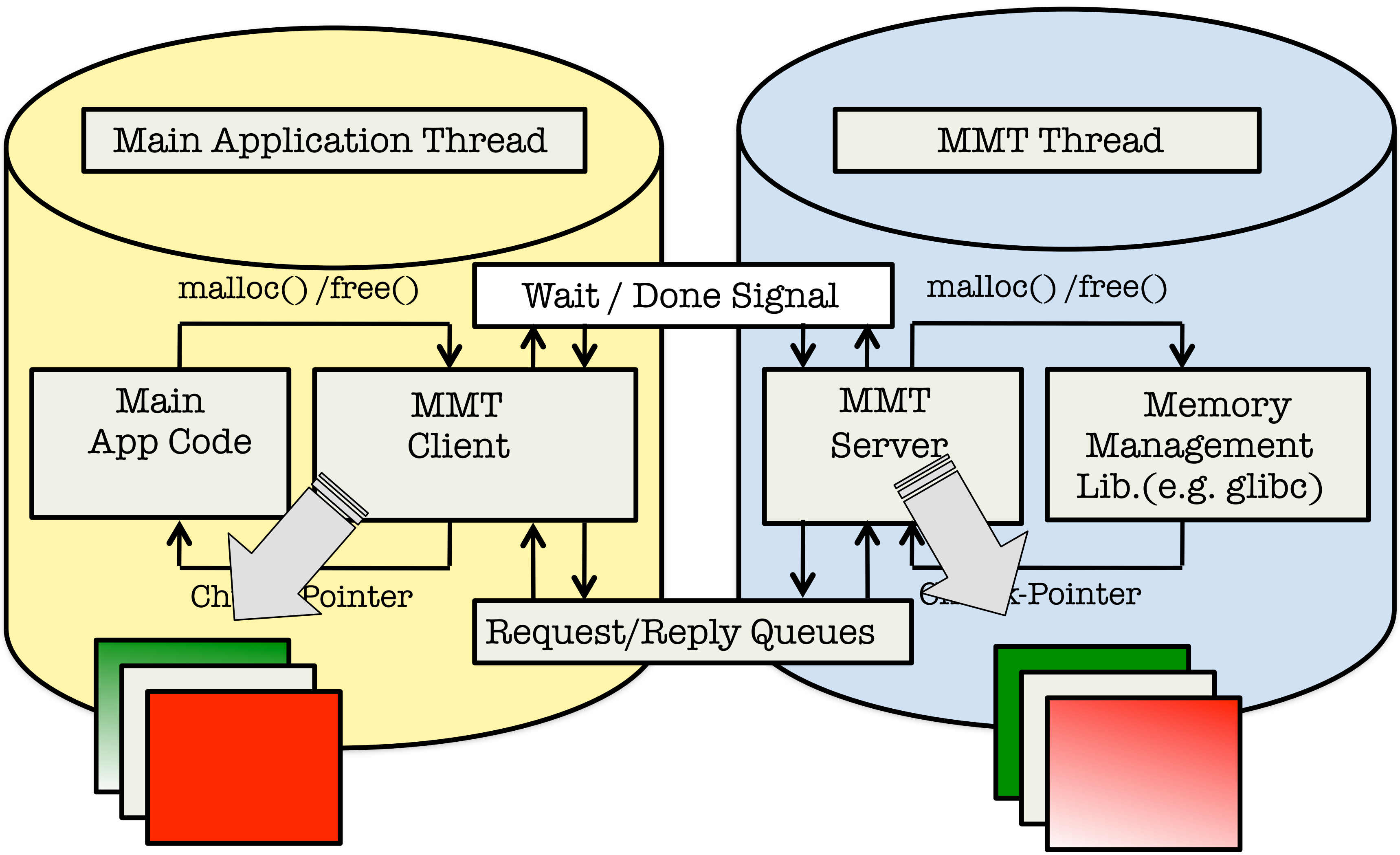
Bulk Memory Management



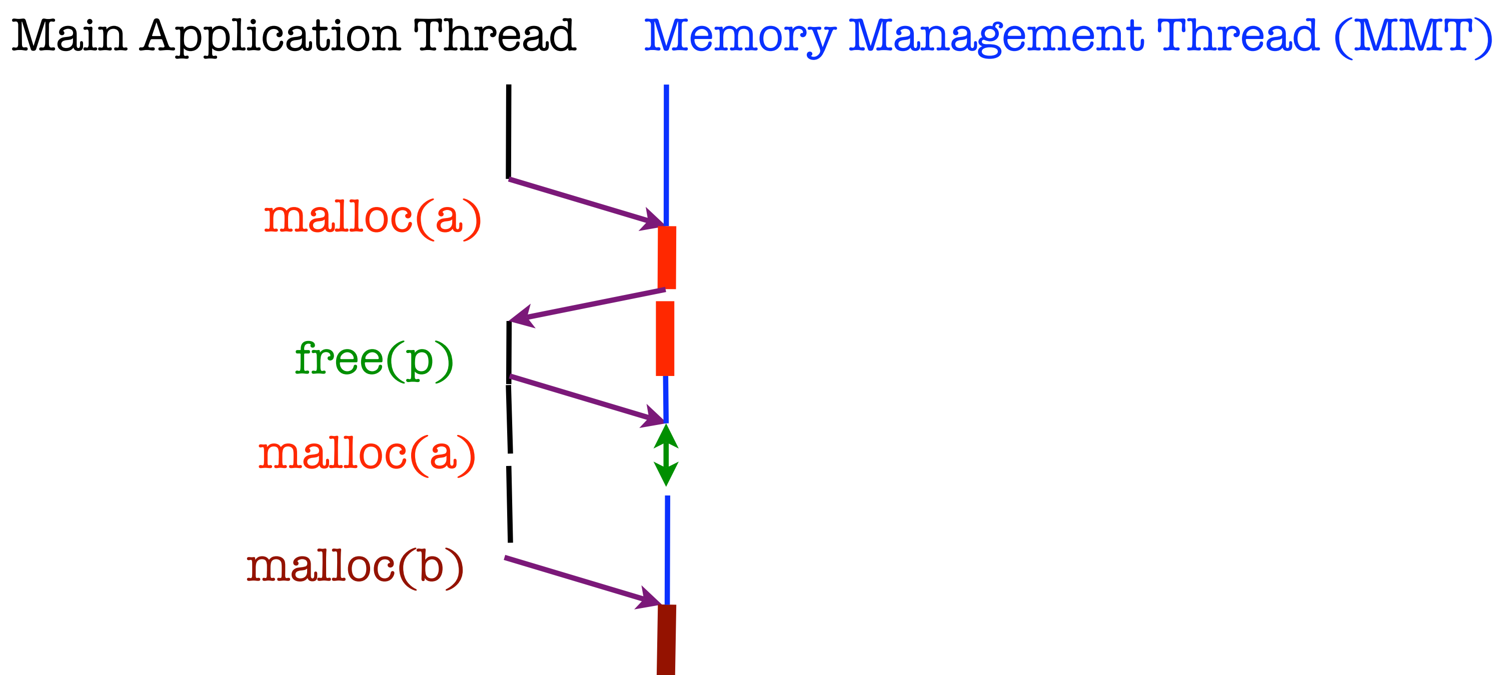
Bulk Memory Management



Bulk Memory Management

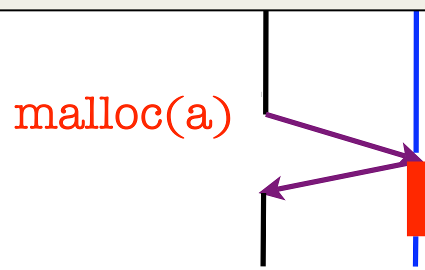


Bulk Memory Management

