Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy*

Snehasish Kumar, Hongzhou Zhao†, Arrvindh Shriraman
Eric Matthews*, Sandhya Dwarkadas†, Lesley Shannon*

School of Computing Sciences, Simon Fraser University
†Department of Computer Science, University of Rochester
*School of Engineering Science, Simon Fraser University

Abstract

The fixed geometries of current cache designs do not adapt to the working set requirements of modern applications, causing significant inefficiency. The short block lifetimes and moderate spatial locality exhibited by many applications result in only a few words in the block being touched prior to eviction. Unused words occupy between 17—80% of a 64K L1 cache and between 1%—79% of a 1MB private LLC. This effectively shrinks the cache size, increases miss rate, and wastes on-chip bandwidth. Scaling limitations of wires mean that unused-word transfers comprise a large fraction (11%) of on-chip cache hierarchy energy consumption.

We propose Amoeba-Cache, a design that supports a variable number of cache blocks, each of a different granularity. Amoeba-Cache employs a novel organization that completely eliminates the tag array, treating the storage array as uniform and morphable between tags and data. This enables the cache to harvest space from unused words in blocks for additional tag storage, thereby supporting a variable number of tags (and correspondingly, blocks). Amoeba-Cache adjusts individual cache line granularities according to the spatial locality in the application. It adapts to the appropriate granularity both for different data objects in an application as well as for different phases of access to the same data. Overall, compared to a fixed granularity cache, the Amoeba-Cache reduces miss rate on average (geometric mean) by 18% at the L1 level and by 18% at the L2 level and reduces L1—L2 miss bandwidth by ≃46%. Correspondingly, Amoeba-Cache reduces on-chip memory hierarchy energy by as much as 36% (mcf) and improves performance by as much as 50% (art).

1 Introduction

A cache block is the fundamental unit of space allocation and data transfer in the memory hierarchy. Typically, a block is an aligned fixed granularity of contiguous words (1 word = 8bytes). Current processors fix the block granularity largely based on the average spatial locality across workloads, while taking tag overhead into consideration. Unfortunately, many applications (see Section 2 for details) exhibit low— moderate spatial locality and most of the words in a cache block are left untouched during the block's lifespan. Even for applications with good spatial behavior, the short lifespan of a block caused by cache geometry limitations can cause low cache utilization. Technology trends make it imperative that caching efficiency improves to reduce wastage of interconnect bandwidth. Recent reports from industry [2] show that on-chip networks can contribute up to 28% of total chip power. In the future an L2 — L1 transfer can cost up to 2.8× more energy than the L2 data access [14, 20]. Unused words waste ≃ 11% (4%—21% in commercial workloads) of the cache hierarchy energy.

*This material is based upon work supported in part by grants from the National Science and Engineering Research Council, MARCO Gigascale Research Center, Canadian Microelectronics Corporation, and National Science Foundation (NSF) grants CNS-0834451, CCF-1016902, and CCF-1217920.

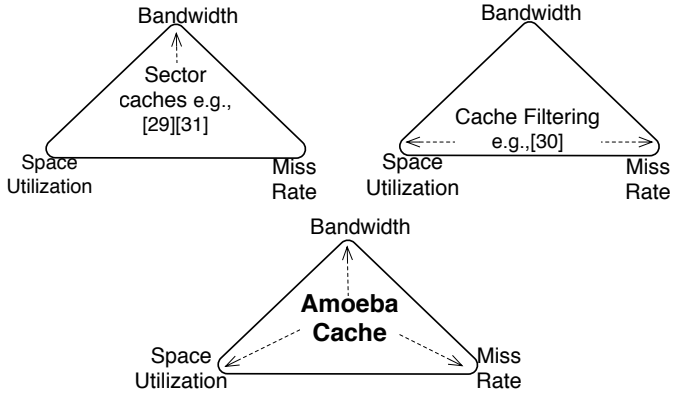


Figure 1: Cache designs optimizing different memory hierarchy parameters. Arrows indicate the parameters that are targeted and improved compared to a conventional cache.

Figure 1 organizes past research on cache block granularity along the three main parameters influenced by cache block granularity: miss rate, bandwidth usage, and cache space utilization. Sector caches have been used to [32, 15] minimize bandwidth by fetching only sub-blocks but miss opportunities for spatial prefetching. Prefetching [17, 29] may help reduce the miss rate for utilized sectors, but on applications with low—moderate or variable spatial locality, unused sectors due to misprediction, or unused regions within sectors, still pollute the cache and consume bandwidth. Line distillation [30] filters out unused words from the cache at evictions using a separate word-granularity cache. Other approaches identify dead cache blocks and replace or eliminate them eagerly [19, 18, 13, 21]. While these approaches improve utilization and potentially miss rate, they continue to consume bandwidth and interconnect energy for the unutilized words. Word-organized cache blocks also dramatically increase cache associativity and lookup overheads, which impacts their scalability.

Determining a fixed optimal point for the cache line granularity at hardware design time is a challenge. Small cache lines tend to fetch fewer unused words, but impose significant performance penalties by missing opportunities for spatial prefetching in applications with high spatial locality. Small line sizes also introduce high tag overhead, increase lookup energy, and increase miss processing overhead (e.g., control messages). Larger cache line sizes minimize tag overhead and effectively prefetch neighboring words but introduce the negative effect of unused words that increase network bandwidth. Prior approaches have proposed the use of multiple caches with different block sizes [33, 12]. These approaches require word granularity caches that increase lookup energy, impose high tag overhead (e.g., 50% in [33]), and reduce cache efficiency when there is good spatial locality.

In this paper, we propose a novel cache architecture, *Amoeba-Cache*, to improve memory hierarchy efficiency by supporting fine-grain (per-miss) dynamic adjustment of cache block size and the # of blocks per set. To enable variable granularity blocks within the same cache, the tags maintained per set need to grow and shrink as the # of blocks/set vary. *Amoeba-Cache* eliminates the conventional tag array and collocates the tags with the cache blocks in the data array. This enables us to segment and partition a cache set in different ways: For example, in a configuration comparable to a traditional 4-way 64K cache with 256 sets (256 bytes per set), we can hold eight 32-byte cache blocks, thirty-two 8-byte blocks, or any other collection of cache blocks of varying granularity. Different sets may hold blocks of different granularity, providing maximum flexibility across address regions of varying spatial locality. The *Amoeba-Cache* effectively filters out unused words in a conventional block and prevents them from being inserted into the cache, allowing the resulting free space to be used to hold tags or data of other useful blocks. The *Amoeba-Cache* can adapt to the available spatial locality; when there is low spatial locality, it will hold many blocks of small granularity and when there is good spatial locality, it can adapt and segment the cache into a few big blocks.

Compared to a fixed granularity cache, *Amoeba-Cache* improves cache utilization by 90% - 99% for most applications, saves miss rate by up to 73% (omnetpp) at the L1 level and up to 88% (twolf) at the LLC level, and reduces miss bandwidth by up to 84% (omnetpp) at the L1 and 92% (twolf) at the LLC. We compare against other approaches such as Sector Cache and Line distillation and show that *Amoeba-Cache* can optimize miss rate and bandwidth better across many applications, with lower hardware overhead. Our synthesis of the cache controller hit path shows that *Amoeba-Cache* can be implemented with low energy impact and 0.7% area overhead for a latency- critical 64K L1.

The overall paper is organized as follows: § 2 provides quantitative evidence for the acuteness of the spatial locality problem. § 3 details the internals of the *Amoeba-Cache* organization and § 4 analyzes the physical implementation overhead. § 5 deals with wider chip-level issues (i.e., inclusion and coherence). § 6 — § 10 evaluate the *Amoeba-Cache*, commenting on the optimal block granularity, impact on overall on-chip energy, and performance improvement. § 11 outlines related work.

2 Motivation for Adaptive Blocks

In traditional caches, the cache block defines the fundamental unit of data movement and space allocation in caches. The blocks in the data array are uniformly sized to simplify the insertion/removal of blocks, simplify cache refill requests, and support low complexity tag organization. Unfortunately, conventional caches are inflexible (fixed block granularity and fixed # of blocks) and caching efficiency is poor for applications that lack high spatial locality. Cache blocks influence multiple system metrics including bandwidth, miss rate, and cache utilization. The block granularity plays a key role in exploiting spatial locality by effectively prefetching neighboring words all at once. However, the neighboring words could go unused due to the low lifespan of a cache block. The unused words occupy interconnect bandwidth and pollute the cache, which increases the # of misses. We evaluate the influence of a fixed granularity block below.

2.1 Cache Utilization

In the absence of spatial locality, multi-word cache blocks (typically 64 bytes on existing processors) tend to increase cache pollution and fill the cache with words unlikely to be used. To quantify this pollution, we segment the cache line into words (8 bytes) and track the words touched before the block is evicted. We define utilization as the average # of words touched in a cache block before it is evicted. We study a comprehensive collection of workloads from a variety of domains: 6 from PARSEC [3], 7 from SPEC2006, 2 from SPEC2000, 3 Java workloads from DaCapo [4], 3 commercial workloads (Apache, SpecJBB2005, and TPC-C [22]), and the Firefox web browser. Subsets within benchmark suites were chosen based on demonstrated miss rates on the fixed granularity cache (i.e., whose working sets did not fit in the cache size evaluated) and with a spread and diversity in cache utilization. We classify the benchmarks into 3 groups based on the utilization they exhibit: Low (<33%), Moderate (33%—66%), and High (66%+) utilization (see Table 1).

Table 1: Benchmark Groups

Group	Utilization %	Benchmarks
Low	0 — 33%	art, soplex, twolf, mcf, canneal, lbm, omnetpp
Moderate	34 — 66%	astar, h2, jbb, apache, x264, firefox, tpc-c, freqmine, fluidanimate
High	67 — 100%	tradesoap, facesim, eclipse, cactus, milc, ferret

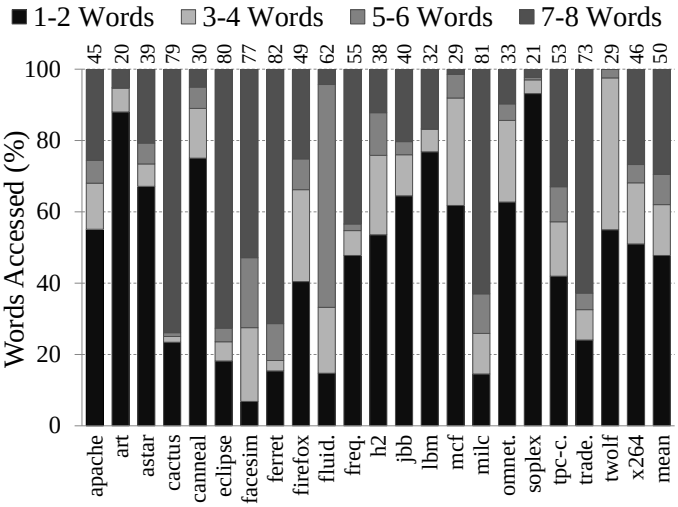


Figure 2: Distribution of words touched in a cache block. Avg. utilization is on top. (Config: 64K, 4 way, 64-byte block.)

