Design and Analysis of Spatially-Partitioned Shared Caches

by

Nathan Beckmann

B.S., Computer Science B.S., Mathematics
University of California, Los Angeles (2008)

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 2015

Author.....
Department of Electrical Engineering and Computer Science
August 28, 2015

Certified by.....
Daniel Sanchez
Assistant Professor
Thesis Supervisor

Accepted by.....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Abstract

Data movement is a growing problem in modern chip-multiprocessors (CMPs). Processors spend the majority of their time, energy, and area moving data, not processing it. For example, a single main memory access takes hundreds of cycles and costs the energy of a thousand floating-point operations. Data movement consumes more than half the energy in current processors, and CMPs devote more than half their area to on-chip caches. Moreover, these costs are increasing as CMPs scale to larger core counts.

Processors rely on the on-chip caches to limit data movement, but CMP cache design is challenging. For efficiency reasons, most cache capacity is shared among cores and distributed in banks throughout the chip. Distribution makes cores sensitive to *data placement*, since some cache banks can be accessed at lower latency and lower energy than others. Yet because applications require sufficient capacity to fit their working sets, it is not enough to just use the closest cache banks. Meanwhile, cores compete for scarce capacity, and the resulting interference, left unchecked, produces many unnecessary cache misses.

This thesis presents novel architectural techniques that navigate these complex tradeoffs and reduce data movement. First, *virtual caches* spatially partition the shared cache banks to fit applications' working sets near where they are used. Virtual caches expose the distributed banks to software, and let the operating system schedule threads and their working sets to minimize data movement. Second, analytical replacement policies make better use of scarce cache capacity, reducing expensive main memory accesses: *Talus* eliminates performance cliffs by guaranteeing convex performance, and *EVA* uses planning theory to derive the optimal replacement metric under uncertainty. These policies improve performance and make qualitative contributions: Talus is cheap to predict, and so lets cache partitioning techniques (including virtual caches) work with high-performance cache replacement; and EVA shows that the conventional approach to practical cache replacement is sub-optimal.

Designing CMP caches is difficult because architects face many options with many interacting factors. Unlike most prior caching work that employs best-effort heuristics, we reason about the tradeoffs through analytical models. This analytical approach lets us achieve the performance and efficiency of application-specific designs across a broad range of applications, while further providing a coherent theoretical framework to reason about data movement. Compared to a 64-core CMP with a conventional cache design, these techniques improve end-to-end performance by up to 76% and an average of 46%, save 36% of system energy, and reduce cache area by 10%, while adding small area, energy, and runtime overheads.

Thesis Supervisor: Daniel Sanchez
Title: Assistant Professor

Contents

Acknowledgments

First and foremost, I owe thanks to my research advisor, Professor Daniel Sanchez. I switched research areas to work with Daniel rather late in my graduate career. Daniel quickly brought me up to speed and guided me towards important and challenging problems. Daniel has shown me how to leverage my mathematical background while effectively communicating the research to the architecture community. As a result, I have been able to pursue research that I find personally exciting, and the last few years have been easily the most productive of my graduate study. On a personal level, Daniel is also a good friend.

Prior to working with Daniel, I did my masters and early doctoral study advised by Professor Anant Agarwal, and continued this work advised by Professors Frans Kaashoek and Nickolai Zeldovich. I am deeply grateful to each of my advisors for their support, and I have learnt much from their different approaches to research. Anant showed me how to "think big" about research problems, and Frans and Nickolai taught me to maintain healthy skepticism and deep, technical rigor.

This thesis covers the research I have done since September 2012 while working with Prof. Daniel Sanchez [12–15]. Not included is my earlier work with Profs. Agarwal, Kaashoek, and Zeldovich [16, 90, 108, 121, 152–154, 164]. I am grateful to my many co-authors: Harshad Kasture, Po-An Tsai, David Wentzlaff, Charles Gruenwald III, George Kurian, Jason Miller, Hank Hoffmann, Lamia Youseff, Adam Belay, Christopher Johnson, Filippo Sironi, Davide Bartolini, Christopher Celio, Kevin Modzelewski, and James Psota. It has been a pleasure sharing ideas, working together, and socializing over the past seven years.

Professor Randall Davis has been my academic advisor from when I first arrived at MIT, helping me select my first classes in September 2008, through several changes in research supervision, to eventually completing this thesis. Prof. Davis always made sure I was on the path to complete a successful thesis, and has been an invaluable source of support.

I was able to take classes from some of the best minds in Computer Science and Economics as part of my degree. I thank EECS Profs. Arvind, Joel Emer, Michael Sipser, Robert Berwick, Charles Leiserson, Silvio Micali and Economics Profs. Jonathan Gruber, Christopher Knittel, and Anne McCants.

I would also like to thank the administrative staff, both of my advisors (Mary MacDavitt, Cree Bruins, and Sally Lee) and in central administration (Janet Fischer and Alicia Duarte) who helped me in innumerable small ways to complete my PhD.

Last but not least, I thank my family and friends who helped me focus when I needed to, and helped me relax when I didn't. In particular, I am in debt to my partner and best friend, Deirdre Connolly. Dee has tolerated years of graduate stipends, long deadlines, and uncertainty so that I could finish my degree. I love you! And of course, I cannot forget my sidekick, Arya the Fluffy Corgi, who has been my research assistant for the past three years. Woof!

Introduction 1

THIS thesis tackles the growing cost of data movement in modern chip-multiprocessors (CMPs). Sequential processor performance peaked in the early 2000s, forcing computer architects to turn to parallelism to sustain performance growth [3]. This shift was primarily motivated by power concerns, as the end of Dennard scaling meant it was no longer practical to increase core frequency with each technology generation [24, 39, 46]. Although core frequency and complexity have since plateaued, the number of cores in CMPs has increased exponentially following Moore's law. The concomitant increase in memory references from these additional cores places mounting pressure on the memory system.

Yet current memory system designs do not scale their performance to match demand. Data movement consumes more than 50% of chip energy in conventional designs, caches take more than 50% of chip area, and still many applications spend most of their time waiting for data. The performance gap arises for two main reasons. Most significantly, *off-chip bandwidth* is limited by power constraints, so it has increased much more slowly than core count. Main memory references are thus increasingly expensive and frequently limit processor performance. To cope with this growing problem, future CMPs must satisfy more memory requests using on-chip caches or memories. Unfortunately, *on-chip cache accesses* are also becoming more expensive. CMP caches are split into banks that are distributed across the chip and connected by an on-chip network (see Figure 1.1). The network diameter increases with additional cores, and hence the average network distance between caches is increasing with larger core counts. We find that on-chip network latency often limits performance and consumes significant energy, even when using state-of-the-art cache designs.

To achieve good CMP performance scaling, future memory systems must address both of these problems. Indeed, without major architectural innovations, data movement may soon consume so much energy that it limits performance for power reasons alone—in current processors, moving 256 B across the on-chip network consumes 15× as much energy as a floating-point multiply-add, and a single DRAM access consumes 1000× as much [33, 78]. Prior work has focused on computational efficiency for general-purpose processors and domain-specific accelerators. But the memory system dominates power for many workloads, and hence data movement must be reduced to limit power consumption as well.

To summarize, raw computing power has never been cheaper, but it is increasingly difficult to supply cores with the data they demand. A large volume of work in new technologies and architectures focuses on reducing data movement: Emerging technologies like high-bandwidth off-chip links [86] and 3D stacking [21] provide a large increase in off-chip bandwidth. Commercial processors devote an increasing fraction of chip area to caches. And architectural innovations have reduced off-chip memory accesses [44, 69, 79, 124, 125, 130, 141, 151, 159] and on-chip latency [9, 10, 27, 29, 30, 35, 53, 64, 106, 123, 165]. Yet the design of future memory systems remains an open question.

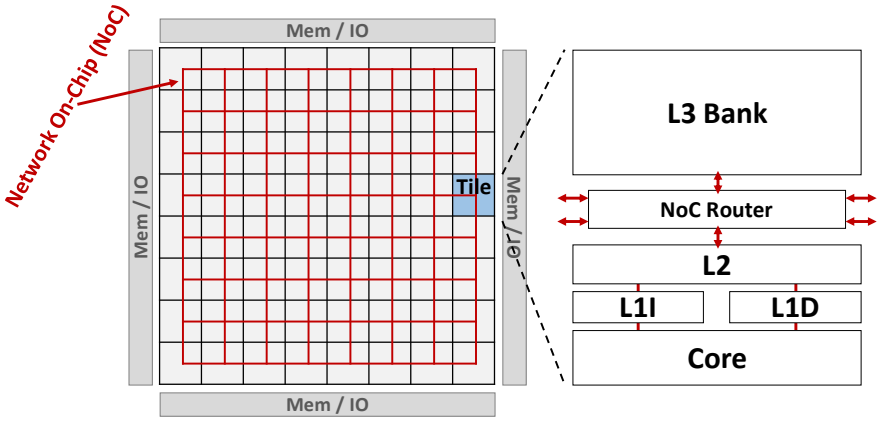


Figure 1.1: An example 64-core CMP. Cores are distributed throughout the processor and connected by an on-chip network. Each core has a private cache hierarchy (L1s and L2), and cores share the L3 cache. The L3 cache is split into banks that are distributed throughout the chip.

1.1 Challenges

On-chip memories keep data close to cores and are thus critical to minimize data movement. Ideally, programs or compilers would manage the on-chip memories (i.e., as a scratchpad), since in theory they have the most information about how memory is used. But in practice scratchpads are too cumbersome for programmers and too difficult for compilers to use effectively [92], particularly in shared systems.

The central challenge is that programs are themselves often unpredictable, and it is better in most cases to manage on-chip resources as a cache. Doing so shifts the responsibility of managing on-chip memories from the programmer to the memory system, which is more convenient for the programmer but loses application-level information. Thus there is a need for robust, general-purpose techniques that reduce data movement, without foreknowledge of how the application accesses memory.

Most caching techniques use heuristics drawn from observations of common application behaviors. However, applications are diverse, and optimizing for the common case leaves significant performance on the table. Adaptive heuristics recover some of this performance by choosing among a few fixed policies, but there is an opportunity for substantial further improvement.

Organizing shared cache banks: To capture frequently accessed data, the cache must be large enough to hold the application's working set, but ideally no larger, since large caches take longer to access. To navigate this tradeoff, the on-chip cache is traditionally organized as a hierarchy of increasingly larger and slower caches. Hierarchy allows a simple, fixed design to benefit a wide range of applications, because working sets settle at the smallest (and hence fastest and most efficient) level where they fit. However, rigid hierarchies also cause significant overheads because each level adds latency and energy even when it does not capture the working set.

The tradeoff between cache size and latency/energy is particularly relevant to multicore CMPs (see Figure 1.1). Each core has its own private cache levels (the L1s and L2), but the bulk of cache capacity is shared among cores (the L3) so that applications with large working sets have sufficient capacity available to them. The shared cache typically takes upwards of 50% of chip area [89] and adopts a non-uniform cache access (NUCA [85]) design. NUCAs divide capacity into banks and then distribute banks across the chip. Each bank is a self-contained cache that contains a small fraction of the overall shared capacity. This distribution exposes a fine-grain tradeoff between cache size and access latency/energy, since for any given core some banks are further away than others.

State-of-the-art NUCA designs use simple, hierarchical heuristics. In particular, they treat the on-chip cache banks as separate levels of a cache hierarchy. An application may thus require multiple cache lookups to reach the bank where its working set fits. Because the latency and energy difference across banks is small, every lookup is similarly expensive and hierarchy imposes significant costs. Future CMPs cannot afford to have multiple bank lookups to access data, but must exploit the distribution of cache banks to keep data near where it used—*how should cache designs harness this growing heterogeneity?*

Improving performance within a single cache level: With sufficient associativity, a cache's performance is determined by its replacement policy. The optimal replacement policy, Belady's MIN [17, 104], is impractical because it requires knowledge of the future. Practical replacement policies must cope with *uncertainty*, never knowing precisely when candidates will be referenced. The challenge is uncertainty itself, since with complete information the optimal policy is simple: evict the candidate that is referenced furthest in the future. The most common approach to replacement under uncertainty, which was already well-established in the 1970s [4], is to predict when candidates will be referenced and evict the candidate that is *predicted* to be referenced furthest in the future.

In practice, policies make predictions using a number of different heuristics. Traditional policies like least-recently used (LRU) perform poorly at the last-level cache (i.e., the L3 in Figure 1.1) [80, 83, 125]. They also exhibit performance cliffs, where performance suddenly improves beyond a threshold cache size, that waste cache space and make it difficult to manage shared caches.

Much recent work has explored techniques to improve upon LRU through various empirical, best-effort heuristics [44, 69, 73, 79, 125, 141, 159]. These policies tend to perform similarly on average, but no policy dominates across different applications. Indeed, these policies give no guarantees of good performance and often perform poorly on particular access patterns. Moreover, there is substantial room for improvement, since these polices tend to close only around 40% of the gap between random and optimal replacement. Clearly, best-effort heuristics are not making the best use of available information. But to substantially improve performance, we require new insights—*what is the right approach to cache replacement under uncertainty?*

Contributions 1.2

This thesis presents novel architectural techniques to reduce data movement. Specifically, we make contributions in two areas:

- *Jigsaw* spatially partitions the distributed cache banks into *virtual caches*, placing each application's working set in nearby cache banks. Virtual caches are a single, coherent approach to harness the increasing heterogeneity of on-chip memories.

- Analytical replacement policies make better use of scarce cache capacity and provide qualitative benefits over best-effort heuristics. *(i) Talus* guarantees convex cache performance, letting well-studied policies like LRU perform competitively with state-of-the-art heuristics without sacrificing their attractive features. *(ii) EVA* is derived from first principles and gives a useful analytical framework for cache replacement under uncertainty. We show that predicting when candidates will be referenced is sub-optimal, and that EVA is in fact the right metric. EVA is practical to implement and substantially outperforms state-of-the-art heuristics.

Our goal is to match or exceed the performance of application-specific designs while performing well across many applications. To achieve this, we take an analytical approach: These techniques use simple but highly configurable hardware mechanisms, lightweight hardware monitors that profile applications,

and analytical models that periodically reconfigure the mechanisms to adapt to running programs. We find this design minimizes data movement at low overhead. In total, this thesis constitutes a comprehensive and robust foundation to scale memory system performance in future CMPs.

Jigsaw: Non-uniform cache architectures (NUCAs) present an opportunity to customize the cache hierarchy to the application. We propose that heterogeneity should be exposed to software, allowing an operating system-level runtime to organize cache banks into *virtual caches* customized to applications. Each virtual cache is sized to fit an application's working set and placed in nearby cache banks to minimize access latency.

Jigsaw (Chapter 3) implements virtual caches via *(i)* a lightweight software runtime that spatially partitions cache banks into virtual caches, and *(ii)* modest hardware support that provides a configurable layer of indirection between an address and its on-chip location. In software, finding the best virtual cache configuration is NP-hard, and Jigsaw's runtime must complete in at most a few milliseconds. Jigsaw uses novel partitioning algorithms and a simple latency model to perform within 1% of idealized, impractical solutions. In hardware, a small structure called the virtual cache translation buffer (VTB) maps addresses to banks. The VTB stores a small number of bank ids (e.g., 128 ids), and maps each cache line to a single bank by hashing its address to select a bank id. Unlike prior NUCAs, in Jigsaw addresses do not move when accessed, so Jigsaw requires only a *single cache lookup* to access data. Our results show that virtual cache hierarchies improve performance by up to 76% and geometric mean of 46% over commercial designs, save 36% of system energy, and add less than 3% area overhead to the last-level cache.

Virtual caches are an *application-centric* rather than *system-centric* approach to CMP caching. Virtual caches are configured based on how applications accesses memory: e.g., applications with smaller working sets get smaller virtual caches, regardless of the number of banks in the system. This has the important implication that *virtual cache performance and energy scale independently of system size*. Virtual caches thus provide asymptotically better memory system performance scaling than prior NUCA schemes.

Cache replacement policies: To improve cache replacement, our critical insight is that *the access stream at the last-level cache is effectively randomized* because lower cache levels strip out the short-term temporal locality. The last-level cache, which dominates cache capacity, can therefore be accurately modeled as a random process. This simple idea opens the door to several techniques.

First, we build Talus, a partitioning scheme that guarantees *convex cache performance* (Chapter 4). Convexity means that cache space yields diminishing returns in hit rate, eliminating performance cliffs and improving performance. Convexity thus improves performance and simplifies cache partitioning because simple algorithms (e.g., hill climbing) yield globally optimal configurations.

In addition to convexity, Talus's main contribution is letting cache partitioning achieve good replacement performance within each partition. This is important because recent replacement policies, which speed up some applications by more than 20%, are incompatible with cache partitioning: To allocate cache capacity intelligently, partitioning techniques like Jigsaw require predictions of miss rate at different partition sizes. This requirement has confined partitioning to well-studied policies like LRU that are cheap to predict. Talus transforms LRU into a policy competitive with recent empirical heuristics, resolving the unintended conflict between partitioning and high-performance replacement. Talus thus merges two independent lines of cache research. It is the first scheme to do so.

Talus works by randomly sampling accesses between two *shadow partitions* that are invisible to software. We present an intuitive model for how sampling scales cache performance, and use this result to derive the sampling rate and shadow partition sizes that yield convex performance. We prove that Talus works on arbitrary replacement policies and partitioning schemes, and use this framework to

further prove that *(i)* optimal replacement is convex and *(ii)* bypassing, a common replacement heuristic, is sub-optimal.

Second, we study optimal replacement under uncertainty, exploring the limits of practical cache replacement. Specifically, we model that reuse distances[1] are drawn independently from a small set of probability distributions. This memory reference model is simple enough to analyze and accurate enough to yield useful insights. In particular, it is easy to show within this model that the standard approach to cache replacement under uncertainty (i.e., predicting when candidates will be referenced) is sub-optimal. In appendices, we validate this model by using it to accurately predict cache behavior across many workloads, replacement policies, and cache sizes.

Using planning theory, we show that the optimal replacement policy is to replace candidates based on their *economic value added* (EVA, Chapter 5), or how many more hits each candidate yields over the average candidate. This intuitive metric reconciles the two main tradeoffs in replacement: the probability that a candidate will hit (its benefit) and the cache space it will consume (its cost). We show that EVA follows from a Markov decision process (MDP [122]) model of cache replacement. EVA is the first rigorous study of cache replacement under uncertainty since the 1970s [4], and gives a general principle of optimal cache replacement under uncertainty that should prove useful in designing future policies.

We implement EVA using a small structure called the *eviction priority array*. Candidates' ages are used to look up their eviction priority during replacement, and the candidate with the highest priority is evicted. We have implemented a small circuit to compute eviction priorities and synthesized it in a 65 nm commercial manufacturing process. EVA adds roughly 1% area and energy overhead over the last-level cache. We find that EVA widely outperforms prior replacement policies, closing 57% of the gap between random and optimal replacement on average vs. 41-47At equal performance, EVA saves 10% cache area over the best alternative policy.

Two common themes emerge from these techniques. The first and most important theme is the use of analytical models to predict performance and find a configuration—e.g., virtual cache placement or replacement policy—that yields the best performance. The second theme is how these models are realized in implementation. We develop simple but highly configurable hardware mechanisms (viz., the VTB and eviction priority array), and then infrequently configure these mechanisms to implement the desired policy. We demonstrate that our approach yields both significant quantitative and qualitative benefits over empirical heuristics. These themes therefore serve as a blueprint for designing robust, high-performance memory systems.

[1] A reference's *reuse distance* is the number of accesses between references to the same address. Reuse distances are an intuitive starting point because they correspond to candidates' time until reference, which is the basis for optimal replacement.

Background 2

This chapter presents the relevant prior work on CMP memory systems that we build and improve on. This work falls into several categories: cache architecture, cache replacement policies, cache partitioning, cache modeling, and non-uniform cache architectures.

Modern cache architecture 2.1

Modern multicores feature multilevel cache hierarchies (refer back to Figure 1.1, pg. 12). Each core has one or two levels of small, fast, private caches (L1 and L2). Private caches are backed by a much larger last-level cache (LLC, L3 in Figure 1.1) that contains the bulk of cache capacity, and often consumes more than half of chip area [89]. The LLC is shared among cores so that applications with large working sets have sufficient capacity available to them.

Cache architecture varies greatly across levels. The purpose of the private levels is to cache the program's most frequently accessed data at low latency and high bandwidth, so they are kept small and simple—and therefore fast. Specifically, both the L1 and L2 adopt a familiar, set-associative architecture and use simple replacement policies like random replacement, least-recently used (LRU), not-recently-used (NRU), or approximations thereof (e.g., pseudo-LRU).

The cache architecture is more sophisticated at the LLC. Since off-chip misses are expensive, various caching techniques are attractive for the LLC that are a poor design for the private levels. In particular, the LLC adopts a non-uniform cache access (NUCA) design, distributing its capacity in *banks* throughout the chip and connected by the on-chip network (NoC). Each bank is a self-contained cache containing a small fraction of shared capacity. Splitting the LLC into banks provides sufficient bandwidth to all cores at reasonable overheads; a monolithic cache with sufficient read and write ports would be prohibitively expensive.

While distributed banks are more efficient, they introduce *data placement* as a major design concern. Since each individual bank is fairly small, its access latency is typically not much worse than the L2 caches (say, 9 cycles vs. 6 cycles), and their energy is similar as well. Most of the latency and energy of an LLC access is spent in the on-chip network, not accessing the bank. For example, with 3 cycles per network hop, the round-trip time across an 8×8 mesh is 84 cycles. In other words, when accessing a distant cache bank, more than 90% of the latency can come from the on-chip network. Commercial cache designs often ignore data placement and simply hash addresses across cache banks, but clearly performance can be improved by adapting data placement to the access pattern.

The architecture of individual LLC banks is also quite different from the private cache levels. The most significant change is the replacement policy, discussed in detail below. In addition, the LLC adopts an array organization that ensures high effective associativity. For instance, by hashing addresses before indexing the array, the cache pseudo-randomly spreads accesses across sets and reduces conflicts [84, 89, 140, 156]. Other designs further increase effective associativity without adding ways. For instance, skew-associative

caches [133] use a different hash function for each way, mixing replacement candidates across sets; and zcaches [129] use cuckoo hashing to select many more replacement candidates than physical ways.

From the perspective of an analytical memory system design, there are two important implications. First, private caches capture most "hot" data, so the LLC's access stream is stripped of short-term temporal correlations: a second access to the same line requires that the line be first evicted from the L1 and L2. Second, since modern LLCs use hashing and achieve high effective associativity, replacement candidates form a representative sample of cached lines [14, 124, 129]. Details of array organization, which have been the focus of prior analytical models [1, 41, 104, 132, 160], are relatively unimportant in modern LLCs. We leverage this insight by modeling replacement as a probabilistic process affecting individual lines.

2.2 Replacement policies

The optimal replacement policy, Belady's MIN [17, 104], relies on a perfect oracle to replace the line that will be reused furthest in the future. Since an oracle is not generally available, practical policies must cope with uncertainty, never knowing precisely when or if a candidate will be referenced.

Broadly speaking, prior work has taken two approaches to practical cache replacement. On the one hand, architects can develop a probabilistic model of how programs reference memory and then solve for the optimal replacement policy within this reference model. On the other hand, architects can observe programs' behaviors and find best-effort heuristics that perform well on common access patterns. These two approaches are complementary: analytical work yields insight into how to approach replacement, while empirical work tunes the policy so that it performs well on real applications at low overhead. But the vast majority of research has been on the empirical side; analytical work dates from the early 1970s [4].

Caches traditionally use simple heuristics like least-recently used (LRU) or least-frequently used (LFU) to choose a victim. These policies are well studied and have some attractive features. For instance, LRU is provably within a constant factor of optimal replacement [135], and it obeys the *stack property* [104]—i.e., on the same reference stream, a smaller LRU cache will hold a strict subset of a larger LRU cache—which makes it cheap to monitor [124].

Unfortunately, LRU performs poorly at the last-level cache (LLC) because the private caches filter most temporal locality [14, 80, 83], and without temporal locality LRU causes thrashing and performance cliffs. For example, iterating sequentially over a 1 MB array gets zero hits with less than 1 MB of cache, but every access suddenly hits at 1 MB. Recent years have seen a resurgence of cache replacement research, which has been quickly adopted in commercial multicores [65, 156]. These policies propose many mechanisms and heuristics that attempt to emulate optimal replacement.

2.2.1 Replacement theory

In the 1970s, Aho et al. [4] studied page replacement within the *independent reference model* (IRM), which assumes that pages are accessed non-uniformly with known probabilities. They model cache replacement as a Markov decision process, and show that the optimal policy, A_0, is to evict the page with the lowest reference probability. In the context of web caching, Bahat et al. [7] extend IRM to support non-uniform miss costs (e.g., due to network congestion). They show that with non-uniform costs, the IRM still yields a simple, optimal policy.

Aho et al. is the work most closely related to EVA (Chapter 5), but we observe that the independent reference model is a poor fit for processor caches. Capturing dynamic behavior is crucial for good performance, but the IRM does not. We thus first formalize a reference model that tractably captures

dynamic behavior. Then, like Aho et al., we use Markov decision processes to find the optimal replacement policy.

Empirical replacement policies 2.2.2

Many policies follow MIN's example and try to predict candidates' time until reference. This longstanding approach was described by Aho et al. [4] as *the informal principle of optimality*: "the candidate to be replaced is that which has the longest expected time until next reference", and they showed it was optimal in a simple memory reference model discussed below. We observe that, details aside, empirical policies use three broad techniques:

- **Recency:** Recency prioritizes recently used lines over old ones. LRU uses recency alone, which has obvious pathologies (e.g., thrashing and scanning [40, 125]). Most high-performance policies combine recency with other techniques.
- **Classification:** Some policies divide lines into separate classes, and treat lines of each class differently. For example, several policies classify lines as reused or non-reused [69, 83]. Classification works well when classes have markedly different access patterns.
- **Protection:** When the working set does not fit in the cache, some policies choose to protect a portion of the working set against eviction from other lines to avoid thrashing. Protection is equivalent to thrash-resistance [69, 125].

Empirical policies implement these techniques in different ways. DIP [125] enhances LRU by dynamically detecting thrashing using set dueling, and protects lines in the cache by inserting most lines at low priority in the LRU chain. DIP inserts a fixed fraction of lines ($\epsilon = 1/32$) at high priority to avoid stale lines. DRRIP [69] classifies between reused and non-reused lines by inserting lines at medium priority, includes recency by promoting lines on reuse, and protects against thrashing with the same mechanism as DIP. SHiP [159] extends DRRIP with more elaborate classification, based on the memory address, PC, or instruction sequence. PDP [44] decides how long to protect lines based on the reuse distance distribution, but does not do classification. IbRDP [79] uses PC-based classification, but does not do protection. Finally, IRGD [141] adapts its policy to the access stream, ranking lines according to their harmonic expected reuse distance. These policies are starting to ship in commercial processors, with recent Intel processors using DRRIP in the LLC [65, 156].

Empirical policies improve over LRU on average, but have some drawbacks. First, these policies use empirically tuned heuristics that may not match application behavior, so they sometimes perform worse than traditional alternatives like LRU. Moreover, there is no common design principle in these policies, and it is unclear which if any take the right approach. Second, the miss curve for these policies cannot be easily estimated in general, which makes them hard to use with partitioning, as we discuss below.

Trends: Beyond their particular techniques, we observe two relevant trends in recent research. First, most empirical policies exploit dynamic behavior by using the candidate's age (the time since it was last referenced) to select a victim. That is, lines enter the cache at a given priority, which changes over time according to how long the line has been in the cache. For example, LRU prefers recently used lines to capture temporal locality; RRIP [69, 159] predicts a longer re-reference interval for older candidates; PDP [44] protects candidates from eviction for a fixed number of accesses; and IRGD [141] uses a heuristic function of ages.

Second, recent empirical policies adapt themselves to the access stream to varying degrees. DIP [125] detects thrashing with set dueling, and avoids thrashing by inserting most lines at low priority. DRRIP [69]

inserts lines at medium priority, preferring lines that have been reused, and avoids thrashing using the same mechanism as DIP. SHiP [159] extends DRRIP by adapting the insertion priority based on the memory address, PC, or instruction sequence. PDP [44] and IRGD [141] use auxiliary monitors to profile the access pattern and periodically recompute their policy.

These two trends show that *(i)* on real access streams, aging reveals information relevant to replacement; and *(ii)* adapting the policy to the access stream improves performance. But these policies do not make the best use of the information they capture, and prior theory does not suggest the right policy. Indeed, we show that with dynamic behavior, *predicting time until reference is the wrong approach*.

In Chapter 4, we use partitioning to improve the performance of prior policies like LRU, achieving competitive performance to empirical policies without sacrificing LRU's attractive properties.

In Chapter 5, we use planning theory to design a practical policy, EVA, that maximizes the cache's hit rate given imperfect information about candidates. EVA is intuitive and inexpensive to implement. In contrast to most empirical policies, it does not explicitly encode particular heuristics (e.g., preferring recently-used lines). Rather, it is a general approach that aims to make the best use of the limited information available, so that prior heuristics arise naturally when appropriate.

2.3 Cache partitioning

Cache partitioning allows software to divide cache space among cores, threads, or types of data, enabling system-wide management of shared caches. For example, partitioning improves performance by favoring applications that cache well, and can provide quality-of-service by isolating applications' working sets. There are several ways to implement partitioning schemes. Cache partitioning requires a *partitioning policy* to select partition sizes, and a *partitioning scheme* to enforce them.

2.3.1 Partitioning schemes

A partitioning scheme should support a large number of partitions with fine-grained sizes, disallow interference among partitions, strictly enforce partition sizes, avoid hurting cache associativity or replacement policy performance, support changing partition sizes efficiently, and require small overheads. Achieving these properties is not trivial.

Several techniques rely on restricting the locations where a line can reside depending on its partition. Way partitioning [28] restricts insertions from each partition to its assigned subset of ways. It is simple, but it supports a limited number of coarsely-sized partitions (in multiples of way size), and partition associativity is proportional to its way count, sacrificing performance for isolation. To avoid losing associativity, some schemes can partition the cache by sets instead of ways [127, 149], but they require significant changes to cache arrays. Alternatively, virtual memory and page coloring can be used to constrain the pages of a process to specific sets [95, 143]. While software-only, these schemes are incompatible with superpages and caches indexed using hashing (common in modern CMPs), and repartitioning requires costly recoloring (copying) of physical pages.

Caches can also be partitioned by modifying the allocation or replacement policies. These schemes avoid the problems with restricted line placement, but most rely on heuristics [101, 158, 161], which provide no guarantees and often require many more ways than partitions to work well. In contrast, Vantage [130] and Futility Scaling [151] leverage the statistical properties of skew-associative caches [133] and zcaches [129] to implement partitioning efficiently. These schemes support hundreds of partitions, provide strict guarantees on partition sizes and isolation, can resize partitions without moves or invalidations, and are cheap to implement (requiring $\approx$1% extra state and negligible logic).

Our techniques are agnostic to partitioning scheme, but we often employ Vantage to partition cache banks because it supports many partition sizes with strict isolation and can reconfigure partition sizes cheaply.

Partitioning policies 2.3.2

Partitioning policies consist of a monitoring mechanism, typically in hardware, that profiles partitions, and a controller, in software or hardware, that uses this information to set partition sizes to maximize some metric, such as throughput [124], fairness [95, 110, 138], security [115], or QoS [77, 93].

Utility-based cache partitioning (UCP) is a frequently used policy [124]. UCP introduces a *utility monitor* (UMON) per core, which samples the address stream and measures the partition's *miss curve*, i.e. the number of misses that the partition would have incurred with each possible number of allocated ways. System software periodically reads these miss curves and repartitions the cache to maximize cache utility (i.e., the expected number of cache hits). Miss curves are often not convex, so deriving the optimal partitioning is NP-hard. UCP decides partition sizes with the Lookahead algorithm, an $O(N^2)$ heuristic that works well in practice, but is too slow beyond small problem sizes. Although UCP was designed to work with way partitioning, it can be used with other schemes [130, 161]. Instead of capturing miss curves, some propose to estimate them with analytical models [138], use simplified algorithms, such as hill-climbing, that do not require miss curves [101], or capture them offline [20], which simplifies monitoring but precludes adaptation. Prior work has also proposed approximating miss curves by their convex fits and using efficient convex optimization instead of Lookahead [20].

In our work, we observed that miss curves are often non-convex, so hill-climbing or convex approximations are insufficient. However, UCP's Lookahead is too slow to handle large numbers of fine-grained partitions. To solve this problem, we reformulate Lookahead in a much more efficient way, making it linear-time (Chapter 3). We also develop a general technique that makes cache performance convex, letting simple and efficient convex optimization methods find the globally optimal cache configuration (Chapter 4, see below).

Replacement policies vs. partitioning 2.3.3

Cache partitioning is often used to improve performance in systems with shared caches [110, 124, 137], and is sometimes compared to thread-aware extensions of several replacement policies [44, 66, 69]. However, cache partitioning has many other uses beyond performance, and is better thought of as an enabling technology for *software control of the cache*. Partitioning strikes a nice balance between scratchpad memories, which yield control to software but are hard to use, and conventional hardware-only caches, which are easy to use but opaque to software. For instance, partitioning has been used to improve fairness [110, 116], implement priorities and guarantee QoS [31, 51, 77, 96], improve NUCA designs [12, 91], and eliminate side-channel attacks [115].

Partitioning is therefore a general tool to help achieve *system-level objectives*. After all, caches consume over half of chip area in modern processors [89]; surely, software should have a say in how they are used. Hardware-only replacement policies simply do not support this—they use policies fixed at design time and cannot know what to optimize for. However, in order to allocate capacity intelligently, partition schemes need predictions of cache performance at different sizes.

Predictability: We say that a replacement policy is *predictable* if the *miss curve*, i.e. its miss rate on a given access stream at different partition sizes, can be estimated efficiently. Miss curves allow partitioning policies to reason about the effect of different partition sizes without actually running and measuring performance at each size. Using miss curves, dynamic partitioning algorithms can find and set the

appropriate sizes without trial and error. Because resizing partitions is slow and the space of choices is very large, predictability is highly desirable.

The need for predictability has confined partitioning to LRU in practice. LRU's miss curve can be cheaply sampled in hardware using utility monitors (UMONs) [124], or in software using address-based sampling [41, 132]. However, recent replacement policies lack this predictability. Since they are designed empirically and do not obey the stack property, there is no simple known monitoring scheme that can sample their miss curve cheaply. Likewise, much work has gone into modeling LRU's performance for general access patterns, but little for empirical policies. For example, DIP was analyzed on cyclic, scanning access patterns [125]. While insightful, this analysis does not extend to general access patterns that do not exhibit cyclic behavior.

Alternatively, non-predictable policies can adapt through slow trial and error [31, 55]. However, these schemes can get stuck in local optima, scale poorly beyond few cores (since the design space grows exponentially), and are unresponsive to changes in application behavior since partitions take tens of milliseconds to be resized. PDP [44] includes a cache model, but it is not accurate enough to use in partitioning (see Section C.6).

2.3.4 Convexity

One of the main flaws in LRU is thrashing, exhibited as performance cliffs where performance abruptly improves beyond a threshold size. Not only do performance cliffs cause performance problems, they also make cache management NP-hard and difficult to reason about.

By contrast, convexity means that *the miss curve's slope shrinks as cache space grows* [139]. Convexity implies that the cache yields diminishing returns on performance with increasing cache space, and thus implies the absence of performance cliffs. It has two other important benefits: it makes simple allocation algorithms (e.g., hill climbing) optimal, and it avoids all-or-nothing behavior, improving fairness. Although the benefits of convexity may not be obvious, prior work shows that there is latent demand for convex cache behavior in order to simplify cache management.

Convexity is not a substitute for predictability. Some empirical policies are mostly convex in practice, but without some way of predicting their performance, such policies cannot be effectively controlled by software to achieve system-level objectives.

High-performance cache replacement and partitioning are consequently at loggerheads: techniques that improve single-thread performance are incompatible with those used to manage shared caches. This is unfortunate, since in principle they should be complementary. A key contribution of Talus (Chapter 4) is to eliminate this conflict by capturing much of the benefit of high-performance policies without sacrificing LRU's predictability.

Simple resource allocation: Without convexity, partitioning cache capacity to maximize performance is an NP-complete problem [139]. Existing algorithms yield approximate and often good solutions, but they are either inefficient [124] or complex (Appendix A, [110]). This makes them hard to apply to large-scale systems or multi-resource management [20]. Alternatively, prior work consciously ignores non-convexity and applies convex optimization [20, 31], sacrificing performance for simplicity and reduced overheads. As we will see in Section 4.5, applying convex optimization to LRU wipes out most of the benefits of partitioning. Thus, it is not surprising that prior techniques that use hill climbing or local optimization methods to partition LRU find little benefit [20, 31], and those that use more sophisticated methods, such as Lookahead or dynamic programming, report significant gains [12, 77, 110, 124, 130, 161].

Talus eliminates this tradeoff, as cheap convex optimization (e.g., hill climbing) finds the optimal partition sizes [23, 139]. Talus thus simplifies cache management.

Fairness: Even if optimization complexity is not a concern, convexity also avoids all-or-nothing behavior in allocations and improves fairness. For example, imagine a system with a 32 MB cache running two instances of `libquantum`, an application that scans over 32 MB. If the cache is unpartitioned, both applications will evict each other's lines, getting no hits (both will have effective capacity below the 32 MB cliff). Partitioning can help one of the applications by giving it the whole 32 MB cache, but this is unfair. Any other choice of partition sizes will yield no benefit. Imbalanced partitioning [116] finds that this is common in parallel applications, and proposes to allocate most cache space to a single thread of a homogeneous parallel application to improve performance. Convex miss curves make this unnecessary: with a convex miss curve, giving 16 MB to each instance of `libquantum` accelerates both instances equally and maximizes the cache's hit rate. In general, with homogeneous threads and convex miss curves, the maximum-utility point is equal allocations, which is also the most fair.

In summary, convexity not only provides smooth, stable behavior, but makes optimal resource allocation cheap, improves fairness, and leads to better allocation outcomes.

Cache modeling 2.4

As discussed above, *predictability* is a prerequisite to unlock the myriad benefits of cache partitioning. It is inexpensive to monitor LRU's performance in hardware, but not for empirical replacement policies. Fortunately, analytical cache performance models offer a promising alternative.

Most prior work has developed analytical cache models using *stack distance distributions* [1, 41, 104, 132, 160]. The stack distance of an access is the number of *unique* addresses accessed after the last reference to the same address. For example, in the sequence ABCBDDA the stack distance of A is four, as B, C, and D are accessed between both accesses to A. Stack distances are meaningful for LRU: in a fully-associative LRU cache of S lines, accesses with stack distance $\leq S$ will be hits, and all others will be misses.

However, no prior work gives a general way to predict recent empirical policies. DIP gives a simple performance analysis on scanning access patterns (i.e., iterating over an array), but does not consider performance on real applications, nor does their analysis generalize to such cases. PDP employs an analytical model to choose for how long to protect lines in the cache. PDP's model is simple, but it is inaccurate on common access patterns (Appendix C). Moreover, even if it were accurate, it is limited to modeling protecting distances, and therefore does not solve the general problem of modeling different replacement policies.

Stack distances are inherently meaningful for LRU, but not for recent empirical policies. Our insight is that reuse distances can be used to model cache behavior for a much wider range of replacement policies. The reuse distance of an access is the total number of references after the last reference to the same address. For example, in ABCBDDA the reuse distance of access A is six. Reuse distances correspond to lines' ages: a line with reuse distance d will hit at age d if it is not first evicted. Reuse distance distributions cannot be trivially translated into miss rates. *Our cache model's key innovation is to perform this translation for a broad class of age-based policies* (Appendix C). Prior work has used reuse distances only in a limited context: Sen et al. [132] use reuse distances to model random replacement, but use stack distances for LRU. By contrast, we model all policies through a single framework based on reuse distances. Since we are able to model LRU, our model demonstrates that reuse distances dominate stack distances in predictive power.

2.5 Non-uniform cache access (NUCA) architectures

Partitioning alone scales poorly because it does not optimize placement. NUCA techniques [8, 85] reduce the access latency of large distributed caches, and have been the subject of extensive research. Static NUCA (S-NUCA) [85] simply spreads the data across all banks with a fixed line-bank mapping, and exposes a variable bank access latency. Commercial designs often use S-NUCA [89]. Dynamic NUCA (D-NUCA) schemes improve on S-NUCA by adaptively placing data close to the requesting core [9, 10, 27, 29, 30, 35, 53, 64, 106, 123, 165]. They involve a combination of *placement*, *migration*, and *replication* strategies. Placement and migration dynamically place data close to cores that use it, reducing access latency. Replication makes multiple copies of frequently used lines, reducing latency for widely read-shared lines (e.g., hot code), at the expense of some capacity loss.

2.5.1 Shared- vs private-based NUCA

D-NUCA designs often build on a *private-cache baseline*. Each NUCA bank is treated as a private cache, lines can reside in any bank, and coherence is preserved through a directory-based or snoopy protocol, which is often also leveraged to implement NUCA techniques. For example, Adaptive Selective Replication [9] controls replication by probabilistically deciding whether to store a copy of a remotely fetched line in the local L2 bank; Dynamic Spill-Receive [123] can spill evicted lines to other banks, relying on remote snoops to retrieve them. These schemes are flexible, but they require all LLC capacity to be under a coherence protocol, so they are either hard to scale (in snoopy protocols), or incur significant area, energy, latency, and complexity overheads (in directory-based protocols).

In contrast, some D-NUCA proposals build on a *shared-cache baseline* and leverage virtual memory to perform adaptive placement. Cho and Jin [30] use page coloring and a NUCA-aware allocator to map pages to specific banks. Hardavellas et al. [53] find that most applications have a few distinct classes of accesses (instructions, private data, read-shared, and write-shared data), and propose R-NUCA, which specializes placement and replication policies for each class of accesses on a per-page basis, and significantly outperforms NUCA schemes without this access differentiation. Shared-baseline schemes are simpler, as they require no coherence for LLC data and have a simpler lookup mechanism. However, they may incur significant overheads if remappings are frequent or limit capacity due to restrictive mappings.

2.5.2 Isolation and partitioning in NUCA

Unlike partitioning, most D-NUCA techniques rely on best-effort heuristics with little concern for isolation, so they often improve typical performance at the expense of worst-case degradation, further precluding QoS. Indeed, prior work has shown that D-NUCA often causes significant bank contention and uneven distribution of accesses across banks [10]. We also see this effect in Chapter 3 — R-NUCA has the highest worst-case degradation of all schemes. Dynamic Spill-Receive mitigates this problem with a QoS-aware policy that avoids spills to certain banks [123]. This can protect a local bank from interference, but does not provide partitioning-like capacity control. Virtual Hierarchies rely on a logical two-level directory to partition a cache at bank granularity [103], but this comes at the cost of doubling directory overheads and making misses slower.

Because conventional partitioning techniques (e.g., way partitioning) only provide few partitions and often degrade performance, D-NUCA schemes seldom use them. ASP-NUCA [45], ESP-NUCA [106], and Elastic Cooperative Caching [58] use way partitioning to divide cache banks between private and shared levels. However, this division does not provide isolation, since applications interfere in the shared level.

CloudCache [91] implements virtual private caches that can span multiple banks. Each bank is way partitioned, and partitions are sized with a distance-aware greedy algorithm based on UCP with a limited

Scheme	High capacity	Low Latency	Capacity control	Isolation	Directory-less
Private caches	✗	✓	✗	✓	✗
Shared caches	✓	✗	✗	✗	✓
Partitioned shared caches	✓	✗	✓	✓	✓
Private-based D-NUCA	∽	✓	✓	✗	✗
Shared-based D-NUCA	∽	✓	✗	✗	✓
Virtual caches	✓	✓	✓	✓	✓

Table 2.1: Desirable properties achieved by main cache organizations.

frontier. Unfortunately, CloudCache scales poorly to large virtual caches, as it uses N-chance spilling on evictions, and relies on broadcasts to serve local bank misses, reducing latency at the expense of significant bandwidth and energy (e.g., in a 64-bank cache with 8-way banks, in a virtual cache spanning all banks, a local miss will trigger a full broadcast, causing a 512-way lookup and a chain of 63 evictions).

Virtual caches vs. prior work 2.5.3

We propose virtual caches, software-managed collections of NUCA banks, as an improvement over prior NUCA schemes (Chapter 3). Table 2.1 summarizes the main differences among techniques.

We build on a shared cache baseline to avoid additional coherence. However, instead of mapping pages to locations as in prior work [30, 53], we map pages to *virtual caches*, and decide the physical configuration of the virtual caches independently. This avoids page table changes and TLB shootdowns on reconfigurations, though some reconfigurations still need cache invalidations.

In contrast to prior D-NUCAs, we partition the cache into multiple isolated virtual caches that, due to smart placement, approach the low latency of private caches. Prior schemes often size partitions using hill-climbing (e.g., shadow tags [45] or LRU way hit counters [58]), which can get stuck in local optima, whereas we capture full miss curves to make global decisions.

In contrast to prior partitioned NUCAs, we implement *single-lookup* virtual *shared* caches, providing coordinated placement and capacity management without the overheads of a globally shared directory or multi-level lookups, and performs global (not limited-frontier) capacity partitioning efficiently using novel algorithms (Section 3.3).

Thread scheduling in multicores 2.6

In most cache designs, cache performance is fairly insensitive to thread placement because a thread's location has little impact on how much capacity is available to it or how far away its capacity lies.

Conventional NUCA techniques [85] are concerned with data placement, but do not place threads or divide cache capacity among them. S-NUCA spreads data across banks with a fixed address-bank mapping, exposing variable bank latency. Access latency is thus approximately the average latency to every bank, and although thread placement can change this average latency somewhat (e.g., on open network topologies like meshes), it does not have sufficient impact to gain notice in prior work. D-NUCAs [9, 10, 27, 29, 30, 53, 64, 106, 123, 165] generally either place lines nearby in the local bank or use a fixed, global mapping for lines that do not fit locally (e.g., spilled lines [9, 123] or write-shared data in R-NUCA [53]). There are exceptions, e.g. rotational interleaving in R-NUCA, but these affect only a fraction of cache accesses. Hence a thread's location also has little impact in D-NUCA.

However, recent spatial partitioning techniques such as Jigsaw (Chapter 3), CloudCache [91], and Virtual Hierarchies [103], *are* sensitive to thread placement. To fit applications' working sets, these techniques allocate virtual caches spanning multiple banks and, to minimize access latency, place them near threads. Hence if adjacent threads each access a large virtual cache, then they will compete for capacity in nearby cache banks. This *capacity contention* forces virtual caches to use further-away banks to fit their virtual caches and thereby degrades their access latency. Jigsaw thus also schedules threads to minimize capacity contention, letting virtual caches use closer banks and reducing data movement over the on-chip network.

CRUISE [68] is perhaps the mostly closely related work. CRUISE schedules single-threaded apps in CMPs with multiple fixed-size last-level caches, each shared by multiple cores and unpartitioned. CRUISE takes a classification-based approach, dividing apps in thrashing, fitting, friendly, and insensitive, and applies fixed scheduling policies to each class (spreading some classes among LLCs, using others as filler, etc.). CRUISE bin-packs apps into fixed-size caches, but partitioned NUCA schemes provide flexibly sized virtual caches that can span multiple banks. It is unclear how CRUISE's policies and classification would apply to NUCA. For example, CRUISE would classify `omnet` as thrashing if it was considering many small LLCs, and as insensitive if considering few larger LLCs (see Section 3.1). CRUISE improves on DI [168], which profiles miss rates and schedules apps across chips to balance intensive and non-intensive apps. Both schemes only consider single-threaded apps, and have to contend with the lack of partitioned LLCs.

Other thread placement schemes focus on NUMA systems. NUMA techniques have different goals and constraints than NUCA: the large size and low bandwidth of main memory limit reconfiguration frequency and emphasize bandwidth contention over capacity contention. Tam et al. [144] profile which threads have frequent sharing and place them in the same socket. DINO [22] clusters single-threaded processes to equalize memory intensity, places clusters in different sockets, and migrates pages along with threads. In on-chip NUMA (CMPs with multiple memory controllers), Tumanov et al. [147] and Das et al. [34] profile memory accesses and schedule intensive threads close to their memory controller. These NUMA schemes focus on equalizing memory bandwidth, whereas we find that proper cache allocations cause capacity contention to be the main constraint on thread placement.

Finally, prior work has used high-quality *static mapping* techniques to spatially decompose regular parallel problems. Integer linear programming is useful in spatial architectures [113], and graph partitioning is commonly used in stream scheduling [118, 120]. While some of these techniques could be applied to the thread and data placement problem, they are too expensive to use dynamically, as we will see in Section 3.7.

Jigsaw: Scalable Virtual Caches 3

FOR systems to scale efficiently, data must be close to the computation that uses it. This requires keeping cached data in banks close to threads (to minimize on-chip traffic), while judiciously allocating cache capacity among threads (to minimize cache misses). As we saw in Chapter 2, prior cache designs do not address both issues jointly. On the one hand, prior non-uniform cache access (NUCA) techniques [8–10, 27, 29, 30, 53, 64, 85, 106, 123, 165] have proposed a variety of placement, migration, and replication policies to reduce network distance. However, these best-effort techniques do not explicitly manage capacity between threads, and in fact often result in hotspots and additional interference [10]. On the other hand, prior work has proposed a variety of partitioning techniques [28, 95, 101, 143, 158], but these schemes only work on fully shared caches, and so do not reduce data movement from accessing distributed cache banks.

Moreover, prior work has ignored thread placement in NUCA, which can have a large impact on access latency. Thread placement techniques mainly focus on non-uniform *memory* access (NUMA) [22, 34, 68, 144, 147, 168] and use policies, such as clustering, that do not translate well to NUCA. In contrast to NUMA, where capacity is plentiful but bandwidth is scarce, capacity contention is the main constraint for thread placement in NUCA (Section 3.1).

We present *Jigsaw*, a design that jointly addresses the scalability and interference problems of shared caches. On the hardware side, we partition individual cache banks and add a small, efficient hardware structure that lets software divide the access stream across banks. Jigsaw lets software combine multiple bank partitions into a logical, software-defined cache, which we call a *virtual cache*. By mapping data to virtual caches, and configuring the locations and sizes of the individual bank partitions that compose each virtual cache, software has full control over both where data is placed in the cache, and the capacity allocated to it. Jigsaw efficiently supports reconfiguring virtual caches dynamically and moving data across them, and implements novel monitoring hardware to let software find the optimal virtual cache configuration efficiently.

On the software side, we develop a lightweight system-level runtime that divides data into virtual caches and decides where to place each thread and virtual cache to minimize data movement. We find that to achieve good performance, the system must both place cache capacity well *and* schedule threads to limit capacity contention. This is a complex, multi-dimensional optimization problem that Jigsaw must solve in at most a few milliseconds. We develop novel and efficient resource management algorithms that achieve performance within 1% of impractical, idealized solutions. In particular, we develop *Peekahead*, an exact linear-time implementation of the previously proposed quadratic-time Lookahead algorithm [124], enabling global optimization with non-convex utilities on large caches at negligible overheads.

We evaluate Jigsaw with simulations of 16- and 64-core tiled CMPs. On multiprogrammed mixes of 64 single-threaded workloads, Jigsaw improves weighted speedup by up to 72% (46% gmean) over a conventional shared LLC, up to 51% (32% gmean) over Vantage partitioning [130], up to 53% (32%

gmean) over R-NUCA [53], and up to 25% (20% gmean) over an idealized shared-private D-NUCA organization (IdealSPD) that uses twice the cache capacity of the other schemes [58]. Jigsaw also reduces energy by 36% over a conventional shared LLC. Jigsaw delivers similar benefits on multithreaded application mixes.

Overall, Jigsaw makes the following contributions:

- (Hardware) We design a simple hardware mechanism, the *virtual cache translation buffer* (VTB), that allows software to control the placement of data across cache banks with a single lookup.
- (Hardware) We design miss curve monitors that use geometric sampling to scale to very large NUCA caches efficiently (Section 3.2.3).
- (Hardware) We present novel hardware that enables incremental reconfigurations of NUCA caches, avoiding the bulk invalidations and long pauses that make reconfigurations expensive in prior NUCAs [53, 103].
- (Software) We develop *Peekahead*, an efficient partitioning algorithm that maximizes non-convex utility on very large caches at low overhead.
- (Software) We develop a novel thread and data placement scheme that takes into account both data allocation and access intensity to jointly place threads and data across CMP tiles.

Jigsaw requires simple hardware (3% of LLC area), works transparently to applications, and reconfigures the full chip every few milliseconds with minimal software overheads (0.2% of system cycles). Jigsaw demonstrates that, given the right hardware primitives, software can manage large distributed caches efficiently.

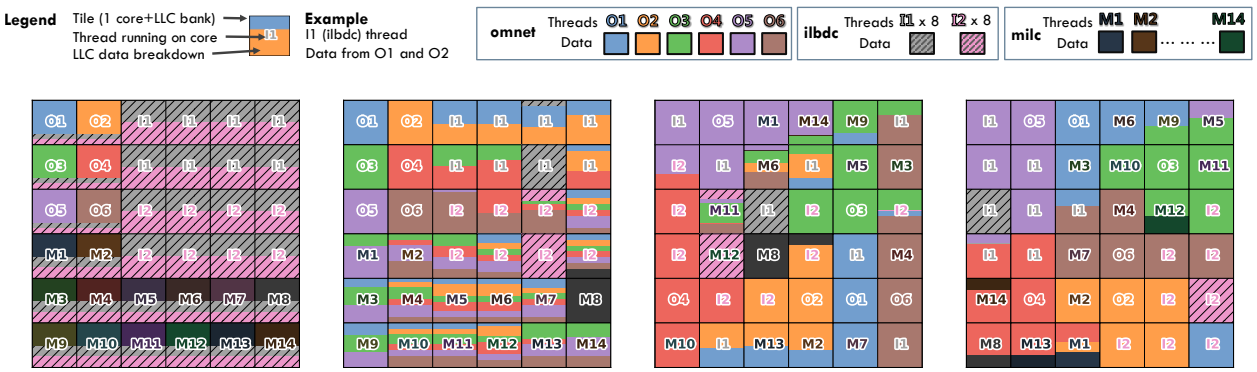
3.1 Motivation

To explore the effect of thread placement on different NUCA schemes, we simulate a 36-core CMP running a specific mix. The CMP is a scaled-down version of the 64-core chip in Figure 3.3, with 6×6 tiles. Each tile has a simple 2-way OOO core and a 512 KB LLC bank. (See Section 3.4 for methodology details.)

We run a mix of single- and multi-threaded workloads. From single-threaded SPEC CPU2006, we run six instances of `omnet` (labeled O1-O6) and 14 instances of `milc` (M1-M14). From multi-threaded SPEC OMP2012, we run two instances of `ilbdc` (labeled I1 and I2) with eight threads each. We choose this mix because it illustrates the effects of thread and data placement—Section 3.5 uses a comprehensive set of benchmarks.

Figure 3.1 shows how threads and data are placed across the chip under different schemes. Each square represents a tile. The label on each tile denotes the thread scheduled in the tile's core (labeled by benchmark as discussed before). The colors on each tile show a breakdown of the data in the tile's bank. Each process uses the same color for its threads and data. For example, in Figure 3.1b, the upper-leftmost tile has thread O1 (colored blue) and its data (also colored blue); data from O1 also occupies parts of the top row of banks (portions in blue).

Figure 3.1a shows the thread and data breakdown under R-NUCA when applications are grouped by type (e.g., the six copies of `omnet` are in the top-left corner). R-NUCA maps thread-private data to each thread's local bank, resulting in very low latency. Banks also have some of the shared data from the multithreaded processes I1 and I2 (shown hatched) because R-NUCA spreads shared data across the chip. Finally, code pages are mapped to different banks using rotational interleaving, though this is not visible in this mix because apps have small code footprints. R-NUCA excels at reducing LLC access latency to



(a) R-NUCA (b) Jigsaw+Clustered (c) Jigsaw+Random (d) Jigsaw

Figure 3.1: Case study: 36-tile CMP with a mix of single- and multi-threaded workloads (omnet×6, milc×14, 8-thread ilbdc×2) under different NUCA organizations and thread placement schemes. Threads are labeled and data is colored by process.

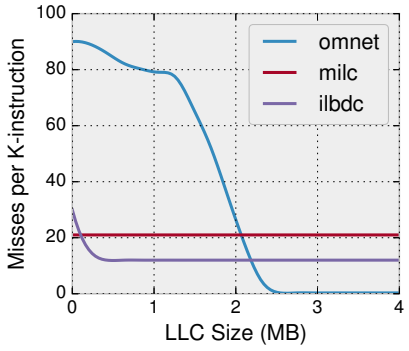


Figure 3.2: Application miss curves.

	omnet	ilbdc	milc	Weighted speedup
R-NUCA	1.09	0.99	1.15	**1.08**
Jigsaw+Clustered	2.88	1.40	1.21	**1.48**
Jigsaw+Random	3.99	1.20	1.21	**1.47**
Jigsaw	4.00	1.40	1.20	**1.56**

Table 3.1: Per-app and weighted speedups for the mix studied.

private data vs. an S-NUCA cache, which spreads all accesses among banks. This helps `milc` and `omnet`, as shown in Figure 3.1. Overall, R-NUCA speeds up this mix by 8% over S-NUCA.

But R-NUCA does not use capacity efficiently in this mix. Figure 3.2 shows why, giving the miss curves of each app. Each miss curve shows the misses per kilo-instruction (MPKI) that each process incurs as a function of LLC space (in MB). `omnet` is very memory-intensive, and suffers 85 MPKI below 2.5 MB. However, over 2.5 MB, its data fits in the cache and misses turn into hits. `ilbdc` is less intensive and has a smaller footprint of 512 KB. `milc` gets no cache hits no matter how much capacity it is given—it is a streaming application. In R-NUCA, `omnet` and `milc` apps get less than 512 KB, which does not benefit them, and `ilbdc` apps use more capacity than they need.

