ARCHITECTURAL SUPPORT FOR A VARIABLE GRANULARITY CACHE MEMORY SYSTEM

by

Snehasish Kumar
B.Tech, Biju Patnaik University of Technology, 2010

a Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

APPROVAL

Name:	Snehasish Kumar
Degree:	Master of Science
Title of Thesis:	Architectural support for a variable granularity cache memory system
Examining Committee:	Dr. Hu Kaers Chair
	Dr. Arrvindh Shriraman, Senior Supervisor
	Dr. Alexandra Federova, Supervisor

Date Approved:

Abstract

Memory in modern computing systems are hierarchial in nature. Maintaining a memory hierarchy enables the system to service frequently requested data from a small low latency store located close to the processor. The design paradigms of the memory hierachy have been mostly unchanged since their inception in the late 1960's. However in the meantime there have been significant changes in the tasks computers perform and the way they are programmed. Modern computing systems perform more data centric tasks and are programmed in higher level languages which introduce many layers of abstraction between the programmer and the system.

Waste in the memory hierarchy refers to the under utilised space in the memory system and consequently wasted energy and time. The data access patterns of modern workloads are increasingly less uniform which makes it hard to design a memory hierarchy with rigid design principles that performs optimally for a wide range of workloads. The problem is exacerbated by the implications of the growing fraction of dark silicon on a processor chip.

This dissertation proposes and evaluates the benefits of a novel architecture for the on chip memory hierarchy which would allow it to dynamically adapt to the requirements of the application. We propose a design that can support a variable number of cache blocks, each of a different granularity. It 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). The design 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.

Compared to a fixed granularity cache, improves cache utilization to 90% - 99% for most applications, saves miss rate by up to 73% at the L1 level and up to 88% at the LLC level, and reduces miss bandwidth by up to 84% at the L1 and 92% at the LLC. Correspondingly reduces on-chip memory hierarchy energy by as much as 36% and improves performance by as much as 50%.

To whomever whoever reads this!

"Don't worry, Gromit. Everything's under control!"

— *The Wrong Trousers*, Aardman Animations, *1993*

Acknowledgments

Here go all the people you want to thank.

Contents

List of Tables

List of Figures

List of Programs

Preface

Here go all the interesting reasons why you decided to write this thesis.

Chapter 1

Introduction

Memory systems are an integral part of computer architecture whose overall design and organisation have remained unchanged since their inception. Early mainframe computers in the 1960's were known to use a hierarchical memory organisation. The memory technologies included semi-conductor, magnetic core, drum and disc. Initially, accessing memory was only slightly slower than register access however as the difference grew, the need to mitigate the delay incurred for a memory access became extremely important. The rate at which computations were performed kept increasing, however the rate at which data was fed to the processor from the memory system did not grow at the same rate. In order to alleviate the effects of slow memory, smaller faster memory was built close to the processor to cache frequently used data. Caching was used to fetch data and instructions into the fastest memory on CPU accesses to take advantage of faster lookups on reuse. The first documented use of a data cache was in the IBM System/360 Model 85 [16].

In modern systems, with multiple levels in their memory hierarchy, a couple of levels closest to the processor are made from static random access memory (SRAM) which increase in size as the distance from the processor increases. These are placed on the same die as the processor chip. To service a memory request not present in the previous levels, a request is sent off chip to fetch the data from main memory, constructed from dynamic random access memory (DRAM). The main memory is usually several magnitudes of order larger than the largest cache present on the chip. This can be afforded by the lower cost of DRAM compared to SRAM. However, DRAM lookups are slower and require more power to retain data. The speed, cost and size for each level in the memory hierarchy can be visualized as shown in Fig 1.1.

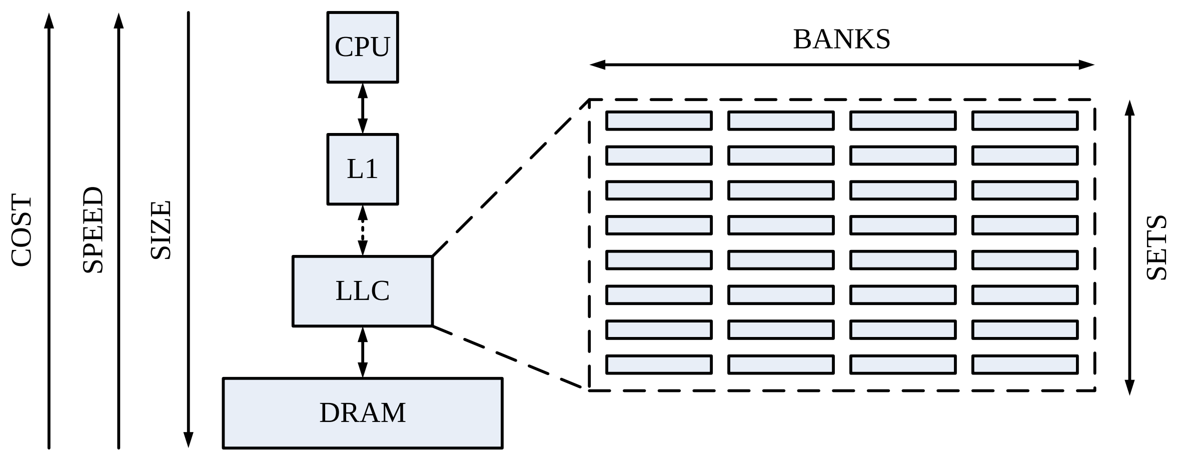


Figure 1.1: **Canonical Memory Hierarchy** – Moving down through the hierarchy, away from the processor, the levels are larger but slower. At each level the storage may be monolithic or sub divided in to "banks" for lower indexing overhead.

1.1 Cache Memory Systems

Most processors access data at the granularity of 4 to 8 bytes at a time. In order to exploit locality present in programs, caches are designed to retain small amounts of data close to the processor for fast access. The management of data in the cache is determined by a suitably selected replacement scheme. Caches are given designations to indicate their level in the memory hierarchy. The closest cached data stores to the processor are given lower numerical designations starting from *L1* and increases as their distance from the processor increases. The last level of cache is often abbreviated as the *LLC*. Each level in the cache hierarchy is linked in a daisy chain fashion, as shown in Fig 1.1, where there is an option for the data that is being cached to be replicated or not. The design choices can be enumerated as:

1. Inclusive Caches : Lower level caches (further from the processor) replicate the cache lines present (although the data may be stale) present in the higher level caches. Inclusive caches can be found in Intel Sandy Bridge processors.

2. Exclusive Caches : Caches lower in the hierarchy are guaranteed to not contain the cache lines present in the higher levels of the hierarchy. Present in the AMD architectures such as the Athlon processors.

3. Non-Inclusive Caches : Also known as *Non-Exclusive* or *Accidentally Inclusive*, were used for a while in Intel architectures prior to the Intel P6. A lower level cache may or may not include a block cached at a higher level in the cache hierarchy.

Caches are designed to take advantage of reuse of data by speeding up subsequent access to the same datum. They also speed up accesses to nearby data which may be fetched into the cache depending on its operating policy. The different types of locality which caches try to exploit can be enumerated as :

1. Temporal Locality : Some applications tend to reuse the same data items over and over again during the course of their execution. This principle is the cornerstone for caching. Cache management policies usually implemented take into account the recency of data reuse to take a decision on what data is to be retained in the cache. Modern cache hierarchies implement a form of the *Least Recently Used* algorithm to manage the contents of the cache.

2. Spatial Locality : Due to conventional imperative programming paradigms, data is usually managed by grouping datum together in data structures, the fields of which are accessed in close proximity in the source code. Thus in order to exploit this pattern when a datum is requested, it is normal behaviour for the cache to bring in a contiguous region, $32 - 128$ bytes in size, which contains the datum. The contiguous region of data brought into the cache is referred to as a *cache block* or *cache line.* The Intel Pentium 3 processors used a 32 byte line size which was increased to 64 bytes from Pentium 4. The IBM Power7 architectures use a 128 byte cache line where as the Intel Itanium2 uses a 64 bytes cache line size at the L1 and 128 byte cache line size at the L2 and L3.

According to the place where a new cache line can be inserted into the cache, the cache can have varying associativity. If the policy requires that a certain block from memory can map only to a specific entry in the cache, it is known as a *direct mapped* cache [Fig 1.2(a)]. On the other end of the spectrum, if a certain block from memory can map to any entry in the cache it is known as a *fully associative* cache [Fig 1.2(c)]. It is easy to see that fully associative caches are the most flexible, however they incur significant costs in terms of latency and area overhead for standard cache operations. A cache lookup for a specific block is analogous to checking each item in a collection for a possible match. Conventional caches are organised as a 2-dimensional data structure where the rows are called *sets*. Within each set there are a fixed number of cache blocks. The number of blocks in each set is the degree of associativity of the cache. Each possible entry in a set is called a *way*. Thus a direct mapped cache has associativity of 1 where as a fully associative cache is one whose associativity is equal to the total number of cache blocks that the given cache can possibly hold.

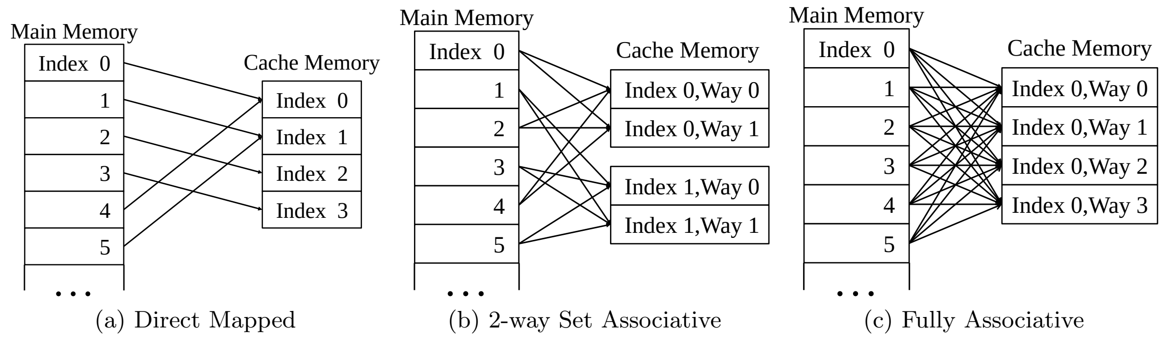


Figure 1.2: Each block in memory maps to (a) a single entry (way) in the cache (b) one of 2 possible entries (ways) in the cache (c) any entry (way) in the cache

1.2 Motivation for change

In conventional 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

Table 1.1: 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

bandwidth and pollute the cache, which increases the # of misses. We evaluate the influence of a fixed granularity block below.

