DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

Geraldo F. Oliveira
Juan Gómez-Luna   Lois Orosa   Saugata Ghose
Nandita Vijaykumar   Ivan Fernandez   Mohammad Sadrosadati
Onur Mutlu

SAFARI

ETH Zürich  UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN  UNIVERSITY OF TORONTO  UNIVERSITAS PALACITAN  UNIVERSIDAD DE MÁLAGA
The Executive Summary highlights the importance of identifying data movement bottlenecks in modern systems. The problem statement emphasizes the need for methodologies to distinguish between different sources of data movement bottlenecks and to determine the most suitable mitigation techniques. The discussion is framed under several key sections, including goals, key approach, key contributions, and a brief overview of DAMOV (a benchmark suite with 144 functions). The text references SAFE and DAMOV, pointing towards a GitHub repository for more detailed information.
Outline

1. Data Movement Bottlenecks
2. Methodology Overview
3. Application Profiling
4. Locality-Based Clustering
5. Memory Bottleneck Analysis
6. Case Studies
Data movement bottlenecks happen because of:

- Not enough data **locality** → ineffective use of the cache hierarchy
- Not enough **memory bandwidth**
- High average **memory access time**
Data Movement Bottlenecks (2/2)

**Compute-Centric Architecture**

- Abundant DRAM bandwidth
- Shorter average memory access time

**Memory-Centric Architecture**

**Near-Data Processing (NDP)**

SAFARI
The goal of Near-Data Processing (NDP) is to mitigate data movement

- Abundant DRAM bandwidth
- Shorter average memory access time
The goal of Near-Data Processing (NDP) is to mitigate data movement.

**Near-DRAM-banks processing for general-purpose computing**

- **UPMEM (2019)**

  - 0.9 TOPS compute throughput

**Near-DRAM-banks processing for neural networks**

- **Samsung FIMDRAM (2021)**

  - 1.2 TFLOPS compute throughput

---


When to Employ Near-Data Processing?

Mobile consumer workloads (GoogleWL^2)

Graph processing (Tesseract^1)

Neural networks (GoogleWL^2)

Databases (Polynesia^5)

DNA sequence mapping (GenASM^3; GRIM-Filter^4)

Time series analysis (NATSA^6)

Near-Data Processing

Identifying Memory Bottlenecks

• Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP

• Existing approaches are not comprehensive enough
Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is *bounded* by **compute** or **memory** units

![Diagram](image_url)

- **Memory Roof**: $y = BW \times AI$
- **Compute Roof**: $y = \text{Peak System Throughput}$

- **Compute Bound** → Not suitable for NDP
- **Memory Bound** → Suitable for NDP
Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is *bounded by compute or memory* units
Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is *bounded* by **compute** or **memory** units.

Memory Bound applications are **faster on CPU**, or performance **depends** ✗

Compute Bound applications are **faster on CPU** ✓

Memory Bound applications are **faster on NDP** ✓

Compute Bound applications have **similar performance** on CPU/NDP or performance **depends** ✗
Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is 
  bounded by compute or memory units

Roofline model **does not accurately account** for the **NDP suitability** of memory-bound applications.

- **Memory Bound** applications are faster on **NDP**
- **Compute Bound** applications have **similar performance** on CPU/NDP or performance depends

Arithmetic intensity (OPS/byte)

SAFARI
Limitations of Prior Approaches (2/2)

• Application with a last-level cache MPKI $> 10$ → memory intensive and benefits from NDP
Limitations of Prior Approaches (2/2)

- Application with a last-level cache **MPKI > 10** → **memory intensive** and benefits from NDP

Applications with low MPKI can be faster on NDP; have similar performance on CPU/NDP or; performance can depends

Applications with low MPKI are faster on CPU

Applications with high MPKI are faster on NDP

**SAFARI**
Limitations of Prior Approaches (2/2)

- Application with a last-level cache MPKI > 10 → memory intensive and benefits from NDP.

Applications with low MPKI can be faster on NDP;

LLC MPKI does not accurately account for the NDP suitability of memory-bound applications.
Identifying Memory Bottlenecks

• Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP

• Existing approaches are not comprehensive enough
The Problem

• Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP

No available methodology can comprehensively:

- **identify** data movement bottlenecks

- **correlate** them with the **most suitable**
  data movement mitigation mechanism
Our Goal: develop a methodology to:

- methodically identify sources of data movement bottlenecks
- comprehensively compare compute- and memory-centric data movement mitigation techniques
Key Approach

• New **workload characterization methodology** to analyze:
  - data movement bottlenecks
  - suitability of different data movement mitigation mechanisms

• Two main profiling strategies:

  - **Architecture-independent profiling:** characterizes the memory behavior *independently* of the underlying *hardware*.

