MIT Libraries | DSpace@MIT

MIT Open Access Articles

Livia: Data-Centric Computing Throughout the Memory Hierarchy

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Lockerman, Elliot, Feldmann, Axel, Bakhshalipour, Mohammad, Stanescu, Alexandru, Gupta, Shashwat et al. 2020. "Livia: Data-Centric Computing Throughout the Memory Hierarchy." International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS.

As Published: 10.1145/3373376.3378497

Publisher: Association for Computing Machinery (ACM)

Persistent URL: https://hdl.handle.net/1721.1/143865.2

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of Use: Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.

Massachusetts Institute of Technology

Livia: Data-Centric Computing Throughout the Memory Hierarchy

Elliot Lockerman[1], Axel Feldmann[2*], Mohammad Bakhshalipour[1], Alexandru Stanescu[1], Shashwat Gupta[1], Daniel Sanchez[2], Nathan Beckmann[1]

[1] Carnegie Mellon University
{elockerm, astanesc, beckmann}@cs.cmu.edu
{mbakhsha, shashwag}@andrew.cmu.edu

[2] Massachusetts Institute of Technology
{axelf, sanchez}@csail.mit.edu

Abstract

In order to scale, future systems will need to dramatically reduce data movement. Data movement is expensive in current designs because *(i)* traditional memory hierarchies force computation to happen unnecessarily far away from data and *(ii)* processing-in-memory approaches fail to exploit locality.

We propose *Memory Services*, a flexible programming model that enables data-centric computing throughout the memory hierarchy. In Memory Services, applications express functionality as graphs of simple tasks, each task indicating the data it operates on. We design and evaluate *Livia*, a new system architecture for Memory Services that dynamically schedules tasks and data at the location in the memory hierarchy that minimizes overall data movement. Livia adds less than 3% area overhead to a tiled multicore and accelerates challenging irregular workloads by 1.3× to 2.4× while reducing dynamic energy by 1.2× to 4.7×.

CCS Concepts • **Computer systems organization → Processors and memory architectures**.

Keywords memory; cache; near-data processing.

ACM Reference Format:
Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, Nathan Beckmann. 2020. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3373376.3378497

* — This work was done while Axel Feldmann was at CMU.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378497

1 Introduction

Computer systems today are increasingly limited by data movement. Computation is already orders-of-magnitude cheaper than moving data, and the shift towards leaner and specialized cores [17, 22, 36, 39] is exacerbating these trends. Systems need new techniques that dramatically reduce data movement, as otherwise data movement will dominate system performance and energy going forward.

Why is data so far from compute? Conventional CPU-based systems reduce data movement via deep, multi-level cache hierarchies. This approach works well on programs that have hierarchical reuse patterns, where smaller cache levels filter most accesses to later levels. However, these systems *only process data on cores*, forcing data to traverse the full memory hierarchy before it can be processed. On such systems, programs whose data doesn't fit in small caches often spend nearly all their time shuffling data to and fro.

Since such compute-centric systems are often inefficient, prior work has proposed to do away with them and place cores close to memory instead. In these *near-data processing (NDP)* or *processing-in-memory (PIM)* designs [13, 14, 29, 72], cores enjoy fast, high-bandwidth access to nearby memory. PIM works well when programs have little reuse and when compute and data can be spatially distributed. However, eschewing a cache hierarchy makes PIM far less efficient on applications with significant locality and complicates several other issues, such as synchronization and coherence. In fact, prior work shows that for many applications, conventional cache hierarchies are far superior to PIM [5, 40, 90, 97].

Computing near data while exploiting locality: In this work, we propose the next logical step, which lies between these two extremes: reducing data movement by performing compute *throughout the memory hierarchy*—near caches large and small as well as near memory. This lets the system perform computation at the location in the memory hierarchy that minimizes data movement, synchronization, and cache pollution. Critically, this can mean moving computation to data *or* moving data to computation, and in some cases moving both.

Prior work has already shown that performing computation within the memory hierarchy is highly beneficial.

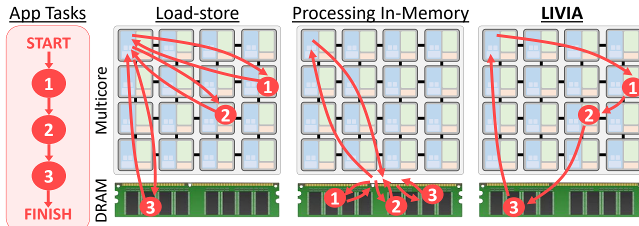


Fig. 1. Livia minimizes data movement by executing tasks at their "natural" location in the memory hierarchy.

GPUs [94] and multicores [93] perform atomic memory operations on shared caches to reduce synchronization costs, and caches can be repurposed to accelerate highly data-parallel problems [1], like deep learning [24] and automata processing [87]. However, prior techniques are limited to performing *fixed* operations at a *fixed* location in the cache hierarchy.

To benefit a wide swath of applications, data-centric systems must overcome these limitations. First, the memory hierarchy must be *fully programmable.* Applications should be able to easily extend the memory interface via a simple programming model. Second, the system must *decide where to perform computation*—not the application programmer! The best location to perform a given computation depends on many factors (e.g., data locality, cache size, etc.). It is very difficult for programmers to reason about these factors. Caches, not scratchpads, have become ubiquitous because they relieve programmers from the burden of placing data; data-centric systems must provide similar ease-of-use for placing computation.

Our approach: We overcome these limitations through a combination of software and hardware. First, we propose Memory Services, a flexible programming interface that facilitates computation throughout the memory hierarchy. Memory Services break application functionality into a graph of short, simple tasks. Each task is associated with a memory location that determines where it will execute in the memory hierarchy, and can spawn additional tasks to implement complex computations in a fork-join or continuation-passing fashion. Memory Services ask programmers *what* operations to perform, but not *where* to perform them.

Second, we present Livia,[1] an efficient architecture for Memory Services. Livia introduces specialized *Memory Service Elements* (MSEs) throughout the memory hierarchy. Each MSE consists of a *controller*, which schedules tasks and data in their best location in the memory hierarchy, and an *execution engine*, which executes the tasks themselves.

Fig. 1 illustrates how Memory Services reduce data movement over prior architectures. The left shows a chain of three dependent tasks which, together, implement a higher-level operation (e.g., a tree lookup; see Sec. 2). Suppose tasks ① and ② have good locality and task ③ does not. The baseline load-store architecture (left) executes tasks on the top-left core, and so must move all data to this core. Hence, though it caches data for ① and ② on-chip, it incurs many expensive round-trips that add significant data movement. PIM (middle) moves tasks closer to data but sacrifices locality in tasks ① and ②, incurring additional expensive DRAM accesses. In contrast, Livia (right) minimizes data movement by executing tasks in their natural location in the memory hierarchy, exploiting locality and eliminating unnecessary shuffling of data between tasks.

In this paper, we present Memory Services that accelerate a suite of challenging *irregular* computations. Irregular computations access memory in unpredictable patterns and are dominated by data movement. We use this suite of irregular workloads to explore the design space of the Memory Service programming model and Livia architecture, as well as to demonstrate the feasibility of mapping Memory Services onto our specialized MSE hardware.

Beyond irregular computations, we believe that Memory Services can accelerate a wide range of tasks, such as background systems (e.g., garbage collection [60], data dedup [86]), cache optimization (e.g., sophisticated cache organizations [77, 80, 81], specialized prefetchers [6, 98, 99]), as well as other functionality that is prohibitively expensive in software today (e.g., work scheduling [62], fine-grain memoization [28, 102]). We leave these to future work.

Contributions: This paper contributes the following:

1. We propose the *Memory Service* programming model to facilitate data-centric programming throughout the memory hierarchy. We define a simple API for Memory Services and develop a library of Memory Services for common irregular data structures and algorithms.
2. We design Livia, an efficient system architecture for the Memory Services model. Livia distributes specialized *Memory Service Elements* (MSEs) throughout the memory hierarchy that schedule and execute Memory Service tasks.
3. We explore the design space of MSEs. This leads us to a unique hybrid CPU-FPGA architecture that distributes reconfigurable logic throughout the memory hierarchy.
4. We evaluate Memory Services for our irregular workloads against prior multicore and processing in-memory (PIM) designs. With only 3% added area, Livia improves performance by up to 1.3× to 2.4× while reducing dynamic energy by up to 1.2× to 4.7×.

Road map: Sec. 2 motivates Memory Services on a representative irregular workload. Secs. 3 and 4 describe Memory Services and Livia. Sec. 5 presents our experimental methodology, and Sec. 6 evaluates Livia. Sec. 7 discusses related work, and Sec. 8 concludes with directions for future work.

[1] So named for the messenger pigeon, a variety of *Columba livia.*

2 Background and Motivation

We begin by discussing prior approaches to reduce data movement, and why they fall short on irregular computations.

2.1 Data movement is a growing problem

Data movement fundamentally limits system performance and cost, because moving data takes orders-of-magnitude more time and energy than processing it [2, 15, 21, 25, 50, 83]. Even on high-performance, out-of-order cores, system performance and energy are often dominated by accesses to the last-level cache (LLC) and main memory [39, 50]. This trend is amplified by the shift towards specialized cores that significantly reduce the time and energy spent on computation [39, 48, 50].

Irregular workloads are important and challenging: Data movement is particularly challenging on applications that access data in irregular and unpredictable patterns. These include many important workloads in, e.g., machine learning [32, 58, 68], graph processing [54, 59], and databases [92].

Though locality is often present in these workloads [8], standard techniques to reduce data movement struggle. Irregular prefetchers [44, 47, 96] can hide data access latency, but they do not reduce overall data movement [62]. Moreover, irregular workloads are poorly suited to common accelerator designs [18, 65]. Their data-dependent control does not map well onto an array of simple, replicated hardware with centralized control, and their unpredictable memory accesses render scratchpad memories ineffective.

2.2 Motivating example: Lookups in a binary tree

To motivate Livia's approach and illustrate the challenges faced by prior techniques, we consider how different systems behave on a representative workload: looking up items in the binary tree depicted in Fig. 2. What is the best way to map such a tree onto a memory hierarchy?

