Understanding Object-level Memory Access Patterns Across the Spectrum

Xu Ji, Chao Wang, Nosayba El-Sayed (MIT/QCRI), Xiaosong Ma, Youngjea Kim, Sudharshan Vazhkudai, Xue Wei, Daniel Sanchez
Why are we obsessed with memory?

Challenge: Memory performance not improving as fast as computation

- Top supercomputer bandwidth/FLOP?
  - 0.85 [1997] ➔ 0.5 [2011] ➔ 0.002 [2018]

Source: www.top500.org

Opportunity: Today’s systems have deep memory hierarchies, combining different technologies (SRAM, DRAM, NVM, HBM, etc.)

Optimizing programs and systems for efficient memory access is crucial
Optimizing requires understanding

- Understand program’s memory allocation and access behaviours (temporally & spatially)

- Limitations of prior profiling studies?
  - Collected/Sampled memory traces to derive specific info
    - **Disconnected from program semantics**
  - Limited to specific input size / application domain

**Contributions**

- **New profiling approach** bridging program high-level semantics & low-level memory accesses
- **Comprehensive study** covering multiple app domains (HPC, datacenter, commercial, desktop) & problem sizes
Two-Level program memory profiling

void eo_fermion_force( double eps,  
    int nflavors, field_offset x_off ){  
    /* Allocate temporary vectors */  
    for(mu=0;mu<8;mu++)  
        tempvec[mu] = (su3_vector *)  
            calloc(sites_on_node, size);  
    /*copy x_off to a temporary vector*/  
    temp_x = (su3_vector *)  
        calloc(sites_on_node, size);  
    ...  
}

Program Variables & Computation Semantics

Memory Objects

Memory References

Object to variable mapping

Address to object mapping
Key Definitions

- **Object**: Contiguous heap memory region allocated by dynamic alloc functions; e.g. `malloc()`, `calloc()`, or `realloc()`

- **Variable** [Barrett et al. SIGPLAN’93]: Entity grouping all objects allocated within same call-stack (i.e. function names & return addresses)

```c
void eo_fermion_force( double eps,
    int nflavors, field_offset x_off ){
    ... /* Allocate temporary vectors */
    for (mu=0; mu<8; mu++)
        tempvec[mu] = (su3_vector *)
            calloc(sites_on_node, sizeof(su3_vector));

    /* copy x_off to a temporary vector */
    temp_x = (su3_vector *)
        calloc(sites_on_node, sizeof(su3_vector));
}
```

**Call-Stack (Var₁)**
- getalloc.so: calloc()+0x12e
- ./milc: eo_fermion_force()+0xb7
- ./milc: update()+0xd2
- ./milc: main()+0xb2

**Call-Stack (Var₂)**
- getalloc.so: calloc()+0x12e
- ./milc: eo_fermion_force()+0xd7
- ./milc: update()+0xd2
- ./milc: main()+0xb2
Roadmap

- Introduction
- Profiling Tool Overview
- Profiling Methodology
- Results and Analysis
Two-pass profiling tool

(1) First Pass (collect per-object info)

Application Allocation calls Custom memory allocation library

Object info (size, lifetime, call-stack)

(2) Second Pass (target variable info)

Application Allocation calls Custom memory allocation library

Mem references Target object address intervals Target variable call-stacks

Custom Pintool (Runtime object access analysis)

Off-line processing (Target variable selection: top 10 in mem utilization)

Target variable access characteristics
(Locality, Access density, Sequentiality, R/W-ratio...)

Smaller overheads than traditional tracing (creates TBs in minutes)

Final output
Roadmap

- Introduction
- Profiling Tool Overview
- Profiling Methodology/Workloads
- Results and Analysis
Profiled workloads

38 Applications:

- NPB: 7 Fortran apps
- Real-world HPC: LAMMPS, mpiBLAST, gromacs
- SPEC (Scientific-Comp): 5 C/C++ apps
- SPEC (Commercial/Desktop): 12 C/C++ apps
- PARSEC: 7 C/C++ apps
- PBBS [SPAA’12]: spanningForest, BFS
- TailBench [IISWC’16]: silo, dnn

Profile executions under 3 problem sizes:

- Small, medium, large (based on available datasets)
Overall variable/object behaviour

Counts, sizes, lifetimes, concurrent objects, etc.
Profiling methodology and targets

- **Overall variable/object behaviour**
  - Counts, sizes, lifetimes, concurrent objects, etc.

