Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies

Appears in the Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018

Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies

Po-An Tsai, Changping Chen, Daniel Sanchez
Massachusetts Institute of Technology
{poantsai, cchen, sanchez}@csail.mit.edu

***Abstract*—Conventional multicores rely on deep cache hierarchies to reduce data movement. Recent advances in die stacking have enabled near-data processing (NDP) systems that reduce data movement by placing cores close to memory. NDP cores enjoy cheaper memory accesses and are more area-constrained, so they use shallow cache hierarchies instead. Since neither shallow nor deep hierarchies work well for all applications, prior work has proposed systems that incorporate both. These asymmetric memory hierarchies can be highly beneficial, but they require scheduling computation to the right hierarchy.**

We present AMS, an adaptive scheduler that automatically finds high-quality thread-to-hierarchy mappings. AMS monitors threads, accurately models their performance under different hierarchies and core types, and adapts algorithms first proposed for cache partitioning to produce high-quality schedules. AMS is cheap enough to use online, so it adapts to program phases, and performs within 1% of an exhaustive-search scheduler. As a result, AMS outperforms asymmetry-oblivious schedulers by up to 37% and by 18% on average.

***Index Terms*—Cache hierarchies, near-data processing, asymmetric systems, scheduling, analytical performance modeling.**

I. Introduction

Data movement has become a key bottleneck for computer systems. For example, an off-chip main memory access costs 1000× more energy and takes 100× more time than a double-precision multiply-add [19]. Without a drastic reduction in data movement, memory accesses and communication will limit the scalability of future systems [31].

Conventional systems rely on *deep* multi-level cache hierarchies to reduce data movement. These hierarchies often take over half of chip area and are dominated by a multi-megabyte last-level cache (LLC). Deep hierarchies avoid costly memory accesses when they can accommodate the program's working set. But when the working set does not fit in any cache level, deep hierarchies add latency and energy for no benefit [67].

Recently, placing cores closer to main memory has become a feasible alternative to deep hierarchies. Advances in die-stacking technology [12] allow tightly integrating memory banks and cores or specialized processors, an approach known as near-data processing (NDP). NDP cores enjoy lower latency and energy to the memory stacked above them, but have limited area and power budgets [26, 62]. These factors naturally bias NDP systems not only towards efficient cores [22, 25], but also towards *shallow* hierarchies with few cache levels between cores and memories.

Shallow hierarchies substantially outperform deep ones when the working set does not fit in a large on-chip LLC, but they work poorly for cache-friendly applications. Consequently, prior work [4, 25, 34, 71, 73] has proposed *asymmetric* memory hierarchies that combine deep and shallow hierarchies within a single system. For example, Google recently proposed to use such asymmetric systems for consumer workloads [14]. Fig. 1 shows an example system. This system includes a conventional processor die with a deep cache hierarchy, connected to several memory stacks, each with a small number of NDP cores and a shallow cache hierarchy in its logic layer. The processor die and memory stacks are connected using a silicon interposer.

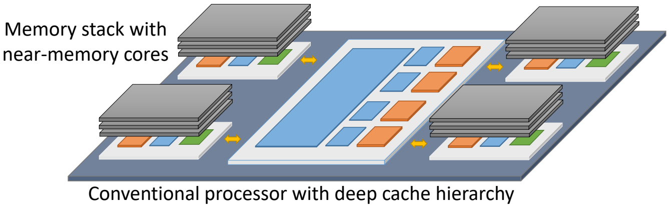

Memory stack with near-memory cores
Conventional processor with deep cache hierarchy

Fig. 1: A system with an asymmetric memory hierarchy.

Asymmetric hierarchies provide ample opportunity to improve the performance and efficiency of memory-intensive applications. We find that mapping threads to the correct hierarchy improves their performance per Joule by up to 2.8× and by 40% on average (Sec. IV). However, achieving this potential requires mapping threads to the right hierarchy dynamically. As shown in prior work, the same application can prefer different hierarchies depending on its input [4]. Moreover, colocated applications can compete for resources in either hierarchy, which affects their preferences. Thus, it is unreasonable to expect programmers or users to make this choice manually. Instead, *the system should automatically schedule threads to the right hierarchy*.

Nonetheless, this scheduling problem is quite challenging, as it has a large, non-convex decision space (i.e., which threads use the shallow hierarchy, which threads share the deep hierarchy). Much prior work has studied dynamic resource management and scheduling for systems with *symmetric* memory hierarchies [9, 37, 50, 69, 74]. And prior work on asymmetric systems [16, 69] focuses only on asymmetric *cores*, not memory hierarchies.

To address this problem, we introduce AMS, a novel thread scheduler for systems with asymmetric memory hierarchies (Sec. V). The key insight behind AMS is that the problem of modeling a thread's preferences to different hierarchies under contention bears a strong resemblance to the cache partitioning problem. Therefore, AMS leverages both working set profiling techniques and allocation algorithms from previous partitioning techniques, even though AMS does not partition any cache.

Specifically, we show that by sampling a thread's miss curve, the number of misses a thread would incur at different cache sizes, we can effectively model a thread's performance over different hierarchies under contention without trial and error. We then extend this model to handle other asymmetries (i.e., core types), proposing a novel analytical model that integrates both memory hierarchy and core asymmetries.

AMS uses the proposed model to remap threads periodically, improving performance and efficiency. We contribute two different mapping algorithms. First, AMS-Greedy is a simple scheduler that performs multiple rounds of cache partitioning and greedily maps threads to hierarchies. Second, AMS-DP leverages dynamic programming to explore the full space of configurations efficiently, finding the optimal schedule given the performance model. AMS-Greedy is cheap, scales well to large systems, and performs within 1% of AMS-DP. While AMS-DP is more expensive, it is still practical in small systems and serves as the upper bound of AMS-Greedy.

Evaluation results (Sec. VII) show that, on a 16-core system, AMS outperforms an asymmetry-oblivious baseline by up to 37% and by 18% on average. AMS adapts to program phases and handles core and cache contention in asymmetric hierarchies well, outperforming state-of-the-art schedulers. Specifically, AMS outperforms a scheduler that extends LLC-aware CRUISE [37] to NDP systems by up to 18% and by 7% on average; and AMS outperforms the PIE [69] heterogeneous-core-aware scheduler by up to 13% and by 6% on average.

II. Background and related work

We now review related work in NDP systems and scheduling algorithms, the areas that AMS draws from.

A. PIM and NDP systems

Processing-in-memory (PIM) systems proposed to integrate processors and DRAM arrays in the same die. PIM systems were studied extensively in the 90s. J-Machine [20], EXECUBE [44], and IRAM [45] proposed to integrate processors and main memory, while Active Pages [53], DIVA [28], and FlexRAM [39] instead proposed to enhance traditional processors using memory chips with coprocessors. Though compelling, PIM was unsuccessful due to the difficulties of integrating high-speed logic and DRAM [66].

With the success of 3D integration using through-silicon vias [12], the idea of processing-in-memory has been revisited recently in the context of die-stacked DRAM. Recent near-data processing (NDP) research has focused on two directions: *(i)* how to exploit the massive bandwidth of NDP systems within their limited area and power budgets, and *(ii)* how to integrate NDP systems with conventional systems.

Die-stacking technology offers lower latency, lower energy, and much higher bandwidth between the logic layer and the memory stack than conventional off-chip memories. However, it also imposes limited area and thermal budgets in the logic layer. This new tradeoff is attractive for data-intensive applications. However, without careful engineering, it is difficult to saturate the available bandwidth to fully utilize the potential of NDP systems. Thus, one important research question in recent NDP work is *what form of computation to put in the logic layer* to best balance programmability, performance/efficiency, and design constraints. On the one hand, several designs focus on general-purpose NDP systems that use simple cores [22, 25, 55], GPUs [72, 73], and reconfigurable logic [24, 26]. On the other hand, multiple projects design NDP systems tailored to important emerging workloads, such as graph analytics [3, 52], neural networks [27, 42], and sparse data structures [33, 35].

Although die-stacking technology has made NDP systems practical, not all applications can benefit from NDP. Therefore, another research direction has been *how to support an asymmetric system composed of both NDP and conventional chips*. For example, LazyPIM [15] studies how to provide coherence within asymmetric systems. PIM-enabled instructions [4], TOM [34], and Pattnaik et al. [54] focus on *how to map computation* across systems with asymmetric hierarchies. PIM-enabled instructions proposes new instructions and hardware support to decide when to offload specific instructions to in-memory, fixed-function accelerators to maximize locality. TOM proposes a combination of compiler, runtime, and hardware techniques to offload computation and place data to balance bandwidth in GPU-based asymmetric systems. Similarly, Pattnaik et al. propose to combine compiler techniques and a runtime affinity prediction model to schedule kernels for asymmetric systems.

Like this prior work, AMS focuses on how to schedule threads across an asymmetric system to maximize system-wide performance. Unlike this prior work, AMS aims to schedule threads *with no program modifications and transparently to users*, similar to how OS-level schedulers manage symmetric systems, as recent work on OS for NDP systems advocates [7].

B. Cache, NUMA, and heterogeneity-aware thread schedulers

Scheduling applications under different constraints has been studied extensively in many contexts. The closest techniques to AMS are cache-contention-aware, NUMA-aware, and heterogeneous-core-aware schedulers.