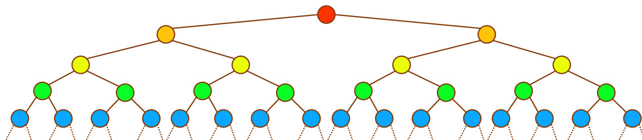


Fig. 2. A self-balancing search tree.

The ideal mapping places the most frequently accessed nodes in the tree closest to the requesting core, as illustrated in Fig. 3. This placement is ideal because it makes the best use of the most efficient memories (i.e., the smallest caches).

Ideal data movement: We can now consider how much data movement tree lookups will incur in the best case. The highlighted path in Fig. 3 shows how a single lookup must traverse from the root to a leaf node, accessing larger caches along the way as it moves down the memory hierarchy. Hence, the ideal data movement is the cost of walking nodes along this lookup path: accessing each cache/memory plus traversing the NoC. This cost is ideal because it considers only the cost of loading each node and proceeding to the next, ignoring the cost of locating nodes (i.e., accessing directories) and processing them (i.e., executing lookup code).

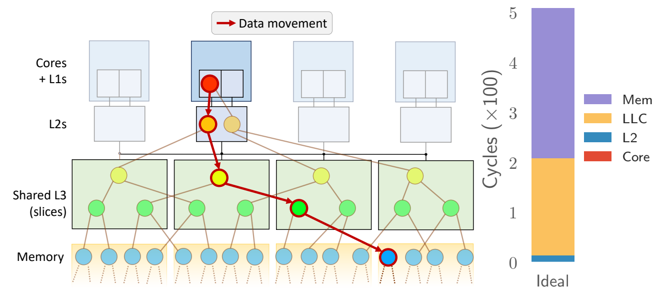


Fig. 3. Ideal data movement: The tree lookup walks the memory hierarchy, from the root in the L1 to the leaf in memory.

Modeling methodology: Throughout this section, we compare the time per lookup for a 512 MB AVL tree [20, 23] on a 64-core system with a 32 MB LLC and mesh on-chip network. (See Sec. 5 for further details.) We model how each system performs lookups, following the figures, by adding up the average access cost to access the tree at each level of the memory hierarchy. This simple model matches simulation.

Fig. 3 shows that Ideal data movement is dominated by the LLC and memory, primarily in the NoC. We now consider how practical systems measure up to this Ideal.

2.3 Current systems force needless data movement

Traditional multicore memory hierarchies are, in one respect, not far from Ideal. Fig. 4 shows how, in a conventional system, each level of the tree eventually settles at the level of the memory hierarchy where it ought to—at least, most of the time. The problem is that, since lookup code only executes on cores, data is never truly settled. This has several harmful effects: data moves unnecessarily far between lookups, each lookup must check multiple caches along the hierarchy, and each lookup evicts other useful data in earlier cache levels.

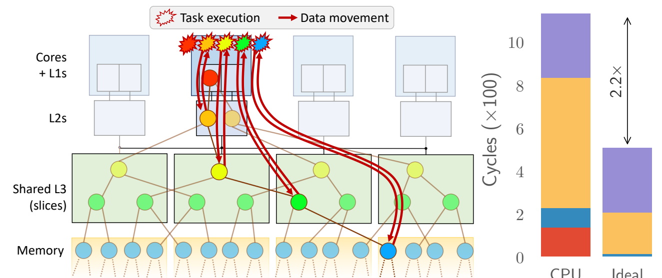


Fig. 4. Compute-centric systems frequently move data long distances between cores and the memory hierarchy.

The net effect of these design flaws is illustrated by the red arrows in Fig. 4, showing how data repeatedly moves up and down the memory hierarchy. Fig. 4 also shows the time per lookup. The traditional multicore is 2.2× worse than Ideal,

adding cycles to execute lookup code on cores and in the NoC moving data between cores and the LLC.

Finally, note that replacing cores with an accelerator would not be very effective because data movement is the main problem. Even if an accelerator or prefetcher could eliminate all the time spent on cores, lookups would still take 1.9× longer than Ideal data movement dictates.

2.4 Processing in-memory fails to exploit locality

Processing in-memory (PIM) avoids the inefficiency of a conventional cache hierarchy by executing lookups near memory, below the shared LLC. While many variations of these systems exist, a common theme is that they do away with deep cache hierarchies, preferring to access memory directly. This may benefit streaming computations, but it cedes abundant locality in irregular workloads like tree lookups. For these workloads, PIM does not capture our notion of processing in data's natural location—namely, the caches.

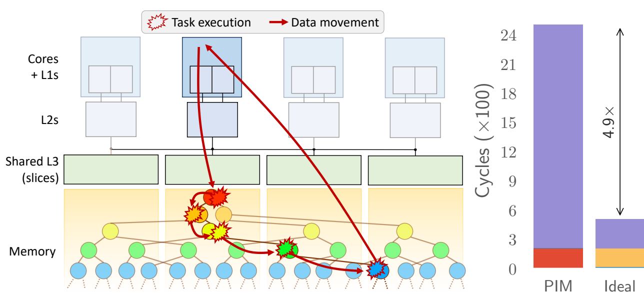


Fig. 5. Processing in-memory (PIM) sacrifices locality.

The result is significantly worse data movement for PIM systems. Fig. 5 shows how a pure PIM approach incurs expensive DRAM accesses, where Ideal has cheap cache accesses, and still incurs NoC traffic between memory controllers. The result is a slowdown of 4.9× over Ideal. In fact, this is optimistic, as our model ignores limited bandwidth at the root.

2.5 Prior processing in-cache approaches fall short

Hybrid PIM designs [5, 33] process data on cores if the data is present in the LLC, and migrate them to execute near-memory otherwise. Fig. 6 illustrates how lookups execute on EMC [33]. The first few levels of the tree execute on a core, like a compute-centric system, until the tree falls off-chip and is offloaded to the memory controller, like PIM.

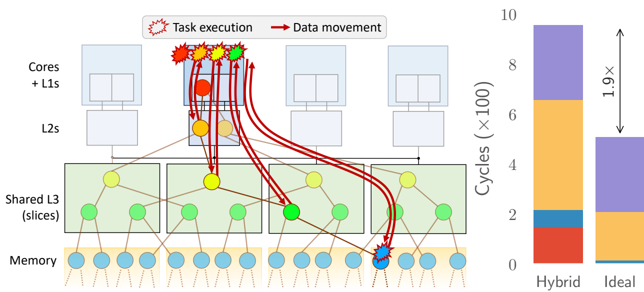


Fig. 6. Hybrid PIM still incurs unnecessary data movement.

One might think that these hybrid designs capture most of the benefit of Memory Services for tree lookups, but Fig. 6 shows this is not so. Because these designs adopt the compute-centric design for data that fits on-chip, they still incur much of its inefficiency. Overall, Hybrid-PIM is only 23% better than a compute-centric system, and still 1.9× worse than Ideal.

The unavoidable conclusion is that compute must be distributed throughout the memory hierarchy, rather than clustered at its edges, so that lookups can execute in-cache where the data naturally resides. Unfortunately, prior ***in-cache computing designs*** are too limited to significantly reduce data movement on the irregular workloads we consider. The few fully programmable near-cache designs focus on coherence [55, 74, 82] or prefetching [6, 99], not on reducing data movement. The others only support a few operations, e.g., remote memory operations (RMOs) for simple tasks like addition [5, 37, 42, 51, 57, 79, 94] or, more recently, logical operations using electrical properties of the data array [1, 24]. Most designs operate only at the LLC, and none actively migrate tasks and data to their best location in the hierarchy.

These designs stream instructions one-by-one from cores. This means that, for each node in the tree, *instructions must move to caches* and *data must return to cores* (i.e., to decide which child to follow). Hence, though these designs accelerate part of task execution in-cache, their overall data movement still looks like Fig. 6 and faces the same limitations.

2.6 Memory Services are nearly Ideal

Irregular workloads require a different approach. Frequent data movement to and from cores must be eliminated. Instead, cores should offload a high-level computation into the memory hierarchy, with no further communication until the entire computation is finished.

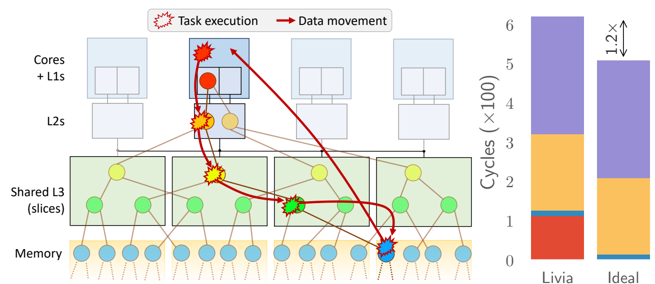


Fig. 7. Memory Services on Livia are nearly ideal.

Fig. 7 shows how Livia executes tree lookups as a Memory Service. The Memory Service breaks the lookup into a chain of tasks, where each task operates at a single node of the tree (see Fig. 9) and spawns tasks *within the memory hierarchy* to visit the node's children. The lookup begins executing on the core, and, following the path through the tree, spawned tasks migrate down the hierarchy to where nodes have settled. (Sec. 4 explains how Livia schedules tasks within the memory hierarchy and migrates data to its best location.)

Comparing to Fig. 3, Livia looks very similar to Ideal. Livia adds time only to *(i)* execute lookup code and *(ii)* access directories to locate data in the hierarchy. These overheads are small, so Livia achieves near-Ideal behavior.

To sum up, Memory Services express complex computations as graphs of dependent tasks, and Livia hardware schedules these tasks to execute at the best location in the memory hierarchy. Together, these techniques eliminate unnecessary data movement while capturing locality when it is present. Livia thus minimizes data movement by enabling data-centric computing throughout the memory hierarchy.

3 Memory Services API

We describe Memory Services starting from the programmer's view, and then work our way down to Livia's implementation of Memory Services.

Execution model: Memory Services are designed to accelerate workloads that are bottlenecked on data accesses, which follow a pattern of: loading data, performing a short computation, and loading more data. Memory Services include this data access explicitly within the programming model so that tasks can be proactively scheduled near their data.

