Technical Report

UCAM-CL-TR-800
ISSN 1476-2986

Number 800

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Improving cache utilisation

James R. Srinivasan

June 2011

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 2011 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Jesus College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Microprocessors have long employed caches to help hide the increasing latency of accessing main memory. The vast majority of previous research has focussed on increasing cache hit rates to improve cache performance, while lately decreasing power consumption has become an equally important issue. This thesis examines the lifetime of cache lines in the memory hierarchy, considering whether they are **live** (will be referenced again before eviction) or **dead** (will not be referenced again before eviction). Using these two states, the **cache utilisation** (proportion of the cache which will be referenced again) can be calculated.

This thesis demonstrates that cache utilisation is relatively poor over a wide range of benchmarks and cache configurations. By focussing on techniques to improve cache utilisation, cache hit rates are increased while overall power consumption may also be decreased.

Key to improving cache utilisation is an accurate predictor of the state of a cache line. This thesis presents a variety of such predictors, mostly based upon the mature field of branch prediction, and compares them against previously proposed predictors. The most appropriate predictors are then demonstrated in two applications:

- Improving victim cache performance through filtering
- Reducing cache pollution during aggressive prefetching

These applications are primarily concerned with improving cache performance and are analysed using a detailed microprocessor simulator. Related applications, including decreasing power consumption, are also discussed, as are the applicability of these techniques to multiprogrammed and multiprocessor systems.

Acknowledgements

I would like to thank my supervisor, Simon Moore, for his advice, support and patience. Thanks also to my numerous friends and colleagues in the Computer Architecture Group.

Thanks to my previous employer, Microsoft Research, for a never-ending source of distractions and to my current employer, 2d3 Ltd, for their support and flexibility.

I would also like to thank my examiners, Prof Ian Watson and Dr Robert Mullins.

I would like to dedicate this dissertation to my father, without whom it would never have been started, and to my wife, Sarah, without whom it would never have been finished.

The work described in this dissertation was funded by the Cambridge MIT Institute.

Contents

1 Introduction

This dissertation demonstrates that cache utilisation is an important performance and power-related metric which current cache management techniques fail to adequately address by focusing simply on the ordering of cache line accesses, rather than the absolute and relative times of those accesses.

Having first defined cache utilisation and a variety of associated time-based metrics, efficient run-time predictors for these metrics are introduced and two specific techniques are evaluated to exploit these predictors which each improve overall cache and hence processor performance. Finally, a preliminary analysis of cache utilisation in multiprogramming and multiprocessor systems is presented, as well as a review of additional applications.

In this chapter, the overall background to caching is discussed and the contribution of the work described by this dissertation is highlighted. A brief overview of the remaining content is then provided together with a description of the terminology that will be used.

1.1 Motivation

Caching is a popular technique employed in a wide range of applications throughout computer systems in an attempt to hide the full cost of accessing some relatively slow device or connection by seeking a different capacity/speed tradeoff. While the penalty of accessing the device behind the cache is usually a time penalty due to constrained bandwidth or latency, it may equally well be some other penalty such as a financial cost or energy penalty. Caches are typically of smaller capacity but faster speed than the device which they are backing. By keeping suitable items in the cache, the full cost of accessing the backing device may be largely hidden.

Caches are found in a wide variety of applications including processor caches (hiding the latency of accessing main memory as well as other levels of cache), disk, database buffer and virtual memory caches (hiding the latency of a disk access) and web caches (hiding the latency and potential financial cost of a long-distance network access). In each of these examples, the potential benefit provided by the cache depends on ensuring that items which will be referenced in the near future are found in the cache, rather than incurring the full cost of an access to the device behind the cache. Ensuring the cache is populated with such useful items is of paramount importance, and often relies upon exploiting spatial and temporal locality, the properties that the same, or similar, items will be accessed again in the near future.

The exact techniques used to manage the cache vary widely according to the properties of both the device and the properties of the requests being made to the device. For

example, the time available to make a processor cache replacement decision is much shorter than that available to make a disk buffer cache replacement decision so different policies are used. While this dissertation concentrates on the processor cache, various techniques from other applications of caching are also briefly discussed and the techniques developed here may well be applicable for other caches.

With increasing processor clock frequencies, the increasing latency to access main memory has long been a significant obstacle to continually improving overall system performance. Though the rate of clock frequency increase has slowed lately, there is still a substantial gap (the so-called **memory gap**), while other microarchitectural innovations to improve performance such as multiple cores and simultaneous multithreading (SMT) impose yet more demands on the already pressured memory subsystem.

More recently, power consumption has become a topic of concern to all processor architects, from embedded devices, through mainstream desktop computers to blade and other high-performance server applications. As the size of processor caches is naïvely increased to improve performance, static power consumption together with yield and reliability concerns place constraints on the economically achievable performance. While this dissertation concentrates on performance-improving applications, a brief discussion of analogous power-decreasing applications is also presented.

1.2 Contribution

The main contributions of the work described in this dissertation are as follows:

- A detailed characterisation of the distribution and scaling of cache utilisation (and other related metrics) across a wide variety of benchmarks and cache configurations
- Novel, accurate and practical predictors for metrics related to cache utilisation
- Two applications of such predictors which improve overall processor performance, evaluated against other comparative techniques across a wide variety of benchmarks
- A preliminary analysis of cache utilisation in multiprogrammed and multiprocessor environments

1.3 Outline

The remainder of this dissertation is organised as follows:

- **Chapter 2** provides a thorough background to processor caches, reviewing previous work which relates to and motivates this dissertation.

- **Chapter 3** starts by demonstrating that processor caches have a significant impact on overall system performance. The improvements made by traditional cache scaling are then examined. Various metrics relating to the liveness of a cache lines are defined, which rely upon distinguishing live cache lines (those which will be accessed again prior to eviction) from dead cache lines (those which will not be accessed again prior to eviction). The distribution of these metrics between different caches, cache configurations and benchmarks is detailed and discussed, as is a metric for cache utilisation. Scaling of cache utilisation with traditional cache parameters is also examined.

- **Chapter 4** examines the predictability of the previously defined cache line lifetime metrics. Having established that the task of predicting such metrics is feasible, a variety of existing and novel predictors are introduced.

- **Chapter 5** details two performance-improving applications for cache line lifetime predictors and evaluates them against other comparative techniques. The first application is a filtered victim cache, in which space in the victim cache is selectively allocated based upon the predicted liveness of the cache line being evicted. The second application decouples prefetch target selection from prefetch victim selection, in order to reduce cache pollution under aggressive prefetching.

- **Chapter 6** concludes this dissertation by reviewing the major findings and results as well as suggesting areas for future research, particularly cache line lifetime predictor applications associated with reducing power consumption. In addition, a preliminary analysis of cache utilisation in multiprogrammed and multiprocessor environments is presented.

1.4 Terminology

Some terminology relating to caches has already been used in this dissertation. Most terms will be defined when they are used but to avoid any confusion, a small number of key terms are defined here and subsequent uses of these terms in this dissertation shall be with reference to these definitions.

A **cache line** is the smallest item of data which may be stored and transferred as a single unit within the cache. For the purposes of this dissertation, **cache block** is another term widely used for the same entity but only the former phrase will be used unless it refers specifically to previous work.

A **cache set** of a set-associative cache is the collection of cache lines to which a single line may map. Thus a n-way set-associative cache of size m contains $\frac{m}{n}$ sets.

The informal term **average** refers to the **arithmetic mean** when dealing with absolute values, and the **geometric mean** when dealing with proportional or relative values.

Finally, when using abbreviations, the units of a quantity measured in bytes is abbreviated to **B** whereas the units of a quantity measured in bits is abbreviated to **b**. For example, 8 kb and 1 kB both denote the same quantity. In keeping with convention, bytes are largely used in this dissertation other than when discussing memory density which is more often measured in terms of bits.

Background 2

2.1 Overview

The aim of this chapter is to provide a summary of cache designs as implemented in microprocessors today, as well as focusing on previous relevant research and their likely significance given current and anticipated technology trends.

This chapter begins by introducing the familiar "memory wall" problem, and puts it into context with regard to current and anticipated technology trends. The basics of cache design are then briefly reviewed through examining the parameters describing traditional caches, and their influence on cache performance is considered. Methods to quantify both cache and overall system performance are then discussed. Finally, a wide variety of previous work associated with both increasing cache performance and decreasing cache power consumption is reviewed and discussed. Due to the expansive nature of previous cache-related research, only work directly relevant to that described in this dissertation is considered.

2.2 The "Memory Wall" Problem

The **memory wall** refers to the increasing gap between the speed of logic and the speed of main memory, first spelt out by Wulf and McKee in 1995 [WM95]. However, this gap has existed for a long time. In the early days of computing main memories, constructed using mercury delay lines, were much slower than the technology used to construct logic. Later memory technologies such as cathode ray tubes and core memories improved the situation somewhat but there was still a significant gap. With the introduction of semiconductor memories in the early 1970s, memory performance caught up substantially but still lagged logic performance. Since then, the focus on developing memory technology has been on increasing density rather than decreasing latency (indeed the two are somewhat contradictory), and the gap between main memory speed and processor speed has been ever-widening. In addition, recent trends and microarchitectural innovations such as multithreading and multiple cores have only increased the pressure on the memory subsystem.

2.2.1 Moore's Law

In 1965 Gordon Moore observed that the complexity[1] of an integrated circuit of minimum cost had approximately doubled over the previous five years and Moore predicted that such growth was attainable for at least the next ten years [Moo65]. Now

[1] In terms of the number of components

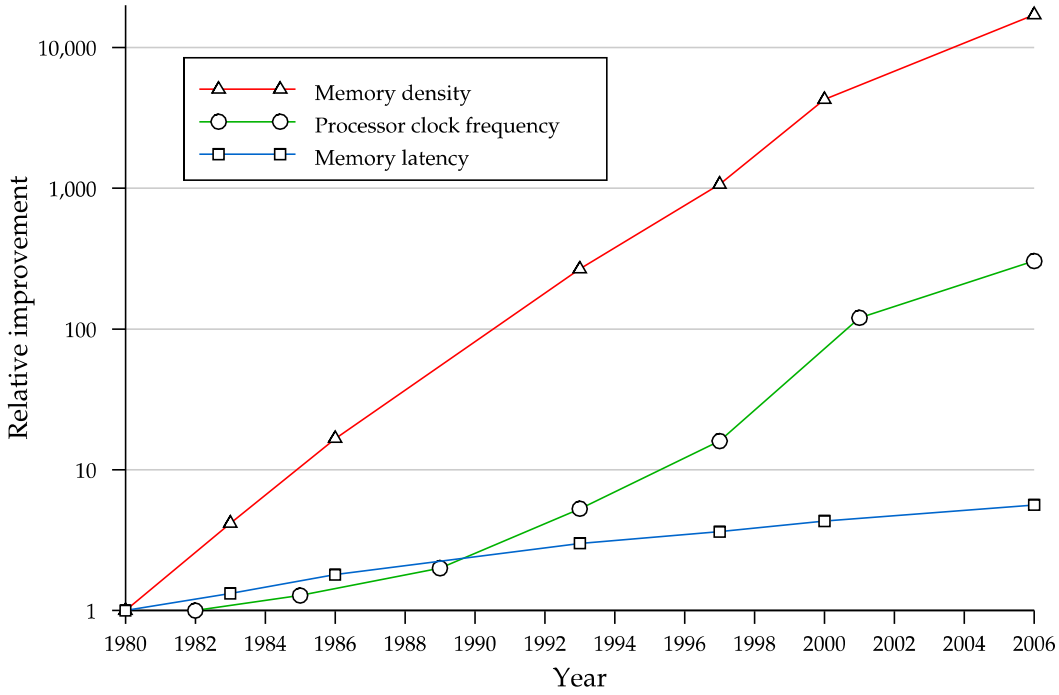

Figure 2.1: Relative improvements of memory density, processor clock frequency and memory latency, 1980—2006

termed **Moore's Law**, it is most often formulated in terms of the number of transistors on a integrated circuit doubling every 18 months, but has also been applied more generally to any exponential growth when using a variety of other metrics throughout the computer industry, including processor clock frequencies, hard disk capacity, DRAM density and network bandwidth. Figure 2.1 shows the improvements in memory density, processor clock frequency and memory latency from 1980 to 2006 using the data detailed in Appendix C. Each metric is normalised relative to the earliest value available for that metric. Note that the y-axis of the plot uses a logarithmic scale.

The first observation from Figure 2.1 is that memory density has grown enormously since 1980, by a factor of almost 20,000. The memory density line is approximately straight, indicating exponential growth and the steep slope of the line suggests a large base. Processor clock frequency has also increased significantly over the period, by a factor of 300. This line is less straight than the memory density line, but is still indicative of exponential growth. The kink evident in the early 2000s may represent marketing rather than technology trends since processor clock frequency was viewed by many consumers as being synonymous with overall processor performance, a fallacy known as the "megahertz myth". Finally, Figure 2.1 shows that memory latency has improved, but by considerably less than either memory density or processor clock frequency, a mere factor of five over the entire twenty-six year period. The line is also slightly curved, hinting at less than exponential growth. The divergent behaviour of the processor clock frequency line compared to the memory latency line signifies that the gap between the two has grown exceedingly rapidly and is only accentuated by the logarithmic scale.

2.3 Technology Trends

A variety of current and anticipated technology trends are now examined and their impact on cache design is investigated. Though presented separately, these trends are by no means independent and their interactions are also discussed.

2.3.1 Frequency Scaling

As shown by Figure 2.1, processor clock frequencies have increased rapidly over the past twenty-six years. This has been facilitated by both improvements in fabrication technologies supplied by smaller transistors with less capacitance courtesy of Moore's Law, as well as microarchitectural innovations such as superpipelining coupled with accurate branch prediction. However, despite decreasing capacitance and voltage scaling, dynamic power consumption has proved to impose a hard limit on the performance benefits that may be achieved through frequency scaling alone, and it has been observed that the rate of processor clock frequency increase has slowed significantly in recent years.

2.3.2 Increased Power Consumption

Figure 2.2 shows the improvements (or in the case of power consumption, the increase) of the number of transistors per die, power consumption, process geometry and die area for the processors detailed in Appendix C from 1982 to 2006. Again, each metric is normalised relative to the earliest value available and a logarithmic scale is used for the y-axis.

Over this time period the number of transistors has grown substantially, by a factor of almost 10,000, and the line is fairly straight, indicating exponential growth as in the original formulation of Moore's Law. Both die area and process geometries have also improved, but much more unevenly and by much smaller factors, approximately 10 and 200 respectively over the entire period. Both are heavily constrained by other factors including yield, reliability, and non-recurring expenditure. Note that the number of transistors is proportional to the product of the die area and the square of the process geometry.

As was mentioned in Section 2.3.1, power consumption has also increased significantly, by a factor of over 100, and is now a primary, if not *the* primary, concern of processor architects. Overall power consumption is made up of **dynamic** and **static** (also known as leakage) components. Dynamic power consumption, the energy expended by transistors as they switch, was previously the dominant factor, increasing due to increased clock frequencies as well as other factors such as the disparity between wire and transistor scaling. Many techniques exist to manage dynamic power consumption, two of the most popular being clock gating and dynamic voltage scaling. Recently, static power consumption, the energy continually expended by transistors, has become equally significant, due to smaller process geometries which have inherently more static leakage current. Decreasing the supply voltage, a common technique to decrease dynamic power consumption, has the side-effect of increasing static power

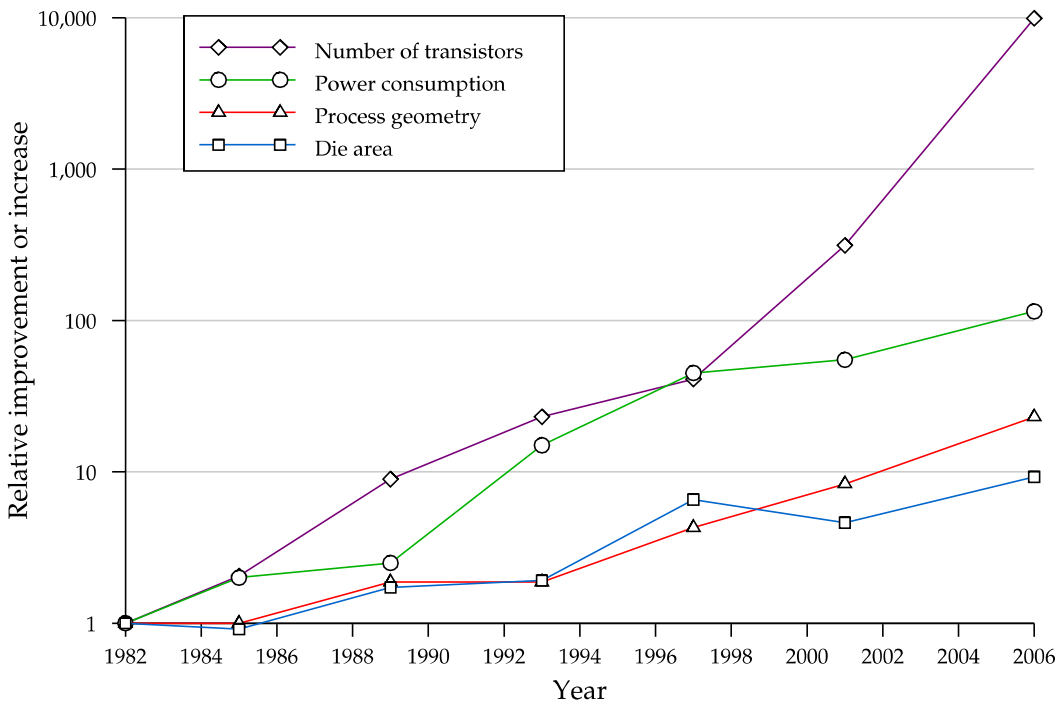

Figure 2.2: Relative improvements or increases of the number of transistors, power consumption, process geometry and die area, 1982—2006

consumption, hence a suitable trade-off is required. Both forms of power consumption are increased due to larger die areas, another trend shown in Figure 2.2.

2.3.3 Increased Cache Size

In an attempt to hide the widening gap between main memory access times and processor frequencies previously discussed, caches have been widely deployed in both high performance and increasingly embedded processors. There is an inherent tradeoff between the size of the cache (larger caches have higher hit rates hence better overall performance) and latency (larger caches tend to have higher latencies which can decrease overall performance). It is interesting to note that caching, when effective, can reduce overall power consumption by reducing execution time as well as reducing the number of off-chip accesses which are particularly costly in embedded platforms.

Figure 2.3 shows the increase in cache size and latency for three levels of cache based on the data detailed in Appendix C. Again, each metric is normalised relative to the earliest value available but unlike Figures 2.1 and 2.2, a linear scale is used on the y-axis.

Overall, cache sizes have not increased and cache latencies, measured in terms of processor cycles, have not improved anywhere near as quickly as the other metrics previously covered. Cache size growth towards the earlier dates is slow, or indeed negative in the case of the first-level cache from 1984 to 1989. This is mainly due to the earliest caches being off-chip which enabled a higher capacity, but with a correspondingly

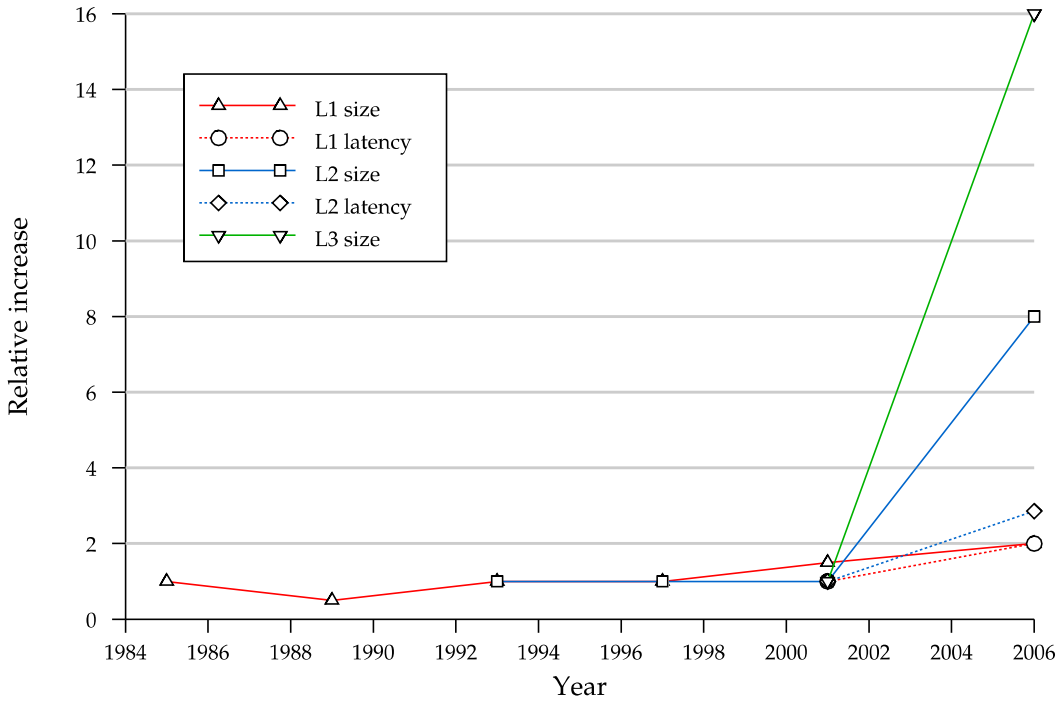

Figure 2.3: Relative increases of cache size and latency, 1984—2006

much higher latency. As the number of transistors available on the die grew, caches were moved on-chip where they had much smaller latencies. The number of cache levels is also shown to increase over time, from a single level in 1985 to three in 2006. As will be discussed later, multiple levels of cache are necessary to provide multiple points in the design space, trading off latency against size. Third-level caches, a relatively recently implemented technique, illustrated by only two datapoints are growing rapidly, again due to the rapid increase in the number of available transistors per die and today the cache accounts for the majority of the transistor budget, as exemplified by Figure 2.4, illustrating the floorplan of a modern Intel Itanium microprocessor.

Figure 2.3 also show cache latencies increasing over time as the requirement to achieve high cache hit rates outweighs that to minimise cache latency due to the implementation of microarchitectural latency-tolerating techniques such as out-of-order execution. However, not only is absolute cache latency increasing, but the variability of cache latency due to increasing wire delays is also becoming significant, a topic which will be further discussed later in Section 6.1.1.

2.3.4 Decreased Clock Cycles per Instruction

Original **Complex Instruction Set Computer** (CISC) microprocessors may take many cycles to execute each instruction. With the advent of **Reduced Instruction Set Computer** (RISC) architectures, the introduction of **pipelining** and **superpipelining**, together with later trends such as **superscalar execution**, the number of cycles taken to execute an instruction (CPI) has decreased rapidly such that multiple instructions

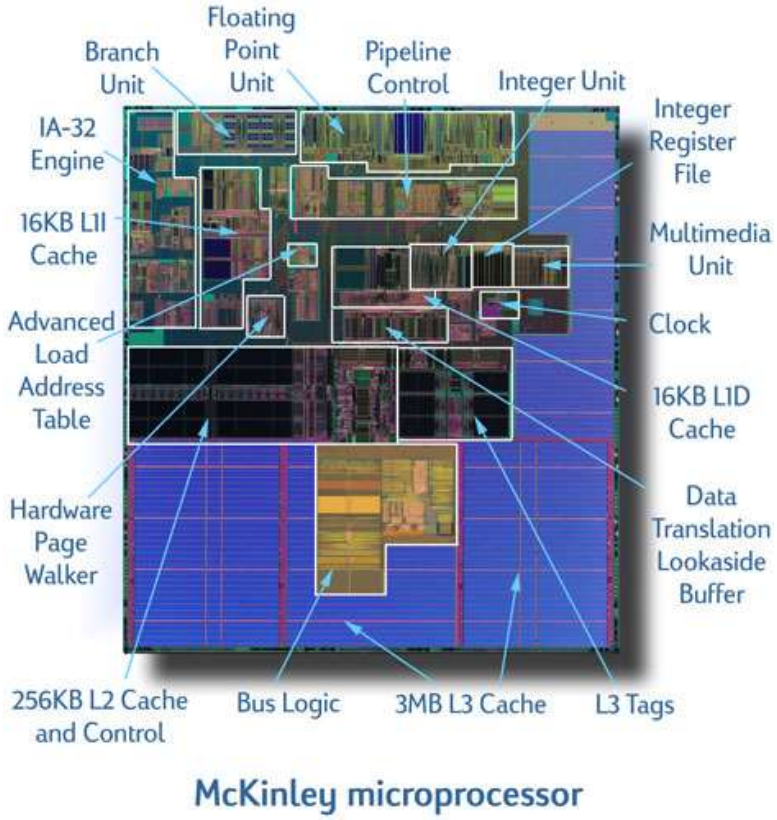

Figure 2.4: Floorplan of the Intel Itanium microprocessor. Adapted from [Lab03]

may be executed in each cycle. The number of instructions executed per cycle (IPC, the inverse of CPI) is now far more commonly quoted. Thus for each cycle that the processor is stalled whilst waiting for data from the memory subsystem, not only are an increasing number of clock cycles being wasted due to the widening difference in speed between logic and memory, but the number of instructions which *could* have been executed in this time is growing as well, making cache misses even more expensive.

2.3.5 Increased Number of Cores

Following the end of frequency scaling to improve performance, high-level parallelism has been exploited by increasing the number of processor cores on a single die, a phenomenon known as **multicore**, a trend which many believe will continue into **many-core** architectures [Par06]. Though hampered by immature software tools to express such parallelism, well-designed applications demonstrate almost linear scaling with the number of cores. In the general-purpose microprocessor marketplace, two or four cores are common. In the embedded, scientific and technical computing marketplaces, where parallelisation can be aided using detailed application knowledge, commercial processors with 64 cores are available [BEA+08], and research processors with 80 cores [VHR+07].

Continuing the theme of increased parallelism, hardware to explicitly support multiple threads, known as **symmetric multithreading** (SMT) or **chip multithreading** (CMT)

further increase the number of concurrent tasks being performed by the processor as a whole. Various interconnection networks have been implemented but currently most follow the traditional multiprocessor model of small private caches for each core followed by larger caches shared between multiple cores and finally main memory interfaces to supply the shared cache. The interconnection requirements of such a bus-based approach are clearly not scalable, leading to the development of **on-chip networks** to more efficiently move data around. Returning to the overall technology trend, it is conceivable that rather than cache sizes and the number of cache levels continuing to scale, the amount of cache per core will remain reasonably constant and the number of cores (plus cache) will simply increase. Thus it would be desirable to make more efficient use of the limited cache resources available to each core.

The specific challenges introduced by multicore architectures include an increase in on-chip memory traffic to support cache coherency, as well as increasing off-chip bandwidth to support multiple concurrent threads of execution. This latter trend is particularly problematic, since the amount of off-chip bandwidth available is dependent on the pin count of the package, which is predicted to grow far slower than the number of cores within the package [RKB+09].

2.3.6 Summary

The primary technology trend of those previously discussed is increasing power consumption, which drives several of the other technology trends. For high-performance microprocessors, the power consumption limit is imposed by the ability to effectively and economically cool the device, while for embedded microprocessors other factors such as battery life may also be relevant. The prevalence of frequency scaling has diminished substantially in recent years, with overall system performance improved by other trends such as increasing cache size, or increasing the number of cores.

2.4 Cache Basics

Having discussed the various technology trends which necessitate and influence caches, this section briefly reviews traditional cache designs and examines the various cache parameters which influence performance.

In its simplest sense, a cache is a smaller but faster memory which is placed between a processor and its main memory and which seeks a different tradeoff in the design space between capacity and latency. A cache contains a subset of main memory which can be accessed quickly and providing the contents of the cache correspond closely to the requests being made by the processor, the majority of memory references can be satisfied by the cache (cache **hits**) and the apparent latency of accessing main memory is much reduced. On the other hand, if the requests being made by the processor cannot be found in the cache (cache **misses**), then the full latency (and energy) penalty of accessing main memory is imposed. Clearly, keeping "useful" data in the cache is of paramount importance.

Originally a single level of caching was adequate, but as the speed of logic outpaced the speed of memories, multiple levels with different capacity/latency tradeoffs are

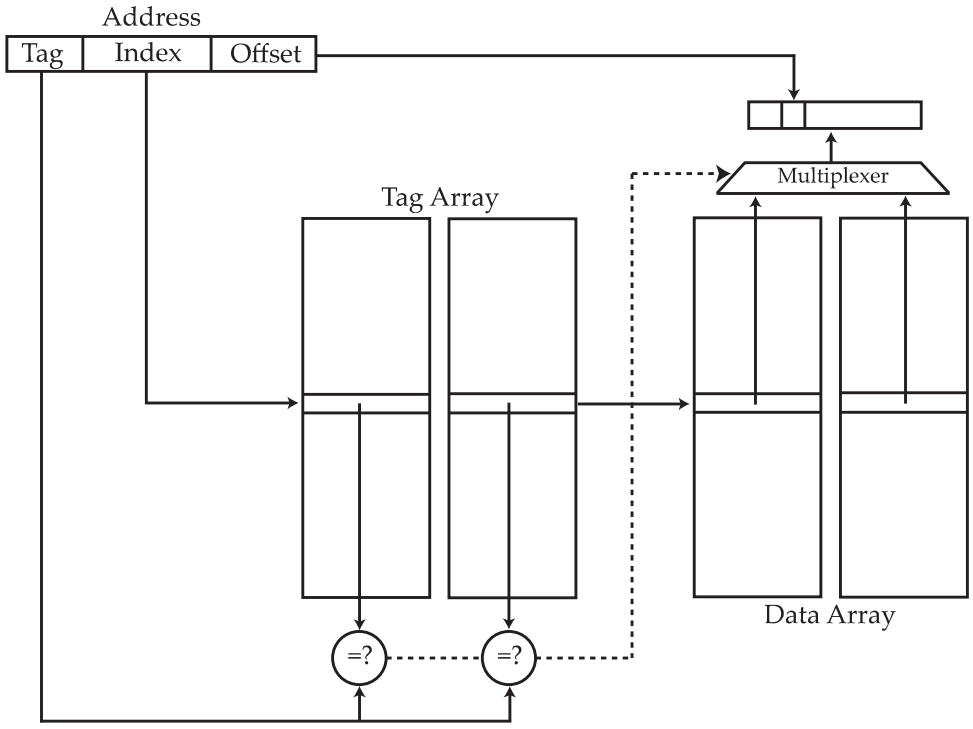

Figure 2.5: Simplified modern two-way set-associative cache

now common. For example, a first-level cache may favour a small latency (prioritising latency over capacity), whereas a second-level cache may favour a low miss rate (prioritising capacity over latency). These caches are usually **inclusive**, such that any data at level n must also be contained at level $n + 1$[2]. In most of the following discussion, the remarks made are independent of which level of cache is being considered.

Figure 2.5 shows an example of a modern cache, adapted from the Alpha 21264 cache as presented by Hennessy and Patterson [HP03].

The target address of each reference is split into three components. The **index** is used to first index into the **tag array** yielding a set of potential tags, one for each degree of associativity of the cache. The retrieved tags are then compared against that from the target address to determine if the requested data can be found in the cache. In parallel, the same index is used into the **data array** to retrieve the cache lines which *may* contain the requested data. The outcome of the tag matching indicates which cache line, if any, contains the requested data. Finally, the **offset** is used to index into the cache line to return the exact word (or halfword, byte etc.) requested. If none of the tags match, a memory request is submitted to the next level in the memory hierarchy (which may be main memory or another cache). When this request is satisfied, a suitable cache line is chosen by the **replacement policy** to be **evicted** and replaced by the new cache line.

While the tag and data lookups usually occur in parallel for optimum performance, they are sometimes performed sequentially to reduce power consumption.

[2]Although the copy of the data at level n may be more recent than that at level $n + 1$

2.4.1 Basic Cache Parameters

The basic parameters influencing the performance of a cache are as follows:

Cache Size

The size of the data and tag arrays has a significant impact on cache performance, with larger caches tending to have higher hit rates. As previously discussed in Section 2.3.3, cache sizes have increased somewhat but strict latency limitations have placed restrictions on cache sizes. Power consumption, economic yield[3] and the reliability of large caches are now also significant issues.

Cache Line Size

Increasing the cache line size, while keeping the overall data array size constant, can increase the hit rate if the application frequently accesses neighbouring words in the larger cache line. However, increasing the cache line size might cause an overall performance degradation due to increased cache fill time and increased bus traffic, as well as reducing the total number of unique cache lines that may be stored simultaneously.

Associativity

The **associativity** of a cache refers to the number of potential locations within the cache to which a cache line with a particular index may be stored.

- **Direct mapped** caches can place a cache line only at a single location within the data array.
- **Set-associative** caches can place a cache line at a limited number of locations within the data array. The number of sets determines the number of potential locations. The cache depicted in Figure 2.5 is known as a two-way set-associative cache since there are two potential locations for each cache line in the data array and so there are two tags for each row in the tag array.
- **Fully-associative** caches can place a cache line anywhere within the data array.

Hallnor and Reinhardt present an interesting implementation of full associativity known as an **Indirect Index Cache** (IIC) [HR00]. Full associativity is achieved via a level of indirection whereby each tag entry contains a pointer to an arbitrary location into the data array. A hash function is applied to the requested address, resulting in an index into a table of tag entries. To reduce the penalty of hash collisions, a small number of tag entries are stored in each row of the tag array. To handle the occurrence of overflowing these entries, tag entries may be chained resulting in an infrequent penalty of having to traverse the chain. Replacement in the IIC is handled entirely by software,

[3] Economic yield refers to the fraction of acceptable devices fabricated compared to the number of total devices fabricated. Increasing cache size by increasing die area and/or decreasing process geometry increases the incidence of defects per device, hence decreases the overall economic yield.

allowing sophisticated algorithms to be implemented. Hallnor and Reinhardt propose generational replacement whereby cache lines are pooled according to their frequency of reference and the least frequently accessed cache lines are replaced.

Hallnor and Reinhardt identify the following advantages of their IIC:

- Software-management techniques may be applied to implement sophisticated replacement algorithms. Caches of lower associativity have far fewer choices about where to place a cache line so the potential benefit of software management is limited in a set-associative cache.
- A fully-associative cache can lock (pin) data in the cache much more efficiently than caches of lower associativity. Locking is used to provide performance guarantees in real-time systems and may also be used to avoid side-channel attacks such as that detailed by Bernstein [Ber05].
- A shared, fully-associative cache can be easily partitioned into arbitrarily sized pieces. Such resizeable shared caches would be applicable in chip multiprocessors.

A **pseudoassociative** cache is similar to a direct-mapped cache, but in the case of a cache miss the address undergoes a simple transformation (e.g. the most significant bit of the index is inverted) and a second cache access performed. If this second access hits, the latency is twice that if the first access had hit, but still smaller than accessing the next level in the memory hierarchy. However, if this second access misses, the next level in the hierarchy must be accessed with twice the penalty due to the additional cache access.

Reference Stream

The properties of the stream of references from the CPU (or equally from the previous level of cache in the memory hierarchy) is a key determinant of cache performance. If the reference stream exhibits significant **temporal locality** (i.e. once accessed, references to the same address are likely in the near future) or **spatial locality** (i.e. once accessed, references to neighbouring addresses are likely in the near future), the cache hit rate and hence overall performance is high. As such, much effort has been put into **cache blocking**, designing algorithms for popular applications which operate on "cache-sized" blocks.

Some applications exhibit very poor temporal and spatial locality, in particular **streaming** applications such as graphics and network processing, and these applications perform much better with increased effective memory bandwidth rather than decreased effective memory latency. While caches are primarily intended to reduce latency, now that many caches are implemented on-chip their effective bandwidth is far higher than that of off-chip main memory. Therefore, providing data can be satisfied by the cache (for example, by prefetching which is discussed later), caches can be of benefit to streaming applications as well.

Further down the memory hierarchy, the filtering effects of cache hits at higher levels can cause a significant impact on the characteristics of the reference stream observed

at the lower level. Whilst the impact on traditional cache performance is usually small, it is important to account for this filtering effect when designing more complex cache architectures.

Replacement Policy

For set-associative and fully-associative caches a choice needs to be made when evicting existing data from the cache and bringing in new data to replace it. The performance impact is highly dependent upon the characteristics of the application. Popular options include:

- **First In, First Out** (FIFO) evicts the oldest line to have been brought into the set or cache.
- **Last In, First Out** (LIFO) evicts the youngest line to have been brought into the set or cache.
- **Least Recently Used** (LRU) evicts the least recently referenced line in the set or cache
- **Random** evicts a random line from the set or cache.

Some microprocessors allow the replacement policy to be varied by software according to the expected reference stream. For example, the Cell microprocessor allows the programmer to set up a **replacement management table** which, based on address space ranges, identifies the candidates for replacement [JB07]. Using this scheme, a streaming application may choose LIFO replacement for its cache lines, while a critical interrupt handler may choose LRU or even pin its cache lines by specifying an empty list of replacements.

Non-blocking Caches

Early caches were blocking i.e. no further cache accesses could be made until any outstanding cache miss was satisfied. As reference streams became more demanding, this condition placed a significant restriction on the performance benefit of introducing a cache. Instead, **non-blocking** (also known as **hit under miss** or **lockup-free**) caches were introduced which are able to still operate even with multiple outstanding cache misses. The most common implementation of a non-blocking cache uses **Miss State Holding Registers** (MSHRs), proposed by Kroft [Kro81]. Each **primary** miss (the first miss to a cache line) allocates an MSHR and subsequent misses to the same cache line, **secondary** misses, are appended to the existing MSHR. The number of outstanding misses before the cache is forced to block is determined by the number of MSHRs, a design-time decision made balancing the high (usually full) associativity of the MSHRs against the latency of accessing both the highly-associative MSHRs and the next level in the memory hierarchy.

Write Policies

A **write through** cache passes all store operations to the next level in the memory hierarchy immediately, whereas a **write back** cache tracks which cache lines are **dirty** (i.e. those which have been modified during their residency in the cache) and only passes on store operations for modified cache lines when the line is evicted. A **write allocate** cache will allocate a cache line for a store access which misses in the cache whereas a **no-write allocate** cache will not.

2.4.2 Cache Miss Classification

Having detailed some of the parameters which impact cache performance, it is useful to classify the various types of cache misses according to their (actual or perceived) cause. Such a classification provides some insight into how cache performance may be improved. Hill and Smith proposed a simple scheme [HS89] using three categories, known as the three C's:

- **Conflict** misses are those made by a set-associative cache but not by the same-sized fully-associative cache
- **Capacity** misses are those made by a limited-size fully-associative cache but not by an infinite-sized cache
- **Compulsory** misses are those still made by an infinite sized cache

In the interests of completeness, a fourth C has since been introduced when dealing with cache-coherent multiprocessor systems. **Coherence** misses are those made due to the implementation of the cache coherency protocol but since the work described in this dissertation focuses primarily on uniprocessor systems, this fourth category will not be considered further.

With respect to the basic cache parameters already introduced, the following general trends may be observed:

- Increasing the cache size decreases the number of capacity misses, at the expense of more complicated cache logic and hence potential increases in latency.
- Increasing the cache associativity decreases the number of conflict misses but again at the expense of more complicated cache logic and hence potential increases in latency.
- Increasing the cache line size may decrease the number of compulsory misses but may also increase the number of conflict misses.
- Prefetching, a technique which is discussed later, can decrease the number of compulsory misses.

The overall trend is that bigger, more associative caches have higher hit rates, but there is a tradeoff between size, associativity and the cache latency which must be minimised to provide data in a timely manner to the processor.

2.5 Quantifying Performance and Power Consumption

When previously discussing cache performance, only the cache **hit rate** (the ratio of cache hits compared to the total number of cache accesses) was considered. In this section, various other performance and power consumption metrics are discussed.

2.5.1 Latency and Bandwidth

The two properties of a cache which are most key to attacking the memory wall are the **latency** (the time taken to access a single item from the cache given a hit) and the **bandwidth** (the rate at which data can be transferred to or from the cache). As shown by Figure 2.3, cache latency, measured in processor cycles, has been reasonably constant, if not growing slightly, over recent years. On the other hand, cache and memory bandwidth have been growing exponentially, as detailed by Patterson [Pat04]. For most current applications, the available memory bandwidth is adequate, whereas if the apparent memory latency could be decreased then significant overall performance benefits could be achieved.

Both latency and bandwidth are closely tied to the specific physical implementation of the cache and its bus. Cache implementation simulators such as CACTI v3 [SJ01] can provide estimates of the likely values when fabricated at a particular technology node, but given the complexity of cache implementations on the most cutting-edge process technologies, not least the rising issue of wire delay in large caches, such simplistic cache implementation simulators are becoming less applicable. As a result, more advanced cache implementation simulators which incorporate more parameters have been developed, including CACTI v6 [MBJ09]. However, exact figures can only be obtained by considering detailed circuit models describing the precise physical implementation.

2.5.2 Average Memory Access Time

Hennessy and Patterson [HP03] define the **average memory access time** as:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

where the **hit time** is the time to service a hit in the cache, the **miss rate** is the fraction of references which result in a miss and the **miss penalty** is the additional time incurred by a miss. Thus to decrease the average memory access time, the hit time must be decreased (lower hit latency), the miss rate must be decreased or the miss penalty must be decreased.

Miss rates and the average memory access time can be estimated using cache access simulators such as Dinero [EH03] or Cheetah [AL03]. These simulators are provided with a reference stream of time-stamped memory accesses and for each memory access the simulator determines whether the access would hit or miss in the given cache configuration. The reference stream may be extracted using specialised hardware from an existing system, instrumented binaries on an existing system or via simulation of

a proposed system. Since the performance of the memory subsystem can influence the timing and ordering of the memory reference trace, particularly with aggressive out-of-order processors, a more system-wide simulator is often preferred.

2.5.3 Misses per 1,000 Instructions

The disadvantage of the miss rate and average memory access time metrics previously discussed is that they take no account of the frequency of memory references in the instruction stream. The overall contribution towards system performance of a cache with a given miss rate (or average memory access time) depends heavily upon how often memory references are made when compared to other (arithmetic and control flow) instructions. A system which makes very few memory references may find a higher miss rate (or average memory access time) tolerable whereas a system which makes very frequent memory references may not. For this reason, the number of **Misses Per 1,000 Instructions** (MPKI) is commonly used as an indicator of specific cache performance whilst trying to be independent of the properties of the instruction stream and other latency-tolerating schemes which the processor may employ.

2.5.4 Instructions Per Cycle

A common metric used throughout computer architecture to describe the overall performance of a processor is to measure the average number of **Instructions committed Per clock Cycle** (IPC). This is clearly dependent on a vast range of parameters, from those describing the detailed microarchitecture of the processing core, to those covering the cache hierarchy and memory subsystem, to the properties of the specific code running on the processor. However, the impact of just one seemingly small aspect, in this case the cache architecture, can be isolated by keeping all other relevant factors constant.

2.5.5 Power Consumption

As power consumption becomes a major limitation to further increasing processor performance, much research has turned to investigating techniques to reduce overall power consumption, approaching the problem from both low-level transistor optimisations and also high-level system-wide techniques. Since caches are large, regular structures they are particularly amenable to relatively simple power-saving techniques. However, without resorting to detailed circuit models it is rather difficult to estimate power consumption numerically, although some tools such as Wattch [BTM00] estimate dynamic power consumption by breaking the cache into constituent elements (such as memories, buses etc.) and accounting for accesses to each element. Static power consumption, becoming a more significant factor, is now increasingly considered by more modern tools such as HotLeakage [LPZ+04] and McPAT [LAS+09].

2.6 Improving Cache Performance

Since the performance of the memory subsystem is critical to that of the whole system, there has been a vast range of previous work covering caches, some of which have developed into entire avenues of research in themselves. One relatively early survey published by Smith in 1982 [Smi82] covers the basic cache parameters already described as well as briefly touching upon prefetching, an area which will be discussed shortly.

Hennessy and Patterson found more than 5,000 research papers covering the field of processor caches from 1989—2001 but restricted their discussion only to those which had been implemented in commercially viable computers [HP03]. Unfortunately the topic of this dissertation affords no such luxury. Therefore the techniques described in this section are limited to those of direct relevance to the subsequent work.

2.6.1 Victim and Other Multilateral Caches

Introduced by Jouppi and Eustace, the victim cache is a small fully-associative cache placed between two levels of memory in the hierarchy [Jou90]. Whenever a cache line is evicted, it is first stored temporarily in the victim cache. If a request for that cache line occurs in the near future, it may be satisfied by the victim cache rather than having to access the next level in the memory hierarchy. Effectively, the victim cache adds further associativity to selected cache sets, aiming to reduce the number of conflict misses. Stiliadis and Varma extended Jouppi's victim cache by selectively placing incoming cache lines in either the main (first-level) cache or victim cache depending on their predicted future usage [SV94]. The same predictor also allows for interchange of cache lines between the main cache and the victim cache. The prediction algorithm is based on the modified cache replacement scheme known as **dynamic exclusion** proposed by McFarling [McF92]. Each cache line has an associated **hit bit** which indicates whether or not there was a hit to that cache line the *last* time it was resident in the L1 cache. While resident in the L1 cache, each line also has an associated **sticky bit** which is cleared if that line is preferentially treated compared to some other conflicting line. The replacement policy considers both these bits and tries to determine which of two conflicting lines is more likely to be accessed soonest in the future. If cache line B is accessed, there are three possible cases, as detailed by Stiliadis and Varma's pseudo-code:

```
Case 1: Hit in main cache:

        hit[B] = 1; sticky[B] = 1;
        (update hit and sticky bits)

Case 2: Miss in main cache, hit in victim cache:

        Let A be the conflicting line in the main cache;
        if sticky[A] == 0 then
                interchange A and B;
                sticky[B] = 1; hit[B] = 1;
```

```
        else
                if hit[B] == 0 then
                        sticky[A] = 0
                else
                        interchange A and B;
                        sticky[B] = 1; hit[B] = 0;
                end if
        end if

Case 3: Miss in both main and victim caches:

        Let A be the conflicting line in the main cache;
        if sticky[A] == 0
                move A to victim cache;
                transfer B to main cache;
                sticky[B] = 1; hit[B] = 1;
        else
                if hit[B] == 0 then
                        transfer B to victim cache;
                        sticky[A] = 0
                else
                        move A to victim cache;
                        transfer B to main cache;
                        sticky[B] = 1; hit[B] = 0;
                end if
        end if
```

Stiliadis and Varma find that such a scheme performs well for instruction caches, but data caches, which are typically subject to a much less structured reference stream, do not show such large improvements.

John and Subramanian propose a related scheme, the **annex cache** [JS97]. Like the victim cache, this scheme uses a small highly-associative buffer, termed the annex cache, which operates alongside a traditional main cache. Incoming cache lines are first placed in the annex cache and only promoted to the main cache if they are referenced twice in succession without any references being made to conflicting cache lines. The aim of this scheme is to keep higher-usage lines in the main cache by preventing them from being evicted by lower-usage lines. This scheme is shown to perform well for the instruction cache, results for the data cache are not provided.

Hu *et al.* investigate what they term **timekeeping techniques** to manage a victim cache [HKM02]. This approach is closely related to the work presented later in Chapters 4 and 5. Their technique uses a simple predictor which deems a conflict miss to have occurred if, on eviction, the time since the last access to the cache line being evicted is less than a fixed threshold. Such lines are then allocated space in the victim cache while other lines are passed directly on to the next level in the memory hierarchy. This approach is designed to maximise the number of conflict misses satisfied by the victim cache since they are most likely to be those cache lines which are reused in the near future.

Collins and Tullsen propose another approach to manage the victim cache through the use of a **Miss Classification Table** (MCT) which predicts whether a cache line being evicted is due to either a conflict or a capacity[4] miss [CT99]. The MCT simply stores the partial tag of the last cache line evicted from each cache set; if the partial tag of a newly-evicted line matches the partial tag of the previous line which was evicted from that cache set, it is predicted as a conflict miss. Collins and Tullsen investigate three victim cache policies which exploit their miss classification prediction. The first policy does not swap cache lines between the main and victim caches when a victim cache hit occurs to a cache line which is predicted to be a conflict miss. The second policy bypasses the victim cache when a cache line is evicted from the main cache which is predicted not to be a conflict miss. The third policy combines the first two policies simultaneously. The average speedup observed for all three policies is moderate, compared to a conventional victim cache.

Khan *et al.* propose a **virtual victim cache** which places evicted cache lines in the adjacent cache set [KJBF10]. The specific position chosen is the most-recently used position containing an invalid or predicted dead cache line, or the least-recently used position if no cache lines in the adjacent set are invalid or predicted dead. The term **dead** refers to the cache line not being accessed again prior to eviction, and is further explored in Chapter 3. Khan *et al.* use a trace-based predictor which combines the program counter values of all instructions touching a particular cache line which then indexes two tables of counters with a different hashing algorithm for each. The sum of the two counter values is then compared against a fixed threshold to determine whether the cache line is predicted as dead or not. Trace-based predictors and similar counters are further discussed in Chapter 4.

Victim caches are one example of **multilateral** caches, where multiple data stores are used in parallel with some scheme to choose between them. One of the earliest multilateral caches was the **dual data cache**, proposed by González *et al.*, which predicts whether data references will show spatial or temporal locality [GAV95]. Separate caches are maintained for each type of locality and there is also the option to bypass the cache entirely if no locality is predicted. This approach is designed for vector processors whose types of locality are relatively easy to predict with adequate accuracy and coverage.

Rivers *et al.* review several multilateral architectures based on data reuse [RTT+98]. **Cachable Non-Allocatable** (CNA), first proposed by Tyson *et al.* [TFMP95], allocates lines based on the reuse behaviour of the cache line previously accessed by an instruction with the same program counter. If any words within that previous cache line were reused, then the cache line is placed within the large main cache, otherwise it is placed in a much smaller secondary cache. Very similar in concept is Rivers and Davidson's **Non-Temporal Streaming** (NTS) cache [RD96] which allocates lines based on the previous behaviour of the cache line with the same effective address. If during its previous residency, no word within the cache line was reused then it is allocated to a smaller secondary cache rather than the larger main cache. Finally, Johnson and Hwu's **Memory Address Table** (MAT) cache [JH97] tags cache lines as either frequently or infrequently accessed on the granularity of macroblocks (contiguous groups of cache lines expected to have similar reuse patterns). A counter, indicative of "usefulness", is maintained for each macroblock. Every cache access to that macroblock increments the counter.

[4]Compulsory misses are grouped together with capacity misses

Each incoming cache line as a result of a cache miss has its counter value[5] compared with the resident cache line. If it is greater, the cache line is replaced (the incoming cache line is predicted to be more useful) otherwise the incoming cache line is placed in a separate smaller data store (the incoming cache line is predicted to be less useful). Comparing the three schemes to the traditional victim cache, Rivers *et al.* find that for direct-mapped first-level caches the victim cache still performs best overall but if the first-level cache is made two-way set-associative, address reuse information schemes (NTS and MAT) perform better than a program counter reuse information scheme (CNA) or a traditional victim cache. A second set of simulations by Tam *et al.* show similar results [TRS+99]. This second study also investigates a near-optimal, but not implementable[6], scheme known as **pseudo-opt**. Again, the address reuse based predictors performed better than those based on program counters, but both exhibited a significant lag behind the performance of the pseudo-opt scheme.

One more recent multilateral cache architecture scheme is **Allocation By Conflict** (ABC), proposed by Tam *et al.* [TVTD01]. This approach attempts to prevent an incoming cache line from replacing a conflicting cache line which is still being actively referenced. The least recently used cache line in a set is replaced only if it has not been reaccessed since the last miss reference to that set which did not cause a replacement. Otherwise, the incoming cache line is placed in a separate, smaller cache. This approach generally outperforms those previously discussed. However, Tam *et al.* also show that random allocation performs surprisingly well.

2.6.2 Prefetching

Traditional cache architectures are **demand fetch**, cache lines are only brought into the cache when they are explicitly requested by the processor. Alternatively, cache lines may also be proactively fetched into the cache, anticipating that they will be used in the near future. Such a technique is known as **prefetching** and has been widely employed in high-performance memory subsystems. VanderWiel and Lilja survey a variety of prefetching techniques, concluding that no single strategy considered is optimal in all cases [VL00]. Smith also includes prefetching in his earlier survey of caches [Smi82]. Prefetches must be:

- **Useful**: the prefetched data must be referenced in the near future.
- **Timely**: the prefetch must occur early enough such that the data is ready in the cache when the corresponding demand fetch occurs. Equally, the prefetch should not occur too early such that it occupies space which would be better utilised by storing other data.
- **Little Overhead**: the prefetch must not interfere with the standard demand fetch model to the detriment of overall performance.

[5] Counter values for each macroblock are stored in a separate table indexed by macroblock address

[6] The pseudo-opt scheme is not implementable since it relies on future knowledge to make its decisions

As prefetch schemes become more aggressive, the overhead of prefetching can become significant. This overhead may be expressed in terms of competition for bandwidth with demand fetches, as well as competition for space (**cache pollution**) if the prefetched and demand-fetched data share the same cache. Prefetching techniques can be divided into three categories as follows.

Software Prefetching

Many high-performance microprocessors provide explicit prefetching instructions. Examples include loads targeting registers `R31` or `F31` on the Alpha 21264 [Cor99], `lfetch` on the Intel Itanium [Cor06] and `dcbt` on the IBM PowerPC [WSM+05]. The operation of such instructions is very similar to a traditional load from memory, but no register is allocated to receive the resulting data and any exceptions raised are suppressed[7]. These instructions may be used explicitly by a programmer seeking to hand-optimise their code, in library code, or they may be inserted automatically as part of an optimisation pass made by a compiler. They are particularly effective when employed in loops responsible for large array computations, a common task in scientific code, which would otherwise exhibit poor cache performance. By taking advantage of the referencing pattern known at compile time, suitable prefetch instructions may be issued early to ensure that the subsequent demand fetches will hit in the cache. While it has been shown that software prefetching can be effective in certain cases, the more varied reference patterns of general applications make suitable compiler analyses difficult. Compared to other types of prefetching, prefetch instructions often add to the execution time of the program since they must pass through the processor pipeline.

More specific cache control facilities are becoming available to the programmer. These include the non-temporal `nt1` and `nta` load hints on the Intel Itanium [Cor06], data cache block set to zero (`dbcz`), cache line flush (`clf`) and cache line invalidate (`cli`) instructions on the IBM PowerPC [WSM+05] and the **cache operations register** and the **cache lockdown registers** on the ARM11 family of processors [ARM06]. However, as with software prefetch instructions, their use is limited to either explicit programmer invocation or implicit library or compiler optimisations.

Hardware Prefetching

While the future knowledge available to software prefetching can be very effective in some circumstances, it is not always detailed enough to be more widely applicable. Hardware prefetching takes advantage of the detailed information available at runtime and speculates about future memory references based upon those observed in the past.

One of the simplest hardware prefetching schemes is **One Block Lookahead** (OBL) which issues a prefetch for cache line $i + 1$ whenever cache line i is referenced by a demand-fetch. This is slightly different from simply doubling the cache line size since the two consecutive cache lines will be treated separately by the replacement policy, but essentially as one entity by the allocation policy. **Tagged prefetch** is one popular

[7] To preserve correctness, prefetches are only considered as hints that a particular address will be referenced.

implementation of OBL which issues a prefetch for cache line $i+1$ whenever cache line i is demand-fetched *or* upon the first reference to the previously prefetched cache line i. Referring to the list of desirable prefetch properties, if the reference stream simply increments consistently by small amounts (as is common in large array calculations), OBL can perform well. However, OBL prefetches may not be timely if the loop iterating over memory is very tight. It is possible to prefetch all cache lines up to $i + k$ where k is the **degree of prefetching**, but this may impose a significant overhead even if most of the cache lines for which prefetches are issued are already resident in the cache.

In the same paper that introduced victim caches, Jouppi also proposed **stream buffers**, a FIFO structure into which prefetched cache lines are placed avoiding any cache pollution [Jou90]. Only the head of the stream buffer is considered by the processor and on a hit the cache line is transferred into the main cache and all entries moved up by one, a prefetch being issued to fill the empty tail slot. On a miss, the whole structure is flushed and new data brought into the buffer. This approach works well if the reference stream exhibits long sequences of references to incrementing addresses. However, even the simplest scientific calculations typically involve multiple such streams with interleaved accesses. For these applications, Jouppi extends the single stream buffer to a set-associative **multi-way stream buffer** which can cope with such reference patterns. These reference patterns are common on vector machines and an extension to Jouppi's multi-way stream buffer to filter stream buffer allocation and cope with non-unit strides described by Parlacharla and Kessler [PK94] was implemented on the Cray T3E system. The problem of non-unit strides has also been tackled by numerous other researchers. VanderWiel and Lilja identifies Chen and Baer's approach [CB95] as the most aggressive proposed at the time of their survey [VL00]. A **reference prediction table** is used to store observed strides from recent load instructions. A simple state machine determines when a consistent non-unit stride has been achieved for a particular instruction and proceeds to issue prefetches for addresses based on that observed stride. While this scheme can perform well for certain looping constructs, it only issues prefetches one iteration in advance which may not be adequate to ensure the prefetch is timely. To remedy this, Chen and Baer propose a **lookahead program counter** which runs ahead of the conventional program counter and uses the existing branch prediction hardware. The degree to which the lookahead program counter runs ahead of the conventional program counter depends on the latency to prefetch data from the next level of the memory hierarchy.

Seeking to leverage the varying spatial locality observed both between and within applications, Kumar and Wilkerson describe a **Spatial Footprint Detector** (SFD) [KW98]. This structure tracks the use of cache lines within larger contiguous groups, known as **sectors**, using **spatial footprints**, a bit-vector showing which cache lines within the sector have been used. Rather than just fetching the desired cache line on a miss, the SFD is used to decide which additional cache lines within a sector to prefetch, speculating that since those lines have been useful in the past, they will be useful again in the future. The SFD itself is constructed from a table of previously observed spatial footprints, indexed by a combination of bits from the reference address and the instruction causing the miss. This approach is somewhat similar to OBL with a higher degree of prefetching, but the spatial footprint predictor helps to reduce cache pollution by selectively fetching those cache lines deemed to be useful. Chen *et al.* later propose a very similar prefetcher based upon a slightly refined spatial footprint predictor which they term a **spatial pattern predictor** [CYFM04].

