Data-Center Resource Disaggregation

Yiying Zhang
Outline

• Disaggregation Overview

• OS for disaggregation: LegoOS [OSDI’18]

• Hardware for disaggregation: Clio [ASPLOS’22]

• PL for disaggregation: Mira [SOSP’23]

• Serverless on disaggregation: next time

• Project progress report due 2/15, quiz 2 on 2/14
Cloud Has Always Been Very Serverful

AWS Services

Deployment & Management

Mobile Services

Application Services

Enterprise Applications

Administration & Security

Analytics

Foundation Services

Network Computing

OS/Hypervisor

Very Serverful
**Problem:** Allocating resources from server boxes

- Multi-dimensional bin packing is inefficient
- Cannot independently scale different types of resources

![Diagram showing resource allocation across three machines](image)

- Waiting Job

~20%-60% CPU/Memory usage *

*Stats from Google and Alibaba datacenter traces*
Incorporating Heterogeneous Hardware & Managing

More Problem: Incorporating Heterogeneous Hardware
Making applications smaller + accessing resource in remote servers

Current:

- Machine1
- Machine2
- Machine3

Not solving HW infra problems
Hard to scale independently
Hard to manage
Room for performance improvement

Microservice
Hardware Resource Disaggregation:

Breaking monolithic servers into distributed, network-attached hardware components
Allocate resources precisely as needed, and when needed

Manage resources as independent pools

- Rightsizing
- Scale freely
- Low $ cost
- Independent (easy) mgmt
A New Paradigm: A Truly Serverless Cloud

distributed and *(logically)*
disaggregated applications

distributed and *(functionally)*
disaggregated system software

distributed and *(heterogeneous)*
disaggregated hardware
An End-to-End Server-less Datacenter

• Application: a serverless approach with Scad [under submission]
• OS: a splitkernel approach with LegoOS [OSDI’18]
• Networking: a consolidation approach with SuperNIC [in preparation]
• Hardware: a hardware-software co-design approach with Clio [ASPLOS’22]
Outline

- OS for disaggregation: LegoOS [OSDI’18]
- Hardware for disaggregation: Clio [ASPLOS’22]
- PL for disaggregation: Mira [SOSP’23]
- Serverless on disaggregation: next time
An End-to-End Server-less Datacenter

- Application: a serverless approach with Scad [under submission]
- OS: a splitkernel approach with LegoOS [OSDI’18]
- Networking: a consolidation approach with SuperNIC [in preparation]
- Hardware: a hardware-software co-design approach with Clio [ASPLOS’22]
How to manage disaggregated hardware?
Hard to Support Disaggregation with Existing OSes

Assume local accesses to resources
Don’t manage distributed resources
No fine-grained failure handling
When hardware is disaggregated

The OS should be also
OS

- Process Mgmt
- Virtual Memory System
- File & Storage System
- Network
Faster
Better
Customizable

Process Mgmt
Network

Virtual Memory System
Network

File & Storage System
Network

Management
The Splitkernel Architecture

Split OS functions into monitors

Run each monitor at h/w device

Messaging across components

Applications responsible for inter-component coherence

Distributed resource management

Failure handling

- Processor Monitor (CPU)
- GPU Monitor (GPU)
- New Hardware
- Messaging across components
- Fast network (Ethernet, InfiniBand, Optics)
- Memory Monitor
- NVM Monitor
- HDD Monitor
- SSD Monitor
- Applications responsible for inter-component coherence
- Distributed resource management
- Failure handling
LegoOS

A New Distributed OS for Disaggregated Hardware
How should *Lego*OS appear to users?

As a giant machine?

As a set of virtual nodes (*vNodes*)
Each can run a set of (*Linux-compatible*) processes

As a set of hardware devices?
Protection domain

Distributed across multiple hardware components

Fine-grained, on-demand resource allocation
LegoOS Design

1. Clean separation of OS and hardware functionalities

2. Build monitors with hardware constraints

3. RDMA-based message passing for both kernel and applications

4. Two-level distributed resource management

5. Memory failure tolerance through replication and checkpointing

Shan et al., “LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation,” OSDI’18
Separate Processor and Memory
Separate Processor and Memory

Disaggregate DRAM
Separate Processor and Memory

Separate and move hardware units to memory component
Separate Processor and Memory

Virtual Memory System

Processor
- CPU
- CPU
- Last-Level Cache

Network

Memory
- TLB
- MemCtrl
- DRAM
- PT
Separate Processor and Memory

Separate and move virtual memory system to memory component
Separate Processor and Memory

$pComponent$

Only see virtual memory addr
All levels of cache are virtual cache

Every LLC miss now takes one RTT!

