Jenga: Software-Defined Cache Hierarchies

Po-An Tsai*
MIT CSAIL
poantsai@csail.mit.edu

Nathan Beckmann*
CMU SCS
beckmann@cs.cmu.edu

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

ABSTRACT

Caches are traditionally organized as a rigid hierarchy, with multiple levels of progressively larger and slower memories. Hierarchy allows a simple, fixed design to benefit a wide range of applications, since working sets settle at the smallest (i.e., fastest and most energy-efficient) level they fit in. However, rigid hierarchies also add overheads, because each level adds latency and energy even when it does not fit the working set. These overheads are expensive on emerging systems with heterogeneous memories, where the differences in latency and energy across levels are small. Significant gains are possible by specializing the hierarchy to applications.

We propose Jenga, a reconfigurable cache hierarchy that dynamically and transparently specializes itself to applications. Jenga builds *virtual cache hierarchies* out of heterogeneous, distributed cache banks using simple hardware mechanisms and an OS runtime. In contrast to prior techniques that trade energy and bandwidth for performance (e.g., dynamic bypassing or prefetching), Jenga *eliminates* accesses to unwanted cache levels. Jenga thus improves both performance and energy efficiency. On a 36-core chip with a 1 GB DRAM cache, Jenga improves energy-delay product over a combination of state-of-the-art techniques by 23% on average and by up to 85%.

CCS CONCEPTS

• **Computer systems organization** → *Multicore architectures*;

KEYWORDS

Cache, Hierarchy, Heterogeneous memories, NUCA, Partitioning

ACM Reference format:
Po-An Tsai, Nathan Beckmann, Daniel Sanchez. 2017. Jenga: Software-Defined Cache Hierarchies. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080214

1 INTRODUCTION

With the coming end of Moore's Law, designers are increasingly turning towards architectural specialization to scale performance and energy efficiency. Memory accesses often limit the performance and efficiency of current systems, and the trend towards lean and specialized cores is placing mounting pressure on the energy and latency of memory accesses [15, 39]. Consequently, cache hierarchies are becoming more sophisticated in two key dimensions. First, caches are becoming increasingly distributed and non-uniform (NUCA [41]): each core enjoys cheap accesses to physically-close cache banks, but accesses to faraway banks are expensive. Second, cache hierarchies are starting to combine multiple technologies with disparate tradeoffs, such as SRAM and DRAM cache banks [24, 44, 62, 63].

*Po-An Tsai and Nathan Beckmann contributed equally to this work. This work was done while Nathan Beckmann was at MIT.

ISCA '17, June 24-28, 2017, Toronto, ON, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080214

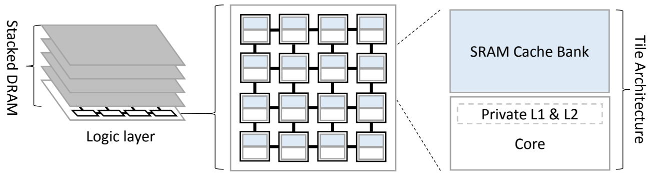


Figure 1: A modern multicore with distributed, on-chip SRAM banks and a 3D-stacked DRAM cache.

The problem: Ideally, these heterogeneous, distributed cache banks should be managed to approach the performance of *application-specific cache hierarchies* that hold working sets at minimum latency and energy. However, conventional systems are far from this ideal: they instead implement a *rigid* hierarchy of increasingly larger and slower caches, fixed at design time and managed by hardware. Rigid hierarchies are attractive because they are transparent to software, and they worked well in the past because systems had few cache levels with widely different sizes and latencies. However, the differences in size and latency are smaller in modern systems, so rigid hierarchies are accordingly less attractive.

For example, consider the tiled multicore in Fig. 1, with 16 MB of distributed on-chip SRAM cache banks and 1 GB of distributed stacked DRAM cache banks. Several problems arise when these banks are organized as a rigid two-level hierarchy, i.e., with on-chip SRAM as an L3 and stacked DRAM as an L4. The root problem is that many applications make poor use of one or more cache levels, and often *do not want hierarchy*. For example, an application that scans over a 32 MB array should ideally use a single cache level, sized to fit its 32 MB working set and placed as close as possible. The 16 MB SRAM L3 in Fig. 1 *hurts its performance and energy efficiency*, since it adds accesses without yielding many hits.

Prior work has explored a variety of techniques to hide the cost of unwanted hierarchy. These techniques try to hide the added latency by issuing speculative accesses up the hierarchy. For example, dynamic cache bypassing [34, 40] in Fig. 1 would check the SRAM L3 and stacked DRAM L4 in parallel, and *must* check both for correctness. While such techniques can improve performance by hiding latency, they do not eliminate the unnecessary accesses, which *still consume energy and bandwidth*. As systems are increasingly power-limited, techniques that trade energy for performance are correspondingly less attractive.

The opportunity: We begin by characterizing the benefits of application-specific hierarchies over rigid ones (Sec. 2). We find that the optimal hierarchy varies widely across applications, both in the number of levels and their sizes. Rigid hierarchies must cater to the conflicting needs of different applications, and even the best rigid hierarchy hurts applications that desire a markedly different configuration, hurting performance by up to 51% and energy-delay product (EDP) by up to 81%. Moreover, applications often have a strong preference for a hierarchical or flat design. Using the right number of levels yields significant improvements (up to 18% in EDP). These results hold even with techniques that mitigate the impact of unwanted hierarchy, such as prefetching and hit/miss prediction [53].

Architects can thus significantly improve performance and energy efficiency by specializing the cache hierarchy to individual applications. However, specialization traditionally sacrifices the convenience of general-purpose designs, while also hurting the performance of applications outside the target domain. Ideally, we would like the benefits of specialization without paying these costs.

Our solution: The key idea of this paper is that architects should treat all heterogeneous cache memories as a single pool, not as a rigid hierarchy, and build specialized hierarchies from this pool using the memory best suited to each application. To achieve this goal, we present Jenga, a reconfigurable cache architecture that builds single- or multi-level *virtual cache hierarchies* tailored to each active application (Sec. 3). Jenga does so while remaining transparent to applications, achieving the benefits of specialization without sacrificing the ease of use of general-purpose systems.

Jenga consists of hardware and software components. Jenga hardware requires small changes over prior work [7, 8, 42], since Jenga software does most of the heavy lifting to build virtual cache hierarchies. Specifically, Jenga builds on Jigsaw [7, 8], which constructs *single-level virtual caches* out of *homogeneous SRAM* banks (Sec. 4).

Contributions: We make two main contributions. First, we propose that new memory technologies should not be added as a rigid hierarchy. Instead, heterogeneous memories are better organized as a pool of resources, out of which appropriate hierarchies are then built. We are the first to study this opportunity in detail. In particular, Jigsaw does not consider hierarchy in any respect.

Second, we design new algorithms that build application-specific virtual cache hierarchies from heterogeneous and spatially distributed physical cache banks. Specifically, Jenga's OS runtime configures virtual hierarchies using:

- *Adaptive hierarchy allocation* (Sec. 6.2), which efficiently finds the right number of virtual cache levels and the size of each level given the demand on the memory system.
- *Bandwidth-aware data placement* (Sec. 6.3), which accounts for the limited bandwidth of emerging heterogeneous memories. Our approach avoids hotspots that hamper prior techniques, which consider only limited capacity.

These algorithms are Jenga's main technical contribution. Finding efficient and effective algorithms required tackling several challenges, e.g., how to reduce the dimensionality of the optimization problem in hierarchy allocation, and how to consider alternatives without backtracking or multiple passes in data placement.

Results: We evaluate Jenga on a 36-core chip with 18 MB of SRAM and 1.1 GB of 3D-stacked DRAM. Compared to a combination of state-of-the-art NUCA and DRAM cache techniques, Jigsaw and Alloy [53], Jenga improves full-system EDP by 23% on average and by up to 85%; and improves performance by 9% on average and by up to 35%. These benefits come at modest complexity over Jigsaw. Jenga's benefits over simpler techniques are even higher.

By contrast, adding a DRAM cache to a rigid hierarchy improves performance, but often degrades energy efficiency. We conclude that the rigid, multi-level hierarchies in current systems are ill-suited to many applications. Future memory systems should instead be reconfigurable and expose their heterogeneity to software.

2 MOTIVATION

Jenga's reconfigurable cache hierarchy offers two main benefits. First, Jenga frees the hierarchy from having to cater to the conflicting needs of different applications. Second, Jenga uses hierarchy only when beneficial, and adopts a appropriately-sized flat organization otherwise. But do programs really desire widely different hierarchies, and do they suffer by using a rigid one? And how frequently do programs prefer a flat design to a hierarchy? To answer these questions and quantify Jenga's potential, we first study the best application-specific hierarchies on a range of benchmarks, and compare them to the best overall rigid hierarchy.

Methodology: We consider a simple single-core system running SPEC CPU2006 apps. (We evaluate multi-program and multi-threaded workloads later.) The core has fixed 32 KB L1s and a 128 KB L2. To find the best hierarchy for each app, we consider NUCA SRAM and stacked-DRAM caches of different sizes: from 512 KB to 32 MB for SRAM, and from 128 MB to 2 GB for stacked DRAM. We evaluate all possible single- and two-level combinations of these memories (i.e., using an SRAM or DRAM L3, optionally followed by an SRAM or DRAM L4). Latency, energy, and area are derived using CACTI [52] and CACTI-3DD [10]. Each SRAM cache bank is 512 KB, and each stacked DRAM vault is 128 MB, takes the area of 4 tiles, and uses the latency-optimized Alloy design [53]. Larger caches have more banks, placed as close to the core as possible and connected through a mesh NoC. Sec. 7 details our methodology further.

Larger caches are more expensive [26, 27]: Area and static power increase roughly linearly with size, while access latency and energy scale roughly with its square root [65]. SRAM caches from 512 KB to 32 MB have access latencies from 9 to 45 cycles and access energies from 0.2 to 1.7 nJ, and stacked DRAM caches from 128 MB to 2 GB have access latencies from 42 to 74 cycles and energies from 4.4 nJ to 6 nJ. Monolithic caches of the same size yield similar figures.

We make the following key observations:

1. The optimal application-specific hierarchy varies widely in size and number of levels. We sweep all single- and two-level cache hierarchies, and rank them by system-wide EDP, which includes core, cache, and main memory static and dynamic power. Fig. 2 reports the best hierarchy for each application. Applications want markedly different hierarchies: eight out of the 18 memory-intensive applications we consider prefer a single-level organization, and their preferred sizes vary widely, especially at the L3.

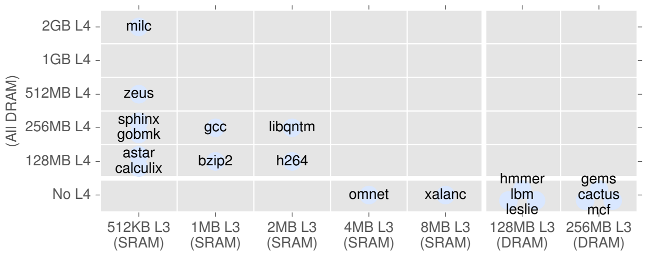


Figure 2: The best cache hierarchy (by EDP) for SPEC CPU2006.