Contention-aware schedulers [37, 49, 74] classify and colocate compatible applications under the same memory domain to avoid interference. For example, CRUISE [37] dynamically schedules single-thread applications in systems with multiple LLCs (e.g., multi-socket systems). CRUISE classifies applications into four categories according to their LLC behavior (insensitive, thrashing, fitting, and friendly). It then applies fixed scheduling policies to each class. As we will see, classification-based techniques do not work well in asymmetric systems, where application preferences (and thus classes) are affected by contention from other colocated applications. They also fail to handle same-class applications with different preference degrees (strong/weak).

NUMA-aware schedulers have different goals and constraints than asymmetry-aware schedulers. Since memory bandwidth is scarce in NUMA systems, prior work focuses on how to schedule threads across NUMA nodes to reduce bandwidth contention, similar to TOM for GPU-based asymmetric systems. Tam et al. [64] profile which threads have frequent sharing and place them in the same socket. DINO [13] clusters single-thread processes to equalize memory intensity, places clusters in different sockets, and migrates pages along with their threads. AsymSched [46] studies NUMA systems with asymmetric interconnects, migrating threads and pages to use the best-connected nodes. These NUMA schemes focus on off-chip memory bandwidth utilization, while AMS focuses on the asymmetry between deep and shallow hierarchies.

Finally, scheduling techniques for systems with heterogeneous cores [16, 69] focus on making the best use of asymmetric *core microarchitectures* like big.LITTLE. Due to the area and power limits of memory stacks, asymmetric systems often employ heterogeneous cores [25, 55], where the processor die has not only a deeper hierarchy but more powerful cores than the NDP stacks. AMS focuses on asymmetric *memory hierarchies*, but its performance model can be easily extended to consider other asymmetries. Specifically, we extend it with PIE's model [69] to handle asymmetry in both core types and memory hierarchies (Sec. V-B).

III. Baseline asymmetric system

To make our discussion concrete, we first describe the asymmetric system we target in this work, shown in Fig. 1 and Fig. 2. The processor die is similar to current multicores: each core has its own private caches, and all cores share a multi-megabyte last-level cache (LLC). The processor die is connected to several memory stacks using high-speed SerDes links. Each stack has multiple DRAM dies and a logic layer with several memory controllers and NDP cores. These NDP cores have only private caches due to the area and power constraints of the logic layer [2]. This system uses an interposer, but AMS would also work with other configurations, e.g., using off-package stacks.

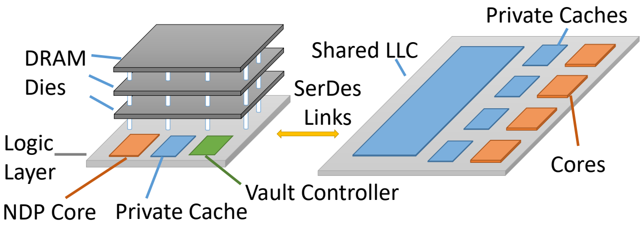


Fig. 2: Baseline system with an asymmetric memory hierarchy.

A. Memory stacks with NDP cores

We assume a memory stack design similar to HMC 2.0's [36]. Memory is organized in several vertical slices called *vaults*. Each DRAM vault is managed by and accessed through a vault controller in the logic layer. Vault controllers are connected via an all-to-all crossbar, as Fig. 3 shows. In addition to vault controllers, we assume the logic layer also has multiple low-power, lean OOO cores, such as Silvermont [40] or Cortex A57 [6]. Those cores have the same ISA as the processor die, so they can run programs without help from the main processor.

Like prior work in NDP systems using die-stacking techniques [25, 42, 55, 73], we conservatively assume the logic layer has a power and area budgets of 10 W and 50 mm^2 for components other than vault controllers and interconnect. This budget supports up to 4 NDP cores in the logic layer, connected to the system via the crossbar.

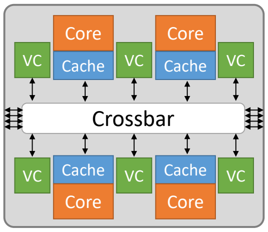


Fig. 3: Logic layer of each memory stack.

Why general-purpose cores? General-purpose cores make it easy for programmers to adapt their applications to this asymmetric memory hierarchy [25, 55]. Since both NDP cores and conventional cores use the same ISA, threads can migrate between hierarchies without recompilation or dynamic binary translation. This enables a smooth transition from traditional systems to asymmetric systems.

B. Coherence in NDP private caches

Deep and shallow hierarchies share the same physical address space, so their caches must be kept coherent to ensure correctness. However, using conventional directory-based coherence would either require NDP cores to check a remote directory even when performing local memory accesses, or require processor-die cores to check a memory-side directory on the memory stacks, adding area and traffic overheads that would limit the benefits of NDP [15].

To avoid these overheads, we perform *software-assisted coherence* similar to prior work [25]. Each virtual memory page is classified as either thread-private, shared read-only, or shared read-write. NDP cores can cache data from thread-private and shared read-only pages without violating coherence. For simplicity, shared read-write pages are considered uncacheable by NDP cores, which access them through the LLC to preserve coherence with processor-die caches. This classification technique has also been used to reduce coherence traffic [18] and to improve data placement in NUCA caches [8, 30]. We use the same dynamic classification mechanism as this prior work: Pages start private to the thread that allocates them. Upon a read from any other thread, the page is reclassified as shared read-only, and upon a write from any other thread, the page is reclassified as shared read-write. Reclassifications are done through TLB shootdowns [8, 30], which flush the page from private caches. Finally, when a thread moves from the processor die to an NDP core, its dirty LLC lines are flushed.

IV. Motivation

Although technology advances have enabled systems with asymmetric memory hierarchies, what is their potential benefit? Moreover, how critical is to schedule threads to the right hierarchy? In this section, we answer these questions by characterizing the benefits of an asymmetric system for memory-intensive applications. We also show that the ideal scheduler should *(i)* identify the right hierarchy for each thread, *(ii)* adapt to execution phases, and *(iii)* consider resource contention among threads.

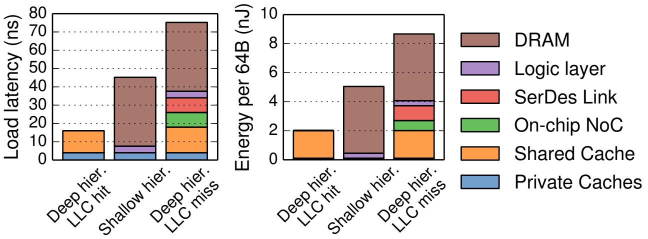


Fig. 4: Latency and energy of deep hierarchy LLC hits, shallow hierarchy memory accesses, and deep hierarchy memory accesses.

A. Asymmetry in access latency and energy

One of the key differences between deep and shallow hierarchies is the multi-megabyte LLC in the processor die. The performance offered by deep and shallow hierarchies largely depends on how frequently accesses hit in the LLC when using the deep hierarchy. Fig. 4 shows the latency and energy breakdowns of a memory reference in three situations with increasing costs: an LLC hit in the deep hierarchy, a stacked memory access in the shallow hierarchy, and an LLC miss (and off-chip stacked memory access) in the deep hierarchy (Sec. VII-A details the methodology for these costs).

Fig. 4 shows that an LLC miss from the deep hierarchy is the worst-case scenario: the system incurs the latency of an LLC lookup for no benefit, then it must traverse the on-chip network and off-chip SerDes link, reach the DRAM vault memory controller, wait for DRAM to serve the data, and finally wait for the response to make its way back. By contrast, a memory access from the shallow hierarchy (i.e., an NDP core) is 40% faster, because it is not subject to the LLC lookup, on-chip network, or SerDes link latencies. Nevertheless, stacked DRAM is significantly slower than on-chip SRAM, so an LLC hit in the deep hierarchy is 65% faster than a DRAM access from the shallow hierarchy. Energy breakdowns follow similar trends as latency breakdowns.

These costs show that shallow hierarchies complement deep ones, but do not uniformly outperform them. If a thread's working set does not fit in the LLC and fits in a local memory stack, a shallow hierarchy works best. But if the LLC can satisfy a substantial number of accesses, a deep hierarchy will be more attractive.

B. Effect of asymmetry on application preferences

We now simulate several memory-intensive applications to see how they can exploit memory asymmetry. We model a deep hierarchy with 32 KB private L1s, 256 KB L2s, and a shared 16 MB LLC in the conventional processor. The shallow hierarchy only has private L1 and L2 caches (see Sec. VII-A for methodology details). Both hierarchies use 2-way OOO cores. We later evaluate heterogeneous cores and multithreaded applications; our goal here is to study memory asymmetry independently. These cores with their private caches consume less than 2.5W and $10mm^2$ per core, which is practical to fabricate in the logic layer of 3D-stacked DRAM [2].

We simulate the 18 memory-intensive SPEC CPU2006 benchmarks that have >5 L2 MPKI and 8 benchmarks from the Problem-Based Benchmark Suite [61], which contains memory-intensive graph algorithms. Since the choice of hierarchy affects both performance and efficiency, we use performance per Joule (Perf/J), i.e., the inverse of energy-delay product, to characterize the differences across hierarchies.

Applications have strong hierarchy preferences. Fig. 5 shows the Perf/J of representative applications when running on the shallow hierarchy, relative to the Perf/J when running on the deep hierarchy. Some applications strongly prefer the deep hierarchy. For example, `xalancbmk` has a working set of about 6 MB, so it benefits significantly from the 16 MB LLC in the deep hierarchy. `xalancbmk`'s Perf/J on the shallow hierarchy is almost 2× (-50%) worse than on the deep hierarchy. By contrast, `soplex` has a much larger working set that cannot fit in the 16 MB LLC. It thus always prefers the shallow hierarchy, which provides 2× higher Perf/J than the deep hierarchy.