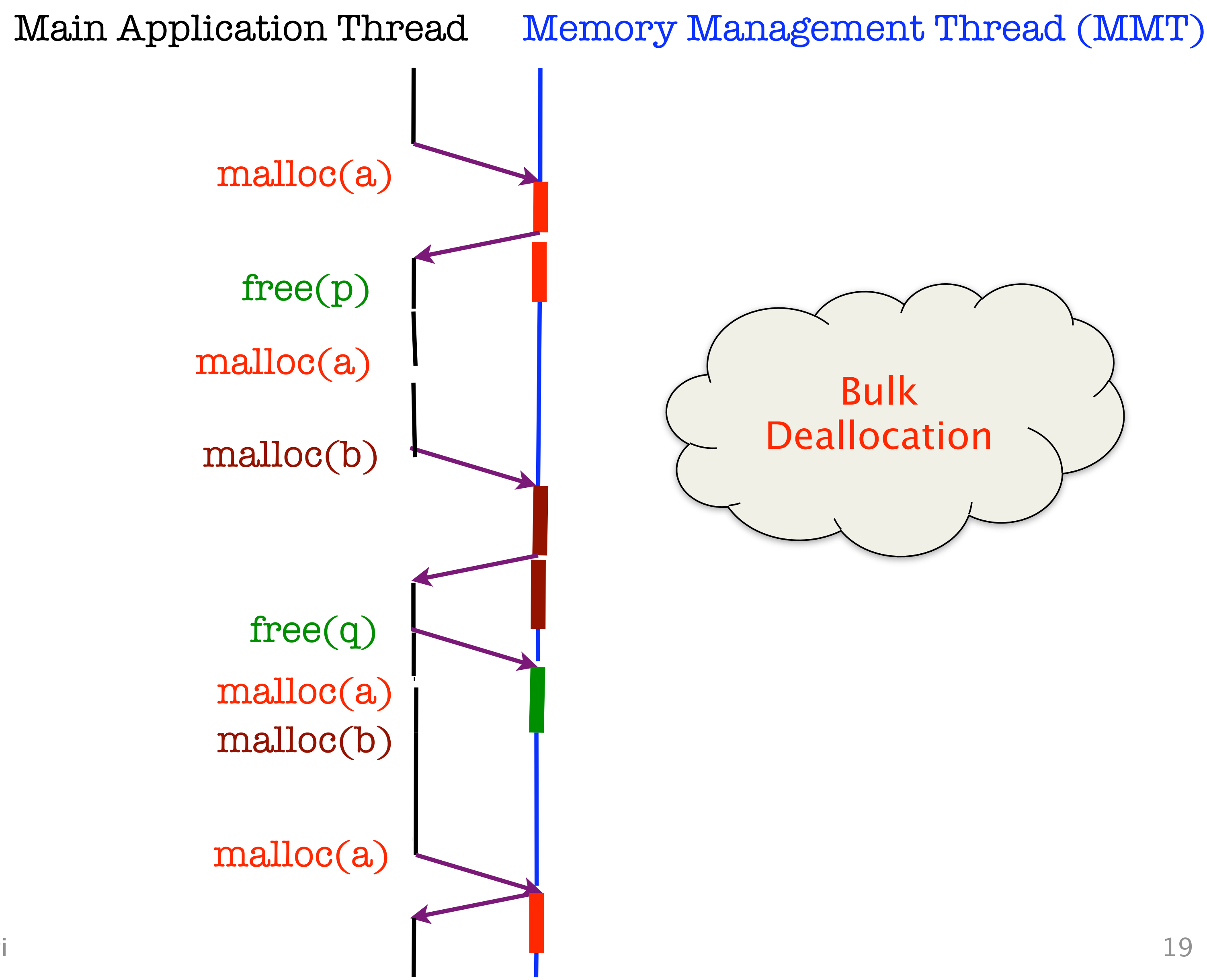


Bulk Speculative Preallocations

- Effect of Miss-speculative allocations and implications
- Starting point of bulk speculative allocations
- Speculative bulk preallocation bucket size



