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 [3].

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.

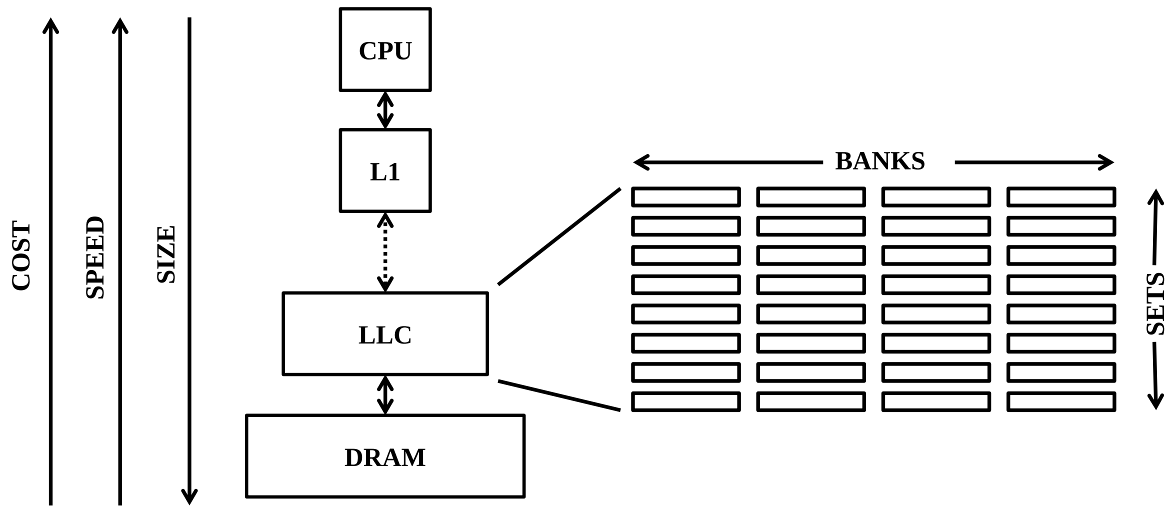


Figure 1.1: **Canonical Memory Hierarchy** – Moving down through the hierarchy, away from the processor, the levels are larger but slower. At each level the storage may be monolithic or sub divided in to "banks" for lower indexing overhead.

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.

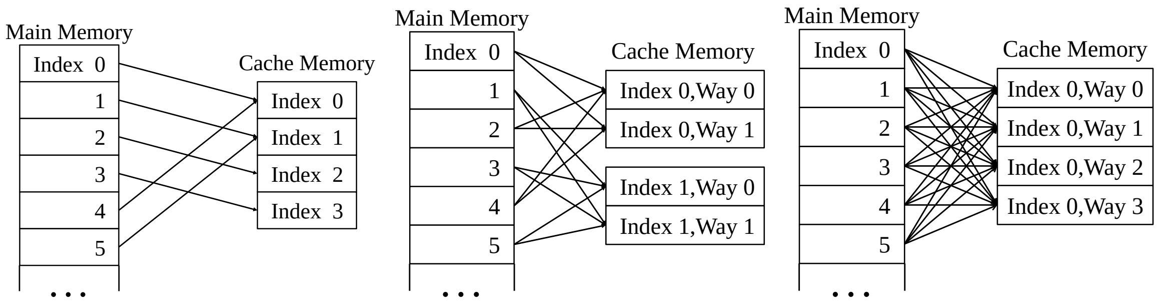
3. Non-Inclusive Caches : Also known as *Non-Exclusive* or *Accidentally Inclusive*, were used for a while in Intel architectures prior to the Intel P6. A lower level cache may or may not include a block cached at a higher level in the cache hierarchy.

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 (See section **??**). 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 [5]), 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; $\simeq$70% of the blocks in astar have 1-2 words accessed at the time of eviction, whereas $\simeq$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