Figure 2 shows the histogram of words touched at the time of eviction in a cache line of a 64K, 4-way cache (64-byte block, 8 words per block) across the different benchmarks. Seven applications have less than 33% utilization and 12 of them are dominated (>50%) by 1-2 word accesses. In applications with good spatial locality (cactus, ferret, tradesoap, milc, eclipse) more than 50% of the evicted blocks have 7-8 words touched. Despite similar average utilization for applications such as astar and h2 (39%), their distributions are dissimilar; ≃70% of the blocks in astar have 1-2 words accessed at the time of eviction, whereas ≃50% of the blocks in h2 have 1-2 words accessed per block. Utilization for a single application also changes over time; for example, ferret's average utilization, measured as the average fraction of words used in evicted cache lines over 50 million instruction windows, varies from 50% to 95% with a periodicity of roughly 400 million instructions.

2.2 Effect of Block Granularity on Miss Rate and Bandwidth

Cache miss rate directly correlates with performance, while under current and future wire-limited technologies [2], bandwidth directly correlates with dynamic energy. Figure 3 shows the influence of block granularity on miss rate and bandwidth for a 64K L1 cache and a 1M L2 cache keeping the number of ways constant. For the 64K L1, the plots highlight the pitfalls of simply decreasing the block size to accommodate the Low group of applications; miss rate increases by 2× for the High group when the block size is changed from 64B to 32B; it increases by 30% for the Moderate group. A smaller block size decreases bandwidth proportionately but increases miss rate. With a 1M L2 cache, the lifetime of the cache lines increases significantly, improving overall utilization. Increasing the block size from 64→256 halves the miss rate for all application groups. The bandwidth is increased by 2× for the Low and Moderate.

Since miss rate and bandwidth have different optimal block granularities, we use the following metric: $\frac{1}{MissRate \times Bandwidth}$ to determine a fixed block granularity suited to an application that takes both criteria into account. Table 2 shows the block size that maximizes the metric for each application. It can be seen that different applications have different block granularity requirements. For example, the metric is maximized for apache at 128 bytes and for firefox (similar utilization) at 32 bytes. Furthermore, the optimal block sizes vary with the cache size as the cache lifespan changes. This highlights the challenge of picking a single block size at design time especially when the working set does not fit in the cache.

2.3 Need for adaptive cache blocks

Our observations motivate the need for adaptive cache line granularities that match the spatial locality of the data access patterns in an application. In summary:

- Smaller cache lines improve utilization but tend to increase miss rate and potentially traffic for applications with good spatial locality, affecting the overall performance.
- Large cache lines pollute the cache space and interconnect with unused words for applications with poor spatial locality, significantly decreasing the caching efficiency.
- Many applications waste a significant fraction of the cache space. Spatial locality varies not only across applications but also within each application, for different data structures as well as different phases of access over time.

Table 2: Optimal block size. Metric: $\frac{1}{\text{Miss}-\text{rate} \times \text{Bandwidth}}$

64K, 4-way	
Block	Benchmarks
32B	cactus, eclipse, facesim, ferret, firefox, fluidanimate,freqmine, milc, tpc-c, tradesoap
64B	art
128B	apache, astar, canneal, h2, jbb, lbm, mcf, omnetpp, soplex, twolf, x264
1M, 8-way	
Block	Benchmarks
64B	apache, astar, cactus, eclipse, facesim, ferret, firefox, freqmine, h2, lbm, milc, omnetpp, tradesoap, x264
128B	art
256B	canneal, fluidanimate, jbb, mcf, soplex, tpc-c, twolf

3 Amoeba-Cache : Architecture

The *Amoeba-Cache* architecture enables the memory hierarchy to fetch and allocate space for a range of words (a variable granularity

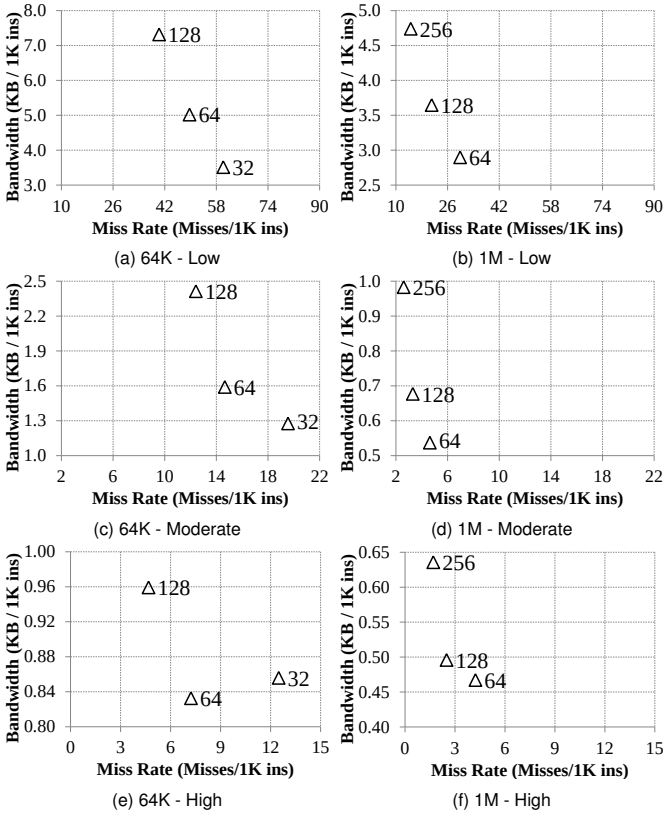


Figure 3: Bandwidth vs. Miss Rate. (a),(c),(e): 64K, 4-way L1. (b),(d),(f): 1M, 8-way LLC. Markers on the plot indicate cache block size. Note the different scales for different groups.

cache block) based on the spatial locality of the application. For example, consider a 64K cache (256 sets) that allocates 256 bytes per set. These 256 bytes can adapt to support, for example, eight 32-bytes blocks, thirty-two 8-byte blocks, or four 32-byte blocks and sixteen 8-byte blocks, based on the set of contiguous words likely to be accessed. The key challenge to supporting variable granularity blocks is how to grow and shrink the # of tags as the # of blocks per set vary with block granularity? *Amoeba-Cache* adopts a solution inspired by software data structures, where programs hold metadata and actual data entries in the same address space. To achieve maximum flexibility, *Amoeba-Cache* completely eliminates the tag array and collocates the tags with the actual data blocks (see Figure 4). We use a bitmap (`T? Bitmap`) to indicate which words in the data array represent tags. We also decouple the conventional valid/invalid bits (typically associated with the tags) and organize them into a separate array (V? : Valid bitmap) to simplify block replacement and insertion. V? and T? bitmaps both require 1 bit for very word (64bits) in the data array (total overhead of 3%). *Amoeba-Cache* tags are represented as a range (`Start` and `End` address) to support variable granularity blocks. We next discuss the overall architecture.

3.1 Amoeba Blocks and Set-Indexing

The *Amoeba-Cache* data array holds a collection of varied granularity *Amoeba-Block*s that do not overlap. Each *Amoeba-Block* is a 4 tuple consisting of `<Region Tag, Start, End, Data-Block>` (Figure 4). A `Region` is an aligned block of memory of size `RMAX` bytes. The boundaries of any *Amoeba-Block* block (`Start` and `End`) always will lie within the regions' boundaries. The minimum granularity of the data in an *Amoeba-Block* is 1 word

and the maximum is RMAX words. We can encode Start and End in $log_2(RMAX)$ bits. The set indexing function masks the lower $log_2(RMAX)$ bits to ensure that all *Amoeba-Block*s (every memory word) from a region index to the same set. The Region Tag and Set-Index are identical for every word in the *Amoeba-Block*. Retaining the notion of *sets* enables fast lookups and helps elude challenges such as synonyms (same memory word mapping to different sets). When comparing against a conventional cache, we set RMAX to 8 words (64 bytes), ensuring that the set indexing function is identical to that in the conventional cache to allow for a fair evaluation.

3.2 Data Lookup

Figure 5 describes the steps of the lookup. ❶ The lower $log_2(RMAX)$ bits are masked from the address and the set index is derived from the remaining bits. ❷ In parallel with the data read out, the T? bitmap activates the words in the data array corresponding to the tags for comparison. Note that given the minimum size of a *Amoeba-Block* is two words (1 word for the tag metadata, 1 word for the data), adjacent words cannot be tags. We need $\frac{N_{words}}{2}$ 2-1 multiplexers that route one of the adjacent words to the comparator ($\in$ operator). The comparator generates the hit signal for the word selector. The $\in$ operator consists of two comparators: a) an aligned Region tag comparator, a conventional == (64 - $log_2 N_{sets}$ - $log_2 RMAX$ bits wide, e.g., 50 bits) that checks if the *Amoeba-Block* belongs to the same region and b) a $Start <= W < END$ range comparator ($log_2 RMAX$ bits wide; e.g., 3 bits) that checks if the *Amoeba-Block* holds the required word. Finally, in ❸, the tag match activates and selects the appropriate word.

3.3 Amoeba Block Insertion

On a miss for the desired word, a spatial granularity predictor is invoked (see Section 5.1), which specifies the range of the *Amoeba-Block* to refill. To determine a position in the set to slot the incoming block we can leverage the V? (Valid bits) bitmap. The V? bitmap has 1 bit/word in the cache; a "1" bit indicates the word has been allocated (valid data or a tag). To find space for an incoming data block we perform a substring search on the V? bitmap of the cache set for contiguous 0s (empty words). For example, to insert an *Amoeba-Block* of five words (four words for data and one word for tag), we perform a substring search for *00000* in the V? bitmap / set (e.g., 32 bits wide for a 64K cache). If we cannot find the space, we keep triggering the replacement algorithm until we create the requisite contiguous space. Following this, the *Amoeba-Block* tuple (Tag and Data block) is inserted, and the corresponding bits in the T? and V? array are set. The 0s substring search can be accomplished with a lookup table; many current processors already include a substring instruction [11, PCMISTRI].