Bulk Memory Management



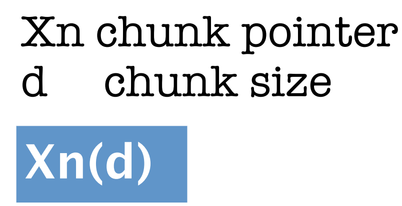
Understanding the Interaction

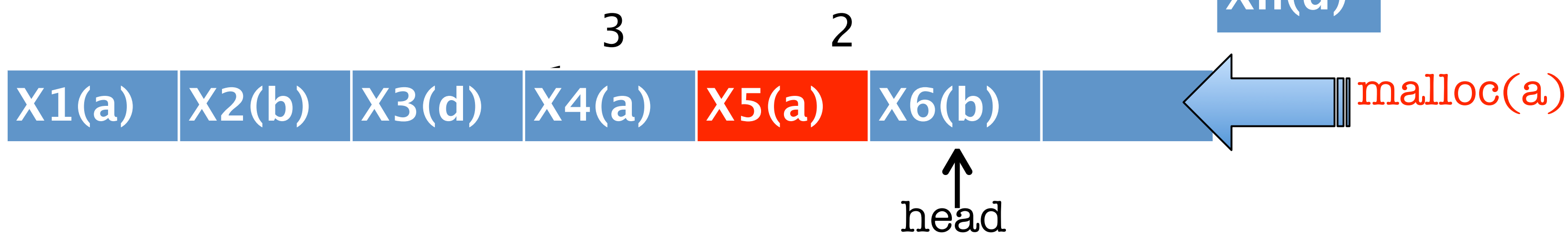
Applying Bulk Speculative Allocation in Conjunction with Bulk Deallocation