  - **Architecture-dependent profiling:** evaluates the *impact of the system configuration* on the memory behavior.
Methodology Overview

User Input
- Target Application
  - Source Code
    - User Input

Step 1: Application Profiling
- Profiler
  - roi_begin
  - roi_end

Step 2: Locality-based Clustering
- DRAM Bandwidth
- DRAM Latency
- L1/L2 Cache Capacity
- L3 Cache Contention
- Compute Bound

Methodology Output
- Memory Bottleneck Classes

Step 3: Memory Bottleneck Class
- Memory Traces
- Scalability Analysis

DAMOV-SIM Simulator
- ld 0xFF
- st 0xAF
- ld 0xFF
- st 0xAF
- ld 0xFF

Temp.
- Locality
- LFMR
  - Low
  - High
- …

Profiler
- Step 1
- Application Profiling

SAFARI

Methodology Output
- Memory Bottleneck Classes

Step 2
- Locality-based Clustering

Step 3
- Memory Bottleneck Class
Methodology Overview

User Input
- Target Application
  - Source Code
  - Profiler
  - roi_begin
  - roi_end

Step 1
Application Profiling
- Memory Traces
- Temporal Locality
- Spatial Locality

Step 2
Locality-based Clustering
- DRAM Bandwidth
- DRAM Latency
- L1/L2 Cache Capacity
- L3 Cache Contention
- Compute-bound

Step 3
Memory Bottleneck Class.

Methodology Output
- Memory Bottleneck Classes

SAFARI

DAMOV-SIM Simulator
- ld 0xFF
- st 0xAF
- ld 0xFF
- st 0xAF
- ld 0xFF

# Cores

Memory Traces

Scalability Analysis

Profiler

Application Profiling

ROI

Last to First Miss Ratio (LFMR)
Step 1: Application Profiling

**Goal:** Identify **application functions** that suffer from **data movement bottlenecks**

Hardware Profiling Tool: **Intel VTune**

**MemoryBound:**
CPU is stalled due to load/store
Methodology Overview

1. **Application Profiling**
   - Step 1
   - User Input: Target Application, Source Code
   - Profiler:
     - roi_begin
     - roi_end
   - Output: Memory Traces

2. **Locality-based Clustering**
   - Step 2
   - DAMOV-SIM Simulator
   - # Cores
   - L1/L2 Cache Capacity
   - L3 Cache Contention
   - DRAM Bandwidth
   - DRAM Latency
   - Temporal Locality
   - Spatial Locality

3. **Memory Bottleneck Class**
   - Step 3
   - Methodology Output
   - Memory Bottleneck Classes
   - Logical Flow
   - High
   - Low

SAFARI
Step 2: Locality-Based Clustering

- **Goal**: analyze application’s memory characteristics

**Spatial Locality**

**Memory Trace**

```
0 1 2 3 4 5
```

```
stride profile(1)+= 1
```

- **Low spatial locality**
  - Stride Profile Histogram
- **High spatial locality**
  - Stride Profile Histogram

---

Step 2: Locality-Based Clustering

- **Goal**: analyze application’s memory characteristics

**Spatial Locality**

- **Low spatial locality**: Lower frequency of reuse and a more even distribution of strides.

  - **Stride Profile Histogram**: Low frequency for small strides and higher frequency for larger strides.

- **High spatial locality**: Higher frequency for small strides and lower frequency for larger strides.

**Temporal Locality**

- **Low temporal locality**: Lower frequency of reuse over time.

  - **Reuse Profile Histogram**: Low frequency for recent accesses and a steady increase over time.

- **High temporal locality**: Higher frequency for recent accesses and a more even distribution over time.