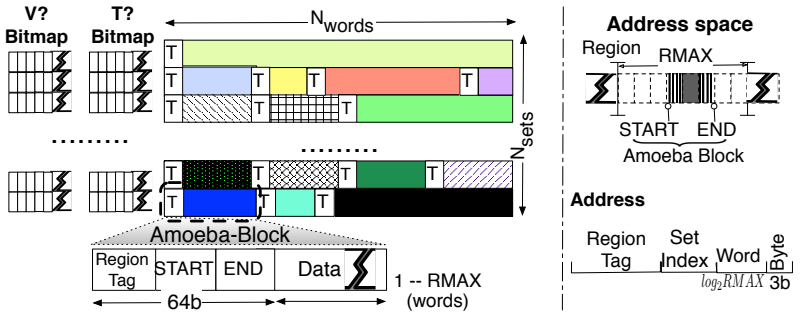
3.4 Replacement : Pseudo LRU

To reclaim the space from an *Amoeba-Block* the tag bits T? (tag) and V? (valid) bits corresponding to the block are unset. The key issue is identifying the *Amoeba-Block* to replace. Classical pseudo-LRU algorithms [16, 26, 16] keep the metadata for the replacement algorithm separate from the tags to reduce port contention. To be compatible with pseudo-LRU and other algorithms such as DIP [27] that work with a fixed # of ways, we can logically partition a set in *Amoeba-Cache* into N_{ways}. For instance, if we partition a 32 word cache set into 4 logical ways, any access to an *Amoeba-Block* tag found in words 0 —7 of the set is treated as an access to logical way 0. Finding a replacement candidate involves identifying the selected replacement way and then picking (possibly randomly) a candidate *Amoeba-Block* . More refined replacement algorithms that require per-tag metadata can harvest the space in the tag-word of the *Amoeba-Block* which is 64 bits wide (for alignment purposes) while physical addresses rarely extend beyond 48 bits.

3.5 Partial Misses

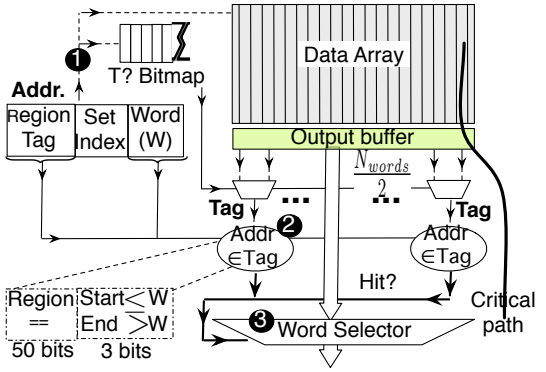
With variable granularity data blocks, a challenging although rare case (5 in every 1K accesses) that occurs is a *partial miss*. It is observed primarily when the spatial locality changes. Figure 6 shows an example. Initially, the set contains two blocks from a region R, one *Amoeba-Block* caches words 1 –3 (Region:R, START:1 END:3) and the other holds words 5 –6 (Region:R START:5 END:6). The CPU reads word 4, which misses, and the spatial predictor requests an *Amoeba-Block* with range START:0 and END:7. The cache has *Amoeba-Block*s that hold subparts of the incoming *Amoeba-Block*, and some words (0, 4, and 7) need to be fetched.

Amoeba-Cache removes the overlapping sub-blocks and allocates a new *Amoeba-Block*. This is a multiple step process: ❶ On a miss, the cache identifies the overlapping sub-blocks in the cache using the tags read out during lookup. $\cap \neq NULL$ is true if $START_{new} < END_{Tag}$ and $END_{new} > START_{Tag}$ (New = incoming block and



Left: Cache Layout. Size: $N_sets*N_words*8 bytes$. T? (Tag map) 1 bit/word. Is word a a tag? Yes(1). V? (Valid map): 1 bit/word. Is word allocated? Yes(1). Total overhead: 3%. **Right:** Address space layout: Region: Aligned regions of size RMAX words (×8 bytes). *Amoeba-Block* range cannot exceed Region boundaries. Cache Indexing: Lower $log_2 RMAX$ bits masked and set index is derived from remaining bits; all words (and all *Amoeba-Block*s) within a region index to the same set.

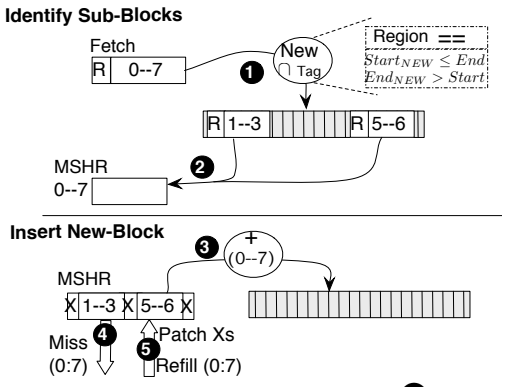
Figure 4: Amoeba-Cache Architecture.



Lookup steps. ❶ Mask lower $log_2 RMAX$ bits and index into data array. ❷ Identify tags using T? bitmap and initiate tag checks ($\in$) operator (Region tag comparator ==? and $\leq$ range comparator). ❸ Word selection within *Amoeba-Block*.

Figure 5: Lookup Logic

Tag = *Amoeba-Block* in set). ❷ The data blocks that overlap with the miss range are evicted and moved one-at-a-time to the MSHR entry. ❸ Space is then allocated for the new block, i.e., it is treated like a new insertion. ❹ A miss request is issued for the entire block (START:0 — END:7) even if only some words (e.g., 0, 4, and 7) may be needed. This ensures request processing is simple and only a single refill request is sent. ❺ Finally, the incoming data block is patched into the MSHR; only the words not obtained from the L1 are copied (since the lower level could be stale).



❶ Identify blocks overlapping with `New` block. ❷ Evict overlapping blocks to MSHR. ❸ Allocate space for new block (treat it like a new insertion). ❹ Issue refill request to lower level for entire block. ❺ Patch only newer words as lower-level data could be stale.

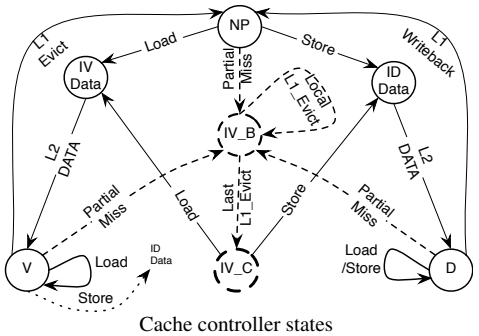
Figure 6: Partial Miss Handling. Upper: Identify relevant sub-blocks. Useful for other cache controller events as well, e.g., recalls. Lower: Refill of words and insertion.

4 Hardware Complexity

We analyze the complexity of *Amoeba-Cache* along the following directions: we quantify the additions needed to the cache controller, we analyze the latency, area, and energy penalty, and finally, we study the challenges specifically introduced by large caches.

4.1 Cache Controller

The variable granularity *Amoeba-Block* blocks need specific consideration in the cache controller. We focus on the L1 controller here, and in particular, partial misses. The cache controller manages operations at the aligned RMAX granularity. The controller permits only one in-flight cache operation per RMAX region. In-flight cache operations ensure no address overlap with stable *Amoeba-Block*s in order to eliminate complex race conditions. Figure 7 shows the L1 cache controller state machine. We add two states to the default protocol, `IV_B` and `IV_C`, to handle partial misses. `IV_B` is a blocking state that blocks other cache operations to RMAX region until all relevant *Amoeba-Block*s to a partial miss are evicted (e.g., 0–3 and 5–7 blocks in Figure 6). `IV_C` indicates partial miss completion. This enables the controller to treat the access as a full miss and issue the refill request. The other stable states (I, V, D) and transient states (`IV_Data` and `ID_Data`) are present in a conventional protocol as well. `Partial-miss` triggers the clean-up operations (`1` and `2` in Figure 6). `Local_L1_Evict` is a looping event that keeps retriggering for each *Amoeba-Block* involved in the partial miss; `Last_L1_Evict` is triggered when the last *Amoeba-Block* involved in the partial miss is evicted to the MSHR. A key difference



Cache controller states

State	Description
NP	*Amoeba-Block* not present in the cache.
V	All words corresponding to *Amoeba-Block* present and valid (read-only)
D	Valid and atleast one word in *Amoeba-Block* is dirty (read-write)
IV_B	Partial miss being processed (blocking state)
IV_Data	Load miss; waiting for data from L2
ID_Data	Store miss; waiting for data. Set dirty bit.
IV_C	Partial miss cleanup from cache completed (treat as full miss)
Amoeba-specific Cache Events	
Partial miss: Process partial miss.	
Local_L1_Evict: Remove overlapping *Amoeba-Block* to MSHR.	
Last_L1_Evict: Last *Amoeba-Block* moved to MSHR. Convert to full miss and process load or store.	

Bold and Broken-lines: *Amoeba-Cache* additions.

Figure 7: Amoeba Cache Controller (L1 level).

between the L1 and lower-level protocols is that the Load/Store event in the lower-level protocol may need to access data from multiple *Amoeba-Block*s. In such cases, similar to the partial-miss event, we read out each block independently before supplying the data (more details in § 5.2).

4.2 Area, Latency, and Energy Overhead

The extra metadata required by *Amoeba-Cache* are the T? (1 tag bit per word) and V? (1 valid bit per word) bitmaps. Table 3 shows the quantitative overhead compared to the data storage. Both the T? and V? bitmap arrays are directly proportional to the size of the cache and require a constant storage overhead (3% in total). The T? bitmap is read in parallel with the data array and does not affect the critical path; T? adds 2%—3.5% (depending on cache size) to the overall cache access energy. V? is referred only on misses when inserting a new block.

Table 3: *Amoeba-Cache* Hardware Complexity.

Cache configuration			
	64K (256by/set)	1MB (512by/set)	4MB (1024by/set)
Data RAM parameters			
Delay	0.36ns	2ns	2.5 ns
Energy	100pJ	230pJ	280pJ
Amoeba-Cache components (CACTI model)			
T?/V? map	1KB	16KB	64KB
Latency	0.019ns (5%)	0.12ns (6%)	0.2ns (6%)
Energy	2pJ (2%)	8pJ (3.4%)	10pJ (3.5%)
LRU	$\frac{1}{8}$KB	2KB	8KB
Lookup Overhead (VHDL model)			
Area	0.7%	0.1%	
Latency	0.02ns	0.035ns	0.04ns

% indicates overhead compared to data array of cache. 64K cache operates in *Fast mode*; 1MB and 4MB operate in *Normal mode*. We use 32nm ITRS HP transistors for 64K and 32nm ITRS LOP transistors for 1MB and 4MB.

We synthesized[1] the cache lookup logic using Synopsys and quantify the area, latency, and energy penalty. *Amoeba-Cache* is compatible with *Fast* and *Normal* cache access modes [28, -access-mode config], both of which read the entire set from the data array in parallel with the way selection to achieve lower latency. *Fast* mode transfers the entire set to the edge of the H-tree, while *Normal* mode, only transmits the selected way over the H-tree. For synthesis, we used the Synopsys design compiler (Vision Z-2007.03-SP5).