1.2.1 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 [2], 7 from SPEC2006, 2 from SPEC2000, 3 Java workloads from DaCapo [3], 3 commercial workloads (Apache, SpecJBB2005, and TPC-C [18]), 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 1.1).

Figure 1.3 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

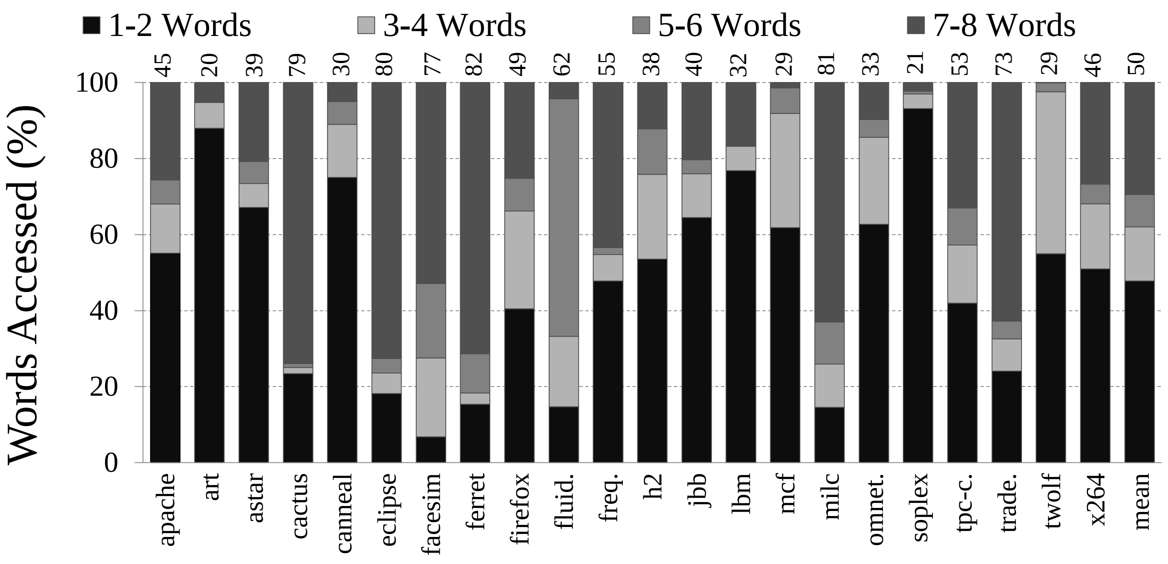


Figure 1.3: Distribution of words touched in a cache block. Avg. utilization is on top. (Config: 64K, 4 way, 64-byte block.)

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.

1.2.2 Causes of poor cache utilization

Applications display poor cache utilization due to inefficient data structure access patterns. This could be due to

1. Programming practices : The array of structs (AoS) approach is a common programming practice. While performing computations upon the array if all elements of the struct are not referenced in close proximity, it could cause poor cache utilisation. A relevant example can be seen in listing 1.1.

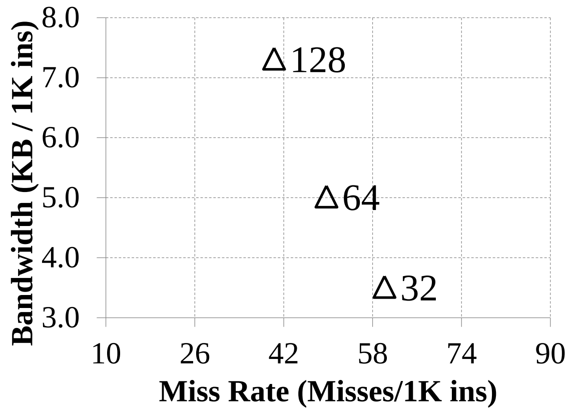
2. Incorrect assumptions about hardware : Hardware conscious code which attempts to optimise for cache behaviour based on assumptions should be ported carefully. We found **streamcluster** of the PARSEC application suite, by default, attempt to optimise cache behaviour for 32 byte cache line sizes. This finding was also reported by Liu and Berger[17].

3. Compiler directives : For better cache performance, sometimes compilers can attempt to allocate aligned blocks of memory. This functionality is exported to the programmer as `posix_memalign` by *GCC*, `__aligned_malloc` by *MSVC* and `ippMalloc` by *ICC*. These allocators may leave gaps filled with garbage values which are picked up by the cache when an entire line is fetched, thus reducing the effective caching capacity and reducing utilization.

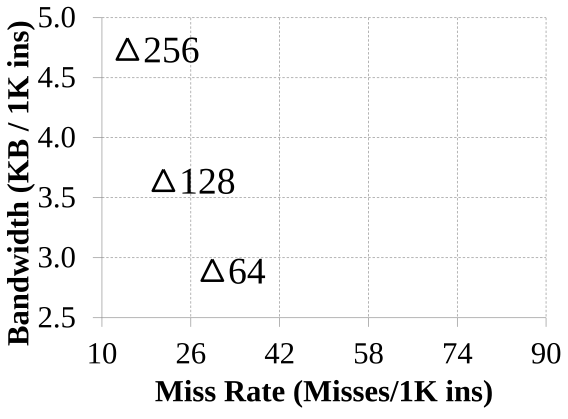
4. Interaction with cache geometry : Due to the set associative nature of conventional caches, a set can only contain a fixed number of ways. For instance, if a large amount of data is accessed in a strided fashion which happens to map to the same set, will cause evictions even though there may be space available for use in the other sets of the cache. This shortens the lifetime of the cache blocks in the selected set and may reduce utilization.

```
/* blackscholes.c:354 */
for (i=0; i<numOptions; i++)
{
    otype[i]      = (data[i].OptionType == 'P') ? 1 : 0;
    sptprice[i]   = data[i].s;
    strike[i]     = data[i].strike;
    rate[i]       = data[i].r;
    volatility[i] = data[i].v;
    otime[i]      = data[i].t;
}
```

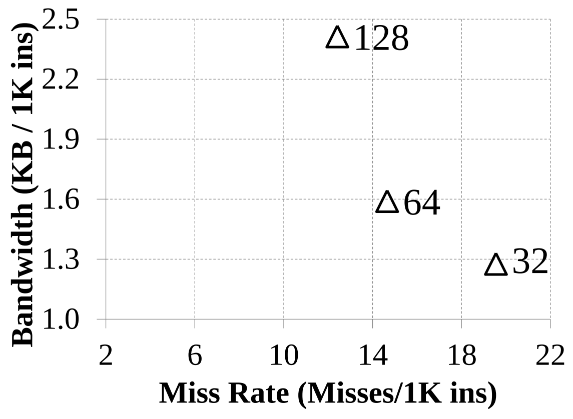
Listing 1.1: Code snippet from the initialisation phase of **blackscholes** benchmark from the PARSEC 2.1 [2] application suite. The code references each *OptionData* structure in the data array where the first 6 fields (24 bytes, as observed on a x86-64 machine with Ubuntu and gcc version 4.4.7) are referenced out of each struct which contains 9 fields (36 bytes). The problem is exacerbated as the *OptionData* structure is allocated as a single chunk and is not cache aligned. The rest of the program demonstrates good cache behaviour.



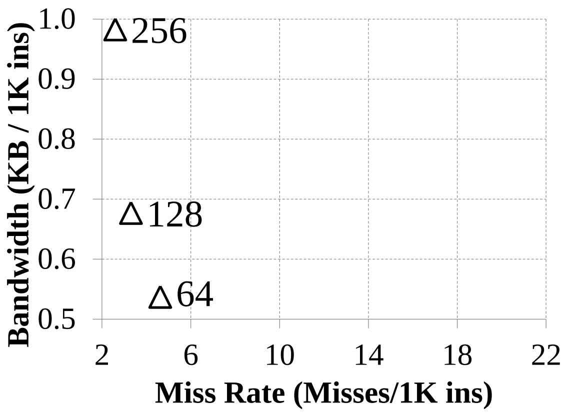
(a) 64K - Low



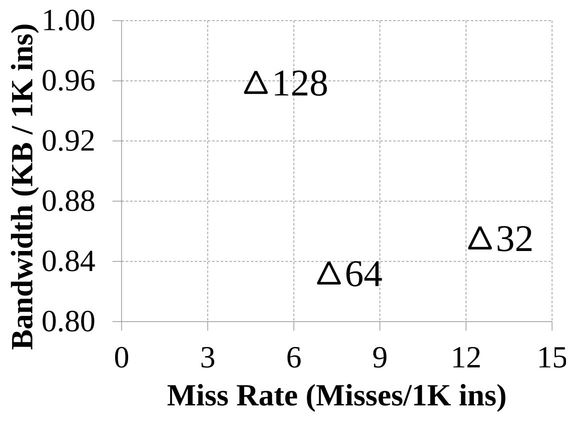
(b) 1M - Low



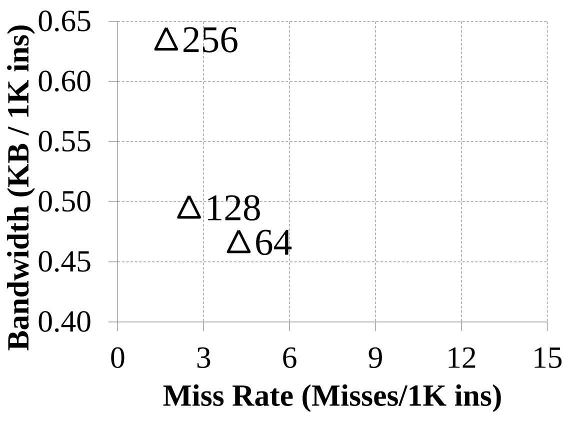
(c) 64K - Moderate



(d) 1M - Moderate



(e) 64K - High



(f) 1M - High

Figure 1.4: 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.

1.2.3 Effect of Block Granularity on Miss Rate and Bandwidth

Cache miss rate directly correlates with performance, while under current and future wire-limited technologies, bandwidth directly correlates with dynamic energy. Figure 1.4 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.

