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

SIMON FRASER UNIVERSITY
Fall 2012

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: _____

Partial Copyright Licence

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 [8].

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.

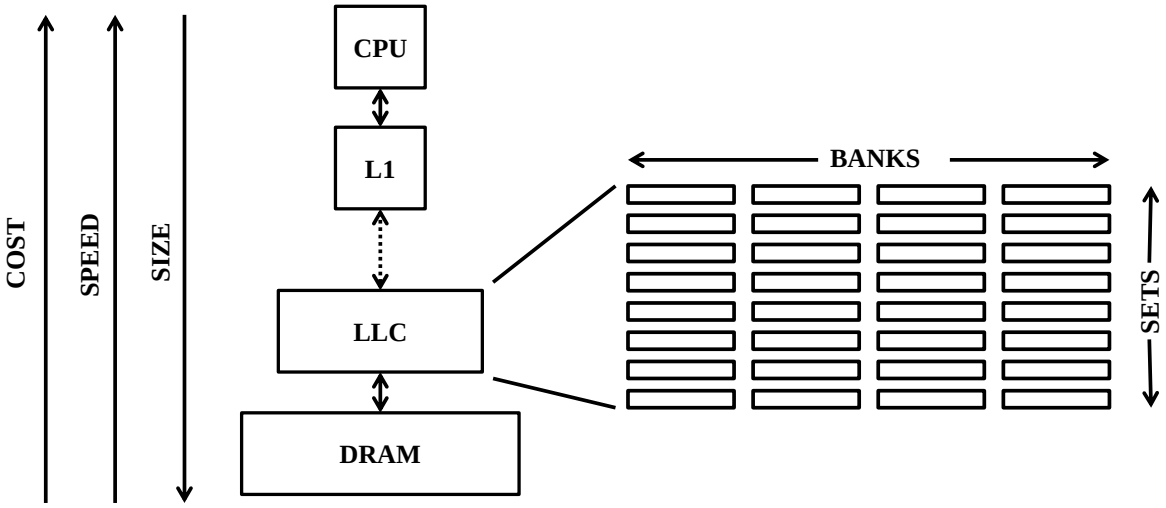


Figure 1.1: **Canonical Memory Hierarchy** – Moving down through the hierarchy, away from the processor, the levels are larger but slower. At each level the storage may be monolithic or sub divided in to "banks" for lower indexing overhead.

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.

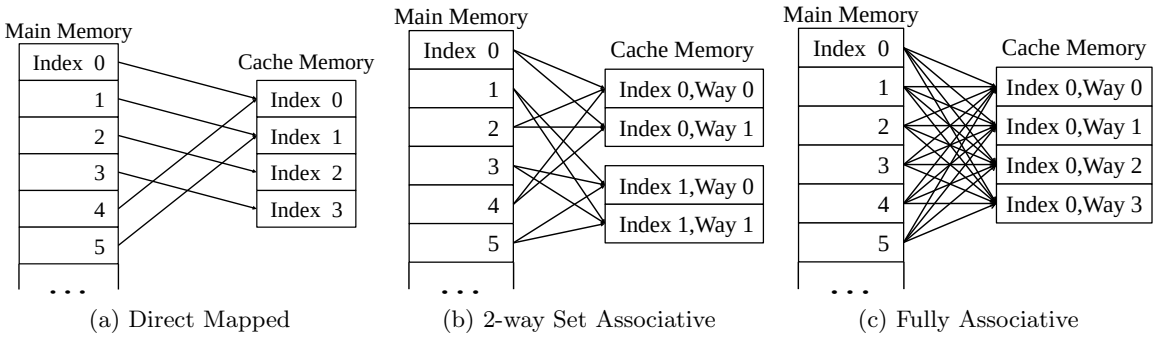
Caches are designed to take advantage of reuse of data by speeding up subsequent access to the same datum. They also speed up accesses to nearby data which may be fetched into the cache depending on its operating policy. The different types of locality which caches try to exploit can be enumerated as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

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.