Figure 5 shows *Amoeba-Cache*ś lookup hardware on the critical path; we compare it against a fixed-granularity cache's lookup logic (mainly the comparators). The area overhead of the *Amoeba-Cache* includes registering an entire line that has been read out, the tag operation logic, and the word selector. The components on the critical path once the data is read out are the 2-way multiplexers, the $\in$ comparators, and priority encoder that selects the word; the T? bitmap is accessed in parallel and off the critical path. *Amoeba-Cache* is made feasible under today's wire-limited technology where the cache latency and energy is dominated by the bit/word lines, decoder, and H-tree [28]. *Amoeba-Cache*'s comparators, which operate on the entire cache set, are 6$\times$ the area of a fixed cache's comparators. Note that the data array occupies 99% of the overall cache area. The critical path is dominated by the wide word selector since the comparators all operate in parallel. The lookup logic adds 60% to the conventional cache's comparator time. The overall critical path is dominated by the data array access and *Amoeba-Cache*'s lookup circuit adds 0.02ns to the access latency and $\simeq$ 1pJ to the energy of a 64K cache, and 0.035ns to the latency and $\simeq$2pJ to the energy of a 1MB cache. Finally, *Amoeba-Cache* amortizes the energy penalty of the peripheral components (H-tree, Wordline, and decoder) over a single RAM.

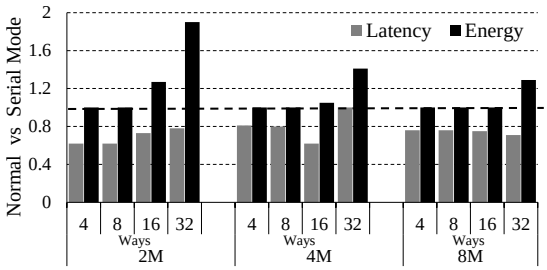
Amoeba-Cache's overhead needs careful consideration when implemented at the L1 cache level. We have two options for handling the latency overhead a) if the L1 cache is the critical stage in the pipeline, we can throttle the CPU clock by the latency overhead to ensure that the additional logic fits within the pipeline stage. This ensures that the number of pipeline stages for a memory access does not change with respect to a conventional cache, although all instructions bear the overhead of the reduced CPU clock. b) we can add an extra pipeline stage to the L1 hit path, adding a 1 cycle overhead to all memory accesses but ensuring no change in CPU frequency. We quantify the performance impact of both approaches in Section 6.

4.3 Tag-only Operations

Conventional caches support tag-only operations to reduce data port contention. While the *Amoeba-Cache* merges tags and data, like many commercial processors it decouples the replacement metadata and valid bits from the tags, accessing the tags only on cache lookup. Lookups can be either CPU side or network side (coherence invalidation and Wback/Forwarding). CPU-side lookups and writebacks ($\simeq$ 95% of cache operations) both need data and hence *Amoeba-Cache* in the common case does not introduce extra overhead. *Amoeba-Cache* does read out the entire data array unlike serial-mode caches (we discuss this issue in the next section). Invalidation checks and snoops can be more energy expensive with *Amoeba-Cache* compared to a conventional cache. Fortunately, coherence snoops are not common in many applications (e.g., 1/100 cache operations in SpecJBB) as a coherence directory and an inclusive LLC filter them out.

[1]We do not have access to an industry-grade 32nm library, so we synthesized at a higher 180nm node size and scaled the results to 32 nm (latency and energy scaled proportional to Vdd (taken from [36]) and Vdd^2 respectively).

4.4 Tradeoff with Large Caches



Baseline: Serial. $\leq$ 1 Normal is better. 32nm, ITRS LOP.

Figure 8: Serial vs Normal mode cache.

Large caches with many words per set ($\equiv$ highly associative conventional cache) need careful consideration. Typically, highly associative caches tend to serialize tag and data access with only the relevant cache block read out on a hit and no data access on a miss. We first analyze the the tradeoff between reading the entire set (normal mode), which is compatible with *Amoeba-Cache* and only the relevant block (serial mode). We vary the cache size from 2M—8M and associativity from 4(256B/set) — 32 (2048B/set). Under current technology constraints (Figure 8), only at very high associativity does serial mode demonstrate a notable energy benefit. Large caches are dominated by H-tree energy consumption and reading out the entire set at each sub-bank imposes an energy penalty when bitlines and wordlines dominate (2KB+ # of words/set).

Table 4: % of direct accesses with fast tags

	64K(256by/set)		1MB(512by/set)		2MB(1024 by/set)	
# Tags/set	2	4	4	8	8	16
Overhead	1KB	2KB	2KB	16KB	16KB	32KB
Benchmarks						
Low	30%	45%	42%	64%	55%	74%
Moderate	24%	62%	46%	70%	63%	85%
High	35%	79%	67%	95%	75%	96%

Amoeba-Cache can be tuned to minimize the hardware overhead for large caches. With many words/set the cache utilization improves due to longer block lifetimes making it feasible to support *Amoeba-Block*s with a larger minimum granularity ($>$ 1 word). If we increase minimum granularity to two or four words, only every third or fifth word could be a tag, meaning the # of comparators and multiplexers reduce to $\frac{N_{words/set}}{3}$ or $\frac{N_{words/set}}{5}$. When the minimum granularity is equal to max granularity (RMAX), we obtain a fixed granularity cache with $N_{words/set}/RMAX$ ways. Cache organizations that collocate all the tags together at the head of the data array enable tag-only operations and serial *Amoeba-Block* accesses that need to activate only a portion of the data array. However, the set may need to be compacted at each insertion. Recently, Loh and Hill [23] explored such an organization for supporting tags in multi-gigabyte caches.

Finally, the use of *Fast Tags* help reduce the tag lookups in the data array. Fast tags use a separate traditional tag array-like structure to cache the tags of the recently-used blocks and provide a pointer directly to the *Amoeba-Block*. The # of *Fast Tags* needed per set is proportional to the # of blocks in each set, which varies with the spatial locality in the application and the # of bytes per set (more details in Section 6.1). We studied 3 different cache configurations (64K 256B/set, 1M 512B/set, and 2M 1024B/set) while varying the number of fast tags per set (see Table 4). With 8 tags/set (16KB

overhead), we can filter 64—95% of the accesses in a 1MB cache and 55— 75% of the accesses in a 2MB cache.

5 Chip-Level Issues

5.1 Spatial Patterns Prediction

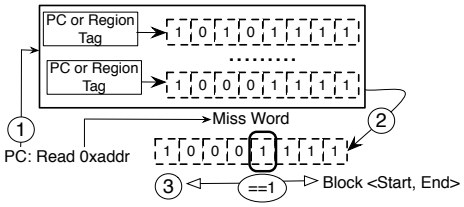


Figure 9: Spatial Predictor invoked on a *Amoeba-Cache* miss

Amoeba-Cache needs a spatial block predictor, which informs refill requests about the range of the block to fetch. *Amoeba-Cache* can exploit any spatial locality predictor and there have been many efforts in the compiler and architecture community [7, 17, 29, 6]. We adopt a table-driven approach consisting of a set of access bitmaps; each entry is `RMAX` (maximum granularity of an *Amoeba-Block*) bits wide and represents whether the word was touched during the lifetime of the recently evicted cache block. On a miss, the predictor will search for an entry (indexed by either the miss PC or region address) and choose the range of words to be fetched on a miss on either side (left and right) of the critical word. The PC-based indexing also uses the critical word index for improved accuracy. The predictor optimizes for spatial prefetching and will overfetch (bring in potentially untouched words), if they are interspersed amongst contiguous chunks of touched words. We can also bypass the prediction when there is low confidence in the prediction accuracy. For example, for streaming applications without repeated misses to a region, we can bring in a fixed granularity block based on the overall global behavior of the application. We evaluate tradeoffs in the design of the spatial predictor in Section 6.2.

5.2 Multi-level Caches

We discuss the design of inclusive cache hierarchies including multiple *Amoeba-Cache*s; we illustrate using a 2-level hierarchy. Inclusion means that the L2 cache contains a superset of the data words in the L1 cache; however, the two levels may include different granularity blocks. For example, the Sun Niagara T2 uses 16 byte L1 blocks and 64 byte L2 blocks. *Amoeba-Cache* permits non-aligned blocks of variable granularity at the L1 and the L2, and needs to deal with two issues: a) L2 recalls that may invalidate multiple L1 blocks and b) L1 refills that may need data from multiple blocks at the L2. For both cases, we need to identify all the relevant *Amoeba-Block*s that overlap with either the recall or the refill request. This situation is similar to a Nigara's L2 eviction which may need to recall 4 L1 blocks. *Amoeba-Cache*'s logic ensures that all *Amoeba-Block*s from a region map to a single set at any level (using the same RMAX for both L1 and L2). This ensures that L2 recalls or L1 refills index into only a single set. To process multiple blocks for a single cache operation, we use the step-by-step process outlined in Section 3.5 (❶ and ❷ in Figure 6). Finally, the L1-L2 interconnect needs 3 virtual networks, two of which, the L2→L1 data virtual network and the L1→L2 writeback virtual network, can have packets of variable granularity; each packet is broken down into a variable number of smaller physical flits.

5.3 Cache Coherence

There are three main challenges that variable cache line granularity introduces when interacting with the coherence protocol: 1) How is the coherence directory maintained? 2) How to support variable granularity read sharing? and 3) What is the granularity of write invalidations? The key insight that ensures compatibility with a conventional fixed-granularity coherence protocol is that a *Amoeba-Block* always lies within an aligned RMAX byte region (see § 3). To ensure correctness, it is sufficient to maintain the coherence granularity and directory information at a fixed granularity $\leq$ RMAX granularity. Multiple cores can simultaneously cache any variable granularity *Amoeba-Block* from the same region in *S*hared state; all such cores are marked as sharers in the directory entry. A core that desires exclusive ownership of an *Amoeba-Block* in the region uses the directory entry to invalidate every *Amoeba-Block* corresponding to the fixed coherence granularity. All *Amoeba-Block*s relevant to an invalidation will be found in the same set in the private cache (see set indexing in § 3). The coherence granularity could potentially be $<$ RMAX so that false sharing is not introduced in the quest for higher cache utilization (larger RMAX). The core claiming the ownership on a write will itself fetch only the desired granularity *Amoeba-Block*, saving bandwidth. A detailed evaluation of the coherence protocol is beyond the scope of the current paper.