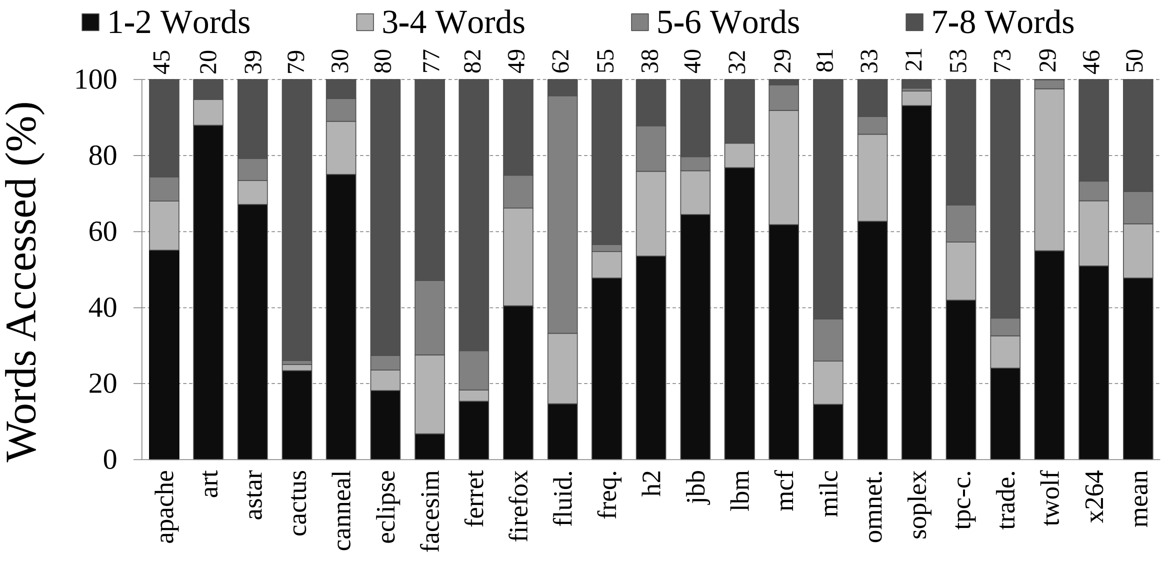


Figure 1.3: Distribution of words touched in a cache block. Avg. utilization is on top. (Config: 64K, 4 way, 64-byte block.)

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[4].

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 [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.

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.

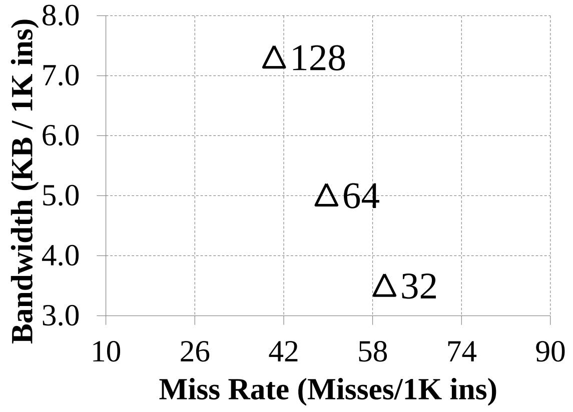
1.2.4 Need for adaptive cache blocks

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:

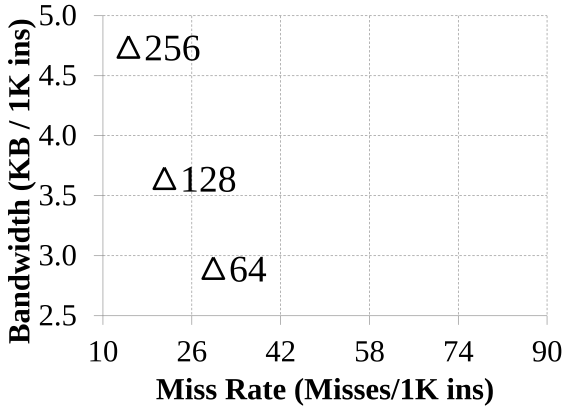
- Smaller cache lines improve utilization but tend to increase miss rate and potentially traffic for applications with good spatial locality, affecting the overall performance.
- Large cache lines pollute the cache space and interconnect with unused words for applications with poor spatial locality, 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

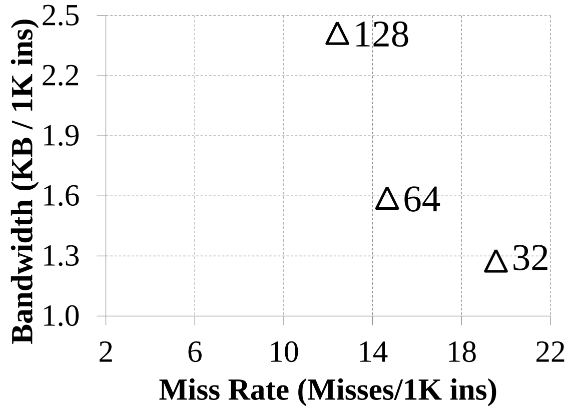
In Chapter **??**, we look more closely at the organisation and operations of a conventional cache and provide an overview of related work. Chapter 2 describes the Amoeba Cache architecture. The experimental setup and implementation of the simulator are described in Chapter 3. The experimental results of an exhaustive evaluation of the Amoeba-Cache is presented in Chapter 4. Conclusions and future work are outlined in Chapter 5.