---

Methodology Overview

**Step 1: Application Profiling**

- User Input: Source Code, Target Application
- Methodology Output: LFMR
- Memory Traces: ROI begin, ROI end

**Step 2: Locality-based Clustering**

- DRAM Bandwidth
- DRAM Latency
- L1/L2 Cache Capacity
- L3 Cache Contention
- Compute-bound

**Step 3: Memory Bottleneck Class**

- Arithmetic Intensity
- LLC MPKI
- Last-to-First Miss Ratio (LFMR)

**SAFARI**

**DAMOV-SIM Simulator**

- # Cores
- Scalability Analysis
Step 3: Memory Bottleneck Classification (1/2)

**Arithmetic Intensity (AI)**
- floating-point/arithmetic operations per L1 cache lines accessed
  → shows computational intensity per memory request

**LLC Misses-per-Kilo-Instructions (MPKI)**
- LLC misses per one thousand instructions
  → shows memory intensity

**Last-to-First Miss Ratio (LFMR)**
- LLC misses per L1 misses
  → shows if an application benefits from L2/L3 caches
Step 3: Memory Bottleneck Classification (2/2)

**Goal:** identify the specific sources of data movement bottlenecks

- **Scalability Analysis:**
  - 1, 4, 16, 64, and 256 out-of-order/in-order host and NDP CPU cores
  - 3D-stacked memory as main memory

DAMOV-SIM Simulator

Integrated ZSim and Ramulator

**Configuration 1: Host CPU System**

**Configuration 2: NDP System**

DAMOV-SIM: [https://github.com/CMU-SAFARI/DAMOV](https://github.com/CMU-SAFARI/DAMOV)
Outline

1. Data Movement Bottlenecks
2. Methodology Overview
3. Application Profiling
4. Locality-Based Clustering
5. Memory Bottleneck Analysis
6. Case Studies
Step 1: Application Profiling

- We analyze 345 applications from distinct domains:
  - Graph Processing
  - Deep Neural Networks
  - Physics
  - High-Performance Computing
  - Genomics
  - Machine Learning
  - Databases
  - Data Reorganization
  - Image Processing
  - Map-Reduce
  - Benchmarking
  - Linear Algebra
  ...

SAFARI
Memory Bound Functions

- We analyze **345 applications** from distinct domains
- **Selection criteria:** clock cycles > **3%** and Memory Bound > **30%**

We find **144 functions** from a total of **77K functions** and select:
- 44 functions → **apply steps 2 and 3**
- 100 functions → **validation**
Outline

1. Data Movement Bottlenecks
2. Methodology Overview
3. Application Profiling
4. Locality-Based Clustering
5. Memory Bottleneck Analysis
6. Case Studies
Step 2: Locality-Based Clustering

We use K-means to cluster the applications across both spatial and temporal locality, forming two groups:

1. Low locality applications (in orange)
2. High locality applications (in blue)
Step 2: Locality-Based Clustering

We use K-means to cluster the applications across both spatial and temporal locality, forming two groups:

1. Low locality applications (in orange)
2. High locality applications (in blue)

The closer a function is to the **bottom-left corner** → less likely it is to **take advantage** of a deep cache hierarchy.
Outline

1. Data Movement Bottlenecks
2. Methodology Overview
3. Application Profiling
4. Locality-Based Clustering
5. Memory Bottleneck Analysis
6. Case Studies
Step 3: Memory Bottleneck Analysis

Memory Bottleneck Class

Temporal Locality

LFMR

MPKI

Low

High

MPKI

Low

AI

Low

High

AI

AI

Decreasing

Low

Increasing

High

Low

Low

Low

Low

Low

Low

High
Step 3: Memory Bottleneck Analysis

Memory Bottleneck Class

1a: DRAM Bandwidth
1b: DRAM Latency
1c: L1/L2 Cache Capacity
2a: L3 Cache Contention
2b: L1 Cache Capacity
2c: Compute-Bound

Temporal Locality

LFMR

MPKI

High

Low

Decreasing

Increasing

High

Low

High

Low

Low

Low

Low

Low

Low

Low

Low

High

Low

SAFARI
Step 3: Memory Bottleneck Analysis

Six classes of data movement bottlenecks:

- each class ↔ data movement mitigation mechanism

Memory Bottleneck Class:

1a: DRAM Bandwidth
1b: DRAM Latency
1c: L1/L2 Cache Capacity
2a: L3 Cache Contention
2b: L1 Cache Capacity
2c: Compute-Bound
Step 3: Memory Bottleneck Analysis

Temporal Locality

Memory Bottleneck Class

1a: DRAM Bandwidth

1b: DRAM Latency

1c: L1/L2 Cache Capacity

2a: L3 Cache Contention

2b: L1 Cache Capacity

2c: Compute-Bound
Class 1a: DRAM Bandwidth Bound (1/2)

- High MPKI → high memory pressure
- Host scales well until bandwidth saturates
- NDP scales without saturating alongside attained bandwidth

**Host**

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Performance</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NDP**

