Developing a Framework for Analyzing Data Movement within a Memory Management Runtime for Data-Intensive Applications

NVMW 2015
March 2, 2015

Brian Van Essen, Ming Jiang, Maya Gokhale
Center for Applied Scientific Computing

LLNL-PRES-667941
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC
Motivation

Analyzing application’s data movement

- Understand how much performance is “left on the table”
- Correlate performance changes with I/O behavior
  - Develop an introspection/interrogation framework

DI-MMAP

- Enable scalable out-of-core computations for data-intensive computing
- Effectively integrate non-volatile random access memory into the HPC node’s memory architecture
- Address data-intensive computing scalability challenges:
  - Use node-local NVRAM to support larger working sets
  - DRAM-cached NVRAM to extend main memory
Data-intensive memory-map runtime (DI-MMAP)

A high-performance alternative to Linux mmap:
- performance scales with increased concurrency
  - both multi-threading and multi-process
- performance does not degrade under memory pressure
- associate data structures with distinct di-mmap buffers

DI-MMAP features:
- loadable kernel module
- a fixed sized page buffer (independent of Linux page cache)
- minimal dynamic memory allocation
- a simple FIFO buffer replacement policy
- preferential caching for frequently accessed pages
Linux & DI-MMAP Basic Design

- Linux memory map runtime:
  - Optimized for shared libraries
  - Does not expect memory-mapped data to churn
  - Does not expect memory-mapped data to exceed memory capacity

- DI-MMAP:
  - Optimized for frequent evictions
  - Optimized for highly concurrent access
  - Expects to churn memory-mapped data
Buffer management applies caching techniques to NVRAM pages

Minimize the amount of effort needed to find a page to evict:

- In the steady state a page is evicted on each page fault
- Track recently evicted pages to maintain temporal reuse
- Allow bulk TLB operations to reduce inter-processor interrupts
DI-MMAP beats Linux mmap (usually)

- Random I/O (LRIOT)
  - DI-MMAP approaches direct I/O performance level

- Graph Traversal - Breadth First Search (HavoqGT)
  - 7.44x better on single node multi-threaded test
  - 10-13% better on single node multi-process
  - 2.4x improvement on distributed multi-process

- Metagenomics search (LMAT)
  - 4x faster for multi-threaded search

- Streamline tracing
  - Single-node, multi-threaded, optimized data layout
  - 10-15% better than tuned Linux memory-map

Why doesn’t streamline tracing see bigger improvement?
Streamline tracing is an important tool for visualizing and analyzing flow fields

- Computing parallel streamlines out-of-core efficiently is difficult
  - Data set size
  - Seeding density and distribution
  - Flow field complexity

- Underlying cause: *irregular* and *data-dependent* access patterns
Tuning Streamline Tracing Algorithm

- Static vs. Dynamic OpenMP scheduling
- Order of processing seed points (work partitioning)
- Data-sensitive scheduling for seed points
  - Work on seed points that already have data in the buffer
- Tracing full streamlines vs. segmented streamline
  - Keep exploring part of the simulation space while it is cached

Tuning DI-MMAP runtime

- Buffer sizes
  - Hotpage vs. primary
  - Victim vs. primary
- Queuing algorithm
  - FIFO vs. …
Understanding performance optimizations

- Which optimizations worked well (and why)
  - How much performance is left on the table

- Gather runtime metrics
  - Without significantly perturbing application runtime

- Allow application to interrogate DI-MMAP framework
DI-MMAP Introspection Interface

- Memory-mapped interfaces
  - Lightweight access
  - Real-time
  - Low-overhead
  - Available via devfs

- Page Status Histogram
  - major / minor faults
  - cache residency
  - data valid
  - numa-node

- Fault Sequence Window
  - log faults
  - log evictions

---

DI-MMAP Buffer

Page: Location Hashable

Status Histogram

- Hotpage FIFO
- Primary FIFO
- Eviction Queue
- Writeback Queue
- Free Page List
- Physical Page Array