Memory Services express application functionality as a graph of short, simple, dependent tasks. This is implemented by letting each task spawn further tasks and pass data to them. Each task gets its own execution context and runs concurrently (and potentially in parallel) with the thread that spawns it. This model supports both fork/join and continuation-passing programming styles. To simplify programming, Memory Services execute in a cache-coherent address space like any other thread in a conventional multicore system.

Invoking tasks: Applications are able to invoke memory services tasks using ms_invoke, which has the C-like type shown in Fig. 8. ms_invoke runs the ms_function_t called fn on data residing at address data_ptr with additional arguments args. Before calling ms_invoke, the caller initializes a future via ms_future_init that indicates where the eventual result of the task will be returned. ms_invoke also takes flags, which can be currently be used to indicate *(i)* that the task will need EXCLUSIVE permissions to modify data_ptr, or that *(ii)* the task is STREAMING and will not reuse data_ptr. These are both hints to the system that improve task and data scheduling, but do not affect program correctness.

```
typedef void (*ms_function_t)(
    T* data_ptr, ms_future_t* future, U... args);
void ms_invoke(ms_function_t fn, int flags,
    T* data_ptr, ms_future_t* future, U... args);
void ms_future_init(ms_future_t* future);
void ms_return(ms_future_t* future, R result);
R ms_wait(ms_future_t future);
```

Fig. 8. Memory Services API. T, U, and R are user-defined types.

Communicating results: Memory Service tasks return values to their invoker by fulfilling the future through the ms_send API (Fig. 8). The invoker obtains this value explicitly by calling ms_wait. ms_invoke calls are asynchronous, but a simple wrapper function could be placed around an ms_invoke and accompanying ms_wait to allow for a synchronous programming model similar to an RPC system. Futures can be passed among invoked tasks until the result is eventually returned to the invoker (see, e.g., Fig. 7).

Example: Tree lookup. Consider the following simplified binary tree lookup function using the API in Fig. 8:

```
void lookup(node_t* node, ms_future_t* res, int key) {
    if(node->key == key) {
        ms_return(res, node);
    } else if(node->key < key) {
        ms_invoke(lookup, 0, node->left, res, key);
    } else {
        ms_invoke(lookup, 0, node->right, res, key);
    }
}
...
ms_future_t res;
ms_future_init(&res);
ms_invoke(lookup, /*flags*/ 0, &root, &res, /*key*/ 42);
node_t* result_node = (node_t*) ms_wait(res);
```

Fig. 9. Memory Service code for a simple binary tree lookup.

This example shows that Memory Service code looks quite similar to a naïve implementation of the same code on a baseline CPU system, and our experience has been that, for data structures and algorithms well-suited to Memory Services, conversion has been a mechanical process.

Memory Services on FPGA: We map Memory Services onto FPGA through high-level synthesis (HLS). For FPGA execution, it is especially important to identify the hot path. Any execution that strays from this hot path will raise a flag that causes execution to fall back to software at a known location. To aid HLS, each task is decomposed into a series of pure functions that map easily into combinational logic. This decomposition effectively produces a state machine with one of the following actions at each transition: invoking another task, waiting upon a future, reading memory, writing memory, raising the fallback flag, or task completion. For our applications, this transformation is trivial (e.g., the tree lookup in Fig. 9), but some tasks would be split into multiple stages [18]. Table 1 shows the results from HLS for our benchmarks. Memory Services require negligible area and execute in at most a few cycles, letting small FPGAs accelerate a wide range of irregular workloads.

Limitations: Memory Services are currently designed to minimize data movement for a single data address per task. Many algorithms and data structures decompose naturally

Benchmark	C LoC	Area (mm^2)	Cycles @ 2.4 GHz
AVL tree	20	0.00203	4
Linked list	15	0.00212	3
PageRank	5	0.00185	4
Message queue	4	0.00178	1

Table 1. High-level synthesis on the Memory Services considered in this paper. Designs are mapped from C through Vivado HLS to Verilog. Latency was taken from Vivado for a Xilinx Virtex Ultrascale. For area, designs were synthesized with VTR [75] onto a Stratix-IV FPGA model, scaled to match the more recent Stratix-10 [52, 85].

into a graph of such tasks [46]. In the future, we plan to extend Memory Services to accelerate *multi-address* tasks through in-network computing [1, 42, 70] and online data scheduling [9, 63]. Additionally, we have thus far ported benchmarks to the Memory Services API and done HLS by hand, but these transformations should be amenable to a compiler pass in future work.

4 Livia Design and Implementation

This section explains the design and implementation of Livia, our architecture to support Memory Services. Livia is a tiled multicore system where each tile contains an out-of-order core (plus its private cache hierarchy), one bank of the shared distributed LLC, and a Memory Service Element (MSE) to accelerate Memory Service tasks. Livia makes small changes to the OoO core and introduces the MSEs, which are responsible for scheduling and executing tasks in the memory hierarchy.

Fig. 10 illustrates the design, showing how Memory Services migrate tasks to execute in-cache where it is most efficient. The top-left core invokes operation f infrequently on data x, so f is sent to execute on x's tile when invoked. By contrast, the bottom-left core invokes f frequently on data y, so the data y is cached in the bottom-left core's private caches and f executes locally. *All of this scheduling is done transparently in hardware without bothering the programmer.*

Livia heavily leverages the baseline multicore system to simplify its implementation. It is always safe to execute Memory Services on the OoO cores—in other words, executing on MSEs is "merely" a data-movement optimization. This property lets Livia fall back on OoO cores when convenient, so that Livia includes the simplest possible mechanisms that accelerate common-case execution.

4.1 Modifications to the baseline system

Livia modifies several components of the baseline multicore system to support Memory Services.

ISA extensions: Livia adds the following instruction:

invoke <function>

The ms_invoke API maps to the invoke instruction, which takes a pointer to the function being called, the flags, a data_ptr that the function will operate on, a pointer to the future that will hold the result, and the user-defined arguments.

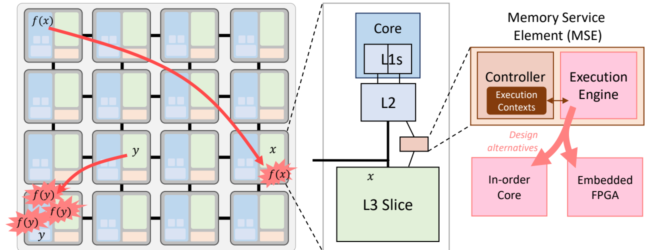


Fig. 10. Livia adds a *Memory Service Element (MSE)* to each tile in a multicore. MSEs contain a *controller* that migrates tasks and data to minimize data movement, and an *execution engine* that efficiently executes tasks near-data. The MSE execution engine is implemented as either a simple in-order core or a small embedded FPGA.

While the function is encoded in the instruction, the other arguments are passed in registers following the system's calling convention. (E.g., we use the x86-64 Sys-V variadic function call ABI, where RAX also holds the number of integer registers being passed and the flags.)

invoke first probes the L1 data cache to check if data_ptr is present. If it is, invoke turns into a vanilla function call, simply loading the data and then calling the specified function. If not, then invoke offloads the task onto MSEs. To do so, it assembles a packet containing: the function pointer (64 b), the data_ptr pointer (64 b), the future pointer (64 b), the flags (2 b), and additional arguments. All of our services fit in a packet smaller than a cache line. The core sends this packet to the MSE on the local tile, which is thereafter responsible for scheduling and executing the task.

ms_send and ms_wait can be implemented via loads and stores. ms_wait spins on the future, waiting for a ready flag to be set. Like other synchronization primitives, ms_wait yields to other threads while spinning or quiesces the core if no threads are runnable. ms_send writes the result into the future pointer using a store-update instruction that updates the value in remote caches rather than allocating it locally. store-update is similar to a remote store [37], except that it pushes updates into a private cache, rather than the home LLC bank, if the data has a single sharer. store-update lets ms_send communicate values without any further coherence traffic in the common case. Sending results through memory is done to simplify interactions with the OS (Sec. 4.4).

Compilation and loading: invoke instructions are generated from the ms_invoke API, which is supplied as a library. FPGA bitstreams are bundled with application code as a fat binary and FPGAs are configured when a program is loaded or swapped-in.

Coherence: Livia uses clustered coherence [61], with each tile forming a cluster. OoO cores and MSEs snoop coherence traffic to each tile. Directories also track MSEs in the memory controllers as potential sharers. This design keeps MSEs coherent with minimal extra state over the baseline.

Work-shedding: Falling back to OoO cores: In several places, Livia simplifies its design by relying on OoO cores to run tasks in exceptional conditions, e.g., when an MSE suffers a page fault. We implement this by triggering an interrupt on a nearby OoO core and passing along the architectural state of the task. This is possible in the FPGA design because each bitstream corresponds to a known application function.

4.2 MSE controller

We now describe the new microarchitectural component introduced by Livia, the Memory Service Element (MSE). Fig. 10 shows the high-level design: MSEs are distributed on each tile and memory controller, and each consists of a controller and an execution engine. We first describe the MSE controller, then the MSE execution engine.

The MSE controller handles invoke messages from cores and other MSEs. It has two jobs: *(i)* scheduling work to minimize data movement, and *(ii)* providing architectural support to simplify programming.

4.2.1 Scheduling tasks and data across the memory hierarchy to minimize data movement

We first explain how Livia schedules tasks in the common case, then explain how Livia converges over time to a near-optimal schedule and handles uncommon cases.

Common case: The MSE decides where to execute a task by walking the memory hierarchy, following the normal access path for the requested data_ptr in the invoke instruction. At each level of the hierarchy, the MSE proceeds based on whether the data is present. When the data is present, the MSE controller schedules the task to run locally. Since MSEs are on each tile and memory controller, there is always an MSE nearby. When the data is not present, the MSE controller migrates the task to the next level of the hierarchy (except as described below) and repeats. The same process is followed for tasks invoked from MSEs, except that the MSEs bypass the core's private caches when running tasks from the LLC.