Table 1.2: Optimal block size. Metric: $\frac{\mathbf{1}}{\mathbf{Miss-rate \times Bandwidth}}$

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

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 1.2 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.

1.2.4 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.

1.3 Dissertation Outline

Chapter 2 describes the *Amoeba-Cache* architecture whilst comparing it with conventional architecture and looking at related work. The hardware complexity, implementation issues and simulator infrastructure are discussed in Chapter 3. The experimental results of an exhaustive evaluation of the *Amoeba-Cache* is presented in Chapter 4. Conclusions and future work are outlined in Chapter 5.

Chapter 2

Amoeba Cache Architecture

As described in § 1.1, a conventional cache organises the data array into a 2 dimensional structure. A transparently addressed cache uses the same namespace (memory address space layout) as the main memory. The blocks which are stored in the sets are *tagged* with the aligned start address of block present in the main memory. The *tags* for the cache blocks currently present in the cache set are maintained in a separate array. When a search is being performed to find out whether a required physical address is present in the cache, the tag array is looked up to determine a cache hit or a cache miss. The organization of the cache set and tag array is shown in Figure 2.1. The effective address is the virtual address supplied by the CPU of the required datum. The component bits of the effective address is segmented into 3 parts which form the *Virtual Page Number(VPN)*, set number and byte offset. The set number and byte offset are looked up in the tag array while the VPN is looked up in the *Translation Lookaside Buffer(TLB)* to check that the current process has brought in the corresponding page and it is valid. The organisation described (and shown in Fig 2.1) is virtually indexed, physically tagged organisation where the lookup logic does not include the TLB translation in the critical path to enable faster searches. There are other organisations such as virtually indexed, virtually tagged and physically indexed, physically tagged which are uncommon due to inherent issues with their design. The tradeoff for a virtually indexed, physically tagged cache is that it can only grow in size with an increase in the associativity of each set, or an increase in the size of each cache block. The Intel Sandy Bridge architecture is known to use a virtually indexed, physically tagged cache organisation.

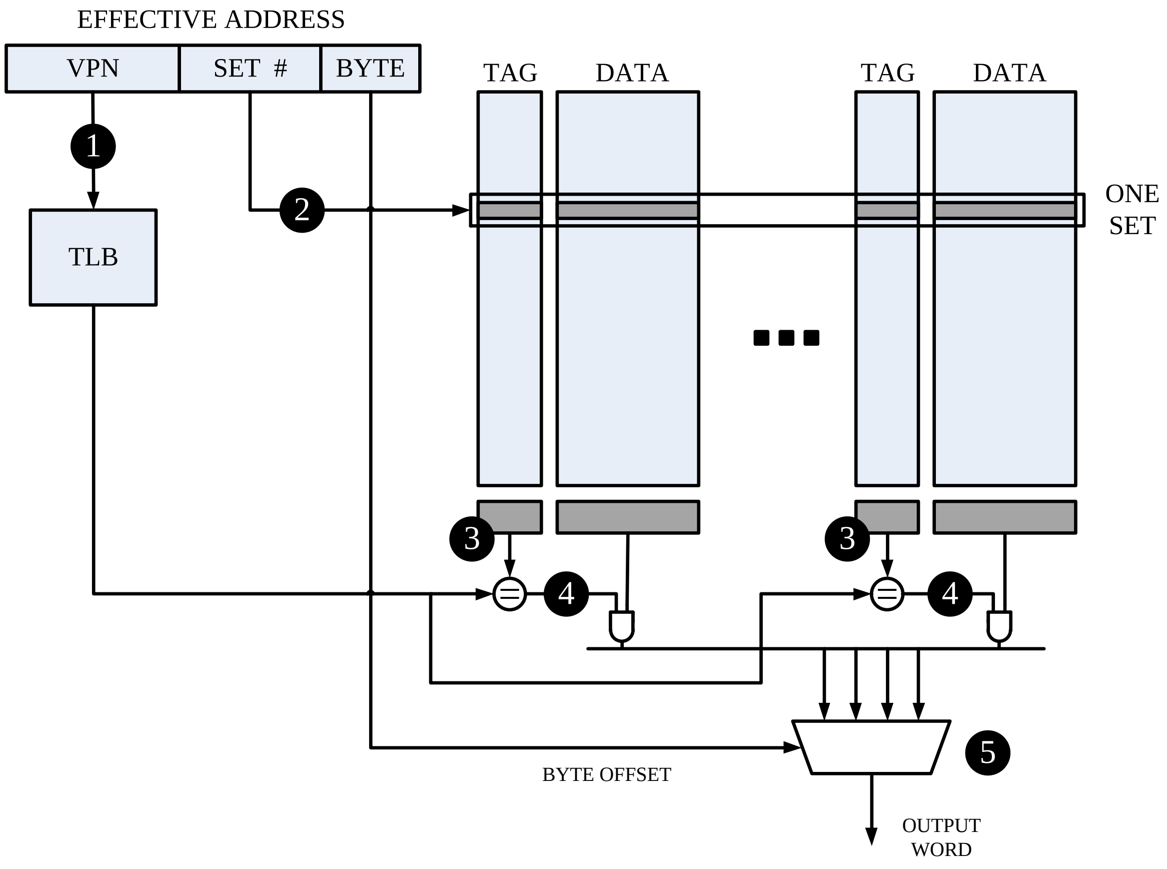


Figure 2.1: **Conventional N-Way Set-Associative Cache** ❶ The Virtual Page Number (VPN) is used to look up the entry in the Translation Lookaside Buffer (TLB) ❷ According to the number of sets in the cache, the following bits from the address are used to look up the corresponding set from the cache ❸ The tags read out from the set are compared with the translation from the TLB and tested for equality ❹ The corresponding cache block is forwarded to the output buffer for the tag which matches the TLB lookup ❺ Using the byte offset from the CPU, the mutiplexer selects the correponding critical word

In contrast to a conventional cache, the *Amoeba-Cache* architecture enables the memory hierarchy to fetch and allocate space for a range of words (i.e. a variable granularity 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 challenges to realising the *Amoeba-Cache* architecture are

1. To support a variable number of blocks per set
2. To support a variable granularity for each block
3. To support a variable number of tags, which correspond to the blocks in the set

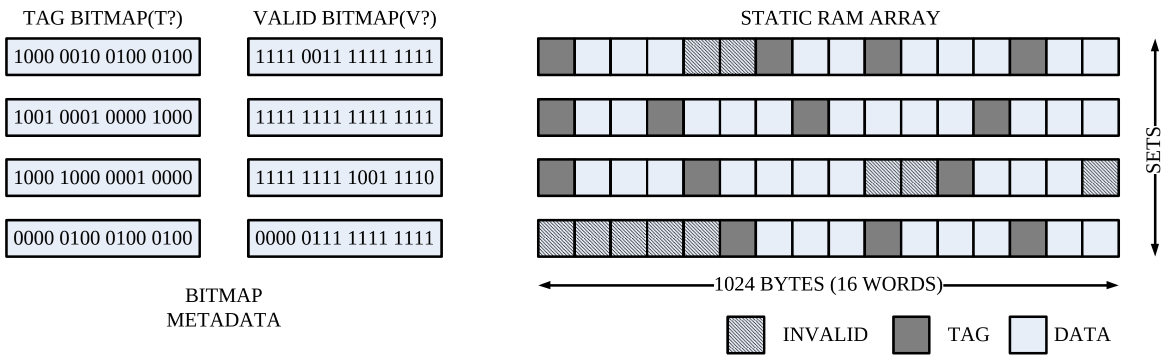
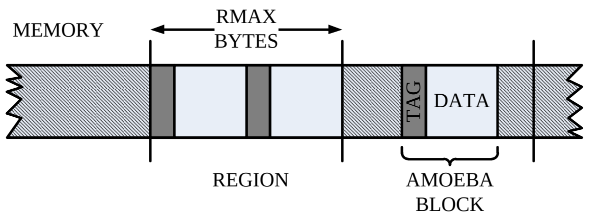


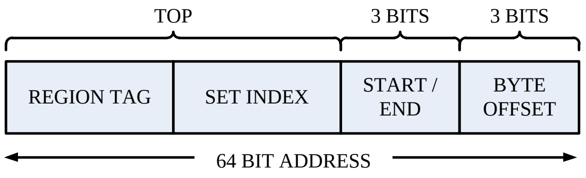
Figure 2.2: **Amoeba Cache Overview** The static RAM (SRAM) array where the tags and data are colocated is shown on the right. The `T? Bitmap` and the `V? Bitmap` for the *Amoeba-Cache* are shown on the left. Each block in the SRAM array represents 8 bytes (1 word). In this specific example, an *Amoeba-Cache* is shown with 4 sets and 1024 bytes per set. The invalid, data and tag words (marked in the SRAM array) are tracked by setting the corresponding bits in the `T? and V? Bitmaps`. This information is maintained in order to simplify cache operations such as insertion and refill.

The *Amoeba-Cache* adopts a solution inspired by software data structures, where programs hold meta-data and actual data entries in the same address space. To achieve maximum flexibility, the *Amoeba-Cache* completely eliminates the tag array and collocates the tags with the actual data blocks (see Figure 2.2). To distinguish which words are data words and which ones are tags within the set, a bitmap data structure is used (labeled `T? Bitmap` in Fig 2.2). For each word in the set which is a tag, the corresponding bit in the `T? Bitmap` is set. The conventional valid/invalid bits are also decoupled (typically associated with the tags) and organized into a separate array (labeled `V? Bitmap` in Fig 2.2) to simplify block replacement and insertion. *Amoeba-Cache* tags are composed of a `Region Tag` and a tuple which consists of the `Start` and `End` address of the variable granularity cache block. The data block immediately follows the tag word as shown in Fig 2.2. The following sections provide more detail about the *Amoeba-Cache* architecture and how cache operations are performed.

2.1 Amoeba Blocks and Set-Indexing



(a) Memory Regions



(b) Addressing

Figure 2.3: (a) The linear memory address space is segmented into Regions. The *Amoeba-Block*s are constrainted to have their start and end within a single memory region. (b) 64 bits are used to encode the Region Tag, the Set Index, the start or end word and the word offset in the tag for an *Amoeba-Block*.