- **Focused study of major variables**
  - Top-10 vars in memory footprint (peak combined memory consumed by concurrent objects)
  - Discard vars whose largest object < 4KB

- **Profiling parallel applications**
  - Multi-process parallel codes (e.g. mpiBLAST, LAMMPS, NPB)
  - Multi-threaded support (e.g. PARSEC, PBBS)
    - Allocation & access patterns very similar across processes/threads
    - Focused our analysis on single executions
Roadmap

- Introduction
- Profiling Tool Overview
- Profiling Methodology
- Results and Analysis
Object counts and sizes

- Critical for estimating runtime cost of data placement & memory allocation optimizations

### Application Counts and Sizes

<table>
<thead>
<tr>
<th>Application</th>
<th>#variables</th>
<th>#objects</th>
<th>#objects per major variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpiBLAST (SC)</td>
<td>1,284</td>
<td>5,961,518</td>
<td>2.3</td>
</tr>
<tr>
<td>LAMMPS (SC)</td>
<td>693</td>
<td>3,004</td>
<td>27</td>
</tr>
<tr>
<td>CG (SC)</td>
<td>12</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>gobmk</td>
<td>43</td>
<td>118,627</td>
<td>1</td>
</tr>
<tr>
<td>perlbench</td>
<td>7,268</td>
<td>59,080,718</td>
<td>911,292</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>4,802</td>
<td>135,155,557</td>
<td>407,561</td>
</tr>
</tbody>
</table>

Sci-Comp have fewer #objects & larger average object sizes

### Implication

With fewer & larger objects, Sci-Comp apps are candidates for runtime memory object placement optimizations
What about object *lifetime*?

- Distributions are skewed
  - > 95% of objects have sizes smaller than 1% of largest obj.
  - Majority of large objects had lifetimes under 5% of exec time

**Target objects in Sci-Comp vs. Others?**
- Target objects in *Sci-Comp* tend to be long-lived (*Sci-Comp* median=0.99 vs. Others median=0.19)

- **Long-lived** objects in *Sci-Comp* appealing for mem allocation optimizations (pay optimization overhead once!)

- **Short-lived** objects in *commercial/desktop* allow more flexible placement & interleaved utilization of faster memory layers
Problem size scaling

- **Goal:** Understand program memory behaviour changes under different problem sizes

- **Method:** Monitor size of the *largest object* in each major variable

- Classify each profiled variable as:
  - Fixed
  - Scaling
  - Irregular
Sci-Comp exhibit highly uniform scaling behaviour (up to 3 scaling rates); “Fixed” → footprint-consuming data structures not tied to problem size, or size scaling incurs more computation (e.g. more chess moves)

☑ Sci-Comp: behaviour more predictable with varying size
☑ Others: Optimizations based on offline profiling are challenging
Data Structure Types (DSTs)

1-D hash table

Key
Jim
Bill
Joy

Hash Func

Hash Table

1-D buffer

Buffer

Index
Vector

1-D index & vector

Index
Vector

1-D bitmap

Bitmap

1
0
1

2-D or MD matrix

Matrix

2-D collection

Collection
Which data types are more popular?

MD-Matrices prominent for Sci-Comp, while 1D DSTs are common in Others

Insights on data types can be exploited by tools for code annotation (by compiler/programmer) to assist runtime data placement
Object “Density” Scaling

- Object density = total reference volume / size
  - Useful in determining object _worthiness_ to be placed on faster/closer layers

- How does density scale with input size?