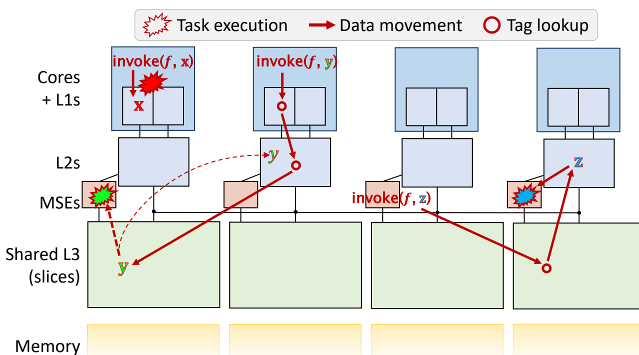


Fig. 11. Livia's task scheduling for three invokes on different data: x, y, and z. Livia schedules tasks by following the normal lookup path and migrates data to its best location through sampling.

Fig. 11 shows three examples. invoke(f,x) finds x in the L1 and so executes f(x) on the main core. invoke(f,y) checks for y in the L1 and L2 before finding y in the LLC and executing f(y) on the nearby MSE. (The thin dashed line to the L2 is explained below.) invoke(f,z) comes from an MSE, not a core, so it bypasses the private caches and checks the LLC first.

Assuming data has settled in its "natural" location in the memory hierarchy, this simple procedure schedules all tasks to run near-data with minimal added data movement over Ideal. Note that races are not a correctness issue in this scheduling algorithm, since once a scheduling decision is made, the MSE controller issues a coherent load for the data.

Migrating data to its natural location: The problem is that data must migrate to its natural location in the hierarchy and settle there. In general, finding the optimal data layout is a very challenging problem. Prior work [5] has addressed this problem in the LLC by replicating the cache tags; Livia takes a simpler and much less expensive approach.

When the data is not present, the MSE controller flips a weighted coin, choosing whether to *(i)* migrate the task to the next level of the hierarchy, or *(ii)* run the task locally. With ϵ probability ($\epsilon = 1/32$ in our implementation), the task is scheduled locally and the MSE controller fetches the data with the necessary coherence permissions. The thin dashed line in Fig. 11 illustrates this case, showing that invoke(f,y) will occasionally fetch the data into the L2. Similar to prior caching policies [73], we find that this simple, stateless policy gradually migrates data to its best location in the memory hierarchy. For data that is known to have low locality, the programmer can disable sampling by passing the STREAMING flag to ms_invoke. (See Sec. 6.5 for a sensitivity study.)

When data is cached elsewhere: Sometimes the MSE controller will find that the data is cached in another tile's L2. If the L2 has a shared (i.e., read-only) copy and the EXCLUSIVE flag is not set, then the LLC also has a valid copy and the MSE controller can schedule the task locally in the LLC. Otherwise, the MSE controller schedules the task to execute on the remote L2's tile. This is illustrated by invoke(f,z) in Fig. 11, which shows how Livia locates z in a remote L2 and executes f(z) on the remote tile's MSE.

Spawning tasks in memory controllers: When data resides off-chip, tasks will execute in the memory controller MSEs, where they may spawn additional tasks. Scheduling tasks in the memory controller MSEs is challenging because these MSEs lie below the coherence boundary. To find if a spawned task's data resides in the LLC, a naïve design must schedule spawned tasks back on their home LLC bank. However, this naïve design adds significant data movement because tasks spawned in the memory controllers tend to access data that is not in the LLC and so usually end up back in the memory controller MSEs anyway.

To accelerate this common case, Livia speculatively forwards spawned tasks to their home memory controller MSE, which will immediately schedule a memory read for the requested data. In parallel, Livia checks the LLC home node for coherence. If the data is present, the task executes in the LLC; otherwise, the LLC adds the new memory controller MSE as a sharer. Either way, the LLC notifies the memory controller MSE accordingly. The memory controller MSE will wait until it has permissions before executing the task. This approach hides coherence permission checks behind memory latency at the cost of modest complexity (Sec. 6.5).

4.2.2 Architectural support for task execution

Once a scheduling decision is made, the MSE runs the task by, in parallel: loading the requested data, allocating the task an execution context in local memory, and finally starting execution. The MSE controller hides the task startup overhead with the data array access. If a task runs a long-latency operation (a load or ms_wait), the MSE controller deschedules it until the response arrives. If the MSE controller ever runs out of local storage for execution contexts, it sheds incoming tasks to an idle hardware thread on the local OoO core [100] or, if the local OoO core is overloaded, to the invoking core to apply backpressure.

Implementation overhead: The MSE controller contains simple logic and its area is dominated by storage for execution contexts. To support one outstanding execution context from each core (a conservative estimate), the MSE requires approximately $64\,\text{B} \times 64\,\text{cores} = 4\,\text{KB}$ of storage.

4.3 MSE execution engine

The MSE execution engine is the component that actually runs tasks throughout the cache hierarchy. We consider two design alternatives, depicted in Fig. 10: *(i)* in-order cores and *(ii)* embedded FPGAs. The former is the simplest design option, whereas the latter delivers higher performance.

4.3.1 In-order core

The first design option is to execute tasks on a single-issue in-order core placed near cache banks and memory controllers. This core executes the same code as the OoO cores, though, to reduce overheads and simplify context management in the MSE controller, each task is allocated a minimal stack sufficient only for local variables and a small number of function calls. If a thread would ever overrun its stack, it is shed to a nearby OoO core (see "Work-shedding" above).

Implementation overhead: We assume single-issue, in-order cores similar to an ARM Cortex M0, which require approximately 12,000 gates [7]. This is a small area and power overhead over a wide-issue core, comparable to the size of its L1 data cache [6].

4.3.2 FPGA

The in-order core is a simple and cheap design point, but it pays for this simplicity in performance. Since a single application request may invoke a chain of many tasks, Livia is sensitive to MSE execution latency (see Sec. 6). We therefore consider a specialized microarchitectural design that replaces the in-order core with a small embedded FPGA. Memory Service tasks take negligible area (Table 1), letting us configure the fabric when a program is loaded or swapped in. An interesting direction for further study is the area-latency tradeoff in fabric design [4] and fabrics that can swap between multiple designs efficiently [30], but these are not justified by our current workloads given their negligible area.

Implementation overhead: As indicated in Table 1, Memory Services map to small FPGA designs. Among our benchmarks, the largest area is still less than $0.01\,\text{mm}^2$. Hence, a small fabric of $0.1\,\text{mm}^2$ (3% area overhead on a 64-tile system at $200\,\text{mm}^2$) can support more than 10 concurrent services.

4.4 System integration

Livia's design includes several mechanisms for when Memory Services interact with the wider system.

Virtual memory: Tasks execute in an application's address space and dereference virtual addresses. The MSE controller translates these addresses by sharing the tile's L2 TLB. MSEs located on memory controllers include their own small TLBs. We assume memory is mapped through huge pages so that a small TLB (a few KB) suffices, as is common in applications with large amounts of data [49, 56, 69].

Interrupts: Memory Service tasks are concurrent with application threads and execute in their own context (Sec. 3). Hence, Memory Services do not complicate precise interrupts on the OoO cores. Memory Service tasks can continue executing on the MSEs while an OoO core services an I/O interrupt and, since they pass results through memory, can even complete while the interrupt is being processed. Faults from within a Memory Service task are handled by shedding the task to a nearby OoO core, as described above.

OS thread scheduling: Futures are allocated in an application's address space, and results are communicated through memory via a store-update. This means it is safe to deschedule threads with outstanding tasks, because the response will be just be cached and processed when the thread is rescheduled. Moreover, the thread can be rescheduled on any core without correctness concerns. Memory Service tasks are descheduled when an application is swapped out by sending an inter-process interrupt (IPI) that causes MSEs to shed tasks from the swapped-out process to nearby OoO cores.

Cores	64 cores, x86-64 ISA, 2.4 GHz, OOO Goldmont μarch (3-way issue, 78-entry IQ/ROB, 16-entry SB, ... [3])
L1	32 KB, 8-way set-assoc, split data and instruction caches
L2	128 KB, 8-way set-assoc, 2-cycle tag, 4-cycle data array
LLC	32 MB (512 KB per tile), 8-way set-assoc, 3-cycle tag, 5-cycle data array, inclusive, LRU replacement
NoC	mesh, 128-bit flits and links, 2/1-cycle router/link delay
Memory	4 DRAM controllers at chip corners; 100-cycle latency

Table 2. System parameters in our experimental evaluation.

5 Experimental Methodology

Simulation framework: We evaluate Livia in execution-driven microarchitectural simulation via a custom, cycle-level simulator, which we have validated against end-to-end performance models (like Sec. 2) and through extensive execution traces. Tightly synchronized simulation of dozens of concurrent execution contexts (e.g., 64 cores + 72 MSEs) restricts us to simulations of hundreds of millions of cycles.

System parameters: Except where specified otherwise, our system parameters are given in Table 2. We model a tiled multicore system with 64 cores connected in a mesh on-chip network. Each tile contains a main core that runs application threads (modeled after Intel Goldmont), one bank of the shared LLC, and MSEs (to ease implementation, our simulator models MSEs at both the L2 and LLC bank). MSE engines are modeled as simple IPC=1 cores or FPGA timing models, as appropriate to the evaluated system. We conduct several sensitivity studies and find that Livia's benefits are robust to a wide range of system parameters.

Workloads: We have implemented four important data-access-bottlenecked workloads as Memory Services: lock-free AVL trees, linked lists, PageRank, and producer-consumer queues. We evaluate these workloads on different data sizes, system sizes, and access patterns. These workloads are described in more detail as they are presented in Sec. 6.

Each workload first warms up the caches by executing several thousand tasks, and we present results for a representative sample of tasks following warm-up. To reduce simulation time for Livia, our warm-up first runs several thousand requests on the main cores using normal loads and stores before running additional Livia warm-up tasks via invoke. This fills the caches quickly, and we have confirmed that this methodology matches results run with a larger number of Livia warm-up tasks.

Systems: We compare these workloads across five systems:
- CPU: A baseline multicore with a passive cache hierarchy that executes tasks in software on OoO cores.
- PIM: A near-memory system that executes tasks on simple cores within memory controllers.
- Hybrid-PIM: A hybrid design that executes tasks on OoO cores when they are cached on-chip, or on simple cores in memory controllers otherwise.
- Livia-SW: Our proposed design with MSE execution engines implemented as simple cores.
- Livia-FPGA: Our proposed design with MSE execution engines implemented as embedded FPGAs.

