Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2015-035 December 19, 2015

Jenga: Harnessing Heterogeneous Memories through Reconfigurable Cache Hierarchies

Nathan Beckmann, Po-An Tsai, and Daniel Sanchez

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Jenga: Harnessing Heterogeneous Memories through Reconfigurable Cache Hierarchies

Nathan Beckmann Po-An Tsai Daniel Sanchez
Massachusetts Institute of Technology
{beckmann,poantsai,sanchez}@csail.mit.edu

ABSTRACT

Conventional memory systems are 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, because working sets settle at the smallest (and fastest) level they fit in. However, rigid hierarchies also cause significant overheads, because each level adds latency and energy even when it does not capture the working set. In emerging systems with heterogeneous memory technologies such as stacked DRAM, these overheads often limit performance and efficiency.

We propose Jenga, a reconfigurable cache hierarchy that avoids these pathologies and approaches the performance of a hierarchy optimized for each application. Jenga monitors application behavior and dynamically builds *virtual cache hierarchies* out of heterogeneous, distributed cache banks. Jenga uses simple hardware support and a novel software runtime to configure virtual cache hierarchies.

On a 36-core CMP with a 1 GB stacked-DRAM cache, Jenga outperforms a combination of state-of-the-art techniques by 10% on average and by up to 36%, and does so while saving energy, improving system-wide energy-delay product by 29% on average and by up to 96%.

1. INTRODUCTION

Memory accesses often limit the performance and efficiency of current multicores, and the trend towards lean and specialized cores is placing mounting pressure on the energy and latency of memory accesses [14, 33]. Consequently, cache hierarchies are becoming more sophisticated in two key dimensions. First, cache hierarchies are starting to combine multiple technologies with disparate tradeoffs, such as SRAM and stacked DRAM [22, 38]. Second, hierarchies are becoming increasingly distributed and non-uniform (NUCA [35]): each core enjoys cheap accesses to physically-close cache banks, but accesses to far-away banks are expensive.

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 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, and 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

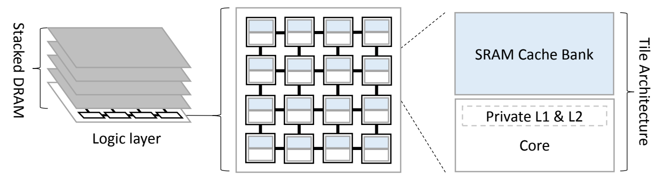


Figure 1: A modern multicore with distributed, on-chip SRAM banks and a 3D-stacked DRAM cache.

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*, since it adds cache accesses without yielding many hits.

We begin by characterizing the benefits of application-specific hierarchies over rigid ones (Sec. 2). We find that the optimal application-specific hierarchy varies widely, both in the number of levels and their sizes. Rigid hierarchies are forced to cater to the conflicting needs of diverse applications, and even the best-performing rigid hierarchy hurts applications that desire a markedly different configuration, degrading 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 [47].

To exploit this opportunity, we present Jenga, a reconfigurable cache architecture that builds single- or multi-level *virtual cache hierarchies* tailored to each application (Sec. 3). Jenga provides four desirable properties: *(i)* Jenga is flexible, adopting multi-level hierarchies for hierarchy-friendly applications, and a single level for hierarchy-averse ones. *(ii)* Jenga manages scarce capacity wisely among competing applications, placing data close to where it is used and preventing harmful interference. *(iii)* Jenga continuously monitors and reconfigures hierarchies, adapting to dynamic application behavior. And *(iv)* Jenga is cheap to implement.

Jenga builds on prior work on partitioned NUCA caches [6, 8, 36], specifically Jigsaw [6, 8], which constructs *single-level virtual caches* out of *homogeneous SRAM* banks (Sec. 4). Jigsaw performs well in its proposed context. However, it does not handle heterogeneous banks, cannot construct multi-level hierarchies, and does not account for the limited bandwidth of stacked DRAM. Jenga solves these problems through three novel contributions:

- We design straightforward hardware extensions that let software define multi-level virtual cache hierarchies, monitor their behavior, and reconfigure them on the fly (Sec. 5).
- We present *adaptive hierarchy allocation* (Sec. 6.2), which finds the right number of virtual cache levels and the size of each level given the demand on the memory system.
- We introduce *bandwidth-aware data placement* (Sec. 6.3) to account for limited bandwidth when placing data among banks, avoiding hotspots that hamper existing techniques that only consider limited cache capacity.

We evaluate Jenga on a 36-core CMP with 18 MB of on-die SRAM and 1.1 GB of 3D stacked DRAM. (Jenga works equally well on other configurations, e.g. “2.5D” DRAMs connected via an interposer.) Compared to a combination of state-of-the-art NUCA and stacked DRAM techniques, Jigsaw and Alloy [47], Jenga improves performance by 10% on average and by up to 36%. Jenga also reduces energy, improving system-wide EDP by 29% on average and by up to 96%. By contrast, adding a DRAM cache as a rigid hierarchy improves performance, but degrades energy efficiency. We also show that each of our contributions is key to achieve consistently high performance. We conclude that the rigid, multi-level organization of current systems is ill-suited to many applications. Perhaps future memory systems should not be organized as a rigid hierarchy, but rather as a flexible memory system with heterogeneity exposed 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 (later sections evaluate multi-program and multi-threaded workloads). The core has fixed 32 KB L1s and a 128 KB L2. To find the best hierarchy for each app, we consider both NUCA SRAM L3 and stacked DRAM L4 caches of different sizes. Latency, energy, and area are derived using CACTI [43] 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 [47]. Larger caches have more banks, placed as close to the core as possible and connected through a mesh NoC. Sec. 7.1 details our methodology further.

Larger caches are more expensive [24, 25]: Area and static power increase roughly linearly with size, while access latency and energy scale roughly with its square root [54]. We evaluate SRAM caches from 512 KB to 32 MB with access latencies from 9 to 45 cycles and energy from 0.2 to 1.7 nJ, and stacked DRAM caches from 128 MB to 2 GB with access latencies from 42 to 74 cycles and access 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 energy-delay product (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: seven out of the 18 memory-intensive applications we consider prefer a single-level organization, and their preferred sizes vary widely, especially at the L3. (Because Alloy is direct-mapped, some apps prefer larger caches than their working set, e.g. `libquantum` prefers 256 MB of stacked DRAM for its 32 MB working set.)

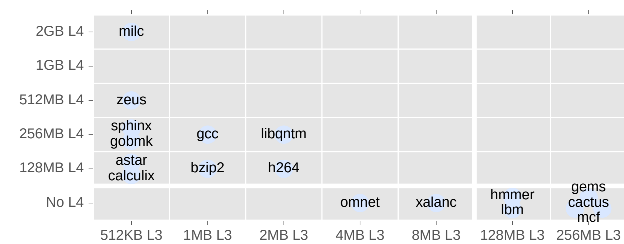


Figure 2: Best application-specific hierarchies (by EDP) vary greatly in the number of levels and their sizes.