Other prefetching techniques targeted to more complex reference patterns include those aimed at pointer-intensive code, becoming more prevalent with the increased use of object-oriented languages which commonly require various levels of indirection. Both Harrison and Mehrotra [HM94] and Roth *et al.* [RMS98] describe methods which aim to identify loads that access linked data structures. The specific type of loads identified are those whose values are subsequently used as base addresses for other load instructions. A common example is the traversal of a linked-list data structure. Roth *et al.* conclude that such prefetching can improve performance, but the prefetches issued may not always be timely depending on the amount of processing taking place on each node in the data structure.

One final class of hardware prefetch techniques targeted at complex reference patterns are **correlation**-based techniques, whereby observed microarchitectural events are used to to infer the future addresses that will be accessed, much in the same way that correlation-based branch predictors use the direction of past branches to predict the direction of future ones. Such techniques were first proposed by Charney and Reeves [CR95] and there have been a number of implementations since. Joseph and Grunwald model the addresses accessed as a Markov process [JG97], requiring a large amount of storage but the performance benefit gained is greater than that of dedicating the equivalent amount of storage to simply enlarge the cache. This technique can be very aggressive, generating up to four prefetch targets at once. Lai *et al.* propose a correlation scheme which combines a **dead-block predictor** together with a **Dead-Block Correlating Prefetcher** (DBCP) [LFF01]. The dead-block predictor consists of a history entry for each cache line which stores a fixed-size encoded instruction trace, known as a **signature**, for the sequence of memory operations which have touched that cache line. The address being accessed and the encoded history is then used to index a dead-block table which predicts whether the cache line is dead or not. A **dead** cache line is defined to be one which will not be referenced again prior to eviction. Extending this further, a **dead-block correlating address predictor** adds another entry to the dead-block table which is a prediction of which cache line to prefetch if the current cache line is deemed dead. In order to achieve adequate accuracy and coverage, the size of the dead-block table is relatively large, 2MB on-chip or 7.6MB off-chip with corresponding latencies.

Analogous to dead block prediction, Lai and Falsafi propose a **last-touch predictor** which seeks to identify the last reference to a cache line in a cache-coherent distributed shared memory system prior to its invalidation by another processor [LF00]. Cache lines which are predicted to have had their last-touch are speculatively self-invalidated to reduce the impact of coherence misses. The predictor itself is similar to a traditional two-level branch predictor, with the first-level table holding the current signatures for each line and the second-level table holding previously-observed last-touch signatures on a per-line or global basis. A signature consists of the truncated addition of all program counter values which have touched that line since the last coherence miss.

Hu *et al.* propose a timekeeping scheme which uses a simple 1-miss history per cache frame and a predictor which, given this history, specifies which cache line to prefetch and critically a prediction of when to schedule the prefetch based on the expected lifetime of the cache line currently resident in the cache [HKM02]. Comparing an 8kB timekeeping correlation table against a 2MB dead-block correlating prefetcher, Hu *et al.* found similar if not better performance for a much smaller storage requirement. Again seeking to reduce the storage required, Hu *et al.* later proposed **Tag Correlat-**

ing Prefetchers (TCP) which track the history of cache tags (rather than complete addresses), comparing a 8kB TCP to a 2MB DBCP and finding that the smaller prefetcher generally outperforms the larger at the L2 level. When considering prefetching into the L1 from the L2, a dead-block predictor was found to be necessary to avoid excessive cache pollution but the additional improvement to overall performance was slight.

Combined Hardware/Software Prefetching

As the complexity of prefetchers increases, some more recent research has looked at combining the imprecise future knowledge available to the compiler with the detailed run-time information available to hardware. Chen proposes a general **programmable prefetch engine** consisting of a **run ahead table** populated using explicit software instructions [CB95]. Each entry of the run ahead table specifies a prefetch stream which may be initiated should the current program counter match that stored for the stream. The types of stream targeted by Chen are simple constant strides which are readily detectable by compiler techniques. A more advanced technique is presented by VanderWiel and Lilja through the use of a **Data Prefetch Controller** (DPC) [VL99]. The DPC is programmed by the processor with a set of prefetch streams which have been determined at compile-time. To ensure the DPC and processor remain synchronised, the issuing of prefetch requests is based upon specific **trigger blocks**, again analysed at compile-time, which allow the establishment of a producer-consumer relationship. A yet more complex approach is proposed by Solihin *et al.* which uses a **user-level memory thread**, implementing a correlation-based prefetcher and running on a dedicated general-purpose processor in memory [SLT02]. Wang's PhD dissertation considers cooperative management of all cache aspects [Wan04], focusing specifically on replacement policies and prefetching. Compiler hints are passed on to a dedicated prefetch engine which is responsible for scheduling the required types of prefetches. The hints described by Wang include **spatial** (prefetch the area around a load), **pointer** (follow pointers in the loaded cache line) and **recursive** (fetch a data structure recursively).

Another combined hardware/software approach to prefetching is the concept of a **runahead** mode or a **hardware scout**, proposed by Dundas and Mudge [DM97] and implemented in the IBM POWER6 [CN10]. Upon a cache miss, an in-order processor usually has to block until the data can be supplied by the memory hierarchy. Instead, the processor enters runahead mode, in which instructions continue to be fetched and executed, aiming to prefetch useful data into the caches.

One final approach is a software-programmable **helper thread**, which runs in parallel with the main computation. This thread is a more generic version of VanderWiel and Lilja's Data Prefetch Controller, and allows arbitrarily complicated reference patterns to be generated [XIC09]. Chappell *et al.* propose, but do not evaluate, helper threads in a **Simultaneous Subordinate Microthreading** environment, which is characterised by "additional work [being] done to enhance the performance of microarchitectural structures, solely for the benefit of the primary thread" [CSK+99].

2.7 Decreasing Power Consumption

As previously discussed, larger and larger on-chip caches are occupying proportionally more die area and so are responsible for proportionally more power consumption. Traditional techniques to tackle dynamic power consumption include clock gating and reducing the supply voltage, well-known approaches which have been widely deployed as described by Borkar [Bor01]. More recently, static power consumption has become increasingly significant as process geometries shrink. While quantitative results of power-saving techniques will not be provided in this dissertation, some of the microarchitectural constructs are reusable from performance-improving applications, as well as being an important application area for future cache utilisation research. Various relevant approaches to reducing static power consumption in caches are now considered.

Circuit-level research has provided two families of techniques to reduce the static power consumption of a memory element. **State-preserving** techniques retain the contents of the memory but require additional latency to return to the **active** state in which the contents of the memory can be accessed. Cache lines in this state are typically known as **drowsy**. Examples of such techniques include **auto-backgate-controlled multi-threshold CMOS** [NMT+98], **dynamic voltage scaling** [FKM+02] and **data retention gated-ground** [ALR02]. **State-destroying** techniques do not preserve the contents of the memory but allow greater power savings to be made. This state is typically known as **sleep**. Examples of such techniques include **gated-V_{dd}** [PYF+00] and **gated-V_{ss}** [LPZ+04]. These techniques can be coupled with a variety of microarchitectural predictors in order to determine when to transition cache lines into the various low-power states as follows.

2.7.1 State-Destroying Techniques

Reducing power consumption using state-destroying techniques provides the highest potential reductions in power consumption, but at the cost of necessitating explicit safety mechanisms in case of a misprediction. This is often accomplished by relying on the inclusion property of the memory hierarchy, having ensured any modified data is written back to the appropriate level. Powell *et al.* present the first cache design which integrates microarchitectural and circuit-level techniques to reduce static power consumption [PYF+00]. Their **dynamically resizeable instruction cache** changes the size of the cache at run-time, using a gated V_{dd} approach to effectively switch off those cache lines which are not required. The size of the cache is determined by periodically examining the miss rate, decreasing the size of the cache if the miss rate is determined to be acceptable or increasing the size of the cache if the miss rate is unacceptable. Resizing the cache is achieved by changing the boundary between the tag and index portions of a memory address during the cache indexing procedure, effectively limiting the number of cache sets. Since the tag size must be allowed to vary, the tag array is sized such that it is capable of storing the largest tag size possible (corresponding to the smallest cache size allowed), which can be a source of additional power consumption if the cache size is relatively large.

Albonesi proposes a related technique known as **selective cache ways**, which varies the associativity of a set-associative cache in order to reduce power consumption [Alb99].

The decision about which cache ways to enable is controlled by a machine register, accessible to the program or operating system and Albonesi suggests that software is responsible for setting this register according to the current demands of the application.

Yang *et al.* propose a later scheme [YFPV02] which combines the **selective-ways** [Alb99] and **selective-sets** [PYF+00] approaches to allow a finer degree of control over cache size. Comparing **static** and **dynamic** resizing, Yang *et al.* find that static resizing, driven by profiling the application, generally performs better than a dynamic scheme which periodically examines cache miss rates.

Kaxiras *et al.* present the first time-based approach to reducing cache leakage power [KHM01]. Observing that cache lines are frequently storing data which will not be referenced before the line is evicted, they aim to identify such lines and put them into a state-destroying low-power mode via a gated-V_{dd} technique. Two predictors are presented, the first puts a line into the low-power state after a fixed number of cycles have passed without a reference to it, while the second uses an exponentially increasing scheme per cache line, attempting to capture the variability in usage both between cache lines in the same application as well as between cache lines in difference applications. This first predictor is shown to perform well and is the basis for predictors used in other work with both state-destroying and state-preserving implementations.

Zhou *et al.* propose **Adaptive Mode Control** (AMC), a scheme which applies state-destroying power-saving techniques to the data array whilst leaving the tag array always active [ZTRC03]. This allows cache misses due to the power-saving scheme (**sleep misses**) to be differentiated from misses which would have occurred regardless of the state of the corresponding line in the data array (**ideal misses**). Zhou *et al.* propose an adaptive scheme to transition cache lines into the low-power state based upon the cache decay mechanism previously proposed by Kaxiras *et al.* [KHM01]. A fixed target is placed upon the number of sleep misses compared to ideal misses and the period after which to transition a cache line into the low-power state is adjusted accordingly to achieve this target. If too many sleep misses are occurring, cache lines are placed into the low-power mode after a longer period of time and *vice versa*, trading off potential power savings against the performance impact.

Velusamy *et al.* build on Kaxiras *et al.*'s cache decay mechanism and Zhou *et al.*'s adaptive mode control by introducing formal feedback control theory resulting in their **Integral Miss Control** (IMC) approach [VSP02]. This technique differs from AMC by taking a formal approach to the construction of the feedback control path using traditional engineering approaches. Performance is similar to that of AMC, but Velusamy *et al.* argue that the rigorous design process results in less time-consuming ad-hoc tuning of parameters and that such an approach copes better with unexpected inputs. However, they only present results for four SPEC95 and four (integer) SPEC2000 benchmarks.

Also aiming to improve upon prior cache decay and adaptive mode control mechanisms, Abella *et al.* propose a low-power scheme controlled by **Inter-Access Time per Access Count** (IATAC) [AGVO05]. Observing that second-level cache access patterns are similar between cache lines, this approach aims to track global state. While the details of the exact method are rather lengthy, the basic process is to observe the inter-access times for each line and classify the line depending on the number of accesses to the line since it was brought into the cache. This classification then directly provides a prediction for the decay interval, the period of time which must elapse without any

intervening accesses before the cache line is placed in the low-power state. The circuit-level mechanism used by Abella *et al.* to place cache lines in such a low-power state is gated-V_{dd}.

2.7.2 State-Preserving Techniques

In contrast to state-destroying techniques, state-preserving techniques do not have the potential to reduce power consumption as much, but can afford an aggressive prediction mechanism, improving practical performance by relying on the lower latency of transitioning the cache line from the drowsy state to the active state rather than the latency of fetching the cache line from the next level in the memory hierarchy.

Flautner *et al.* present a **drowsy** cache which uses dynamic voltage scaling to place individual cache lines in a low-power, state-preserving mode which requires just a single cycle to transition into the active state in which data can be read [FKM+02]. Given this low latency implementation, they propose a simple prediction scheme in which all cache lines are periodically placed into the drowsy state and awakened upon their first access. Such a scheme is shown to perform reasonably for the first-level instruction cache while no such scheme is required for the second-level cache — the performance impact of keeping all cache lines in the drowsy state is minimal due to the low latency of transition into the active state.

Petit *et al.* seek to improve upon the fixed decay intervals proposed by Kaxiras *et al.* [KHM01] using a 4-way set-associative first-level data cache in which no cache lines, the most recently used cache line or the two most recently used cache lines are in the active state per cache set while the remaining cache lines remain in the drowsy state [PSSK05]. The decision about how many cache lines to activate is made periodically based upon the observation that the number of cache lines touched per fixed period is reasonably constant e.g. if only one cache line was touched in the previous period, only activate one cache line for the next period. This approach has a similarly small performance impact as the original cache decay mechanism while providing slightly superior leakage energy savings.

Similarly, Bhadauria *et al.* observe that the majority of cache references are to cache lines which themselves have been most recently referenced and seek to improve upon Kaxiras *et al.*'s fixed decay intervals via a **reuse distance** policy which keeps the previous n most recently accessed cache lines awake, with the remaining cache lines in a state-preserving, low-power mode [BMST06]. This approach requires little additional hardware since the value of n is typically very small. For the second-level cache, Bhadauria *et al.* find that keeping a single cache line awake is adequate to achieve significant leakage savings with minimal performance impact. For the first-level cache, keeping five lines awake yields the best performance/leakage tradeoff.

Besides investigating spatial locality as a means to guide prefetching, Chen *et al.* also propose the use of a **spatial pattern predictor** to reduce cache leakage energy [CYFM04]. In this application, individual words within a cache line may be placed into a low-power, state-preserving mode by using a data retention gated-ground technique as described by Agarwal *et al.* [ALR02]. The spatial pattern predictor tracks usage of words within each cache line and when a cache line is brought in to the cache, only

those words which are predicted to be referenced are kept in the active state. The predictor itself uses two tables, similar to that described by Kumar and Wilkerson [KW98], indexed by concatenating the low-order bits of the program counter of the instruction causing the miss with the offset within the cache line of the requested word.

2.7.3 Combined Techniques

Several studies have sought to either compare state-preserving and state-destroying techniques or use both in an attempt to gain the benefits of each.

Li *et al.* propose a dynamic voltage scaling scheme to implement a state-preserving low-power mode on a cache line granularity [LKT+02]. They propose five different management strategies which use a combination of this state-preserving low-power mode together with a state-destroying low-power mode to exploit the duplication of data between the first and second-level caches, the principle of inclusion. The strategy found to save the most energy places the second-level cache line in a state-preserving low-power mode whenever the contents of that cache line is transferred into the first-level cache. A combination of this technique and Kaxiras *et al.*'s cache decay mechanism [KHM01] is also shown to provide further energy savings.

In a second study, Li *et al.* compare the drowsy cache and cache decay in the context of vulnerability to soft errors, primarily induced by external radiation [LDV+04]. The dynamic voltage scaling technique used by the drowsy cache increases the number of soft errors reaching the datapath, since the critical amount of collected charge is proportional to the supply voltage. However, the cache decay technique actually *decreases* the number of soft errors reaching the datapath since cache lines are resident in the first-level cache for shorter periods, decreasing the time for which they are exposed. Li *et al.* then propose an energy-efficient optimisation to decrease the likelihood of soft errors by using an **early-write-back** policy.

In a third study, Li *et al.* also compare state-preserving and state-destroying leakage control in caches using an adapted leakage model which includes the exponential effects of temperature [LPZ+04]. Gated-V_{SS} is used in preference to gated-V_{dd} since this technique effectively eliminates bitline leakage. At small second-level cache latencies, gated-V_{SS} (state-destroying) is superior to the drowsy cache (state-preserving) in terms of both the overall leakage savings and the performance degradation. As the latency increases, the drowsy cache surpasses gated-V_{SS}. As the temperature increases, the normalised energy savings increase, as would be expected. Gated-V_{SS} almost eliminates leakage regardless of temperature, whereas cache decay's leakage is non-trivial at low temperatures and increases exponentially with increasing temperature. However, the use of a fixed decay interval prevents gated-V_{SS} from realising its full energy savings, hence there is little relative difference between the two as the temperature is increased. Li *et al.* proceed to show that varying the decay interval per benchmark (by up to a factor of 32) can improve performance considerably but do not suggest methods by which this could be achieved other than relying on previous work. Finally, gate oxide thickness is also shown to have a significant influence on whether state-preserving or state-destroying techniques fare best.

Hanson *et al.* compare several static power-saving techniques in the context of microprocessor caches [HHA+03]. The first applies a simple dual-V_t scheme to the cache,

using higher threshold, lower power consumption transistors in memory cells with lower threshold, higher power consumption transistors elsewhere in the cache. This scheme treats all cache lines equally, so no additional misses are incurred but the cache latency is increased by one cycle to accommodate the slower switching speeds of the higher threshold transistors. Gated-V_{dd} [PYF+00] and multi-threshold-CMOS [NMT+98] are also examined, using the simple cache decay policy proposed by Kaxiras *et al.* [KHM01] which transitions the cache line into a low-power state after a fixed period with no intervening accesses to the line. The dual-V_t approach is shown to provide the best energy savings for the second-level cache but the performance degradation incurred by the additional cycle of latency significantly increases both program execution time and hence energy consumption when applied to the first-level caches. Gated-V_{dd} and multi-threshold-CMOS perform comparably for the first-level caches but is found to be sensitive to the choice of the decay interval after which to transition cache lines into the low-power state.

Meng *et al.* present a limit study investigating how much leakage power can be reduced by applying **active**, **drowsy** (state-preserving) and **sleep** (state-destroying) power modes judiciously to individual cache lines [MSK05]. Given full future knowledge (i.e. which addresses will be referenced and crucially when they will be referenced) and the transition and leakage parameters of each of the power modes, the decision about which mode to best place a cache line in can be made as soon as it is brought into the cache. As might be expected, if the line is going to be referenced again soon, it is best to use the active mode, if it will not be referenced for a long time it is best to use the sleep mode and if it will be referenced in some medium-term timeframe then it is best to use the drowsy mode. The exact time boundaries between these three modes is determined by the energy consumed by each mode, the power consumed by switching between modes and the time taken to switch between modes. For the first-level data cache, instruction cache and the second-level unified cache Meng *et al.* find that 99.1%, 96.4% and 97.7% of leakage power respectively may be eliminated, providing an upper bound for implementable leakage saving schemes.

Finally, many performance optimisations can increase power consumption and there is often a tradeoff to be made. This tradeoff becomes significantly more complex when considering energy as well as power, since the performance optimisation has the potential to reduce the total execution time therefore reducing overall energy consumption even if power consumption is slightly raised. Kadayif *et al.* examine the relationship between prefetching and power saving by turning off cache lines [KKL06]. This approach applies different power-saving policies to prefetched and demand fetched cache lines. The **speculative state-preserving** scheme places prefetched lines in a low-power, state-preserving mode, adding a single cycle of latency when they are subsequently accessed. The **lazy, state-destroying** scheme places prefetched cache lines in the active mode for a short period of time after which they are transitioned into a low-power, state-destroying mode. Finally, the **predictive hybrid** scheme exploits the observation that **useful**[8] prefetches very frequently follow other useful prefetches to the same cache line. If the previous prefetch was non-useful then the current cache line is placed in a state-preserving, low-power mode. If the previous prefetch was useful and the previously prefetched line was accessed soon after being prefetched, the cache line is placed in the active mode. If the previous prefetch was useful and the

[8] A useful prefetched cache line is referenced before the cache line is evicted

previously prefetched line was not accessed soon after being prefetched, the cache line is first placed in a state-preserving, low-power mode for some period related to how long it took the previously prefetched line to be accessed, following which it is placed in the active mode. This third scheme is shown to yield the most energy savings with little performance degradation.

2.8 Other Time-Based Techniques

In this section a variety of other cache management techniques which utilise not only the ordering of memory references, but also their absolute timing are reviewed.

Lee *et al.* propose **eager writeback**, a technique which speculatively writes dirty lines to main memory ahead of eviction [LTF00]. This policy can be considered as a compromise between write-through (immediately writing dirty lines to main memory) and writeback (writing dirty lines to main memory only when evicted). It is important to note that eager writeback is always correct, the limiting cases of its operation being either the traditional write-through or writeback behaviour. The best candidates for eager writeback are those cache lines which will not be written to again prior to their eviction. Such cache lines are identified by examining the LRU replacement stack, finding that on their benchmarks the probability of rewriting a dirty least-recently used cache line is generally small. While for some benchmarks the actual probability of rewriting such a cache line may be numerically large, the actual number of such cases is very small.

The performance benefit introduced by eager writeback comes from better distributed bandwidth utilisation to alleviate memory bus congestion. For the SPEC95 benchmarks, little benefit is gained by eager writeback. This is explained by SPEC95 being a poor candidate for memory system performance studies due to its small working sets. Instead, Lee *et al.* present a pair of kernels. The first kernel is based upon the traditional graphics rendering pipeline, utilising graphics hardware only for rasterisation. The second kernel is intended to be representative of a streaming application, looping over two large arrays. Both kernels show reasonable performance improvements from eager writeback. When additional traffic is injected onto the bus, increasing the potential for contention, more significant performance improvements are observed. Such a scenario occurs in many systems where there are multiple devices trying to access the memory bus simultaneously, for example a graphics accelerator using Direct Memory Access (DMA) to access main memory.

Al-Zoubi *et al.* demonstrate a significant gap between the performance of an optimal cache replacement policy and that which is achieved by currently implemented cache replacement policies [AZMM04]. While there has been much work on general cache replacement policies, several specific time-based cache replacement policies are detailed below.

Lin and Reinhardt note the increasing gap between Least Recently Used (LRU) and optimal (OPT) cache replacement policies as cache size and associativity grow and investigate last-touch references under the two policies [LR02]. All references can be partitioned into three disjoint subsets: references which are not last-touches under either policy, references that are last-touches under both policies and references which are

last-touches under LRU but not OPT. The converse of the latter case is shown to be impossible. In order to attempt to predict last-touch references, a variety of trace-based signatures are introduced, composed of the (past and present) addresses referenced and the (past and present) program counters of the instructions that reference them. An ideal predictor is then introduced which simply accumulates signatures known to be last-touches. Such a predictor is shown to be adequate at the first-level data cache, but the increased associativity and filtered reference stream at the unified second-level cache leads to poor accuracy and coverage. Some *future* knowledge can also be added to the predictor, slightly improving predictability, since the time at which the prediction is required (when a cache miss occurs and a replacement needs to be found) is later than the time of the last-touch to the line. Instead, a **Last-Touch History** (LTH) bit vector is presented which tracks the last-touch history for each cache line. The LTH vector simply stores a sequence of bits indicating if the previous references were last-touches or not. The LTH vector is then used as a signature, having filtered out references to the most recently used line to reduce the storage overhead, in the same manner as before.

Given a last-touch predictor, Lin and Reinhardt proceed to modify the replacement policy by either **early eviction** or **late retention**, with a fallback LRU policy. The early eviction policy replaces the first non-MRU line whose last reference is predicted to be an OPT last-touch. The late retention policy does not replace the line whose last reference is predicted to be an LRU last-touch but not an OPT last-touch. In general, late retention policies are more prone to degradation by inaccurate prediction than early eviction policies, since late retention requires the eviction of one additional line for each miss that takes place while the line is being retained, whereas early eviction suffers at most one additional miss for each misprediction. While both policies improve the miss rate compared to LRU, the early eviction policy performs slightly better than the late retention policy.

Liu and Yeung also investigate last-touch predictors to improve cache replacement decisions [LY09]. They form a signature from the **reuse distance history** of each cache set, as well as the program counter of the current reference. The reuse distance is defined as the the number of unique memory locations referenced between two references to the same memory location. This signature is then used to index into a table of previously-observed last-touch signatures. A **shadow tag array** is required in order to track the history of recently evicted cache lines. Liu and Yeung find that the best victim for eviction is the most-recently used (MRU) cache line which is predicted to have had its last-touch. Their **Reuse Distance Last-Touch Predictor** is further extended to a direct **Reuse Distance Predictor** by storing a prediction of the reuse distance itself, alongside the previously-observed last-touch signatures. Similarly, Petoumenos *et al.* propose a reuse distance predictor for improved cache replacement based upon tracking reuse distances of the most frequent instructions which access memory [PKK09]. Yet another cache replacement enhancement is proposed by Jaleel *et al.*, who predict the **re-reference interval** of cache lines [JTSE10].

Takagi and Hiraki propose **inter-reference gap distribution replacement**, a time-based cache replacement algorithm designed to improve upon traditional LRU replacement [TH04]. The **Inter-Reference Gap** (IRG) is defined as the time difference between successive references to a cache line. Time is tracked by the number of accesses to the cache, hence is not absolute. While each memory block has its own probability distribution of IRG, to reduce the overheads of storage, cache lines are classified into several

disjoint classes:

- **OT**: cache lines with a reference count of one
- **TT**: cache lines with a reference count of two
- **ST**: cache lines with a reference count greater than two
- **MT**: cache lines with a reference count greater than some threshold
- **PS**: cache lines with a stable IRG. This is determined when the difference between consecutive IRGs is less than some threshold and when this situation occurs consecutively at least a second threshold number of times.

The classes are listed in increasing order of priority such that a cache line is placed in the class with the highest priority whose criteria it satisfies. The IRG probability distribution for each class is divided into fixed size bins and the distribution statistics for each class is recorded at run-time. Using the probability distribution of each class, the IRG can be estimated, and the cache line chosen for replacement is the one with the smallest reciprocal of IRG.

Kharbutli and Solihin propose using counters per cache line to improve the performance of the cache replacement policy [KS05]. When a counter exceeds an adaptive threshold, the line is considered eligible for eviction. In the absence of any eligible lines, a fallback LRU policy is used. Two counters are proposed. The first increments on every access to the cache set in which the cache line is contained, while the second increments upon every access to the cache line itself. The two counter-based approaches are shown to yield significant performance improvements compared to both standard LRU and two sequence-based replacement policies — Lin and Reinhardt's last-touch history predictor [LF00] and Lai and Falsafi's dead block predictor [LFF01]. A replacement policy using absolute time shows a small overall performance improvement compared to the simple counter-based approach. All predictors are limited to approximately the same storage size of 64kB which goes some way towards explaining the lacklustre performance of the sequence-based predictors, acknowledging that large prediction tables are required for reasonable coverage and accuracy.

2.9 Cache Efficiency

The cache utilisation metric investigated in Chapter 3 is also known as **cache efficiency**, first defined by Burger *et al.* [BGK95]. While much subsequent research has cited this paper, very little work has directly addressed cache efficiency. One exception is Liu *et al.*'s recent work prefetching into cache lines which are predicted not to be referenced again prior to eviction, known as **dead** cache lines [LFHB08]. A **cache burst** is defined as the period of time between a cache line becoming MRU and subsequently becoming non-MRU, and cache bursts are found to be more predictable than regular references, hence last-touch predictors based on cache bursts have superior performance compared to last-touch predictors based on regular references. Notably, Liu *et al.* detail the quantitative improvement in cache efficiency which their scheme provides.

2.10 Summary

This chapter began by introducing several highly-significant technology trends which have influenced the development of microprocessors over the past fifty years and which are anticipated to continue to do so for the medium to long term future. The basics of cache architectures were reviewed and several approaches to improve cache performance and decrease power consumption were discussed focusing specifically on victim caches, prefetching and transitioning cache lines into low-power states. Finally, several other cache management techniques were reviewed which exemplify time-based approaches to improving cache performance, primarily by improving cache replacement decisions.

Cache Utilisation 3

3.1 Overview

The aim of this chapter is to define and examine various cache line lifetime metrics, including cache utilisation, and to examine the distribution and scaling of these metrics across various cache configurations.

This chapter begins with a description of the method used to explore the concepts and results presented in this dissertation. This method is first used to evaluate the impact of cache performance on overall system performance. The improvements made by a traditional cache hierarchy are identified and the remaining gap between scaling a traditional cache hierarchy and a perfect memory system is examined. Several cache line lifetime metrics are defined and their distributions are examined across different benchmarks. These metrics are based around the liveness of a cache line i.e. whether it will be accessed again prior to eviction, and include cache utilisation, a measure of the effectiveness or efficiency of the cache. Finally, the relationship between the cache miss rate and cache utilisation as well as the scaling of cache utilisation with conventional cache parameters are examined.

Subsequent chapters describe various approaches to predict the liveness of a cache line, and how these predictions may be exploited in optimisations designed to efficiently enhance cache and hence system performance by improving cache utilisation.

3.2 Method

Simulation is widely used in computer architecture due to the high cost[1] of fabricating actual semiconductor devices for architectural exploration. Simulation may be applied along a wide spectrum of models, typically trading off accuracy against execution time and development effort. Very detailed **circuit-level simulators** such as HSPICE and PSPICE employ sophisticated transistor models to enable highly-accurate results, but due to long execution times and complex parameterisation are only suitable for very small parts of a design. At a higher level, **Hardware Description Languages** (HDLs) such as Verilog and VHDL may be used to model larger systems, with correspondingly shorter execution times and development effort, but with less accurate results than circuit-level simulation. Finally, more abstract models of real hardware can be created using traditional high-level programming languages such as C++ or Java. While such abstract models are typically much quicker to develop and simulate compared to either circuit-level simulation or hardware description languages, care must be taken to

[1] Both financial cost and the cost in terms of time

ensure that the model does not implement functionality which would be unsuitable (in terms of delay, area or power consumption) to implement in actual hardware.

One hybrid approach between a full hardware implementation and a software simulation is hardware emulation by a suitably reconfigurable platform. This approach is taken by the RAMP (Research Accelerator for Multiple Processors) project in which Field-Programmable Gate Arrays (FPGAs) provide the reconfigurable substrate upon which a multiprocessor system is implemented [WPO+07]. In Wawrzynek *et al.*'s relative comparison of different approaches for parallel research, software simulation scores highly in all factors other than "credibility of result". However, this lack of credibility is largely due to the parallel nature of the system under examination, whereas the work in this dissertation concentrates on single processor systems.

Unless stated otherwise, all results in this dissertation were generated using a modified version of the M5 Simulator System v1 [BHR03] with a detailed processor model based upon the venerable SimpleScalar v3 `sim-outorder` Alpha model [ALE02]. The simulator operates in **syscall-emulation** mode, so any functionality that would be performed by the operating system is emulated by the host system, rather than being faithfully simulated. The alternative would be **full-system simulation**, not supported by M5 v1, in which an entire operating system is run upon the simulated processor, greatly increasing simulation times unless approaches such as **checkpointing** are used, whereby the entire system state can be persisted and recalled under different processor configurations. Since the work in this dissertation is primarily concerned with single benchmarks rather than multiprogramming, and since the benchmarks under consideration are designed to place few demands on the operating system itself, this approach is reasonable. Virtual memory, and hardware to support it, are not modelled.

The M5 Simulator processor core, memory hierarchy and buses were mostly unchanged. Additional statistics were added to keep track of live and dead cache lines. New simulation objects were created to model the victim cache and the various predictors evaluated, while the existing prefetcher was modified to model the prefetch victim selection scheme detailed in Chapter 5. Where appropriate, modifications were tested using unit tests of individual pieces of functionality, followed by bespoke hand-written assembler to perform larger-scale testing.

All results in this dissertation were generated by simulations running on the University of Cambridge Computer Laboratory's processor bank which consists of numerous virtual machines, each configured with a single processor and between 1 GB and 4 GB of memory. The Xen hypervisor is used for virtualisation, providing a reliable, high-performance virtual machine [BDF+03]. The number of machines available in the processor bank varies according to other system demands, but is usually between 40 and 90. The processor bank is managed using the Condor workload management system [TTL05].

3.2.1 Benchmarks

For the majority of this dissertation the popular SPEC CPU2000 suite of integer and floating point benchmarks is used [Hen00]. This suite is designed to measure the performance of the processor core, memory subsystem (excluding any disk-based virtual memory) and compiler. To clarify, the term **benchmark** (e.g. `gcc_scilab`) will be used

Functional Units	6 Integer ALUs 2 Integer multiply/divide units 4 Floating-point ALUs 2 Floating-point multiply/divide units 2 Memory read/write ports
Instruction Queue Size	128 instructions
Fetch/Decode/Issue/Commit Width	8 instructions/cycle
Reorder Buffer Size	128 instructions
Load/Store Queue Size	64 entries
Clock Frequency	4 GHz

Table 3.1: Summary of the baseline processor core parameters

to specify a particular **application** (e.g. `gcc`) with a particular **workload** (e.g. `scilab`). Almost all the simulated binaries are compiled with the Compaq compiler suite using the peak SPEC2000 settings. However, `gcc_200` is compiled using the standard GNU gcc compiler with peak SPEC2000 settings since the binary produced by the Compaq compiler suite exposes unresolved issues in the SimpleScalar processor core model. Details of individual benchmarks can be found in Appendix A. Unlike much other research, the full results for each individual application together with each individual workload for that application are considered separately. The benchmarks are executed using the reference set of inputs, with the first one billion cycles executed using a simplified processor model to warm up the caches and to bypass any initialisation code before running the next two billion cycles with a detailed processor model and recording the desired statistics[2]. All benchmarks produced some text or file output which was compared to the reference output provided by the SPEC CPU2000 suite in order to help verify functional correctness.

3.2.2 Baseline System Configuration

The full baseline system parameters are listed in Appendix B and summaries of the processor core, branch predictor and memory subsystem parameters are provided in Tables 3.1, 3.2 and 3.3 respectively. The latter consists of separate first-level (L1) caches for data (DL1) and instructions (IL1) and a unified second-level (L2) cache.

The parameters are based upon the Alpha 21264 microprocessor [Cor99], excluding the microclustered architecture, but with some updates to accommodate recent and shortly anticipated technology trends as previously discussed in Section 2.3. While these parameters may appear slightly conservative, they are aligned with the demands of the SPEC CPU2000 suite. For example, excessively increasing the L1 cache sizes beyond the benchmark's workload would result in a very sparse, unrepresentative, reference stream being presented to the L2 cache.

The load/store queue provides load-bypassing and store-to-load forwarding, but does not provide a load wait table as found in the Alpha 21264 microprocessor [Cor99].

[2]Except for `perlbmk_2` which is fast-forwarded for one billion cycles then runs to completion in 650 million cycles in the baseline processor configuration

Type	Hybrid local/global predictor
Meta-predictor size	4 kB
Global history register size	12 bits
Local history size	10 bits
Local history registers	1024
Branch target buffer size	4 kB
Branch target buffer associativity	4-way
Return address stack size	16 entries

Table 3.2: Summary of the baseline branch predictor parameters

Data Level 1 (DL1)	**Total size**	128 kB
	Line width	64 bytes
	Associativity	2-way
	Latency	2 cycles
	Replacement policy	Least recently used
	Prefetcher	None
Instruction Level 1 (IL1)	**Total size**	128 kB
	Line width	64 bytes
	Associativity	2-way
	Latency	1 cycle [a]
	Replacement policy	Least recently used
	Prefetcher	None
Unified Level 2 (L2)	**Total size**	1 MB
	Line width	64 bytes
	Associativity	8-way
	Latency	12 cycles
	Replacement policy	Least recently used
	Prefetcher	None
Main Memory	**Total size**	Assumed infinite
	Latency	200 cycles

[a] Actual Alpha 21264 IL1 latency is 2 cycles but this is mostly hidden by line and way prediction which are not modelled

Table 3.3: Summary of the baseline memory subsystem parameters

3.3 Quantifying the Impact of Memory Latency

The need to reduce the impact of increasing main memory latency is the primary motivation for the work described in this dissertation. This section examines to what extent current cache architectures can mitigate large main memory latencies. The overall performance gains that *would* be achieved by theoretically perfect caches is also investigated, demonstrating that there is still significant progress to be made in the arena of cache and memory subsystem design.

Figure 3.1 shows the average number of instructions committed per cycle (IPC) with the baseline configuration over the SPEC CPU2000 integer and floating-point benchmarks. The improvements that *would* be achieved by a perfect L2 cache and perfect L1 caches are also shown. In this case, a **perfect** cache is defined as a cache in which every access, including the first access to a cache line, hits with the fixed latency of that particular cache. The further improvements that would be made by a perfect memory subsystem in which every access hits in the cache with a fixed latency of a single cycle are also shown.

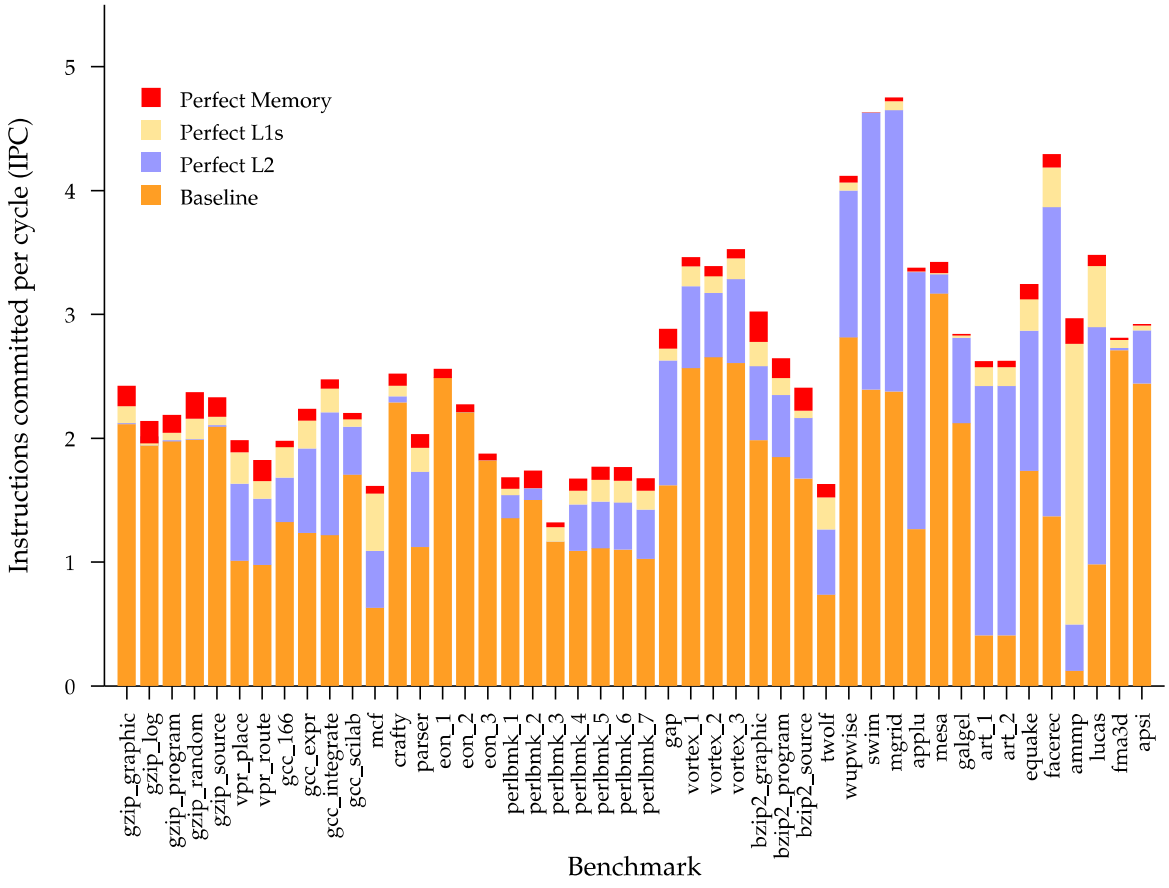

Figure 3.1: Average number of instructions committed per cycle (IPC)

It is interesting to note that the IPC attained varies considerably both between applications, as well as within applications but between different workloads, e.g. the seven `perlbmk` benchmarks and the five `gcc` benchmarks show markedly different results. Differing IPC values are to be expected between applications due to the inherent difference in the amount of **Instruction Level Parallelism** (ILP) that may be exploited, but differing IPC values within applications but between workloads is more unexpected and is an example of where previous research which considers only applications as a whole may be misleading.

For the majority of benchmarks, a perfect L2 cache improves IPC considerably, with an average of 26% for the integer and 150% for the floating-point benchmarks. As would be expected, those benchmarks with the greatest improvements are typically those which exhibit a high L2 miss rate, as shown in Figure 3.2. For example, `mcf` and `art` both exhibit high L2 miss rates and correspondingly higher IPC improvements. It is interesting to note that the floating-point benchmarks have a higher average L2 miss rate than the integer benchmarks, but they are sometimes omitted from previous research.

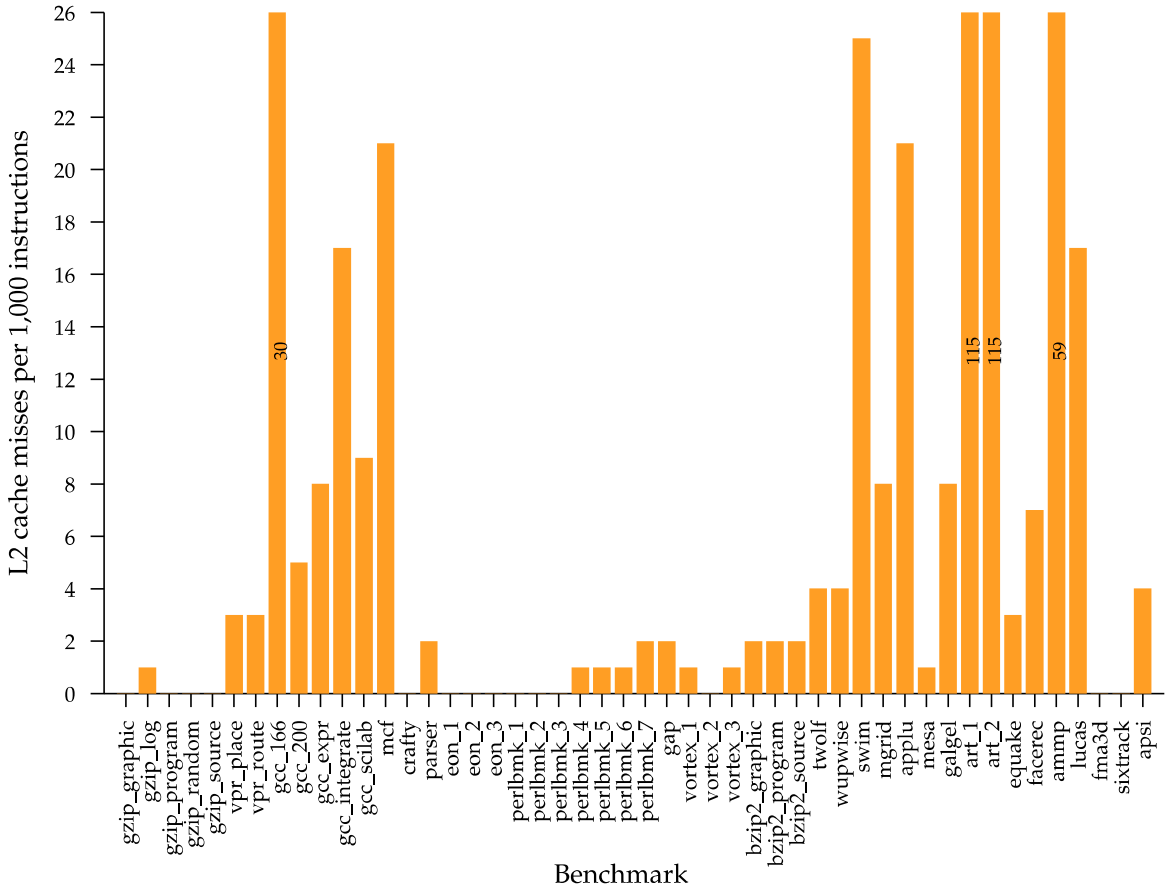

Figure 3.2: Number of L2 cache misses per 1,000 instructions committed

Perfect L1 caches show less of an incremental IPC improvement than for a perfect L2 cache, indicating that for the benchmarks studied, the processor is able to tolerate the

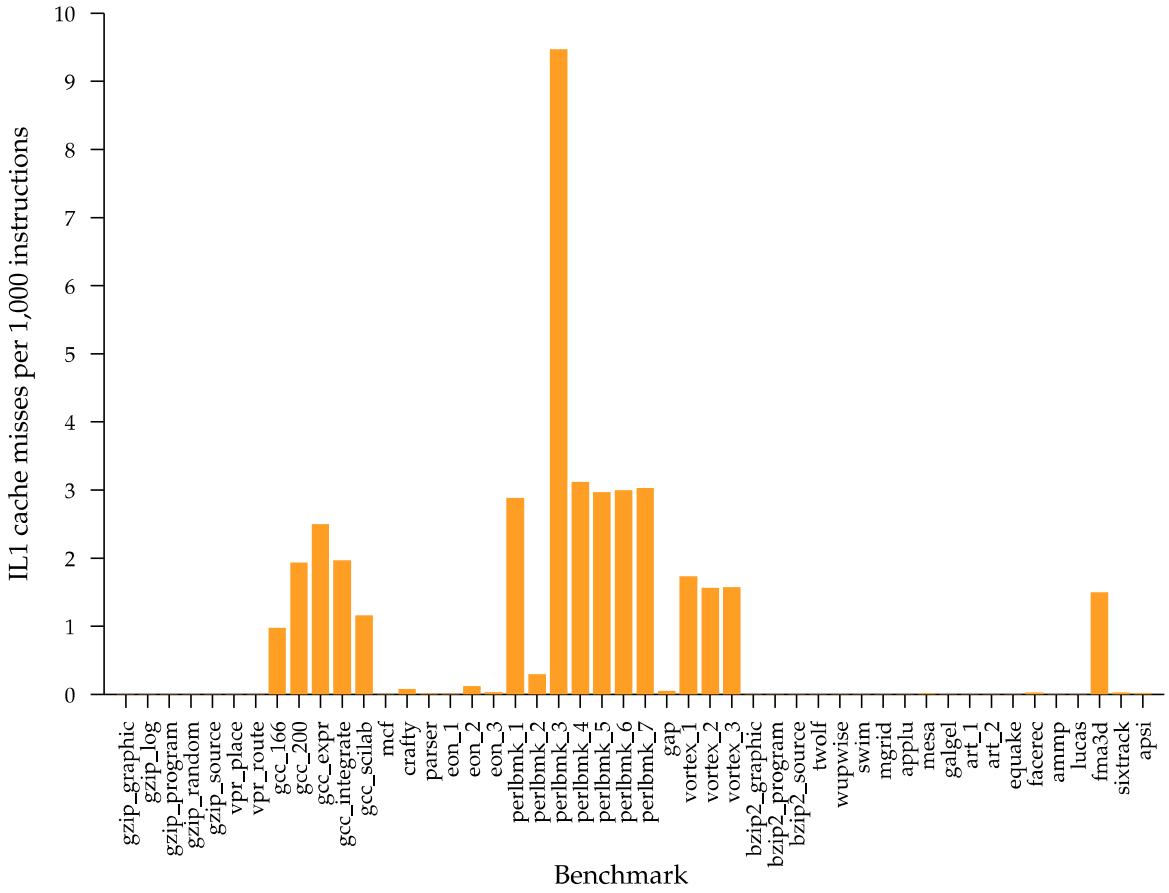

Figure 3.3: Number of IL1 cache misses per 1,000 instructions committed

small miss latency of the L1 caches. Drilling down into the individual miss rates for the L1 caches, the IL1 cache, shown in Figure 3.3, exhibits a uniformly low miss rate over all benchmarks. While the result for the `perlbmk_3` benchmark may look relatively high compared to the other benchmarks, the absolute value (9.5 misses per 1,000 instructions) is small compared to the results for the DL1 and L2 caches.

Compared to both the IL1 and L2 caches, the miss rates for the DL1 cache, shown in Figure 3.4, are considerably higher. The combined miss rate for the L1 caches is simply the sum of the miss rates of the DL1 and IL1 caches, and given the small IL1 miss rates is approximately equal to the DL1 miss rates. Indeed, again as would be expected, those benchmarks with high DL1 miss rates show correspondingly better IPC improvements with perfect L1 caches e.g. `mcf` and `ammp`, but in general the improvements are smaller than those achieved by a perfect L2 cache. `art` is an interesting example of the interaction between L1 and L2 caches. Both miss rates are very high for the application, indicating much traffic to main memory, but a perfect L2 cache yields a much greater IPC benefit than perfect L1 caches. This is because with a perfect L2 cache, the impact of a L1 cache miss is greatly reduced since the access will never require a lengthy access to main memory.

Finally, a perfect memory hierarchy in which every access is satisfied in a single cycle

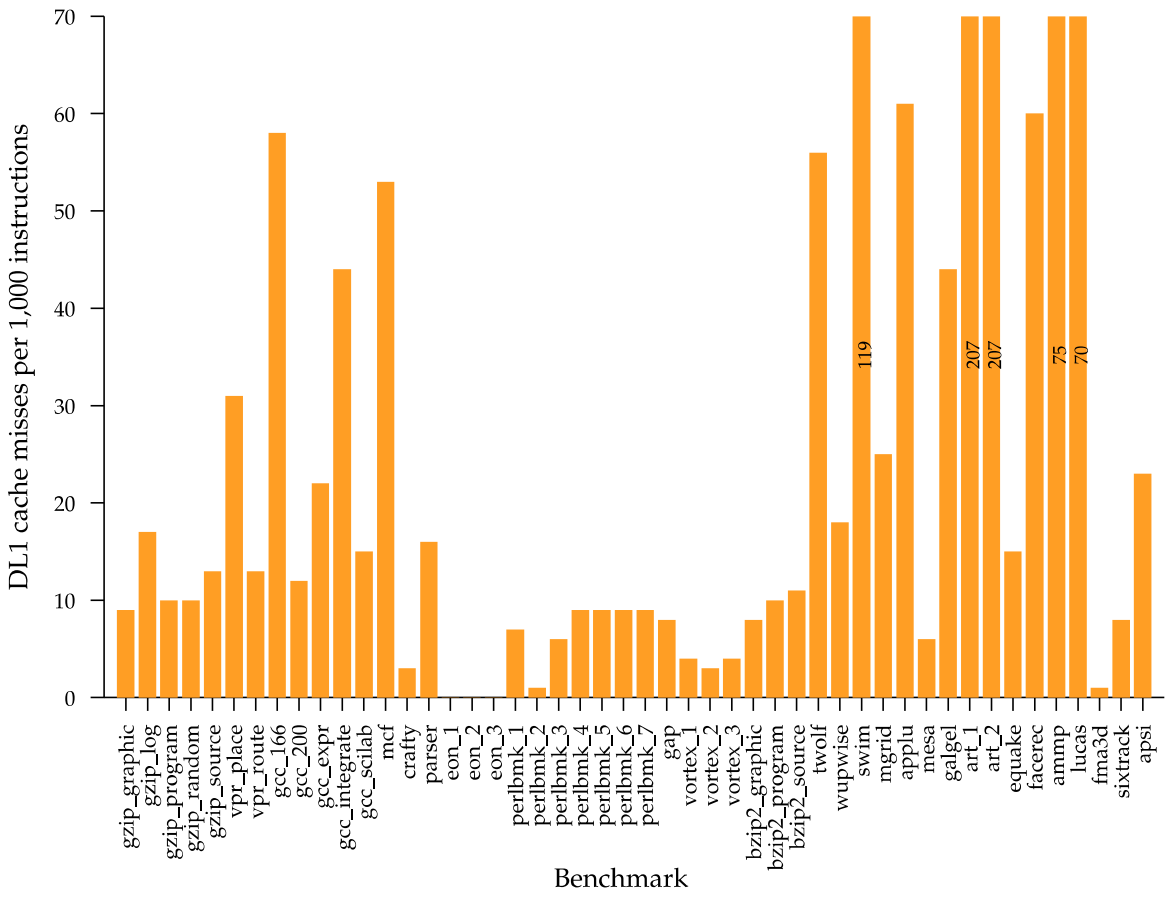

Figure 3.4: Number of DL1 cache misses per 1,000 instructions committed

shows marginal further improvement in most cases. Those few benchmarks where reasonable increases in IPC are observed are those which especially stress the memory hierarchy by requiring particularly large numbers of accesses to be satisfied with as little latency as possible e.g. `gzip` and `bzip2`.

At this point it is worth noting that there is still much scope for further overall performance improvement beyond enhancing the memory hierarchy. The theoretical maximum IPC of the baseline configuration is 8 instructions per cycle, however, other factors such as limited benchmark ILP, imperfect branch prediction and functional unit capabilities place additional limits on that which is achievable.

3.3.1 Scaling Cache Size and Associativity

Increasing cache size and associativity were two of the key technology trends previously identified in Section 2.3. The cache sizes and associativities are now scaled in line with these projected trends and the impact on overall system performance is examined. The L1 caches are scaled more conservatively than the L2 cache since minimising latency is of prime importance for L1 caches, whereas the significantly increased size and hence hit rate of the L2 cache can help outweigh any latency increases. Table 3.4 details the ten cache configurations examined. Latencies remain the same in all cases at the baseline values.

Figure 3.5 shows the contribution to IPC improvement made by each of the cache configurations relative to the previously defined perfect memory. For example, Configuration B increases the baseline IPC of `gzip_graphic` to approximately 40% of the increase that would be achieved with a perfect memory. Subsequent cache configurations (C to I) do little more to help, with Configuration I lagging the perfect memory by approximately 55%.

Changing cache configurations typically increases performance, but the size of the increase is very variable depending on the benchmark and the cache configuration being considered. Some benchmarks (e.g. `vpr_place`, `twolf` and `galgel`) benefit greatly from Configuration A, whereas others (e.g. `mcf`, `mesa` and `facerec`) require Configuration E or later to obtain significant performance improvements. Even Configuration I, the most extreme configuration, still leaves much room for further improvement for many benchmarks.

Cache Configuration	L1 Caches		L2 Cache	
	Size	Associativity	Size	Associativity
Baseline	128 kB	2-way	1024 kB	8-way
A	128 kB	2-way	2048 kB	8-way
B	256 kB	4-way	2048 kB	16-way
C	256 kB	4-way	4096 kB	16-way
D	512 kB	8-way	4096 kB	32-way
E	512 kB	8-way	8192 kB	32-way
F	1024 kB	16-way	8192 kB	32-way
G	1024 kB	16-way	16384 kB	64-way
H	2048 kB	32-way	16384 kB	64-way
I	2048 kB	32-way	32768 kB	128-way

Table 3.4: Cache hierarchy configurations

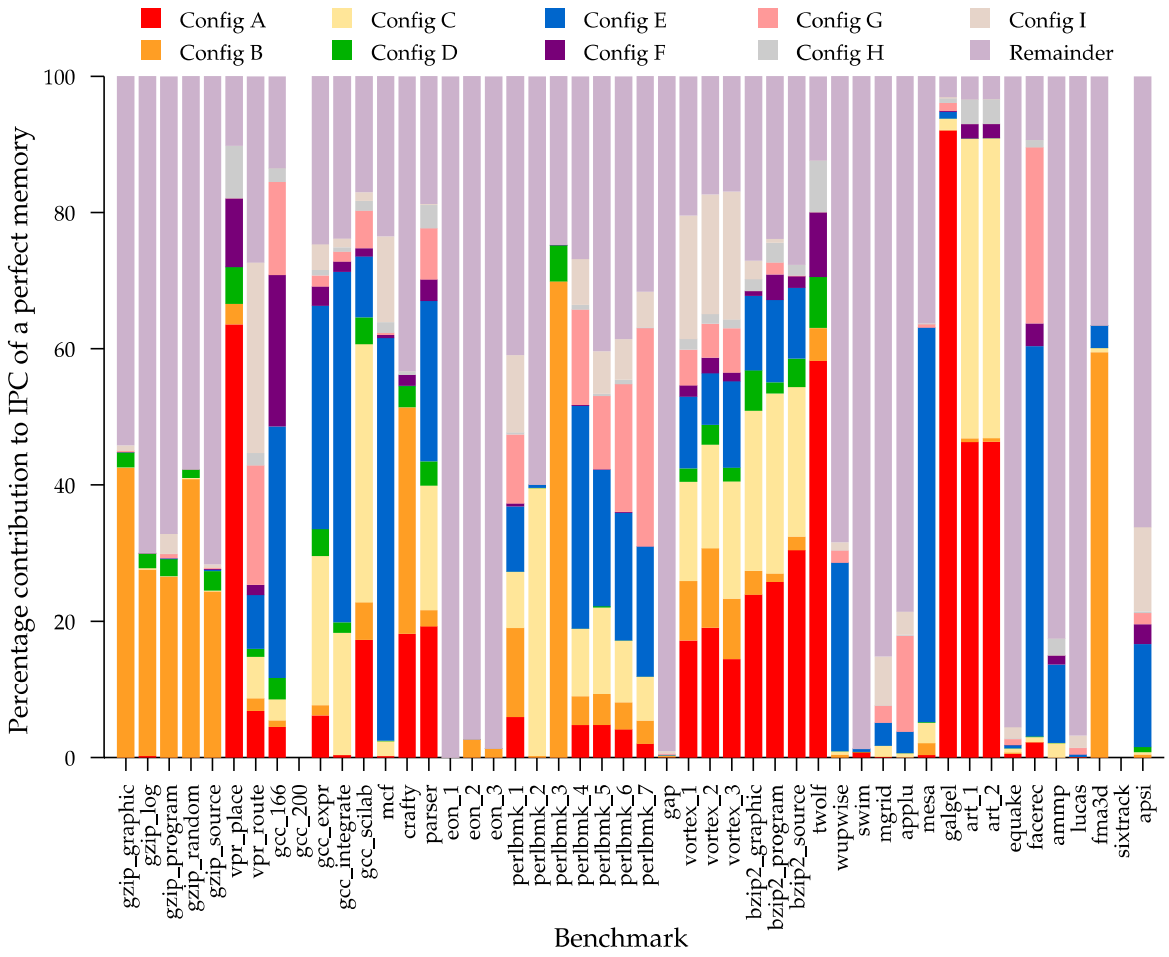

Figure 3.5: Contribution of different cache configurations towards the IPC of a perfect memory

In an isolated number of cases, increasing the cache associativity without increasing the cache size, e.g. for the L2 cache from Configuration A to Configuration B, can lead to small decreases in IPC since there is potentially more contention for fewer uniquely addressable cache lines. Such cases are of too small magnitude to be apparent in Figure 3.5.

In reality, access latencies could not be kept constant and would increase if cache size and/or associativity were increased, thus these results are very optimistic and IPC would rise far more slowly, probably decreasing in some cases as the performance detriment of an increased latency overtook the performance benefit of a decreased miss rate.