$mComponent$

manage virtual and physical memory
What about network delay?

Throughput on par with DRAM

Latency is still ~10x worse

Time to rethink memory hierarchy

* Data points from DDR standard and EthernetAlliance
Separate Performance and Capacity

Improved performance

Another level of Extended Cache, or ExCache

Small (few GBs) but high bandwidth (HBM)

Hardware-managed hit path, OS for misses
Separate *Performance* and *Capacity*

**pComponent**
- CPU
- Last-Level Cache
- HBM ExCache

**mComponent**
- TLB
- MemCtrl
- Virtual Memory System
- PT
- DRAM

**Improved performance**

**Scaled and managed independently**

**Enlarged capacity**
Two-Level Distributed Resource Management

Global Process Management

Global Memory Management

Global Storage Management

Global Resource Manager

Network

Process Monitor

Process Monitor

Memory Monitor

Memory Monitor

Storage Monitor

Storage Monitor

Coarse-grain

Fine-grain
User Process Virtual Address Space

fix-sized, **coarse-grain** virtual region (vRegion) (e.g., 1GB)

Home mComponent oversees vma allocation
- Knows vRegions available space and owners

GMM assigns vRegions to mComponents
- which become the **owners**

Owner mComponent of a vRegion
- **Fine-grain** virtual memory allocation
- **On-demand** physical memory allocation

**brk 1.5GB**
**brk 100MB**
**write 500MB at virt-addr 0**
Implementation and Evaluation

Implemented from scratch, drivers ported from Linux

Unmodified TensorFlow running ImageNet and Cifar-10

Benefits of OS+hardware-level disaggregation!

- Single-machine Linux with unlimited memory (optimal perf)
- Linux swap to local ramdisk
- Swapping to remote memory (InfiniSwap [NSDI’17])

Systems in comparison

- 1.3x - 2.1x perf improvement compared to swapping to remote or local memory
- Improves perf/$ by up to 1.4x over single-machine Linux

$ cost saving of ~50%, estimated by hardware and energy required to meet the same level of app resource reqs

* Similar results on ImageNet (total memory 70GB)
LegoOS Summary

Separate and distribute core OS functionalities

Customize for applications and hardware

Co-design OS, dist sys, and hardware

Enable/encourage future h/w and OS research

Shan et al., “LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation,” OSDI’18 best paper
Outline

• OS for disaggregation: LegoOS [OSDI’18]
• Hardware for disaggregation: Clio [ASPLOS’22]
• PL for disaggregation: Mira [SOSP’23]
• Serverless on disaggregation: next time
Existing Disaggregated Memory Systems

- LegoOS [OSDI ’18]
- FastSwap [EuroSys ’20]
- AIFM [OSDI ’20]
- Semeru [OSDI ’20]

5.1 Implementation
We implemented Kona as a C library that interposes on an application’s memory allocation handling page faults [80]. The Kona server and were implemented in 3,136 lines of code (LoC). The Kona server and were implemented in 3,136 lines of code (LoC). The Kona server and were implemented in 3,136 lines of code (LoC). The Kona server and were implemented in 3,136 lines of code (LoC). The Kona server and were implemented in 3,136 lines of code (LoC). The Kona server and were implemented in 3,136 lines of code (LoC).

5.1 Hardware Emulation
Since there is no real resource disaggregation hardware, we emulate aggregated hardware components using commodity servers by limiting their internal hardware usages. For example, to emulate controllers for mComponents and sComponents, we limit the usable cores of a server to two. To emulate pComponents, we limit the amount of usable main memory of a server and configure it as LegoOS software-managed ExCache.