Jigsaw uses capacity more efficiently, giving 2.5 MB to each instance of `omnet`, 512 KB to each `ilbdc` (8 threads), and near-zero capacity to each `milc`. Each application is given its own virtual cache, which is placed in nearby cache banks to minimize access latency. Figure 3.1b uses the same thread placement as R-NUCA and shows how Jigsaw tries to place data close to the threads that use it. By using partitioning, Jigsaw can share banks among multiple types of data without introducing capacity contention. However, the `omnet` threads in the corner heavily contend for capacity of neighboring banks, and their data is placed farther away than if they were spread out. Clearly, when capacity is managed efficiently, thread placement has a large impact on capacity contention and achievable latency. Nevertheless, because `omnet`'s data now fits in the cache, its performance vastly improves, by 2.88× over S-NUCA (its AMAT

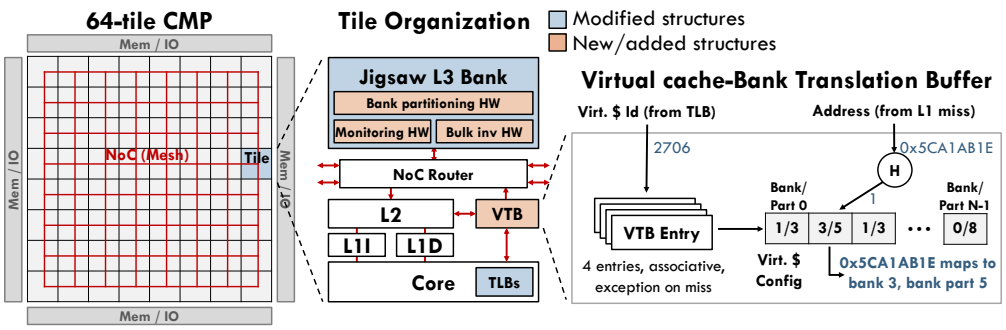


Figure 3.3: Jigsaw overview: target tiled CMP, tile configuration with microarchitectural changes and additions introduced by Jigsaw, and virtual cache-bank translation buffer (VTB).

improves from 15.2 to 3.7 cycles, and its IPC improves from 0.22 to 0.61). `ilbdc` is also faster, because its shared data is placed close by instead of across the chip; and because `omnet` does not consume memory bandwidth anymore, `milc` instances have more of it and speed up moderately (Figure 3.1). Overall, Jigsaw speeds up this mix by 48% over S-NUCA.

Figure 3.1c shows the effect of randomizing thread placement to spread capacity contention among the chip. `omnet` instances now have their data in neighboring banks (1.2 hops on average, instead of 3.2 hops in Figure 3.1b) and enjoy a 3.99× speedup over S-NUCA. Unfortunately, `ilbdc`'s threads are spread further, and its performance suffers relative to clustering threads (Figure 3.1). This shows why one policy does not fit all: depending on capacity contention and sharing behavior, apps prefer different placements. Specializing policies for single- and multithreaded apps would only be a partial solution, since multithreaded apps with large per-thread footprints and little sharing also benefit from spreading.

Finally, Figure 3.1d shows how Jigsaw places threads to further reduce data movement. Jigsaw spreads `omnet` instances across the chip, avoiding capacity contention, but clusters `ilbdc` instances across their shared data. Jigsaw achieves a 4× speedup for `omnet` and a 40% speedup for `ilbdc`. Jigsaw speeds up this mix by 56%.

In summary, this case study shows that partitioned NUCA schemes use capacity more effectively and improve performance, but they are sensitive to thread placement, as threads in neighboring tiles can aggressively contend for capacity.

3.2 Jigsaw hardware

Jigsaw exposes on-chip caches to software and enables their efficient management using a small set of primitives. First, Jigsaw lets software explicitly divide a distributed cache in collections of bank partitions, which we call *virtual caches*. Virtual caches can be dynamically reconfigured by changing the size of each bank partition. Second, Jigsaw provides facilities to map data to virtual caches, and to quickly migrate data among virtual caches. Third, Jigsaw implements virtual cache monitoring hardware to let software find the optimal virtual cache configuration efficiently.

3.2.1 Virtual caches

Figure 3.3 illustrates the overall organization of Jigsaw. Jigsaw banks can be divided in bank partitions. Jigsaw is agnostic to the partitioning scheme used, as well as the array type and replacement policy. As discussed in Chapter 2, in our evaluation we select Vantage partitioning due to its ability to partition banks at a fine granularity with minimal costs.

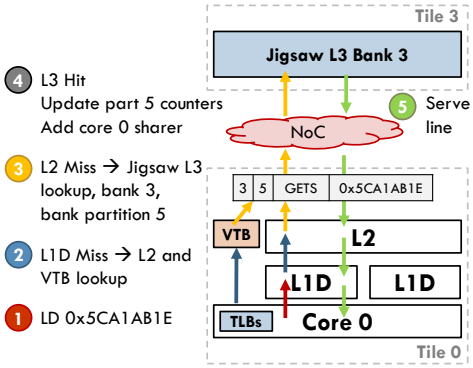


Figure 3.4: Jigsaw L3 access, including VTB lookup in parallel with L2 access.

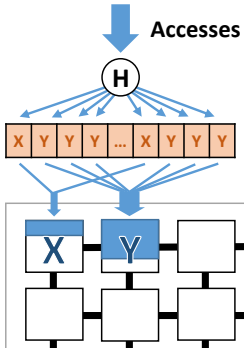


Figure 3.5: The VTB spreads accesses across capacity.

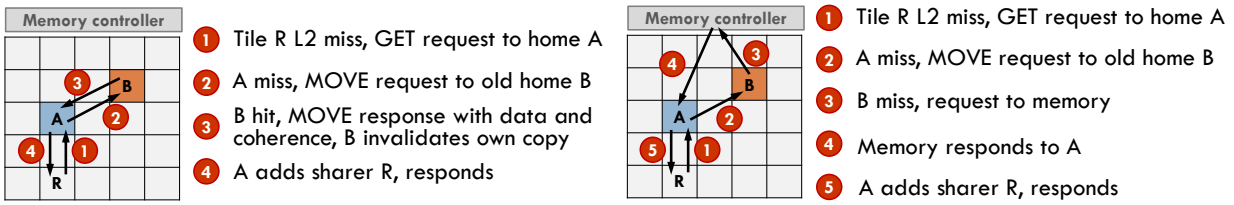
Virtual caches are configurable collections of bank partitions, visible to software. Each virtual cache has a unique id number and comprises a set of bank partitions that can be sized independently. The virtual cache size is the sum of its bank partition sizes. The virtual cache id is independent from the individual partition ids.

We could exploit virtual caches in two ways. On the one hand, we could assign cores to virtual caches, having virtual caches behave as virtual private caches. This is transparent to software, but would require a coherence directory for LLC data. On the other hand, we can map data to virtual caches. This avoids the need for coherence beyond the private (L2) caches, as each line can only reside in a single location. Mapping data to virtual caches also enables specializing virtual caches to different types of data (e.g., shared vs. thread-private [53]). For these reasons, we choose to map data to virtual caches. This has the added benefit that an LLC access requires only *one cache bank lookup* in Jigsaw.

Jigsaw leverages the virtual memory subsystem to map data to virtual caches. Figure 3.3 illustrates this implementation, highlighting the microarchitectural structures added and modified. Specifically, we add a virtual cache id to each page table entry, and extend the TLB to store the virtual cache id. Virtual cache ids are needed in L2 accesses, so these changes should not slow down page translations.

On a miss on the private cache levels, a per-core *virtual cache-bank translation buffer* (VTB) finds the bank the line maps to, as well as its bank partition. Figure 3.3 depicts the per-core VTBs. Each VTB has a small number of resident virtual caches descriptors. Like in a software-managed TLB, an access to a non-resident virtual cache causes an exception, and system software can refill the VTB. As we will see in the next section, supporting a small number of resident virtual caches per core (typically 4) is sufficient. Each virtual cache descriptor consists of an array of N bank and bank partition ids. To perform a translation, Jigsaw hashes the address and uses the hash value to select an array entry. We take the VTB translation latency out of the critical path by doing it speculatively on L2 accesses. Figure 3.4 details the different steps involved in a Jigsaw cache access.

There are several interesting design dimensions in the VTB. First, the hash function can be as simple as bit-selection. However, to simplify virtual cache management, the VTB should divide the requests into sub-streams *with statistically similar access patterns*. A more robust hash function can achieve this. Specifically, we use an H3 hash function (H in Figure 3.3), which is universal and efficient to implement in hardware [26]. All VTBs implement the same hash function. Second, increasing N, the number of entries in a virtual cache descriptor, lets us fine-tune the load we put on each bank to adapt to heterogeneous bank partition sizes. For example, Figure 3.5 shows a virtual cache consisting of bank partitions X and Y. X is 128 KB and Y is 384 KB (3× larger). By setting one-fourth of VTB entries to X and three-fourths to Y, Y receives 3× more accesses than X. Thus, together X and Y behave like a 512 KB cache.



(a) Demand move, old bank hit.

(b) Demand move, old bank miss.

Figure 3.6: Messages and protocol used on incremental reconfigurations: demand moves when old bank hits or misses.

In our implementation, we choose N equal to twice the number of banks, so virtual caches spanning few bank partitions can be finely tuned to bank partition sizes, but large virtual caches that span most banks cannot. For a 64-core system with 64 banks in which each bank has 64 partitions, bank and bank partition ids are 6 bits, and each virtual cache descriptor takes 768 bits (96 bytes). Supporting four virtual caches can be done with less than 400 bytes, a 0.2% storage overhead over the private cache sizes. Alternatively, more complex weighted hash functions or more restrictive mappings can reduce this overhead.

Coherence: Jigsaw maintains coherence using two invariants. First, in-cache directories track the contents of private caches. 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 bank and then to main memory). For example, if a line mapped to the top-right bank in Figure 3.3, then accesses to that line will go that bank regardless of which core issued the access. This mapping only changes infrequently, when the system is reconfigured. Having all accesses to a given address follow the same path maintains coherence automatically.

3.2.2 Dynamic adaptation

So far we have seen how Jigsaw works on a static configuration. To adapt dynamically, however, we must also support both reconfiguring a virtual cache and remapping data to another virtual cache.

Virtual cache reconfiguration: Virtual caches can be changed in two dimensions. First, per-bank partition sizes can be dynamically changed. This concerns the bank partitioning technique used (e.g., in Vantage, this requires changing a few registers [130]), and is transparent to Jigsaw. Second, the virtual cache descriptor (i.e., the mapping of lines to bank partitions) can also change at runtime, to change either the bank partitions that comprise the virtual cache, or the load put on each bank partition. We observe that with few hardware modifications, we can spatially reconfigure the cache incrementally, without pausing cores, by moving lines to their new location. To our knowledge, this is the first scheme to achieve on-chip data migration through moves without requiring a coherence directory.

The key idea is to, upon a reconfiguration, temporarily treat the cache as a two-level hierarchy. We add a *shadow VC descriptor* to each VTB entry, doubling the number of descriptors. Upon reconfiguration, each core copies the VC descriptors into the shadow descriptors, and updates the normal VC descriptors with the new configuration. When the shadow descriptors are active, the VTB finds both the current and previous banks and bank partitions for the line, and, if they are different, sends the old bank id along with its request. Figure 3.6 illustrates this protocol. If the current bank misses, it forwards the request to the old bank instead of memory. If the old bank hits, it sends both the line and its coherence state to the

new bank, invalidating its own copy. This moves the line to its new location. We call this a *demand move* (Figure 3.6a). If the old bank misses, it forwards the request to the memory controller (Figure 3.6b). Demand moves have no races because all requests follow the same path through both virtual levels, i.e. they access the same sequence of cache banks. If a second request for the same line arrives at the new bank, it is queued at the MSHR that's currently being used to serve the first request.

Demand moves quickly migrate frequently used lines to their new locations, but the old banks must still be checked until all lines have moved. This adds latency to cache accesses. To limit this cost, banks walk the array in the background and incrementally invalidate lines whose location has changed. These *background invalidations* are not on the critical path and can proceed at a comparatively slow rate. For example, by scanning one set every 200 cycles, background invalidations finish in 100 K cycles. Background invalidations begin after a short period (e.g., 50 K cycles) to allow frequently-accessed lines to migrate via demand moves. After banks walk the entire array, cores stop using the shadow VTB descriptors and resume normal operation.

In addition to background invalidations, we also experimented with background moves, i.e. having banks send lines to their new locations instead of invalidating them. However, we found that background moves and background invalidations performed similarly—most of the benefit comes from not pausing cores to invalidate lines that have moved. We prefer background invalidations because they are simpler: background moves require additional state at every bank (viz., *where* the line needs to be moved, not just that its location has changed), and require a more sophisticated protocol (as there can be races between demand and background moves).

Page remapping: To classify pages dynamically (Section 3.3), software must also be able to remap a page to a different virtual cache. A remap is similar to a TLB shootdown: the initiating core quiesces other cores where the virtual cache is accessible with an IPI; it then issues a bulk invalidation of the page. Once all the banks involved finish the invalidation, the core changes the virtual cache in the page table entry. Finally, quiesced cores update the stale TLB entry before resuming execution. Page remaps typically take a few hundred cycles, less than the associated TLB shootdown, and are rare in our runtime, so their performance effects are negligible.

Invalidations due to both remappings and reconfigurations could be avoided with an extra directory between Jigsaw and main memory. Section 3.5 shows that this is costly and not needed, as reconfiguration overheads are negligible.

Monitoring 3.2.3

In order to make reasonable partitioning decisions, software needs monitoring hardware that gives accurate and timely information. As discussed in Chapter 2, utility monitors (UMONs) [124] are an efficient way to gather miss curves. Prior partitioning schemes use per-core UMONs [124, 130], but this is insufficient in Jigsaw, as virtual caches can be accessed from multiple cores, and different cores often have wildly different access patterns to the same data. Moreover, as CMPs scale to larger core counts, the total size of the LLC makes conventional UMONs prohibitively expensive.

Monitoring miss curves in large CMPs is challenging. To allocate capacity efficiently, we should manage it in small chunks (e.g., the size of the L1s) so that it isn't over-allocated where it produces little benefit. This is crucial for VCs with small working sets, which see large gains from a small size and no benefit beyond. Yet we also need miss curves that cover the full LLC because a few VCs may benefit from taking most capacity. These two requirements—fine granularity and large coverage—are problematic for existing monitors.

UMONs were originally designed to work with set-associative caches, and worked by sampling a small but statistically significant number of sets. UMONs gather the miss curve by counting hits at each

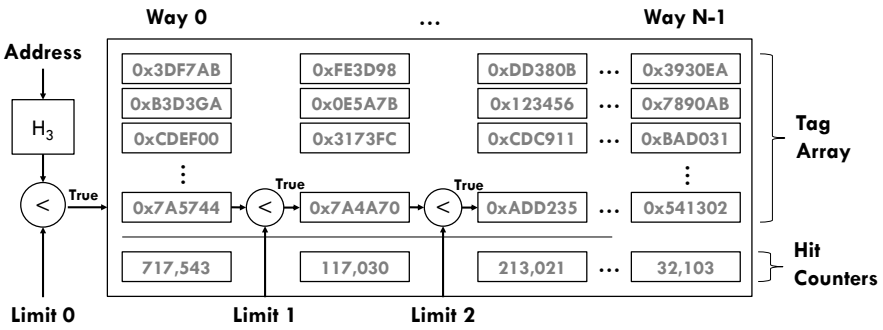


Figure 3.7: GMONs augment UMONs with per-way *limit registers* that vary the sampling rate across ways.

way. UMONs can also be used with other cache designs [130] by sampling a fraction of cache accesses at the UMON. Given high enough associativity, a sampling ratio of UMON lines : S behaves like a cache of S lines (Chapter 4, Theorem 4). Moreover, the number of UMON ways determines the resolution of the miss curve: an W-way UMON yields $W+1$-point miss curves. UMONs monitor a fixed cache capacity per way, and would require a prohibitively large associativity to achieve both fine detail and large coverage. Specifically, in a UMON with W ways, each way models $1/W$ of LLC capacity. To allocate a 32 MB LLC in 64 KB chunks, a conventional UMON needs 512 ways to have enough resolution. This is expensive to implement, even for infrequently used monitors.

Instead, we develop a novel monitor, called a *geometric monitor* (GMON). GMONs need fewer ways—64 in our implementation—to model capacities from 64 KB up to 32 MB. This is possible because GMONs vary the sampling rate across ways, giving both fine detail for small allocations and large coverage, while using many fewer ways than conventional UMONs. Figure 3.7 shows this design: A GMON consists of small set-associative, tag-only array. Instead of storing address tags, GMON tags store 16-bit hashed addresses. GMONs also have a *limit register* per way. The limit registers progressively decrease across ways, and are used to filter out a fraction of lines per way as follows. In a conventional UMON, when an address is inserted or moved up to the first way, all other tags are moved to the next way. (This requires potentially shifting as many tags as ways, but only a small fraction of accesses are monitored, so the energy impact is small.) In a GMON, on each move, the hash value of the tag is checked against the way's limit register. If the value exceeds the limit register, the tag is discarded instead of moving to the next way, and the process terminates. Discarding lines achieves a variable, decreasing sampling rate per way, so GMONs model an increasing capacity per way (Chapter 4, Theorem 4 and [81]).

We set limit registers to decrease the sampling rate by a factor $\gamma < 1$, so the sampling rate at way w is $k_w = \gamma^w$ less than at way zero (Section A.4), and then choose γ to cover the full cache capacity. For example, with a 32 MB LLC, a 64-way GMON with $\gamma \approx 0.95$ covers the full cache while having the first way model 64 KB. Modeled capacity per way grows by 26×, from 0.125 to 3.3 banks. This means GMONs miss curves are sparse, with high resolution at small sizes, and reduced resolution at large sizes. We find these GMONs work as well as the impractical UMONs described above (Section 3.7).

We place each virtual cache's monitor in a fixed location on the chip to avoid clearing or migrating monitor state. Since cores already hash lines to index the VTB, we add the initial limit value (corresponding to k_0) to the VTB so cores know when to forward the hashed address over the network to the GMON. Monitoring is off the critical path, so this has no performance impact as long as the traffic is kept low.

Each monitor array consists of $s = 1$ K lines organized into $W = 64$ ways. Lines are 16-bit, hashed, partial tags. To achieve a minimum allocation of 64 KB with 512 KB LLC banks, we choose $k_0 = 1/64$. This means monitor traffic is one-sixty-fourth of a S-NUCA cache, and in fact less since each update is only 16 bits. With 512 KB LLC banks (8 K×64 B lines) and two monitors per bank, this is 0.8% overhead, in line with prior work [124].

Jigsaw software 3.3

Virtual caches are a general mechanism with multiple potential uses (e.g., maximizing throughput or fairness, providing strict process isolation, implementing virtual local stores, or avoiding side-channel attacks). In this work, we design a system-level runtime that leverages Jigsaw to jointly improve cache utilization and access latency transparently to user-level software. The runtime first classifies data into virtual caches, then periodically decides how to size and where to place each virtual cache.

Virtual caches and page mapping 3.3.1

Jigsaw defines three types of virtual caches: global, per-process, and per-thread. Jigsaw maps pages accessed by multiple processes (e.g., OS code and data, library code) to the (unique) global virtual cache. Pages accessed by multiple threads in the same process are mapped to a per-process virtual cache. Finally, each thread has a per-thread virtual cache. With this scheme, each core's VTB uses three entries, but there are a large number of virtual caches in the system.

Similar to R-NUCA [53], page classification is done incrementally and lazily, at TLB/VTB miss time. When a thread performs the first access to a page, it maps it to its per-thread virtual cache. If another thread from the same process tries to access the page, the page is remapped to the per-process virtual cache. On an access from a different process (e.g., due to IPC), the page is remapped to the global virtual cache. When a process finishes, its virtual caches are deallocated and bulk-invalidated.

A simple cost model for thread and data placement 3.3.2

Jigsaw configures virtual caches to minimize overall data movement. As discussed in Section 2.6, classification-based heuristics are hard to apply to NUCA. Instead, Jigsaw uses a simple analytical cost model that captures the effects of different configurations on *total memory access latency*, and uses it to find a low-cost solution. This latency is better analyzed as the sum of on-chip (L2-to-LLC) and off-chip (LLC-to-memory) latencies.

Off-chip latency: Assume a system with T threads and D virtual caches (VCs). Each thread t accesses VC d at a rate $a_{t,d}$ (e.g., 50 K accesses in 10 ms). If VC d is allocated s_d lines in the cache, its miss ratio is $M_d(s_d)$ (e.g., 10% of accesses miss). Then the total off-chip access latency is:

$$\text{Off-chip latency} = \sum_{t=1}^{T} \sum_{d=1}^{D} a_{t,d} \times M_d(s_d) \times \text{MemLatency} \tag{3.1}$$

where MemLatency is the latency of a memory access. This includes network latency, and relies on the average distance of all cores to memory controllers being the same (Figure 3.3). Extending Jigsaw to NUMA would require modeling a variable main memory latency in Equation 3.1.

On-chip latency: Each VC's capacity allocation s_d consists of portions of the N banks on chip, so that $s_d = \sum_{b=1}^{N} s_{d,b}$. The capacity of each bank B constrains allocations, so that $B \geq \sum_{d=1}^{D} s_{d,b}$. Limited bank capacities sometimes force data to be further away from the threads that access it, as we saw in Section 3.1. Because the VTB spreads accesses across banks in proportion to capacity, the number of accesses from thread t to bank b is $\alpha_{t,b} = \sum_{d=1}^{D} \frac{s_{d,b}}{s_d} \times a_{t,d}$. If thread t is placed in a core c_t and the

Figure 3.8: Overview of Jigsaw's periodic reconfiguration procedure.

(a) Dependency cycle.

(b) Cycle broken.

Figure 3.9: Thread and data placement are interdependent. Jigsaw breaks this cycle by placing data twice, first optimistically and second after thread placement is known.

network distance between two tiles t_1 and t_2 is $D(t_1, t_2)$, then the on-chip latency is:

$$\text{On-chip latency} = \sum_{t=1}^{T} \sum_{b=1}^{N} \alpha_{t,b} \times D(c_t, b) \tag{3.2}$$

3.3.3 Overview of Jigsaw reconfigurations

Figure 3.8 gives an overview of a Jigsaw reconfiguration. Scalable geometric monitors sample the miss curves of each virtual cache. An OS runtime periodically reads these miss curves and uses them to jointly place VCs and threads using a 4-step procedure that accounts for both access latency and capacity contention. Finally, this runtime uses hardware support to move cache lines to their new locations.

With the cost model above, the virtual cache configuration problem is to choose the c_t (thread placement) and $s_{t,b}$ (VC size and data placement) that minimize total latency, subject to the given constraints. However, finding the optimal solution is NP-hard [59, 117], and different factors are intertwined. For example, the size of VCs and the thread placement affect how close data can be placed to the threads that use it.

Jigsaw takes a multi-step approach to disentangle these interdependencies. Jigsaw first adopts optimistic assumptions about the contention introduced by thread and data placement, and chooses the size of virtual caches under these optimistic assumptions. Jigsaw then gradually refines these assumptions to produce the final data and thread placement. Figure 3.9 shows this schematically: given VC sizes, thread and data placement are interdependent because the best placement for either depends on the other's placement. Jigsaw breaks the dependency cycle by performing placement twice. Specifically, reconfigurations consist of four steps:

1. *Latency-aware allocation* divides capacity among VCs assuming data is compactly placed (no capacity contention).

2. *Optimistic contention-aware VC placement* places VCs among banks to avoid capacity contention. This step produces a rough picture of where data should be in the chip, e.g. placing `omnet` VCs far away enough in Figure 3.1 to avoid the pathological contention in Figure 3.1b.

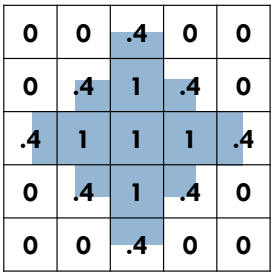
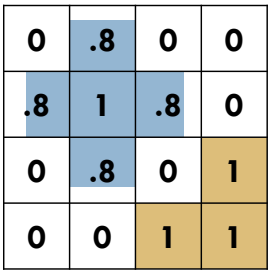
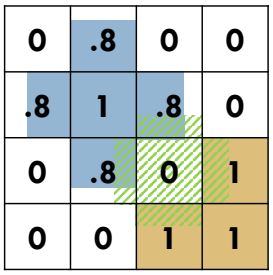


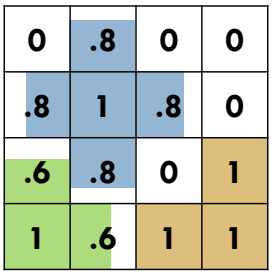
Figure 3.10: Optimistic virtual cache placement.



(a) Partial optimistic data placement



(b) Estimating contention for VC



(c) VC placed near least contended tile

Figure 3.11: Optimistic, contention-aware virtual cache placement.

3. *Thread placement* uses the optimistic VC placement to place threads close to the VCs they access. For example, in Figure 3.1d, this step places `omnet` applications close to the center of mass of their data, and clusters `ilbdc` threads around their shared data.

4. *Refined VC placement* improves on the previous data placement to, now that thread locations are known, place data closer to minimize on-chip latency. For example, a thread that accesses its data intensely may swap allocations with a less intensive thread to bring its data closer; while this increases latency for the other thread, overall it is beneficial.

By considering data placement twice (steps 2 and 4), Jigsaw accounts for the circular relationship between thread and data placement. We were unable to obtain comparable results with a single VC placement step; and Jigsaw performs almost as well as impractically expensive schemes, such as integer linear programming (Section 3.7). Jigsaw uses arbitrary distance vectors, so it works with arbitrary topologies. However, to make the discussion concrete, we use a mesh topology in the examples.

Latency-aware capacity allocation 3.3.4

As we saw in Section 3.1, VC sizes have a large impact on *both* off-chip latency and on-chip latency. Prior work has partitioned cache capacity to reduce cache misses [124], i.e. off-chip latency. However, it is well-known that larger caches take longer to access [54, 56, 146]. Most prior partitioning work has targeted fixed-size LLCs with constant latency. But capacity allocation in NUCA caches provides an opportunity to also reduce on-chip latency: if an application sees little reduction in misses from a larger VC, the additional network latency to access it can negate the benefits of having fewer misses.

In summary, larger allocations have two competing effects: decreasing off-chip latency and increasing on-chip latency. This is illustrated in Figure 3.12, which shows the average memory access latency to one VC (e.g., a thread's private data). Figure 3.12 breaks latency into its off- and on-chip components, and shows that there is an optimal size that minimizes total latency.

This has two important consequences. First, unlike in other D-NUCA schemes, it is sometimes better to leave cache capacity *unused*. Second, incorporating on-chip latency changes the curve's shape and also its marginal utility [124], leading to different cache allocations even when all capacity is used.

Jigsaw allocates capacity from *total memory latency curves* (the sum of Equation 3.1 and Equation 3.2) instead of miss curves. However, Equation 3.2 requires knowing the thread and data placements, which are unknown at the first step of reconfiguration, i.e. when sizing VCs. Jigsaw instead uses an optimistic on-chip latency curve, found by compactly placing the VC around the center of the chip and computing the resulting average latency. For example, Figure 3.10 shows the optimistic placement of an 8 ⅕-bank VC accessed by a single thread, with an average distance of 1.27 hops. Optimistic on-chip latency curves are constant, so we precompute them from the network topology.

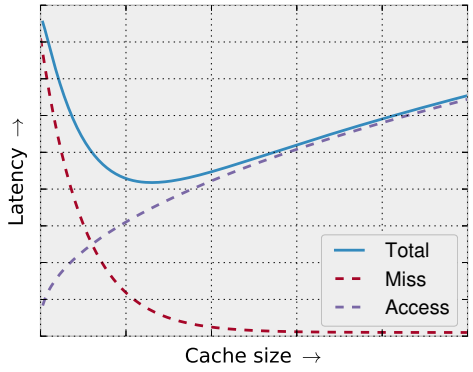


Figure 3.12: Memory access latency broken into access latency (increasing) and miss latency (decreasing).

With this simplification, Jigsaw partitions capacity to minimize total memory latency. While these allocations account for on-chip latency, they generally underestimate it due to capacity contention. Nevertheless, we find that this scheme works well because the next steps are effective at limiting contention.

Conceptually, sizing virtual caches is no different than in UCP: the runtime computes the per-virtual cache latency curves, then runs Lookahead [124] to compute the virtual cache sizes that minimize latency. Unfortunately, using Lookahead is unfeasible. Lookahead greedily allocates space to the partition that provides the highest *utility per unit* (hits per allocation quantum). Because miss curves are not generally convex, on each step Lookahead traverses each miss curve looking for the maximum utility per unit it can achieve with the remaining unallocated space. This results in an $O(P \cdot S^2)$ run-time, where P is the number of partitions and S is the cache size in allocation quanta, or "buckets". With way partitioning, S is small (the number of ways) and this is an acceptable overhead. In Jigsaw, banks can be finely partitioned, and we must consider all banks jointly. Lookahead is too inefficient at this scale.

Virtual cache sizing with Peekahead: To address this, we develop the *Peekahead* algorithm, an exact $O(P \cdot S)$ implementation of Lookahead. We leverage the insight that the point that achieves the maximum utility per unit is the next one in the *convex hull* of the miss curve. For example, Figure 3.13 shows a non-convex miss curve (blue) and its convex hull (red). With an unlimited budget, i.e., abundant unallocated cache space, and starting from A, D gives maximal utility per unit (steepest slope); starting from D, H gives the maximal utility per unit; and so on along the convex hull. With a limited budget, i.e. if the remaining unallocated space limits the allocation to S', the point that yields maximum utility per unit is the next one in the convex hull of the miss curve in the region $[0, S']$. For example, if we are at D and are given limit S' between F and G, the convex hull up to S' is the line $\overline{DFS'}$ and F yields maximal utility per unit. Conversely, if S' lies between G and H, then the convex hull is $\overline{DS'}$, S' is the best option, and the algorithm terminates (all space is allocated).

If we know these *points of interest* (POIs), the points that constitute all reachable convex hulls, traversing the miss curves on each allocation becomes unnecessary: given the current allocation, the next relevant POI always gives the maximum utility per unit. For example, in Figure 3.13, the only POIs are A, D, F, and H; Table 3.2 shows all possible decisions. Figure 3.14 shows several example miss curves and their POIs. Note that some POIs do not lie on the full convex hull (dashed lines), but are always on the convex hull of some sub-domain. The animation at [11] shows how to construct all intermediate convex hulls from the POIs.

Peekahead first finds all POIs in $O(S)$ for each partition. This is inspired by the three coins algo-

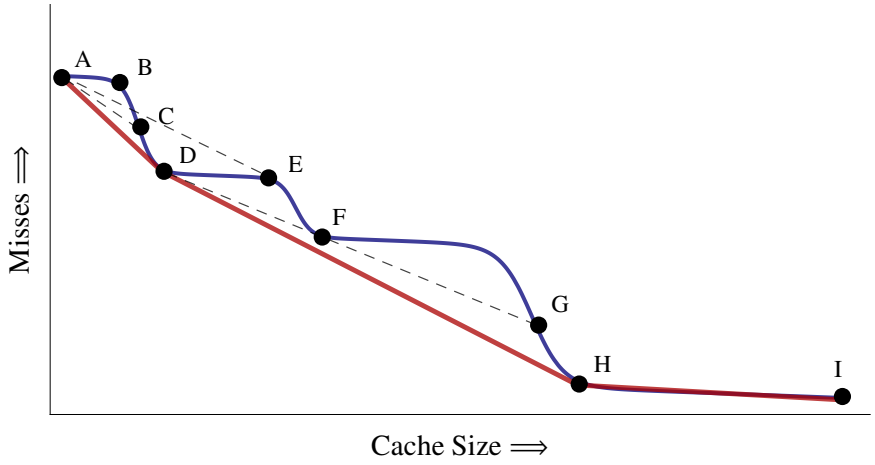


Figure 3.13: Non-convex miss curve (blue), and its convex hull (red).

		Maximum allocation, S'				
		$S' < D$	$D \le S' < F$	$F \le S' < G$	$G \le S' < H$	$H \le S'$
Start	**A**	S'	D	D	D	D
	D	-	S'	F	S'	H
	F	-	-	S'	-	-
	H	-	-	-	-	S'

Table 3.2: Maximal utility allocations for Figure 3.13 across the entire domain from all possible starting positions.

rithm [105]. For example, we construct the convex hull $\overline{ADHI}$ in Figure 3.13 by considering points from left to right. At each step, we add the next point to the hull, and then backtrack to remove previous points that no longer lie on the hull. We begin with the line $\overline{AB}$. C is added to form $\overline{ABC}$, and then we backtrack. Because B lies above $\overline{AC}$, it is removed, leaving $\overline{AC}$. Similarly, D replaces C, leaving $\overline{AD}$. Next, E is added to form $\overline{ADE}$, but since D lies below $\overline{AE}$, it is *not* removed. Continuing, F replaces E, G replaces F, H replaces G, and finally I is added to give the convex hull $\overline{ADHI}$.

We extend this algorithm to build all convex hulls over $[0, X]$ for any X up to S, which produces all POIs. We achieve this in $O(S)$ by not always deleting points during backtracking. Instead, we mark points in convex regions with the x-coordinate at which the point becomes obsolete, termed the *horizon* (e.g., F's horizon is G). Such a point is part of the convex hull up to its horizon, after which it is superseded by the higher-utility-per-unit points that follow. However, if a point is in a concave region then it is not part of any convex hull, so it is deleted (e.g., C and G).

Appendix A lists the algorithms in detail, proves their correctness, and analyzes their asymptotic performance.

Optimistic contention-aware VC placement 3.3.5

Once VC sizes are known, Jigsaw first finds a rough picture of how data should be placed around the chip to avoid placing large VCs close to each other. The main goal of this step is to *inform thread placement* by avoiding VC placements that produce high capacity contention, as in Figure 3.1b.

To this end, we sort VCs by size and place the largest ones first. Intuitively, this works well because larger VCs can cause more contention, while small VCs can fit in a fraction of a bank and cause little

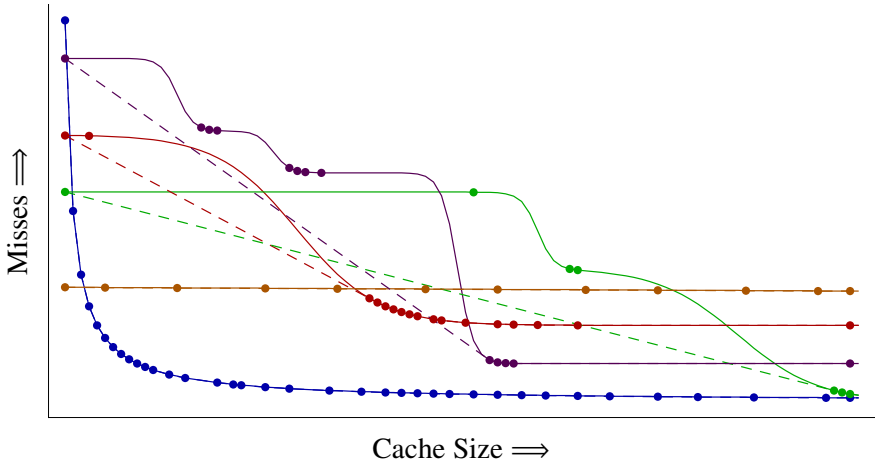


Figure 3.14: Points of interest (POIs) for several example miss curves. Dashed lines denote their convex hulls.

contention. For each VC, the algorithm iterates over all the banks, and chooses the bank that yields the least contention with already-placed VCs as the center of mass of the current VC. To make this search efficient, we approximate contention by keeping a running tally of *claimed capacity* in each bank, and relax capacity constraints, allowing VCs to claim more capacity than is available at each bank. With N banks and D VCs, the algorithm runs in $O(N \cdot D)$.

Figure 3.11 shows an example of optimistic contention-aware VC placement at work. Figure 3.11a shows claimed capacity after two VCs have been placed. Figure 3.11b shows the contention for the next VC at the center of the mesh (hatched), where the uncontended placement is a cross. Contention is approximated as the claimed capacity in the banks covered by the hatched area—or 3 ⅗ in this case. To place a single VC, we compute the contention around that tile. We then place the VC around the tile that had the lowest contention, updating the claimed capacity accordingly. For instance, Figure 3.11c shows the final placement for the third VC in our example.

3.3.6 Thread placement

Given the previous data placement, Jigsaw tries to place threads closest to the center of mass of their accesses. Recall that each thread accesses multiple VCs, so this center of mass is computed by weighting the centers of mass of each VC by the thread's accesses to that VC. Placing the thread at this center of mass minimizes its on-chip latency (Equation 3.2).

Unfortunately, threads sometimes have the same centers of mass. To break ties, Jigsaw places threads in descending *intensity-capacity product* (sum of VC accesses × VC size for each VC accessed). Intuitively, this order prioritizes threads for which low on-chip latency is important, and for which VCs are hard to move. For example, in the Section 3.1 case study, `omnet` accesses a large VC very intensively, so `omnet` instances are placed first. `ilbdc` accesses moderately-sized shared data at moderate intensity, so its threads are placed second, clustered around their shared VCs. Finally, `milc` instances access their private VCs intensely, but these VCs are tiny, so they are placed last. This is fine because the next step, refined VC placement, can move small VCs to be close to their accessors very easily, with little effect on capacity contention.

For multithreaded workloads, this approach clusters shared-heavy threads around their shared VC, and spreads private-heavy threads to be close to their private VCs. Should threads access private and shared data with similar intensities, Jigsaw places threads relatively close to their shared VC but does

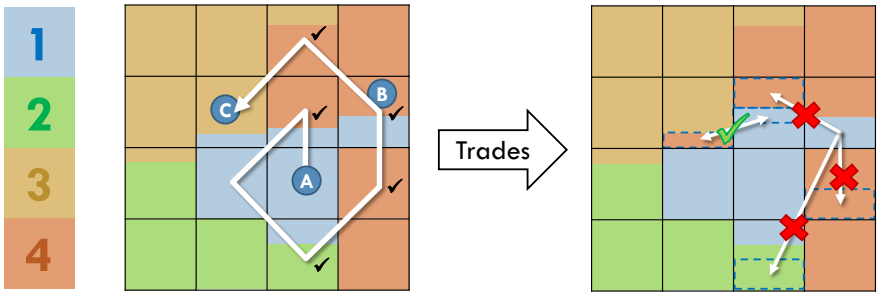


Figure 3.15: Trading data placement: Starting from a simple initial placement, VCs trade capacity to move their data closer. Only trades that reduce total latency are permitted.

not tightly cluster them, avoiding capacity contention among their private VCs.

Refined VC placement 3.3.7

Finally, Jigsaw performs a round of detailed VC placement to reduce the distance between threads and their data.

Jigsaw first simply iterates over VCs, placing capacity as close to threads as possible without violating capacity constraints. This greedy scheme is a reasonable starting point, but produces sub-optimal placements. For example, a thread's private VC always gets space in its local bank, regardless of the thread's memory intensity. Also, shared VCs can often be moved at little or no cost to make room for data that is more sensitive to placement. This is because moving shared data farther away from one accessing thread often moves it closer to another.

Furthermore, unlike in previous steps, it is straightforward to compute the effects of moving data, since we have an concrete placement to compare against. Jigsaw therefore looks for beneficial trades between pairs of VCs after the initial, greedy placement. Specifically, Jigsaw computes the latency change from trading capacity between VC_1 at bank b_1 and VC_2 at bank b_2 using Equation 3.2. The change in latency for VC_1 is:

$$\Delta\text{Latency} = \frac{\text{Accesses}}{\text{Capacity}} \times \big(D(\text{VC}_1, b_1) - D(\text{VC}_1, b_2)\big)$$

The first factor is VC_1's accesses per byte of allocated capacity, which we also call the *intensity*. Multiplying by this factor accounts for the number of accesses that are affected by moving capacity, which varies between VCs. The equation for VC_2 is similar, and the net effect of the trade is their sum. If the net effect is negative (lower latency is better), then the VCs swap bank capacity.

Naïvely enumerating all possible trades is prohibitively expensive, however. Instead, Jigsaw performs a bounded search by iterating over all VCs: Each VC spirals outward from its center of mass, trying to move its data closer. At each bank b along the outward spiral, if the VC has not claimed all of b's capacity then it adds b to a list of desirable banks. These are the banks it will try to trade into later. Next, the VC tries to move its data placed in b (if any) closer by iterating over closer, desirable banks and offering trades with VCs that have data in these banks. If the trades are beneficial, they are performed. The spiral terminates when the VC has seen all of its data, since no farther banks will allow it to move any data closer.

Figure 3.15 illustrates this for an example CMP with four VCs and some initial data placement. For example, to perform a bounded search for VC1, we spiral outward starting from VC1's center of mass at bank A, and terminate at VC1's farthest data at bank C. Desirable banks are marked with black checks on

the left of Figure 3.15. We only attempt a few trades, shown on the right side of Figure 3.15. At bank B, VC1's data is two hops away, so we try to trade it to any closer, marked bank. For illustration, suppose none of the trades are beneficial, so the data does not move. This repeats at bank C, but suppose the first trade is now beneficial. VC1 and VC4 trade capacity, moving VC1's data one hop closer.

This approach gives every VC a chance to improve its placement. Since any beneficial trade must benefit one party, it would discover all beneficial trades. However, for efficiency, in Jigsaw each VC trades only once, since we have empirically found this discovers most trades. Finally, this scheme incurs negligible overheads, as we will see in Section 3.7.

These techniques are cheap and effective. We also experimented with more expensive approaches commonly used in placement problems: integer linear programming, simulated annealing, and graph partitioning. Section 3.7 shows that they yield minor gains and are too expensive to be used online.

3.3.8 Putting it all together

Hardware overheads: Implementing Jigsaw as described imposes small overheads, which are a well worth the achieved system-wide performance and energy savings:

- Each bank is partitioned. With 512 KB banks and 64-byte lines, Vantage adds 8 KB of state per bank to support 64 bank partitions [12] (each tag needs a 6-bit partition id and each bank needs 256 bits of per-partition state).
- Each tile's VTB is 588 bytes: 576 bytes for 6 VC descriptors (3 normal + 3 shadow) and 12 bytes for the 3 tags.
- Jigsaw uses 4 monitors per bank. Each GMON has 1024 tags and 64 ways. Each tag is a 16-bit hash value (we do not store full addresses, since rare false positives are fine for monitoring purposes). Each way has a 16-bit limit register. This yields 2.1 KB monitors, and 8.4 KB overhead per tile.

In total, Jigsaw requires 17.1 KB of state per tile (2.9% of the space devoted to the tile's bank) and simple logic. Overheads are similar to prior partitioning-based schemes [12, 124].

Software overheads: Periodically (every 25 ms in our implementation), a software runtime wakes up on core 0 and performs the steps in Figure 3.8. Reconfiguration steps are at most quadratic on the number of tiles (Section 3.3.5), and most are more efficient. Software overheads are small, 0.2% of system cycles, and are detailed in Section 3.7 and Table 3.5.

Jigsaw without partitioning: If partitioned banks are not desirable, Jigsaw can be used as-is with non-partitioned NUCA schemes [30, 64, 103]. To allow more VCs than threads, we could use several smaller banks per tile (e.g., 4×128 KB), and size and place VCs at bank granularity. This would eliminate partitioning overheads, but would make VTBs larger and force coarser allocations. We evaluate the effects of such a configuration in Section 3.7. Jigsaw's thread placement could also be used in spilling D-NUCAs [123], though the cost model (Section 3.3.2) would need to change to account for the multiple cache and directory lookups.

3.4 Experimental Methodology

Modeled systems: We perform microarchitectural, execution-driven simulation using zsim [131], an x86-64 simulator based on Pin [98], and model tiled CMPs with 16 and 64 cores and a 3-level cache

Cores	16 / 64 cores, x86-64 ISA, 2 GHz, in-order IPC =1 except on memory accesses / Silvermont-like OOO [75]: 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
L1 caches	32 KB, 8-way set-associative, split D/I, 3-cycle latency
L2 caches	128 KB private per-core, 8-way set-associative, inclusive, 6-cycle latency
L3 cache	1 MB/512 KB per tile, 4-way 52-candidate zcache, 9 cycles, inclusive, S-NUCA/R-NUCA/Vantage/Jigsaw/idealized shared-private D-NUCA with 2× capacity (IdealSPD)
Coherence protocol	MESI protocol, 64 B lines, in-cache directory, no silent drops; sequential consistency
Global NoC	8×8 mesh, 128-bit flits and links, X-Y routing, 3-cycle pipelined routers, 1-cycle links
Memory controllers	1/4 MCUs, 1 channel/MCU, 120 cycles zero-load latency, 12.8 GB/s per channel

Table 3.3: Configuration of the simulated 16-core/64-core CMPs.

hierarchy, as shown in Figure 3.3. We use both simple in-order core models and detailed OOO models similar to Silvermont [75]. The 64-core CMP, with parameters shown in Table 3.3, is organized in 64 tiles, connected with an 8×8 mesh network-on-chip (NoC), and has 4 memory controllers at the edges. The scaled-down 16-core CMP has 16 tiles, a 4×4 mesh, and a single memory controller. The 16-core CMP has a total LLC capacity of 16 MB (1 MB/tile), and the 64-core CMP has 32 MB (512 KB/tile). The 64-core system is similar to Knights Landing [62]. We use McPAT [94] to derive the area and energy numbers of chip components (cores, caches, NoC, and memory controller) at 22 nm, and Micron DDR3L datasheets [107] to compute main memory energy. With simple cores, the 16-core system is implementable in 102 mm^2 and has a typical power consumption of 10-20 W in our workloads, consistent with adjusted area and power of Atom-based systems [49].

Cache implementations: Experiments use an unpartitioned, shared (static NUCA) cache with LRU replacement as the baseline (S-NUCA). We compare Jigsaw with Vantage, a representative partitioned design, and R-NUCA, a representative shared-baseline D-NUCA design. Because private-baseline D-NUCA schemes modify the coherence protocol, they are hard to model. Instead, we model an idealized shared-private D-NUCA scheme, IdealSPD, with 2× the LLC capacity. In IdealSPD, each tile has a private L3 cache of the same size as the LLC bank (512 KB or 1 MB), a fully provisioned 5-cycle directory bank that tracks the L3s, and a 9-cycle exclusive L4 bank (512 KB or 1 MB). Accesses that miss in the private L3 are serviced by the proper directory bank (traversing the NoC). The L4 bank acts as a victim cache, and is accessed in parallel with the directory to minimize latency. This models D-NUCA schemes that partition the LLC between shared and private regions, but gives the full LLC capacity to both the private (L3) and shared (L4) regions. Herrero et al. [58] show that this idealized scheme *always* outperforms several state-of-the-art private-baseline D-NUCA schemes that include shared-private partitioning, selective replication, and adaptive spilling (DCC [57], ASR [9], and ECC [58]), often by significant margins (up to 30%).

Vantage and Jigsaw both reconfigure every 50 M cycles (25 ms). Vantage uses utility-based cache partitioning (UCP) [124]. R-NUCA is configured as proposed [53] with 4-way rotational interleaving and page-based reclassification. Jigsaw and R-NUCA use the page remapping support discussed in Section 3.2, and Jigsaw uses demand-moves with background invalidations. Jigsaw uses thread-private and per-process virtual caches. In all configurations, banks use 4-way 52-candidate zcache arrays [129] with H3 hash functions, though results are similar with more expensive 32-way set-associative hashed arrays.

Workloads and metrics: We simulate mixes of single- and multi-threaded workloads. For single-threaded mixes, we use a similar methodology to prior partitioning work [124, 130]. We pin each application to a specific core, and fast-forward all applications for 20 billion instructions. We use a fixed-work methodology and equalize sample lengths to avoid sample imbalance, similar to FIESTA [60]: First, we run each application in isolation, and measure the number of instructions I_i that it executes in 1 billion cycles. Then, in each experiment we simulate the full mix until all applications have executed at least I_i instructions, and consider only the first I_i instructions of each application when reporting aggregate metrics. This ensures that each mix runs for at least 1 billion cycles. Our per-workload performance metric is $perf_i = IPC_i$.

For detailed study of the 16-core, in-order system, we classify all 29 SPEC CPU2006 workloads into four types according to their cache behavior: insensitive (`n`), cache-friendly (`f`), cache-fitting (`t`), and streaming (`s`) as in [130, Table 2], and build random mixes of all the 35 possible combinations of four workload types. We generate four mixes per possible combination, for a total of 140 mixes.

For mixes on the 64-core, out-of-order system, we use the 16 SPEC CPU2006 apps with ≥5 L2 MPKI: `bzip2`, `gcc`, `bwaves`, `mcf`, `milc`, `zeusmp`, `cactusADM`, `leslie3d`, `soplex`, `calculix`, `GemsFDTD`, `libquantum`, `lbm`, `astar`, `omnet`, `sphinx3`, and `xalancbmk`. We simulate mixes of 1–64 randomly selected apps.

For multi-threaded mixes, we simulate SPEC OMP2012, PARSEC, SPLASH-2, and BioParallel workloads. In-order results use ten parallel benchmarks from PARSEC [18] (`blackscholes`, `canneal`, `fluidanimate`, `swaptions`), SPLASH-2 (`barnes`, `ocean`, `fft`, `lu`, `radix`), and BioParallel [67] (`svm`), and out-of-order results use SPEC OMP2012. Since IPC is not a valid measure of work in multithreaded workloads [5], we instrument each app with heartbeats that report global progress (e.g., when each timestep or transaction finishes). For each app, we find the smallest number of heartbeats that complete in over 1 billion cycles from the start of the parallel region when running alone. This is the ROI. We then run the mixes by fast-forwarding all apps to the start of their parallel regions, and running the full mix until all apps complete their ROI. To avoid biasing throughput by ROI length, our per-application performance metric is $perf_i = ROItime_{i,alone}/ROItime_i$.

We report throughput and fairness metrics: normalized throughput, $\sum_i perf_i / \sum_i perf_{i,base}$, and weighted speedup, $(\sum_i perf_i/perf_{i,base})/N_{apps}$, which accounts for fairness [124, 136]. To achieve statistically significant results, we introduce small amounts of non-determinism [5], and perform enough runs to achieve 95% confidence intervals ≤1% on all results.

3.5 Evaluation without Thread Placement

We first compare Jigsaw against alternative cache organizations on the 16-core, in-order CMP to demonstrate the benefits of virtual caches and analyze their performance in detail. Simple, IPC=1 cores let us study the memory system in detail without confounding factors (e.g., memory-level parallelism) introduced by out-order cores. These results do *not* include Jigsaw's contention-aware thread placement; they instead focus on the benefits of virtual caches alone. In Section 3.6, we evaluate single- and multi-threaded mixes with thread placement on the 64-core, out-of-order CMP to show how Jigsaw's performance and energy savings increase with larger systems.

3.5.1 Single-threaded mixes on 16-core, in-order CMP

Performance across all mixes: Figure 3.16 summarizes both throughput and weighted speedup for the cache organizations we consider across the 140 mixes. Each line shows the performance improvement of a single organization against the shared S-NUCA baseline. For each line, workload mixes (the x-axis)

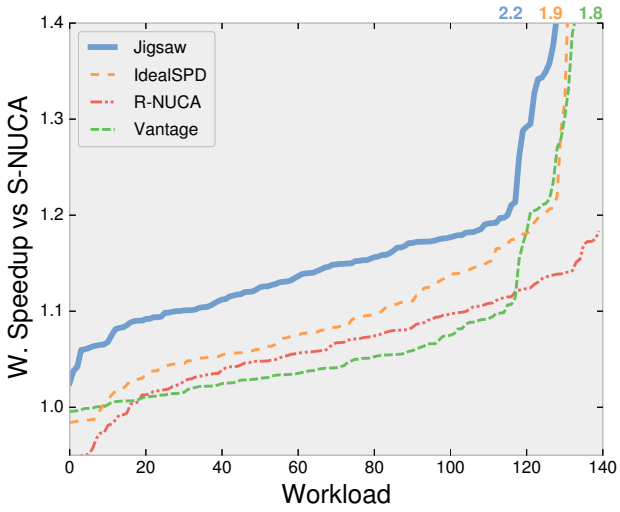


Figure 3.16: Weighted speedup of Jigsaw, Vantage, R-NUCA, and IdealSPD (with 2× cache space) over the S-NUCA baseline, for 140 SPEC CPU2006 mixes on the 16-core chip with in-order cores.

are sorted according to the improvement achieved. Lines are sorted independently, so these graphs give a concise summary of improvements, but should not be used for workload-by-workload comparisons among schemes.

Figure 3.16 shows that Jigsaw is beneficial for all mixes, and achieves large throughput and fairness gains: up to 2.2× weighted speedup over an unpartitioned shared cache. Overall, Jigsaw achieves gmean weighted speedups of 18.4%, Vantage achieves 8.2%, R-NUCA achieves 6.3%, and IdealSPD achieves 11.4%.

Performance of memory-intensive mixes: These mixes have a wide range of behaviors, and many access memory infrequently. For the mixes with the highest 20% of memory intensities (aggregate LLC MPKIs in S-NUCA), where the LLC organization can have a large impact, the achieved gmean weighted speedups are 29.2% for Jigsaw, 19.7% for Vantage, 4.8% for R-NUCA, and 14% for IdealSPD. Jigsaw and Vantage are well above their average speedups, R-NUCA is well *below*, and IdealSPD is about the same. Most of these mixes are at the high end of the lines in Figure 3.16 for Jigsaw and Vantage, but not for R-NUCA. R-NUCA suffers on memory-intensive mixes because its main focus is to *reduce LLC access latency, not MPKI*, and IdealSPD does not improve memory-intensive mixes much because it provides *no capacity control in the shared region*.

Figure 3.17 illustrates this observation, showing the performance of each mix versus memory intensity. Performance is measured by weighted speedup over S-NUCA, and memory intensity is measured by S-NUCA's average memory access time. The figure also includes the best-fit linear regression, which shows the performance-memory intensity correlation for each cache organization. Jigsaw and Vantage both perform increasingly well with increasing memory intensity, exhibiting similar, positive correlation. Vantage is the worst-performing scheme at low memory intensity, but under high intensity it is only bested by Jigsaw. In constract, R-NUCA's performance degrades with increasing memory intensity due to its limited capacity. Finally, IdealSPD shows only slightly increasing performance versus S-NUCA despite having twice the cache capacity. In [58] IdealSPD was (predictably) shown to outperform other D-NUCA schemes by the largest amount on mixes with high memory intensity. Therefore, it's unclear if actual D-NUCA schemes would realize even the modest improvement with increasing memory intensity seen in Figure 3.17. Jigsaw performs similarly to D-NUCA schemes at low memory intensity, and at high

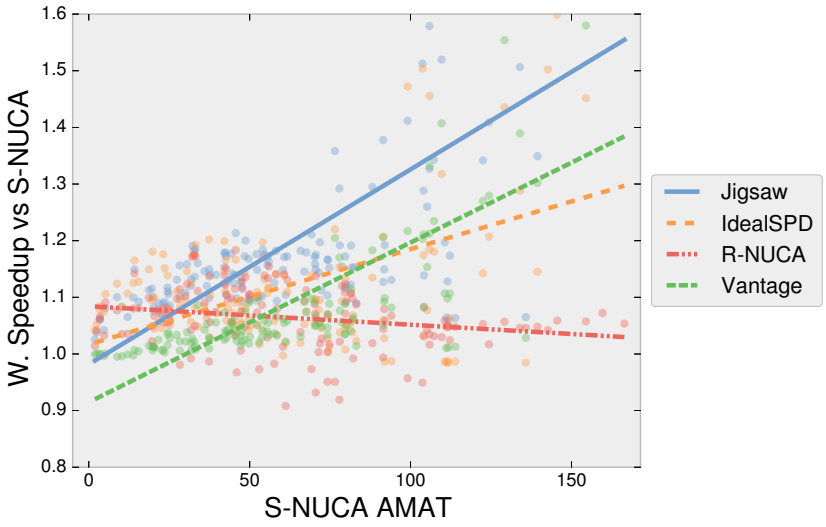


Figure 3.17: Performance with increasing memory intensity. Weighted speedup is plotted vs. S-NUCA's average memory access time for each mix on the 16-core chip. Lines indicate the best-fit linear regression.

intensities is clearly the best organization.

Performance breakdown: To gain more insight into these differences, Figure 3.18 shows a breakdown of execution time for nine representative mixes. Each bar shows the total number of cycles across all workloads in the mix for a specific configuration, normalized to S-NUCA's (the inverse of each bar is throughput over S-NUCA). Each bar further breaks down where cycles are spent, either executing instructions or stalled on a memory access. Memory accesses are split into their L2, NoC, LLC, and memory components. For R-NUCA and Jigsaw, we include time spent on reconfigurations and remappings, which is negligible. For IdealSPD, the LLC contribution includes time spent in private L3, directory, and shared L4 accesses.

We see four broad classes of behavior: First, in *capacity-insensitive mixes* (e.g., `fffn0`, `ssff3`, and `snnn3`) partitioning barely helps, either because applications have small working sets that fit in their local banks or have streaming behavior. Vantage thus performs much like S-NUCA on these mixes. R-NUCA improves performance by keeping data in the closest bank (with single-threaded mixes, R-NUCA behaves like a private LLC organization without a globally shared directory). Jigsaw maps virtual caches to their closest banks, achieving similar improvements. IdealSPD behaves like R-NUCA on low memory intensity mixes (e.g., `fffn0`) where the shared region is lightly used so all data lives in local banks. With increasing memory intensity (e.g. `ssff3` and `snnn3`) its directory overheads (network and LLC) begin to resemble S-NUCA, so much so that its overall performance matches S-NUCA at `snnn3`. For all remaining mixes in Figure 3.18, IdealSPD has similar network and LLC overheads to S-NUCA.

Second, *capacity-critical mixes* (e.g., `stnn0`) contain applications that do not fit within a single bank, but share the cache effectively without partitioning. Here, Vantage and IdealSPD show no advantage over S-NUCA, but R-NUCA in particular performs poorly, yielding higher MPKI than the shared S-NUCA baseline. Jigsaw gets the benefit of low latency, but without sacrificing the MPKI advantages of higher capacity.