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 <`RegionTag, Start, End, Data-Block`> (Figure 2.2). The first 3 components of the tuple are equivalent to a tag in a conventional cache. 8 bytes are allocated (1 word) for each tag. In order to simplify cache lookups for *Amoeba-Block*s, the address space is partitioned into `Regions`. A `Region` is an aligned block of memory of size `RMAX` bytes. The boundaries of any *Amoeba-Block* block (`Start` and `End`) are constrained to lie within the regions' boundaries. The minimum granularity of the data in an *Amoeba-Block* is 1 word and the maximum is `RMAX` words. The `Start` and `End` can be encoded 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, `RMAX` is set 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.

2.2 Data Lookup

When data is referenced by the CPU, a cache lookup takes place in order to determine whether the required datum is present in the cache or not (resulting in a cache hit or miss). The operation percolates down the memory hierarchy until a cache returns a hit or the backing store supplies the datum required. Fig 2.1 shows a conventional cache which operates in *Fast Mode*, where the contents of the entire set is read out into the output buffer in parallel with the tag lookup. Megabyte sized caches, with larger sets, may want to avoid the extra cost of reading out all ways to the output buffer and wait until the tag lookup completes to read out only the required way from the set over the H-tree (for more details see § 3.1.2). Though this saves energy, it serializes the lookup and takes longer. The delay is usually tolerated as the *Serial Mode* lookup is often implemented in the L2 caches or lower in the memory heirarchy. Another approach which minimises energy whilst still reading out the way in parallel is *way prediction*[24, 36], commonly used in processors manufactured by MIPS.

In contrast to a conventional cache, the *Amoeba-Cache* needs to lookup tags from the SRAM array to determine a hit or miss. The metadata stored in the `T? Bitmap` is used during the lookup operation in the *Amoeba-Cache*. Figure 2.4 describes the steps of the lookup procedure in an *Amoeba-Cache*. The overheads incurred due to the extra stages introduced in the critical path are evaluated in chapter 3.1.

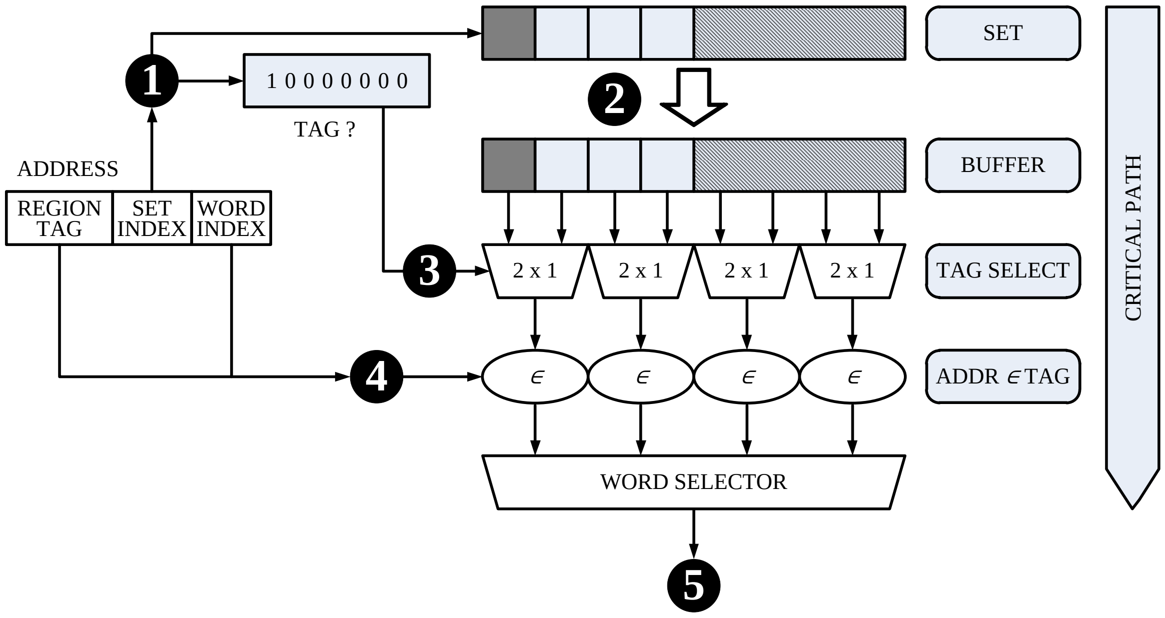


Figure 2.4: **Amoeba Cache Lookup** : The incoming effective address is segmented into the region tag, set index and word offset. ❶ The `Tag? Bitmap` is looked up to determine which words in the activated set are required for the tag comparison. Note that given the minimum size of a *Amoeba-Block* is two words (1 word for the tag metadata, 1 word for the data), adjacent words cannot be tags. Given this constraint, the number of 2-1 multiplexers required to route one of the adjacent words to the comparator ($\in$ operator), is equal to half the number of words in the set. ❷ Simultaneously, the set is activated and the contents are latched onto the output buffer. ❸ The appropriate tag words are selected with the input from the the `Tag? Bitmap`. ❹ 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_2N_{sets} - log_2RMAX 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_2RMAX bits wide; e.g., 3 bits) that checks if the *Amoeba-Block* holds the required word. Finally, in ❺, the tag match activates and selects the appropriate word. The critical path (as indicated on the left) includes the read out from the set, the tag selectors and the $\in$ operation.

2.3 Block Insertion

On a miss for the desired word, a spatial granularity predictor is invoked (see § 2.6), which specifies the range of the *Amoeba-Block* to refill. To determine a position in the set to slot the incoming block use the information contained in the `Valid? 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, a substring search is performed 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), a substring search is performed for *00000* in the `V? Bitmap` / set (e.g., 32 bits wide for a 64K cache). If the required amount of space is not found, the replacement algorithm is repeatedly triggered until the required amount of contiguous space is found. Following this, the *Amoeba-Block* tuple (Tag and Data block) is inserted, and the corresponding bits in the `T? and V? Bitmaps` are set. The 0s substring search can be accomplished with a lookup table; many current processors already include a substring instruction. Intel SSE 4.2 include the instruction, `PCMPISTRI`, which can accomplish the required task.

2.4 Replacement Policy

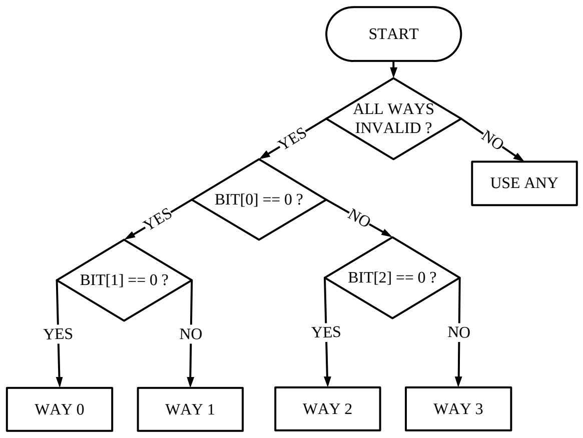
The replacement policy of a cache determines which blocks are evicted when new data is brought in and there is no room to store it. The replacement algorithm tries to make an optimal choice where it evicts a blocks which is not expected to be used in the near future. The most optimal choice possible would be to remove a block which is not going to be referenced in the program again or which is going to be referenced farthest in the future in comparison to the other cache blocks. The optimal algorithm is also known as *"Belady's Optimal Algorithm"*, named after Hungarian computer scientist, Laszlo Belady.

The Least Recently Used (LRU) algorithm is a popular choice for conventional caches and is based on the principle of temporal locality of reference. The hardware tracks data reference to the different cache blocks and when a new insertion request arrives, the least recently used block in the set is evicted. Implementing an LRU cache in software is relatively easy with the use of a linked list. Hardware manufacturers commonly implement the *Pseudo-LRU*(PLRU) which has a lower metadata overhead and is a reasonable approximation of

LRU. The tree based PLRU was implemented in processors such as the Intel 80486 and many processors in the PowerPC family.

2.4.1 Pseudo-LRU

The tree based Pseudo-LRU algorithm was developed as an approximation of LRU due to the exponentially increasing complexity of storing the LRU state for increasing number of ways (an N_{way} cache would require $N!$ states). For multi threaded processor designs and megabyte sized caches, it is often desirable to have a high level of associativity. For Pseudo-LRU, one bit indicates whether the most recent reference is to a line in either the first half or the second half of the lines in a set; then, this technique is logically applied recursively, resulting in a logical binary tree with N-1 nodes (thus N-1 bits are required to represent a state in tree-PLRU). A 4-way set associative can be encoded for PLRU replacement with 3 bits. Each bit represents one branch point in a binary decision tree; let 1 represent that the left side has been referenced more recently than the right side, and 0 vice-versa.



STATE TRANSITION TABLES

STATE	REPLACE
00x	WAY 0
01x	WAY 1
1x0	WAY 2
1x1	WAY 3

‘x’ means don’t care

(a)

REFER	STATE
WAY 0	11_
WAY 1	10_
WAY 2	0_1
WAY 3	0_0

‘_’ means unchanged

(b)

Figure 2.5: **Pseudo-LRU decision tree and state transition table** : The flowchart on the left shows the decision making process based on the values of the 3 PLRU bits in a 4-way associative cache. The state transition tables on the right are (a) way to replace for given state (b) state to transition to on reference to a way. Reproduced from Intel Embedded Pentium Processor Family Dev. Manual[7].

2.4.2 *Amoeba-Cache* Replacement Policy

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 [13, 22] 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 [26] that work with a fixed number of ways, a set in *Amoeba-Cache* can be logically partitioned into N_{ways}. For instance, partitioning 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*. This procedure is repeated until the required amount of space is made available. 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.

2.5 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 2.6 shows an example. Initially, the set contains two blocks from the region R, one *Amoeba-Block* caches words 0 — 2 (Region:R Start:0 End:2) and the other caches words 6 and 7 (Region:R Start:6 End:7). Let us assume that the CPU reads word 4, which misses, and the spatial pattern 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 only some words (4,5 and 5) need to be fetched.