Across all applications, always using the shallow hierarchy improves gmean Perf/J over the deep hierarchy by 15%. However, always using the hierarchy that offers the best average Perf/J for each application improves gmean Perf/J by 31% (Fig. 5, right), doubling the improvement achieved by always using the shallow hierarchy. This result shows that it is important to schedule applications to the right hierarchy.

Dynamic scheduling unlocks the full potential of asymmetric hierarchies. Some applications have multiple phases, each with different memory behaviors and working sets. For example, as shown in Fig. 6, `GemsFDTD` prefers the shallow hierarchy before it reaches 53 billion instructions, and prefers the deep hierarchy afterward. Therefore, running `GemsFDTD` on either hierarchy statically does not yield major benefits.

To show the impact of these dynamic preferences, we implement a dynamic scheduler that always runs the application on the best hierarchy for each 50 ms phase. This substantially improves applications like `GemsFDTD` and `refine`. Of the 26 applications, 12 (46%) prefer different hierarchies over different phases. Overall, our dynamic scheduler improves gmean Perf/J by 40%, more than the 31% achieved by static decisions.

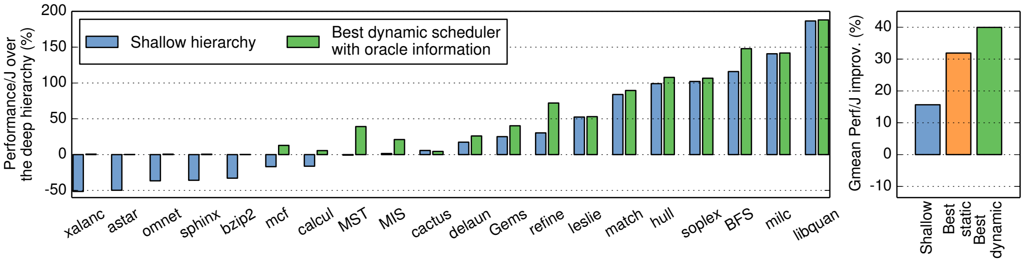


Fig. 5: Performance per Joule (Perf/J) relative to the deep hierarchy. Higher is better.

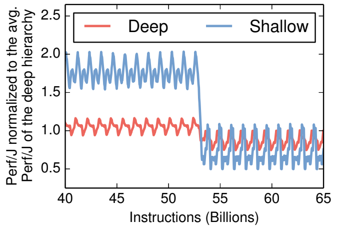


Fig. 6: Perf/J traces of `GemsFDTD`, relative to the deep hierarchy.

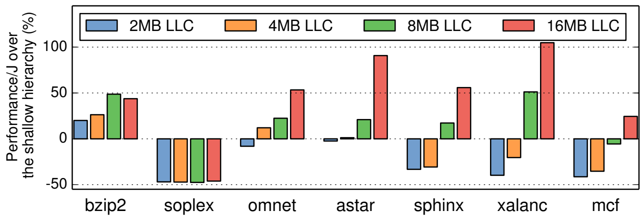


Fig. 7: Performance per Joule of deep hierarchies with different LLC sizes, relative to the shallow hierarchy.

Application preferences are sensitive to contention. The above results consider a single application, but in real-world workloads, multiple applications are colocated in a single system and compete for shared resources, such as LLC capacity. To study this effect, we sweep the LLC size of the deep hierarchy to mimic capacity contention among applications.

Fig. 7 shows the Perf/J improvement of deep hierarchies with different LLC capacities over the shallow hierarchy. We select 7 representative benchmarks. The first application (`bzip2`) always benefits from deep hierarchies due to its cache-friendly working sets. The next application (`soplex`) instead always prefers a shallow hierarchy due to its streaming behavior.

By contrast, the other applications have very different preferences across LLC capacities. For example, `omnetpp` benefits from LLCs $\geq$ 4 MB, while `sphinx3` benefits from LLCs $\geq$ 8 MB, and `mcf` only benefits from a 16 MB LLC. And even when applications prefer the deep hierarchy, their degree of preference also changes significantly with available capacity (e.g, 8 MB vs. 16 MB for `astar`).

This result shows that when applications are colocated, resource contention can dramatically change their preferences. It also shows why prior classification-based schedulers can cause pathologies with asymmetric hierarchies. For example, CRUISE can first classify and schedule `mcf` to the deep hierarchy, then later schedule others that cause capacity contention and make `mcf` strongly prefer the shallow hierarchy.

In summary, these results show that applications have strong preferences for the type of hierarchy, and that these preferences change over time and with available resources. These observations guide AMS's design.

V. AMS: Adaptive Scheduling for Asymmetric Memory Systems

AMS realizes the potential of asymmetric memory hierarchies by accounting for contention and dynamic behavior when mapping threads to cores. The key insight behind AMS is that the problem of modeling a thread's preferences to different memory hierarchies on-the-fly and under contention bears a strong resemblance to the dynamic cache partitioning problem. Therefore, unlike other schedulers, AMS leverages both working set profiling techniques and allocation algorithms that were originally proposed for cache partitioning, even though AMS does not perform cache partitioning.

Fig. 8 shows an overview of AMS. AMS has both hardware and software components. AMS hardware consists of simple hardware utility monitors [56] to profile per-thread *miss curves*, which reflect the number of misses a thread would incur under different cache sizes. AMS software then uses this information to remap threads periodically, on each scheduling quantum (every 50 ms in our implementation). This process consists of two phases. In the first phase, AMS software uses miss curves to accurately estimate a thread's performance on both shallow and deep hierarchies and under different amounts of LLC contention (Sec. V-A). Miss curves allow AMS software to produce these estimates without trial and error (i.e., AMS does not run a thread in both hierarchies to infer its preferences).

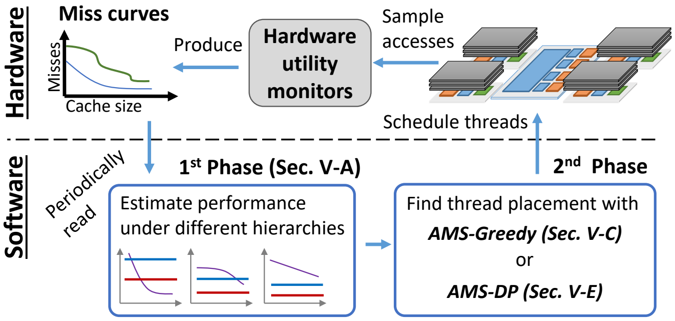


Fig. 8: AMS overview.

In the second phase, AMS software uses these estimates to find a thread placement that achieves high system-wide performance. We present two thread placement algorithms: AMS-Greedy performs multiple rounds of cache partitioning and uses its outcomes to progressively and greedily map threads to hierarchies (Sec. V-C), while AMS-DP leverages dynamic programming to explore the full space of configurations efficiently, finding the optimal schedule given the predicted preferences (Sec. V-E). Though AMS-DP is more expensive than AMS-Greedy, it is practical to use in small systems and serves as AMS-Greedy's upper bound.

To simplify the explanation, we first focus on systems with homogeneous cores running single-thread applications. Sec. V-B extends AMS to heterogeneous cores, Sec. V-D extends AMS to multithreaded workloads, and Sec. V-F discusses other scenarios, such as oversubscribed systems.

A. Estimating performance under asymmetric hierarchies

To model thread preferences, it is crucial to understand the utility of the processor die's LLC for each thread. To this end, AMS leverages UMONs [56] to produce miss curves. Each UMON is a set-associative tag array with per-way hit counters. UMONs leverage LRU's stack property to profile different cache sizes simultaneously. AMS adds a 4 KB UMON to each core. Each UMON samples private cache misses and produces a miss curve that covers the range of possible capacities available to the thread (from no capacity to the full LLC). We choose UMONs for their low overhead and high accuracy, but AMS could use other miss curve profiling techniques [9, 23, 65].

AMS models thread performance using total memory access latency, a cost function derived with miss curves. AMS can also optimize other cost functions, such as core cycles, as we will show in Sec. V-B. AMS uses miss curves to derive cost

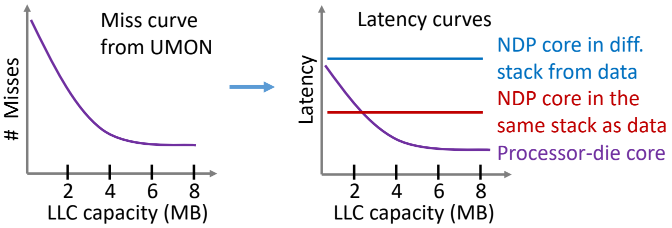


Fig. 9: Example latency curves for processor-die and NDP cores.

functions for all relevant scenarios. Since NDP and processor-die cores have the same private caches, we focus on memory references after the private cache levels.

If a thread runs on a processor-die core, its latency depends on how much LLC capacity is available. Specifically, the total latency in cycles as a function of LLC capacity s, which we call the *latency curve*, is:

$$L^{proc}(s) = A \cdot Lat^{LLC} + M(s) \cdot Lat^{mem,proc}$$

where A is the number of accesses that miss in the private cache levels (i.e., the number of LLC accesses in this case), Lat^{LLC} is the average latency of a single LLC access, $M(s)$ is the number of LLC misses given capacity s, and $Lat^{mem,proc}$ is the average latency of a single access to off-chip main memory. $M(s)$ is the miss curve, and A, the number of LLC accesses, is simply $A = M(0)$ (with no LLC capacity, all LLC accesses miss). Note that this formula covers the *total* amount of cycles spent in memory references in a given interval, not the average latency. This is because it is important to account for the rate at which accesses happen, not only their unit cost. For example, a thread that has infrequent misses from its private caches will have low values for A and $M(s)$, and thus will incur a low penalty from different thread placements, even if most of the few accesses it performs miss in the LLC.