2. Rigid hierarchies sacrifice performance and efficiency. The rigid hierarchy that maximizes gmean EDP across these applications consists of a 512 KB SRAM L3 and a 256 MB DRAM L4. This is logical, since six (out of 18) apps want a 512 KB L3 and seven want a 256 MB L3 or L4. However, Fig. 3 shows that applications that desire a different hierarchy can do significantly better: up to 81% better EDP and 51% higher performance. By comparing Fig. 2 and Fig. 3, we see that the potential benefit is directly correlated to how different the application-specific hierarchy is from the rigid one.

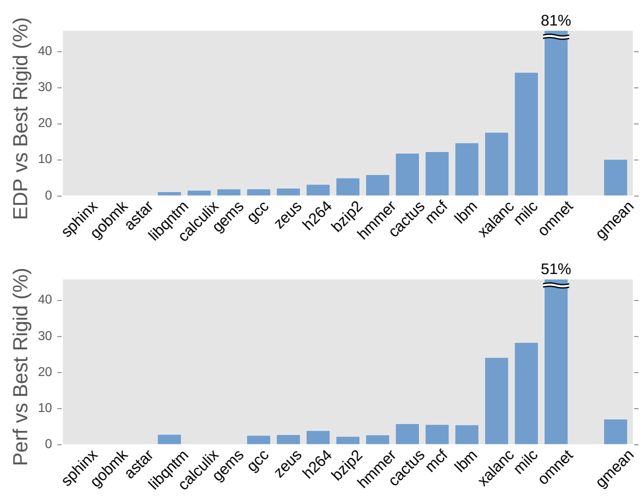


Figure 3: EDP and performance improvements of the best (EDP-optimal) application-specific cache hierarchy over the best rigid hierarchy (512 KB L3 + 256 MB L4).

Fig. 3 also shows that, 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. We will exploit this correlation by optimizing for performance in Jenga; as we will show later, this strategy also helps EDP.

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.

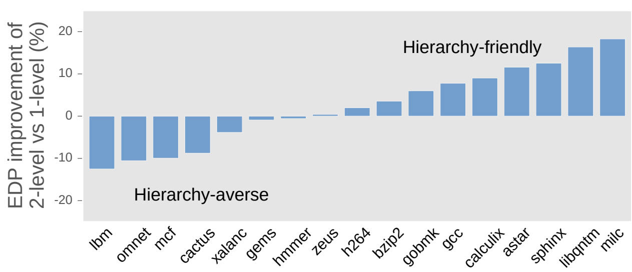


Figure 4: EDP improvement of the best two-level application-specific hierarchy over the best single-level one. The best hierarchy is the better of the two.

Fig. 4 shows that applications often have strong preferences about hierarchy: 6 out of the 18 applications are *hierarchy-averse*, and two-level organizations degrade their EDP by up to 13%. 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 L3), the best rigid two-level hierarchy (a 512 KB L3 plus a 256 MB 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).

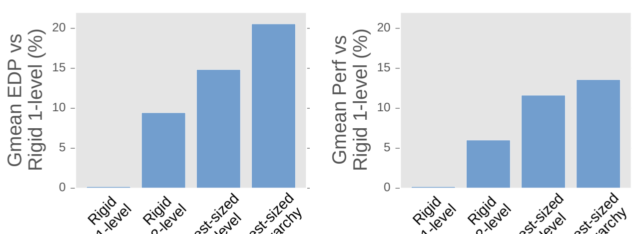


Figure 5: Gmean EDP and performance improvements of rigid 2-level, application-specific 1-level, and best application-specific hierarchies.

Overall, hierarchy offers only modest benefits in rigid designs since it is hampered by hierarchy-averse applications: just 9% improved gmean EDP and 6% 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 15% gmean EDP and 11% performance. Building multi-level hierarchies (when appropriate) yields further improvements, by 20% gmean EDP and 13% gmean performance. This motivates the need for virtual cache *hierarchies*.

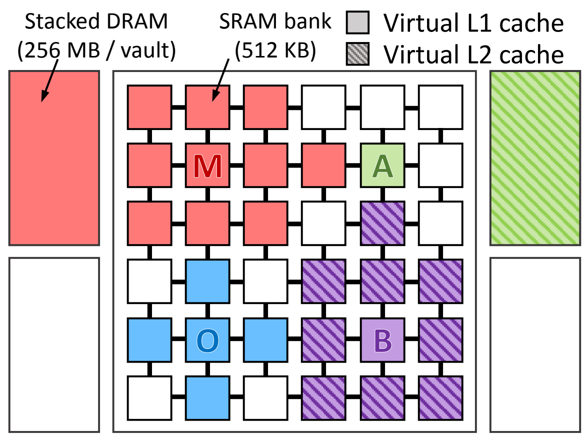


Figure 6: A 36-core Jenga system running four applications. Jenga gives each a custom *virtual cache hierarchy*.

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 bank. There are four stacked DRAM vaults. Jenga builds a custom *virtual cache hierarchy* out of the shared cache banks (i.e., 512 KB SRAM banks and stacked DRAM vaults, 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 placement caches its working set at minimum latency and energy. Misses from `omnet` go directly to main memory, *and do not access stacked DRAM*. 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 vault. Crucially, although `mcf`'s virtual cache uses both SRAM and stacked DRAM, it is still accessed as a single-level cache, 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 stacked DRAM vault 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 stacked DRAM.

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 techniques [35] reduce the latency and energy of large caches. Static NUCA (S-NUCA) [35] spreads data across all banks with a fixed line-bank mapping, and exposes a variable bank access latency. S-NUCA is simple, but only reduces latency and energy by a constant factor. Dynamic NUCA (D-NUCA) schemes improve on S-NUCA by adaptively placing data close to the requesting core [3,4,5,9,11,12,23,29,41,45,55] 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 [5,35], with each level consisting of banks at a given distance. However, these schemes caused excessive data movement and thrashing [5]. 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 [17,41,55], or spilling lines to other banks and relying on a global directory to access them [45]. Finally, Cho and Jin [12], Awasthi et al. [3], R-NUCA [23] and Jigsaw [6] 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 multiple-lookup NUCAs [6, 8, 23]. This effect is analogous to results in Sec. 2: multiple-lookup 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 enough capacity to fit the working set at minimum latency and energy. In other words, single-lookup D-NUCAs try to find the best-sized, single-level hierarchy (Fig. 5). In particular, Jigsaw [6, 8] addresses this problem by letting software define *virtual caches*.

However, as we have seen in Sec. 2, systems with heterogeneous memory technologies introduce a wider tradeoff in latency and capacity. Thus, hierarchy is sometimes desirable, so long as it is used only when beneficial.

Stacked DRAM: Prior work has proposed using stacked DRAM as either OS-managed memory [1,16,31,53] or an extra layer of cache [18,30,39,47]. When used as a cache, the main challenge is its high access latency.