- Waiting time for asynchronous allocation request
  - Bulk Deallocation and Bulk Preallocation Synergy
- Finding idle cycles for bulk preallocation and bulk deallocation
  - Bulk preallocation and deallocation bucket sizes
- Program Cache Reference Locality
  - Preallocation versus Prefetching
  - Preallocation versus bulk delayed deallocation
  - Bulk Deallocation versus chunk reuse possibility

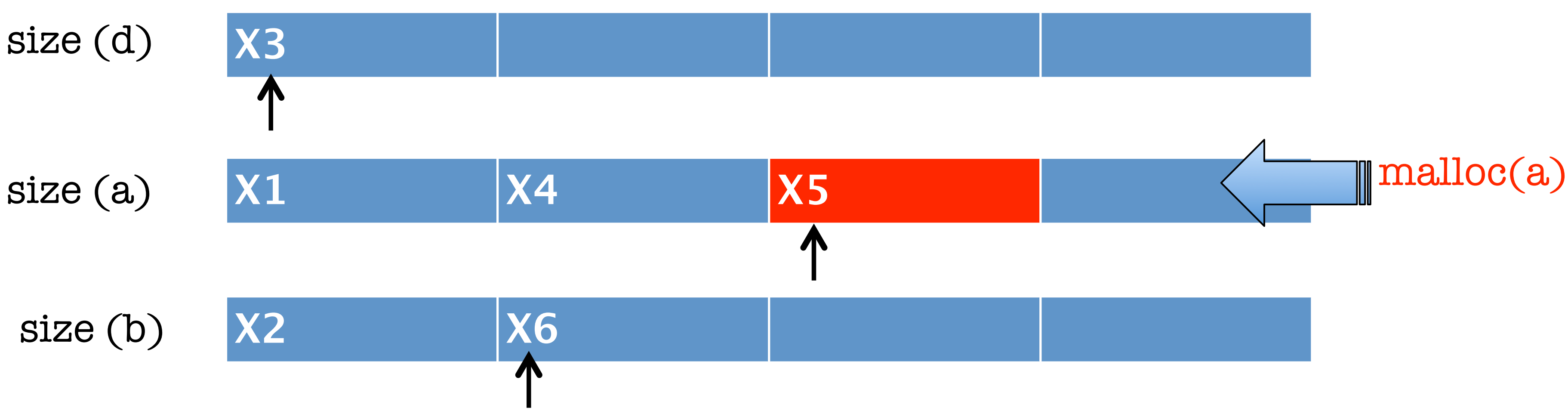
Improving Cache Reference Locality

Satisfying new malloc() request by first checking shared bulk deallocation queue (until depth of 3), before looking into preallocated chunks.





Satisfying new malloc() request by first checking the head of size specific bulk deallocation queue, before looking into preallocated chunks.



Outline

- Motivation
- Memory Management Thread : MMT Approach
- MMT Design and Implementation
- Evaluation
  - Performance : MMT Approach
  - Understanding Decoupling Effects
- Conclusion and Future Work

Evaluation

Benchmark	Language	Input
cfrac	C	A 35 digit number
deltaBlue	C++	1000000
gawk	C	large.awk
mst	C	8192 nodes
Perimeter	C	13 levels
Roboop	C++	bench
Treeadd	C	27 levels