6 Evaluation

Framework We evaluate the *Amoeba-Cache* architecture with the Wisconsin GEMS simulation infrastructure [25]; we use the in-order processor timing model. We have replaced the SIMICS functional simulator with the faster Pin [24] instrumentation framework to enable longer simulation runs. We perform timing simulation for 1 billion instructions. We warm up the cache using 20 million accesses from the trace. We model the cache controller in detail including the transient states needed for the multi-step cache operations and all the associated port and queue contention. We use a Full— LRU replacement policy, evicting *Amoeba-Block*s in LRU order until sufficient space is freed up for the block to be brought in. This helps decouple our observations from the replacement policy, enabling a fairer comparison with other approaches (Section 9). Our workloads are a mix of applications whose working sets stress our caches and includes SPEC- CPU benchmarks, Dacapo Java benchmarks [4], commercial workloads (SpecJBB2005, TPC-C, and Apache), and the Firefox web browser. Table 1 classifies the application categories: Low, Moderate, and High, based on the spatial locality. When presenting averages of ratios or improvements, we use the geometric mean.

6.1 Improved Memory Hierarchy Efficiency

Result 1:*Amoeba-Cache increases cache capacity by harvesting space from unused words and can achieve an 18% reduction in both L1 and L2 miss rate.*

Result 2:*Amoeba-Cache adaptively sizes the cache block granularity and reduces L1↔L2 bandwidth by 46% and L2↔Memory bandwidth by 38%.*

In this section, we compare the bandwidth and miss rate properties of *Amoeba-Cache* against a conventional cache. We evaluate two types of caches: a **Fixed** cache, which represents a conventional set-associative cache, and the *Amoeba-Cache*. In order to isolate the benefits of *Amoeba-Cache* from the potentially changing accuracy of the spatial predictor across different cache geometries, we use utilization at the next eviction as the spatial prediction, determined from a prior run on a fixed granularity cache. This also ensures that the spatial granularity predictions can be replayed across multiple simulation runs. To ensure equivalent data storage space, we set the *Amoeba-Cache* size to the sum of the tag array and the data array in a conventional cache. At the L1 level (64K), the net capacity of

the *Amoeba-Cache* is 64K + 8*4*256 bytes and at the L2 level (1M) configuration, it is 1M + 8*8*2048 bytes. The L1 cache has 256 sets and the L2 cache has 2048 sets.

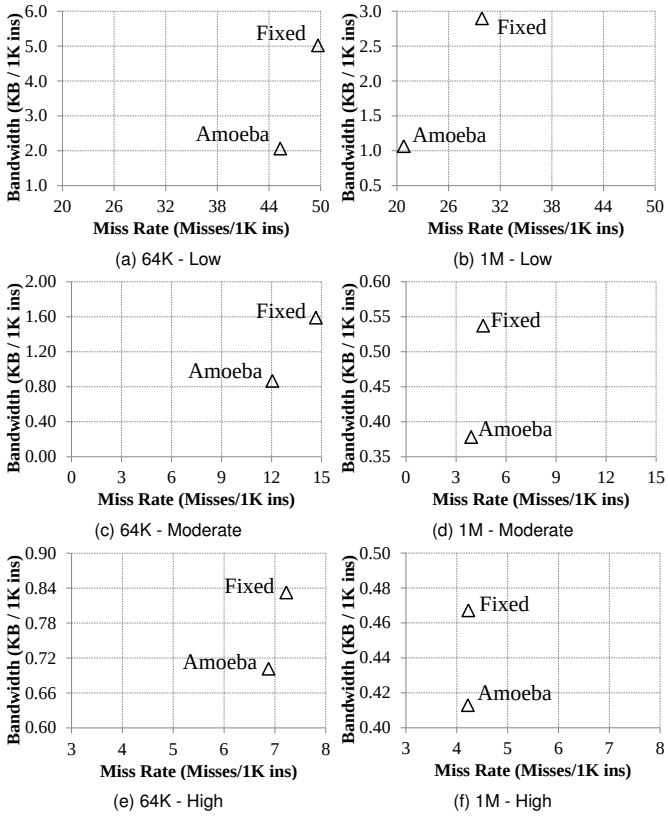


Figure 10: Fixed vs. Amoeba (Bandwidth and Miss Rate). Note the different scale for different application groups.

Figure 10 plots the miss rate and the traffic characteristics of the *Amoeba-Cache*. Since *Amoeba-Cache* can hold blocks varying from 8B to 64B, each set can hold more blocks by utilizing the space from untouched words. *Amoeba-Cache* reduces the 64K L1 miss rate by 23%(stdev:24) for the Low group, and by 21%(stdev:16) for the moderate group; even applications with high spatial locality experience a 7%(stdev:8) improvement in miss rate. There is a 46%(stdev:20) reduction on average in L1↔L2 bandwidth. At the 1M L2 level, *Amoeba-Cache* improves the moderate group's miss rate by 8%(stdev:10) and bandwidth by 23%(stdev:12). Applications with moderate utilization make better use of the space harvested from unused words by *Amoeba-Cache*. Many low utilization applications tend to be streaming and providing extra cache space does not help lower miss rate. However, by not fetching unused words, *Amoeba-Cache* achieves a significant reduction (38%(stdev:24) on average) in off-chip L2↔Memory bandwidth; even High utilization applications see a 17%(stdev:15) reduction in bandwidth. Utilization and miss rate are not, however, always directly correlated (more details in § 8).

With *Amoeba-Cache* the # of blocks/set varies based on the granularity of the blocks being fetched, which in turn depends on the spatial locality in the application. Table 5 shows the avg.# of blocks/set. In applications with low spatial locality, *Amoeba-Cache* adjusts the block size and adapts to store many smaller blocks. The 64K L1

Table 5: Avg. # of *Amoeba-Block* / Set

# Blocks/Set	64K Cache, 288 B/set
4—5	ferret, cactus, firefox, eclipse, facesim, freqmine, milc, astar
6—7	tpc-c, tradesoap, soplex, apache, fluidanimate
8—9	h2, canneal, omnetpp, twolf, x264, lbm, jbb
10—12	mcf, art
	1M Cache, 576 B/set
3—5	eclipse, omnetpp
8—9	cactus, firefox, tradesoap, freqmine, h2, x264, tpc-c
10—11	facesim, soplex, astar, milc, apache, ferret
12—13	twolf, art, jbb, lbm , fluidanimate
15—18	canneal, mcf

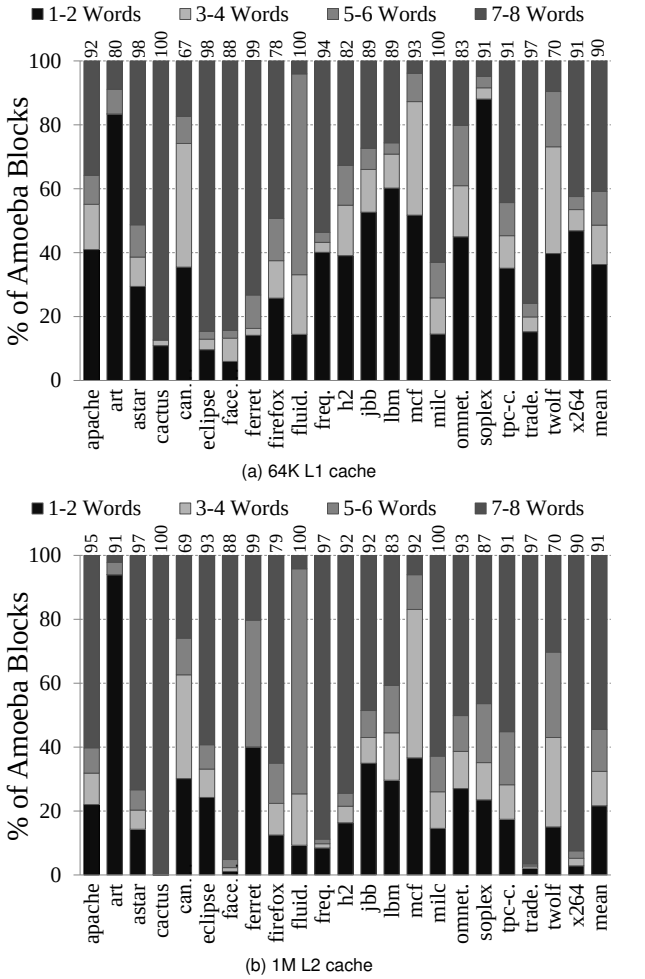


Figure 11: Distribution of cache line granularities in the 64K L1 and 1M L2 Amoeba-Cache. Avg. utilization is on top.

Amoeba-Cache stores 10 blocks per set for mcf and 12 blocks per set for art, effectively increasing associativity without introducing fixed hardware overheads. At the L2, when the working set starts to fit in the L2 cache, the set is partitioned into fewer blocks. Note that applications like eclipse and omnetpp hold only 3—5 blocks per set on average (lower than conventional associativity) due to their low miss rates (see Table 8). With streaming applications (e.g., canneal), *Amoeba-Cache* increases the # of blocks/set to >15 on average. Finally, some applications like apache store between 6—7 blocks/set with a 64K cache with varied block sizes (see Figure 11): approximately 50% of the blocks store 1-2 words and 30% of the blocks store 8 words at the L1. As the size of the cache increases and thereby the lifetime of the blocks, the *Amoeba-Cache* adapts to store larger size blocks as can be seen in Figure 11.

Utilization is improved greatly across all applications (90%+ in many cases). Figure 11 shows the distribution of cache block granularities in *Amoeba-Cache*. The *Amoeba-Block* distribution matches the word access distribution presented in § 2). With the 1M cache, the larger cache size improves block lifespan and thereby utilization, with a significant drop in the % of 1—2 word blocks. However, in many applications (tpc-c, apache, firefox, twolf, lbm, mcf), up to 20% of the blocks are 3–6 words wide, indicating the benefits of adaptivity and the challenges faced by Fixed.

6.2 Overall Performance and Energy

Result 3:*Amoeba-Cache improves overall cache efficiency and boosts performance by 10% on commercial applications*[2]*, saving up to 11% of the energy of the on-chip memory hierarchy. Off-chip L2↔memory energy sees a mean reduction of 41% across all workloads (86% for art and 93% for twolf)* .

We model a two-level cache hierarchy in which the L1 is a 64K cache with 256 sets (3 cycles load-to-use) and the L2 is 1M, 8192 sets (20 cycles). We assume a fixed memory latency of 300 cycles. We conservatively assume that the L1 access is the critical pipeline stage and throttle CPU clock by 4% (we evaluate an alternative approach in the next section). We calculate the total dynamic energy of the *Amoeba-Cache* using the energy #s determined in Section 4 through a combination of synthesis and CACTI [28]. We use 4 fast tags per set at the L1 and 8 fast tags per set at the L2. We include the penalty for all the extra metadata in *Amoeba-Cache*. We derive the energy for a single L1—L2 transfer ((6.8pJ per byte) from [2,28].The interconnect uses full-swing wires at 32nm, 0.6V.