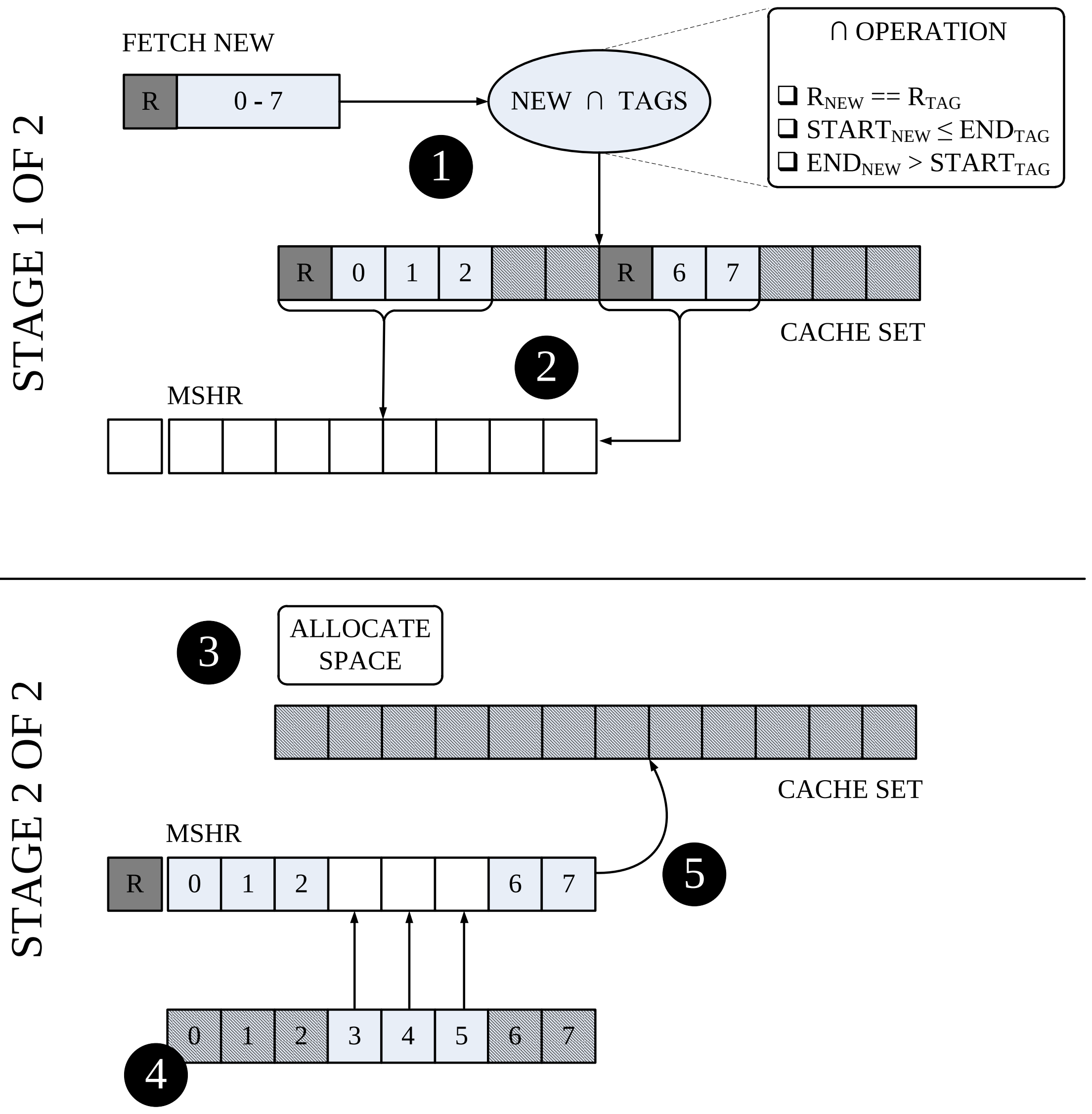


Figure 2.6: **Partial miss processing** : The partial miss processing is shown as 2 separate stages. Stage 1 deals with the identification and eviction (to MSHR) of all overlapping blocks. Stage 2 issues the miss, allocates space for the new *Amoeba-Block* and patches in the missing words from a lower level in the memory hierarchy before copying the data back into the cache set. Due to the infrequent nature of partial misses the implementation does not optimise for fetching only the missing words.

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. The $\cap$(in) operation returns false if there are no overlapping blocks. The $\cap$ operation consists of an equality check for the region (R), $Start_{New} \leq End_{Tag}$ and $End_{New} > Start_{Tag}$ (where *New* refers to the fetch request from the spatial pattern predictor and *Tag* refers to the existing *Amoeba-Block*). ❷ The data blocks that overlap with the miss range are evicted and moved one-at-a-time to the miss status holding register (MSHR) entry. ❸ Space is then allocated for the new block, i.e., it is treated like a new insertion. A miss request is issued for the entire block (Start:0 — End:7) even if only some words (e.g., 3, 4 and 5) may be needed. This ensures request processing is simple and only a single refill request is sent. ❹ The incoming data block is patched into the MSHR; only the words not obtained from the L1 (words 3,4 and 5 as indicated in Fig2.6) are copied (since the lower level could be stale). ❺ The entire block is copied back into the SRAM array.

2.6 Spatial Pattern Predictor

The *Amoeba-Cache* uses 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 [5, 14, 25, 4]. A tabular approach is adopted, as shown in Fig 2.7, 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. The entry is indexed with information gleaned from either the program counter (PC) which initiated the cache miss or region tag for the effective address or both. On a miss, the predictor will search for an entry and choose the range of words to be fetched on a miss on either side (left and right) of the critical word. The predictor optimizes for spatial prefetching and will overfetch (bring in potentially untouched words), if they are interspersed amongst contiguous chunks of touched words. The prediction can also be bypassed when there is low confidence in the prediction accuracy, i.e. when there may not be an entry in the table. Tradeoffs in the design of the spatial predictor are evaluated in § **??**.

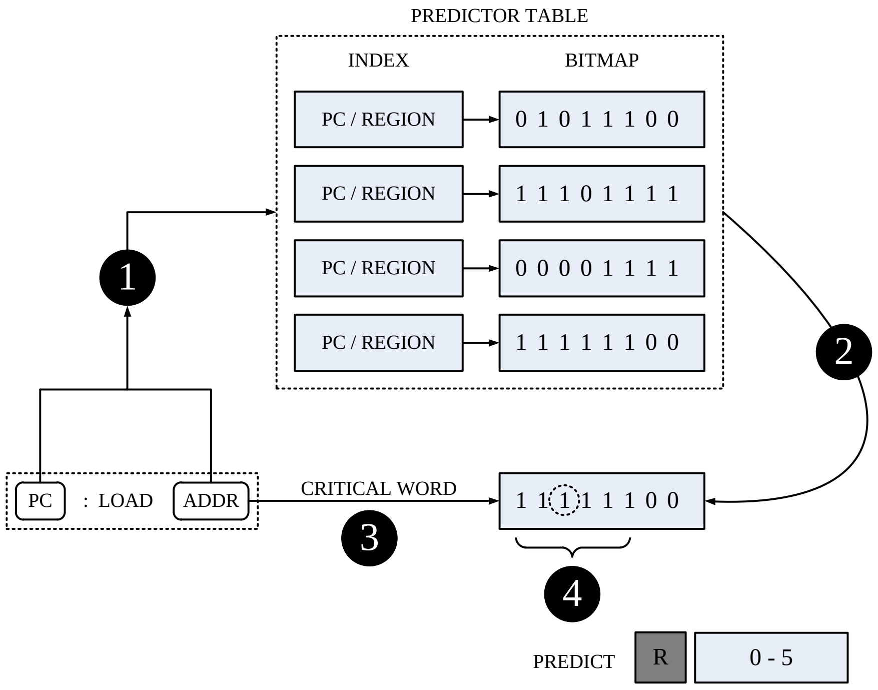


Figure 2.7: ***Amoeba-Cache*** **Predictor** : On a cache miss, the *Amoeba-Cache* predictor performs the following steps. ❶ Lookup the table using bits from the program counter or region tag or a combination of both. ❷ If an entry is present for the calculated index, the corresponding bitmap is read out from the table. The bitmap is updated on eviction of a cache block with the pattern of words accessed by the CPU in the cache block during its lifetime. ❸ The critical word (word requested by the instruction) is examined to ensure it is part of the marked portion of the bitmap. The range(`START` and `END`) for the new block is extended to include all marked words in the bitmap. ❹ The predictor then requests the cache to issue a miss for words 0 – 5 based on the bitmap pattern.

2.7 Related Work

This section describes prior work in the area which closely relate to the *Amoeba-Cache*.

2.7.1 Line Distillation

Qureshi et al[27] proposed a cache organisation dubbed as the *Distill Cache* and a technique known as *Line Distillation* to discard only the unused words from a cache line upon eviction. Their proposed organisation (as shown in Fig 2.8) includes a *Line Organised Cache(LOC)* which stores blocks at the default line granularity (such as 64 bytes per line) and a *Word*

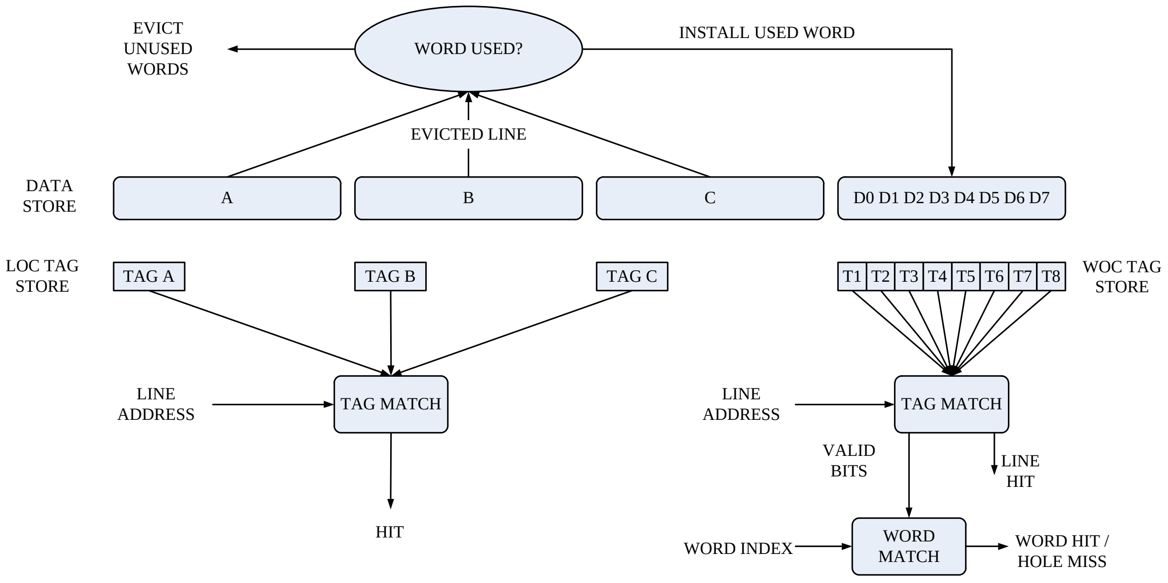


Figure 2.8: **Line Distillation Organisation** : The figure shows a single set from a 4-way *Distill Cache*. The LOC consists of ways A,B and C and the WOC consists of way D. A line evicted from the LOC ways is checked for suitable words to retain in the WOC. A lookup in the WOC includes a check for the line address as well as the word index to qualify as a hit.