Much recent work has focused on the structure of cache arrays. Several schemes [18,30,39,40] place tags in SRAM, reducing latency at the cost of SRAM capacity. Alloy [47] 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.

Some prior work reduces the overhead of hierarchy. Dynamic cache bypassing [30, 34] will not install lines at specific levels when they are predicted non-reused. However, these schemes must still check each level for correctness, wasting considerable energy and bandwidth (contrast with `omnet` in Sec. 3, which bypasses stacked DRAM entirely). Similarly, hit/miss prediction [47] reduces miss penalty by speculatively issuing main memory accesses in parallel with lookups. These techniques improve performance, but are wasteful on mispredictions and also must check all levels for correctness. Jenga complements these techniques (we use hit/miss prediction in our evaluation), but improves upon them by *eliminating hierarchy when it is not useful*.

5. JENGA HARDWARE

Jenga consists of hardware and software components. In hardware, Jenga extends Jigsaw [6,8] in straightforward ways to support DRAM cache banks and multi-level virtual hierarchies. We now present these hardware components, emphasizing differences from Jigsaw at the end of the section. Sec. 6 presents Jenga's OS runtime.

Overview: Jenga hardware provides four facilities. First, it lets software organize collections of cache banks into *virtual cache hierarchies* (VHs) with one or more levels. Second, Jenga hardware lets software map data pages to those virtual hierarchies. All accesses to a page then go through the virtual hierarchy. Third, Jenga hardware provides monitors that gather the miss curves of each virtual hierarchy. Fourth, Jenga hardware provides fast reconfiguration mechanisms.

Fig. 7 shows the tiled CMP we use to present Jenga. Each tile has a core, a directory bank, and an SRAM cache bank. The CMP also has distributed stacked DRAM vaults. Jenga also supports other configurations, e.g. "2.5D" systems that connect stacked DRAM vaults via an interposer (Sec. 7.6).

Virtual hierarchies: To dynamically adapt the memory system, Jenga allows software to define many virtual hierarchies cheaply (several per thread), sized at fine granularity and placed among physical cache banks. Jenga supports virtual hierarchies of one or two levels, called VL1 and VL2 respectively. (Recall that Jenga operates in the shared cache banks—VL1 and VL2 are different from each core's private L1s and L2.) Furthermore, to support more VHs than physical banks, each bank can be optionally partitioned. Partitioning lets multiple VHs share a single physical bank without interference.

In our evaluation, we partition SRAM but do not partition stacked DRAM. This is for two reasons: *(i)* SRAM capacity is scarce and highly contended, but stacked DRAM is not, and *(ii)* high associativity is expensive in DRAM caches [47], making partitioning more costly.

The key hardware support is the *virtual hierarchy table* (VHT), a small structure that provides a configurable layer of indirection between a line's address and its physical location. Jenga also makes minor changes to other system components.

Mapping data to banks: Software maps data to VHs using the virtual memory system. Each page table entry is extended with a VH id. On a private cache miss, the core uses the line address and its VH id to find its VL1 bank and (possibly) VL2 bank, using the virtual hierarchy table (VHT), shown in Fig. 8.

Each of the VHT entries consists of two arrays of N bank ids each; one array for the VL1 configuration, and another for the VL2 configuration. As shown in Fig. 8a, to find the bank and bank partition ids the address is hashed, and the hash value (mod N) selects the bucket. Having more buckets than physical banks allows Jenga to spread accesses across bank partitions in proportion to their capacities. For example, Fig. 8c shows a VL1 consisting of bank partitions X and Y. X is 128 KB and Y is 384 KB (3× larger). By setting one-fourth of VHT entries in the VL1 descriptor to X and three-fourths to

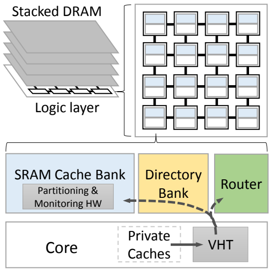
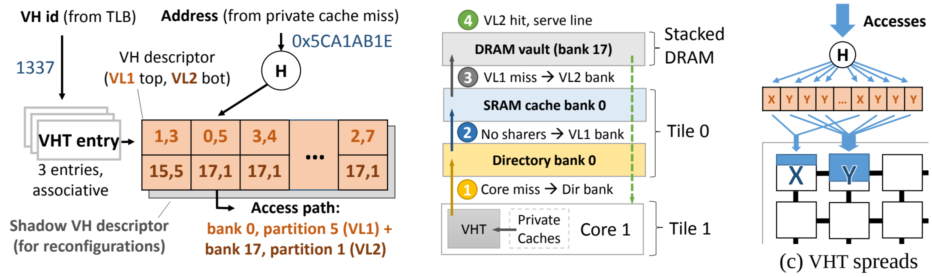


Figure 7: 16-tile Jenga with on-chip SRAM and stacked DRAM banks.



(a) VHT organization and example lookup (b) Access path for a VL2 hit (c) VHT spreads accesses across capacity

Figure 8: The virtual hierarchy table (VHT) finds the VL1 and VL2 banks for each private cache miss. Jenga sets VHT entries to spread accesses evenly across virtual cache capacity.

Y, Y receives 3× more accesses than X. Thus, together X and Y behave like a 512 KB cache [6, 7].

Because VHT accesses are narrow (12 bits at 36 tiles), Jenga accesses the VHT in parallel with every private L2 access so that misses can be routed to their VL1 bank immediately. Fig. 8b shows how the VHT controls the directory, VL1, and VL2 banks traversed on each access.

Types of VHs: Jenga's OS-level runtime creates one thread-private VH per thread, one per-process VH for each process, and a global VH. Data used by a single thread is mapped to its thread-private VH; data used by multiple threads in the same process is mapped to the per-process VH; and data used by multiple processes is mapped to the global VH. Pages are reclassified efficiently [6] (e.g., when a thread-private page is accessed by another thread, it is remapped to the per-process VH), though in steady state this happens rarely.

Coherence: 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. However, this requires a careful mapping of lines to directory banks to keep latency low.

Jenga maintains coherence using two invariants. First, private cache misses check the directory before accessing the VL1. This ensures private caches stay coherent. Second, in the shared levels, lines do *not* migrate in response to accesses. Instead, between reconfigurations, all accesses to the same line follow the same path (i.e., through the same VL1 and VL2 banks). For example, in Fig. 7, if a line maps to the top-right SRAM bank, accesses to that line will go to that bank regardless of which core issued the access. This mapping only changes infrequently, when the VH is reconfigured. Having all accesses to a given address follow the same path maintains coherence automatically. In particular, note that no directory lookups are needed between VL1 and VL2 accesses.

To reduce directory latency, Jenga assigns a directory bank near the VL1 bank. For example, in Fig. 7, VL1 accesses that map to the SRAM cache bank in the top-right tile also use the top-right tile's directory bank; accesses to VL1 DRAM banks use directory banks near the DRAM bank's TSVs. This optimization is critical to Jenga's scalability and performance: if directories were mapped statically, then directory latency would increase with system size and eliminate most 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* [15].