Third, in *partitioning-friendly mixes* (e.g., `ftnn2`, `sssf3`, and `ttnn2`) each application gets different utility from the cache, but no single application dominates LLC capacity. Partitioning reduces MPKI slightly, whereas R-NUCA gets MPKI similar to the shared S-NUCA baseline, but with lower network latency. IdealSPD performs somewhere between Vantage and S-NUCA because it does not partition within the

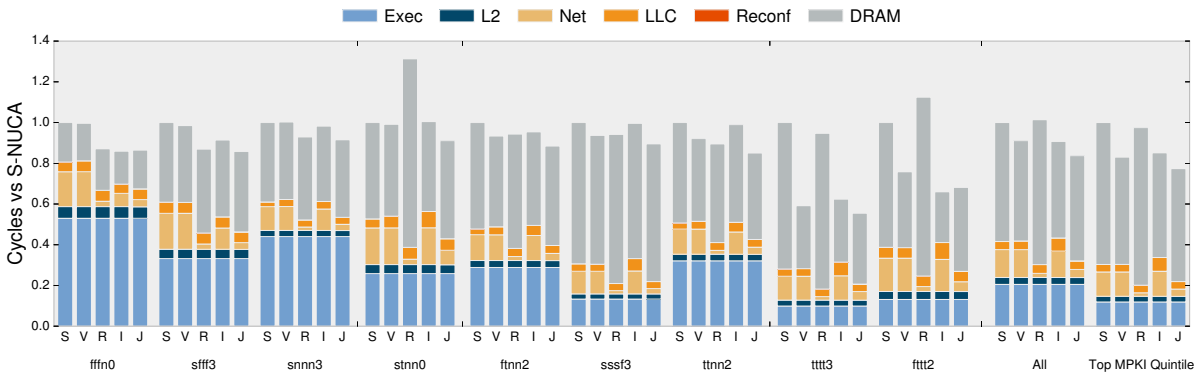


Figure 3.18: Execution time breakdown of S-NUCA (S), Vantage (V), R-NUCA (R), IdealSPD (I), and Jigsaw (J), for representative 16-core single-thread mixes. Cycles are normalized to S-NUCA's (*lower is better*).

shared region. Jigsaw captures the benefits of both partitioning and low latency, achieving the best performance of any scheme.

`ftnn2` is typical, where Vantage gives modest but non-negligible gains, and IdealSPD matches this performance by allowing each application a private bank (capturing high-locality accesses) and dividing shared capacity among applications (giving additional capacity to high-intensity apps). R-NUCA gets very low latency, but at the cost of additional MPKI which makes it ultimately perform worse than Vantage. Jigsaw gets the benefits of both partitioning and low latency, and achieves the best performance. `sssf3` shows a similar pattern, but IdealSPD's directory overheads (specifically directory accesses, included in LLC access time) hurt its performance compared to Vantage.

`ttnn2` demonstrates the importance of capacity control. Vantage shows significant reductions in DRAM time, but IdealSPD resembles (almost identically) the shared S-NUCA baseline. This performance loss is caused by the lack of capacity control within the shared region.

Fourth, *partitioning-critical mixes* (e.g., `tttt3` and `ftttt2`) consist of cache-fitting apps that perform poorly below a certain capacity threshold, after which their MPKI drops sharply. In these mixes, a shared cache is ineffective at dividing capacity, and partitioning achieves large gains. R-NUCA limits apps to their local bank and performs poorly. Jigsaw is able to combine the advantages of partitioning with the low latency of smart placement, achieving the best performance.

In some mixes (e.g., `ftttt2`), IdealSPD achieves a lower MPKI than Vantage and Jigsaw, but this is an artifact of having twice the capacity. Realistic shared-private D-NUCA schemes will always get less benefit from partitioning than Vantage or Jigsaw, as they partition between shared and private regions, but do not partition the shared region among applications. This effect is fairly common, indicating that our results may overestimate the top-end performance of private-baseline NUCA schemes.

Finally, Figure 3.18 shows the average breakdown of cycles across all mixes and for the mixes with the top 20% of memory intensity. They follow the trends already discussed: Vantage reduces DRAM time but not network latency. R-NUCA reduces network latency significantly, but at the cost of additional DRAM time. IdealSPD performs similarly to Vantage; in particular, its global directory overheads are significant. Jigsaw is the only organization to achieve both low DRAM time and low network latency. The main difference on high memory intensity mixes is that Vantage and Jigsaw reduce DRAM time significantly more than other organizations.

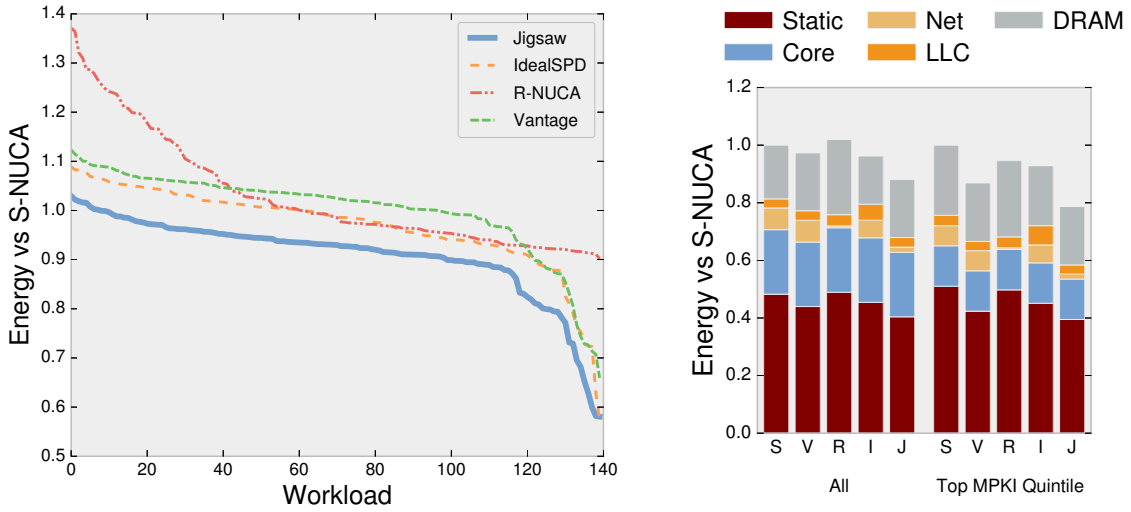


Figure 3.19: System energy across LLC organizations on 16-core, in-order chip: per-mix results, and average energy breakdown across mixes. Results are normalized to S-NUCA's energy (*lower is better*).

Energy: Figure 3.19 shows the system energy (full chip and main memory) consumed by each cache organization for each of the 140 mixes, normalized to S-NUCA's energy. Lower numbers are better. Jigsaw achieves the largest energy reductions, up to 42%, 10.6% on average, and 22.5% for the mixes with the highest 20% of memory intensities. Figure 3.19 also shows the per-component breakdown of energy consumption for the different cache organizations, showing the reasons for Jigsaw's savings: its higher performance reduces static (leakage and refresh) energy, and Jigsaw reduces both NoC and main memory dynamic energy. These results do not model UMON or VTB energy overheads because they are negligible. Each UMON is small and is accessed infrequently. VTBs are less than 400 bytes, and each VTB lookup reads only 12 bits of state.

3.5.2 Multi-threaded mixes on 64-core, in-order CMP

Figure 3.20 shows weighted speedup results of different organizations on 40 random mixes of four 16-thread workloads in a 64-core CMP with in-order cores. We include two variants of Jigsaw: one with a single per-process virtual cache (Jigsaw (P)), and another with additional thread-private virtual caches as discussed in Section 3.3 (Jigsaw). Jigsaw achieves the highest improvements of all schemes. Overall, gmean weighted speedups are 8.9% for Jigsaw, 2.6% for Vantage, 4.7% for R-NUCA, and 5.5% for IdealSPD.

Unlike the single-threaded mixes, most applications are capacity-insensitive and have low memory intensity; only `canneal` is cache-friendly, and `ocean` is cache-fitting. This is why we model a 32 MB LLC: a 64 MB LLC improves weighted speedup by only 3.5%. Longer network latencies emphasize smart placement, further de-emphasizing MPKI reduction. Consequently, Vantage yields small benefits except on the few mixes that contain `canneal` or `ocean`. IdealSPD enjoys the low latency of large local banks as well as a large shared cache, but read-write sharing is slower due to the deeper private hierarchy and global directory, ultimately yielding modest improvements. This drawback is characteristic of all private-based D-NUCA schemes. On the other hand, R-NUCA achieves low latency and, unlike the single-threaded mixes, does not suffer from limited capacity. This is both because of lower memory intensity and because R-NUCA uses shared cache capacity for data shared among multiple threads.

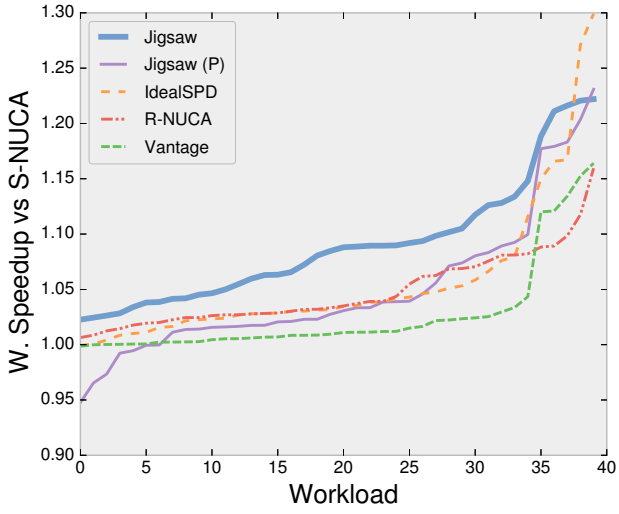


Figure 3.20: Weighted speedup of Jigsaw (w/ and w/o per-thread virtual caches), Vantage, R-NUCA, and IdealSPD (2× cache space) over S-NUCA, for 40 4×16-thread mixes on 64-core chip with in-order cores.

Jigsaw (P) does better than Vantage, but worse than Jigsaw due to the lack of per-thread virtual caches. Jigsaw achieves lower network latency than R-NUCA and outperforms it further when partitioning is beneficial. Note that R-NUCA and Jigsaw reduce network latency by different means. R-NUCA places private data in the local bank, replicates instructions, and spreads shared data across all banks. Jigsaw just does placement: per-thread virtual caches in the local bank, and per-process virtual caches in the local quadrant of the chip. This reduces latency more than placing data throughout the chip and avoids capacity loss from replication. Because there is little capacity contention, we tried a modified R-NUCA that replicates read-only data (i.e., all pages follow a Private → Shared Read-only → Shared Read-write classification). This modified R-NUCA improves weighted speedup by 8.5% over S-NUCA, bridging much of the gap with Jigsaw. While Jigsaw could implement fixed-degree replication a là R-NUCA, we defer implementing an adaptive replication scheme (e.g., using cost-benefit analysis and integrating it in the runtime) to future work.

Summary of results 3.5.3

Figure 3.21 summarizes our findings with in-order cores, using 16-core mixes. For each cache organization, each mix is represented by a single point. Each point's x-coordinate is its LLC and main memory latency (excluding network) normalized to S-NUCA, and the y-coordinate is its network latency normalized to S-NUCA; lower is better in both dimensions. This representation tries to decouple each organization's intrinsic benefits in MPKI and latency reduction from the specific timing of the system. Overall, we draw the following conclusions:

- *Vantage is able to significantly reduce MPKI, but has no impact on network latency.* Vantage partitions within each bank, but does not trade capacity between banks to reduce access latency. In many mixes, network time far exceeds the savings in DRAM time, and a scheme that improved data placement could yield much greater improvements.

- *R-NUCA achieves low network latency, but at the cost of increased MPKI for a significant portion of mixes*. Often the losses in MPKI exceed the savings in network latency, so much so that R-NUCA has the worst-case degradation of all schemes.

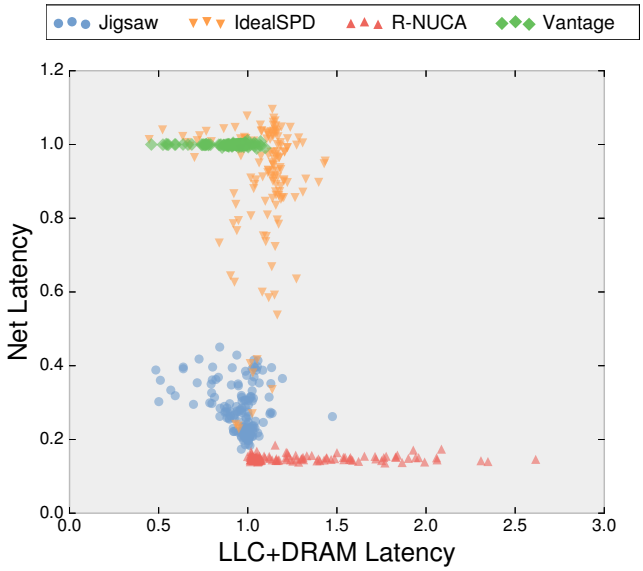


Figure 3.21: Intrinsic MPKI and network latency reduction benefits of Jigsaw, Vantage, R-NUCA, and IdealSPD (with 2× cache space) over S-NUCA. Each point shows the average LLC + memory latency (x) and network latency (y) of one mix normalized to S-NUCA's (*lower is better*).

- *IdealSPD is able to act as either a private-cache or shared-cache organization, but cannot realize their benefits simultaneously.* IdealSPD can match the main memory performance of Vantage on many mixes (albeit with twice the capacity) and R-NUCA's low latency on some mixes. However, IdealSPD struggles to do both due to its shared/private dichotomy, shown by its ⅂-shape in Figure 3.21. Mixes can achieve low latency only by avoiding the shared region. With high memory intensity, global directory overheads become significant, and it behaves as a shared cache.
- *Jigsaw combines the latency reduction of D-NUCA schemes with the miss reduction of partitioning*, achieving the best performance on a wide range of workloads. Indeed, for most mixes, Jigsaw is the only scheme that simultaneously improves network and access latency.

3.6 Evaluation with Thread Placement

We now evaluate Jigsaw on larger systems with out-of-order cores. We show that Jigsaw's performance advantage increases on these larger systems and that virtual caches allow thread placement to further reduce data movement.

3.6.1 Single-threaded mixes on 64-core, out-of-order CMP

Figure 3.22 shows the distribution of weighted speedups that Vantage, IdealSPD, R-NUCA, and Jigsaw achieve in 50 mixes of 64 randomly-chosen, memory-intensive SPEC CPU2006 applications. We find that S-NUCA, Vantage, IdealSPD, and R-NUCA are insensitive to thread placement (performance changes by $\leq 1\%$): S-NUCA, Vantage, and IdealSPD spread accesses among banks in the shared level, and R-NUCA because its policies cause little contention, as explained in Section 3.1. Therefore, we report results for both with a *random* scheduler, where threads are placed randomly at initialization, and stay pinned.

Figure 3.22 shows that Jigsaw significantly improves system performance, achieving 46% gmean weighted speedup and up to 76%. IdealSPD achieves 21% gmean weighted speedup and up to 42%,

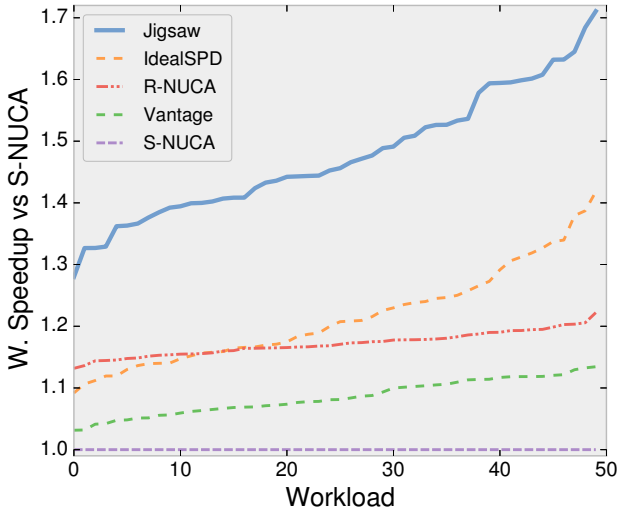


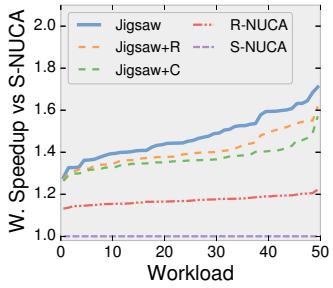
Figure 3.22: Weighted speedup of Jigsaw, Vantage, R-NUCA, and IdealSPD (2× cache space) over S-NUCA, for 50 mixes of 64 SPEC CPU2006 apps on a 64-core CMP with OOO cores.

R-NUCA achieves 18% gmean weighted speedup and up to 23%, and Vantage achieves 8.4% gmean weighted speedup and up to 13%. These results mirror those seen with 16 in-order cores, except with a larger system, placement becomes even more important. Vantage reduces misses and thereby increases performance, but it trails the D-NUCA schemes. IdealSPD and R-NUCA keep some data in the local bank, but misses from the local bank are expensive (going either to far-away cache banks or to off-chip main memory). IdealSPD and R-NUCA thus achieve similar speedups on average. Because of this, and since IdealSPD's extra cache capacity is increasingly unrealistic at larger system sizes, we only consider R-NUCA in the remaining evaluation. Meanwhile, Jigsaw gives each application enough capacity to fit its working set in nearby cache banks, and achieves the largest performance advantage of all schemes. Jigsaw's performance improvement is 2× the improvement of the best alternative.

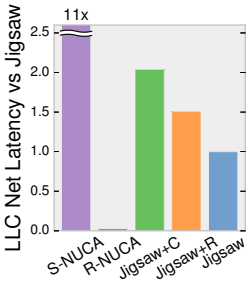
Additionally, we evaluate the effect of thread placement in Jigsaw by running Jigsaw with three schedulers: *random* (Jigsaw+R) and *clustered* (Jigsaw+C), as in Section 3.1, and Jigsaw's capacity-aware thread scheduler. As we will see, neither random nor clustered schedules are better in general—different mixes prefer one over the other.

Figure 3.23a shows that Jigsaw's capacity-aware thread scheduling significantly improves system performance. Jigsaw+R achieves 38% gmean weighted speedup and up to 64%, and Jigsaw+C achieves 34% gmean weighted speedup and up to 59%. Jigsaw+C shows near-pathological behavior, as different instances of the same benchmark are placed close by, introducing capacity contention and hurting latency when they get large VCs (as in Figure 3.1b). Jigsaw+R avoids this behavior and performs better, but Jigsaw avoids capacity contention much more effectively and attains higher speedups across all mixes. Jigsaw widely outperforms R-NUCA, as R-NUCA does not manage capacity efficiently in heterogeneous workload mixes (Section 3.1).

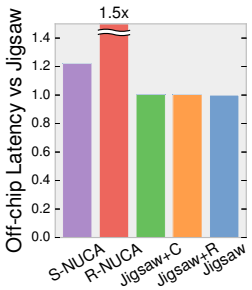
Figure 3.23 gives more insight into these differences. Figure 3.23b shows the average network latency incurred by LLC accesses across all mixes (Equation 3.2), normalized to Jigsaw, while Figure 3.23c compares off-chip latency (Equation 3.1). S-NUCA incurs 11× more on-chip network latency than Jigsaw on L2-LLC accesses, and 23% more off-chip latency. R-NUCA classifies most pages as private and maps them to the nearest bank, so its network latency for LLC accesses is negligible. However, the lack of capacity management degrades off-chip latency by 46% over Jigsaw. Jigsaw+C, Jigsaw+R and Jigsaw achieve similar off-chip latency, but Jigsaw+C and Jigsaw+R have 2× and 51% higher on-chip network



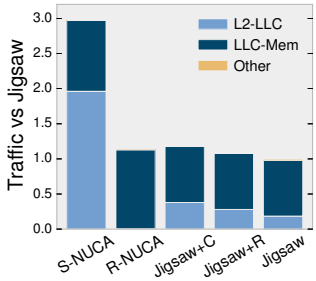
(a) Distribution of weighted speedups over S-NUCA.



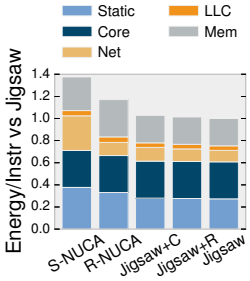
(b) Avg. on-chip latency on LLC accesses.



(c) Avg. off-chip latency (due to LLC misses).



(d) Breakdown of avg. on-chip network traffic per instruction.



(e) Breakdown of avg. energy per instruction.

Figure 3.23: Evaluation of S-NUCA, R-NUCA, and Jigsaw across 50 mixes of 64 SPEC CPU2006 apps on a 64-core CMP.

latency for LLC accesses than Jigsaw.

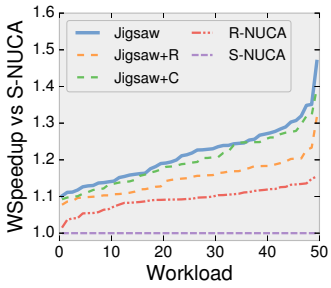
Figure 3.23d compares the network traffic of different schemes, measured in flits, and split in L2-to-LLC and LLC-to-memory traffic. S-NUCA incurs 3× more traffic than Jigsaw, most of it due to LLC accesses. For other schemes, traffic due to LLC misses dominates, because requests are interleaved across memory controllers and take several hops. We could combine these schemes with NUMA-aware techniques [34, 36, 100, 144, 147, 150] to further reduce this traffic. Though not explicitly optimizing for it, Jigsaw achieves the lowest traffic.

Because Jigsaw improves performance and reduces network and memory traffic, it reduces energy as well. Figure 3.23e shows the average energy per instruction of different organizations. Static energy (including chip and DRAM) decreases with higher performance, as each instruction takes fewer cycles. S-NUCA spends significant energy on network traversals, but other schemes make it a minor overhead; and R-NUCA is penalized by its more frequent memory accesses. Overall, Jigsaw+C, Jigsaw+R and Jigsaw reduce energy by 33%, 34% and 36% over S-NUCA, respectively.

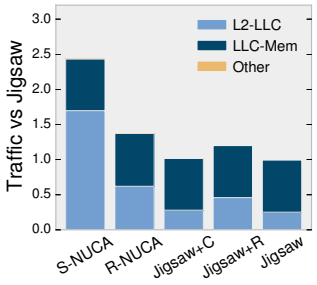
Jigsaw benefits apps with large cache-fitting footprints, such as `omnet`, `xalanc`, and `sphinx3`, the most. They require multi-bank VCs to work well, and benefit from lower access latencies. Apps with smaller footprints benefit from the lower contention, but their speedups are moderate.

3.6.2 Multithreaded mixes

Figure 3.24a shows the distribution of weighted speedups for 50 mixes of eight 8-thread SPEC OMP2012 applications (64 threads total) running on the 64-core CMP. Jigsaw achieves gmean weighted speedup of 21%. Jigsaw+R achieves 14%, Jigsaw+C achieves 19%, and R-NUCA achieves 9%. Trends are reversed:

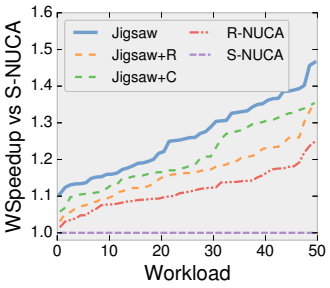


(a) Weighted speedups

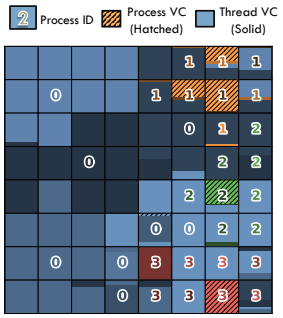


(b) Traffic breakdown

Figure 3.24: Weighted speedup distribution and traffic breakdown of 50 mixes of eight 8-thread SPEC OMP2012 apps on a 64-core CMP.



(a) Weighted speedups.



(b) Thread and data placement.

Figure 3.25: Weighted speedups for 50 mixes of four 8-thread SPEC OMP2012 apps (32 threads total) on a 64-core CMP, and case study with private-heavy and shared-heavy apps.

on multi-threaded benchmarks, Jigsaw works better with clustered thread placement than with random (S-NUCA and R-NUCA are still insensitive). Jigsaw sees smaller benefits over Jigsaw+C. Figure 3.24b shows that they get about the same network traffic, while others are noticeably worse.

Figure 3.25a shows the distribution of weighted speedups with under-committed system running mixes of four 8-thread applications. Jigsaw increases its advantage over Jigsaw+C, as it has more freedom to place threads. Jigsaw dynamically clusters or spreads each process as the context demands: shared-heavy processes are clustered, and private-heavy processes are spread out. Figure 3.25b illustrates this behavior by showing the thread and data placement of a specific mix, where one of the apps, `mgrid` (process P0), is private and intensive, and the others, `md` (P1), `ilbdc` (P2), and `nab` (P3) access mostly shared data. Jigsaw gives most capacity to `mgrid`, spreads its threads over the CMP, and tightly clusters P1–3 around their shared data.

From the results of Section 3.6.1 and Section 3.6.2, we can see that Jigsaw+R and Jigsaw+C help different types of programs, but no option is best in general. Yet by jointly placing threads and data, Jigsaw always provides the highest performance across all mixes. Thus, beyond improving performance, Jigsaw provides an important advantage in *guarding against pathological behavior incurred by fixed policies*.

Buckets	32	64	128	256	512	1024	2048	4096	8192
Lookahead	0.87	2.8	9.2	29	88	280	860	2,800	10,000
Peekahead	0.18	0.30	0.54	0.99	1.9	3.6	7.0	13	26
Speedup	4.8×	9.2×	17×	29×	48×	77×	125×	210×	380×
AllHulls %	87	90	92	95	96	98	99	99	99.5

Table 3.4: Performance of Peekahead and UCP's Lookahead [124]. Results given in M cycles per invocation.

Threads / Cores	16 / 16	16 / 64	64 / 64
Capacity allocation (M cycles)	0.30	0.30	1.20
Thread placement (M cycles)	0.29	0.80	3.44
Data placement (M cycles)	0.13	0.36	1.85
Total runtime (M cycles)	0.72	1.46	6.49
Overhead @ 25 ms (%)	0.09	0.05	0.20

Table 3.5: Jigsaw runtime analysis. Avg M cycles per invocation of each reconfiguration step, total runtime, and relative overhead.

3.7 Jigsaw analysis

Peekahead scaling: Table 3.4 shows the average core cycles required to partition the cache using both UCP Lookahead and Peekahead algorithms. To run these experiments, we use the miss curves from the 140 16-core mixes at different resolutions. Conventional Lookahead scales near quadratically (3.2× per 2× buckets), while Peekahead scales sublinearly (1.9× per 2× buckets). AllHulls dominates Peekahead's run-time, confirming linear asymptotic growth (Appendix A).

Reconfiguration overheads: Table 3.5 shows the CPU cycles spent, on average, in each of the steps of the reconfiguration procedure. Overheads are negligible: each reconfiguration consumes a mere 0.2% of system cycles. Our results use 64-way GMONs, where Peekahead is 9.2× faster than Lookahead. Peekahead's advantage increases quickly with resolution. Sparse GMON curves improve Peekahead's runtime, taking 1.2 M cycles at 64 cores instead of the 7.6 M cycles it would require with 512-way UMONs. Overall, Peekahead takes less than 0.1% of system cycles in reconfigurations at both 16 and 64 cores, imposing negligible overheads. Although thread and data placement have quadratic runtime, they are practical even at thousands of cores (1.2% projected overhead at 1024 cores).

Alternative thread and data placement schemes: We have considered more computationally expensive alternatives for thread and data placement. First, we explored using ILP to produce the best achievable data placement. We formulate the ILP problem by minimizing Equation 3.2 subject to the bank capacity and VC allocation constraints, and solve it in Gurobi [52]. ILP data placement improves weighted speedup by 0.5% over Jigsaw on 64-app mixes. However, Gurobi takes about 219 M cycles to solve 64-cores, far too long to be practical. We also formulated the joint thread and data placement ILP problem, but Gurobi takes at best tens of minutes to find the solution and frequently does not converge.

Since using ILP for thread placement is infeasible, we have implemented a simulated annealing [155] thread placer, which tries 5000 rounds of thread swaps to find a high-quality solution. This thread placer is only 0.6% better than Jigsaw on 64-app runs, and is too costly (6.3 billion cycles per run).

We also explored using METIS [76], a graph partitioning tool, to jointly place threads and data. We were unable to outperform Jigsaw. We observe that graph partitioning methods recursively divide

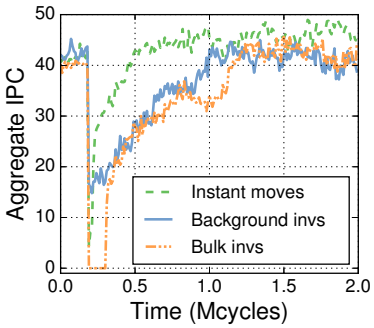


Figure 3.26: IPC throughput of a 64-core CMP with various data movement schemes during one reconfiguration.

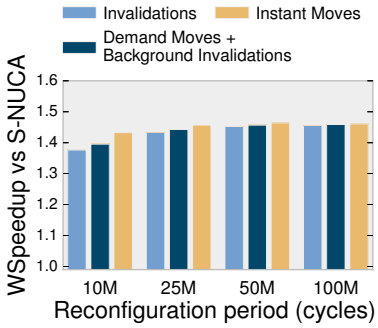


Figure 3.27: Weighted speedup of 64-app mixes for various data movement schemes vs. reconfiguration period.

threads and data into equal-sized partitions of the chip, splitting around the center of the chip first. Jigsaw, by contrast, often clusters one application around the center of the chip to minimize latency. In trace-driven runs, graph partitioning increases network latency by 2.5% over Jigsaw.

Geometric monitors: 1K-line, 64-way GMONs match the performance of 256-way UMONs. UMONs lose performance below 256 ways because of their poor resolution: 64-way UMONs degrade performance by 3% on 64-app mixes. In contrast, unrealistically large 16K-line, 1K-way UMONs are only 1.1% better than 64-way GMONs.

Reconfiguration schemes: We evaluate several LLC reconfiguration schemes: demand moves plus background invalidations (as in Jigsaw), bulk invalidations (pausing cores until all moved lines are invalidated, as in [12]), and idealized, instant moves. The main benefit of demand moves is avoiding global pauses, which take 114 K cycles on average, and up to 230 K cycles. While this is a 0.23% overhead if reconfigurations are performed every 50 M cycles (25 ms), many applications cannot tolerate such pauses [38, 114]. Figure 3.26 shows a trace of aggregate IPC across all 64 cores during one representative reconfiguration. This trace focuses on a small time interval to show how performance changes right after a reconfiguration, which happens at 200 K cycles. By serving lines with demand moves, Jigsaw prevents pauses and achieves smooth reconfigurations, while bulk invalidations pause the chip for 100 K cycles in this case. Besides pauses, bulk invalidations add misses and hurt performance. With 64 apps (Figure 3.23), misses are already frequent and per-thread capacity is scarce, so the average slowdown is 0.5%. Note that since SPEC CPU2006 is stable for long phases, these results may underestimate overheads for apps with more dynamic behavior. Figure 3.27 compares the weighted speedups of different schemes when reconfiguration intervals increase from 10 M cycles to 100 M cycles. Jigsaw outperforms bulk invalidations, though differences diminish as reconfiguration interval increases.

We also evaluated backing Jigsaw with a directory. We optimistically model an ideal, 0-cycle, fully-provisioned directory that causes no directory-induced invalidations. The directory enables migrations between lines in different banks after a reconfiguration, and avoids all bulk and page remapping invalidations. At 50 M cycles, directory-backed Jigsaw improves performance by 1.7% over bulk invalidations. We conclude that a directory-backed Jigsaw would not be beneficial. Even efficient implementations of this directory would require multiple megabytes and add significant latency, energy, and complexity. However, our workloads are fairly stable, we pin threads to cores, and do not overcommit the system. Other use cases (e.g., overcommitted systems) may change the tradeoffs.

Bank-partitioned NUCA: Jigsaw can be used without fine-grained partitioning (Section 3.3.8). With the parameters in Table 3.3 but 4 smaller banks per tile, Jigsaw achieves 36% gmean weighted speedup (up to 49%) over S-NUCA in 64-app mixes, vs. 46% gmean with partitioned banks. This difference is mainly due to coarser-grain capacity allocations, as Jigsaw allocates full banks in this case.

3.8 Summary

We have presented Jigsaw, a cache organization that addresses the scalability and interference issues of distributed on-chip caches. Jigsaw lets software define virtual caches of guaranteed size and placement and provides efficient mechanisms to monitor, reconfigure, and map data to virtual caches. We have developed an efficient, novel software runtime that uses these mechanisms to achieve both the latency-reduction benefits of NUCA techniques and the hit-maximization benefits of controlled capacity management. As a result, Jigsaw significantly outperforms state-of-the-art NUCA and partitioning techniques over a wide range of workloads. Jigsaw can potentially be used for a variety of other purposes, including maximizing fairness, implementing process priorities or tiered quality of service, or exposing virtual caches to user-level software to enable application-specific optimizations.

Talus: A Simple Way to Remove Cliffs in Cache Performance

4

CACHES can be a major headache for architects and programmers. Unlike most system components (e.g., frequency or memory bandwidth), caches often do not yield smooth, diminishing returns with additional resources (i.e., capacity). Instead, they frequently cause *performance cliffs*: thresholds where performance suddenly changes as data fits in the cache.

Cliffs occur, for example, with sequential accesses under LRU. Imagine an application that repeatedly scans a 32 MB array. With less than 32 MB of cache, LRU always evicts lines before they hit. But with 32 MB of cache, the array suddenly fits and every access hits. Hence going from 31 MB to 32 MB of cache suddenly increases hit rate from 0% to 100%. The SPEC CPU2006 benchmark `libquantum` has this behavior. Figure 4.1 shows `libquantum`'s *miss curve* under LRU (solid line), which plots misses per kilo-instruction (MPKI, y-axis) against cache size (MB, x-axis). `libquantum`'s miss curve under LRU is constant until 32 MB, when it suddenly drops to near zero. Cliffs also occur with other access patterns and policies.

Performance cliffs produce three serious problems. First, cliffs waste resources and degrade performance. Cache space consumed in a plateau does not help performance, but wastes energy and deprives other applications of that space. Second, cliffs cause unstable and unpredictable performance, since small fluctuations in effective cache capacity (e.g., due to differences in data layout) result in large swings in performance. This causes confusing performance bugs that are difficult to reproduce [32, 61, 111], and makes it hard to guarantee quality of service (QoS) [63, 77]. Third, as we saw in Chapter 3, cliffs greatly complicate cache management, because without convex miss curves optimal allocation is an NP-complete problem.

Two areas of prior work address performance cliffs in caches: high-performance replacement policies and cache partitioning. High-performance replacement policies have addressed many of the common pathologies of LRU [44, 69, 125, 159]. These policies achieve good performance and often avoid cliffs, but due to their empirical design they are difficult to predict and sometimes perform worse than LRU. The loss of predictability is especially unfortunate, since performance predictions are needed for efficient cache partitioning.

Partitioning handles cliffs by avoiding operating on plateaus. For example, faced with the miss curve in Figure 4.1, efficient partitioning algorithms will allocate either 32 MB or 0 MB, and nowhere in between. This ensures cache space is either used effectively (at 32 MB) or is freed for use by other applications (at 0 MB). Partitioning thus copes with cliffs, but still suffers from two problems: First, cliffs force "all-or-nothing" allocations that degrade fairness. Second, since optimal partitioning is NP-complete, partitioning algorithms are forced to use expensive or complex approximations [12, 110, 124].

Cliffs are not a necessary evil: optimal cache replacement (MIN [17]) does not suffer them. Rather, cliffs are evidence of the difficulty in using cache space effectively. Eliminating cliffs would be highly desirable, since it would put resources to good use, improve performance and fairness, increase stability,

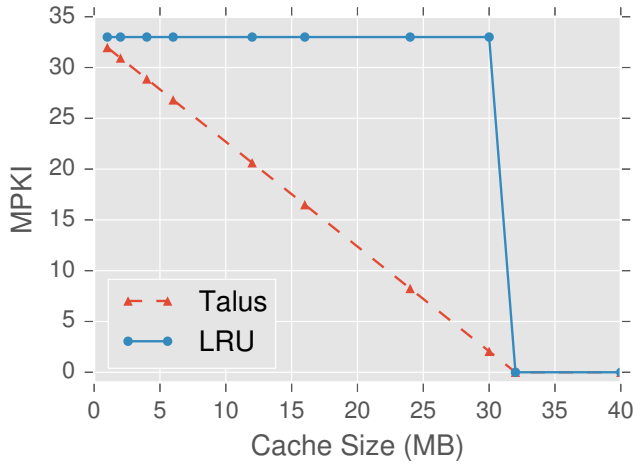


Figure 4.1: Performance of `libquantum` over cache sizes. LRU causes a performance cliff at 32 MB. Talus eliminates this cliff.

and—perhaps most importantly in the long term—make caches easier to reason about and simpler to manage.

We observe that performance cliffs are synonymous with *non-convex miss curves*. A convex miss curve has slope that shrinks with increasing capacity. By contrast, non-convex miss curves have regions of small slope (plateaus) followed by regions of larger slope (cliffs). Convexity means that additional capacity gives smooth and diminishing hit rate improvements.

We present Talus, a simple partitioning technique that ensures convex miss curves and thus eliminates performance cliffs in caches. Talus achieves convexity by partitioning *within* a single access stream, as opposed to prior work that partitions *among* competing access streams. Talus divides accesses between two *shadow partitions*, invisible to software, that emulate caches of a larger and smaller size. By choosing these sizes judiciously, Talus ensures convexity and improves performance. Our key insight is that only the miss curve is needed to do this. We make the following contributions:

- We present Talus, a simple method to remove performance cliffs in caches. Talus operates on miss curves, and works with any replacement policy whose miss curve is available.
- We prove Talus's convexity and generality under broad assumptions that are satisfied in practice.
- We design Talus to be *predictable*: its miss curve is trivially derived from the underlying policy's miss curve, making Talus easy to use in cache partitioning.
- We prove that optimal cache replacement is convex.
- We contrast Talus with bypassing, a common replacement technique. We derive the optimal bypassing scheme and show that Talus is superior, and discuss the implications of this result on the design of replacement policies.
- We develop a practical, low-overhead implementation of Talus that works with existing partitioning schemes and requires negligible hardware and software overheads.
- We evaluate Talus under simulation. Talus transforms LRU into a policy free of cliffs and competitive with state-of-the-art replacement policies [44, 69, 125]. More importantly, Talus's convexity simplifies cache partitioning algorithms, and automatically improves their performance and fairness.

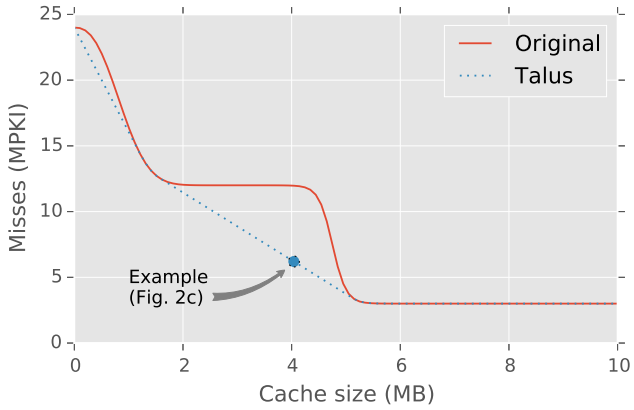


Figure 4.2: Example miss curve from an application with a cliff at 5 MB. Section 4.1 shows how Talus smooths this cliff at 4 MB.

In short, Talus is the first approach to offer *both* the benefits of high-performance cache replacement and the versatility of software control through cache partitioning.

Talus Example 4.1

This section illustrates how Talus works with a simple example. Talus uses partitioning to eliminate cache performance cliffs. Unlike prior work that partitions capacity among different cores, Talus partitions *within a single access stream*. It does so by splitting the cache (or, in partitioned caches, each software-visible partition) into two hidden *shadow partitions*. It then controls the size of these partitions and how accesses are distributed between them to achieve the desired performance.

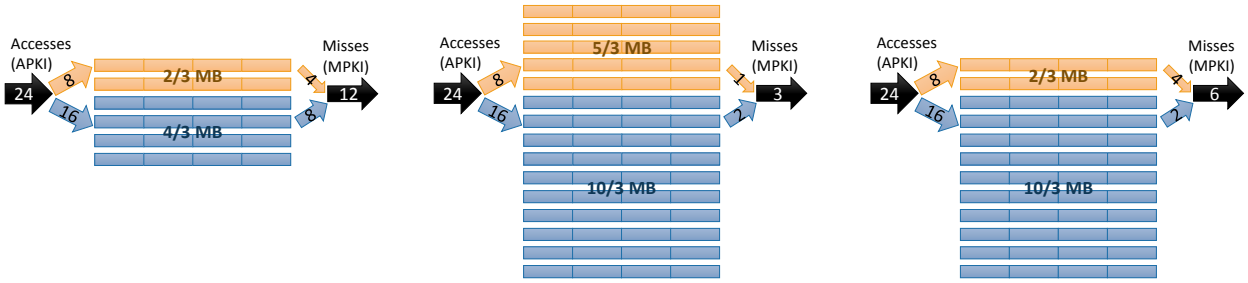
Talus computes the appropriate shadow partition configuration using miss curves; in this example we use the miss curve in Figure 4.2. Figure 4.2 is the miss curve of LRU on an application that accesses 2 MB of data at random, and an additional 3 MB sequentially. This results in a performance cliff around 5 MB, when MPKI suddenly drops from 12 to 3 once the application's data fits in the cache. Since the cache gets 12 MPKI at 2 MB, there is no benefit from additional capacity from 2 until 5 MB. Hence at 4 MB, half of the cache is essentially wasted since it could be left unused with no loss in performance.

Talus can eliminate this cliff and improve performance. Specifically, in this example Talus achieves 6 MPKI at 4 MB. The key insight is that LRU is inefficient at 4 MB, but LRU is efficient at 2 MB and 5 MB. Talus thus *makes part of the cache behave like a 2 MB cache, and the rest behave like a 5 MB cache.* As a result, the 4 MB cache behaves like a combination of efficient caches, and is itself efficient.

Significantly, Talus requires only the miss curve to ensure convexity. Talus is totally blind to the behavior of individual lines, and does *not* distinguish between lines' usage (e.g., sequential vs. random-access data), nor is Talus tied to particular replacement policies.

Talus traces out the *convex hull* [87] of the original miss curve, the dotted line in Figure 4.2. The convex hull of a curve is the smallest convex shape that contains the curve. Intuitively, it is the curve produced by stretching a taut rubber band across the curve from below. The convex hull thus connects points on the original curve, bridging its non-convex regions. The convex hull of a curve can be cheaply found with a single pass through the original curve using the three-coins algorithm [105].

Figure 4.3 shows how Talus works in this example. First, we describe how the cache behaves at 2 MB and 5 MB, and then show how Talus combines parts of these caches at 4 MB. Figure 4.3a shows the original 2 MB cache, split into parts by sets in a $1:2$ ratio. Figure 4.2 indicates this application accesses the cache at rate of 24 accesses per kilo-instruction (APKI). With a hashed cache, incoming



(a) Original cache at 2 MB. (b) Original cache at 5 MB. (c) Talus cache at 4 MB.

Figure 4.3: Performance of various caches for the miss curve in Figure 4.2. Figure 4.3a and Figure 4.3b show the original cache (i.e., without Talus), conceptually dividing each cache by sets, and dividing accesses evenly across sets. Figure 4.3c shows how Talus eliminates the performance cliff with a 4 MB cache by dividing the cache into partitions that *behave like the original* 2 MB (top) and 5 MB (bottom) caches. Talus achieves this by dividing accesses in *dis*-proportion to partition size.

accesses will be evenly split across sets, so accesses are also split at a $1:2$ ratio between the top and bottom sets. Specifically, the top third of sets receive $24/3 = 8$ APKI, and the bottom two thirds receive $24 \times 2/3 = 16$ APKI (left side of Figure 4.3a). Figure 4.2 further indicates a miss rate of 12 MPKI at 2 MB. Misses are also distributed approximately in proportion to accesses, so the top sets produce $12/3 = 4$ MPKI and the bottom sets $12 \times 2/3 = 8$ MPKI (right side of Figure 4.3a).

Figure 4.3b is similar, but for a 5 MB cache. This cache is also split by sets at a $1:2$ ratio. It achieves 3 MPKI, coming from the top sets at 1 MPKI and the bottom sets at 2 MPKI.

Finally, Figure 4.3c shows how Talus manages the 4 MB cache using set partitioning. The top sets behave like the top sets of the 2 MB cache (Figure 4.3a), and the bottom sets behave like the bottom sets of the 5 MB cache (Figure 4.3b). This is possible because Talus does *not* hash addresses evenly across sets. Instead, Talus distributes them in the same proportion as the original caches, at a $1:2$ ratio between top : bottom. Hence the top sets in Figure 4.3c operate *identically* to the top sets in Figure 4.3a. (They receive the same accesses, have the same number of sets and lines, etc.) In particular, the top sets in Figure 4.3c and Figure 4.3a have the same miss rate of 4 MPKI. Similarly, the bottom sets between Figure 4.3c and Figure 4.3b behave identically and have the same miss rate of 2 MPKI. Hence the total miss rate in Figure 4.3c is $4 + 2 = 6$ MPKI (instead of the original 12 MPKI). This value lies on the convex hull of the miss curve, as shown in Figure 4.2.

In this example, Talus partitions by set. However, with sufficient associativity, capacity is the dominant factor in cache performance, and cache organization or partitioning scheme are less important. So while this example uses set partitioning, Talus works with other schemes, e.g. way partitioning.

The only remaining question is how Talus chooses the partition sizes and sampling rates. 4 MB lies at a ratio of $2:1$ between the end points 2 MB and 5 MB. Although not obvious, the partitioning ratio should be the inverse, $1:2$. We derive and explain this in detail in the next section.

4.2 Talus: Convexity by Design

Figure 4.4 summarizes the parameters Talus controls. It shows a single application accessing a cache of size s, employing some replacement policy that yields a miss curve $m(s)$. For example, $m(1\text{ MB})$ gives the miss rate of a 1 MB cache. Talus divides the cache into two shadow partitions of sizes s_1 and s_2, where $s = s_1 + s_2$. Each shadow partition has its own miss curve, $m_1(s_1)$ and $m_2(s_2)$ respectively. Furthermore, Talus inserts a fraction ρ of the access stream into the first shadow partition, and the remaining $1 - \rho$

into the second.

We now show how, for any given size s, we can choose s_1, s_2, and ρ to achieve performance on the convex hull of the original miss curve. We develop the solution in three steps. First, we show how miss curves change when the access stream is divided among shadow partitions. Second, we show how to choose partition sizes to linearly interpolate cache performance between any two points of the original miss curve. Third, we show that by choosing these points appropriately, Talus traces the original miss curve's convex hull. Because these claims are broad, independent of particular applications or replacement policies, we present rigorous proofs.

General assumptions 4.2.1

Before we get started, we need to make some basic assumptions about cache performance and application behavior. These assumptions are often implicit in prior work, and as we shall see, are well-supported by experiments (Section 4.5).

Assumption 1. Miss curves are stable over time, and change slowly relative to the reconfiguration interval.

Talus uses the miss curve sampled in a given interval (e.g., 10 ms) to adjust its configuration for the next interval. If the access stream drastically changes across intervals, Talus's decisions may be incorrect. In practice, most applications have stable miss curves. Dynamic partitioning techniques and PDP [44] also make this assumption.

Assumption 2. For a given access stream, a partition's miss rate is a function of its size alone; other factors (e.g., associativity) are of secondary importance.

With reasonable associativity, size is the main factor that affects performance. Assumption 2 is inaccurate in some cases, e.g. way partitioning with few ways. Our implementation (Section 4.4) describes a simple way to satisfy this in practice, justified in our evaluation (Section 4.5). This assumption is made in prior partitioning work [12, 130] that uses UMONs to generate miss curves without using way partitioning.

This assumption also means that, although we prove results for Talus on an unpartitioned cache, our results also apply to individual partitions in a partitioned cache.

Assumption 3. Sampling an access stream produces a smaller, *statistically self-similar* access stream.

In large last-level caches, hits and misses are caused by accesses to a large collection of cache lines. No single line dominates accesses, as lower-level caches filter temporal locality. For example, if a program accesses a given line very frequently, that line will be cached in lower levels and will not produce last-level cache accesses. Thus, by pseudo-randomly sampling the access stream (i.e., by hashing addresses), we obtain an access stream with similar statistical properties to the full stream [81]. This holds in practice for large caches and good hash functions [26, 84, 129, 148]. Assumption 3 is extensively used, e.g. to produce miss curves cheaply with UMONs [124], dynamically switch replacement policies using set dueling [125], or accelerate trace-driven simulation [81]. Talus uses this property to reason about the behavior of the shadow partitions.

Miss curves of shadow partitions 4.2.2

Using the previous assumptions, we can easily derive the relationship between the partition's full miss curve, $m(s)$, and the miss curves of an individual shadow partition, $m'(s')$. Intuitively, a shadow

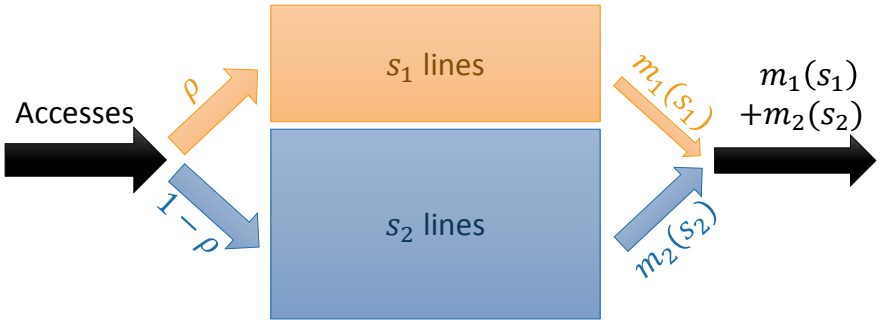


Figure 4.4: Talus divides cache space in two partitions of sizes s_1 and s_2, with miss rates $m_1(s_1)$ and $m_2(s_2)$, respectively. The first partition receives a fraction ρ of accesses.

partitioned cache with accesses split pseudo-randomly in proportion to partition size behaves the same as an unpartitioned cache (see Figure 4.3a and Figure 4.3b). In particular, misses are also split proportionally between shadow partitions. So if the partition gets a fraction $\rho = s'/s$ of accesses, its miss curve is $m'(s') = s'/s\ m(s) = \rho\ m\big(s'/\rho\big)$. This relation holds when accesses are distributed disproportionately as well:

Theorem 4. *Given an application and replacement policy yielding miss curve $m(s)$, pseudo-randomly sampling a fraction ρ of accesses yields miss curve $m'(s')$:*

$$m'(s') = \rho\ m\left(\frac{s'}{\rho}\right) \tag{4.1}$$

Proof. Since a sampled access stream is statistically indistinguishable from the full access stream (Assumption 3) and capacity determines miss rate (Assumption 2), it follows that misses are distributed evenly across capacity. (If it were otherwise, then equal-size partitions would exist with different miss rates, exposing either statistical non-uniformity in the access stream or sensitivity of miss rate to factors other than capacity.) Thus, following the discussion above, Equation 4.1 holds for proportionally-sampled caches.

Equation 4.1 holds in general because, by assumption, two partitions of the same size s' and sampling rate ρ must have the same miss rate. Hence a disproportionally-sampled partition's miss rate is equal to that of a partition of a larger, proportionally-sampled cache. This cache's size is s'/ρ, yielding Equation 4.1. □

4.2.3 Convexity

To produce convex cache performance, we trace the miss curve's *convex hull* (e.g., the dotted lines in Figure 4.2). Talus achieves this by linearly interpolating between points on the miss curve's convex hull, e.g. by interpolating between $\alpha = 2$ MB and $\beta = 5$ MB in Figure 4.2.

We interpolate cache performance using Theorem 4 by splitting the cache into two partitions, termed the α and β shadow partitions (Figure 4.4), since Talus configures them to behave like caches of size α and β. We then control the sizes of the shadow partitions, s_1 and s_2, and the sampling rate, ρ (into the α partition), to achieve the desired miss rate. First note from Theorem 4 that the miss rate of this system

is:

$$\begin{aligned} m_{\text{shadow}}(s) &= m_1(s_1) + m_2(s_2) \\ &= \rho\, m\left(\frac{s_1}{\rho}\right) + (1-\rho)\, m\left(\frac{s-s_1}{1-\rho}\right) \end{aligned} \tag{4.2}$$

Lemma 5. *Given an application and replacement policy yielding miss curve $m(\cdot)$, one can achieve miss rate at size s that linearly interpolates between any two points on on the curve, $m(\alpha)$ and $m(\beta)$, where $\alpha \le s < \beta$.*

Proof. To interpolate, we must anchor terms in m_{shadow} at $m(\alpha)$ and $m(\beta)$. We wish to do so for all m, in particular injective m, hence anchoring the first term at $m(\alpha)$ implies:

$$m(s_1/\rho) = m(\alpha) \Rightarrow s_1 = \rho\,\alpha \tag{4.3}$$

Anchoring the second term at $m(\beta)$ implies:

$$m\left(\frac{s-s_1}{1-\rho}\right) = m(\beta) \Rightarrow \rho = \frac{\beta - s}{\beta - \alpha} \tag{4.4}$$

Substituting these values into the above equation yields:

$$m_{\text{shadow}} = \frac{\beta - s}{\beta - \alpha}\, m(\alpha) + \frac{s - \alpha}{\beta - \alpha}\, m(\beta) \tag{4.5}$$

Hence as s varies from α to β, the system's miss rate varies proportionally between $m(\alpha)$ to $m(\beta)$ as desired. □

Theorem 6. *Given a replacement policy and application yielding miss curve $m(s)$, Talus produces a new replacement policy that traces the miss curve's convex hull.*

Proof. Set sizes α and β to be the neighboring points around s along m's convex hull and apply Lemma 5. That is, α is the largest cache size no greater than s where the original miss curve m and its convex hull coincide, and β is the smallest cache size larger than s where they coincide. □

Finally, we now apply Theorem 6 to Figure 4.3 to show how the partition sizes and sampling rate were derived for a $s = 4$ MB cache. We begin by choosing $\alpha = 2$ MB and $\beta = 5$ MB, as these are the neighboring points on the convex hull (Figure 4.2). ρ is the "normalized distance to β": 4 MB is two-thirds of the way to 5 MB from 2 MB, so one third remains and $\rho = 1/3$. s_1 is the partition size that will emulate a cache size of α with this sampling rate. By Theorem 4, $s_1 = \rho\ \alpha = 2/3$ MB. Finally, s_2 is the remaining cache space, $10/3$ MB. This size works because—by design—it emulates a cache of size β: the second partition's sampling rate is $1 - \rho = 2/3$, so by Theorem 4 it models a cache of size $s_2/(1-\rho) = 5$ MB.

Hence, a fraction $\rho = 1/3$ of accesses behave like a $\alpha = 2$ MB cache, and the rest behave like a $\beta = 5$ MB cache. This fraction changes as s moves between α and β, smoothly interpolating performance between points on the convex hull.

Theoretical Implications 4.3

A few interesting results follow from the previous section. All of the following should be taken to hold approximately in practice, since the assumptions only hold approximately.

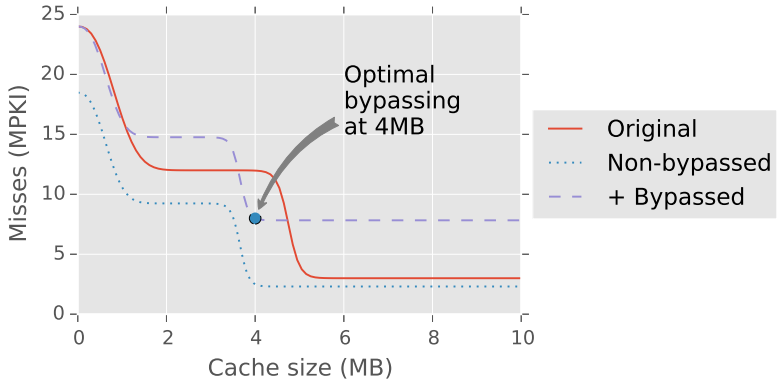


Figure 4.5: Optimal bypassing at 4 MB for the Figure 4.2 miss curve.

4.3.1 Predictability enables convexity

Theorem 6 says that so long as the miss curve is available, it is fairly simple to ensure convex performance by applying Talus. While Talus is general (we later show how it convexifies SRRIP, albeit using impractically large monitors), it is currently most practical with policies in the LRU family [132]. Talus motivates further development of high-performance policies for which the miss curve can be cheaply obtained.

4.3.2 Optimality implies convexity

Theorem 6 gives a simple proof that optimal replacement (MIN) is convex. We are not aware of a prior proof of this fact.

Corollary 7. *Optimal cache replacement is convex.*

Proof. By contradiction. If it were not convex, by Theorem 6 some shadow partitioning would exist that traced its convex hull (we need not compute the shadow partition sizes or sampling rates that trace it to guarantee its existence). This convex hull would achieve fewer misses than the optimal policy. □

4.3.3 Talus vs. Bypassing

As discussed in Chapter 2, high-performance replacement policies frequently use bypassing or low-priority insertions to avoid thrashing [44, 69, 125]. Theorem 4 explains why bypassing is an effective strategy. This theorem says that by bypassing some lines, non-bypassed lines behave like a larger cache (since $s/\rho \geq s$). So one can essentially get a larger cache for some lines at the price of missing on bypassed lines.

Figure 4.5 gives an example using Figure 4.2. Once again, the cache is 4 MB, which lies on a cliff that yields no improvement over 2 MB. At 5 MB, the working set fits and misses drop substantially. Bypassing improves performance in this instance. There are two effects: *(i)* Non-bypassed accesses (80% in this case) behave as in a 5 MB cache and have a low miss rate (dotted line). *(ii)* Bypassed accesses (20% in this case) increase the miss rate by the bypass rate (dashed line). The net result is a miss rate of roughly 8 MPKI—better than without bypassing, but worse than the 6 MPKI that Talus achieves.