Figure 3.5 shows relative improvements in performance, and as such needs to be interpreted carefully. Recalling Figure 3.1, some benchmarks (e.g. `gzip`, `eon`, `mesa` and `fma3d`) show very little benefit from even a perfect memory, and typically have low miss rates in all three caches. Therefore even a large IPC improvement relative to that of a perfect memory may not correspond to a large absolute increase in performance. Other benchmarks (e.g. `vpr`, `gcc`, `art` and `facerec`) show large benefit from a perfect

memory, typically have higher miss rates in all three caches, and are helped by increasing cache size and associativity. This is most likely due to more conflict and capacity misses being made, compared to compulsory misses. One final group of benchmarks (e.g. `wupwise`, `swim`, `mgrid`, `applu`, `ammp` and `lucas`) also show large benefit from a perfect memory, also typically have higher miss rates in all three caches, but are not helped by increasing cache size and associativity. This is most likely due to more compulsory misses being made, compared to conflict or capacity misses. These latter cases demonstrate that even optimistic cache scaling rarely comes close to the performance improvements that would be made by a perfect cache, hinting that such naïve scaling, with associated power consumption and reliability concerns, is not the best approach to improving overall performance.

3.4 Describing the Lifetime of a Cache Line

In this section a set of definitions is developed to describe the lifetime of a cache line, using similar terminology to previous work by Hu *et al.* [HKM02]. A cache line is **born** when it is fetched from a lower level in the memory hierarchy and placed in the cache. Subsequent read and write hits occur during which time the cache line is known as **live**, with the period between consecutive hits being known as the **access interval**. At some later point in time the cache line is evicted to make space for a new line, and throughout the period between the last hit and the eviction the line is known as **dead**. Thus the period of time from the cache line being fetched until its last hit is the **live time**, and the period of time from its last hit until its eviction is the **dead time**. The whole period of time that the cache line is resident in the cache is known as its **generation**. These definitions are illustrated in Figure 3.6.

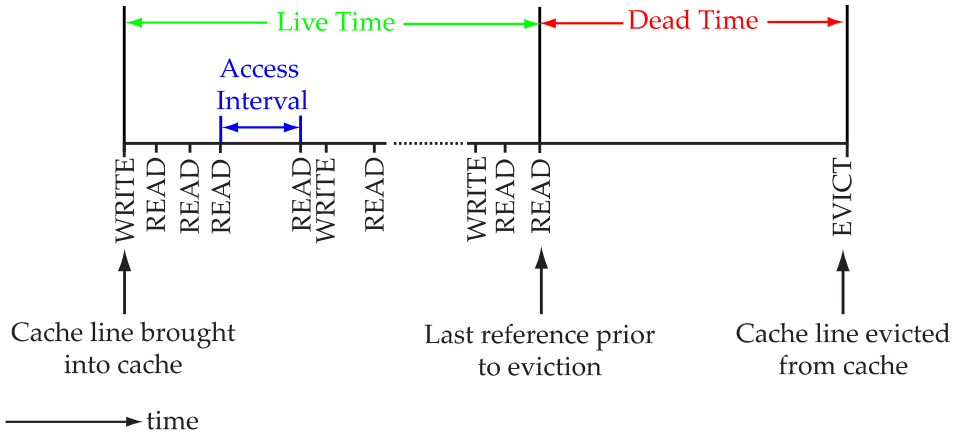

Figure 3.6: Cache line lifetime metric definitions

These metrics are now recorded for each of the benchmarks in the SPEC CPU2000 suite for the baseline configuration. For each cache and benchmark combination, the cumulative distribution of the metric is plotted. A cumulative plot is used in order to simultaneously examine the behaviour of multiple benchmarks since a traditional histogram would be unwieldy with 48 different benchmarks. Those benchmarks of

specific interest are highlighted in the legend. In addition, to capture both large and small scale behaviour, the majority of plots use a logarithmic scale on the x-axis. Some benchmarks demonstrate clustering of values, indicated by flat sections on the cumulative plot. One of the simplest cases is a **bimodal** distribution, observed by Hu *et al.* for all of the metrics *on average* [HKM02]. In this type of distribution, there are two clusters of values, usually at opposing ends of the range under examination. The boundary between these two clusters appears as a "step" on the cumulative plot. More than two clusters leads to multiple steps. Note that the use of the logarithmic scale can be misleading, and care must be taken to verify the rate of change of apparent steps, particularly those appearing towards the right-hand side of the plot.

Each of the cache line lifetime metrics is now examined in turn, and the cumulative distributions for each of the three caches in the memory hierarchy is examined and discussed. Critically, the live and dead times of a cache line can only be determined once that line has been evicted, since at an arbitrary point in time the next event for a cache line may be a hit (indicating that the line was live) or an eviction (indicating that the line was dead). At the end of the simulation, the generations of each cache line are artificially ended to ensure that cache lines with very long live or dead times are not inadvertently missed, since these values will only be known when the cache line is evicted. By assuming cache lines are dead at the end of the simulation, an inaccuracy is potentially introduced. However, given the vast number of different generations encountered during the benchmark compared to those still resident in the cache at the end of the simulation, this inaccuracy is negligible.

3.4.1 Live Time Distributions

Figures 3.7, 3.8 and 3.9 show the cumulative distributions of live times for the DL1, IL1 and L2 caches respectively. All three plots use a logarithmic scale on the x-axis and show a wide variety of differing behaviour for the different benchmarks. Each of the three caches is now discussed in turn.

DL1 Cache

Figure 3.7 shows the cumulative distributions of live times for the DL1 cache for each of the benchmarks. The proportion of cache lines with very short live times (defined here as less than 10 cycles) varies considerably between benchmarks, from 2.5% (`gcc_integrate`) to 99.3% (`ammp`). Conversely, the proportion of cache lines with very long live times (defined for the L1 caches as 10,000 cycles or more) varies almost as much, from 0.0016% (`art_1`) to 61% (`eon_3`). In between, a variety of behaviour is observed. Some benchmarks show multiple steps characteristic of a clustered distribution (e.g. `gcc_166` and `mgrid`) whereas others show more continuous behaviour (e.g. `crafty` and `gzip_program`).

Hu *et al.* report that, *on average*, 58% of live times are less than 100 cycles [HKM02]. At the same boundary, Figure 3.7 shows substantial variation between benchmarks, diminishing the significance of quoting just an average figure, with between 14% (`mgrid`) and 99.4% (`ammp`) of cache lines having live times less than 100 cycles.

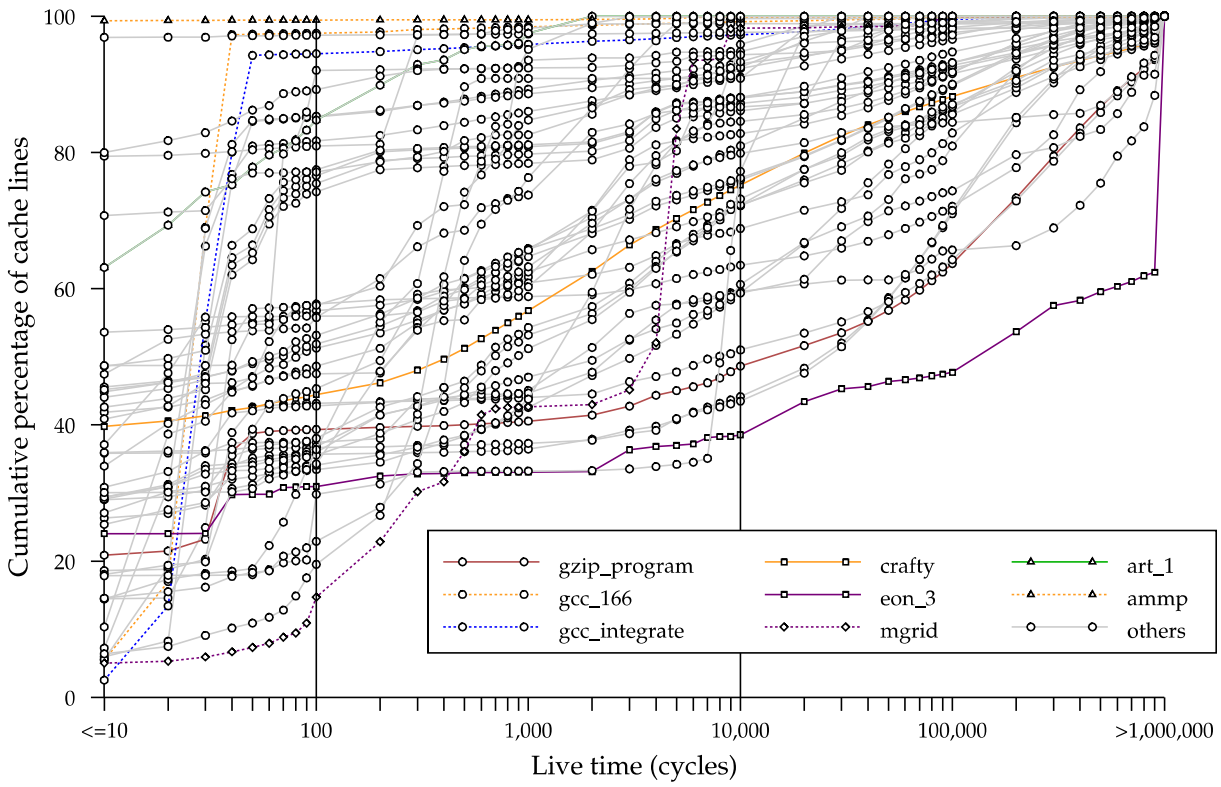

Figure 3.7: Cumulative distributions of live times for the DL1 cache

IL1 Cache

Figure 3.8 shows the cumulative distributions of live times for the IL1 cache for each of the benchmarks. The range of very short live times is slightly narrower than the range for the DL1 cache, whereas the range of very long live times is considerably wider than the DL1 cache, with 7.4% (`perlbmk_2`) to 93% (`applu`) of cache lines having live times greater than 10,000 cycles.

The overall trend is for IL1 live times to be considerably longer than DL1 live times. This is to be expected since the instruction cache typically exhibits more repetitive access patterns, hence better cache line reuse than the data cache, so cache lines are resident longer whilst actively being referenced. At the 100 cycle boundary previously discussed, between 4.9% (`applu`) and 82% (`art_1` and `art_2`) of lines are live.

Compared to the DL1 results, there are fewer pronounced steps other than for a few selected benchmarks (e.g. `swim` and `fma3d`) indicating that the IL1 live times are not as clustered as the DL1 live times.

L2 Cache

Figure 3.9 shows the cumulative distributions of live times for the L2 cache for each of the benchmarks. The first observation is that the proportion of cache lines with very short live times varies hugely between benchmarks, from 0.62% (`perlbmk_2`) to 99.2%

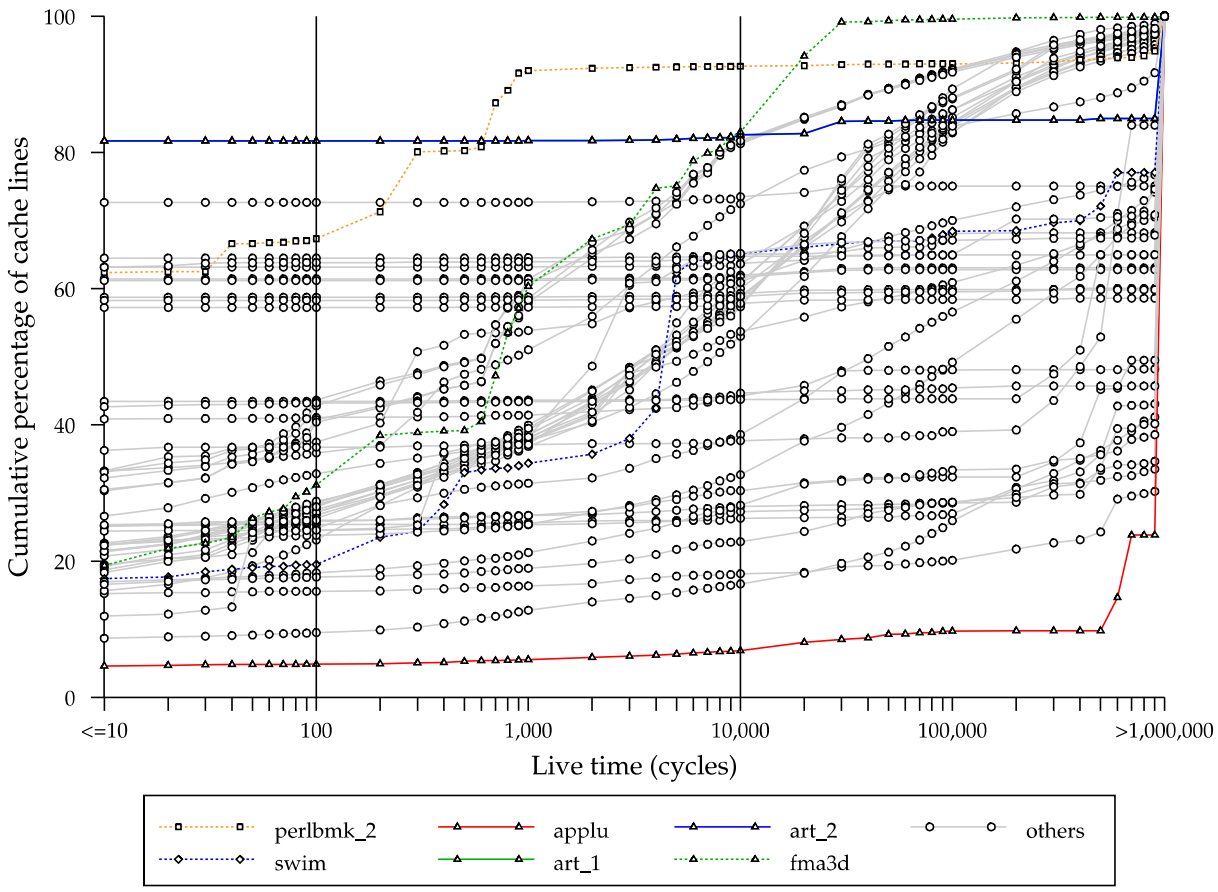

Figure 3.8: Cumulative distributions of live times for the IL1 cache

(ammp). Almost all benchmarks show no live times between 10 and 10,000 cycles as indicated by the flat lines on the cumulative plot. Above 10,000 cycles, some benchmarks show sudden increases (e.g. facerec, mesa and fma3d) whereas the increase in the proportion of cache lines with longer live times is slower for other benchmarks (e.g. vortex_2 and crafty).

The proportion of cache lines with long live times shows similar behaviour as for the IL1 cache, with between 0% (swim) and 99.4% (perlbmk_2) of cache lines having live times greater than 1,000,000 cycles. The overall tendency is for L2 live times to be considerably longer than both DL1 and IL1 live times. This is to be expected since the L2 cache reference stream is filtered by the inclusive L1 caches, therefore cache accesses are fewer and sparser, leading to longer live times.

3.4.2 Dead Time Distributions

Figures 3.10, 3.11 and 3.12 show the cumulative distributions of dead times for the DL1, IL1 and L2 caches respectively. Note that like the live time plots previously discussed, Figures 3.10 and 3.11 use a log scale on the x-axis, however, Figure 3.12 uses a linear scale due to the nature of the distribution. The first general observation is that dead

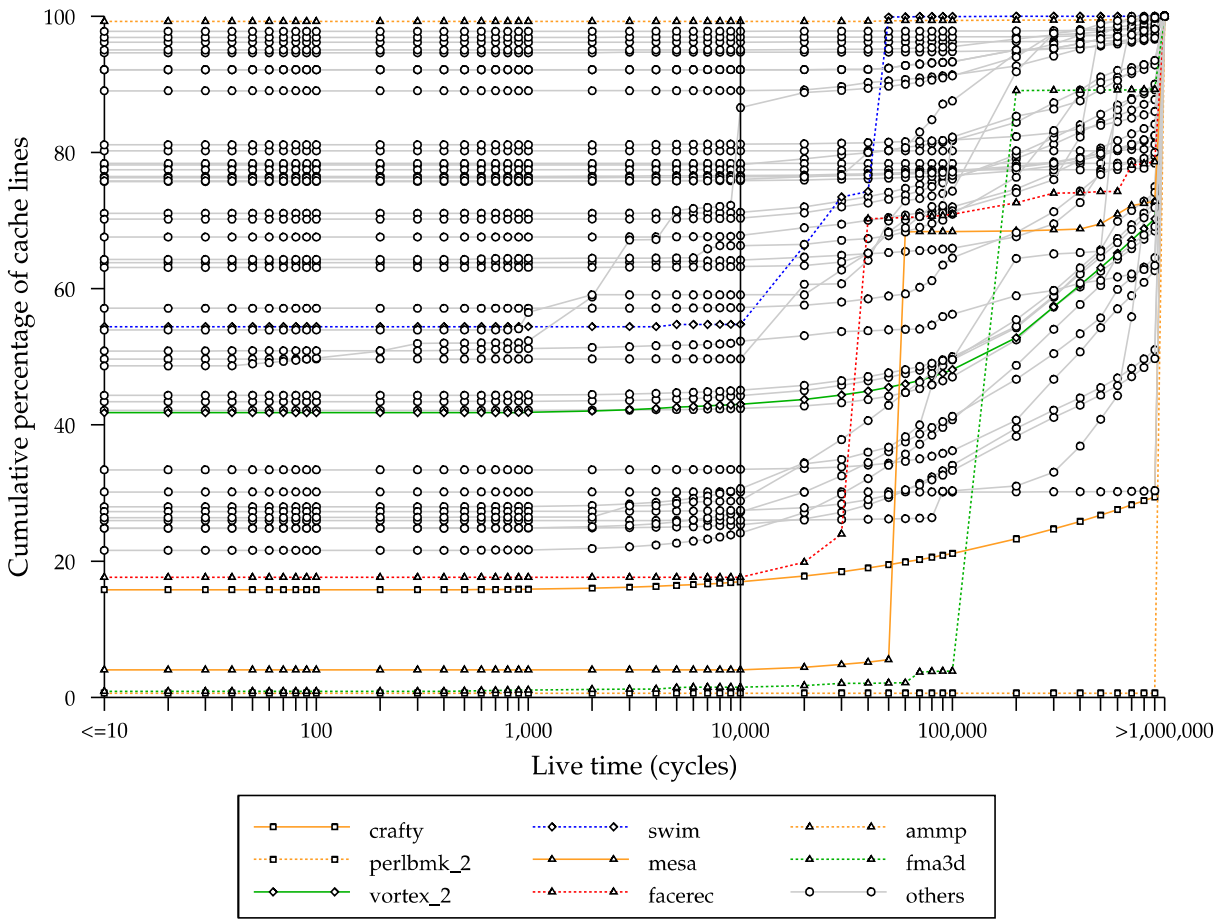

Figure 3.9: Cumulative distributions of live times for the L2 cache

times are, on average, considerably longer than live times, indicating **generational** behaviour. This term is borrowed from the field of garbage collection and is defined by Kaxiras *et al.* as "a flurry of use when first brought in, and then a period of dead time between their last access and the point where a new data item is brought into that cache location" [KHM01]. Generational behaviour was also found by Hu *et al.* [HKM02]. The distributions of dead times are also highly variable between different benchmarks and caches, and the details of each is now discussed in turn.

DL1 Cache

Figure 3.10 shows the the cumulative distributions of dead times for the DL1 cache for each of the benchmarks. Compared to the live times for the DL1 cache, dead times are, on average, one to two orders of magnitude larger. Hu *et al.* consider a dead time of less than 100 cycles to be small, finding 31% of dead times less than this value. Figure 3.10 shows that between 0% (`art_1` and `art_2`) and 12% (`equake`) of cache lines have dead times less than this threshold. Some steps are visible on the plot (e.g. `gcc_166`, `lucas` and `perlbmk_6`), indicative of clustering. The proportion of cache lines with dead times between 10,000 and 100,000 cycles rises rapidly for the majority

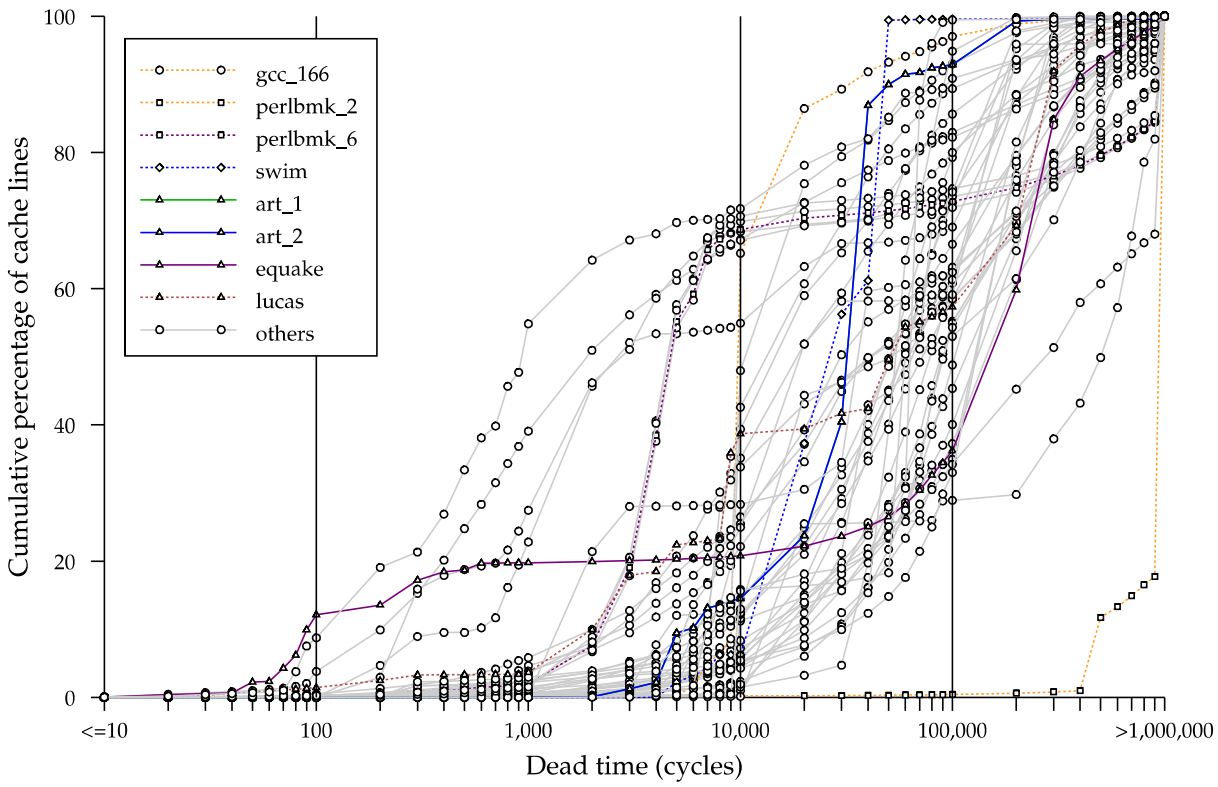

Figure 3.10: Cumulative distributions of dead times for the DL1 cache

of benchmarks. Between 0.44% (`swim`) and 99.6% (`perlbmk_2`) of cache lines have dead times greater than 100,000 cycles.

IL1 Cache

Figure 3.11 shows the the cumulative distributions of dead times for the IL1 cache for each of the benchmarks. Two distinct behaviours are apparent — approximately half of the benchmarks (e.g. `eon_2` and `perlbmk_1`) have a distribution of IL1 dead times shorter than the DL1 dead times while the other half of the benchmarks (e.g. `gzip_log` and `bzip2_program`) have a distribution of dead times considerably longer than the DL1 dead times. At the 100 cycle boundary previously discussed, the IL1 behaviour is somewhat similar to the DL1 behaviour, with 0% (`vpr_place` and several others) to 3.5% (`eon_1`) of cache lines having dead times less than 100 cycles. From the 1,000 cycle point onwards, the proportion of cache lines with longer dead times rises quickly for some benchmarks but much more slowly for others, with between 0.054% (`fma3d`) and 99.9% (`art_1` and `art_2`) of cache lines have dead times greater than 100,000 cycles.

L2 Cache

Figure 3.12 shows the cumulative distributions of dead times for the L2 cache for each of the benchmarks. Again, it is clear that overall dead times are considerably

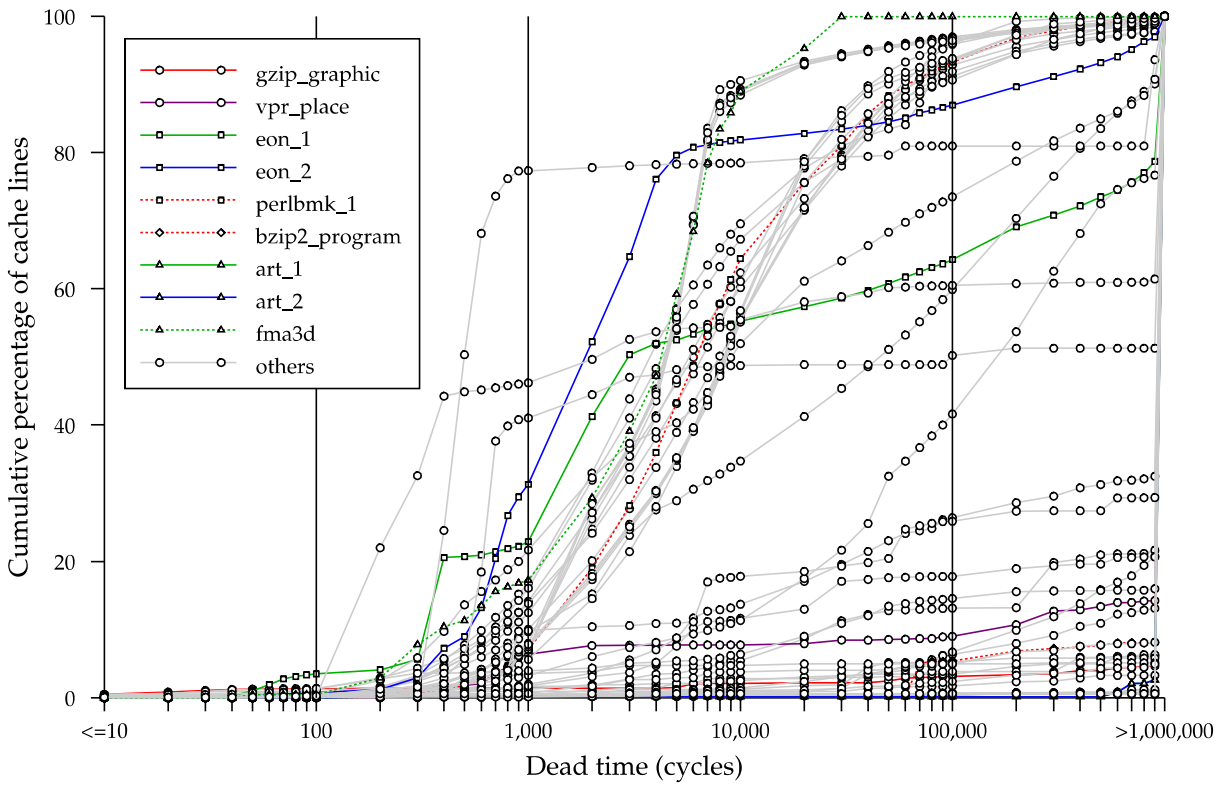

Figure 3.11: Cumulative distributions of dead times for the IL1 cache

longer than live times. Unlike the live time distribution for the L2 cache, the proportion of cache lines having short dead times (less than 100 cycles) is very small, from 0% (`gzip_log` and numerous others) to 0.00011% (`crafty`). Some benchmarks show evidence of clustering between dead times of 200,000 and 400,000 cycles (e.g. `art_2`, `swim` and `mesa`) whereas others show more continual increases (e.g. `facerec`, `bzip2_graphic` and `sixtrack`). The proportion of cache lines with dead times greater than 900,000 cycles varies widely, from 0.015% (`swim`) to 99.9% (`eon_3`).

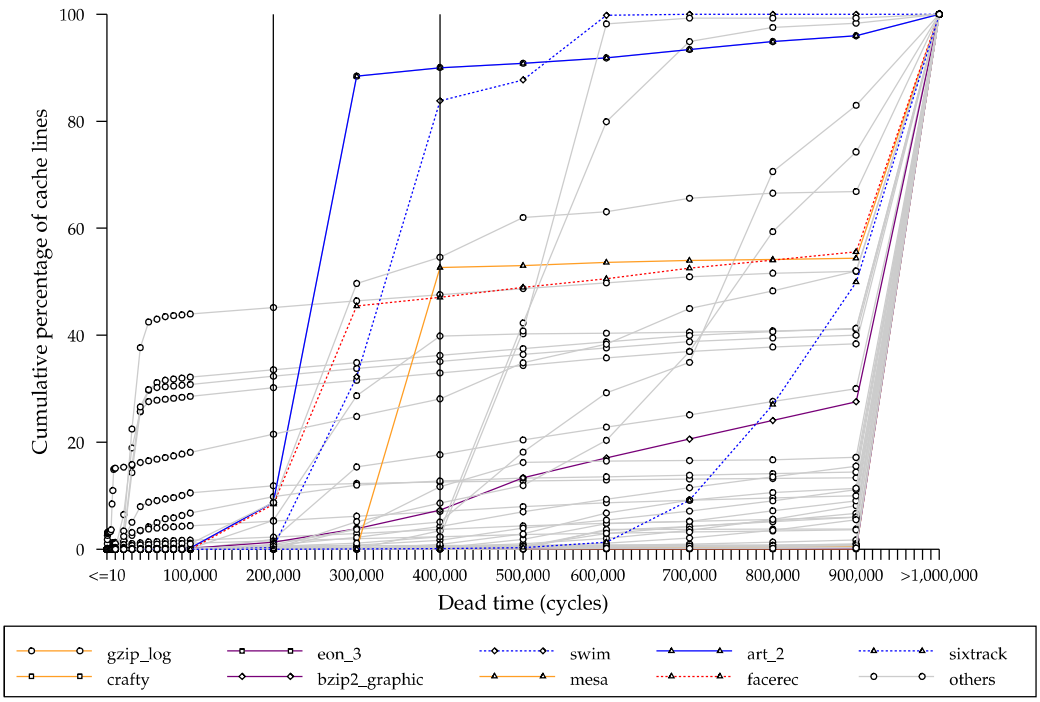

Figure 3.12: Cumulative distributions of dead times for the L2 cache

3.4.3 Access Interval Distributions

Figures 3.13, 3.14 and 3.15 show the cumulative distributions of access intervals for the DL1, IL1 and L2 caches respectively. All three plots use log scales on the x-axis, however the range of the DL1 and IL1 plots is considerably narrower than previously (up to 10,000 cycles) whereas the L2 plot uses the full range (up to 1,000,000 cycles). Compared to the live and dead time distributions previously discussed, there is generally more consistency between benchmarks, particularly for the DL1 and IL1 caches. As would be expected from their definition, access intervals are much smaller than either live or dead times, indicating that cache lines are repeatedly accessed numerous times prior to eviction. The access interval distributions for each of the caches is now discussed in turn.

DL1 Cache

Figure 3.13 shows the cumulative distributions of access intervals for the DL1 cache for each of the benchmarks. For all benchmarks there is a significant proportion of cache lines with very short access intervals (in this case defined to be less than 10 cycles). This value varies from 45% (`art_1` and `art_2`) to 92% (`gcc_166`). From the start, the rate of increase across all benchmarks remains reasonably constant and by 150 cycles all benchmarks are within 20% of each other. There is little evidence of steps in the plot, other than for `galgel`, indicating that access intervals for the DL1 cache are not generally clustered. Between 0.0018% (`art_1`) and 2.5% (`gzip_random`) of cache lines have access intervals in excess of 10,000 cycles.

IL1 Cache

Figure 3.14 shows the cumulative distributions of access intervals for the IL1 cache for each of the benchmarks. Similar trends are shown as for the DL1 cache, though with slightly less consistency between benchmarks. Between 57% (`vortex_3`) and 95% (`galgel`) of cache lines have access intervals of less than 10 cycles. The cumulative proportion of cache lines with a given access interval then increases somewhat uniformly across benchmarks. Few clear steps are observed, indicating again that access intervals for the IL1 cache are not generally clustered. Some benchmarks show remarkably uniform increases over a wide range (e.g. `vortex_3`) whereas others are more sporadic (e.g. `mgrid`). At the upper limit of the x-axis, between 0.0040% (`lucas`) and 1.6% (`perlbmk_3`) of cache lines have access intervals greater than 10,000 cycles.

L2 Cache

Figure 3.15 shows the cumulative distributions of access intervals for the L2 cache for each of the benchmarks. Note that this figure is plotted over a larger range of access intervals than the previous two figures. There is considerably more variation between benchmarks than the previous distributions for the DL1 and IL1 caches. As would be expected, the access intervals for the L2 cache are significantly longer than the access intervals for the DL1 and IL1 caches, reflecting the fact that the L2 cache is accessed

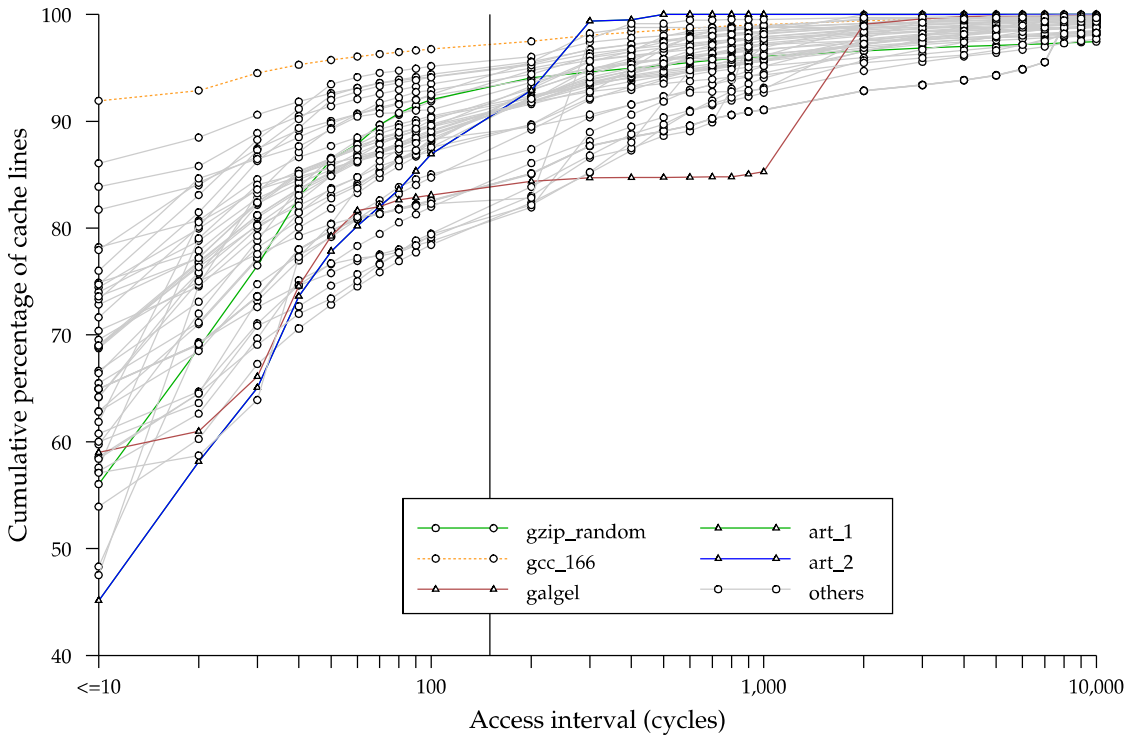

Figure 3.13: Cumulative distributions of access intervals for the DL1 cache

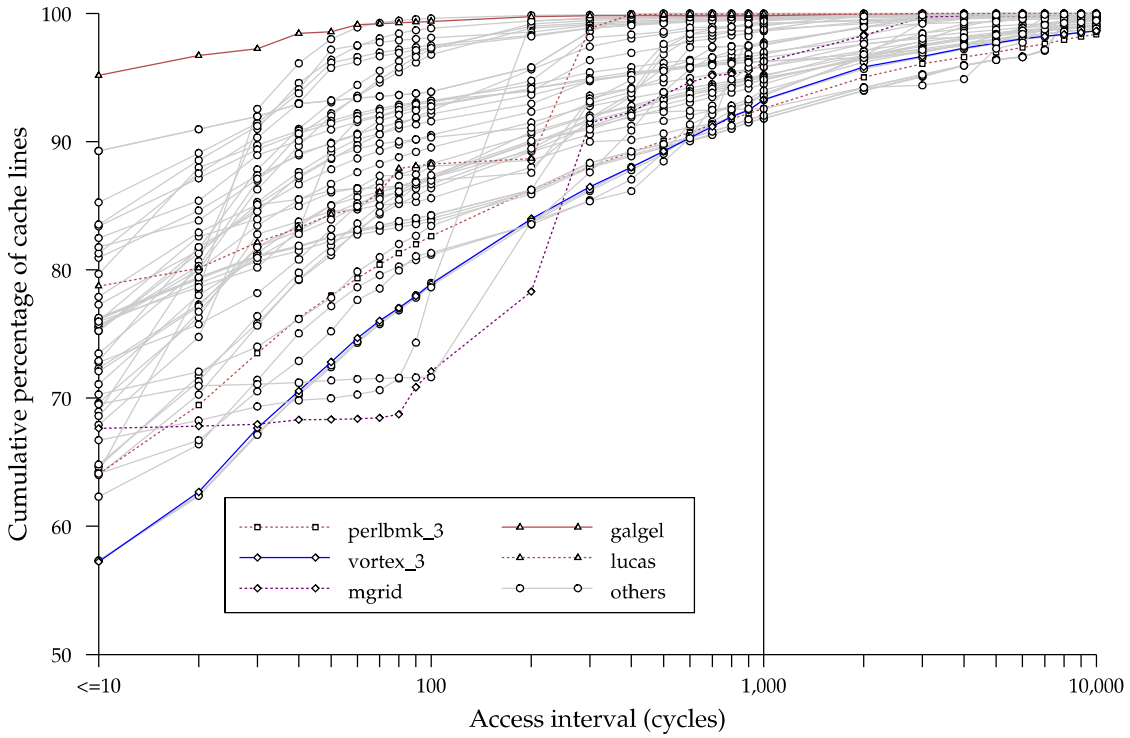

Figure 3.14: Cumulative distributions of access intervals for the IL1 cache

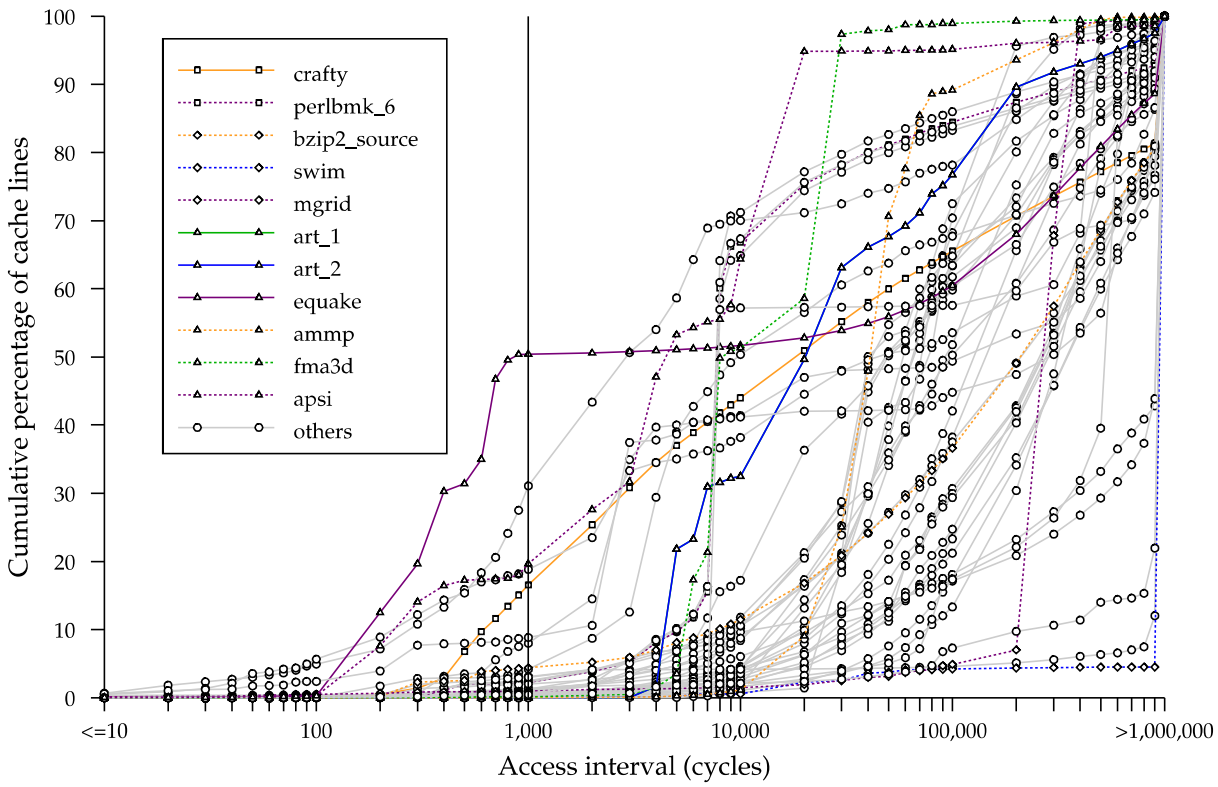

Figure 3.15: Cumulative distributions of access intervals for the L2 cache

much less frequently due to the filtering effect of the L1 caches. The proportion of cache lines with short access intervals (defined here as less than 1,000 cycles) varies between 0% (`art_1` and `art_2`) and 50% (`equake`). Some benchmarks show steps on the cumulative plot, indicating clustered behaviour (e.g. `fma3d`, `apsi` and `mgrid`) whereas others show more continuous increases (e.g. `crafty`, `perlbmk_6` and `bzip2_source`). At the limit of the x-axis, between 0.14% (`ammp`) and 96% (`swim`) of cache lines have access intervals in excess of 900,000 cycles, though the majority of benchmarks exhibit a proportion of less than 30%.

3.5 Cache Utilisation

Having distinguished live and dead cache lines, it is now possible to determine what proportion of the overall cache is actively in use at any point in time, an indication of how well utilised or effective the cache is.

One simple metric is to take the ratio of the cumulative live time to the cumulative generation time (i.e. the sum of the cumulative live time and the cumulative dead time) for the whole cache. These formulas are formally defined in Equations 3.1 and 3.2.

$$\text{Cache utilisation} = \frac{\text{Cumulative live time}}{\text{Cumulative generation time}} \tag{3.1}$$

$$\text{Cumulative generation time} = \text{Cumulative live time} + \text{Cumulative dead time} \tag{3.2}$$

Such a metric takes no account of varying behaviour over time, but does provide an estimate of the potential for improving cache performance or decreasing power consumption by exploiting such information. This ratio is now examined for the cache hierarchy of the baseline configuration.

Based on the metrics previously discussed, namely the cache miss rate and the overall IPC, it is not immediately clear what utilisation results are to be expected. A high utilisation, caused by long live times compared to dead times, may be accompanied by a low miss rate (high data reuse throughout the cache) or a high miss rate (much "churn" in the cache with many evictions). A low utilisation, caused by short live times compared to dead times, may be accompanied by a low miss rate (high data reuse but concentrated on a small portion of the cache) or a high miss rate (low data reuse throughout the cache). Thus utilisation is influenced not only by the magnitude of the hit rate, but also how widely distributed in the cache those hits are being made.

3.5.1 DL1 Cache

Figure 3.16 shows the cache utilisation of the DL1 cache for each benchmark. The utilisation varies greatly over the benchmarks, from 0.29% (`art_1` and `art_2`) to 72% (`perlbmk_3`). Somewhat surprisingly, cache utilisation also varies considerably within applications, but with different workloads where similar results might be expected e.g. for the `gzip` benchmarks. The average utilisation across all benchmarks is 25%.

In general, if a benchmark has a low miss rate, it will have a high utilisation and *vice versa*, but this relationship is not always consistent across all benchmarks. For example, while `art_1`, `art_2` and `lucas` all have high miss rates and low utilisation, `gap` has a low miss rate and low utilisation.

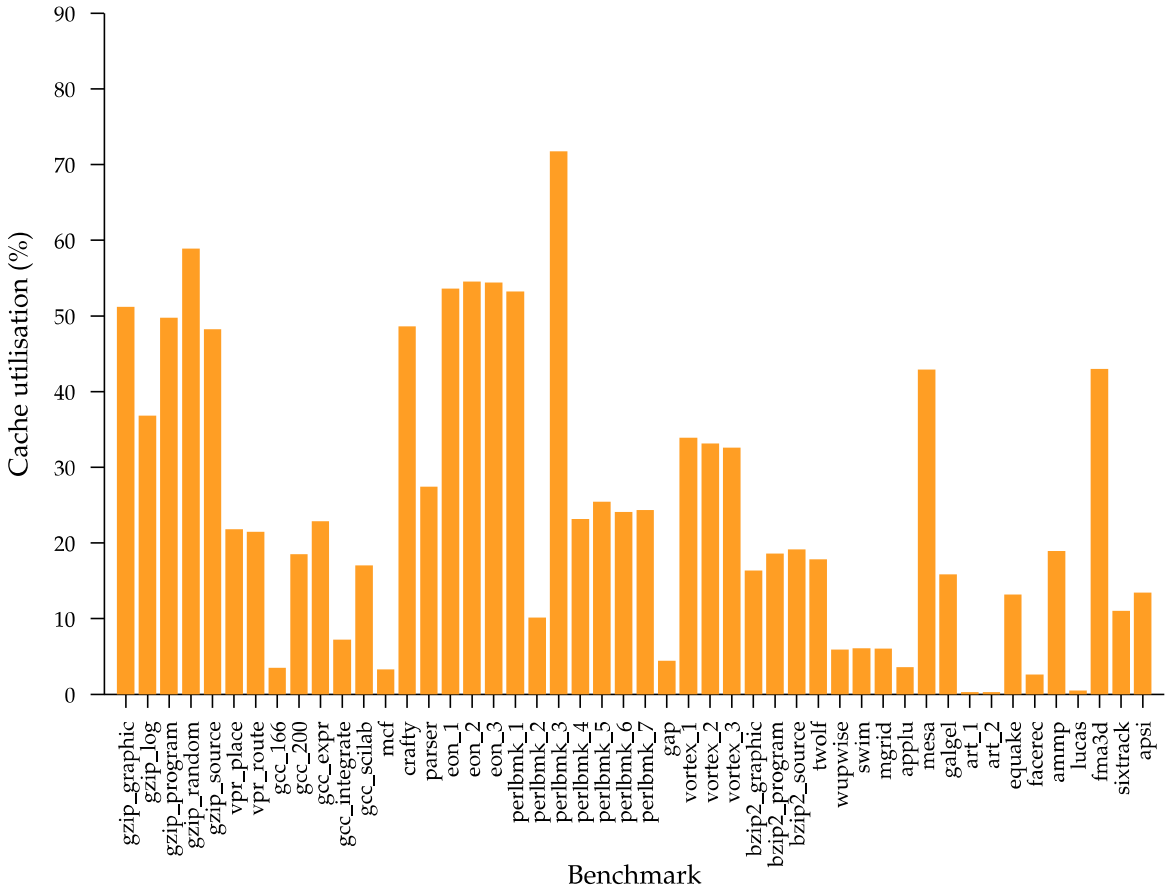

Figure 3.16: Cache utilisation in the DL1 cache

3.5.2 IL1 Cache

Figure 3.17 shows the cache utilisation of the IL1 cache for each benchmark. The utilisation is generally more consistent than in the DL1 cache, with numerous benchmarks between 60% and 75%, peaking at 78% for `perlbmk_3`. Interestingly, `perlbmk_3` also has the highest miss rate. A few benchmarks show comparatively poor utilisation, with minima of 7.7% (`art_1` and `art_2`), 13% (`wupwise`) and 16% (`gzip_random`). No clear relationship between the cache miss rate and utilisation is apparent. The average utilisation is 48%, considerably higher than for the DL1 cache.

3.5.3 L2 Cache

Figure 3.18 shows the cache utilisation of the L2 cache for each benchmark. The overall range is similar to the range of the DL1 cache utilisation, from 2.9% (`lucas`) to 79% (`crafty` and `sixtrack`) with an average of 27%. However, the relationship of low miss rates being coupled with high utilisation and *vice versa* observed in the DL1 cache is not as apparent in the L2 cache. In numerous cases (e.g. `eon`, `gap` and `mesa`), a low miss rate accompanies a low utilisation.

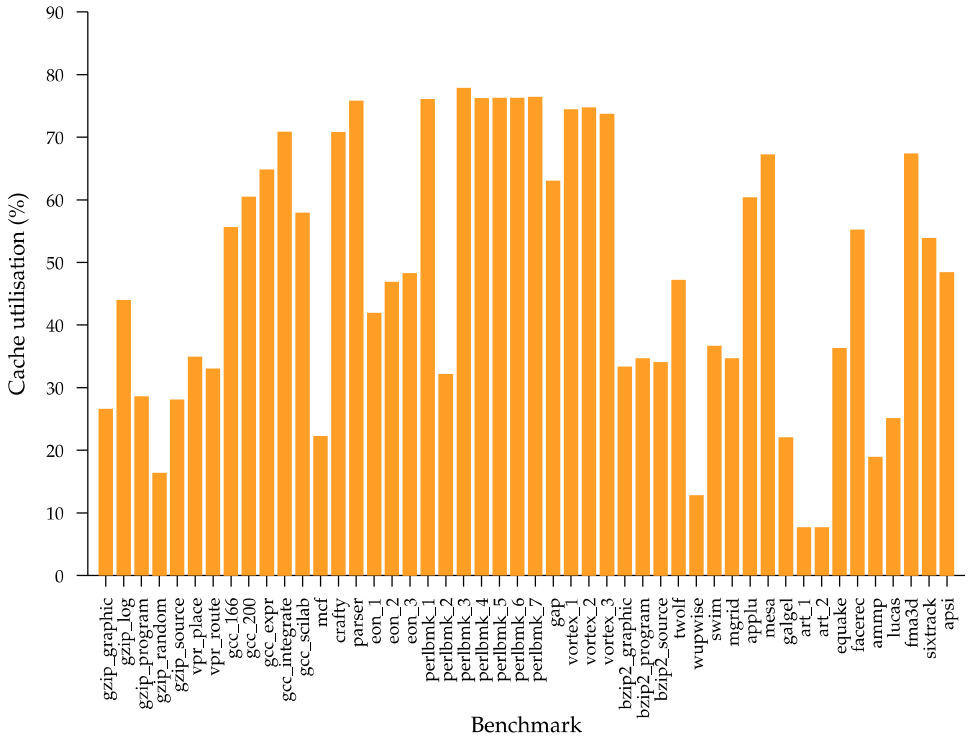

Figure 3.17: Cache utilisation in the IL1 cache

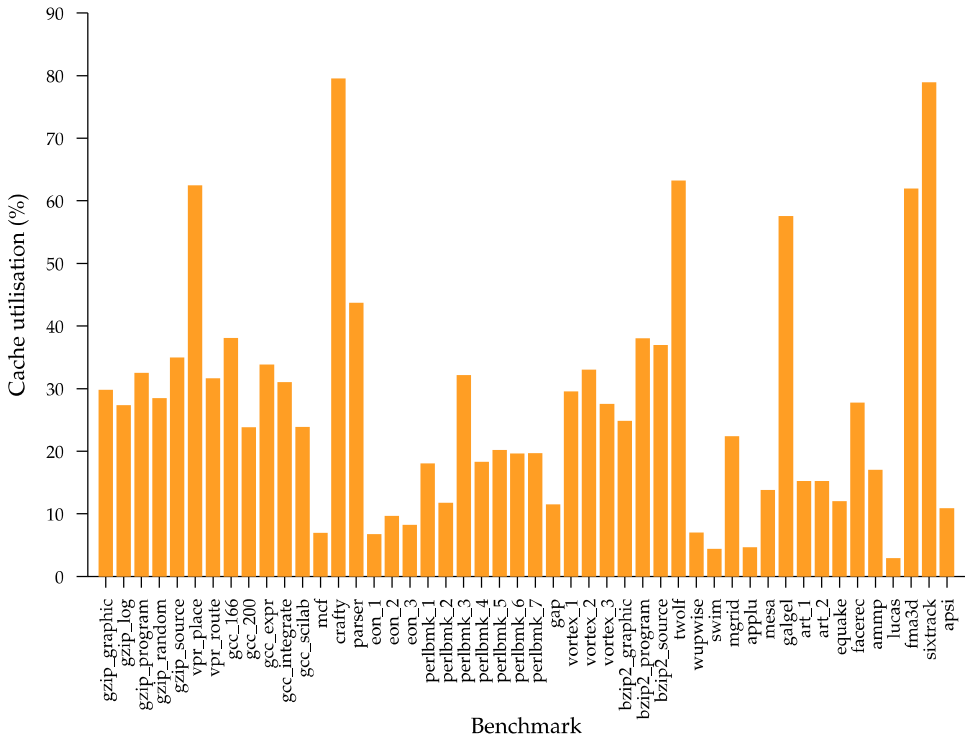

Figure 3.18: Cache utilisation in the L2 cache

3.5.4 Relationship Between Cache Miss Rate and Cache Utilisation

The relationship between cache miss rate and cache utilisation warrants further investigation, since the majority of current cache performance optimisations are aimed at decreasing the miss rate rather than improving utilisation. Figures 3.19, 3.20 and 3.21 plot the cache utilisation against the cache miss rate (expressed as the number of misses per 1,000 instructions) for the DL1, IL1 and L2 caches respectively. Since only the general trends are of interest, rather than specific results, and for clarity, individual benchmarks are not labelled.

DL1 Cache

Figure 3.19 plots the cache utilisation against the cache miss rate of the DL1 cache for each benchmark. For benchmarks with low miss rates, cache utilisation is variable, from approximately 5% to 70%, while for benchmarks with higher miss rates, cache utilisation is generally low, below approximately 20%. There are no benchmarks with high cache miss rates and correspondingly high cache utilisations.

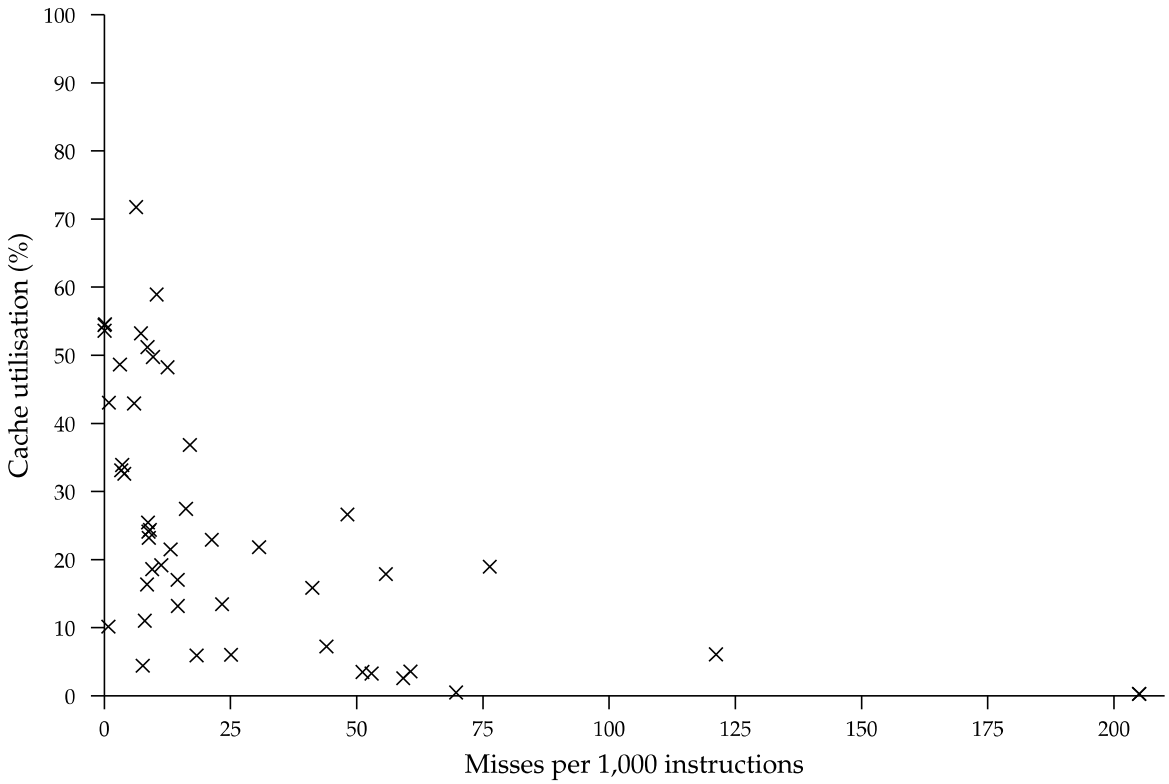

Figure 3.19: Cache utilisation *vs* miss rate in the DL1 cache

IL1 Cache

Figure 3.20 plots the cache utilisation against the cache miss rate of the IL1 cache for each benchmark. Since the miss rate varies greatly, a logarithmic scale is used on the

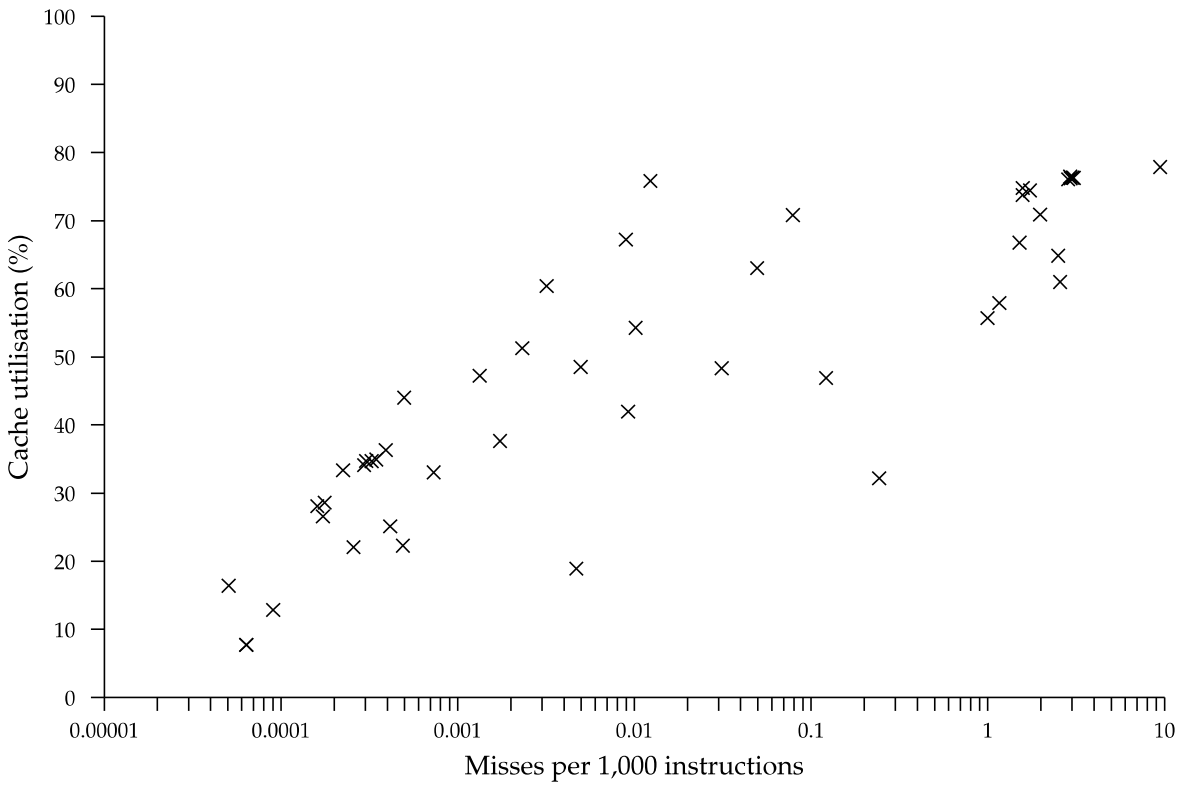

Figure 3.20: Cache utilisation *vs* miss rate in the IL1 cache

x-axis. In this case, there is a clear trend that a high cache miss rate tends to have a correspondingly high cache utilisation. Benchmarks with a low miss rate have a utilisation under approximately 20%, while those with a high miss rate have a utilisation over approximately 70%. Benchmarks with moderate miss rates have a more variable utilisation, from approximately 30% to 70%. There are no benchmarks with a low miss rate and a high utilisation, nor a high miss rate and a low utilisation.