If the thread runs on an NDP core, all A private cache misses go to memory. Thus, the thread's latency curve is simply $L^{NDP} = A \times Lat^{mem,NDP}$. Because NDP cores do not access a shared LLC, this curve does not change with s. However, because the system has multiple memory stacks, the average latency per memory access, $Lat^{mem,NDP}$, depends on the core's stack as well as the placement of the application's data. We use a simple algorithm that makes most NDP memory accesses local by biasing data placement to particular stacks. We describe this algorithm in Sec. VI. AMS simply computes $Lat^{mem,NDP}$ as the weighted average of the number of application pages on each stack, times the latency to access that stack.

Fig. 9 shows three example latency curves for a particular thread: the curve for processor-die cores and two curves for two NDP cores on different stacks. These latency curves encode a thread's preferences under different scenarios. For example, if the LLC is very contended and leaves no capacity, this thread prefers NDP cores. But with 2 MB of LLC capacity, only the NDP core closest to its data is better (i.e., has a lower latency). Finally, if the thread can use over 4 MB of LLC capacity, it prefers to run on a processor-die core with the deep hierarchy. Moreover, the latency difference between $L^{proc}(s)$ and L^{NDP} at each point also indicates how strong the preference is.

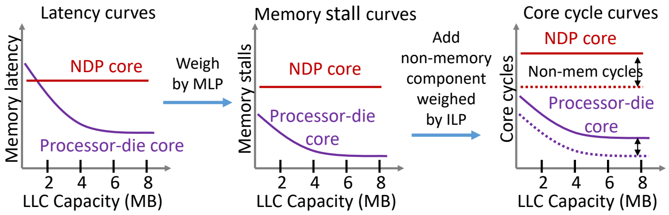


Fig. 10: To handle heterogeneous cores, AMS transforms the latency curves into CPI curves using PIE's performance model.

We find that this model matches Fig. 7's results. Therefore, this model lets AMS predict how applications perform under different decisions without directly profiling or sampling their performance under various colocation combinations.

B. *Handling heterogeneous cores*

Although AMS focuses on asymmetric hierarchies, we must also consider core asymmetry, as NDP cores are typically simpler than processor-die cores. Fortunately, it is easy to extend AMS to handle heterogeneous cores. We combine AMS's model with the cycles-per-instruction (CPI) estimation techniques from PIE [69], which targets heterogeneous cores but assumes a *symmetric memory hierarchy*.

To map threads across heterogeneous cores, PIE estimates each thread's CPI on different core types. Its model consists of a memory component, estimated with the core's memory-level parallelism (MLP), and a non-memory component, estimated with the core's instruction-level parallelism (ILP).

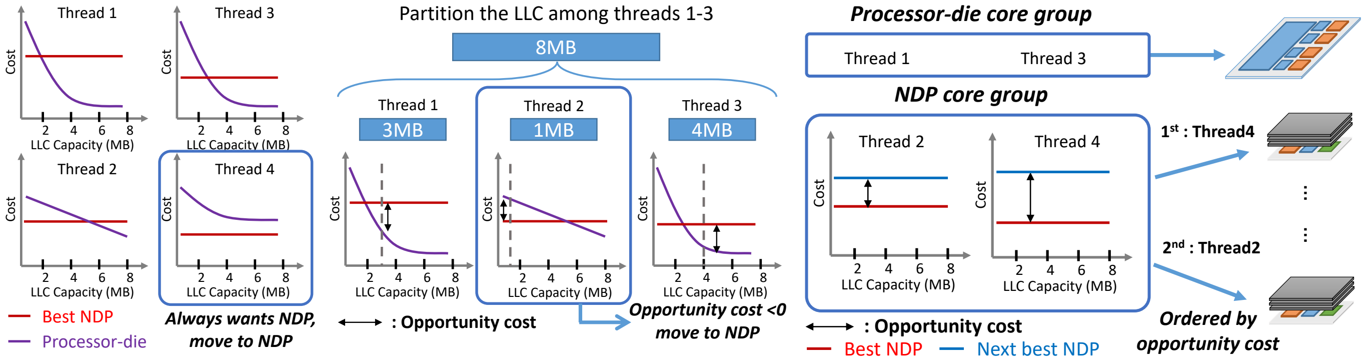
AMS with PIE models the total cycles spent across core types and LLC sizes. It thus works on *core cycle curves* instead of memory latency curves. Fig. 10 shows this transformation. The memory component of each curve comes from AMS's latency curve weighted using PIE's estimated MLP, and the non-memory component uses PIE's estimated ILP. This requires collecting non-memory stall cycles using standard hardware counters (as in PIE). Core cycle curves unify asymmetries in *both* cores and memory hierarchies. They can be transformed into other cost curves as needed (e.g., using time instead of cycles to model cores running at different frequencies).

C. *AMS-Greedy: Mapping threads via cache partitioning*

Given the cost function (total latency or core cycle) curves of all threads in the system, we can evaluate a schedule by calculating the *total cost* it incurs. Finding the best mapping in asymmetric hierarchies can be modeled as minimizing the total cost over all possible thread mappings. We present two optimizers for this problem, one based on greedy optimization, and another based on dynamic programming (Sec. V-E).

AMS-Greedy works by performing multiple rounds of cache partitioning. On each round, the algorithm identifies the threads that benefit the least from the deep hierarchy and schedules them away from the processor die. Fig. 11 illustrates AMS-Greedy's algorithm with a 4-thread example.

AMS-Greedy begins with all threads mapped to the deep hierarchy (the processor die). We denote the cost curves for thread i as $C_i^{proc}(s)$ for the **processor die** and $C_i^{best\,NDP}$ for the **best NDP stack**. First, AMS-Greedy finds threads that always



(a) Find threads that always want the NDP hierarchy.

(b) Partition the LLC to find threads with minimal opportunity cost to use the NDP hierarchy.

(c) Schedule threads in the NDP core group ordered by maximum opportunity cost.

Fig. 11: An example of how AMS-Greedy schedules 4 threads with different latency curves.

prefer the shallow hierarchy. These are the threads for which $C_i^{proc}(s) > C_i^{best\,NDP}$ across all possible LLC capacities s (e.g., thread 4 in Fig. 11a). AMS-Greedy moves these threads off the processor die.

The remaining threads can benefit from the LLC if they have enough capacity available. But there may not be enough LLC capacity or enough processor-die cores to satisfy all threads. Therefore, AMS-Greedy progressively moves threads to NDP cores, stopping either when the remaining threads have sufficient LLC capacity and cores or when NDP cores fill up.

AMS-Greedy uses cache partitioning for this goal. Specifically, it partitions the LLC using the Peekahead algorithm [8] (a linear-time implementation of quadratic-time UCP Lookahead [56]). AMS-Greedy uses the processor-die cost curves ($C_i^{proc}(s)$ for thread i) to drive the partitioning. This way, the partitioning algorithm finds a set of partition sizes s_i that seeks to minimize total cost ($\sum C_i^{proc}(s_i)$). For example, in Fig. 11b, threads 1, 2, and 3 receive partition sizes of 3, 1, and 4 MB.

Intuitively, partitioning naturally finds threads that should give up the processor die. For example, if a thread has no capacity after partitioning the LLC, that means fitting its working set is too costly compared to other options. We should thus move it to an NDP core and let others share the LLC.

Thus, AMS-Greedy ranks threads by their *opportunity cost*, the extra cost they incur when moving to the best NDP core:

$$\mathit{Opportunity\ cost}_i = C_i^{best\,NDP} - C_i^{proc}(s_i)$$

AMS-Greedy moves all threads with a negative opportunity cost to NDP cores as long as NDP cores are not oversubscribed. These threads have lower cost in NDP cores than with s_i LLC capacity (e.g., thread 2 in Fig. 11b). If there is no such thread but the processor die is still oversubscribed, AMS-Greedy moves the thread with the smallest opportunity cost.

If after a round of partitioning and moving threads there are still more threads than the number of processor-die cores, AMS-Greedy performs another round of partitioning and movement *among the remaining threads*. This process repeats until the processor die is not oversubscribed.

Finally, AMS-Greedy tries to map the threads on NDP cores to their most favorable stack. Threads are again prioritized by opportunity cost: threads with the largest difference between their latencies in the **best** and **next-best** NDP stacks are placed first. For example, in Fig. 11c, thread 4 has a larger opportunity cost than thread 2 and is mapped first.

AMS-Greedy works well because it shares the same goal as partitioning: identifying the threads that benefit the least (or not at all) from the LLC. AMS-Greedy leverages these algorithms to minimize total cost at each round. Greedily moving threads out may not yield the optimal solution because the problem is not convex. Nonetheless, we find AMS-Greedy generates high-quality results because opportunity cost captures the degree of preference accurately. AMS-Greedy also scales well: its runtime complexity is $O(N^2 S)$, where N is the number of threads and S is the number of LLC segments ($O(NS)$ per round of cache partitioning [8] and $O(N)$ for up to N rounds).

Prior work has also leveraged partitioning algorithms for other purposes, such as tailoring the cache hierarchy to each application [67] and performing dynamic data replication in NUCA caches [68]. AMS-Greedy shares similar insights in using miss curves and partitioning, but it focuses on scheduling in asymmetric systems and does not partition the cache.