2. Rigid hierarchies sacrifice performance and energy efficiency. The rigid hierarchy that maximizes gmean EDP on these applications consists of a 512 KB SRAM L3 and a 256 MB DRAM L4. This is logical, since six (out of 18) applications want a 512 KB SRAM L3 and seven want a 256 MB DRAM L3 or L4. But Fig. 3 shows that applications that desire a different hierarchy can do significantly better: up to 81% better EDP and 51% higher performance. Comparing Fig. 2 and Fig. 3 shows that these gains are highly correlated to how different the application-specific hierarchy is from the rigid one.

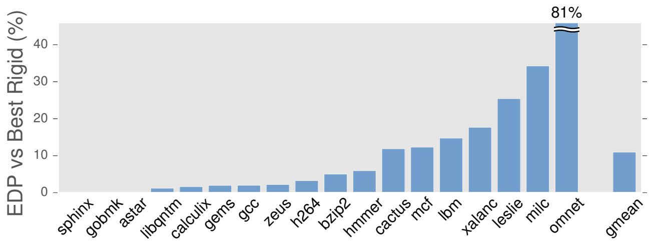


Figure 3: EDP improvements of application-specific hierarchies vs. the best rigid hierarchy.

With application-specific hierarchies, performance and EDP are highly correlated. This occurs because better hierarchies save energy by reducing expensive off-chip misses, and improving performance also reduces the contribution of static power to total energy. Data movement reductions are thus reflected in lower latency and energy; Jenga exploits this by optimizing for access latency. However, unlike prior work, Jenga reduces latency by *eliminating* wasteful accesses, not by hiding their latency.

3. Applications have strong preferences about hierarchy. Fig. 2 showed that many applications prefer a single- or a two-level hierarchy. But is this a strong preference? In other words, what would we lose by fixing the number of levels? To answer this question, Fig. 4 reports, for each application, the EDP of its best application-specific, two-level hierarchy relative to the EDP of its best single-level, L3-only hierarchy.

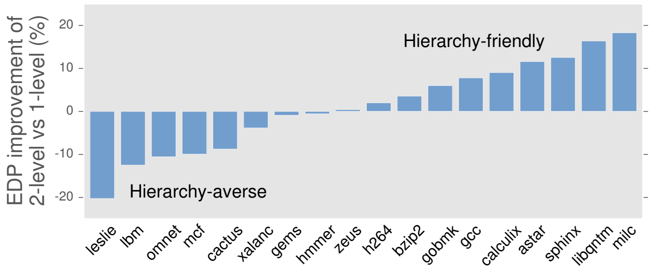


Figure 4: EDP of the best two-level, application-specific hierarchy vs. the best single-level one.

Fig. 4 shows that applications often have strong preferences about hierarchy: 7 out of the 18 applications are *hierarchy-averse*, and two-level organizations degrade their EDP by up to 20%. Others are *hierarchy-friendly* and see significant EDP gains, of up to 18%. This shows that fixing the hierarchy leaves significant performance on the table, and motivates adaptively choosing the right number of levels.

Putting it all together: Fig. 5 compares the gmean EDP and performance gains of the best rigid single-level hierarchy (a 128 MB DRAM L3), the best rigid two-level hierarchy (a 512 KB SRAM L3 plus a 256 MB DRAM L4), the best application-specific single-level cache size (i.e., L3 size with no L4), and the best application-specific hierarchy (L3 size and L4 size, if present, from Fig. 2).

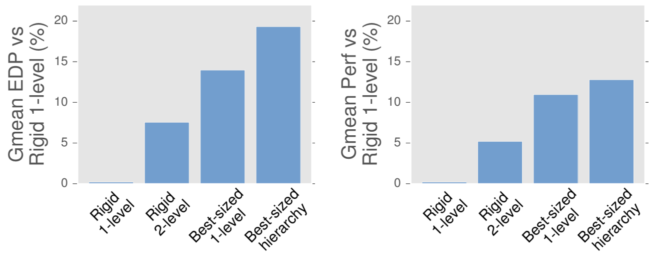


Figure 5: Gmean EDP/perf. of the best rigid hierarchies of 1 or 2 levels vs. application-specific hierarchies of 1 or (up to) 2 levels.

Overall, hierarchy offers only modest benefits in rigid designs since it is hampered by hierarchy-averse applications: just 7% higher gmean EDP and 5% performance. In contrast, application-specific hierarchies substantially improve performance and efficiency. Even a single-level cache of the appropriate size solidly outperforms a rigid hierarchy, by 14% gmean EDP and 11% performance. Building multi-level hierarchies (when appropriate) yields further improvements, by 19% gmean EDP and 13% gmean performance. This motivates the need for specialized virtual cache *hierarchies*.

3 JENGA OVERVIEW

Fig. 6 shows a 36-tile Jenga system running four applications. Each tile has a core, a private cache hierarchy (L1s and L2) and a 512 KB SRAM cache bank. There are four 256 MB DRAM cache banks, e.g., stacked DRAM vaults connected by an interposer. Jenga builds a custom *virtual cache hierarchy* out of the shared cache banks—i.e., the 512 KB SRAM and 256 MB DRAM banks, excluding private caches—for each application according to how it accesses memory. Letters show where each application is running (one per quadrant), colors show where its data is placed, and hatching indicates the second virtual hierarchy level, when present.

Jenga builds a single-level virtual cache for two apps. `omnet` (lower-left) uniformly accesses a small working set, so it is allocated a single-level virtual cache in nearby SRAM banks. This keeps its working set at minimum latency and energy. Misses from `omnet` go directly to main memory, *and do not access DRAM cache banks*.

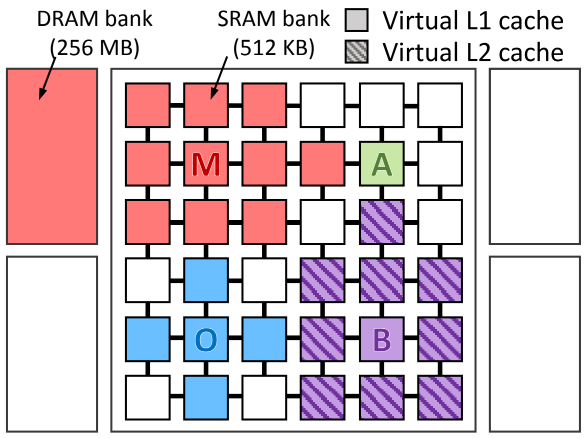


Figure 6: A 36-tile Jenga system running four applications. Jenga gives each a custom *virtual cache hierarchy*.

Similarly, mcf (upper-left) uniformly accesses its working set, so it is also allocated a single-level virtual cache—except its working set is much larger, so its data is placed in both SRAM banks and the nearest DRAM bank. Crucially, although mcf's virtual cache uses both SRAM and DRAM, it is still accessed as a single cache level (with accesses spread uniformly across its capacity), and misses go directly to main memory.

Jenga builds two-level virtual hierarchies for the other apps. astar (upper-right) accesses a small working set intensely and a larger working set less so, so Jenga allocates its local SRAM bank as the first level of its hierarchy (VL1), and its closest DRAM bank as the second level (VL2). astar thus prefers a hierarchy similar to the best rigid hierarchy in Sec. 2, although this is uncommon (Fig. 2). Finally, bzip2 has similar behavior, but with a much smaller working set. Jenga also allocates it a two-level hierarchy—except placed entirely in SRAM banks, saving energy and latency over the rigid hierarchy that uses DRAM banks.

Later sections explain Jenga's hardware mechanisms that control data placement and its OS runtime that chooses where data should be placed. We first review relevant prior work in multicore caching and heterogeneous memory technologies.

4 BACKGROUND AND RELATED WORK

Non-uniform cache access (NUCA) architectures: NUCA [41] techniques reduce the latency and energy of large caches. Static NUCA (S-NUCA) [41] spreads data across all banks with a fixed line-bank mapping, and exposes a variable bank access latency. S-NUCA is simple, but suffers from large average network distance. Dynamic NUCA (D-NUCA) schemes improve on S-NUCA by adaptively placing data close to the requesting core [3, 5, 6, 9, 11, 12, 25, 32, 49, 54, 66] using a mix of placement, migration, and replication techniques.

D-NUCAs and hierarchy: D-NUCAs often resemble a hierarchical organization, using multiple lookups to find data, and suffer from similar problems as rigid hierarchies. Early D-NUCAs organized banks as a fine-grain hierarchy [6, 41], with each level consisting of banks at a given distance. However, these schemes caused excessive data movement and thrashing [6]. Later techniques adopted coarser-grain hierarchies, e.g., using the core's local bank as a private level and all banks as a globally shared level [19, 49, 66], or spilling lines to other banks and relying on a global directory to access them [54]. Finally, Cho and Jin [12], Awasthi et al. [3], R-NUCA [25] and Jigsaw [7] do away with hierarchy entirely, adopting a *single-lookup* design: at a given time, each line is mapped to a fixed cache bank, and misses access main memory directly.

In systems with non-uniform SRAM banks, single-lookup NUCAs generally outperform directory-based NUCAs [7, 8, 25, 51]. This effect is analogous to results in Sec. 2: directory-based D-NUCAs suffer many of the same problems as rigid hierarchies, and single-lookup D-NUCAs eliminate hierarchy.

The key challenge in single-lookup designs is balancing off-chip and on-chip data movement, i.e., giving just enough capacity to fit the working set at minimum latency and energy. In particular, Jigsaw [7, 8] achieves this by letting software define single-level *virtual caches*. Jenga builds on Jigsaw, so we explain it in greater depth in later sections. However, systems with heterogeneous memories (e.g., DRAM cache banks) introduce a wider tradeoff in latency and capacity. Hierarchy is thus sometimes desirable, so long as it is used only when beneficial.

Stacked-DRAM caches: Recent advances in 3D stacking [44] and silicon interposers [36] have made stacked-DRAM caches practical. Prior work has proposed using stacked DRAM as either OS-managed memory [1, 17, 35, 64] or an extra layer of cache [20, 34, 45, 53]. When used as OS-managed memory, it must be allocated at page boundaries and hence suffers internal fragmentation. When used as a cache, the main challenge is its high access latency.

To address its latency, much recent work has focused on the structure of cache arrays. Several schemes [20, 34, 45, 47] place tags in SRAM, reducing latency at the cost of SRAM capacity. Alloy [53] uses a direct-mapped organization with tags adjacent to data, reducing latency at the cost of additional conflict misses. Jenga abstracts away details of array organization and is orthogonal to these techniques. While our evaluation uses Alloy caches, Jenga should also apply to other DRAM cache architectures and memory technologies.

Mitigating the cost of hierarchy: Some prior work hides the latency cost of hierarchy, but *does not eliminate the unnecessary accesses* to unwanted cache levels that are the root cause.

Dynamic cache bypassing [34, 40] will not install lines at specific levels when they are predicted to not be reused. Similarly, some cache replacement policies [18, 33, 55] bypass lines or insert them at low priority. However, these schemes still *must* check each level for correctness, so they do not eliminate unnecessary accesses.