Organised Cache(WOC) which stores blocks at the granularity of a word (usually 8 bytes). The WOC is similar in spirit to the *Victim Cache*[12] proposed by Jouppi. The victim cache is a small, fully associative cache which is usually coupled with a direct mapped cache and stores the victim of a cache miss. It was used to alleviate problems caused by conflict misses. The WOC however, does not cache the entire block. The WOC only caches the words within the cache block which were touched by the application at the time of eviction. Whether the words will be installed in the WOC or the entire line is discarded is determined by the number of words which were touched. The authors propose a *median threshold filtering* policy, where the words are retained in the WOC on eviction only if the number of words touched in the cache line from the LOC is less than the median number of words touched in a cache block in the application. The authors propose the implementation of the *Distill Cache* at the L2 as the design of the L1 data cache is is heavily constrained by cycle time and out of order processors already cover some of the latency of the cache misses. Their findings show an average improvement of 12% in IPC by reducing cache misses for a 1MB 8-way L2 by 30%.

The *Distill Cache* is similar to the *Amoeba-Cache* in the sense that in a single cache it supports word granularity blocks and line granularity blocks as two extremes. The *Amoeba-Cache* supports varying granularity blocks of sizes 1–$RMAX$ words (see § 2.1). The WOC always incurs a high overhead in terms of a single tag for each of the data words in the WOC which is similar to the worst case scenario for an *Amoeba-Cache* where all the data words in the cache set are of size 1 word. Veidenbaum et al[32], propose the entire cache be word organised and propose an online algorithm to prefetch required words. This approach however has a built in tag overhead of 50% and requires energy intensive associative searches.

2.7.2 Sector Caches

The original cache designs[16] were essentially sector caches in nature was due to the technology available at the time (the discrete transistor logic for sectors was easier to implement) where the cache consisted of sectors (address tags) and subsectors (or blocks with valid bits). They were deprecated in favor of set associative caches which have superior performance in terms of miss rate and because many subsectors went unused due to early eviction of a sector (for a 64K data cache with 256 byte sector frame and 64 byte sector size had only 74% utilization[28]).

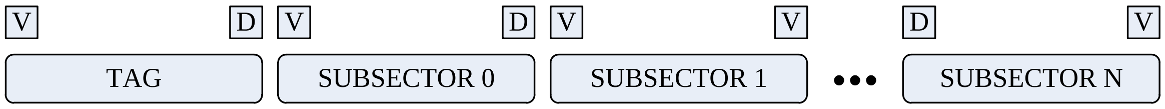


Figure 2.9: **Sector Frame** : Diagram of a single sector frame (reproduced from [28]). D is the dirty bit and V is the valid bit associated with the subsector. The first documented use of a cache in a computing system, the IBM 360/85, was 16 KBytes in size and consisted of 16 sector frames of 16 subsectors, each of size 64 byte blocks.

In the 90's interest in sector cache design was revived by Rothman et al[28] and Seznec[29, 30] due to their suitability for large cache design. Sector caches are also desirable due to reduced bus traffic and smaller latency overheads. The Intel Pentium 4, SUN SPARC and IBM PowerPC G4/G5 all incorporated sector designs into the cache memory hierarchy. The Power7 architecture also employs a sector design at the L2 with 32 bytes per subsector. Sector caches can also be combined with victim caches as proposed by Lai et al[15] to reduce miss rates. Prior work on spatial pattern predictors by Kumar et al[14], Pujara et al[25] and Yoon et al[34, 35] use sector caches as the substrate for their proposals.

Sector caches provide variable granularity caching at the granularity of the subsector size. However, the space freed up by unused subsector cannot be reused by a different sector frame. The *Amoeba-Cache* is able to provide adaptive granularity caching at a granularity of a word and the unused space can be used by *Amoeba-Block*s from other regions. Decoupled sector caches[29] help reduce the number of invalid subsectors within a sector by increasing the number of tags per sector. Compared to the *Amoeba-Cache*, the tag space is a constant overhead and limits the number of invalid sectors that can be eliminated. Pujara et al[25] consider a word granularity sector cache and use a predictor to try and bring in only the used words. Our observations show (see Fig **??**) that smaller granularity sectors increase misses and optimisations that prefetch can pollute the cache and interconnect with unused words.

2.7.3 Indirect Index Caches

Indirect index caches(IIC) were proposed by Hallnor et al.[10] as a practical, fully associative, software managed secondary cache system that does not require OS or application intervention. Their goal was to provide LLCs the benefits of full associativity and software management similar to DRAM architectures. Their motivation was the magnitude of the increasing gap in latency between LLC and DRAM which is similar to the existing gap between DRAM and disk. In a conventional cache each way is statically associated with a tag entry and indicates whether the way is valid. In the IIC, the tag entries are not associated with particular data entries (ways). A tag entry in the IIC contains a pointer to the data block, i.e. an index into the cache's data array. And since a tag entry can indicate any data array location, the cache is fully associative. Figure 2.10 illustrates an example IIC design.

The tag storage requirement for the IIC is greater than a conventional cache as the position of an entry in the tag store for the IIC is not related to the corresponding physical address of the block. Assuming a 48-bit physical address for a 1M cache with 256 byte blocks, the tag entry in the primary tag storage, as shown in Figure 2.10, consists of tag bits ($48 - \log_2(256) = 40$ bits), status bits (3 bits - valid, dirty and unreferenced), the index of the data block in the data array ($\log_2(1M/256) = 12$ bits), chain pointer ($\log_2(1M/256) = 12$ bits) and replacement policy data storage (32 bits). Thus each tag entry is 99 bits in size. The latency overheads for the IIC are caused by : serial access of tag and data, additional hit and miss latency overhead due to hash table lookups and additional miss latency due to software management.

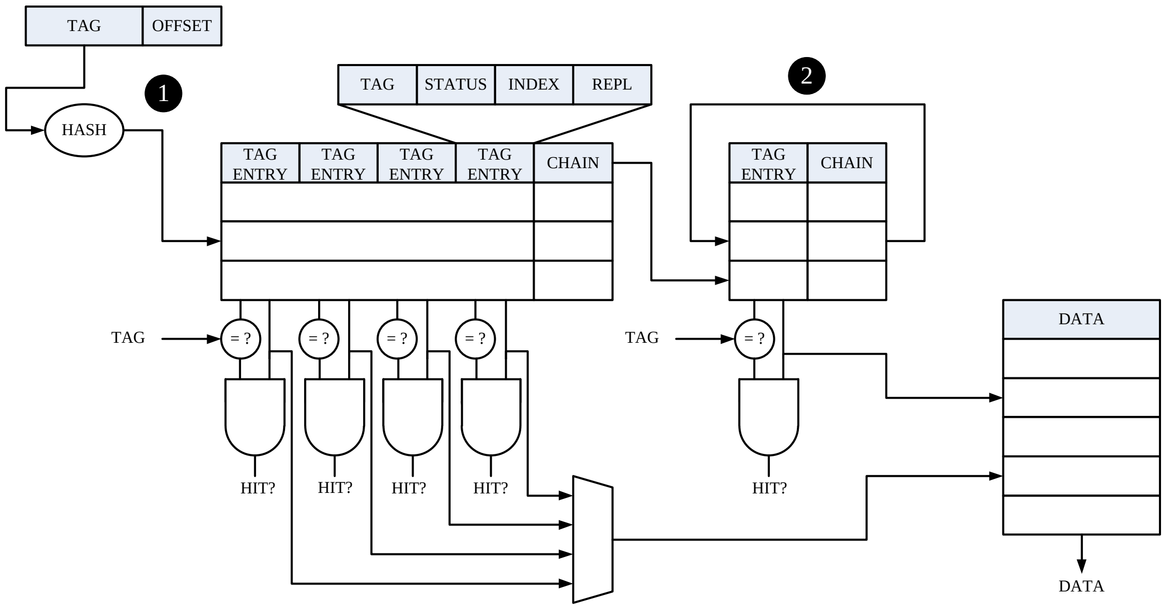


Figure 2.10: **Indirect Index Cache** : The tag array for an IIC is split into two parts. A primary hash table and a secondary block for chaining. ❶ On each access, the block tag is hashed to obtain a primary table index. The primary table is associative, so a single search accesses the first few entries (4 in the example figure) of the hash chain. Collisions beyond this depth are chained into the secondary hash storage as shown in ❷.

In 2004, Hallnor et al[11] proposed a compressed memory hierarchy based on the IIC. Their design to support compression in the LLC was heavily inspired by IBM's Memory Expansion Technology(MXT)[31]. The IIC already supported the random placement of blocks within a set with the index of the data block being stored in tag entry. With suitable modifications, the IIC was made to support compressed, variable granularity blocks.

The IIC-Compressed design is similar to the *Amoeba-Cache* in the sense that it can cache variable granularity data blocks. However, the increased tag structure complexity and nature of serialised access to tags and data detract from its merits. The goals of caching variable granularity data blocks are different for the *Amoeba-Cache*(bandwidth and energy reduction) and the IIC(caching compressed blocks).

2.7.4 Other related work

Recently Yoon et al. have proposed an adaptive granularity DRAM architecture[34]. This provides the support necessary for supporting variable granularity off-chip requests from an

Amoeba-Cache-based LLC. Some research [9, 6] 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. [5]) to restructure data for improved spatial efficiency. There have also been efforts from the architecture community to predict spatial locality [25, 33, 14, 35], which can be used 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].

