AIFM: High-Performance, Application-Integrated Far Memory

Zain (Zhenyuan) Ruan*  Malte Schwarzkopf†  Marcos K. Aguilera ‡  Adam Belay*

*MIT CSAIL  †Brown University  ‡VMware Research
In-Memory Applications

Data Analytics
- pandas

Web Caching
- redis

Database
- VoltDB

Graph Processing
- Powergraph
Memory Is Inelastic

➢ Limited by the server physical boundary.
Memory Is Inelastic

• Limited by the server physical boundary.

➢ Applications cannot overcommit memory.

Opening a 20GB file for analysis with pandas

I am currently trying to open a file with pandas and python for machine learning purposes it would be ideal for me to have them all in a DataFrame. My RAM is 32 GB. I keep getting memory errors.
Memory Is Inelastic

- Limited by the server physical boundary.
- Applications cannot overcommit memory.

Expensive solution: overprovision memory for peak usage.

Opening a 20GB file for analysis with pandas

I am currently trying to open a file with pandas and python for machine learning purposes it would be ideal for me to have them all in a DataFrame. My RAM is 32 GB. I keep getting memory errors.
Trending Solution: Far Memory

➢ Leverage the idle memory of remote servers (with fast network).
Existing Far-Memory Systems Perform Poorly

➢ Real-world Data Analytics from Kaggle.

Normalized Performance

ideal
Existing Far-Memory Systems Perform Poorly

- Real-world Data Analytics from Kaggle.
  - Provision 25% of working set in local mem

![Normalized Performance Graph]

- Ideal: 70% of performance wasted
  - state-of-the-art
Existing Far-Memory Systems Perform Poorly

- Real-world Data Analytics from Kaggle.
  - Provision 25% of working set in local mem.

Goal: reclaim the wasted performance.
Existing Far-Memory Systems Perform Poorly

- Real-world Data Analytics from Kaggle.
  - Provision 25% of working set in local mem.
- Goal: reclaim the wasted performance.

![Diagram showing 70% of performance wasted and comparison with ideal]

- 70% of performance wasted
- State-of-the-art
- AIFM (this work)
Existing Far-Memory Systems Perform Poorly

• Real-world Data Analytics from Kaggle.
  • Provision 25% of working set in local mem.

➢ Goal: reclaim the wasted performance.
Why Do Existing Systems Waste Performance?

• Problem: based on **OS paging**.
  – Semantic gap.
  – High kernel overheads.
Challenge 1: Semantic Gap

➢ Page granularity → R/W amplification.
Challenge 1: Semantic Gap

- Page granularity → **R/W amplification.**

OS  ➔  Page

App  ➔  Page
Challenge 1: Semantic Gap

- Page granularity $\rightarrow$ **R/W amplification.**

  - OS lacks app knowledge $\rightarrow$ **hard to prefetch, etc.**
Challenge 1: Semantic Gap

• Page granularity → R/W amplification.

➤ OS lacks app knowledge → hard to prefetch, etc.

A sequence of random memory accesses.

App

OS
Challenge 2: High Kernel Overheads
Challenge 2: High Kernel Overheads

➢ Expensive page faults.

![Diagram showing interaction between APP, Remote Object, Kernel, and Page Fault Handler]

1. APP
2. 1 μs
3. Page Fault Handler (8 μs)
Challenge 2: High Kernel Overheads

- **Expensive page faults.**
  - Busy Polling for in-kernel net I/O → **burn CPU cycles.**

![Diagram]

1. APP
2. Remote Object
3. Page Fault Handler (8 μs)
4. Net (6 μs)
Design Space

Transparency

Existing OS paging systems

Perf.
Design Space

Manually manage objects with RDMA

Existing OS paging systems
Design Space

- Manually manage objects with RDMA
- AIFM (this work)
- Existing OS paging systems
AIFM’s Design Overview

Key idea: swap memory using a userspace runtime.
AIFM’s Design Overview

Key idea: swap memory using a userspace runtime.