Other techniques hide the latency of unwanted levels by overlapping it with speculative accesses up the hierarchy. Prefetchers and hit/miss prediction [53] fall in this category. Since mispredictions consume energy and bandwidth, these techniques essentially trade energy for lower latency. Like bypassing, they also must check all levels for correctness. So not only do they not eliminate unnecessary accesses—they *add* unnecessary accesses on mispredictions.

Jenga's advantage: Jenga complements these techniques while improving upon them (e.g., we use hit/miss prediction in our evaluation and study the effect of prefetchers). However, unlike prior work, Jenga *eliminates* wasteful accesses. For example, omnet in Sec. 3 avoids DRAM cache accesses entirely. By eliminating wasteful accesses, Jenga sidesteps this latency-energy tradeoff and improves both performance *and* energy efficiency. Compared to previously proposed reconfigurable caches [2, 4], Jenga handles heterogeneous and spatially distributed cache banks, and works with multiple applications. These factors greatly complicate reconfigurations.

5 JENGA HARDWARE

Jenga's key idea is to treat cache banks as a single resource pool from which software can build application-specific cache hierarchies. To realize this goal, Jenga's hardware needs to be flexible and reconfigurable at low cost. We thus base Jenga's hardware on Jigsaw [7, 8], which supports application-specific single-level SRAM caches. Jenga extends Jigsaw in straightforward ways to support heterogeneous (e.g., SRAM and DRAM) cache banks and multi-level virtual hierarchies. We now present these hardware components, emphasizing differences from Jigsaw at the end of the section. See prior work [7, 8] for details of Jigsaw's hardware.

Fig. 7 shows the tiled multicore we use to present Jenga. Each tile has a core, a directory bank, and an SRAM cache bank. The system also has distributed DRAM cache banks (connected through TSVs or an interposer). We first present steady-state operation in Jenga, then discuss support for coherence and reconfigurations.

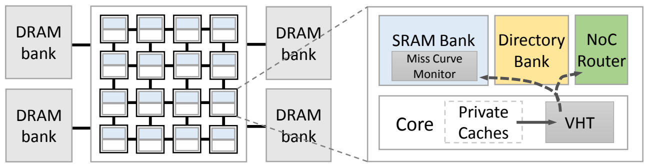


Figure 7: 16-tile Jenga system with distributed on-chip SRAM banks and four DRAM banks (e.g., stacked DRAM vaults).

Virtual cache hierarchies: Jenga builds *virtual cache hierarchies* (VHs) by combining parts of physical SRAM banks and stacked DRAM banks, as shown in Fig. 6. Our implementation partitions SRAM banks but not DRAM banks. This is because *(i)* SRAM capacity is scarce and highly contended but DRAM capacity is not, so partitioning is less beneficial in DRAM banks, and *(ii)* high associativity is expensive in DRAM caches [53], making partitioning more costly.

Data are mapped to a specific VH using the virtual memory system. Jenga maps each page to a particular VH, and adds a VH id to each page table entry and TLB entry to keep track of this mapping. In our Jenga implementation, there are three kinds of VHs: each thread has a thread-private VH, all threads in the same process share a process VH, and all threads in the system share a global VH. This simple scheme captures common access patterns (e.g., letting Jenga place private data near its thread) with few VHs per thread. Each VH can have one or two levels in the shared cache banks, which we call VL1 and VL2 (these are different from cores' private L1s and L2).

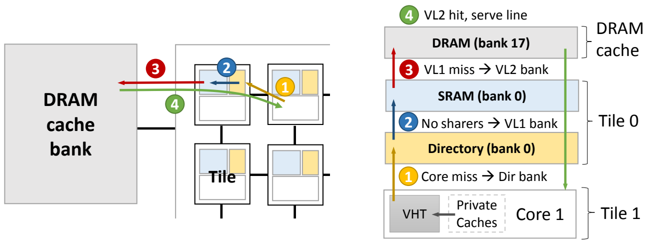


Figure 8: Example access to a virtual hierarchy (VH) that misses in the VL1 and hits in the VL2. An access first checks the directory bank, then its VL1 and (upon a miss) VL2 locations.

Accessing VHs: Fig. 8 shows an example memory access that misses in the VL1 and hits in the VL2. The access begins with a private cache miss in tile 1. First, tile 1's core consults its *virtual hierarchy table* (VHT) to find the path the access should follow (see below). Each address belongs to a single VH and has unique VL1 and VL2 locations—in Jenga, data does *not* migrate upon an access. In this example, the address maps to tile 0 for directory and VL1, so tile 1 next forwards the request to tile 0. At tile 0, we first check the directory bank to maintain coherence and, in parallel, access the VL1 by querying the SRAM bank. The request misses in both directory and SRAM banks, indicating that no copy of the data is present in any private cache, so tile 0 forwards the request to the VL2 bank, DRAM bank 17, located nearby. The request hits, and the data is returned to tile 1 via the SRAM and directory banks at tile 0.

The VHT stores the configuration of the virtual hierarchies that the running thread can access. Fig. 9 shows its organization. The VHT implements a configurable hash function that maps an address to its unique location, similar to Jigsaw's VTB [8]. In our implementation, it is a fully associative, three-entry lookup table indexed by VH id

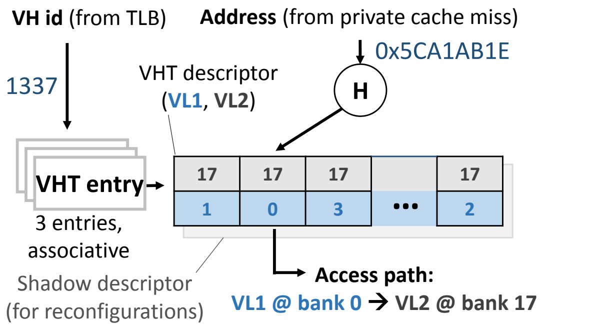

VH id (from TLB)
Address (from private cache miss)
0x5CA1AB1E
1337
VHT descriptor (VL1, VL2)
H
VHT entry
3 entries, associative
17 17 17 17
1 0 3 ••• 2
Shadow descriptor (for reconfigurations)
Access path:
VL1 @ bank 0 → VL2 @ bank 17

Figure 9: The virtual hierarchy table (VHT) maps addresses to their unique VL1 and (if present) VL2 locations, storing the bank ids and partition ids (not shown).

(letting each thread access its private, process, and global VHs). Each entry contains the VH's *descriptor*, itself two arrays of N bank ids for the VL1 and VL2 locations ($N = 128$ buckets in our evaluation). The address is hashed to a bucket to find its access path (as in Fig. 8). The VHT is accessed in parallel with the private L2 to hide its latency.

Maintaining coherence: In Jenga all accesses to a single address follow the same path beyond the private caches, automatically maintaining coherence in the shared levels. Therefore, directories are needed only to maintain coherence in the private caches. Unlike Jigsaw, which uses in-cache directories, Jenga uses separate directory banks to track the contents of private caches. Separate directories are much more efficient when the system's shared cache capacity greatly exceeds its private capacity.

To reduce directory latency, Jenga uses a directory bank near the VL1 bank, similar to prior dynamic directory schemes [16, 48]. Specifically, when the VL1 access is to an SRAM bank, Jenga uses the directory bank in the same tile, as in Fig. 8. When the VL1 access is to a DRAM bank, Jenga uses a directory bank in nearby tiles. For example, VL1 accesses to the top-left DRAM bank in Fig. 7 would use the four directory banks in the top-left quadrant. Directory bank ids are always derived from the VL1 bank id.

Dynamic directory placement is critical to Jenga's scalability and performance. If directories used a naïve static placement, then directory latency would increase with system size and eliminate much of Jenga's benefit. Instead, this dynamic directory mapping means that access latency is determined only by working set size and *does not increase with system size*.

Directory bank lookups are only required when an access to a shared VH misses in the private caches: VL1 and VL2 cache lines are not under a coherence directory, so no directory lookups are needed on VL1 or VL2 misses. And accesses to a thread-private VH do not need coherence [14] and skip the directory.

Monitoring VHs: Like Jigsaw, Jenga uses distributed utility monitors [56] to gather the miss curve of each VH. Miss curves let Jenga's software runtime find the right virtual hierarchies without trial and error. A small fraction (~1%) of VHT accesses are sampled into these monitors. Specifically, we use geometric monitors (GMONs) [8], which let us accurately monitor large caches at low overhead.

Supporting fast reconfigurations: Periodically (every 100 ms), Jenga software updates the configuration of some or all VHs by updating the VHT descriptors at all cores. Jenga uses the same techniques as Jigsaw to make reconfigurations incremental and efficient [8, Sec. IV.H]. Following a reconfiguration, the system enters a transient period where VHTs retain the old VH descriptors (in the shadow descriptors in Fig. 9), and VL1/VL2 accesses check both the new and old locations of the line. If the line is found at the old location during a reconfiguration, it is migrated to its new location.

Concurrently, directory and cache banks walk their tag arrays, invalidating lines that have moved to a new bank, and forward these invalidations to directories, which in turn invalidate private caches as needed. Once these tag array walks finish (which takes just a few milliseconds), the system resumes steady-state operation, with cores checking a single location at each level.

Supporting page reclassifications: Like Jigsaw and R-NUCA, Jenga uses a simple technique to map pages to VHs [7, 14, 25]. All pages start in the private VH of the thread that allocates them and are upgraded lazily: the first access from another thread in the same process upgrades the page to the process VH, and the first access from another process's thread upgrades the page to the global VH. To maintain coherence on an upgrade, the OS issues a bulk invalidation of the page (which invalidates its lines in both shared and private levels) and performs a TLB shootdown to invalidate the page's translations. These reclassifications are expensive but very rare in steady-state operation (<1% runtime overhead).

Jenga extensions over Jigsaw hardware: In summary, Jenga makes simple changes to Jigsaw's hardware. The main differences are:

- Jenga supports two-level virtual cache hierarchies.
- Jenga does not partition DRAM cache banks.
- Jenga uses non-inclusive caches and separate, dynamically-mapped directory banks, which are more efficient than in-cache directories given large DRAM cache banks.

Hardware overheads: In our implementation, each VH descriptor has $N = 128$ buckets and takes 384 bytes, 192 per virtual level (128 2×6-bit buckets, for bank and partition ids). A VHT has three entries, each with two descriptors (normal and shadow, as explained above). The VHT thus requires $\sim$2.4 KB. We use 8 KB GMONs, and have two monitors per tile. In total, Jenga adds $\sim$20 KB per tile, less than 720 KB for a 36-tile chip, and just 4% overhead over the SRAM cache banks.