Finally, pages known to be private to a single thread do not need coherence and skip the directory entirely.

Monitoring: Jenga uses utility monitors [46] to gather the miss curve of each VH. Miss curves allow finding the right virtual hierarchies without trial and error. A small fraction (∼1%) of VHT accesses are sampled into these monitors. We use geometric monitors (GMONs) [8], set-associative structures that sample unevenly across ways to achieve both large coverage and fine resolution with a moderate number of ways.

Reconfiguration support: Periodically (every 100 ms), Jenga software changes the configuration of some or all VHs. Hardware support makes reconfigurations efficient [8]. Directory and cache banks walk their tag arrays and invalidate lines whose location has changed. To avoid pausing cores while this happens, each core copies the VH descriptors into the *shadow* descriptors, and updates the primary VH descriptors. While banks are invalidating old entries, accesses that miss in the line's new bank also check the old bank using shadow descriptors. When all banks finish their invalidations, cores stop checking old locations. The shadow descriptors are thus only in use briefly during reconfiguration (e.g., for a few ms every 100 ms), and a single set of shadow descriptors suffices.

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 bank partition ids). A VHT has 3 entries, as each thread only accesses 3 different VHs. Each of the 3 entries has two descriptors (shadow descriptors, shown in Fig. 8), making the VHT ∼2.4 KB. We use 8 KB GMONs, and have two monitors per tile. In total, Jenga adds ∼20 KB per tile, less than 720 KB for a 36-tile CMP, 4% overhead over the SRAM cache banks.

Jenga extensions over Jigsaw hardware: Much of Jenga's hardware is similar to Jigsaw. The main extensions are:

- Jenga supports two-level virtual cache hierarchies.
- Jenga supports partitioned or unpartitioned 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.

6. JENGA SOFTWARE

Periodically (e.g., every 100 ms), Jenga's software runtime reconfigures virtual hierarchies to minimize data movement. Each reconfiguration consists of four steps, shown in Fig. 9:

1. Read miss curves from hardware monitors.
2. Divide cache capacity into one- or two-level virtual cache hierarchies (Sec. 6.2). This algorithm returns the number of levels and size of each level, but does not compute where they are placed.

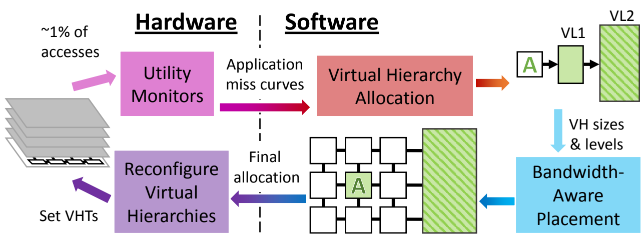


Figure 9: Overview of Jenga reconfigurations. Hardware profiles applications; software periodically reconfigures virtual hierarchies to minimize total access latency.

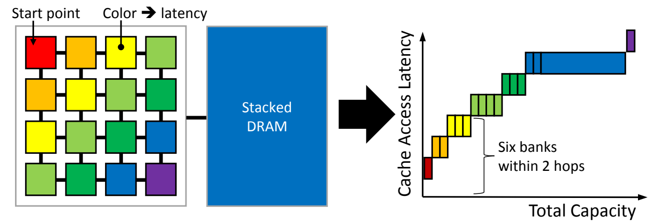


Figure 11: Jenga models access latency by sorting capacity according to latency, producing the *marginal latency curve* that yields the latency to the next available bank. Averaging this curve gives the average access latency.

3. Place each virtual cache hierarchy in cache banks, accounting for the limited bandwidth of stacked DRAM (Sec. 6.3).
4. Initiate a reconfiguration by updating the VHTs.

The resulting Jenga runtime is cheap, taking 450 lines of code and 0.4% of system cycles (Sec. 6.4).

Jenga makes major extensions to Jigsaw's runtime to support hierarchies and cope with limited stacked DRAM bandwidth. We begin by briefly reviewing Jigsaw's algorithms.

6.1 Jigsaw Algorithms

Jigsaw chooses virtual cache sizes to minimize end-to-end access latency [8]. Fig. 10 shows that latency consists of two components: 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 the virtual cache. Jigsaw uses these curves to allocate capacity among virtual caches, trying to minimize total latency with the Peekahead algorithm [6]. Since the same trends hold for energy, Jigsaw also reduces energy and improves EDP.

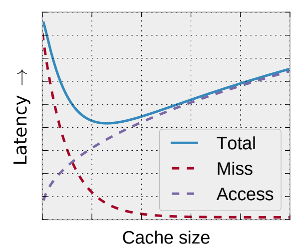


Figure 10: Access latency broken into cache access latency (increasing) and miss latency (decreasing).

Jigsaw constructs the miss latency curve from the hardware miss curve monitors. The miss latency at a given cache size is just the expected number of misses (read directly from monitor) times the memory latency.

Jigsaw constructs the cache access latency curve using the system configuration. Fig. 11 shows how. Starting from each tile (e.g., top-left in figure), Jigsaw 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 because its average value from 0 to s gives the average access latency to a cache of size s.

After sizing virtual caches, the last step is to place virtual caches in physical cache banks. Jigsaw places data in two passes. First, virtual caches take turns greedily grabbing capacity in their most favorable banks. Second, virtual caches trade capacity to move more-intensely accessed data closer to where it is used, reducing access latency [8].

Jenga-SINGLE: Even without major changes, Jigsaw's algorithms can already outperform rigid hierarchies on most apps. We call this scheme Jenga-SINGLE, which uses Jigsaw's algorithms to build single-level virtual caches out of SRAM and stacked DRAM. Jenga-SINGLE is the simplest scheme that exploits the opportunity presented in Sec. 2: As we saw, the best-sized cache often outperforms the best rigid two-level hierarchy. We find that Jenga-SINGLE suffices for applications that are hierarchy-averse or do not stress memory bandwidth.

Jenga-SINGLE places working sets near applications, using whatever bank types are most appropriate. In other words, Jenga-SINGLE just treats stacked DRAM vaults as a different "flavor" of cache bank—there is no hierarchy beyond the private caches. For example in Fig. 6, Jenga-SINGLE would place `omnet`'s working set in nearby SRAM, avoiding stacked DRAM, and spread `mcf`'s across SRAM and stacked DRAM.

Jenga-SINGLE requires trivial changes to Jigsaw's software: First, we must model banks with different access latencies and capacities, i.e. SRAM banks vs. stacked DRAM vaults. Second, we must model the network latency to TSVs (or interposer I/Os in "2.5D" systems). That is, stacked DRAM vaults are also NUCA, and in fact a stacked DRAM vault can have lower latency than far-away SRAM banks. Fig. 11 already incorporates these extensions by accounting for the latency and capacity of stacked DRAM in the marginal latency curve.