In general, Talus is superior to bypassing: while bypassing matches Talus under some conditions, it cannot outperform the convex hull of the miss curve, and often does worse. For example, Figure 4.6 shows the miss curves that optimal bypassing (dashed line) and Talus (convex hull) achieve.

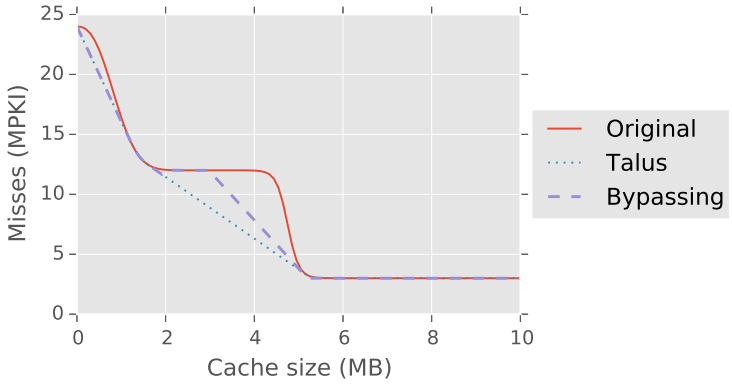


Figure 4.6: Comparison of Talus (convex hull) vs. optimal bypassing for the miss curve in Figure 4.2.

Corollary 8. *Bypassing on a miss curve $m(s)$ achieves performance no better than the miss curve's convex hull.*

Proof. Consider a cache that accepts a fraction ρ of accesses and bypasses $1-\rho$. This is equivalent to sampling a fraction ρ of accesses into a "partition of size s" (the full cache), and sending the remainder to a "partition of size zero" (bypassed). By Theorem 4, this yields a miss rate of:

$$m_{\text{bypass}}(s) = \rho\ m(s/\rho) + (1-\rho)\, m(0/(1-\rho)) \tag{4.6}$$

Letting $s_0 = s/\rho$, $m_{\text{bypass}}(s) = \rho\ m(s_0) + (1-\rho)m(0)$. Thus, m_{bypass} describes a line connecting points $(0, m(0))$ and $(s_0, m(s_0))$. Both points are in the original miss curve m, and by definition, m's convex hull contains all lines connecting any two points along m [87]. Thus, $m_{\text{bypass}}(s)$ either lies on the convex hull or above it (as in Figure 4.6). □

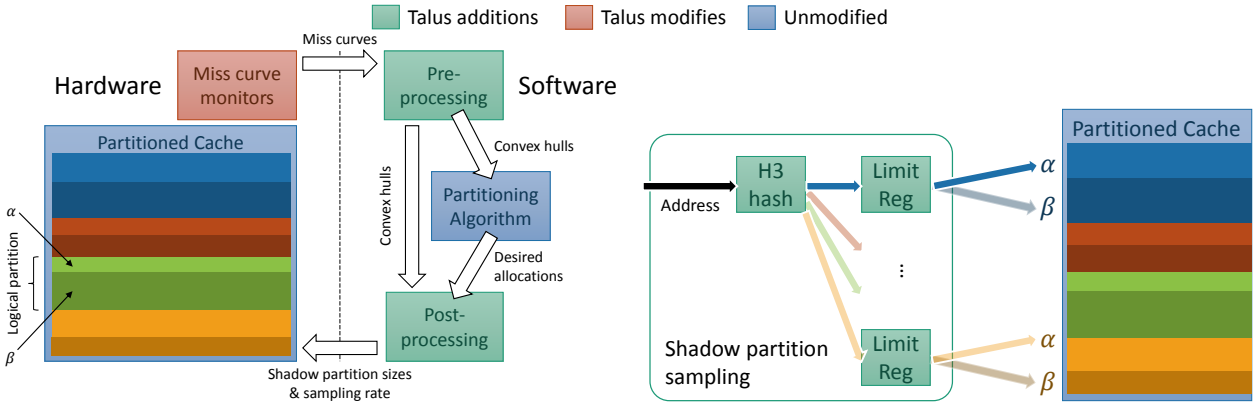
Because Talus traces the convex hull, it performs at least as well as optimal bypassing. However, this claim comes with a few important caveats: most replacement policies do not really bypass lines. Rather, they insert them at low priority. The distinction is sometimes relatively unimportant (e.g., in DIP [125]), but it can be significant. For example, with a high miss rate many lines can occupy the lowest priority in DRRIP, so a "bypassed" line may not even be the first evicted.

Additionally, unlike Assumption 3, most policies do not sample by address, but instead sample lines via other methods not strictly correlated to address. For example, DIP inserts lines at high priority every 32$^{\text{nd}}$ miss, regardless of address [125]. Assumption 3 and hence Theorem 4 are consequently less accurate for these policies.

Finally, this corollary also says that Talus will perform as well as optimal bypassing *on that policy*. It says nothing about the performance of Talus vs. bypassing for different baseline policies, although the intuition behind Corollary 8 is still useful to reason about performance in such cases. For example, PDP comes close to our description of optimal bypassing, so one might expect Talus on LRU to always outperform PDP. However, PDP evicts MRU among protected lines and sometimes outperforms Talus on LRU. Hence this result is a useful but inexact tool for comparing policies (see Section 4.5.3).

Practical Implementation 4.4

Talus builds upon existing partitioning solutions, with minor extensions in hardware and software. Our implementation is illustrated in Figure 4.7 and described in detail below.



(a) Overall system.

(b) Sampling into shadow partitions.

Figure 4.7: Talus implementation: pre- and post- processing steps in software, and simple additions and extensions to existing partition schemes in hardware.

4.4.1 Software

Talus wraps around the system's partitioning algorithm. Talus does not propose its own partitioning algorithm. Instead, Talus allows the system's partitioning algorithm—whatever it may be—to safely assume convexity, then realizes convex performance. This occurs in pre- and post-processing steps.

In the pre-processing step, Talus reads miss curves from hardware monitors (e.g., UMONs [124] or the SRRIP monitors described later) and computes their convex hulls. These convex hulls are passed to the system's partitioning algorithm, which no longer needs to concern itself with cliffs.

In the post-processing step, Talus consumes the partition sizes generated by the partitioning algorithm, and produces the appropriate shadow partition sizes and sampling rates. It does so using Theorem 6 and the convex hulls.

4.4.2 Hardware

Talus works with existing partitioning schemes, either coarse-grained (e.g., set [95, 127] or way [6, 28, 124] partitioning) or fine-grained (e.g., Vantage [130] or Futility Scaling [151]).

Talus extends these by *(i)* doubling the number of partitions in hardware, *(ii)* using two shadow partitions per "logical" (i.e., software-visible) partition, and *(iii)* adding one configurable sampling function to distribute accesses between shadow partitions. The sampling function consists of an 8-bit hash function (we use inexpensive H3 [26] hashing) and an 8-bit *limit register* per logical partition. Each incoming address is hashed. If the hash value lies below the limit register, the access is sent to the α partition. Otherwise it is sent to the β partition.

Deviation from assumptions: The theory relies on a few assumptions (Section 4.2.1) that are good approximations, but not exact. We find that our assumptions hold within a small margin of error, but this margin of error is large enough to cause problems if it is not accounted for. For example, although applications are stable between intervals, they do vary slightly, and sampling adds small deviations, even over many accesses. Unaccounted for, these deviations from theory can "push β up the performance cliff," seriously degrading performance.

We account for these deviations by adjusting the sampling rate ρ to build in a margin of safety into our implementation. (The effect of adjusting ρ by X% is to decrease α by X% and increase β by X%.) We have empirically determined an increase of 5% ensures convexity with little loss in performance.

Talus on way partitioning: Talus works on way partitioning, but way partitioning can somewhat egregiously violate Assumption 2. Specifically, way partitioning forces coarse-grain allocations that can significantly reduce associativity. This means that the coarsened shadow partition sizes will not match the math, and Talus will end up interpolating between the wrong points. We address this by recomputing the sampling rate from the final, coarsened allocations: $\rho = s_1/\alpha$.

Talus on Vantage: Vantage partitioning supports many partitions sized at line granularity, and is a good fit for Talus. However, Vantage does not partition a small fraction of the cache, known as the unmanaged region (10% of the cache in our evaluation). Vantage can give no guarantees on capacity for this region. Hence, at a total capacity of s, our Talus-on-Vantage implementation assumes a capacity of $s' = 0.9s$. Using Talus with Futility Scaling [151] would avoid this complication.

Inclusion: Talus can cause performance anomalies with inclusive LLCs. By sampling accesses into a small partition, Talus can back-invalidate lines that are frequently reused in lower cache levels and cause LLC accesses that aren't reflected in the sampled miss curve. We use non-inclusive caches to avoid this problem. Alternatively, one could sample the miss curve at lower cache levels (e.g., at the L2), or lower-bound the emulated sizes of the shadow partitions.

Monitoring 4.4.3

We gather LRU miss curves with utility monitors [124] (UMONs). We use 64-way, 1 K line UMONs to monitor the full LLC size. However, if this was the only information available, Talus would miss the behavior at larger sizes and be unable to trace the convex hull to these sizes. This matters for benchmarks with cliffs beyond the LLC size (e.g., `libquantum`).

Miss curve coverage: To address this, we add a second monitor per partition that samples accesses at a much lower rate. By Theorem 4, this models a proportionally larger cache size. With a sampling rate of $1:16$ of the conventional UMON, we model 4× LLC capacity using just 16 ways.

With 32-bit tags, monitoring requires 5 KB per core (4 KB for the original UMON plus 1 KB for the sampled one).

Other replacement policies: Talus can work with other policies, but needs their miss curve. UMONs rely on LRU's stack property [104] to sample the whole curve with one monitor, but high-performance policies do not obey the stack property. We evaluate Talus with SRRIP by using multiple monitor arrays, one per point on the miss curve. By sampling at different rates, each monitor models a cache of a different size. We use 64-point curves, which would require $64 \times 4 = 256$ KB of monitoring arrays per core, too large to be practical. CRUISE [68] takes a similar approach (using set sampling instead of external monitors) to find the misses with both half of the cache and the full cache, in effect producing 3-point miss curves; producing higher-resolution curves would be similarly expensive. Perhaps future implementations can reduce overheads by using fewer monitors and dynamically adapting sampling rates, but this is non-trivial and orthogonal to our purpose: demonstrating that Talus is agnostic to replacement policy.

Overhead analysis 4.4.4

Talus adds small hardware overheads. Doubling the number of partitions adds negligible overhead in way partitioning [6, 28]; in Vantage, it requires adding an extra bit to each tag (each tag has a partition

Cores	1 (**ST**) or 8 (**MP**), 2.4 GHz, Silvermont-like OOO [75]: 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
L1 caches	32 KB, 8-way set-associative, split D/I, 4-cycle latency
L2 caches	128 KB priv per-core, 8-way set-assoc, inclusive, 6-cycle
L3 cache	Shared, non-inclusive, 20-cycle; 32-way set-assoc with way partitioning or 4/52 zcache with Vantage; 1 MB/core
Coherence	MESI, 64 B lines, no silent drops; sequential consistency
Main mem	200 cycles, 12.8 GBps/channel, 1 (**ST**) or 2 (**MP**) channels

Table 4.1: Configuration of the simulated systems for single-threaded (**ST**) and multi-programmed (**MP**) experiments.

id) and adding 256 bits of state per partition [130]. Adaptive sampling requires an 8-bit hash function and an 8-bit limit register per partition. Monitors need 5 KB/core, of which only 1 KB is specific to Talus (to cover larger sizes). In the evaluated 8-core system with an 8 MB LLC, extra state adds up to 24.2 KB, a 0.3% overhead over the LLC size.

Talus adds negligible software overheads. First, Talus computes the convex hulls in linear time in size of the miss curve using the three-coins algorithm [105]. Second, it computes the shadow partition sizes by finding the values for α and β (logarithmic time in size of convex hull) and a few arithmetic operations for Theorem 6 (constant time). These overheads are a few thousand cycles per reconfiguration (every 10 ms), and in return, enable simpler convex optimization.

4.5 Evaluation

We evaluate Talus in a variety of settings, to demonstrate the following claims from Chapter 4:

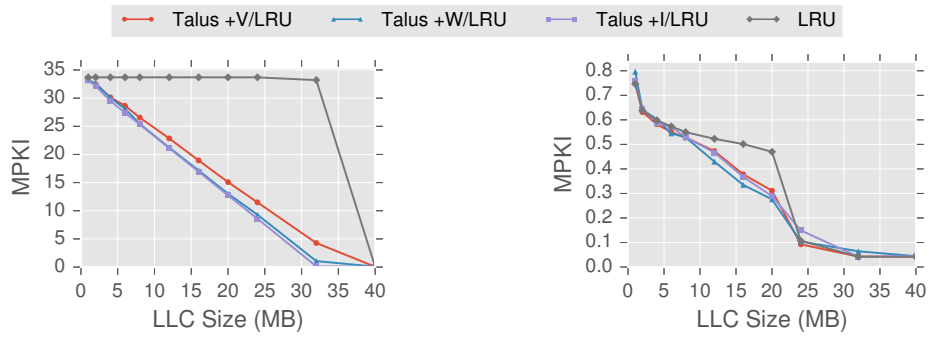
- Talus avoids performance cliffs, and is agnostic to replacement policy and partitioning scheme.
- Talus on LRU achieves performance competitive with high-performance policies and avoids pathologies.
- Talus is *both* predictable and convex, so simple convex optimization improves shared cache performance and fairness.

Talus is the first approach to combine high-performance cache replacement with the versatility of cache partitioning.

4.5.1 Methodology

We use zsim [131] to evaluate Talus in single-threaded and multi-programmed setups. We simulate systems with 1 and 8 OOO cores with parameters in Table 4.1.

We perform single-program runs to evaluate Talus's convexity and performance on LLCs of different sizes. We also simulate SRRIP, DRRIP, and PDP, which are implemented as proposed. DRRIP uses $M = 2$ bits and $\epsilon = 1/32$ [69]. For fairness, these policies use external auxiliary tag directories (DRRIP) and monitors (PDP) with the same overheads as Talus monitors (Section 4.4). We use SPEC CPU2006 apps, executed for 10 B instructions after fast-forwarding 10 B instructions.



(a) `libquantum` (b) `gobmk`

Figure 4.8: Talus on LRU replacement with various hardware policies: Vantage (V), way partitioning (W), and ideal (I). On all schemes, Talus closely traces LRU's convex hull.

We also perform multi-programmed runs to evaluate Talus's cache partitioning improvements (simplicity, performance, and fairness). We compare LRU, TA-DRRIP [69], and several cache partitioning schemes with and without Talus. Our methodology resembles prior work [66, 130]. We run random mixes of SPEC CPU2006 apps, with 1 B instructions per app after a 20 B fast-forward. We use a fixed-work methodology: all apps are kept running until all finish 1 B instructions, and we only consider the first 1 B instructions of each app to report aggregate performance. We report weighted speedup relative to LRU, $(\sum_i IPC_i/IPC_{i,LRU})/N_{apps}$, which accounts for throughput and fairness; and harmonic speedup, $1/\sum_i(IPC_{i,LRU}/IPC_i)$, which emphasizes fairness [124, 136]. We repeat runs to achieve 95% confidence intervals $\leq$1%.

Talus yields convex miss curves 4.5.2

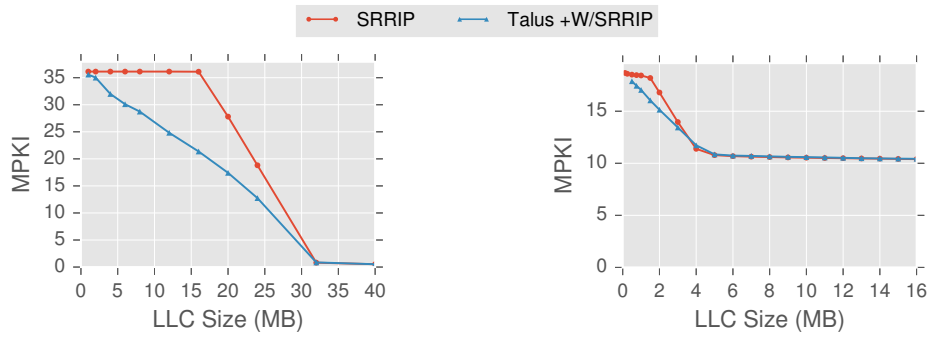
We first evaluate the miss curves (MPKI vs. LLC size) Talus produces in different settings and demonstrate its convexity.

Talus is agnostic to partitioning scheme: Figure 4.8 shows Talus with LRU on two representative SPEC CPU2006 apps, `libquantum` and `gobmk` (other apps behave similarly). We evaluate Talus on three partitioning schemes: Vantage (Talus+V/LRU), way partitioning (Talus+W/LRU), and idealized partitioning on a fully-associative cache (Talus+I/LRU).

In all schemes, Talus avoids LRU's performance cliffs, yields smooth, diminishing returns, and closely traces LRU's convex hull. Because Vantage partitions only 90% of the cache, Talus+V/LRU's performance lies slightly above the convex hull. This is particularly evident on `libquantum`. Likewise, variation between reconfiguration intervals causes small deviations from the convex hull, especially evident on `gobmk`.

One might be concerned that small non-convexities cause problems when using convex optimization with Talus. However, recall from Figure 4.7 that Talus produces convex miss curves from LRU's in a pre-processing step, *not* from measuring Talus itself. The distinction is subtle but important: these curves are guaranteed to be convex; Figure 4.8 shows that Talus achieves performance very close to these curves in practice. Thus, the system's partitioning algorithm can assume convexity with confidence that Talus will achieve the promised performance.

Talus is agnostic to replacement policy: Figure 4.9 shows Talus with SRRIP replacement and way partitioning (Talus+W/SRRIP) on `libquantum` and `mcf`. The purpose of this experiment is to demonstrate



(a) `libquantum` (b) `mcf`

Figure 4.9: Talus on SRRIP with Vantage partitioning. Talus smooths the cliff with SRRIP on `libquantum` and `mcf`.

that Talus is agnostic to replacement policy; SRRIP is hard to predict, so we use impractical 64-point monitors (Section 4.4.3). As with LRU, Talus eliminates the cliff in `libquantum` and `lbm` and traces SRRIP's convex hull. Finally, note that DRRIP achieves similar performance to Talus+V/SRRIP with lower monitoring overheads, but DRRIP is unpredictable and not guaranteed to be convex in general, so it lacks the partitioning benefits of Talus on LRU.

Talus is agnostic to prefetching: We have reproduced these results using L2 adaptive stream prefetchers validated against Westmere [131]. Prefetching changes miss curves somewhat, but does not affect any of the assumptions that Talus relies on.

Talus is agnostic to multi-threading: We have run Talus with LRU on the multi-threaded benchmark suite SPEC OMP2012, where it achieves convex miss curves similar to Figure 4.8 and Figure 4.10. With non-inclusive caches and directories to track the L2 contents, shared data is served primarily from other cores, rather than through the LLC. Hence, Talus's assumptions hold, and it works equally well on multi-threaded apps.

We have only observed significant non-convexities in Talus on benchmarks with exceptionally low memory intensity (e.g., on `povray` and `tonto`, which have <0.1 L2 MPKI). These benchmarks do not access the LLC frequently enough to yield statistically uniform access patterns across shadow partitions. However, since their memory intensity is so low, non-convexities on such applications are inconsequential.

In the remainder of the evaluation, we use Talus with Vantage partitioning and LRU replacement (Talus+V/LRU).

4.5.3 Talus with LRU performs well on single programs

MPKI: Figure 4.10 shows MPKI curves from 128 KB to 16 MB for six SPEC CPU2006 benchmarks. We compare Talus+V/LRU with several high-performance policies: SRRIP, DRRIP, and PDP, and include LRU for reference.

Generally, Talus+V/LRU performs similarly to these policies. On many benchmarks (not shown), all policies perform identically to LRU, and Talus+V/LRU matches this performance. On `perlbench`, `libquantum`, `lbm`, and `xalancbmk` (and others not shown), Talus+V/LRU outperforms LRU and matches the high-performance replacement policies.

On `perlbench`, `cactusADM`, `libquantum`, and `lbm`, Talus+V/LRU outperforms one or more high-performance policies. `perlbench` and `cactusADM` are examples where PDP performs poorly. These

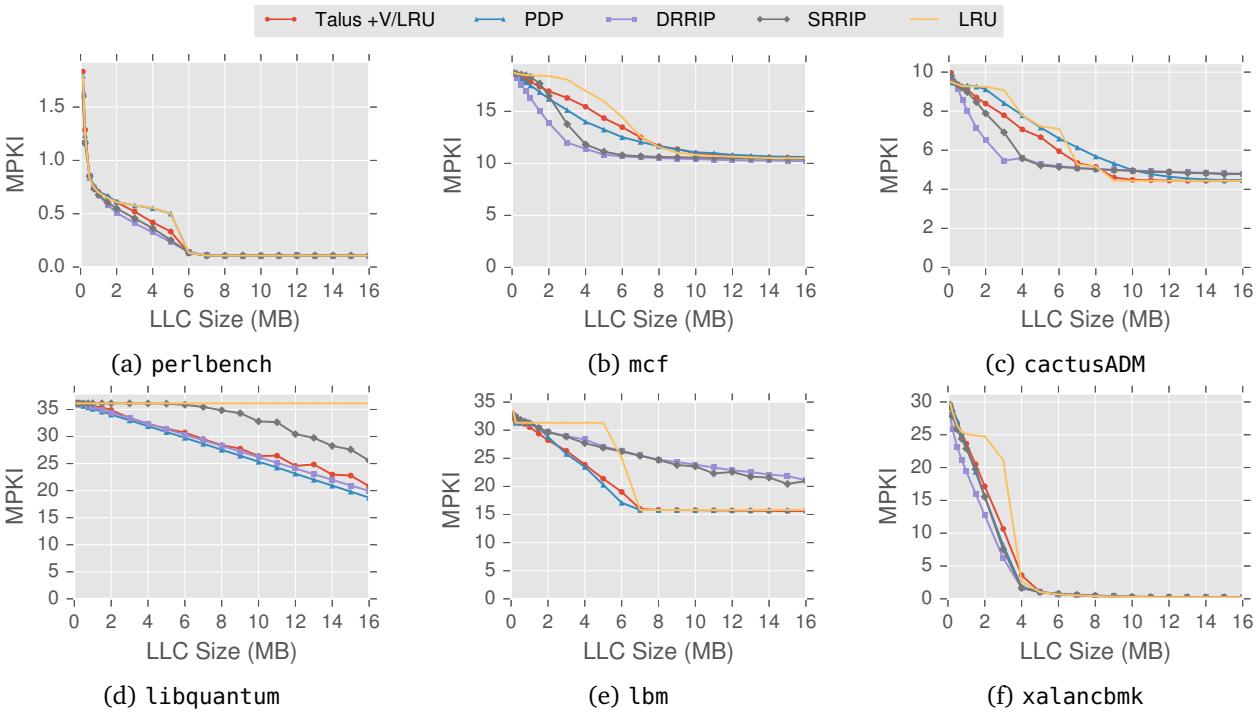


Figure 4.10: Misses per kilo-instruction (MPKI) of Talus (+V/LRU) and high-performance replacement policies on representative SPEC CPU2006 benchmarks from 128 KB to 16 MB. Talus achieves good performance and avoids cliffs.

benchmarks have a cliff following a convex region in the LRU miss curve. Since PDP is based on bypassing, it cannot achieve convex performance on such applications, and Talus+V/LRU outperforms it (Section 4.3.3). `lbm` is an example where RRIP policies perform poorly (`Gems` also shows this behavior). We found that DRRIP is largely convex on SPEC CPU2006, but this comes at the price of occasionally under-performing LRU and sacrificing its predictability.

Since Talus+V/LRU works by simply avoiding cliffs in LRU, it never degrades performance over LRU, and outperforms RRIP on such benchmarks. However, this is a limitation as well as a strength: Talus's performance is limited by the performance of the replacement policy it operates on. On some benchmarks this is significant. For instance, `mcf` and `cactusADM` benefit tremendously from retaining lines that have been reused. LRU does not track this information, so it is not available to Talus+V/LRU. Policies that capture reused lines (RRIP especially, somewhat for PDP) will outperform Talus+V/LRU on such benchmarks. This limitation is *not* inherent to Talus, though—any predictable replacement policy that captured such information would allow Talus to exploit it.

IPC: Figure 4.11 shows IPC vs. LRU for the policies shown in Figure 4.10 at 1 MB and 8 MB. These sizes correspond to the per-core LLC capacity, and the LLC capacity of the 8-core CMP. (Beyond 8 MB differences among policies steadily diminish.) The left side of Figure 4.11 shows the IPC vs. LRU for every benchmark that is affected at least 1%, and the right side shows the gmean speedup across all 29 SPEC CPU2006 benchmarks.

Talus+V/LRU improves performance whenever other policies do, and gives similar benefits, except for a few cases where DRRIP outperforms all others at 1 MB. Significantly, Talus+V/LRU never causes large degradations, while at 8 MB other policies all cause significant degradations for some benchmark.

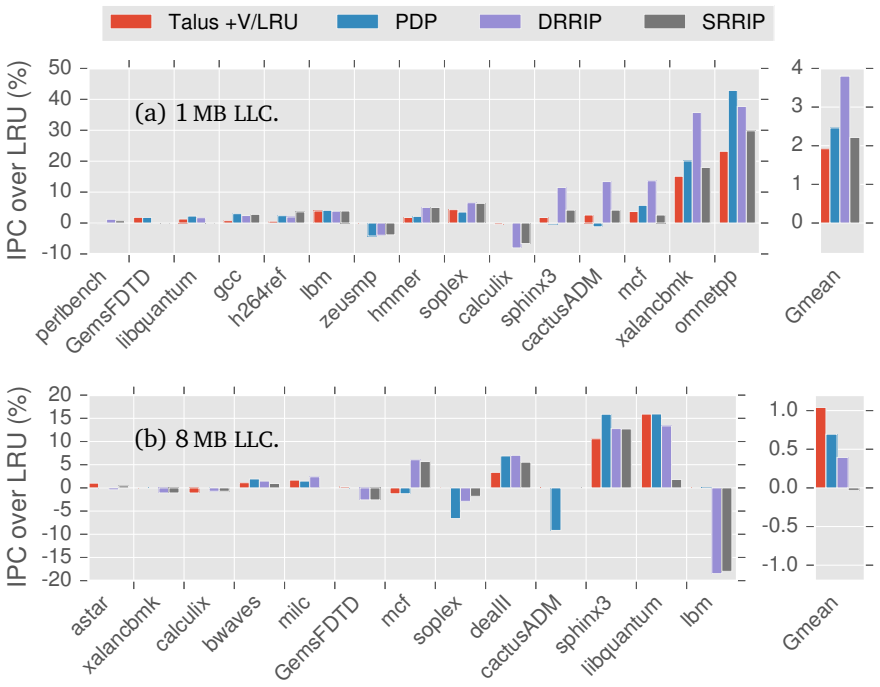


Figure 4.11: IPC improvement over LRU of Talus (+V/LRU) and high-performance replacement policies over all SPEC CPU2006 benchmarks. (Only apps with >1% IPC change shown.) Talus achieves competitive performance and avoids degradations.

Averaging over SPEC CPU2006, Talus+V/LRU gives IPC benefits comparable to other policies. At 1 MB, it is similar to PDP and SRRIP, but trails DRRIP (1.9% vs. 2.4%, 2.2%, 3.8%, respectively); at 8 MB, it outperforms PDP, SRRIP, and DRRIP (1.0% vs. 0.69%, -0.03%, 0.39%).

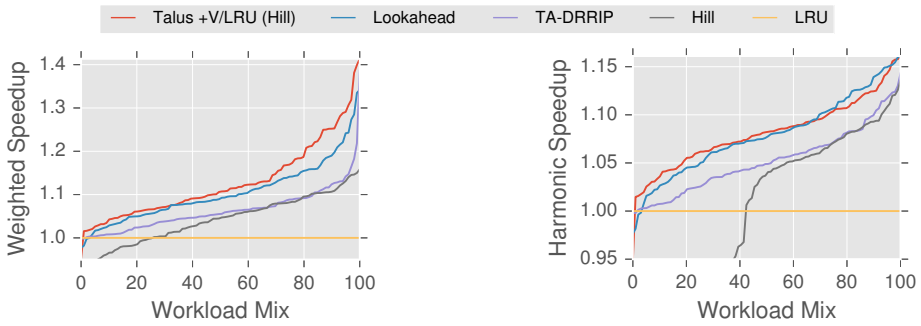
In summary, Talus avoids LRU's inefficiencies and approaches state-of-the-art replacement policies, without adopting an empirical design that sacrifices LRU's predictability.

4.5.4 Talus simplifies cache management and improves the performance and fairness of managed LLCs

We now evaluate Talus+V/LRU on an 8-core CMP with a shared LLC. These experiments demonstrate the qualitative benefits of Talus: Talus is *both* predictable and convex, so simple partitioning algorithms produce excellent outcomes. We apply partitioning to achieve two different goals, performance and fairness, demonstrating the benefits of *software control of the cache*. Hardware-only policies (e.g., TA-DRRIP) are fixed at design time, and cannot adapt to the changing needs of general-purpose systems. Both of Talus's properties are essential: predictability is required to partition effectively, and convexity is required for simple algorithms to work well.

Performance: Figure 4.12 shows the weighted (left) and harmonic (right) speedups over unpartitioned LRU for 100 random mixes of the 18 most memory intensive SPEC CPU2006 apps. Figure 4.12 is a *quantile plot*, showing the distribution of speedups by sorting results for each mix from left to right along the x-axis.

We compare Talus+V/LRU, partitioned LRU, and TA-DRRIP. Moreover, we compare two partitioning algorithms: hill climbing and Lookahead [124]. Hill climbing is a simple algorithm that allocates cache capacity incrementally, giving capacity to whichever partition benefits most from the next little bit. Its



(a) Weighted speedup.

(b) Harmonic speedup.

Figure 4.12: Weighted and harmonic speedup over LRU of Talus (+V/LRU), partitioned LRU (using both Lookahead [124] and hill climbing), and thread-aware DRRIP.

implementation is a trivial linear-time for-loop. Lookahead is a quadratic heuristic that approximates the NP-complete solution of the non-convex optimization problem. Equivalent algorithms achieve linear-time common case performance, at the cost of extra complexity [12]. Talus+V/LRU uses hill climbing, and partitioned LRU is shown for both Lookahead and hill climbing.

Weighted speedups over LRU are up to 41%/gmean of 12.5% for hill climbing on Talus+V/LRU, 34%/10.2% for Lookahead on LRU, 39%/6.3% for TA-DRRIP, and 16%/3.8% for hill climbing on LRU. Thus Talus+V/LRU achieves the best performance of all schemes with simple hill climbing, whereas partitioned LRU sees little benefit with hill climbing. This is caused by performance cliffs: with non-convexity, hill climbing gets stuck in local optima far worse than the best allocation. In contrast, since Talus+V/LRU is convex, hill climbing is optimal (in terms of LLC hit rate).

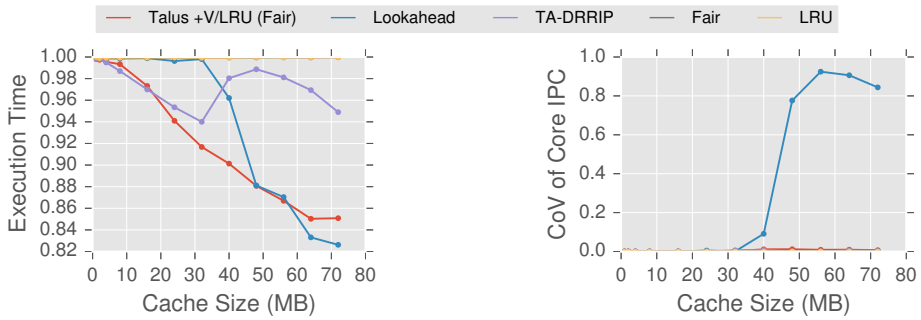
Lookahead avoids this pitfall, but must make "all-or-nothing" allocations: to avoid a cliff, it must allocate to sizes past the cliff. This means that if the most efficient allocation lies beyond the cache capacity, Lookahead must ignore it. Talus+V/LRU, in contrast, can allocate at intermediate sizes (e.g., along the LRU plateau) and avoid this problem. Hence Talus+V/LRU outperforms Lookahead by 2%. Finally, TA-DRRIP under-performs partitioning, trailing Talus+V/LRU by 5.6% gmean over all mixes.

Talus+V/LRU also improves fairness, even while optimizing for performance, illustrated by harmonic speedups in Figure 4.12(b). Harmonic speedups over LRU are gmean 8.0% for hill climbing on Talus+V/LRU, 7.8% for Lookahead on LRU, 5.2% for TA-DRRIP, and -1.8% for hill climbing on LRU. Talus+V/LRU modestly outperforms Lookahead in harmonic speedup. In contrast, TA-DRRIP's harmonic speedup is well below partitioned schemes, while hill climbing on LRU actually degrades performance over an unpartitioned cache.

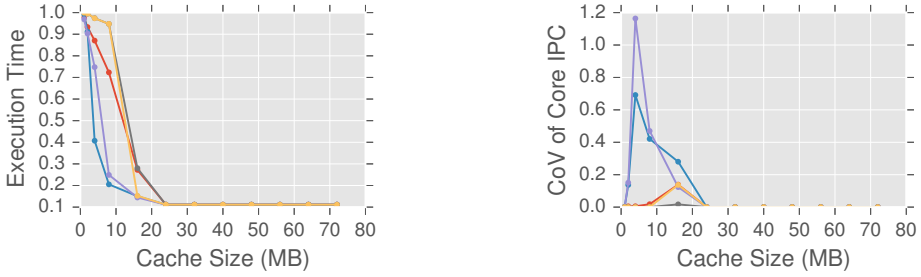
The only scheme that is competitive with naïve hill climbing on Talus+V/LRU is Lookahead, which is expensive, or alternatives like Peekahead (Section 3.3), which are more complex. This shows that, by ensuring convexity, *Talus makes high-quality partitioning simple and cheap*.

Fairness: Figure 4.13 shows three case studies of eight copies of benchmarks on the 8-core system with LLC sizes from 1 MB to 72 MB. This system represents a homogeneous application in a setting where fairness is paramount. Figure 4.13 shows three benchmarks: `libquantum`, `omnetpp`, and `xalancbmk`, all of which have a performance cliff under LRU. We compare the execution time of a fixed amount of work per thread for various schemes (lower is better) vs. unpartitioned LRU with a 1 MB LLC. As before, we compare Talus+V/LRU against partitioned LRU and TA-DRRIP. In this case, we employ fair partitioning (i.e., equal allocations) for LRU and Talus+V/LRU, and Lookahead on LRU.

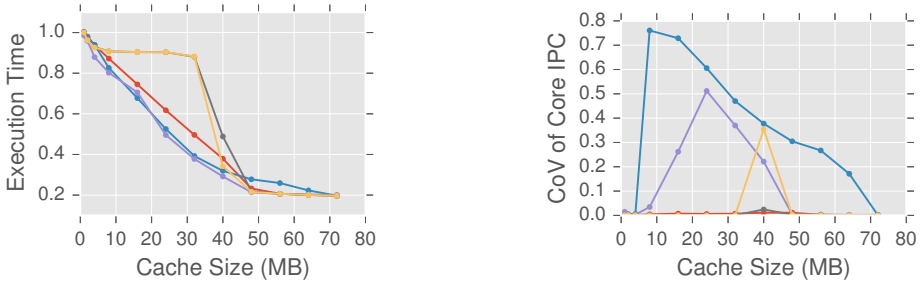
`libquantum` has a large performance cliff at 32 MB. With eight copies of `libquantum`, unpartitioned



(a) `libquantum`.



(b) `omnetpp`.



(c) `xalancbmk`.

Figure 4.13: Fairness case studies: 8 copies of a benchmark with fair partitioning (Talus+V/LRU and LRU) and Lookahead (LRU). We plot performance as execution time vs. unpartitioned LRU (left); and (un-)fairness as the coefficient of variation of per-core IPC (right). In both cases, lower is better.

LRU does not improve performance even at 72 MB, as the applications interfere in the cache (Figure 4.13a, left). Likewise, fair partitioning with LRU provides no speedup, since each application requires 32 MB to perform well. Lookahead improves performance, but at the cost of fairness: after 32 MB, Lookahead gives the entire cache to one application, boosting its performance without benefiting the others. This leads to a large gap between the IPCs of different cores, shown by the coefficient of variation of core IPC (i.e., the standard deviation of IPCs divided by the mean IPC; Figure 4.13a, right). TA-DRRIP thrashes in the cache, and never settles on any single core to prioritize.

In contrast, Talus+V/LRU gives steady gains with increasing LLC size, achieving the best performance except when Lookahead briefly outperforms it (due to Vantage's unmanaged region; see Section 4.4). Unlike Lookahead, fair partitioning speeds up each copy equally, so Talus+V/LRU does not sacrifice fairness to improve performance.

This pattern repeats for `omnetpp` (Figure 4.13b) and `xalancbmk` (Figure 4.13c), which have cliffs at 2 MB and 6 MB, respectively. In non-convex regions, Lookahead improves performance, but at the cost of grossly unfair allocations. Unlike in `libquantum`, TA-DRRIP improves performance in these benchmarks,

but also at the cost of fairness. Note that unpartitioned LRU is occasionally unfair around the performance cliff. This occurs when one copy of the benchmark enters a vicious cycle: it misses in the cache, slows down, is unable to maintain its working set in the cache, and thus misses further. Hence most cores speed up, except for one or a few unlucky cores that cannot maintain their working set, resulting in unfair performance. Over all sizes, Talus+V/LRU gives steady performance gains and never significantly sacrifices fairness.

These results hold for other benchmarks. Most are barely affected by partitioning scheme and perform similarly to LRU. Across all benchmarks, Talus+V/LRU with fair partitioning has at most 2% coefficient of variation in core IPC and averages 0.3%, whereas Lookahead/TA-DRRIP respectively have up to 85%/51% and average 10%/5%. While Lookahead or TA-DRRIP reduce execution time in some cases, they sacrifice fairness (e.g., `omnetpp` in Figure 4.13b). Imbalanced partitioning [116] time-multiplexes unfair allocations to improve fairness, but Talus offers a simpler alternative: naïve, equal allocations.

Hence, as when optimizing performance, Talus makes high-quality partitioning simple and cheap. We have shown that Talus, using much simpler algorithms, improves shared cache performance and fairness over prior partitioning approaches and thread-aware replacement policies. Both of Talus's qualitative contributions—predictability and convexity—were necessary to achieve these outcomes: Predictability is needed to make informed decisions, and convexity is needed to make simple algorithms work well.

Summary 4.6

Convexity is important in caches to avoid cliffs and thereby improve cache performance and simplify cache management. We have presented Talus, a novel application of partitioning that ensures convexity. We have proven Talus's convexity rigorously under broadly applicable assumptions, and discussed the implications of this analytical foundation for designing replacement policies. Talus improves performance over LRU and, more importantly, allows for simple, efficient, and optimal allocation of cache capacity. Talus is the first approach to combine the benefits of high-performance cache replacement with the versatility of cache partitioning.

EVA: A Practical Cache Replacement Policy using Planning Theory

5

LAST-LEVEL caches consume significant resources, typically over 50% of chip area [89], so it is crucial that they are managed efficiently. Prior work has approached cache replacement from both theoretical and practical standpoints. Unfortunately, there is *a large gap between theory and practice*.

From a theoretical standpoint, the optimal replacement policy is Belady's MIN [17, 104], which evicts the candidate referenced furthest in the future. But MIN's requirement for perfect knowledge of the future makes it impractical. In practice, policies must cope with uncertainty, never knowing exactly when candidates will be referenced. Most policies emulate MIN by evicting the candidate with the largest *predicted* time until reference.

Theoretically-grounded policies account for uncertainty by using a simplified, statistical model of the reference stream in which the optimal policy can be found analytically. The key challenge is defining a model that faithfully captures the important tradeoffs, yet is simple enough to analyze. For instance, prior work uses the *independent reference model* (IRM), which assumes that candidates are referenced independently with static probabilities. The IRM-optimal policy is to evict the candidate with the lowest reference probability [4]. Though useful in other areas (e.g., web caches [7]), IRM is inadequate for processor caches because it assumes reference probabilities do not change over time.

Instead, replacement policies for processor caches are designed empirically, using heuristics based on observations of common-case access patterns [44, 69, 73, 79, 125, 141, 159]. We observe that, unlike the IRM, these policies do not assume static reference probabilities. Instead, they *exploit dynamic behavior* through various aging mechanisms. While often effective, high-performance policies lack a theoretical foundation. Each policy performs well on particular programs, yet no policy dominates overall, suggesting that these policies are not making the best use of available information.

In this chapter, we seek to *bridge theory and practice* in cache replacement with two key contributions. First, we describe a simple reference model that captures the important tradeoffs of dynamic cache behavior. We use planning theory to show that the correct approach is to replace candidates by their *economic value added* (EVA); i.e., how many more hits one expects from each candidate vs. the average. Second, we design a practical implementation of this policy and show it outperforms existing policies.

Because EVA improves replacement decisions, it needs less space than prior policies to achieve the same performance, and thus saves area. Figure 5.1 shows the cache area breakdown for different replacement policies at iso-performance, including all replacement overheads. Specifically, each policy achieves the same average misses per kilo-instruction (MPKI) on SPEC CPU2006 as a 4 MB cache using random replacement (see Section 5.6.1 for details). EVA saves 10% area over the best practical alternative, SHiP [159]. Prior policies have focused on reducing replacement overheads [44, 69, 73, 159], but Figure 5.1 shows that these constitute roughly 1% of cache area. Far more area can be saved by making better use of the other 99% of cache area—i.e., by improving replacement decisions—even if doing so increases

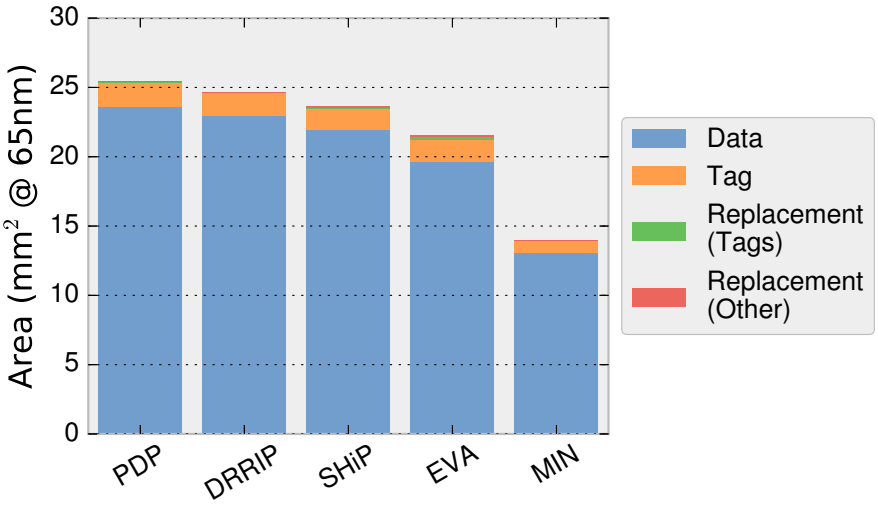


Figure 5.1: Cache area breakdown for different replacement policies at the same performance. Because EVA improves performance, it saves area over prior policies. Replacement overheads are negligible for all policies.

overheads. Indeed, Figure 5.1 shows that MIN could consume a third of the cache and yet still save area.

However, in order for larger overheads to be a good investment, we need a policy that makes good use of them. EVA's analytical design achieves this, and EVA benefits more than prior policies from additional replacement state. (Figure 5.1 compares the most area-efficient configuration of each policy.) Hence, despite modestly increasing replacement overheads, EVA yields substantial area savings over prior policies.

Contributions: We make the following contributions:

- We formalize a simple memory reference model, the *iid reuse distance model*, that captures the important dynamic tradeoffs of real reference streams within practical constraints (Section 5.1).
- We show that predicting time until reference, the conventional approach to replacement under, is a flawed strategy (Section 5.2). We discuss the two main tradeoffs in cache replacement: hit probability and cache space, and describe how EVA reconciles them with a single intuitive metric (Section 5.3).
- We describe a practical implementation of EVA, which we have synthesized in a 65 nm commercial process. EVA adds less than 0.5% area overhead on a 1 MB cache, plus 1% overhead in tags vs. 2-bit SHiP [159]. Unlike prior policies, EVA does not require set sampling or auxiliary tagged monitors (Section 5.5).
- We evaluate EVA against prior high-performance policies on SPEC CPU2006 and OMP2012 over many cache sizes (Section 5.6). EVA reduces LLC misses over existing policies at equal area, closing 57% of the gap from random replacement to MIN vs. 47% for SHiP [159], 41% for DRRIP [69], and 42% for PDP [44]. *Fewer misses translate into large area savings—gmean 8% over SHiP across LLC sizes (Figure 5.1.)*
- Finally, we formulate cache replacement within the iid reuse distance model as a Markov decision process (MDP), and show that EVA follows from MDP theory, though an exact solution is impractical (Section 5.4).

Overall, our results show that formalizing cache replacement yields practical benefits. Beyond our particular design, we expect replacement by economic value added (EVA) to be useful in the design of future high-performance policies.

The iid reuse distance model 5.1

This section introduces the iid reuse distance model, a memory reference model intended to capture the main features and constraints of real LLC access streams. Refer to Section 2.2.1 for background in replacement theory.

Motivation for a new model 5.1.1

The independent reference model assumes that replacement candidates have distinct reference probabilities that are constant over time. While this model may be appropriate for web caching [7], where references come from many, independent web clients, it is a poor fit for multicore caches, where references come from at most a few threads.

Within a single thread, reference probabilities change rapidly over time and cache lines tend to behave similarly. For example, consider a program that scans over a 100 K-line array. This is an anti-recency pattern: with less than 100 K lines, LRU evicts every line before it hits and yields zero hits. The optimal replacement policy is to protect a fraction of the array so that some lines age long enough to hit. Doing so can significantly outperform LRU on scanning patterns (e.g., achieving 80% hit rate with a 80 K-line cache). But protection works only because lines have dynamic behavior: their reference probability *increases* as they age.

The independent reference model does not capture such information. In this example, each address is accessed once every 100 K accesses, so each has the same "reference probability". Thus, the IRM-optimal policy, which evicts the candidate with the lowest reference probability, cannot distinguish among lines and would replace them at random. Moreover, the IRM-optimal policy relies on identifying the reference probabilities of every address, which would be difficult to track in hardware. Clearly, a new model is needed.

Model description 5.1.2

We propose the *iid reuse distance model*, a model better suited to processor caching. Rather than assuming cache lines behave non-uniformly with static reference probabilities, the model asserts they behave *homogeneously* and share *dynamic* reference probabilities. Specifically, the model assumes that reuse distances are drawn independently from a common probability distribution, called the *reuse distance distribution*. Figure 5.2 and Table 5.1 summarize the nomenclature used.

Time is measured in accesses. A reference's *reuse distance* is the time (not unique addresses, cf. stack distances [41]) between references to the same address. A line's *age* is the time since its contents were last referenced. Thus a line's reuse distance always equals its age plus its time until reference. For example, in Figure 5.2, the dashed vertical line shows A at age 3. A has a reuse distance of four (the second A in ABBDA is four accesses after the first), so its time until reference is 1.

We divide time for each line into *lifetimes*, the idle periods between hits or evictions. For example, in Figure 5.2, A enters the cache at time $t = 0$, hits at $t = 3$ and $t = 7$, and is evicted by C at $t = 10$ (final column of figure). This produces three lifetimes for A, in order: length three, ending with a hit; length four, ending with a hit; and length three, ending with an eviction to insert C. In Section 5.2.2, we show that the ages of hits and evictions, and their implied lifetimes, frame the main tradeoffs for cache replacement.

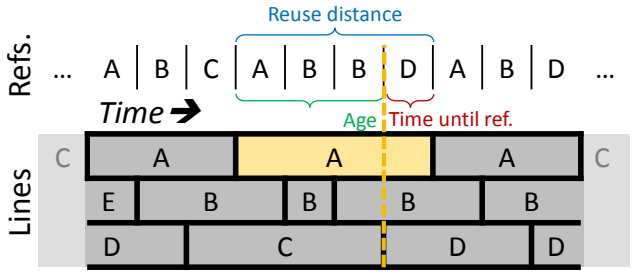


Figure 5.2: Model terminology on a simple, 3-line cache. Terms are shown for the line holding A following the third reference to B (dashed line). Each thick-bordered box is a lifetime.

Symbol	Meaning	
$\mathrm{P}[y]$	The probability of event y.	
$\mathrm{E}[X]$	The expected value of random variable (rv) X.	
$\mathrm{P}_X(x)$	The probability that rv X equals x, $\mathrm{P}[X=x]$.	
$\mathrm{P}_{X,Y}(x,y)$	The joint probability rv X equals x and rv Y equals y.	
$\mathrm{P}_X(x\|y)$	The conditional probability that rv X equals x given y.	
RV	**Meaning**	
D	Reuse distance of an access.	Input
H	Age at which a line hits.	Output
E	Age at which a line is evicted.	Output
A	Age of a line.	Internal
L	Lifetime of a line (see text).	Internal

Table 5.1: A summary of our notation. D is a property of the access stream; A, H, and E are properties of cached lines.

Definition. The *iid reuse distance model* asserts that references' reuse distances are independent. Additionally, it divides references into a small number of classes, and asserts that reuse distances are identically distributed within each class. In class c, reuse distances d are distributed according to $\mathrm{P}[D_c = d]$.

The iid model describes how candidates' reuse probability changes over time, and thus captures dynamic behavior. It is a simple, analyzable model that yields useful insights: we use it to model arbitrary age-based replacement policies (Appendix C), to obtain explicit, closed-form equations for miss rate under different access patterns (Appendix D), and PDP [44] also uses a similar model (although they do not formalize it to study the optimal policy). These results validate that the model usefully describes how programs reference the LLC.

The iid model is reasonably accurate because the private cache levels strip out most short-term temporal locality, often leaving complex and irregular access patterns at the LLC. Moreover, by dividing references into classes, the iid model captures relevant heterogeneity without adding much complexity. References could be classified in many ways; e.g., by thread, instruction vs. data, or compiler or programmer hints. However, for simplicity, in this work we classify candidates into those that have been reused and those that have not. Prior work has shown this simple scheme is effective [69, 82]; but note that unlike prior work, we do *not* automatically prefer reused over non-reused candidates—we use classification to describe how different lines behave, but rank all candidates within a common analytical framework.

Motivating identical distributions: The iid reuse distance model assumes lines behave identically within a class, but in reality some addresses are referenced more frequently than others or are accessed in distinct patterns. Nevertheless, this is a justifiable simplification at the LLC, since the most frequently accessed lines will be pinned in the lower levels of the cache hierarchy. Thus the LLC naturally sees an access stream stripped of short-term temporal locality [14, 80, 83]. Prior work has assumed caches exhibit statistically uniform behavior for other purposes [14, 44, 124, 125], and our evaluation justifies its use here.

Assuming identical distributions lets us focus on candidates' dynamic behavior at the cost of failing to distinguish between some candidates. Contrast this with the IRM [4], which focuses on heterogeneity at the cost of dynamic behavior. Prior work suggests that for processor LLCs, dynamic behavior is more important (Section 2.2), so the iid reuse distance model is a better fit. Moreover, recall that the iid model supports some heterogeneity via reference classes.

Motivating independence: The iid reuse distance model assumes reuse distances are independent, but this is inaccurate since reuse distances must not conflict with each other. Consider the access stream ABx. If A has reuse distance 2, then x is A; but if B has reuse distance 1, then x is B. Thus if A has reuse distance 2, then B cannot have reuse distance 1—their reuse distances are not independent.

A fully accurate model would have consider the history of reuse distances to avoid these collisions, which would be prohibitively expensive. But in practice the number of replacement candidates (e.g., 16 for a 16-way cache) is a small fraction of the cache, so collisions are rare. Appendix B approximates the error introduced by ignoring conflicts and shows it is small.

The iid reuse distance model is not a perfect description of programs, but that is not our goal. Rather, the model is a tool through which we explore the important tradeoffs in dynamic cache replacement. What matters is that is accurate *enough* to yield useful insights, while still being simple enough to analyze. Section 5.4 discusses the tradeoff between accuracy and complexity in depth. We first build the intuition behind EVA—what are the main tradeoffs in dynamic cache replacement, and what is the right replacement strategy?

Replacement under uncertainty 5.2

Many high-performance replacement policies rank candidates by predicting their time until reference, e.g. RRIP [69, 159] and IbRDP [79]. (RRIP stands for re-reference interval prediction, i.e. time until reference prediction, and RDP for reuse distance prediction.) While this approach is optimal with perfect information (MIN) and within the IRM (A_0), we now show it leads to sub-optimal performance within the iid model.

Flaws in predicting time until reference 5.2.1

A simple counterexample illustrates why predictions of time until reference are inadequate. It might seem obvious to some readers, but the prevalence of predicted time until reference among architects and in prior work suggests otherwise.

Suppose the replacement policy has to choose a victim from two candidates: A is referenced immediately with probability 9/10, and in 100 accesses with probability 1/10; and B is always referenced in two accesses. In this case, the best choice is to evict B, betting that A will hit immediately, and then evict A if it does not. Doing so yields an expected hit rate of 9/10, instead of 1/2 from evicting A. Yet

A's expected time until reference is $1 \times 9/10 + 100 \times 1/10 = 10.9$, and B's is 2. Thus, according to their predicted time until reference, A should be evicted. This is wrong.

Predictions fail because they ignore the possibility of future evictions. When behavior is uncertain and changes over time, the replacement policy can learn more about candidates as they age. This means it can afford to gamble that candidates will hit quickly and evict them if they do not. But expected time until reference ignores this insight, and is thus influenced by large reuse distances that will never be reached in the cache. In this example, the expected time until reference is skewed by reuse distance 100. *But the optimal policy never keeps lines this long.* Indeed, the value "100" plays no part in calculating the cache's maximum hit rate, so it is wrong for it to influence replacement decisions.

Such situations arise in practice and can be demonstrated on simple examples (Appendix B) and real programs (Section 5.6). This example uses expected time until reference to make predictions, as it is the most intuitive and commonly used heuristic, but other intuitive prediction metrics fail on similar counterexamples. The optimal policy should only consider what will actually happen to lines in the cache, *given its own replacement decisions*. This feedback loop differentiates replacement under uncertainty from replacement with perfect information (i.e., MIN).

Note that the problem is not a matter of prediction quality—it is sub-optimal even if one exactly computes the expected time until reference. In other words, introducing uncertainty around dynamic behavior invalidates the informal principle of optimality (Section 2.2.2). We are unaware of any prior in-depth discussion of its failings,[1] nor of alternative principles to guide replacement policy design.

5.2.2 Fundamental tradeoffs in replacement

All replacement policies have the same goal and face the same constraints. Namely, they try to maximize the cache's hit rate subject to limited cache space. We can develop the intuition behind EVA by considering each of these in greater depth. The following discussion applies to all replacement policies, regardless of memory reference model.

First, we introduce two random variables, H and L. H is the age at which the line hits, undefined if it is evicted; and L is the age at which the line's lifetime ends, whether by hit or eviction. For example, $\mathrm{P}[H = 8]$ is the probability the line hits at age 8; and $\mathrm{P}[L = 8]$ is the probability that it either hits or is evicted at age 8. (We discuss how to practically sample these distributions in Section 5.5.)

Policies try to maximize the cache's hit rate, which necessarily equals the average line's hit probability:

$$\text{Hit rate} = \mathrm{P}[\text{hit}] = \sum_{a=1}^{\infty} \mathrm{P}[H = a], \tag{5.1}$$

but are constrained by limited cache space. Since every access starts a new lifetime, the average lifetime equals the cache size (Section B.3):

$$S = \sum_{a=1}^{\infty} a \cdot \mathrm{P}[L = a] \tag{5.2}$$

Comparing these two equations, we see that hits are beneficial irrespective of their age, yet the cost in space increases in proportion to age (the factor of a in Equation 5.2). So to maximize the cache's hit rate, the replacement policy must attempt to both maximize hit probability *and* limit how long lines spend in the cache. From these considerations, one can see why MIN is optimal.

[1] Aho et al. [4] acknowledge the existence of counterexamples "under arbitrary assumptions" in an isolated remark, but do not elaborate. Moreover, [4] repeatedly emphasizes that A_0 conforms to the informal principle, and leaves the informal principle's performance outside the IRM to future work.

But how should one trade off between these competing, incommensurable objectives in general? Obvious generalizations of MIN like expected time until reference or expected hit probability are inadequate, since they only account for one side of the tradeoff.

We resolve this problem by viewing time spent in the cache as forgone hits, i.e. as the opportunity cost of retaining lines. We thus rank candidates by their *economic value added* (EVA), or how many hits the candidate yields over the "average candidate". Section 5.4 shows this intuitive formulation follows from MDP theory, and in fact maximizes the cache's hit rate.

EVA replacement policy 5.3

EVA essentially views retaining a line in the cache as an investment, with the goal of retaining the candidates that yield the highest profit (measured in hits). Since cache space is a scarce resource, we need to account for how much space each candidate consumes. We do so by "charging" each candidate for the time it will spend in the cache (its remaining lifetime). We charge candidates at a rate of a line's average hit rate (i.e., the cache's hit rate divided by its size), since this is the long-run opportunity cost of consuming cache space. Therefore, the EVA for a candidate of age a is:

$$\text{EVA}(a) = \text{P}\big[\text{hit}|\text{age } a\big] - \frac{\text{Hit rate}}{S} \times \text{E}\big[L - a|\text{age } a\big] \tag{5.3}$$

Example: To see how EVA works, suppose that a candidate has a 20% chance of hitting in 10 accesses, 30% chance of hitting in 20 accesses, and a 50% chance of being evicted in 32 accesses. We would expect to get 0.5 hits from this candidate, but these come at the cost of the candidate spending an expected 24 accesses in the cache. If the cache's hit rate were 40% and it had 16 lines, then a line would yield 0.025 hits per access on average, so this candidate would cost an expected $24 \times 0.025 = 0.6$ forgone hits. Altogether, the candidate yields an expected net $0.5 - 0.6 = -0.1$ hits—its *value added* over the average candidate is negative! In other words, retaining this candidate would tend to *lower* the cache's hit rate, even though its chance of hitting (50%) is larger than the cache's hit rate (40%). It simply takes up too much cache space to be worth the investment.