PIM and Hybrid-PIM are implemented basically as Livia-SW with MSEs at the L2 and LLC disabled. The CPU system executes each benchmark via normal loads and stores, and all other systems use our new invoke instruction.

Metrics: We present results for execution time and dynamic execution energy, using energy parameters from [89]. Where possible, we breakdown results to show where time and energy are spent throughout the memory hierarchy. We focus on dynamic energy because Livia has negligible impact on static power and to clearly distinguish Livia's impact on data movement energy from its overall performance benefits.

6 Evaluation

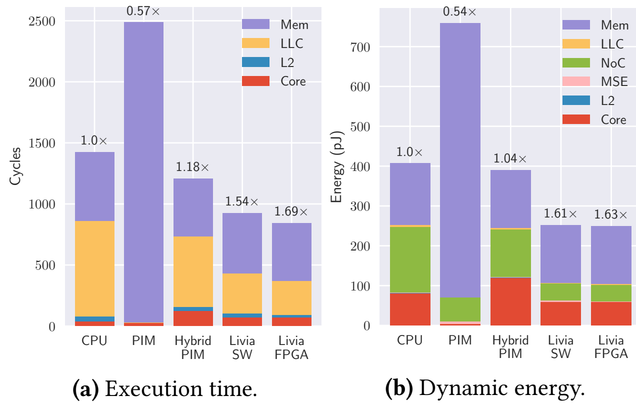
We evaluate Livia to demonstrate the benefits of the Memory Service programming model and Livia hardware on four irregular workloads that are bottlenecked by data movement. We will show that performing computation throughout the memory hierarchy provides dramatic performance and energy gains. We will also identify several important areas where the current Livia architecture can be improved. Some results are described only in text due to limited space; these can be found online in a technical report [19].

6.1 Lock-free-lookup AVL tree

We first consider a lock-free AVL search tree [23]. Binary-search trees like this AVL tree are popular data structures, despite being bottlenecked by pointer chasing, which is difficult to accelerate. In addition to the usual child/parent pointers, pointers to successors and predecessor nodes are used to locate the correct node in the presence of tree rotations, allowing concurrent modifications to the tree structure. We implemented this tree as a Memory Service in three functions: the root function walks a single level of the tree, invoking itself on the child node pointer, or returning the correct node if a matching key is found; two other functions follow successor/predecessor pointers until the correct node is found (or a sentinel, in the rare case that a race is detected).

Livia accelerates trees dramatically: We evaluated a 512 MB tree (≈8.5 million nodes) on a 64-tile system. Fig. 12 shows the average number of cycles and dynamic execution energy for a single thread to walk the tree on a uniform distribution, broken down into components across the system. The graph also shows, in text, each system's improvement vs. CPU.

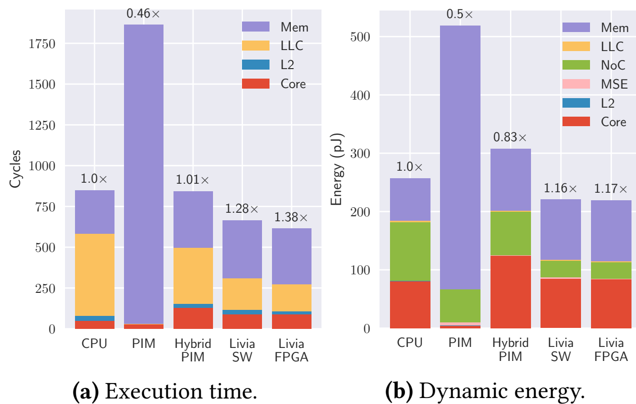
PIM takes nearly 2× as long as CPU because it cannot leverage the locality present in nodes near the root. Hybrid-PIM gives some speedup (18%), but its benefit is limited by the high NoC latency for both the in-CPU and in-memory-controller portions of its execution. Hybrid-PIM spends more



(a) Execution time. **(b)** Dynamic energy.

Fig. 12. AVL tree lookups on 64 tiles with uniform distribution.

time and energy in cores due to the overhead of invoke instructions. Livia-SW and Livia-FPGA perform significantly better, accelerating lookups by 54% and 69%, respectively. Livia drastically reduces NoC traversals, and Livia-FPGA additionally reduces cycles spent in computation at each level of the tree. This leads to similar dynamic energy improvements of 61% and 63%, respectively.



(a) Execution time. **(b)** Dynamic energy.

Fig. 13. AVL tree lookups on 64 tiles with Zipfian distribution.

Livia works well across access patterns: Fig. 13 shows that these benefits hold on the more cache-friendly YCSB-B workload, which accesses keys following a Zipf distribution ($\alpha = 0.9$). Due to increased temporal locality, performance improves in absolute terms for all systems. More tree levels fit in the core's private caches, reducing the opportunity for Livia to accelerate execution. Nevertheless, Livia still sees the largest speedups and energy savings. In contrast, PIM gets little speedup because it does not exploit locality, making it relatively worse compared to CPU (now over 2× slower).

Livia works well at different data and system sizes: Fig. 14 shows how Livia performs as we scale the system size. Scaling the LLC allows more of the tree to fit on-chip, but since binary trees grow exponentially with depth, this benefit is outweighed by the increasing cost of NoC traversals. Livia is effective at small tree sizes and scales the most gracefully of all systems because it halves NoC traversals for tasks executing in the LLC and memory. Going from 16 tiles to 144 tiles, Livia-FPGA's speedup improves from 53% to 87%, whereas both PIM and HybridPIM do relatively worse as the system scales because they do not help with NoC traversals.

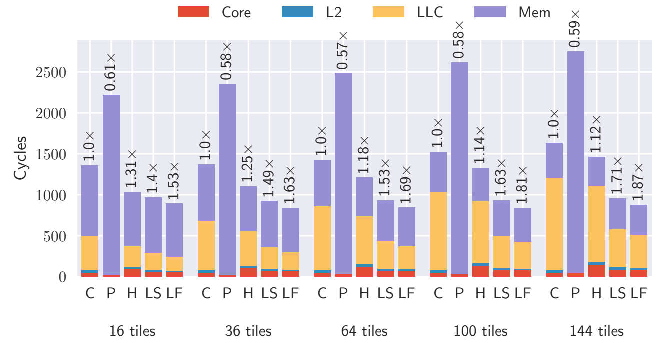


Fig. 14. Avg. lookup cycles on 512 MB AVL tree vs. system sizes.

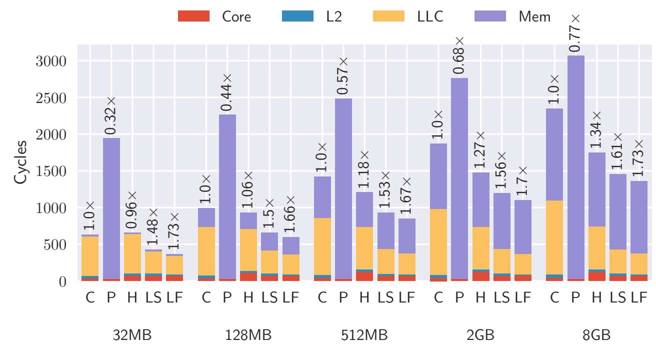


Fig. 15. Avg. lookup cycles on AVL tree at 64 tiles vs. tree size.

Fig. 15 shows the effect of scaling tree size on a 64-tile system, going from a tree that fits in cache to one 256× larger than the LLC. PIM and HybridPIM both improve as a larger fraction of the tree resides in memory, getting up to 34% speedup. Livia maintains consistent speedups of gmean 54%/70% on small to large trees. This is because Livia accelerates tree walks throughout the memory hierarchy, halving NoC traversals in the LLC, and, since tree levels grow exponentially in size with depth, a large fraction of tree levels remain on-chip even at the largest tree sizes.

Livia's benefits remain substantial even when doing extra work: Most applications use the result of a lookup as input to some other computation. We next consider whether Livia's speedup on lookups translates into end-to-end performance gains for such applications. Fig. 16 evaluates a workload that, after completing a lookup, loads an associated value via regular loads on the main core. This extra processing introduces additional delay and cache pollution.

Despite this, Fig. 16 shows that Livia still provides significant end-to-end speedup, even when the core loads an additional 1 KB for each lookup. (This value size corresponds to an overall tree size of 8 GB.) Livia's benefits remain substantial for two reasons. First, the out-of-order core is able to accelerate the value computation much more effectively

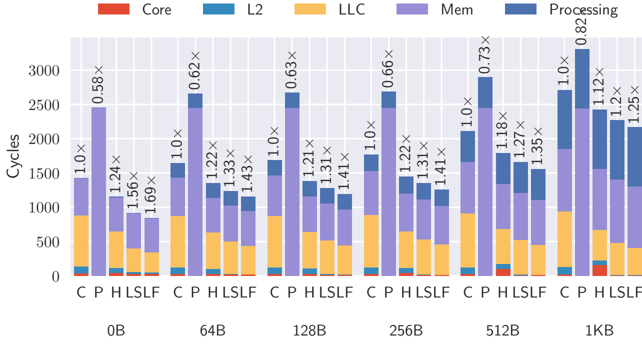


Fig. 16. Avg. cycles to lookup a key and then load the associated value on the main core. Results are for a 512 MB tree at 64 tiles, for different value sizes along the x-axis.

than the tree lookup, so the performance loss from accessing a 1 KB value is much less than an accessing an equivalent amount of data in a tree lookup. Second, the cache pollution from the value is not as harmful as it seems. Though the value flushes tree nodes from the L1 and L2, a significant fraction of the tree remains in the LLC. As a result, the time it takes to do a lookup (i.e., ignoring "Processing" in Fig. 16), only degrades slightly as value grows larger.

6.2 Linked lists

We next consider linked lists. Despite their reputation for poor performance, linked lists are commonly used in situations where pointer stability is important, as standalone data structures, embedded in a larger data structure (e.g., separate chaining for hash maps), or used to manage raw memory without overhead (e.g., free lists in an allocator).