---

Fault Sequence Window

Record Fault

---

Fault Statistics

Evicted Page Statistics

<table>
<thead>
<tr>
<th>fault_seq</th>
<th>V</th>
<th>T</th>
<th>D</th>
<th>Q</th>
<th>FC</th>
<th>DevID</th>
<th>pgOffset</th>
<th>V</th>
<th>T</th>
<th>D</th>
<th>Q</th>
<th>FC</th>
<th>DevID</th>
<th>pgOffset G</th>
</tr>
</thead>
<tbody>
<tr>
<td>3641503</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>F</td>
<td></td>
<td>2211.</td>
<td>636008 0x000000efc6000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>8</td>
<td>2211.</td>
<td>636008 0x000000efcf</td>
</tr>
<tr>
<td>3641504</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>F</td>
<td></td>
<td>2211.</td>
<td>636008 0x000000efc6000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>6</td>
<td>2211.</td>
<td>636008 0x000000f8c9</td>
</tr>
<tr>
<td>3641505</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>F</td>
<td></td>
<td>2211.</td>
<td>636008 0x000000efc6000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>3</td>
<td>2211.</td>
<td>636008 0x000000e8e18</td>
</tr>
<tr>
<td>3641506</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>F</td>
<td></td>
<td>2211.</td>
<td>636008 0x000000efc6000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>6</td>
<td>2211.</td>
<td>636008 0x000000c64d</td>
</tr>
<tr>
<td>3641507</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>F</td>
<td></td>
<td>2211.</td>
<td>636008 0x000000efc6000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>4</td>
<td>2211.</td>
<td>636008 0x000000c640</td>
</tr>
</tbody>
</table>
Runtime Metrics

- Capture general runtime states that indicate:
  - page reuse
  - I/O concurrency
  - Buffer contention

- Track specific minor fault events in DI-MMAP runtime:
  - recovery
  - collision
  - reuse

- How do we relate metrics to performance:
  - Major page faults dominate runtime
  - Minor page faults provide additional insight into I/O patterns
Major Fault: Timing of basic page fault

- Major page faults
  - ~1us for fault handler
  - ~50-75us for read data
Minor Fault: Recovering pages about to be evicted

- **Clean page**
  - Recover from victim queue

- **Dirty page**
  - Recover from writeback queue

- **No data movement required**
Minor Fault: Collision on same page

Early second fault

Late second fault

- Second thread waits on first thread
  - No additional I/O concurrency
Show the performance from major faults

- Processing seed points in row-major order:
  - Major faults generally correlate well with wall clock time

- Processing seed points randomly:
  - Number of major faults is much closer than difference in wall clock time
Show the performance impact of different types of minor faults

- OpenMP Dynamic scheduling
  - Recovers more pages
  - Fewer fault collisions on mid-sized streamlines

- Performance at mid-sized streamlines is better with static scheduling? Why???
Preliminary Conclusions: nothing is clear cut

- Page fault collisions – good or bad
  - Indicate page reuse
  - Limit potential I/O concurrency
  - Impact probably varies with number of application threads
    - Does thread oversubscription makes this always good?

- Page fault recovery
  - Indicates temporal reuse (good)
  - Does it mean that runtime is failing to capture hot pages or that the buffer is too small
Profiling overhead

- DI-MMAP overhead is minimal

- Application:
  - Records unique address in per thread STL map
  - Post-processing interrogation of DI-MMAP
  - Page residency uses lightweight pre-access check
  - ~37% overhead at 16384 length
Mapping metrics back to application space

- Visualize (overlay) active buffer pages on application data structures
  - Streamlines are shown from seed point (silver sphere) to termination
  - Pink cubes show the active pages in the buffer for the current time step

- Spatial distribution of buffered pages w.r.t. streamlines illustrate reuse within streamline clusters
  - Identifiers potential opportunity for intelligent pre-fetching
- Fully tracing long streamlines serially leads to less data reuse between seeds
Streamline Tracing: what’s left on the table

- There is relatively low amount of page reuse in turbulent regions
  - Ratio of major faults to unique pages is relatively high
  - Minor fault profile – ratio of minor faults to major faults is low
  - Visualization shows interplay (or lack thereof) of streamlines

- Data-dependent seed scheduling
  - Pick seeds from neighborhood that have ready data

- Segment streamline processing
  - Partially trace streamlines – split long streamlines into multiple short streamlines
  - Process seed points with data-dependent scheduling

- Both techniques should:
  - Reduce number of major page faults
  - Increase the number of minor recovery page faults
Next steps: runtime

- Further correlate runtime metrics with performance variation

- Enable runtime to determine if application changes made execution better
  - Isolate bottlenecks
  - Identify opportunities for improvement

- Representing I/O behavior in application environment
  - Improve algorithm / scheduling optimization through visualization
Minor Fault: Reusing the same page

PTE missing triggers second fault.

Occurs when:
- PTE not established for second thread
- Multiple processes access the same page