(a) Direct Mapped (b) 2-way Set Associative (c) Fully Associative

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 [1], 7 from SPEC2006, 2 from SPEC2000, 3 Java workloads from DaCapo [2], 3 commercial workloads (Apache, SpecJBB2005, and TPC-C [10]), 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

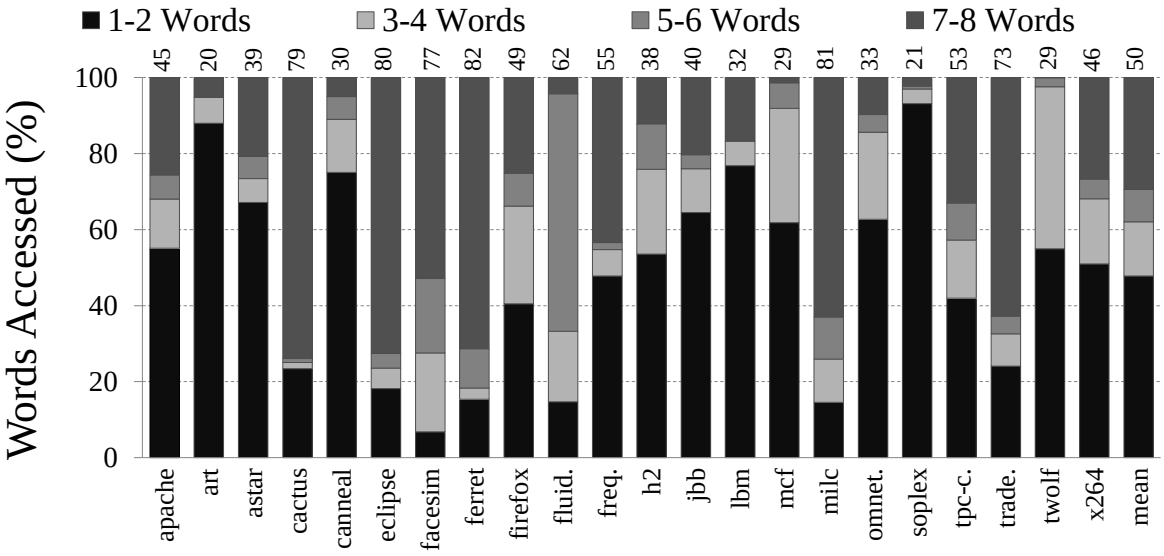


Figure 1.3: Distribution of words touched in a cache block. Avg. utilization is on top. (Config: 64K, 4 way, 64-byte block.)

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.