D. *Handling multithreaded workloads*

We have so far considered only single-threaded processes. AMS can handle multithreaded processes with extra UMONs and simple modifications to AMS-Greedy.

Multithreaded workloads share data within the process, so per-core UMONs may overestimate the size of the working set. To solve this, AMS adds an extra UMON per core to profile shared data. To distinguish private vs. shared data, we leverage the per-page data classification scheme used for coherence (Sec. III-B). Cache misses to private data are sampled to the per-core UMON, and misses to shared data are sampled to a UMON shared by all threads in the process (although this UMON is not local to the core, this imposes negligible traffic because only ∼1% of the misses are sampled).

Using the number of private cache misses to thread-private data and shared data, AMS first classifies processes as thread-private-intensive or shared-intensive. AMS then treats thread-private-intensive processes as multiple independent threads. This is sensible because these processes have little data sharing, so they behave similarly to multiple single-threaded processes.

By contrast, AMS-Greedy groups all the threads of each shared-intensive process into a single unit when making decisions. The algorithm considers the miss curve for shared data only, and performs placement decisions for all its threads at once (considering the opportunity cost of all threads). This ensures that threads that share data intensively stay together.

NDP cores access shared read-write data pages through the LLC for coherence. This makes the processor die preferable under our model for workloads dominated by shared read-write data. However, many multithreaded applications are well-structured: threads write mostly disjoint data and mainly use thread-private or shared read-only pages. These applications often prefer NDP cores.

E. AMS-DP: Mapping threads via dynamic programming

Dynamic programming (DP) [10, 17] is an optimization technique that solves a problem recursively, by dividing it into smaller subproblems. Each subproblem is solved only once, and its result is memoized and reused whenever the subproblem is encountered again. Memoization lets DP explore the full space of possible choices efficiently. Because DP considers all possible choices, it finds the globally optimal solution. By contrast, greedy algorithms take locally optimal decisions but may end up with a globally suboptimal one. However, not all problems are amenable to DP: the problem must have the property that an optimal solution can be computed efficiently given optimal solutions of its subproblems. Often, the difficulty lies in casting the problem in a way that meets this property.

Our second AMS variant, AMS-DP, leverages dynamic programming to find the optimal solution. We again exploit the similarities between scheduling and cache partitioning by building on Sasinowski et al. [59], who show that DP can solve cache partitioning optimally in polynomial time.

Cache partitioning can be solved with DP because it has discrete decisions, at the size of cache segments (e.g., cache ways or lines). This property allows dividing the partitioning problem into subproblems. For example, partitioning a 4 MB cache among eight threads can be divided into partitioning two caches (e.g., of 2 MB each or of 1 MB + 3 MB) to two groups of four threads. The smallest subproblem is just allocating some amount of capacity to a single thread.

Similarly, scheduling threads to cores also has discrete decisions. One thread can occupy only one core and leave the rest to other threads. This property allows dividing a scheduling problem into subproblems that schedule smaller groups of threads across smaller systems. The smallest subproblem is scheduling a thread to a single core, given C^{NDP}, $C^{proc}(s)$, and some amount s of remaining LLC capacity.

Our insight is that since these two problems have discrete decisions, we can combine them together and solve a bigger DP problem to partition the cache and schedule threads at the same time, which is very similar to scheduling in asymmetric systems as we discussed. Thus, solving this DP problem leads to the optimal partitioning and scheduling.

For the rest of the section, we use the same terminology as Sasinowski et al. [59]. See [10] for more details on DP.

The key recurrence relation that lets Sasinowski et al. use DP is as follows. If $M_{i,j}$ is the minimum cost achieved by partitioning j segments among the first i threads, and $C_i(s_i)$ is the cost of the i^{th} thread when allocated s_i segments, then:

$$M_{i,j} = \min_{s_i}\{M_{i-1,j-s_i} + C_i(s_i)\}$$

This recurrence shows that the minimum cost $M_{i,j}$ is the minimum of all possible combinations of subproblems: the cost of thread i with s_i cache segments and the minimum cost of using $j - s_i$ cache segments for the first $i-1$ threads. By solving each $M_{i,j}$ bottom-up, we reach the optimal partitioning for $M_{N,S}$, where N is the number of threads and S is the number of cache segments in the system.

In our case, we want to not only partition the cache (conceptually, to prevent cache contention) but also to schedule threads. Therefore, we extend the recurrence by *adding dimensions for processor-die cores and NDP cores*. Cores are just another type of discrete resource to allocate. However, different cores, even NDP cores in different stacks, should be treated differently. Suppose the system has one processor die and one NDP stack. We define $M_{i,j,k_{proc},k_{ndp}}$ as the minimum cost when partitioning j cache segments to the first i threads *and scheduling them with exactly k_{proc} processor-die cores and k_{ndp} NDP cores*. The recurrence above becomes:

$$M_{i,j,k_{proc},k_{ndp}} = \min\{\min_{s_i}\{M_{i-1,j-s_i,k_{proc}-1,k_{ndp}} + C_i^{\text{proc}}(s_i)\},\\ M_{i-1,j,k_{proc},k_{ndp}-1} + C_i^{\text{NDP}}\}$$

This recurrence states that, if we were to schedule the i^{th} thread on a processor-die core, we can allocate some LLC capacity s_i to it and leave the remaining capacity to the first $i-1$ threads. This decision makes total cost to be the cost $C_i^{proc}(s_i)$ for thread i plus the minimum cost of scheduling the first $i-1$ threads with $j - s_i$ capacity.

If the thread is instead scheduled on an NDP core, it takes no LLC capacity, and incurs cost C_i^{NDP} for the thread plus the minimum cost to schedule the first $i-1$ threads with 1 fewer NDP core available. Finally, the minimum cost to schedule i threads is simply the minimum of those two scheduling choices.

This recurrence considers a single NDP stack, but adding more stacks as extra dimensions is straightforward (e.g., $M_{i,j,k_{proc},k_{ndp,1},k_{ndp,2}}$ with two stacks). This lets each thread use its different costs to each stack.

Using this recurrence, AMS-DP performs standard bottom-up DP to find the optimal thread-to-core mapping. While conceptually simple, AMS-DP scales poorly: every group k (i.e., processor-die or NDP stack) adds a new dimension to the DP algorithm. This causes $O(N \cdot S \cdot k_{proc} \prod_t k_{ndp,t})$ running time, where N is the number of threads and S is the number of cache segments. Thus, AMS-DP is practical only in small systems. On larger systems, AMS-DP serves as the upper bound, but simpler techniques like AMS-Greedy are needed.

F. Discussion

Our evaluation focuses on long-running, memory-intensive batch workloads, but AMS should work in other scenarios with minor changes. First, in oversubscribed systems with more runnable threads than cores, AMS only needs to consider the active threads in each quantum. A thread's miss curve can be saved when it is descheduled so that the thread can be mapped to the right core when it is rescheduled later. Second, kernel threads and short-lived threads or processes can evict any long-running thread in the system. Since they run for a fraction of the scheduling quantum, their impact is minimal. Finally, to handle latency-critical workloads with real-time needs, AMS can be combined with techniques that partition the cache to maintain SLOs instead of maximizing throughput, such as Heracles [48] or Ubik [41].

VI. Data placement for asymmetric hierarchies

NDP cores are most effective when they access their local memory stack. This requires adopting a data placement scheme that minimizes remote accesses.

Data placement is a widely studied topic in non-uniform memory access (NUMA) systems. Prior work [13, 21, 70] has proposed various data migration and replication techniques to reduce remote accesses. Other NUMA work [1, 46] focuses on balancing available bandwidth among applications.

The key difference between NUMA and NDP systems is bandwidth. Because memory bandwidth is scarce, NUMA systems are limited by bandwidth to local memory, and prior work finds that it is important to spread pages evenly across NUMA nodes to reduce bandwidth contention [21, 46]. By contrast, NDP systems suffer a different problem: the NDP cores in each stack enjoy plentiful bandwidth to the memory stacked directly atop them [26], but the bandwidth *across* stacks is very limited [34]. In this case, it is more important to reduce *inter-stack* traffic than *intra-stack* traffic, so the key constraint is ensuring that NDP cores have local accesses.

Since relocating pages is expensive, our data placement algorithm avoids migrating pages and uses simple heuristics to keep data local. Its goal is to keep pages from the same thread in as few stacks as possible, so that NDP cores have most local accesses. When a thread starts, the system builds up a dynamic preference list of memory stacks in the order from which we fulfill memory allocations. This preference list is refreshed when a memory stack is depleted.

When a new thread starts, AMS picks the memory stack with the greatest remaining capacity as the most preferred source. This ensures that threads that can benefit from the shallow hierarchy are able to leverage it and those that prefer the deep hierarchy are not penalized. Next on the list are the nearby stacks. In Fig. 1, these are those on the same side of the processor die. In the example in Fig. 12, an NDP-friendly application can be scheduled on an NDP core in stacks 1 or 2 to have high-bandwidth and low-latency accesses. Finally, if stacks on one side are exhausted, we allocate pages to stacks on the opposite side of the chip.

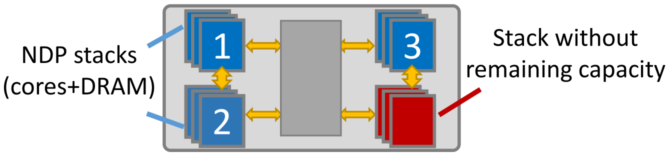


Fig. 12: Example data placement preference list. Memory stacks with free pages are shown in **blue** and full stacks in **red**.

TABLE I: Configuration of the simulated system.