Livia accelerates linked lists dramatically: Because of their $O(N)$ lookup time, in practice linked lists are short and embedded in larger data structures. To evaluate this scenario, we first consider an array of 4096 linked-lists, each with 32 elements (8 MB total). To perform a lookup, we generate a key and scan the corresponding list. Keys are chosen randomly, following either a uniform or Zipfian distribution.

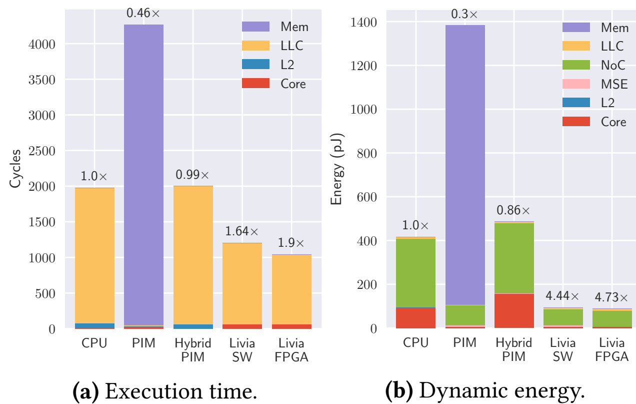


Fig. 17. Linked-list lookups on 64 tiles with uniform distribution.

Fig. 17 shows the uniform distribution. The data fits in the LLC, so PIM is very inefficient. Hybrid-PIM also sees no benefit vs. CPU because the working set fits on-chip. (Its added energy is due to invoke instructions.) Meanwhile, both Livia-SW and Livia-FPGA see large speedups of 64% and 90%, respectively, because task execution is moved off energy-inefficient cores and NoC traversals are greatly reduced. Livia improves energy by 4.4× and 4.7×. Energy savings exceed performance gains because, in addition to avoiding a NoC traversal for each load, Livia also eliminates an *eviction*, which is not on the critical path but shows up in the energy.

In fact, Fig. 17 is potentially quite pessimistic, since Livia can achieve much greater benefits in some scenarios. Initially, we unintentionally allocated linked-list nodes so that they were located in adjacent LLC banks. With this allocation, Livia could traverse a link in the list with a single NoC hop (vs. a full NoC round-trip for CPU), achieving dramatic speedups of 5× or more. In Fig. 17, we eliminate this effect by randomly shuffling nodes, but we plan in future work to explore techniques that achieve a similar effect by design.

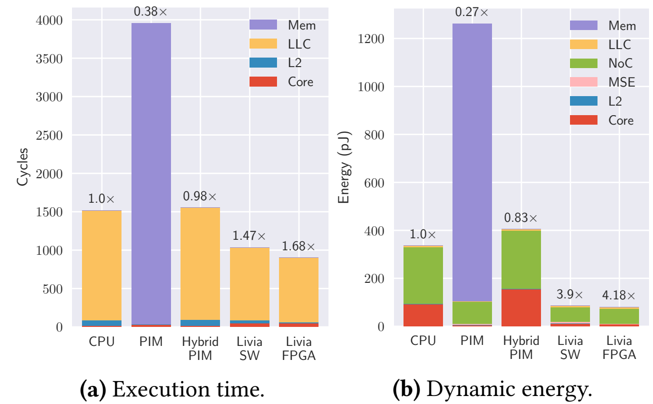


Fig. 18. Linked-list lookups on 64 tiles with Zipfian distribution.

Livia works well across access patterns: Fig. 18 shows linked-list results with the YCSB-B (Zipfian) workload. Unlike the AVL tree (Fig. 13), the Zipfian pattern has less impact on linked-list behavior. This is because linked-list lookups still quickly fall out of the private caches and enter the LLC.

Livia works well at different data and system sizes: Similar to the AVL tree, Figs. 19 and 20 show how results change when scaling the system and data size, respectively. Specifically, we scale input size by increasing the number of lists. (Results are similar when scaling the length of each list.)

As before, Livia's benefits grow with system size as NoC traversals become more expensive. However, as data size increases, Livia's benefits decrease from 1.9× to 1.54×. This is because the lists fall out of the LLC and lookups must go to memory, which starts to dominate lookup time. PIM gets speedup only at the largest list size, when most tasks execute in-memory. HybridPIM gets modest speedup, up to 36%, but significantly under-performs Livia even when most of the lists reside in memory.

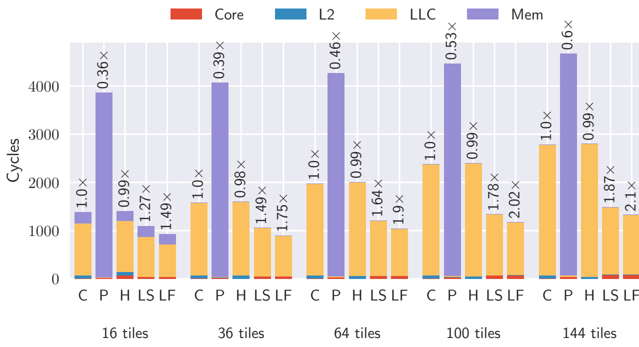


Fig. 19. Avg. lookup cycles on 4096 linked-lists vs. system sizes.

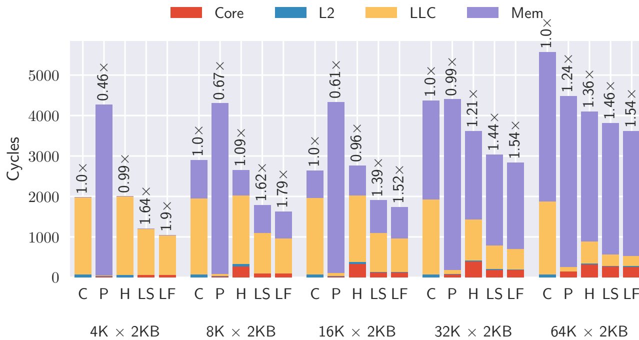
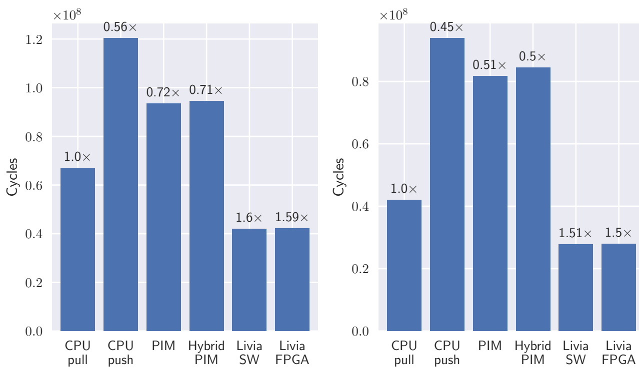


Fig. 20. Avg. cycles per linked-list lookup at 64 tiles vs. input size.

6.3 Graph analytics: PageRank

We next consider PageRank [66], a graph algorithm, running on synthetic graphs [16] that do not fit in the LLC at 16/64 cores. We compare multithreaded CPU push and pull implementations [84] to a push-based implementation written as a Memory Service, which accelerates edge updates by pushing them into the memory hierarchy where they can execute efficiently near-data, similar to recent work [64, 100, 101].



(a) 2M vert/20M edge, 16 tiles. **(b)** 4M vert/40M edge, 64 tiles.

Fig. 21. PageRank on synthetic random graphs at 16/64 cores.

Livia accelerates graph processing dramatically: Fig. 21 shows the time needed to process one full iteration after warming up the cache for one iteration. We do not break down cycles across the cache hierarchy due to the difficulty in identifying which operations are on the critical path.

Livia is 60%/51% faster than CPU-Pull at 16/64 tiles and 2.8×/3.4× faster than CPU-Push. Notice that pull is faster than push in the baseline CPU version, but the push-based Memory Service is better than both. (PIM and HybridPIM are both faster than CPU-Push, but still slower than CPU-Pull.) This is because push-based implementations in the CPU version incur a lot of expensive coherence traffic to invalidate vertices in other tiles' private caches and must update data via expensive atomic operations. Livia avoids this extra traffic by executing most updates in-place in the caches. PageRank tasks run efficiently in software, so Livia-FPGA does not help much.

Curiously, Livia's speedup decreases at 64 tiles. This is because of backpressure that frequently sheds work back to remote cores (Sec. 4.2.2). In particular, we find that the four MSEs at the memory controllers are overloaded with 64 tiles. In the future, we plan to avoid this issue through smarter work-shedding algorithms and higher-throughput MSEs at the memory controllers.

Finally, CPU-Push, CPU-Pull, Livia-SW, and Livia-FPGA get dynamic energy all within 10% of each other (omitted for space), whereas PIM and HybridPIM add modest dynamic energy (20-35%). This is due to instruction energy from shed tasks, and the relatively large fraction of data accesses that go to memory for PageRank. However, bear in mind that Livia still achieves significant end-to-end energy savings because its improved performance reduces static energy: at 16/64 tiles, Livia-FPGA improves overall energy by 1.24×/1.16× vs. CPU-Pull and by 1.82×/1.78× vs. CPU-Push.

6.4 Producer-consumer queue

Many irregular applications are split into stages and communicate among stages via producer-consumer queues [78]. These queues perform poorly on conventional invalidation-based coherence protocols, since each push and pop incurs at least two round-trips with the directory. With multiple producers, the number of coherence messages can be much worse than two. Fortunately, Memory Services give a natural way to implement queues without custom hardware support for message passing. We implement multi-producer, single-consumer queues by invoking pushes on the queue itself. Only a single NoC traversal is on the critical path, instead of three (two round-trips, partially overlapped) in CPU systems. All queue operations occur on the consumer's tile, avoiding unnecessary coherence traffic and leaving the message in the receiver's L2, where it can be quickly retrieved.

Livia accelerates producer-consumer queues dramatically: Fig. 22 shows the latency to push and pop an item from the queue on systems with 16 to 144 tiles. To factor out directory placement, we measure the latency between opposite corners of the mesh network. These results are therefore worst-case, but within a factor of two of expected latency.