All existing works use **server** to **Build/Emulate** disaggregated memory devices
How about real hardware?
Disaggregated Memory Hardware

Features
- Standalone
- Host memory
- Directly connect to network
- Shared by applications

Desired goals
- High throughput
- Low avg and tail latency
- Scalability and capacity
- Low cost
- Easy to use and versatile
What we built: **Clio** [ASPLOS’22]

a **hardware-based disaggregated memory** system that virtualizes, protects, and manages disaggregated memory at **standalone memory nodes**
virtual memory interface

remote_alloc(pid, size)
read/write(pid, VA)
key-value & other high-level API

Data Path
Virtualize
Protect
Multiplex

Offload Path
Computation offload
Extended APIs

Control Path
Allocation
Distributed Support
Metadata

Compute Node

Memory Node (Clio Board)
Main Idea:

Eliminate *state* from hardware

“*state*”:

_Metadata stored on the memory node that need to be accessed or updated when processing requests._
Benefit of Removing State

- Minimizing state can **reduce cost, reduce tail latency**, and improve **scalability**

- Avoiding inter-request state can **make the pipeline smooth**
How to eliminate state from MN hardware

Overall Approach: Co-designing hardware, network, and software

1. Reduce state in disaggregated memory protocol
2. Move state to compute node
3. Remove state from critical path
4. Optimize hard-to-remove state to bounded size
How to eliminate state from MN hardware

Overall Approach: Co-designing hardware, network and software

1. Reduce state in disaggregated memory protocol
2. Move state to compute node
3. Remove state from critical path
4. Optimize hard-to-remove state to bounded size
Reduce state in disaggregated memory protocol >

Asymmetric Memory Request Protocol

**Observation:** accesses to MNs are always in the request-response style
Asymmetric Memory Request Protocol

**Observation:** accesses to MNs are always in the request-response style

Asymmetric RPC-style, connection-less network protocol
Reduce state in disaggregated memory protocol >

Network Ordering

One write with two packets: \( P1 \) \( P2 \)

App Process

Client Side Stack

Compute Node

Asymmetric

\( P1/P2 \) \( P2/P1 \)

RDMA Network

Per-Connection Metadata

Net Rx Buffer (reorder)

Net Tx Buffer

Congestion Control

Virtual Memory System

On-Board DRAM

Memory Node

- **Observation**: Memory requests can tolerate certain network reordering
• **Observation:** Memory requests can tolerate certain network reordering

Release networking ordering requirements
Move state to Compute Node >

Congestion and Flow Control

- **Observation**: CN knows the size of both requests and responses

Move congestion and in-cast control to CN side
Move state to Compute Node >
Handle Retry

- **Observation**: Network losses are rare and fully observed by CN
  Let CN side software handling retry
Move state to Compute Node >

Handle Retry

App Process

ClioLib (*congestion/in-cast ctrl, retry, …*)

Asymmetric
Allow reorder

ETH & PHY

Clio Network

Mainly only PHY & ETH!

On-Board DRAM

Virtual Memory System

Compute Node

Memory Node
How to eliminate state from MN hardware

1. Reduce state in system protocol
2. Move state to compute node
3. Remove state from critical path
4. Optimize hard-to-remove state to bounded size
Remove state from critical path >

Splitting Fast Path and Slow Path

- remote_alloc(pid, size)
- read/write (pid, va)

**Clio Net**

**Virtual Memory System**

Metadata requests
- Stateful, flexible, less strict latency

Data requests (Performance critical!)
- Strict latency and throughput

**Memory Node**

- **Observation**: Metadata and data requests have different state and performance requirements
Remove state from critical path >

Splitting Fast Path and Slow Path

remote_alloc(pid, size)
read/write (pid, va)

Splitting virtual memory system into fast path and slow path
Remove state from critical path >

Splitting Fast Path and Slow Path

Solution: Splitting virtual memory system into fast path and slow path
“eliminate state” summary

• Reduce state in system protocol: Disaggregated protocol, consistency model, …

• Move state to compute node: Congestion control, retry, dependency check, …

• Remove state from critical path: Hardware pagefault, memory region, …

• Optimize state to bounded size: Hash-based pagetable, atomic operations, …

Low Performance Overhead
Throughput ✓ Latency ✓ Tail Latency ✓ Low Cost ✓ Scalability ✓ Flexibility?
Extend computation offloading

App Process

remote_alloc(size)
read/write AP_VA

library (req retry, ordering)

Ethernet NIC

CN (server or device)

Network

remote_alloc
read/write
atomic_rd/wr/cas

ASIC (fast path)

Net

TLB

Page Fault Handler

async buffer

Virtual Mem Alloc

Phys Mem Alloc

ARM (slow path)

Memory

AP_VA

-> PA

MN (device)
Extend computation offloading

Flexibility

pointer_chase
key_value_get/put
...

remote_alloc
read/write
atomic_rd/wr/cas

App Process

remote_alloc(size)
read/write AP_VA

library (req retry, ordering)