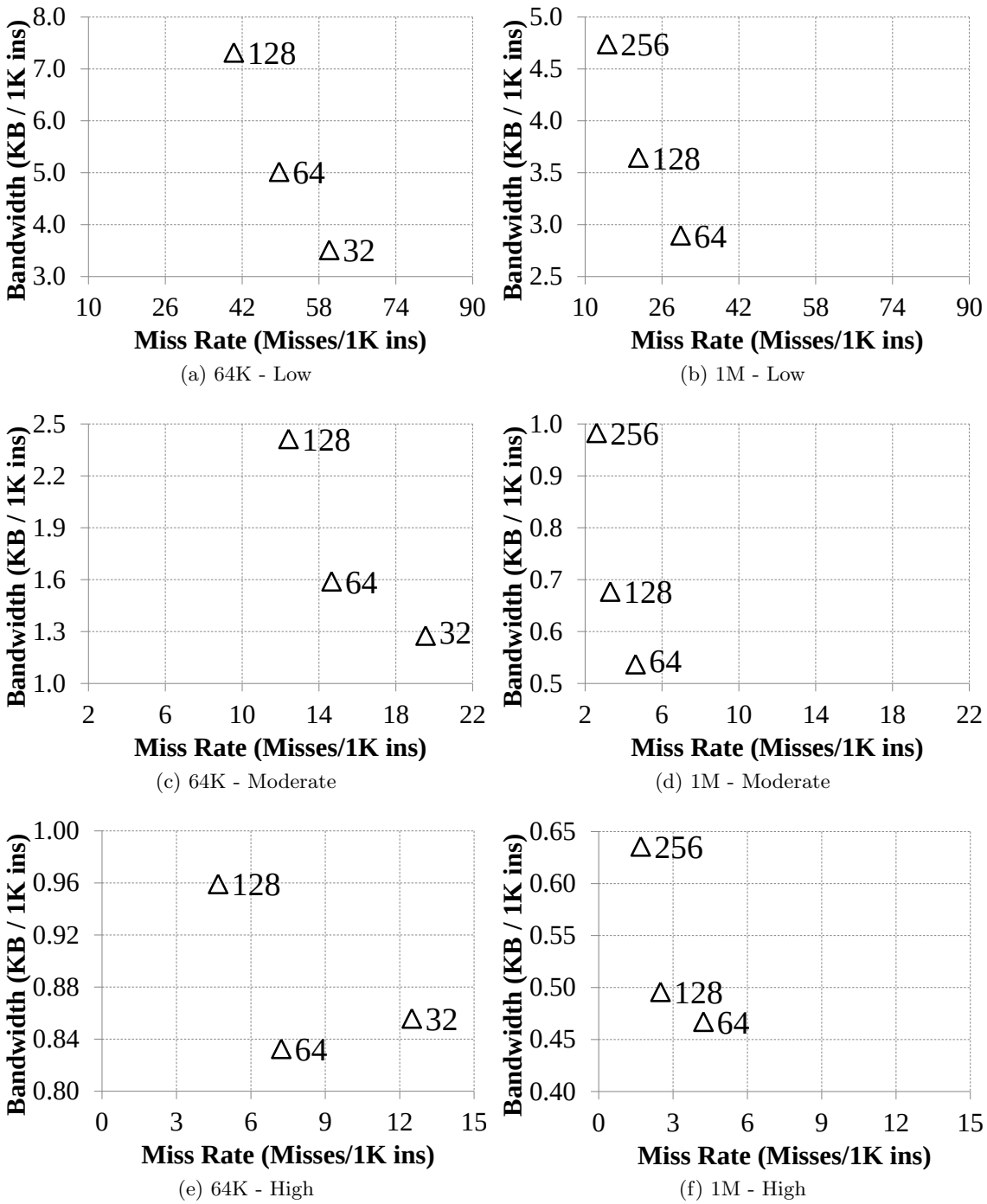
2. Incorrect assumptions about hardware : Hardware conscious code which attempts to optimise for cache behaviour based on assumptions should be ported carefully. We found **streamcluster** of the PARSEC application suite, by default, attempt to optimise cache behaviour for 32 byte cache line sizes. This finding was also reported by Liu and Berger[9].

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.

```
1  /* blackscholes.c:354 */
2  for (i=0; i<numOptions; i++)
3  {
4      otype[i]      = (data[i].OptionType == 'P') ? 1 : 0;
5      sptprice[i]   = data[i].s;
6      strike[i]     = data[i].strike;
7      rate[i]       = data[i].r;
8      volatility[i] = data[i].v;
9      otime[i]      = data[i].t;
10 }
```

Listing 1.1: Code snippet from the initialisation phase of **blackscholes** benchmark from the PARSEC 2.1 [1] 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 Section 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.