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Performance</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DRAM bandwidth bound applications:
NDP does better because of the higher internal DRAM bandwidth
Class 1a: DRAM Bandwidth Bound (2/2)

- High LFMR → L2 and L3 caches are inefficient
- Host’s energy consumption is dominated by cache look-ups and off-chip data transfers
- NDP provides large system energy reduction since it does not access L2, L3, and off-chip links

**DRAM bandwidth bound applications:**
NDP does better because it eliminates off-chip I/O traffic
Step 3: Memory Bottleneck Analysis

Memory Bottleneck Class

1a: DRAM Bandwidth

1b: DRAM Latency

1c: L1/L2 Cache Capacity

2a: L3 Cache Contention

2b: L1 Cache Capacity

2c: Compute-Bound

Temporal Locality

LFMR

MPKI

AI

Low

Decreasing

High

Increasing

High

Low
Class 1b: DRAM Latency Bound

- High LFMR $\rightarrow$ L2 and L3 caches are inefficient
- Host scales well but NDP performance is always higher
- NDP performs better than host because of its lower memory access latency

**DRAM latency bound applications:**

host performance is hurt by the cache hierarchy and off-chip link
Step 3: Memory Bottleneck Analysis

Temporal Locality

1a: DRAM Bandwidth
1b: DRAM Latency
1c: L1/L2 Cache Capacity

2a: L3 Cache Contention
2b: L1 Cache Capacity
2c: Compute-Bound
Class 1c: L1/L2 Cache Capacity

- Decreasing LFMR $\rightarrow$ L2/L3 caches turn efficient
- NDP scales better than the host at low core counts
- Host scales better than NDP at high core counts
- Host performs better than NDP at high core counts since it reduces memory access latency via data caching

L1/L2 cache capacity bottlenecked applications:
NDP is higher performance when the aggregated cache size is small

SAFARI
Step 3: Memory Bottleneck Analysis

Memory Bottleneck Class

1a: DRAM Bandwidth

1b: DRAM Latency

1c: L1/L2 Cache Capacity

2a: L3 Cache Contention

2b: L1 Cache Capacity

2c: Compute-Bound
Class 2a: L3 Cache Contention

- Increasing LFMR → L2/L3 caches turn inefficient
- Host scales better than the NDP at low core counts
- NDP scales better than host at high core counts
- NDP performs better than host at high core counts since it reduces memory access latency

L3 cache contention bottlenecked applications: at high core counts, applications turn into DRAM latency-bound
Step 3: Memory Bottleneck Analysis

Temporal Locality

- LFMR
  - Low
  - Decreasing
  - MPKI
  - High
  - High
  - AI
  - Low
  - Low

Memory Bottleneck Class

1a: DRAM Bandwidth

1b: DRAM Latency

1c: L1/L2 Cache Capacity

2a: L3 Cache Contention

2b: L1 Cache Capacity

2c: Compute-Bound

SAFARI
Class 2b: L1 Cache Capacity

- Low LFMR, MPKI; high temporal locality → efficient L2/L3 caches, low memory intensity
- Low AI → few operations per byte
- Host and NDP performance are similar → L1 dominates average memory access time

L1 cache capacity bottlenecked applications:
NDP can be used to reduce the host overall SRAM area
Step 3: Memory Bottleneck Analysis

**Memory Bottleneck Class**

1a: DRAM Bandwidth

1b: DRAM Latency

1c: L1/L2 Cache Capacity

2a: L3 Cache Contention

2b: L1 Cache Capacity

2c: Compute-Bound

**Temporal Locality**

- Low LFMR → High MPKI → High AI → Low DRAM Bandwidth
- Low LFMR → Low MPKI → Low AI → Low DRAM Latency
- Decaying LFMR

**High**

- Low LFMR → Low MPKI → Low AI → Low L1/L2 Cache Capacity
- High LFMR → High MPKI → High AI → High L3 Cache Contention

**Low**

- Low LFMR → Low MPKI → Low AI → Low L1 Cache Capacity
- High LFMR → High MPKI → High AI → High Compute-Bound
Class 2c: Compute-Bound

- Low LFMR, MPKI; high temporal locality
  → efficient L2/L3 caches, low memory intensity

- High AI → many operations per byte

- Host performs better than NDP because computation dominates execution time

**Compute-bound applications:** benefit highly from cache hierarchy; NDP is *not* a good fit
Step 3: Memory Bottleneck Analysis

Temporal Locality

- Low LFMR → High MPKI → High AI
  - 1a: DRAM Bandwidth
  - 1b: DRAM Latency
  - 1c: L1/L2 Cache Capacity