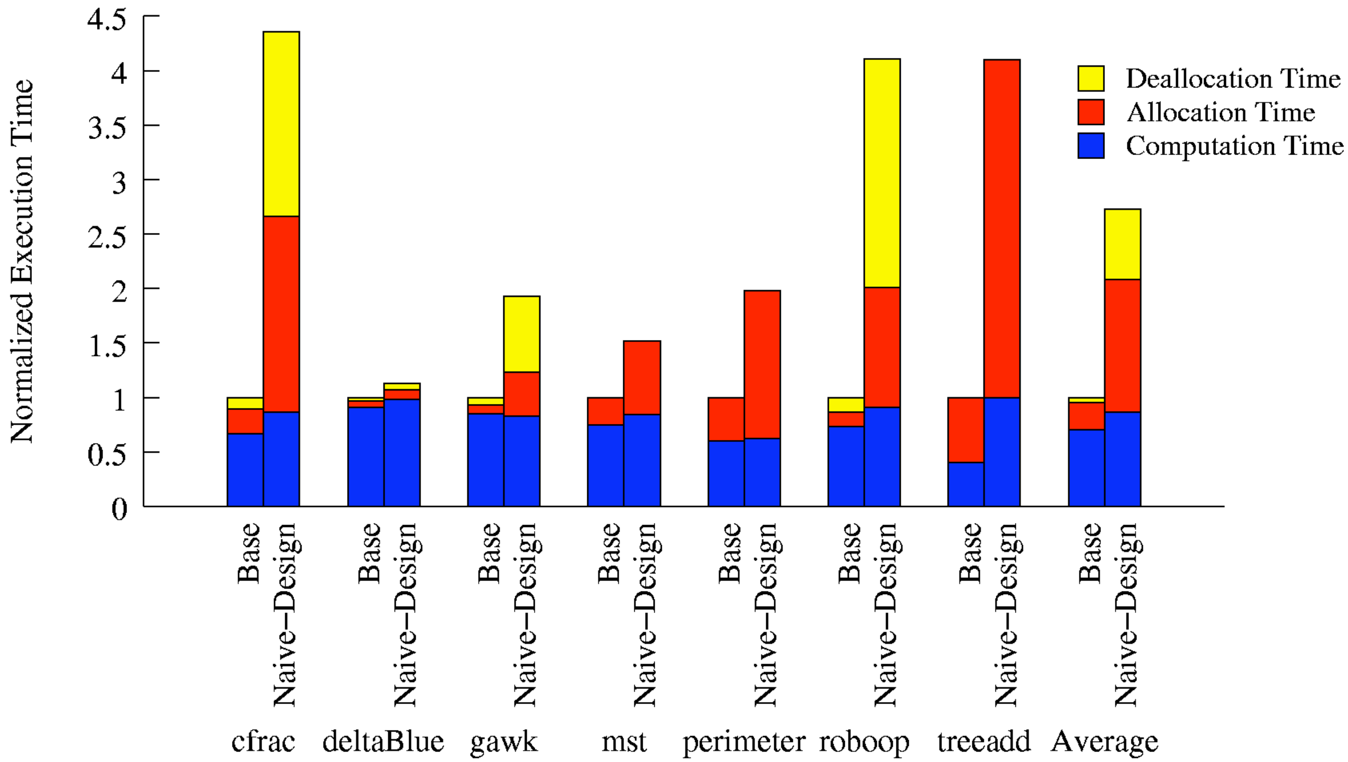
Machine: 2.4GHz Intel Core 2 Quad machine,
Linux Version 2.6.18
Compiler : gcc -O3 optimization level