6 JENGA SOFTWARE

Jenga hardware provides a flexible reconfigurable substrate, and leaves the challenging job of specializing hierarchies to software. Periodically, Jenga's OS-level software runtime reconfigures virtual hierarchies to minimize data movement, considering limited resources and applications' behavior. Jenga software runs concurrently with other useful work in the system. Each reconfiguration consists of four steps, shown in Fig. 10:

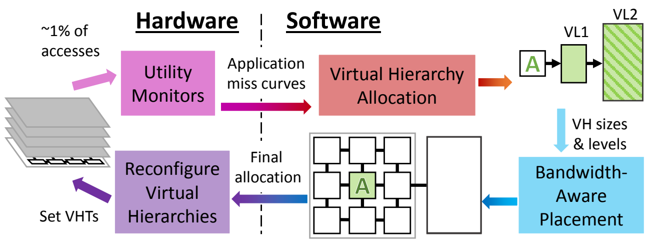


Figure 10: Overview of Jenga reconfigurations. Hardware profiles applications; software periodically reconfigures virtual hierarchies to minimize total access latency.

(1) Read miss curves from hardware monitors.
(2) Divide cache capacity into one- or two-level virtual cache hierarchies (Sec. 6.2). This algorithm finds the number of levels for each VH and the size of each level, but does not place them.
(3) Place each virtual cache hierarchy in cache banks, accounting for the limited bandwidth of DRAM banks (Sec. 6.3).
(4) Initiate a reconfiguration by updating the VHTs.

Jenga makes major extensions to Jigsaw's runtime to support hierarchies and cope with limited DRAM bank bandwidth. We first explain how Jenga integrates heterogeneity into Jigsaw's latency model and then explain Jenga's new algorithms.

6.1 Jenga's latency model

Jenga allocates capacity among VHs to minimize end-to-end access latency. Jenga models latency through two components (Fig. 11): time spent on cache misses, which *decreases* with cache size; and time spent accessing the cache, which *increases* with cache size (larger virtual caches must use further-away banks). Summing these yields the *total access latency curve* of a virtual cache. Both Jenga and Jigsaw use these curves to allocate capacity among applications, trying to minimize total system latency. Since the same trends hold for energy, Jenga also reduces energy and improves EDP.

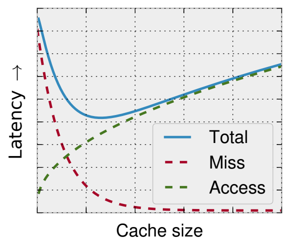


Figure 11: Access latency.

We construct these curves as follows: The miss latency curve is computed from the hardware miss curve monitors, since the miss latency at a given cache size is just the expected number of misses (read from monitors) times the memory latency. Jenga constructs the cache access latency curve for individual levels using the system configuration. Fig. 12 shows how. Starting from each tile (e.g., top-left in Fig. 12), Jenga sorts banks in order of access latency, including both network and bank latency. This yields the *marginal latency curve*; i.e., how far away the next closest capacity is at every possible size. The marginal latency curve is useful since its average value from 0 to s gives the average latency to a cache of size s.

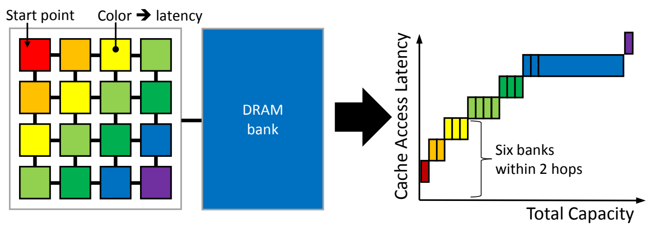
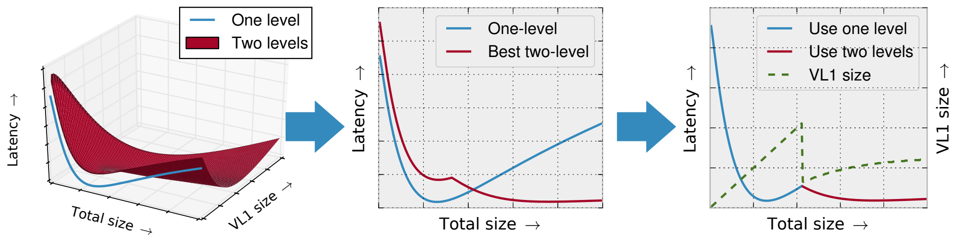


Figure 12: Jenga models access latency by sorting capacity according to latency, producing the *marginal latency curve* that yields the latency to the next available bank. Averaging this curve gives the average access latency.

Jigsaw uses the total access latency curves to allocate SRAM capacity among virtual caches. Jigsaw then places virtual caches across SRAM banks in two passes. First, virtual caches take turns greedily grabbing capacity in their most favorable banks. Second,



(a) Full hierarchy model (b) Choosing the best VL1 (c) Choosing the best hierarchy

Figure 13: Jenga models the latency of each virtual hierarchy with one or two levels. *(a)* Two-level hierarchies form a surface, one-level hierarchies a curve. *(b)* Jenga then *projects* the minimum latency across VL1 sizes, yielding two curves. *(c)* Finally, Jenga uses these curves to select the best hierarchy (i.e., VL1 size) for every size.

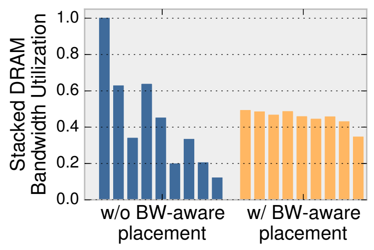


Figure 14: Distribution of bandwidth across DRAM vaults on `lbm`. Jenga removes hotspots by modeling queuing latency at each vault.

virtual caches trade capacity to move more-intensely accessed data closer to where it is used, reducing access latency [8].

Jenga makes a few modifications to this framework to support heterogeneity. First, Jenga models banks with different access latencies and capacities. Second, Jenga models the latency over TSVs or an interposer to access DRAM banks. These changes are already illustrated in Fig. 12. They essentially let the latency model treat DRAM banks as different "flavor" of cache bank. These modifications can be integrated with Jigsaw's runtime to produce virtual caches using heterogeneous memories, but without hierarchy. Sec. 7.4 evaluates these simple changes, and, as shown in Sec. 2, an appropriately sized, single-level cache performs well on many apps. However, since apps are often hierarchy-friendly and since DRAM banks also have limited bandwidth, there is room for significant improvement.

6.2 Virtual hierarchy allocation

Jenga decides whether to build a single- or two-level hierarchy by modeling the latency of each and choosing the lowest. For two-level hierarchies, Jenga must decide the size of both the first (VL1) and second (VL2) levels. The tradeoffs in the two-level model are complex [65]: A larger VL1 reduces misses, but increases the latency of *both* the VL1 and VL2 since it pushes the VL2 to further-away banks. The best VL1 size depends on the VL1 miss penalty (i.e., the VL2 access latency), which depends on the VL2 size. And the best VL2 size depends on the VL1 size, since VL1 size determines the access pattern seen by the VL2. The best hierarchy is the one that gets the right balance. This is not trivial to find.

Jenga models the latency of a two-level hierarchy using the standard formulation:

$$\begin{aligned}\text{Latency} = {} & \text{Accesses} \times \text{VL1 access latency} \\ & + \text{VL1 Misses} \times \text{VL2 access latency} \\ & + \text{VL2 Misses} \times \text{Memory latency}\end{aligned}$$

We model VL2 misses as the miss curve at the VL2 size. This is a conservative, *inclusive* hierarchy model. In fact, Jenga uses non-inclusive caches, but non-inclusion is hard to model.[1]

The VL2 access latency is modeled similarly to the access latency of a single-level virtual cache (Fig. 12). The difference is that, rather than averaging the marginal latency starting from zero, we average the curve starting from the VL1 size (VL2s are placed after VL1s).

[1]Alternatively, Jenga could use exclusive caches, in which the VL2 misses would be reduced to the miss curve at the combined VL1 and VL2 size. However, exclusion adds traffic between levels [60], a poor tradeoff with DRAM banks.

Fig. 13 shows how Jenga builds hierarchies. Jenga starts by evaluating the latency of two-level hierarchies, building the *latency surface* that describes the latency for every VL1 size and total size (Fig. 13(a)). Next, Jenga projects the best (i.e., lowest latency) two-level hierarchy along the VL1 size axis, producing a curve that gives the latency of the best two-level hierarchy for a given total cache size (Fig. 13(b)). Finally, Jenga compares the latency of single- and two-level hierarchies to determine at which sizes this application is hierarchy-friendly or -averse (Fig. 13(c)). This choice in turn implies the hierarchy configuration (i.e., VL1 size for each total size), shown on the second *y*-axis in Fig. 13(c).

With these changes, Jenga models the latency of a two-level hierarchy in a single curve, and thus can use the same partitioning algorithms as in prior work [7, 56] to allocate capacity among virtual hierarchies. The allocated sizes imply the desired configuration (the VL1 size in Fig. 13(c)), which Jenga places as described in Sec. 6.3.

Efficient implementation: Evaluating every point on the surface in Fig. 13(a) is too expensive. Instead, Jenga evaluates a few well-chosen points. Our insight is that there is little reason to model small changes in large cache sizes. For example, the difference between a 100 MB and 101 MB cache is often inconsequential. Sparse, *geometrically spaced* points can achieve nearly identical results with much less computation.

Rather than evaluating every configuration, Jenga first computes a list of candidate sizes to evaluate. It then only evaluates configurations with total size or VL1 size from this list. The list is populated by geometrically increasing the spacing between points, while being sure to include points where the marginal latency changes (Fig. 12).

Ultimately, our implementation at 36 tiles allocates >1 GB of cache capacity by evaluating just ∼60 candidate sizes per VH. This yields a mesh of ∼1600 points in the two-level model. Our sparse model performs within 1% of an impractical, idealized model that evaluates the entire latency surface.

6.3 Bandwidth-aware data placement

The final improvement Jenga makes is to account for bandwidth usage. In particular, DRAM banks have limited bandwidth compared to SRAM banks. Since Jigsaw ignores differences between banks, it often spreads bandwidth unevenly across DRAM banks, producing hotspots that sharply degrade performance.

The simplest approach to account for limited bandwidth is to dynamically monitor bank access latency, and then use these monitored latencies in the marginal latency curve. However, monitoring does not solve the problem, it merely causes hotspots to shift between

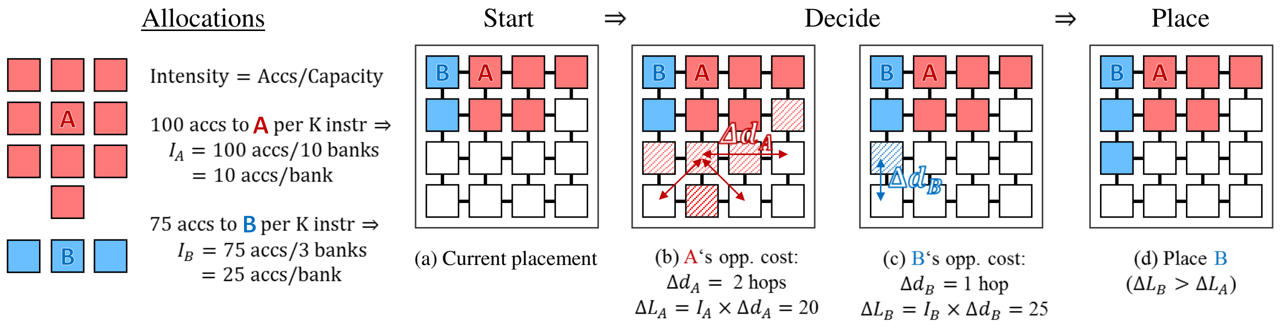

Allocations
Intensity = Accs/Capacity
100 accs to A per K instr ⇒
I_A = 100 accs/10 banks = 10 accs/bank
75 accs to B per K instr ⇒
I_B = 75 accs/3 banks = 25 accs/bank
Start ⇒ Decide ⇒ Place
A
B
Δd_A
Δd_B
(a) Current placement
(b) A's opp. cost: Δd_A = 2 hops, $\Delta L_A = I_A \times \Delta d_A = 20$
(c) B's opp. cost: Δd_B = 1 hop, $\Delta L_B = I_B \times \Delta d_B = 25$
(d) Place B ($\Delta L_B > \Delta L_A$)

Figure 15: Jenga reduces total access latency by considering two factors when placing a chunk of capacity: *(i)* how far away the capacity will have to move if not placed, and *(ii)* how many accesses are affected (called the *intensity*).