- Low LFMR → Decreasing MPKI → Low AI
  - 2a: L3 Cache Contention

- High LFMR → Increasing MPKI → Low AI
  - 2b: L1 Cache Capacity
  - 2c: Compute-Bound
Step 3: Memory Bottleneck Analysis

DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

GERALDO F. OLIVEIRA¹, JUAN GÓMEZ-LUNA¹, LOIS OROSA¹, SAUGATA GHOSE², NANDITA VIJAYKUMAR³, IVAN FERNANDEZ¹,⁴, MOHAMMAD SADROSADATI¹, and ONUR MUTLU¹

¹ETH Zürich, Switzerland
²University of Illinois Urbana-Champaign, USA
³University of Toronto, Canada
⁴University of Malaga, Spain

Corresponding author: Geraldo F. Oliveira (e-mail: geraldod@inf.ethz.ch).

SAFARI
Methodology Validation

**Goal:** evaluate the **accuracy** of our workload characterization methodically on a large set of functions

**Two-phase validation:**

**Phase 1:**
- Calculate thresholds \( (T) \)
- Temporal Locality
- Arithmetic Intensity
- LLC MPKI
- Last-to-First Miss Ratio

**Calculate**
- 44 functions

**Phase 2:**
- Classify
- \( T_{\text{Temporal Locality}} \)
- \( T_{\text{Arithmetic Intensity}} \)
- \( T_{\text{LLC MPKI}} \)
- \( T_{\text{Last-to-First Miss Ratio}} \)

**Classify**
- 100 functions

**High accuracy:**
our methodology accurately classifies 97% of functions into one of the six memory bottleneck classes
• Effect of the last-level cache size
  - Large L3 cache size (e.g., 512 MB) can mitigate some cache contention issues

• Summary of our workload characterization methodology
  - Including workload characterization using in-order host/NDP cores

• Limitations of our methodology

• Benchmark diversity
More in the Paper

• Effect of the last-level cache size
  - Large L3 cache size (e.g., 512 MB) can mitigate some cache contention issues

• Summary of our workload characterization methodology
  - Including workload characterization using in-order host/NDP cores

• Limitations of our methodology

• Benchmark diversity
Case Studies

• Many open questions related to NDP system designs:
  - Interconnects
  - Data mapping and allocation
  - NDP core design (accelerators, general-purpose cores)
  - Offloading granularity
  - Programmability
  - Coherence
  - System integration
  - ...

• Goal: demonstrate how DAMOV is useful to study NDP system designs

Case Studies

Load Balance and Inter-Vault Communication on NDP

NDP Accelerators and Our Methodology

Different Core Models on NDP Architectures

Fine-Grained NDP Offloading
Load Balance and Inter-Vault Communication on NDP

portion of the memory requests an NDP core issues go to remote vaults → increases the memory access latency for the NDP core

NDP Accelerators and Our Methodology

Different Core Models on NDP Architectures

Fine-Grained NDP Offloading
Load Balance and Inter-Vault Communication on NDP

NDP Accelerators and Our Methodology
NDP accelerator is faster than compute-centric accelerator for Class 1a and 1b applications; slower for Class 2c
→ key observations hold for other NDP architectures

Different Core Models on NDP Architectures

Fine-Grained NDP Offloading
Load Balance and Inter-Vault Communication on NDP

NDP Accelerators and Our Methodology

Different Core Models on NDP Architectures

using in-order cores limits performance of some applications

→ static instruction scheduling cannot exploit memory parallelism

Fine-Grained NDP Offloading
Load Balance and Inter-Vault Communication on NDP

NDP Accelerators and Our Methodology

Different Core Models on NDP Architectures

Fine-Grained NDP Offloading

few basic blocks are responsible for most of LLC misses

→ offloading such basic blocks to NDP are enough to improve performance
Case Studies

Load Balance and Inter-Vault Communication on NDP
portion of the memory requests an NDP core issues go to remote vaults
→ increases the memory access latency for the NDP core

NDP Accelerators and Our Methodology
NDP accelerator is faster than compute-centric accelerator for Class 1a and 1b applications; slower for Class 2c
→ key observations hold for other NDP architectures

Different Core Models on NDP Architectures
using in-order cores limits performance of some applications
→ static instruction scheduling cannot exploit memory parallelism

Fine-Grained NDP Offloading
few basic blocks are responsible for most of LLC misses
→ offloading such basic blocks to NDP are enough to improve performance
Case Studies

Load Balance and Inter-Vault Communication on NDP

NDP Accelerators and Our Methodology
NDP accelerator is faster than compute-centric accelerator for Class 1a and 1b applications; slower for Class 2c
→ key observations hold for other NDP architectures

Different Core Models on NDP Architectures

Fine-Grained NDP Offloading
**Goal:** evaluate compute-centric versus NDP accelerators

---

NDP Accelerators and Our Methodology

- Goal: evaluate compute-centric versus NDP accelerators

The performance of NDP accelerators are in line with the characteristics of the memory bottleneck classes:

Our memory bottleneck classification can be applied to study other types of system configurations.