Profiler: Oprofile for hardware performance counters

Offloaded allocator: Doug Lea's allocator

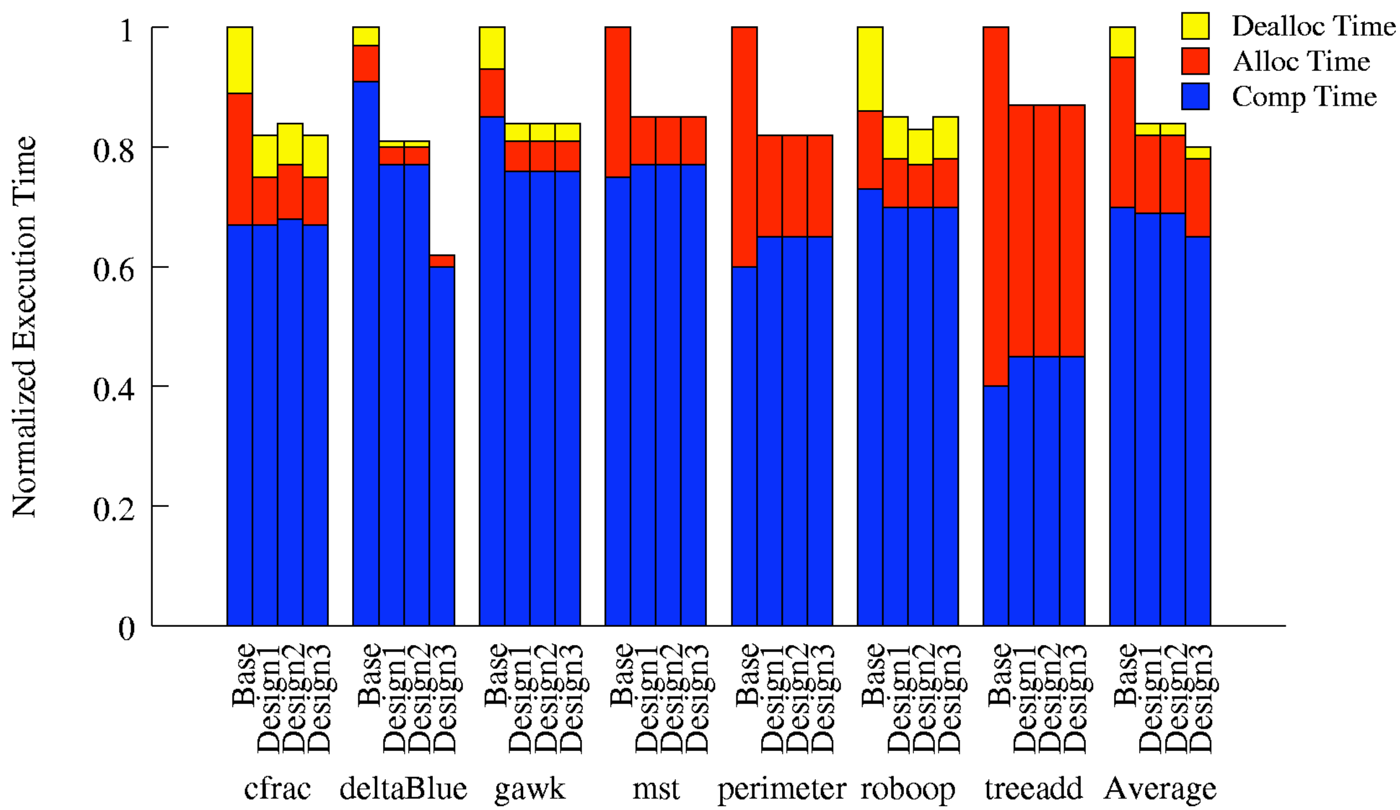
Basic MMT Performance

Synchronous allocation and deallocation using MMT Approach



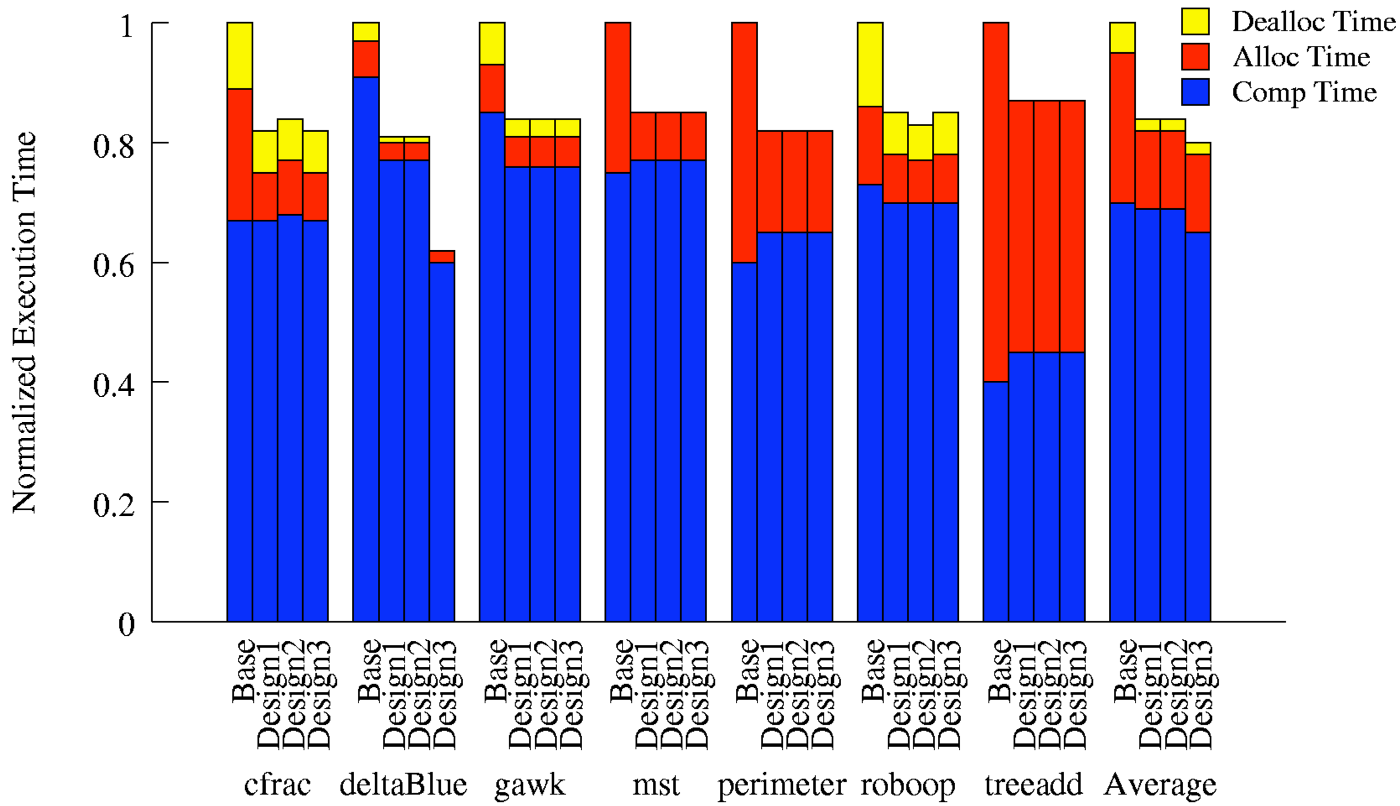
Average Slow-down 2.73x, high offloading latency and synchronization cost