Now suppose that the candidate was not evicted (maybe there was an even less attractive candidate), and it appeared again $t = 16$ accesses later. Since time has passed and it did not hit at $t = 10$, its expected behavior changes. It now has a $30\%/(100\% - 20\%) = 37.5\%$ chance of hitting in $20 - 16 = 4$ accesses, and a $50\%/(100\% - 20\%) = 62.5\%$ chance of being evicted in $32 - 16 = 16$ accesses. Hence we expect 0.375 hits, a lifetime of 13.5 accesses, and forgone hits of $13.5 \times 0.025 = 0.3375$. In total, the candidate yields net 0.0375 hits over the average candidate—after not hitting at $t = 10$, it has become more valuable! This example shows that EVA can change over time, sometimes in unexpected ways.

We have limited our consideration to the candidate's current lifetime in these examples. But conceptually there is no reason for this, and we should also consider future lifetimes. We can get away with not doing so within a single class, because the iid reuse distance model asserts that candidates behave identically following a reference. We thus assume a candidate's EVA (its difference from the average candidate) is zero after its current lifetime. We extend EVA to classification in Section 5.3.2.

Computing EVA 5.3.1

We can generalize these examples using conditional probability and the random variables H and L introduced above. (Section 5.5 discusses how we sample these distributions.) To compute EVA, we compute the candidate's expected current lifetime and its probability of hitting in that lifetime.

Following conventions in MDP theory, we denote EVA at age a as $h(a)$ and the cache's hit rate as $\overline{g}$. Let $r(a)$ be the expected number of hits, or reward, and $c(a)$ the forgone hits, or cost. Then from Equation 5.3:

$$\text{EVA}(a) = h(a) = r(a) - c(a) \tag{5.4}$$

The reward $r(a)$ is the expected number of hits for a line of age a. This is simple conditional probability; a line of age a restricts the sample space to lifetimes at least a accesses long:

$$r(a) = \frac{\text{P}[H > a]}{\text{P}[L > a]} \tag{5.5}$$

The forgone hits $c(a)$ are $\overline{g}/S$ times the expected lifetime:

$$c(a) = \frac{\overline{g}}{S} \times \frac{\sum_{x=1}^{\infty} x \cdot \text{P}[L = a + x]}{\text{P}[L > a]} \tag{5.6}$$

To summarize, we select a victim by comparing each candidate's EVA (Equation 5.4) and evict the candidate with the lowest EVA. Note that our implementation does not evaluate Equation 5.4 during replacement, but instead precomputes ranks for each age and updates ranks infrequently. Moreover, Equations 5.5 and 5.6 at age a can each be computed incrementally from age $a+1$, so EVA requires just a few arithmetic operations per age (Section 5.5).

5.3.2 EVA with classification

We now extend EVA to support multiple reference classes, within which reuse distances are iid (Section 5.1). Specifically, we discuss in detail how to extend EVA to support reused/non-reused classification. The main difference is that lines do *not* regress to the mean at the end of their current lifetime; rather, their EVA depends on whether they hit or are evicted, since this determines if they transition to a reused or non-reused lifetime, respectively.

We denote the EVA of reused lines as $h^R(a)$, and the EVA of non-reused lines as $h^{NR}(a)$. We refer to class C whenever either R or NR apply. With classification, the terms of EVA are further conditioned upon a line's class. For example, the reward for a reused line is:

$$r^R(a) = \frac{\text{P}\big[H > a | \text{reused}\big]}{\text{P}\big[L > a | \text{reused}\big]} \tag{5.7}$$

Forgone hits and non-reused lines are similarly conditioned.

Without classification, we were able to consider only the current lifetime, assuming all lines behaved identically following a reference. When distinguishing among classes, this is untenable, and we must consider all future lifetimes because each class may differ from the average. The EVA for a single lifetime of class C is unchanged:

$$h_\ell^C(a) = r^C(a) - c^C(a), \tag{5.8}$$

but the future lifetimes are now important. Specifically, if a lifetime ends in a hit, then the next will be a reused lifetime. Otherwise, if it ends in an eviction, then the next will be non-reused. Thus if the miss rate of class C is m_C, then the EVA (extending to infinity) is:

$$h^C(a) = \overbrace{h_\ell^C(a)}^{\text{Current lifetime}} + \overbrace{(1 - m_C) \cdot h^R(0)}^{\text{Hits} \to \text{Reused}} + \overbrace{m_C \cdot h^{NR}(0)}^{\text{Misses} \to \text{Non-reused}} \tag{5.9}$$

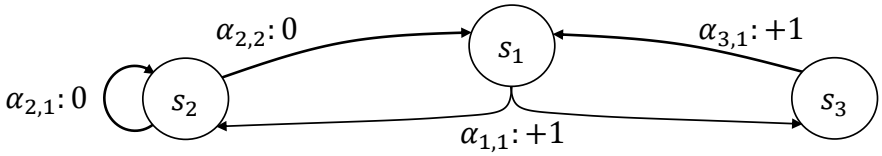


Figure 5.3: Example MDP with three states. Actions transition between states probabilistically (e.g., $\alpha_{1,1}$ to s_2 or s_3) and produce a reward (adjacent to edge label). A state may allow multiple actions (e.g., s_2).

Moreover, the average line's EVA is zero by definition, and reused lines are simply those that hit. So if the cache's miss rate is m, then:

$$0 = (1-m) \cdot h^R(0) + m \cdot h^{NR}(0) \tag{5.10}$$

From Equation 5.9 and Equation 5.10, it follows that:

$$h^R(a) = h_\ell^R(a) + \frac{m - m_R}{m_R} \cdot h_\ell^R(0) \tag{5.11}$$

$$h^{NR}(a) = h_\ell^{NR}(a) + \frac{m - m_{NR}}{m_R} \cdot h_\ell^R(0) \tag{5.12}$$

These equations have simplified terms, so their interpretation is not obvious. Essentially, these equations compare the rate that a class is creating additional reused lines ($m - m_C$) to the rate at which lines are leaving the reused class (m_R), and their ratio is how similar the class is to reused lines.

Summary: Classification ultimately amounts to adding a constant to the unclassified, per-lifetime EVA. If reused and non-reused lines have the same miss rate, then the added terms cancel and EVA reverts to Equation 5.4. EVA thus incorporates classification without a fixed preference for either class, instead adapting its policy based on observed behavior.

Although we have focused on reused/non-reused classification, these ideas apply to other classification schemes as well, so long as one is able to express how lines transition between classes (e.g., Equation 5.9).

Cache replacement as an MDP 5.4

We now formulate cache replacement in the iid reuse distance model as a Markov decision process (MDP). We use the MDP to show that *(i)* EVA maximizes hit rate and *(ii)* our implementation follows from a relaxation of the MDP. Finally, we discuss some nice properties that follow from this formulation.

Background on Markov decision processes 5.4.1

MDPs [122] are a popular framework to model decision-making under uncertainty, and can yield optimal policies for a wide variety of settings and optimization criteria. MDPs are frequently used in operations research, economics, robotics, and other fields.

As the name suggests, MDPs extend Markov chains to model decision-making. An MDP consists of a set of states S. In state $s \in S$, an action α is taken from the set A_s, after which the system transitions to state s' with probability $P(s'|s, \alpha)$. Finally, each action gives a reward $r(s, \alpha)$.

Figure 5.3 illustrates the key elements of MDPs with a simple example. This MDP has three states (s_i) and four actions (action $\alpha_{i,j}$ denotes the j^{th} action in state s_i). Actions trigger state transitions, which may be probabilistic and have an associated reward. In this example, all actions except $\alpha_{1,1}$ are

deterministic, but $\alpha_{1,1}$ can cause a transition to either s_2 or s_3. Actions $\alpha_{1,1}$ and $\alpha_{3,1}$ give a reward, while $\alpha_{2,1}$ and $\alpha_{2,2}$ do not. To maximize rewards in the long run, the optimal policy in this case is to take actions $\alpha_{1,1}$, $\alpha_{2,2}$, and $\alpha_{3,1}$ in states s_1, s_2, and s_3, respectively.

The goal of MDP theory is to find the best actions to take in each state to achieve a given objective, e.g. to maximize the total reward. After describing the cache replacement MDP, we give the relevant background in MDP theory needed to describe the optimal cache replacement policy.

5.4.2 The MDP

Our MDP models a single set of cache lines with state transitions on each access. Each access to the set can be a hit or a miss. A miss yields no reward, and allows the replacement policy to choose a victim. A hit yields reward of 1. The optimal policy maximizes the long-run average reward (i.e., the hit rate).

State space: Every line in the set has reuse distance that is iid. The optimal replacement policy (MIN [17]) is to evict the line with the largest time until next reference. Thus the state of the MDP is simply the time until reference of each line in the set.

With associativity of W, each state q is a tuple:

$$q = (t_1, t_2 \dots t_W) \tag{5.13}$$

where each t_i is the time until reference of the i^{th} line in the set, i.e. an integer greater than zero. If t_i is 1, then the i^{th} line hits on the next access.

This state is unobservable to any practical replacement policy, but it is the proper starting point to maximize cache performance. Any practical policy must instead operate on a partially-observable MDP (POMDP [109]) whose states s reflect *beliefs* about the true state q. In general, the states of a POMDP are probability distributions over all possible q, and beliefs are updated from observations upon every state transition. This framework lets POMDPs inherit the theoretical results of MDPs; however, solving POMDPs directly is unfeasible even for small problems, and is in fact undecidable in general [99]. Fortunately, careful consideration of the information available to a replacement policy within the iid reuse distance model lets us simplify beliefs considerably.

The time until reference of each line is its reuse distance minus its age, and by assumption every reuse distance within a single class is iid. A line's only distinguishing feature is its age a and class c, and this only implies (since reuse distances are iid) that its reuse distance must be larger than a (otherwise it would have hit). In other words, upon observing that a line of age a did not hit, the POMDP learns only that the line's reuse distance is greater than a. Hence the probability that a line of age a and class c has time until reference t is the conditional probability:

$$\mathrm{P}\big[D_c = t + a | \text{age } a\big] = \frac{\mathrm{P}\big[D_c = t + a\big]}{\mathrm{P}\big[D_c \geq a\big]} \tag{5.14}$$

Consequently, each belief-state s of the POMDP can be represented:

$$s = (c_1, a_1, c_1, a_2 \dots c_W, a_W), \tag{5.15}$$

where each a_i/c_i is the age/class, rather than time until reference, of the i^{th} line. This state space encodes the information directly observable to the replacement policy. It is sufficient within our reference model to fully describe beliefs about the underlying state q, since lines are independent and Equation 5.14 requires only the line's age a_i and class c_i to compute the probability distribution of its time until reference t_i.

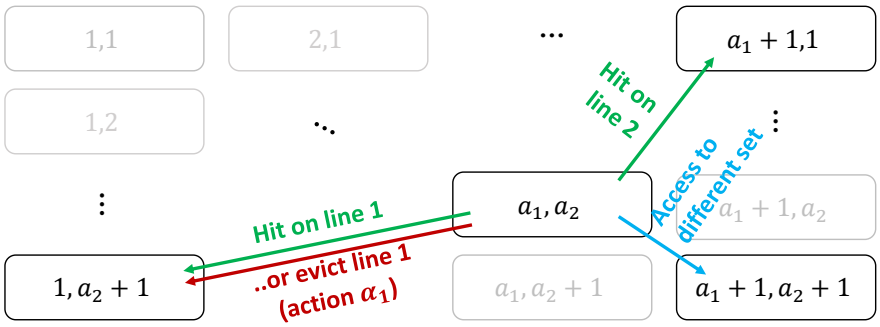


Figure 5.4: Cache replacement MDP for a 2-line set with a single reference class. The state $s = (a_1, a_2)$ is lines' ages; each hit gives reward 1; a line's age resets to 1 upon hit or eviction.

State transitions: Every access causes a state transition. In all states, there are W actions available to the replacement policy, $\alpha_1 \ldots \alpha_W$, corresponding to evicting the i^{th} line upon a miss. However, these actions only affect the next state $s' = (c'_1, a'_1 \ldots c'_W, a'_W)$ when an access to the set misses; otherwise, the next state is independent of replacement action. There are three possible outcomes to every cache access (Figure 5.4):

1. *Accesses to a different set:* In a cache with many sets, most accesses are to other sets. For these accesses, each line ages by one, i.e. a_i ages to $a'_i = a_i + 1$ and stays in the same class.

2. *Hits:* If the access is to the set, then lines may hit. Each line hits with probability that its time until reference is zero (Equation 5.14). Each line that hits then transitions to age 1 and may transition to a new class. Lines that do not hit age by one, as above.

3. *Evictions:* If the access is to the set and no line hits, then the replacement policy must choose a line to evict. To evict line j, it takes action α_j. In this case, line j transitions to age $a'_j = 1$ with the incoming reference's class, and all other lines age by one.

The reward in this MDP is the probability of having hit on any line, and is independent of action:

$$r(s, \alpha_j) = \sum_{i=1}^{W} \mathrm{P}\big[\text{hit } i | s\big] \tag{5.16}$$

It is unnecessary to explicitly compute the state transition probabilities to describe the optimal policy. Instead, we qualitatively describe the optimal policy and show our implementation approximates it.

Related work: Prior work has used MDPs to study replacement within the IRM [4, 7]. Our MDP is similar, but with the critical difference that we tractably model dynamic behavior. This is essential to achieve good performance. Prior approaches are too complex: Aho et al. [4] propose modeling the reference history to capture dynamic behavior, but deem this approach intractable and do not consider it in depth.

Since the IRM assumes static, heterogeneous reference probabilities, their MDP states are the *addresses* cached. In contrast, we use the iid reuse distance model, so we ignore addresses. Instead, the MDP states are the *ages* and *classes* cached. This lets us tractably capture dynamic behavior and produce a high-performance policy.

5.4.3 Background in optimal MDP policies

This section gives a brief overview of MDP theory, narrowly focused on our application of MDPs to cache replacement. For a comprehensive treatment, see Puterman [122]. MDP theory describes the optimal policy for many optimization criteria. We focus on the policy that maximizes the *long-run average reward*, as this represents the cache's hit rate in our model.

The goal of MDP theory is to describe the optimal policy for a given optimization criterion and give algorithms to solve for it explicitly. Depending on usage, different criteria are appropriate. For instance, one intuitive criterion is *expected total reward*, which maximizes received rewards over a finite or infinite horizon. In our MDP, the reward represents the probability of hitting in the cache at each state. The total reward over a finite horizon is thus the total number of hits after a specified number of accesses. To optimize the cache's hit rate, the appropriate optimization criterion is the *long-run average reward*.

Actions are chosen following a policy π. In general, the policy can be randomized or deterministic, and depend on the full history of states and actions or only the current state. Stationary (history-independent), deterministic policies are the simplest: such policies select an action as a function of the current state $\alpha = \pi(s)$. For the long-run average reward criterion, there always exists an optimal policy that is stationary and deterministic, so we limit consideration to such policies.

The optimal policy depends on the optimization criterion, but the general procedure is common across criteria. Each criterion has an associated optimality equation, which assigns a value to each state based on actions taken. For example, the optimality equation for the expected total reward after T actions is $v^T(s)$:

$$v^T(s) = \max_{\alpha \in A_s} \left\{ r(s,\alpha) + \sum_{s' \in S} \mathrm{P}(s'|s,\alpha) v^{T+1}(s') \right\} \tag{5.17}$$

Equation 5.17 says that the maximum total reward is achieved by taking the action that maximizes the immediate reward plus the expected total future reward. For an infinite horizon, Equation 5.17 drops the superscripts and becomes recursive on v.

The significance of the optimality equation is that any v^* that satisfies the optimality equation gives the maximum achievable value over all policies. Hence, a solution v^* directly yields an optimal policy: take the action chosen by the max operator in Equation 5.17. Optimality equations therefore summarize all relevant information about present and future rewards.

Optimality equations also suggest an obvious algorithm to find v^* via iteration to a fixed point, called *value iteration*. For the MDPs we consider, value iteration is guaranteed to converge within arbitrary precision.[2] However, value iteration is computationally expensive and our MDP has a large number of states, so we approximate the optimal solution, as discussed below.

Long-run average reward: The long-run average reward can be intuitively described using the expected total reward. If the expected total reward after T actions is $v^T(s)$, then the expected average reward, or *gain*, is $g^T(s) = v^T(s)/T$. The long-run average reward is just the limit as T goes to infinity. However, since all states converge to the same average reward in the limit, the gain is constant $g(s) = \overline{g}$ and does not suffice to construct an optimal policy. That is, the cache converges to the same hit rate from every initial state, so the long-run hit rate cannot by itself distinguish among states.

To address this, MDP theory introduces the *bias* of each state $h(s)$, which represents how much the total reward from state s differs from the mean. In other words, after T accesses one would expect a total reward (i.e., cumulative hits) of $T\,\overline{g}$; the bias is the asymptotic difference between the expected

[2] Our cache model is a unichain MDP, which can be made aperiodic following the transformation in [122, §8.5.4].

total reward from s and this value:

$$h(s) = \lim_{T\to\infty} v^T(s) - T\,\overline{g} \tag{5.18}$$

In the limit, the bias is a finite but generally non-zero number. The bias is also called the *transient reward*, since in the long run, it becomes insignificant relative to the total reward $v^T(s)$ once all states have converged to the average reward $\overline{g}$.

Perhaps surprisingly, this transient reward indicates how to achieve the best long-run average reward. Optimizing the bias is equivalent to optimizing the expected total reward after T actions, and choosing T large enough effectively optimizes the long-run average reward (to within arbitrary precision). For rigorous proofs, see [122, §8]. The optimality equation for the long-run average reward is:

$$\overline{g} + h(s) = \max_{\alpha\in A_s}\left\{ r(s,\alpha) + \sum_{s'\in S} \mathrm{P}(s'|s,\alpha)h(s') \right\} \tag{5.19}$$

Since solving Equation 5.19 yields an optimal policy, the optimal policy for long-run average reward MDPs is to *take the action that maximizes the immediate reward plus expected bias.*

A practical implementation of the MDP 5.4.4

This policy can be solved for explicitly, but doing so at run-time is impractical. Given a maximum reuse distance of $d_{\max}$, this MDP has ${d_{\max}}^W$ states. Even after compressing the state space (e.g., by removing unnecessary ordering of lines within the set), it has intractably many states. The MDP formulation is useful for its insight, but not as a practical policy itself.

Since the immediate reward (Equation 5.16) is independent of action, the optimal replacement policy is to maximize the expected future bias (Equation 5.19). Bias is the expected reward minus the expected average reward over the same number of actions—*bias is exactly EVA* (Section 5.3). Moreover, MDP theory guarantees that this approach maximizes the hit rate in the long run.

Since it is impractical to directly solve the MDP, we now discuss how we arrived at our implementation.

Ranking candidates: The main approximation is to decompose the policy from *MDP states* to *individual candidates*. That is, rather than selecting a victim by looking at all ages simultaneously, our implementation ranks candidates individually and evicts the one with highest rank. This approach is common in practical policies [44, 69, 141, 159], and we believe it introduces little error to the MDP solution.

The idea is that *(i)* hits occur to individual lines, so the cache's EVA is simply the sum of every line's EVA; *(ii)* we can thus maximize the cache's EVA by retaining the lines with the highest EVA; and *(iii)* following MDP theory, this maximizes the cache's hit rate.

EVA as a function of age: Moreover, the iid reuse distance model lets us approximate each line's EVA as a function of its age and class. In the iid model, these form a concise summary of beliefs about its time until reference. Since LLCs have high associativity and evictions continuously filter the pool of candidates, candidates will be compared against statistically similar alternate victims. Thus the EVA of the "average line"—the expectation conditioned only on age and class—is informative when selecting a victim.

EVA per lifetime: The final approximation is to consider EVA one lifetime at a time. This follows directly from Equation 5.18 by assuming the EVA is zero after a reference and letting $T \to \infty$.

Although these approximations do not follow necessarily from the iid model, we believe they are accurate in practice for large, highly-associative LLCs. Our evaluation indicates that the ranking approach is empirically effective.

5.4.5 Generalized optimal replacement

MDPs are capable of modeling a wide variety of memory reference models and cache replacement problems. We have presented an MDP for the iid reuse distance model, but it is easy to extend MDPs to other contexts. For example, prior work models the IRM with uniform [4] and non-uniform [7] replacement costs. With small modifications, our MDP can model a compressed cache [119], where candidates have different sizes and conventional prediction-based policies are clearly inadequate. The above discussion applies equally to such problems, and shows that EVA is a general principle of optimal replacement under uncertainty.

In particular, EVA subsumes prior optimal policies, i.e. MIN and A_0 (the IRM-optimal policy), in their respective memory reference models. MIN operates on lines' times until reference—the unobservable q states in Section 5.4.2. Since all hits yield the same reward, EVA is maximized by minimizing forgone hits, i.e. by minimizing time until reference. Hence a smaller time until reference is clearly preferable. In some sense, MIN is "buying hits as cheaply as possible". MDP theory guarantees that this greedy approach maximizes the long-run hit rate, and thus that MIN is optimal. In the IRM, A_0 operates on lines' reference probabilities. Reward is proportional to hit probability and forgone hits are inversely proportional to hit probability, so EVA is maximized by evicting lines with the lowest hit probability. EVA thus yields the optimal policy in both reference models.

The real contribution of the MDP framework, however, lies in going beyond prior reference models. Unlike predicting time until reference, the MDP approach does not ignore future evictions. It therefore gives an accurate accounting of all outcomes. In other words, EVA generalizes MIN.

5.4.6 Qualitative gains from planning theory

Convergence: MDPs are solved using iterative algorithms that are guaranteed to converge to an optimal policy within arbitrary precision. One such algorithm is *policy iteration* [88], wherein the algorithm alternates between computing the value v^T for each state and updating the policy. Our implementation uses an analogous procedure, alternatively monitoring the cache's behavior (i.e., "computing v^T") and updating ranks. Although this analogy is no strict guarantee of convergence, it gives reason to believe that our implementation converges on stable applications, as we have observed empirically.

Convexity: We proved in Chapter 4 that MIN is convex (Corollary 7). This proof does not rely on any specific properties of MIN, it only requires that *(i)* the policy is optimal in some model and *(ii)* its miss rate can be computed for any arbitrary cache size. Since EVA is iid-optimal and the MDP yields the miss rate, the proof holds for EVA as well. Thus EVA is convex, provided that the iid model and other assumptions hold. Empirically, we observe convex performance in the idealized implementation, and minor non-convexities with coarsened ages.

5.5 Implementation

Figure 5.5 shows the structures required to implement EVA. We first describe how our implementation ranks candidates during normal operation, then how we sample the hit and lifetime distributions to update ranks. We use a small table to rank candidates and periodically compute EVA to update this table. Since EVA requires the distribution of hits and evictions, we also count cache events under normal operation. However, we do *not* devote a fraction of sets to monitoring alternative policies (cf. [69, 125]), *nor* do we require auxiliary tags to monitor properties independent of the replacement policy (cf. [14, 44, 142]). This is possible because of EVA's analytical foundation: it converges to the right policy

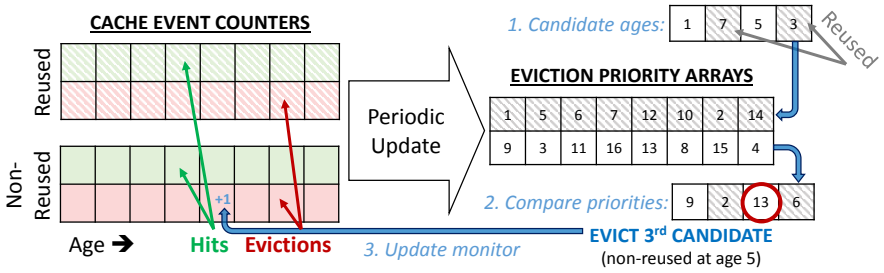


Figure 5.5: An efficient implementation of EVA.

simply by monitoring its own behavior (Section 5.4.6, cf. [159]). As a result, our implementation makes full use of the entire cache and eliminates overheads from monitors.

Coarse ages: We use global, coarsened timestamps [129, 130]. The cache controller has a k-bit timestamp counter ($k = 7$ in our prototype) that it increments every A accesses, and each cache tag has a k-bit timestamp field. A is chosen large enough to make wrap-arounds rare but give enough resolution (e.g., $A = S/32$).

Ranking: To rank candidates cheaply, we use a small *eviction priority array*. During replacement, each line's age is the difference between the current timestamp and its timestamp. We use each age to index into the priority array, and evict the candidate with the highest eviction priority. We assign ranks by order of EVA, i.e. if age a has higher rank than b, then $h(a) < h(b)$.

To work with classification, we add a reused bit to each cache tag, and use two priority arrays to store the priorities of reused and non-reused lines. Eviction priorities require $2^{k+1} \times (k+1)$ bits, or 256 B with $k = 7$.

The eviction priority array is dual ported to support peak memory bandwidth. With 16 ways, we can sustain one eviction every 8 cycles, for 19.2 GBps per LLC bank.

Event counters: To sample the hit and lifetime distributions, we add two arrays of 16-bit counters ($2^k \times 16\,\text{b} = 256\,\text{B}$ per array) that record the age histograms of hits and evictions. When a line hits or is evicted, the cache controller increments the counter in the corresponding array. These counters are periodically read to update the eviction priorities. To support classification, there are two arrays for both reused and non-reused lines, or 1 KB total with $k = 7$.

Updates: Periodically (every 128 K accesses), we recompute the EVA for every age and class. First, we obtain the hit and lifetime distributions ($\text{P}[H = a]$ and $\text{P}[L = a]$) from the counters. Then, we compute EVA in a small number of arithmetic operations per age. Finally, we sort the result to find the eviction priority of each age and update the eviction priority array.

These steps can be performed either in hardware or software. Algorithm 1 gives pseudo-code for the update algorithm. Updates occur in three passes over ages. In the first pass, we compute m_R, m_{NR}, and $m = 1 - \overline{g}$ by summing counters. Second, we compute lifetime EVA incrementally: Equation 5.8 requires four additions, one multiplication, and one division per age. Third, classification (e.g., Equation 5.9) adds one more addition per age.

In hardware, updates are off the critical path and can thus afford to be simple. Figure 5.6 shows a circuit that computes EVA using a single adder and a small ROM microprogram. We use fixed-point arithmetic, requiring $2^{k+1} \times 32$ bits to store the results plus seven 32-bit registers, or 1052 B with $k = 7$.

Algorithm 1. EVA's update algorithm.

Inputs: hitCtrs, evictionCtrs — event counters, ageScaling — age coarsening factor
Returns: rank — eviction priorities for all ages and classes

```
1:  function UPDATE
2:      m, m_R, m_NR ← MISSRATES(hitCtrs, evictionCtrs)        ▷ Miss rates from summing over counters.
3:      perAccessCost ← (1 − m)/S × ageScaling
4:      for c ∈ {reused, nonReused}:                           ▷ Compute EVA going backwards over all ages.
5:          expLifetime, hits, events ← 0
6:          for a ← 2^k to 1:
7:              expectedLifetime += events
8:              eva[c, a] ← (hits − perAccessCost × expectedLifetime)/events
9:              hits += hitCtrs[c, a]
10:             events += hitCtrs[c, a] + evictionCtrs[c, a]
11:     Δh_R ← (m − m_R)/m_R × eva[reused, 0]                  ▷ Differentiate classes.
12:     Δh_NR ← (m − m_NR)/m_R × eva[reused, 0]
13:     for a ← 1 to 2^k:
14:         eva[reused, a] += Δh_R
15:         eva[nonReused, a] += Δh_NR
16:     order ← ARGSORT(eva)                                   ▷ Finally, rank ages by EVA.
17:     for i ← 1 to 2^(k+1):
18:         rank[order[i]] ← 2^(k+1) − i
19:     return rank
```

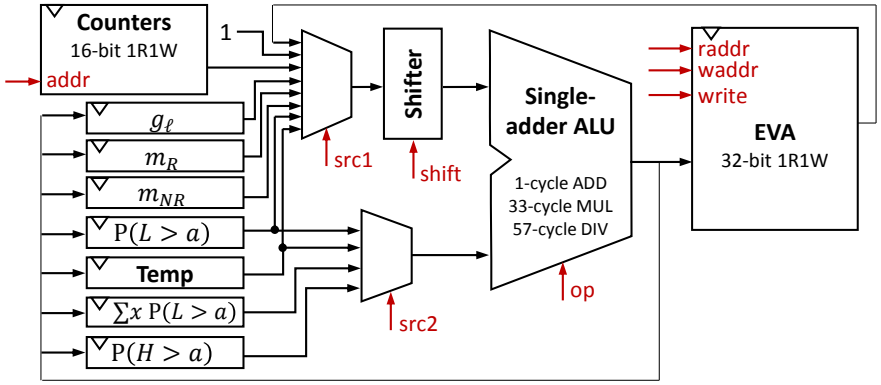


Figure 5.6: Datapath for EVA update circuit. Control signals (red, some not shown) come from ROM.

We have also implemented a small FSM to compute the eviction priorities (not shown), which performs an argsort using the merge sort algorithm, adding $2 \times 2^{k+1} \times (k+1)$ bits, or 512 B.

In software, updates can use low-priority threads to minimize interference. Software updates may be preferable to integrate EVA with other system objectives (e.g., cache partitioning [151]), to reduce implementation complexity, or on systems with dedicated OS cores (e.g., the Kalray MPPA-256 [37] or Fujitsu Sparc64 XIfx [163]).

Overheads: Our implementation imposes small overheads. The additional state is 1 KB for counters, 256 B for priority arrays, and 1.5 KB for hardware updates. We have implemented all components and synthesized them in a commercial process at 65 nm. We lack access to an SRAM compiler, so we use CACTI 5.3 [146] for all SRAMs (using register files instead of SRAM for all memories makes the circuit 4× larger). Ranking and counters run at 2 GHz; updates at 1 GHz. Table 5.2 shows the area and energy for each component at 65 nm. Absolute numbers should be scaled to reflect more recent technology nodes. We compute overhead relative to a 1 MB LLC using area from a 65 nm Intel E6750 [43] and energy from

	Area		Energy	
	(mm^2)	(% 1 MB LLC)	(nJ / LLC miss)	(% 1 MB LLC)
Ranking	0.010	0.05%	0.014	0.6%
Counters	0.025	0.14%	0.010	0.4%
Updates	0.052	0.30%	380 / 128 K	0.1%

Table 5.2: Implementation overheads at 65 nm.

Cores	Westmere-like OOO [131] at 4.2 GHz; 1 (ST) or 8 (MT)
L1 caches	32 KB, 8-way set-assoc, split D/I, 1-cycle
L2 caches	Private, 256 KB, 8-way set-assoc, inclusive, 7-cycle
L3 cache	Shared, 1 MB–8 MB, non-inclusive, 27-cycle; 16-way, hashed set-assoc
Coherence	MESI, 64 B lines, no silent drops; seq. consistency
Memory	DDR-1333 MHz, 2 ranks/channel, 1 (ST) or 2 (MT) channels

Table 5.3: Configuration of the simulated systems for single- (ST) and multi-threaded (MT) experiments.

CACTI. Overheads are small, totaling 0.5% area and 1.1% energy over a 1 MB LLC. Total leakage power is 2.2 mW. Even with one LLC access every 10 cycles, EVA adds just 7 mW at 65 nm, or 0.01% of the E6750's 65 W TDP [43].

Updates are off the critical path and complete in a few tens of K cycle in either hardware or software (vs. M cycles between updates). The longest step is sorting the priority arrays.

Each tag requires 8 b for replacement ($k = 7$ b timestamp plus one reused bit). This is a 1% overhead over 2-bit DRRIP; note other schemes use timestamps like EVA [44, 130, 141]. Moreover, our evaluation shows that *EVA saves cache space over prior policies at iso-performance*.

Complexity: A common concern with analytical policies like EVA is their complexity. However, we should be careful to distinguish between conceptual complexity—equations, MDPs, etc.—and implementation complexity. EVA was fully implemented in Verilog by a non-expert in one week. The circuit takes a few hundred lines of Verilog. A software implementation of updates is even simpler and takes only 68 lines of C++.

The only performance-sensitive components of EVA are ranking and counters, which are trivial to implement. Updates are off the critical path and use a single-cycle, single-ALU design. Complexity is low compared to microcontrollers shipping in commercial processors (e.g., Intel Turbo Boost [128]). Moreover, software updates eliminate even this modest complexity.

Evaluation 5.6

We now evaluate EVA over diverse benchmark suites and configurations. We show that EVA performs consistently well across benchmarks, and consequently outperforms existing policies, closes the gap with MIN, and saves area at iso-performance.

Methodology 5.6.1

We use zsim [131] to simulate systems with 1 and 8 OOO cores with parameters shown in Table 5.3. We simulate LLCs with sizes from 1 to 8 MB. The single-core chip runs single-threaded SPEC CPU2006 apps,

while the 8-core chip runs multi-threaded apps from SPEC OMP2012. Our results hold across different LLC sizes, benchmarks (e.g., PBBS [134]), and with a strided prefetcher validated against real Westmere systems [131].

Policies: We compare EVA with MIN, random, LRU, RRIP variants (DRRIP and SHiP), and PDP, which are implemented as proposed. We sweep configurations for each policy and select the one that is most area-efficient at iso-performance. DRRIP uses $M = 2$ bits and $\epsilon = 1/32$ [69]. SHiP uses $M = 2$ bits and PC signatures [159] with idealized, large history counter tables. DRRIP is only presented in text because it performs similarly to SHiP, but occasionally slightly worse. These policies' performance degrades with larger M. PDP uses an idealized implementation with large timestamps.

Area: Except where clearly noted, our evaluation compares policies' performance against their *total cache area at 65 nm*, including all replacement overheads. For each LLC size, we use CACTI to model data and tag area, using the default tag size of 45 b. We then add replacement tags and other overheads, taken from prior work. When overheads are unclear, we use favorable numbers for other policies: DRRIP and SHiP add 2-bit tags, and SHiP adds 1.875 KB for tables, or 0.1 mm^2. PDP adds 3-bit tags and 10 K NAND gates, or 0.02 mm^2. LRU uses 8-bit tags. Random adds no overhead. Since MIN is our upper bound, we also grant it zero overhead. Finally, EVA adds 8-bit tags and 0.09 mm^2, as described in Section 5.5.

Workloads: We execute SPEC CPU2006 apps for 10 B instructions after fast-forwarding 10 B instructions. Since IPC is not a valid measure of work in multi-threaded workloads [5], we instrument SPEC OMP2012 apps with heartbeats. Each completes a region of interest (ROI) with heartbeats equal to those completed in 1 B cycles with an 8 MB, LRU LLC (excluding initialization).

Metrics: We report misses per thousand instructions (MPKI) and end-to-end performance; for multi-threaded apps, we report MPKI by normalizing misses by the instructions executed on an 8 MB, LRU LLC. We evaluate *how well policies use information* by comparing against random and MIN; these policies represent the extremes of no information and perfect information, respectively. We report speedup using IPC (single-threaded) and ROI completion time (multi-threaded).

5.6.2 Single-threaded results

Figure 5.7 plots MPKI vs. cache area for ten representative, memory-intensive SPEC CPU2006 apps. Each point on each curve represents increasing LLC sizes from 1 to 8 MB. First note that the total cache area at the same LLC size (i.e., points along x-axis) is hard to distinguish across policies. This is because replacement overheads are small—less than 3% of total cache area in all cases.

In most cases, MIN outperforms all practical policies by a large margin. Excluding MIN, some apps are insensitive to replacement policy; e.g., `zeusmp` performs similarly across all practical policies. On others, random replacement and LRU perform similarly; e.g., `mcf` and `libquantum`. In fact, random often outperforms LRU.

EVA performs consistently well: SHiP and PDP improve performance by correcting LRU's flaws on particular access patterns. Both perform well on `libquantum` (a scanning benchmark), `sphinx3`, and `xalancbmk`. However, their performance varies considerably across apps. For example, SHiP performs particularly well on `perlbench`, `mcf`, and `cactusADM`. PDP performs particularly well on `GemsFDTD` and `lbm`, where SHiP exhibits pathologies and performs similar to random replacement.

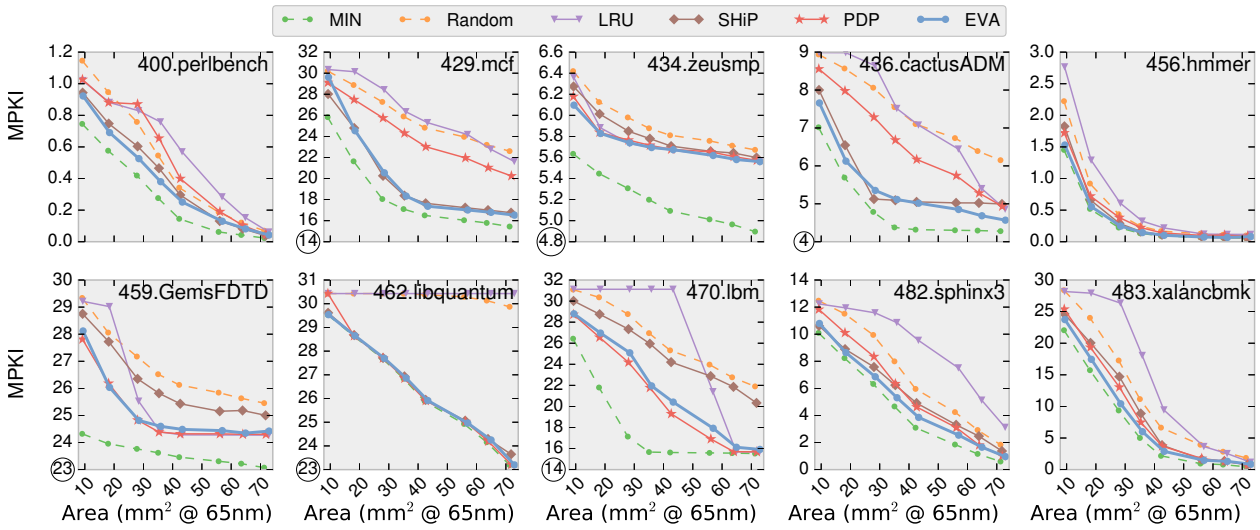


Figure 5.7: Misses per thousand instructions (MPKI) vs. total cache area across sizes (1 MB–8 MB) for MIN, random, LRU, SHiP, PDP, and EVA on selected memory-intensive SPEC CPU2006 benchmarks. (Lower is better.)

EVA matches or outperforms SHiP and PDP on most apps and cache sizes. This is because EVA is optimal within a model that accurately represents many programs, so the right replacement strategies naturally emerge from EVA where appropriate. As a result, EVA successfully captures the benefits of SHiP and PDP within a common framework, and sometimes outperforms both. Since EVA performs consistently well, but SHiP and PDP do not, EVA achieves the lowest MPKI of all policies on average.

The cases where EVA performs slightly worse arise for two reasons. First, in some cases (e.g., `mcf` with small caches), the access pattern changes significantly between policy updates. EVA can take several updates to adapt to the new pattern, during which performance suffers. But in most cases the access pattern changes slowly, and EVA performs well. Second, our implementation coarsens ages at a fixed granularity that can cause wraparounds for some apps (e.g., `lbm`). We observe that larger timestamps fix the problem, and so should adaptive age coarsening.

EVA edges closer to optimal replacement: Figure 5.8 compares the practical policies against MIN, showing the average MPKI gap over MIN across the most memory-intensive SPEC CPU2006 apps[3]—i.e., each policy's MPKI minus MIN's at equal area. One would expect a practical policy to fall somewhere between random replacement (no information) and MIN (perfect information). But LRU actually performs *worse than random* at many sizes because private caches strip out most temporal locality before it reaches the LLC, leaving scanning patterns that are pathological in LRU. In contrast, both SHiP and PDP significantly outperform random replacement. Finally, EVA performs best over all cache areas. On average, EVA closes 57% of the random-MIN MPKI gap. In comparison, DRRIP (not shown) closes 41%, SHiP 47%, PDP 42%, and LRU –9%.

EVA saves cache space: Because EVA improves performance, it requires a smaller cache than other policies to achieve a given level of performance. Figure 5.9 shows the *iso-MPKI total cache area* of each policy, i.e. the area required to match random replacement's average MPKI for different LLC sizes (lower

[3]All with ≥3 L2 MPKI; Figure 5.7 plus `bzip2`, `gcc`, `milc`, `gromacs`, `leslie3d`, `gobmk`, `soplex`, `calculix`, `omnetpp`, and `astar`.

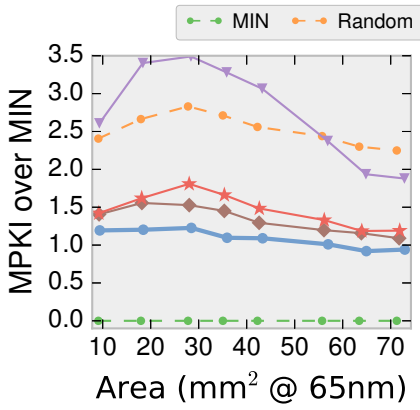


Figure 5.8: Avg MPKI above MIN for each policy on SPEC CPU2006 (lower is better).

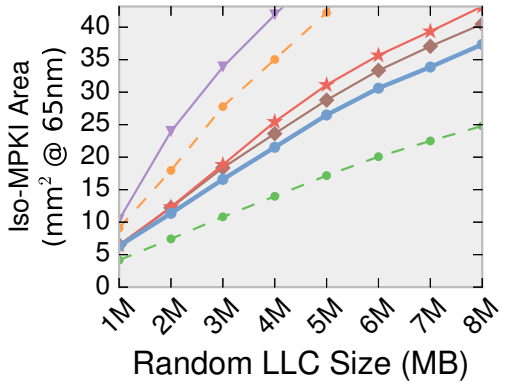


Figure 5.9: Iso-MPKI cache area: avg MPKI equal to random at different LLC sizes.

is better). For example, a 21.5 mm^2 EVA cache achieves the same MPKI as a 4 MB cache using random replacement, whereas SHiP needs 23.6 mm^2 to match this performance. (Refer back to Figure 5.1 for a detailed area breakdown.)

EVA is the most area efficient over the full range. On average, EVA saves 41% total cache area over random, 46% over LRU, 12% over PDP, 10% over DRRIP (not shown), and 8% over SHiP, the best practical alternative. However, note that MIN saves 48% over EVA, so there is still room for improvement.

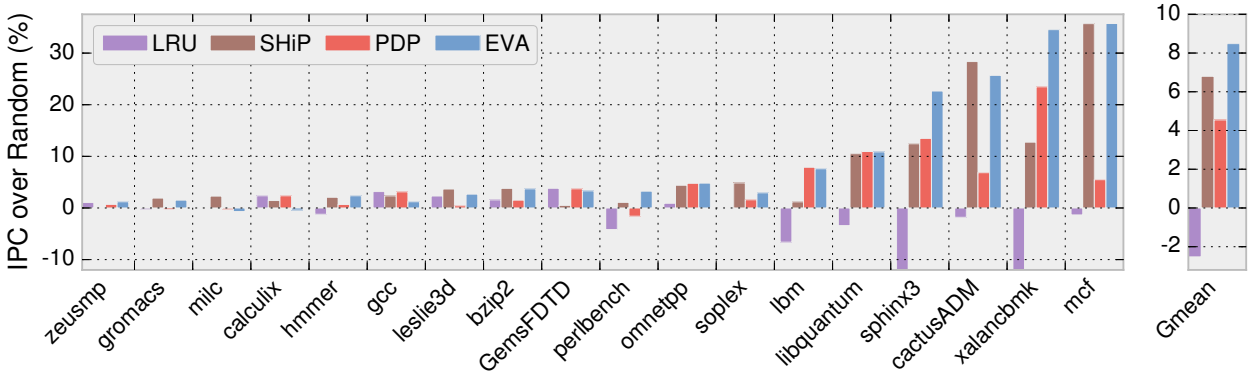


Figure 5.10: Performance over SPEC CPU2006 on a 4 MB LLC. (Only apps with >1% difference shown.)

EVA achieves the best end-to-end performance: Figure 5.10 shows the IPC speedups over random replacement on a 4 MB LLC. Only benchmarks that are sensitive to the replacement policy are shown, i.e. benchmarks whose IPC changes by at least 1% under some policy. Each policy uses a 4 MB LLC, so this experiment is not iso-area, but we still find the results meaningful since area varies by less than 2% across policies. (For a fixed policy, small changes in LLC size yield negligible performance differences. This is why small performance improvements yield large cache area savings.)

EVA achieves consistently good speedups across apps, while prior policies do not. SHiP performs poorly on `xalancbmk` and `lbm`, and PDP performs poorly on `mcf` and `cactusADM`. Consequently, EVA achieves the best speedup overall. Gmean speedups on sensitive apps (those shown) are for EVA 8.4%, DRRIP (not shown) 6.7%, SHiP 6.8%, PDP 4.5%, and LRU –2.3%.

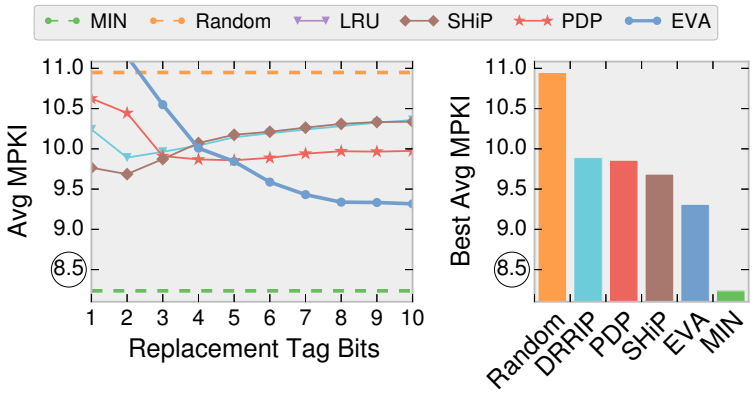


Figure 5.11: Avg MPKI for different policies at 4 MB vs. tag overheads (lower is better).

EVA makes good use of additional state: Figure 5.11 sweeps the number of tag bits for different policies and plots their average MPKI at 4 MB. The figure shows the best configuration on the right. (The trends are the same at other sizes.) Prior policies achieve peak performance with 2 or 3 bits, after which their performance plateaus or degrades.

EVA requires more tag bits to perform well, but its peak performance exceeds prior policies by a good margin. Comparing the best configurations, EVA's improvement over SHiP is 1.8× greater than SHiP's improvement over DRRIP. EVA with 8 b tags performs as well as an idealized implementation, yet still adds small overheads. These overheads more than pay for themselves, saving area at iso-performance (Figure 5.9).

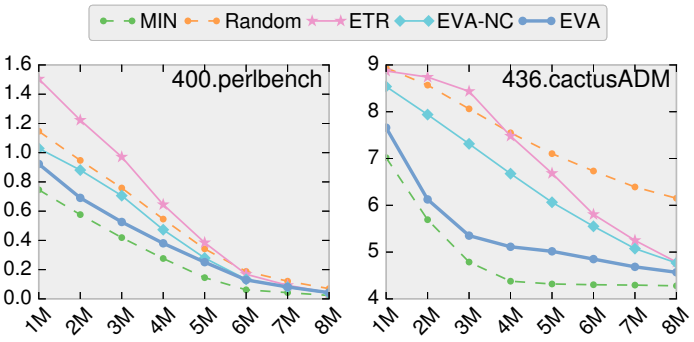


Figure 5.12: Comparison of EVA (with and without classification) vs. predicting time until reference (ETR).

EVA outperforms simple predictions: Figure 5.12 compares EVA, EVA without classification (EVA-NC), and an idealized prediction policy that ranks lines by their expected time until reference (ETR) given their age.[4]

Figure 5.12 shows results for just two apps: `perlbench` and `cactusADM`. ETR performs worse than random replacement in both cases, whereas EVA and EVA-NC rarely do so (except for noise on insensitive apps). Across all applications, ETR performs poorly compared to high-performance policies: it closes just 29% of the random-MIN MPKI gap, and its gmean IPC speedup vs. random at 2 MB is 1.4%. It handles

[4]We use large, idealized monitors. References with immeasurably large reuse distances use twice the maximum measurable distance in Equation 5.14.

simple access patterns well (e.g., `libquantum`), but fails on more complex patterns. These results support the claim that EVA is the correct generalization of MIN under uncertainty.

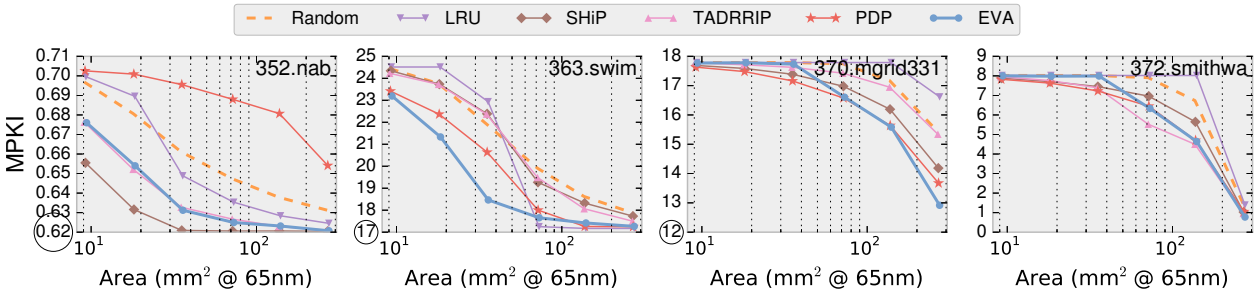


Figure 5.13: Misses per thousand instructions (MPKI) vs. total cache area across sizes (1 MB, 2 MB, 4 MB, 8 MB, 16 MB, and 32 MB) for random, LRU, SHiP, PDP, and EVA on selected SPEC OMP2012 benchmarks. (Lower is better.)

5.6.3 Multi-threaded results

Figure 5.13 extends our evaluation to multi-threaded apps from SPEC OMP2012. Working set sizes vary considerably, so we consider LLCs from 1 to 32 MB, with area shown in log scale on the x-axis. All qualitative claims from single-threaded apps hold for multi-threaded apps. Many apps are streaming or access the LLC infrequently; we discuss four representative benchmarks from the remainder.

As with single-threaded apps, SHiP and PDP improve performance on different apps. SHiP outperforms PDP on some apps (e.g., `nab`), and both perform well on others (e.g., `smithwa`). Unlike in single-threaded apps, however, DRRIP and thread-aware DRRIP (TADRRIP) outperform SHiP. This difference is largely due to a single benchmark: `smithwa` at 8 MB.

EVA performs well in nearly all cases and achieves the highest speedup. On the 7 OMP2012 apps that are sensitive to the replacement policy[5], the gmean speedup over random for EVA is 4.5%, DRRIP (not shown) 2.7%, TA-DRRIP 2.9%, SHiP 2.3%, PDP 2.5%, and LRU 0.8%.

5.7 Summary

We have argued for cache replacement by *economic value added* (EVA). We showed that predicting time until reference, a common replacement strategy, is sub-optimal; and we used first principles of caching and MDP theory to motivate EVA as an alternative principle. We developed a practical implementation of EVA that outperforms existing high-performance policies nearly uniformly on single- and multi-threaded benchmarks. EVA thus gives a theoretical grounding for practical policies, bridging the gap between theory and practice.

[5]Figure 5.13 plus `md`, `botsspar`, and `kd`.

Future Work 6

TECHNIQUES in this thesis open many directions for future work. Virtual caches are a natural framework in which to incorporate heterogeneous memories and allow for shared memory to scale to thousands of cores. Analytical cache replacement also has many potential extensions.

Virtual caches and heterogeneous memories: Systems are starting to ship with large on-chip memories. For example, 3D-stacked DRAM offers GBs of capacity at low energy and high bandwidth. Forthcoming technologies such as phase-change memory PCM offer similar promise. It is unclear, however, how to incorporate these memories into CMP memory systems.

Prior work has proposed using stacked DRAM either as OS-managed memory [2, 42, 72, 157] or an extra layer of cache [48, 71, 97, 126]. When used as a cache, the main challenge is how to use stacked DRAM for those applications that benefit without adding significant latency to main memory accesses for those that do not. Avoiding stacked DRAM when it is not beneficial is critical because the difference between stacked DRAM and main memory is small. Hence a conventional stacked DRAM cache must have a very high hit rate in order to improve performance.

Virtual caches naturally incorporate heterogeneous memories. Instead of treating new memories as an additional layer of cache, we can add them to the pool of capacity that is allocated among virtual caches. Thus, applications that need large capacity will use stacked DRAM when it is beneficial, but if no application requires the capacity, it will simply be left unused.

Virtual caches may thus harness the benefits of heterogeneous memories without the latency and energy penalties of a rigid cache hierarchy.

Virtual cache hierarchies: Virtual caches give each application a cache sized and placed according to how it uses memory. In this sense, virtual caches remove the need for cache hierarchy, since hierarchy is the traditional design to adapting to different working set sizes. However, cache hierarchy is also beneficial for individual applications that access data structures of different sizes. For these applications, a single-level virtual cache is an incomplete solution.

To improve the performance of these applications and let them fully exploit heterogeneous memories like stacked DRAM, we can extend virtual caches to build *virtual cache hierarchies*. By chaining together several virtual caches, we can size each level and place it in cache banks to further reduce data movement.

The key challenge in virtual cache hierarchies would be managing the additional complexity in reconfigurations. To configure a single-level virtual cache, Jigsaw starts by computing its size. No single parameter describes a multi-level hierarchy, however, and we instead need the size of each level. This is a multi-dimensional optimization problem that is more computationally difficult to solve.

Program-level data classification: Both virtual caches and EVA rely on coarse-grain classification schemes to distinguish between different types of memory references. Jigsaw relies on sharing pattern

at a page granularity, so any page that is referenced by another thread is permanently upgraded to the process-shared VC and loses data placement information. EVA uses reused/non-reused classification which, although simple and effective, could be potentially improved.

One approach is to extend classification further up the software stack. Cache-based systems generally assume an opaque interface to software, and give programmers no way to express semantic information about how they use memory. Prior work that allows such hints exposes a static interface, where the programmer or compiler say whether or not to cache a given memory reference. This interface, while effective on regular programs, falls foul of the same problems as scratchpads: when program behavior is unpredictable, it performs worse than having no hints at all. And programs are unpredictable for many reasons: input-dependent behavior, phase changes, irregular access patterns, changing resources in shared systems, or sheer complexity that makes them difficult to analyze.

Much like Jigsaw improved upon prior D-NUCAs by giving a layer of indirection between a class and its placement, we can improve on cache hints by providing a layer of indirection between the *semantic hint* and the *caching policy*. That is, we can allow programmers to express semantic relationships between memory references, and let the cache dynamically decide the right policy for each class.

In practice, this would mean letting programs define their own virtual caches (in Jigsaw) or classes (in EVA). This strategy would let each actor do what it knows best: programs express the "big picture" about how they use memory, and the cache adapts to dynamic behavior in response. The goal is to get the most information at the least cost: programmers and compilers often know very well how they use memory, but have no good way to express it in current systems; while dynamic caching polices know the system resources available, but have no good way to capture program semantics in current systems.

Scaling shared memory systems to thousand+ cores: One of Jigsaw's main advantages is that it allows cache performance to scale independent of system size. Programs with small working sets can use nearby cache banks, without any global scaling bottlenecks. This qualitative advantage changes the marginal tradeoffs in system design in favor of larger systems, since adding cores results in smaller penalty than in prior cache designs. In particular, it should be possible to scale Jigsaw to very large systems with more than a thousand cores, possibly spread across multiple boards or racks and connected by high-speed interconnects, without harming the performance of individual applications. For Jigsaw to work well on these large systems, it will be particularly important to scale parallel runtime systems to encourage locality—without locality of reference, no caching scheme can reduce data movement.

Cache-aware algorithms: Similarly, since virtual cache performance changes with the number of threads sharing data and the size of the working set, virtual caches shift marginal algorithmic tradeoffs. Virtual caches make algorithms sensitive to their degree of parallelism in a way that S-NUCA cache designs do not. Algorithms may therefore want to focus more on data locality, so that virtual caches can keep data placed near threads that use it. Algorithms may also want to restrain their parallelism to increase data locality if the marginal speedup from additional threads does not compensate for the additional latency to shared data.

Applying Talus to other policies: Talus is currently limited by its baseline policy, which is in practice LRU. LRU does not capture certain access patterns (e.g., reused vs. non-reused heterogeneity), and so this information is not available to Talus. There is no fundamental reason, however, why Talus cannot divide the cache into reused and non-reused partitions [83]. Doing so may allow Talus to match SHiP's performance without sacrificing LRU's predictability.

Extending EVA to new contexts: We have evaluated EVA in the traditional context of cache replacement, but other replacement problems have received attention in prior work. For example, compressed caches [119] have candidates of different sizes, and web caching [7] has non-uniform replacement costs. These differing costs change the optimal policy, but in ways that are easily captured by MDPs. SSDs are widely used as hard disk caches, and misses are very expensive (milliseconds, or millions of cycles). SSD caches can thus afford much larger replacement overheads than LLC caches, and analytical techniques like EVA, combined with predictive modeling (Appendix C), could be particularly attractive. It should be straightforward, therefore, to extend EVA to these contexts.

Conclusion 7

THIS thesis has presented novel techniques to reduce data movement. We have shown that an analytical approach can improve performance and provide further qualitative benefits over purely empirical, best-effort designs. In particular, we have presented the following contributions:

- **Virtual caches** are a flexible way to manage non-uniform cache architectures. Virtual caches schedule data across cache banks to fit applications' working sets in nearby cache banks. Virtual caches thus reduce both cache access latency and miss rate, the two sources of data movement.

  Virtual caches require lightweight hardware mechanisms: chiefly the VTB, a small table to configure the placement of data, reconfiguration support, and small monitors to sample miss curves. These mechanisms grant software complete control over the data layout, but finding the best layout is itself a challenging problem. We have developed efficient algorithms to solve this problem, improving on standard partitioning algorithms.

  We reduced data movement even further through **contention-aware thread scheduling**, which reduces contention for scarce cache capacity by separating threads with large working sets.