L2 Cache

Figure 3.21 plots the cache utilisation against the cache miss rate of the L2 cache for each benchmark. Again, since the miss rate varies greatly, a logarithmic scale is used on the x-axis. The results for this cache are more variable than those for either of the L1 caches. There are no benchmarks with a low miss rate and a high utilisation, likewise nor are there any benchmarks with a high miss rate and a high utilisation. For moderate miss rates, cache utilisation varies from approximately 5% to 80%, while for particularly low or high miss rates, it is typically below 20%.

3.5.5 Impact of Scaling Cache Size on Utilisation

The above results for the DL1, IL1 and L2 caches indicate that there is no simple relationship between cache utilisation and cache miss rate, therefore the impact of con-

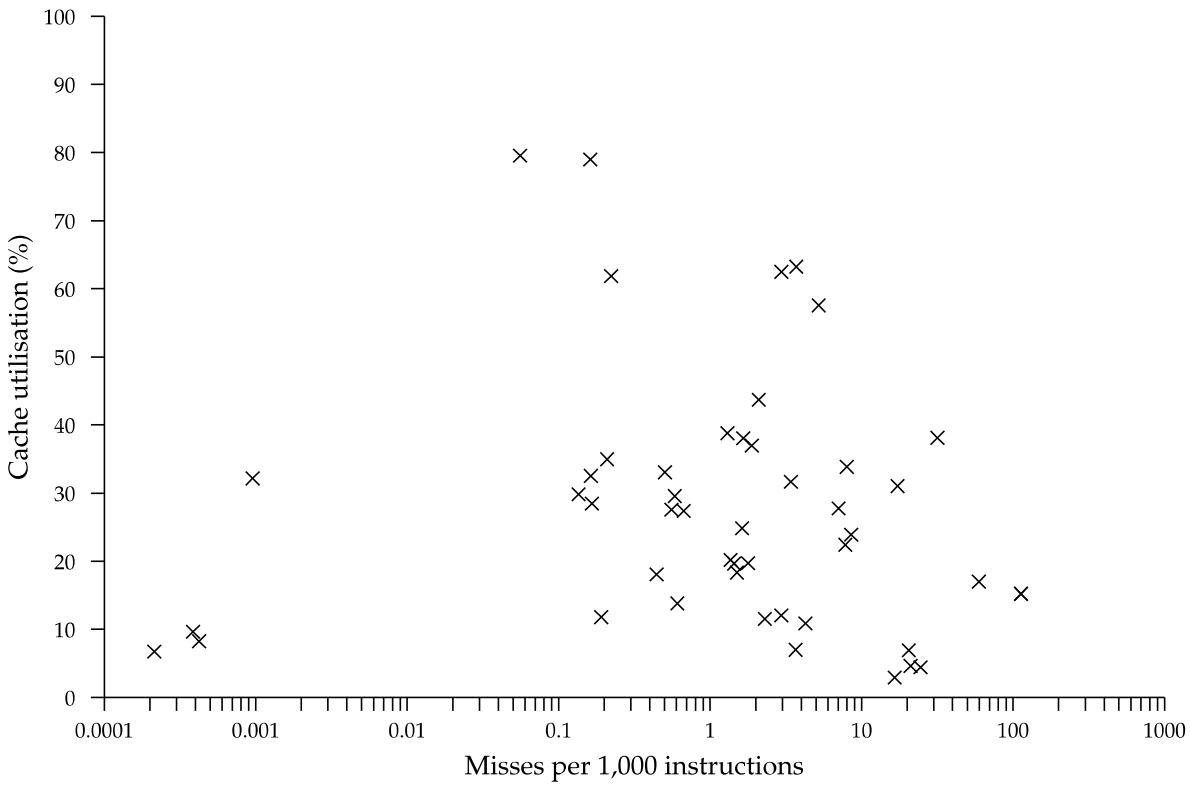

Figure 3.21: Cache utilisation *vs* miss rate in the L2 cache

ventional techniques to reduce the cache miss rate on cache utilisation is difficult to anticipate. For this reason, cache utilisation is now examined against scaling cache size and associativity. Unlike Section 3.3.1, cache size and associativity are scaled independently in an attempt to separate their behaviour.

DL1 Cache

Figure 3.22 plots cache utilisation against cache size (from 8 kB to 1024 kB) for the DL1 cache for each benchmark. For ease of analysis, the benchmarks are clustered into four separate groups, each of which shows similar behaviour. Group A (top-left, 17 benchmarks) shows a single peak of cache utilisation for each benchmark, typically from 32 kB to 256 kB. Either side of this peak cache utilisation falls markedly. Group B (top-right, 14 benchmarks) shows a single trough of cache utilisation for each benchmark, typically from 64 kB to 256 kB. Either side of this trough cache utilisation rises markedly. Group C (bottom-left, 9 benchmarks) shows continually decreasing cache utilisation, for some benchmarks this decrease is sharp while for others it is very shallow. Finally, Group D (bottom-right, 8 benchmarks) shows more varied behaviour for each benchmark, with differing magnitudes and direction of change.

It might be expected that the most common behaviour, that of a single peak in Group A, would be more common. Scaling cache size without changing utilisation will result in decreasing the number of capacity misses, up to a point at which the number of conflict

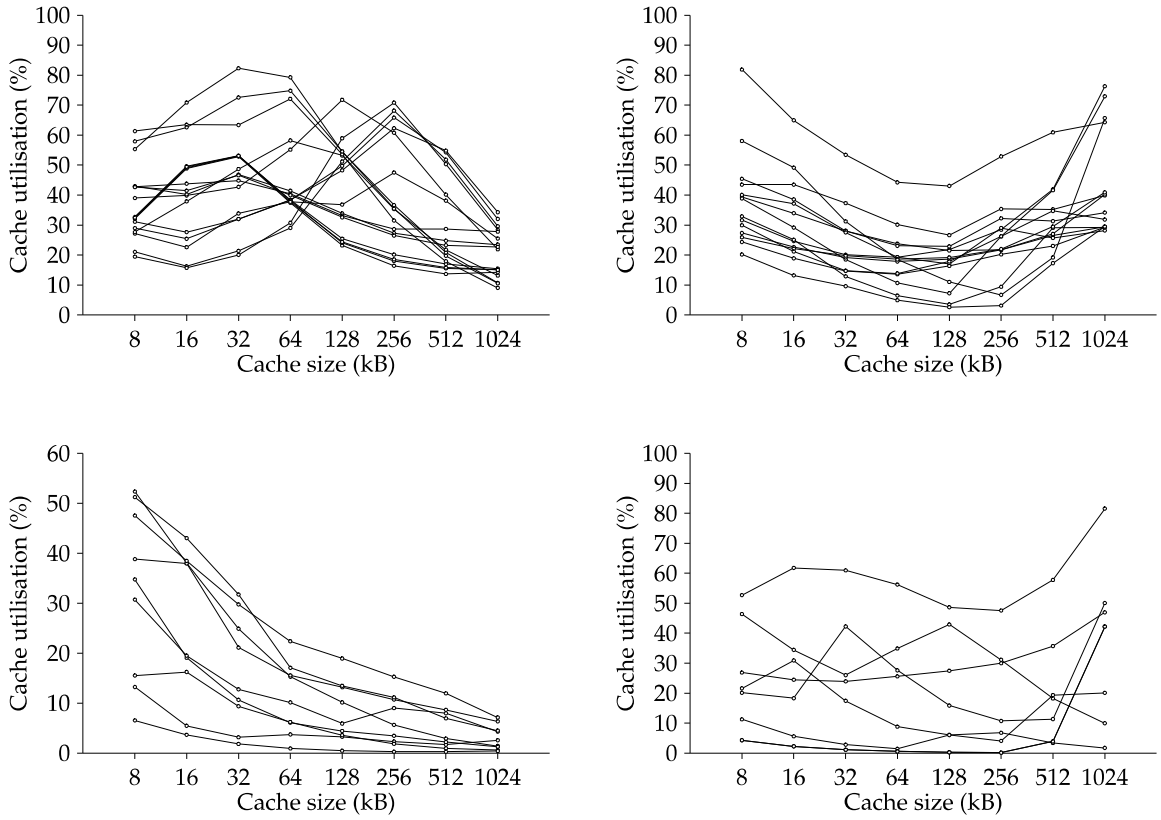

Figure 3.22: Cache utilisation *vs* cache size for the DL1 cache. Similar trends are clustered into four different groups (A, B, C, D) from top-left to bottom-right.

misses becomes prevalent. Further increasing the cache size results in falling utilisation since there are fewer additional hits being made in the additional cache lines. To some extent, this may account for the continually decreasing behaviour of Group C, if it is assumed that a peak in cache utilisation lies at cache sizes smaller than 8 kB. However, the behaviour of Groups B and D, comprising almost half of the benchmarks, is more difficult to explain in such a simplistic manner.

IL1 Cache

Figure 3.23 plots cache utilisation against cache size (from 8 kB to 1024 kB) for the IL1 cache for each benchmark. For ease of analysis, the benchmarks are clustered into three separate groups, each of which shows similar behaviour. Group A (top-left, 29 benchmarks) shows a single peak of cache utilisation for each benchmark, typically from 32 kB to 128 kB. Either side of this peak, cache utilisation falls steeply in most cases, but more gently in others. Group B (top-right, 15 benchmarks) shows continually decreasing cache utilisation, which is steep in almost every case. Finally, Group C (bottom, 4 benchmarks) shows more varied behaviour for each benchmark, with differing magnitudes and directions of change. Groups A and B for the IL1 cache correspond to the same behaviours exhibited by Groups A and C of the DL1 cache respectively, and a similar explanation may be inferred.

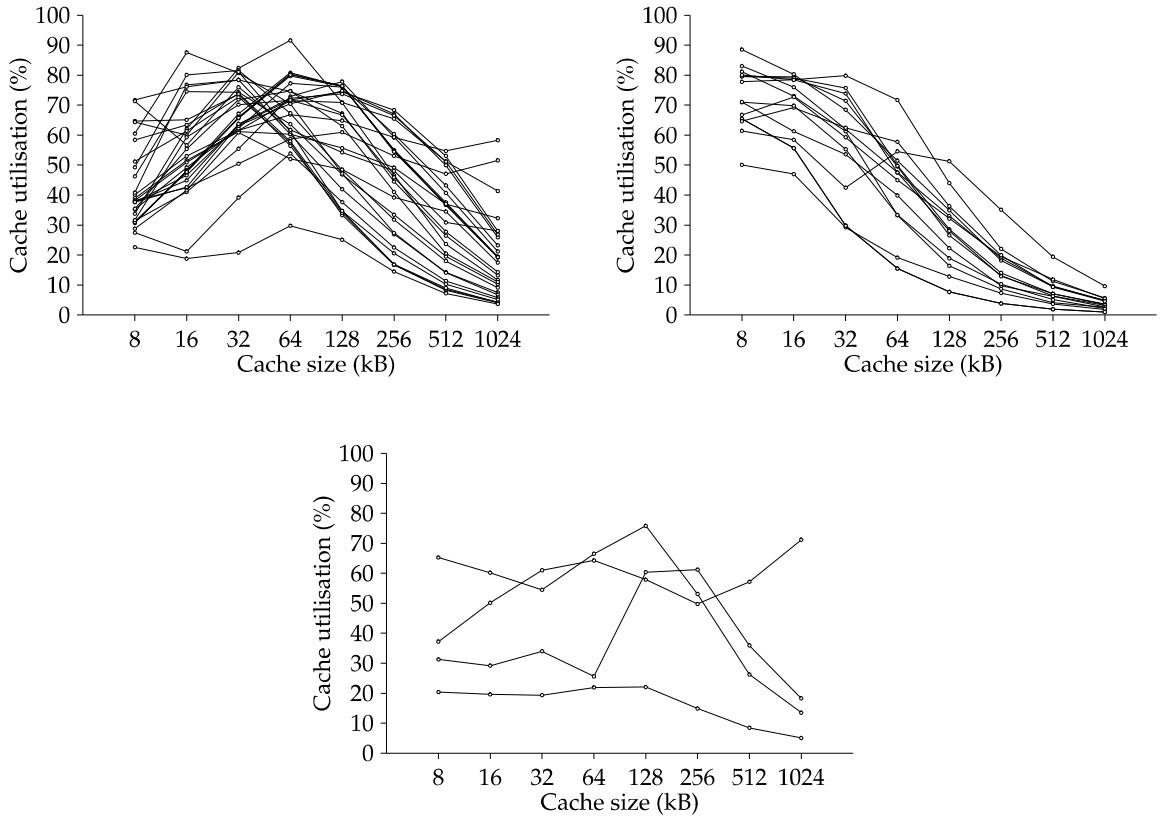

Figure 3.23: Cache utilisation *vs* cache size for the IL1 cache. Similar trends are clustered into three different groups (A, B, C) from top-left to bottom.

L2 Cache

Figure 3.24 plots cache utilisation against cache size (from 128 kB to 32768 kB) for the L2 cache for each benchmark. For ease of analysis, the benchmarks are clustered into five separate groups, each of which shows similar behaviour. Group A (top-left, 18 benchmarks) shows a single peak of cache utilisation for each benchmark, typically from 1024 kB to 8192 kB, although with a fair degree of variation. Either side of this peak, cache utilisation falls markedly. Group B (top-right, 3 benchmarks) shows continually decreasing cache utilisation at a relatively slow rate. Group C (middle-left, 6 benchmarks) shows continually increasing cache utilisation, mostly at an increasing rate but in one case at a decreasing rate. Group D (middle-right, 4 benchmarks) shows a single trough of cache utilisation. Either side of this trough, cache utilisation increases at varying rates. Finally, Group E (bottom, 17 benchmarks) shows very varied behaviour for each benchmark.

The above results show that there is no single clear trend between cache utilisation and cache size. Depending on the benchmark, cache utilisation may rise or fall as the cache size is changed. For some benchmarks, long-term trends (e.g. peaks, troughs, continual increase or continual decrease) are apparent, but for some benchmarks, particularly in the L2 cache, no long-term trend is apparent.

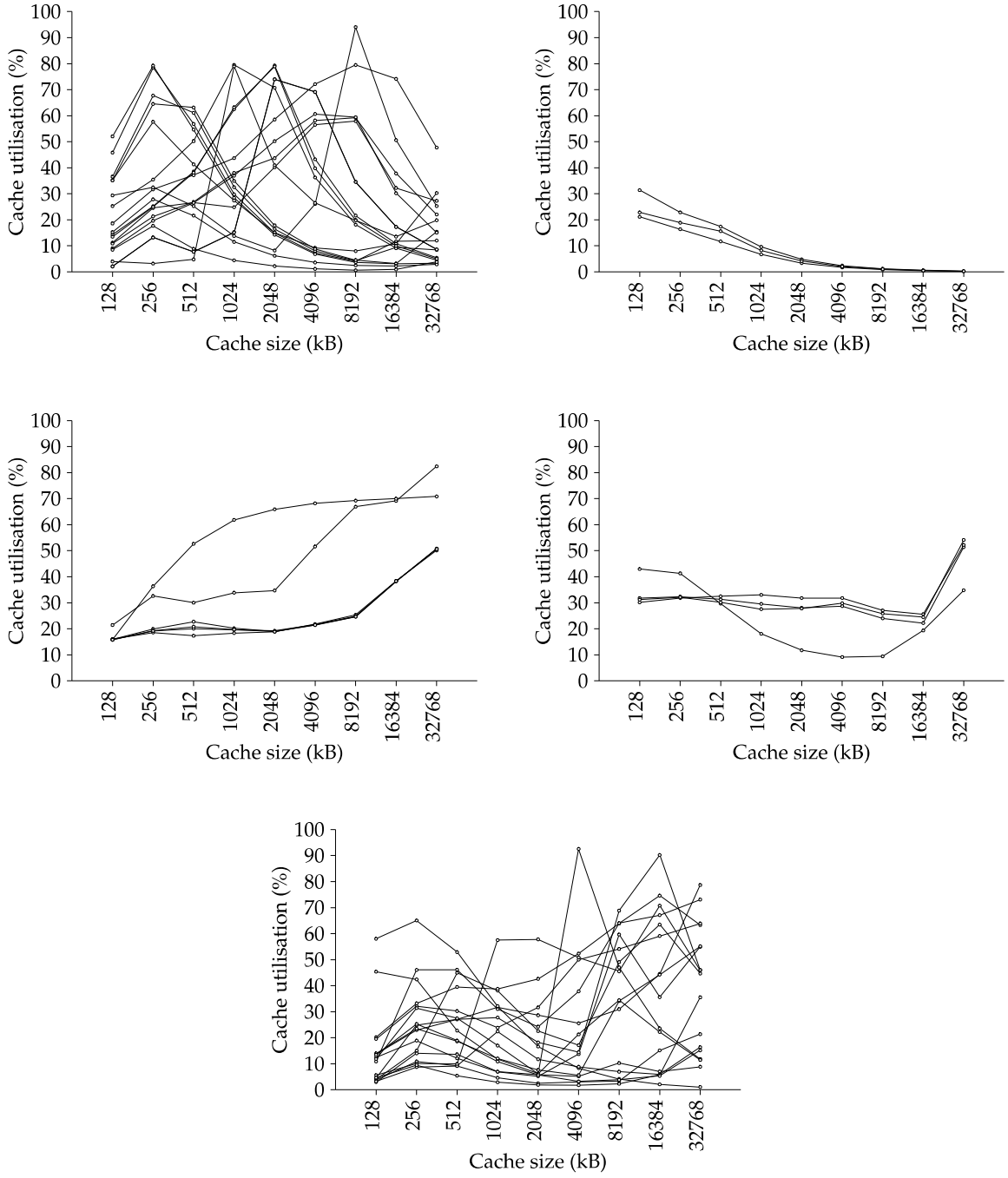

Figure 3.24: Cache utilisation *vs* cache size for the L2 cache. Similar trends are clustered into five different groups (A, B, C, D, E) from top-left to bottom.

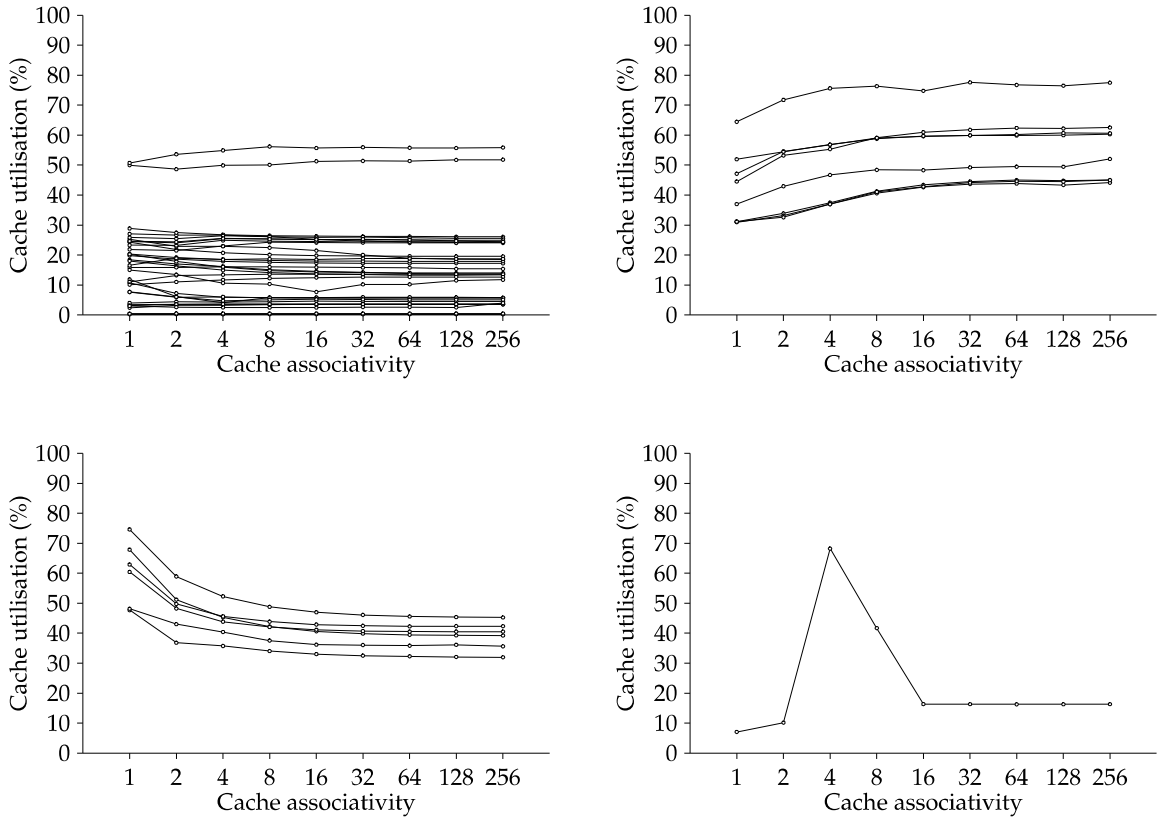

Figure 3.25: Cache utilisation *vs* cache associativity for the DL1 cache. Similar trends are clustered into four different groups (A, B, C, D) from top-left to bottom-right.

3.5.6 Impact of Scaling Cache Associativity on Utilisation

Figures 3.25, 3.26 and 3.27 plot cache utilisation against cache associativity for the DL1, IL1 and L2 caches respectively.

DL1 Cache

Figure 3.25 plots cache utilisation against cache associativity (from 1 to 256 way) for the DL1 cache for each benchmark. For ease of analysis, the benchmarks are clustered into four separate groups, each of which shows similar behaviour. Group A (top-left, 33 benchmarks) shows very little variation over the whole range of cache associativities examined. Group B (top-right, 8 benchmarks) shows continually mildly increasing cache utilisation. Group C (bottom-left, 6 benchmarks) shows continually decreasing cache utilisation. Finally, Group D (bottom-right, 1 benchmark) shows a single peak of cache utilisation for a 4-way cache with cache utilisation falling markedly either side.

The predominant behaviour is for cache utilisation to remain constant over the range of cache associativities examined. This is slightly unexpected, since increasing cache associativity without increasing cache size would be expected to decrease the number of conflict misses, resulting in an overall increase in cache utilisation since conflict misses have been shown to have shorter dead times than capacity misses [HKM02].

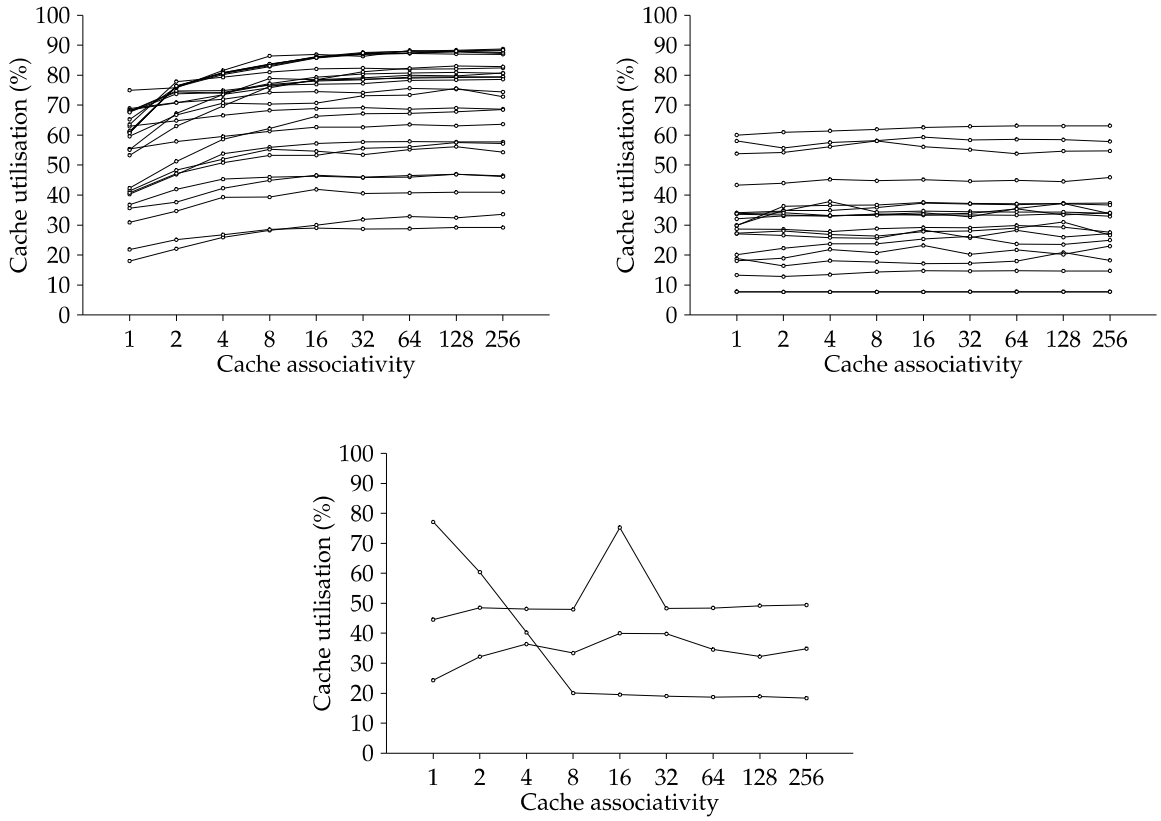

Figure 3.26: Cache utilisation *vs* cache associativity for the IL1 cache. Similar trends are clustered into three different groups (A, B, C) from top-left to bottom.

IL1 Cache

Figure 3.26 plots cache utilisation against cache associativity (from 1 to 256 way) for the IL1 cache for each benchmark. For ease of analysis, the benchmarks are clustered into three separate groups, each of which shows similar behaviour. Group A (top-left, 26 benchmarks) shows cache utilisation continually increasing, mostly at a slow rate, over the entire range of cache associativities examined. Group B (top-right, 19 benchmarks) shows little variation of cache utilisation over the entire range of cache associativities examined. Finally, Group C (bottom, 3 benchmarks) shows more varied behaviour.

L2 Cache

Figure 3.27 plots cache utilisation against cache associativity (from 1 to 1024 way) for the L2 cache for each benchmark. For ease of analysis, the benchmarks are clustered into four separate groups, each of which shows similar behaviour. Group A (top-left, 35 benchmarks) shows little variation of cache utilisation over the entire range of cache associativities examined. Group B (top-right, 9 benchmarks) shows continually decreasing cache utilisation, almost always at a very slow rate. Group C (bottom-left, 3 benchmarks) shows continually increasing cache utilisation, again almost always at a very slow rate. Finally, Group D (bottom-right, 1 benchmark) shows more varied behaviour.

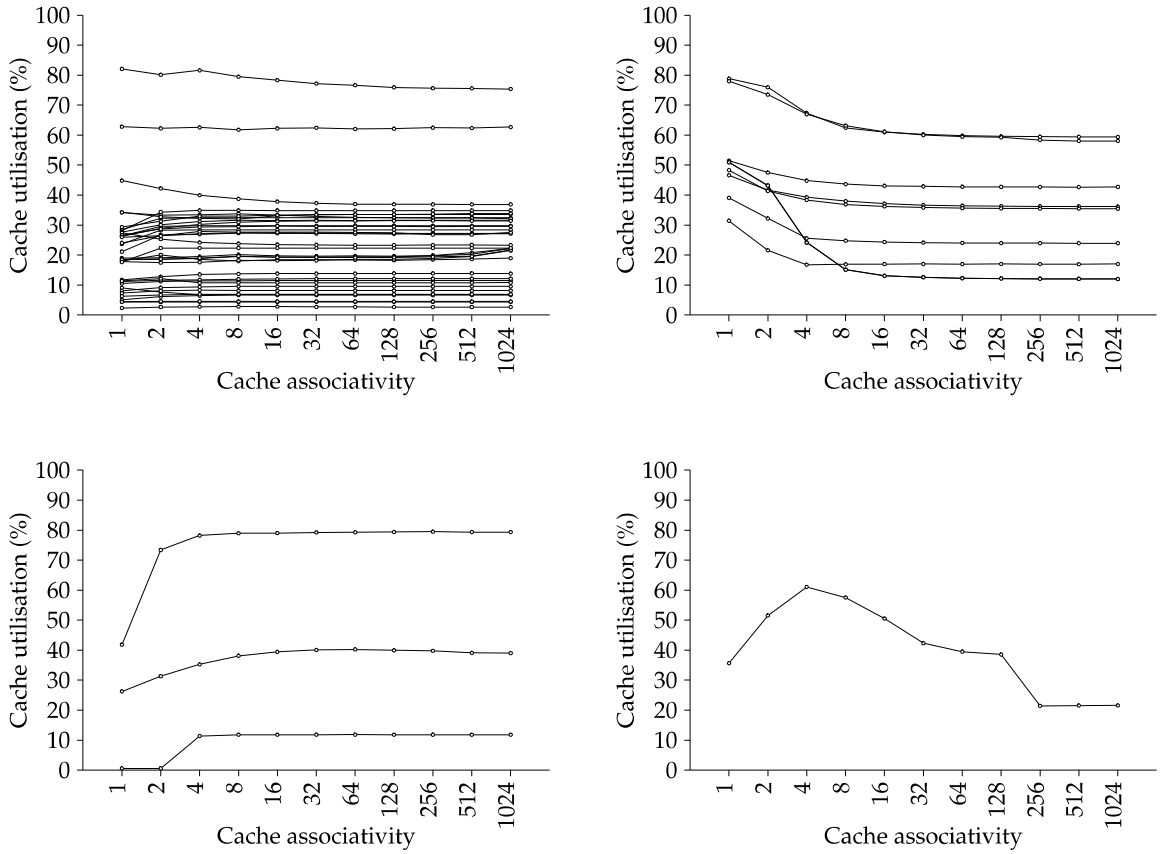

Figure 3.27: Cache utilisation *vs* cache associativity for the L2 cache. Similar trends are clustered into four different groups (A, B, C, D) from top-left to bottom-right.

Scaling of cache associativity tends to produce a more consistent impact on cache utilisation than scaling of cache size, with the most common outcome being little overall change in cache utilisation.

3.6 Summary

This chapter started by detailing the simulation method used to generate the results presented in this dissertation. Cache performance was shown to have a significant impact on overall system performance by comparing the performance of the baseline processor configuration to that which would be achieved with varying levels of perfect caches. Having discussed the miss rates of the individual caches and benchmarks, the results of a projected scaling of cache size and associativity were shown to still lag the performance of a perfect memory hierarchy considerably, indicating that naïve scaling alone cannot continually improve performance.

Having defined metrics to describe the lifetime of cache lines, the distributions of these metrics were discussed and much variation found, something which has not been considered by previous work. Using the live and dead times previously defined, a simple metric for cache utilisation was formulated. Cache utilisation was shown to vary considerably between benchmarks and caches, with the instruction cache (average 48%) showing much better utilisation than the data caches (average 26%). This is to be expected, since access patterns to the instruction cache are typically much more regular than those to the data caches.

The relationship between cache miss rates and utilisation was also investigated, with no simple relationship being applicable for all benchmarks and caches. This indicates that traditional techniques to decrease cache miss rates do not necessarily increase utilisation. The impact of scaling cache size on cache utilisation was examined, as was the impact of scaling cache associativity. The former was found to result in a variety of long-term trends depending on the particular benchmark and cache combination, while the latter was found to generally have little impact.

Subsequent chapters will describe methods to predict cache line lifetime metrics, as well as applications of those predictors. The general intention is to improve cache utilisation, and the impact on cache miss rates will also be detailed.

4 Prediction

4.1 Overview

The aim of this chapter is to devise methods to accurately predict two useful cache line lifetime-related properties using reasonable additional hardware constructs. These predictors will be used in two performance-enhancing applications detailed in Chapter 5.

The chapter begins by examining the inherent predictability of cache line lifetime metrics by looking at how future cache line behaviour relates to that observed in the past. Two predictability metrics are defined and evaluated over the set of benchmarks. Having determined the innate predictability of cache line lifetime metrics, two specific predictor requirements are characterised in order to support the applications which will be detailed in Chapter 5. The first predictor is a binary predictor, a family of predictors which have been examined in great depth by previous research into branch prediction. The second predictor is a value predictor, again a family of predictors which have been examined by previous research. Potential implementations of the two predictors are discussed, both those based upon previous work as well as several novel predictor implementations. Actual evaluation of specific predictor implementations, which is dependent on the application, is left until Chapter 5.

4.2 Introduction

Prediction, or **speculation**, is a technique widely employed by computer architects in applications to both improve performance and decrease power consumption. For example, **branch prediction** is widely used to avoid pipeline stalls in modern microprocessors. This technique predicts whether a branch instruction will be taken (or not) and fetches the subsequent instruction sequence accordingly. When the actual outcome of the branch is known, if the prediction was found to be incorrect, the state changes made by the incorrect instructions being executed must be rolled back and the correct instruction sequence fetched and executed. Due to the highly repetitive behaviour of typical programs, branch prediction usually achieves very high accuracy and so mispredictions are fortunately rare. Another form of prediction is **load value prediction**, introduced by Lipasti *et al.* [LWS96], which seeks to reduce the average memory access latency by predicting the contents of a memory location before the actual contents can be supplied by the memory hierarchy.

Prefetching is another speculation technique that predicts which memory addresses will be accessed in the near future and proactively fetches those addresses into higher levels of the memory hierarchy, before the actual memory instruction accessing those

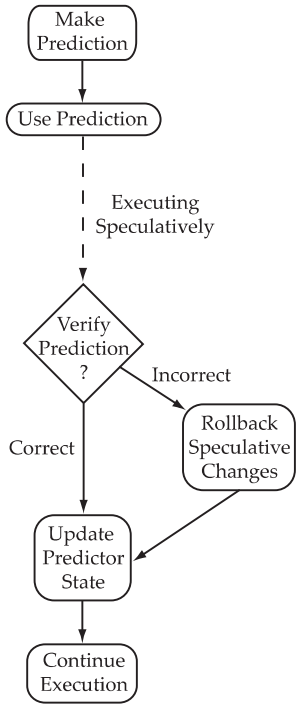

Figure 4.1: Overview of general speculation

addresses is encountered. Prefetching can be considered a form of speculative execution, albeit without requiring any support for rolling back processor state, since prefetches are always safe as errors such as page faults are ignored. However, a poor prefetching implementation can cause a negative performance and/or power impact by competing with regular memory accesses for both cache space and bus bandwidth. Aggressive prefetching is widely deployed in both hardware and software in modern computer systems and has already been detailed in Section 2.6.2.

Figure 4.1 shows a generic overview of speculation in computer architecture. Once a prediction is made and used, the processor may not be executing the correct instructions with the correct operands. At some later point when the validity of the prediction can be established, these speculatively executed instructions may be either **committed** if the prediction was correct or **squashed** should the prediction turn out to have been incorrect. In either case, the predictor state is updated and execution can continue until the next prediction is required. In a dynamically scheduled superscalar processor, rollback of incorrectly speculated instructions can be expensive, involving discarding all instructions dependent on that incorrect prediction. Thus, for example, accurate branch prediction is critical to achieving high performance in such processors.

4.3 Predictability of Cache Line Behaviour

Most methods of predicting events in computer architecture rely upon making observations of past events and using these observations to infer what future events will

occur. For example, at a high level, many branch predictors look at how often a particular branch was taken in the recent past and use this information to decide whether that particular branch will be taken next time it is encountered. Such an approach relies upon past events being indicative of future events. In this section, cache line lifetime behaviour is examined to determine if this is the case. Hu *et al.* [HKM02] examined one of the simplest metrics for predictability — the difference between consecutive values for each unique cache line address[1]. This same metric is now used to examine the predictability of live times, dead times and access intervals for all three caches in the baseline configuration.

For each cache, a categorised distribution indicating the magnitude of difference values falling into various ranges is plotted. The central category represents a difference of zero i.e. the previous value was the same as the current. Moving outwards in both the positive and negative directions (right and left respectively), the differences are clustered such that, for example, the data point numbered 60 corresponds to the percentage of differences between 51 (one more than the previous data point) and 60. Such distributions may conventionally be plotted as histograms, however, such a representation is unwieldy when dealing with many benchmarks concurrently. The categories are defined such that the x-axis uses a partially logarithmic scale.

Given regular reference patterns, one might expect cache line lifetime metrics to be very consistent, so that the difference observed is small, illustrated by a peak in the centre of the difference distribution plots which follow. Less regular reference patterns will lead to a wider range of differences being observed, spreading the distribution over a wider range.

4.3.1 Live Time Predictability

Figures 4.2, 4.3 and 4.4 show the categorised distributions of live time differences for each benchmark for the DL1, IL1 and L2 caches respectively. Many benchmarks show evidence of clustering of live time differences around zero, while some show significant clusters at other points. This behaviour is now discussed for each cache in turn.

DL1 Cache

Figure 4.2 shows the categorised distributions of live time differences for the DL1 cache for each benchmark. There were enough cache replacements made by all benchmarks to provide previous live times, so all benchmarks are represented. Numerous benchmarks exhibit a pronounced central peak in the 0 (i.e. identical) difference category, led by `ammp` and `perlbmk_2` with 99.4% and 96.8% respectively. Some benchmarks have a much smaller central value, including `gap` and `applu` with 0.47% and 1.45% respectively. The mean proportion of identical successive live times across all benchmarks is 36.9%.

Moving outwards, there are few other significant peaks until the extremes of the distributions are reached. A few benchmarks show considerable proportions of live time differences falling outside the range under study. These include `eon_3` and `gzip_program`

[1] A unique cache line address is simply the memory address aligned to the size of a cache line

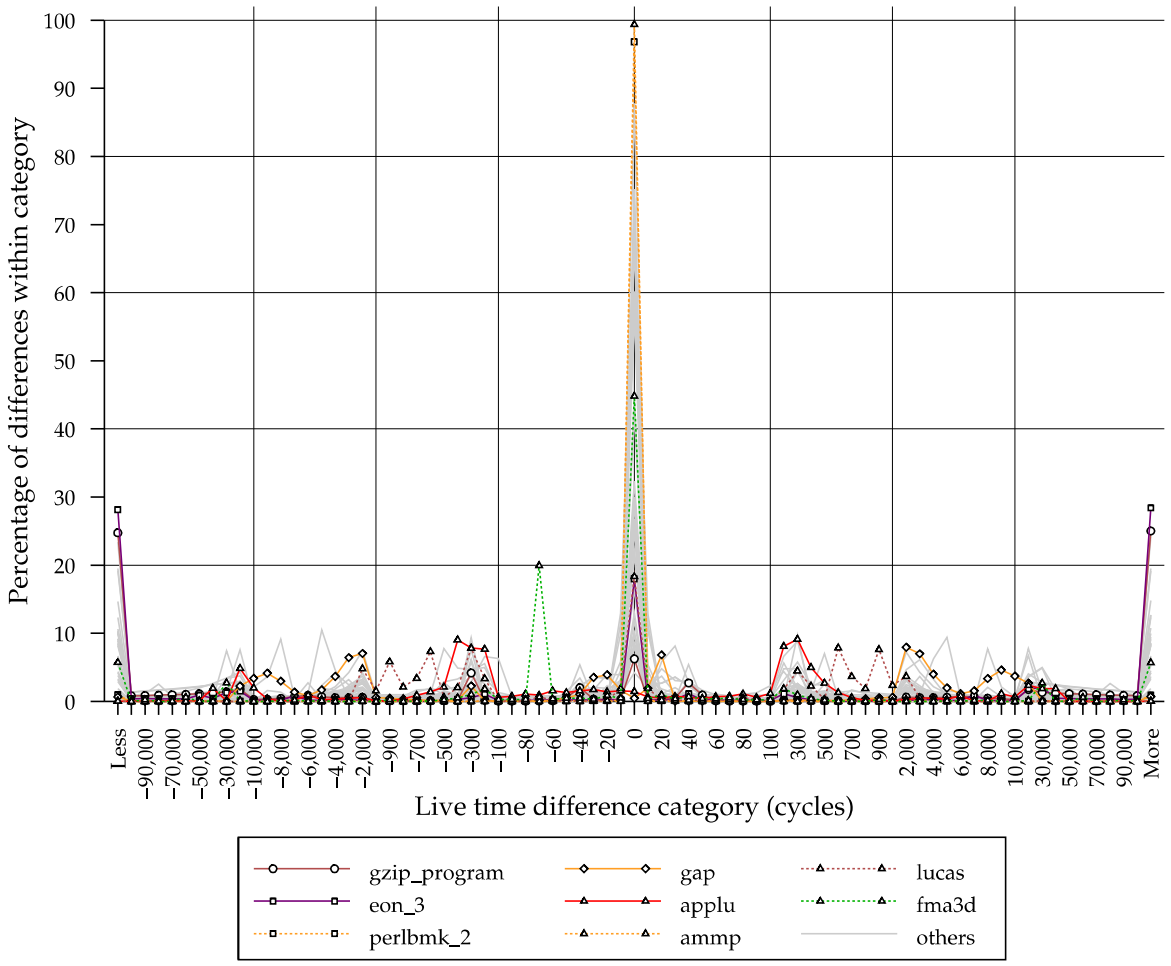

Figure 4.2: Categorised distributions of live time differences for the DL1 cache

with a significant number of differences being larger than 100,000 cycles (28.7% and 26.4% respectively). The mean proportion of differences in this category is 6.51%.

At the other extreme, differences less than -100,000 cycles, `eon_3` and `gzip_program` also lead, with 28.4% and 25.0% respectively. The mean proportion of differences in this category is 6.38%, similar to the other extreme.

Several benchmarks exhibit interesting symmetrical distributions, such as `applu` and `gap`. This may indicate the presence of alternating live time values. For example, the sequence 10,5,10,5,10,5 would give rise to differences of -5,5,-5,5,-5.

IL1 Cache

Figure 4.3 shows the categorised distributions of live time differences for the IL1 cache for each benchmark. At first glance, the distributions seem rather similar to those observed for the DL1 cache with large central peaks for many benchmarks and relatively large values at the extremes for a few others.

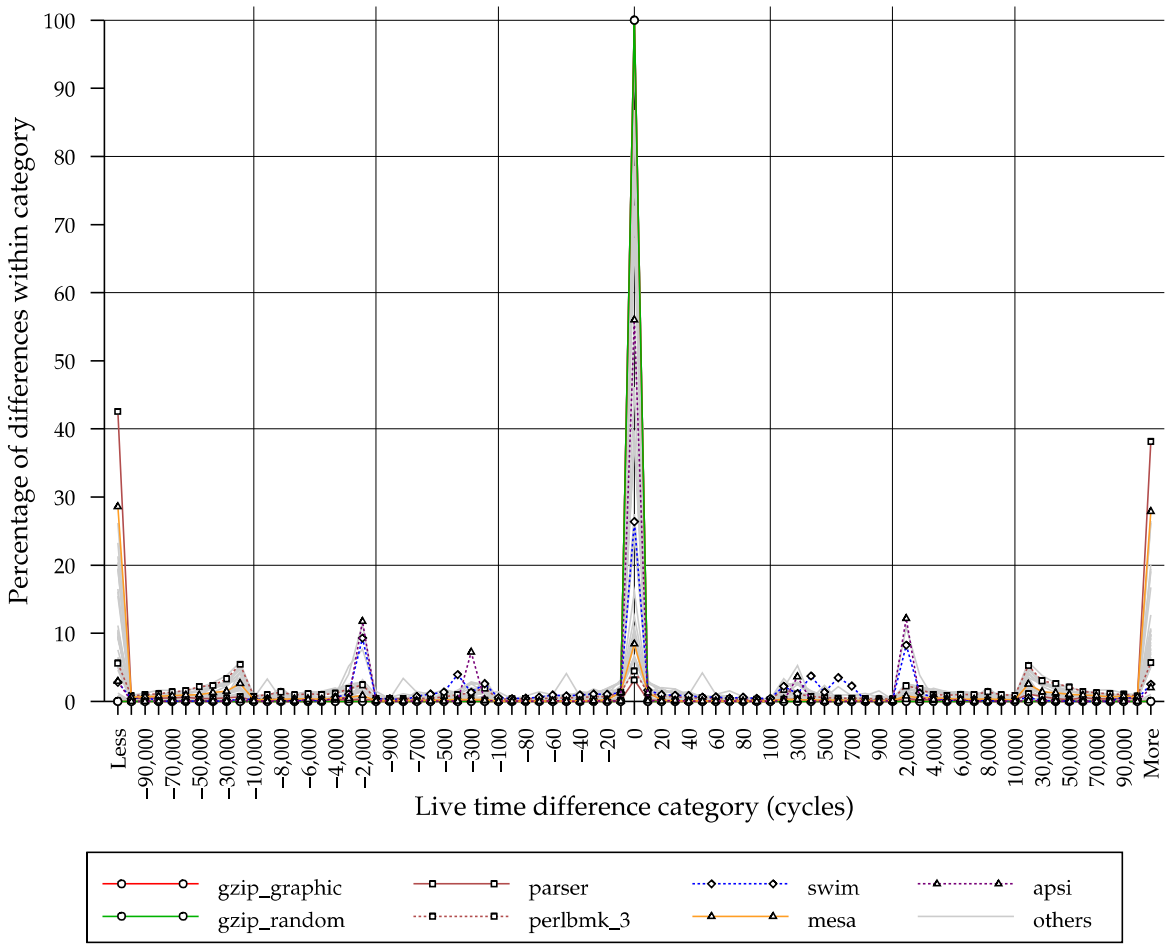

Figure 4.3: Distributions of live time differences for the IL1 cache

Several benchmarks demonstrated sufficiently repetitive behaviour that *all* of their live times were identical to their previous values. These benchmarks include `gzip_graphic` and `gzip_random`, as well as `mcf`, `art_1` and `art_2`. However, recalling Figure 3.3, these benchmarks have very low IL1 cache miss rates, so make very few IL1 cache replacements, thus these live time differences cover very few actual pairs of live times. Equally, other benchmarks with similarly low IL1 cache miss rates exhibited very small proportions of live times in this central category. One example of this is `parser`, with only 3.14% of identical live time differences, while the next smallest benchmark in this category, `perlbmk_3` with 4.46%, actually has the highest IL1 cache miss rate. The mean proportion of identical successive live times across all benchmarks is 47.5%.

Moving away from the central peaks, the distributions are rather similar to the DL1 cache, although the proportion of differences between the extremes and the central maxima are somewhat reduced. This is to be expected since recalling Figures 3.7 and 3.8, on average, IL1 cache live times are longer than DL1 cache live times, thus their consecutive differences will be, on average, larger.

Both `swim` and `apsi` exhibit small symmetrical peaks around +2,000 and -2,000 cycles difference, again perhaps due to alternating behaviour.

A few benchmarks have a significant proportion of differences in each of the extreme categories. At the negative extreme, less than -100,000 cycles, `parser` and `mesa` lead with 42.5% and 28.6% of differences respectively. The mean proportion of differences in this category across all benchmarks is 8.7%.

The behaviour at the positive extreme of the distribution is largely similar, with the maximum proportion of differences again belonging to `parser` and `mesa`, with 38.2% and 27.9% of live time differences being larger than 100,000 cycles respectively. The mean proportion of live time differences in this category is 8.11%, very similar to the negative extreme.

L2 Cache

Figure 4.4 shows the categorised distributions of live time differences for the L2 cache for each benchmark. The overall appearance is similar to that of the DL1 and IL1 caches, with large central peaks for many benchmarks and relatively little significant proportions encountered until the extremes are reached.

Again, several benchmarks demonstrated sufficiently repetitive behaviour that *all* of their live times were identical to their previous values. These benchmarks were `eon_1`, `eon_2` and `eon_3` and recalling Figure 3.2, these benchmarks had a very low L2 cache miss rate. Other benchmarks with a large proportion of consecutive live times being identical are `ammp`, `gzip_random` and `gzip_graphic`, with 99.6%, 99.2% and 98.8% respectively. A few benchmarks have a very small number of differences in this category, including `perlbmk_2`, `gap` and `mesa` with 0.068%, 0.23% and 7.74% respectively. The mean proportion of identical successive live times across all benchmarks is 49.4%.

There are very few significant peaks encountered between the central category and the extremes, other than a pair of symmetrical peaks for `swim` at -50,000 and 50,000 cycles. Compared to the DL1 and IL1 cache results, significantly more differences fall outside the range under study, which is to be expected since recalling Figures 3.7, 3.8 and 3.9, L2 cache line live times are considerably longer than DL1 cache line live times, and somewhat longer than IL1 cache line live times, therefore the magnitude of their successive differences is likely to be larger.

At the negative extreme, less than -100,000 cycles, `perlbmk_2` and `fma3d` have the largest proportion of differences, with 64.7% and 61.5% respectively. The benchmarks previously reported as having significant central peaks clearly have very small proportions in this category. The mean proportion of live time differences in this category is 23.4%, considerably larger than that in the corresponding categories of either the DL1 or IL1 distributions.

At the positive extreme, more than 100,000 cycles, similarly large proportions of differences are encountered, led by `gap`, `crafty` and `parser` with 50.9%, 44.6% and 44.0%. The mean proportion in this category is 16.6%, also considerably higher than that in the corresponding categories of either the DL1 or IL1 distributions.

4.3.2 Dead Time Predictability

Figures 4.5, 4.6 and 4.7 show the categorised distributions of differences between consecutive dead times for each benchmark for the DL1, IL1 and L2 caches respectively.

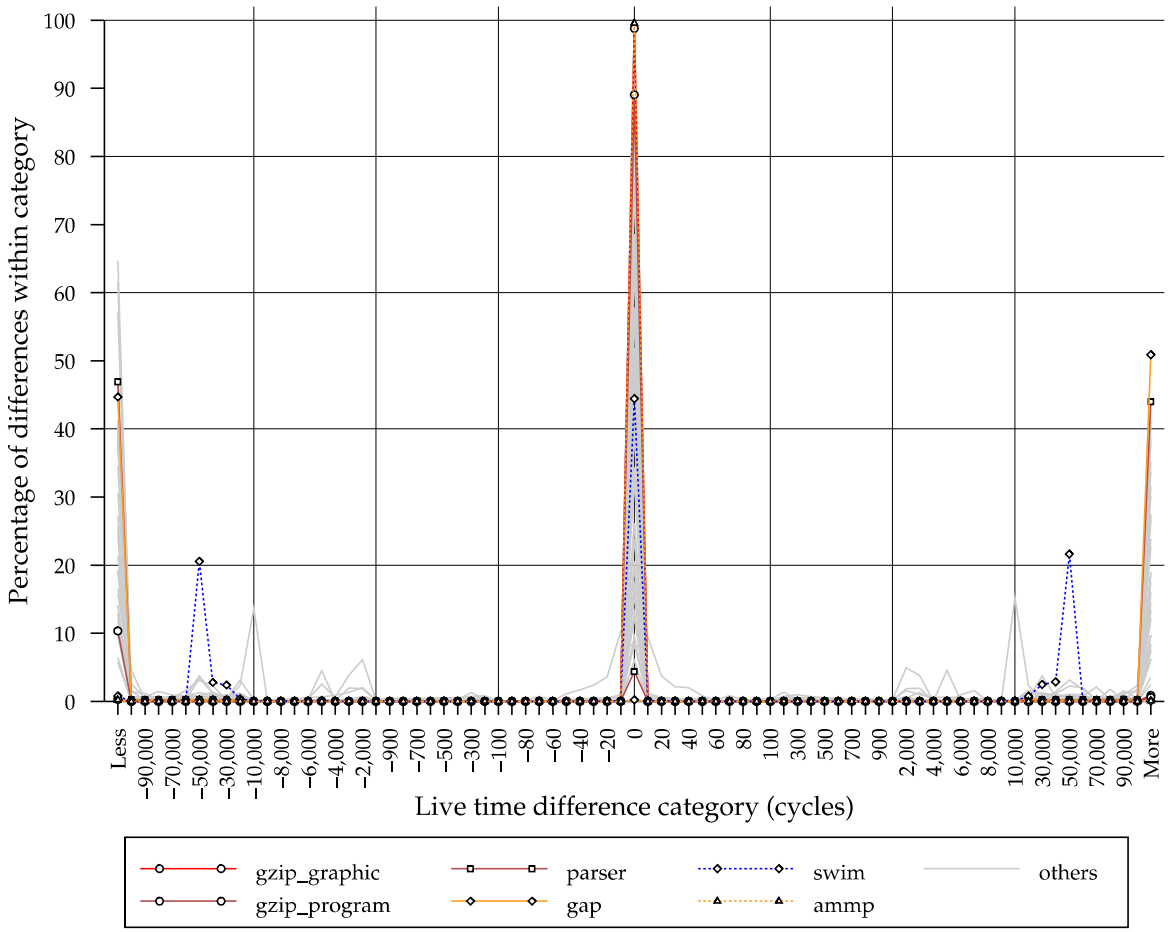

Figure 4.4: Distributions of live time differences for the L2 cache

Like the previous live time differences, dead time differences show evidence of clustering around small values, but overall dead times differences seem slightly larger than live time differences. This is to be somewhat expected since the magnitude of dead times is considerably larger than the magnitude of live times as discussed in Section 3.4.2. Each of the dead time difference distributions is now discussed in turn.

DL1 Cache

Figure 4.5 shows the categorised distributions of dead time differences for the DL1 cache for each benchmark. Like the live time differences shown in Figure 4.2, significant numbers of cache lines have small dead time differences, illustrated by the central peaks in the distribution. The largest peak occurs in the central category of zero difference for `ammp` with 82.8% and `gcc_166` with 51.5%. Other benchmarks have much smaller values in this category, the smallest being `gap` with only 0.00025%. Indeed the average proportion of differences in this category is 8.13%, considerably smaller than the 36.9% in the same category for live time differences.

Moving outwards, there are few significant clusters encountered other than for `swim`,

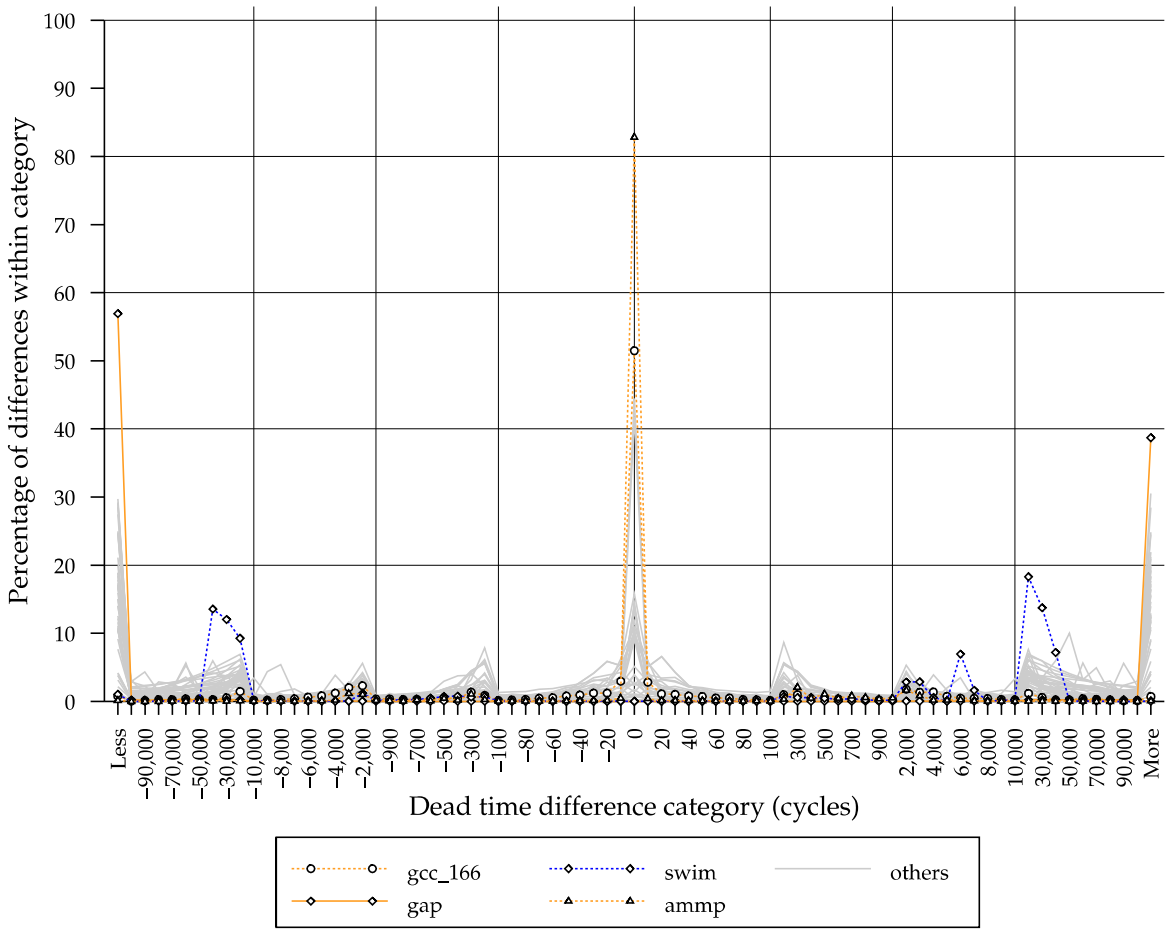

Figure 4.5: Distributions of dead time differences for the DL1 cache

which exhibits asymmetrical peaks around -30,000 and 30,000 cycles. The proportion of differences falling outside the positive extreme, more than 100,000 cycles, varies from 38.7% for `gap` to 0.025% for `swim`, with an average of 13.5%.