MMT Performance



Design1	Bulk Preallocation and deallocation at MMT
Design2	Design1 + immediate recycling using shared bulk deallocation queue
Design3	Design1 + immediate recycling using size specific bulk deallocation queue

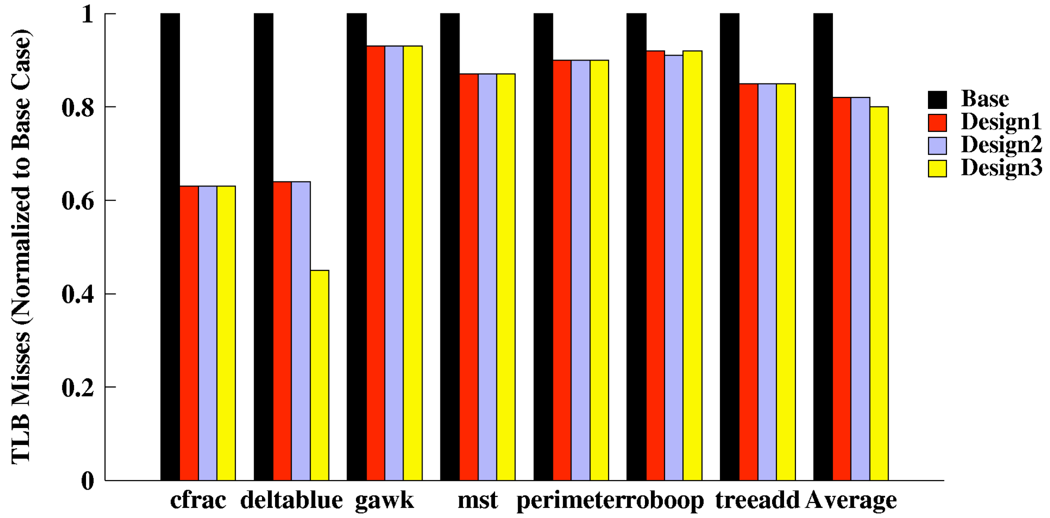
MMT Performance

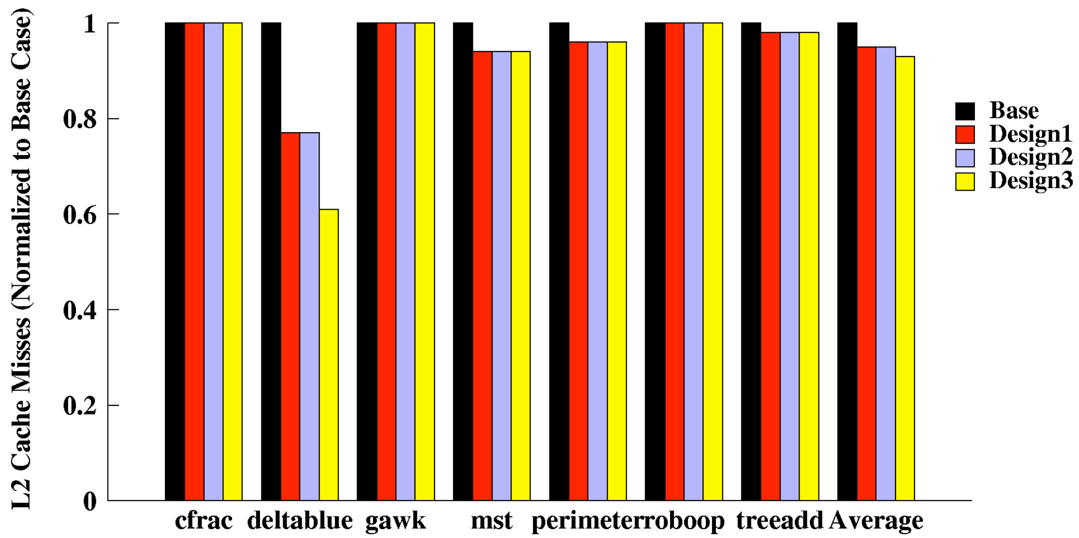


Design 1: Average speed up ratio 1.19x, Best Case 1.25x
Allocation and Deallocation costs reduced by a factor of two