- We have explored a principled approach to cache replacement. We showed how to use partitioning to eliminate performance cliffs, capturing much of the benefit of high-performance replacement policies while retaining the attractive properties of well-studied policies like LRU. Our technique uses a simple model of sampled access streams to guarantee **predictable, convex cache performance**, letting cache partitioning benefit from high-performance replacement, while simplifying partitioning algorithms and improving fairness. Our technique adds just a single hardware hash function and a lightweight software runtime to existing cache partitioning schemes.

  We also used planning theory to argue for **replacement by economic value added** (EVA), and presented a practical implementation of this policy. To develop EVA, we introduced the iid reuse distance model, a simple memory reference model that faithfully captures the important facets of dynamic cache behavior. In appendices, we validated this model by accurately predicting the performance of several policies across many workloads. We use Markov decision processes (MDPs) to model cache replacement within the iid model and show that EVA maximizes the cache's hit rate. EVA has a simple implementation, inspired by MDP algorithms, and improves performance and saves area over prior policies.

These contributions provide both *(i)* quantitative performance and energy benefits over the state-of-the-art and *(ii)* qualitative advantages over best-effort heuristics. Moreover, by taking an analytical approach, we have developed a coherent theoretical framework to reason about data movement.

Appendices

Jigsaw Algorithms A

Algorithm listings A.1

Algorithm 2 shows the complete Peekahead algorithm. First, ALLHULLS preprocesses each virtual cache's miss curve and computes its POIs. Then, PEEKAHEAD divides cache space across virtual caches iteratively using a max-heap. In practice, ALLHULLS dominates the run-time of Algorithm 2 at $O(P \cdot S)$, as Appendix 3.7 confirms.

AllHulls: In ALLHULLS, the procedure is as described previously: Two vectors are kept, pois and hull. pois is the full list of POIs whereas hull contains only the points along the current convex hull. Each POI is represented by its coordinates and its horizon, which is initially ∞.

The algorithm proceeds as described in the example. Points are always added to the current hull (lines 18-19). Previous points are removed from hull by backtracking (lines 7-17). A point (x, y) is removed from pois when it lies in a concave region. This occurs when it is removed from hull at $x + 1$ (line 13). Otherwise, the point lies in a convex region and is part of a sub-hull, so it its horizon is set to x, but it is not deleted from pois (line 15).

Peekahead: PEEKAHEAD uses POIs to implement lookahead. POIs for partition p are kept in vector pois$[p]$, and current$[p]$ tracks its current allocation. heap is a max-heap of the maximal utility per unit steps for each partition, where a step is a tuple of: (partition, utility per unit, step size, POI).

PEEKAHEAD uses two helper routines. NEXTPOI gives the next relevant POI for a partition from its current allocation and given remaining capacity. It simply iterates over the POIs following current$[p]$ until it reaches capacity (line 28), a valid POI is found (line 29), or no POIs remain (line 31) which indicates a concave region, so it claims all remaining capacity. ENQUEUE takes the next POI and pushes a step onto heap. This means simply computing the utility per unit and step size.

The main loop repeats until capacity is exhausted taking steps from the top of heap. If capacity has dropped below the best next step's requirements, then the step is skipped and the best next step under current capacity enqueued (line 40).

Placement: Algorithm 3 shows the complete partitioning algorithm, in particular Jigsaw's data placement algorithm following the invocation of PEEKAHEAD to obtain the virtual cache sizes, or "budgets". The goal of the algorithm is for each virtual cache to exhaust its budget on banks as close to the source as possible. The source is the core or "center of mass" of cores that generate accesses to a virtual cache. The distance of banks from the source is precomputed for each partition and passed as the lists $D_1 ... D_P$. Each bank is given an inventory of space, and virtual caches simply take turns making small "purchases" from

Algorithm 2. The Peekahead algorithm. Compute all reachable convex hulls and use the convexity property to perform Lookahead in linear time. Letters in comments refer to points in Figure 3.13. $\overline{AB}$ is the line connecting A and B.

Inputs: A single miss curve: M, Cache size: S
Returns: POIs comprising all convex hulls over $[0,X]\ \forall\ 0 \le X \le S$

```
1:  function AllHulls(M, S)
2:      start ← (0, M(0), ∞)                                  ▷ POIs are (x, y, horizon)
3:      pois[...] ← {start}                                    ▷ Vector of POIs
4:      hull[...] ← {pois.Head}                                ▷ Current convex hull; references pois
5:      for x ← 1 to S:
6:          next ← (x, M(x), ∞)
7:          for i ← hull.Length − 1 to 1:                      ▷ Backtrack?
8:              candidate ← hull[i]
9:              prev ← hull[i − 1]
10:             if candidate is not Below (prev next):
11:                 hull.PopBack()                             ▷ Remove from hull
12:                 if candidate.x ≥ x − 1:
13:                     pois.PopBack()                         ▷ Not a POI (C, G)
14:                 else:
15:                     candidate.horizon ← x − 1              ▷ POI not on hull (F)
16:             else:
17:                 break                                      ▷ POI and predecessors valid (for now)
18:         pois.PushBack(next)                                ▷ Add POI
19:         hull.PushBack(pois.Tail)
20:     return pois

Inputs: Partition miss curves: M_1...M_P, Cache size: S
Returns: Partition allocations: A[...]
21: function Peekahead(M_1...M_P, S)
22:     pois[...] ← {AllHulls(M_1, S)...AllHulls(M_P, S)}
23:     current[...] ← {pois[1].Head...pois[p].Head}           ▷ Allocations
                       (P times)
24:     A[...] ← {0...0}
25:     heap ← MakeHeap()                                      ▷ Steps sorted by ΔU
26:     function NextPOI(p)
27:         for i ← current[p] + 1 to pois[p].Tail:
28:             if i.x > current[p].x + S: break               ▷ No space left
29:             if i.horizon > current[p].x + S: return i      ▷ Valid POI
30:         x ← current[p].x + S                               ▷ Concave region; take S
31:         return (x, M_p(x), ∞)
32:     function Enqueue(p)
33:         next ← NextPOI(p)
34:         ΔS ← next.x − current[p].x
35:         ΔU ← (current[p].y − next.y) / ΔS
36:         heap.Push((p, ΔU, ΔS, next))
37:     Enqueue([1...P])
38:     while S > 0:                                           ▷ Main loop
39:         (p, ΔU, ΔS, next) ← heap.Pop()
40:         if S ≥ ΔS:                                         ▷ Allocate if we have space
41:             current[p] ← next
42:             A[p] ← A[p] + ΔS
43:             S ← S − ΔS
44:         Enqueue(p)
45:     return A[...]
```

Algorithm 3. Jigsaw's partitioning policy. Divide the LLC into virtual caches to maximize utility and locality. Virtual caches use budgets produced by Peekahead to claim capacity in nearby bank partitions in increments of Δ_0.

Inputs: Partition miss curves: $M_1 ... M_P$, Cache size: S, Num. banks: B, Banks sorted by distance: $D_1 ... D_P$
Returns: Virtual cache allocation matrix: $\left[A_{p,b}\right]$ where $1 \le p \le P$ and $1 \le b \le B$

function PARTITION($M_1 ... M_P, S, B$)
  budget[...] $\leftarrow$ PEEKAHEAD($M_1 ... M_P, S$)
  inventory[...] $\leftarrow \overbrace{\left\{\frac{S}{B}, \frac{S}{B}, \frac{S}{B} ... \frac{S}{B}\right\}}^{B \text{ times}}$
  $d[...] \leftarrow \{D_1.\text{HEAD} ... D_P.\text{HEAD}\}$ ▷ Prefer closer banks
  $A \leftarrow [0]_{\substack{1 \le p \le P \\ 1 \le b \le B}}$
  while $\sum$budget > 0 **:**
    for $s \leftarrow 1$ **to** P **:**
      $b \leftarrow d[i]$ ▷ Closest bank
      if inventory$[b] > \Delta_0$ **:**
        $\Delta \leftarrow \Delta_0$ ▷ Have space; take Δ_0
      else :
        $\Delta \leftarrow$ inventory$[b]$ ▷ Empty bank; move to next closest
        $d[i] \leftarrow d[i]+1$
      $A_{p,b} \leftarrow A_{p,b} + \Delta$
      budget$[s] \leftarrow$ budget$[s] - \Delta$
      inventory$[b] \leftarrow$ inventory$[b] - \Delta$
  return A

banks until all budgets are exhausted. Peekahead dominates the run-time of the complete algorithm at $O(P \cdot S)$.

Correctness A.2

The correctness of Algorithm 2 relies on ALLHULLS constructing all relevant convex hulls and PEEKAHEAD always staying on a convex hull. The logic is as follows:

- The highest utility per unit step for a miss curves comes from the next point on its convex hull.
- If a point (x, y) is on the convex hull of $[0, Y]$, then it is also on the convex hull of $[0, X]$ for any $x \le X \le Y$.
- POIs from ALLHULLS and repeated calls to NEXTPOI with capacity $0 \le S' \le S$ produce the convex hull over $[0, S']$.
- Remaining capacity S' is decreasing, so current$[p]$ always lies on the convex hull over $[0, S']$.
- Therefore PEEKAHEAD always makes the maximum-utility move, and thus correctly implements lookahead.

Definition. A *miss curve* is a decreasing function $M : [0, S] \to \mathbb{R}$.

Definition. Given a miss curve M and cache sizes x, y, the *utility per unit* between x and y is defined as:

$$\Delta U(M, x, y) = \frac{M(y) - M(x)}{x - y}$$

Which in plain language is simply the number of additional hits per byte gained between x and y.

We now define what it means to implement the lookahead algorithm. Informally, a lookahead algorithm is one that successively chooses the maximal utility per unit allocation until all cache space is exhausted.

Definition. Given P partitions, miss curves $M_1 ... M_P$, remaining capacity, S' and current allocations $x_1 ... x_P$, a *lookahead move* is a pair (p, s) such that partition p and allocation $0 < s \leq S'$ give maximal utility per unit:

$$\forall q : \Delta U(M_p, x_p, x_p + s) \geq \Delta U(M_q, x_q, x_q + s)$$

Definition. Given P partitions, miss curves $M_1 ... M_P$, cache size S, a *lookahead sequence* is a sequence $L = (p_1, s_1), (p_2, s_2) ... (p_N, s_N)$ such that the *allocations for p at i*, $x_{p,i} = \sum_{i=1}^{N} \{s_i : p_i \equiv p\}$, and *remaining capacity at i*, $S_i = \sum_{i=1}^{N} s_i$, satisfy the following:

1. For all i, (p_i, s_i) is a lookahead move for $M_1 ... M_P$, $S' = S - S_i$, and $x_{1,i} ... x_{P,i}$.
2. $S_N = S$.

Definition. A *lookahead algorithm* is one that produces a lookahead sequence for all valid inputs: $M_1 ... M_P$, and S.

Theorem 9. *UCP Lookahead is a lookahead algorithm.*

Proof. UCP Lookahead is the literal translation of the definition. At each step, it scans each partition to compute the maximum utility per unit move from that partition. It then takes the maximum utility per unit move across all partitions. Thus at each step is makes lookahead moves, and proceeds until capacity is exhausted. □

We now show that PEEKAHEAD implements lookahead.

Lemma 10. *Given a miss curve M and starting from point A on the convex hull over $[0, S]$, the first point on the miss curve that yields maximal utility per unit is the next point on the miss curve's convex hull.*

Proof. Let the convex hull be H and the next point on the convex hull be B.

Utility per unit is defined as negative slope. Thus, maximal utility per unit equals minimum slope. Convex hulls are by definition convex, so their slope is non-decreasing. The minimum slope of the convex hull therefore occurs at A. Further, the slope of the hull between A and B is constant, so B yields the maximal utility per unit.

To see why it is the first such point, consider that all points on M between A and B lie above H. If this were not the case for some C, then the slope between A and C would be less than or equal to that between A and B, and C would lie on H. Therefore since all points lie above H, none can yield the maximal utility per unit. □

Lemma 11. *For any miss curve M, if (x, y) is on H_Y, the convex hull of M over $[0, Y]$ then it is also on H_X, the convex hull of M over $[0, X]$, for any $x \leq X \leq Y$.*

Proof. By contradiction. Assume (x, y) is on H_Y but not on H_X. Then there must exist on $M : [0, X] \to \mathbb{R}$ points $A = (a, M(a))$ and $B = (b, M(b))$ such that $a < x < b$ and (x, y) lies above the line $\overline{AB}$. But $[0, X] \subset [0, Y]$ so A and B are also in $[0, Y]$. Thus (x, y) is not on H_Y. □

Lemma 12. *In ALLHULLS on a miss curve M over domain $[0, S]$, upon completion of iteration $x \equiv X$, hull contains H_X, the convex hull of M over $[0, X]$.*

Proof. All points in $[0,X]$ are added to hull, so we only need to prove that all points *not* on the hull are removed. By the definition of a convex hull, a point $(a, M(a))$ is on H_X iff it is not below the line connecting any two other points on $[0,X]$. Now by induction:

Base case: At $X = 1$, hull contains the points $(0, M(0))$ and $(1, M(1))$. These are the only points on $[0,1]$, so hull contains H_1.

Induction: Assume that hull contains H_X. Show that after iteration $X+1$, hull contains H_{X+1}. This is guaranteed by backtracking (lines 7-17), which removes all points on H_X not on H_{X+1}.

Let $H_X = \{h_1, h_2 ... h_N\}$ and $h_{N+1} = (X+1, M(X+1))$. Then next $\equiv h_{N+1}$ and, at the first backtracking iteration, candidate $\equiv h_N$ and prev $\equiv h_{N-1}$.

First note that $H_{X+1} \subset H_X \cup \{h_{N+1}\}$. This follows from Lemma 11; any point on H_{X+1} other than h_{N+1} *must* lie on H_X.

Next we show that backtracking removes all necessary points by these propositions:

(i) *If candidate lies is on H_{X+1} then all preceding points are on H_{X+1}.*
Backtracking stops when a single point is valid (line 17). We must show that no points that should be removed from hull are missed by terminating the loop. This is equivalent to showing

$$h_i \notin H_{X+1} \Rightarrow h_{i+1} ... h_N \notin H_{X+1}$$

Let h_i be the first point on H_X not on H_{X+1}. Then there exists $i < k \leq N+1$ such that h_i lies above $\ell_k = \overline{h_{i-1}h_k}$. Further, all h_j for $i < j \leq N$ lie above $\ell_i = \overline{h_{i-1}h_i}$. But since h_i lies above ℓ_k, ℓ_i is above ℓ_k to the right of h_i. Since all $i < j \leq N$ lie above ℓ_i, $k = N+1$. Therefore all h_j for $i < j \leq N$ lie above $\overline{h_{i-1}h_{N+1}}$.

(ii) *Points below $\overline{\text{prev next}}$ lie on H_{X+1}.*
Backtracking removes all points that do not lie below $\overline{\text{prev next}}$. We must show this is equivalent to testing all pairs of points in $[0,X]$.
Note that for all $1 \leq i \leq N$, $M : [0,X] \to \mathbb{R}$ lies on or above $\overline{h_i h_{i+1}}$ by the definition of a convex hull.
Consider cases if candidate $\in H_{X+1}$:
candidate $\in H_{X+1}$: By the first proposition, prev is also on H_{X+1}. So by the previous note, M lies above both $\overline{\text{prev candidate}}$ and $\overline{\text{candidate next}}$. Therefore candidate lies below $\overline{\text{prev next}}$.
candidate $\notin H_{X+1}$: If prev $\in H_{X+1}$ then by definition of a convex hull, candidate lies above $\overline{\text{prev next}}$. If not, then let h_i be the last point on H_{X+1} before candidate and h_k the first point on H_{X+1} after candidate. Then candidate lies above $\overline{h_i h_k}$. Note that $h_k \equiv$ next by the previous proposition.
h_i is also before prev because no points on H_X lie between candidate and prev. Furthermore, since next lies below $\overline{h_i h_{i+1}}$ but above $\overline{h_{i-1}h_i}$, $\overline{h_{i+1}\text{ next}}$ lies above $\overline{h_i\text{ next}}$. H_X has non-decreasing slope and M is decreasing, so $\overline{h_j\text{ next}}$ lies above $\overline{h_{j-1}\text{ next}}$ for all $i < j \leq N$. In particular, $\overline{\text{candidate next}}$ lies above $\overline{\text{prev next}}$ and thus candidate lies above $\overline{\text{prev next}}$.
$\therefore$ candidate $\in H_{X+1} \iff$ candidate is below $\overline{\text{prev next}}$

(iii) *Backtracking removes all points on H_X not on H_{X+1}.*
Backtracking continues until candidate is on H_{X+1} or hull is empty. By the previous two propositions, this removes all such points.

Finally, h_{N+1} is added to hull (line 19), and hull contains H_{X+1}.

$\therefore \mathsf{hull} \equiv H_X$. □

Lemma 13. ALLHULLS *on a miss curve M over domain $[0,S]$ produces POIs for which H_X, the convex hull over $[0,X]$, is equivalent to:*

$$h_i = \begin{cases} (0, M(0)) & \text{if } i \equiv 1 \\ \text{NEXTPOI}(h_{i-1}) & \text{otherwise} \end{cases}$$

Proof. Consider how NEXTPOI works. It simply filters pois to the set

$$\begin{aligned} V = &\{p : p \in \mathsf{pois} \text{ and } p.x < X \text{ and } p.horizon > X\} \\ &\cup \{(X, M(X), \infty)\} \end{aligned}$$

By Lemma 12, we must show V is the same as the contents of hull at iteration $x \equiv X$.

First note that whenever points are removed from hull, they are either removed from pois (line 13) or their horizon is set (line 15). If a point obsolesces at $x \equiv \chi$, then its horizon is always $\chi - 1$.

$V \subset H_X$: Show that all points in $[0,X]$ not in H_X are either deleted from pois or have their horizon set smaller than X.

By Lemma 12, at completion of iteration $x \equiv X$, hull contains H_X. Therefore all points not on H_X have been removed from hull and are either deleted from pois or have horizon set to a value less than X.

$H_X \subset V$: Show that no point in H_X is deleted from pois or has a horizon set smaller than X.

Points are only deleted or have their horizon set when they are removed from hull. By Lemma 12, no point on H_X is removed from hull before $x \equiv X$. So the earliest a point in H_X could be removed from hull is at $X+1$, and its horizon would be X.

Furthermore no point earlier than X will be deleted at $X+1$, because ALLHULLS only deletes a point $(a, M(a))$ if it obsolesces at $x \equiv a+1$. If the point $(X, M(X))$ is deleted from pois, then NEXTPOI will replace it (via the union in $V = ...$).

$\therefore H_X = V$. □

Lemma 14. *current[p] in Algorithm 2 always lies on the convex hull over $[0,S]$ over all iterations.*

Proof. current$[p]$ is initially set to pois$[p]$.HEAD, which lies on all convex hulls for M_p. It is subsequently updated by next = NEXTPOI(current$[p]$). Thus for some i, current$[p] = h_i$ as in Lemma 13, and current$[p]$ lies on the convex hull over $[0,S]$. Note that S is decreasing over iterations of the main loop, so by Lemma 11 it is on the hull for all iterations. □

Theorem 15. PEEKAHEAD *is a lookahead algorithm.*

Proof. By Lemma 14, we know that NEXTPOI is called always starting from a point on the convex hull over remaining capacity, $[0,S]$. By Lemma 10 and Lemma 13, NEXTPOI returns the maximal utility per unit move for the partition. By Lemma 11 we know that so long as $s \leq S$, this move remains on the convex hull and by Lemma 10 remains the maximal utility per unit move. If $s > S$ then PEEKAHEAD recomputes the best move.

Because heap is a max-heap, heap.POP is the maximal utility per unit move across partitions. Thus each move made by PEEKAHEAD is a lookahead move. PEEKAHEAD does not terminate until $S \equiv 0$.

Furthermore, PEEKAHEAD is guaranteed forward progress, because at most P moves can be skipped before a valid move is found and S is reduced.

Therefore for all inputs PEEKAHEAD produces a lookahead sequence. □

Asymptotic run-time A.3

ALLHULLS has asymptotic run-time of $O(S)$, using the standard argument for the three-coins algorithm. All vector operations take constant time. Each point is added at most once in the main loop to pois and hull, giving $O(S)$. Finally backtracking removes points from hull at most once (and otherwise checks only the tail in $O(1)$), so *over the lifetime of the algorithm* backtracking takes $O(S)$. The overall run-time is thus $O(S+S) = O(S)$.

Now consider the run-time of PEEKAHEAD. Line 22 alone is $O(P \cdot S)$ (P invocations of ALLHULLS). The run-time of the remainder is complicated by the possibility of skipping steps (line 40). We first compute the run-time assuming no steps are skipped. In this case, NEXTPOI iterates over POIs at most once during the lifetime of the algorithm in $O(P \cdot S)$ total. PUSH (line 36) takes $O(\log P)$, and the main loop invokes $O(S)$ iterations for $O(\log P \cdot S)$. Thus altogether the run-time is $O(P \cdot S + \log P \cdot S) = O(P \cdot S)$.

If steps are skipped, then NEXTPOI can take $O(S)$ per skip to find the next POI. Further, $O(P)$ steps can be skipped per decrement of S. So it may be possible to construct worst-case miss curves that run in $O(P \cdot S^2)$. Note, however, that the number of skips is bounded by the number of concave regions. Miss curves seen in practice have at most a few concave regions and very few skips. The common case run-time is therefore bounded by ALLHULLS at $O(P \cdot S)$.

Indeed, Table 3.4 shows that the main loop (lines 38-44) runs in sub-linear time and ALLHULLS dominates, taking over 99% of execution time for large problem sizes. Contrast this with the common-case $O(P \cdot S^2)$ performance of UCP.

Analysis of geometric monitors A.4

Let s be the number of lines in the GMON, k_0 be its sampling rate, γ be the rate at which the sampling rate decreases per way, and W be the number of ways. At way w, the GMON models $m_w = s/(W \cdot k_0 \cdot \gamma^w) = m_0/\gamma^w$ lines. Thus the modeled lines per way increase geometrically, and a GMON with W ways models in total:

$$\text{Modeled capacity} = \sum_{w=0}^{W} m_w = \sum_{w=0}^{W} \frac{m_0}{\gamma^w} = \frac{m_0}{\gamma^W} \cdot \frac{1-\gamma^{W+1}}{1-\gamma}$$

The last factor tends to $1/(1-\gamma)$ very quickly with W, so modeled capacity grows $O(1/\gamma^W)$, while the low end maintains fidelity of m_0 according to the choice of k_0.

GMONs thus allow us to model a large cache *and* make efficient, small allocations. GMONs curves are sparse, containing only W points (where W is much smaller than in conventional UMONs of similar fidelity). This reduces software overheads considerably, since Peekahead's run-time is proportional to curve size (Appendix A.3).

EVA Appendices B

Quantifying reference model error B.1

The iid reuse distance model assumes each reference's reuse distance is distributed independently of the others, but this is not so (see Appendix 5.1). We can get a sense for the magnitude of the error through some simple calculations. We will focus on a single set, assuming that considering more than a single set is impossible for a practical replacement policy. The error arises because hits to different lines are disjoint events, but the iid reuse distance model treats them as independent. Hence the number of hits in the iid model follows a binomial distribution [19], and multiple hits can occur on a single access. Suppose the hit rate per line is a constant g, the cache size is S, and the associativity is W. In reality, the miss probability is $1-Wg$ and the hit probability for a single set is Wg. But using a binomial distribution, the miss probability is $(1-g)^W$ and single hit occurs with probability $Wg(1-g)^{W-1}$. Hence, the models do not agree on the probabilities of these events. The error term is the "missing probability" in the binomial model, i.e. the probability of more than one hit. To quantify this error, consider the linear approximation of the binomial distribution around $g = 0$:

$$(1-g)^W = 1 - Wg + O(g^2)$$
$$Wg(1-g)^{W-1} = Wg + O(g^2)$$

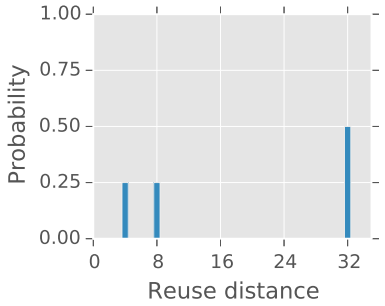
When g is small, $O(g^2)$ is negligible, and the binomial distribution approximates the disjoint event probabilities. Since the average per-line hit rate is small, at most $1/S \ll 1/W$, the error from assuming independence is negligible.

An explicit counterexample to the informal principle of optimality B.2

Figure B.1 lays out in detail a counterexample to the informal principle of optimality. It shows a particular case where evicting lines by their expected time until reference lowers the cache's hit rate. This example is somewhat contrived to make it as simple as possible, but as our evaluation shows (Figure 5.12), the central problem arises in real programs as well.

We consider a cache consisting of a single line that has the option of bypassing an incoming accesses. The replacement policy always chooses between two candidates: *(i)* the currently cached line at age a, or *(ii)* the incoming reference. Hence all deterministic replacement policies degenerate to choosing an age at which they evict lines, a_E.

Moreover, it is trivial to compute the cache's hit rate as a function of a_E. All reuse distances less than



(a) Reuse distance distribution

Policy Retain for…	**Hit rate** Hit prob. / Avg. lifetime
4 accesses	$\frac{1/4}{4} = \frac{1}{16}$
8 accesses	$\frac{1/2}{4\times1/4+8\times3/4} = \frac{1}{14}$
32 accesses	$\frac{1}{4\times1/4+8\times1/4+32\times1/2} = \frac{1}{19}$

(b) Hit-rate of policies

	Expected time until reference
New access	$4\times1/4+8\times1/4+32\times1/2 = 19$
Line at age 4	$(8\times1/3+32\times2/3)-4 = 25.33\ldots$

(c) Expected time until reference

Figure B.1: Counter-example to the informal principle of optimality, where replacing lines by their expected time until reference leads to sub-optimal hit rate. (a) Reuse distance probability distribution. (b) Hit rate of different policies; retaining lines for 8 accesses is optimal. (c) Expected time until reference for different lines; the informal principle says to evict lines after 4 accesses (since 25.33…>19), and is not optimal.

or equal to a_E hit, and all others are evicted. The hit rate is:[1]

$$\mathrm{P}[\text{hit}] = \frac{\mathrm{P}\left[D \le a_E\right]}{a_E \cdot \mathrm{P}\left[D > a_E\right] + \sum_{a=1}^{a_E} a_E \cdot \mathrm{P}[D = a]} \tag{B.1}$$

We consider Equation B.1 on an access stream with a trimodal reuse distance distribution that takes three distinct values at different probabilities. (Three values are needed to construct a counterexample; the informal principle of optimality is correct on bimodal reuse distance distributions.) The example we choose is shown in Figure B.1a. This reuse distance distribution is:

$$\mathrm{P}[D = d] = \begin{cases} 0.25 & \text{If } d = 4 \\ 0.25 & \text{If } d = 8 \\ 0.5 & \text{If } d = 32 \end{cases}$$

The sensible age at which to evict lines are thus either 4, 8, or 32. Figure B.1b computes the hit rate for each. The best choice is to evict lines after 8 accesses, i.e. $a_E = 8$. Figure B.1c shows the expected time until reference for new accesses and those at age 4. Accesses have a lower expected time until reference when new than at age 4, so the informal principle would choose $a_E = 4$. This choice degrades the hit rate, and the informal principle of optimality is therefore wrong.

The reason why the informal principle makes this decision is that it incorrectly uses the long reuse distance (i.e., 32) when computing the expected time until reference of lines at age 4. This is incorrect because if the cache does the right thing—evict lines at $a_E = 8$—then lines will never make it to age 32. Hence its not clear what 32 is doing influencing replacement decisions.

If the maximum age reached in the cache is 8, then only events that occur before age 8 should be

[1] Equation B.1 is quite similar to the model PDP uses to determine protecting distances, but note that this formula does not generalize well to caches with more than one line (Appendix C.6).

taken under consideration. Our contribution is to produce a framework that generalizes to these cases.

How limited space constrains replacement B.3

We assume inclusive caches where every access results in either a hit or eviction. In other words, no accesses bypass the cache. Thus, by definition, every access starts a new lifetime. If one visualizes the contents of an S-line cache over T accesses, then there are $S \cdot T$ line-access *cells*—i.e., a cell is a line's contents at a particular time (Figure 5.2). Each line is broken into lifetimes by hits and evictions. We can now show the desired result by counting cells across lifetimes.

Proposition 1. *The cache size is equal to the expected lifetime length:*

$$S = \sum_{a=1}^{\infty} a \cdot \mathrm{P}[L = a] \tag{B.2}$$

Proof. Each lifetime of length a takes a cells by definition. Let n_a be the number of lifetimes of length a after T accesses. In total, there are $S \cdot T$ cells. Since every cell is part of some lifetime:

$$S \cdot T = \sum_{a=1}^{\infty} n_a \cdot a$$

Now divide through by T and observe, since all T accesses are part of some lifetime, that $\lim_{T\to\infty} \frac{n_a}{T} = \mathrm{P}[L = a]$. □

A Cache Model for Modern Processors C

UNDERSTANDING the LLC's behavior is critical to achieve system objectives. Accurate predictions of cache behavior enable a large number of optimizations, including of single-threaded performance [14, 41], shared cache performance [12, 110, 124, 137], fairness [110, 116], quality of service [12, 77], security [115], etc.. These optimizations can be performed in many ways, e.g. by explicitly partitioning the shared cache in hardware [6, 101, 130] or software [28], or through job scheduling to avoid interference in cache accesses [102, 162, 166].

Unfortunately, cache behavior is difficult to predict because it depends on many factors, both of the application (e.g., its access pattern) and the cache (e.g., its size, associativity, and replacement policy). Existing analytical cache models [1, 41, 104, 132, 160] tend to focus on traditional, set-associative caches using simple replacement policies like least-recently used (LRU), pseudo-LRU, or random replacement. *But modern processors do not use LRU (or pseudo-LRU) for the LLC.*

Modern LLCs instead employ high-performance replacement policies that greatly improve cache performance over traditional policies like LRU (Appendix 2.2). These designs are available in commercial processors [65, 156], and a new model is needed to understand their behavior.

We present a cache model that accurately predicts the behavior of high-performance replacement policies on modern LLCs. Our model leverages two key observations: First, each core's private cache hierarchy filters accesses before they reach the LLC, capturing most temporal locality [14, 80, 83]. Thus, LLC accesses are free of short-term temporal correlations. (This is also why LRU is a poor policy for the LLC; LRU relies on temporal locality, and at the LLC, there is little.) Second, modern LLCs use hashing to map lines to sets [84, 156], reducing hotspots. Existing models often focus on the effects of low associativity, i.e. conflict misses, which can be hard to capture. But with hashing, modern LLCs have near-uniform behavior across sets and high effective associativity [129], making conflict misses a second-order concern (Appendix 2.1).

These two observations mean that *modern LLCs can be modeled as a pseudo-random process*: by capturing highly correlated accesses, private caches essentially randomize the accesses seen by the LLC; and hashing and high effective associativity mean that the replacement candidates constitute a representative sample of cached lines. Our model therefore computes the probability of various events (hits and evictions) occurring to *individual lines* as they age. This approach lets us model arbitrary age-based policies, including familiar policies like LRU and several high-performance policies, and abstracts away the details of array organization.

These features set our model apart from existing models. Existing models target set-associative LRU caches, and many model the performance of *entire sets* using *stack distances*, which measure the number of unique addresses between references to the same line. Stack distances are meaningful for set-associative LRU caches, but have little meaning for modern LLCs. In particular, it is unclear how to model high-performance policies through stack distances (Appendix 2.4).

Our model is built from logically distinct components, each of which captures a separate aspect of

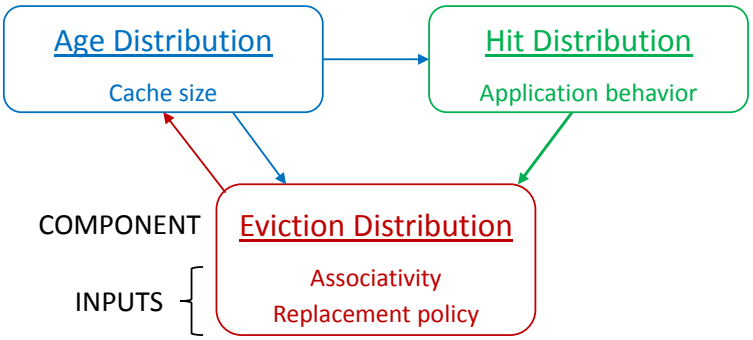


Figure C.1: Our model consists of three inter-dependent probability distributions. Each distribution captures distinct model inputs. Arrows denote dependence, e.g. A → B means "A depends on B".

cache behavior (Appendix C.2). For instance, to model a new replacement policy, our model only requires simple changes to a single parameter, the ranking function (Appendix C.3). We present an efficient, practical implementation of our model (Secs. C.4 and C.5) and thoroughly validate it against synthetic and real benchmarks (Appendix C.6). Finally, we present two case studies: *(i)* cache partitioning with high-performance replacement (Appendix C.7); and *(ii)* improving the replacement policy itself (Appendix C.9). In summary, we offer an efficient way to predict the performance of high-performance policies, allowing them to enjoy the many benefits that prior work has demonstrated for LRU.

C.1 Overview

Figure C.1 shows the high-level design of our cache model. The input to the model is the cache architecture (its size, associativity, and replacement policy) and a concise description of the access stream. Specifically, we describe the access stream by its *reuse distance distribution*; i.e., for each distance d, how many accesses have reuse distance d.

From these inputs, our model produces a concise description of the cache's behavior. Specifically, the model produces the cache's *hit and eviction distributions*; i.e., for each age a, how many accesses are hit or evicted at age a, respectively. The hit and eviction distributions directly yield the cache's performance: the cache's hit rate is the sum over the hit distribution. Additionally, they give a rich picture of the cache's behavior that can be used to improve cache performance(Appendix C.9).

Internally, the model uses three distributions: the hit and eviction distributions (already discussed) and the age distribution of cached lines (i.e., the probability that a randomly selected line has age a). We define age as the number of accesses since a line was last referenced. These distributions are interdependent and related to one another by simple equations. Each incorporates a particular model input and conveys its constraints: *(i)* the age distribution incorporates the cache size, and constrains modeled capacity; *(ii)* the hit distribution incorporates the access stream and constrains when lines can hit; and *(iii)* the eviction distribution incorporates the replacement policy and constrains how long lines stay in the cache. The model is solved by iterating to a fixed point. When the distributions converge, the hit and eviction distributions accurately describe the cache's behavior.

We build the model in stages. First, we introduce the model on a specific example. Second, we develop the model formulation for LRU. Third, we generalize it to other policies. Finally, we show to solve the model efficiently.

Request	A	A	B	C	B	D	B	C
Lines	A	A				D		
			B		B		B	
				C				C
Time	1	2	3	4	5	6	7	8

Table C.1: Steady state behavior for a 3-line LRU cache on a simple, repeating access pattern. Live lines are colored green, and dead lines red.

Request	A	A	B	C	B	D	B	C
Ages	1	1	2	3	*4*	1	2	*3*
	3	4	1	2	1	2	1	2
	2	3	4	1	2	3	4	1
Time	1	2	3	4	5	6	7	8

Table C.2: Ages per line in Table C.1 (hits underlined, *evictions* italic).

Example C.1.1

Table C.1 shows the behavior of a 3-line LRU cache on a simple, repeating access pattern. Time (measured in accesses) increases from left to right and wraps around (e.g., time 9 would be identical to 1). Live lines, those that eventually hit, are colored green; dead lines, those that do not, are red. For example, the first line is live (green) at time 1 because A hits at time 2. However, D evicts A at time 6, so A is dead (red) in 2–5. Similarly, A evicts D at time 1, so D is dead in 6–9. B and C always hit, so they are always live. This divides the cache into *lifetimes*, the intervals from last use until hit or eviction. For example, A's lifetime starting at time 1 ends after a single access when A hits at time 2; this starts a new lifetime that ends after four accesses when D evicts A at time 6.

We can redraw Table C.1 showing the ages of each line at each access (Table C.2). In steady state, 6 out of 8 accesses are hits. These hits (underlined) come at ages 1 (A once), 2 (B twice), and 4 (C twice and B once). The other 2 accesses are evictions (*italic*), at ages 3 (D) and 4 (A).

This information lets us compute the hit and eviction distributions, denoted by random variables H and E, respectively. These distributions yield the probability that a (randomly selected) access will hit or be evicted at a given age. For example, one quarter of accesses hit at age 2 (B at times 4 and 6), so the hit distribution at age 2 is $P_H(2) = 1/4$. Table 5.1 summarizes our notation and the most important distributions in the model.

Table C.3 gives the three model distributions for the example in Table C.1. For the hit and eviction distributions, we turn counts into probabilities by dividing by 8 accesses. Since every lifetime ends

Age	a	1	2	3	4	5	6	7	8	**Sum**
Distribution	$P_D(a)$	1/8	2/8	–	3/8	–	–	1/8	1/8	**1**
	$P_A(a)$	1/3	7/24	5/24	1/6	–	–	–	–	**1**
	$P_H(a)$	1/8	1/4	–	3/8	–	–	–	–	3/4
	$P_E(a)$	–	–	1/8	1/8	–	–	–	–	1/4

Table C.3: Steady-state distributions for the cache in Table C.1.

in a hit or eviction, the hit and eviction distributions together sum to 1, but separately sum to less than 1. We express the distribution inclusive of both hits and evictions as the *lifetime distribution*, $P_L(a) = P_H(a) + P_E(a)$ (not shown in Table C.3). L is the age at which a line is either hit or evicted, i.e. the age when its lifetime ends. Finally, we can compute the age distribution by counting the ages of lines and dividing by 3 lines × 8 accesses = 24.

These distributions tell us how the cache performs on this access pattern: its hit rate is the sum over the hit distribution, 3/4. Moreover, the distributions also say how lines behave: e.g., no lines make it past age 4, despite some having reuse distance of 8. Such information will prove valuable in applying the model (Appendix C.9).

In this example, we have computed the distributions by brute force, but this method is too expensive for our purposes. We instead model the relationships among distributions analytically and solve them by iteration. This example gives the intuition behind the model, but note that *our independence assumptions are accurate only for large caches*. So while our model is accurate on real systems (Appendix C.6), some model components are inaccurate on simple examples like Table C.1.

C.2 Basic model (for LRU)

This section presents the model for caches with LRU replacement. We present the complete equations for the age and hit distributions, and the eviction distribution equations for LRU replacement. Appendix C.3 extends the eviction distribution to model arbitrary age-based replacement policies.

C.2.1 Model assumptions

First, we make a few simplifying assumptions that make cache modeling tractable, chiefly about the independence of certain events. These assumptions are motivated by the properties of modern LLCs (Appendix 2.1).

We assume each access has reuse distance d distributed identically and independently according to the distribution $P_D(d)$. This assumption is not exactly correct in practice, but since private caches filter accesses before they reach the LLC, it is a good approximation for large caches ([14], Appendix C.6). In return, it greatly simplifies the probability calculations, allowing a simple model to capture diverse access patterns.

Similarly, we model an idealized, *random-candidates* cache, where replacement candidates are drawn at random from cached lines. This is a good model for modern LLCs, where the replacement candidates form a representative sample of cached lines. The random-candidates model is a direct analog of skew-associative caches or zcaches, but is also a good approximation for hashed, set-associative caches with many ways [129]. Although the accuracy of this assumption depends slightly on the cache array architecture, it is a reasonable simplifying assumption for modern LLCs.

C.2.2 Age distribution

The age distribution is used internally by the model to constrain cache capacity. It is presented first because it is the simplest to compute from the other distributions.

Since ages measure the time since a line was last referenced, a line reaches age a if and only if it is not hit or evicted for at least a accesses. Hence the probability of a line having age a, $P_A(a)$, is proportional to the fraction of lines that survive at least a accesses in the cache without being referenced. The probability of surviving *exactly* x accesses is given by the lifetime distribution at age x, $P_L(x) = P_H(x) + P_E(x)$, and the probability of surviving *at least* a accesses is the probability over all $x \geq a$. The age distribution $P_A(a)$ is thus proportional to $P[L \geq a]$, which determines the age distribution up to a constant factor.

To find the constant factor, note that every access necessarily produces a line of age 1 (whether a hit or miss), since we define ages as the number of accesses since a line was last referenced. Since ages increase upon each access, there is always exactly one line of age 1 in the cache. Hence if the cache has S lines, the age distribution at age 1 is $\mathrm{P}_A(1) = 1/S$.

Combining the two results, we find the age distribution for a cache of S lines is:

$$\mathrm{P}_A(a) = \frac{\mathrm{P}[L \geq a]}{S} = \sum_{x=a}^{\infty} \frac{\mathrm{P}_H(x) + \mathrm{P}_E(x)}{S} \tag{C.1}$$

For example, in Appendix C.1.1, by counting cells we find that $\mathrm{P}_A(3) = 5/24$. That is, of the twenty-four cells depicted in Table C.2, age 3 appears five times. Equation C.1 gives another way to arrive at the same conclusion without counting cells: The probability of a hit or eviction at age 3 or greater is $3/8 + 1/8 + 1/8 = 5/8$ (Table C.3). Dividing by $S = 3$ gets $\mathrm{P}_A(3) = 5/24$, as desired. This argument can be generalized to yield an alternative derivation of Equation C.1 and show $\sum_{a=1}^{\infty} \mathrm{P}_A(a) = 1$ (Appendix B.3).

Hit distribution C.2.3

We now show how to compute when hits occur for a given access pattern, again assuming the other distributions are known. The hit distribution is perhaps the most important product of the model, since it yields the cache's hit rate.

A line will eventually hit if it is not evicted. Intuitively, a line's hit probability depends on its reuse distance (longer distances have greater opportunity to be evicted) and the cache's eviction distribution (i.e., at what age does the replacement policy evict lines?). Moreover, by definition, a line with reuse distance d must hit at age d, if it hits at all. Thus the hit probability is just the reuse distance probability minus the probability of being evicted, or:

$$\begin{aligned} \mathrm{P}_H(d) &= \mathrm{P}_D(d) - \mathrm{P}[\text{evicted}, D = d] \\ &= \mathrm{P}_D(d) - \sum_{a=1}^{d-1} \mathrm{P}_{D,E}(d, a) \end{aligned} \tag{C.2}$$

It may be tempting to simply subtract the eviction probability below d in the above equation. That is, $\mathrm{P}_H(d) = \mathrm{P}_D(d) \times (1 - \mathrm{P}[E < d])$. This is incorrect; eviction age and reuse distance are not independent.

To proceed, we require the critical insight that candidates of age a look alike to the replacement policy. In other words, the replacement policy does not know candidates' reuse distances, only that lines of age a have reuse distance $d > a$. Evicting at age a thus only implies that the line's reuse distance was at least a, i.e. $E = a \Rightarrow D > a$. From this insight:

$$\begin{aligned} \mathrm{P}_{D,E}(d, a) &= \mathrm{P}_D\big(d | E = a\big)\, \mathrm{P}_E(a) \\ &= \mathrm{P}_D\big(d | D > a\big)\, \mathrm{P}_E(a) && \text{(Insight)} \\ &= \frac{\mathrm{P}_D(d) \cdot \mathrm{P}_E(a)}{\mathrm{P}[D > a]} && \text{(Simplify)} \end{aligned} \tag{C.3}$$

This insight assumes that other, spurious correlations between eviction age and reuse distance can be safely ignored. Hence, Equation C.3 is accurate for the large LLCs seen in real systems (Appendix C.6), but it is inaccurate for the small example in Appendix C.1.

Finally, we substitute this result into the above equation: Summing over all a below d counts all the lines of reuse distance d that are evicted before hitting. Since the rest hit, subtracting these from $\mathrm{P}_D(d)$

yields the hit probability:

$$\mathrm{P}_H(d) = \mathrm{P}_D(d) \times \left(1 - \sum_{a=1}^{d-1} \frac{\mathrm{P}_E(a)}{\mathrm{P}[D > a]}\right) \tag{C.4}$$

C.2.4 Eviction distribution in LRU

The eviction distribution accounts for the replacement process and is the most complex part of the model. It models both the selection of replacement candidates (i.e., associativity) and the selection of a victim among candidates (i.e., the replacement policy). For clarity, we begin by presenting the model for LRU only.

To compute the eviction distribution, we assume that candidates are drawn randomly from the cache, as discussed above. Among these, LRU simply evicts the oldest, so the eviction probability at age a is the probability that both an access misses and a is the oldest age among candidates. We also assume that cache misses and the oldest age are independent. This is reasonable because a large cache is negligibly affected by the behavior of a few candidates.

Hence to find the eviction probability at age a, we need to know the probabilities that the access misses and the oldest candidate has age a: $\mathrm{P}_E(a) = \mathrm{P}[\text{miss}] \cdot \mathrm{P}_{\text{oldest}}(a)$. The first factor is trivial, since

$$\mathrm{P}[\text{miss}] = 1 - \mathrm{P}[\text{hit}] = 1 - \sum_{a=1}^{\infty} \mathrm{P}_H(a) \tag{C.5}$$

The challenge lies in finding the distribution of oldest ages, $\mathrm{P}_{\text{oldest}}(a)$. Replacement candidates are drawn randomly from the cache, so each has age identically and independently distributed according to $\mathrm{P}_A(a)$. The oldest age is just the maximum of W iid random variables, where W is the associativity. The maximum of iid random variables is a well-known result [50]: the oldest age among replacement candidates is less than a iff all candidates are of age less than a. Thus given W candidates, $\mathrm{P}[\text{oldest} < a] = \mathrm{P}[A < a]^W$. To get the distribution from the cumulative distribution, differentiate:

$$\mathrm{P}_{\text{oldest}}(a) = \mathrm{P}[\text{oldest} < a+1] - \mathrm{P}[\text{oldest} < a] \tag{C.6}$$

Altogether, the eviction distribution at age a for LRU is:

$$\begin{aligned} \mathrm{P}_E(a) &= (1 - \mathrm{P}[\text{hit}])\, \mathrm{P}_{\text{oldest}}(a) \\ &= (1 - \mathrm{P}[\text{hit}]) \left(\mathrm{P}[\text{oldest} < a+1] - \mathrm{P}[\text{oldest} < a]\right) \\ \Rightarrow \mathrm{P}_E(a) &= (1 - \mathrm{P}[\text{hit}]) \left(\mathrm{P}[A < a+1]^W - \mathrm{P}[A < a]^W\right) \end{aligned} \tag{C.7}$$

C.2.5 Summary

Equations C.1, C.4, and C.7 form the complete model for LRU replacement. The age distribution incorporates the cache size and depends on the hit and eviction distributions. The hit distribution incorporates the access stream and depends on the eviction distribution. The eviction distribution incorporates the cache's associativity and replacement policy (LRU) and depends on the hit and age distributions.

Other replacement policies C.3

We now extend the eviction distribution to support arbitrary age-based policies, like those discussed in Appendix 2.2.

Ranking functions C.3.1

To support other policies, we must abstract the replacement policy in a way that can be incorporated into the model. We do so through a *ranking function*, $R : \text{age} \to \mathbb{R}$, which gives an eviction priority to every age. By convention, higher rank means higher eviction priority.

Ranking functions capture many existing policies. For example, LRU's ranking function is $R_{\text{LRU}}(a) = a$ (or any other strictly increasing function). This represents LRU because it ensures that older lines will be preferred for eviction. Similarly, a constant ranking function produces random replacement, e.g. $R_{\text{random}}(a) = 0$. Ranking functions can also capture many high-performance replacement policies.

PDP [44] protects lines up to an age d_p, known as the protecting distance. It prefers to evict lines older than the protecting distance, but if none are available among candidates, it evicts the youngest line. Thus PDP's ranking function decreases up to the protecting distance (d_p), upon which it jumps to a large value and increases thereafter:

$$R_{\text{PDP}}(a) = \begin{cases} d_p - a & \text{If } a < d_p \\ a & \text{If } a \geq d_p \end{cases} \tag{C.8}$$

IRGD [141] ranks lines using an analytical formulation based on the reuse distance distribution. IRGD essentially ranks lines by their expected reuse distance, but since in practice very large reuse distances can't be measured, IRGD uses a weighted harmonic mean instead of the conventional, arithmetic mean. This lets it ignore immeasurably large reuse distances, since they have small impact on the harmonic mean.[1] Its ranking function is:

$$R_{\text{IRGD}}(a) = \text{P}[D > a] \times \left(\sum_{x=1}^{\infty} \frac{\text{P}_D(a+x)}{a+x} \right)^{-1} \tag{C.9}$$

Rank functions thus model any age-based policy, but not all high-performance policies are strictly age-based. Our model can support such policies (e.g., RRIP [69, 159]) with specialized extensions. However, this paper presents the general framework, and we leave extensions to future work.

From the ranking function and age distribution, we can produce a *rank distribution* that gives the probability a line will have a given rank. It is then possible to generalize Equation C.7. While LRU evicts the oldest replacement candidate, in general the cache evicts the maximum rank among candidates.

Generalized eviction distribution C.3.2

Generalizing the eviction distribution is a straightforward substitution from "oldest age" in LRU to "maximum rank" in general. If a line of age a is evicted, then the maximum rank among candidates must be $R(a)$. Additionally, R may rank several ages identically (i.e., $R(a) = R(b)$), so we must ensure that the candidate had age a (not age b).

This consideration is important because, in practice, continuous ranks are quantized in units of Δr, increasing the possibility that several ages map to indistinguishable ranks. For example, if ranks can take

[1] Takagi et al. [141] express IRGD somewhat differently, but the two formulations are equivalent.

values in $[0, 256)$ (e.g., LRU with 8-bit ages), then an efficient model implementation might quantize ranks into regions as $[0, 8), [8, 16) \ldots [248, 256)$. Each region has size $\Delta r = 8$, and many ages may have the "same rank" as far as the model is concerned.

We account for indistinguishable ranks by using the joint distribution of rank and age to avoid double counting:

$$\mathrm{P}_E(a) = (1 - \mathrm{P}[\text{hit}]) \cdot \mathrm{P}_{\text{max rank},A}\big(R(a), a\big) \tag{C.10}$$

The joint distribution is in turn:

$$\mathrm{P}_{\text{max rank},A}\big(R(a), a\big) = \mathrm{P}_{\text{max rank}}\big(R(a)\big) \cdot \frac{\mathrm{P}_A(a)}{\mathrm{P}_{\text{rank}}\big(R(a)\big)} \tag{C.11}$$

$\mathrm{P}_A(a)/\mathrm{P}_{\text{rank}}\big(R(a)\big)$ is the fraction of ranks belonging to age a in the rank quantum containing $R(a)$ (roughly its Δr-neighborhood). Multiplying by this fraction eliminates double counting. This equation should simply be thought as the analog to $\mathrm{P}_{\text{oldest}}(a)$ in LRU.

As in LRU, the challenge lies in finding $\mathrm{P}_{\text{max rank}}(r)$. To start, we compute the *rank distribution* in the cache from the age distribution. Since ranks depend on age, the probability that a line's rank equals r is just the total probability of ages with rank r:

$$\mathrm{P}_{\text{rank}}(r) = \sum_{a:R(a)=r} \mathrm{P}_A(a) \tag{C.12}$$

Next, the cumulative distribution of maximum rank is computed just as $\mathrm{P}_{\text{oldest}}(a)$ in LRU as the maximum of iid rvs:

$$\mathrm{P}[\max \text{rank} < r] = \mathrm{P}[\text{rank} < r]^W \tag{C.13}$$

Finally, the distribution of maximum rank is obtained by discrete differentiation [25]:

$$\mathrm{P}_{\text{max rank}}(r) = \frac{\mathrm{P}[\max \text{rank} < r + \Delta r] - \mathrm{P}[\max \text{rank} < r]}{\Delta r} \tag{C.14}$$

(In LRU, the oldest age distribution uses $\Delta r = 1$.)

These formulae fill in all the terms to compute the generalized eviction distribution:

$$\mathrm{P}_E(a) = \big(1 - \mathrm{P}[\text{hit}]\big) \times \left(\frac{\mathrm{P}_A(a)}{\mathrm{P}_{\text{rank}}\big(R(a)\big)}\right) \times \left(\frac{\mathrm{P}\big[\text{rank} < R(a) + \Delta r\big]^W - \mathrm{P}\big[\text{rank} < R(a)\big]^W}{\Delta r}\right) \tag{C.15}$$

C.3.3 Discussion

The complete cache model is given by the age (Equation C.1), hit (Equation C.4), and eviction (Equation C.15) distributions. These equations describe a cache using an arbitrary, age-based replacement policy. Our model forms a system of equations that describe a valid solution, but does not yield this solution directly.

The implicit nature of our model has benefits. The equations organize the model into logical components. Each distribution is responsible for a specific model input: the age distribution for the cache size, the hit distribution for the access pattern, and the eviction distribution for replacement (both associativity and replacement policy). This makes it easy to adapt the model to new cache architectures. For example, a new replacement policy only requires a new ranking function, and all appropriate changes naturally propagate through the eviction, hit, and age distributions. Likewise, new applications change

Algorithm 4. The cache model simultaneously solves the cache's age, hit, and eviction distributions by iteration.

Inputs: S - Cache size; W - associativity; R - rank function; rdd - reuse distance distribution; prd - with classification, total probability of class (Appendix C.8), or 1 otherwise.
Returns: Hit and eviction distributions, hit and evict.

```
1:  function MODEL
2:      age, hit, evict, h' ← SEED                          ▷ Initialize distributions.
3:      while not CONVERGED:
4:          h ← h'                                          ▷ Hit rate from last iteration.
5:          h' ← 0                                          ▷ x' is solution of x for this iteration.
6:          crd ← rdd[1]                                    ▷ Cumulative D probability, P[D ≤ a].
7:          evBelow ← 0                                     ▷ Prob. line evicted at D=a in Equation C.4.
8:          age'[1] ← prd/S
9:          for a ← 1 to N:
10:             hit'[a] ← rdd[a] (1 − evBelow)
11:             evict'[a] ← (1−h) maxRankDist[R(a)] (age'[a] / rankDist[R(a)])
12:             age'[a+1] ← age'[a] − (hit'[a] + evict'[a])/S
13:             h' ← h' + hit'[a]
14:             evBelow ← evBelow + evict'[a]/(prd − crd)
15:             crd ← crd + rdd[a+1]
16:         age, hit, evict ← AVERAGE(age', hit', evict')
17:     return hit, evict
```

only the reuse distance distribution.

However the drawback is that, since these equations are not explicit, their solution is not entirely obvious. We solve the system through iteration to a fixed point, discussed next.

Model solution C.4

All components of the model are interdependent, and a general, closed-form solution is unlikely to exist. We solve the model by iterating to a fixed point, *simultaneously* solving the three distributions age by age (Algorithm 4). This simultaneous solution tightly couples the solution of each distribution to the others, maintaining their relationships. That is, each distribution is computed from the others as the solution evolves, rather than from the distributions at the last iteration. Only the hit rate and rank distribution are fixed across iterations. We find this tight coupling improves convergence time.

At each iteration, Algorithm 4 solves Equation C.1, Equation C.4 and Equation C.15 for age a in constant time, building on the solution from age $a-1$. Algorithm 4 uses the following recurrence relation derived from Equation C.1:

$$\mathrm{P}_A(a+1) = \mathrm{P}_A(a) - \frac{\mathrm{P}_H(a) + \mathrm{P}_E(a)}{S} \tag{C.16}$$

This allows the age distribution to be updated to reflect the hit and eviction distributions as the solution evolves, which in turn influences the solution of the hit and eviction distributions. The hit and eviction distributions are thus constrained, and negative feedback loops are imposed on over-estimation. Sums in other equations are similarly broken across iterations so that each age is solved in constant time. For example, the variable evBelow is the inner sum in Equation C.4.

We seed the first iteration with sensible but arbitrary parameters (e.g., hit rate of 50%). To avoid oscillating around a stable solution, in each iteration we average the old and new distributions using an exponentially weighted moving average. We have empirically determined that a coefficient of ⅓ yields good performance. We detect convergence when the hit rate stays within a 10^{-3} range for ten iterations. Finally, the model sets a floor of 0 for all probabilities during solution. In practice, Algorithm 4

reliably converges to the steady-state distributions after a few iterations (typically 20-40) on hundreds of thousands of distributions from real workloads (Appendix C.6).

While involved, iteration is computationally cheap: in practice, we use and monitor coarse ages (see below) for which N-point distributions with $N \approx 64$–256 suffice, and each iteration runs in linear time on the size of the distributions.

C.4.1 Convergence

Our model is designed with generality in mind, but this comes at the cost of complicating some theoretical properties. Equation C.1, Equation C.4, and Equation C.15 form a non-linear system (particularly Equation C.15) operating in many dimensions (N points per distribution and multiple distributions). Moreover, the model accepts arbitrary N-point vectors as input (the reuse distance distribution and ranking function). Demonstrating the convergence of fixed point iteration for non-linear systems is difficult. Generally, it involves reducing the system to a contraction mapping of some relevant model parameter [70]. Although several intuitive parameters are attractive (e.g., hit rate or modeled cache size), we cannot yet prove a contraction mapping on these parameters in general—indeed, it seems that for some degenerate ranking functions (not those in Appendix C.3.1), the model does not converge.

We instead take an empirical approach. We evaluate our model on a diverse suite of real applications and demonstrate its accuracy and utility in that context. Since the model is solved online at regular intervals, our evaluation represents hundreds of thousands of model solutions. Thus we conclude that the model converges on distributions seen in practice. We leave rigorous convergence conditions to future work.

C.4.2 Increased step size

Reuse distances in practice can be quite large, and naïve iteration over all ages would be quite expensive. Moreover, age-by-age iteration is wasteful, since there are large age ranges where few events occur or the event probabilities are roughly constant. Modeling such ranges in detail is unnecessary, since they can be approximated by assuming constant event probabilities throughout the range. This observation allows the solution to take a large step over many ages at once and greatly reduces N, the number of points that need to be solved in Algorithm 4. Reducing N is important when applying the model online, e.g. as part of a runtime system (Appendix C.7).

The concept is quite similar to adaptive step size in numerical solutions of differential equations [25]. For example, Figure C.2 shows a solution of all three model distributions for a synthetic benchmark (solid lines). These distributions are *coarsened* by increasing the step size (dashed lines), producing a good approximation with much less computation. Indeed, Figure C.2 shows that $N = 32$ is sufficient to model this access pattern, even though ages go up to 500. This is possible because there are large regions (e.g., ages 0-100) where few events occur. There is no reason to model these regions in great detail. Instead, we adaptively divide ages into regions, modeling regions of high activity at fine granularity and others at coarse granularity.