At the other extreme, less than -100,000 cycles, the proportion of differences varies from 56.9% for `gap` again, to 0.19% for `ammp`, with an average of 14.4%. In general, as would be expected, those benchmarks with a large proportion of small differences have a small proportion of large differences and *vice versa*.

IL1 Cache

Figure 4.6 shows the categorised distributions of dead time differences for the IL1 cache for each benchmark. The same set of benchmarks previously found to have *all* their live times to be identical also find all their dead times to be identical. Other benchmarks with high proportions of identical dead times include `gzip_source` and `wupwise` with 99.9% and 99.8% respectively. Behaviour in this category varies significantly, with `applu` and `perlbmk_3` having 0% and 0.014% of identical dead times. The average proportion of identical dead times is 41.1%, however, as has already been indicated,

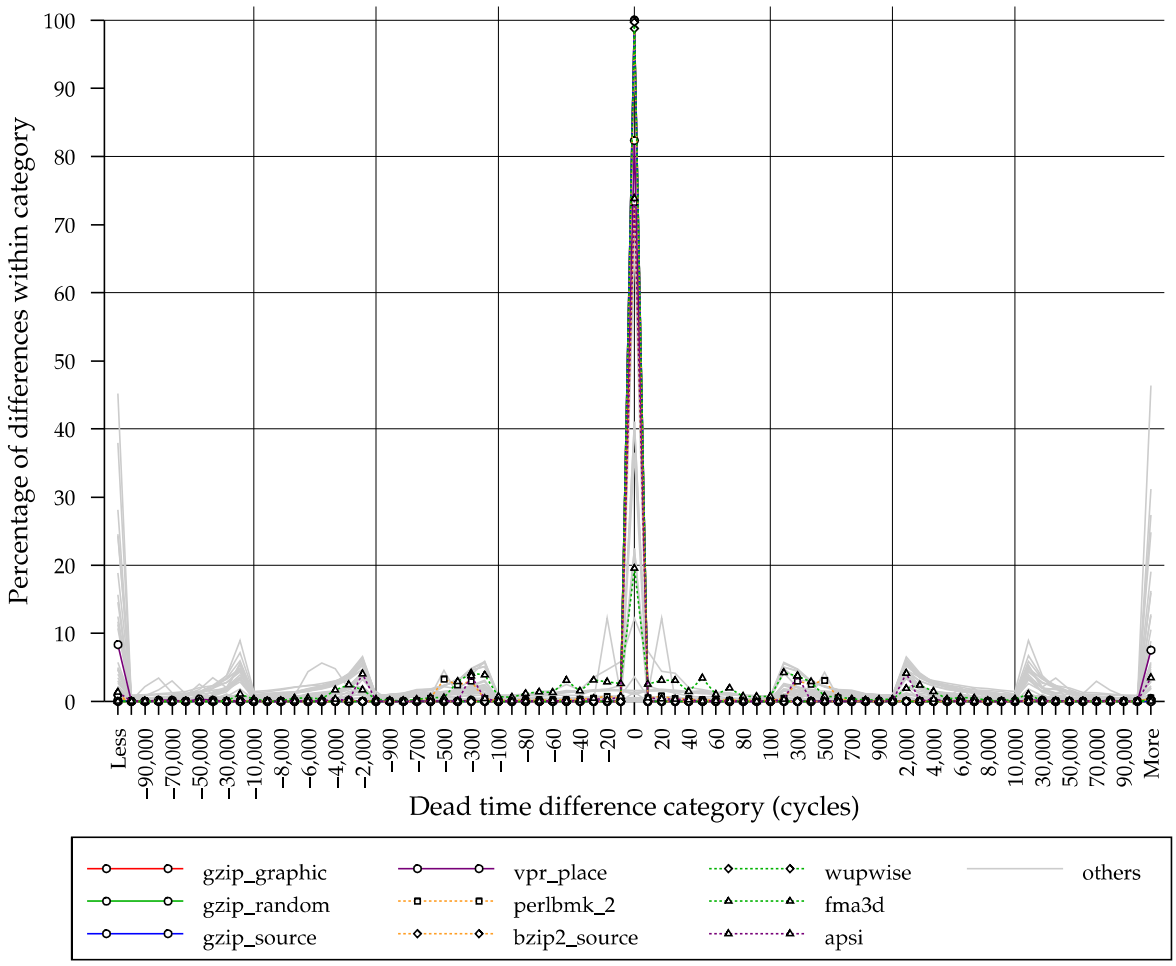

Figure 4.6: Distributions of dead time differences for the IL1 cache

this proportion is highly variable between benchmarks.

Moving outwards, several benchmarks show small symmetrical clusters around -20 and 20 cycles, -200 and 200 cycles, -2,000 and 2,000 cycles and -20,000 and 20,000 cycles. At the positive extreme, more than 100,000 cycles, the proportion of differences varies from 46.3% for `parser` to 0% for those benchmarks with all identical dead times. The average proportion of differences in this category is 6.4%. At the negative extreme, less than 100,000 cycles, the proportion of differences varies from 45.2% for `parser` again, to 0% for the same set of benchmarks previously listed. The average proportion of differences in this category is 6.9%. The proportions of differences at the extremes are comparable to those found for the IL1 live time differences, and smaller than the DL1 dead time differences.

L2 Cache

Figure 4.7 shows the categorised distributions of dead time differences for each benchmark for the L2 cache. Overall, the general behaviour is similar to that of the L2 cache

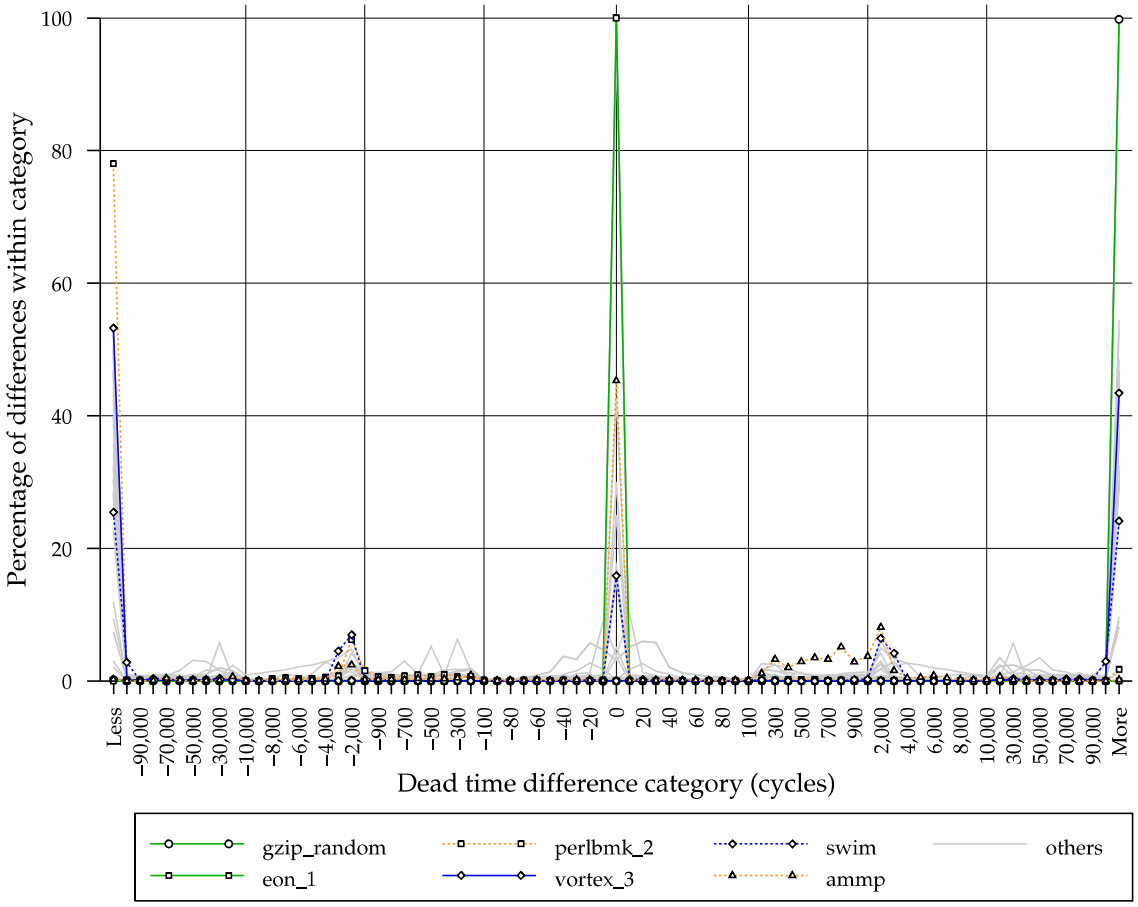

Figure 4.7: Distributions of dead time differences for the L2 cache

live time differences, with significant clusters in the centre and at the extremes of the distributions. Three benchmarks, `eon_1`, `eon_2` and `eon_3`, find *all* their dead times to be identical, indicated by a value of 100% in the central category. Recalling Figure 3.2, these benchmarks have tiny L2 miss rates, therefore the number of generations contributing to these distributions will be very small. Most benchmarks have a much smaller proportion of identical dead times, including all the `gzip` benchmarks, `crafty`, `parser`, `perlbmk_2`, `gap` and `fma3d` which have none. The average proportion of identical dead times is 13.0%.

Moving further outwards from the centre of the distributions, few significant clusters are encountered before the extremes are reached. The proportion of differences at the positive extreme, more than 100,000 cycles, varies from 99.8% for `gzip_random` to 0% for the `eon` benchmarks. The average proportion of differences in this category is 38.5%. The proportion of differences at the other extreme, less than -100,000 cycles, varies from 78.0% for `perlbmk_2` to 0%, again for the `eon` benchmarks. The average proportion of differences in this category is 30.5%. The proportion of dead time differences at the extremes is larger than those found at the extremes of the L2 live time difference distributions, as well as those found at the extremes of the DL1 and IL1 dead time difference distributions.

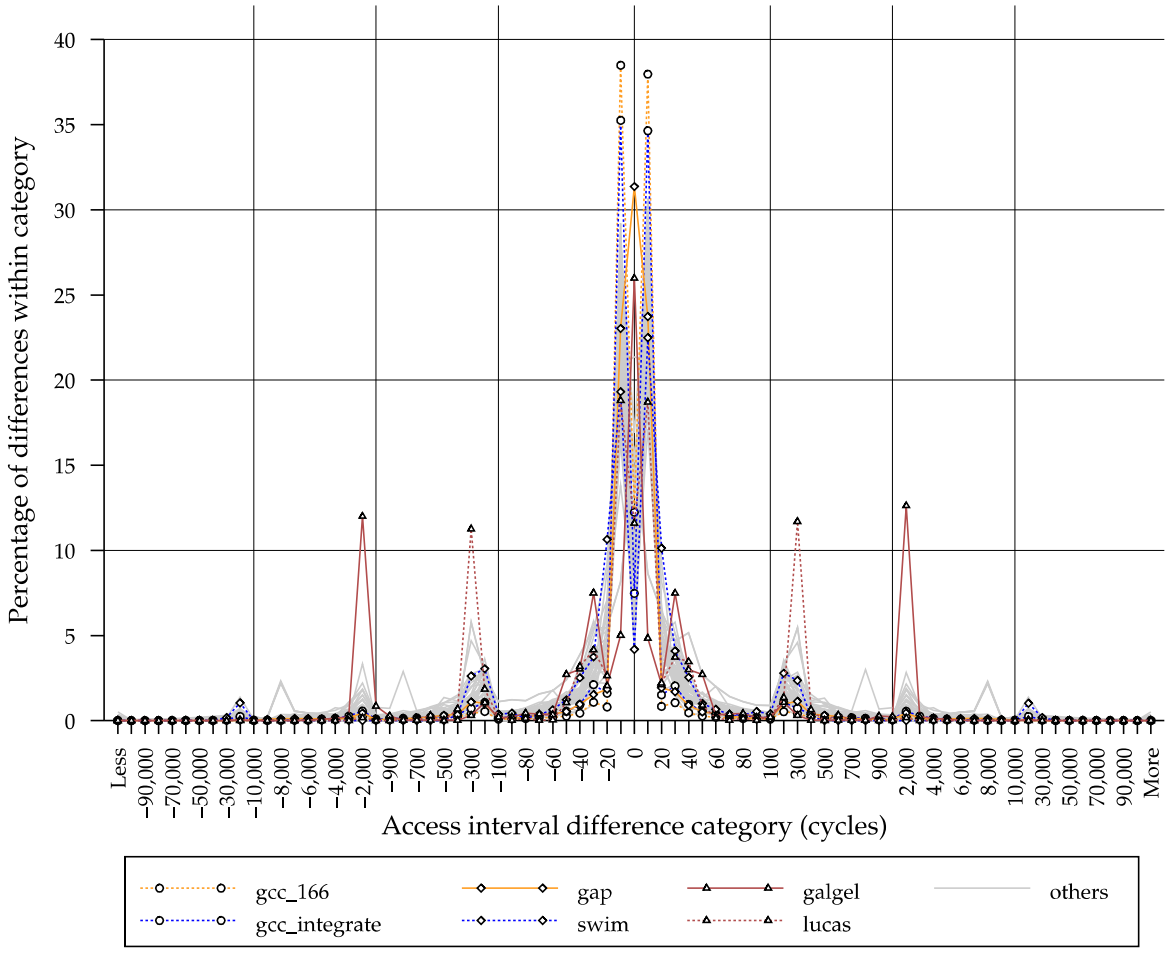

Figure 4.8: Distributions of access interval differences for the DL1 cache

4.3.3 Access Interval Predictability

Figures 4.8, 4.9 and 4.10 show the categorised distributions of access interval differences for each benchmark for the DL1, IL1 and L2 caches respectively. Overall, access interval differences seem considerably more varied than either live or dead time differences, and this is reflected in the fact that the y-axis scale is truncated. Each of the caches is now discussed in turn.

DL1 Cache

Figure 4.8 shows the categorised distributions of access interval differences for the DL1 cache for each benchmark. Unlike the previous distributions of live time or dead time differences, the central clusters are more widely spread. Indeed, the largest single cluster is found for `gcc_166` with 38.5% of access interval differences being between -1 and -10 cycles. For this benchmark, there is an almost identical symmetrical peak in the cluster from 1 to 10 cycles, with 38.0% of differences. Such symmetrical clusters are also exhibited by other benchmarks including `gcc_integrate`, `swim`, `galgel`

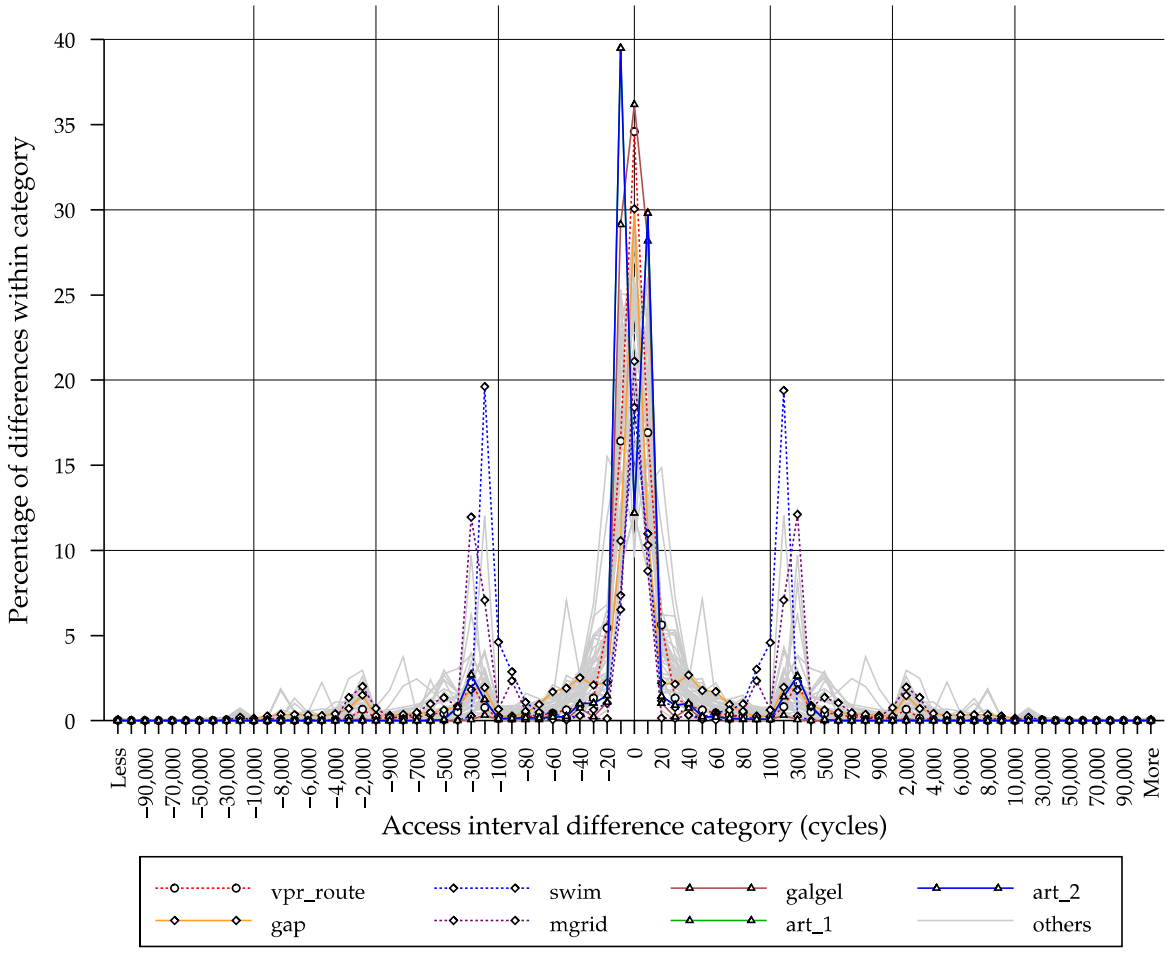

Figure 4.9: Distributions of access interval differences for the IL1 cache

and `lucas`. As previously discussed, symmetrical values such as these indicate alternating values of access intervals. The largest proportion of identical access interval differences is found for `gap` with 31.4%, although the average proportion in this category is only 9.77%, considerably less than the averages of 23.0% and 23.9% found in the neighbouring categories of -10 and 10 respectively. Indeed, fully 80% of access interval differences are found between -100 and 100 cycles. There are negligible proportions of differences at the extremes of the distributions.

IL1 Cache

Figure 4.9 shows the categorised distributions of access interval differences for the IL1 cache for each benchmark. Like the distributions of DL1 cache access interval differences, the vast majority of differences are concentrated around the central area. Again, the largest single cluster is found in the -10 category, with `art_1` and `art_2` both having 39.5% of their overall differences. The next largest proportion comes for `galgel` with 36.2% of access intervals being identical. The average proportion of identical access intervals is 19.8%, considerably higher than the corresponding value of 9.77% for

the DL1 cache access interval differences. The neighbouring categories, -10 and 10 cycles, both have an average of 17.7%. On average, 79% of access interval differences are found between -100 and 100 cycles, almost identical to the proportion for the DL1 cache.

Fewer symmetrical peaks are apparent, although those for `swim` and `mgrid` are particularly large. Again, there are negligible proportions of differences at the extremes of the distributions.

L2 Cache

Figure 4.10 shows the distributions of access interval differences for the L2 cache for each benchmark. Compared to the DL1 and IL1 distributions, far fewer differences have small values, indicated by the lack of central clusters. Recalling Figure 3.15, L2 cache access intervals are far larger than DL1 or IL1 cache access intervals, primarily due to the filtering effect of these first-level caches. Therefore, their differences are expected to be correspondingly larger.

Only two benchmarks show a significant proportion of identical access intervals, `facerec` and `mgrid` with 38.4% and 34.1% respectively. Most other benchmarks show a negligible proportion in this category, with an average of 2.64%. Compared to the DL1 and IL1 cache access interval difference distributions which found, on average, approximately 80% of their differences from -100 to 100 cycles, the L2 distribution finds only 7.56% in the same range. Further out, `perlbmk_2`, `art_1`, `art_2` and `swim` show small clusters, some of which are symmetrical. However, the bulk of the distributions is found at the extremes.

At the positive extreme, more than 100,000 cycles, the proportion of differences varies from 41.3% for `twolf` to 0.111% for `ammp`, with an average of 20.2%. At the negative extreme, less than 100,000 cycles, the proportion varies from 64.1% for `lucas` to 0.166% for `ammp`, with an average of 21.1%.

4.3.4 Summary

Using the same predictability metric as Hu *et al.* [HKM02], namely the difference between consecutive values for each unique cache line, it is clear the predictability of live times, dead times and access intervals varies considerably depending on the particular cache line lifetime metric, the particular cache and the particular benchmark under examination. The following high-level conclusions can be drawn from the results presented above.

- Live times are typically more similar than dead times.
- IL1 cache behaviour is typically more predicable than either DL1 or L2 cache behaviour, particularly for dead times.
- DL1 and IL1 cache access intervals are typically far more similar than L2 cache access intervals, and more similar than any live or dead times.

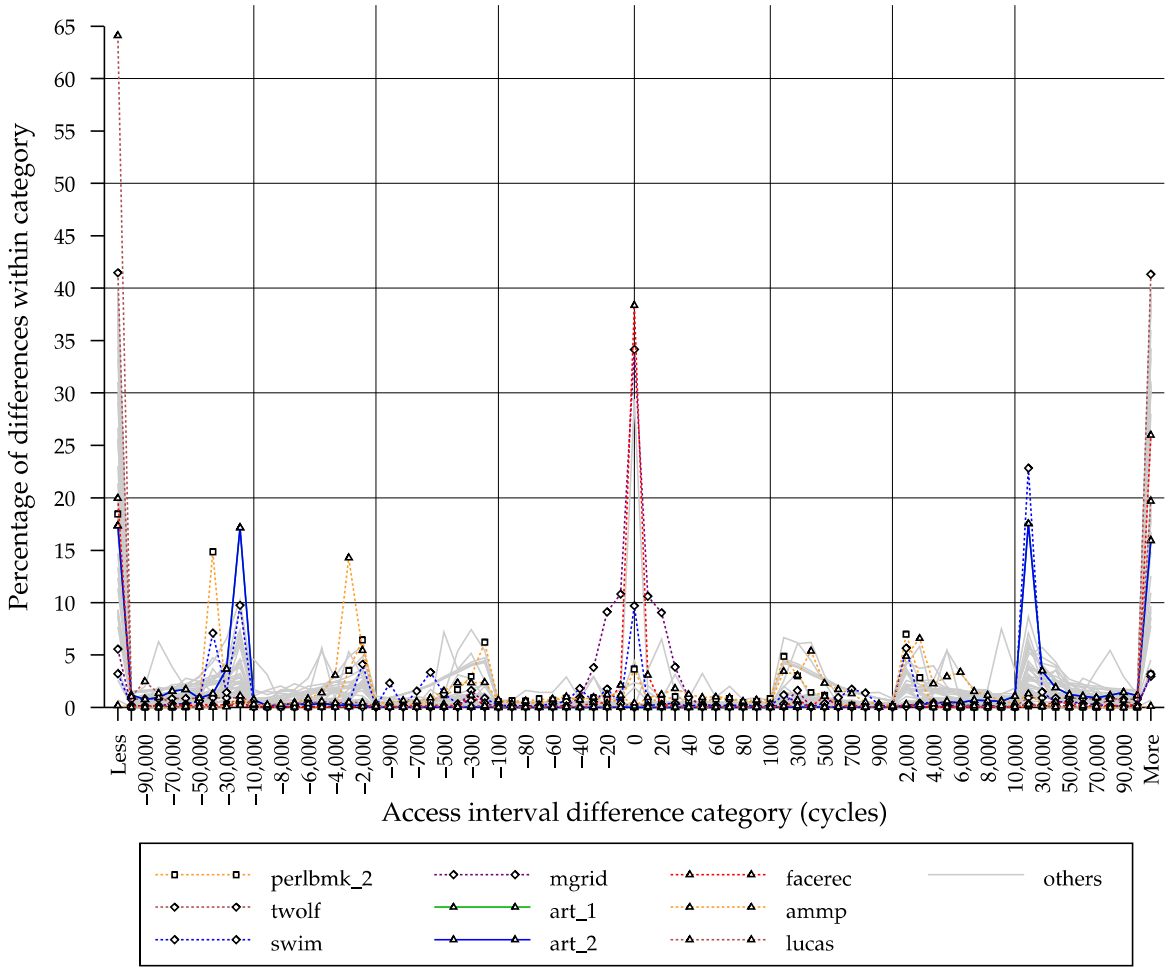

Figure 4.10: Distributions of access interval differences for the L2 cache

4.4 Alternative Predictability Metric

The predictability metric used in the preceding section, namely the difference between consecutive values, is one simple method to assess predictability. However, this approach does not take the magnitude of the original value into account. A small difference may be significant if the original value was small, but relatively insignificant if the original value was large. Instead, this section attempts to take the magnitude of values into account by investigating the ratio of the current value to the previous value.

Figure 4.11 clusters this ratio into various ranges using interval notation. For example, the $[1, 2)$ range indicates that the ratio is greater or equal to one, and less than two. For brevity, live times, dead times and access intervals for all three caches are shown in a single figure using multiple subfigures. The three rows correspond to live times, dead times and access intervals, from top to bottom respectively. The three columns correspond to the DL1, IL1 and L2 caches, from left to right respectively. Unlike the previous discussion of predictability, individual benchmarks are not highlighted, rather general trends are discussed. The results for each benchmark are normalised and expressed as a percentage falling into each range.

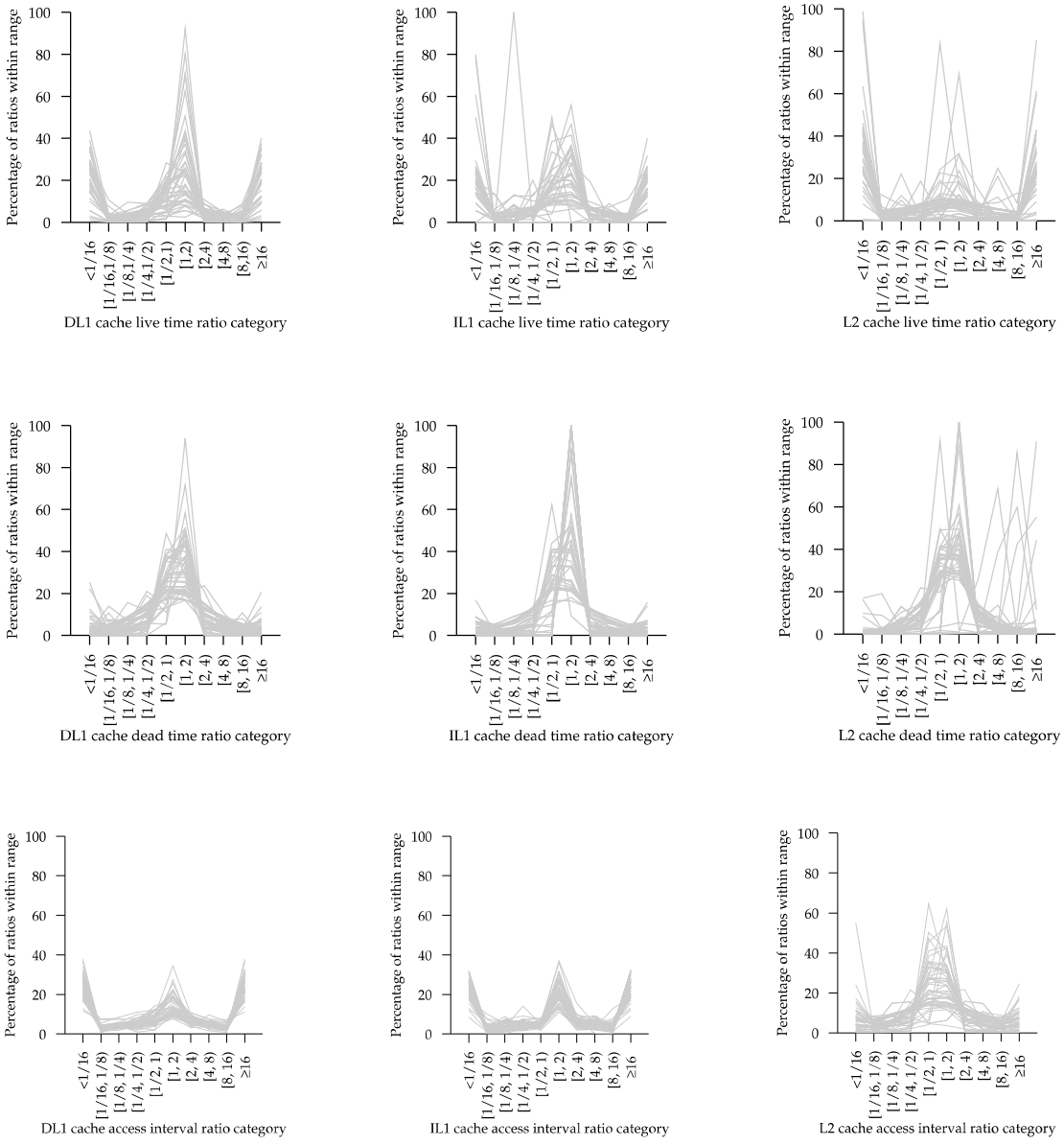

Figure 4.11: Proportions of live time, dead time and access interval ratios falling into various ranges for different caches. Rows are live times, dead times and access intervals from top to bottom. Columns are DL1, IL1 and L2 caches from left to right.

Reviewing the three high-level conclusions drawn in Section 4.3.4 using the original predictability metric, it is clear that the apparent predictability using this alternative metric is different and should be modified as follows.

- Live times are more similar than dead times for the DL1 cache, but less similar for the IL1 and L2 caches.
- DL1 cache live times are more similar than IL1 or L2 cache live times.
- IL1 cache dead times are more similar than DL1 cache dead times, and similar to L2 cache dead times.

- Access interval behaviour for the DL1 and IL1 cache is similar, with approximately one third of ratios falling in each of the centre category and two extreme categories of the distributions.
- L2 cache access intervals are more similar than DL1 or IL1 cache access intervals.

The two different metrics of predictability give rather conflicting results, indicating that such an abstract attempt to assess the likely performance of a predictor depends highly on how predictability is defined. Indeed, these two predictability metrics are only applicable to timekeeping-based predictors such as those proposed by Hu *et al.* [HKM02]. For signature-based predictors, the approach taken by Lin and Reinhardt [LR02] would be more appropriate. Counter-based predictors, proposed by Kharbutli and Solihin [KS05] but not examined in this dissertation, would require yet another predictability metric. Instead, a more practical approach is now taken to implement and assess predictors of cache line lifetime metrics.

4.5 Predictors

Having examined the predictability of live times, dead times and access intervals using two different predictability metrics, the following sections detail mechanisms for efficient online prediction of cache line lifetime behaviour. The questions of *what* to predict, *when* to predict it and crucially *how* to predict it are addressed, including a discussion of how to evaluate such predictors. The various predictors presented are heavily influenced by the mature fields of branch prediction and value prediction, two topics which are also reviewed.

What to Predict

Chapter 3 detailed several metrics describing the lifetime of cache lines within the memory hierarchy. These were the live times, dead times and access intervals, as defined in Section 3.4. The primary aim of this dissertation is to show that predicting these, or closely related, metrics enables a number of performance optimisations, and that suitable predictors may be implemented using reasonable additional hardware resources. Chapter 5 details two performance enhancements which rely on the following predictors, while Chapter 6 outlines other potential applications of these predictors.

The specific predictor examined in this dissertation is a **liveness predictor**, which predicts whether a specific cache line is live i.e. will be referenced again prior to eviction. Liveness predictors may be **direct**, in which the liveness of a cache line is predicted exactly, or **indirect** in which another cache line lifetime metric is predicted from which a liveness prediction can be inferred.

Related work sometimes uses different terms for what are essentially the same predictors and these terms will be mentioned where necessary.

When to Predict

It is initially tempting to try and compare predictors in isolation without regarding their potential applications. However, such an approach soon runs into a practical difficulty — when should the predictor be invoked? For branch prediction, the answer to this question is simple — a prediction is required whenever a branch instruction is fetched, in order to maintain a constant stream of instructions in the processor's pipeline. Thus the performance of a branch predictor is dependent not only upon its structure, but also upon the distribution of branches in the instruction stream. Unlike branch prediction, cache line lifetime predictors have different applications, and both the mode of predictor invocation and the cache lines on which the predictor is invoked will vary between applications.

In addition, when assessing the suitability of a predictor for a particular application, the penalty for a misprediction must be taken into account. The implementation must also justify additional hardware resources — for example, the addition of a prediction scheme should not only improve overall performance, but also the additional die area overhead should not be better used by simply expanding the existing cache size.

While most previously proposed predictors are invoked on every access to a cache line, Liu *et al.* argue that prediction should be delayed as long as possible in order to exploit additional historical events that may take place between the prediction being made, and the result of the prediction being required [LFHB08].

For these reasons, while candidate predictors are described in this chapter, actual evaluation of predictors is deferred until their precise applications are determined in Chapter 5.

How to Predict

Having addressed the questions of what and when to predict, the critical question of how to make predictions is now considered. Section 4.3 demonstrated that consecutive values of live times, dead times and access intervals are reasonably consistent, so a prediction mechanism which uses this historical information to predict future events would seem appropriate.

Two approaches are considered — the first uses **binary prediction**, to directly predict the liveness of a cache line, while the second uses **value prediction** to indirectly predict the liveness of a cache line through prediction of the live time or access interval for the cache line. These two approaches are now considered in turn.

4.6 Binary Prediction

Direct liveness predictors are examples of a binary predictor i.e. a predictor for an event with only two definitive outcomes. The two definitive outcomes of a binary predictor are **correct** (the prediction matched the observation) and **incorrect** (the prediction differs from the observation). A third outcome, **unknown**, is possible in the case that the predictor is unable to make a definitive prediction at the time it is required. This latter case may occur during the initial training phase when there is insufficient history of past events to make a timely prediction. These possible outcomes are combined to yield the traditional **accuracy** and **coverage** metrics used widely in computer architecture to evaluate binary predictors.

Accuracy is defined as the fraction of predictions made which turn out to be correct i.e.

$$accuracy = \frac{correct}{correct + incorrect}$$

Coverage is defined as the fraction of predictions which are able to be made i.e.

$$coverage = \frac{correct + incorrect}{correct + incorrect + unknown}$$

Binary predictors with high accuracy and coverage are frequently required for the task of branch prediction, a very mature field which is now briefly reviewed.

4.6.1 Branch Prediction

Branch prediction is one example of speculative execution whereby a continuous stream of instructions is fetched into the processor's pipeline before determining whether the correct stream of instructions is being followed. A **branch predictor** has a binary output, predicting only whether a particular branch will be **taken** or **not-taken** before the actual branch condition can be checked. Since accurate control flow speculation is paramount to achieving high performance in dynamically scheduled superscalar processors, instruction fetch units are being integrated with branch prediction and prefetch units, leading to the development of **trace caches**, proposed by Rotenberg *et al.*, which store streams of decoded instructions in the order in which they have been previously executed rather than conventional program order [RBS96].

Note that computation of the target address (dependent on the prediction) takes place later in the pipeline, so a separate structure, a **branch target buffer** may be used in an attempt to further reduce the branch penalty. Similarly, a **return address predictor** may be used to cope with the common and important special case of indirect branches returning from procedure calls.

Static Branch Prediction

The simplest form of branch prediction is **static** i.e. independent of any run-time processor state. Examples of such a predictor include an **always-taken** (or equally **never-taken**) predictor, separate branch instructions used by a compiler to hint whether a particular branch will be taken (or not), or the simple **heuristic** that "backwards branches are taken, forwards branches are not taken" which copes reasonably well with simple nested looping constructs.

Dynamic Branch Prediction

Dynamic branch prediction takes previous run-time program behaviour into account when predicting whether a particular branch will be taken (or not) and as such, relies heavily upon past program behaviour being indicative of future behaviour.

One of the simplest dynamic branch predictors is a table of **bimodal** counters, indexed by the significant low-order bits of the program counter of a branch instruction. Just two bits are usually used in each counter, which is incremented if a branch is known to be taken and decremented if a branch is known to be not taken, saturating at 0 and 3. A branch is predicted as being taken if the corresponding counter entry is 2 or 3, otherwise it is predicted as not being taken.

Alternatively, a **local history** predictor may be used. One example of such a predictor uses a table of n-bit shift registers, indexed by the significant low-order program counter bits. Each shift register stores the taken (1) or not-taken (0) history for the corresponding branches. The value of the shift register is then used to index a second table containing bimodal counters as previously described. Such a configuration is termed "Per address history table, global pattern table" or **PAg** by Yeh *et al.* [YP92]. Other similar configurations are illustrated in Figure 4.12 using Yeh *et al.*'s classification which

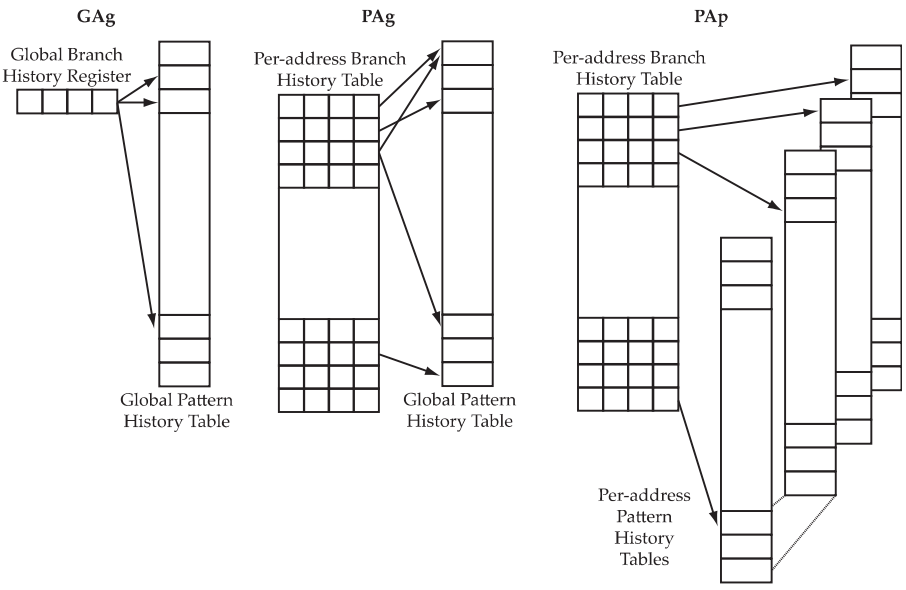

Figure 4.12: Branch predictor classification proposed by Yeh *et al.* [YP92]

differentiates between storing state per-address (or subset of addresses) and globally (i.e. for all addresses).

Global state may also be combined with per-address state by concatenating (**gselect**) or XORing (**gshare**) global branch history with significant low-order program counter bits. Finally, independent global and per-address predictors may be combined as proposed by McFarling , with a third **meta** or **choice** predictor to choose between the available predictions should they disagree [McF93].

4.6.2 Static Direct Liveness Predictors

First considering the simple static predictors, Table 4.1 shows some approximate equivalents to each of the static branch predictors previously described when applied as a liveness predictor.

Branch Predictor	**Liveness Predictor**	**Name**
Always taken	Always live	`AlwaysLive`
Never taken	Never live	`NeverLive`
Compiler taken / not taken hints	Compiler live/ dead hints	n/a
”Backwards taken, forwards not taken” heuristic	“Stack references live, heap references dead” heuristic	`StackLive` `HeapDead`

Table 4.1: Equivalent static branch and liveness predictors

The `AlwaysLive` and `NeverLive` predictors correspond to applications in which the speculative action is either always or never performed and are included for completeness rather than actual implementation, and these scenarios are evaluated without the prediction overhead.

Those predictors which require compiler interaction necessitate potentially complex compile-time analysis and are considered out of the scope of this dissertation, however, remain interesting areas for future work.

The novel `StackLiveHeapDead` heuristic predictor is indirectly motivated by Lee and Tyson's region-based caching [LT00] which partitions the cache depending on the memory region (stack, global or heap) being accessed. Lee and Tyson show that average generation time (measured in terms of the number of accesses) and average miss rates for the three memory regions varies considerably, with stack addresses exhibiting long generation times and low miss rates while heap addresses exhibit shorter generation times and higher miss rates.

4.6.3 Dynamic Direct Liveness Predictors

Static direct liveness predictors are simple to implement and provide complete coverage, however they are expected to perform poorly in general-purpose applications since they are unable to adapt to the dynamic run-time behaviour of the application. Predictors which directly predict liveness are more commonly known as **Last-Touch Predictors** (LTPs), which predict if a particular memory operation is the last one to touch a cache line prior to its eviction. Lai and Falsafi proposed the first last-touch predictors, for the application of self-invalidation in a cache coherent multiprocessor system [LF00]. Their approach builds up a **trace** of accesses to a cache line, encoded as a **signature**, and attempts to match this signature against previously observed signatures of last-touch references. Each element within Lai and Falsafi's trace is simply the program counter of each instruction touching the cache line, and the signature is formed by truncated addition of these values. The previously observed signatures can either be stored per cache line (in a PAp-like organisation) or globally (in a PAg-like organisation). The advantage of the PAg-like organisation is that storage is more effectively utilised since the number of signatures per cache line may vary widely across cache lines. However, there is a risk of **subtrace aliasing** whereby sections of two otherwise different traces may be identical, resulting in identical signatures and potential mispredictions. Attempting to gain the benefits of both organisations, Lai *et al.* propose hashing the signature with the address being accessed in their **Dead-Block Predictor** (DBP) [LFF01]. While the dead-block predictor uses the same signatures as Lai and Falsafi's last-touch predictor, the **Dead-Block Correlating Prefetcher** (DBCP) adds the two most recent addresses previously mapped to the cache line [LFF01].

Extending signature construction further, Lin and Reinhardt investigate a variety of different signatures to predict last-touch references [LR02]. Table 4.2 lists the signature elements they consider.

Different numbers of these elements may be combined in different ways. Lin and Reinhardt investigate thirteen combinations using concatenation to combine elements while Lai *et al.* [LFF01] and Lai and Falsafi [LF00] use truncated addition[2] to combine multiple elements in a trace. Concatenation is clearly highly space inefficient, requiring approximately one machine word per element, but allows unique signature identification while truncated addition risks the potential of interference between two

[2] **Truncated addition** adds two words of the same size but ignores any carry from the most significant bit

Name	Description
`Addr`	The target address of the current reference
`AddrAny`	The target address of the previous reference to any target address
`AddrSet`	The target address of the previous reference to the current cache set
`AddrPC`	The target address of the previous reference with the current program counter
`PC`	The program counter of the current reference
`PCAny`	The program counter of the previous reference to any address
`PCSet`	The program counter of the previous reference to the current cache set
`PCAddr`	The program counter of the previous reference to the current address
`PCAnyPrev`	The program counter of the second most recent reference to any target address
`PCSetPrev`	The program counter of the second most recent reference to the current cache set
`PCAddrPrev`	The program counter of the second most recent reference to the current address

Table 4.2: Elements of signatures investigated by Lin and Reinhardt [LR02]

traces with different signature elements but the same overall signature. Such interference may be **constructive** (identical signatures corresponding to the same type of event), increasing accuracy, or **destructive** (identical signatures corresponding to different events), decreasing accuracy. Somewhat paradoxically, both forms of interference increase coverage.

Even if the number of elements in each signature is limited to three, the total number of potential signatures is over 150 which doubles if the two methods of combining elements is considered. Rather than an exhaustive comparison of signatures, a list of previously proposed signatures which have been shown to perform well are considered, as listed in Table 4.3. The `1PC2PAddr` signature, shown to perform well by Lin and Reinhardt's rather theoretical study, is omitted from consideration since implementation would require tracking the previous address accessed by every single memory instruction.

Note that Hu *et al.*'s signatures are not used as a conventional last-touch predictor, but to index into a table of predicted live times, as described shortly in Section 4.7.2.

Last-touch Predictor Implementation

Lin and Reinhardt [LR02] concentrate on the inherent predictability of last-touches using signatures, so consider an unbounded store of signatures. Other related work is more concerned with implementing signature-based predictors for particular applications, so detail the specific structure used to store signatures. Hu *et al.* observe little

Name	Elements	Citation
`1Addr`	Address of current reference	[LR02]
`2SAddr`	Address of current reference Address of previous reference to the same set	[HKM02]
`1Addr3APC`	Address of current reference PC of current reference PCs of two previous references to same address	[LF00] [LFF01]
`1PC2Addr`	Address of current reference PC of current reference Address of previous reference to any address	[LR02]
`1PC`	PC of current reference	[LF00]

Table 4.3: Signature combinations investigated in this dissertation

variation with size so use an 8kB, 8-way set associative **address correlation table**, indexed by part of the signature [HKM02]. The remainder of the signature is used as a tag to disambiguate potentially conflicting entries. Lai *et al.* consider two implementations — a 2 MB, 8-way set-associative **on-chip correlation table** and a 7.6 MB, 16-way set-associative **off-chip correlation table** [LFF01]. Approximately 60% of the storage is occupied by tags and history, the rest is used to predict which cache line to prefetch in their particular performance optimisation. As previously described, Lai and Falsafi use a two-level structure inspired by branch predictors, adding up to 1 MB of storage [LF00].

Each predictor is made up of two logical tables. The first table stores the current signature associated with each cache line, while the second stores previously-observed signatures which lead to a dead cache line, known as last-touch signatures. On each cache access, the current signature table is updated for the corresponding cache line(s). On a cache miss, the signature of the cache line being replaced is transferred to the last-touch signature table. On predictor invocation, the last-touch signature table is searched for a signature corresponding to the cache line under consideration. This leads to the observation that unlike much previous work, predictor update (on each cache access) and predictor invocation (dependent on application) can potentially be decoupled. This allows more history to be included before the prediction is made, potentially improving accuracy and coverage, as well as allowing more complex, highly-associative but slower structures to store previously-observed signatures.

The last-touch predictor implementation chosen is similar to that used by Hu *et al.*, with the signature used to index a set-associative table of previously observed signatures. The last-touch signature table is very similar to the tag array of a conventional cache and is illustrated in Figure 4.13. If the signature is found in the last-touch signature table, the cache line is predicted to be dead, otherwise it is predicted to be live. Entries in the last-touch signature table are managed using a least-recently used replacement policy. It is desirable to add a bimodal counter to reduce hysteresis, as used in branch prediction. However, this would require frequent updates of the last-touch signature table which may not be possible.

The last-touch predictor actions can be summarised as follows:

- **On cache hit**: Update current signature.

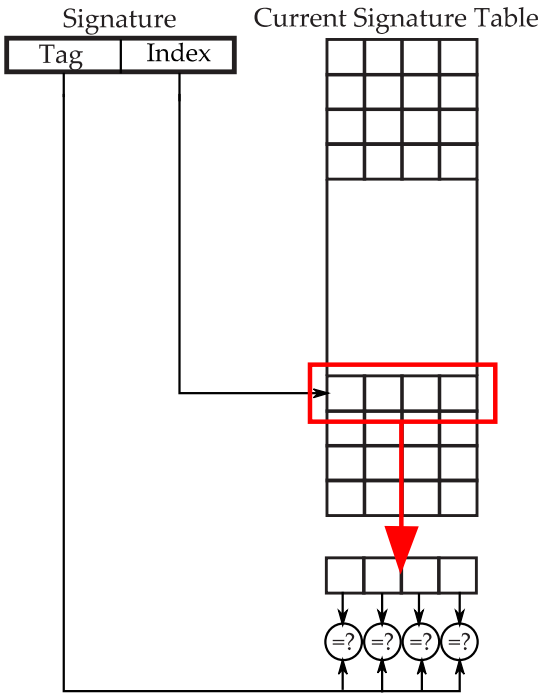

Figure 4.13: Four-way set-associative last-touch signature table

- **On cache miss**: Update current signature & transfer to last-touch signature table.
- **On prediction**: Query the last-touch signature table for the current signature of the corresponding cache line.

Other Related Work

Lee *et al.* propose a scheme to even out writeback traffic by speculatively writing back dirty cache lines before they are evicted from the cache [LTF00]. The cache lines chosen for early writeback are those in the least-recently used position in the replacement stack already maintained for each cache line. Cache lines may either become eligible for early writeback when transitioning into the LRU state, or periodically via an **autonomous eager writeback** scheme. This predictor may easily be cast into the liveness predictor framework by considering a predictor which considers some fraction of the LRU stack to be live or dead. In the `MRUlive` predictor, the n most recently used lines are considered to be live. The primary advantage of this predictor is that it requires very little additional hardware above that which is already implemented for the LRU cache replacement algorithm.

4.7 Value Prediction

As previously described, branch prediction takes place in the instruction decode (ID) stage of the classic RISC five-stage pipeline. However, even in this relatively shallow pipeline, it is unable to entirely eliminate the penalty of a taken branch, since the

branch predictor is only invoked once the branch instruction has been decoded, during which time the next instruction has already been fetched from the incremented program counter location, which for a taken branch is the incorrect address. In order to further reduce the taken branch penalty, branch prediction may be moved into the instruction fetch (IF) stage, the first stage in the classic RISC five-stage pipeline. In this case, a **branch target buffer (BTB)** is used which augments the existing branch predictor by speeding up delivery of the predicted instruction stream. The BTB is indexed by the current program counter and if a match is found, the corresponding instruction is predicted to be a branch, with each BTB entry containing data to help reduce the branch penalty. Perleberg examines the design space and complexity of BTBs, ranging from low-complexity containing just prediction information for each entry, through medium-complexity incorporating predicted target addresses for each entry and finally to high-complexity which also incorporate instructions from the predicted target address [PS93]. The degree of complexity of the BTB determines the extent to which it can reduce or entirely eliminate branch penalties, but equally the hardware resources required for implementation. Since mispredictions may occur, the processor pipeline must support squashing incorrect instructions, and also updating of the BTB contents if either the predicted direction or the predicted branch target address is incorrect.

Branch target prediction is an example of **value prediction**[3], whereby the event being predicted can have numerous possible definitive outcomes, in contrast to the previously discussed family of binary predictors which only have two possible definitive outcomes. In the case of branch targets, the number of possible outcomes is theoretically any instruction in the entirety of the program. However, the extremely repetitive behaviour exhibited by many programs allows branch target prediction to exhibit acceptable performance.

Another example of value prediction is the technique of **load value prediction**, first introduced by Lipasti *et al.* [LWS96]. This approach speculatively predicts the *value* of memory locations before the actual value can be provided by the memory subsystem. If the prediction is correct, the access latency of that memory reference can be eliminated. However, if the prediction is incorrect, that load instruction and any dependent instructions must be replayed. This technique was soon extended by Lipasti and Shen to apply to all register-producing instructions and is now known simply as **value prediction** [LS98]. This corresponds to the most general case of predictor, since any value within the range of a register may be possible as an outcome. For a typical machine, this may be 2^{32} or 2^{64} but again, due to the repetitive behaviour exhibited by many programs, acceptable performance is achievable.

Yet another example of value prediction is the technique of **prefetching**, as previously detailed in Section 2.6.2. Again, the range of potential outcomes is vast, essentially any memory address, but the repetitive behaviour exhibited by many programs allows reasonable performance to be achieved. Prefetching implementations are always safe, in the sense that a misprediction never results in incorrect control or data flow, so no rollback support is required. However, poor prefetching accuracy will have significant negative performance and power consumption effects due to contention with demand fetch instructions.

Rather than directly predicting the liveness of a cache line, value predictors may be

[3]The term **value prediction** is being used here in a more general sense than that implied by the original value prediction work of Lipasti and Shen [LS98]

used to predict the value of a particular cache line lifetime parameter, which can be converted into a prediction of liveness by considering other information such as the time a cache line was brought into the cache, as well as the current time. As with binary predictors, these predictors may be split into static predictors (which are independent of run-time behaviour) and dynamic predictors (which depend upon run-time behaviour). These two classes of predictor are now discussed in turn.

4.7.1 Static Indirect Liveness Predictors

Previous work by both Kaxiras *et al.* [KHM01] and Hu *et al.* [HKM02] propose a simple static dead line predictor which predicts that cache lines which have not been accessed for more than a fixed number of cycles are dead. Kaxiras *et al.* determine this threshold by considering the field of competitive algorithms and using the generic policy of taking action when the extra cost incurred by *not* taking action up until the current time is equal to the extra cost that would be incurred if action is taken and that action turns out to be wrong. This policy bounds the worst case cost within a factor of two of the optimal algorithm. The costs referred to in this policy vary by application and like Kaxiras *et al.*, a variety of fixed thresholds will be examined in the `Threshold` predictor. Hu *et al.* examine the distributions of access intervals and dead times, settling on a fixed threshold between the two of 5120 cycles in order to yield adequate coverage and accuracy.

The performance of a given threshold may be estimated by considering the cumulative distribution of live and dead times for each cache, Figures 3.7, 3.8, 3.9, 3.10, 3.11 and 3.12. Intuitively, if the threshold is too low, the accuracy of the predictor will be low while the coverage will be high. Conversely, if the threshold is too high, the coverage of the predictor will be low while the accuracy will be high. The `Threshold` predictor effectively predicts a cache line as dead if it has not been accessed for more than the threshold number of cycles. It can equally be applied as a liveness predictor, predicting a cache line to be live if it has not been accessed for less than the threshold number of cycles.

Besides the actual value of the threshold used, the `Threshold` predictor has another parameter, the point in time which the threshold value is relative to. Both Hu *et al.* and Kaxiras *et al.* use the time at which the cache line was last accessed. Alternatively, the time at which the cache line was first brought into the cache may be used, referred to as the **fetch time**, and effectively yielding a predictor of a cache line's live time. The various options for the `Threshold` predictor are summarised in Table 4.4.

Value Predicted	**Threshold Reference**	**Semantics**	**Citation**
Access Interval	Last Reference	$(now - lastref) < threshold \rightarrow live$	[KHM01] [HKM02]
Live Time	Fetch time	$(now - fetchtime) < threshold \rightarrow live$	Novel

Table 4.4: Threshold predictor semantics

4.7.2 Dynamic Indirect Liveness Predictors

Rather than operating with a single fixed threshold, Kaxiras *et al.* investigate dynamic approaches whereby the threshold is modified at run-time in response to observed behaviour [KHM01]. The threshold is increased if the current value is judged to be too pessimistic, and decreased if the current value is judged to be too optimistic. Kaxiras *et al.* assume a misprediction has occurred if a cache line is accessed shortly after being predicted as dead. However, this approach is unsuitable for the applications described in Chapter 5, since once a cache line has been predicted as dead, the optimisation has already taken place thus tracking mispredictions with `Threshold` predictors is difficult without resorting to duplicating large architectural elements such as the cache tag array.

Dynamic Live Time Predictor

Instead, the approach taken by Hu *et al.* is followed and considerably extended. Rather than adaptively varying a threshold, the live time of a cache line is directly predicted. Thus knowing when the cache line was brought into the cache, together with the current time, the predicted liveness can be inferred. The history for a cache line is summarised using a trace-based signature which indexes a live time prediction table, similar to the last-touch predictor. Recalling Figure 4.11, the live time for a DL1 or IL1 cache line is often less than twice the previous live time for that cache line, therefore the predicted live time will be twice the previous live time, the same heuristic used by Hu *et al.*

The dynamic live time predictor actions can be summarised as follows:

- **On cache hit**: Update current signature.
- **On cache miss**: Calculate live time of the outgoing cache line & transfer to live time prediction table, indexed by current signature.
- **On prediction**: Query the live time prediction table with the current signature of the corresponding cache line.

The method by which the live-time prediction table is organised is similar to that used for the last-touch prediction table, illustrated in Figure 4.13. If a live time cannot be found for a particular signature, the corresponding cache line is assumed to be dead.

Unlike the DL1 and IL1 cache live times, predictability of consecutive L2 cache live times is considerably lower so an alternative method is sought.

Dynamic Access Interval Predictor

As well as dynamic prediction of live times using trace-based signatures, access intervals can also be predicted. Figure 4.11 shows that consecutive access intervals to the L2 cache are far more predictable than consecutive live times, therefore a novel alternative dynamic access interval predictor is more applicable.

The dynamic access interval predictor actions can be summarised as follows:

- **On cache hit**: Update current signature, calculate new access interval & transfer to access interval prediction table, indexed by previous signature.
- **On cache miss**: Do nothing.
- **On prediction**: Query the access interval prediction table with the current signature of the corresponding cache line.

As with the dynamic live time predictor, the previously observed access interval is doubled and combined with the time of the last access to a cache line and the current time, providing a predictor for liveness as required.

4.8 Summary

This chapter began by examining the inherent predictability of live times, dead times and access intervals using two different predictability metrics. The questions of *what* to predict, *when* to predict it and crucially *how* to predict it were addressed, followed by a detailed examination of a variety of different predictors. Binary predictors of liveness were presented, inspired by branch prediction, as well as value predictors of live times and access intervals. Implementation details and tradeoffs were discussed, but final choices depend upon the particular applications which are described in the next chapter.