DRAM banks at each reconfiguration. Keeping a moving average can reduce this thrashing, but since reconfigurations are infrequent, averaging makes the system unresponsive to changes in load.

We conclude that a proactive approach is required. Jenga achieves this by placing data incrementally, accounting for queueing effects at DRAM banks on every step with a simple M/D/1 queue latency model. This technique eliminates hotspots on individual DRAM banks, reducing queuing delay and improving performance.

Incremental placement: Optimal data placement is an NP-hard problem. Virtual caches vary greatly in how sensitive they are to placement, depending on their access rate, the size of their allocation, which tiles access them, etc. Accounting for all possible interactions during placement is challenging. We observe, however, that the main tradeoffs are the size of the virtual cache, how frequently it is accessed, and the access latency at different cache sizes. We design a heuristic that accounts for these tradeoffs.

Jenga places data incrementally. At each step, one virtual cache gets to place some of its data in its most favorable bank. Jenga selects the virtual cache that has the highest *opportunity cost*, i.e., the one that suffers the largest latency penalty if it cannot place its data in its most favorable bank. This opportunity cost captures the cost (in latency) of the space being given to another virtual cache.

Fig. 15 illustrates a single step of this algorithm. The opportunity cost is approximated by observing that if a virtual cache does not get its favored allocation, then its entire allocation is shifted further down the marginal latency curve. This shift is equivalent to *moving a chunk of capacity from its closest available bank to the bank just past where its allocation would fit*. This heuristic accounts for the size of the allocation and distance to its nearest cache banks.

For example, the step starts with the allocation in Fig. 15(a). In Fig. 15(b) and Fig. 15(c), each virtual cache (A and B) sees where its allocation would fit. Note that it does not actually place this capacity, it just reads its marginal latency curve (Fig. 12). It then compares the distance from its closest available bank to the next available bank (Δd, arrows), which gives *how much additional latency* is incurred if it does not get to place its capacity in its favored bank.

However, this is only half of the information needed to approximate the opportunity cost. We also need to know *how many accesses pay this latency penalty*. This is given by the intensity I of accesses to the virtual cache, computed as its access rate divided by its size. All told, we approximate the opportunity cost as: $\Delta L \approx I \times \Delta d$.

Finally, in Fig. 15(d), Jenga chooses to place a chunk of B's allocation since B's opportunity cost is larger than A's. Fig. 15 places a full bank per step; our Jenga implementation places at most 1/16th of a bank per step.

Bandwidth-aware placement: To account for limited bandwidth, we update the latency to each bank at each step. This may change which banks are closest (in latency) from different tiles, changing where data is placed in subsequent iterations. Jenga thus spreads accesses across multiple DRAM banks, equalizing their access latency.

We update the latency using a simple M/D/1 queueing model. Jenga models SRAM banks having unlimited bandwidth, and DRAM banks having 50% of peak bandwidth (to account for cache overheads [13], bank conflicts, suboptimal scheduling, etc.). Though more sophisticated models could be used, this model is simple and avoids hotspots.

Jenga updates the bank's latency on each step after data is placed. Specifically, placing capacity s at intensity I consumes $s \times I$ bandwidth. The bank's load ρ is the total bandwidth divided by its service bandwidth μ. Under M/D/1, queuing latency is $\rho/(2\mu \times (1-\rho))$ [23]. After updating latencies, Jenga sorts banks for later steps. Resorting is cheap because each bank moves at most a few places.

Fig. 14 shows a representative example of how Jenga balances accesses across DRAM vaults on `lbm`. Each bar plots the access intensity to different DRAM vaults with (right) and without (left) bandwidth-aware placement. Jigsaw's placement algorithm leads to hotspots, overloading some vaults while others are idle, whereas Jenga evenly spreads accesses across vaults. Jenga improves performance by 10%, energy by 6%, and EDP by 17% in this case. Similar results hold for other apps (e.g., `omnet` and `xalanc`, Sec. 7.4).

6.4 Implementation discussion

OS integration: Jenga's software runs in the OS, and should be tightly integrated with OS thread scheduling to ensure data stays near the threads that use it [8]. Jenga does not complicate thread migration or context switches: threads access distinct VHs, and caches do *not* need to be flushed when a new thread is swapped in. Instead, only their VHT entries must be loaded (3 entries total).

We focus our evaluation on long-running batch workloads, but Jenga should work in other scenarios with minor changes. First, in oversubscribed systems, background threads and processes (e.g., kernel threads, short-lived threads, or short jobs) should use the global VH to limit monitoring and reconfiguration overheads. Second, for interactive workloads with real-time requirements, Jenga's reconfiguration algorithm can incorporate policies that partition cache capacity to maintain SLOs instead of maximizing throughput, like

Cores	36 cores, x86-64 ISA, 2.4 GHz, Silvermont-like OOO [37]: 8B-wide ifetch; 2-level bpred with 512×10-bit BHSRs + 1024×2-bit PHT, 2-way decode/issue/rename/commit, 32-entry IQ and ROB, 10-entry LQ, 16-entry SQ; 371 pJ/instruction, 163 mW/core static power [43]
L1 caches	32 KB, 8-way set-associative, split data and instruction caches, 3-cycle latency; 15/33 pJ per hit/miss [52]
Prefetchers (Fig. 23 only)	16-entry stream prefetchers modeled after and validated against Nehalem [59]
L2 caches	128 KB private per-core, 8-way set-associative, inclusive, 6-cycle latency; 46/93 pJ per hit/miss [52]
Coherence	MESI, 64 B lines, no silent drops; sequential consistency; 4K-entry, 16-way, 6-cycle latency directory banks for Jenga; in-cache L3 directories for others
Global NoC	6×6 mesh, 128-bit flits and links, X-Y routing, 2-cycle pipelined routers, 1-cycle links; 63/71 pJ per router/link flit traversal, 12/4 mW router/link static power [43]
SRAM banks	18 MB, one 512 KB bank per tile, 4-way 52-candidate zcache [57], 9-cycle bank latency, Vantage partitioning [58]; 240/500 pJ per hit/miss, 28 mW/bank static power [52]
Stacked DRAM banks	1152 MB, one 128 MB vault per 4 tiles, Alloy with MAP-I DDR3-3200 (1600 MHz), 128-bit bus, 16 ranks, 8 banks/rank, 2 KB row buffer; 4.4/6.2 nJ per hit/miss, 88 mW/vault static power [10]
Main memory	4 DDR3-1600 channels, 64-bit bus, 2 ranks/channel, 8 banks/rank, 8 KB row buffer; 20 nJ/access, 4 W static power [50]
DRAM timings	t_{CAS}=8, t_{RCD}=8, t_{RTP}=4, t_{RAS}=24, t_{RP}=8, t_{RRD}=4, t_{WTR}=4, t_{WR}=8, t_{FAW}=18 (all timings in t_{CK}; stacked DRAM has half the t_{CK} as main memory)

Table 1: Configuration of the simulated 36-core system.

Ubik [38]. However, these scenarios are beyond the scope of this paper, and we leave them to future work.

Software overheads: Jenga's configuration algorithm, including both VH allocation and bandwidth-aware placement, completes in 40 M aggregate core cycles, or 0.4% of system cycles at 36 tiles. It is trivial to parallelize across available cores, and it scales well with system size, taking 0.3% of system cycles at 16 tiles. These overheads are accounted for in our evaluation.

Jenga's runtime runs *concurrently* with other useful work on the system, and only needs to pause cores to update VHTs (i.e., a few K cycles every 100 ms, given the hardware support for incremental reconfigurations in Sec. 5).

7 EVALUATION

Modeled system: We perform microarchitectural, execution-driven simulation using zsim [59] and model a 36-core system with on-chip SRAM and stacked DRAM caches, as shown in Fig. 1. Each tile has a lean 2-way OOO core similar to Silvermont [37] with private L1 instruction and data caches, a unified private L2, and a 512 KB SRAM bank (same MB/core as Knights Landing [62]). We use a mesh with 2-stage pipelined routers and 1-cycle links (like [25, 28] and more aggressive than [3, 7, 49]). Table 1 details the system's configuration.

We compare five different cache organizations: *(i)* Our baseline is an S-NUCA SRAM L3 without stacked DRAM. *(ii)* We add a stacked DRAM Alloy cache with MAP-I hit/miss prediction [53]. These organizations represent rigid hierarchies.

The next two schemes use Jigsaw to partially relax the rigid hierarchy. Specifically, we evaluate a Jigsaw L3 both *(iii)* without stacked DRAM and *(iv)* with an Alloy L4 (we call this combination JigAlloy). Hence, SRAM adopts an application-specific organization, but the stacked DRAM (when present) is still treated as a rigid hierarchy. (We also evaluated R-NUCA [25], but it performs uniformly worse than Jigsaw.) Finally, we evaluate *(v)* Jenga.

All organizations that use Alloy employ MAP-I memory access predictor [53]. When MAP-I predicts a cache miss, main memory is accessed in parallel with the stacked DRAM cache. Jenga uses MAP-I to predict misses to VL2s in SRAM as well as DRAM. We account for all reconfiguration overheads (in both software and hardware).

Workloads: Our workload setup mirrors prior work [8]. First, we simulate copies (SPECrate) and mixes of SPEC CPU2006 apps. We use the 18 SPEC CPU2006 apps with $\geq$5 L2 MPKI (as in Sec. 2) and fast-forward all apps in each mix for 20 B instructions. We use a fixed-work methodology and equalize sample lengths to avoid sample imbalance, similar to FIESTA [29]. We first find how many instructions each app executes in 1 B cycles when running alone, I_i. Each experiment then runs the full mix until all apps execute at least I_i instructions, and we consider only the first I_i instructions of each app when reporting system-wide performance and energy.

Second, we simulate the multi-threaded SPEC OMP2012 apps that are cache-sensitive (those with at least 5% performance difference across schemes). We instrument each app with heartbeats that report global progress (e.g., when each timestep or transaction finishes) and run each app for as many heartbeats as the baseline system completes in 1 B cycles after the start of the parallel region.