Case Studies

Load Balance and Inter-Vault Communication on NDP

portion of the memory requests an NDP core issues go to remote vaults
→ increases the memory access latency for the NDP core

NDP Accelerators and Our Methodology

NDP accelerator is faster than compute-centric accelerator for Class 1a and 1b applications; slower for Class 2c

→ key observations hold for other NDP architectures

Different Core Models on NDP Architectures

using in-order cores limits performance of some applications
→ static instruction scheduling cannot exploit memory parallelism

Fine-Grained NDP Offloading

few basic blocks are responsible for most of LLC misses
→ offloading such basic blocks to NDP are enough to improve performance
Load Balance and Inter-Vault Communication on NDP

NDP core issues go to remote vaults → increases the memory access latency for the NDP core.

NDP accelerators are faster than compute-centric accelerators for Class 1a and 1b applications; slower for Class 2c applications → key observations hold for other NDP architectures.

Different Core Models on NDP Architectures

Using inorder cores limits performance of some applications → static instruction scheduling cannot exploit memory parallelism.

Fine-Grained NDP Offloading

Few basic blocks are responsible for most of LLC misses → offloading such basic blocks to NDP are enough to improve performance.
DAMOV is Open-Source

- We open-source our benchmark suite and our toolchain

DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing.

The DAMOV benchmark suite is the first open-source benchmark suite for main memory data movement-related studies, based on our systematic characterization methodology. This suite consists of 144 functions representing different sources of data movement bottlenecks and can be used as a baseline benchmark set for future data-movement mitigation research. The applications in the DAMOV benchmark suite belong to popular benchmark suites, including BWA, Chai, Darknet, GASE, Hardware Effects, Hashjoin, HPCC, HPCG, Ligra, PARSEC, Parboil, PolyBench, Phoenix, Rodinia, SPLASH-2, STREAM.
DAMOV is Open-Source

• We open-source our benchmark suite and our toolchain

Get DAMOV at:

https://github.com/CMU-SAFARI/DAMOV

DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing.

The DAMOV benchmark suite is the first open-source benchmark suite for main memory data movement-related studies, based on our systematic characterization methodology. This suite consists of 144 functions representing different sources of data movement bottlenecks and can be used as a baseline benchmark set for future data-movement mitigation research. The applications in the DAMOV benchmark suite belong to popular benchmark suites, including BWA, Chai, Darknet, GASE, Hardware Effects, Hashjoin, HPCC, HPCG, Ligra, PARSEC, Parboil, PolyBench, Phoenix, Rodinia, SPLASH-2, STREAM.
**Problem:** Data movement is a major bottleneck in modern systems. However, it is **unclear** how to identify:

- different sources of data movement bottlenecks
- the most suitable mitigation technique (e.g., caching, prefetching, near-data processing) for a given data movement bottleneck

**Goals:**
1. Design a methodology to **identify** sources of data movement bottlenecks
2. Compare compute- and memory-centric data movement mitigation techniques

**Key Approach:** Perform a large-scale application characterization to identify **key metrics** that reveal the sources to data movement bottlenecks

**Key Contributions:**
- Experimental characterization of 77K functions across 345 applications
- A **methodology** to characterize applications based on data movement bottlenecks and their relation with different data movement mitigation techniques
- **DAMOV:** a **benchmark suite** with **144 functions** for data movement studies
- Four case-studies to highlight DAMOV’s applicability to open research problems

**Conclusion**

DAMOV: [https://github.com/CMU-SAFARI/DAMOV](https://github.com/CMU-SAFARI/DAMOV)
DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

Geraldo F. Oliveira
Juan Gómez-Luna  Lois Orosa  Saugata Ghose
Nandita Vijaykumar  Ivan Fernandez  Mohammad Sadrosadati
Onur Mutlu

SAFARI
DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV Benchmark Suite and DAMOV-SIM Tutorial
Outline