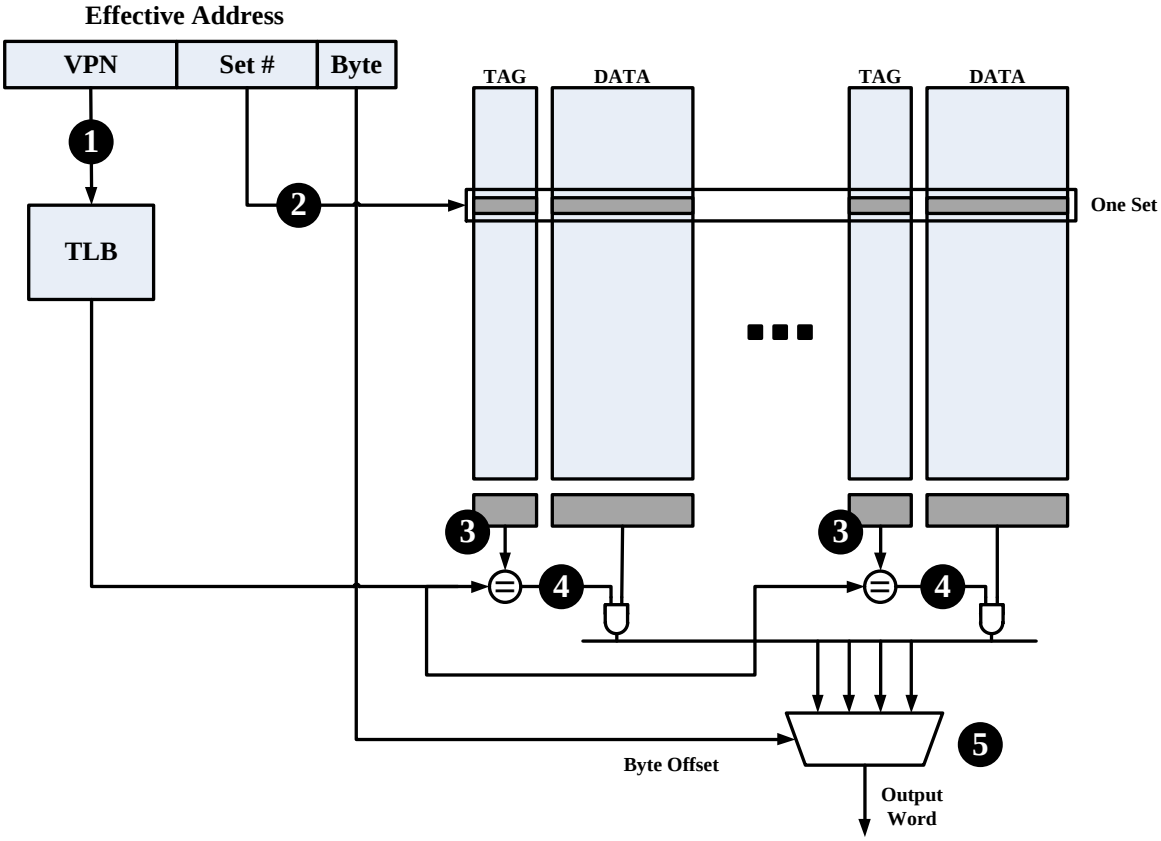


Figure 2.1: **Conventional N-Way Set-Associative Cache** ❶ The Virtual Page Number (VPN) is used to look up the entry in the Translation Lookaside Buffer (TLB) ❷ According to the number of sets in the cache, the following bits from the address are used to look up the corresponding set from the cache 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

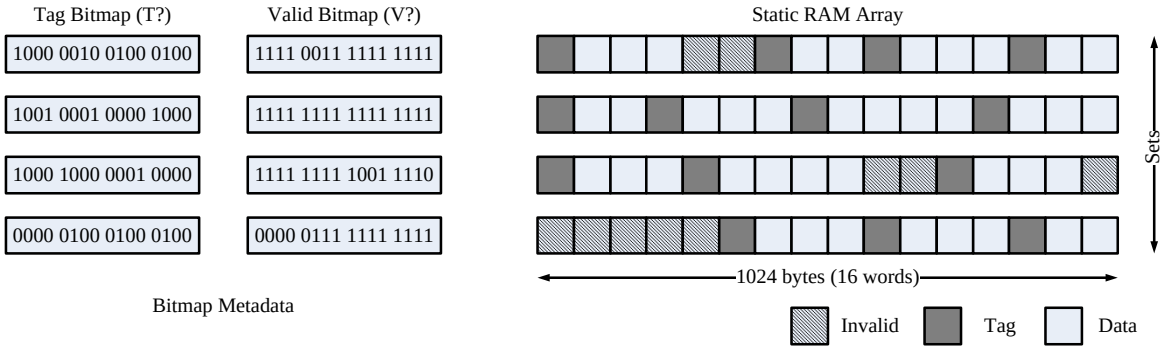
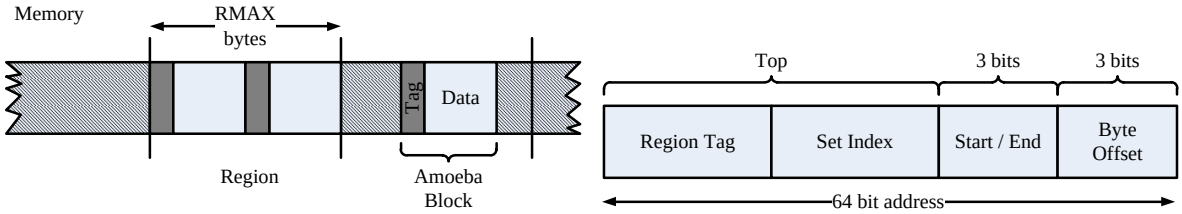