The various predictors introduced in this chapter are summarised in Table 4.5. The `StackLiveHeapDead` and `DualThresholdLiveTime` predictors are described in Chapter 5.

	Direct	**Indirect**
Static	`AlwaysLive`	`ThresholdLiveTime`
	`NeverLive`	`ThresholdAccessInterval`
	`StackLive`	`DualThresholdLiveTime`
	`HeapDead`	
	`StackLiveHeapDead`	
Dynamic	`LastTouch_*`	`LiveTime_*`
	`MRULive`	`AccessInterval_*`

Table 4.5: Summary of predictors

5 Applications

5.1 Overview

The aim of this chapter is to demonstrate that prediction of cache line lifetime metrics may be used in two different applications to decrease cache miss rates, and to improve overall cache utilisation and system performance.

Having previously defined various cache line lifetime metrics in Chapter 3 and described methods to predict them in Chapter 4, this chapter shows how these predictors may be applied in two different applications. The first application is a filtered victim cache, in which allocation of an outgoing cache line to the victim cache is made depending on its predicted live or dead state. The second application is selection of prefetch victims, in which a victim cache line is chosen for replacement depending upon its predicted live or dead state. Accuracy and coverage of predictors for both applications are examined in detail, with full results presented for the best-performing predictors together with a comparison against relevant previous research.

5.2 Victim Cache Management

Section 2.6.1 described the well-established concept of a victim cache, a small highly-associative cache placed in the refill path of a conventional **main cache** which retains cache lines recently evicted from the main cache, and which aims to reduce the number of conflict misses by effectively adding further associativity to selected cache sets. In this application, a liveness predictor is invoked during the management of a victim cache. The liveness predictor speculates whether a cache line being replaced in the main cache is still live (i.e. will be accessed again prior to eviction) and if so, the cache line is stored in the victim cache rather than being immediately evicted to the next level in the memory hierarchy.

A conventional victim cache design allocates all cache lines being replaced in the main cache to the victim cache, regardless of usage patterns. It is anticipated that the proposed **Filtered Victim Cache** (`FVC`) can increase the benefits provided by a conventional victim cache by making more efficient use of the limited storage available through only retaining those cache lines which will be shortly reused.

5.2.1 Method

As previously, results are presented from simulations over the entire SPEC CPU2000 integer and floating-point benchmark suite. The benchmarks are executed using the

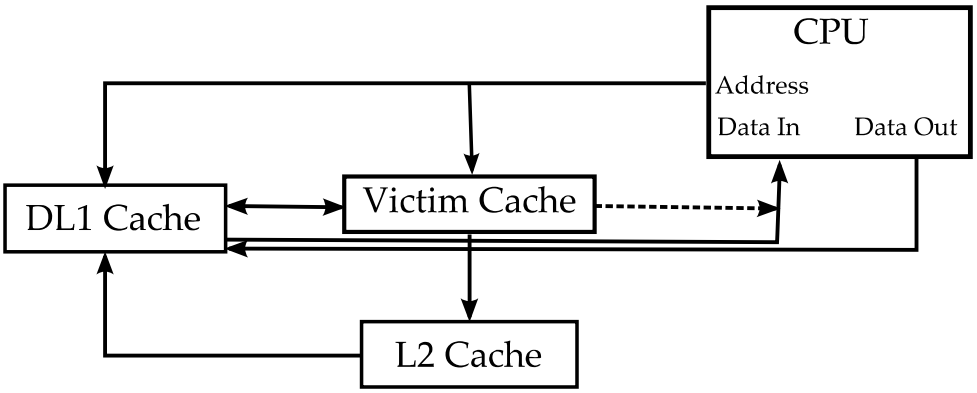

Figure 5.1: Simple victim cache design. Adapted from Hennessy and Patterson [HP03].

reference set of inputs, with the first one billion cycles executed using a simplified processor model to warm up the caches and to bypass any initialisation code before running the next two billion cycles with a detailed processor model.

Victim caches are mostly beneficial alongside main caches of limited associativity with high numbers of conflict misses, such as the L1 caches of the baseline system configuration. The L2 cache has higher associativity, hence fewer conflict misses, hence a victim cache would be less beneficial. In addition, the IL1 cache generally has a very low overall miss rate, hence a victim cache would only provide very limited benefit.

Even limiting the victim cache to implementation alongside the DL1 cache, limited benefits are anticipated since the baseline DL1 cache miss rate is still relatively low. Instead, the DL1 cache size and associativity are decreased to 32 kB and direct-mapped respectively, in order to best gain any benefits from a victim cache. This **revised baseline** configuration is similar to that used by Hu *et al.* with similar benchmarks [HKM02].

The benefits provided by a conventional victim cache are first investigated. The accuracy and coverage of various predictors are then evaluated, and full results for the best performing predictors are evaluated against relevant previous research.

5.2.2 Conventional Victim Cache

Figure 5.1 shows the design of a simple victim cache (`VC`), adapted from Hennessy and Patterson [HP03].

The victim cache is accessed in parallel with the DL1 cache[1] and despite its high degree of associativity, its small size allows similar latency compared to accessing the larger main cache. If an access misses in the DL1 cache but hits in the victim cache, the corresponding cache lines are swapped. If an access misses in both the DL1 cache and the victim cache, the requested cache line is fetched from the L2 cache and placed in the DL1 cache, potentially evicting a cache line which is placed in the victim cache. This

[1]Note that some victim cache designs access the DL1 cache and victim cache sequentially to reduce power consumption. However, in this section, it is assumed that they are accessed in parallel to minimise latency and hence maximise performance.

may evict another cache line from the victim cache to the L2 cache. Thus, as shown in Figure 5.1, datapaths are required as follows:

- From the DL1 cache to the CPU
- From the CPU to the DL1 cache
- Between the DL1 cache and the victim cache (bidirectional)
- From the L2 cache to the DL1 cache
- From the victim cache to the L2 cache

While an additional bypass datapath from the victim cache directly to the CPU may be included (shown dashed in Figure 5.1), it would add to the critical path of a DL1 cache hit, which is hopefully the most frequent occurrence. Therefore the latency of a victim cache hit is the sum of the time taken to access the victim cache itself and the time taken to swap cache lines between the DL1 and victim caches. In line with previous work, it is assumed that the latter component can be satisfied in a single cycle. The access time of the victim cache itself is estimated by the CACTI 4.2 cache modelling tool [TTJ06], the most recent version of CACTI to support fully-associative caches, assuming a 45 nm fabrication process. Using this tool, the largest victim cache which can be accessed in a single cycle has eight cache line entries, and when added to the single cycle to swap cache lines gives an overall victim cache latency identical to the DL1 main cache.

A variety of replacement policies may be used for the victim cache, but again, in line with previous work, traditional least-recently used (LRU) is assumed. The performance of the conventional victim cache configuration is now assessed using three different metrics.

Miss Rate

Figure 5.2 shows the percentage decrease of the DL1 cache miss rate for the conventional victim cache configuration, compared to the revised baseline configuration, for each benchmark. As previously, the miss rate is measured as the number of Misses Per Thousand Instructions (MPKI). As would be expected, for every benchmark the conventional victim cache configuration reduces the miss rate. The average reduction in miss rate is 23%, and varies from 1.4% for `swim` and 1.5% for `art_1` and `art_2`, to 97% for `perlbmk_2` and 57% for `eon_3`.

Cache Utilisation

Figure 5.3 shows the percentage point change of the DL1 cache utilisation for the conventional victim cache configuration, compared to the revised baseline configuration. Note that since cache utilisation is already measured as a percentage, Figure 5.3 shows the percentage *point* change, i.e. the numerical difference between the two respective percentages.

The general trend is that the conventional victim cache configuration has very little impact on DL1 cache utilisation. This is not surprising, since only a small number of

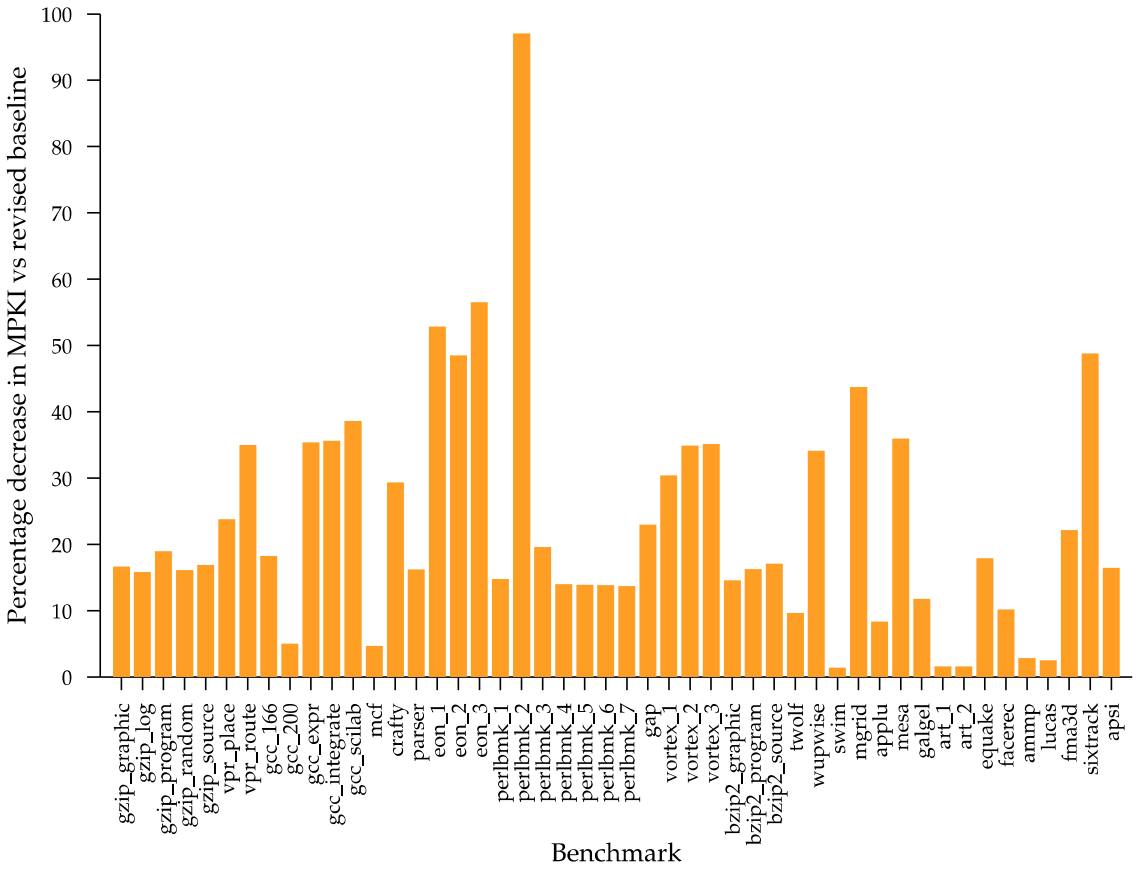

Figure 5.2: DL1 percentage miss rate decrease for conventional victim cache configuration *vs* revised baseline configuration

additional cache lines are provided by the victim cache, hence the time for which a cache line is resident in the victim cache is short, hence the potential impact on cache utilisation is small. The average change in cache utilisation is 0.042%, ranging from -0.65% for `perlbmk_6` to 4.0% for `perlbmk_2`.

Instructions Committed per Cycle

Having examined the impact of the conventional victim cache configuration on the DL1 cache miss rate and the DL1 cache utilisation, the overall system performance is examined using the number of Instructions committed Per Cycle (IPC) metric previously discussed. Figure 5.4 shows the percentage change in IPC for the conventional victim cache configuration, compared to the revised baseline configuration.

The overall trend is that the conventional victim cache configuration improves IPC, although generally by a small amount. Recalling Figure 3.1, this behaviour is to be expected, since very limited IPC improvements were encountered even with *perfect* L1 caches, indicating that the overall system is tolerant of the L1 cache latency and miss rates.

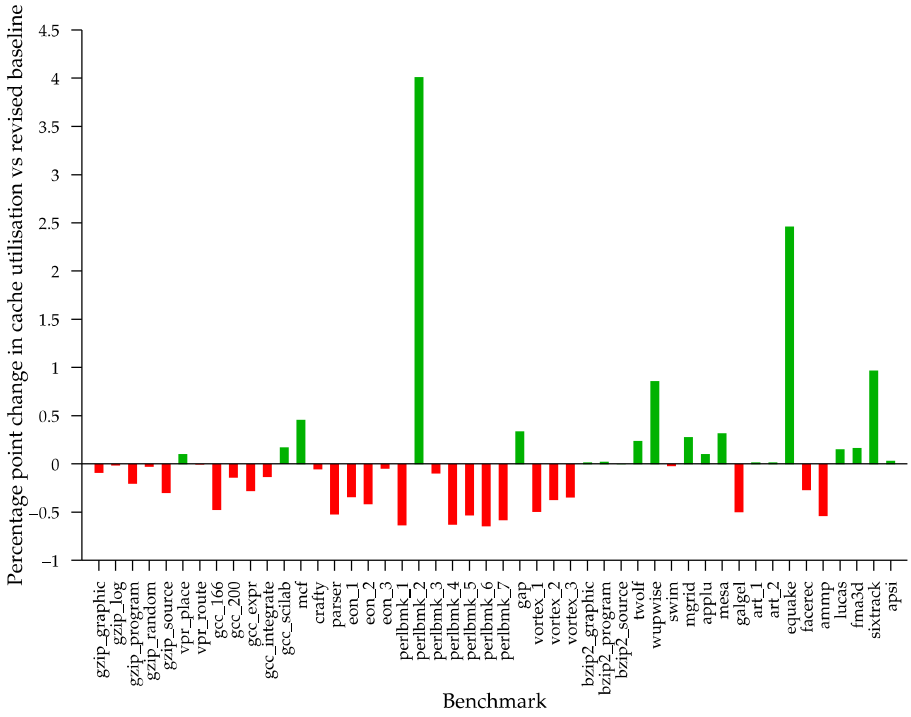

Figure 5.3: DL1 cache utilisation percentage point change for conventional victim cache configuration *vs* revised baseline configuration

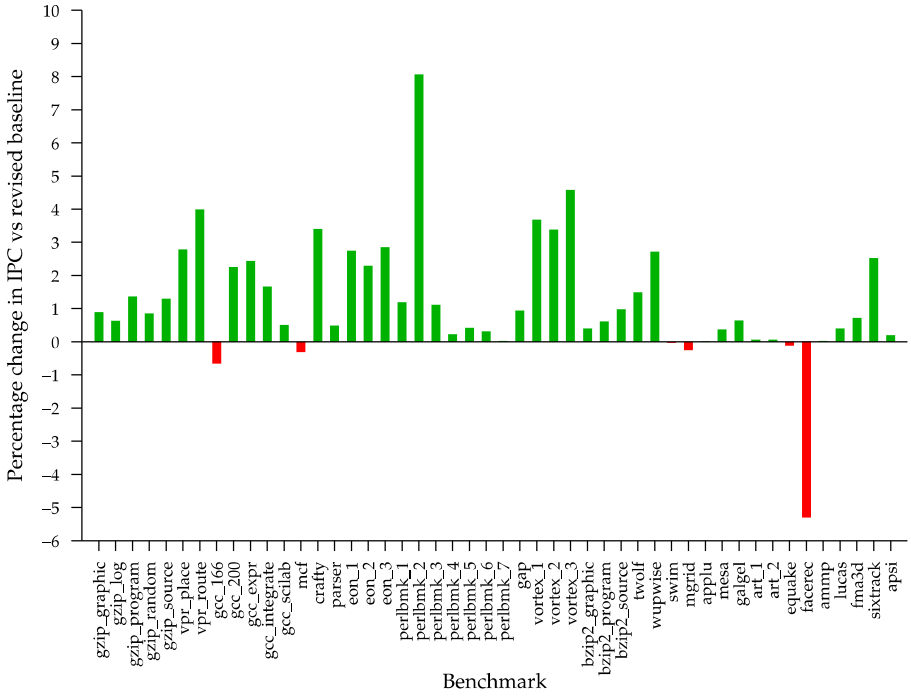

Figure 5.4: IPC change for conventional victim cache configuration *vs* revised baseline configuration

The average IPC improvement is 1.2%, ranging from -5.3% for `facerec` and -0.66% for `gcc_166`, to 4.6% for `vortex_3` and 8.0% for `perlbmk_2`. The small decreases in IPC, also observed by Hu *et al.* [HKM02], are most likely due to second-order effects, including reordering of memory load operations and the fact that the simulation is run for a fixed number of cycles, not instructions, therefore the revised baseline and conventional victim cache configurations will run a slightly different mix of instructions.

5.2.3 Predictor Coverage & Accuracy

The relevant predictors described in Chapter 4 are now evaluated, measuring their coverage and accuracy as previously defined. The filtered victim cache is simulated, together with its associated predictor, however, actual filtering of cache lines into the victim cache is not performed in order to determine the definitive state of each cache line.

Whenever a cache line is placed in the victim cache through the eviction of a cache line from the DL1 cache, a prediction is attempted for that cache line. The prediction is verified when the cache line is either accessed in the victim cache (i.e. it was live at the time of prediction) or it is evicted from the victim cache (i.e. it was dead at the time of prediction).

A prediction *could* be made whenever an access misses in the DL1 cache but hits in the victim cache, resulting in the two cache lines being swapped. However, in such a situation it is unclear what action to take with the result of the prediction. Cache lines which are predicted to be dead could be speculatively written back to the L2 cache, however, as observed by Lee *et al.* [LTF00], this is likely to provide very limited benefit in general-purpose applications.

The various predictors are now evaluated in turn. For this filtered victim cache application, dynamic predictors which require large storage overheads are very unlikely to be as effective as simply enlarging the existing DL1 cache or victim cache. For this reason, only static predictors are considered. Also note that the `MRUlive` predictor is not evaluated for this application since the cache lines evicted from the DL1 cache are *always* the least recently used due to the direct-mapped DL1 cache of the revised baseline configuration.

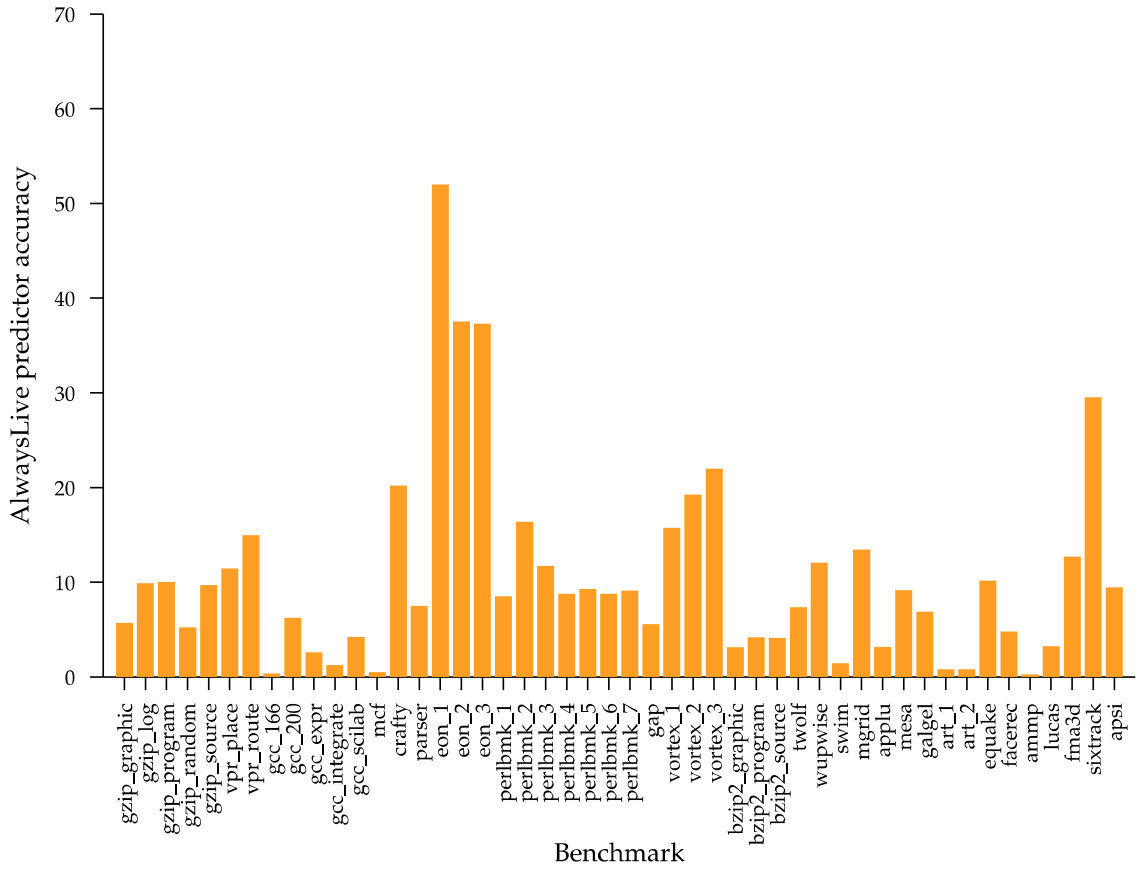

Figure 5.5: `AlwaysLive` predictor accuracy

`AlwaysLive` Predictor

The `AlwaysLive` predictor, included for completeness, predicts that a cache line is always live. As such, it has full coverage since it does not rely upon any history being tracked. It is equivalent to always allocating a victim cache line for every cache line being replaced in the DL1 cache. Figure 5.5 shows the accuracy of the `AlwaysLive` predictor applied to the DL1 victim cache.

In general, the `AlwaysLive` predictor accuracy is low, indicating that allocation of a victim cache line for every cache line being replaced in the DL1 cache is frequently the wrong choice, since that cache line will not be referenced during its time in the victim cache. Predictor accuracy varies considerably between different applications, but is reasonable consistent within applications but with different workloads. The average accuracy is 11%, and varies from 0.27% for `ammp` to 52% for `eon_1`.

`NeverLive` Predictor

The `NeverLive` predictor is the converse of the `AlwaysLive` predictor, and predicts that a cache line is never live, hence should never be allocated space in the victim cache. Again, it has full coverage since it does not rely upon any history being tracked.

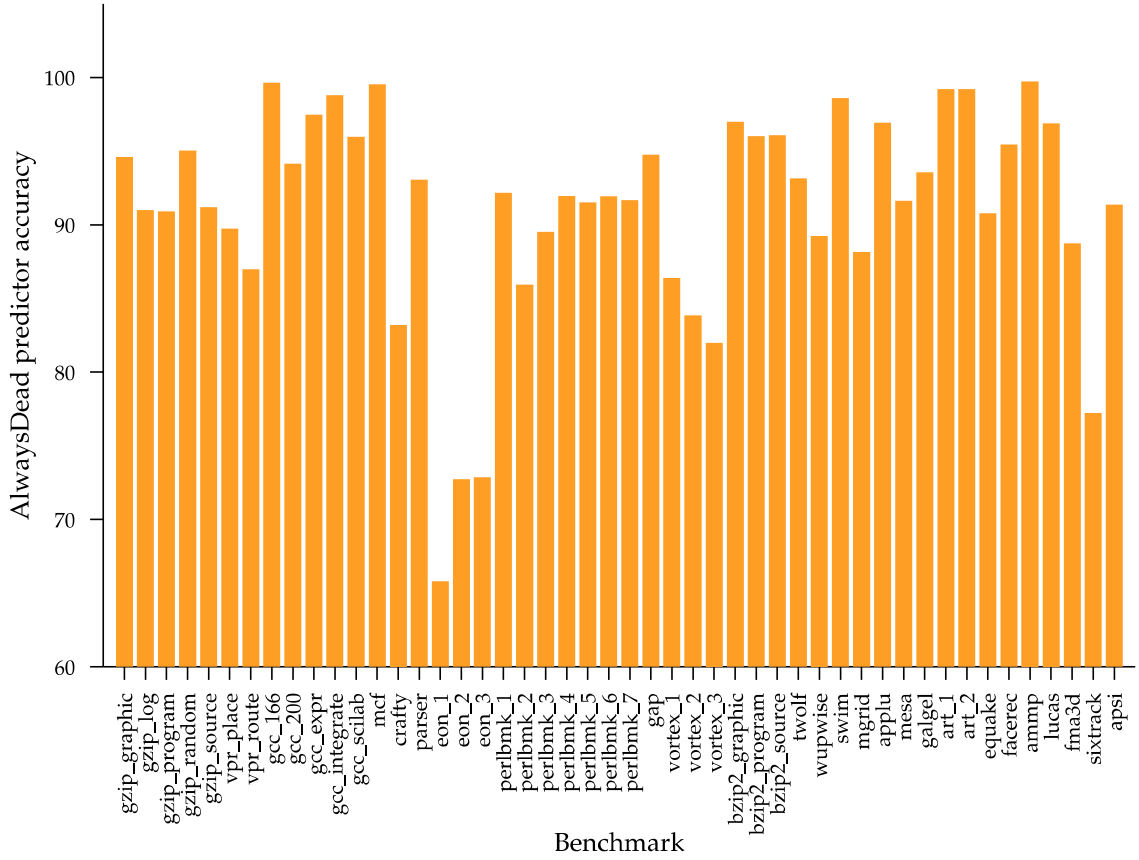

Figure 5.6: `NeverLive` predictor accuracy

Figure 5.6 shows the accuracy for the `NeverLive` predictor applied to the DL1 victim cache. As would be expected, the results are the opposite of those previously obtained for the `AlwaysLive` predictor, confirming the observation that allocation of a victim cache line is frequently the wrong choice and indicating that a filtered victim cache has much potential.

`StackLiveHeapDead` Predictor

The `StackLiveHeapDead` heuristic predictor predicts that a cache line is live if its address falls into the stack segment of the address space, and dead if its address falls into the heap segment of the address space. If its address falls into any other address space, a prediction cannot be made. This approach provides almost complete coverage, since addresses outside the heap and stack segments are typically very rarely encountered in the DL1 cache for the benchmarks considered. Figure 5.7 shows the accuracy of the `StackLiveHeapDead` predictor applied to the DL1 victim cache, the observed coverage being indistinguishable from complete.

Accuracy is fairly high but very variable, with an average of 83%, and ranging from 5% for `gcc_166` to 99.6% for `ammp`. The observed accuracy is lower than that of the `NeverLive` predictor, indicating that the heuristic assumption may not be correct.

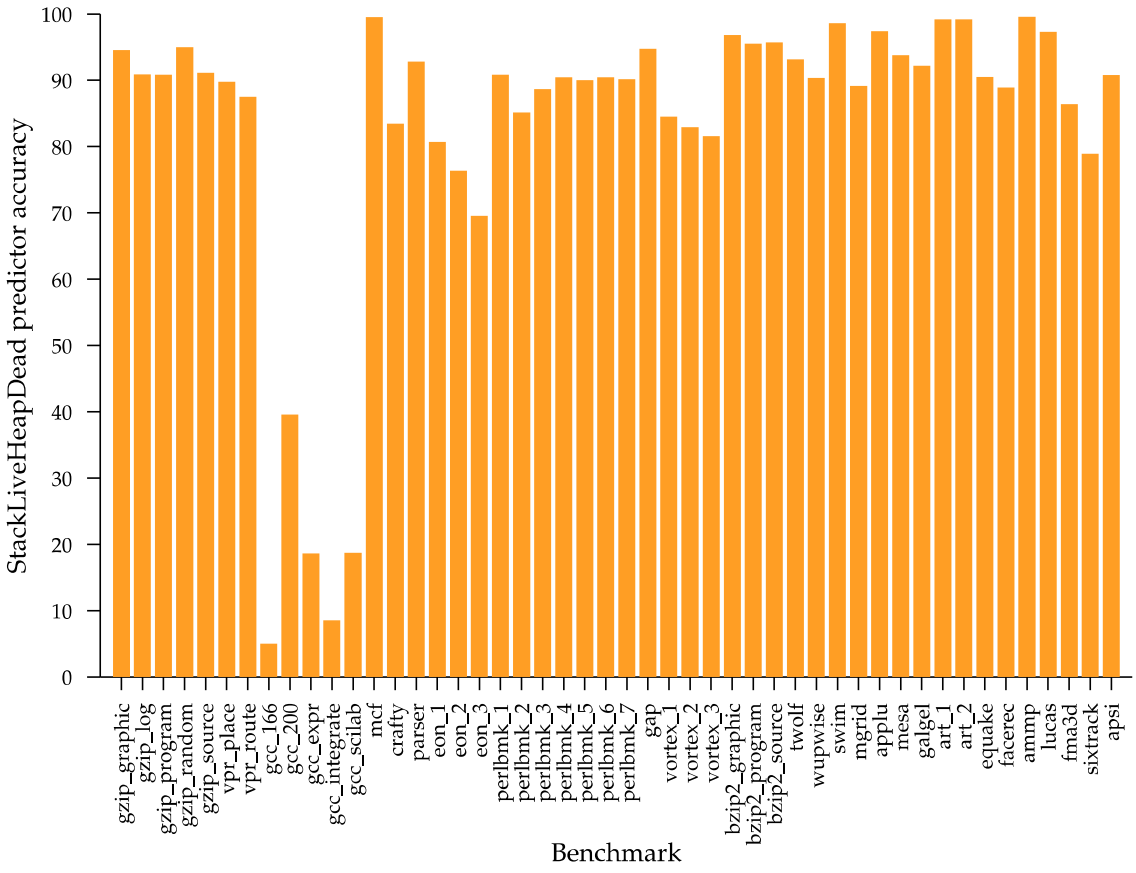

Figure 5.7: `StackLiveHeapDead` predictor accuracy

The validity of the heuristic assumption can be assessed by splitting the predictor into two different components.

The `StackLive` predictor considers the location of the cache line within a program's address space and predicts it to be live if the cache line corresponds to the **stack** segment, otherwise a prediction cannot be made. Figure 5.8 shows the coverage and accuracy of the `StackLive` predictor applied to the DL1 victim cache.

Coverage is generally very low, and varies widely from 0.0014% for `art_1` and `art_2` to 95% for `gcc_166`, with an average of 13%. Likewise, accuracy is generally low, with an average of 35%, varying widely from 0% for `art_1` and `art_2` to 99.8% for `lucas`. The low coverage indicates that most benchmarks make relatively few stack references, while the low accuracy indicates that cache lines in the stack address space are more likely to be dead than live, as is the general overall trend.

The `HeapDead` predictor considers the location of the cache line within a program's address space and predicts it to be dead if the cache line corresponds to the **heap** segment, otherwise a prediction cannot be made. Figure 5.9 shows the coverage and accuracy of the `HeapDead` predictor applied to the DL1 victim cache.

Coverage is mostly very high, over 90%, although is low for the `gcc` and `eon` benchmarks, indicating these benchmarks make few heap references. The average coverage

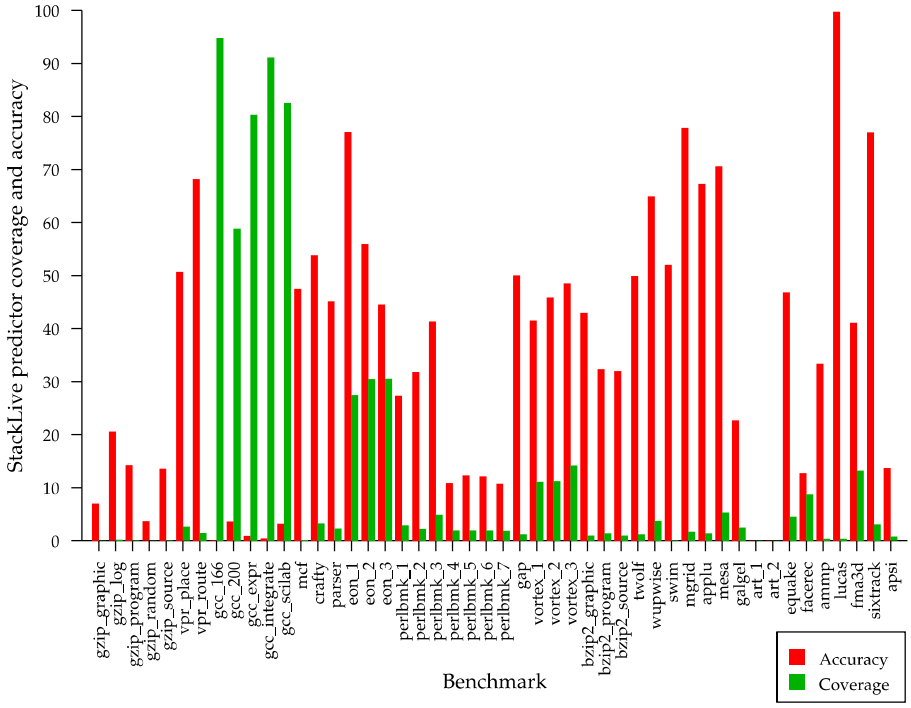

Figure 5.8: `StackLive` predictor coverage and accuracy

Figure 5.9: `HeapDead` predictor coverage and accuracy

is 87%, and ranges from 5.2% for gcc_166 to 100% for art_1 and art_2. Accuracy is consistently high, with an average of 92%, ranging from 79% for sixtrack to 100% for ampp.

For most benchmarks, the number of heap references greatly exceeds the number of stack references, indicated by the coverage results for the individual StackLive and HeapDead predictors. This means that the StackLiveHeapDead predictor very often predicts a cache line to be dead, hence its performance tends towards that of the NeverLive predictor. Indeed, the infrequently occurring but mostly incorrect StackLive predictions mean that the performance is slightly worse than that of the NeverLive predictor.

ThresholdLiveTime Predictor

The ThresholdLiveTime predictor predicts that a cache line is live if the time interval since it was brought into the cache is less than the threshold value, otherwise the cache line is predicted as dead. The particular value for the threshold parameter can be determined by considering the distribution of live and dead cache lines in the victim cache.

Figure 5.10 shows the distribution of live and dead cache lines, entering the victim cache, clustered by their live time when entering the victim cache, for the gzip_graphic benchmark which has been chosen as being representative of the general behaviour. The threshold value must be chosen to maximise the number of dead cache lines correctly predicted as dead (the red area to the right of the threshold value) whilst minimising the number of live cache lines incorrectly predicted as dead (the green area to the right of the threshold). For the filtered victim cache application, incorrectly predicting a live cache line as dead *will* lead to an additional cache miss, whereas incorrectly predicting a dead cache line as live only *may* lead to an additional cache miss, should it evict a live cache line prematurely.

As has already been observed from the AlwaysLive and NeverLive predictors, the vast majority of cache lines in the victim cache turn out to be dead. Detailed examination of the data presented in Figure 5.10 indicates that a threshold of 512 cycles is most appropriate.

DualThresholdLiveTime Predictor

Further careful examination of Figure 5.10 indicates that the behaviour of live cache lines in the victim cache exhibits a bimodal distribution, with significant numbers of live cache lines having both very small and very large live times, but very few in between. This behaviour cannot be captured with the ThresholdLiveTime predictor previously proposed, since only a single threshold is available for consideration. Instead, a DualThresholdLiveTime predictor is evaluated, which predicts that a cache line is live if its live time is less than the lower threshold *or* more than the upper threshold, otherwise the cache line is predicted as dead. From Figure 5.10, the lower threshold remains at 512 cycles, while the upper threshold is at 100,000 cycles.

Figure 5.11 compares the accuracy of the ThresholdLiveTime and DualThresholdLiveTime predictors for the revised baseline configuration. The ThresholdLive-

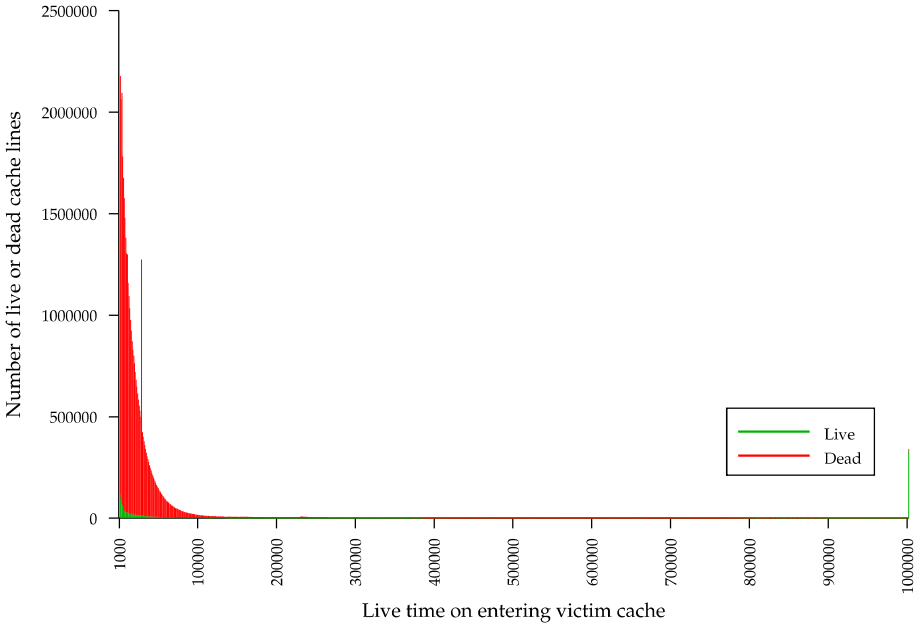

Figure 5.10: Distribution of live and dead cache lines in the victim cache clustered by live time for `gzip_graphic`

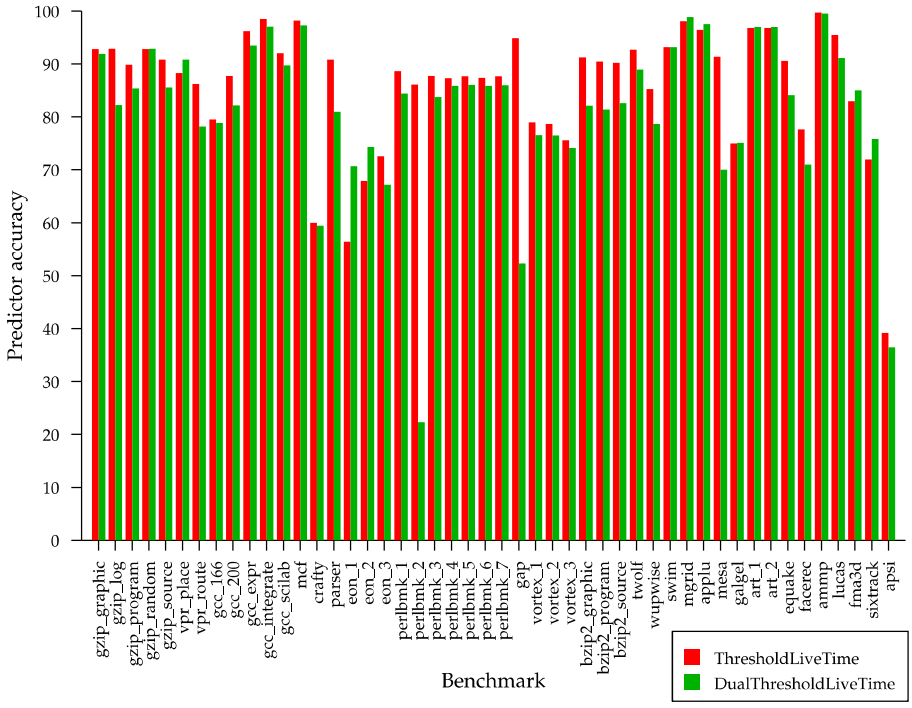

Figure 5.11: Accuracy of the `ThresholdLiveTime` and `DualThresholdLiveTime` predictors

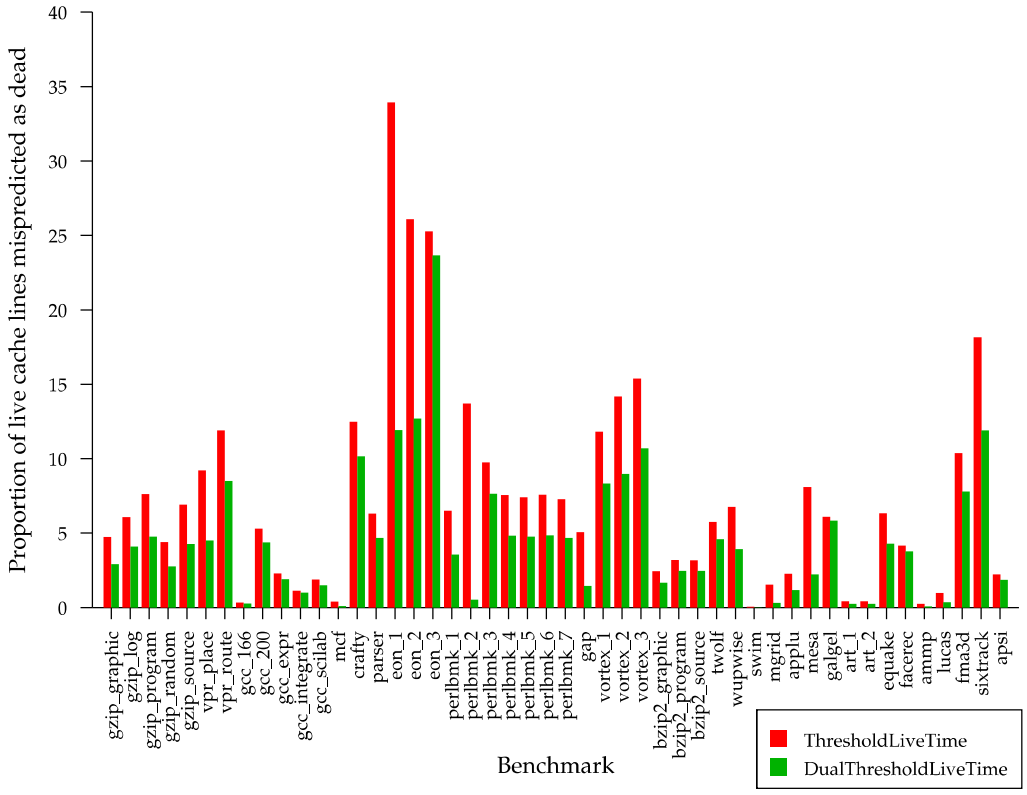

Figure 5.12: Proportion of live cache lines mispredicted as dead by the `ThresholdLiveTime` and `DualThresholdLiveTime` predictors

`Time` predictor and `DualThresholdLiveTime` predictor accuracies are usually similar and fairly high, with an average of 86% and 81% respectively. The accuracy of the `ThresholdLiveTime` predictor usually exceeds that of the `DualThresholdLiveTime` predictor, but for a few benchmarks the converse is observed. In two cases, `perlbmk_2` and `gap`, the accuracy of the `DualThresholdLiveTime` predictor is considerably lower than that of the `ThresholdLiveTime` predictor.

As has already been mentioned, mispredicting a live cache line as dead will result in an additional cache miss. Figure 5.12 compares the proportion of such occurrences for the revised baseline configuration with the `ThresholdLiveTime` and `DualThresholdLiveTime` predictors. While the overall accuracy of the `DualThresholdLiveTime` predictor is lower than that of the `ThresholdLiveTime` predictor, the `DualThresholdLiveTime` predictor makes far fewer expensive mispredictions of live cache lines as dead, with an average of 4.6% compared to 7.4%.

`ThresholdAccessInterval` Predictor

The `ThresholdAccessInterval` predictor predicts that a cache line is live if the time interval since it was last referenced is less than the threshold value, otherwise the cache line is predicted as dead. The particular value for the threshold parameter can be determined by considering the distribution of live and dead cache lines in the victim

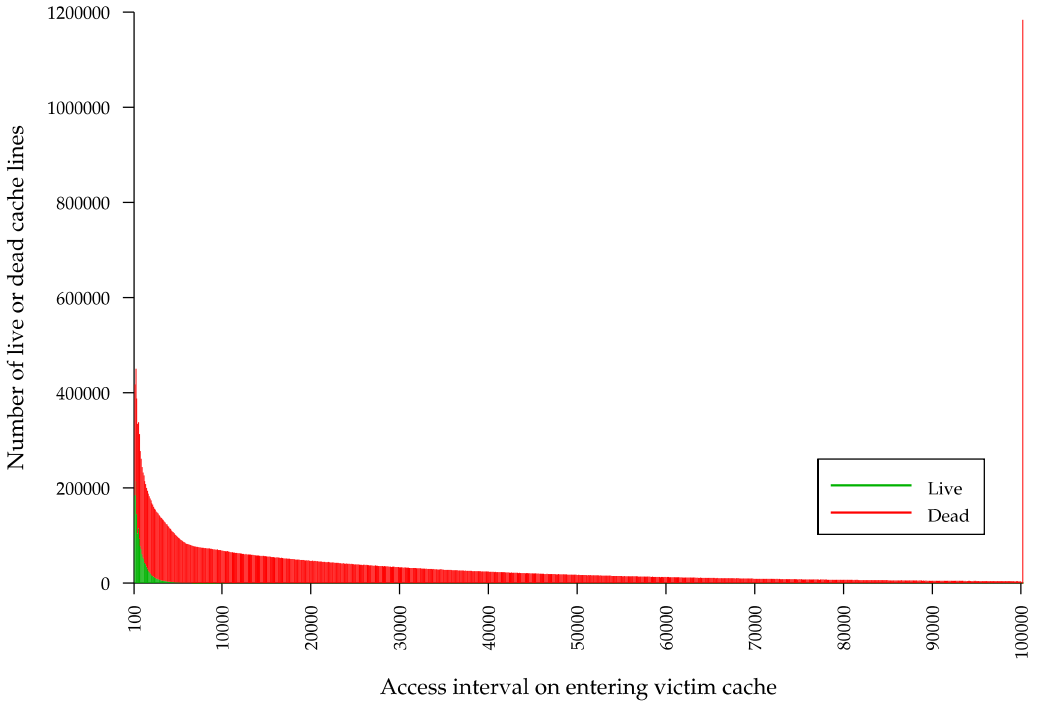

Figure 5.13: Distribution of live and dead cache lines in the victim cache clustered by access interval for `twolf`

cache.

Figure 5.13 shows the distribution of live and dead cache lines, entering the victim cache, clustered by their access interval when entering the victim cache, for the `twolf` benchmark which has been chosen as being representative of the general behaviour. As with the `ThresholdLiveTime` predictor, the threshold value must be chosen to maximise the number of dead cache lines correctly predicted as dead (the red area to the right of the threshold value) whilst minimising the number of live cache lines incorrectly predicted as dead (the green area to the right of the threshold). Compared to Figure 5.10, live and dead cache lines seem more separable using access intervals rather than live times. In this case, the threshold value is chosen as 32 cycles.

Unlike the `ThresholdLiveTime` predictor, the live cache lines do not exhibit a bi-modal distribution, hence the `ThresholdAccessInterval` predictor is adequate. Figure 5.14 shows the accuracy of the `ThresholdAccessInterval` predictor for the revised baseline configuration for each benchmark. Accuracy is generally high for most benchmarks, with an average of 92% and ranging from 73% for `eon_1` to 99.8% for `ammp`. Accuracy of the `ThresholdAccessInterval` predictor is considerably higher than that of the `ThresholdLiveTime` or `DualThresholdLiveTime` predictors.

Figure 5.15 shows the proportion of live cache lines mispredicted as dead by the `ThresholdAccessInterval` predictor. The average proportion of live cache lines mispredicted as dead is 6.7%, and ranges from 0.21% for `swim` to 27% for `eon_1`. This is considerably higher than the average of 4.6% observed for the `DualThresholdLiveTime` predictor, but less than the average of 7.4% observed for the `ThresholdLive-`

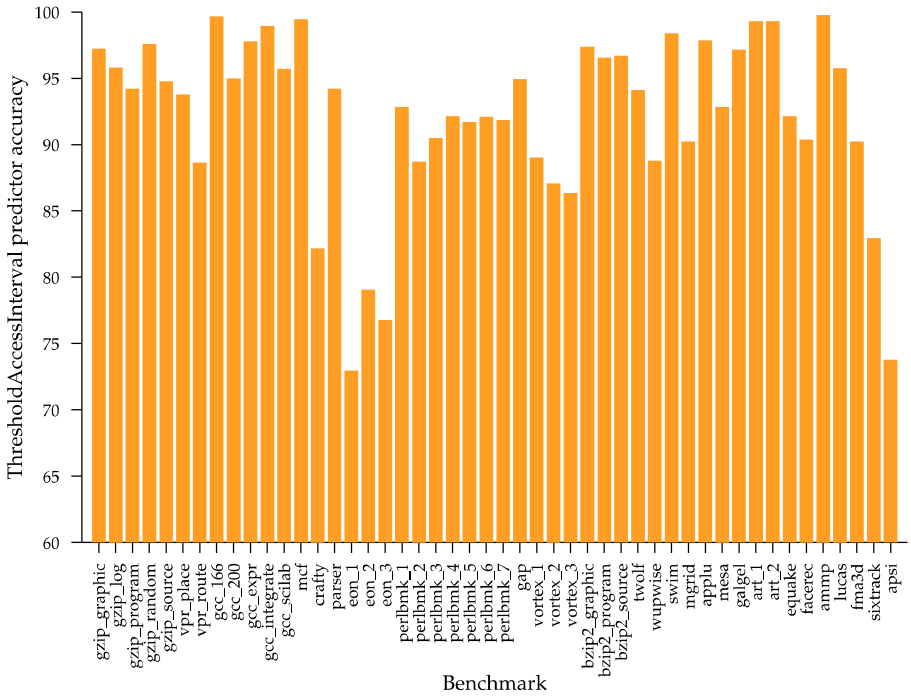

Figure 5.14: Accuracy of the `ThresholdAccessInterval` predictor

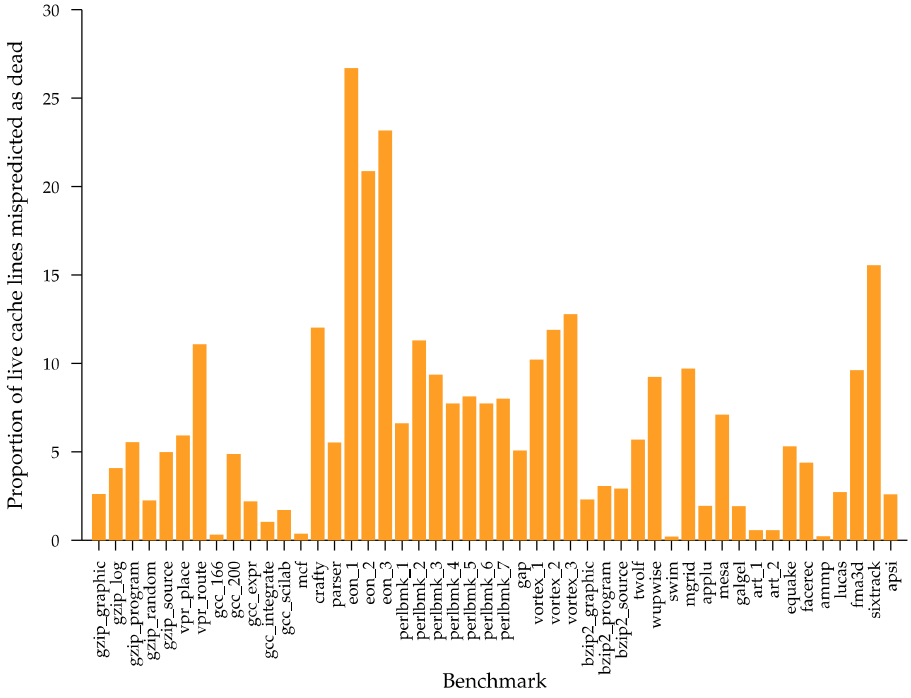

Figure 5.15: Proportion of live cache lines mispredicted as dead by the `ThresholdAccessInterval` predictor

`Time` predictor.

5.2.4 Overall Performance

The most promising predictors evaluated above are the `ThresholdLiveTime`, `DualThresholdLiveTime` and `ThresholdAccessInterval` predictors. These three predictors are now evaluated in terms of the change in DL1 cache miss rate and the change in IPC. Due to the limited effect of the victim cache on overall cache utilisation, this metric is not examined.

Miss Rate

Figure 5.16 shows the percentage change in the number of misses per thousand instructions (MPKI) for a filtered victim cache compared to the conventional victim cache. In almost all cases, all three predictors substantially reduce the miss rate, by an average of 23%, 24% and 21% respectively. Their performance is usually similar, and where it differs, the novel `DualThresholdLiveTime` predictor typically outperforms the novel `ThresholdLiveTime` predictor and the `ThresholdAccessInterval` predictor previously proposed by Hu *et al.* [HKM02]. In addition, the novel predictors do not require updating on every cache hit, hence are potentially more power-efficient. In one case, `perlbmk_1`, the cache miss rate increases substantially for all three predictors.

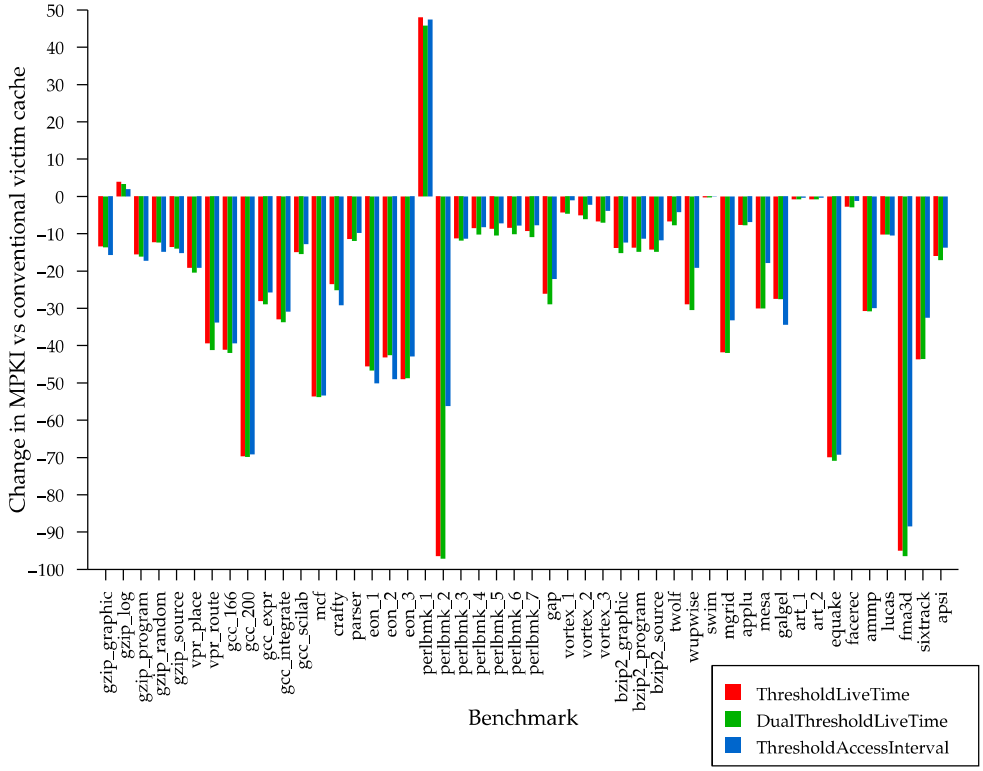

Figure 5.16: Percentage change in MPKI *vs* conventional victim cache

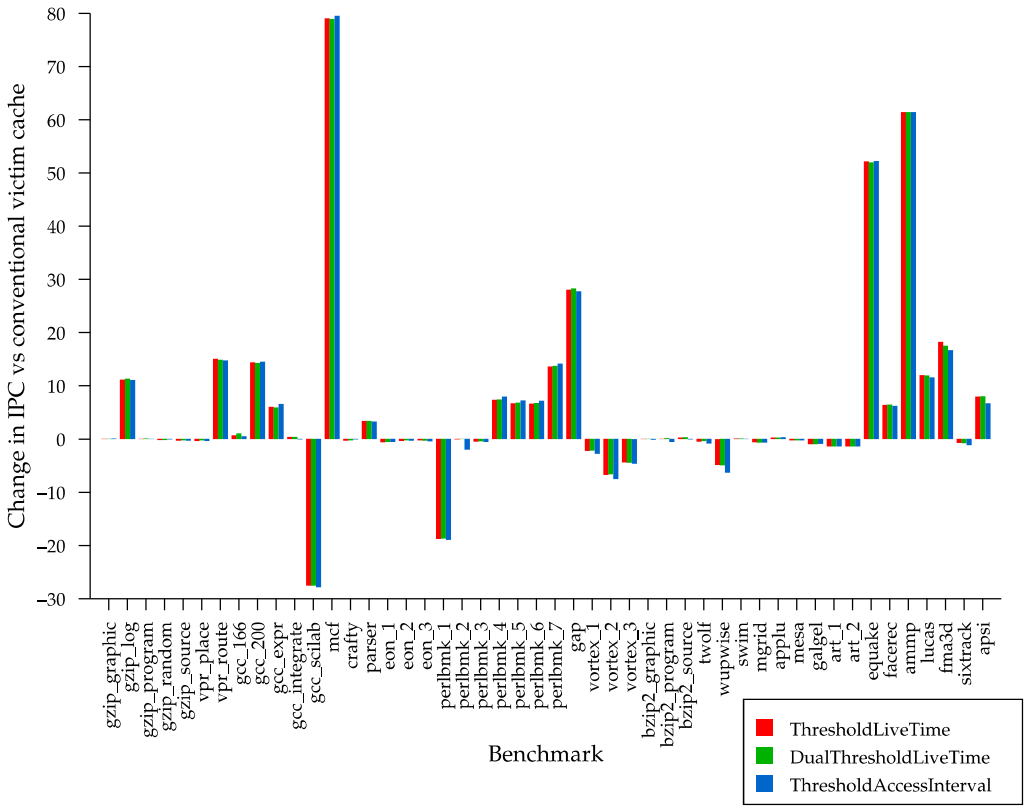

Figure 5.17: Percentage change in IPC *vs* conventional victim cache

Instructions Committed per Cycle

Figure 5.17 shows the percentage change in the number of Instructions committed Per Cycle (IPC) for a filtered victim cache compared to the conventional victim cache. In most cases, the IPC increases similarly for each of the three predictors. The average change in IPC is 5.6%, 5.6% and 5.8% for the `ThresholdLiveTime`, `DualThresholdLiveTime` and `ThresholdAccessInterval` predictors respectively. The average change in IPC between the baseline system configuration and the conventional victim cache is just 1.2%, hence the filtered victim cache improves the average performance significantly with little hardware overhead.