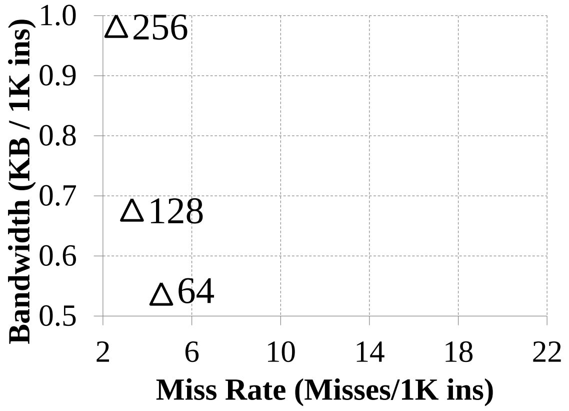
(a) 64K - Low



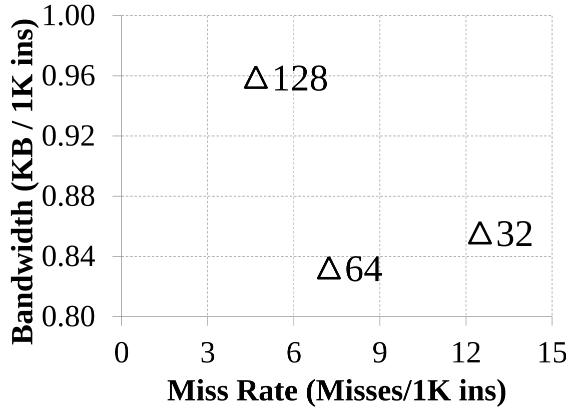
(b) 1M - Low



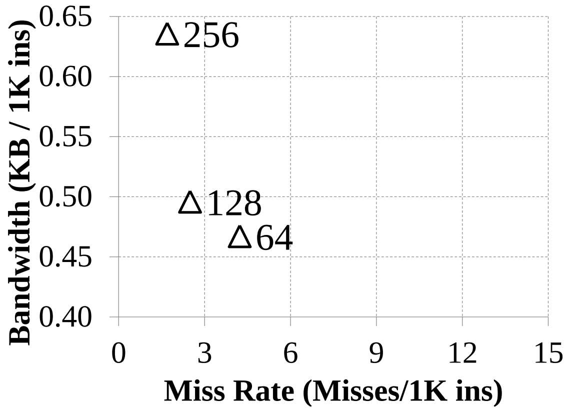
(c) 64K - Moderate



(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.

Chapter 2

Amoeba Cache Architecture

2.1 Fundamental Building Blocks

2.2 Cache Management

2.2.1 Standard Operations

2.2.2 Hardware Aids

2.2.3 Replacement Policy

2.3 Spatial Pattern Predictor

2.3.1 Oracle

2.3.2 Region based Approach

2.3.3 Program Counter based Approach

2.4 Hierarchical Amoeba Cache Architecture

2.5 On-chip Interconnection Network

2.6 Multi Core and Coherence

2.7 Related Work

2.7.1 Sector Caches

2.7.2 Cache Filtering

2.7.3 Spatial Access Pattern Prediction

2.7.4 Software based approaches

Chapter 3

Implementation

3.1 Application Traces

3.1.1 Intel Pin

3.1.2 Generating a memory access trace

3.1.3 Workload selection

3.2 GEMS Infrastructure

3.2.1 Introduction

3.2.2 Components

3.2.3 SLICC

3.2.4 Amoeba-Single Protocol

Chapter 4

Evaluation

4.1 Best Effort - Oracle

4.1.1 Miss Rate - Performance

4.1.2 Bandwidth - Energy

4.2 Amoeba Cache vs Other Approaches

4.2.1 Sector Caches

4.2.2 Sector Caches with Prefetching

4.2.3 Line Distillation

4.2.4 Multi Cache

4.3 A feasible online approach

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

[4] 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.

[5] 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.