• Downloading and Installing

• Compiling the Workloads

• Installing DAMOV-SIM

• Generating Configuration Files

• Executing Simulations and Collecting Statistics
Outline

• Downloading and Installing

• Compiling the Workloads

• Installing DAMOV-SIM

• Generating Configuration Files

• Executing Simulations and Collecting Statistics
Downloading and Installing

**Step 1:** Clone DAMOV’s GitHub repository

```
$ git clone https://github.com/CMU-SAFARI/DAMOV.git
$ cd DAMOV/
$ ls
get_workloads.sh  LICENSE  README.md  simulator
```

**Step 2:** Download the workloads

```
$ ./get_workloads.sh
Downloading DAMOV workloads -- Part 1
Cloning into 'damovworkloadspart1'...
...

bwa  chai-cpu  Darknet  GASE-master  hardware-effects  hpcc  hpcg
ligra  multicore-hashjoins-0.1  parboil  parsec-3.0  phoenix
PolyBench-ACC  rodinia_3.1  STREAM  zsim_hooks.h
```
Outline

• Downloading and Installing

• Compiling the Workloads

• Installing DAMOV-SIM

• Generating Configuration Files

• Executing Simulations and Collecting Statistics
Compiling the Workloads

- **Note 1:** Each workload might have its own set of dependencies

- **Step 3:** Compile applications with `compile.py`

  ```
  $ cd workloads/STREAM/
  $ python compile.py
  $ ls
  stream_add  stream.c  stream_copy  stream_scale  stream_triad
  compile.py
  ``

- **Note 2:** `compile.py` generates a single binary per memory-bound function
Outline

• Downloading and Installing

• Compiling the Workloads

• **Installing DAMOV-SIM**

• Generating Configuration Files

• Executing Simulations and Collecting Statistics
### Installing DAMOV-SIM

**Step 4: Install DAMOV-SIM**

```bash
$ cd simulator/
$ sudo ./scripts/setup.sh
$ ./scripts/compile.sh
...
```

`scons: done building targets.`

**To execute a simulation:**

```bash
$ ./build/opt/zsim configuration_file
```
Outline

• Downloading and Installing

• Compiling the Workloads

• Installing DAMOV-SIM

• Generating Configuration Files

• Executing Simulations and Collecting Statistics
• DAMOV-SIM simulates different system configurations
  - *Host*: a host CPU with private L1/L2 and shared L3 caches → **fixed LLC size**
  - *Host with prefetcher*: same as *Host* with a stream prefetcher
  - *Host NUCA*: a host CPU with private L1/L2 and shared L3 caches organized in a 2D mesh network → **LLC size increases as a factor of the core count**
  - *NDP*: an NDP system with private L1 cache

• We provide template configuration files under `simulator/templates`

<table>
<thead>
<tr>
<th></th>
<th>in-order</th>
<th>out-out-order</th>
<th>accelerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>template_host_inorder.cfg</td>
<td>template_host_ooo.cfg</td>
<td>template_host_accelerator.cfg</td>
</tr>
<tr>
<td>Host with pref</td>
<td>template_host_prefetch_inorder.cfg</td>
<td>template_host_prefetch_ooo.cfg</td>
<td>template_host_prefetch_accelerator.cfg</td>
</tr>
<tr>
<td>Host with NUCA</td>
<td>template_host_nuca_inorder.cfg</td>
<td>template_host_nuca.cfg</td>
<td>N/A</td>
</tr>
<tr>
<td>NDP</td>
<td>template_pim_inorder.cfg</td>
<td>template_pim_ooo.cfg</td>
<td>template_pim_accelerator.cfg</td>
</tr>
</tbody>
</table>
• **Step 5.1**: Create a command file
  - Command file format = \{benchmark, application, function, binary path, inputs\}

```bash
$ cd simulator/
$ cat command_files/stream_cf

stream Add Add PIM_ROOT/STREAM/stream_add THREADS
stream Copy Copy PIM_ROOT/STREAM/stream_copy THREADS
stream Scale Scale PIM_ROOT/STREAM/stream_scale THREADS
stream Triad Triad PIM_ROOT/STREAM/stream_triad THREADS
```
**Step 5.2:** Generate configuration files using `generate_config_files.py`.