Figure 2.2: **Amoeba Cache Overview** The static RAM (SRAM) array where the tags and data are colocated is shown on the right. The `T? Bitmap` and the `V? Bitmap` for the *Amoeba-Cache* are shown on the left. Each block in the SRAM array represents 8 bytes (1 word). In this specific example, we show an *Amoeba-Cache* 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, we use a bitmap data structure (labeled `T?`

`Bitmap` in Fig 2.2). For each word in the set which is a tag, we set the corresponding bit in the `T? Bitmap`. We also decouple the conventional valid/invalid bits (typically associated with the tags) and organize them 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. We allocate 8 bytes (1 word) for each tag. In order to simplify cache lookups for *Amoeba-Block*s, we partition the address space 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. 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.

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 correct way from the set. 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*[12, 15], 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.

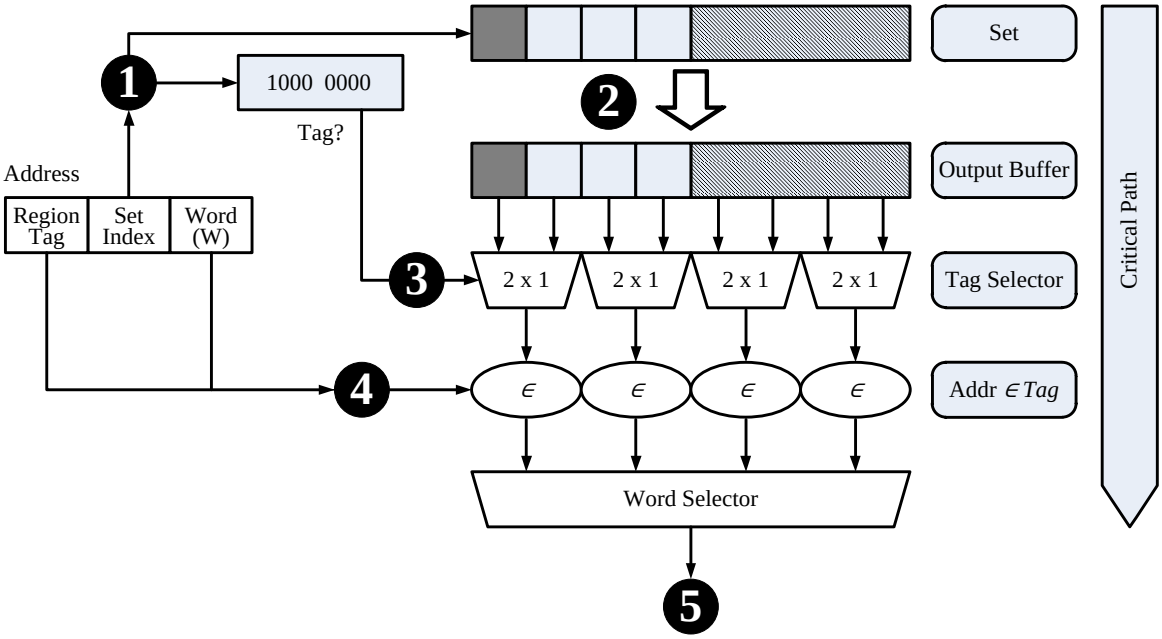


Figure 2.4: **Amoeba Cache Lookup** : The incoming effective address is segmented into the region tag, set index and word offset. ❶ The `Tag? Bitmap` is looked up to determine which words in the activated set are required for the tag comparison. Note that given the minimum size of a *Amoeba-Block* is 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 - $log_2 N_{sets}$ - $log_2 RMAX$ 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 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 Section 2.6), 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 `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 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? 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