We use weighted speedup [61] as our performance metric, and energy-delay product (EDP) to summarize performance and energy gains. We use McPAT 1.3 [43] to derive the energy of cores, NoC, and memory controllers at 22 nm, CACTI [52] for SRAM banks at 22 nm, CACTI-3DD [10] for stacked DRAM at 45 nm, and Micron DDR3L datasheets [50] for main memory. This system takes 195 mm^2 with typical power consumption of 60 W in our workloads, consistent with area and power of scaled Silvermont-based systems [31, 37].

7.1 Multi-programmed copies

Fig. 16 shows compares the EDP of different cache organizations when running 36 copies of one SPEC CPU2006 app on the 36-tile system. We select apps that exhibit a range of hierarchy-friendly and hierarchy-averse behaviors to better understand how Jenga performs vs. rigid hierarchies, and present full results later. Fig. 17 shows traffic breakdowns that give more insight into these results. Each bar reports on-chip network traffic (in flits) relative to the baseline. Furthermore, each bar shows a breakdown of traffic by source-destination pairs, e.g., from private L2s to 3D-stacked DRAM.

The first two apps in Fig. 16 are hierarchy-friendly. `astar` has two working sets with different sizes (512 KB and 10 MB). It accesses the small working set very frequently and the large one less so. Therefore, `astar` prefers a two-level hierarchy. Alloy does not help much because SRAM banks absorb most accesses (little SRAM-to-Mem traffic), but Jigsaw helps `astar` significantly by placing the small working set in the local SRAM bank, eliminating L2-to-SRAM traffic and improving EDP by 2×. JigAlloy performs close to Jigsaw, since Alloy helps `astar` little. Jenga improves EDP even further (by 2.4×) because it places the 10 MB working set in a nearby DRAM bank, reducing VL2 access latency vs. Alloy.

`bzip2` has a smooth miss curve with many misses at 512 KB, few misses at 2 MB, and virtually no misses beyond 8 MB. Alloy's extra DRAM capacity helps substantially, replacing SRAM-to-Mem traffic with more efficient SRAM-to-3D-DRAM traffic. Jigsaw helps by placing data in the local SRAM bank, eliminating L2-to-SRAM traffic. Even though Jigsaw cannot reduce misses significantly, placing data in the local SRAM bank lets Jigsaw *miss quickly*, modestly improving

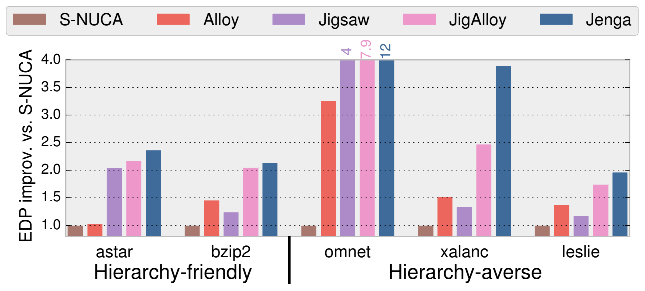


Figure 16: EDP improvement on copies of SPEC CPU2006.

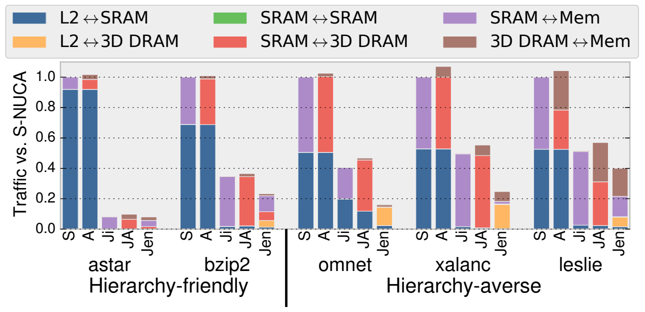


Figure 17: Traffic breakdown on copies of SPEC CPU2006.

EDP by 24%. JigAlloy combines these benefits, eliminating both L2-to-SRAM traffic and SRAM-to-Mem traffic, and improves EDP by 2×. Jenga improves EDP even further than JigAlloy because it can allocate capacity across the SRAM and DRAM boundary: for most of `bzip2`'s execution, each VH comprises a 2 MB VL1 that is split across the local SRAM bank and part of the closest DRAM bank (note the L2-to-3D-DRAM traffic). The VL1 is backed by an 8 MB VL2 also allocated in the closest DRAM bank.

The next three apps in Fig. 16 are hierarchy-averse. `omnet` has single 2.5 MB working set, so the 18 MB SRAM cannot fit all working sets. Alloy's extra capacity helps avoid most main memory misses, improving EDP by 3×. Jigsaw cannot fit all working sets, but it can increase throughput by allocating SRAM to a few apps and thereby avoid thrashing. This unfair allocation improves gmean performance significantly, by 4× in EDP. JigAlloy combines these benefits to improve EDP by nearly 8×.

Jenga improves EDP much more—by 12×—because it is not bound by rigid hierarchy. Jenga adopts a very different configuration than JigAlloy: most copies of `omnet` get *single-level* VHs with capacity split across SRAM and DRAM banks (L2-to-3D-DRAM instead of SRAM-to-3D-DRAM traffic). This hierarchy eliminates unnecessary accesses to SRAM and saves significant energy, though a few copies still get two-level VHs as in JigAlloy. Jenga achieves the same weighted speedup as JigAlloy, but does so with 54% less energy.

`xalanc` behaves similarly to `omnet` but with a 6 MB working set. This larger working set changes the relative performance of Alloy and Jigsaw: Alloy is able to fit all working sets and improves EDP by 50%, but Jigsaw is unable to fit many copies in SRAM, so its EDP improvement is lower than Alloy's. While JigAlloy still benefits from each scheme, its EDP improvement is only 2.5×. Meanwhile, Jenga improves EDP by nearly 4× by skipping SRAM and allocating single-level VHs in DRAM banks.

Finally, `leslie` has a very large working set (64 MB), and even the 1.1 GB DRAM cache cannot fit the 36 working sets. S-NUCA is dominated by main memory accesses, and both Alloy and Jigsaw are of limited help. Alloy reduces main memory accesses, and Jigsaw reduces miss time, but each improve EDP by less than 50%. JigAlloy combines the benefits of each, but Jenga does better by fitting as many copies in DRAM banks as possible and bypassing the remainder to main memory, reducing total traffic vs. JigAlloy.

These selected applications show that Jenga adapts to a wide range of behaviors. Different applications demand very different hierarchies, so the best rigid hierarchy is far from optimal for individual applications. Jenga improves performance and energy efficiency over JigAlloy by specializing the hierarchy to active applications, and its ability to allocate hierarchies out of a single resource pool is critical, e.g., letting it build VL1s using both SRAM and DRAM banks in `omnet`.

These results hold across other benchmarks. Fig. 19 shows comprehensive results when running 36 copies of all 18 memory-intensive SPEC CPU2006 apps on the 36-tile chip. Jenga improves performance over S-NUCA by gmean 47% and over JigAlloy by gmean 8%; and Jenga improves energy over S-NUCA by gmean 31%, and over JigAlloy by gmean 13%. Overall, Jenga achieves the highest EDP improvement, improving by 2.2× over the S-NUCA baseline, and *over JigAlloy by 23% and by up to 85%* (on `libquantum`).

7.2 Multi-programmed mixes

Fig. 18 shows the distribution of EDP improvement and weighted speedups over 20 mixes of 36 randomly-chosen memory-intensive SPEC CPU2006 apps. Each line shows the results for a single scheme over the S-NUCA baseline. For each scheme, workload mixes (the *x*-axis) are sorted according to the improvement achieved. Hence these graphs give a concise summary of results, but do not give a direct comparison across schemes for a particular mix.

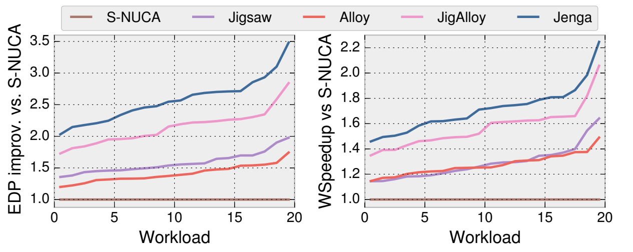


Figure 18: EDP improvement and weighted speedup of 20 mixes of 36 randomly-chosen SPEC CPU2006 apps.

Multi-programmed mixes present more challenges than copies of the same application. In a mix, different applications prefer different hierarchies and potentially different cache bank types. Although it is hard to study 36 different applications at the same time, we observe similar trends as we have shown earlier with copies: Alloy helps applications that require larger capacity, and Jigsaw improves performance by allocating SRAM resources among applications. JigAlloy combines these benefits, but Jenga further improves over JigAlloy by allocating resources in both SRAM and DRAM and creating a specialized hierarchy for each application.

Jenga thus improves EDP on all mixes over the S-NUCA baseline by up to 3.5×/gmean 2.6×, and *over JigAlloy by 25%/20%*. Jenga improves EDP because it is both faster and more energy-efficient than prior schemes. Jenga improves weighted speedup over S-NUCA by up to 2.2×/gmean 70%, and *over JigAlloy by 13%/9%*.