```bash
$ python scripts/generate_config_files.py command_files/stream_cfg
$ ls config_files/
host_accelerator  host_inorder  host_ooo  pim_accelerator
pim_inorder  pim_ooo

$ ls config_files/host_ooo/
no_prefetch  prefetch

$ ls config_files/host_ooo/no_prefetch/
stream

$ ls config_files/host_ooo/no_prefetch/stream
1  16  256  4  64

$ ls config_files/host_ooo/no_prefetch/stream/4/
Add_Add.cfg  Copy_Copy.cfg  Scale_Scale.cfg  Triad_Triad.cfg
```
• **Step 5.3: Verify the configuration file**

```bash
$ cat config_files/host_ooo/no_prefetch/stream/4/Add_Add.cfg

// This system is similar to a 6-core, 2.4GHz Westmere with 10 Niagara-like cores attached to the L3
sys = {
    lineSize = 64;
    frequency = 2400;

    cores = {
        core = {
            type = "000";
            cores = 4;
            icache = "l1i";
            dcache = "l1d";
        }
    }
...}
```

**type = \{000, Timing, Accelerator\}**
Generating Configuration Files (4/4)

• Step 5.3: Verify the configuration file

```
$ cat config_files/host_ooo/no_prefetch/stream/4/Add/Add.cfg
...
caches = {
  l1d = {
    caches = 4;
    size = 32768;
    array = {
      type = "SetAssoc";
      ways = 8;
    };
    latency = 4;
  };
  l1i = { ... };
  l2 = { ... };
  l3 = { ... };
  mem = {
    type = "Ramulator";
    ramulatorConfig = "ramulator-configs/HMC-config.cfg";
    latency = 1;
  };
};
...
```

- Define cache size and organization
- Use Ramulator as the memory controller, and HMC as the main memory
• **Step 5.3:** Verify the configuration file

```bash
$ cat config_files/host_ooo/no_prefetch/stream/4/Add_Add.cfg
...
sim = {
pimMode = false;
stats = "zsim_stats/host_ooo/no_prefetch/4/stream_Add_Add";
phaseLength = 1000;
maxOffloadInstrs = 1000000000L;
maxTotalInstrs = 1000000000L;
...}
};

process0 = {
command = "'/home/safari/DAMOV/workloads//STREAM/stream_add 4";
startFastForwarded = True;
};
```

- `pimMode` enables host (\(pimMode = false\)) or NDP (\(pimMode = true\)) execution

- Simulation termination condition

- Command that will be simulated
Outline

• Downloading and Installing

• Compiling the Workloads

• Installing DAMOV-SIM

• Generating Configuration Files

• Executing Simulations and Collecting Statistics
Executing a Simulation

- **Step 6**: Run STREAM Add in a host CPU with 4 cores

```
$ ./build/opt/zsim config_files/host_ooo/no_prefetch/stream/4/Add_Add.cfg
```

- **Step 7**: Run STREAM Add in an NDP system with 4 cores

```
$ ./build/opt/zsim config_files/pim_ooo/stream/4/Add_Add.cfg
```

- **Current DAMOV-SIM limitation**: no support for concurrent execution on host and NDP cores
Collecting Statistics (1/2)

**Step 8.1**: Check the statistics stored under `zsim_stats/`

```bash
$ ls zsim_stats/host_ooo/no_prefetch/4/
stream_Add_Add.dramRequestsPerPhase  stream_Add_Add.out.cfg
stream_Add_Add.ramulator.stats  stream_Add_Add.zsim.out

$ ls zsim_stats/pim_ooo/4/
stream_Add_Add.dramRequestsPerPhase  stream_Add_Add.out.cfg
stream_Add_Add.ramulator.stats  stream_Add_Add.zsim.out
```
Step 8.2: Filter key metrics to compare host and NDP execution using `get_stats_per_app.py`.

```
$ python scripts/get_stats_per_app.py
zsim_stats/host_ooo/no_prefetch/4/stream_Add_Add.zsim.out

------------------
Summary
------------------
Instructions: 1000000939
Cycles: 450320568
**IPC: 2.22064238247**
L3 Miss Rate (%): 99.9992276064
L2 Miss Rate (%): 100.0
L1 Miss Rate (%): 73.5630799769
L3 MPKI: 23.433444996
LFMR: 0.999993129532

$ python scripts/get_stats_per_app.py
zsim_stats/pim_ooo/4/stream_Add_Add.zsim.out

------------------
Summary
------------------
Instructions: 1000005774
Cycles: 280045288
**IPC: 3.57087162988**
```

NDP speedup over CPU = 3.57/2.22 = 1.6x
DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV Benchmark Suite and DAMOV-SIM Tutorial

SAFARI