Ethernet NIC

CN (server or device)

Network

FPGA (offload)

Extended API

Async buffer

ASIC (fast path)

Net

TLB

Page Fault Handler

Async buffer

ARM (slow path)

Virtual Mem Alloc

Phys Mem Alloc

Memory

PT

AP_VA -> PA

MN (device)

Flexibility

remote_alloc
read/write
atomic_rd/wr/cas

App Offload

Async buffer

App Offload

Ethernet NIC

Page Fault Handler

Async buffer

Virtual Mem Alloc

Phys Mem Alloc

Memory

PT

AP_VA -> PA

MN (device)
Implementation and Application

- Fast path and extended path implemented in hardware using SpinalHDL
- Prototype with Xilinx ZCU106 ARM-FPGA board

- Implemented five applications
  - Image compression
  - Multi-version object store
  - Key-value store
  - Pointer dereferencing
  - Data analytics operation

"Clio prototype on the Xilinx ZCU106 board"
Evaluation Results

Basic Performance

- **100Gbps** throughput, **2.8µs (avg) 3.2µs (p99)** latency
- Orders of magnitude **lower tail latency** than RDMA
- Outperforms Clover [ATC’20], LegoOS [OSDI’18], and HERD [SIGCOMM’14]
Evaluation Results

Concurrent Clients and Memory Size

- Clio provides bounded access time for data requests
- Clio scales well with concurrent clients and total memory size

Bounded Tail Latency even under scale
Clio Summary

• Real benefits of hardware resource disaggregation comes from real hardware
• Building OS functionaliites in hardware is feasible but needs new design
• The nature of disaggregation indicates new opportunities and challenges.
• Co-designing software and hardware systems is key in building real hardware.

**Clio** is a starting point for more real disaggregated hardware
Outline

- OS for disaggregation: LegoOS [OSDI’18]
- Hardware for disaggregation: Clio [ASPLOS’22]
- PL for disaggregation: Mira [SOSP’23]
- Serverless on disaggregation: next time
Existing Far Memory Systems

• How to **run applications with** far memory?

• Existing approaches in two broad categories:

1. New memory programming model:
   AIFM [OSDI22], Clio [ASPLOS22]

2. Transparent far memory:
   LegoOS [OSDI18], FastSwap [EuroSys21], MemLiner [OSDI22]
New Far Mem Programming Model

• Programs rewritten to access far memory with new APIs

  + Explicitly express access pattern
  - Require programmers’ effort
  - Cannot adapt to dynamic behavior

 Overhead on both human and runtime
Transparent Far Memory

- Transparetnly swap to far memory at runtime or OS layer

+ No program changes
- Unaware of application semantics
- No adaptation to different applications

Inefficient far memory operation
(read/write amplification, no or wrong prefetch, …)
How to build a transparent and efficient far memory system?
Solution: Behavior-Guided Far Memory via Program Analysis, Compiling, and Profiling
Mira: Behavior-Guided Far Memory

- **Program analysis** for capturing static behavior
- **Profiling** for capturing dynamic behavior
- **Compiler** for behavior-guided transparent far memory generation

- Program analysis for capturing static behavior: access sequence, locality...
- Profiling for capturing dynamic behavior: Object size, execution overhead...

Diagram:
- Unmodified Application
- Program Analysis
- Compiling
- Execute and Profile
- Mira Generated Program
  - Application on Far Memory
  - Configured Runtime System
Mira: Behavior-Guided Far Memory

- **Program analysis** for capturing static behavior
- **Profiling** for capturing dynamic behavior
- **Compiler** for behavior-guided transparent far memory generation
Mira: Behavior-Guided Far Memory

- Captures and utilizes a full view of **program behavior**
- Outperforms state-of-the-art far memory systems by 43% to 70%

* Data collected on the DataFrame application, with local memory size 20%
Challenge:

How to generate efficient far memory program based on program behaviors?
The Key to Far Memory Performance

• Far memory has **non-trivial overhead**

```c
x = arr[i];
```

~50ns
The Key to Far Memory Performance

- Far memory has **non-trivial overhead**
The Key to Far Memory Performance

- Far memory has **non-trivial overhead**
- **Accessing and caching** policies are key to far memory performance

Problem: Existing system use **single unified global cache**
Problem of a Unified, Global Cache