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Semantic gap</td>
<td>Remoteable Data structure library</td>
</tr>
<tr>
<td>(Amplification, Hard to prefetch)</td>
<td></td>
</tr>
<tr>
<td>2. Kernel overheads</td>
<td>Userspace runtime</td>
</tr>
<tr>
<td>(page faults, busy poll for net I/O)</td>
<td></td>
</tr>
<tr>
<td>3. Impact of Memory Reclamation</td>
<td>Pauseless evacuator</td>
</tr>
<tr>
<td>(pause app threads)</td>
<td></td>
</tr>
<tr>
<td>4. network BW &lt; DRAM BW</td>
<td>Remote Agent</td>
</tr>
</tbody>
</table>
AIFM in Action

App User- Level Thread 0

Local Memory

Far Memory
1. Remoteable Data Structure Library

➢ Solved challenge: semantic gap.

App User-Level Thread 0 ➞ Remoteable Data Structure

Local Memory

Far Memory
1. Remoteable Data Structure Library

➢ Solved challenge: semantic gap.

Local Memory

Far Memory
1. Remoteable Data Structure Library

➢ Solved challenge: semantic gap.

Local Memory

Far Memory
1. Remoteable Data Structure Library

➢ Solved challenge: semantic gap.

App User-Level Thread 0 → library API → Remoteable Data Structure

- App Semantics
- Prefetcher

Local Memory

Far Memory

Obj 0

Obj 1

Ptr 0

 Ptr 1
2. Userspace Runtime

➢ Solved challenge: kernel overheads.
2. Userspace Runtime

- Solved challenge: kernel overheads.

![Diagram showing the relationship between App User-Level threads, library API, Remoteable Data Structure, and pointers to objects in local and far memory.]

Local Memory

Far Memory
2. Userspace Runtime

- Solved challenge: kernel overheads.
3. Pauseless Evacuator

➢ Solved challenge: impact of memory reclamation.

Local Memory *(close to full)*

Far Memory
3. Pauseless Evacuator

➢ Solved challenge: performance impact of memory reclamation.

Local Memory (close to full)

Far Memory
3. Pauseless Evacuator

- Solved challenge: impact of memory reclamation.

Diagram:
- App User-Level Thread 0
  - Yield
  - library API
- App User-Level Thread 1
  - Yield
- Remoteable Data Structure
  - App Semantics
  - Prefetcher
  -.Ptr 0 → Obj 0
  -.Ptr 1 → Obj 1
  -... 
  -.Ptr N → Obj N
- Local Memory
- Far Memory
- Pauseless Evacuator
3. Pauseless Evacuator

➢ Solved challenge: impact of memory reclamation.

Remoteable Data Structure

- App Semantics
- Prefetcher

App User-Level Thread 0

- library API
- Yield

App User-Level Thread 1

Local Memory

Far Memory

Obj 0

- Ptr 0
- Obj 1
- Ptr 1
- ... (Ptr N)
- Obj N

Pauseless Evacuator

Evacuate

Evacuate
3. Pauseless Evacuator

- Solved challenge: impact of memory reclamation.

```plaintext
Local Memory

App User-Level Thread 0

library API

Yield

App User-Level Thread 1

Remoteable Data Structure

App Semantics

Prefetcher

Ptr 0

Obj 0

Ptr 1

...

Ptr N

Pauseless Evacuator

Far Memory

Obj 1

...

Obj N
```
4. Remote Agent

- Solved challenge: network BW < DRAM BW.
4. Remote Agent

➢ Solved challenge: network BW < DRAM BW.
4. Remote Agent

➢ Solved challenge: network BW < DRAM BW.

- App User-Level Thread 0
  - library API
  - yield
- App User-Level Thread 1
  - yield

Local Memory
- e.g., Copy Obj 1
- Remote Agent

Far Memory
- Remote Agent
  - Obj 0
  - Obj 1
  - Obj N
  - Pauseless Evacuator

Remoteable Data Structure
- App Semantics
- Prefetcher
- Ptr 0
- Ptr 1
- ... 
- Ptr N