6.2 Virtual Hierarchy Allocation

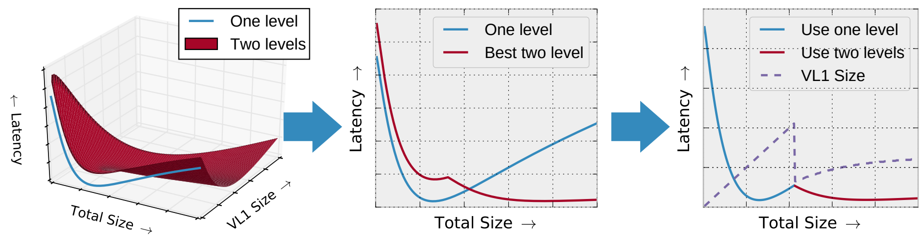
Jenga-SINGLE works well for many apps, but for memory-intensive or hierarchy-friendly applications, there is room for improvement. Jenga extends Jigsaw to build two-level hierarchies for hierarchy-friendly applications, significantly improving performance and EDP for these apps.

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 [54]: 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) Full hierarchy model (b) Choose best VL1 (c) Choose best hierarchy

Figure 12: 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.

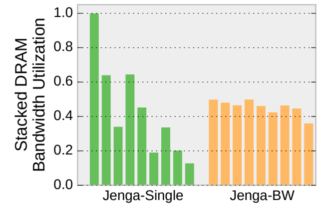


Figure 13: Distribution of bandwidth across DRAM vaults on `lbm`. Jenga-SINGLE suffers from hotspots, while Jenga-BW doesn't.

a conservative, *inclusive* hierarchy model. In fact, Jenga uses non-inclusive caches, but modeling non-inclusion is difficult. 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 [51], a poor tradeoff with stacked DRAM.

The VL2 access latency is modeled similarly to the access latency of a single-level virtual cache (Fig. 11). The difference is that rather than averaging the marginal latency starting from zero, we average the curve starting from the VL1 size (since the VL2 is placed after the VL1).

Fig. 12 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. 12a). 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. 12b). Finally, Jenga compares the latency of single- and two-level hierarchies to determine at which sizes this application is hierarchy-friendly or -averse (Fig. 12c). This choice in turn implies the hierarchy configuration (i.e. VL1 size for each total size), shown on the second y-axis in Fig. 12c.

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 [6, 46] to allocate capacity between virtual hierarchies. The allocated sizes imply the desired configuration (the VL1 size in Fig. 12c), which Jenga finally places as described in Sec. 6.3.

Efficient implementation: Evaluating every point on the surface in Fig. 12a 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. 11).

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, stacked DRAM has limited bandwidth compared to SRAM. Since Jenga-SINGLE ignores differences between banks, it produces pathologies for some apps that access large working sets intensely.

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 DRAM vaults at each reconfiguration. Keeping a moving average can reduce this thrashing, but since reconfigurations are relatively 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 stacked DRAM on every step with a simple M/D/1 queue latency model. This technique, called Jenga-BW, eliminates hotspots on individual stacked DRAM vaults, 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, and 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 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. 14 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.

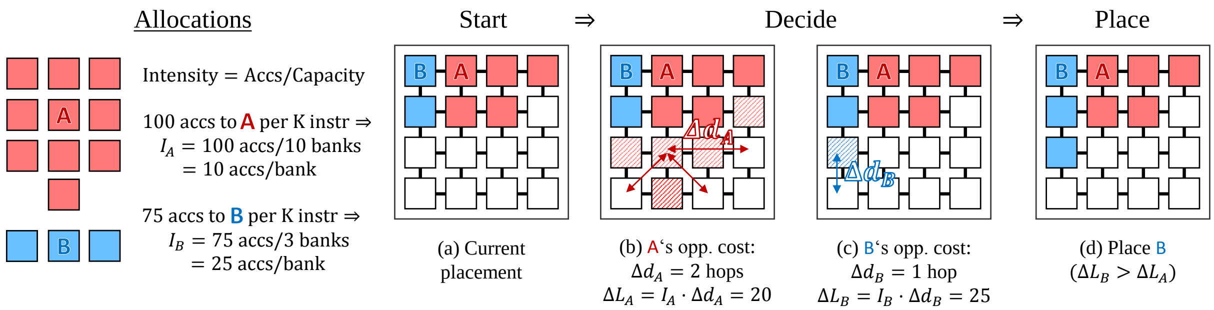


Figure 14: Jenga reduces total access latency by considering two factors when placing a chunk of capacity: *(i)* how far away the capacity will have to move if not placed, and *(ii)* how many accesses are affected (called the *intensity*).

For example, the step starts with the allocation in Fig. 14a. In Fig. 14b and Fig. 14c, each virtual cache (A and B) sees where its allocation would fit on chip. Note that it does not actually place this capacity, it just reads its marginal latency curve (e.g., Fig. 11). 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. We approximate the opportunity cost as: $\Delta L \approx I \times \Delta d$.

In Fig. 14d, Jenga chooses to place a chunk of B's allocation since B's opportunity cost is larger than A's. Fig. 14 places a full bank at each step; our Jenga implementation places at most $1/16^{\text{th}}$ 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 vaults, 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 vaults 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 $\rho/(2\mu \times (1-\rho))$ [21,44]. After updating the bank latency, Jenga sorts banks for later steps. Re-sorting is cheap because a single bank moves by at most a few places.

Fig. 13 shows a representative example of how Jenga-BW balances accesses across DRAM vaults on `lbm`. Each bar plots the access intensity to different DRAM vaults in Jenga-SINGLE (green) and Jenga-BW (yellow). Jenga-SINGLE leads to hotspots, overloading some vaults while others are idle, whereas Jenga-BW evenly spreads accesses across vaults. As a result, Jenga-BW improves performance by 10%, energy by 6%, and EDP by 17%. Similar results hold for other apps (e.g., `omnet` and `xalanc`, Sec. 7.5).

Cores	36 cores, x86-64 ISA, 2.4 GHz, Silvermont-like OOO [32]: 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 [37]
L1 caches	32 KB, 8-way set-associative, split D/I, 3-cycle latency; 15/33 pJ per hit/miss [43]
L2 caches	128 KB private per-core, 8-way set-associative, inclusive, 6-cycle latency; 46/93 pJ per hit/miss [43]
Coherence	MESI, 64 B lines, no silent drops; sequential consistency
Global NoC	6×6 mesh, 128-bit flits and links, X-Y routing, 3-cycle pipelined routers, 1-cycle links; 63/71 pJ per router/link flit traversal, 12/4 mW router/link static power [37]
SRAM banks	18 MB, one 512 KB bank per tile, 4-way 52-candidate zcache [48], 9-cycle bank latency, Vantage partitioning [49]; 240/500 pJ per hit/miss, 28 mW/bank static power [43]
Stacked DRAM banks	1152 MB, one 128 MB vault per 4 tiles, Alloy with MAP-I DDR3-3200 (1600 MHz bus), 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 [42]
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 CMP.