For `perlbmk_1` which saw a substantial rise in miss rate, the IPC is correspondingly lowered by approximately 20%. One other benchmark, `gcc_scilab`, shows a similar decrease in IPC but no corresponding increase in miss rate. This warrants further investigation, and may be due to the fact that not all memory operations are of the same **criticality**, as found by Srinivasan *et al.* [SJLW01].

Storage Overhead

The three predictors evaluated above each require a timestamp to be stored for each cache line, corresponding to the time at which the cache line was brought into the cache (for the `ThresholdLiveTime` and `DualThresholdLiveTime` predictors), or

the time at which the cache line was last accessed (for the `ThresholdAccessInterval` predictor). This timestamp is stored with per-cycle precision using a 64-bit value. Therefore the additional storage overhead is 4 kB, which could equally well be used to increase the size of the DL1 cache, or increase the size of the victim cache. However, a much coarser timestamp would probably suffice, as used by Hu *et al.* [HKM02], leading to negligible storage overhead.

5.3 Prefetching Victim Selection

Section 2.6.2 described the concept of prefetching, proactively fetching cache lines from the next level in the memory hierarchy before a corresponding demand fetch instruction is encountered. Prefetching is most effective when there are a large number of compulsory misses; indeed, it is the only common cache improvement technique which can reduce the number of compulsory misses. However, prefetching can potentially harm overall system performance in two respects. First, prefetch accesses compete with demand fetch accesses for bus bandwidth. Second, and more significant, prefetch accesses compete with demand fetch accesses for cache space. It is this second aspect which is addressed here.

Previous work has combined prediction of dead cache lines with a prefetcher to select **prefetch targets** which can replace predicted dead cache lines, known as **prefetch victims**, for example in Lai *et al.*'s Dead Block Correlating Prefetcher (DBCP) [LFF01]. An alternative approach is pursued here, in which prefetch target selection and prefetch victim selection are decoupled. An aggressive conventional prefetcher issues prefetches, providing the prefetch targets. When a prefetch request is satisfied, it is placed in the cache set in the least-recently used position which is predicted to contain a dead cache line. If no such position exists, the prefetched cache line is discarded.

Many prefetching algorithms of varying complexity have been proposed. In this dissertation, **tagged prefetching** is used since it is simple to implement and can provide significant performance gains, but also significant performance losses. Tagged prefetch issues a prefetch for cache lines $i+1$, $i+2$...$i+n$ whenever cache line i is demand-fetched *or* upon the first reference to the previously prefetched cache line i. The parameter n is known as the **degree** of prefetching, and determines how many cache lines are prefetched at a time.

5.3.1 Method

As previously, results are presented from simulations over the entire SPEC CPU2000 integer and floating-point benchmark suite. The benchmarks are executed using the reference set of inputs, with the first one billion cycles executed using a simplified processor model to warm up the caches and to bypass any initialisation code before running the next *one* billion cycles with a detailed processor model for predictor evaluation, otherwise *two* billion cycles are used. Previous simulations in this dissertation have run the detailed processor model for two billion cycles, but this is reduced in order to reduce the overall simulation time with numerous different predictors.

Prefetching is generally of most benefit if the latency to access the lower level of memory is high, and there is adequate capacity to store prefetched cache lines without excessive **cache pollution**. For this reason, prefetching is evaluated in the L2 cache of the baseline system configuration.

The benefits provided by conventional tagged prefetching are first investigated. The accuracy and coverage of various predictors are then evaluated, and full results for the best performing predictors are evaluated against relevant previous research.

The `MRULive` predictor is not evaluated since it would require significant changes to the replacement policy of the cache to avoid thrashing when inserting prefetched cache lines. The question of where to place both demand-fetched and prefetched cache lines in the least-recently used (LRU) stack is still open, and an additional area where liveness predictors may be applicable [QJP+07].

5.3.2 Conventional Tagged Prefetching

Conventional tagged prefetching is now evaluated using the miss rate, cache utilisation and instructions committed per cycle metrics. The degree of prefetching is chosen to be four, which is a fairly aggressive value. Candidate prefetches are allowed to check the DL1 and IL1 cache before being issued, and the latency of calculating prefetch addresses is just a single cycle. The prefetcher is invoked on every cache access. Up to 100 prefetches may be queued at once.

Miss Rate

Figure 5.18 shows the percentage change in the L2 cache miss rate, measured as the number of misses per thousand instructions (MPKI), for the conventional tagged prefetch configuration. For the majority of benchmarks, a significant reduction is observed, up to -94% for `gcc_integrate`, with an average of -31%. However, a few benchmarks show a significant increase in miss rate, up to 143% for `perlbmk_3`.

As with the conventional victim cache, some care must be taken when interpreting these figures, and the absolute miss rate must be considered as well as the percentage change. For example, while `perlbmk_3` shows the largest percentage increase in miss rate, the miss rate both in the baseline configuration and with the conventional tagged prefetcher is actually very small.

Cache Utilisation

Figure 5.19 shows the percentage point change in cache utilisation for the conventional tagged prefetcher compared to the baseline configuration. The general trend is that the conventional tagged prefetcher decreases cache utilisation, by a maximum of -44% for `fma3d` and an average of -6%. In some cases, cache utilisation increases slightly, by up to 6% for `gcc_scilab`. The interpretation of these results is that in most cases the prefetcher is bringing dead lines into the cache, as well as prematurely evicting live cache lines. The decoupled prefetch victim selection scheme investigated here can help with the latter issue, but not with the former since that largely relies upon a better prefetching algorithm.

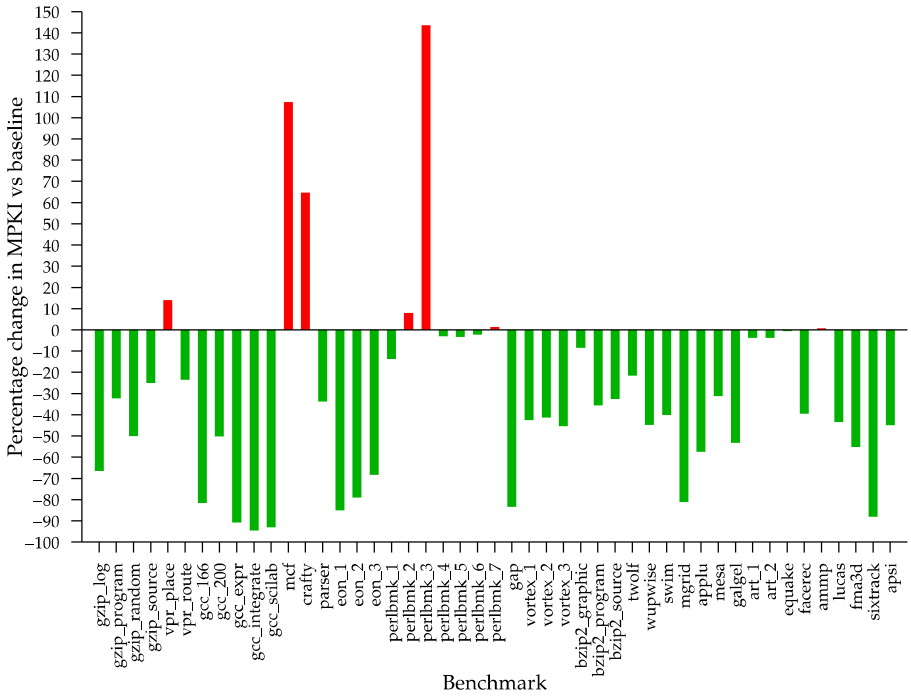

Figure 5.18: Percentage change in L2 cache miss rate *vs* baseline

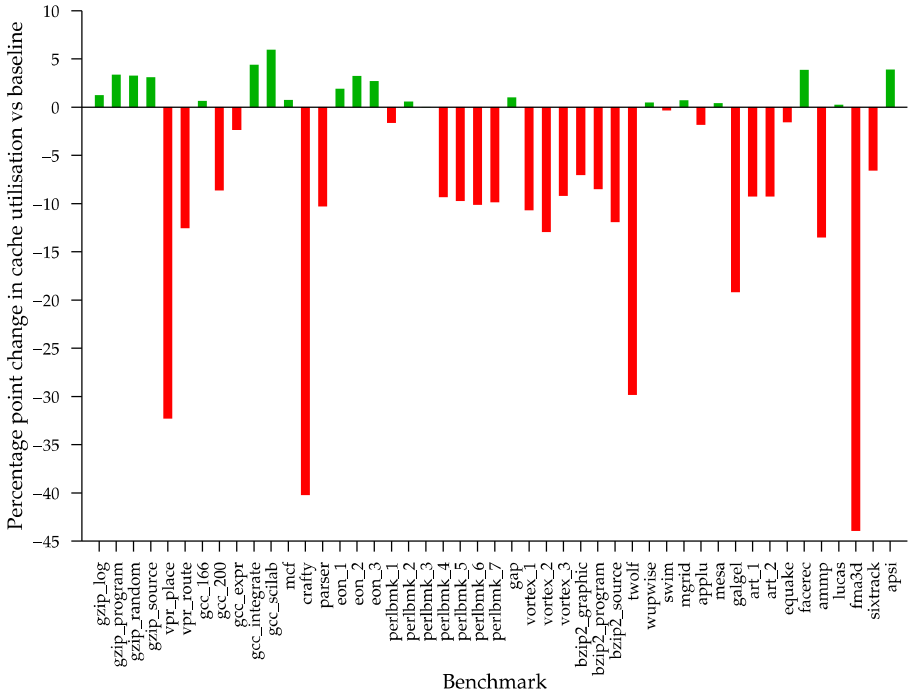

Figure 5.19: Percentage point change in L2 cache utilisation *vs* baseline

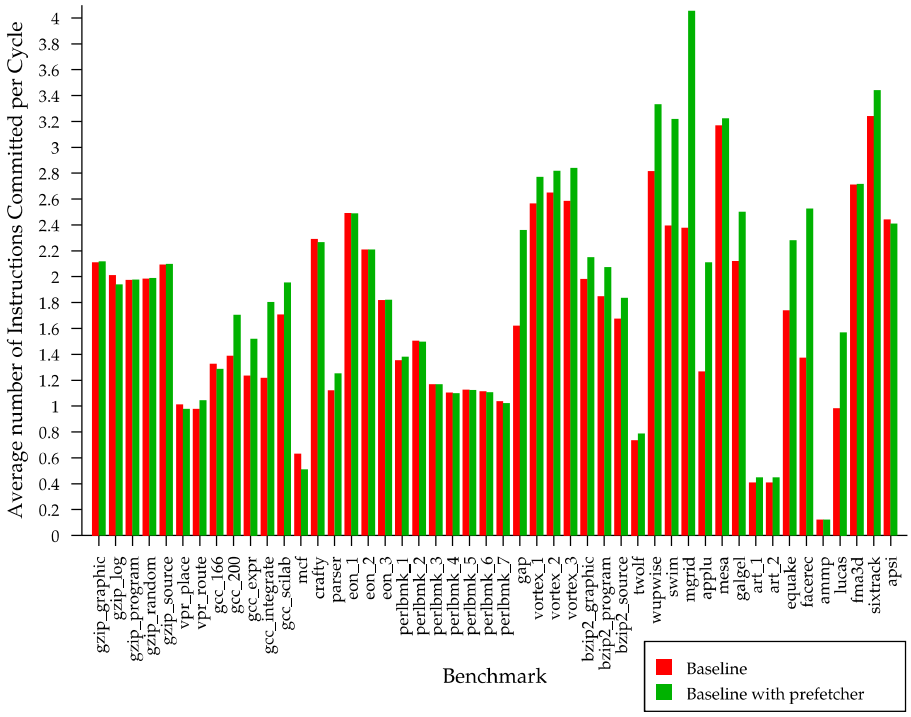

Figure 5.20: IPC for conventional tagged prefetcher compared to baseline configuration

Instructions Committed per Cycle

Figure 5.20 shows the average number of Instructions committed Per Cycle (IPC) for the conventional tagged prefetcher compared to the baseline configuration. In most cases, the conventional tagged prefetcher increases IPC by up to 84% for `facerec`, with an average of 13%. For `perlbmk_3` which showed a significant percentage increase in the cache miss rate, no change in IPC is observed since the absolute change in cache miss rate was negligible. Several benchmarks show a small decrease in IPC, up to -20% for `mcf`. Those benchmarks which most benefit from the conventional prefetcher tend to be the floating-point benchmarks on the right-hand side of Figure 5.20, with typically more regular access patterns than the integer benchmarks on the left-hand side.

5.3.3 Static Liveness Predictors

`NeverLive` Predictor

Figure 5.21 shows the accuracy of the `NeverLive` predictor applied to prefetch victim selection in the baseline prefetching configuration. Overall accuracy is generally high, indicating that the least-recently used cache line in each set of the L2 cache is frequently dead. The average accuracy is 95%, and varies from 64% for `gcc_166` to 100% for the `eon` benchmarks and `fma3d`. However, for these latter benchmarks, exceedingly few prefetched cache lines are either referenced or evicted during execution of the benchmark, and are omitted from subsequent results.

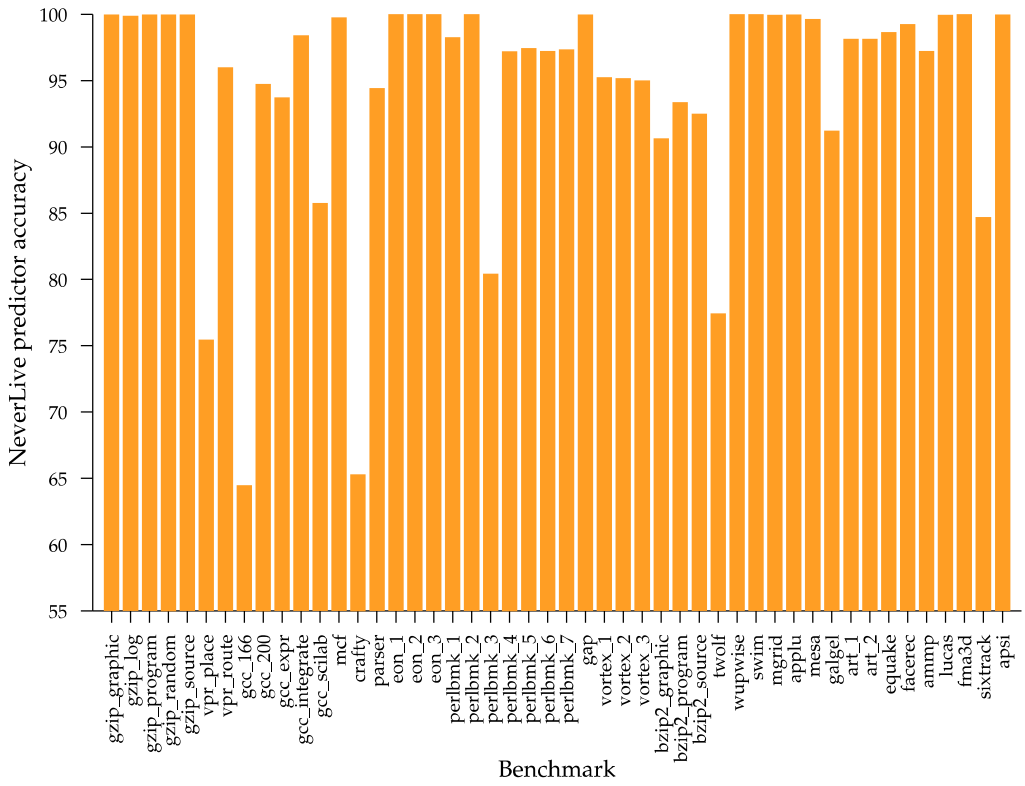

Figure 5.21: `NeverLive` predictor accuracy

Other Static Liveness Predictors

The `StackLiveHeapDead`, `ThresholdLiveTime` and `ThresholdAccessInterval` predictors previously evaluated for the filtered victim cache application were also evaluated for selection of prefetch victims. The results, not presented here for reasons of space, were generally poor, with low accuracy over most benchmarks. This is most likely due to the inability to adjust the threshold for a particular benchmark or phase of a benchmark. As has already been seen in Chapter 3, results for the L2 cache are more variable than those for the DL1 cache, hence the remainder of this section concentrates on dynamic predictors.

5.3.4 Dynamic Direct Liveness Predictors

The dynamic direct liveness predictors evaluated in this section are also known as **Last-Touch Predictors** (LTPs) or **Dead Block Predictors** (DBPs). Section 4.6.3 described how various elements from historical addresses and program counter values could be combined into a **signature**. A **last-touch signature table** contains signatures which have previously been observed to have been last-touches. In this section, an 8-way set-associative last-touch signature table is used, with a total of 16,384 sets. The tag for each signature is stored as a 49-bit value, and the four least significant bits are ignored during tag comparison in order to promote constructive interference. In total,

approximately 1 MB of signature storage is provided. The access time of the last-touch signature table, estimated using the CACTI 4.2 cache modelling tool [TTJ06] assuming a 45 nm fabrication process, is 10 cycles.

`LastTouch_1Addr` Predictor

Figure 5.22 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LastTouch_1Addr` predictor. As with victim cache filtering, mispredicting a live cache line as dead has a much larger potential negative impact than the converse, since a mispredicted dead cache line *will* cause an additional cache miss, whereas a mispredicted live cache line *may* cause an additional cache miss if an access is made to the prefetch target.

The average accuracy is 65%, but varies considerably between benchmarks, from 24% to 93%. Coverage generally tracks accuracy, with an average of 68% and ranging from 23% to 99%. The proportion of mispredicted dead prefetch victims is low for most benchmarks, with an average of 4.2%, but is considerable for several benchmarks with a maximum of 33%.

`LastTouch_2SAddr` Predictor

Figure 5.23 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LastTouch_2SAddr` predictor. The average accuracy is 55%, but varies considerably between benchmarks, from 10% to 95%. Again, coverage generally tracks accuracy, with an average of 54% and ranging from 6.9% to 95%.

Again, the proportion of mispredicted dead prefetch victims is low for most benchmarks, with an average of 2.4% and a maximum of 21%, less than the values for the `LastTouch_1Addr` predictor.

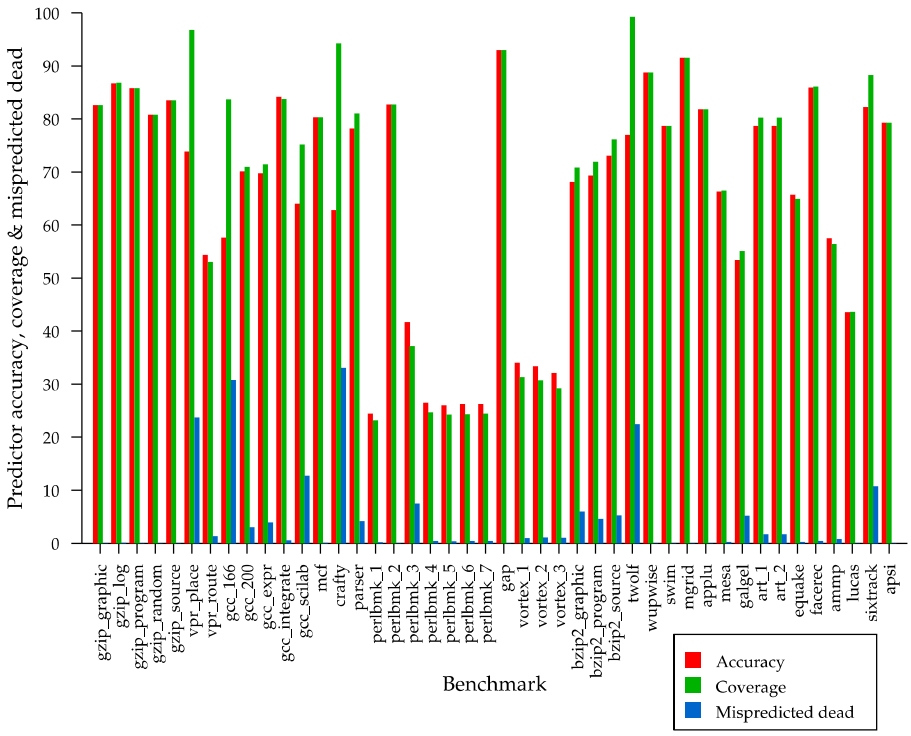

Figure 5.22: `LastTouch_1Addr` predictor performance

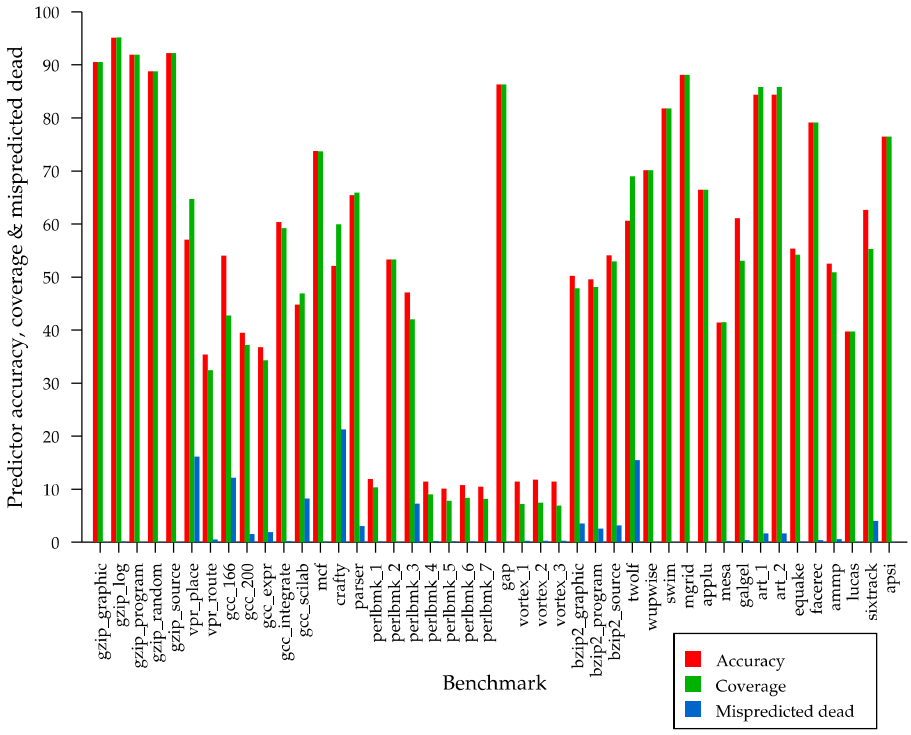

Figure 5.23: `LastTouch_2SAddr` predictor performance

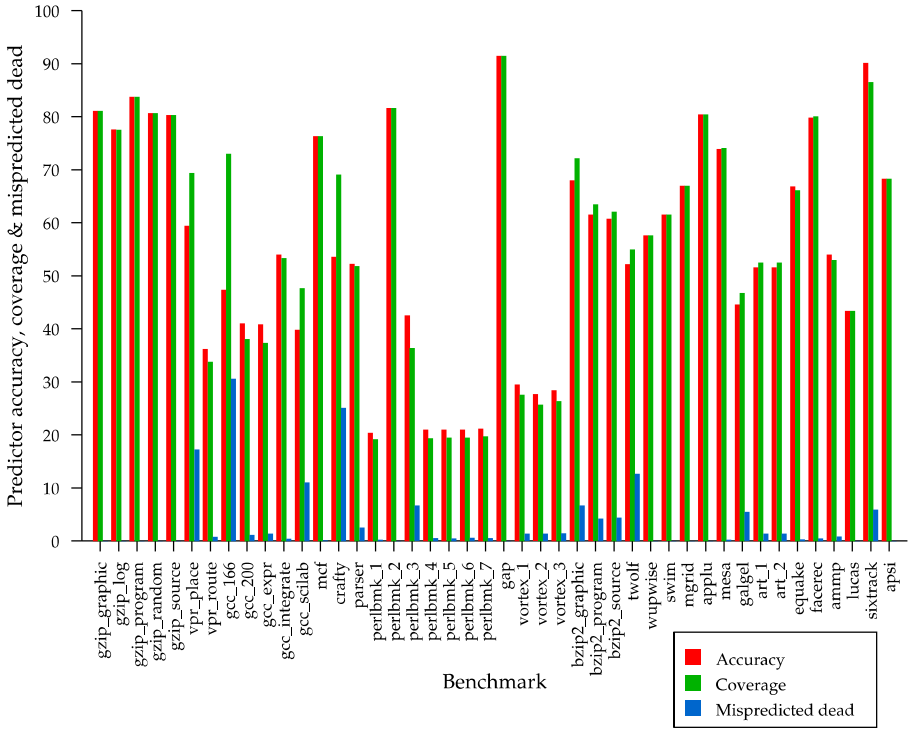

Figure 5.24: `LastTouch_1Addr3APC` predictor performance

`LastTouch_1Addr3APC` Predictor

Figure 5.24 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LastTouch_1Addr3APC` predictor. The average accuracy is 56%, but varies considerably between benchmarks, from 20% to 91%. Again, coverage mostly closely tracks accuracy, with an average of 56% and ranging from 19% to 91%. Again, the proportion of mispredicted dead prefetch victims is low for most benchmarks, with an average of 3.4% and a maximum of 30%.

`LastTouch_1PC2Addr` Predictor

Figure 5.25 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LastTouch_1PC2Addr` predictor. Accuracy is high for several benchmarks, low for several others, and varies considerably for the remainder. The average accuracy is 43%, ranging from 5% to 85%. Again, coverage tracks accuracy with an average of 42%, ranging from 5.4% to 85%. Again, the proportion of mispredicted dead prefetch victims is usually low, with an average of 2.2% and a maximum of 19%.

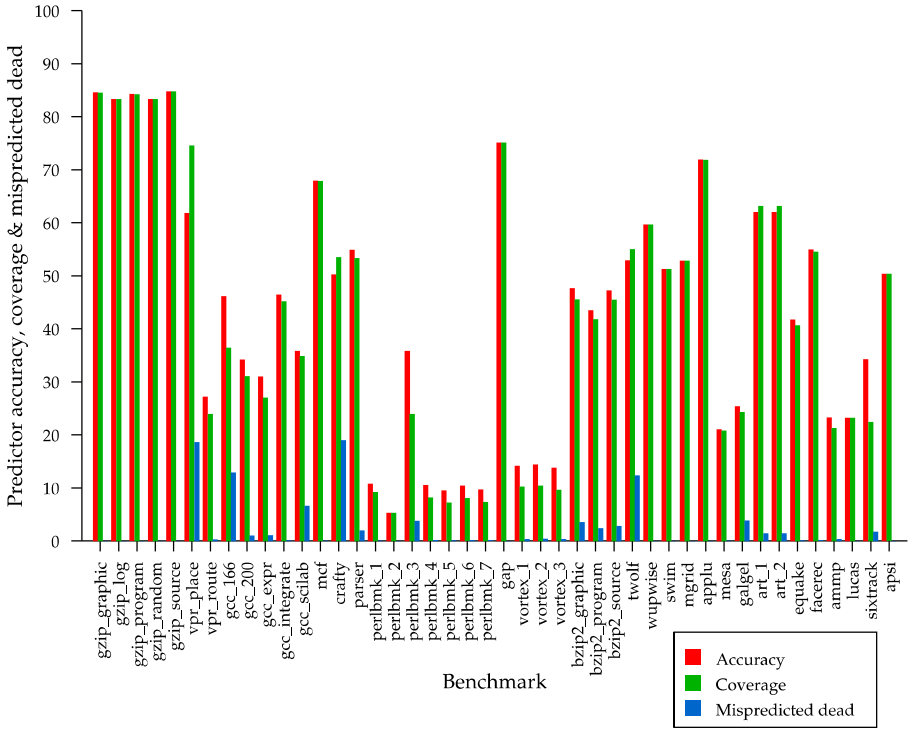

Figure 5.25: `LastTouch_1PC2Addr` predictor performance

`LastTouch_1PC` Predictor

Figure 5.26 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LastTouch_1PC` predictor. Accuracy is very high for most benchmarks, with an average of 94%, ranging from 65% to 99%. Coverage is also very high, with an average on 99.7% and ranging from 98.0% to 99.9%. The proportion of mispredicted dead prefetch victims is usually very low, with an average of 5.4%, but significant for a few benchmarks with a maximum of 35%.

Summary

Overall, the `LastTouch_1PC` predictor performs best over the set of benchmarks evaluated. It combines high accuracy and coverage with relatively few mispredicted dead prefetch victims.

5.3.5 Dynamic Live Time Predictors

The dynamic live time predictor extends the previously evaluated dynamic direct liveness predictor by storing the previously observed live time with each corresponding entry in the signature table. The current live time is predicted to be double that previously observed, the same heuristic used by Hu *et al.* [HKM02]. Live times are stored

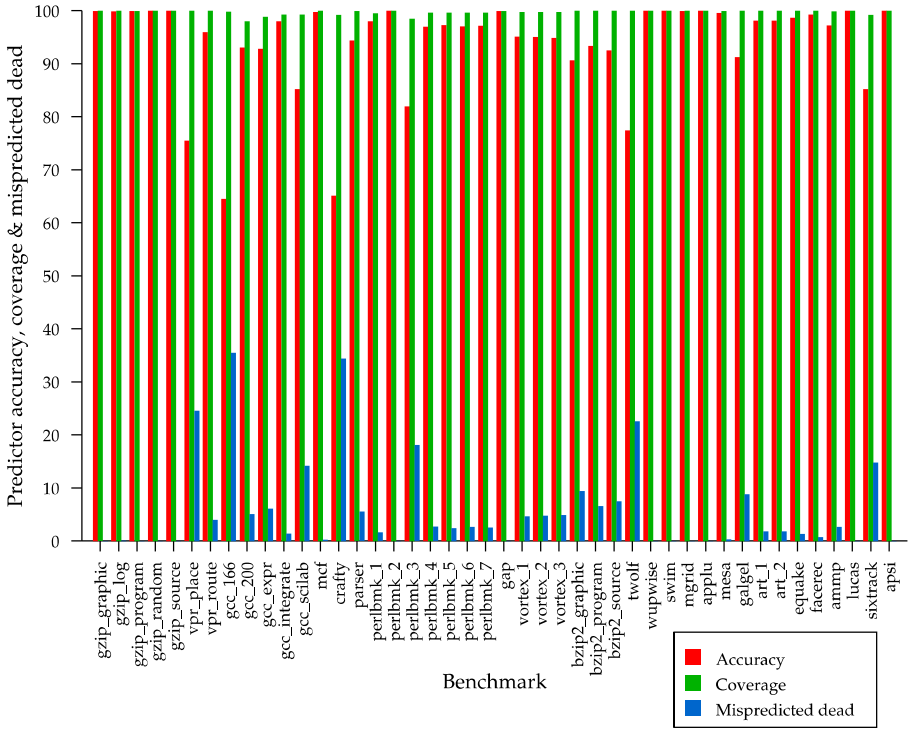

Figure 5.26: `LastTouch_1PC` predictor performance

at a resolution of a single cycle — a much coarser resolution is probably acceptable, but remains an area for future work. The same signature table size, organisation and latency is used for the dynamic live time predictors as for the dynamic direct liveness predictors. If a matching signature cannot be found in the signature table, the cache line is predicted as dead. This is based upon the earlier observation that most cache lines which are candidate prefetch victims are dead, but is the converse of the behaviour of the dynamic direct liveness predictor.

`LiveTime_1Addr` Predictor

Figure 5.27 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LiveTime_1Addr` predictor. Accuracy is generally moderate but variable, with an average of 64% and ranging from 20% to 93%. Coverage mostly tracks accuracy, but is significantly different in a few cases. The average coverage is 67%, ranging from 20% to 98%. The proportion of mispredicted dead prefetch victims is usually very low, with an average of 4.4%, but significant for a few benchmarks with a maximum of 26%.

`LiveTime_2SAddr` Predictor

Figure 5.28 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as

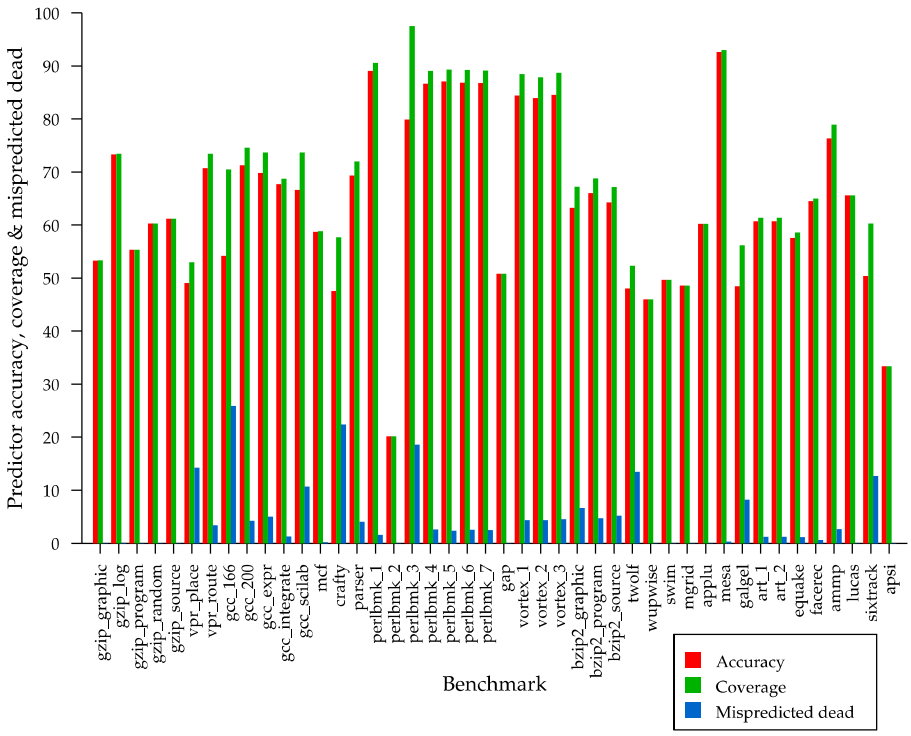

Figure 5.27: `LiveTime_1Addr` predictor performance

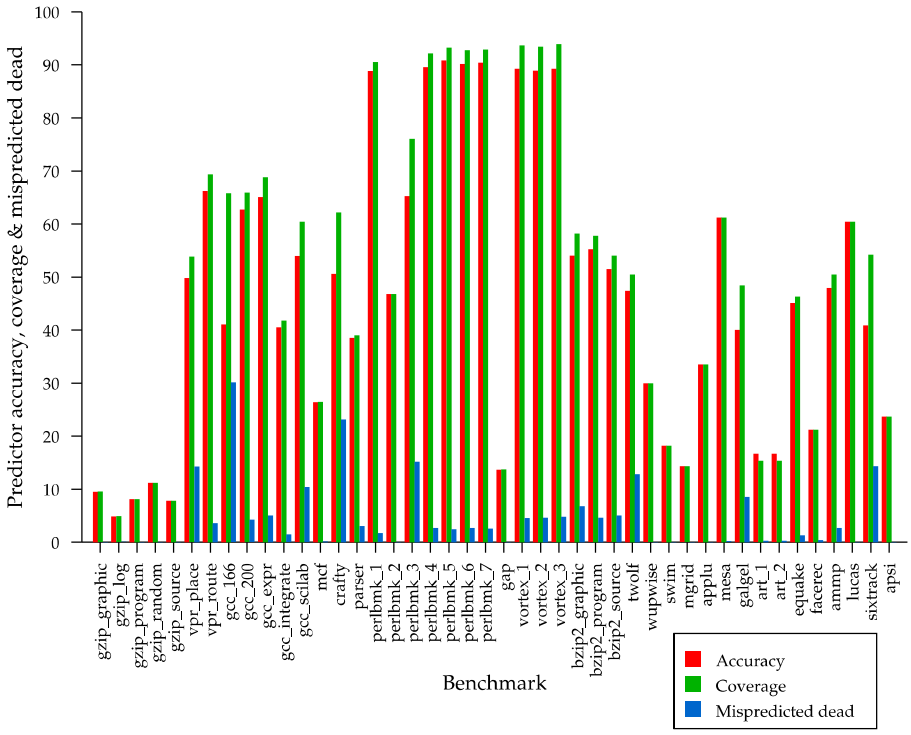

Figure 5.28: `LiveTime_2SAddr` predictor performance

dead which are actually live for the `LiveTime_2SAddr` predictor, as used by Hu *et al.* [HKM02]. Accuracy is fairly low, with an average of 47% and ranging from 4.9% to 91%. Coverage mostly follows accuracy, with an average of 50%, ranging from 4.9% to 94%. The proportion of mispredicted dead prefetch victims is, as usual, low, with an average of 4.4% and a maximum of 30%.

`LiveTime_1Addr3APC` Predictor

Figure 5.29 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LiveTime_1Addr3APC` predictor. As with the previous `LiveTime_2SAddr` predictor, accuracy is fairly low, with an average of 46%, ranging from 8.5% to 82%. Coverage follows accuracy fairly closely, with an average of 48%, ranging from 8.5% to 83%. The proportion of mispredicted dead prefetch victims is low, with an average of 3.7% and a maximum of 21%.

`LiveTime_1PC2Addr` Predictor

Figure 5.30 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LiveTime_1PC2Addr` predictor. Accuracy is higher than that for the previous `LiveTime_1Addr3APC` and `LiveTime_2SAddr` predictors, with an average of 58%, ranging from 15% to 95%. Coverage tracks accuracy for most benchmarks, with an average of 61%, ranging from 15% to 95%. The proportion of mispredicted dead prefetch victims is generally low, with an average of 4.6% and a maximum of 31%.

`LiveTime_1PC` Predictor

Figure 5.31 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `LiveTime_1PC` predictor.

Accuracy and coverage are both generally very low, with an average of 11% and 10% respectively. The proportion of mispredicted dead prefetch victims is generally low, with an average of 2.4% and a maximum of 22%.

Summary

Overall, the novel `LiveTime_1Addr` predictor performs best over the set of benchmarks evaluated. The `LiveTime_2SAddr` predictor proposed by Hu *et al.* is significantly worse [HKM02]. However, the `LiveTime_1Addr` predictor does not perform as well as the best `LastTouch` predictor examined.

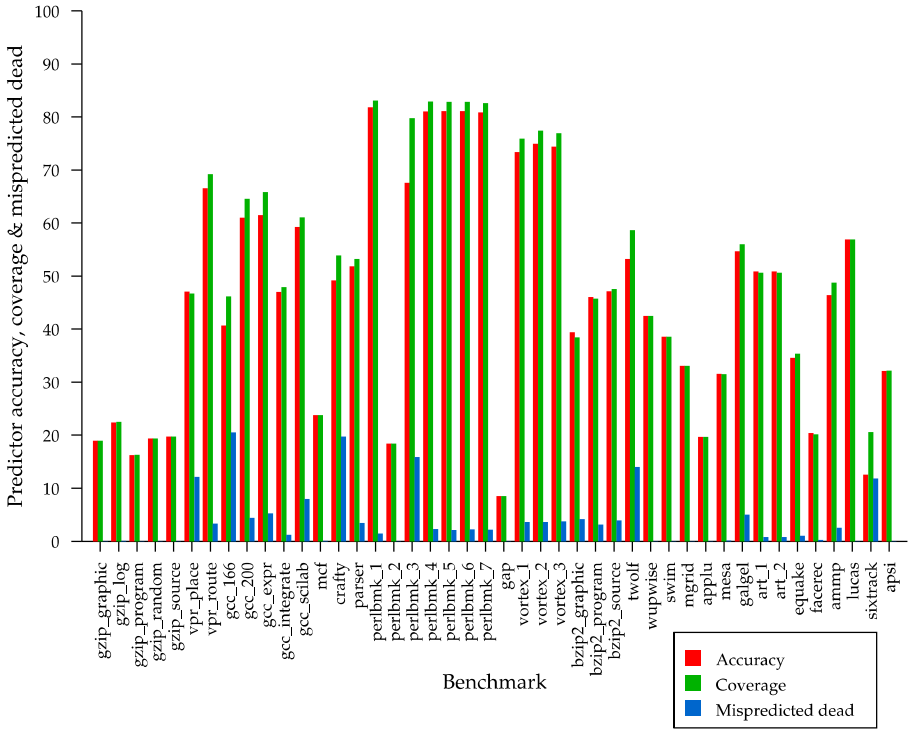

Figure 5.29: `LiveTime_1Addr3APC` predictor performance

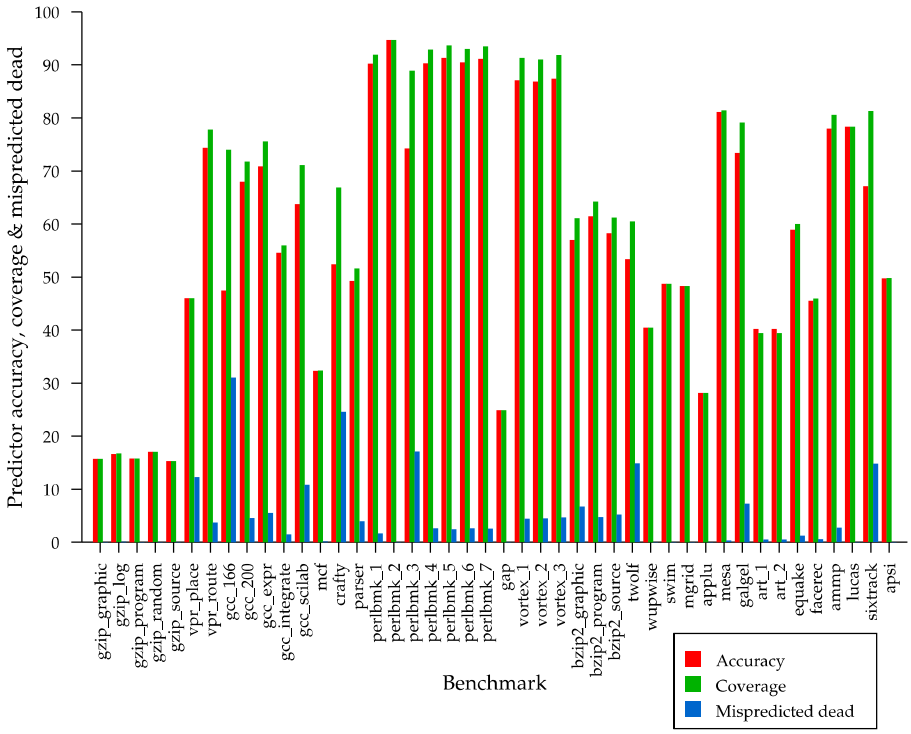

Figure 5.30: `LiveTime_1PC2Addr` predictor performance

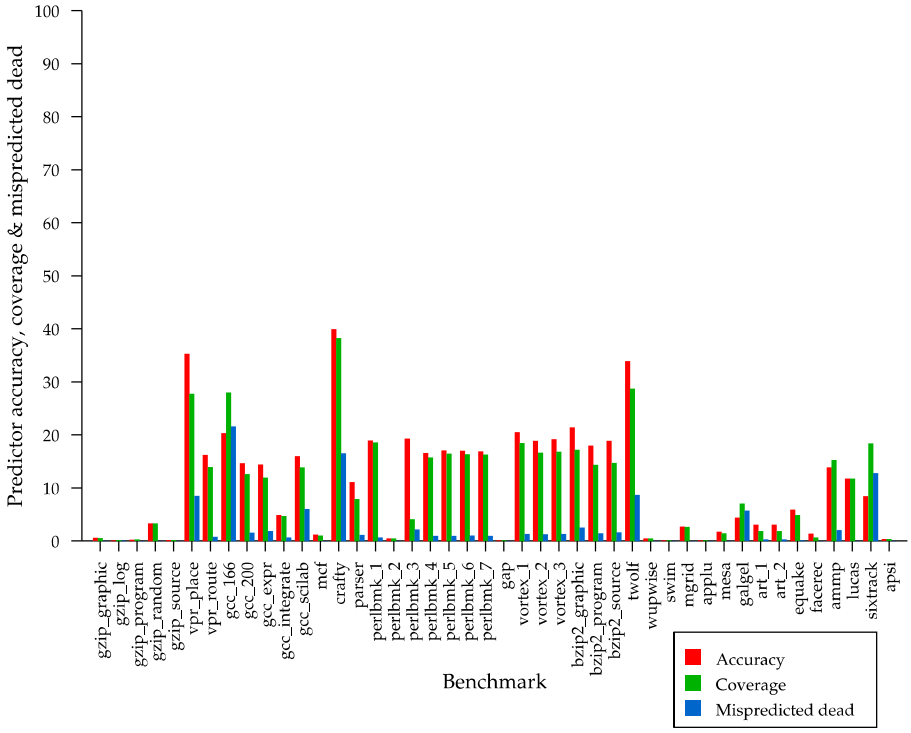

Figure 5.31: `LiveTime_1PC` predictor performance

5.3.6 Dynamic Access Interval Predictors

The dynamic access interval predictor extends the previously evaluated dynamic direct liveness predictor by storing the previously observed access interval with each corresponding entry in the signature table. The current access interval is predicted to be double that previously observed, a similar heuristic to that used by Hu *et al.* [HKM02]. Access intervals are stored at a resolution of a single cycle — a much coarser resolution is probably acceptable, but remains an area for future work. The same signature table size, organisation and latency is used for the dynamic access interval predictors as for the dynamic direct liveness predictors and dynamic live time predictors. Again, if a matching signature cannot be found in the signature table, the cache line is predicted as dead.

`AccessInterval_1Addr` Predictor

Figure 5.32 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `AccessInterval_1Addr` predictor. Overall accuracy is fairly high, with an average of 65%, but a disappointing minimum of 20% and a maximum of 92%. As previously, coverage mostly follows accuracy, with an average of 67%, ranging from 20% to 98%. The proportion of mispredicted dead prefetch victims is generally low, with an average of 4.1% and a maximum of 20%.

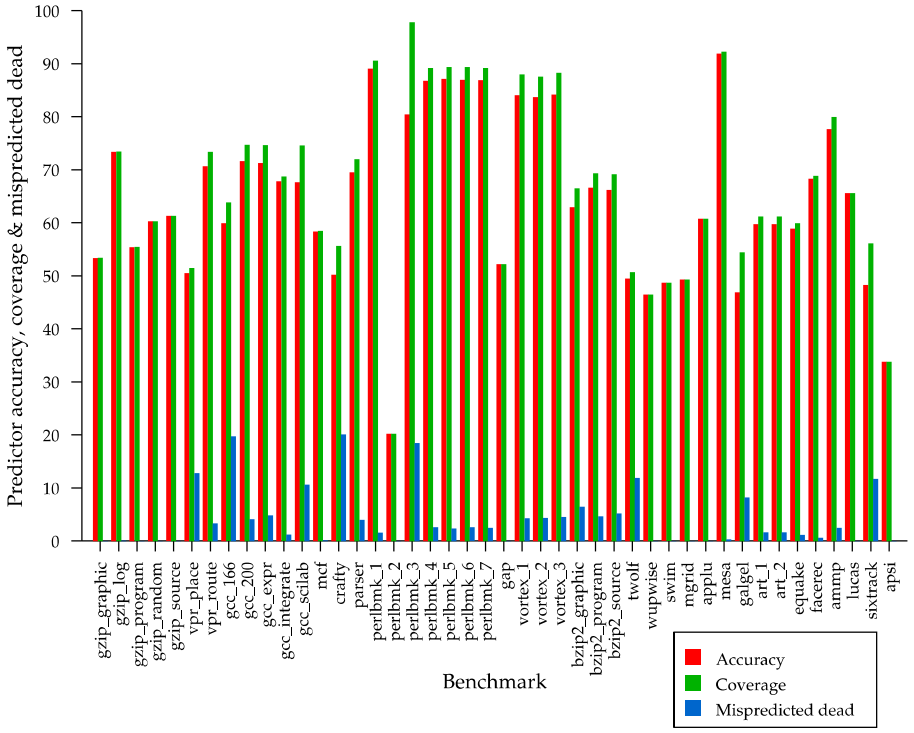

Figure 5.32: `AccessInterval_1Addr` predictor performance

`AccessInterval_2SAddr` Predictor

Figure 5.33 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `AccessInterval_2SAddr` predictor.

Accuracy is fairly high, with an average of 68%, ranging from 34% to 94%. As usual, coverage generally tracks accuracy, with an average of 72%, ranging from 34% to 97%. The proportion of mispredicted dead prefetch victims is, as usual, generally low, with an average of 4.9% and a maximum of 29%.

`AccessInterval_1Addr3APC` Predictor

Figure 5.34 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `AccessInterval_1Addr3APC` predictor. Accuracy is generally fairly high but disappointing low in one case. The average accuracy is 70%, ranging from 22% to 91%. Coverage mostly follows accuracy, with an average of 72%, ranging from 22% to 99%. The proportion of mispredicted dead prefetch victims is generally low, with an average of 4.4% and a maximum of 24%.

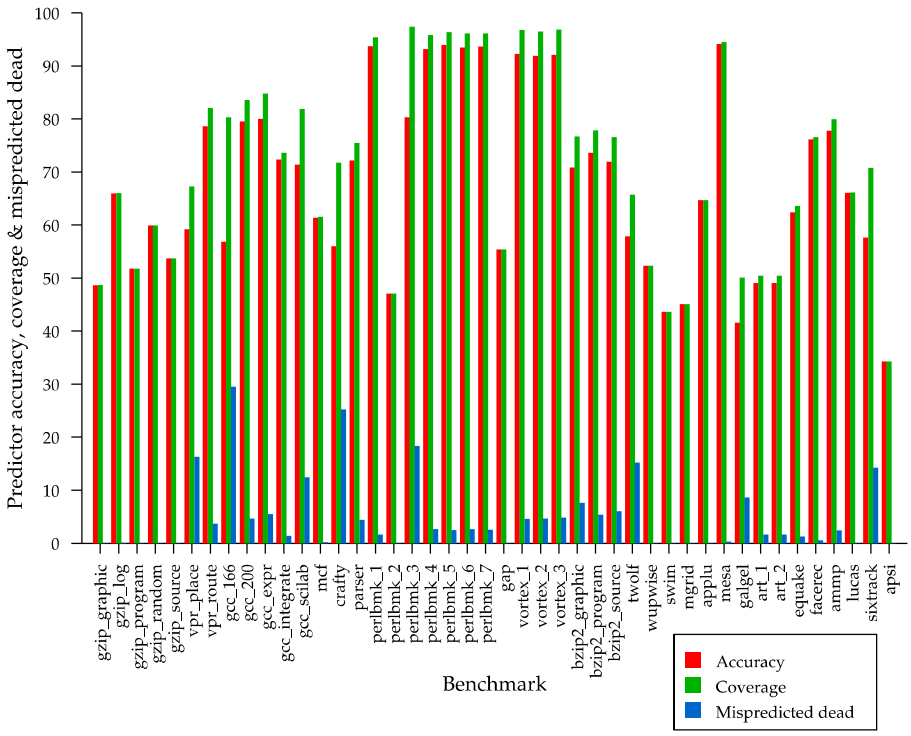

Figure 5.33: `AccessInterval_2SAddr` predictor performance

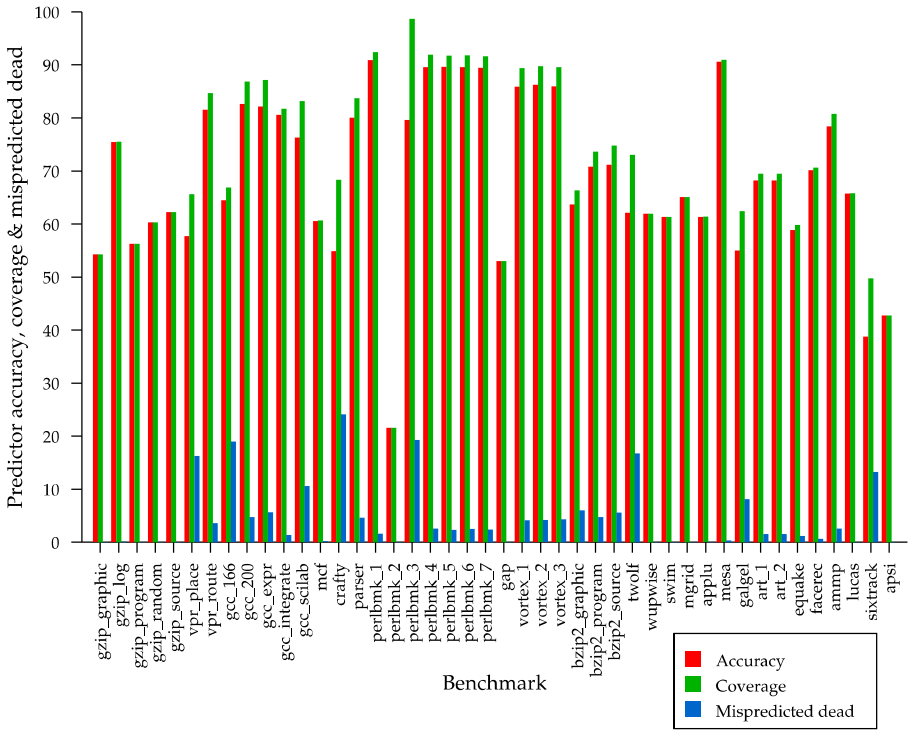

Figure 5.34: `AccessInterval_1Addr3APC` predictor performance

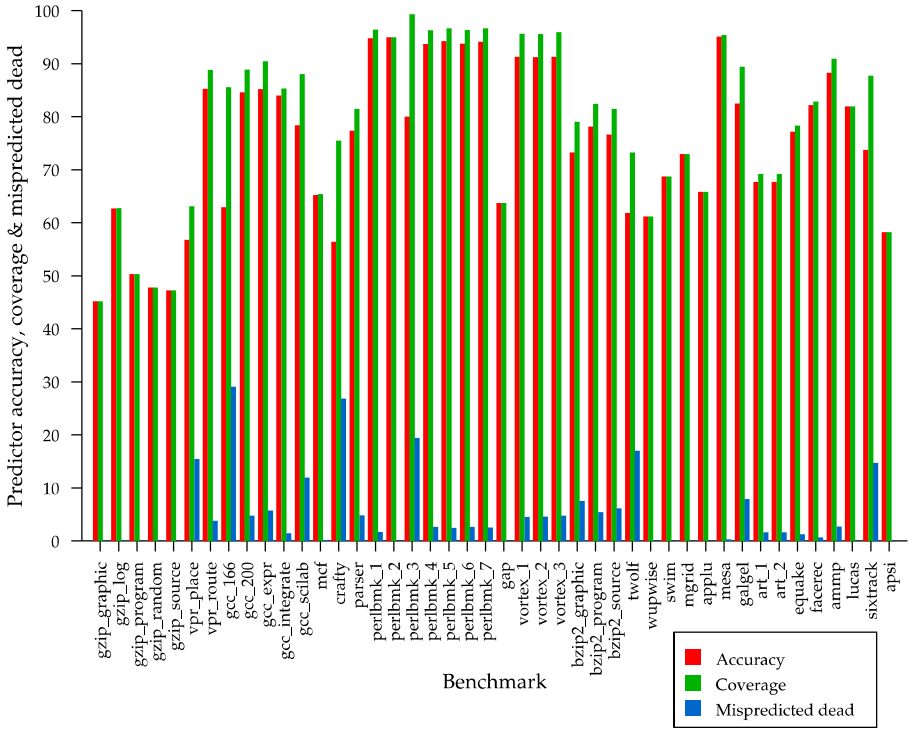

Figure 5.35: `AccessInterval_1PC2Addr` predictor performance

`AccessInterval_1PC2Addr` Predictor

Figure 5.35 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `AccessInterval_1PC2Addr` predictor. Accuracy is fairly high, with an average of 75% and ranging 45% to 95%, considerably higher than any of the previous `AccessInterval` predictors evaluated. Coverage mostly follows accuracy, but is considerably higher in a few cases. The average coverage is 75%, ranging from 45% to 99%. The proportion of mispredicted dead prefetch victims is generally low, with an average of 4.9% and a maximum of 29%.

`AccessInterval_1PC` Predictor

Figure 5.36 shows the predictor accuracy, coverage (in terms of the proportion of times a cache line is predicted to be dead) and the proportion of cache lines predicted as dead which are actually live for the `AccessInterval_1PC` predictor. Accuracy is generally fairly low, with an average of 44%, ranging from a disappointing 5.7% to 72%. Coverage tracks accuracy, with an average of 45% and with the same minimum and maximum values as for accuracy. The proportion of mispredicted dead prefetch victims is generally low, with an average of 3.5% and a maximum of 21%.

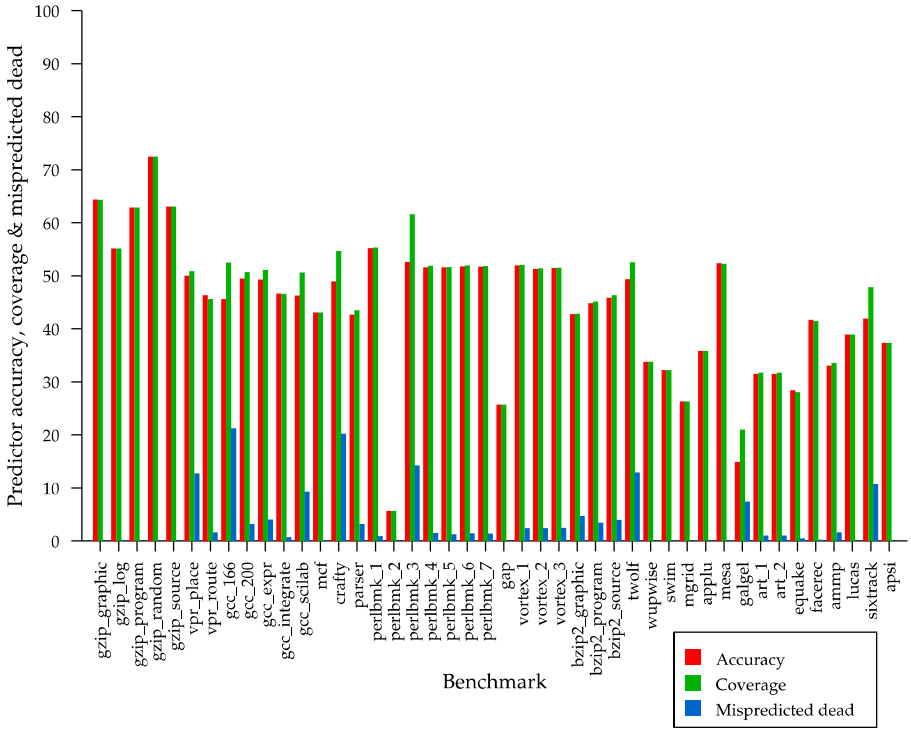

Figure 5.36: `AccessInterval_1PC` predictor performance

Summary

Overall, the novel `AccessInterval_1PC2Addr` predictor performs best over the set of benchmarks evaluated, and indeed performs better than any of the novel or previously-proposed `LiveTime` predictors, although it does not perform as well as the `LastTouch_1PC` predictor.