Chapter 3

Implementation

This chapter describes the additional hardware complexity added to an *Amoeba-Cache* with respect to a conventional cache in order to support variable granularity *Amoeba-Block*s. The chapter also describes the simulation infrastructure used to evaluate the performance of the *Amoeba-Cache*.

3.1 Hardware complexity

The complexity of the *Amoeba-Cache* is analysed along the following directions:

- Additional complexity of the cache controller
- Area, latency and energy overhead
- Challenges of megabyte sized *Amoeba-Cache*s

3.1.1 Cache Controller

Each level of the cache memory hierarchy incorporates a finite state machine (FSM) called the *cache controller* to track the state of the blocks currently being cached. The cache controller is the interface between the CPU, cache and DRAM as shown in Fig 3.1(a). The figure depicts a look through hierarchy where the processor is isolated from the system, thus allowing the processor to run with data from the cache while another bus master is acessing main memory. Fig 3.1(b) shows an overview of the implementation logic for the cache controller. Using the current state of the block and the incoming request (which could be from the CPU, DRAM or a remote cache) a new state is set for the cache block. The state transition diagram of an L1 *Amoeba-Cache* is shown in Fig 3.2. The FSM for the *Amoeba-Cache* is a superset of an equivalent FSM for a conventional cache (without cache coherence; more details in § 3.1.6).

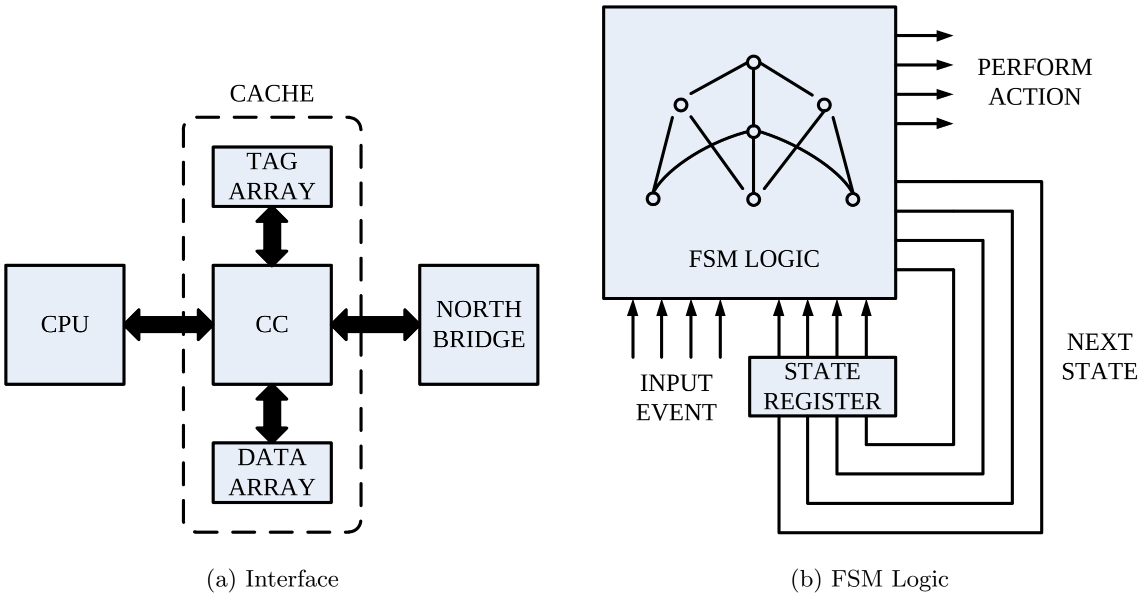
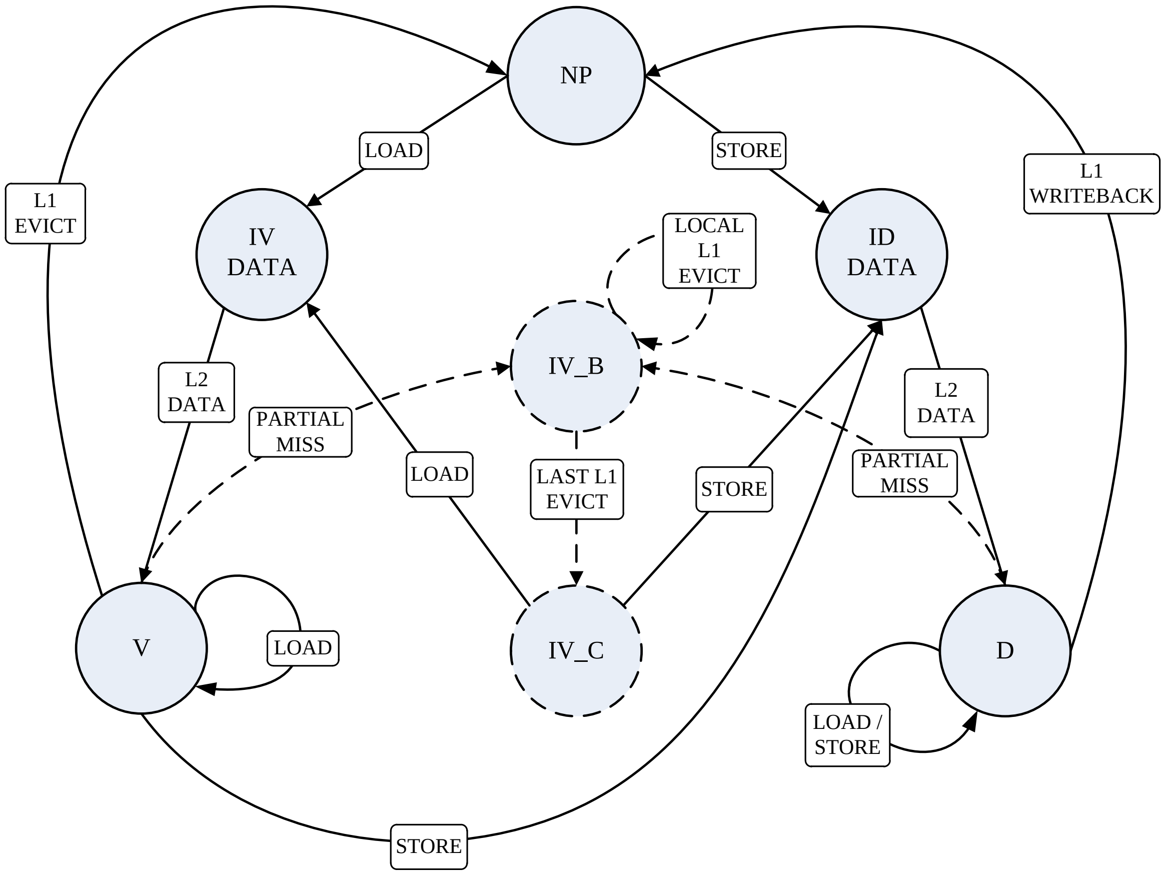


Figure 3.1: Fig (a) on the left shows a high level block diagram of the cache controller's (CC) position in the memory hierarchy. The *Northbridge* is used to manage data communcations between the CPU and motherboard and is also where the memory controller resides. In newer architectures such as the Intel Sandy Bridge, the *Northbridge* has been integrated on chip. Fig (b) on the right depicts a cache controller where the input data path from the cache (event) is used along with the current state to perform the corresponding action and change the state of the block as required.

The cache controller manages operations at the aligned RMAX granularity. The controller permits only one in-flight cache operation per RMAX region, i.e. transition buffer entries in the cache controller are indexed using the region bits. In-flight cache operations ensure no address overlap with stable *Amoeba-Block*s in order to eliminate complex race conditions. Fig 3.2 shows the L1 cache controller state machine for the *Amoeba-Cache*.



L1 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-Cache* specific 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.

Figure 3.2: **L1 Cache Controller for *Amoeba-Cache*** : The *Amoeba-Cache* specific states and events are marked by dashed lines. The *Amoeba-Cache* specific events are described in the table.

A simple default single core protocol is assumed which contains blocks in either `Valid (V)`, `Not Present (NP)` and `Dirty (D)` as shown in Fig 3.2. There are also two transient states `IV Data`(block transitions from `Invalid/NP` to `Valid`, waiting for `Data` response from memory; cache miss caused by a load) and `ID Data`(block transitions from `Invalid/NP` to `Dirty`, waiting for `Data` response from memory; cache miss caused by a store). To handle partial misses, unique to the *Amoeba-Cache*, two new states are added to the default protocol, `IV_B` and `IV_C`. `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. `IV_C` indicates partial miss completion. This enables the controller to treat the access as a miss and issue the refill request. The `Partial Miss` event triggers the clean-up operations (Stage 1 and Stage 2 in Figure 2.6). `Local_L1_Evict` is an event that keeps being triggered 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 (see Stage 2 of Fig 2.6).

A key difference 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, each block is read out independently before supplying the data (more details in § 3.1.5).

3.1.2 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 as shown in Fig 2.2. Table 3.1 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 siz) to the overall cache access energy. `V?`is referred only on misses when inserting a new block.

The *Amoeba-Cache* lookup logic was synthesized[1] using Synopsys to quantify the area, latency and energy penalty. *Amoeba-Cache* is compatible with *Fast* and *Normal* cache access modes [23], 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

[1]Actual synthesis was at 180nm node size and the results were scaled to 32 nm (latency and energy scaled proportional to Vdd (taken from [8]) and Vdd^2 respectively). For synthesis, we used the Synopsys design compiler (Vision Z-2007.03-SP5).

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

Table 3.1: ***Amoeba-Cache* hardware overheads** : Percentage indicates overhead compared to data array of a cache. 64K cache operates in *Fast mode*; 1MB and 4MB operate in *Normal mode.* International Technology Roadmap for Semiconductors (ITRS) specified 32nm High Performance (HP) transistors are assumed for 64K cache and 32nm ITRS Low Output Power (LOP) transistors for 1MB and 4MB.

H-tree, while *Normal* mode, only transmits the selected way over the H-tree.

Fig 2.4 shows *Amoeba-Cacheś* lookup hardware overhead on the critical path. The *Amoeba-Cache* lookup logic is compared against a conventional 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 [23]. *Amoeba-Cache*'s comparators, which operate on the entire cache set, are 6$\times$ the area of a fixed cache's comparators. In a conventional cache, 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 $\simeq$ 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.

