Pinpointing Data Locality Problems Using Data-centric Analysis

Xu Liu
XL10@rice.edu
Department of Computer Science
Rice University
Outline

• Introduction to data-centric analysis
• Implementation
• Experiments
• Conclusion
Code-centric and data-centric

- Code-centric analysis
  - Attribute metrics to statements
  - Identify problematic statements, loops or subroutines
- Data-centric analysis
  - Attribute metrics to data structures
  - Identify problematic data structures

![Diagram showing code-centric and data-centric analysis]
Data-centric analysis is necessary

• Code-centric analysis is not enough
  – Difficult to understand locality problems with data
    • Example: A=B+C:
      – Cannot tell which array is responsible for cache misses
      – Does one array have a bad layout?
      – Can one array’s access patterns be changed to increase locality?

• Data-centric analysis can provide more detail
  – Attribute metrics to specific variables
  – Data-centric optimization strategies
    • Transform access pattern to shorten reuse distance
    • Transform data layout to better support the usage pattern
    • Adjust the allocation to address NUMA issues
      – Uneven mapping of data to memory modules
Related work

- **StatCache**
  - **Approach**
    - Wrap all load/store instructions
    - Simulate memory hierarchy
    - Compute reuse distance to find temporal locality problems
  - **Disadvantages:**
    - Assume fully associative caches, LRU replacement
    - Large overhead and low accuracy, especially for shared caches

- **Memphis**
  - Measure using AMD’s Instruction-based Sampling (IBS)
  - Map metrics to static data
  - Identify NUMA problems by monitoring remote accesses
  - **Disadvantages:**
    - Require user to add instrumentation to their source code
    - Only associate measurements with static data
    - Somewhat narrowly focused on NUMA problems
Outline

• Introduction to data-centric analysis
• Implementation
• Experiments
• Conclusion
Introduction to AMD’s IBS

- Hardware provided by AMD Opteron cores
  - Periodically tag the instruction and monitor its execution
    - Record events (cache miss, TLB miss, etc.) and latencies
  - Record precise IP at the sample point

- IBS fetch: select a fetch
  - Monitor instruction cache and instruction TLB utilization
  - Fetch latency

- IBS op: select an op
  - Monitor data cache and data TLB utilization
  - Effective address for load/store instructions
  - Data load latency
  - Branch information
Data collection

- Data at allocation points
  - Heap data
    - Wrap malloc family functions
      - Record memory ranges allocated
      - Associate each data range with its allocation point
    - Record of the call path of heap allocation
  - Static data
    - Use libelf to record address ranges for static data
      - Record memory ranges allocated

- Data at IBS samples (only load/store op)
  - Precise IP
    - Unwind the call stack and record the call path
  - Effective address touched by load/store instructions
    - Associate the IBS sample to the allocation point
  - Cache miss, cache miss latency, TLB miss, data location (local/remote L3 cache/memory)
Measurement post-processing

- **Post mortem**
  - Group all samples touching the same data structure
  - Find the least common ancestor of these samples in the calling context tree

- **Display:**
  - Allocation point of data structures (for heap allocation)
  - All uses of the data in the call paths from the least common ancestor
  - Metrics
Experimental results

• Hardware
  – CPU: AMD Opteron family 10h (Barcelona)

• Applications
  – S3D: direct numerical simulation of turbulent combustion
    • Pathscale compiler
    • Data structures are mostly allocated on the heap
  – PFLOTRAN: model multi-scale, multi-phase, multi-component subsurface reactive flows
    • GNU compiler
    • Data structures are mostly allocated on the heap
  – Flash
    • Intel compiler
    • Data structures are mostly allocated statically (static data section)
Conclusions

• Innovative method to pinpoint data locality problems
  – Which data has bad performance
    • Find data structure allocation and attribute all metrics to it
    • Sort the metric values (latency, cache/TLB, NUMA info) of all data structures
  – Where does the poor performance happen
    • Decompose metrics to each use in the call path
    • Sort the uses of the data according to metric values

• Advantages
  – Low overhead
    • S3D: 24.0%, flash: 13.7%
  – High accuracy
    • No simulation
    • No “skid”

• NOT confined to AMD Opteron: Intel Nehalem with PEBS