Compared to the CPU baseline, Livia-SW and Livia-FPGA accelerate producer-consumer queues by roughly 2× across

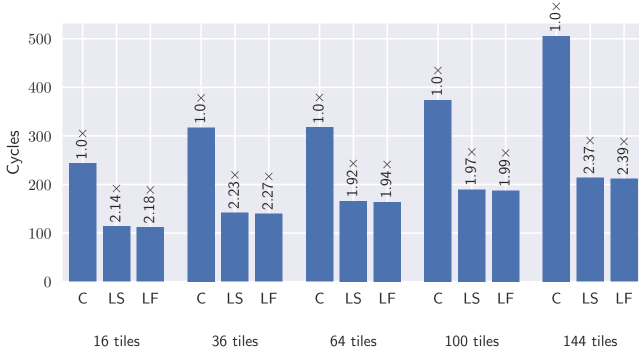


Fig. 22. Avg. latency for producer-consumer queue vs. system size.

all system sizes. This is because Livia eliminates NoC delay due to coherence traffic. With multiple senders, even larger speedup can be expected as senders can conflict and invalidate the line mid-push on the CPU system.

Note that push uses the STREAMING flag (Sec. 3) to prevent data from being fetched into the sender's L2. Without this hint, we observe modest performance degradation (roughly 25%, depending on system size), as data is occasionally migrated to the wrong cache via random sampling.

6.5 Sensitivity studies

Livia is insensitive to core microarchitecture: We ran the 512 MB AVL tree on 64 tiles modeling Silvermont, Goldmont, Ivy Bridge, and Skylake core microarchitectures (graph omitted due to space). We found that core microarchitecture had negligible impact, since Livia targets benchmarks dominated by data-dependent loads. On such workloads, increasing issue width is ineffective because simple, efficient core designs already extract all available memory-level parallelism.

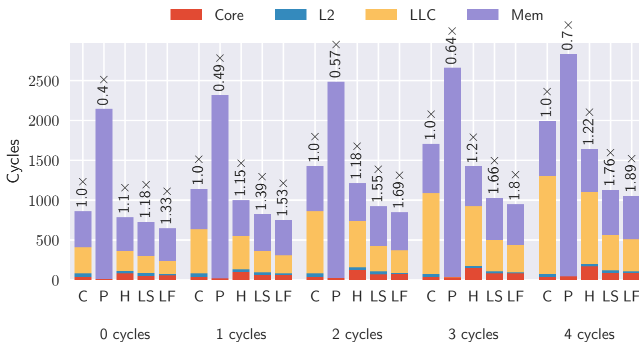


Fig. 23. Avg. lookup cycles on a 512 MB AVL tree at 64 tiles. Livia's benefits increase with larger on-chip routing delay.

Livia's benefits increase with network delay: Next, Fig. 23 considers the impact of increasing NoC delay on Livia's results, e.g., due to congestion. We found, unsurprisingly, that Livia's benefits grow as the NoC becomes more expensive, achieving up to 89% speedup as routing delay grows. However, Livia still provides substantial benefit on lightly loaded networks—even with zero router delay, Livia-FPGA gets 33% speedup for the AVL tree at 64 tiles.

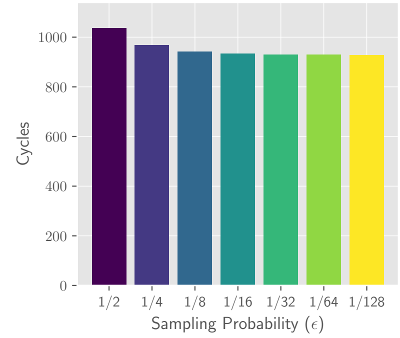


Fig. 24. Avg. lookup cycles on a 512 MB AVL tree at 64 tiles, after warm-up, for Livia-SW with different sampling probability. Livia-SW's steady-state performance is insensitive to $\epsilon < 1/8$.

Livia's steady-state performance is insensitive to sampling probability: Fig. 24 shows the steady-state performance of Livia on an AVL tree, after several million requests to warm up the caches, with different sampling probability. Similar to adaptive cache replacement policies [45, 73], we find that Livia is not very sensitive to sampling rate, so long as sampling is not too frequent. That said, we have observed that sampling can take a long time to converge to the steady-state, so ϵ should not be too small. Looking forward, Livia would benefit from data-migration techniques that can respond quickly to changes in the application's access pattern.

Livia's speculative memory prefetching is effective: Livia hides LLC coherence checks by performing them in parallel with a DRAM load for tasks spawned at memory controllers. We found that, on the AVL tree and linked list benchmarks at 64 tiles, this technique consistently hides latency equivalent to 26% of the DRAM load. The savings depend on the cost of coherence checks: speculation saves latency equal to 13% of the DRAM load at 16 tiles, but up to 38% at 144 tiles. Speculation is accurate on input sizes larger than the LLC (e.g., >99% accuracy for 512 MB AVL tree), but can be inaccurate for small inputs that just barely do not fit in the LLC (worst-case: 45% accuracy). We expect a simple adaptive mechanism would avoid these problem cases.

7 Related work

We wrap up by putting Livia in the context of related work in data-centric computing and a taxonomy of accelerators.

7.1 Data-centric computing

Data-centric computing has a long history in computer systems. There is a classic tradeoff in system design: should we move compute to the data, or data to the compute?

At one extreme, conventional CPU-based systems adopt a purely compute-centric design that always moves data to cores. At the other extreme, spatial dataflow architectures adopt a purely data-centric design that always moves data to compute [53, 88]. Many designs have explored a middle ground between these extremes. For example, hierarchical dataflow designs [67, 76] batch groups of instructions together for efficiency, and there is a large body of prior work on scheduling tasks to execute near data [10, 26, 46, 63]. Similarly, active messages (AMs) [91] and remote-procedure

calls (RPCs) [12] execute tasks remotely, often to move them closer to data.

Livia also takes a middle ground, adding a dash of data-centric design to existing systems. Livia improves on prior data-centric systems in two respects. First, we rely on the Memory Service model to *statically* identify which functions are well-suited to execute within the memory hierarchy, rather than always migrating computation to data (unlike, e.g., EM2 [53] and pure dataflow). Second, we rely on cache hardware to *dynamically* discover locality and schedule computation and data, rather than statically assigning tasks to execute at specific locations [12, 46, 91] or schedule compute infrequently in large chunks [10]. We believe this division of work between hardware and software strikes the right balance, letting each do what it does best [63].

7.2 Schools of accelerator design

The recent trend towards architectural specialization has led to a proliferation of accelerator designs. We classify these designs into three categories: *co-processor*, *in-core*, and *in-cache*. Livia falls into the under-explored *in-cache* category.
Co-processor designs treat an accelerator as a concurrent processor that is loosely integrated with cores. Co-processors can be discrete cards accessed over PCIe (e.g., GPUs and TPUs [48]), or IP blocks in a system-on-chip. Co-processor designs yield powerful accelerators, but make communication between cores and the accelerator expensive. PIM [38, 41] falls into this category, as do existing designs that integrate a powerful reconfigurable fabric alongside a CPU in order to accelerate large computations [27, 43, 65, 71, 95]. In contrast, Livia integrates many small reconfigurable fabrics throughout the memory hierarchy to accelerate short tasks.
In-core designs treat an accelerator as a "mega functional unit" that is tightly integrated with cores [31, 34, 35]. A good example is DySER [31], which integrates a reconfigurable spatial array into a core's pipeline to accelerate commonly executed hyperblocks. However, in-core designs like DySER often do not interface with memory at all, whereas Livia focuses entirely on interfacing with memory to accelerate data-heavy, irregular computations. In-core acceleration is insufficient for these workloads (Sec. 2.3).
In-cache designs are similar to in-core designs in that they tightly integrate an accelerator with an existing microarchitectural component. The difference is that the accelerator is tightly integrated into the memory hierarchy, not the core.

This part of the design space is relatively unexplored. As discussed in Sec. 2.4, prior approaches are limited to a few operations and still require frequent data movement between cores and the memory hierarchy to stream instructions and fetch results [1, 24, 37, 51, 57, 79, 94]. Livia further develops the in-cache design school of accelerators by providing a fully programmable memory hierarchy that captures locality at all levels and eliminates unnecessary communication between cores and caches. As a result, Livia accelerates a class of challenging irregular workloads that have remained beyond the reach of existing accelerator designs.

8 Conclusion and Future Work

This paper has presented Memory Services, a new programming model that enables near-data processing throughout the memory hierarchy. We designed Livia, an architecture for Memory Services that introduces simple techniques to dynamically migrate tasks and data to their best location in the memory hierarchy. We showed that these techniques significantly accelerate several challenging irregular workloads that are at the core of many important applications. Memory Services open many avenues for future work:

New applications: This paper focuses on irregular workloads, but there are a wide range of other workloads amenable to in-cache acceleration. Prior work contains many examples: e.g., garbage collection [60], data deduplication [86], and others listed in Sec. 1. Unfortunately, it is unlikely that general-purpose systems will implement specialized hardware for these tasks individually. We intend to expand Livia and Memory Services into a general-purpose platform for in-cache acceleration of these applications.

Productive programming: This paper presented an initial exploration of the Memory Service programming model targeted at expert programmers with deep knowledge of their workloads and of hardware. We plan to explore enhancements to the model and compiler that will make Memory Services more productive for the average programmer, e.g., by providing transactional semantics for chains of tasks [11] or by extracting Memory Services from lightly annotated code.

Improved microarchitecture: Finally, we see abundant opportunities to refine Livia's current design. Livia employed existing cores and FPGAs to simplify its design and demonstrate the basic potential of Memory Services. However, FPGAs are not the final answer for MSE execution, and we have left several important issues in the MSE controller unexplored. We plan to explore more efficient architectures for task execution, such as CGRAs [4, 65, 71] with wider datapaths. We will also develop policies in the MSE controller to manage resources and avoid harmful interference across co-running applications. Finally, our evaluation highlighted several opportunities to improve the microarchitecture, e.g., by designing more responsive data-migration techniques.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. We also thank Graham Gobieski for his help with high-level synthesis. This work was supported by NSF award CAREER-1845986 and CAREER-1452994.

References

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute caches. In *Proc. HPCA-23*.

[2] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. 2000. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. ISCA-27*.

[3] Agner Fog. 2020. The microarchitecture of Intel, AMD and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf.

[4] Elias Ahmed and Jonathan Rose. 2004. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12, 3 (2004), 288–298.

[5] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Proc. ISCA-42*.