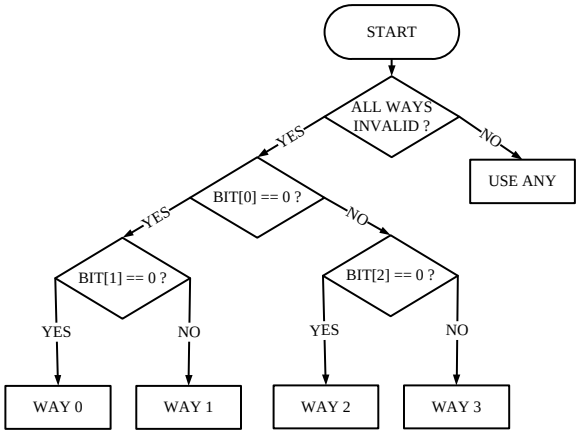
The replacement policy of a cache determines which blocks are evicted when new data is brought in and there is no room to store it. The replacement algorithm tries to make an optimal choice where it evicts a blocks which is not expected to be used in the near future. The most optimal choice possible 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[5].

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 [6, 11] 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 [14] that work with a fixed number 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*. 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.

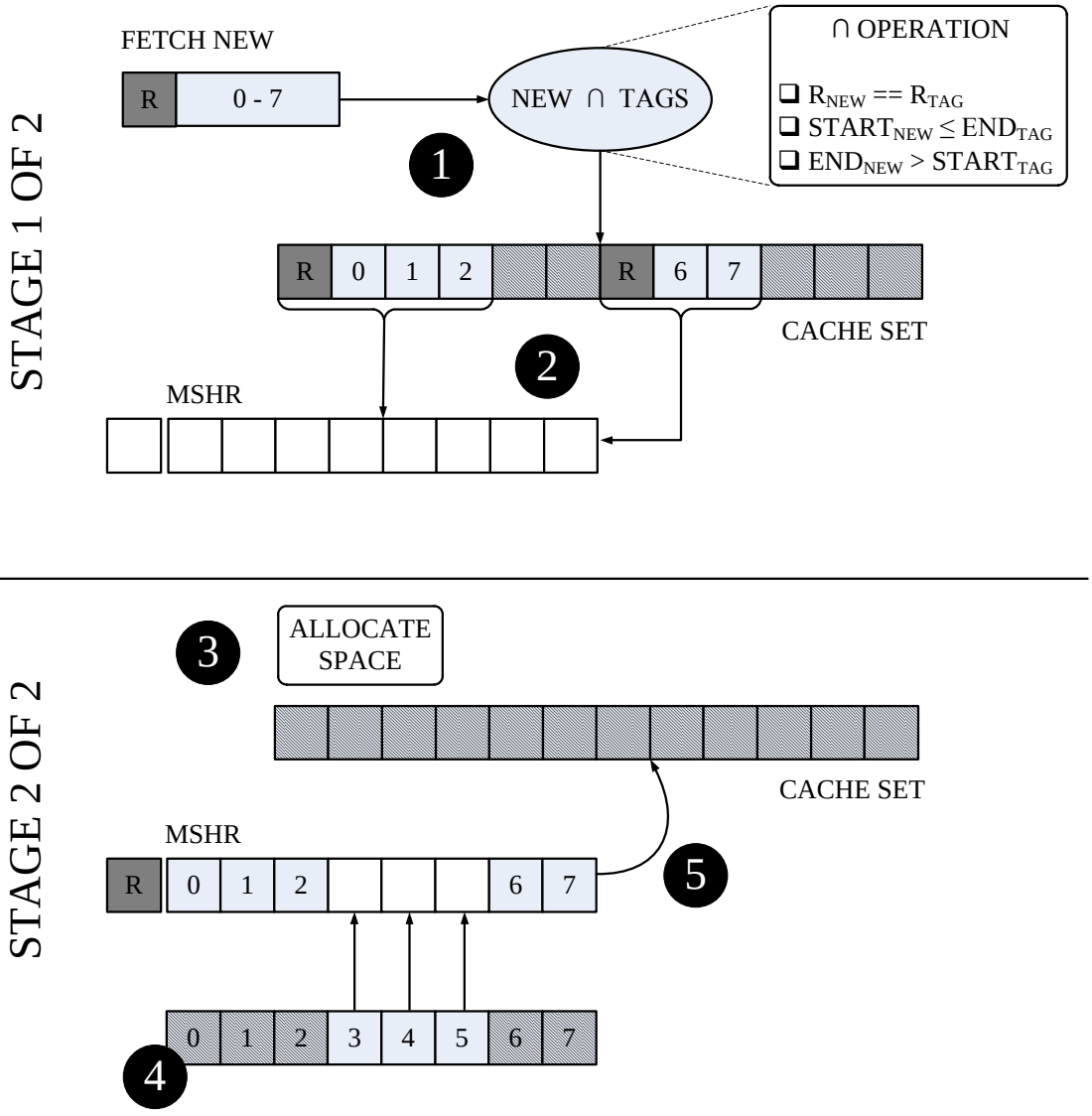


Figure 2.6: **Partial miss processing** : The partial miss processing is shown as 2 separate stages. Stage 1 deals with the identification and eviction (to MSHR) of all overlapping blocks. Stage 2 issues the miss, allocates space for the new *Amoeba-Block* and patches in the missing words from a lower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