6.4 Overheads

Jenga's configuration runtime, including both VH allocation and bandwidth-aware placement, takes less than 450 lines of C++ and completes in just 40 Mcycles, or 0.4% of system cycles at 36 tiles. It also scales nearly in proportion to system size, taking 0.3% of system cycles at 16 tiles.

7. EVALUATION

7.1 Experimental Methodology

Modeled system: We perform microarchitectural, execution-driven simulation using zsim [50], and model a 36-core CMP with on-chip SRAM and stacked DRAM caches, as shown in Fig. 7. Each tile has one lean 2-way OOO core similar to Silvermont [32] with private L1 instruction and data caches and a unified L2. Table 1 details the system's configuration.

We compare seven 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 [47]. 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)* with-

Figure 15: Simulation results on 36 concurrent copies SPEC CPU2006 apps (rate mode).

out 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 have also evaluated R-NUCA [23] in the L3, and as in prior work [6, 8] it performs worse than Jigsaw.)

Finally, we evaluate Jenga variants: *(v)* Jenga-SINGLE, which does not build hierarchies; *(vi)* Jenga-BW, which accounts for limited stacked DRAM bandwidth; and *(vii)* Jenga, which also builds two-level hierarchies when beneficial.

All organizations that use Alloy employ MAP-I memory access predictor [47]. 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.

Jigsaw and Jenga use Vantage [49] to partition SRAM cache banks. Since stacked DRAM capacity is abundant, Jenga does not partition stacked DRAM.

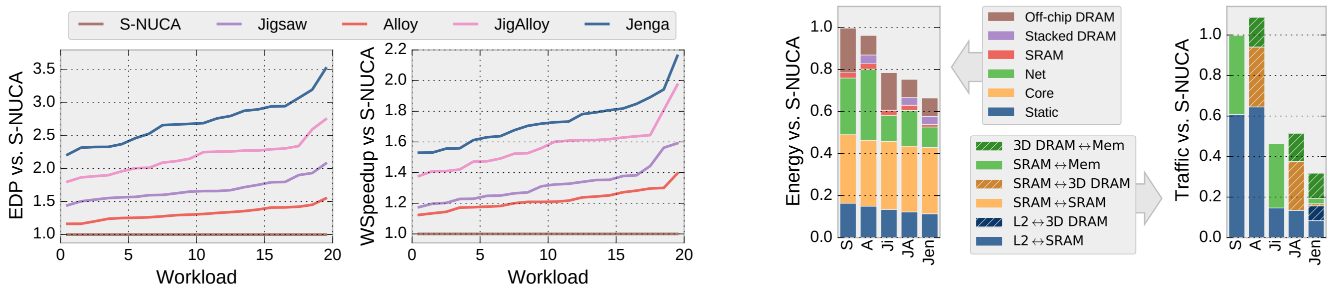
Workloads: Our workload setup mirrors prior work [8]. We simulate mixes and copies of SPEC CPU2006 apps. We use the 18 SPEC CPU2006 apps with ≥5 L2 MPKI (Fig. 15) and fast-forward all apps in each mix for 20 B instructions. We use a fixed-work methodology and equalize sample lengths to avoid sample imbalance, similar to FIESTA [26]. 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 consider only the first I_i instructions of each app to report performance.

We simulate multithreaded SPEC OMP2012 apps that are sensitive to improvement in cache hierarchy (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 [52] as our performance metric, and EDP improvement ($^{Perf \cdot Energy_{base}}/_{Perf_{base} \cdot Energy}$) to summarize performance and energy gains. We use McPAT 1.3 [37] to derive the energy of cores, NoC, and memory controllers at 22 nm, CACTI [43] for SRAM banks at 22 nm, CACTI-3DD [10] for stacked DRAM at 45 nm, and Micron DDR3L datasheets [42] for main memory. We report energy consumed to perform a fixed amount of work. This system is implementable in 195 mm^2 with typical power consumption of 60 W in our workloads, consistent with area and power of scaled Silvermont-based systems [28, 32].

7.2 Multi-programmed repeats

Fig. 15 shows results when running 36 copies of each SPEC CPU2006 app on the 36-tile CMP. Fig. 15a shows that Jenga achieves the highest EDP, improving by 2.3× and up to 12× over the S-NUCA baseline (on `omnet`). Jenga improves gmean EDP over JigAlloy by 29% and up to 96% (on `libquantum`). These gains come from being both faster and more efficient:



(a) Energy-delay product and weighted speedup.

(b) Average energy and traffic over all mixes.

Figure 16: Results on 20 mixes of 36 randomly-chosen SPEC CPU2006 apps: (a) EDP and (b) weighted speedup distributions, and (c) average energy and network traffic by component. Results are relative to the S-NUCA L3 baseline.

Fig. 15b shows that Jenga improves performance over S-NUCA by up to 3.6×%/gmean 52%, and over JigAlloy by up to 36%/gmean 9%; and Fig. 15c shows that Jenga improves energy over S-NUCA by up to 70%/gmean 34%, and over JigAlloy by up to 38%/gmean 15%.

Vs. S-NUCA and Alloy, Jenga shows that reconfigurable caches significantly outperform conventional, rigid hierarchies. Vs. Jigsaw, Jenga shows that expanding virtual caches across heterogeneous memories further reduces data movement. Finally, vs. JigAlloy, Jenga shows that a reconfigurable hierarchy can make much better use of heterogeneity and non-uniform access latency than a rigid one.

Fig. 15d gives traffic breakdowns that help explain these benefits. Each bar shows the NoC traffic broken down by source-destination pairs, e.g. traffic from the L2 to stacked DRAM or from SRAM to memory. Jigsaw greatly reduces L2-to-SRAM traffic because it places data in nearby cache banks. Alloy greatly reduces the traffic to memory, because stacked DRAM captures most working sets. JigAlloy combines these benefits, but actually adds traffic on average vs. Jigsaw due to stacked DRAM misses that access both stacked DRAM and main memory. Since stacked DRAM accesses are cheaper than main memory, this is usually a good tradeoff.

However, Jenga reduces traffic further by skipping SRAM or stacked DRAM entirely when they are not beneficial. Since Jenga eliminates hierarchy, it also eliminates SRAM-to-stacked DRAM traffic. L2 accesses instead go directly to stacked DRAM, when it is beneficial (e.g., `libquantum`). Likewise, applications that do not benefit from stacked DRAM simply do not access it (e.g., `astar`). Some other applications (e.g. `gems`) use both SRAM and DRAM as their *single-level* virtual caches and thus benefit more from heterogeneous memories than rigid hierarchy.

Comparing schemes, note that Alloy and JigAlloy sometimes consume more energy than their SRAM-only counterparts (`gems`, `libquantum`, `milc`). In contrast, Jenga saves energy, since it only uses banks when they are beneficial. Prior dynamic bypassing schemes must check all levels for correctness, so they do not provide these benefits (Sec. 4).

7.3 Multi-programmed mixes

Figs. 16a shows the distribution of EDP and weighted speedups over 20 mixes of 36 randomly-chosen memory-intensive SPEC CPU2006 apps. Each line shows the speedup of 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 performance, but do not give a direct comparison across schemes for a particular mix.