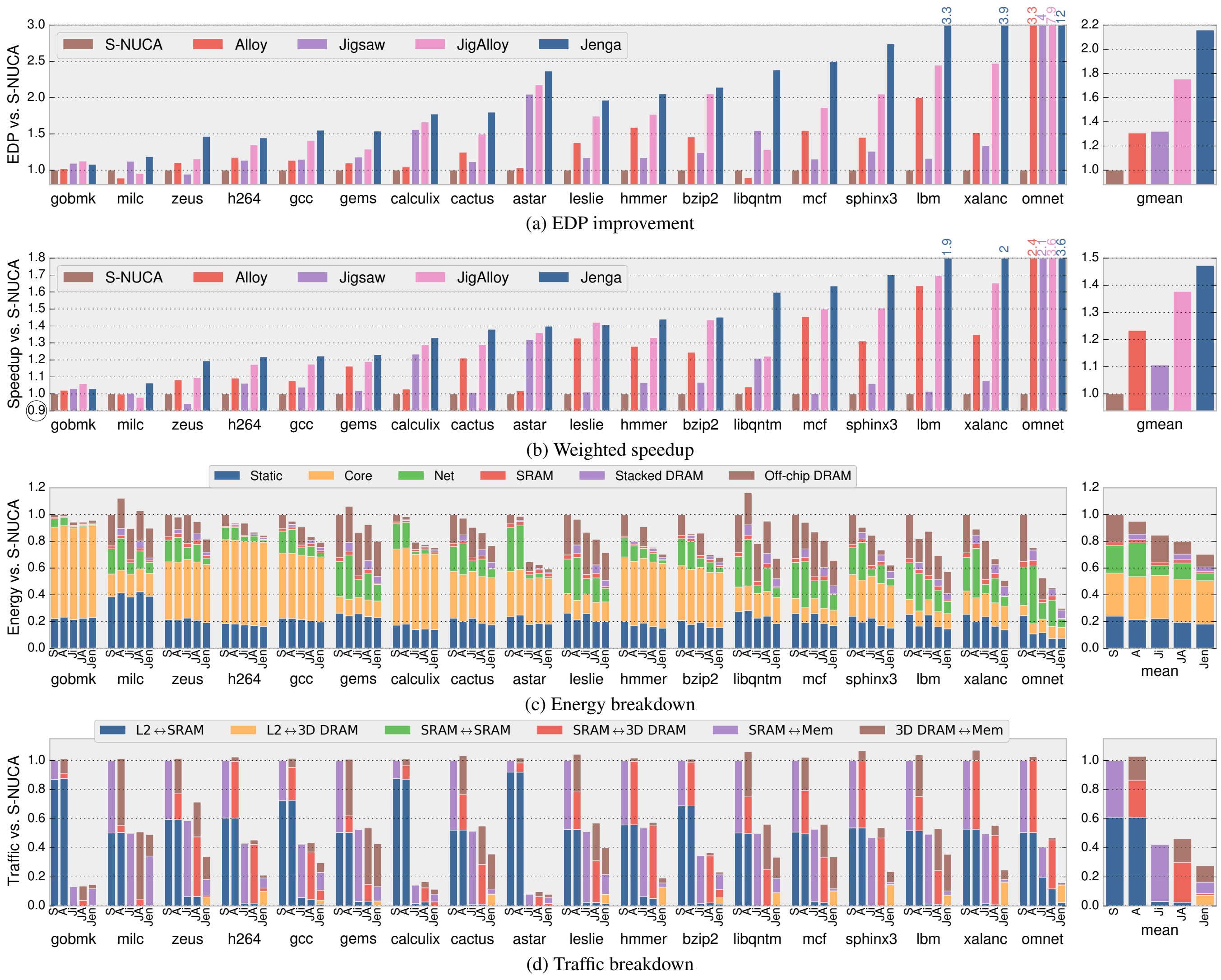


Figure 19: Simulation results on 36 concurrent copies SPEC CPU2006 apps (rate mode).

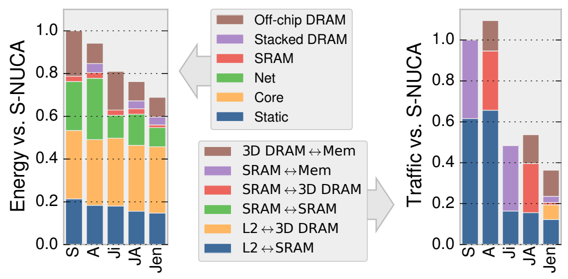


Figure 20: Average energy and traffic over the 20 mixes.

Fig. 20 compares the average energy and network traffic across mixes for each scheme. Jenga's benefits come from three factors: *(i)* improving performance so that less static energy is required to perform the same amount of work, *(ii)* placing data to reduce network energy and traffic, and *(iii)* eliminating wasteful accesses by adopting an application-specific hierarchy. In multiprogrammed mixes, wasteful accesses are mostly to SRAM banks for applications whose data settles in DRAM banks, as can be seen Jenga's lower SRAM energy vs. JigAlloy. However, in multiprogrammed mixes, Jenga's biggest energy benefit comes from data placement, as reflected in lower network energy and total traffic. (Individual apps show larger differences, e.g., Jenga eliminates accesses to DRAM banks in some multithreaded apps, see below.)

As a result, Jenga reduces energy by 31% over the S-NUCA baseline, and *by 10% over JigAlloy*. Jenga reduces network traffic by 64% over S-NUCA, and by *33% over JigAlloy*.

We have also evaluated Jenga on 20 mixes selected from all 29 SPEC CPU2006 apps. In such mixes, some apps are cache-insensitive, so the differences across schemes are smaller. Nevertheless, Jenga improves EDP over S-NUCA/JigAlloy by 76%/17% on average.

7.3 Multi-threaded applications

Jenga's benefits carry over to multi-threaded apps. Fig. 21 reports EDP improvements, speedups, and energy breakdowns for the seven cache-sensitive SPEC OMP2012 benchmarks.

Multi-threaded applications add an interesting dimension, as data is either thread-private or shared among threads. For example, `swim` and `smithwa` are apps dominated by thread-private data. Jigsaw helps them by placing private data near threads. Alloy helps when their private data does not fit in SRAM (`smithwa`). JigAlloy combines these benefits, but Jenga performs best by placing private data intelligently across SRAM and stacked DRAM. For apps with more frequent accesses to shared data (e.g. `ilbdc` and `mgrid`), Jenga improves their EDP by selecting the right banks to hold this shared data and by avoiding hotspots in DRAM banks.

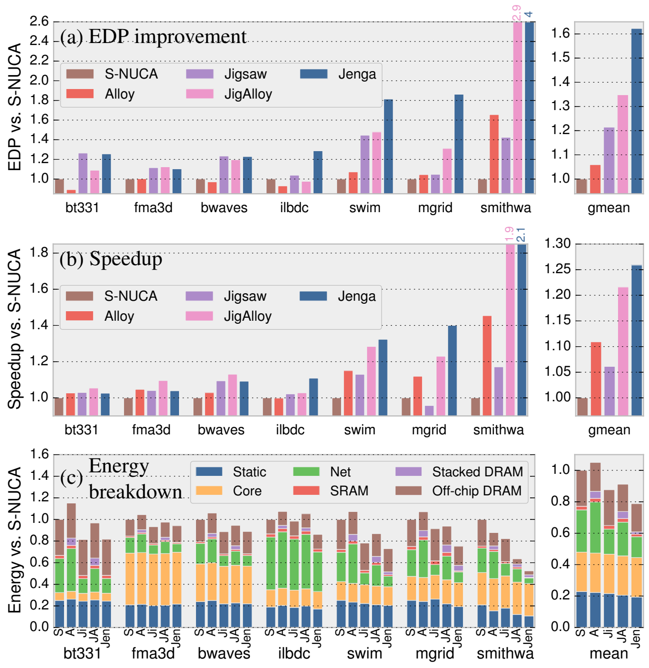


Figure 21: Simulation results for SPEC OMP2012 applications on several rigid hierarchies and Jenga.

Therefore, Jenga improves EDP by up to 4.1×/gmean 62%, and *over JigAlloy by 42%/20%*. Jenga improves performance over S-NUCA by up to 2.1×/gmean 26%, and *over JigAlloy by 30%/5%*. Jenga's energy savings on multi-threaded apps exceed its performance improvements: Jenga reduces energy by 23% over the S-NUCA baseline, and *by 17% over JigAlloy*.

7.4 Sensitivity studies

Factor analysis: Fig. 22 compares the performance of JigAlloy and different Jenga variants on copies of SPEC CPU2006: *(i)* Jenga-Single, which uses Jenga's hardware but Jigsaw's OS runtime (modified to model heterogeneous banks) to allocate a *single-level* hierarchy; *(ii)* Jenga-BW, which enhances Jenga-Single with bandwidth-aware placement; and *(iii)* Jenga, which uses both hierarchy allocation and bandwidth-aware placement. This analysis shows the benefits of Jenga's new algorithms.

Overall, though Jenga-Single suffices for some apps, bandwidth-aware placement and dynamic hierarchy allocation give significant gains on many applications. Jenga-BW avoids bandwidth hotspots and significantly outperforms Jenga-Single on memory-intensive

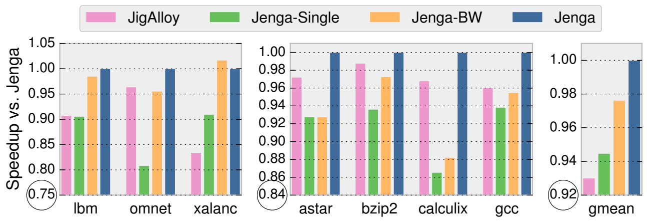


Figure 22: Performance of different Jenga techniques.

apps. Jenga-BW improves performance on `lbm`, `omnet`, and `xalanc` by 8%, 18%, and 10%, respectively.

Virtual hierarchy allocation further improves results for hierarchy-friendly apps. Jenga improves performance on `astar`, `bzip2`, `calculix`, and `gcc` by 7%, 4%, 14%, and 8% over Jenga-BW, respectively. Since they lack two-level hierarchies, Jenga-Single and Jenga-BW perform worse than JigAlloy on these apps.

The right side of Fig. 22 presents the gmean performance across all of SPEC CPU2006. Jenga-Single by itself already outperforms JigAlloy by 2%, showing the benefits of treating all memories as a single resource pool, but Jenga's new techniques are crucial to achieve the full benefits across all apps.

Prefetchers: Fig. 23 shows the EDP improvements of different schemes with stream prefetchers, where Jenga improves gmean EDP on copies/mixes of SPEC CPU2006 over S-NUCA by 2.2×/3.2×and *over JigAlloy by 23%/19%*.

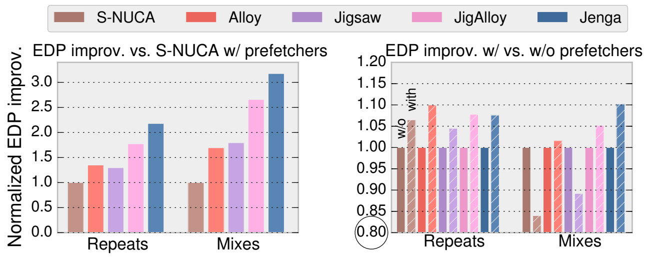


Figure 23: Sensitivity study for a system with prefetchers.

Prefetchers degrade EDP for schemes without stacked DRAM, as mispredicted prefetches waste bandwidth to main memory. This is shown on the right of the figure, where prefetchers degrade EDP by over 10% for S-NUCA and Jigsaw. We thus compare against schemes without prefetchers to be fair to prior work. Stacked DRAM's abundant bandwidth allows prefetchers to improve EDP, and Jenga's benefits over JigAlloy are similar with or without prefetchers.

Jigsaw SRAM L3 + Jigsaw DRAM L4: To show the advantage of putting all resources in a single pool, we also implement a rigid hierarchy where SRAM L3 and DRAM L4 are managed by two independent sets of Jigsaw hardware/software, which we call JigsawX2. Note that JigsawX2 has 2× hardware/software overheads vs. Jenga.

We find that JigsawX2 improves EDP over JigAlloy by only 6% on copies and by 4% on mixes. Jenga improves EDP over JigsawX2 by 13% on copies and by 17% on mixes. JigsawX2 performs poorly for three reasons: *(i)* It cannot allocate cache levels across multiple technologies. JigsawX2 is still a rigid hierarchy (SRAM L3 + DRAM L4). Since SRAM and DRAM banks are managed independently, JigsawX2 cannot build caches that span SRAM and DRAM (e.g., `bzip2` in Sec. 7.1). *(ii)* Jigsaw's data placement algorithm is bandwidth-agnostic, so it often causes hotspots in DRAM banks and performs worse than JigAlloy. For example, JigsawX2 degrades EDP by 28% over JigAlloy for copies of `omnet` due to imbalanced bandwidth utilization. *(iii)* Finally, JigsawX2's uncoordinated allocations can lead to thrashing. Although JigsawX2 carefully staggers reconfigurations to avoid unstable decisions, we find that this is insufficient in some cases, e.g., `xalanc` suffers from oscillating L3 and L4 allocations. These results show the importance of taking globally coordinated decisions to unlock the full potential of heterogeneous memories.