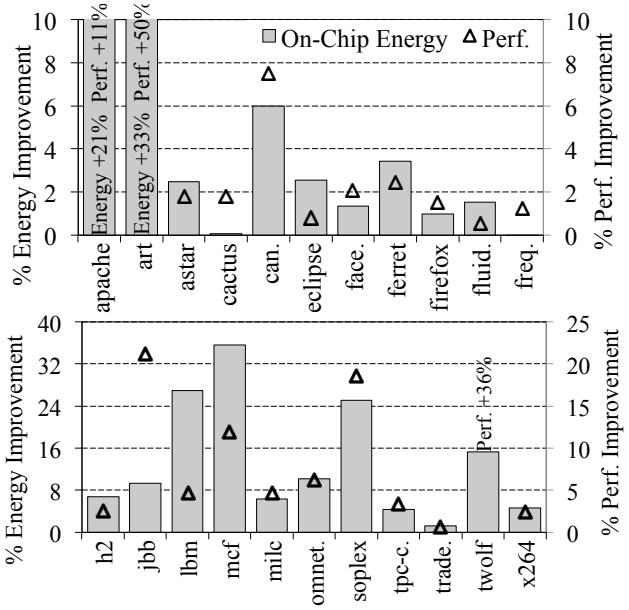


Figure 12: % improvement in performance and % reduction in on-chip memory hierarchy energy. Higher is better. Y-axis terminated to illustrate bars clearly. Baseline: Fixed, 64K L1, 1M L2.

Figure 12 plots the overall improvement in performance and reduction in on-chip memory hierarchy energy (L1 and L2 caches, and L1↔L2 interconnect). On applications that have good spatial locality (e.g., tradesoap, milc, facesim, eclipse, and cactus), *Amoeba-Cache* has minimal impact on miss rate, but provides significant bandwidth benefit. This results in on-chip energy reduction: milc's L1↔L2 bandwidth reduces by 15% and its on-chip energy reduces by 5%. Applications that suffer from cache pollution under Fixed (apache, jbb, twolf, soplex, and art) see gains in performance and energy. Apache's performance improves by 11% and on-chip energy reduces by 21%, while SpecJBB's performance improves by 21% and energy reduces by 9%. Art gains approximately 50% in performance. Streaming applications like mcf access blocks with both low and high utilization. Keeping out the unused words in the under-utilized blocks prevents the well-utilized cache blocks from being evicted; mcf's performance improves by 12% and on-chip energy by 36%.

Extra cache pipeline stage. An alternative strategy to accommodate *Amoeba-Cache*'s overheads is to add an extra pipeline stage to the cache access which increases hit latency by 1 cycle. The cpu clock frequency entails no extra penalty compared to a conventional cache. We find that for applications in the moderate and low spatial locality group (for 8 applications) *Amoeba-Cache* continues to provide a performance benefit between 6—50%. milc and canneal suffer minimal impact, with a 0.4% improvement and 3% slowdown respectively. Applications in the high spatial locality group (12 applications) suffer an average 15% slowdown (maximum 22%) due to the increase in L1 access latency. In these applications, 43% of the instructions (on average) are memory accesses and a 33% increase in L1 hit latency imposes a high penalty. Note that all applications continue to retain the energy benefit. The cache hierarchy energy is dominated by the interconnects and *Amoeba-Cache* provides notable bandwidth reduction. While these results may change for an out-of-order, multiple-issue processor, the evaluation suggests that *Amoeba-Cache* if implemented with the extra pipeline stage is more suited for lower levels in the memory hierarchy other than the L1.

Off-chip L2↔Memory energy The L2's higher cache capacity makes it less susceptible to pollution and provides less opportunity for improving miss rate. In such cases, *Amoeba-Cache* keeps out unused words and reduces off-chip bandwidth and thereby off-chip energy. We assume that the off-chip DRAM can provide adaptive granularity transfers for *Amoeba-Cache*'s L2 refills as in [35]. We use a DRAM model presented in a recent study [9] and model 0.5nJ per word transferred off-chip. The low spatial locality applications see a dramatic reduction in off-chip energy. For example, twolf sees a 93% reduction. On commercial workloads the off-chip energy decreases by 31% and 24% respectively. Even for applications with high cache utilization, off-chip energy decreases by 15%.

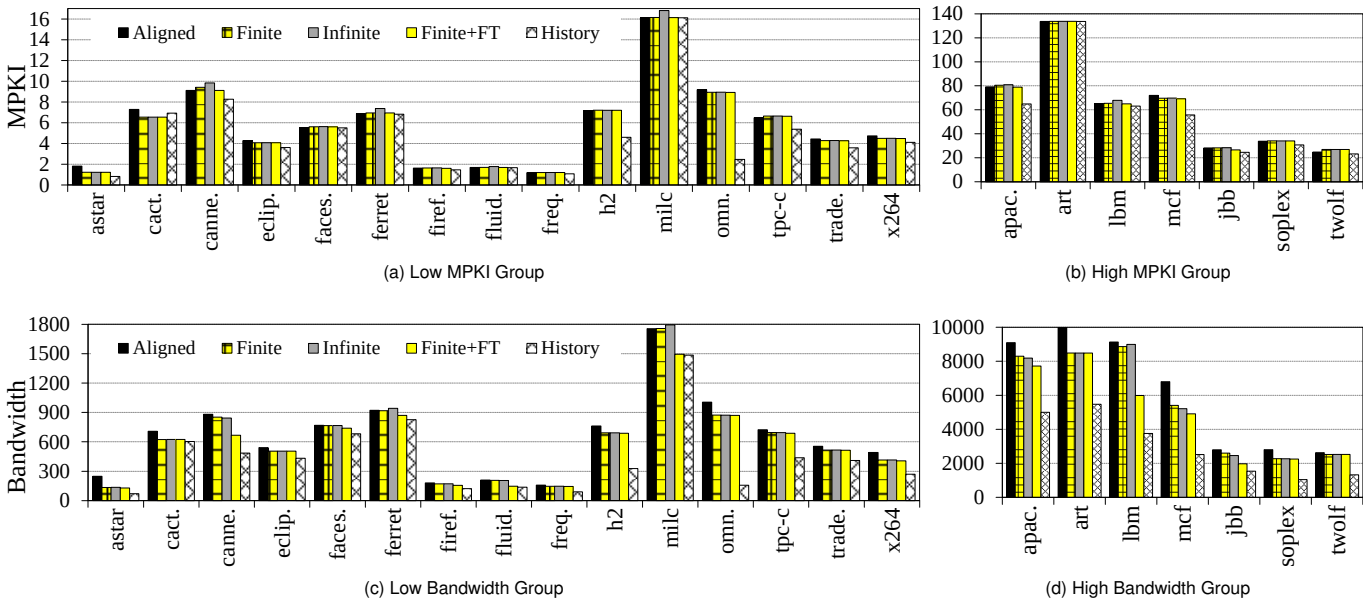
7 Spatial Predictor Tradeoffs

We evaluate the effectiveness of spatial pattern prediction in this section. In our table-based approach, a pattern history table records spatial patterns from evicted blocks and is accessed using a prediction index. The table-driven approach requires careful consideration of the following: prediction index, predictor table size, and training period. We quantify the effects by comparing the predictor against a baseline fixed-granularity cache. We use a 64K cache since it induces enough misses and evictions to highlight the predictor tradeoffs clearly.

7.1 Predictor Indexing

A critical choice with the history-based prediction is the selection of the predictor index. We explored two types of predictor indexing a) a PC-based approach [6] based on the intuition that fields in a data structure are accessed by specific PCs and tend to exhibit similar spatial behavior. The tag includes the the PC and the critical word index:

[2]"Commercial" applications includes Apache, SpecJBB and TPC-C.



ALIGNED: fixed-granularity cache (64B blocks). FINITE: *Amoeba-Cache* with a REGION predictor (1024 entry predictor table and 4K region size). INFINITE: *Amoeba-Cache* with an unbounded predictor table (REGION predictor).FINITE+FT is FINITE augmented with hints for predicting a default granularity on compulsory misses (first touches). HISTORY: *Amoeba-Cache* uses spatial pattern hints based on utilization at the next eviction, collected from a prior run.

$((PC >> 3) << 3)+\frac{(addr\%64)}{8}$. and b) a Region-based (REGION) approach that is based on the intuition that similar data objects tend to be allocated in contiguous regions of the address space and tend to exhibit similar spatial behavior. We compared the miss rate and bandwidth properties of both the PC (256 entries, fully associative) and REGION (1024 entries, 4KB region size) predictors. The size of the predictors was selected as the sweet spot in behavior for each predictor type. For all applications apart from cactus (a high spatial locality application), REGION-based prediction tends to overfetch and waste bandwidth as compared to PC-based prediction, which has 27% less bandwidth consumption on average across all applications. For 17 out of 22 applications, REGION-based prediction shows 17% better MPKI on average (max: 49% for cactus). For 5 applications (apache, art, mcf, lbm, and omnetpp), we find that PC demonstrates better accuracy when predicting the spatial behavior of cache blocks than REGION and demonstates a 24% improvement in MPKI (max: 68% for omnetpp).

7.2 Predictor Table

We studied the organization and size of the pattern table using the REGION predictor. We evaluated the following parameters a) region size, which directly correlates with the coverage of a fixed-size table, and b) the size of the predictor table, which influences how many unique region patterns can be tracked, and c) the # of bits required to represent the spatial pattern.

Large region sizes effectively reduce the # of regions in the working set and require a smaller predictor table. However, a larger region is likely to have more blocks that exhibit varied spatial behavior and may pollute the pattern entry. We find that going from 1KB (4096 entries) to 4KB (1024 entries) regions, the 4KB region granularity decreased miss rate by 0.3% and increased bandwidth by 0.4% even though both tables provide the same working set coverage (4MB). Fixing the region size at 4KB, we studied the benefits of an unbounded table. Compared to a 1024 entry table (FINITE in Figure 6.2), the unbounded table increases miss rate by 1% and decreases bandwidth by 0.3% . A 1024 entry predictor table (4KB region granularity per-entry) suffices for most applications. Organizing the 1024 entries as a 128-set×8-way table suffices for eliminating associativity related conflicts (<0.8% evictions due to lack of ways).

Focusing on the # of bits required to represent the pattern table, we evaluated the use of 4-bit saturation counters (instead of 1-bit bitmaps). The saturation counters seek to avoid pattern pollution when blocks with varied spatial behavior reside in the same region. Interestingly, we find that it is more beneficial to use 1-bit bitmaps for the majority of the applications (12 out of 22); the hysteresis introduced by the counters increases training period. To summarize, we find that a REGION predictor with region size 4KB and 1024 entries can predict the spatial pattern in a majority of the applications. CACTI indicates that the predictor table can be indexed in 0.025ns and requires 2.3pJ per miss indexing.

