Cross-Layer Memory Management to Reduce DRAM Power Consumption

Michael Jantz
Assistant Professor
University of Tennessee, Knoxville
Introduction

• Assistant Professor at UT since August 2014
• Before UT
  – PhD in Computer Science at KU (July 2014)
  – Intern at Intel Corporation (2012 – 2013)
• Research interests:
  – Compilers (optimization, phase ordering)
  – Operating Systems (kernel instrumentation, memory and power management)
  – Runtime Systems (dynamic compilation, object mgmt.)
• Courses taught:
  – Compilers (COSC 461), Discrete Structures (COSC 311)
Outline

• Compiler Optimization Phase Ordering
• Dynamic Compilation
• Cross-Layer Memory Management
  – Motivation
  – Design
  – Experimental Evaluation
• Future Directions
• Conclusions
Compiler Optimization
Phase Ordering
Phase Ordering

• Compiler optimizations operate in phases
  – Phases interact with each other
  – Phase ordering: different phase orderings produce different quality code

• Problem: finding the best ordering for each function or program takes a very long time
  – Iterative search is the most common technique
Exploiting Phase Interactions

• Our approach: identify and exploit phase interactions during search

• Major contributions:
  – Reduce exhaustive phase ordering search time
  – Increase applicability and effectiveness of individual optimization phases
  – Improve phase ordering heuristics

• Publications: LCTES ‘10 [1], CASES ‘10, [2]
Dynamic Compilation

Java

C#

.NET
Tradeoffs in Dynamic Compilation

• Managed language applications (e.g. Java)
  – Distributed as machine-independent codes
  – Require compilation at runtime
• Dynamic compilation policies involve *tradeoffs*
  – Can potentially slow down overall performance
  – Must consider several factors when setting policy:
    • Compiling speed and quality of compiled code
    • Execution frequency of individual methods
    • Availability of compilation resources
Dynamic Compilation Strategies

• Conducted multiple studies on how, when, and if to compile program methods

• Employ industrial-grade Java VM (HotSpot)

• Major studies:
  – Performance potential of phase selection in dynamic compilers (VEE '13-A [5])
  – Dynamic compilation strategy on modern machines (TACO, Dec. '13 [6])
Cross-Layer Memory Management
A Collaborative Approach to Memory Management

• Memory has become a significant player in power and performance
• Memory power management is challenging
• Propose a collaborative approach between applications, operating system, and hardware:
  – Applications – communicate memory usage intent to OS
  – OS – re-architect memory mgmt. to interpret application intent and manage memory over hardware units
  – Hardware – communicate hardware layout to the OS to guide memory management decisions
A Collaborative Approach to Memory Management

- Implemented framework by re-architecting a recent Linux kernel
- Experimental evaluation
- Publications: VEE ’13-B [7], Linux Symposium ‘14 [8], manuscript in submission [9]
• CPU and Memory are most significant players for power and performance
  – In servers, memory power == 40% of total power [10]

• Applications can direct CPU usage
  – threads may be affinitized to individual cores or migrated b/w cores
  – prioritize threads for task deadlines (with nice)
  – individual cores may be turned off when unused

• Surprisingly, much of this flexibility does not exist for controlling memory
Example Scenario

• System with database workload with 512GB DRAM
  – All memory in use, but only 2% of pages are accessed frequently
  – CPU utilization is low

• How to reduce power consumption?
Challenges in Managing Memory Power

• Memory refs. have temporal and spatial variation

• At least two levels of virtualization:
  – Virtual memory abstracts away application-level info
  – Physical memory viewed as single, contiguous array of storage

• No way for agents to cooperate with the OS and with each other

• Lack of a tuning methodology
A Collaborative Approach

• Our approach: enable applications to guide mem. mgmt.

• Requires collaboration between the application, OS, and hardware:
  – Interface for communicating application intent to OS
  – Ability to keep track of which memory modules host which physical pages during memory mgmt.

• To achieve this, we propose the following abstractions:
  – Colors
  – Trays
Communicating Application Intent with Colors

- Color = a hint for how pages will be used
  - Colors applied to sets of virtual pages that are alike
  - Attributes associated with each color

- Attributes express different types of distinctions:
  - Hot and cold pages (frequency of access)
  - Pages belonging to data structures with different usage patterns

- Allow applications to remain agnostic to lower level details of mem. mgmt.
Power-Manageable Units Represented as Trays

- **Tray** = software structure containing sets of pages that constitute a power-manageable unit

- Requires mapping from physical addresses to power-manageable units
  - ACPI 5.0 defines memory power state table (MPST) to expose this mapping

- Re-architect a recent Linux Kernel to perform memory management over trays
Application colors pages to indicate a range of pages will be hot.