Jenga improves EDP on all mixes over the S-NUCA baseline, by up to 3.5×/gmean 2.7×, over Alloy by 2.3×/2.0×%, over Jigsaw by 88%/61%, and *over JigAlloy by 31%/25%*.

Jenga improves EDP because it is both faster and more efficient than prior schemes. Jenga improves weighted speedup over S-NUCA by up to 2.2×/gmean 73%, over Alloy by 55%/42%, over Jigsaw by 46%/31%, and *over JigAlloy by 16%/10%*.

Fig. 16b compares the average energy and network traffic across mixes for each scheme. Jenga reduces energy by 33% over the S-NUCA baseline, by 31% over Alloy, by 15% over Jigsaw, and *by 12% over JigAlloy*.

Jenga reduces network traffic by 63% over S-NUCA, by 69% over Alloy, by 30% over Jigsaw, and by *37% over JigAlloy*. Whereas Alloy relies on speculative parallel accesses to reduce latency, Jenga use stacked DRAM only when it is beneficial. Fig. 16b shows that Alloy reduces energy and traffic to main memory, but these gains are essentially replaced by added energy and traffic to stacked DRAM. In contrast, Jenga intelligently manages the placement of data in stacked DRAM and so only uses stacked DRAM when beneficial.

7.4 Multi-threaded applications

Jenga's benefits carry over to multi-threaded apps, shown in Fig. 17. Jenga improves EDP by up to 4.2×/gmean 65%, over Alloy by 2×/51%, over Jigsaw by 3×/36%, and *over JigAlloy by 60%/27%*. Jenga improves performance over S-NUCA by up to 2.1×/gmean 29%, over Alloy by 35/11%, over Jigsaw by 82/14%, and *over JigAlloy by 35%/6%*. Jenga's energy savings on multi-threaded apps exceed its performance improvements: Jenga reduces energy by 22% over the S-NUCA baseline, by 24% over Alloy, by 12% over Jigsaw, and *by 15% over JigAlloy*.

Multi-threaded applications add an interesting dimension, as data is either thread-private or shared among threads. Jigsaw improves performance by placing private data near threads (e.g., `md`). Alloy helps applications that do not fit in SRAM (e.g., `mgrid`, `smithwa`, `swim`). JigAlloy combines these benefits (e.g., `smithwa`, `swim`), but Jenga performs best by placing data intelligently across SRAM and stacked DRAM.

7.5 Jenga Analysis

Factor analysis: Fig. 18 shows the performance of state-of-the-art rigid hierarchies (JigAlloy) and different Jenga variants on repeats of SPEC CPU2006. Overall, Jenga-SINGLE

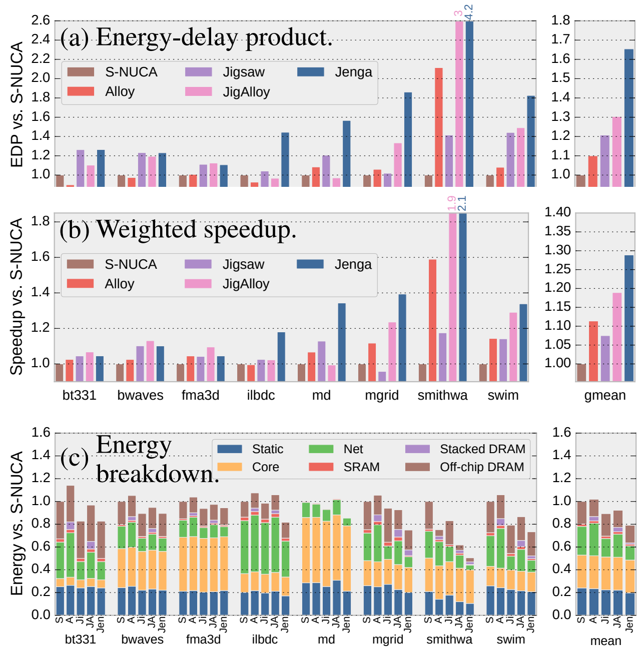


Figure 17: Simulation results for SPEC OMP2012 applications on several rigid hierarchies and Jenga.

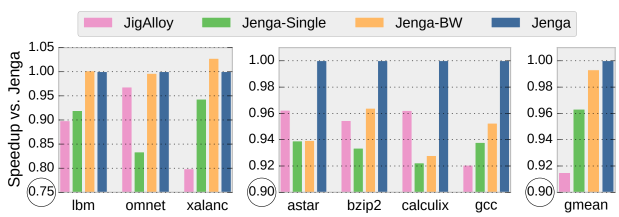


Figure 18: Performance of different Jenga techniques.

suffices for most apps, but bandwidth-aware placement and hierarchy are important to avoid pathologies on the applications shown.

Jenga-BW avoids bandwidth hotspots and significantly outperforms Jenga-SINGLE on memory-intensive apps. Jenga-BW improves performance on `lbm`, `omnet`, and `xalanc`, by 8%, 17%, and 8%, respectively.

Virtual hierarchies further improve results for the few hierarchy-friendly apps: `astar`, `bzip2`, `calculix`, and `gcc` (the system has insufficient capacity for 36 copies of `libquantum` and `milc`). Jenga improves performance by 7%, 4%, 5%, and 8% over Jenga-BW respectively. Without building two-level hierarchies, Jenga-BW would underperform JigAlloy on `astar` and `calculix`.

The gmean improvements are modest because Jenga-SINGLE suffices for most apps: Jenga improves gmean performance over Jenga-SINGLE by 4% and over Jenga-BW by 1%, but Jenga-SINGLE by itself outperforms JigAlloy by 5%.

On mixes, Jenga-BW and Jenga improve gmean EDP by 3% over Jenga-SINGLE. Mixes have very diverse miss curves, so multi-level VHs provide little benefit over a single-level virtual cache: since VL1s are accessed more intensely than VL2s, with diverse access patterns it is almost always the case that some app's VL1 benefits more from capacity than other apps' VL2s, and VL2s are rarely allocated—which speaks to the inefficiency of building a rigid hierarchy for all applications.

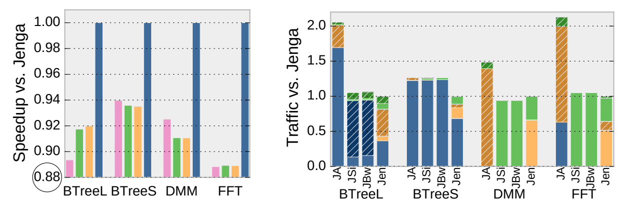


Figure 19: Speedup and traffic breakdown on microbenchmarks. (Legends in Figs. 15 and 18)

Hierarchy-friendly case study: Repeats of SPEC CPU2006 apps are often insensitive to hierarchy, but this is not true of all applications. For example, cache-oblivious algorithms [20] generally benefit from hierarchies because of their inherent recursive structure. To study how Jenga helps such applications, we evaluate three benchmarks: `btree`, whcih 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 [19] to compute the 2D FFT of a 512×512 signal (12 MB footprint).