7.3 Spatial Pattern Training

A widely-used approach to training the predictor is to harvest the word usage information on an eviction. Unfortunately, evictions may not be frequent, which means the predictor's training period tends to be long, during which the cache performs less efficiently and/or that the application's phase has changed in the meantime. Particularly at the time of first touch (compulsory miss to a location), we need to infer the global spatial access patterns. We compare the finite region predictor (FINITE in Figure 6.2) that only predicts using eviction history, against a FINITE+FT: this adds the optimization of inferring the default pattern (in this paper, from a prior run) when there is no predictor information. FINITE+FT demonstrates an avg. 1% (max: 6% for jbb) reduction in miss rate compared to FINITE and comes within 16% the miss rate of HISTORY. In terms of bandwidth FINITE+FT can save 8% of the bandwidth (up

to 32% for lbm) compared to FINITE. The percentage of first-touch accesses is shown in Table 8.

7.4 Predictor Summary

- For the majority of the applications (17/22) the address-region predictor with region size 4KB works well. However, five applications (apache, lbm, mcf, art, omnetpp) show better performance with PC-based indexing. For best efficiency, the predictor should adapt indexing to each application.
- Updating the predictor only on evictions leads to long training periods, which causes loss in caching efficiency. We need to develop mechanisms to infer the default pattern based on global behavior demonstrated by resident cache lines.
- The online predictor reduces MPKI by 7% and bandwidth by 26% on average relative to the conventional approach. However, it still has a 14% higher MPKI and 38% higher bandwidth relative to the HISTORY-based predictor, indicating room for improvement in prediction.
- The 1024-entry (4K region size) predictor table imposes $\simeq$0.12% energy overhead on the overall cache hierarchy energy since it is referenced only on misses.

8 *Amoeba-Cache* Adaptivity

We demonstrate that *Amoeba-Cache* can adapt better than a conventional cache to the variations in spatial locality.

Tuning RMAX for High Spatial Locality A challenge often faced by conventional caches is the desire to widen the cache block (to achieve spatial prefetching) without wasting space and bandwidth in low spatial locality applications. We study 3 specific applications: milc and tradesoap have good spatial locality and soplex has poor spatial locality. With a conventional 1M cache, when we widen the block size from 64 to 128 bytes, milc and tradesoap experience a 37% and 39% reduction in miss rate. However, soplex's miss rate increases by 2$\times$ and bandwidth by 3.1$\times$.

With *Amoeba-Cache* we do not have to make this uneasy choice as it permits *Amoeba-Block*s with granularity 1—RMAX words (RMAX: maximum block size). When we increase RMAX from 64 bytes to 128 bytes, miss rate reduces by 37% for milc and 24% for tradesoap, while simultaneously lowering bandwidth by 7%. Unlike the conventional cache, *Amoeba-Cache* is able to adapt to poor spatial locality: soplex experiences only a 2% increase in bandwidth and 40% increase in miss rate.

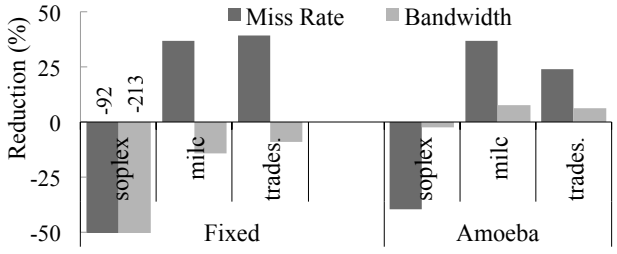


Figure 13: Effect of increase in block size from 64 to 128 bytes in a 1 MB cache

Predicting Strided Accesses Many applications (e.g., firefox and canneal) exhibit strided access patterns, which touch a few words in a block before accessing another block. Strided accesses patterns introduce intra-block holes (untouched words). For instance, canneal accesses $\simeq$10K distinct fixed granularity cache blocks with a specific access pattern, **[--x--x--]** (**x** indicates i^{th} word has been touched). Any predictor that uses the access pattern history has two choices when an access misses on word 3 or 6 a) A miss oriented policy (Policy-Miss) may refill the entire range of words 3–6 and eliminate the secondary miss but bring in untouched words 4–5, increasing bandwidth, and b) a bandwidth focused choice (Policy- BW) that refills only the requested word but will incur more misses. Table 6 compares the two contrasting policies for *Amoeba-Cache* (relative to a Fixed granularity baseline cache). Policy-BW saves 9% bandwidth compared to Policy-Miss but suffer 25-30% higher miss rate.

Table 6: Predictor Policy Comparison

	canneal		firefox	
	Miss Rate	BW	Miss Rate	BW
Policy-Miss	10.31%	81.2%	11.18%	47.1%
Policy-BW	–20.08%	88.09%	–13.44%	56.82%
Spatial Patterns	**[--x--x--] [x--x----]**		**[-x--x---] [x---x---]**	

–: indicates Miss or BW higher than Fixed.

9 *Amoeba-Cache* vs other approaches

We compare *Amoeba-Cache* against four approaches.

- **Fixed-2x**: Our baseline is a fixed granularity cache 2$\times$ the capacity of the other designs (64B block).
- **Sector** is the conventional sector cache design (as in IBM Power7 [15]): 64B block and a small sector size (16bytes or 2 words). This design targets optimal bandwidth. On any access, the cache fetches only the requisite sector, even though it allocates space for the entire line.
- **Sector-Pre** adds prefetching to Sector. This optimized design prefetches multiple sectors based on a spatial granularity predictor to improve the miss rate [17, 29].
- **Multi$** combines features from line distillation [30] and spatial-temporal caches [12]. It is an aggressive design that partitions a cache into two: a line organized cache (LOC) and a word-organized cache (WOC). At insertion time, Multi$ uses the spatial granularity hints to direct the cache to either refill words (into the WOC) or the entire line. We investigate two design points: 50% of the cache as a WOC (Multi$-50) and 25% of cache as a WOC (Multi$-25).

Sector, Sector-Pre, Multi$, and *Amoeba-Cache* all use the same spatial predictor hints. On a demand miss, they prefetch all the sub-blocks needed together. Prefetching only changes the timing of an access; it does not change the miss bandwidth and cannot remove the misses caused by cache pollution.

Energy and Storage The sector approaches impose low hardware complexity and energy penalty. To filter out unused words, the sector granularity has to be close to word granularity; we explored 2words/sector which leads to a storage penalty of $\simeq$64KB for a 1MB cache. In Multi$, the WOC increases associativity to the number of words/block. Multi$-25 partitions a 64K 4-way cache into a 48K 3-way LOC and a 16K 8-way WOC, increasing associativity to 11. For a 1M cache, Multi$-50 increases associativity to 36. Compared to Fixed, Multi$-50 imposes over 3$\times$ increase in lookup latency, 5$\times$ increase in lookup energy, and $\simeq$4$\times$ increase in tag storage. *Amoeba-Cache* provides a more scalable approach to using adaptive cache lines since it varies the storage dedicated to the tags based on the spatial locality in the application.

Miss Rate and Bandwidth Figure 14 summarizes the comparison; we focus on the moderate and low utilization groups of applica-

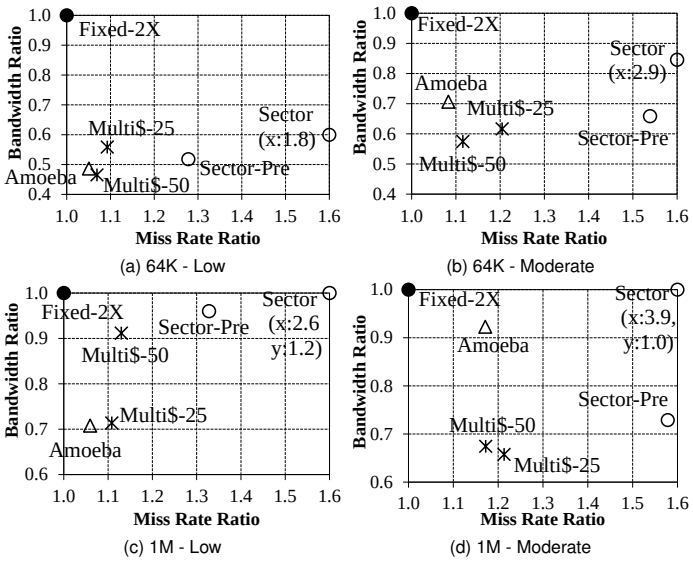


Figure 14: Relative miss rate and bandwidth for different caches. Baseline (1,1) is the Fixed-2x design. Labels: • Fixed-2x, ○ Sector approaches. ∗ : Multi$, △ Amoeba. (a),(b) 64K cache (c),(d) 1M cache. Note the different Y-axis scale for each group.

tions. On the high utilization group, all designs other than Sector have comparable miss rates. *Amoeba-Cache* improves miss rate to within 5%—6% of the Fixed-2× for the low group and within 8%—17% for the moderate group. Compared to the Fixed-2×, *Amoeba-Cache* also lowers bandwidth by 40% (64K cache) and 20% (1M cache). Compared to Sector-Pre (with prefetching), *Amoeba-Cache* is able to adapt better with flexible granularity and achieves lower miss rate (up to 30% @ 64K and 35% @ 1M). Multi$'s benefits are proportional to the fraction of the cache organized as a WOC; Multi$-50 (18-way@64K and 36-way@1M) is needed to match the miss rate of *Amoeba-Cache*. Finally, in the moderate group, many applications exhibit strided access. Compared to Multi-$'s WOC, which fetches individual words, *Amoeba-Cache* increases bandwidth since it chooses to fetch the contiguous chunk in order to lower miss rate.

10 Multicore Shared Cache

We evaluate a shared cache implemented with the *Amoeba-Cache* design. By dynamically varying the cache block size and keeping out unused words, the *Amoeba-Cache* effectively minimizes the footprint of an application. Minimizing the footprint helps multiple applications effectively share the cache space. We experimented with a 1M shared *Amoeba-Cache* in a 4 core system. Table 7 shows the application mixes; we chose a mix of applications across all groups. We tabulate the change in miss rate per thread and the overall change in bandwidth for *Amoeba-Cache* with respect to a fixed granularity cache running the same mix. Minimizing the overall footprint enables a reduction in the miss rate of each application in the mix. The commercial workloads (SpecJBB and TPC-C) are able to make use of the space available and achieve a significant reduction in miss rate (avg: 18%). Only two applications suffered a small increase in miss rate (x264 Mix#2: 2% and ferret Mix#3: 4%) due to contention. The overall L2 miss bandwidth significantly improves, showing 16%—39% reduction across all workload mixes. We believe that the Amoeba-based shared cache can effectively enable the shared cache to support more cores and increase overall throughput. We leave the design space exploration of an Amoeba-based coherent cache for future work.

Table 7: Multiprogrammed Workloads on 1M Shared *Amoeba-Cache*% reduction in miss rate and bandwidth. Baseline: Fixed 1M.