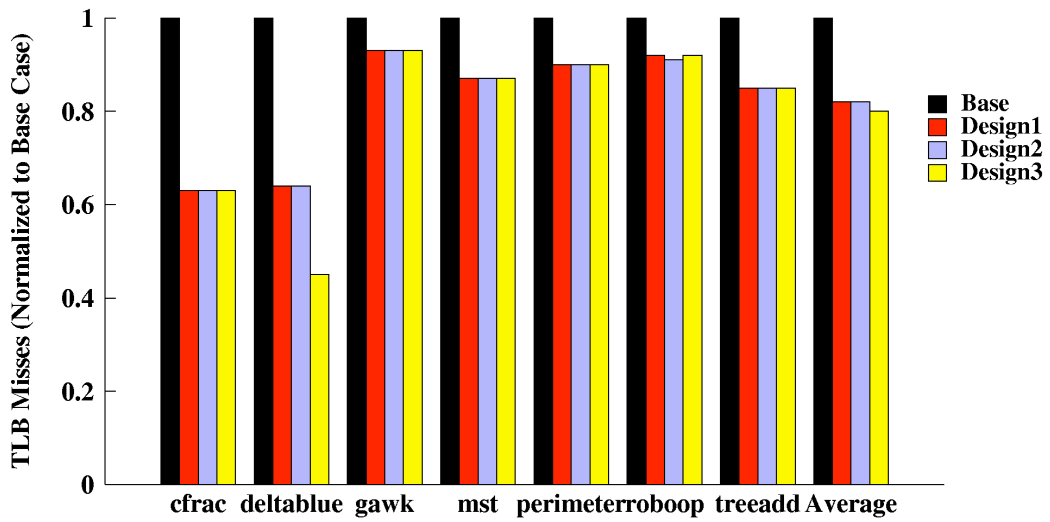
Design 2 and 3: Best Case speed up ratio 1.60x
Synergy between bulk preallocation and bulk deallocation : Cache Locality

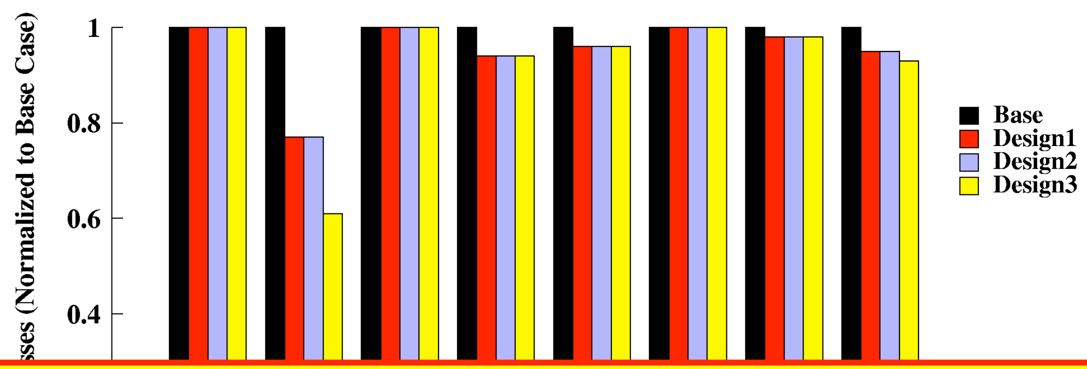
Understanding the Decoupling Effect





Understanding the Decoupling Effect





Decoupling Effect: difference in code and cache behavior of regular computation and memory management routines, reduction in TLB miss rate, L2 cache miss rate and branch mispredictions. Less resource interference.

Outline

- Motivation
- Memory Management Thread : MMT Approach
- MMT Design and Implementation
- Evaluation
- Conclusion and Future Work

Conclusion and Future Work

- Novel MMT Approach for Efficient Dynamic Memory Management for Sequential Application exploiting Multi Core Architecture Parallelism
  - Agnostic to underlying allocator's design and implementation
  - No hardware, compiler support
  - No application specific tuning or source code modification
- Exploiting Parallelism and reducing communication cost using
  - Bulk Preallocation and Bulk Deallocation
- Shows how to exploit fine grain function parallelism and off-load non-critical meta data computation to a dedicated thread
- Extending it to other allocators and Using MMT for high overhead tasks: memory related security checks, profiling, tracing, debugging etc.

Questions

✍

Devesh Tiwari
devesh.dtiwari@ncsu.edu
ARPERS Research Group
Department of Electrical and Computer Engineering
North Carolina State University
http://www.ece.ncsu.edu/arpers