Fig. 19 shows the performance and traffic breakdown for those benchmarks with different cache architectures. Jenga builds a two-level hierarchy *entirely in SRAM* for `btree-S`, `dmm` and `fft`, improving EDP up to 51%, performance by up to 13%, and energy by up to 25% over JigAlloy. For `btree-L`, Jenga places the second level in stacked DRAM, and improves EDP by 62% and performance by 11% over JigAlloy. These apps benefit from hierarchy: vs. Jenga-SINGLE, Jenga improves EDP by 20%, performance by 10%, and energy by 8%. These results demonstrate the need for reconfigurable *hierarchies* for common applications.

	rate-p	rate-e	rate-edp	mix-p	mix-e	mix-edp
Jenga	**1.55**	**1.51**	**2.32**	**1.74**	**1.50**	**2.71**
Jigsaw + Alloy	1.40	1.27	1.75	1.60	1.35	2.26
Jigsaw [6, 8]	1.16	1.24	1.40	1.32	1.27	1.67
Alloy [47]	1.21	1.02	1.23	1.24	1.05	1.35

Table 2: Improvements in performance, energy, and EDP of various schemes over S-NUCA on a 2.5D DRAM system.

7.6 Other System Architectures

We also evaluate Jenga under different a DRAM cache architecture to show that Jenga is effective across different packaging technologies. Specifically, we model a "2.5D", interposer-based DRAM architecture with 4 vaults located at chip edges, totaling 2 GB and 200 GBps of bandwidth, similar to AMD Fiji [2] and NVIDIA Pascal [27] packaging. Table 2 shows the gmean improvement of different schemes over S-NUCA in performance, energy, and EDP under the interposer-based system. Jenga has similar improvements in energy, performance, and EDP as it does in a 3D stacked system.

8. CONCLUSION

Forthcoming memory technologies provide abundant capacity at high bandwidth. However, unless they are managed effectively, their added latency and energy will limit system performance. Rigid hierarchies have inherent flaws in this regard. We have presented Jenga, a system that builds application-specific virtual cache hierarchies out of heterogeneous, distributed cache banks with different packaging techniques. Jenga uses heterogeneous caches when they are beneficial, and avoids their overheads when they are not. As a result, Jenga significantly improves performance and efficiency over the state of the art.

Acknowledgments

This work was supported in part by NSF grants CCF-1318384 and CAREER-1452994, and by a Samsung GRO grant.

9. 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 *HPCA-21*, 2015.

[2] AMD, "AMD Radeon R9 Fury X Graphics Card," 2015.

[3] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *HPCA-15*, 2009.

[4] B. Beckmann, M. Marty, and D. Wood, "ASR: Adaptive selective replication for CMP caches," in *MICRO-39*, 2006.

[5] B. Beckmann and D. Wood, "Managing wire delay in large chip-multiprocessor caches," in *ASPLOS-XI*, 2004.

[6] N. Beckmann and D. Sanchez, "Jigsaw: Scalable Software-Defined Caches," in *PACT-22*, 2013.

[7] N. Beckmann and D. Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance," in *HPCA-21*, 2015.

[8] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling," in *HPCA-21*, 2015.

[9] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *ISCA-33*, 2006.

[10] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *DATE*, 2012.

[11] Z. Chishti, M. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in CMPs," in *ISCA-32*, 2005.

[12] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *MICRO-39*, 2006.

[13] C. Chou, A. Jaleel, and M. K. Qureshi, "BEAR: techniques for mitigating bandwidth bloat in gigascale DRAM caches," in *ISCA-42*, 2015.

[14] W. J. Dally, "GPU Computing: To Exascale and Beyond," in *SC10*, 2010.

[15] A. Das, M. Schuchhardt, N. Hardavellas, G. Memik, and A. Choudhary, "Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores," in *DATE*, 2012.

[16] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *SC10*, 2010.

[17] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *HPCA-13*, 2007.

[18] S. Franey and M. Lipasti, "Tag Tables," in *HPCA-21*, 2015.

[19] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999.

[21] D. Gross, *Fundamentals of queueing theory*. John Wiley & Sons, 2008.

[22] 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.

[23] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA-36*, 2009.

[24] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi, "Database servers on chip multiprocessors: Limitations and opportunities," in *Proc. CIDR*, 2007.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach (5th ed.)*. Morgan Kaufmann, 2011.

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

[27] J.-H. Huang, "Leaps in Visual Computing," in *GTC*, 2015.

[28] Intel, "Knights Landing: Next Generation Intel Xeon Phi," in *SC13*, 2013.

[29] J. Jaehyuk Huh, C. Changkyu Kim, H. Shafi, L. Lixin Zhang, D. Burger, and S. Keckler, "A NUCA substrate for flexible CMP cache sharing," *IEEE Trans. Par. Dist. Sys.*, vol. 18, no. 8, 2007.

[30] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA-40*, 2013.

[31] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian, "CHOP: Adaptive filter-based dram caching for CMP server platforms," in *HPCA-16*, 2010.

[32] D. Kanter, "Silvermont, Intel's Low Power Architecture," 2013.

[33] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, 2011.

[34] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *MICRO-43*, 2010.

[35] C. Kim, D. Burger, and S. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS-X*, 2002.

[36] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *HPCA-17*, 2011.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, in *MICRO-42*, 2009.

[38] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *ISCA-35*, 2008.

[39] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *MICRO-44*, 2011.

[40] N. Madan, L. Zhao, N. Muralimanohar, A. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell, "Optimizing communication and capacity in a 3D stacked reconfigurable cache hierarchy," in *HPCA-15*, 2009.

[41] J. Merino, V. Puente, and J. Gregorio, "ESP-NUCA: A low-cost adaptive non-uniform cache architecture," in *HPCA-16*, 2010.

[42] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.

[43] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO-40*, 2007.

[44] F. Pollaczek, "Über eine aufgabe der wahrscheinlichkeitstheorie." *Mathematische Zeitschrift*, vol. 32, no. 1, 1930.

[45] M. Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," in *HPCA-15*, 2009.

[46] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO-39*, 2006.

[47] M. Qureshi and G. Loh, "Fundamental latency trade-offs in architecting DRAM caches," in *MICRO-45*, 2012.

[48] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *MICRO-43*, 2010.

[49] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *ISCA-38*, 2011.

[50] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *ISCA-40*, 2013.

[51] J. Sim, J. Lee, M. Qureshi, and H. Kim, "FLEXclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion," in *ISCA-39*, 2012.

[52] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *ASPLOS-IX*, 2000.

[53] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. Lee, "An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," in *HPCA-16*, 2010.

[54] L. Yavits, A. Morad, and R. Ginosar, "Cache hierarchy optimization," *Computer Architecture Letters*, vol. 13, no. 2, 2014.

[55] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA-32*, 2005.