[6] Sam Ainsworth and Timothy M Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *Proc. ASPLOS-XXIII*.

[7] ARM. 2019. Cortex M0+. http://www.arm.com/products/processors/cortex-m/cortex-m0plus.php.

[8] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. In *Proc. IISWC*.

[9] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable Software-Defined Caches. In *Proc. PACT-22*.

[10] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *Proc. HPCA-21*.

[11] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: safe multithreaded programming for C/C++. In *Proc. OOPSLA*.

[12] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. 1989. Lightweight remote procedure call. *ACM SIGOPS Operating Systems Review* 23, 5 (1989), 102–113.

[13] Bryan Black. 2013. Die Stacking is Happening!. In *MICRO-46 Keynote*.

[14] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proc. ASPLOS-XXIII*.

[15] David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. ISCA-27*.

[16] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[17] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. ASPLOS-XIX*.

[18] Tao Chen, Shreesha Srinath, Christopher Batten, and G Edward Suh. 2018. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *Proc. MICRO-51*.

[19] CORGi Research Group. 2020. CORGi Research Group Web Page. https://cmu-corgi.github.io/.

[20] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.

[21] William J. Dally. 2010. GPU Computing: To Exascale and Beyond. In *Supercomputing '10, Plenary Talk*.

[22] Benoıt Dupont de Dinechin, Renaud Ayrignac, P-E Beaucamps, Patrice Couvert, Benoıt Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. 2013. A clustered manycore processor architecture for embedded and accelerated applications. In *Proc. of the High Performance Extreme Computing Conference*.

[23] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. In *Proc. PPoPP*.

[24] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural cache: bit-serial in-cache acceleration of deep neural networks. In *Proc. ISCA-45*.

[25] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. 2011. Dark Silicon and The End of Multicore Scaling. In *Proc. ISCA-38*.

[26] K. Fatahalian, D.R. Horn, T.J. Knight, L. Leem, M. Houston, J.Y. Park, M. Erez, M. Ren, A. Aiken, W.J. Dally, and P. Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proc. of the 2006 ACM/IEEE conf. on Supercomputing*.

[27] Michael Feldman. 2018. Intel Ships Xeon Skylake Processor with Integrated FPGA. https://www.top500.org/news/intel-ships-xeon-skylake-processor-with-integrated-fpga/.

[28] Adi Fuchs and David Wentzlaff. 2018. Scaling Datacenter Accelerators With Compute-Reuse Architectures. In *Proc. ISCA-45*.

[29] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *Proc. PACT-24*.

[30] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proc. HPCA-22*.

[31] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *Proc. HPCA-17*.

[32] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pdream, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proc. ISCA-43*.

[33] Milad Hashemi, Eiman Ebrahimi, Onur Mutlu, Yale N Patt, et al. 2016. Accelerating dependent cache misses with an enhanced memory controller. In *Proc. ISCA-43*.

[34] Scott Hauck, Thomas W Fry, Matthew M Hosler, and Jeffrey P Kao. 2004. The Chimaera reconfigurable functional unit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12, 2 (2004), 206–217.

[35] John R Hauser and John Wawrzynek. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)*. IEEE, 12–21.

[36] John Hennessy and David Patterson. 2018. A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development. In *Turing Award Lecture*.

[37] Henry Hoffmann, David Wentzlaff, and Anant Agarwal. 2010. Remote store programming. In *Proc. HiPEAC*.

[38] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. 2016. Accelerating linked-list traversal through near-data processing. In *Proc. PACT-25*.

[39] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *Proc. of the IEEE Intl. Solid-State Circuits Conf. (ISSCC)*.

[40] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *Proc. ISCA-43*.

[41] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *Proc. ICCD*.

[42] Jiayi Huang, Ramprakash Reddy Puli, Pritam Majumder, Sungkeun Kim, Rahul Boyapati, Ki Hwan Yum, and Eun Jung Kim. 2019. Active-Routing: Compute on the Way for Near-Data Processing. In *Proc.*

HPCA-25.
[43] Intel. 2018. Intel Arria 10 Device Datasheet. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_datasheet.pdf.
[44] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proc. MICRO-46*.
[45] Aamer Jaleel, Kevin Theobald, Simon C. Steely Jr, and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proc. of the 37th annual Intl. Symp. on Computer Architecture*.
[46] Mark C Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-Centric Execution of Speculative Parallel Programs. In *Proc. MICRO-49*.
[47] Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. In *Proc. ISCA-24*.
[48] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760* (2017).
[49] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. 2015. Redundant memory mappings for fast access to large memories. In *Proc. ISCA-42*.
[50] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31, 5 (2011).
[51] Richard E Kessler and James L Schwarzmeier. 1993. CRAY T3D: A new dimension for Cray Research. In *Compcon Spring'93, Digest of Papers*.
[52] Farheen Fatima Khan and Andy Ye. 2017. A study on the accuracy of minimum width transistor area in estimating FPGA layout area. *Microprocessors and Microsystems* 52 (2017), 287–298.
[53] Omer Khan, Mieszko Lis, and Srini Devadas. 2010. Em2: A scalable shared-memory multicore architecture. (2010).
[54] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. 2016. Optimizing indirect memory references with milk. In *Proc. PACT-25*.
[55] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, , and John Hennessy. 1994. The Stanford FLASH multiprocessor. In *Proc. ISCA-21*.
[56] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens.. In *Proc. OSDI-12*.
[57] James Laudon and Daniel Lenoski. 1997. The SGI Origin: a ccNUMA highly scalable server. In *Proc. ISCA-24*.
[58] Jing Liu, Mingjing Li, Qingshan Liu, Hanqing Lu, and Songde Ma. 2009. Image annotation via graph learning. *Pattern recognition* (2009).
[59] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *PPL* 17, 01 (2007).
[60] Martin Maas, Krste Asanovic, and John Kubiatowicz. 2018. A Hardware Accelerator for Tracing Garbage Collection. In *Proc. ISCA-45*.
[61] Milo Martin, Mark D Hill, and Daniel J Sorin. 2012. Why on-chip cache coherence is here to stay. *Commun. ACM* (2012).
[62] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proc. MICRO-51*.
[63] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. Whirlpool: Improving dynamic cache management with static data classification. In *Proc. ASPLOS-XXI*.
[64] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Proc. MICRO-52*.
[65] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proc. ISCA-44*.
[66] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: bringing order to the web.* Technical Report. Stanford InfoLab.
[67] Gregory M Papadopoulos and David E Culler. 1990. Monsoon: an explicit token-store architecture. In *Proc. ISCA-17*.
[68] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proc. ISCA-44*.
[69] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proc. ISCA-44*.
[70] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2019. Opportunistic computing in gpu architectures. In *Proc. ISCA-46*.
[71] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Patterns. In *Proc. ISCA-44*.
[72] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, and Vijayalakshmi Srinivasan. 2014. NDC: Analyzing the Impact of 3D-Stacked Memory + Logic Devices on MapReduce Workloads. In *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*.
[73] Moinuddin Qureshi, Aamer Jaleel, Yale Patt, Simon Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proc. ISCA-34*.
[74] Steven K Reinhardt, James R Larus, and David A Wood. 1994. Tempest and Typhoon: User-level shared memory. In *Proc. ISCA-21*.
[75] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. 2012. The VTR project: architecture and CAD for FPGAs from verilog to routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 77–86.
[76] Shuichi Sakai, Kci Hiraki, Y Kodama, T Yuba, et al. 1989. An architecture of a dataflow single chip processor. In *Proc. ISCA-16*.
[77] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling Ways and Associativity. In *Proc. of the 43rd intl. symp. on Microarchitecture*.
[78] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *Proc. PACT-20*.
[79] Steven L Scott. 1996. Synchronization and communication in the T3E multiprocessor. In *Proc. ASPLOS-VII*.
[80] André Seznec. 1993. A case for two-way skewed-associative caches. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture*.
[81] André Seznec. 1994. Decoupled sectored caches: conciliating low tag implementation cost. In *Proc. ISCA-21*.
[82] Ofer Shacham, Zain Asgar, Han Chen, Amin Firoozshahian, Rehan Hameed, Christos Kozyrakis, Wajahat Qadeer, Stephen Richardson, Alex Solomatnikov, Don Stark, Megan Wachs, and Mark Horowitz. 2009. Smart memories polymorphic chip multiprocessor. In *Proc. DAC-46*.
[83] John Shalf, Sudip Dosanjh, and John Morrison. 2011. Exascale computing technology challenges. In *Proc. High Performance Computing for Computational Science (VECPAR)*.
[84] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proc. PPoPP*.

[85] Etienne Sicard. 2017. Introducing 7-nm FinFET technology in Microwind. (2017).

[86] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. 2017. Pageforge: a near-memory content-aware page-merging architecture. In *Proc. MICRO-50*.

[87] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In *Proc. MICRO-50*.

[88] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proc. MICRO-36*.

[89] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-Defined Cache Hierarchies. In *Proc. ISCA-44*.

[90] Po-An Tsai, Changping Chen, and Daniel Sanchez. 2018. Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies. In *Proc. MICRO-51*.

[91] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. 1992. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th annual Intl. Symp. on Computer Architecture*.

[92] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*.

[93] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. 2007. On-chip interconnection architecture of the tile processor. *IEEE micro* (2007).

[94] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU architecture. *IEEE Micro* 31, 2 (2011).

[95] Xilinx. 2017. ZC706 evaluation board for the Zynq-7000 XC7Z045 all programmable SoC user guide. https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf.

[96] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proc. MICRO-48*.

[97] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proc. HPDC*.

[98] Dan Zhang, Xiaoyu Ma, and Derek Chiou. 2016. Worklist-directed Prefetching. *IEEE Computer Architecture Letters* (2016).

[99] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *Proc. ASPLOS-XXIII*.

[100] Guowei Zhang, Virginia Chiu, and Daniel Sanchez. 2016. Exploiting Semantic Commutativity in Hardware Speculation. In *Proc. MICRO-49*.

[101] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proc. MICRO-48*.

[102] Guowei Zhang and Daniel Sanchez. 2018. Leveraging Hardware Caches for Memoization. *Computer Architecture Letters (CAL)* 17, 1 (2018).