We then model the total probability within each coarsened region. For example, rather than computing the hit probability at a single age (e.g., $\mathrm{P}[H = a]$), we compute the hit probability over several ages (e.g., $\mathrm{P}[a \le H < b]$). Remarkably, the model equations are basically unchanged by coarsening. There is a deep reason for this; these equations are connected to a system of differential equations, which can be approximated at arbitrary step sizes (Appendix D). For example, if regions are split at ages $a_1, a_2, a_3, \ldots$

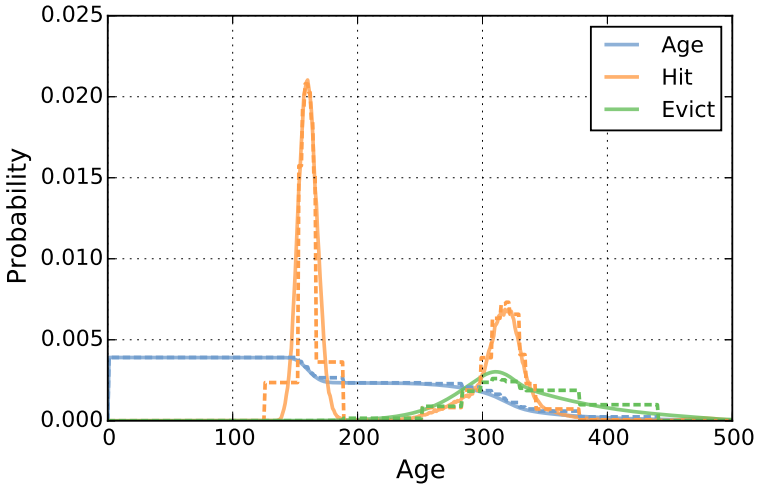


Figure C.2: Model solution with increased step size on a synthetic benchmark. Solid lines show a full-detail solution; dashed lines show a solution with N =32.

then the coarsened hit equation is:

$$\mathrm{P}\left[a_i \le H < a_{i+1}\right] \approx \mathrm{P}\left[a_i \le D < a_{i+1}\right] \times \left(1 - \sum_{j=1}^{i-1} \frac{\mathrm{P}\left[a_j \le E < a_{j+1}\right]}{\mathrm{P}\left[D > a_j\right]}\right) \tag{C.17}$$

This equation is identical in form to the fine-grain hit equation (Equation C.4), except now operating on regions rather than individual ages. Other model equations are similar (Appendix C.11).

Another important question is how to choose the regions. The choice must balance two competing goals: modeling regions with high activity at fine granularity, while modeling other ages in sufficient detail for the model to converge. We address this in two steps. First, we divide all ages evenly into $N/2$ regions. For example, with 8-bit ages and $N = 64$, we first create the 32 regions divided at ages: $0, 8, 16 \ldots 256$. Second, we further divide these regions $N/2$ times to try to equalize the probability of hits and evictions in each region. We sort regions by their probability of hits and evictions, and recursively divide the largest in equal-probability halves $N/2$ times. We find this procedure chooses regions that yield efficient and accurate solutions.

Implementation C.5

We now describe how to integrate our cache model into a full system which we evaluate in simulation. In our validation (Appendix C.6) and case studies (Appendix C.7 and Appendix C.9), the model is used to dynamically model or reconfigure the cache, as shown in Figure C.3. A lightweight hardware monitor samples a small fraction of accesses and produces the application's reuse distance distribution. Periodically (e.g., every 50 ms), a software runtime models the cache's behavior from the sampled reuse distance distribution, which is then used to predict the cache's behavior over the next interval.

This configuration represents just one use case for the model; it can also be used to model cache behavior offline. However, Figure C.3 is the most demanding use case, since it imposes the most stringent run-time requirements and the model must contend with sampling error.

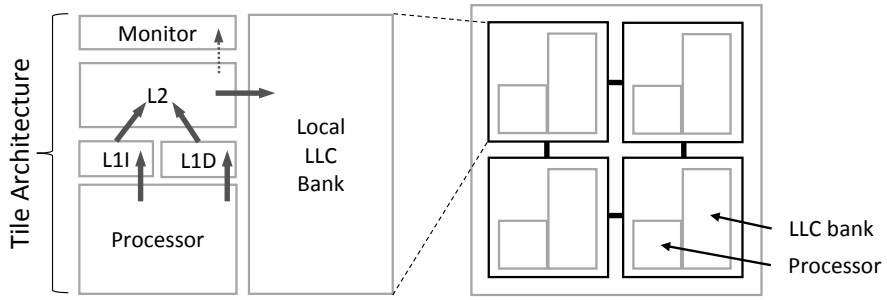


Figure C.3: An example implementation of our model. A lightweight hardware monitor on each tile samples a small fraction of LLC accesses. Our model runs periodically in software.

C.5.1 Application profiling

Our model works with several low-overhead, hardware monitoring schemes. For instance, PDP [44] proposes a FIFO that samples a small fraction of accesses and, when an access hits in the FIFO, records its depth in the FIFO as the reuse distance. Other monitors record stack distance and could be adapted to work with our model. For example, utility monitors [124] are a small, tag-only LRU cache that record stack distance. Geometric monitors [15] extend utility monitors to efficiently monitor very large stack distances. Stack distances can then be approximately translated into reuse distances in software [132]. Alternatively, the monitor itself can be modified to include timestamps and record reuse distance directly; we use such monitors in our validation. In all cases, these monitors impose small overheads, typically around 1% of LLC area.

Finally, other schemes can sample the reuse distance distribution without adding hardware. Software-only schemes can sample access patterns, e.g. through injecting page faults [167]. Offline profiling can record the access trace, e.g. through compiler hooks [41] or dynamic binary translation [145]. These schemes enable our model to work when hardware support is unavailable.

C.5.2 Overheads

Our model incurs modest run-time overheads and small monitoring overheads, similar to prior schemes. The model takes only a few arithmetic operations per age region per iteration ($\sim$25). With $N = 128$ and 30 iterations on average, the model completes in under 100 K arithmetic operations. Since the model runs infrequently (e.g., every 50 ms), this overhead is small ($<$1% of a single core's cycles). If this overhead is too large, N can be reduced or the reconfiguration interval can be increased, typically at little performance loss ([12, 15], Figure C.6). Alternatively, the model can be solved in the background with low-priority threads that interfere minimally with active applications [74]. Finally, computation can be reduced by specializing the model to particular ranking functions.

C.6 Validation

We now validate our model on synthetic and real benchmarks, showing that it is accurate over diverse replacement policies, access patterns, and cache sizes.

C.6.1 Synthetic

Figure C.4 compares the model against simulation of synthetic traces. These experiments demonstrate the model's accuracy in an environment that largely satisfies its assumptions (Appendix C.2.1). We

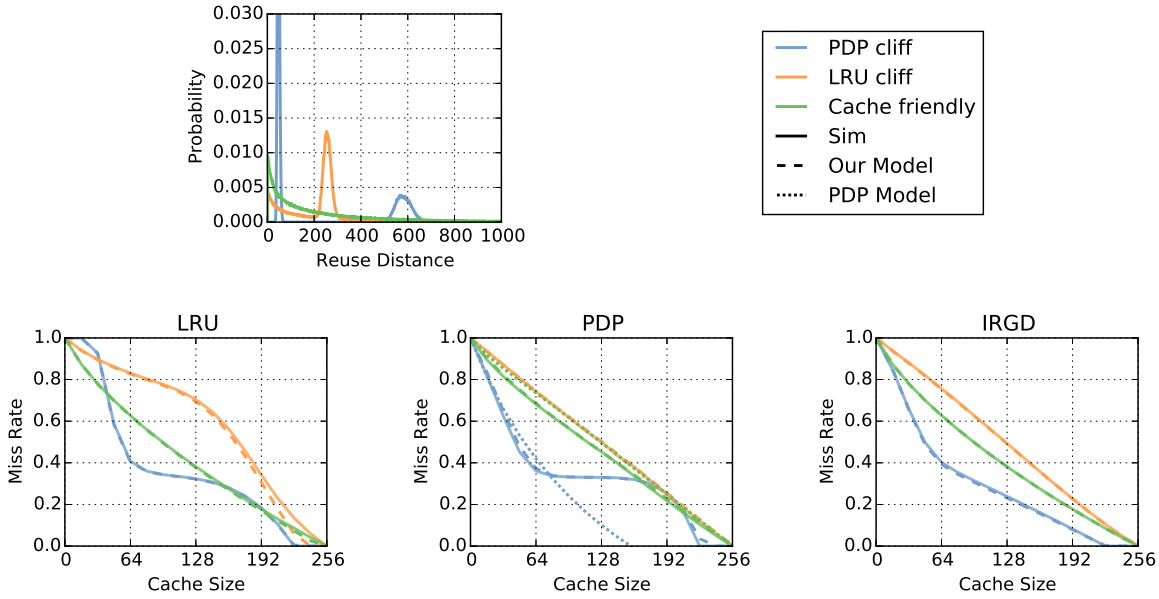


Figure C.4: The cache model on three synthetically generated traces driving small caches using LRU, PDP, and IRGD replacement. Simulation results are shown as solid lines; model predictions as dashed lines; PDP's model as dotted lines.

simulate a small cache that randomly selects replacement candidates; random-candidates is an idealized model of associativity [129] that matches model assumptions. Each trace is pseudorandomly generated to produce the desired reuse distance distribution.

Each trace represents a different access pattern: a cache-friendly pattern and patterns that expose cliffs in LRU and PDP. Their reuse distance distributions are shown on the left. On the right there is one graph per replacement policy. On each graph, simulation results are shown (solid lines) along with model predictions (dashed lines). The PDP graph also includes the predictions of PDP's analytical model (dotted lines).

Our model is accurate on every configuration. The dashed lines are often not visible because model predictions are indistinguishable from simulation results. By contrast, PDP's model exhibits significant error from simulation, failing to distinguish between the LRU-cliff and cache-friendly patterns, and mispredicting the PDP cliff badly.

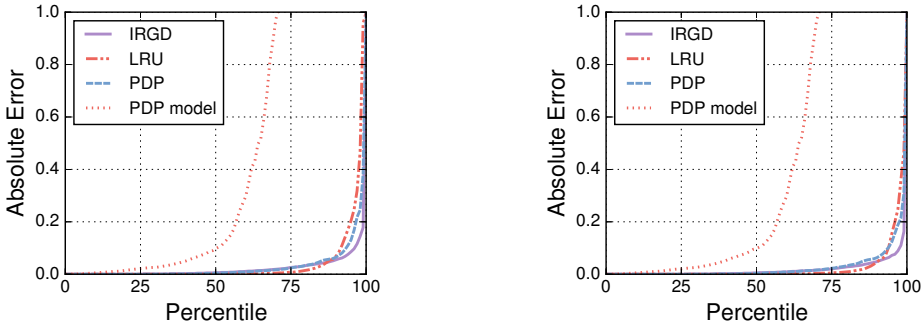
Execution-driven C.6.2

These results carry over to real benchmarks on a full system, where model assumptions only approximately hold.

Methodology: We use zsim [131] to evaluate our model. We perform execution-driven simulation of SPEC CPU2006 benchmarks on OOO cores using 16-way hashed, set-associative last-level caches and parameters given in Table C.4, chosen to mimic Ivy Bridge-EP. We run each benchmark for 10 B instructions after fast-forwarding 10 B, and we perform enough runs to achieve 95% confidence intervals $\leq$1%. All results hold for skew-associative caches [133], zcaches [129], and for systems with prefetching. This methodology also applies to later case studies.

Cores	Westmere-like OOO [131], 2.4 GHz
L1 caches	32 KB, 8-way set-associative, split D/I, 1-cycle latency
L2 caches	128 KB priv. per-core, 8-way set-assoc, inclusive, 6-cycle
L3 cache	Shared, non-inclusive, 20-cycle; 16-way, hashed set-assoc
Coherence	MESI, 64 B lines, no silent drops; sequential consistency
Memory	200 cycles, 12.8 GBps/channel, 1 channel

Table C.4: Configuration of the simulated systems.



(a) 128-point solution. (b) Full solution.

Figure C.5: Model error distribution over 250 K-access intervals. Our model is accurate, while PDP's model (dotted line) has large error.

We evaluate a large range of cache sizes, from 128 KB to 128 MB, and solve the model every 250 K accesses from the sampled reuse distance distribution in that interval. This yields many samples—over 400 K model solutions in all.

Results: Figure C.5 shows the distribution of modeling error (|predicted hit rate − actual hit rate|) in each interval for LRU, PDP, and IRGD. We show results *(a)* for coarsened solutions with $N = 128$ (Appendix C.4.2) and *(b)* full solutions. For 128-point solutions, median error is 0.1%/0.1%/0.6% for LRU/PDP/IRGD, respectively; mean error is 3.3%/3.7%/2.2%; and 90th percentile error is 7.5%/9.9%/6.1%. For full solutions, median error is 0.1%/1.1%/0.6%; mean error is 2.2%/3.5%/1.9%; and 90th percentile error is 4.8%/9.9%/5.5%.

Overall, the model is accurate, and there is modest error from coarsening solutions. Figure C.6 shows the mean and 90th percentile error for different values of N. (Median error is negligible in all

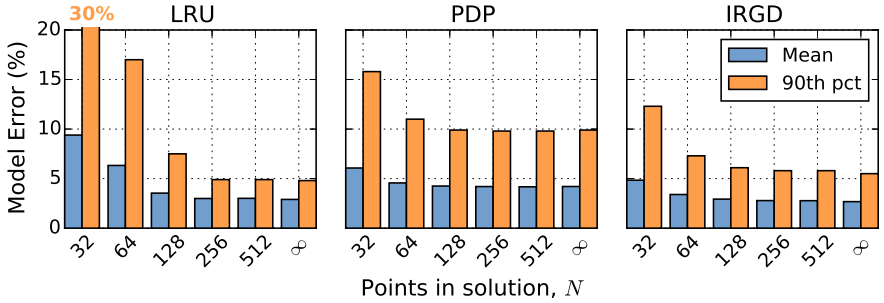


Figure C.6: Sensitivity of model to step size for LRU, PDP, and IRGD, measured by the number of steps, N (Appendix C.4.2). $N = \infty$ means full, age-by-age solution.

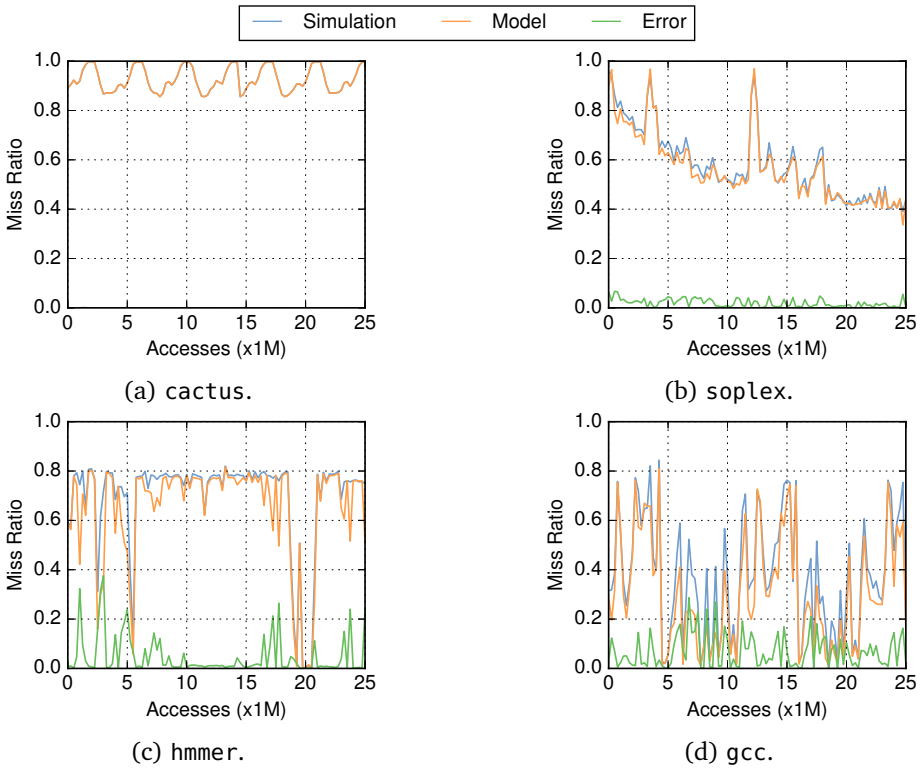


Figure C.7: Trace of simulated and predicted miss ratio for the first 25 M LLC accesses on a 1 MB, LRU LLC. Some benchmarks are unstable over time, introducing model error on some intervals.

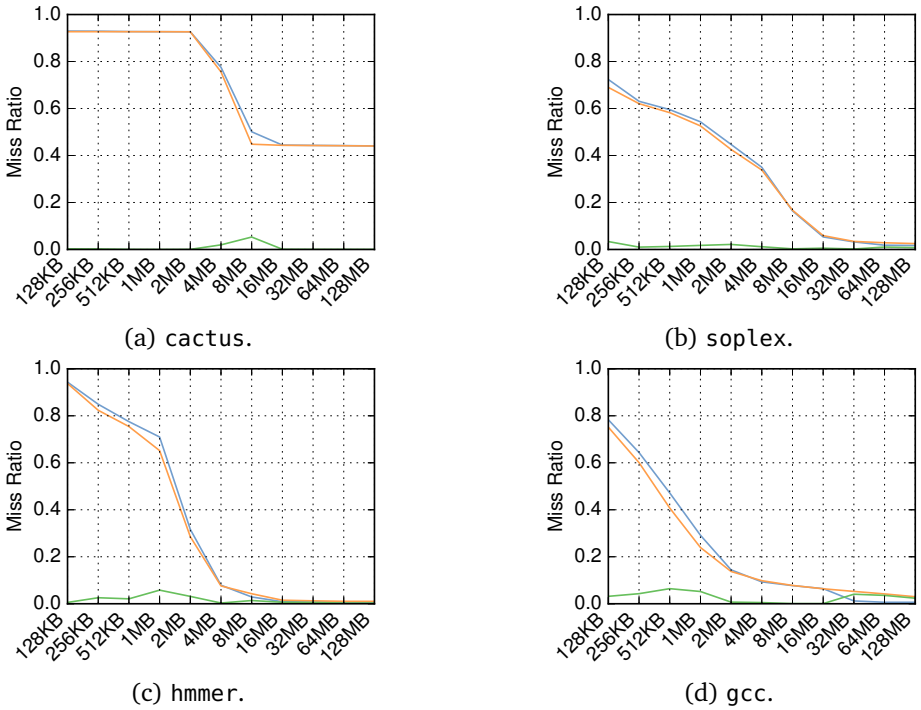
cases.) Our model is fairly insensitive to coarsening, although reducing N below 128 noticeably degrades accuracy on LRU. Skew-associative LLCs improve model accuracy even further.

It is important to emphasize that these results are for 250 K-access *intervals*—it is *not* the case that 10% of *benchmarks* have error above 7.5%/9.9%/6.1%. Rather, 10% of intervals have this error. The distinction is critical. Many benchmarks have an unstable access pattern, and their hit rate changes rapidly between intervals. Figure C.7 shows four representative SPEC CPU2006 apps on a 1 MB, LRU LLC, plotting miss ratio over the first 25 M LLC accesses with $N = 128$-point solutions.[2] Most apps are unstable; `gcc`'s and `hmmer`'s hit rates fluctuate wildly across intervals. Our model does a good job of tracking these fluctuations. (Recall that the model only uses the application's reuse distance distribution; it does not observe the cache's hit rate.) However, since benchmarks are unstable, reuse distance distributions are not representative of equilibrium behavior, and model error is large in some intervals.

This model error tends to average out in the long-run, and indeed our model is quite accurate at predicting each benchmark's miss curve over its entire execution (Figure C.8). For 128-point solutions, the mean model error over 10 B instructions is 1.9%/2.7%/1.1% for LRU/PDP/IRGD, respectively, while the 90th percentile error is 4.7%/6.7%/3.1%. Results for full solutions are similarly reduced, with long-run mean error of 1.2%/2.7%/1.0% and 90th percentile error of 3.3%/6.7%/2.9%.

Hence, the model error presented in Figure C.5 is a conservative assessment of our model's accuracy. We find that in practice the model is accurate and useful at predicting cache behavior, as we now illustrate in two case studies.

[2]Model behavior on PDP and IRGD is essentially identical.



(a) `cactus`. (b) `soplex`.

(c) `hmmer`. (d) `gcc`.

Figure C.8: Simulated and predicted miss ratio over 10 B instructions vs. LRU LLC size. Model is accurate over benchmarks' entire execution.

C.7 Case study: Cache partitioning

In this section and the next, we show how to apply our model to improve cache performance with high-performance replacement policies. This section focuses on shared caches, while the next focuses on single-threaded performance.

Cache partitioning is only effective when miss curves are available—otherwise, software cannot predict the effect of different partition sizes, and therefore cannot size them to achieve system objectives. Our model lets partitioning be used with high-performance replacement policies, which are otherwise difficult to predict (Chapter 4). It can model high-performance replacement policies at arbitrary cache sizes, and thus predicts the replacement policy's miss curve. These miss curves are given to the partitioning algorithm, which can then choose partition sizes that maximize performance (or achieve other objectives).

We evaluate the performance of a shared 4 MB LLC on a 4-core system running 100 random mixes of SPEC CPU2006 applications. We compare four schemes: *(i)* unpartitioned LRU, *(ii)* a representative thread-aware high-performance policy (TA-DRRIP [69]), *(iii)* LRU with utility-based cache partitioning (UCP) [124], and *(iv)* a high-performance policy (IRGD) with utility-based cache partitioning (UCP+IRGD). Partition sizes are set every 50 ms, and we employ a fixed-work methodology [60].

Results: Figure C.9 shows the distribution of weighted and harmonic speedup over the 100 mixes [47], normalized to unpartitioned LRU. TA-DRRIP and UCP outperform the LRU baseline respectively by 3.1%/5.1% on gmean weighted speedup, and 1.1%/3.6% on gmean harmonic speedup. Meanwhile, UCP+IRGD improves gmean weighted speedup by 9.5% and gmean harmonic speedup by 5.3%. Our model combines the single-stream benefits of high-performance cache replacement and the shared-cache benefits of partitioning, outperforming the state-of-the-art.

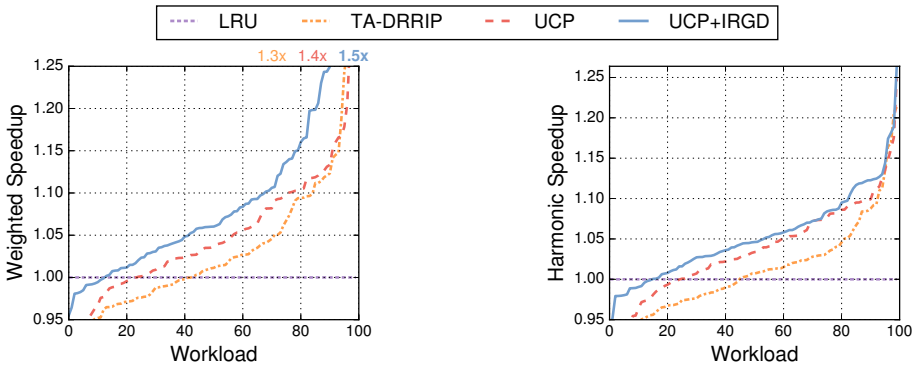


Figure C.9: Speedup for 100 random mixes of SPEC CPU2006 apps. Our model enables partitioning of IRGD, improving performance over the state-of-the-art.

Modeling classification C.8

Recall that we can more accurately describe the access pattern by dividing references into a small number of classes, within which reuse distances are iid (Appendix 5.1). Just as classification added little complexity to EVA, our cache model also easily supports classification. For the most part, each class c can be modeled independently using a per-class reuse distance distribution $\mathrm{P}_{D,c}(d)$ and rank function $R_c(a)$ representing how the policy treats class c. Because classes share the cache, we need two additional refinements. First, the age distribution must change to reflect the relative frequency of accesses to each class. Recall from Appendix C.2.2 that $\mathrm{P}_A(1) = 1/S$ because each access creates a line of age 1. With classification, $\mathrm{P}_{A,c}(1)$ is proportional to the total probability of accesses to class c (Algorithm 4, line 8):

$$\mathrm{P}_{A,c}(1) = \frac{\sum_{d=1}^{\infty} \mathrm{P}_{D,c}(d)}{S} \tag{C.18}$$

Thus, summing across classes, the probability at age 1 is unchanged at $1/S$.

Second, since any access can force an eviction and victims are chosen from all classes, the eviction distribution uses the cache's total hit rate and combined rank distribution. These are obtained by summing over all classes:

$$\mathrm{P}[\text{hit}] = \sum_c \sum_{a=1}^{\infty} \mathrm{P}_{H,c}(a) \tag{C.19}$$

$$\mathrm{P}_{\text{rank}}(r) = \sum_c \sum_{a:R_c(a)=r} \mathrm{P}_{A,c}(a) \tag{C.20}$$

The complete model equations for access class c are given in Appendix C.12. Note that although we focus on reused/non-reused classification, the above is a general framework that admits other classifications.

Predictive reused/non-reused classification C.8.1

Performing reused/non-reused classification is challenging because being reused is a dynamic property of the cache, not an independent property of the access stream. This means we must somehow derive the reuse distance distribution of reused lines $\mathrm{P}_{D,\text{reused}}(d)$ to perform classification, e.g. by using Equation C.9 or computing the protecting distance a là PDP. Modeling this distribution is possible because reused lines

are just those that hit, and the model gives the distribution of hits in the cache (Equation C.4). So with some information about how lines behave after hitting, we can compute the reuse distance distribution of reused lines. We first model this behavior exactly, and then show a practical implementation.

Exact: To model the behavior of lines after they have hit, we need to introduce a new distribution D', which gives a line's *next* reuse distance after the current one. D' is distributed identically to D unless some other condition is specified. Specifically, D' is dependent on the prior reuse distance D, and D is a property of the cache *independent of the replacement policy*. Moreover, $H = d$ implies a prior reuse distance of d, so knowing the distribution of D' conditioned on D lets us reason about how lines will behave after they are reused.

In particular, the *reuse distance transition probabilities* $\mathrm{P}\big[D' = d | D = x\big]$ let us compute the reuse distance distribution of reused lines. These are the probabilities that lines with *previous* reuse distance $D{=}x$ will have a *next* reuse distance of $D'{=}d$. Since reused lines are those that have hit in the cache, the reuse distance distribution of reused lines is:

$$\mathrm{P}_{D,\text{reused}}(d) = \sum_{x=1}^{\infty} \mathrm{P}_H(x) \cdot \mathrm{P}_{D'}\big(d | D{=}x\big) \tag{C.21}$$

The distribution of non-reused lines is just the difference of this and the full reuse distance distribution.

The all-to-all transition probabilities are difficult to gather in practice, however, since they require a two-dimensional histogram indexed by previous reuse distance × next reuse distance. The resulting distributions are expensive to sample and can have significant noise.

Simplified: We account for this by instead using a simpler distribution: the reuse distance distribution of lines that were previously reused at a distance d less than some limit ℓ. We choose the limit to serve as an effective proxy for being reused. Specifically, ℓ covers 90% of hits as computed by the model, $\mathrm{P}[H < \ell] \approx 0.9$. We chose this parameter empirically; performance is relatively insensitive to its value.

We then compute the reuse distance distribution of reused lines, $\mathrm{P}_{D,\text{reused}}(d)$, from whether the reuse came before or after the limit:

$$\mathrm{P}_{D,\text{reused}}(d) \approx \mathrm{P}\big[D' = d, H < \ell\big] + \mathrm{P}\big[D' = d, H \geq \ell\big] \tag{C.22}$$

We compute each term assuming the hit probability of the *current* access is independent of the reuse distance of the *next* access.[3] For example, the former term:

$$\begin{aligned}
\mathrm{P}\big[D' = d, H < \ell\big] &= \mathrm{P}\big[\text{hit} | D' = d, D < \ell\big] \cdot \mathrm{P}\big[D' = d, D < \ell\big] \\
\text{(Independence)} \qquad &= \mathrm{P}\big[\text{hit} | D < \ell\big] \cdot \mathrm{P}\big[D' = d, D < \ell\big] \\
\text{(Simplify)} \qquad &= \frac{\mathrm{P}[H < \ell] \cdot \mathrm{P}\big[D' = d, D < \ell\big]}{\mathrm{P}[D < \ell]} \\
&= \mathrm{P}[H < \ell] \cdot \mathrm{P}\big[D' = d | D < \ell\big]
\end{aligned} \tag{C.23}$$

This allows us to gather only two distributions, $\mathrm{P}_D(d)$ and $\mathrm{P}\big[D' = d, D < \ell\big]$, in order to model reused/non-reused classification.[4] For this model to be informative, ℓ must be chosen intelligently. We choose ℓ to cover most hits (90%).

[3] The same assumption is implicit in the exact derivation above, and follows nearly an identical form to the derivation that follows.

[4] The third distribution, $\mathrm{P}\big[D' = d, D \geq \ell\big]$, is their difference.

Although this limit scheme is a rough approximation of the full transition probabilities, the following case study shows that it captures the benefits of reused/non-reused classification while simplifying modeling and monitoring hardware.

Case study: Improving cache replacement C.9

As demonstrated by prior policies and EVA, classifying lines into classes (e.g., reused vs. non-reused) and treating each class differently can significantly improve replacement policy performance [69, 79, 159]. However, many otherwise attractive policies do not exploit classification, or do so in limited ways, e.g. using fixed heuristics that prioritize certain classes. For instance, PDP [44] suggests classification as a promising area of future work, but lacks a modeling framework to enable it; and IRGD [141] requires several additional monitors to perform classification. Our model provides a simple analytical framework for classification. Specifically, we extend IRGD (Equation C.9) to support reused vs. non-reused classification.

We support classification by modeling the reuse distance distribution of each class, in addition to the hit and eviction distributions. Doing so requires modest changes to the model. Since policies like PDP and IRGD are reuse distance-based, this is sufficient for them to support classification. Crucially, our model captures the highly dynamic interactions *between* classes, which is essential for dynamic classifications like reused vs. non-reused.

Model-based classification has a further advantage. Unlike some prior high-performance policies (e.g., DRRIP [69]), we do not assume that reused lines are preferable to non-reused lines. We instead model the behavior of each class, and rely on the underlying policy (e.g., IRGD) to make good decisions within each class. This approach allows for direct comparison of candidates *across* classes according to their rank, removing the need for heuristics or tunables to penalize or prefer different classes. We thus avoid pathological performance on applications where, contra expectations, reused lines are less valuable than non-reused lines.

Results: Figure C.10 shows the LLC misses per thousand instructions (MPKI) for three representative benchmarks and average performance over all 29 SPEC CPU2006 on caches of size 128 KB to 128 MB (note log scale on x-axis). Lower MPKIs are better. Our model improves on IRGD on most SPEC CPU2006 benchmarks, and never degrades performance. On `cactusADM`, our model enables IRGD to match DRRIP's performance, since this benchmark exhibits clear differences between reused and non-reused lines. On `lbm`, our model provides no additional benefit, but unlike DRRIP, we do not degrade performance compared to LRU. Finally, on `mcf`, our model improves performance, but does not match the performance of DRRIP.

On net, our model makes a slight but significant improvement to IRGD's overall performance. The rightmost figure plots the MPKI over optimal cache replacement (Belady's MIN [17]) averaged over all of SPEC CPU2006. Without classification, DRRIP outperforms IRGD on many cache sizes. With classification, IRGD outperforms DRRIP at every cache size. On average across sizes, IRGD (without classification) and DRRIP perform similarly, closing 44% and 42% of the MPKI gap between LRU and MIN, respectively. With classification, IRGD closes 52% of this gap.

Summary C.10

We have presented a cache model for modern LLCs with high-performance replacement policies. Our model is motivated by observations of modern cache architecture (e.g., of temporal reuse) that allow

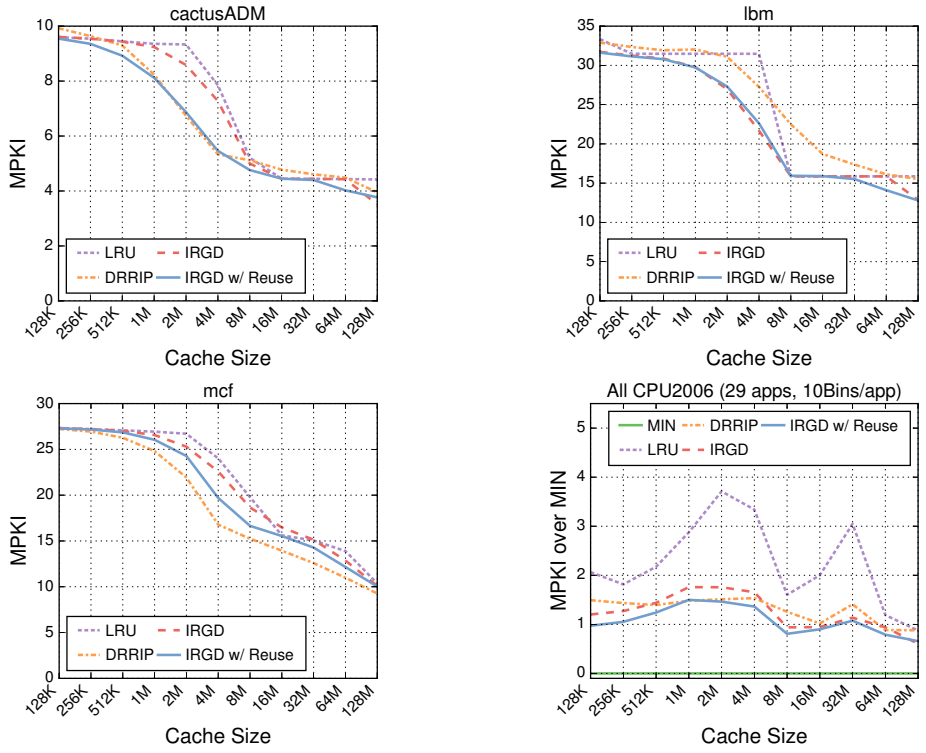


Figure C.10: Misses per thousand instructions (MPKI) vs. cache size for (left) three applications and (right) all of SPEC CPU2006. Lower is better. Our model extends IRGD [141] to support reused/non-reused classification, making it outperform DRRIP on most applications and on gmean MPKI over all of SPEC CPU2006.

us to abstract away details of array organization and focus on modeling the replacement policy. As a result, we capture a broad class of policies at relatively low complexity. We have presented an efficient implementation of the model and thoroughly evaluated its accuracy and implementation tradeoffs. Finally, we showed how to use the model to improve cache performance over state-of-the-art techniques.

Cache Model Equations with Coarsening C.11

The coarsened model equations are obtained by assuming constant event probabilities within a region. Ages are broken into regions at the ages $1 = a_1 < a_2 < \ldots < a_{N+1}$, giving N coarsened age regions. We start from the model equations (Equation C.1, Equation C.4, and Equation C.15) and sum over each region to derive the coarsened equations. For example, the coarsened hit equation for region i is derived as:

$$\mathrm{P}\Big[a_i \le H < a_{i+1}\Big] = \sum_{a=a_i}^{a_{i+1}-1} \mathrm{P}[H = a] \tag{C.24}$$

$$= \sum_{a=a_i}^{a_{i+1}-1} \left(\mathrm{P}[D = a] \times \left(1 - \sum_{x=1}^{a-1} \frac{\mathrm{P}[E = x]}{\mathrm{P}[D > x]} \right) \right)$$

We now model the eviction probability (i.e., the second factor in the above equation) as approximately constant in this region, and pull it out of the sum.

$$\approx \left(\sum_{a=a_i}^{a_{i+1}-1} \mathrm{P}[D = a] \right) \times \left(1 - \sum_{x=1}^{a_i} \frac{\mathrm{P}[E = x]}{\mathrm{P}[D > x]} \right) \tag{C.25}$$

Applying this approximation to the second factor, we break the sum over x across prior regions, and we simplify the first factor.

$$= \mathrm{P}\Big[a_i \le D < a_{i=1}\Big] \times \left(1 - \sum_{j=1}^{i-1} \frac{\mathrm{P}\Big[a_j \le E < a_{j+1}\Big]}{\mathrm{P}\Big[D > a_j\Big]} \right) \tag{C.26}$$

The equation for the coarsened eviction distribution is similar, although it is complicated by ranking functions. A ranking function may map distinct age regions to overlapping rank regions, so there is no strict rank-ordering of ages between two regions. The correct solution to this problem is to compute the rank distribution at full fidelity, i.e. age-by-age. But doing so sacrifices much of the benefit of coarsening, since it means each iteration of the solution must consider every age. We instead coarsen ranks by representing every age within a region by the mean rank over that region, denoted $\bar{R}_i$ for the i^{th} region. If regions are chosen poorly, this approach can be inaccurate, but when regions are chosen intelligently

it is a good approach. The eviction distribution is thus computed:

$$\begin{aligned}\mathrm{P}\big[a_i \le E < a_{i+1}\big] &= \sum_{a=a_i}^{a_{i+1}-1} \mathrm{P}[E=a] \\ &= \mathrm{P}[\text{miss}] \times \sum_{a=a_i}^{a_{i+1}-1} \frac{\mathrm{P}[A=a]}{\mathrm{P}\big[\text{rank} = R(a)\big]} \times \mathrm{P}_{\max\text{rank}}\big(R(a)\big) \\ &\approx \mathrm{P}[\text{miss}] \times \sum_{a=a_i}^{a_{i+1}-1} \frac{\mathrm{P}[A=a]}{\mathrm{P}\big[\text{rank} = \bar{R}_i\big]} \times \mathrm{P}_{\max\text{rank}}\big(\bar{R}_i\big)\end{aligned} \tag{C.27}$$

We model the age distribution as approximately constant in the region, so using Equation C.12 we obtain:

$$\approx \mathrm{P}[\text{miss}] \times \frac{\mathrm{P}\big[A=a_i\big]}{\mathrm{P}\big[\text{rank} = \bar{R}_i\big]} \times \mathrm{P}_{\max\text{rank}}\big(\bar{R}_i\big) \tag{C.28}$$

The age distribution is simple to derive assuming the hit and eviction distributions are constant in the region. The simplest way to derive our coarsened approximation of the age distribution is as an extension of the recurrence relation Equation C.16. In a single step, the age distribution $\mathrm{P}_A(a)$ decreases by $\mathrm{P}_H(a)+\mathrm{P}_E(a)$. Hence in the coarsened region from a_i to a_{i+1}, it decreases by the total hit and eviction probability in the region. Thus Equation C.16 becomes:

$$\mathrm{P}_A\big(a_i\big) \approx \mathrm{P}_A\big(a_{i-1}\big) - \frac{\mathrm{P}\big[a_{i-1} \le H < a_i\big] + \mathrm{P}\big[a_{i-1} \le E < a_i\big]}{S} \tag{C.29}$$

C.12 Model with Classification

The complete model equations for class c are:

$$\mathrm{P}_{A,c}(a) = \frac{\mathrm{P}[L \ge a, c]}{S} = \sum_{x=a}^{\infty} \frac{\mathrm{P}_{H,c}(x) + \mathrm{P}_{E,c}(x)}{S} \tag{C.30}$$

$$\mathrm{P}_{H,c}(d) = \mathrm{P}_{D,c}(d) \cdot \left(1 - \sum_{x=1}^{d-1} \frac{\mathrm{P}_{E,c}(x)}{\mathrm{P}[D > x, c]}\right) \tag{C.31}$$

$$\mathrm{P}_{E,c}(a) = (1 - \mathrm{P}[\text{hit}]) \cdot \frac{\mathrm{P}_{\max\text{rank}}\big(R_c(a)\big) \cdot \mathrm{P}_{A,c}(a)}{\mathrm{P}_{\text{rank}}\big(R_c(a)\big)} \tag{C.32}$$

along with Equation C.19 and Equation C.20. These are simply the original model equations (Equation C.1, Equation C.4, and Equation C.15) with joint probability on class c where appropriate.

Cache Calculus: Modeling Caches as a System of Ordinary Differential Equations D

THE probabilistic cache model presented in the last chapter is accurate on real applications, but its implicit formulation hides how performance changes with different parameters. For example, it is hard to see from the equations themselves how much the hit rate improves with increasing cache size. Explicit, closed-form equations for cache performance are more expressive and easier to reason about. This chapter relaxes the probabilistic cache model into a system of ordinary differential equations (ODEs). We then show how to solve this system for particular access patterns. In doing so, the ODE formulation lets architects leverage efficient, robust numerical analysis techniques [25] to model cache performance.

A continuous approximation of the discrete model D.1

Table D.1 shows how the discrete model from the previous chapter can be relaxed into a continuous model. Discrete probabilities $\mathrm{P}_X(x)$ are replaced by continuous ones $f_X(x)$ and sums are replaced by integrals. At first blush, it may not seem to have simplified matters much. But the integral bounds are simpler than the corresponding sums, and we will use the Fundamental Theorem of Calculus [112] to simplify the model further.

Equation	Discrete	Continuous
Age distribution	$\mathrm{P}_A(a) = \frac{1}{S}\sum_{x=a}^{\infty} \mathrm{P}_H(x) + \mathrm{P}_E(x)$	$f_A(a) = \frac{1}{S}\int_a^{\infty} f_H(x) + f_E(x)\,\mathrm{d}x$
Hit distribution	$\mathrm{P}_H(a) = \mathrm{P}_D(a) \times \left(1 - \sum_{x=1}^{a-1} \frac{\mathrm{P}_E(x)}{\mathrm{P}[D > x]}\right)$	$f_H(a) = f_D(a) \times \int_a^{\infty} \frac{f_E(x)}{\int_x^{\infty} f_D(z)\,\mathrm{d}z}\,\mathrm{d}x$
Eviction distribution	$\mathrm{P}_E(a) = \mathrm{P}[\text{miss}] \times \mathrm{P}_{\max\text{rank}}\big(R(a)\big) \times \frac{\mathrm{P}_A(a)}{\mathrm{P}_{\text{rank}}\big(R(a)\big)}$	$f_E(a) = \mathrm{P}[\text{miss}] \times f_{\max\text{rank}}(R(a)) \times \frac{f_A(a)}{f_{\text{rank}}(R(a))}$

Table D.1: Continuous approximation of discrete model equations.

Interpretation: The discrete model converges to the continuous model as ages become infinitesimally small relative to reuse distances, the cache size, etc.. A natural question is what this limit represents "in the real world". One interpretation is that the continuous equations model cache performance as both the cache and the application's footprint scale indefinitely large, but in the same proportion.

That is, suppose both the cache and the working set increase in size by a factor of two. The cache's hit rate will be unchanged by this transformation, regardless of its replacement policy (Theorem 4,

pg. 62). Likewise, all model distributions are scaled by a factor of two; e.g., if H' is the hit distribution after scaling, then $\mathrm{P}_{H'}(2a) = \mathrm{P}_H(a)$. Hence the "age step" of the discrete model is effectively halved, without any significant impact on the solution. The continuous model is the limit as this process repeats indefinitely.

Example replacement policies: For ease of discussion, we follow through the derivations using random and LRU replacement. In random replacement every age has the same rank, so the eviction distribution simplifies to:

$$f_E(a) = m\, f_A(a), \tag{D.1}$$

where $m = \mathrm{P}[\text{miss}] = \int_0^\infty f_E(x)\,\mathrm{d}x$.

In LRU, rank equals age ($R(a) = a$). Recall that the distribution of maximum rank $f_{\max\text{rank}}(r)$ is the derivative of the cumulative distribution of maximum rank (Appendix C.3, pg. 125). So the eviction distribution simplifies to:

$$f_E(a) = m\,\frac{\mathrm{d}}{\mathrm{d}a}\left(\int_0^a f_A(x)\,\mathrm{d}x\right)^W \tag{D.2}$$

D.2 A system of ordinary differential equations

Now define $\alpha(a) = \int_0^a f_A(x)\,\mathrm{d}x$. That is, α is the cumulative distribution function of age at a. Then by the Fundamental Theorem of Calculus, $\alpha' = f_A(a)$. Table D.2 shows our complete notation for each random variable.

Value	Notation: Discrete	Notation: ODE	
Cache size	S		Unchanged
Associativity	W		Unchanged
Reuse distance distribution	$\mathrm{P}[D < a]$	δ	CDFs ⇒ Variable in ODE
Age distribution	$\mathrm{P}[A < a]$	α	CDFs ⇒ Variable in ODE
Hit distribution	$\mathrm{P}[H < a]$	η	CDFs ⇒ Variable in ODE
Eviction distribution	$\mathrm{P}[E < a]$	ϵ	CDFs ⇒ Variable in ODE
Rank distribution	$\mathrm{P}[\text{rank} < r]$	ρ	CDFs ⇒ Variable in ODE
Miss rate	$\mathrm{P}[\text{miss}]$	m	

Table D.2: Model nomenclature with ordinary differential equations.

D.2.1 Ages

Using this notation to simplify the math, the age distribution equation becomes:

$$\begin{aligned} f_A(a) &= \frac{1}{S}\left(1 - \int_0^a f_H(x) + f_E(x)\,\mathrm{d}x\right) \\ \Rightarrow \alpha' &= \frac{1-(\eta+\epsilon)}{S} \end{aligned} \tag{D.3}$$

Which we differentiate to obtain:

$$\alpha'' = -\frac{\eta' + \epsilon'}{S} \tag{D.4}$$

Hits D.2.2

The hit distribution is more complicated:

$$f_H(a) = f_D(a) \int_a^\infty \frac{f_E(x)}{1 - \int_0^a f_D(z)\,\mathrm{d}z}\,\mathrm{d}x$$
$$\Rightarrow \eta' = \delta' \int_a^\infty \frac{\epsilon'(x)}{1-\delta(x)}\,\mathrm{d}x$$

We add another variable $\beta = \int_0^a \epsilon'(x)/(1-\delta(x))\,\mathrm{d}x$ to eliminate the integral:

$$\eta' = \delta'\,(1-\beta) \tag{D.5}$$
$$\beta' = \frac{\epsilon'}{1-\delta} \tag{D.6}$$

For analytical solutions, we can eliminate β by using second derivatives, along with the Product Rule and Fundamental Theorem of Calculus:

$$\eta'' = \delta''\,(1-\beta) - \delta'\,\beta'$$

And hence, so long as $\delta' > 0$ and $\delta < 1$:

$$\eta'' = \frac{\delta''}{\delta'}\,\eta' - \frac{\delta'}{1-\delta}\,\epsilon' \tag{D.7}$$

Evictions D.2.3

The eviction distribution is, in general, the most complicated:

$$f_E(a) = m \times f_{\text{max rank}}(R(a)) \times \frac{f_A(a)}{f_{\text{rank}}(R(a))}$$
$$= m \times \tfrac{\mathrm{d}}{\mathrm{d}a}\left(\int_0^{R(a)} f_{\text{rank}}(r)\,\mathrm{d}r\right)^W \times \frac{f_A(a)}{f_{\text{rank}}(R(a))}$$

Substituting in $\rho(r) = \int_0^r f_{\text{rank}}(x)\,\mathrm{d}x$ and applying the chain rule:

$$\epsilon' = m \times \tfrac{\mathrm{d}}{\mathrm{d}a}\rho(R(a))^W \times \frac{\alpha'}{\rho'(R(a))}$$
$$= mW\,\rho(R(a))^{W-1}\,R'(a)\,\alpha' \tag{D.8}$$

However, this general form that supports arbitrary rank functions is difficult to work with. To simplify discussion, we consider random and LRU replacement.

Examples: Under random replacement:

$$\begin{aligned} f_E(a) &= m\, f_A(a) \\ \Rightarrow \epsilon' &= m\, \alpha' \end{aligned} \tag{D.9}$$

Differentiating and substituting in Equation D.4:

$$\epsilon'' = -\frac{m}{S}\left(\eta' + \epsilon'\right) \tag{D.10}$$

LRU is more complicated, since the eviction distribution changes with the CDF of ages:

$$\epsilon' = m\, \tfrac{\mathrm{d}}{\mathrm{d}a}\alpha^W = mW\, \alpha^{W-1}\, \alpha' \tag{D.11}$$

Closed-form solutions of LRU are difficult to come by since the eviction distribution is non-linear.

D.3 Modeling cache performance

The above equations form a non-autonomous system of differential equations in five variables (the model distributions α, η, and ϵ plus helper variables β and ρ). This system yields the cache's miss rate as: $m = \lim_{a\to\infty} \epsilon = 1 - \lim_{a\to\infty} \eta$. Note also that $\lim_{a\to\infty} \alpha = 1$.

For random replacement, Equations D.7 and D.10 form a non-autonomous *linear* system of differential equations in just *two* variables (η' and ϵ'). Since linear systems are much simpler to work with analytically, it is easiest to find closed-form solutions for random replacement.

Modeling a particular access pattern therefore corresponds to solving the initial value problem where all variables are initially zero. To solve the system, particularly the eviction distribution, we need to know the miss rate $m = \lim_{a\to\infty} \epsilon$, which is itself a product of a solution. Hence a complete solution of the system is one that maps the *seed miss rate* to the same *modeled miss rate*; i.e., it is a fixed point on m.

In the following examples, we numerically solve the ODEs from the reuse distance distribution of the simulated access stream. Specifically, we use the Runge-Kutta method of order 4 with a step size of $\Delta a = 1$ [25].

We can visualize how the cumulative hit and eviction probabilities (η and ϵ) behave over time by plotting η and ϵ as the x- and y-axes on a graph, called the *probability space*. In this case, modeling the cache corresponds to finding the final *resting position* of (η, ϵ) along the line $\eta + \epsilon = 1$. The analogous *velocity* (η', ϵ') at age a corresponds to the instantaneous hit and eviction probabilities. These graphs are the standard way to visualize vector fields, but they can be unintuitive. We therefore also directly plot the probabilities vs. age, comparing simulation and model results on each graph.

D.3.1 Example: Randomized Scan

Figure D.1 shows an example access pattern that scans over a 350-element array while randomly skipping many elements each iteration, leading to the depicted reuse distance distribution. Reuse distances repeat periodically with exponentially decreasing probability, since larger reuse distances are only reached when an element happens to be skipped in successive scans over the array.

Random replacement: Figure D.3 shows the cache behavior for this pattern on a 125-line cache using random replacement. Figure D.3a shows simulation results over many accesses, and Figure D.3b the model solution. In both figures, the left-hand figure ("position") plots the cumulative probabilities of

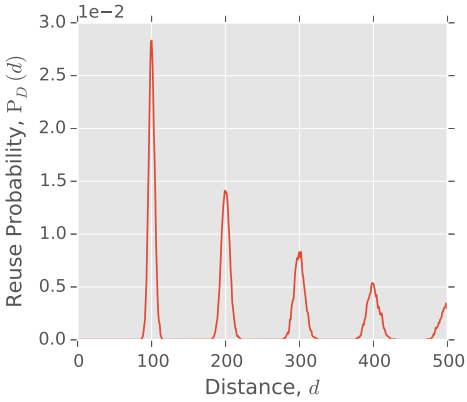


Figure D.1: Reuse distance distribution for a randomized, scanning access pattern.

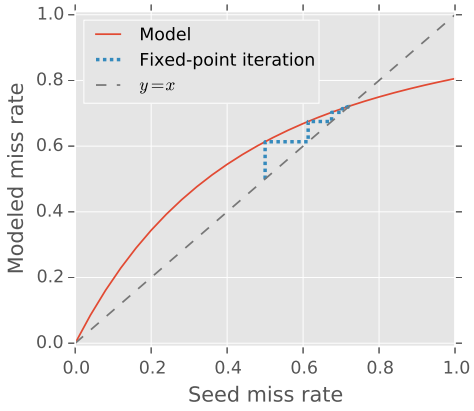
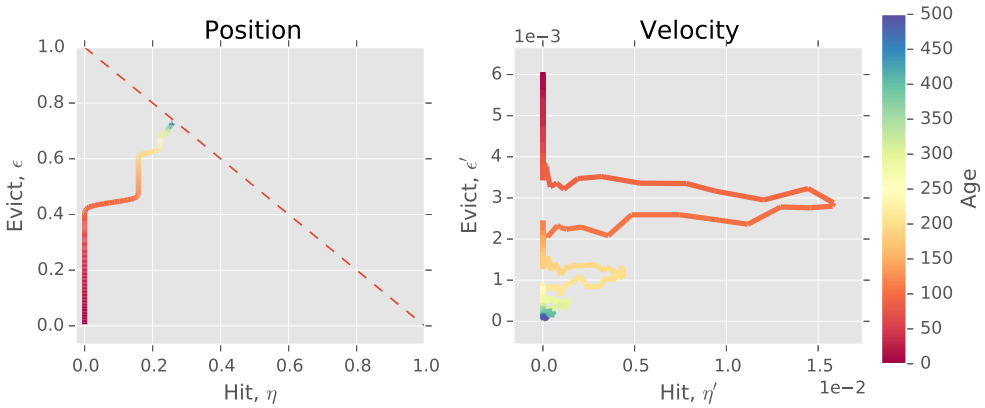
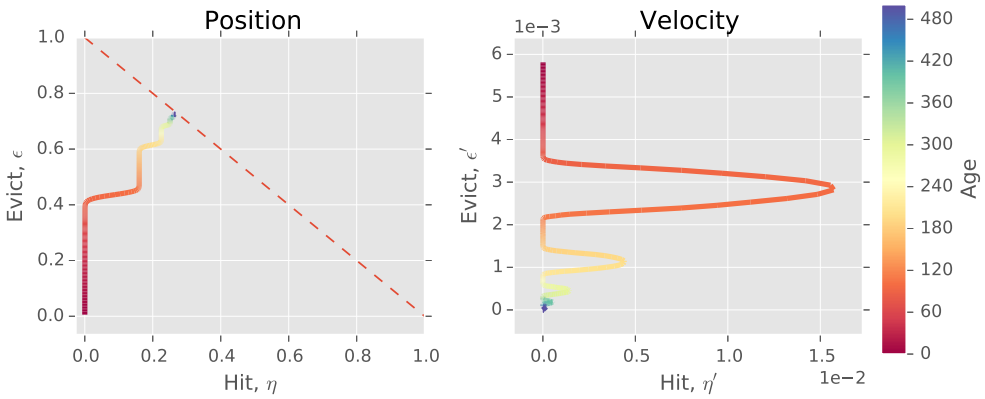


Figure D.2: Fixed-point iteration quickly converges to the simulated miss rate.



(a) Simulation results.



(b) Model solution.

Figure D.3: Positional visualization of a randomized scan on a 125-line cache with random replacement.

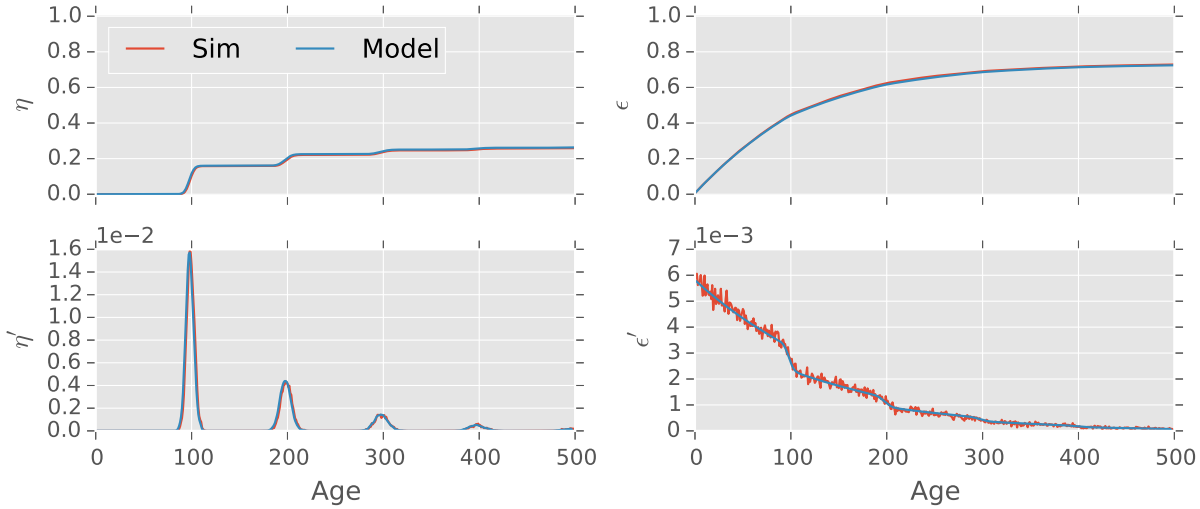


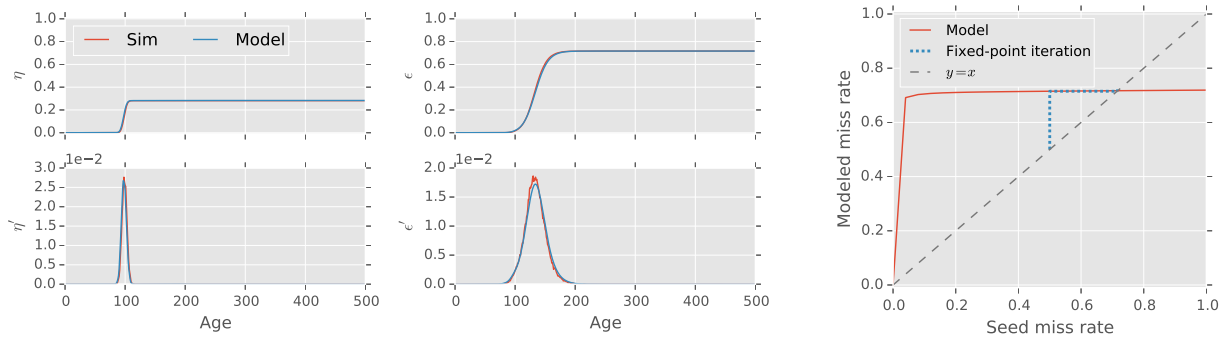
Figure D.4: Temporal visualization of a randomized scan on a 125-line cache with random replacement.

having hit or evicted over ages from 0 to 500. Specifically, it plots hit × eviction probability ($\eta \times \epsilon$). The probabilities converge to a point on the line $\eta + \epsilon = 1$ (dashed), in this example roughly at a cumulative probability for evictions of $m = 0.73$. The right-hand figure ("velocity") plots the instantaneous hit or eviction probabilities over ages from 0 to 500. As can be seen, the eviction probability starts relatively high, while the hit probability is initially zero and peaks following the reuse distance distribution. Both the hit and eviction probabilities rapidly decrease to zero as lines age, as an exponentially smaller share of lines survive to large ages.