Density varies across variables in the same application, and may or may not scale with problem sizes

**Offline profiling** for judging _heat levels_ can be misleading
More interesting memory-access patterns analysis...

[In the paper]

- Temporal locality
- Spatial locality
- Read/Write Ratio
- Sequentiality
- MD-Matrix Access Patterns (SC)
- Impact of observation window-size
Summary

- Object-level profiling provides intuitive insights into program memory behaviour
- Applications have diverse variable-level behaviour
  - Often inconsistent when scaling problem sizes
  - Hybrid offline-online profiling recommended
- Scientific codes do possess different object patterns
  - Relatively small #objects, larger obj size, scaling, long-lived, MD-Matrix DST
- Commonly used HPC benchmarks do not capture complexity in memory object allocation/access of real-world large applications
Object-level profiling provides intuitive insights into program memory behaviour

Applications have diverse variable-level behaviour
- Often inconsistent when scaling problem sizes
- Hybrid offline-online profiling recommended

Scientific codes do possess different object patterns
- Relatively small #objects, larger obj size, scaling, long-lived, MD-Matrix DST

Commonly used HPC benchmarks do not capture complexity in memory object allocation/access of real-world large applications
Backup slides
Bzip2 Result

- **Easy-to-compress vs Hard-to-compress workload**

  - **Temporal Locality**
  - **Spatial Locality**
  - **Access Volume (Bytes)**
  - **Read Ratio**
Access behaviors vary across adjacent 1B-instr. windows; Variation (CoV) stabilizes after ~5B instr. window sizes.

Common window sizes of several 100M instr. may fail to capture program’s memory access behaviour accurately, even during steady-phase
Object Footprint

How early does an application reach its peak allocation?

Most apps reach near-peak footprint early in execution, which can be used to guide cloud/datacenter managing decisions (e.g. VM migration, online profiling)

Majority of applications consume 80% of peak footprint by only 20% of exec time!
### Same-variable Object Behavior

For data placement, variable level information is enough to distinguish object access behavior, especially in access patterns.

<table>
<thead>
<tr>
<th># of objects</th>
<th># of variables</th>
<th>Distribution of CoV in object size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
<td>0</td>
</tr>
<tr>
<td>2-10</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>11-50</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>50-100</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>100-500</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>&gt;500</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

In general, objects associated with same variable are found to possess highly similar behavior, especially in access patterns.
## Top Supercomputer

<table>
<thead>
<tr>
<th>Name</th>
<th>Nodes</th>
<th>Cores per node</th>
<th>Total cores</th>
<th>Memory per node (GB)</th>
<th>Total Memory</th>
<th>Computation (PFLOPS)</th>
<th>Memory/Computation (Bytes/FLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018¹</td>
<td>100,000</td>
<td>1000</td>
<td>1e8</td>
<td>32-64</td>
<td>32PB~64PB</td>
<td>1,000</td>
<td>0.0032~0.0064</td>
</tr>
<tr>
<td>TaihuLight</td>
<td>40,960</td>
<td>260</td>
<td>10,649,600</td>
<td>32</td>
<td>1,310TB</td>
<td>93.0</td>
<td>0.014</td>
</tr>
<tr>
<td>Tianhe-2</td>
<td>16,000</td>
<td>24</td>
<td>384,000</td>
<td>64</td>
<td>1,000TB</td>
<td>33.8</td>
<td>0.029</td>
</tr>
<tr>
<td>Titan</td>
<td>18,688</td>
<td>16</td>
<td>299,008</td>
<td>32</td>
<td>584TB</td>
<td>17.6</td>
<td>0.033</td>
</tr>
</tbody>
</table>

Data Structure Types

- 1-D hash table: Typically identified by variable name or use of hash function;
- 1-D buffer: Array for intermediate storage (typically for input/output data) outside main computation of application/function, identified by name or lifetime plus processing behavior
- 1-D index: Array indexing into another data structure, identified by name or use of indexing operations
- 1-D bitmap: Identified by name or bit-level operations (excluding above three types)
- 1-D vector: Default category for 1-D arrays not identified as above four types
- 2-D matrix: 2-D array storing homogeneous data elements, identified by name or data semantics plus (x; y)-style reference pattern
- 2-D collection: Set of arrays presenting independent objects, such as different attributes of same data entity, identified by name or element data types
Access Density

- Defined as object’s total reference volume divided by its size

Major variables have widely distributed density, even within same application. App often sees its major variables having different scaling behavior across problem sizes.

Different densities bring opportunity of placement, but it’s hard to predict density.
Access Pattern

- More read-intensive than write-intensive
- More sequential than Random
- Relatively high temporal/spatial locality