Copy
Sample Code

```cpp
std::unordered_map<key_t, int> hashtable;
std::array<LargeData> arr;

LargeData foo(std::list<key_t> &keys_list) {
    int sum = 0;
    for (auto key : keys_list) {
        sum += hashtable.at(key);
    }

    LargeData ret = arr.at(sum);
    return ret;
}
```
Sample Code

```cpp
RemHashTable<key_t, int> hashtable;
RemArray<LargeData> arr;

LargeData foo(RemList<key_t> &keys_list) {
    int sum = 0;
    for (auto key : keys_list) {
        sum += hashtable.at(key);
    }

    LargeData ret = arr.at(sum);
    return ret;
}
```
Sample Code

```cpp
RemHashTable<key_t, int> hashtable;
RemArray<LargeData> arr;

LargeData foo(RemList<key_t> &keys_list) {
    int sum = 0;
    for (auto key : keys_list) {
        DerefScope scope;
        sum += hashtable.at(key, scope);
    }
    DerefScope scope;
    LargeData ret = arr.at(sum, scope);
    return ret;
}

Ensure the accessed objects will not be moved by the evacuator.
```
Sample Code

RemHashTable<key_t, int> hashtable;
RemArray<LargeData> arr;

LargeData foo(RemList<key_t> &keys_list) {
  int sum = 0;
  for (auto key : keys_list) {
    DerefScope scope;
    sum += hashtable.at(key, scope);
  }
  DerefScope scope;
  LargeData ret = arr.at(/*don’t cache*/ true>(sum, scope);
  return ret;
}
Sample Code

```cpp
RemHashTable<key_t, int> hashtable;
RemArray<LargeData> arr;

LargeData foo(RemList<key_t> &keys_list) {
    int sum = 0;
    for (auto key : keys_list) {
        DerefScope scope;
        sum += hashtable.at(key, scope);
    }
    DerefScope scope;
    LargeData ret = arr.at(/*don’t cache*/ true)(sum, scope);
    return ret;
}
```

Prefetch list data.
Sample Code

```cpp
RemHashTable<key_t, int> hashtable;
RemArray<LargeData> arr;

LargeData foo(RemList<key_t> &keys_list) {
    int sum = 0;
    for (auto key : keys_list) {
        DerefScope scope;
        sum += hashtable.at(key, scope);
    }
    DerefScope scope;
    LargeData ret = arr.at</*don’t cache*/ true>(sum, scope);
    return ret;
}
```
Sample Code

```cpp
RemHashTable<key_t, int> hashtable;
RemArray<LargeData> arr;

LargeData foo(RemList<key_t> &keys_list) {
    int sum = 0;
    for (auto key : keys_list) {
        DerefScope scope;
        sum += hashtable.at(key, scope);
    }
    DerefScope scope;
    LargeData ret = arr.at(/*don’t cache*/ true>(sum, scope);
    return ret;
}
```
Implementation

➢ Implemented 6 data structures.
  • Array, List, Hashtable, Vector, Stack, and Queue.
Implementation

• Implemented 6 data structures.
  • Array, List, Hashtable, Vector, Stack, and Queue.

➤ Runtime is built on top of Shenango [NSDI’ 19].
Implementation

- Implemented 6 data structures.
  - Array, List, Hashtable, Vector, Stack, and Queue.
- Runtime is built on top of Shenango [NSDI’ 19].
  - TCP far-memory backend.
Implementation

• Implemented 6 data structures.
  • Array, List, Hashtable, Vector, Stack, and Queue.
• Runtime is built on top of Shenango [NSDI’ 19].
• TCP far-memory backend.

➢ LoC: 6.5K (runtime) + 5.5K (data structures) + 0.8K (Shenango)
Evaluation

➢ Setup: 1 compute server + 1 far memory server, 25 GbE.
Evaluation

• Setup: 1 compute server + 1 far memory server, 25 GbE.
• How does AIFM
  ➢ ... perform on applications with different compute intensities?
Evaluation

• Setup: 1 compute server + 1 far memory server, 25 GbE.
• How does AIFM
  • ... perform on applications with different compute intensities?
  ➢ ... compare to the local-only (ideal) system?
Evaluation

• Setup: 1 compute server + 1 far memory server, 25 GbE.
• How does AIFM
  • ... perform on applications with different compute intensities?
  • ... compare to the local-only (ideal) system?
  ➢ ... compare to the state-of-the-art paging system, Fastswap [EuroSys’ 20]?
Performance on Different Compute Intensities

![Graph showing normalized performance vs. microsecunds of compute per far memory access.]
Performance on Different Compute Intensities

![Graph showing normalized performance versus microseconds of compute per far memory access. The line indicates that Fastswap has converged to an ideal state of 1 at approximately 50 μs.](image)

**Legend**
- **Fastswap**

**Graph Details**
- **Normalized Performance**
- **Microseconds of compute per far memory access**

*Converged to 1 at ~50 μs*
Performance on Different Compute Intensities

AIFM hides far memory latency with moderate compute.

