The Impact of Increasing Memory System Diversity on Applications

Gwen Voskuilen

Arun Rodrigues, Mike Frank, Si Hammond

4/25/2017
Diversity in memory hierarchy

- New memory technologies – stacked DRAM, non-volatile
  - Advantages: higher bandwidth, persistent, etc.
  - Disadvantages: expensive, higher latencies

- Solution: multi-level memory (MLM)
  - Mix memories to expose advantages, hide disadvantages
  - Really hard in practice
    - Must carefully locate data based on data characteristics
    - Which data goes into which memory? Who decides?

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Automatic</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>Hardware decides</td>
<td>OS/runtime decides</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application decides</td>
</tr>
</tbody>
</table>
Managing MLM

- Trinity KNL: Small stacked DRAM + large DDR DRAM
  - Application data footprint >> stacked DRAM capacity

- Management requires *identifying and selectively allocating* data needing bandwidth into stacked DRAM

- Study 1: Software management policies
  - Ranging from simple to complex
  - Developed analysis tool, *MemSieve*, to evaluate memory behavior

- Study 2: Hardware caching
  - Insertion/eviction policies
Outline

- Methodology

- Software / manually managed MLM (malloc() based)

- Hardware / automatic managed MLM

- Conclusions
Simulation Methodology

- **MiniApps:** HPCG, PENNANT, SNAP
  - Memory footprints of 1-8GB → sampling required
- **Simulated on two architectures using SST**
  - Lightweight: 72 small cores, mesh, private L1, semi-private L2
  - Heavyweight: 8 big cores, ring, private L1 + L2, shared L3
  - Both: DDR and HMC memories
Outline

- Methodology
  - Software / manually managed MLM
  - Hardware / automatic managed MLM
- Conclusions
Software approaches

Software management

OS / Runtime
- Static
  - Greedily insert pages into HMC
- Dynamic (future work)
  - Greedily insert mallocs into HMC

Programmer
- Static
  - Direct "best" mallocs to HMC
- Dynamic
  - Migrate to put current "best" to HMC

Increased performance?
Tradeoff

- **Automatic (OS)**
  - Easier for programmer
  - Able to capture allocations not under programmer control
    - Library, pre-program start, etc.
  - Page-table complexity; potentially expensive re-mapping
  - No program knowledge → worse performance?
    - Could use programmer hints or runtime profiling but more work

- **Manual (programmer)**
  - More work for programmer, pervasive (?) changes
  - Not able to handle all allocations
  - Possible conflicts between application and library allocations
    - What if libraries decide to manage allocation for internal structures too?
  - Knowledge of program behavior → better performance?
Analysis tool: MemSieve

- Captures an application’s memory accesses and correlates to the application’s memory allocations
  - Filters out cache hits
  - Without simulating full memory hierarchy → 2.5X + faster

- Key measurement: `malloc density`
  - # accesses / size

- Hypothesis: dense mallocs should be put in HMC
  - Assuming similar latencies between HMC and DDR
# Malloc analysis

<table>
<thead>
<tr>
<th></th>
<th>Pennant</th>
<th>HPCG</th>
<th>Snap *</th>
<th>MiniPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malloc count</td>
<td>8B</td>
<td>23M</td>
<td>1B</td>
<td>438K</td>
</tr>
<tr>
<td>Malloc size</td>
<td>32.1 TB</td>
<td>7.43 GB</td>
<td>30GB</td>
<td>7.9GB</td>
</tr>
<tr>
<td>Distinct traces</td>
<td>248</td>
<td>612</td>
<td>323</td>
<td>39043</td>
</tr>
<tr>
<td>Accessed traces</td>
<td>140</td>
<td>146</td>
<td>90</td>
<td>10794</td>
</tr>
<tr>
<td>Size of accessed traces as % total</td>
<td>89.7%</td>
<td>99.987%</td>
<td>89.6%</td>
<td>84%</td>
</tr>
</tbody>
</table>

*Iterations from beginning & middle only

- Many mallocs but few distinct malloc call traces (locations)
- Reasons mallocs are not accessed
  - Same address malloc’d repeatedly $\rightarrow$ cache-resident
  - Malloc was not accessed in profiled section of application
Ideal malloc behavior

- **Good**: A few, small, very dense mallocs
- **Bad**: Many, equally dense mallocs; densest are big

Big density variation: less work to manage

Lots of accesses in a very small region
Malloc density

PENNANT

Density (accesses/byte)

Accesses

Malloc call sites, from most to least dense

HPCG

Density (accesses/byte)

Accesses

Malloc call sites, from most to least dense

Cumulative size (TB)

Accesses

Malloc call sites, from most to least dense

Cumulative size (GB)

Accesses

Malloc call sites, from most to least dense

HPCG

Size (MB)

Accesses

Malloc call sites, from most to least dense
HMC Potential Performance

- **Max: 8X**
- Trends do not change with data set size (1-8GB)

**Heavyweight Architecture: Performance with all HMC**

**Lightweight Architecture: Performance with all HMC**
- Large performance jump from 25% to 50% HMC
- Dynamic migration necessary
Manual allocation: HPCG

- Again, large jump from 25% to 50% HMC
- Page-based & static perform similarly
- Dynamic not better
  - But granularity of migration is large
Manual allocation: SNAP

- **Greedy-page performs the best**
  - Two large mallocs in SNAP, each 42% of total
    - Medium/low density
  - Once they don’t fit, HMC size / malloc strategy doesn’t matter
- **Suggested code change**
  - Break up large mallocs to improve HMC utilization
Topology comparison

- Similar trend
Outline

- Methodology
- Software / manually managed MLM
- Hardware / automatic managed MLM
- Conclusions
Hardware management

- Hardware management of MLM at the page level
  - Cache pages in HMC, page still resides in DDR
  - Compared to block level: lower tracking overhead but higher add/remove overhead

- Focus was hardware caching
  - But, also possible to do caching via OS
  - Usually, less information (hits, misses, etc.)
Automatic Page-Level Swapping

- Addition policies
- Replacement policies

**Addition Policies**
- addT: Simple Threshold
- addMFU: Most Frequently Used
- addRAND: 1:8192 chance
- addMRPU: More Recent Previous Use
- addMFRPU: More Frequent + More Recent Previous Use
- addSC: Deprioritize streams
- addSCF: as addSC + More Frequent

**Replacement Policies**
- FIFO: First-in, First-out
- LRU: Least Recently Used
- LFU: Least Frequently Used
- LFU8: LFU w/ 8-bit counter
- BiLRU: BiModal LRU
- SCLRU: Deprioritize streams
Performance vs. Policy

Replacement policy: little variation

Addition policy: big variation

“What you put in matters more than what you take out”
Larger data sets

- Looked at highest performing addition policies
  - Variants of most-frequently used
  - Baseline: random
  - LRU replacement
Fine Tuning

1. Thresholds
   Pennant Threshold

2. Page size

3. Throttling
   Pennant Page size Effects
   snap-p0 Page size Effects

MLM Performance vs. Threshold (addT/LRU)

Swap Throttling
Conclusions

- Application behavior varies significantly
- Software management is feasible
  - Small to moderate number of dense call sites
  - Static allocation sufficient in some cases, dynamic necessary in others
- For hardware management, addition policy matters most
- For automatic & manual, profiling is instrumental
  - Application managed – helps identify high-bandwidth data
  - Automatically managed – helps identify places where application changes will improve performance
Exceptional service in the national interest