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 ∩(in) operation returns false if there are no overlapping blocks. The ∩ 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

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 [4, 7, 13, 3]. 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 ses 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 4.3.

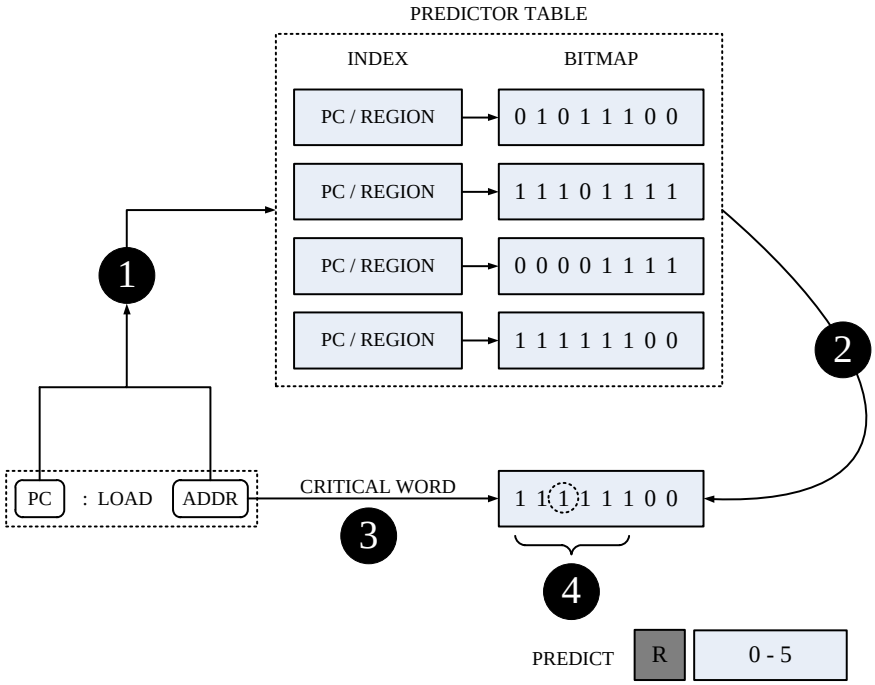


Figure 2.7: ***Amoeba-Cache* Predictor** :

2.7 Related Work

2.7.1 Line Distillation

2.7.2 Sector Caches

2.7.3 Indexed Indirect Caches

2.7.4 Cache Compression

Chapter 3

Hardware Complexity and Simulation

3.1 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.

3.2 Simulation Infrastructure

Chapter 4

Evaluation

4.1 Best Effort - Oracle

4.1.1 Miss Rate - Performance

4.1.2 Bandwidth - Energy

4.2 Comparative Study

4.3 Predictor Tradeoffs

4.4 Multi Core Shared Cache

Chapter 5

Conclusion

5.1 Summary

5.2 Future Work

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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