Hierarchy-friendly case study: Copies of SPEC CPU2006 apps are often insensitive to hierarchy, but this is not true generally. For example, cache-oblivious algorithms [22] generally benefit from hierarchies because of their inherent recursive structure. To study how Jenga helps such applications, we evaluate three benchmarks: `btree`, which performs random lookups on binary trees of two sizes (S, 100K nodes, 13 MB; and L, 1M nodes, 130 MB); `dmm`, a cache-oblivious matrix-matrix multiply on 1K×1K matrices (12 MB footprint); and `fft`, which uses cache-oblivious FFTW [21] to compute the 2D FFT of a 512×512 signal (12 MB footprint).

Fig. 24 shows the EDP improvements for those benchmarks normalized to JigAlloy. Jenga builds a two-level hierarchy *entirely in SRAM* for `btree-S`, `dmm` and `fft`, improving EDP up to 51% over JigAlloy. For `btree-L`, Jenga places the second level in stacked DRAM, and improves EDP by 62% over JigAlloy. These apps benefit from hierarchy: vs. Jenga-Single, Jenga improves EDP by 20%. These results demonstrate that application-specific hierarchies have a more significant effect on cache-intensive, hierarchy-friendly applications.

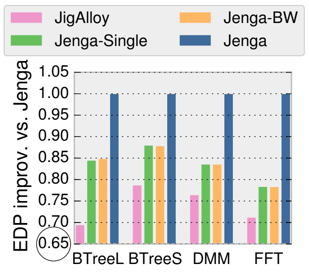


Figure 24: EDP on microbenchmarks.

7.5 Other system architectures

Finally, we evaluate Jenga under different a DRAM cache architecture to show that Jenga is effective across packaging technologies. Specifically, we model a "2.5D", interposer-based DRAM architecture with 8 vaults located at chip edges. In aggregate, this system has 2 GB of on-package DRAM and 200 GBps of bandwidth, and is similar to Intel Knights Landing [62], AMD Fiji [46], and NVIDIA Pascal [30]. Fig. 25 shows the gmean improvement of various schemes in EDP, performance, and energy under the 2.5D system. Jenga achieves similar improvements to the system with 3D-stacked DRAM.

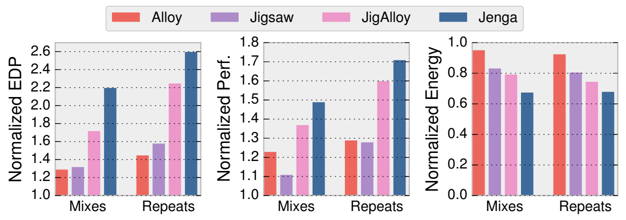


Figure 25: Improvements in EDP, performance, and energy of schemes over S-NUCA on a 2.5D DRAM system.

8 CONCLUSION

Rigid cache hierarchies waste time and energy when cache levels do not fit the working set. These overheads are problematic when differences across levels are small—as they are in emerging technologies like 3D-stacked DRAM. Prior techniques hide the latency of unwanted cache levels through speculation, trading off energy for performance. This path is unsustainable in power-limited systems. We have proposed Jenga, a software-defined, reconfigurable cache hierarchy that adopts an application-specific organization and is transparent to programs. Jenga sidesteps the performance-energy tradeoff by *eliminating* wasteful accesses. As a result, Jenga significantly improves both performance and energy efficiency over the state of the art.

ACKNOWLEDGMENTS

We sincerely thank Christina Delimitrou, Joel Emer, Mark Jeffrey, Harshad Kasture, Anurag Mukkara, Suvinay Subramanian, Guowei Zhang, our shepherd Martha Kim, and the anonymous reviewers for their helpful feedback on prior versions of this manuscript. This work was supported in part by NSF grants CCF-1318384 and CAREER-1452994, a Samsung GRO grant, and a grant from the Qatar Computing Research Institute.

REFERENCES

[1] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W Keckler, and Thomas F Wenisch. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *Proc. HPCA-21*.
[2] David H Albonesi. 1999. Selective cache ways: On-demand cache resource allocation. In *Proc. MICRO-32*.
[3] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. 2009. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Proc. HPCA-15*.
[4] Rajeev Balasubramonian, David H Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. 2003. A dynamically tunable memory hierarchy. *IEEE TOC* 52, 10 (2003).
[5] Bradford M Beckmann, Michael R Marty, and David A Wood. 2006. ASR: Adaptive selective replication for CMP caches. In *Proc. MICRO-39*.
[6] Bradford M Beckmann and David A Wood. 2004. Managing wire delay in large chip-multiprocessor caches. In *Proc. ASPLOS-XI*.
[7] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proc. PACT-22*.
[8] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *Proc. HPCA-21*.
[9] Jichuan Chang and Gurindar S Sohi. 2006. Cooperative caching for chip multiprocessors. In *Proc. ISCA-33*.
[10] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Proc. DATE*.
[11] Zeshan Chishti, Michael D Powell, and TN Vijaykumar. 2005. Optimizing replication, communication, and capacity allocation in CMPs. In *Proc. ISCA-32*.
[12] Sangyeun Cho and Lei Jin. 2006. Managing distributed, shared L2 caches through OS-level page allocation. In *Proc. MICRO-39*.
[13] Chiachen Chou, Aamer Jaleel, and Moinuddin K Qureshi. 2015. BEAR: techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *Proc. ISCA-42*.
[14] Blas A Cuesta, Alberto Ros, María E Gómez, Antonio Robles, and José F Duato. 2011. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proc. ISCA-38*.
[15] William J. Dally. 2010. GPU Computing: To Exascale and Beyond. In *Proc. SC10*.
[16] Abhishek Das, Matt Schuchhardt, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. 2012. Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores. In *Proc. DATE*.
[17] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P Jouppi. 2010. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proc. SC10*.
[18] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *Proc. MICRO-45*.

[19] Haakon Dybdahl and Per Stenstrom. 2007. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *Proc. HPCA-13*.
[20] Sean Franey and Mikko Lipasti. 2015. Tag tables. In *Proc. HPCA-21*.
[21] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. of the IEEE* 93, 2 (2005).
[22] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *Proc. FOCS-40*.
[23] Donald Gross. 2008. *Fundamentals of queueing theory*. John Wiley & Sons.
[24] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted Burton. 2014. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro* 34, 2 (2014).
[25] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proc. ISCA-36*.
[26] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. 2007. Database servers on chip multiprocessors: Limitations and opportunities. In *Proc. CIDR*.
[27] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach (5th ed.)*. Morgan Kaufmann.
[28] Enric Herrero, José González, and Ramon Canal. 2010. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proc. ISCA-37*.
[29] Andrew Hilton, Neeraj Eswaran, and Amir Roth. 2009. FIESTA: A sample-balanced multi-program workload methodology. *Proc. MoBS* (2009).
[30] Jen-Hsun Huang. 2015. Leaps in visual computing. In *Proc. GTC*.
[31] Intel. 2013. Knights Landing: Next Generation Intel Xeon Phi. In *Proc. SC13*.
[32] J. Jaehyuk Huh, C. Changkyu Kim, H. Shafi, L. Lixin Zhang, D. Burger, and S.W. Keckler. 2007. A NUCA substrate for flexible CMP cache sharing. *IEEE TPDS* 18, 8 (2007).
[33] Aamer Jaleel, Kevin Theobald, Simon C. Steely, and Joel Emer. 2010. High performance vache replacement using re-reference interval prediction (RRIP). In *Proc. ISCA-37*.
[34] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? Have it all with footprint cache. In *Proc. ISCA-40*.
[35] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, D Solihin, and Rajeev Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Proc. HPCA-16*.
[36] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel H Loh. 2015. Enabling interposer-based disintegration of multi-core processors. In *Proc. MICRO-48*.
[37] David Kanter. 2013. Silvermont, Intel's low power architecture.
[38] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proc. ASPLOS-XIX*.
[39] 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).
[40] Samira Manabi Khan, Yingying Tian, and Daniel A Jimenez. 2010. Sampling dead block prediction for last-level caches. In *Proc. MICRO-43*.
[41] Changkyu Kim, Doug Burger, and Stephen Keckler. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. ASPLOS-X*.
[42] Hyunjin Lee, Sangyeun Cho, and Bruce R Childers. 2011. CloudCache: Expanding and shrinking private caches. In *Proc. HPCA-17*.
[43] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. MICRO-42*.
[44] Gabriel H Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *Proc. ISCA-35*.
[45] Gabriel H Loh and Mark D Hill. 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proc. MICRO-44*.
[46] Joe Macri. 2015. AMD's next generation GPU and high bandwidth memory architecture: Fury. In *HotChips-27*.
[47] Niti Madan, Li Zhao, Naveen Muralimanohar, Aniruddha Udipi, Rajeev Balasubramonian, Ravishankar Iyer, Srihari Makineni, and Donald Newell. 2009. Optimizing communication and capacity in a 3D stacked reconfigurable cache hierarchy. In *Proc. HPCA-15*.
[48] Michael R Marty and Mark D Hill. 2007. Virtual hierarchies to support server consolidation. In *Proc. ISCA-34*.
[49] Javier Merino, Valentin Puente, and Jose Gregorio. 2010. ESP-NUCA: A low-cost adaptive non-uniform cache architecture. In *Proc. HPCA-16*.
[50] Micron. 2013. 1.35V DDR3L power calculator (4Gb x16 chips). (2013).
[51] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. Whirlpool: Improving dynamic cache management with static data classification. In *Proc. ASPLOS-XXI*.
[52] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proc. MICRO-40*.
[53] Moinuddin Qureshi and Gabriel Loh. 2012. Fundamental latency trade-offs in architecting DRAM caches. In *Proc. MICRO-45*.
[54] Moinuddin K. Qureshi. 2009. Adaptive spill-receive for robust high-performance caching in CMPs. In *Proc. HPCA-15*.
[55] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proc. ISCA-34*.
[56] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. MICRO-39*.
[57] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *Proc. MICRO-43*.
[58] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. ISCA-38*.
[59] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proc. ISCA-40*.
[60] Jaewoong Sim, Jaekyu Lee, Moinuddin K Qureshi, and Hyesoon Kim. 2012. FLEXclusion: Balancing cache capacity and on-chip bandwidth via flexible exclusion. In *Proc. ISCA-39*.
[61] Allan Snavely and Dean M Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. ASPLOS-IX*.
[62] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro* 36, 2 (2016).
[63] Jeff Stuecheli. 2013. POWER8. In *HotChips-25*.
[64] Dong Hyuk Woo, Nak Hee Seong, Dean L Lewis, and H-HS Lee. 2010. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *Proc. HPCA-16*.
[65] Leonid Yavits, Amir Morad, and Ran Ginosar. 2014. Cache hierarchy optimization. *IEEE CAL* 13, 2 (2014).
[66] Michael Zhang and Krste Asanovic. 2005. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proc. ISCA-32*.