Converged to 1 at ~50 μs
AIFM hides far memory latency with moderate compute.
Performance on Different Compute Intensities

- **Fastswap**
- **AIFM**

**13X in Synthetic Web Frontend**
Performance on Different Compute Intensities

- **Fastswap**
- **AIFM**

理想的性能

在合成网络前端有13倍的提升

**Microseconds of compute per far memory access**

**Normalized Performance**
NYC Taxi Analysis (C++ DataFrame)

- DataFrame: data analytical framework, similar to Python Pandas.
NYC Taxi Analysis (C++ DataFrame)

- DataFrame: data analytical framework, similar to Python Pandas.
- Real Kaggle workload
  - Working set size = 31 GB.
  - Modify 1.4K LoC (out of 24.3K LoC), five person-days.
NYC Taxi Analysis (C++ DataFrame)

• DataFrame: data analytical framework, similar to Python Pandas.
• Real Kaggle workload
  • Working set size = 31 GB.
  • Modify 1.4K LoC (out of 24.3K LoC), five person-days.
➢ Relatively low compute intensity → Unable to hide far-mem latency.
NYC Taxi Analysis (C++ DataFrame)

• DataFrame: data analytical framework, similar to Python Pandas.
• Real Kaggle workload
  • Working set size = 31 GB.
  • Modify 1.4K LoC (out of 24.3K LoC), five person-days.
• Relatively low compute intensity → Unable to hide far-mem latency.
  ➢ Keep complex operations local and **offload** very light operations.
  • Significantly reduces expensive data transfer over network.
NYC Taxi Analysis (C++ DataFrame)
NYC Taxi Analysis (C++ DataFrame)

![Normalized Performance vs. Local Memory Ratio](image)

**Legend:**
- Fastswap
- AIFM

**Ideal** performance marked at 100% local memory ratio.
NYC Taxi Analysis (C++ DataFrame)

AIFM achieves near-ideal performance with small local memory.
NYC Taxi Analysis (C++ DataFrame)

AIFM achieves near-ideal performance with small local memory.
NYC Taxi Analysis (C++ DataFrame)

AIFM achieves near-ideal performance with small local memory.
Other Experiments

- Synthetic web frontend: up to **13X end-to-end** speedup.
- Data structures microbenchmarks: up to **61X** speedup.
- Design Drill-Down.

Read our paper for details.
Related Work

• OS-paging systems.
  • Fastswap [EuroSys’ 20], Leap [ATC’ 20]

• Distributed shared memory.
  • Treadmarks [IEEE Computer’ 96]

• Garbage collection (GC).
Conclusion

AIFM: Application-Integrated Far Memory.
Conclusion

• AIFM: Application-Integrated Far Memory.
  ➢ Key idea: swap memory using a userspace runtime.
Conclusion

• AIFM: Application-Integrated Far Memory.
• Key idea: swap memory using a userspace runtime.
  ➢ Data Structure Library: captures application semantics.
Conclusion

• AIFM: Application-Integrated Far Memory.
• Key idea: swap memory using a userspace runtime.
  • Data Structure Library: captures application semantics.
  ➢ Userspace Runtime: efficiently manages objects and memory.
Conclusion

• AIFM: Application-Integrated Far Memory.
• Key idea: swap memory using a userspace runtime.
  • Data Structure Library: captures application semantics.
  • Userspace Runtime: efficiently manages objects and memory.

➢ Achieves 13X end-to-end speedup over Fastswap.
Conclusion

• AIFM: Application-Integrated Far Memory.
• Key idea: swap memory using a userspace runtime.
  • Data Structure Library: captures application semantics.
  • Userspace Runtime: efficiently manages objects and memory.
• Achieves 13X end-to-end speedup over Fastswap.
➢ Code released at https://github.com/AIFM-sys/AIFM

Please send your questions to us
zainruan@csail.mit.edu