Cores	16 cores (8 processor die + 4×2 NDP), x86-64, 2.5 GHz *Silvermont-like OOO* [40]: 8B-wide ifetch, 2-level bpred with 512×10-bit BHSRs + 1K×2-bit PHT, 2-way issue, 36-entry IQ, 32-entry ROB, 32-entry LQ/SQ *Haswell-like OOO* [29]: 16B-wide ifetch, 2-level bpred with 1K×18-bit BHSRs + 4K×2-bit PHT, 4-way issue, 60-entry IQ, 192-entry ROB, 72-entry LQ, 42-entry SQ
L1 caches	32 KB, 8-way set-associative, split data and instruction caches, 3-cycle latency; 15/33 pJ per hit/miss [51]
L2 caches	256 KB private per-core, 8-way set-associative, inclusive, 7-cycle latency; 46/93 pJ per hit/miss [51]
Coherence	MESI, 64 B lines, no silent drops; sequential consistency
Last-level cache	16 MB, 2 MB bank per core, 32-way set-associative, inclusive, 30-cycle latency, TA-DRRIP [38] replacement; 945/1904 pJ per hit/miss [51]
Stacked DRAM	4 GB die, HMC 2.0-like organization, 8 vaults per stack, 64-bit data bus, 6-cycle all-to-all crossbar in the logic layer [36], 2 pJ/bit internal, 8 pJ/bit logic layer [25, 73]
Stack links	160 GBps bidirectional, 10-cycle latency, including 3.2 ns for SerDes [43], 2 pJ/bit [43, 55]
3D DRAM timings	t_{CK}=1.6 ns, t_{CAS}=11.2 ns, t_{RCD}=11.2 ns, t_{RAS}=22.4 ns, t_{RP}=11.2 ns, t_{WR}=14.4 ns

Adopting more sophisticated data placement techniques as in prior NUMA work [21, 46] could increase AMS's benefits. For example, the system could dynamically migrate data to reduce cross-stack accesses from NDP threads. These techniques are orthogonal to AMS, so we leave them to future work.

VII. Evaluation

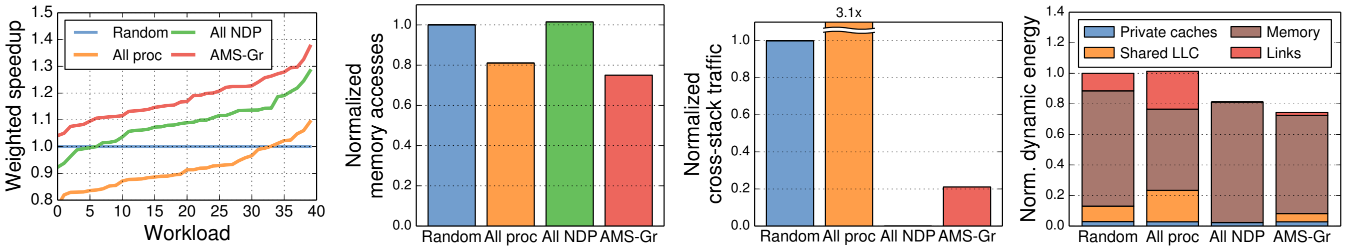
A. Methodology

Modeled system: We perform microarchitectural, execution-driven simulation using zsim [58], and model a 16-core system. The processor die has 8 cores, with private 32 KB L1 and 256 KB L2 caches. All 8 cores share a 16 MB LLC that uses the TA-DRRIP [38] thread-aware replacement policy. The processor die is connected to four NDP stacks via SerDes links, as shown in Fig. 1. Each stack has 4 GB of DRAM and 2 NDP cores with only private caches. Table I details the system's configuration.

We consider systems with both homogeneous and heterogeneous cores. Our homogeneous-core system (Secs. VII-B to VII-D) uses 2-wide OOO cores similar to Silvermont [40]. Our heterogeneous-core system (Sec. VII-E) instead uses 4-wide OOO cores similar to Haswell [29] in the processor die.

Schedulers: We first compare AMS-Greedy against three simple schedulers in Sec. VII-B and Sec. VII-C. First, we use *Random* scheduling as the baseline to which we compare other schedulers. This is a better baseline than the WAS (worst application scheduler) baseline in prior work [37, 69]. Second, *All proc* always runs threads on processor-die cores. Third, *All NDP* always runs threads on NDP cores.

In Sec. VII-D, we compare AMS-Greedy against AMS-DP and a more sophisticated scheduler, CRUISE-NDP. We derive CRUISE-NDP by adapting CRUISE [37] to asymmetric hierarchies. Each scheduling quantum, CRUISE-NDP classifies threads as either cache-insensitive, cache-friendly, cache-fitting,



(a) Weighted speedup. (b) Memory accesses. (c) Cross-stack traffic. (d) Data movement energy.

Fig. 13: Simulation results for different schedulers on 8-app mixes.

or thrashing using the same heuristics as CRUISE (all the necessary information for CRUISE is gathered using UMONs too). CRUISE-NDP then maps thrashing threads to NDP cores, cache-friendly and fitting threads to processor-die cores (prioritizing friendly over fitting), and finally backfills the remaining cores with insensitive threads.

We model migration overheads and find remapping every 50 ms causes negligible overheads, similar to PIE's findings.

Workloads: Our workload setup mirrors prior work [9]. We simulate mixes of SPEC CPU2006 apps and multithreaded apps from SPEC OMP2012 and PARSEC [11]. We evaluate scheduling policies under 50% load (*8 cores occupied*) and 100% load (*16 cores occupied*). We use the 18 SPEC CPU2006 apps with ≥5 L2 MPKI (as in Sec. IV) and fast-forward all apps in each mix for 30 B instructions. We use a fixed-work methodology and equalize sample lengths to avoid sample imbalance, similar to FIESTA [32]. Each application is then simulated for 2 B instructions. Each experiment runs the mix until all apps execute at least 2 B instructions, and we consider only the first 2 B instructions of each app to report performance.

For multithreaded apps, since IPC is not a valid measure of work [5], to perform a fixed amount of work we instrument each application with *heartbeats* that report global progress (e.g., when each timestep or transaction finishes) and run each application for as many heartbeats as *All proc* completes in 2 B cycles after the serial region.

Metrics and energy model: We follow prior work in scheduling techniques and use weighted speedup [63] as our performance metric. We use McPAT 1.3 [47] to derive the energy of cores at 22 nm, and CACTI [51] for caches at 22 nm. We model the energy of 3D-stacked DRAM using numbers reported in prior work [34, 43, 60]. Dynamic energy for NDP accesses is about 10 pJ/bit. We assume that each SerDes link consumes 2 pJ/bit [43, 55].

B. AMS finds the right hierarchy

We first evaluate AMS-Greedy in an undercommitted system with homogeneous cores (8 apps on 16 Silvermont cores) to focus on the effect of memory asymmetry. Fig. 13a shows the distribution of weighted speedups over 40 mixes of 8 randomly chosen memory-intensive SPEC CPU2006 apps. Each line shows the results for a single scheduler over the *Random* baseline. For each scheme, workload mixes (the x-axis) are sorted according to the improvement achieved.

In each mix, different applications want different hierarchies. *All proc* improves only 7 mixes and hurts performance on the other 33 because it never leverages the NDP capability of the asymmetric system. On average, its weighted speedup is 8% worse than the *Random* baseline. *All NDP* benefits some applications significantly (e.g., `soplex` in Fig. 5). However, it sometimes hurts applications that prefer deep hierarchies because it never leverages the LLC. On average, *All NDP* improves weighted speedup by 9% over the baseline.

AMS-Greedy finds the best hierarchy for each application and schedules them accordingly. It thus never hurts performance and improves weighted speedup by up to 37% and by 18% on average over the *Random* baseline.

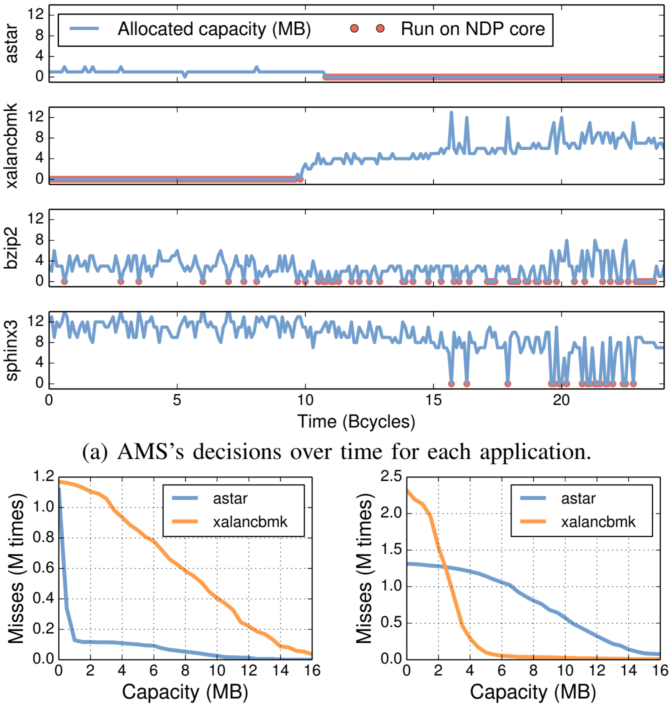
AMS-Greedy achieves significant speedups because it leverages both hierarchies efficiently. Figs. 13b–d give more insight on this. AMS-Greedy uses the LLC as effectively as *All proc* and reduces memory accesses by 26% over the baseline (Fig. 13b). AMS-Greedy also schedules applications to leverage the system's NDP cores when beneficial. It thus eliminates 80% of the cross-stack traffic (Fig. 13c). Overall, AMS-Greedy reduces dynamic data movement energy by 25% over the baseline, while *All proc increases* it by 2% and *All NDP* reduces it by 18% (Fig. 13d).