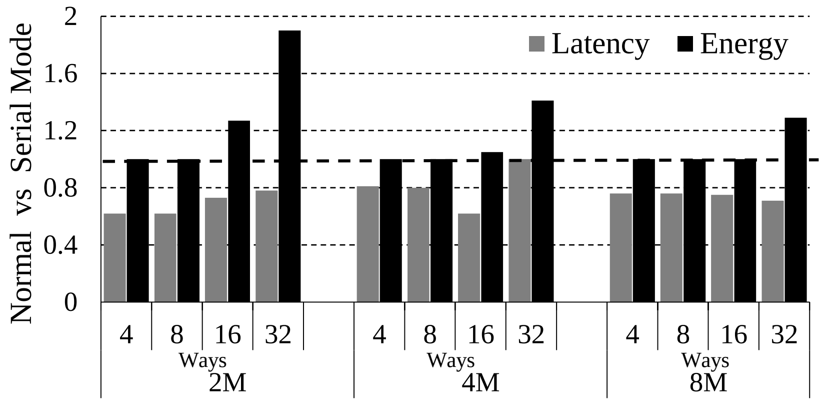
Amoeba-Cache's overhead needs careful consideration when implemented at the L1 cache level. There are two options for handling the latency overhead a) if the L1 cache is the critical stage in the pipeline, the CPU clock can be throttled by the latency overhead to ensure that the additional logic fits 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) an extra pipeline stage can be added to the L1 hit path, adding a 1 cycle overhead to all memory accesses but ensuring no change in CPU frequency. The performance impact of both approaches is quantified in § **??**.

3.1.3 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 invalidations, writebacks or 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 as discussed previously. 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.

3.1.4 Tradeoff with large caches

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. The tradeoff between reading the entire set (normal mode), which is compatible with *Amoeba-Cache*, and only the relevant block (serial mode), is analysed herein. The cache size is varied from 2M—8M and associativity from 4(256B/set) — 32 (2048B/set). Under current technology constraints (Figure 3.3), 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



Baseline: Serial. ≤ 1 Normal is better. 32nm, ITRS LOP.

Figure 3.3: **Serial vs Normal mode cache** : This graph shows the ratio of Serial mode access versus Normal mode access. Thus values less than 1 indicate the serial mode access is more efficient. The cache configurations are grouped in terms of size and subgroups for the number of ways per cache set.

wordlines dominate (more than 2KB of words per set).

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

Table 3.2: Percentage of direct accesses with fast tags

The *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, thus the number of comparators and multiplexers required for lookup 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 [19] explored such an organization for supporting tags in multi-gigabyte caches.

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* similar to *Indirect Index Caches*(see § 2.7.3). The number of *Fast Tags* needed per set is proportional to the number of blocks in each set, which varies with the spatial locality in the application and the number of bytes per set (more details in Section 4.1). Three different cache configurations were studied (64K 256B/set, 1M 512B/set, and 2M 1024B/set) while varying the number of fast tags per set (see Table 3.2). With 8 tags/set (16KB overhead), the fast tags can filter 64—95% of the accesses in a 1MB cache and 55—75% of the accesses in a 2MB cache.

3.1.5 Hierarchical cache memory systems

The *Amoeba-Cache* can be implemented in a hierarchical cache memory system of *Inclusive* nature (see § 1.1). For the *Amoeba-Cache* however, 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 events that may invalidate multiple L1 blocks and b) L1 refills that may need data from multiple blocks at the L2. For both cases, the *Amoeba-Cache* needs 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, the *Amoeba-Cache* uses the step-by-step process outlined in § 2.5 (Stage 1 and Stage 2 in Fig 2.6). 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.

3.1.6 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) that is the granularity of write invalidations? The key insight that ensures compatibility with a conventional fixed-granularity coherence protocol is that an *Amoeba-Block* always lies within an aligned RMAX byte region (see § 2.1). 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 § 2.1). 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. An implementation and detailed evaluation of the coherence protocol is part of future work.

3.2 Simulation Infrastructure

The simulation infrastructure used to evaluate the performance of the *Amoeba-Cache* is described in this section. The *Amoeba-Cache* was implemented using the Wisconsin Multifacet Group's GEMS[21] system simulator infrastructure. To focus on the performance of the memory hierarchy in the studies, an in-order CPU model was used where each non-memory instruction was assumed to have a latency of 1 cycle. The SIMICS frontend was replaced with trace driven frontend. Application binaries were instrumented with PIN[20] to collect information about the memory accesses being made by the application. The address of an access, the type of access, the current instruction count, the contents of the program counter and the size of the memory access were logged in a compressed format.

3.2.1 GEMS-Ruby

Ruby is a component of the GEMS framweork which implements a detailed simulation model for the memory system. It models inclusive/exclusive cache hierarchies with various replacement policies, coherence protocol implementations, interconnection networks, DMA and memory controllers, various sequencers that initiate memory requests and handle responses. The models are modular, flexible and highly configurable. It implements a domain specific language called SLICC (Specification Language for Implementing Cache Coherence) that is used for specifying the cache controller finite state machine. SLICC imposes constraints on the types state machines which can be specified. Apart from protocol specification, SLICC also combines the interaction between the network components with cache memories. Ruby was suitabley modified to support variable granularity accesses and the *Amoeba-Cache* protocol was designed to work with it.

Chapter 4

Evaluation

4.1 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$\leftrightarrow$L2 bandwidth by 46% and L2$\leftrightarrow$Memory bandwidth by 38%.

In this section, the bandwidth and miss rate properties of an *Amoeba-Cache* are compared against a conventional cache. A *Fixed* cache represents a conventional cache which allocates a fixed granularity cache block on a refill request. The accuracy of the spatial pattern predictor is an important factor which governs the accuracy of the *Amoeba-Cache* and is evaluated separately. For the results presented in this section, cache line utilisation statistics, gathered from a prior run of the application on a conventional cache, are used to drive the predictor. This isolates the benefits of the *Amoeba-Cache* from the potentially changing accuracy of the spatial pattern predictor across different cache geometries. This also ensures that the spatial granularity predictions can be replayed across multiple simulation runs. To ensure equivalent data storage space, the *Amoeba-Cache* size is set 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.

Chapter 5

Conclusion

5.1 Summary

5.2 Future Work

Bibliography

[1] A. R. Alameldeen. *Using Compression to Improve Chip Multiprocessor Performance*. PhD thesis, University of Wisconsin, Madison, 2006.

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[4] Chi F. Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. Accurate and complexity-effective spatial pattern prediction. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 276–, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[6] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Intel Corporation. Intel embedded pentium processor family dev. manual, 1998.

[8] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. Cpu db: recording microprocessor history. *Commun. ACM*, 55(4):55–63, April 2012.

[9] Czarek Dubnicki and Thomas J. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 170–180, New York, NY, USA, 1992. ACM.

[10] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th annual international symposium on Computer architecture - ISCA 00*, volume 28, page 107116. ACM Press, Jun 2000.

[11] Erik G. Hallnor and Steven K. Reinhardt. A compressed memory hierarchy using an indirect index cache. Technical report, in Proc. 3rd workshop on Memory performance issues, 2004.

[12] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 364–373, New York, NY, USA, 1990. ACM.

[13] K. Kedzierski, M. Moreto, F.J. Cazorla, and M. Valero. Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[14] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. *SIGARCH Comput. Archit. News*, 26(3):357–368, April 1998.

[15] Chunrong Lai and Shih-Lien Lu. Efficient victim mechanism on sector cache organization. In Pen-Chung Yew and Jingling Xue, editors, *Advances in Computer Systems Architecture*, volume 3189 of *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin / Heidelberg, 2004.

[16] J. S. Liptay. Structural aspects of the system/360 model 85: Ii the cache. *IBM Syst. J.*, 7(1):15–21, March 1968.

[17] Tongping Liu and Emery D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 3–18, New York, NY, USA, 2011. ACM.

[18] Diego R. Llanos. Tpcc-uva: an open-source tpc-c implementation for global performance measurement of computer systems. *SIGMOD Rec.*, 35(4):6–15, December 2006.

[19] Gabriel Loh and Mark D. Hill. Supporting very large dram caches with compound-access scheduling and missmap. *IEEE Micro*, 32(3):70–78, May 2012.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of*

the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[21] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[22] Sun Microsystems. Opensparc t1 processor megacell specification, 2007.

[23] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Ajit Karthik Mylavarapu. Patent 20100049912 : Data cache way prediction, February 2010.

[25] P. Pujara and A. Aggarwal. Increasing the cache efficiency by eliminating noise. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 145 – 154, feb. 2006.

[26] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[27] Moinuddin K. Qureshi, M. Aater Suleman, and Yale N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 250–259, Washington, DC, USA, 2007. IEEE Computer Society.

[28] J .B. Rothman and A.J. Smith. *Sector cache design and performance*, page 124133. IEEE Comput. Soc, 2000.

[29] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio. In *Proceedings of 21 International Symposium on Computer Architecture*, page 384393. IEEE Comput. Soc. Press.

[30] André Seznec. Interleaved sectored caches: reconciling low tag volume and low miss ratio. Rapport de recherche RR-2084, INRIA, 1993.

[31] R.B. Tremaine, T.B. Smith, M. Wazlowski, D. Har, Kwok-Ken Mak, and S. Arramreddy. Pinnacle: Ibm mxt in a memory controller chip. *Micro, IEEE*, 21(2):56 –68, mar/apr 2001.

[32] Alexander V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. Adapting cache line size to application behavior. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 145–154, New York, NY, USA, 1999. ACM.

[33] Matthew A. Watkins, Sally A. Mckee, and Lambert Schaelicke. Revisiting cache block superloading. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 339–354, Berlin, Heidelberg, 2009. Springer-Verlag.

[34] Doe Hyun Yoon, Min Kyu Jeong, and Mattan Erez. Adaptive granularity memory systems. In *Proceeding of the 38th annual international symposium on Computer architecture - ISCA 11*, volume 39, page 295. ACM Press, Jun 2011.

[35] Doe Hyun Yoon, Min Kyu Jeong, Michael B. Sullivan, and Mattan Erez. The dynamic granularity memory system. In *Proceedings of the International Symposium on Computer Architecture (ISCA12)*, June 2012.

[36] Meng-Bing Yu, Era K. Nangia, Michael Ni, and Vidya Rajagopalan. Patent 7594079 : Data cache virtual hint way prediction, and applications thereof, September 2009.