Figure D.3b shows a numerical solution of the ODEs. The model solution closely matches simulation results, and in fact "smooths" some of the noise from simulation (evident in the velocity plots on the right). Figure D.4 shows the same information in Figure D.3, but plots the cumulative and instantaneous probability vs. age with two plots per graph comparing simulation and the ODE solution. For the remaining examples, we will use the temporal visualization (e.g., Figure D.4) because we find them easier to read.

Finally, Figure D.2 shows how ODE solutions converge to the simulated miss rate. The figure plots the seed miss rate used to solve the ODE (x-axis) vs. the modeled miss rate produced from this seed (y-axis). This mapping intersects the line $y = x$ (i.e., fixed points) at two values of m: the cache's miss rate at $m \approx 0.73$ and a degenerate solution at $m = 0$. The non-zero solution is the fixed point where the model has converged. The figure further shows how seeding the model with a miss rate of $m = 0.5$ rapidly converges to the simulated miss rate.

LRU replacement: Figure D.5a shows results on the same access pattern for a 125-line cache using LRU replacement among 16 randomly selected candidates. (A random-candidates cache closely models zcaches [129], skew-associative caches [133], or set-associative caches with many ways.) The cache's miss rate is similar, $m \approx 0.74$, but a closer look reveals that LRU behaves quite differently than random replacement. Figure D.5a shows that hits and evictions occur in a single peak under LRU, unlike in random replacement where they are spread across multiple peaks. Since LRU evicts the oldest line among candidates, lines that reach age ∼125 are quickly evicted and thus essentially none make it past age 175. Meanwhile, lines frequently make it to age 200 and beyond in random replacement (compare Figures D.4 and D.5a).



(a) Event probabilities.

(b) Fixed-point iteration.

Figure D.5: Randomized scanning pattern on a 125-line cache with LRU replacement.

Their different behavior is also reflected in the fixed-point mapping, as shown in Figure D.5b. Random replacement tends to produce smooth curves like Figure D.2, but in LRU the fixed-point mapping is abrupt. This means that LRU converges to the fixed-point almost immediately.

Example: Scans D.3.2

Next, we consider a standard scanning pattern that iterates across an array. Figure D.6 shows the reuse distance distribution: it consists of a single peak at the size of the array (350, in this case).

Figure D.7 shows the simulation and model results for a scan on random replacement. On the left are the event probabilities, and on the right fixed-point iteration. The model is once again accurate. Most lines are evicted before the scan completes, yielding a miss rate of $m = 0.93$, but a few survive long enough to hit, as shown in the small bump in η at age 350 in Figure D.7a.

Figure D.8 shows that the model is also accurate on LRU, where the cache evicts all lines around age 125 and yields no hits.

Example: Random Accesses D.3.3

Next, we consider a random access pattern. Figure D.9 shows the reuse distance distribution: an exponential probability distribution, since the access pattern is memoryless.

Figures D.10 and D.11 show that the model is accurate once again, and fixed point iteration is efficient and behaves similar to other patterns for each policy. Both policies perform the same, since random accesses are agnostic to replacement policy. But they achieve this performance differently. In random replacement, all event probabilities follow an exponential distribution, whereas LRU has its characteristic peak of evictions. In either case, the model accurately captures the cache's behavior.

Example: Stack D.3.4

Finally, we consider a stack access pattern, where the program iterates back and forth across an array. Figure D.12 shows the reuse distance distribution: it is flat, since all reuse distances occur exactly once per iteration. This pattern shows maximal temporal locality, and LRU is in fact the optimal policy.

Figures D.13 and D.14 show that the model is less accurate on stack patterns. The error is modest for random replacement, but quite severe for LRU. Indeed, the model mispredicts the hit rate by nearly a factor of two.

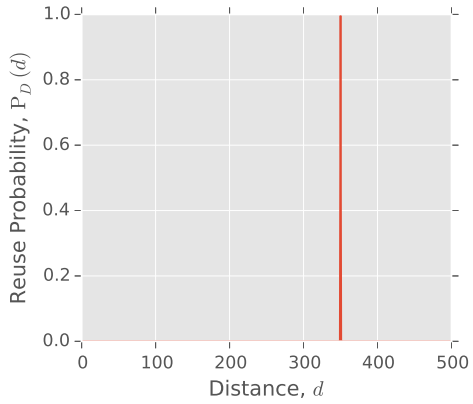
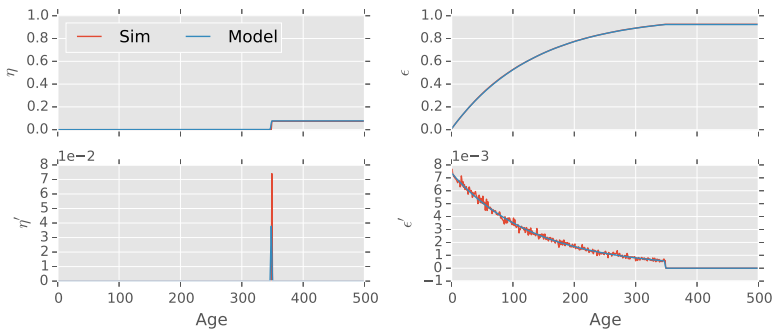
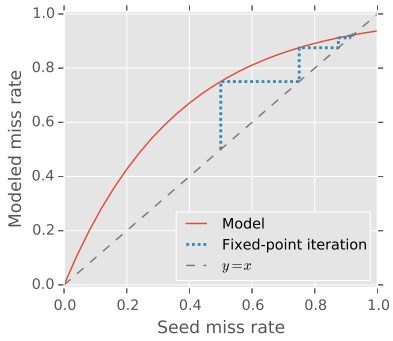


Figure D.6: Reuse distance distribution for a scanning access pattern.

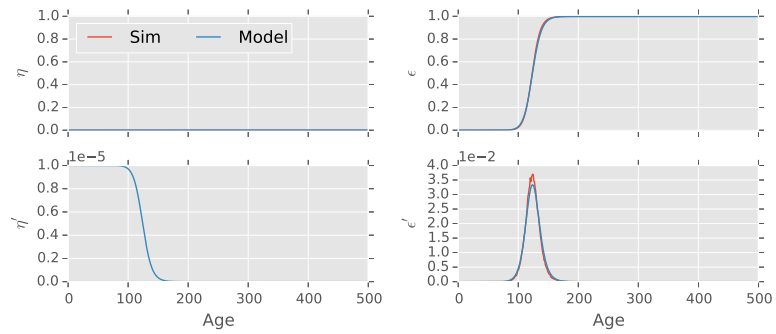


(a) Event probabilities.

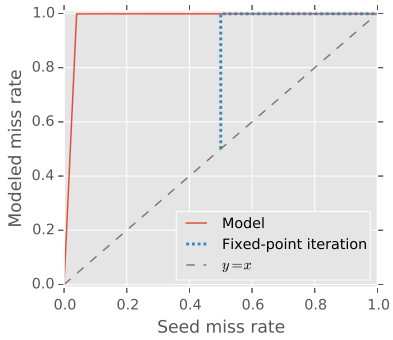


(b) Fixed-point iteration.

Figure D.7: Scanning pattern on a 125-line cache with random replacement.



(a) Event probabilities.



(b) Fixed-point iteration.

Figure D.8: Scanning pattern on a 125-line cache with LRU replacement.

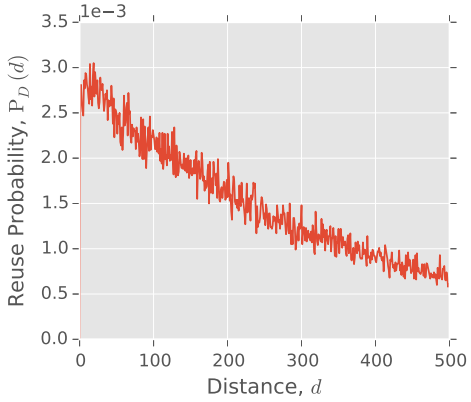
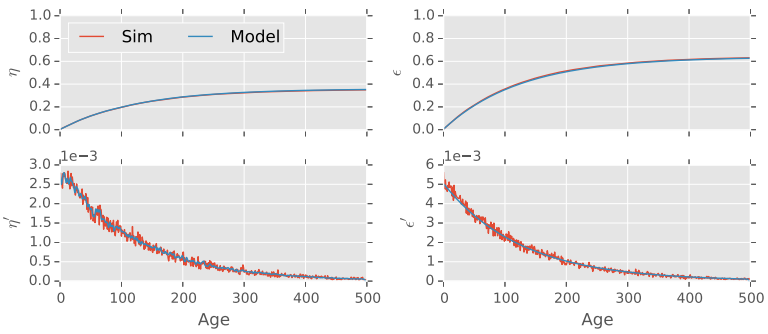
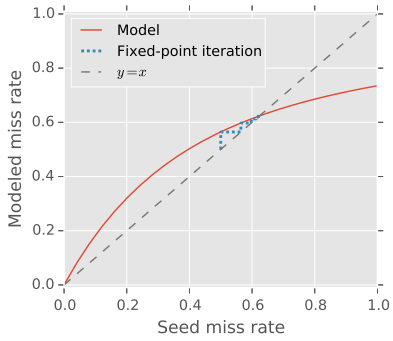


Figure D.9: Reuse distance distribution for a random access pattern.

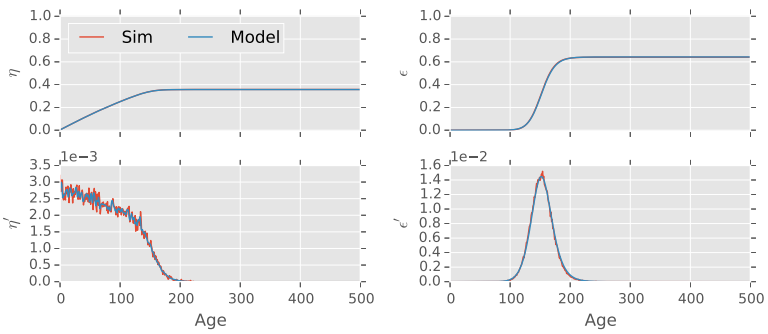


(a) Event probabilities.

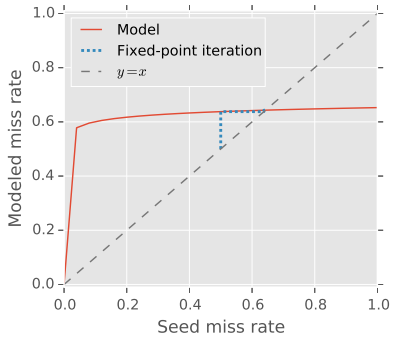


(b) Fixed-point iteration.

Figure D.10: Random access pattern on a 125-line cache with random replacement.



(a) Event probabilities.



(b) Fixed-point iteration.

Figure D.11: Random access pattern on a 125-line cache with LRU replacement.

The error comes not from the ODE formulation, but from the iid reuse distance model itself. Specifically, the reference model asserts that reuse distances are independent, but this assumption is wrong for stack access patterns.

Fortunately, these kinds of patterns are typically captured in the private levels of the cache hierarchy and not seen at the LLC, so the error is not often exposed. When temporal locality is present at the LLC, reused/non-reused classification corrects some of the error introduced by the memory reference model.

D.3.5 Discussion

These examples illustrate how to leverage numerical analysis to accurately model complex access patterns. To produce these results, we used a standard integration method with a fixed step size. Alternatively, we could use integration methods with adaptive step size to increase efficiency while maintaining accuracy. Such methods are well-explored in the numerical analysis literature [25], and directly correspond to the adaptive step size employed in the discrete model (Appendix C.4.2, pg. 128). We hope that the ODEs provide intuition for why the discrete model equations are invariant to step size: in the ODE formulation, the step size can be freely chosen to tradeoff accuracy and efficiency.

The fixed-point mappings (e.g., Figure D.2) also give a highly efficient way to model miss rates that converges in a few steps ($\sim$5), many fewer than the discrete model (which required 20-40).

Despite these attractive features, numerical solutions of the ODEs are not a replacement for the discrete model in Appendix C. In fact, the two approaches are quite similar. For example, we introduced the variable β to simplify the inner integral of the hit equation. β corresponds to the variable evBelow in Algorithm 4, which is used in the discrete model for the same purpose.

Moreover, the results presented thus far have used random or LRU replacement, where the solutions benefit from the simple nature of these policies. In particular, we only need the miss rate to solve the model for both policies, since we do not need to know what happens at later ages to compute the eviction probability for the current age. This lets the model quickly converge using fixed-point iteration on the miss rate alone. In the general case, ranks can increase and decrease over time, and later ages may therefore influence the eviction probabilities. (Indeed, this is commonly the case with EVA from Chapter 5.) We must therefore converge to the *rank distribution* as well as the miss rate. Doing so requires techniques similar to those employed in the discrete model (e.g., maxRankDist in Algorithm 4).

Hence the system of ODEs offers an efficient way to solve some policies (e.g., random and LRU) and gives intuition for the discrete model. But its contributions over the discrete model for arbitrary ranking functions are less clear. However, the ODEs offer one major *qualitiative* contribution over the discrete model: explicit, closed-form solutions.

D.4 Closed-form model solutions of random replacement

We now discuss how to arrive at closed-form solutions for cache performance on particular access patterns and replacement policies. We do so for random replacement, since its linear system of ODEs is the easiest to analyze. We solve Equations D.7 and D.10:

$$\eta'' = \frac{\delta''}{\delta'}\,\eta' - \frac{\delta'}{1-\delta}\,\epsilon'$$
$$\epsilon'' = -\frac{m}{S}\left(\eta' + \epsilon'\right)$$

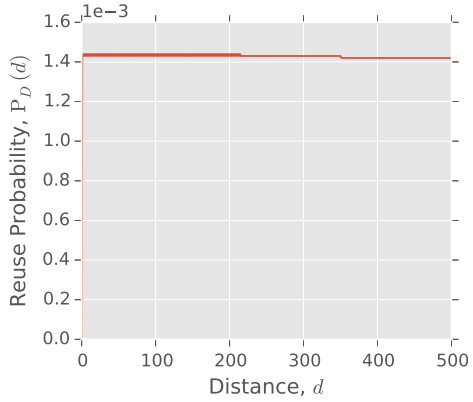
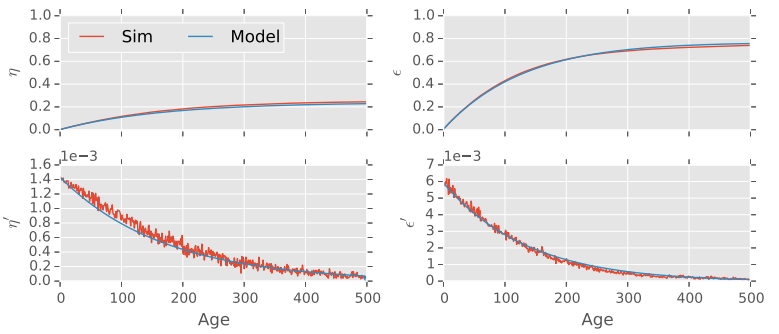
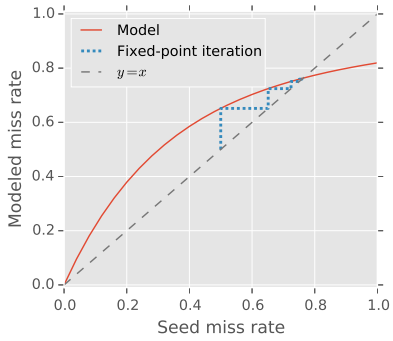


Figure D.12: Reuse distance distribution for a stack access pattern.

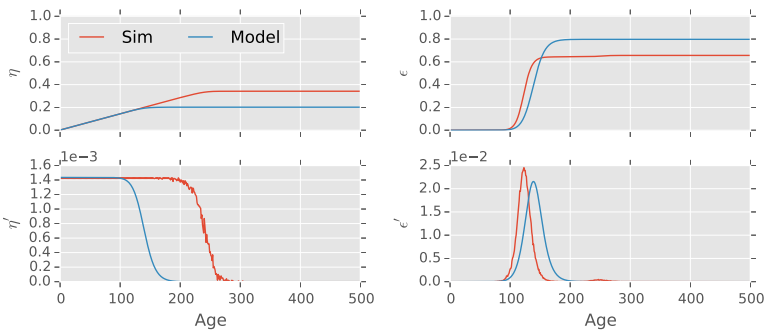


(a) Event probabilities.

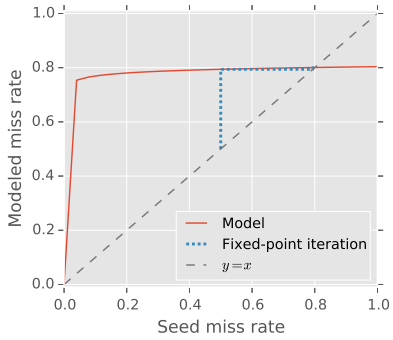


(b) Fixed-point iteration.

Figure D.13: Stack pattern on a 125-line cache with random replacement.



(a) Event probabilities.



(b) Fixed-point iteration.

Figure D.14: Stack pattern on a 125-line cache with LRU replacement.

Where:

$$\eta(0) = \epsilon(0) = 0$$
$$\eta'(0) = \delta'(0) \qquad \epsilon'(0) = \frac{m}{S}$$

The initial rates come from the observations that all lines immediately reused hit, and by substituting $\alpha'(0) = 1/S$ (Appendix C.2.2) into $\epsilon' = m\alpha'$.

One intuitive conclusion that will emerge from the following examples is that the miss rate is scale-invariant. That is, it does not matter in absolute terms how large the working set or cache is. What matters is how large the cache is relative to the working set. Hence, we define $\omega = N/S$ as the relative working set size.

D.4.1 Scans

When the program scans through N items, it has the reuse distance distribution:

$$\delta(d) = \begin{cases} 1 & \text{If } d \geq N \\ 0 & \text{Otherwise} \end{cases} \tag{D.12}$$

We make the following three observations: *(i)* Everything hits at age N if it is not evicted. *(ii)* All evictions occur at ages less than N. And *(iii)* no hits occur at ages less than N. Thus for ages $0 \leq a < N$:

$$\epsilon'' = -\frac{m}{S}\epsilon'$$
$$\Rightarrow \epsilon' = A\,\mathrm{e}^{-ma/S}$$

For some constant A. But we know $\epsilon'(0) = m/S$:

$$\epsilon' = \frac{m}{S}\mathrm{e}^{-ma/S}$$

So for some constant B:

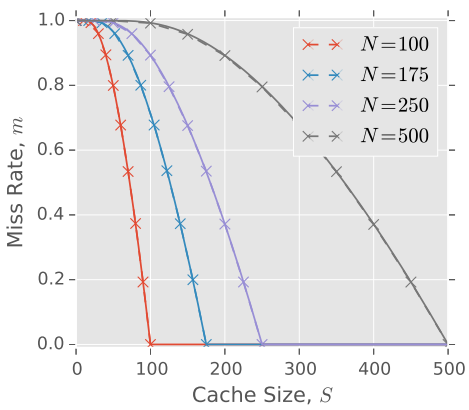
$$\epsilon = B - \mathrm{e}^{-ma/S}$$

We can now solve for the miss rate m:

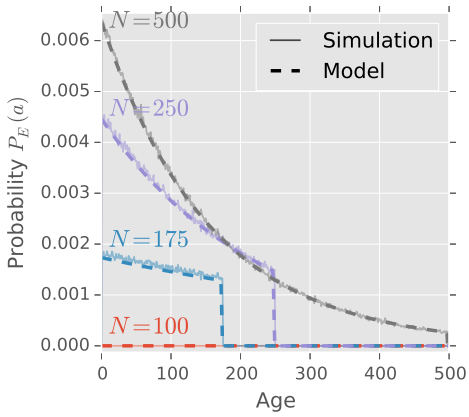
$$m = \lim_{a\to\infty} \epsilon = \int_0^\infty \epsilon'\,\mathrm{d}a$$

Recall that ϵ' is zero at ages larger than N, so:

$$= \int_0^N \epsilon'\,\mathrm{d}a = \epsilon(N) - \epsilon(0)$$
$$= 1 - \mathrm{e}^{-m\omega}$$



(a) Miss curves for four array sizes (N); model (solid lines) vs. simulation (×, dashes).



(b) Eviction probability distribution $\epsilon'(a)$ for cache size $S = 150$.

Figure D.15: Scanning pattern on random replacement.

Finally, we can solve for m using the product logarithm $W_0(x) = z \Rightarrow ze^z = x$:

$$m = 1 + \frac{W_0(-\omega e^{-\omega})}{\omega}, \tag{D.13}$$

One might think that the product logarithm inverts $x\,e^x$, and thus we have derived that $m = 0$. In fact, the product logarithm has two solutions. For the solution we choose (indicated by the zero subscript in W_0), the product logarithm only inverts $x\,e^x$ when $x \leq -1$, but takes a value between 0 and 1 otherwise.

In other words, when the array fits in the cache (i.e., $N \leq S$), the miss rate is zero. But when the array is larger than the cache (i.e., $N > S$), the miss rate is a positive number less than one. This is exactly how one would expect the cache to behave.

Figure D.15a shows example miss curves for different array sizes. The figure shows model predictions (solid lines) as well as simulation results (marked by ×s). The miss rate decreases rapidly until it hits zero when the array fits in the cache. Figure D.15b shows the modeled and simulated eviction distributions for a cache size of $S = 150$. The model closely matches simulation in both cases.

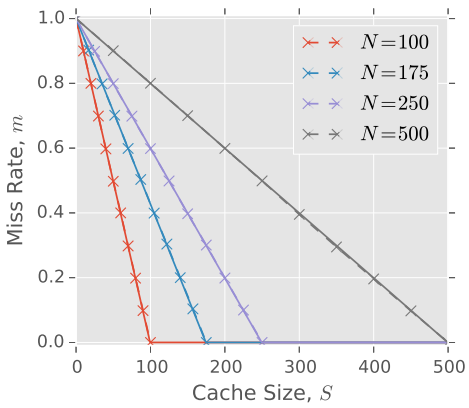
Random accesses D.4.2

Next, we solve for the miss rate when N addresses are accessed at random. As shown above, this yields an exponential distribution of reuse distances. Upon an access, each address has $1/N$ chance of being referenced. To simplify the math, let $\ell = \ln((N-1)/N)$. The reuse distance distribution is:

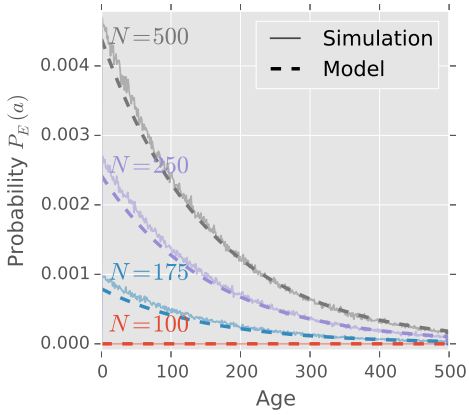
$$\begin{aligned} \delta &= 1 - \left(\frac{N-1}{N}\right)^d = 1 - e^{\ell d} \\ \delta' &= -\ell\,(1-\delta) \\ \delta'' &= -\ell^2\,(1-\delta) \end{aligned} \tag{D.14}$$

Conveniently, the δ terms cancel for hits in Equation D.7:

$$\eta'' = \ell\,(\eta' + \epsilon')$$



(a) Miss curves for four array sizes (N); model (solid lines) vs. simulation (×, dashes).



(b) Eviction probability distribution $\epsilon'(a)$ for cache size $S = 150$.

Figure D.16: Random access pattern on random replacement.

And evictions are unchanged:

$$\epsilon'' = -\frac{m}{S}(\eta' + \epsilon')$$

Hits and evictions thus have a similar form but differ by a constant factor. Assuming an exponential form for each and constraining solutions by the initial conditions above gives:

$$\eta' = -\ell \, \mathrm{e}^{(\ell - m/S)a}$$
$$\epsilon' = \frac{m}{S} \, \mathrm{e}^{(\ell - m/S)a}$$

It is easy to check that these forms satisfy the above equations.

We can now compute the miss rate:

$$m = \int_0^\infty \epsilon' \, \mathrm{d}a = -\frac{m/S}{\ell - m/S},$$

which means either m is zero or, if $S < N$:

$$= 1 + S \ln \frac{N-1}{N}$$

To simplify, consider how logarithms behave with large N. By definition:

$$\mathrm{e} = \lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x,$$

and thus for $x \gg 0$:

$$\mathrm{e} \approx \left(1 + \frac{1}{x}\right)^x \Rightarrow \ln\left(1 + \frac{1}{x}\right) \approx \frac{1}{x}$$

A symmetrical argument shows that:

$$\ln \frac{N-1}{N} \approx -\frac{1}{N}$$

And therefore:

$$m \approx 1 - \frac{1}{\omega} \tag{D.15}$$

In other words, to a good degree of approximation, the miss rate linearly decreases until the entire array fits in the cache, where it becomes zero. Figure D.16 shows several predicted miss curves and eviction distributions, which closely match simulation.

Stack D.4.3

When a program scans back and forth across N items in a last-in-first-out pattern, reuse distances are uniformly distributed up to a maximum value of $2N$. (The first element must wait for two passes over the array before it is re-referenced.) Stack access patterns have a reuse distance distribution of:

$$\delta(d) = \begin{cases} \frac{d}{2N} & \text{If } d < 2N \\ 1 & \text{Otherwise} \end{cases}$$

The derivatives for ages $a < 2N$ are:

$$\begin{aligned} \delta'(d) &= \frac{1}{2N} \\ \delta''(d) &= 0 \end{aligned} \tag{D.16}$$

The constant and zero derivatives greatly simplify hits, but not evictions:

$$\begin{aligned} \eta'' &= \frac{\epsilon'}{a-2N} \\ \epsilon'' &= -\frac{m}{S}(\eta' + \epsilon') \end{aligned}$$

These equations are satisfied by:

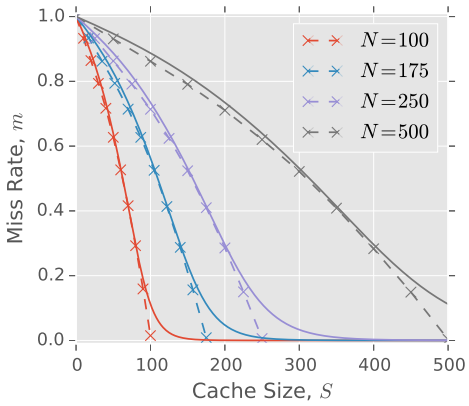
$$\begin{aligned} \eta' &= A\mathrm{e}^{-ma/S} \\ \epsilon' &= A\frac{m(2N-a)}{S}\mathrm{e}^{-ma/S} \end{aligned}$$

for some integration constant A. Solving for $\eta'(0) = \delta'(0) = 1/2N$ yields $A = 1/2N$. Note that, as expected, $\epsilon'(0) = m/S$ and $\epsilon'(2N) = 0$.

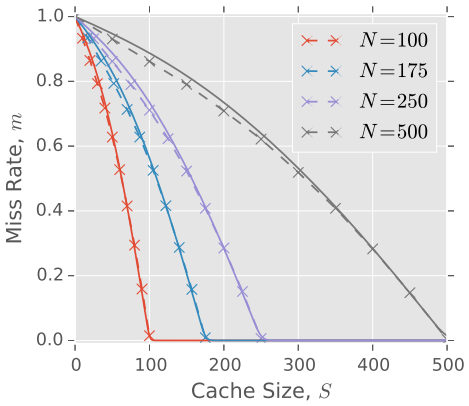
The cache's miss rate is given by:

$$m = \int_0^{2N} \epsilon' \, \mathrm{d}a = 1 + \frac{\mathrm{e}^{-2m\omega} - 1}{2m\omega} \tag{D.17}$$

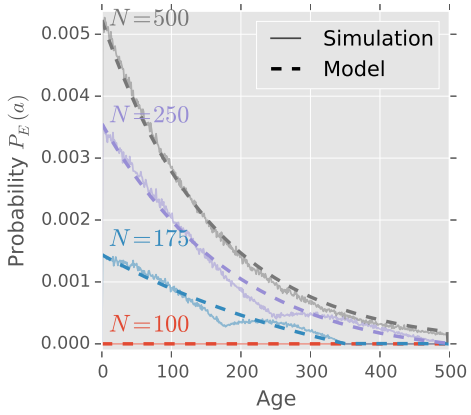
Unlike prior solutions, Equation D.17 does not yield an explicit formula for m. (We were unable to find a solution ourselves, nor could standard symbolic algebra software.) But it can be used to quickly iterate to a fixed-point on m by repeatedly evaluating the right-hand side of Equation D.17, without needing to



(a) Miss curves for four array sizes (N); model (solid lines) vs. simulation (×, dashes); 10 iterations of Equation D.17.



(b) Miss curves for four array sizes (N); model (solid lines) vs. simulation (×, dashes); 100 iterations of Equation D.17.



(c) Eviction probability distribution $\epsilon'(a)$ for cache size $S = 150$.

Figure D.17: Stack access pattern on random replacement.

perform numerical integration.

Figure D.17 compares the model and simulation. We present miss curves for ten and one hundred fixed-point iterations in Figures D.17a and D.17b. Fixed-point iteration takes longer to converge when the array barely fits in the cache, i.e. when $N \approx S$, as seen by comparing miss curves in the two figures. This is true of all fixed-point solutions in this chapter; Equation D.17 simply gives the result of a single iteration without requiring numerical integration. Finally, as discussed above, the iid reuse distance memory reference model introduces some error on stack distributions. This error can be seen in the eviction distributions (Figure D.17c). However, on random replacement the model is still quite accurate, as shown by the miss curves.

Approximations: Since an explicit formula for the miss rate of stack access patterns is unavailable, we now consider how to get accurate approximations. We start by approximating extreme behavior, and then extend these approximations to cover all cache sizes.

Consider how Equation D.17 behaves when $\omega \gg 1$, i.e. when the working set is much larger than

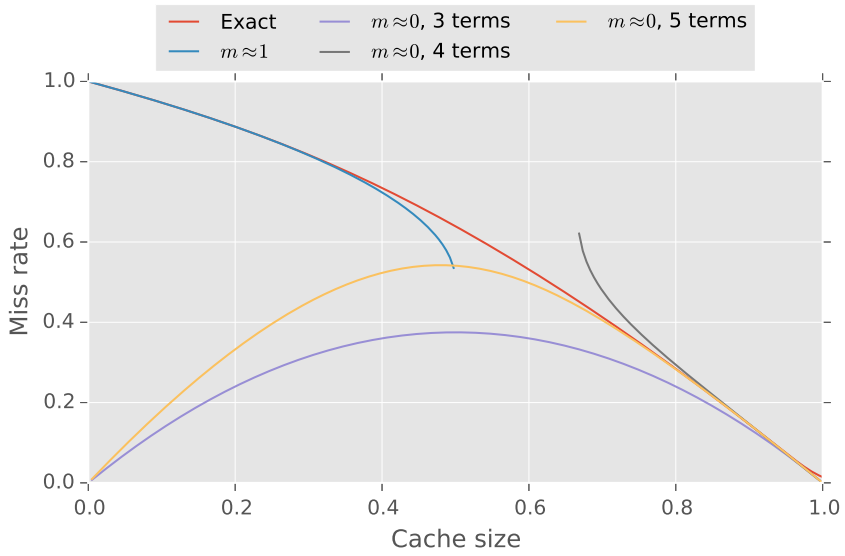


Figure D.18: Miss rate for the stack access pattern approximated at low and high miss rates. Approximations are accurate at extremes, but not moderate miss rates.

the cache. In such cases, the miss rate will be high, so $m\omega \gg 1$ and $e^{-2m\omega} \approx 0$. Thus:

$$\begin{aligned} m &\approx 1 - \frac{1}{2m\omega} \\ \Rightarrow m &\approx \frac{1}{2}\left(1 + \sqrt{1 - \frac{2}{\omega}}\right) \end{aligned} \tag{D.18}$$

Alternatively, when the cache mostly fits the working set, we expand Equation D.17 using the Taylor series for $e^{-2m\omega}$ around $m = 0$ (i.e., the MacLaurin series). For example, using the first three terms of the series and simplifying:

$$\begin{aligned} m &= m\omega - \frac{2}{3}m^2\omega^2 + O(m^3) \\ \Rightarrow m &\approx \frac{3}{2}\frac{\omega - 1}{\omega^2} \end{aligned} \tag{D.19}$$

However, this three-term approximation quickly loses accuracy as the miss rate increases. To increase accuracy, we can use more terms, but even numbers of terms quickly diverge. Using five terms increases accuracy, but still does not accurately predict moderate miss rates (e.g., around $m \approx 0.7$). Adding further terms improves accuracy, but the complexity of the equations does not scale well. (We do not give the approximate solutions with five or more terms because they are verbose.)

Figure D.18 illustrates the approximations so far. "Exact" gives one hundred fixed-point iterations of Equation D.17. "$m \approx 1$" is Equation D.18. "$m \approx 0$" is the Taylor expansions, e.g. $m \approx 0$ with 3 terms is Equation D.19.

To get approximations covering all miss rates, we can once again use the Taylor series, but this time on the previous approximations themselves. For example, the three-term series for Equation D.18 of

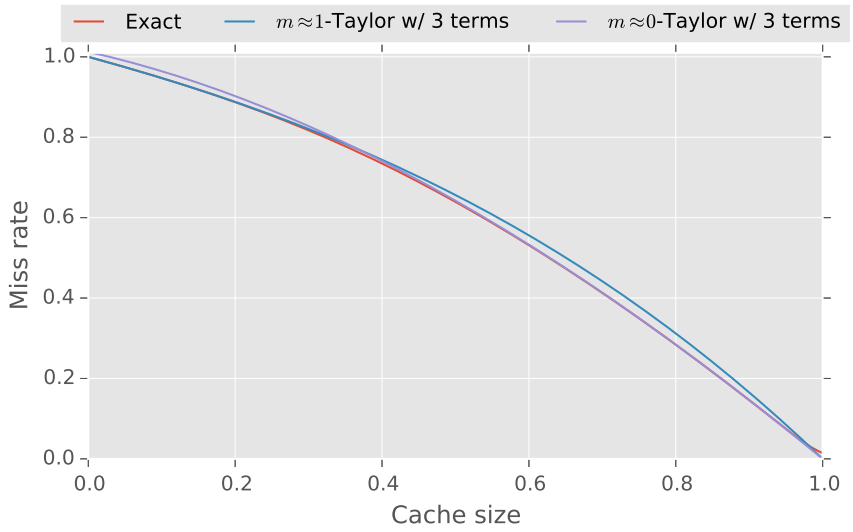


Figure D.19: Second-order approximations of the stack access pattern's miss rate. Approximations are accurate and cover the full range.

$1/\omega$ around $\omega = \infty$ gives:

$$m \approx 1 - \frac{1}{2\omega} - \frac{1}{4\omega^2} - \frac{1}{4\omega^3} + O\left(\frac{1}{\omega^4}\right) \tag{D.20}$$

This is a useful approximation because at $\omega = \infty$ it gives $m \approx 1$ and at $\omega = 1$ it gives $m \approx 0$. Both results match expectations.

Similarly, we can approximate solution of m we just obtained using the five-term Taylor expansion of $e^{-2m\omega}$. Except now we will use the Taylor expansion on $1/\omega$ around $\omega = 1$. We take three terms of the series, yielding:

$$m \approx \frac{9}{80}\left(\frac{1}{\omega} - 1\right)^3 - \frac{3}{8}\left(\frac{1}{\omega} - 1\right)^2 - \frac{3}{2}\left(\frac{1}{\omega} - 1\right) + O\left(\frac{1}{\omega}\right)^4 \tag{D.21}$$

This approximation gives $m \approx 0$ at $S = 1$, as expected, and $m \approx 81/80$ at $S = 0$, which is close to the true value of $m = 1$.

Figure D.19 shows that these second-order approximations cover the moderate miss rates that were missing in Figure D.18. Not only that, they also give good predictions of the miss rate across the entire range. The two approximations are accurate on opposite ends of the spectrum: $m \approx 0$ when the working set fits and $m \approx 1$ when it does not. But even at their worst accuracy, the approximations are quite accurate.

D.5 Closed-form solutions of fully-associative LRU

We can solve for the miss rate of LRU by making one simplifying assumption: we assume that there is a single *eviction age* a_E beyond which lines do not hit. The intuition behind this assumption is that, in a fully associative cache, evictions occur on average every $1/m$ accesses. Thus the oldest line in an LRU cache of size S will be approximately S/m, which gives the eviction age a_E. Of course, since we model cache accesses as a random process, choosing a single age is inaccurate—it would be better to model a probability distribution over a_E. But with large S, evictions will cluster around a single value and we

find the additional complexity of a probabilistic model unnecessary.

Skew-associative caches, zcaches, and hashed caches with many ways perform similar to a fully associative cache. We find that this simple assumption therefore accurately predicts the performance of many different access patterns. However, it does *not* accurately predict the performance of stack access patterns on LRU. The reason for its failure is itself interesting, and highlights the main source of error in the discrete cache model (Appendix C).

The basic framework for this model was presented as Equation B.2 in Appendix B. Cache space is divided between hits and evictions:

$$S = \overbrace{\sum_{a=1}^{\infty} a\ \mathrm{P}_H(a)}^{\text{Hits}} \quad + \quad \overbrace{\sum_{a=1}^{\infty} a\ \mathrm{P}_E(a)}^{\text{Evictions}} \tag{D.22}$$

We denote the space taken by hits and evictions as S_H and S_E, respectively. If we model that all lines are evicted upon reaching a_E, we obtain:

$$\begin{aligned} S_E = m\ a_E = S - S_H = S - \sum_{a=1}^{\infty} a\ \mathrm{P}_H(a) \\ \Rightarrow a_E = \frac{1}{m}\left(S - \sum_{a=1}^{\infty} a\ \mathrm{P}_H(a)\right) \end{aligned} \tag{D.23}$$

And because evictions occur only at a_E:

$$a_E = \frac{1}{m}\left(S - \sum_{a=1}^{a_E} a\ \mathrm{P}_D(a)\right) \tag{D.24}$$

Also:

$$m = \sum_{a=a_E}^{\infty} \mathrm{P}_D(a) = 1 - \sum_{a=1}^{a_E} \mathrm{P}_D(a) \tag{D.25}$$

Thus:

$$a_E = \frac{S - \sum_{a=1}^{a_E} a\ \mathrm{P}_D(a)}{1 - \sum_{a=1}^{a_E} \mathrm{P}_D(a)} \tag{D.26}$$

These equations yield a trivial discrete model for LRU on arbitrary reuse distance distributions: increment a_E starting from $a_E = 1$ until Equation D.26 is satisfied.

In terms of the continuous model presented throughout this chapter, Equation D.26 relaxes to:

$$m = 1 - \delta(a_E) \tag{D.27}$$

$$a_E = \frac{S - \int_0^{a_E} a\ \delta'(a)\,\mathrm{d}a}{1 - \delta(a_E)} \tag{D.28}$$

Finally, before giving examples on particular access patterns, this technique should also apply to fully-associative models of other policies. In particular, it should be straightforward to model PDP, but it should also generalize to an *eviction rank* r_E for well-behaved ranking functions. Thus this approach may generalize to a practical, discrete model of fully-associative replacement policies, obviating the need for expensive powers in the associative discrete model (Equation C.15).

D.5.1 Scans

Scans are an easy place to start, since the behavior of fully associative LRU on a scan is trivial: all misses when the array is larger than the cache, and all hits otherwise. The reuse distance distribution is the step function given by Equation D.12. Applying Equation D.28:

$$\int_0^a x\ \delta'(x)\,\mathrm{d}x = \begin{cases} N & \text{If } a \geq N \\ 0 & \text{Otherwise} \end{cases}$$
$$\Rightarrow a_E = \frac{S - \int_0^{a_E} x\ \delta'(x)\,\mathrm{d}x}{1-\delta(a_E)}$$

Thus, with $S < N$, the only solution is $a_E = S$, which in turn yields $m = 1$.

D.5.2 Random accesses

With random accesses, all replacement policies perform identically. We can show that the fully associative LRU model reproduces Equation D.15 that we obtained for random replacement. Starting from the reuse distance distribution (Equation D.14):

$$\delta = 1 - \mathrm{e}^{\ell d}, \tag{D.29}$$

we apply Equation D.28:

$$\int_0^a x\ \delta'(x)\,\mathrm{d}x = -\frac{1}{\ell}\left(1 + (\ell a - 1)\mathrm{e}^{\ell a}\right)$$
$$\Rightarrow a_E = \frac{S - \int_0^{a_E} x\ \delta'(x)\,\mathrm{d}x}{1-\delta(a_E)} = \frac{\ln(1+S\ell)}{\ell}$$

And then apply Equation D.27:

$$m = 1 - \delta(a_E) = \mathrm{e}^{a_E \ell} = 1 + S\ell$$

Using the approximation we introduced for random replacement, we obtain Equation D.15, as desired:

$$m \approx 1 - \frac{1}{\omega} \tag{D.30}$$

D.5.3 Stack

The model is inaccurate for fully-associative LRU on stack distributions. The reason is surprising: stack distributions on fully-associative LRU *hit at ages much larger than the eviction age*. This is because LRU is the optimal policy for stack access patterns, and it results in a large string of hits. Since there are no cache misses, there is no opportunity for the replacement policy to filter candidates, and candidates survive up to ages much larger than the eviction age. Consider a simple, cyclic access stream: ABCDDCBA. A cache with two lines will hit on half of the accesses, and it will evict lines at age two. *But it will hit at age four!*

We can nevertheless apply the model equations to derive its predictions. The reuse distance distribution is (Equation D.16):

$$\delta'(d) = \frac{1}{2N} \tag{D.31}$$

Applying Equation D.28,

$$\int_0^a x\ \delta'(x)\,\mathrm{d}x = \frac{a^2}{4N}$$

$$\Rightarrow a_E = \frac{S - \int_0^{a_E} x\ \delta'(x)\,\mathrm{d}x}{1-\delta(a_E)} = \frac{S - a_E^2/4N}{1 - a_E/2N} = 2\left(N - \sqrt{N^2 - NS}\right)$$

And applying Equation D.27:

$$m = 1 - \delta(a_E) = \sqrt{1 - S/N}$$

This is incorrect, since the right answer is $m = 1 - S/N$. But if we manually plug in the correct values for the space consumed by hits S_H and the miss rate, then we find the correct value for the eviction age $a_E = S$. This analysis tells us that the error is not with the equations for fully associative LRU, but with the underlying reference model itself. Specifically, stack access patterns have highly *dependent* reuse distances which are *non-uniformly* distributed. Thus, for these kinds of patterns, the iid model is a poor fit.

However, we observe that the only change required for the model to be accurate is to introduce a small number of misses, breaking the string of consecutive hits. These misses give the replacement policy a chance to filter the pool of candidates, and eliminate hits above the eviction age. Since stack patterns cache well in the private levels, we find empirically that pure stack access patterns are rare at the LLC. Thus, in practice the iid reuse distance model is accurate despite this flaw.

Summary D.6

This chapter has shown how to relax the general-purpose, discrete model from the previous section to a continuous system of ordinary differential equations. This formulation lets us use accurate, efficient numerical analysis to model random replacement and LRU, and moreover gives closed-form solutions for common access patterns. Although ODEs are in principle as general as the discrete model, the complexities of modeling the rank distribution make them most beneficial when modeling random or LRU replacement.

Bibliography

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Transactions on Computer Systems (TOCS)*, 7(2):184–215, 1989.

[2] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch. Unlocking bandwidth for GPUs in CC-NUMA systems. In *Proc. HPCA-21*, 2015.

[3] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000.

[4] A. Aho, P. Denning, and J. Ullman. Principles of optimal page replacement. *J. ACM*, 1971.

[5] A. Alameldeen and D. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), 2006.

[6] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proc. MICRO-32*, 1999.

[7] O. Bahat and A. Makowski. Optimal replacement policies for nonuniform cache objects with optional eviction. In *INFOCOM*, 2003.

[8] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: a compiler-managed memory system for raw machines. In *Proc. ISCA-26*, 1999.

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

[10] B. Beckmann and D. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proc. MICRO-37*, 2004.

[11] N. Beckmann. Constructing convex hulls from points of interest. http://people.csail.mit.edu/beckmann/misc/convex_hulls.swf. 2015.

[12] N. Beckmann and D. Sanchez. Jigsaw: Scalable Software-Defined Caches. In *Proc. PACT-22*, 2013.

[13] N. Beckmann and D. Sanchez. A cache model for modern processsors. Technical Report MIT-CSAIL-TR-2015-011, Massachusetts Institute of Technology, 2015.

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

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

[16] N. Z. Beckmann, C. Gruenwald III, C. R. Johnson, H. Kasture, F. Sironi, A. Agarwal, M. F. Kaashoek, and N. Zeldovich. Pika: A network service for multikernel operating systems. Technical Report MIT-CSAIL-TR-2014-002, Massachusetts Institute of Technology, 2014.

[17] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. PACT-17*, 2008.

[19] P. Billingsley. *Probability and measure*. Wiley, 2008.

[20] S. Bird and B. Smith. PACORA: Performance aware convex optimization for resource allocation. In *Proc. HotPar-3*, 2011.

[21] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, et al. Die stacking (3D) microarchitecture. In *Proc. MICRO-39*, 2006.

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

[23] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[24] D. Brooks, V. Tiwari, and M. Martonosi. *Wattch: a framework for architectural-level power analysis and optimizations*. 2000.

[25] R. L. Burden and J. D. Faires. *Numerical Analysis*. Brooks/Cole, 2001.

[26] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Proc. STOC-9*, 1977.

[27] J. Chang and G. Sohi. Cooperative caching for chip multiprocessors. In *Proc. ISCA-33*, 2006.

[28] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proc. DAC-37*, 2000.

[29] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *ISCA-32*, 2005.

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

[31] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ISCA-40*, 2013.

[32] C. Curtsinger and E. Berger. STABILIZER: statistically sound performance evaluation. In *Proc. ASPLOS-XVIII*, 2013.

[33] W. J. Dally. GPU Computing: To Exascale and Beyond. In *SC Plenary Talk*, 2010.

[34] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proc. HPCA-19*, 2013.

[35] S. Das, T. M. Aamodt, and W. J. Dally. SLIP: reducing wire energy in the memory hierarchy. In *Proc. ISCA-42*, 2015.

[36] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proc. ASPLOS-18*, 2013.

[37] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, 2013.

[38] J. Dean and L. Barroso. The Tail at Scale. *CACM*, 56, 2013.

[39] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), 1974.

[40] P. Denning. Thrashing: Its causes and prevention. In *Proc. AFIPS*, 1968.

[41] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proc. PLDI*, 2003.

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

[43] J. Doweck. Inside the CORE microarchitecture. In *Proc. HotChips-18*, 2006.

[44] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proc. MICRO-45*, 2012.

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

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

[47] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3), 2008.

[48] S. Franey and M. Lipasti. Tag Tables. In *Proc. HPCA-21*, 2015.

[49] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A sub-1w to 2w low-power processor for mobile internet devices and ultra-mobile PCs in 45nm hi-k metal gate CMOS. In *ISSCC*, 2008.

[50] C. M. Grinstead and J. L. Snell. *Introduction to probability*. American Mathematical Soc., 1998.

[51] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *Proc. MICRO-40*, 2007.

[52] Gurobi. Gurobi optimizer reference manual version 5.6, 2013.

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

[54] N. Hardavellas, I. Pandis, R. Johnson, and N. Mancheril. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *Proc. CIDR*, 2007.

[55] W. Hasenplaugh, P. S. Ahuja, A. Jaleel, S. Steely Jr, and J. Emer. The gradient-based cache partitioning algorithm. *ACM Trans. on Arch. and Code Opt. (TACO)*, 8(4):44, 2012.

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

[57] E. Herrero, J. González, and R. Canal. Distributed Cooperative Caching. In *Proc. PACT-17*, 2008.

[58] E. Herrero, J. González, and R. Canal. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proc. ISCA-37*, 2010.

[59] H. J. Herrmann. Geometrical cluster growth models and kinetic gelation. *Physics Reports*, 136(3):153–224, 1986.

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

[61] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. MAO–An extensible micro-architectural optimizer. In *Proc. CGO*, 2011.

[62] Intel. Knights Landing: Next Generation Intel Xeon Phi. In *SC Presentation*, 2013.

[63] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proc. SIGMETRICS*, 2007.

[64] 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.*, 18(8), 2007.

[65] S. Jahagirdar, V. George, I. Sodhi, and R. Wells. Power management of the third generation intel core micro architecture formerly codenamed ivy bridge. In *Hot Chips*, 2012.

[66] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proc. of the 17th intl. conf. on Parallel Architectures and Compilation Techniques*, 2008.

[67] A. Jaleel, M. Mattina, and B. Jacob. Last Level Cache (LLC) Performance of Data Mining Workloads On A CMP. In *HPCA-12*, 2006.

[68] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam, S. C. Steely, and J. Emer. CRUISE: Cache replacement and utility-aware scheduling. In *Proc. ASPLOS*, 2012.

[69] A. Jaleel, K. Theobald, S. C. S. Jr, and J. Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proc. of the 37th annual Intl. Symp. on Computer Architecture*, 2010.

[70] M. Javidi. Iterative methods to nonlinear equations. *Applied Mathematics and Computation*, 193(2), 2007.

[71] 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 *Proc. ISCA-40*, 2013.

[72] 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 *Proc. HPCA-16*, 2010.

[73] D. Jiménez. Insertion and promotion for tree-based pseudolru last-level caches. In *Proc. MICRO-46*, 2013.

[74] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proc. ASPLOS-XVI*, 2011.

[75] D. Kanter. Silvermont, IntelâĂŹs Low Power Architecture. 2013.

[76] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20, 1998.

[77] H. Kasture and D. Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *Proc. ASPLOS-19*, 2014.

[78] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5), 2011.

[79] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *Proc. ICCD-25*, 2007.

[80] G. Keramidas, P. Petoumenos, and S. Kaxiras. Where replacement algorithms fail: a thorough analysis. In *Proc. CF-7*, 2010.

[81] R. Kessler, M. Hill, and D. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE T. Comput.*, 43, 1994.

[82] S. Khan, Z. Wang, and D. Jimenez. Decoupled dynamic cache segmentation. In *Proc. HPCA-18*, 2012.

[83] S. M. Khan, Z. Wang, and D. A. Jiménez. Decoupled dynamic cache segmentation. In *Proc. HPCA-18*, 2012.

[84] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proc. of the 10th intl. symp. on High-Performance Computer Architecture*, 2004.

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

[86] R. Kirchain and L. Kimerling. A roadmap for nanophotonics. *Nature Photonics*, 2007.

[87] D. Knuth. *Axioms and hulls*. Springer-Verlag Berlin, 1992.

[88] A. Kolobov. Planning with markov decision processes: An ai perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1), 2012.

[89] N. Kurd, S. Bhamidipati, C. Mozak, J. Miller, T. Wilson, M. Nemani, and M. Chowdhury. Westmere: A family of 32nm IA processors. In *Proc. ISSCC*, 2010.

[90] G. Kurian, N. Beckmann, J. Miller, J. Psota, and A. Agarwal. Efficient Cache Coherence on Manycore Optical Networks. Technical Report MIT-CSAIL-TR-2010-009, Massachusetts Institute of Technology, 2010.

[91] H. Lee, S. Cho, and B. R. Childers. CloudCache: Expanding and shrinking private caches. In *Proc. HPCA-17*, 2011.

[92] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *Proc. ISCA-34*, 2007.

[93] B. Li, L. Zhao, R. Iyer, L. Peh, M. Leddige, M. Espig, S. Lee, and D. Newell. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *J. Par. Dist. Comp.*, 71(5), 2011.

[94] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO-42*, 2009.

[95] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. HPCA-14*, 2008.

[96] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *Proc. ISCA-42*, 2015.

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

[98] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, 2005.

[99] O. Madani, S. Hanks, and A. Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems. AAAI '99/IAAI '99.

[100] Z. Majo and T. R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proc. ISMM*, 2011.

[101] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic shared cache management (PriSM). In *Proc. ISCA-39*, 2012.

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

[103] M. Marty and M. Hill. Virtual hierarchies to support server consolidation. In *Proc. ISCA-34*, 2007.

[104] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[105] A. A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters*, 25(1), 1987.

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

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

[108] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010.

[109] G. Monahan. A survey of partially observable markov decision processes. *Management Science*, 28(1), 1982.

[110] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: A QoS framework for CMP architectures. *ACM SIGOPS Operating Systems Review*, 43(2), 2009.

[111] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. ASPLOS-XIV*, 2009.

[112] I. Newton, D. Bernoulli, C. MacLaurin, and L. Euler. *Philosophiae naturalis principia mathematica*, volume 1. excudit G. Brookman; impensis TT et J. Tegg, Londini, 1833.

[113] T. Nowatzki, M. Tarm, L. Carli, K. Sankaralingam, C. Estan, and B. Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *Proc. PLDI-34*, 2013.

[114] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4), 2010.

[115] D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint archive*, (2005/280), 2005.

[116] A. Pan and V. S. Pai. Imbalanced cache partitioning for balanced data-parallel programs. In *Proc. MICRO-46*, 2013.

[117] P. N. Parakh, R. B. Brown, and K. A. Sakallah. Congestion driven quadratic placement. In *Proc. DAC-35*, 1998.

[118] J. Park and W. Dally. Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures. In *Proc. SPAA-22*, 2010.

[119] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. P. Gibbons, M. A. Kozuch, and T. C. Mowry. Exploiting compressed block size as an indicator of future reuse. In *Proc. HPCA-21*, 2015.

[120] F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proc. HPCN*, 1996.

[121] J. Psota, J. Miller, G. Kurian, H. Hoffman, N. Beckmann, J. Eastep, and A. Agarwal. ATAC: Improving performance and programmability with on-chip optical networks. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010.

[122] M. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. Wiley, 2009.

[123] M. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proc. HPCA-10*, 2009.

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

[125] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proc. ISCA-34*, 2007.

[126] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proc. MICRO-45*, 2012.

[127] P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. In *Proc. ISCA-27*, 2000.

[128] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2), 2012.

[129] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proc. MICRO-43*, 2010.

[130] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. ISCA-38*, 2011.

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

[132] R. Sen and D. A. Wood. Reuse-based online models for caches. In *Proc. SIGMETRICS*, 2013.

[133] A. Seznec. A case for two-way skewed-associative caches. In *ISCA-20*, 1993.

[134] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proc. SPAA*, 2012.

[135] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2), 1985.

[136] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. ASPLOS-8*, 2000.

[137] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. In *Proc. Supercomputing (SC)*, 2009.

[138] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP control: Controlled shared cache management in chip multiprocessors. In *MICRO-42*, 2009.

[139] G. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for cmp/smt systems. *Journal of Supercomputing*, 2004.

[140] Sun Microsystems. UltraSPARC T2 supplement to the UltraSPARC architecture 2007. Technical report, 2007.

[141] M. Takagi and K. Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *Proc. ICS'04*, 2004.

[142] M. Takagi and K. Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *Proc. ICS-18*, 2004.

[143] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *WIOSCA*, 2007.

[144] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proc. Eurosys*, 2007.

[145] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *Proc. ASPLOS-XIV*, 2009.

[146] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[147] A. Tumanov, J. Wise, O. Mutlu, and G. R. Ganger. Asymmetry-aware execution placement on manycore chips. In *SFMA-3*, 2013.

[148] H. Vandierendonck and K. De Bosschere. XOR-based hash functions. *IEEE Trans. on Computers*, 54(7), 2005.

[149] K. Varadarajan, S. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular caches: A caching structure for dynamic creation of app-specific heterogeneous cache regions. In *MICRO-39*, 2006.

[150] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proc. ASPLOS*, 1996.

[151] R. Wang and L. Chen. Futility scaling: High-associativity cache partitioning. In *Proc. MICRO-47*, 2014.

[152] D. Wentzlaff, N. Beckmann, J. Miller, and A. Agarwal. Core Count vs Cache Size for Manycore Architectures in the Cloud. (MIT-CSAIL-TR-2010-008), 2010.

[153] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. A unified operating system for Clouds and Manycore: fos. In *Proceedings of the 1st Workshop on Computer Architecture and Operating Systems*, 2010.

[154] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proc. of the 1st Symp. on Cloud Computing*, 2010.

[155] D. Wong, H. W. Leong, and C. L. Liu. *Simulated annealing for VLSI design*. Kluwer Academic Publishers, 1988.

[156] H. Wong. Intel ivy bridge cache replacement policy, http://bit.ly/1BW8c7V, 2013.

[157] 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 *Proc. HPCA-16*, 2010.

[158] C. Wu and M. Martonosi. A Comparison of Capacity Management Schemes for Shared CMP Caches. In *WDDD-7*, 2008.

[159] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer. Ship: Signature-based hit predictor for high performance caching. In *Proc. MICRO-44*, 2011.

[160] M.-J. Wu, M. Zhao, and D. Yeung. Studying multicore processor scaling via reuse distance analysis. 2013.

[161] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. ISCA-36*, 2009.

[162] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proc. ISCA-40*, 2013.

[163] T. Yoshida, M. Hondou, T. Tabata, R. Kan, N. Kiyota, H. Kojima, K. Hosoe, and H. Okano. SPARC64 XIfx: Fujitsu's Next Generation Processor for HPC. Number 2, 2015.

[164] L. Youseff, N. Beckmann, H. Kasture, C. Gruenwald, D. Wentzlaff, and A. Agarwal. The case for elastic operating system services in fos. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012.

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

[166] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proc. EuroSys*, 2013.

[167] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. ASPLOS-XI*, 2004.

[168] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proc. ASPLOS*, 2010.