C. AMS adapts to application phases

Next, we show how AMS-Greedy adapts to phase changes by examining a 4-app mix. In this workload, we include two applications, `astar` and `xalancbmk`, that have distinct memory behaviors across two long phases, and two other applications, `bzip2` and `sphinx3`, that have short and fine-grained variations over time. To observe time-varying behavior, we simulate this mix for 25 Bcycles.

Fig. 14a shows the traces of scheduling and capacity allocation decisions of AMS-Greedy for all four apps. The upper two traces show that AMS-Greedy takes different decisions for `astar` and `xalancbmk` before and after 10 Bcycles. Before 10 Bcycles, `astar` is mapped on the processor die and `xalancbmk` is mapped to an NDP core. After 10 Bcycles, both change the hierarchy they prefer. The other two apps are more fluctuating, but prefer the deep hierarchy more often.

To explain this phenomenon, Fig. 14b and Fig. 14c show the sampled miss curves for `astar` and `xalancbmk` at 5 and 15 Bcycles. At 5 Bcycles, `astar` has a small working set (the sharp drop around 1 MB), but `xalancbmk` has a large working set (14 MB). Therefore, AMS-Greedy fits the working sets of `astar`, `bzip2`, and `sphinx3` in the LLC, and schedules `xalancbmk` to an NDP core because it prefers the shallow hierarchy when capacity is limited. At 15 Bcycles,



(a) AMS's decisions over time for each application.

(b) Miss curves at 5 Bcycles.

(c) Miss curves at 15 Bcycles.

	astar	xalancbmk	bzip2	sphinx3	weighted speedup
AMS	**1.24**	**1.04**	**1.08**	**1.22**	**1.14**
All NDP	1.07	0.79	0.85	0.86	0.89

(d) Per-app speedups relative to *All proc*.

Fig. 14: Simulation results for the case study.

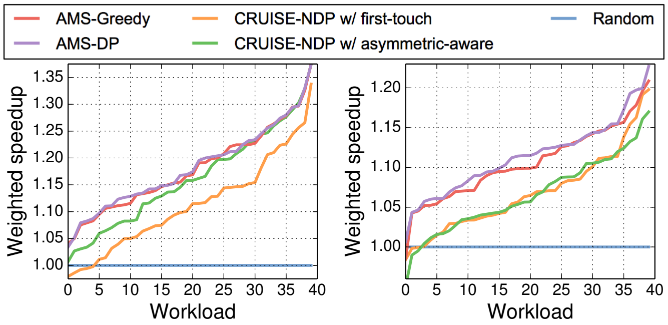
the memory behavior of `astar` and `xalancbmk` essentially switches. `xalancbmk` now has a smaller working set than `astar` (misses drop around 4 MB). Therefore, AMS schedules `astar` to an NDP core and lets `xalancbmk` share the LLC with the other two applications.

This experiment shows that AMS adapts to program phases. Fig. 14d shows AMS's per-app improvements and weighted speedup for this mix, which is 14% better than *All proc*.

D. AMS handles resource contention well

We now compare AMS-Greedy with AMS-DP and CRUISE-NDP (see Sec. VII-A) under different system loads to understand the quality and robustness of these algorithms. It also shows how our data placement for asymmetric hierarchies influences other schedulers.

Fig. 15 shows the weighted speedups over 40 random mixes of 8 apps (50% core utilization) and 16 apps (100% core utilization) for two variants of AMS and CRUISE-NDP with two data placement algorithms: first-touch and asymmetry-aware. The undercommitted system (Fig. 15a) has little resource contention among applications. However, CRUISE-NDP with first-touch, a simple algorithm that always places new pages in the closest stack, hurts performance for 5 mixes and improves weighted speedup by only 11%. AMS-Greedy outperforms CRUISE-NDP by up to 18% and by 7% on average. CRUISE with our asymmetry-aware data placement performs closer to AMS and improves performance by 16%. AMS-Greedy performs very close to AMS-DP, improving performance by 18% on average. This shows that under 50% load, AMS-Greedy very often finds the optimal solution that AMS-DP achieves.



(a) 8-app mixes (50% load).

(b) 16-app mixes (100% load).

Fig. 15: Performance of AMS-Greedy/-DP, and CRUISE-NDP.

When the system is fully loaded (Fig. 15b), resource contention becomes more significant and CRUISE-NDP runs into more frequent pathologies. CRUISE-NDP improves weighted speedup by only 6% over the baseline, and the gap between CRUISE-NDP with either placement and AMS grows. Unlike CRUISE-NDP, AMS-Greedy still produces high-quality schedules, improving average performance by 11%. Finally, AMS-DP is 1% better than AMS-Greedy on average, improving performance by 12%. This result shows that a classification-based policy does not robustly handle contended cores in an asymmetric system. By contrast, AMS-Greedy still performs almost as well as the AMS-DP upper bound.

E. AMS works well with heterogeneous cores

So far, we have considered systems with homogeneous processor-die and NDP cores. However, it is often attractive to use simpler NDP cores than processor-die cores. This causes asymmetry in both cores and the memory hierarchy. As shown in Sec. V-B, AMS's model can be combined with PIE's model to handle this scenario.

Fig. 16 shows results for a 16-core system where processor-die cores are modeled after Haswell (NDP stacks use the same Silvermont-like cores as before). PIE alone neglects asymmetric hierarchies, so it cannot estimate the memory stalls of different cores accurately. Thus, PIE improves performance by only 4% on average. AMS-Greedy alone is oblivious to heterogeneous cores, so it improves performance by only 5% on average. AMS-Greedy with PIE handles both asymmetric cores and memory hierarchies, and thus improves performance the most, by 9% on average and by up to 21% over *Random*.

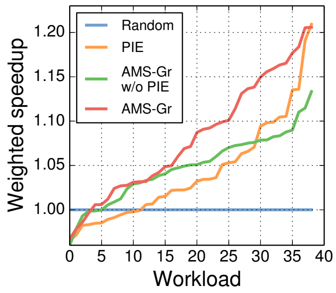


Fig. 16: Performance of AMS with PIE on heterogeneous cores.

TABLE II: Multithreaded workloads and inputs used.

Suite	Benchmark and input
SPECOMP2012	md, bwaves, ilbdc, fma3d, swim, mgrid, smithwa (ref/train), bt, botsspar (ref)
PARSEC	canneal, streamcluster, freqmine (native only)

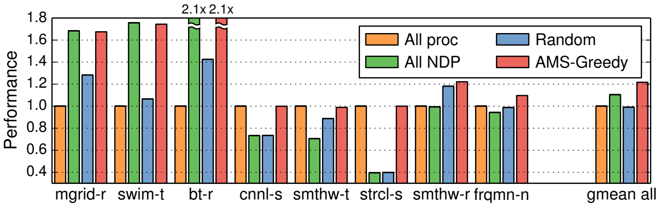


Fig. 17: Performance of representative multithreaded applications under different schedulers.

TABLE III: AMS under various system sizes and loads.

	16-core system			32-core system		
	Greedy	DP	CRUISE	Greedy	DP	CRUISE
50% load	1.18	1.18	1.11	1.19	1.19	1.10
100% load	1.11	1.12	1.06	1.13	1.14	1.06

TABLE IV: AMS under various LLC and memory capacities.

	LLC capacity			Per-stack capacity		
	8 MB	16MB	32MB	2GB	4GB	8GB
50% load	1.20	1.18	1.16	1.18	1.18	1.18
100% load	1.10	1.11	1.11	1.09	1.11	1.11

F. AMS works well with multithreaded apps

We also evaluate AMS on memory-intensive multithreaded workloads. Table II details these workloads. Each workload runs with 8 threads on the same system in Sec. VII-D.

Fig. 17 shows the performance ($\frac{1}{runtime}$) of four scheduling techniques over *All proc* on 8 representative workloads. The first three applications (`mgrid`, `swim`, and `bt`) have large working sets. Therefore, they benefit significantly when using NDP cores, by up to 2.1× over *All proc*. AMS-Greedy detects this preference and correctly schedules them to NDP cores, matching the performance of *All NDP*.

The next three applications (`canneal`, `smithwa-train`, and `streamcluster`) have smaller working sets that fit in the LLC. In this case, it is better to schedule these applications as *All proc* to utilize the LLC, and *All NDP* hurts performance by up to 60%. AMS-Greedy again schedules them correctly and avoids performance degradation.

The last two applications (`smithwa-ref` and `freqmine`) perform very similarly under *All proc* and *All NDP*. Nonetheless, AMS-Greedy improves them further by scheduling only some of the 8 threads on the processor die and the rest to NDP cores. This is because these two workloads have multi-MB thread-private working sets and only some of them fit in the LLC. Therefore, AMS spreads threads across processor die and NDP cores to better use available resources.

Overall, AMS improves gmean performance over *All proc* by 22%, while *All NDP* improves by only 10% and *Random* is 1% worse than *All proc*. These results show that AMS also handles multithreaded workloads under asymmetric hierarchies.

G. AMS sensitivity study

System parameters: Table III shows the performance improvement of AMS and CRUISE-NDP of 16-core and 32-core (16 cores in the processor die + 4×4 NDP cores) systems. AMS-Greedy and AMS-DP are similarly effective in this larger system, and the gap between AMS and CRUISE-NDP remains.