Physical memory allocation and recycling.

OS looks up attribute associated with the virtual pages’ color.

Memory topology represented in the OS using trays.
Experimental Evaluation

- Emulating NUMA API’s
- Memory prioritization for applications
- Reducing DRAM power consumption
  - Power-saving potential of containerized memory management
  - Localized allocation and recycling
  - Exploiting generational garbage collection
Automatic Cross-Layer Memory Management

• Limitations of application guidance:
  – Little understanding of which colors or coloring hints will be most useful for existing workloads
  – All colors and hints must be manually inserted

• Our approach: integrate with profiling and analysis to automatically provide power / bandwidth mgmt.
  – Implemented using the HotSpot JVM
  – Instrumentation and analysis to build memory profile
  – Partition live objects into separately colored regions
Employ the default HotSpot config. for server-class applications
Divide survivor / tenured spaces into spaces for hot / cold objects
• Color spaces on creation or resize
• Partition allocation sites and objects into hot / cold sets
Potential of JVM Framework

• Our goal: evaluate power-saving potential when hot / cold objects are known statically
• MemBench: Java benchmark that uses different object types for hot / cold memory
• “HotObject” and “ColdObject”  
  – Contain memory resources (array of integers)  
  – Implement different functions for accessing mem.
Experimental Platform

• Hardware
  – Single node of 2-socket server machine
  – Processor: Intel Xeon E5-2620 (12 threads @ 2.1GHz)
  – Memory: 32GB DDR3 memory (four DIMM’s, each connected to its own channel)

• Operating System
  – CentOS 6.5 with Linux 2.6.32

• HotSpot JVM
  – v. 1.6.0_24, 64-bit
  – Default configuration for server-class applications
The MemBench Benchmark

• Object allocation
  – Creates “HotObject” and “ColdObject” objects in a large in-memory array
  – # of hots < # of colds (~15% of all objects)
  – Object array occupies most (~90%) system mem.

• Multi-threaded object access
  – Object array divided into 12 separate parts, each passed to its own thread
  – Iterate over object array, only accessing hot objects

• Optional *delay* parameter
MemBench Configurations

• Three configurations
  – Default
  – Tray-based kernel (custom kernel, default HotSpot)
  – Hot/cold organize (custom kernel, custom HotSpot)
• Delay varied from "no delay" to 1000ns
  – With no delay, 85ns between memory accesses
MemBench Performance

- Tray-based kernel has about same performance as default
- Hot/cold organize exhibits poor performance with low delay
• Default and tray-based kernel produce high memory bandwidth when delay is low
• Placement of hot objects across multiple channels enables higher bandwidth
• Hot/cold organize - hot objects co-located on single channel
• Increased delays reduces bandwidth reqs. of the workload
MemBench Energy

- Hot/cold organize consumes much less power with low delay
- Even when BW reqs. are reduced, hot/cold organize consumes less power than other configurations
MemBench Energy

- Significant energy savings potential with custom JVM
- Max. DRAM energy savings of ~39%, max. CPU+DRAM energy savings of ~15%
Results Summary

• Object partitioning strategies
  – Offline approach partitions allocation points
  – Online approach uses sampling to predict object access patterns

• Evaluate with standard sets of benchmarks
  – DaCapo, SciMark

• Achieve 10% average DRAM energy savings, 2.8% CPU+DRAM reduction

• Performance overhead
  – 2.2% for offline, 5% for online
Current and Future Projects in Cross-Layer Memory Management

• Immediate future work: address performance losses of our current approach
  – Improve the online sampling
  – Automatic bandwidth management

• Applications for heterogeneous memory architectures

• Exploit data object placement within each page to improve efficiency
Conclusions

• Research focuses on software systems
  – Compilers, operating systems, and runtime systems

• Cross-layer memory management
  – Achieving power/performance efficiency in memory requires a cross-layer approach
  – First framework to use usage patterns of application objects to steer low-level memory mgmt.
  – Approach shows promise for reducing DRAM energy
  – Opens several avenues for future research in collaborative memory management
Questions?
References

1. Prasad Kulkarni, Michael Jantz, and David Whalley. Improving Both the Performance Benefits and Speed of Optimization Phase Sequence Searches In the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '10), April 2010


References