	Miss	Miss	Miss	Miss	BW
Mix	T1	T2	T3	T4	(All)
jbb×2, tpc-c×2	12.38%	12.38%	22.29%	22.37%	39.07%
firefox×2, x264×2	3.82%	3.61%	–2.44%	0.43%	15.71%
cactus, fluid., omnet., sopl.	1.01%	1.86%	22.38%	0.59%	18.62%
canneal, astar, ferret, milc	4.85%	2.75%	19.39%	–4.07%	17.77%

–: indicates Miss or BW higher than Fixed. T1—T4, threads in the mix; in the order of applications in the mix

11 Related Work

Burger et al. [5] defined cache efficiency as the fraction of blocks that store data that is likely to be used. We use the term cache utilization to identify touched versus untouched words residing in the cache. Past works [6, 29, 30] have also observed low cache utilization at specific levels of the cache. Some works [18, 19, 13, 21] have sought to improve cache utilization by eliminating cache blocks that are no longer likely to be used (referred to as dead blocks). These techniques do not address the problem of intra-block waste (i.e., untouched words).

Sector caches [31, 32] associate a single tag with a group of contiguous cache lines, allowing cache sizes to grow without paying the penalty of additional tag overhead. Sector caches use bandwidth efficiently by transferring only the needed cache lines within a sector. Conventional sector caches [31] may result in worse utilization due to the space occupied by invalid cache lines within a sector. Decoupled sector caches [32] help reduce the number of invalid cache lines per sector by increasing the number of tags per sector. Compared to the Amoeba cache, the tag space is a constant overhead, and limits the # of invalid sectors that can be eliminated. Pujara et al. [29] consider a word granularity sector cache, and use a predictor to try and bring in only the used words. Our results (see Figure 14) show that smaller granularity sectors significantly increase misses, and optimizations that prefetch [29] can pollute the cache and interconnect with unused words.

Line distillation [30] applies filtering at the word granularity to eliminate unused words in a cache block at eviction. This approach requires part of the cache to be organized as a word-organized cache, which increases tag overhead, complicates lookup, and bounds performance improvements. Most importantly, line distillation does not address the bandwidth penalty of unused words. This inefficiency is increasingly important to address under current and future technology dominated by interconnects [14, 28]. Veidenbaum et al. [33] propose that the entire cache be word organized and propose an online algorithm to prefetch words. Unfortunately, a static word-organized cache has a built-in tag overhead of 50% and requires energy-intensive associative searches.

Amoeba-Cache adopts a more proactive approach that enables continuous dynamic block granularity adaptation to the available spatial locality. When there is high spatial locality, the *Amoeba-Cache* will automatically store a few big cache blocks (most space dedicated for data); with low spatial locality, it will adapt to storing many small cache blocks (extra space allocated for tags). Recently, Yoon et

al. have proposed an adaptive granularity DRAM architecture [35]. This provides the support necessary for supporting variable granularity off-chip requests from an *Amoeba-Cache*-based LLC. Some research [10,8] has also focused on reducing false sharing in coherent caches by splitting/merging cache blocks to avoid invalidations. They would benefit from the *Amoeba-Cache* design, which manages block granularity in hardware.

There has a been a significant amount of work at the compiler and software runtime level (e.g. [7]) to restructure data for improved spatial efficiency. There have also been efforts from the architecture community to predict spatial locality [29, 34, 17, 36], which we can leverage to predict *Amoeba-Block* ranges. Finally, cache compression is an orthogonal body of work that does not eliminate unused words but seeks to minimize the overall memory footprint [1].

12 Summary

In this paper, we propose a cache design, Amoeba-Cache, that can dynamically hold a variable number of cache blocks of different granularities. The Amoeba-Cache employs a novel organization that completely eliminates the tag array and collocates the tags with the cache block in the data array. This permits the Amoeba-Cache to trade the space budgeted for the cache blocks for tags and support a variable number of tags (and blocks). For applications that have low spatial locality, *Amoeba-Cache* can reduce cache pollution, improve the overall miss rate, and reduce bandwidth wasted in the interconnects. When applications have moderate to high spatial locality, *Amoeba-Cache* coarsens the block size and ensures good performance. Finally, for applications that are streaming (e.g., lbm), *Amoeba-Cache* can save significant energy by eliminating unused words from being transmitted over the interconnects.

Acknowledgments

We would like to thank our shepherd, André Seznec, and the anonymous reviewers for their comments and feedback.

Appendix

Table 8: *Amoeba-Cache* Performance. Absolute #s.

	MPKI		BW bytes/1K		CPI	Predictor Stats	
	L1 MPKI	L2 MPKI	L1⟷L2 #Bytes/1K	L2⟷Mem #Bytes/1K	Cycles/Ins.	First Touch % Misses	Evict Win. # ins./Evict
apache	64.9	19.6	5,000	2,067	8.3	0.4	17
art	133.7	53.0	5,475	1,425	16.0	0.0	9
astar	0.9	0.3	70	35	1.9	18.0	1,600
cactus	6.9	4.4	604	456	3.5	7.5	162
canne.	8.3	5.0	486	357	3.2	5.8	128
eclip.	3.6	<0.1	433	<1	1.8	0.1	198
faces.	5.5	4.7	683	632	3.0	41.2	190
ferre.	6.8	1.4	827	83	2.1	1.3	156
firef.	1.5	1.0	123	95	2.1	11.1	727
fluid.	1.7	1.4	138	127	1.9	39.2	629
freqm.	1.1	0.6	89	65	2.3	17.7	994
h2	4.6	0.4	328	46	1.8	1.7	154
jbb	24.6	9.6	1,542	830	5.0	10.2	42
lbm	63.1	42.2	3,755	3,438	13.6	6.7	18
mcf	55.8	40.7	2,519	2,073	13.2	0.0	19
milc	16.1	16.0	1,486	1,476	6.0	2.4	66
omnet.	2.5	<0.1	158	<1	1.9	0.0	458
sople.	30.7	4.0	1,045	292	3.1	0.9	35
tpcc	5.4	0.5	438	36	2.0	0.4	200
trade.	3.6	<0.1	410	6	1.8	0.6	194
twolf	23.3	0.6	1,326	45	2.2	0.0	49
x264	4.1	1.8	270	190	2.2	12.4	274

MPKI : Misses / 1K instructions. BW: # words / 1K instructions
CPI: Clock cycles per instruction.
Predictor First touch: Compulsory misses. % of accesses that use default granularity.
Evict window: # of instructions between evictions. Higher value indicates predictor training takes longer.

References

[1] A. R. Alameldeen. Using Compression to Improve Chip Multiprocessor Performance. In *Ph.D. dissertation, Univ. of Wisconsin-Madison*, 2006.

[2] D. Albonesi, A. Kodi, and V. Stojanovic. NSF Workshop on Emerging Technologies for Interconnects (WETI). 2012.

[3] C. Bienia. Benchmarking Modern Multiprocessors. In *Ph.D. Thesis. Princeton University*, 2011.

[4] S. M. Blackburn et al. The DaCapo benchmarks: java benchmarking development and analysis. In *Proc. of the 21st OOPSLA*, 2006.

[5] D. Burger, J. Goodman, and A. Kaigi. The Declining Effectiveness of Dynamic Caching for General-Purpose Workloads. In *University of Wisconsin Technical Report*, 1995.

[6] C. F. Chen et al. Accurate and Complexity-Effective Spatial Pattern Prediction. In *Proc. of the 10th HPCA*. 2004.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proc. of the PLDI*, 1999.

[8] B. Choi et al. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proc. of the PACT*, 2011.

[9] B. Dally. Power, Programmability, and Granularity: The Challenges of ExaScale Computing. In *Proc. of the IPDPS*, 2011.

[10] C. Dubnicki and T. J. Leblanc. Adjustable Block Size Coherent Caches. In *Proc. of the 19th ISCA*, 1992.

[11] A. Fog. Instruction tables. 2012. `http://www.agner.org/optimize/instruction_tables.pdf`.

[12] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. of the ACM Intl. Conf. on Supercomputing*, 1995.

[13] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proc. of the 29th ISCA*, 2002.

[14] C. Hughes, C. Kim, and Y.-K. Chen. Performance and Energy Implications of Many-Core Caches for Throughput Computing. In *IEEE Micro Journal*, 2010.

[15] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's Next-Generation Server Processor. In *IEEE Micro Journal*, 2010.

[16] K. Kedzierski, M. Moreto, F. J. Cazorla, and M. Valero. Adapting Cache Partitioning Algorithms to Pseudo-LRU Replacement Policies. In *Proc. of the IPDPS*, 2010.

[17] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proc. of the 25th ISCA*, 1998.

[18] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proc. of the 27th ISCA*, 2000.

[19] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Proc. of the 22nd ISCA*. 1995.

[20] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. of the 42nd MICRO*, 2009.

[21] H. Liu et al. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proc. of the 41st MICRO*, 2008.

[22] D. R. Llanos and B. Palop. TPCC-UVA: An Open-source TPC-C implementation for parallel and distributed systems. In *Proc. of the 2006 Intl. Parallel and Distributed Processing Symp. PMEO Workshop*, 2006. `http://www.infor.uva.es/~diego/tpcc-uva.html`.

[23] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proc. of the 44th Intl. Symp. on Microarchitecture*, 2011.

[24] 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. of the PLDI*, 2005.

[25] Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. In *ACM SIGARCH Computer Architecture News*, Sept. 2005.

[26] S. Microsystems. OpenSPARC T1 Processor Megacell Specification. 2007.

[27] Moinuddin K. Qureshi et al. Adaptive insertion policies for high performance caching In *Prof. of the 34th ISCA*, 2007.

[28] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proc. of the 40th MICRO*, 2007.

[29] P. Pujara and A. Aggarwal. Increasing the Cache Efficiency by Eliminating Noise. In *Proc. of the 12th HPCA*, 2006.

[30] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *Proc. of the 13th HPCA*, 2007.

[31] J. B. Rothman and A. J. Smith. The pool of subsectors cache design. In *Proc. of the 13th ACM ICS*, 1999.

[32] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *Proc. of the 21st ISCA*, 1994.

[33] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Proc. of the 13th ACM ICS*. 1999.

[34] M. A. Watkins, S. A. Mckee, and L. Schaelicke. Revisiting Cache Block Superloading. In *Proc. of the 4th HIPEAC*. 2009.

[35] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive granularity memory systems: a tradeoff between storage efficiency and throughput. In *Proc. of the 38th ISCA*. 2011.

[36] D. H. Yoon, M. K. Jeong, M. B. Sullivan, and M. Erez. The Dynamic Granularity Memory System. In *Proc. of the 39th ISCA*, 2012.

[36] `http://cpudb.stanford.edu/`