- Objects with different behaviors co-exists in a single program

```cpp
arr = vector<int>;
table = hash<int, int>;
for (i <- 0 to len(arr)){
    idx = arr[i];
    acc += table[idx];
}
```

Sequential, Accessed once, …
Random, No spatial locality, Non-predictable temporal locality …

Need to capture both behaviors
Problem of a Unified, Global Cache

- Objects with different behaviors **co-exists** in a single program
- Different behaviors require different **access and cache** configurations

```c
arr = vector<int>;
table = hash<int, int>;
for (i <- 0 to len(arr)){
    idx = arr[i];
    acc += table[idx];
}
```

**Optimal Access Policies**
- Batching
- No Batching
- Sequential Prefetch
- No program Prefetch

**Optimal Cache Config**
- Small size
- Large size
- Direct
- Full-assoc

Unified cache cannot optimize for all
Solution: Separate local cache into sections based on program behaviors
Solution: Separate Cache Sections

- Leveraging software-managed DRAM cache

- Each section customized for **one single** type of behavior

  With different \{cache size, cache structure, prefetch/eviction policy, \ldots\}

```cpp
arr = vector<int>
```

```
arr = hash<int, int>;
```

Cache Section 1

Cache Section 2
Using Separated Cache Sections

1. **Configure** each separated section according to its behavior
Using Separated Cache Sections

1. Configure each separated section according to its behavior

2. **Generate** accesses based on section configuration and behavior

```java
arr[i]
```

```java
For (batch <- arr) {
    prefetch(nxt_batch)
    ...
}
```
Iterative Cache Section Configuration

1st Iteration:
All data objects are placed into the generic swap section with profiling

```
Application
arr = malloc(...)
tab = malloc(...)
other = malloc(...)

func(tab);
func(arr, tab);
func(other);
```
Iterative Cache Section Configuration

2nd Iteration:

1. Profiling finds optimization targets: highest overhead objects and functions

```
arr = malloc(...)  
tab = malloc(...)  
other = malloc(...)  

func(tab);  
func(arr, tab);  
func(other)
```
Iterative Cache Section Configuration

2\textsuperscript{nd} Iteration:

2. Separate cache sections for selected objects

\begin{itemize}
  \item \texttt{arr} = \textit{malloc}(...)
  \item \texttt{tab} = \textit{malloc}(...)
  \item \texttt{other} = \textit{malloc}(...)
\end{itemize}

\texttt{func(tab)};
\texttt{func(arr, tab)};
\texttt{func(other)}
Iterative Cache Section Configuration

2nd Iteration:

3. Configure cache section based on static and dynamic behavior

Application

arr = malloc(...)
tab = malloc(...)
other = malloc(...)

func(tab);
func(arr, tab);
func(other)

Local DRAM Cache

Default swap section

Section for arr
(linesize = 1K, prefetch=seq, ...)

Static Analysis

Dynamic Behavior

Remote Memory Server

Space for other objects

Space for allocated arr
Iterative Cache Section Configuration

- **In later iterations**, Mira continues the optimization loop
- Until no optimization target is found or maximum iteration is reached
- Evaluated applications converge within 4 iterations
Implementation

- **Compiler** implemented on top of MLIR*
  - Implemented as two new dialects: `remotable` and `rmref`

- **Configurable runtime library** implemented in user space
  - With kernel-bypassing RDMA and `userfaultfd`

- LoC: 7.7K C++ (compiler), 12.1K C++ (runtime)

* MLIR = Multi-Level Intermediate Representation, https://mlir.llvm.org/
**Evaluation**

- Compared **Mira** with **FastSwap**[EuroSys 20] and **AIFM**[OSDI 20]

- On DataFrame, Mira achieves near-optimal performance

**DataFrame Performance**

- Compared to **AIFM**, 
  - *Adaptive code generation* reduces runtime overhead

- Compared to **FastSwap**, 
  - *Section separation* reduce the interference between accesses
  - *Analysis based prefetch* hide more latency
Comparing Typical Applications

- Dataframe reduces interference and could benefit from function offload
- GPT2-Inference has fixed access sequence and statically scheduled
- MCF section isolation and prefetch still benefits indirect access pattern
Summary of Mira

**Mira**: the first attempt to behavior-guided far-memory system

Achieves **application-aware** and **transparent** at the same time!

Program semantic should be captured and utilized in system layers
Concluding Discussion

• Pros and cons of resource disaggregation?

• Do you believe this can be the future?

• New developments in the space: CXL