Table IV reports AMS's speedups under different loads when varying LLC and stacked DRAM sizes. LLC capacity has a higher impact when the system is undercommitted. This is because AMS can map applications to NDP cores with less contention. When the system is fully loaded, improvements are roughly the same over different capacities due to limited scheduling options.

We find that varying capacity in memory stacks does not change the effectiveness of the algorithm. This is due to two reasons. First, AMS successfully avoids capacity contention on memory stacks, and second, our workloads have medium size footprints (<8 GB) and thus, when preferences are properly spread, the capacity of a single stack is rarely exhausted.

Algorithm runtime: AMS-Greedy's runtime overhead scales well with the system size and number of threads to schedule. With 100% load in the 16-core and 32-core systems, AMS-Greedy's overheads are 1 Mcycle (0.1% of system cycles with a scheduling quantum of 50 ms) and 3 Mcycles (0.2%).

However, AMS-DP's overheads increase much more steeply with system size: each scheduling decision takes 10 Mcycles (1%) and 300 Mcycles (19%) for 16-core and 32-core systems, respectively. Thus, AMS-DP is only practical up to 16 cores.

Cache partitioning: Since AMS leverages cache partitioning when scheduling threads, we extend AMS with Vantage [57] to also partition the LLC. We find that cache partitioning improves performance by <1% because AMS already avoids cache pollution by scheduling threads to NDP cores. We thus conclude that AMS is effective without partitioning.

VIII. Conclusion

Advances in die-stacking technology have enabled systems with asymmetric memory hierarchies. We have shown that applications can benefit significantly from these asymmetric systems, but to realize their full potential, scheduling applications to their most suitable hierarchy is essential. We have presented AMS, a scheduling framework that achieves this goal by modeling application preferences to different hierarchies. AMS schedules applications to their best hierarchy and is efficient enough to use online, so it adapts to changing application behavior and outperforms prior scheduling techniques. AMS thus improves the performance of an asymmetric system by up to 37% and by 18% on average.

Acknowledgments

We sincerely thank Maleen Abeydeera, Joel Emer, Yee Ling Gan, Hyun Ryong Lee, Mark Jeffrey, Anurag Mukkara, Suvinay Subramanian, Victor Ying, Guowei Zhang, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CAREER-1452994 and by a grant from the Qatar Computing Research Institute.

References

[1] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture*, 2015.

[2] A. Agrawal, J. Torrellas, and S. Idgunji, "Xylem: Enhancing vertical thermal conduction in 3D processor-memory stacks," in *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture*, 2017.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture*, 2015.

[4] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture*, 2015.

[5] A. Alameldeen and D. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, 2006.

[6] ARM, "Cortex-A57 Processor," https://www.arm.com/products/processors/cortex-a/cortex-a57-processor.php, 2014.

[7] A. Barbalace, A. Iliopoulos, H. Rauchfuss, and G. Brasche, "It's time to think about an operating system for near data processing architectures," in *Proc. of the 16th Workshop on Hot Topics in Operating Systems*, 2017.

[8] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. of the 22nd Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2013.

[9] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture*, 2015.

[10] D. P. Bertsekas, *Dynamic programming and optimal control*. Athena scientific, 1995, vol. 1, no. 2.

[11] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[12] B. Black, "Keynote: Die stacking is happening!" in *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture*, 2013.

[13] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for NUMA-aware contention management on multicore systems," in *Proc. USENIX ATC*, 2011.

[14] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proc. of the 23th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2018.

[15] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, 2017.

[16] J. Cong and B. Yuan, "Energy-efficient scheduling on heterogeneous multi-core architectures," in *Proc. of the ACM/IEEE Intl. Symp. on Low Power Electronics and Design*, 2012.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. MIT press, 2009.

[18] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proc. of the 38th annual Intl. Symp. on Computer Architecture*, 2011.

[19] W. J. Dally, "Keynote: GPU computing: To exascale and beyond," in *Proc. of the ACM/IEEE conf. on Supercomputing*, 2010.

[20] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, "Architecture of a message-driven processor," in *Proc. of the 14th annual Intl. Symp. on Computer Architecture*, 1987.

[21] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on NUMA systems," in *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[22] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian data engine," in *Proc. of the 44th annual Intl. Symp. on Computer Architecture*, 2017.

[23] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proc. of the 24rd IEEE intl. symp. on High Performance Computer Architecture*, 2018.

[24] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture*, 2015.

[25] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. of the 24th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2015.

[26] M. Gao and C. Kozyrakis, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in *Proc. of the 22nd IEEE intl. symp. on High Performance Computer Architecture*, 2016.

[27] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," in *Proc. of the 22th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2017.

[28] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Proc. of the ACM/IEEE conf. on Supercomputing*, 1999.

[29] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation Intel Core processor," *IEEE Micro*, vol. 34, no. 2, 2014.

[30] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009.

[31] M. D. Hill, S. Adve, L. Ceze, M. J. Irwin, D. Kaeli, M. Martonosi, J. Torrellas, T. F. Wenisch, D. Wood, and K. Yelick, "21st century computer architecture," *arXiv preprint arXiv:1609.06756*, 2016.

[32] A. Hilton, N. Eswaran, and A. Roth, "FIESTA: A sample-balanced multi-program workload methodology," *Proc. MoBS*, 2009.

[33] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, "Accelerating linked-list traversal through near-data processing," in *Proc. of the 26th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2016.

[34] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture*, 2016.

[35] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *Proc. of the 34rd Intl. Conf. on Computer Design*, 2016.

[36] Hybrid Memory Cube Consortium, "Hybrid memory cube specification 2.1," http://hybridmemorycube.org/, 2017.

[37] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam, S. C. Steely, and J. Emer, "CRUISE: Cache replacement and utility-aware scheduling," in *Proc. of the 17th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[38] A. Jaleel, K. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. of the 37th annual Intl. Symp. on Computer Architecture*, 2010.

[39] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system," in *Proc. of the 17th Intl. Conf. on Computer Design*, 1999.

[40] D. Kanter, "Silvermont, Intel's low power architecture," in *RealWorldTech*, 2013.

[41] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[42] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture*, 2016.

[43] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proc. of the 22nd Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2013.

[44] P. M. Kogge, "EXECUBE: A new architecture for scaleable MPPs," in *Proc. of the Intl. Conf. on Parallel Processing*, 1994.

[45] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: IRAM," *IEEE Computer*, vol. 30, no. 9, 1997.

[46] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on NUMA systems: Asymmetry matters," in *Proc. of the USENIX Annual Technical Conf.*, 2015.

[47] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture*, 2009.

[48] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture*, 2015.

[49] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. of the 44th annual IEEE/ACM intl. symp. on Microarchitecture*, 2011.

[50] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero, "FlexDCP: A QoS framework for CMP architectures," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, 2009.

[51] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture*, 2007.

[52] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. of the 23rd IEEE intl. symp. on High Performance Computer Architecture*, 2017.

[53] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proc. of the 25th annual Intl. Symp. on Computer Architecture*, 1998.

[54] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for GPU architectures with processing-in-memory capabilities," in *Proc. of the 26th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2016.

[55] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "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*, 2014.

[56] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture*, 2006.

[57] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proc. of the 38th annual Intl. Symp. on Computer Architecture*, 2011.

[58] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. of the 40th annual Intl. Symp. on Computer Architecture*, 2013.

[59] J. E. Sasinowski and J. K. Strosnider, "A dynamic programming algorithm for cache memory partitioning for real-time systems," *IEEE Transactions on Computers*, vol. 42, no. 8, 1993.

[60] M. Scrbak, M. Islam, K. M. Kavi, M. Ignatowski, and N. Jayasena, "Processing-in-memory: Exploring the design space," in *Proc. of the Intl. Conf. on Architecture of Computing Systems (ARCS)*, 2015.

[61] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proc. of the 24th ACM Symp. on Parallelism in Algorithms and Architectures*, 2012.

[62] D. Skarlatos, R. Thomas, A. Agrawal, S. Qin, R. C. N. Pilawa-Podgurski, U. R. Karpuzcu, R. Teodorescu, N. S. Kim, and J. Torrellas, "Snatch: Opportunistically reassigning power allocation between processor and memory in 3D stacks," in *Proc. of the 49th annual IEEE/ACM intl. symp. on Microarchitecture*, 2016.

[63] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreading processor," in *Proc. of the 9th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[64] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proc. of the EuroSys Conf.*, 2007.

[65] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations," in *Proc. of the 14th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[66] J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system: A retrospective paper," in *Proc. of the 30th Intl. Conf. on Computer Design*, 2012.

[67] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-defined cache hierarchies," in *Proc. of the 44th annual Intl. Symp. on Computer Architecture*, 2017.

[68] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Nexus: A new approach to replication in distributed shared caches," in *Proc. of the 27th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2017.

[69] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. of the 39th annual Intl. Symp. on Computer Architecture*, 2012.

[70] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on CC-NUMA compute servers," in *Proc. of the 7th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 1996.

[71] E. Vermij, L. Fiorin, R. Jongerius, C. Hagleitner, J. V. Lunteren, and K. Bertels, "An architecture for integrated near-data processors," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 3, 2017.

[72] C. Xie, S. L. Song, and X. Fu, "Processing-in-memory enabled graphics processors for 3D rendering," in *Proc. of the 23rd IEEE intl. symp. on High Performance Computer Architecture*, 2017.

[73] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proc. of the 23rd intl. symp. on High-performance Parallel and Distributed Computing*, 2014.

[74] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. of the 15th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.