5.3.7 Overall Performance

The best performing predictors in each of the previous categories are the `LastTouch_1PC`, `LiveTime_1Addr` and `AccessInterval_1PC2Addr` predictors whose overall performance is now evaluated for selection of prefetch victims using the usual miss rate, cache utilisation and instructions committed per cycle metrics. Note that these results are not directly comparable to those presented in Section 5.3.2 since these simulations were run for a shorter period of time.

Miss Rate

Figure 5.37 shows the change in miss rate, measured as the number of misses per thousand instructions, for the `LastTouch_1PC`, `LiveTime_1Addr` and `AccessInterval_1PC2Addr` predictors compared to the conventional prefetcher. The change in miss rate varies substantially between benchmarks, and also between predictors.

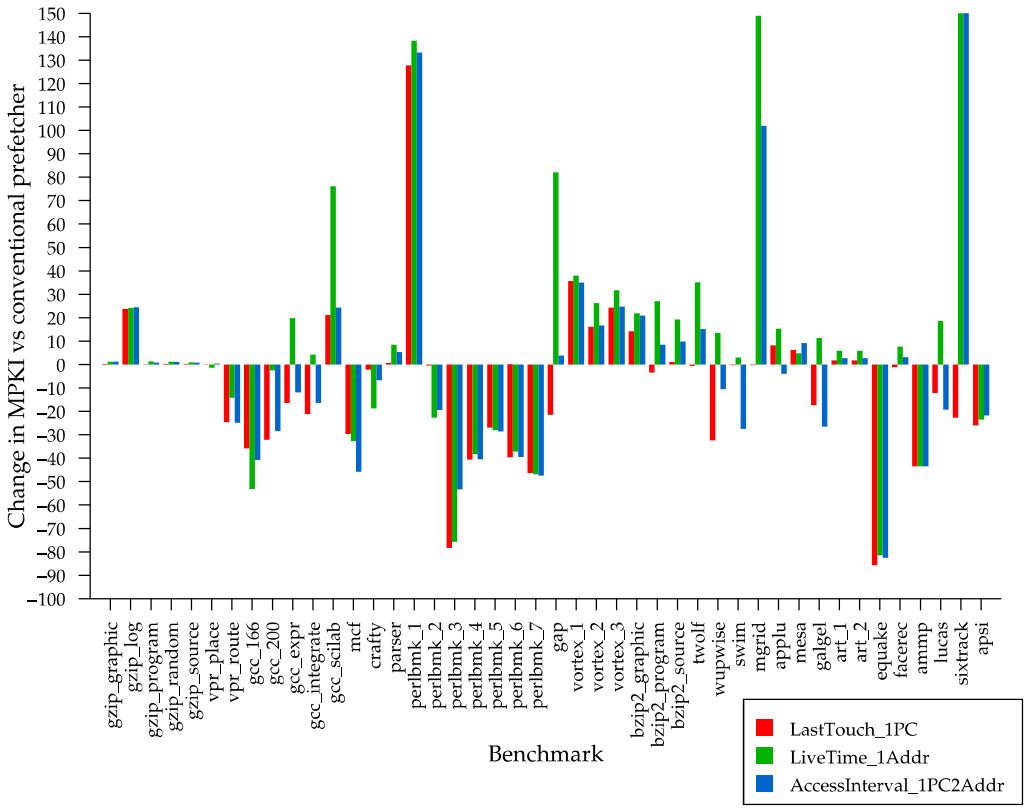

Figure 5.37: Percentage change in miss rate *vs* conventional prefetcher

The average change in miss rate is -6.2%, 10.5% and 0.8% for the three predictors respectively, although for some benchmarks the miss rate decreases substantially (e.g. -86% for `equake`), for other benchmarks it increases substantially (e.g. up to 150% for `sixtrack`), while for the remaining benchmarks very little change is observed. As always, care must be taken to ensure that proportional changes in the miss rate actually correspond to significant changes in the absolute value of the miss rate itself. For example, while the miss rate for `sixtrack` increases substantially, the actual numerical value of the miss rate in all cases is very small. Comparing the three predictors, the `LastTouch_1PC` predictor performs best, followed by the `AccessInterval_1PC2Addr` predictor, followed by the `LiveTime_1Addr`. The unimpressive *average* performance of the latter two benchmarks is heavily influenced by `sixtrack`, `mgrid` and `perlbmk_1`, whose miss rates only increase by a very small amount in absolute terms.

Comparing the change in miss rate to the predictor performance measured in terms of accuracy and the number of mispredicted dead prefetch victims, no clear correlation is apparent, other than the better performing predictors tend to have a better change in miss rate. Some benchmarks with a high proportion of mispredicted dead prefetch victims tend to have a worse change in miss rate, but not all. This is most likely due to the small absolute L2 cache miss rates for some benchmarks, leading to the appearance of large relative changes, but little absolute change.

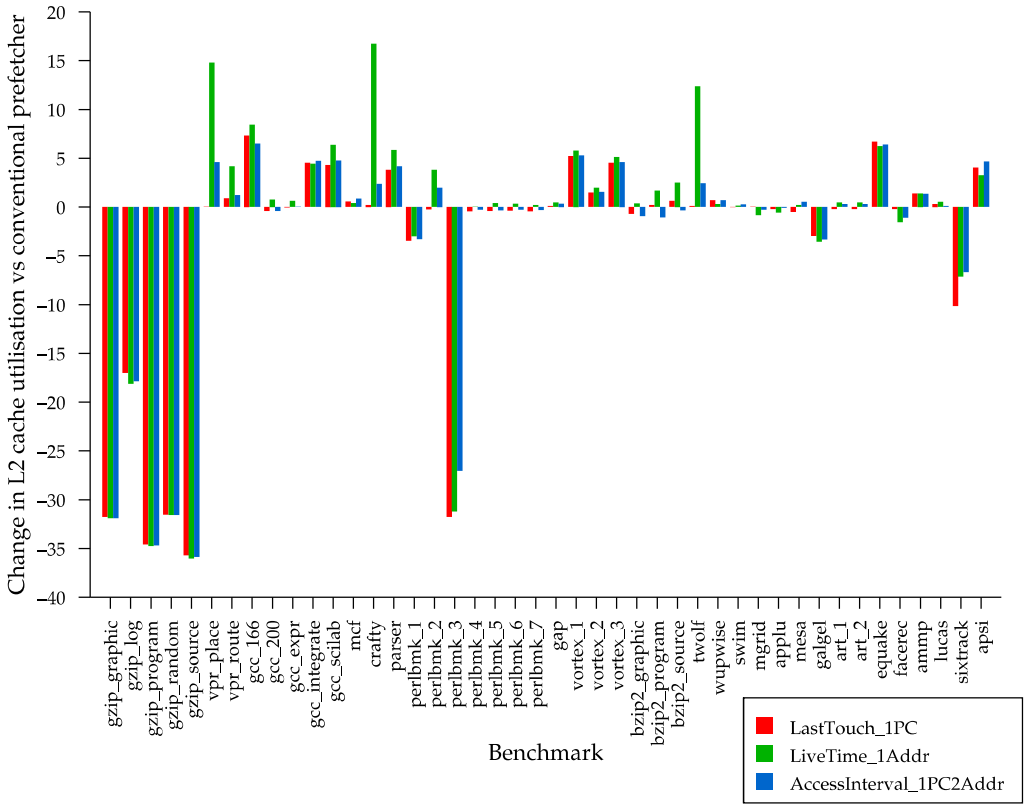

Figure 5.38: Cache utilisation percentage point change *vs* conventional prefetcher

Cache Utilisation

Figure 5.38 shows the percentage point change in the L2 cache utilisation for the `LastTouch_1PC`, `LiveTime_1Addr` and `AccessInterval_1PC2Addr` predictors compared to the conventional prefetcher. There is a substantial decrease in cache utilisation for the `gzip` benchmarks and `perlbmk_3`, otherwise cache utilisation either increases moderately or is unchanged. The average percentage point change in cache utilisation is -3.6%, -2.0% and -3.1% for the three predictors respectively, although this average value is heavily influenced by the substantial decreases already mentioned. The three predictors generally perform similarly, although for a few benchmarks, the `LiveTime_1Addr` predictor significantly outperforms the other two predictors. Again, there is no simple relationship between cache utilisation and the previously observed predictor performance in terms of accuracy and the number of mispredicted dead prefetch victims.

Instructions Committed per Cycle

Figure 5.39 shows the number of Instructions committed Per Cycle (IPC) for the `LastTouch_1PC`, `LiveTime_1Addr` and `AccessInterval_1PC2Addr` predictors compared to the conventional prefetcher. The average change in IPC is 4.2%, -0.3% and 2.7% for the three predictors respectively, although significant variation between both bench-

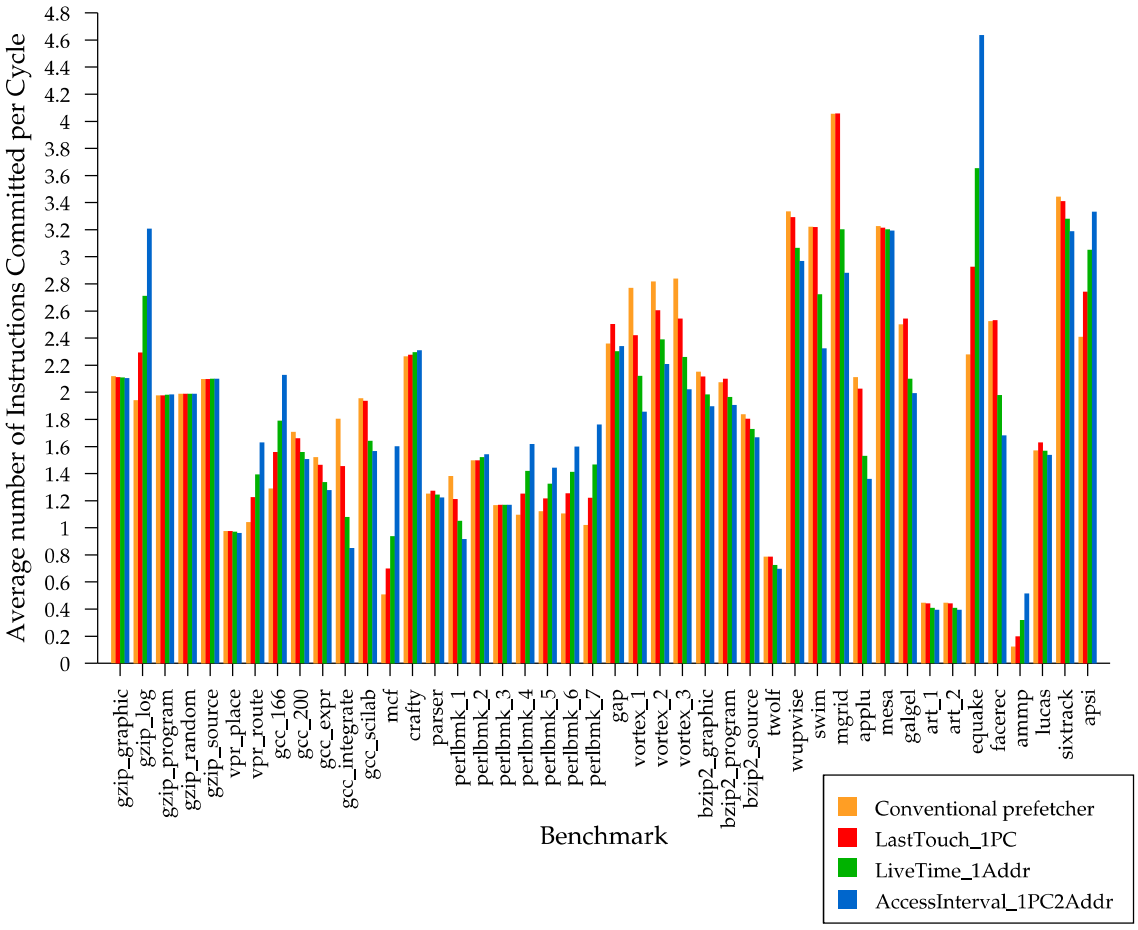

Figure 5.39: IPC compared to conventional prefetcher

marks and predictors is observed. For two benchmarks, `mcf` and `ammp`, a significant increase in IPC is observed, up to 70%. For eight more benchmarks, significant increases in IPC are observed. For 25 benchmarks, little change in IPC is observed, while for the remaining thirteen benchmarks, a moderate decrease in IPC is observed. The change in IPC is related to the change in the absolute miss rate, combined with the criticality of those misses towards overall performance.

Storage Overhead

As already detailed in Section 5.3.4, the three predictors examined above each add approximately 1 MB of storage overhead to the 1 MB L2 cache. This storage could equally well be used to simply double the size of the L2 cache, resulting in fewer conflict and capacity misses, but not compulsory misses. However, the large size and associativity of the L2 cache means that conflict and capacity misses are expected to be relatively rare.

Alternatively, the degree of prefetching could be increased, as well as the prefetch queue size, resulting in more aggressive prefetching. However, overly-aggressive prefetch-

ing would lead to significant cache pollution and bus contention. The decoupled prefetch victim selection scheme detailed here will help ameliorate the former issue, but not the latter, and performance when scaling parameters such as the prefetch degree and prefetch queue size remains an area for future work. Conventional tagged prefetching was chosen as an aggressive, but relatively inaccurate, prefetcher. Numerous other prefetchers described in Section 2.6.2 could also be investigated alongside decoupled prefetch victim selection.

For the `LiveTime_1Addr` and `AccessInterval_1PC2Addr`, timestamps are stored with per-cycle precision using a 64-bit value. Again, a much coarser timestamp would probably suffice, as used by Hu *et al.* [HKM02], leading to either a smaller storage overhead, or increased accuracy by reorganising the table of predicted live times or access intervals.

5.3.8 Future Work

This section has evaluated a wide variety of predictors for use with decoupled prefetch victim selection, presenting overall performance results for the best predictors evaluated. However, a full comparison with other time-based prefetching optimisations such as Hu *et al.*'s timekeeping address correlation table [HKM02] and Lai *et al.*'s dead block correlating prefetcher remain an area for future work, in order to determine the full value of decoupling prefetch target and prefetch victim selection. In addition, the use of partial tag matching in the signature table is another area for future work. Finally, the size and organisation of the last-touch signature table is a third area for future work.

A further area for future work is a more detailed analysis of the `LiveTime` and `AccessInterval` predictor performance, looking at the proportion of times a prediction hits in the signature table in an attempt to separate the predictions made by the signature table contents from the default prediction made when no corresponding value can be found in the signature table.

5.4 Summary

This section examined two applications of predicting the liveness of a cache line. The first application, a filtered victim cache, was evaluated using only static predictors. In terms of reducing the cache miss rate, the novel `DualThresholdLiveTime` predictor typically outperformed the novel `ThresholdLiveTime` predictor and the `ThresholdAccessInterval` predictor previously proposed by Hu *et al.* [HKM02]. All three predictors gave similar results in terms of improving overall performance, increasing IPC by an average of 7.0% compared to an increase of 1.2% for the conventional victim cache when relative to the revised baseline configuration.

The second application, selection of prefetch victims, was evaluated using static and dynamic predictors. Static prediction performance was found to be poor, while the `LastTouch_1PC`, novel `AccessInterval_1PC2Addr` and novel `LiveTime_1Addr`

predictors were found to outperform the `LiveTime_2SAddr` predictor previously proposed by Hu *et al.*. The `LastTouch_1PC` and `AccessInterval_1PC2Addr` predictors were found to increase overall performance by an average of 17% and 16% respectively, compared to an increase of 13% for conventional prefetching, relative to the baseline configuration. The `LiveTime_1Addr` decreased overall performance slightly.

6 Conclusions

The core of this thesis stems from the initial observations (in Chapter 3) that cache utilisation, the proportion of the cache that will be referenced before eviction, is surprisingly poor in current computer systems given that memory system performance is now so critical (as discussed in Chapter 2).

Given that utilisation is so poor for current caches and benchmarks, it was then hypothesised that improvements in cache utilisation might result in efficient improvements in cache hit rates, thereby improving overall performance.

Key to improving cache utilisation is an accurate predictor of the state of a cache line. The inherent predictability of cache line lifetime metrics was examined in Chapter 4, using two different predictability metrics. This forms the second contribution of this work. The questions of *what* to predict, *when* to predict and critically *how* to predict were addressed. A broad range of predictors were then explored, including binary predictors (inspired by successful branch prediction mechanisms), as well as value predictors of live times and access intervals. The precise choice of predictor is dependent upon how it is to be applied, which is the subject of Chapter 5.

The next contribution is in Chapter 5 where the most appropriate predictors were demonstrated in two applications. The first application makes improved use of limited victim cache capacity by selectively allocating space depending on whether the cache line is predicted as being live or dead, known as a filtered victim cache. Two novel predictors were evaluated, one of which frequently outperforms a previously proposed predictor in terms of reducing the cache miss rate. The filtered victim cache improves overall performance relative to the revised baseline system configuration by an average of 7.0%, compared to an average of just 1.2% for a conventional victim cache. The filtered victim cache evaluated requires 4 kB of storage overhead, compared to the 32 kB DL1 cache. However, this storage overhead may be reduced by using considerably coarser resolution timestamps. Victim caches target conflict misses, therefore with higher associativity caches, their benefits are limited and even perfect filtering may not appreciably improve performance.

The second application reduces cache pollution during aggressive prefetching by predicting whether the cache line potentially being replaced is live or dead. A variety of predictors were evaluated, with several novel predictors outperforming a previously proposed predictor in terms of accuracy and costly mispredictions. Three novel predictors were further evaluated, and improved overall performance relative to the baseline system configuration by an average of 17% (for the best predictor), compared to an average of 13% for a conventional prefetcher alone. The predictor overhead was 1 MB of storage, compared to the 1 MB L2 cache. While this storage could have been used to double the size of the L2 cache, it is expected that compulsory misses dominate over conflict and capacity misses for the L2 cache, therefore enabling better prefetching is preferable. By decoupling prefetch target selection and prefetch victim selection, a

more aggressive prefetcher may be used without excessive cache pollution, although performance with different prefetchers remains an area for future investigation.

The final contribution is a preliminary examination of cache utilisation in a chip multiprocessor system, presented in the following section describing areas for future work.

6.1 Future Work

This final section begins by discussing a range of potential future work based upon the cache line lifetime work described thus far. Two additional performance enhancements are described, as well as approaches to reduce overall power consumption. Improved measurement of cache utilisation, as well as the temporal aspects and visualisation of cache line lifetime metrics are discussed. With the almost ubiquitous development of **chip multiprocessors**, cache utilisation in this environment is also evaluated, indicating that the techniques developed throughout this dissertation are even more relevant and applicable to such systems.

6.1.1 Non-Uniform Cache Architectures

Apart from the growing disparity between processor and memory speeds, several other technology trends are necessitating more complicated cache architectures. As process geometries shrink, the reachable area on-die within a single clock cycle shrinks considerably, as demonstrated by Figure 6.1, reproduced from Ho's PhD dissertation [Ho03] which shows the proportion of die area reachable per 16 FO4 clock over a variety of future processes. The signal is assumed to originate at the centre of the die and is constrained to Manhattan routing, hence the diamond-shaped area of reachable distance. The series of decreasing squares shows how the 0.18μm die scales when compared to the other processes. While Ho's model makes various technology assumptions, particularly regarding wire width, it demonstrates the latency of on-chip communication due to wire delays is a highly significant factor when moving to smaller process geometries.

Conventional caches occupy a significant proportion of die area, so the latency to access a particular cache line in such a large on-chip cache is dependent on the exact location of the cache line relative to the processor core. Based on this observation, Kim *et al.* proposed **Non-Uniform Cache Arrays** (NUCA), an array of smaller fixed-latency banks with policies to map data to banks, limit the number of banks that need to be searched and migrate data between banks [KBK02]. Their model indicates that the closest bank of a 16 MB, on-chip L2 cache fabricated using a 50 nm process can be accessed in 4 cycles, while the furthest bank can only be accessed in 47 cycles, the majority of the difference being due to routing to and from the bank, rather than the bank access itself.

Cowell *et al.* later proposed improved policies for mapping, searching and migration [CMB03]. Successful mapping, searching and migration of cache lines within a NUCA cache depends upon accurate prediction of the future usage of cache lines which can be provided by the cache line lifetime predictors described in Chapter 4 and evaluated in Chapter 5. For example, an access interval predictor could be used to both place and

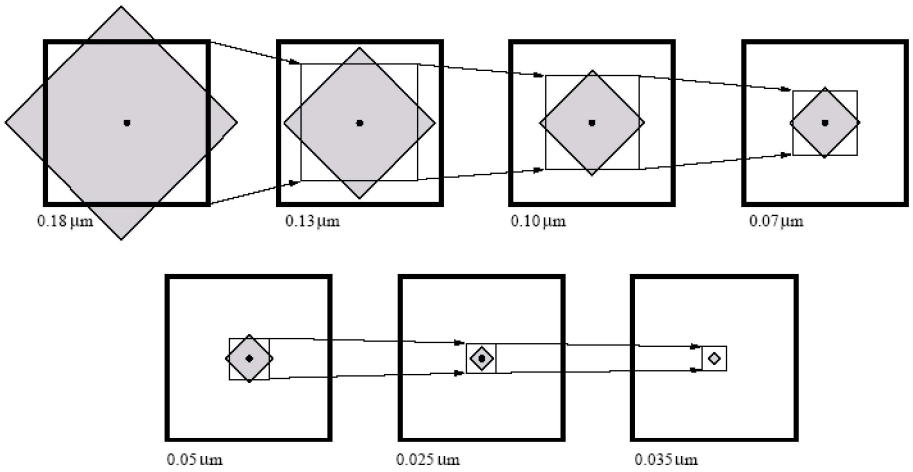

Figure 6.1: Reachable area within a single clock cycle, reproduced from [Ho03].

migrate a cache line so that it is initially placed as far as possible from the processor core (since further cache banks are more numerous), yet migrates towards the processor core ready for when it will be reaccessed. Similarly, a more probabilistic approach to predicting dead cache lines would allow those with higher probability of being dead to be placed further from the processor core than those with a lower probability of being dead.

6.1.2 Cache Replacement

The conventional least-recently used cache replacement policy has long been shown to significantly lag that provided by an optimal algorithm [Bel66] [Puz85], and there has been much previous research on improved cache replacement algorithms. Cache line lifetime metrics may also be used to improve the cache replacement algorithm, for example, choosing to evict cache lines predicted as being dead. However, such an approach would likely require tracking future accesses to evicted cache lines, in order to identify mispredictions. One solution is to use **shadow tags**, as proposed by Puzak [Puz85], which identify if a particular cache access *would* have hit with a different cache organisation, but does not provide the actual cache line contents. The size of the shadow tag array can be significant, requiring a large area and power overhead.

6.1.3 Power Consumption

Besides applications designed to improve performance, there are also numerous applications of cache line lifetime metrics and utilisation predictors in reducing overall power consumption. Increasing power consumption, as previously detailed in Section 2.7, is now a primary design constraint, particularly with increased static power consumption associated with smaller process geometries. As cache sizes grow, the total power consumed by a microprocessor is dominated by the static power consumption of the cache. Liveness predictors may be used to identify those cache lines which

are predicted to be dead, and either transition them into a state-preserving or state-destroying low-power state, as already described in Section 2.7. Evaluation of the predictors described in Chapter 4 for applications to reduce power consumption may be performed using the approach and tools described in Section 2.5.5.

6.1.4 Cache Utilisation Variation

The work described in this dissertation considered cache utilisation over the entire execution of the benchmark, and for each cache as a whole. However, cache utilisation can be broken down in a variety of different ways.

Cache Utilisation Over Time

It has long been observed that applications exhibit different **phases** of execution. Within a single phase, the application exhibits similar behaviour, and between phases, the application exhibits differing behaviour. This behaviour is normally described by parameters such as the cache miss rate, or the range of addresses accessed, but could equally include the cache utilisation. One potential application would be to measure the change in cache utilisation over time, and transition the cache into a low-power state should the cache utilisation fall below some threshold value, indicating that the cache contains a significant proportion of dead (and hence useless) cache lines.

Cache Utilisation Within Caches

Besides varying cache utilisation over time, it is expected that cache utilisation will also vary between different sets within the cache, as well as between different ways in a set-associative cache. For example, cache utilisation is likely to be lower for those cache lines which are further down the least-recently used replacement stack. Again, cache utilisation may be used as a metric to drive selected cache ways or selected cache sets, two methods to decrease power consumption described in Section 2.7.

Cache Utilisation Visualisation

As cache utilisation over time and within specific cache sets and ways is examined, numerous dimensions of data are encountered, leading to problems with visualisation. One potential approach would be to use a **heatmap**, coloured to indicate the live or dead state of a cache line, with cache lines grouped into two dimensions indicating the set and way of each line. As the cache is sampled over time, the heatmap is updated to reflect the current state of each line. However, this visualisation is clearly difficult to convey in printed publications.

6.1.5 Low-Overhead Dynamic Threshold Predictors

The `ThresholdLiveTime` and `ThresholdAccessInterval` predictors described in Chapter 4 and evaluated in Chapter 5 use a single fixed threshold for all cache lines

and all benchmarks. If a link could be established between an easily observable cache parameter, such as the cache miss rate, and the performance of the predictor at a particular threshold, a low-overhead dynamic scheme could be introduced.

6.1.6 Improved Measurement of Cache Utilisation

The use of simulation techniques can only provide an estimate of cache line lifetime metrics and cache utilisation in a real system. The accuracy of this estimate clearly depends upon the fidelity of the simulation, with a tradeoff between accuracy and the time taken to perform the simulation, as discussed in Section 3.2.

More representative results of real systems may be obtained through the use of suitable **hardware performance counters** or through **hardware bus monitoring**, providing a **trace** of program execution and memory operations. A trace of memory operations may be used with a more specific cache simulator, allowing rapid architectural exploration. The advantage of such an approach is that the trace is highly representative of a real application, running on a real system. The primary disadvantage is that since the full system is not simulated, only cache performance results can be obtained and not, for example, the overall system performance. For this reason, trace-based simulation was not used in this dissertation. In addition, second-order effects are not modelled, so that reordering of memory operations due to differing cache performance is not considered.

One potential solution is **hardware-in-the-loop simulation**, whereby the static aspects of the system are provided by hardware, while the dynamic components under investigation are provided by a software emulation. For example, given a suitable platform, the CPU pipeline could be implemented in hardware, providing a rigorous but static simulation, while the cache could be implemented in software, providing a more flexible simulation. Due to the increased effort required for a hardware implementation of a CPU pipeline, this approach is best-suited to using existing models.

6.1.7 Improved Predictor Analysis

The accuracy and coverage metrics reported for the predictors under evaluation are most appropriate if the outcome of mispredictions is similar whether they are a **false positive** or **false negative**. In the case of branch prediction, the cost of mispredicting a not-taken branch as taken, or a taken branch as not-taken, is largely similar. However, in the two applications discussed in Chapter 5, the cost of mispredicting a live cache line as dead is considerably higher than mispredicting a dead cache line as live. This asymmetry was partially addressed in Chapter 5, but a full examination using conventional binary classifier performance metrics such as **sensitivity** and **specificity**, would allow a more informed choice of predictor parameters to be made.

6.1.8 Towards Multicore

The work described so far in this dissertation has concentrated on the analysis and prediction of cache line lifetime metrics and the resulting cache utilisation in single application, single core systems, together with applications aimed at improving overall

system performance. Multiprogramming in a single core environment is not expected to significantly influence cache utilisation since the operating system scheduling quantum size is much greater than live times, dead times or access intervals. Indeed the two billion cycles simulated for each benchmark corresponds to 0.5 seconds at the baseline clock frequency of 4 GHz, while a typical operating system scheduling quantum is around 50 ms, hence the whole simulation corresponds to just ten quanta.

More interesting is multiprogramming in a multiprocessor environment with shared caches. The trend towards **Chip Multiprocessors** (CMPs) discussed in Section 2.3.5 results in an architecture where the amount of cache allocated to each core is actually smaller than in previous architectures, hence making increased cache utilisation ever more important. Preliminary results for cache utilisation in CMPs are obtained as follows.

Method

The M5 Simulator System v1 previously used in this dissertation is unsuitable for simulating CMPs since it does not fully support full-system simulation. Fortunately, since beginning the work described in this dissertation, a much revised version of the simulator has been released, known as M5v2 [BDH+06].

The SPEC CPU2000 benchmark suite previously used in this dissertation is also unsuitable for simulating CMPs since each benchmark only uses a single thread of execution. While multiple benchmarks could be run in parallel on a CMP, dedicated multithreaded benchmarks are more interesting. The Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite consists of 13 modern multithreaded benchmarks intended for use with research CMPs [BKSL08]. Note that at the time of writing, the full set of PARSEC benchmarks has not yet been fully validated on the M5v2 simulator, hence these results should be considered as preliminary and may change with newer releases of the simulator [GHF+09]. In particular, `swaptions` does not run reliably with more than three threads so is omitted, and the remaining benchmarks require validation against reference outputs having modified the simulator to measure cache utilisation.

For each benchmark, the system is started with a simplified processor model without any caches. Once the operating system has booted and the region of interest for the benchmark is encountered, a checkpoint is taken which contains the entire system state. This checkpoint is then read by a detailed processor model including the caches and executed for two billion cycles, or to completion, using the `simsmall` input set.

The system configuration used to evaluate cache utilisation is based upon that used by Blake and Mudge [BM07] which consists of four cores, each with a 64 kB DL1 and IL1 cache, and a shared 2 MB L2 cache. Note that these values are rather conservative for the PARSEC benchmark suite. Each benchmark is run with four threads, to match the number of cores available. Cache coherency is provided by the default M5v2 protocol.

Preliminary Results

Figure 6.2 shows the cache utilisation for the shared L2 cache of the CMP system configuration for each of the PARSEC benchmarks. Direct comparisons with the previous

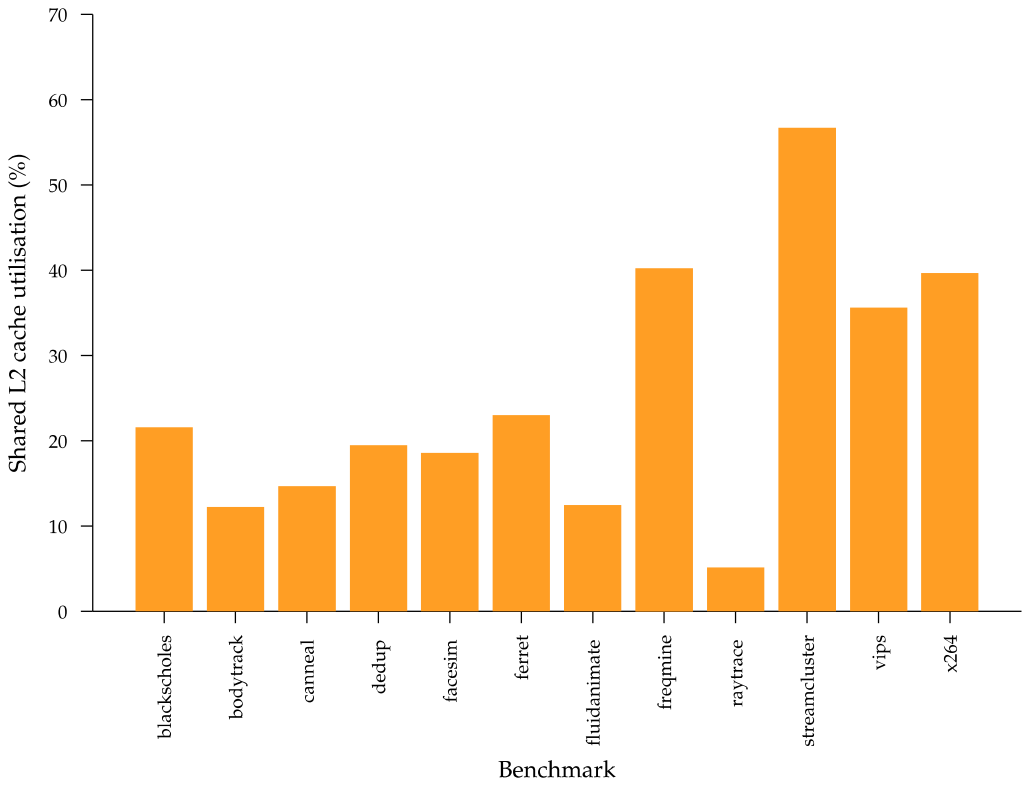

Figure 6.2: Shared L2 cache utilisation for CMP system configuration with PARSEC benchmarks

SPEC CPU2000 results for the baseline system configuration are difficult due to the significant differences between both the system configurations and benchmarks under consideration, but it is clear that cache utilisation in the shared L2 cache is still low, ranging from 5% to 57% with an average of 25%, and indicating that the techniques developed to improve cache utilisation in this dissertation may be equally applicable to CMP systems.

Benchmarks

A

Name	Language	Description[a]
gzip	C	GNU zip compression algorithm, uses Lempel-Ziv coding (LZ77)
vpr	C	FPGA place and route
gcc	C	GNU C compiler, based on v2.7.2.2 targeting Motorola 88100
mcf	C	Combinatorial optimisation / single-depot vehicle scheduling
crafty	C	Chess programme
parser	C	Syntactic parser of English using link grammar
eon	C++	Probabilistic ray tracer
perlbmk	C	Cut-down version of the Perl scripting language, v5.005_03
gap	C	Group theory interpreter
vortex	C	Single-user object-oriented database transaction benchmark
bzip2	C	Compression algorithm based on Julian Seward's bzip2 v0.1
twolf	C	Place and global routing package using simulated annealing

[a] Abridged from http://www.spec.org/cpu2000/CINT2000/

Table A.1: SPEC CPU2000 integer benchmarks

Name	Language	Description[a]
wupwise	Fortran 77	Quantum chromodynamics simulation using the BiCGStab iterative method
swim	Fortran 77	Finite-difference models of the shallow-water equations
mgrid	Fortran 77	3D multi-grid potential field solver
applu	Fortran 77	Solution of five coupled nonlinear PDEs
mesa	C	Software OpenGL library rendering a 3D object from a 2D scalar field
galgel	Fortran 90	Numerical analysis of oscillatory instability of convection
art	C	Adaptive Resonance Theory 2 (ART 2) neural network
equake	C	Simulation of the propagation of elastic waves
facerec	Fortran 90	Face recognition
ammp	C	Molecular dynamics of a protein-inhibitor complex embedded in water
lucas	Fortran 90	Lucas-Lehmer test to check primality of Mersenne numbers
fma3d	Fortran 90	Finite element method to simulate solids and structures
sixtrack	Fortran 77	Tracks particles in an accelerator to check the dynamic aperture
apsi	Fortran 77	Weather prediction

[a] Abridged from http://www.spec.org/cpu2000/CFP2000/

Table A.2: SPEC CPU2000 floating-point benchmarks

Baseline Configuration

B

Example of an entire M5 Simulator configuration file for the baseline configuration executing the `vortex` benchmark.

```
[root]
type=Root
children=cpu0 cpu1 dcache hier icache l2 ram sampler toL2Bus
toMemBus
checkpoint=
clock=4000000000
max_tick=0
output_file=cout
progress_interval=0

[cpu0]
type=SimpleCPU
children=workload
clock=1
dcache=dcache
defer_registration=false
function_trace=false
function_trace_start=0
icache=icache
max_insts_all_threads=0
max_insts_any_thread=0
max_loads_all_threads=0
max_loads_any_thread=0
width=1
workload=cpu0.workload

[cpu0.workload]
type=LiveProcess
cmd=vortex00.peak.ev6 lendian1.raw
env=
executable=/anfs/bigdisc/jrs53/m5/alpha/bin/vortex00.peak.ev6
input=cin
output=cout

[cpu1]
type=FullCPU
```

```
children=branch_pred fupools iq
branch_pred=cpu1.branch_pred
chain_wire_policy=OneToOne
clock=1
commit_model=smt
commit_width=8
dcache=dcache
decode_to_dispatch=15
decode_width=8
defer_registration=false
disambig_mode=normal
dispatch_policy=mod_n
dispatch_to_issue=1
fault_handler_delay=5
fetch_branches=4
fetch_policy=IC
fetch_pri_enable=false
fetch_width=8
fupools=cpu1.fupools
icache=icache
icount_bias=
ifq_size=64
inorder_issue=false
iq=cpu1.iq
iq_comm_latency=1
issue_bandwidth=
issue_width=8
lines_to_fetch=999
loose_mod_n_policy=true
lsq_size=64
max_chains=64
max_insts_all_threads=0
max_insts_any_thread=0
max_loads_all_threads=0
max_loads_any_thread=0
max_wires=64
mispred_recover=3
mt_frontend=true
num_icache_ports=1
num_threads=0
pc_sample_interval=0
prioritized_commit=false
prioritized_issue=false
ptrace=Null
rob_caps=
rob_size=128
storebuffer_size=32
sw_prefetch_policy=enable
```

```
thread_weights=
use_hm_predictor=false
use_lat_predictor=false
use_lr_predictor=true
width=8
workload=cpu0.workload

[cpu1.branch_pred]
type=BranchPred
btb_assoc=4
btb_size=4096
choice_index_bits=12
choice_xor=false
conf_pred_ctr_bits=0
conf_pred_ctr_thresh=0
conf_pred_ctr_type=saturating
conf_pred_enable=false
conf_pred_index_bits=0
conf_pred_xor=false
global_hist_bits=12
global_index_bits=12
global_xor=false
local_hist_bits=10
local_hist_regs=1024
local_index_bits=10
local_xor=false
pred_class=hybrid
ras_size=16

[cpu1.fupools]
type=FuncUnitPool
children=FUList0 FUList1 FUList2 FUList3 FUList4 FUList5
FUList6 FUList7 FUList=cpu1.fupools.FUList0
cpu1.fupools.FUList1 cpu1.fupools.FUList2
cpu1.fupools.FUList3 cpu1.fupools.FUList4
cpu1.fupools.FUList5 cpu1.fupools.FUList6
cpu1.fupools.FUList7
[cpu1.fupools.FUList0]
type=FUDesc
children=opList0
count=6
opList=cpu1.fupools.FUList0.opList0

[cpu1.fupools.FUList0.opList0]
type=OpDesc
issueLat=1
opClass=IntAlu
opLat=1
```

```
[cpu1.fupools.FUList1]
type=FUDesc
children=opList0 opList1
count=2
opList=cpu1.fupools.FUList1.opList0
cpu1.fupools.FUList1.opList1

[cpu1.fupools.FUList1.opList0]
type=OpDesc
issueLat=1
opClass=IntMult
opLat=7

[cpu1.fupools.FUList1.opList1]
type=OpDesc
issueLat=9
opClass=IntDiv
opLat=10

[cpu1.fupools.FUList2]
type=FUDesc
children=opList0 opList1 opList2
count=4
opList=cpu1.fupools.FUList2.opList0
cpu1.fupools.FUList2.opList1 cpu1.fupools.FUList2.opList2

[cpu1.fupools.FUList2.opList0]
type=OpDesc
issueLat=1
opClass=FloatAdd
opLat=3

[cpu1.fupools.FUList2.opList1]
type=OpDesc
issueLat=1
opClass=FloatCmp
opLat=3

[cpu1.fupools.FUList2.opList2]
type=OpDesc
issueLat=1
opClass=FloatCvt
opLat=3

[cpu1.fupools.FUList3]
type=FUDesc
children=opList0 opList1 opList2
```

```
count=2
opList=cpu1.fupools.FUList3.opList0
cpu1.fupools.FUList3.opList1 cpu1.fupools.FUList3.opList2

[cpu1.fupools.FUList3.opList0]
type=OpDesc
issueLat=1
opClass=FloatMult
opLat=4

[cpu1.fupools.FUList3.opList1]
type=OpDesc
issueLat=12
opClass=FloatDiv
opLat=12

[cpu1.fupools.FUList3.opList2]
type=OpDesc
issueLat=20
opClass=FloatSqrt
opLat=20

[cpu1.fupools.FUList4]
type=FUDesc
children=opList0
count=0
opList=cpu1.fupools.FUList4.opList0

[cpu1.fupools.FUList4.opList0]
type=OpDesc
issueLat=1
opClass=MemRead
opLat=1

[cpu1.fupools.FUList5]
type=FUDesc
children=opList0
count=0
opList=cpu1.fupools.FUList5.opList0

[cpu1.fupools.FUList5.opList0]
type=OpDesc
issueLat=1
opClass=MemWrite
opLat=1

[cpu1.fupools.FUList6]
type=FUDesc
```

```
children=opList0 opList1
count=2
opList=cpu1.fupools.FUList6.opList0
cpu1.fupools.FUList6.opList1

[cpu1.fupools.FUList6.opList0]
type=OpDesc
issueLat=1
opClass=MemRead
opLat=1

[cpu1.fupools.FUList6.opList1]
type=OpDesc
issueLat=1
opClass=MemWrite
opLat=1

[cpu1.fupools.FUList7]
type=FUDesc
children=opList0
count=1
opList=cpu1.fupools.FUList7.opList0

[cpu1.fupools.FUList7.opList0]
type=OpDesc
issueLat=3
opClass=IprAccess
opLat=3

[cpu1.iq]
type=StandardIQ
caps=0 0 0 0
prioritized_issue=false
size=128

[dcache]
type=BaseCache
access_dist_bkt=10
access_dist_diff_bkt=10
access_dist_diff_max=100000
access_dist_max=1000000
access_dist_min=0
adaptive_compression=false
addr_range=0:18446744073709551615
assoc=2
block_size=64
compressed_bus=false
compression_latency=0
```

```
dead_dist_bkt=10
dead_dist_diff_bkt=10
dead_dist_diff_max=100000
dead_dist_max=1000000
dead_dist_min=0
diff_dist=true
do_copy=false
hash_delay=1
hier=hier
in_bus=Null
latency=2
lifo=false
live_dist_bkt=10
live_dist_diff_bkt=10
live_dist_diff_max=100000
live_dist_max=1000000
live_dist_min=0
max_miss_count=0
mem_trace=Null
mshrs=16
out_bus=toL2Bus
perfect_cache=false
prefetch_access=false
prefetch_cache_check_push=true
prefetch_data_accesses_only=false
prefetch_degree=1
prefetch_latency=10
prefetch_miss=false
prefetch_past_page=false
prefetch_policy=none
prefetch_serial_squash=false
prefetch_use_cpu_id=true
prefetcher_size=100
prioritizeRequests=false
protocol=Null
reload_dist_bkt=10
reload_dist_max=1000000
reload_dist_min=0
repl=Null
size=131072
split=false
split_size=0
store_compressed=false
subblock_size=0
tgts_per_mshr=16
trace_addr=0
two_queue=false
write_buffers=8
```

```
[exetrace]
print_cpseq=false
print_cycle=true
print_data=true
print_effaddr=true
print_fetchseq=false
print_iregs=false
print_opclass=true
print_thread=true
speculative=false

[hier]
type=HierParams
do_data=false
do_events=true

[icache]
type=BaseCache
access_dist_bkt=10
access_dist_diff_bkt=10
access_dist_diff_max=100000
access_dist_max=1000000
access_dist_min=0
adaptive_compression=false
addr_range=0:18446744073709551615
assoc=2
block_size=64
compressed_bus=false
compression_latency=0
dead_dist_bkt=10
dead_dist_diff_bkt=10
dead_dist_diff_max=100000
dead_dist_max=1000000
dead_dist_min=0
diff_dist=true
do_copy=false
hash_delay=1
hier=hier
in_bus=Null
latency=1
lifo=false
live_dist_bkt=10
live_dist_diff_bkt=10
live_dist_diff_max=100000
live_dist_max=1000000
live_dist_min=0
max_miss_count=0
```

```
mem_trace=Null
mshrs=8
out_bus=toL2Bus
perfect_cache=false
prefetch_access=false
prefetch_cache_check_push=true
prefetch_data_accesses_only=false
prefetch_degree=1
prefetch_latency=10
prefetch_miss=false
prefetch_past_page=false
prefetch_policy=none
prefetch_serial_squash=false
prefetch_use_cpu_id=true
prefetcher_size=100
prioritizeRequests=false
protocol=Null
reload_dist_bkt=10
reload_dist_max=1000000
reload_dist_min=0
repl=Null
size=131072
split=false
split_size=0
store_compressed=false
subblock_size=0
tgts_per_mshr=16
trace_addr=0
two_queue=false
write_buffers=8

[l2]
type=BaseCache
access_dist_bkt=10
access_dist_diff_bkt=10
access_dist_diff_max=100000
access_dist_max=1000000
access_dist_min=0
adaptive_compression=false
addr_range=0:18446744073709551615
assoc=8
block_size=64
compressed_bus=false
compression_latency=0
dead_dist_bkt=10
dead_dist_diff_bkt=10
dead_dist_diff_max=100000
dead_dist_max=1000000
```

```
dead_dist_min=0
diff_dist=true
do_copy=false
hash_delay=1
hier=hier
in_bus=toL2Bus
latency=12
lifo=false
live_dist_bkt=10
live_dist_diff_bkt=10
live_dist_diff_max=100000
live_dist_max=1000000
live_dist_min=0
max_miss_count=0
mem_trace=Null
mshrs=128
out_bus=toMemBus
perfect_cache=false
prefetch_access=false
prefetch_cache_check_push=true
prefetch_data_accesses_only=false
prefetch_degree=1
prefetch_latency=10
prefetch_miss=false
prefetch_past_page=false
prefetch_policy=none
prefetch_serial_squash=false
prefetch_use_cpu_id=true
prefetcher_size=100
prioritizeRequests=false
protocol=Null
reload_dist_bkt=10
reload_dist_max=1000000
reload_dist_min=0
repl=Null
size=1048576
split=false
split_size=0
store_compressed=false
subblock_size=0
tgts_per_mshr=16
trace_addr=0
two_queue=false
write_buffers=8

[ram]
type=BaseMemory
addr_range=0:18446744073709551615
```

```
compressed=false
do_writes=false
hier=hier
in_bus=toMemBus
latency=200
snarf_updates=true
uncacheable_latency=1000

[sampler]
type=Sampler
periods=1000000000 2000000000
phase0_cpus=cpu0
phase1_cpus=cpu1

[serialize]
count=10
cycle=0
dir=cpt.\%012d
period=0

[stats]
descriptions=true
dump_cycle=0
dump_period=0
dump_reset=false
ignore_events=
mysql_db=
mysql_host=
mysql_password=
mysql_user=
project_name=test
simulation_name=test
simulation_sample=0
text_compat=true
text_file=m5stats.txt

[toL2Bus]
type=Bus
clock=1
hier=hier
width=128

[toMemBus]
type=Bus
clock=2
hier=hier
width=16
```

```
[trace]
bufsize=0
dump_on_exit=false
file=cout
flags=
ignore=
start=0
```

Processor and Memory Trends

C

Processor	Year	Clock Frequency (MHz)	Process Geometry (μm)	Power Consumption (W)	Area (mm^2)	Transistors (x1000)
Intel 80286	1982	12.5	1.5	1	47	134
Intel 80386	1985	16	1.5	2	43	275
Intel 80486	1989	25	0.8	2.5	81	1,200
Intel Pentium	1993	66	0.8	15	90	3,100
Intel Pentium Pro	1997	200	0.35	45	308	5,500
Intel Pentium 4	2001	1500	0.18	55	217	42,000
Intel Pentium 4 EE	2006	2800	0.065	115	435	1,328,000

Table C.1: Processor Trends, 1982-2006

Processor	Year	L1 Cache		L2 Cache		L3 Cache	
		Size[a]	Latency	Size	Latency	Size	Latency
Intel 80286	1982	none	n/a	none	n/a	none	n/a
Intel 80386	1985	16 kB	n/a	none	n/a	none	n/a
Intel 80486	1989	8 kB	n/a	none	n/a	none	n/a
Intel Pentium	1993	16 kB	n/a	256 kB	n/a	none	n/a
Intel Pentium Pro	1997	16 kB	n/a	256 kB	n/a	none	n/a
Intel Pentium 4	2001	24 kB	2 cycles	256 kB	7 cycles	1024 kB	n/a
Intel Pentium 4 EE	2006	32 kB	4 cycles	2048 kB	20 cycles	16 MB	n/a

[a]In the case of split instruction and data caches, the total size is reported

Table C.2: Cache Trends, 1982-2006. "n/a" denotes data not available or not applicable

Memory	Year	Latency[a]	Density
DRAM	1980	225 ns	0.06 Mb/chip
Page Mode DRAM	1983	170 ns	0.25 Mb/chip
Fast Page Mode DRAM	1986	125 ns	1 Mb/chip
Fast Page Mode DRAM	1993	75 ns	16 Mb/chip
Synchronous DRAM	1997	62 ns	64 Mb/chip
DDR SDRAM	2000	52 ns	256 Mb/chip
DDR2 SDRAM	2006	40 ns	1 Gb/chip

[a]Latency estimated as row access time plus column access time

Table C.3: Memory Trends, 1980-2006

Data for Tables C.1, C.2 and C.3 taken from [Pat04], [GBCH01], [Lud06] and [Mic06].

Bibliography

[AGVO05] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization*, 2(1):55–77, 2005.

[AL03] Todd Austin and SimpleScalar LLC. Simplescalar 3.0d. http://www.simplescalar.com/, 2003.

[Alb99] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *MICRO 32: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259, Washington, DC, USA, 1999. IEEE Computer Society.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, February 2002.

[ALR02] Amit Agarwal, Hai Li, and Kaushik Roy. DRG-Cache: A data retention gated-ground cache for low power. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, pages 473–478, New York, NY, USA, 2002. ACM Press.

[ARM06] ARM Limited. *ARM1136JF-S and ARM1136J-S Technical Reference Manual*. ARM Limited, 2006.

[AZMM04] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *ACM-SE 42: Proceedings of the 42nd Annual Southeast Regional Conference*, pages 267–272, New York, NY, USA, 2004. ACM Press.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[BDH+06] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey,

D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88 –598, February 2008.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, 1966.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, April 2005.

[BGK95] Douglas C. Burger, James R. Goodman, and Alain Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report 1261, University of Wisconsin-Madison Computer Sciences Department, January 1995.

[BHR03] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-oriented full-system simulation using M5. In *CAECW Proceedings of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*. ACM Press, 2003.

[BKSL08] 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*, October 2008.

[BM07] Geoffrey Blake and Trevor Mudge. Duplicating and verifying LogTM with OS support in the M5 simulator. In *WDDD '07: Proc. 6th Workshop on Duplicating, Deconstructing, and Debunking*, June 2007.

[BMST06] Major Bhadauria, Sally A. McKee, Karan Singh, and Gary Tyson. A precisely tunable drowsy cache management mechanism. In *Proceedings of the Watson Conference on Interaction between Architecture, Circuits, and Compilers*, October 2006.

[Bor01] Shekhar Borkar. Low power design challenges for the decade (invited talk). In *ASP-DAC '01: Proceedings of the 2001 Conference on Asia South Pacific Design Automation*, pages 293–296, New York, NY, USA, 2001. ACM Press.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual Symposium on Computer Architecture*, pages 83–94, 2000.

[CB95] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.

[CMB03] Christopher Cowell, Csaba Andras Moritz, and Wayne Burleson. Improved modeling and data migration for dynamic non-uniform cache accesses. In *Proceedings of the Second Annual Workshop on Duplicating, Deconstructing and Debunking*, 2003.

[CN10] H.W. Cain and P. Nagpurkar. Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 microprocessor. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 203 –212, March 2010.

[Cor99] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*. Compaq Computer Corporation, 1999.

[Cor06] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*. Intel Corporation, 2006.

[CR95] M. Charney and A. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, February 1995.

[CSK+99] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 186–195, Washington, DC, USA, 1999. IEEE Computer Society.

[CT99] Jamison D. Collins and Dean M. Tullsen. Hardware identification of cache conflict misses. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 126–135, Washington, DC, USA, 1999. IEEE Computer Society.

[CYFM04] Chi F. Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. Accurate and complexity-effective spatial pattern prediction. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 276, Washington, DC, USA, 2004. IEEE Computer Society.

[DM97] James Dundas and Trevor Mudge. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 68–75, New York, NY, USA, 1997. ACM.

[EH03] Jan Edler and Mark D. Hill. Dinero IV trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/ markhill/DineroIV/, September 2003.

[FKM+02] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 148–157, Washington, DC, USA, 2002. IEEE Computer Society.

[GAV95] Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS 1995: Proceedings of the 9th International Conference on Supercomputing*, pages 338–347, New York, NY, USA, 1995. ACM Press.

[GBCH01] Stephen H. Gunther, Frank Binns, Douglas M. Carmean, and Jonathan C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 2001.

[GHF+09] Mark Gebhart, Joel Hestness, Ehsan Fatehi, Paul Gratz, and Stephen W. Keckler. Running PARSEC 2.1 on M5. Technical report, The University of Texas at Austin, Department of Computer Science, October 2009.

[Hen00] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[HHA+03] Heather Hanson, M. S. Hrishikesh, Vikas Agarwal, Stephen W. Keckler, and Doug Burger. Static energy reduction techniques for microprocessor caches. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3):303–313, 2003.

[HKM02] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 209–220, Washington, DC, USA, 2002. IEEE Computer Society.

[HM94] Luddy Harrison and Sharad Mehrotra. A data prefetch mechanism for accelerating general-purpose computation. Technical Report 1351, University of Illinois at Urbana-Champaign, May 1994.

[Ho03] Ron Ho. *On-Chip Wires: Scaling and Efficiency*. PhD thesis, Department of Electrical Engineering, Stanford University, August 2003.

[HP03] John Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[HR00] 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*, pages 107–116. ACM Press, 2000.

[HS89] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. In *IEEE Transactions on Computers*, volume 38, pages 1612–1630, 1989.

[JB07] C. R. Johns and D. A. Brockenshire. Introduction to the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 51(5), September 2007.

[JG97] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[JH97] Teresa L. Johnson and Wen-mei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, New York, NY, USA, 1997. ACM Press.

[Jou90] 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 Symposium on Computer Architecture*, pages 364–373, 1990.

[JS97] L. John and A. Subramanian. Design and performance evaluation of a cache assist to implement selective caching. In *Proceedings of the 15th International Conference on Computer Design*, 1997.

[JTSE10] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 60–71, New York, NY, USA, 2010. ACM.

[KBK02] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[KHM01] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 240–251, New York, NY, USA, 2001. ACM Press.

[KJBF10] Samira M. Khan, Daniel A. Jiménez, Doug Burger, and Babak Falsafi. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 489–500, New York, NY, USA, 2010. ACM.

[KKL06] Ismail Kadayif, Mahmut Kandemir, and Feihui Li. Prefetching-aware cache line turnoff for saving leakage energy. In *ASP-DAC '06: Proceedings of the 2006 Conference on Asia South Pacific Design Automation*, pages 182–187, New York, NY, USA, 2006. ACM Press.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organisation. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 81–87, 1981.

[KS05] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement algorithms. In *ICCD '05: Proceedings of the 2005 IEEE International Conference on Computer Design*, pages 61–68, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[KW98] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, Washington, DC, USA, 1998. IEEE Computer Society.

[Lab03] Francois Labonte. Microprocessors through the ages. http://www-vlsi.stanford.edu/group/chips.html, April 2003.

[LAS+09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, New York, NY, USA, 2009. ACM.

[LDV+04] Lin Li, Vijay Degalahal, N. Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. Soft error and energy consumption interactions: A data

cache perspective. In *ISLPED '04: Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 132–137, New York, NY, USA, 2004. ACM Press.

[LF00] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 139–148, New York, NY, USA, 2000. ACM Press.

[LFF01] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, New York, NY, USA, 2001. ACM Press.

[LFHB08] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, Washington, DC, USA, 2008. IEEE Computer Society.

[LKT+02] L. Li, Ismail Kadayif, Yuh-Fang Tsai, Narayanan Vijaykrishnan, Mahmut T. Kandemir, Mary Jane Irwin, and Anand Sivasubramaniam. Leakage energy management in cache hierarchies. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 131–140, Washington, DC, USA, 2002. IEEE Computer Society.

[LPZ+04] Yingmin Li, Dharmesh Parikh, Yan Zhang, Karthik Sankaranarayanan, Mircea Stan, and Kevin Skadron. State-preserving vs. non-state-preserving leakage control in caches. In *DATE '04: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 22–27, Washington, DC, USA, 2004. IEEE Computer Society.

[LR02] Wei-Fen Lin and Steven K. Reinhardt. Predicting last-touch references under optimal replacement. Technical Report CSE-TR-447-02, University of Michigan, 2002.

[LS98] Mikko H. Lipasti and John Paul Shen. Exploiting value locality to exceed the dataflow limit. *Int. J. Parallel Program.*, 26(4):505–538, 1998.

[LT00] Hsien-Hsin S. Lee and Gary S. Tyson. Region-based caching: an energy-delay efficient memory architecture for embedded processors. In *CASES '00: Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 120–127, New York, NY, USA, 2000. ACM.

[LTF00] Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *MICRO 33: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 11–21, New York, NY, USA, 2000. ACM Press.

[Lud06] Christian Ludloff. sandpile.org - the world's leading source for pure technical x86 processor information. http://www.sandpile.org/, October 2006.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138—147, 1996.

[LY09] Wanli Liu and Donald Yeung. Enhancing LTP-driven cache management using reuse distance information. *Journal of Instruction-Level Parallelism*, 11, April 2009.

[MBJ09] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Cacti 6.0: A tool to model large caches. Technical Report HPL-2009-85, HP Laboratories, 2009.

[McF92] Scott McFarling. Cache replacement with dynamic exclusion. In *ISCA '92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 191–200, New York, NY, USA, 1992. ACM Press.

[McF93] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[Mic06] Micron. *Micron DDR2 SDRAM Datasheet*, August 2006.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MSK05] Yan Meng, Timothy Sherwood, and Ryan Kastner. Exploring the limits of leakage power reduction in caches. *ACM Transactions on Architecture and Code Optimization*, 2(3):221–246, 2005.

[NMT+98] Koji Nii, Hiroshi Makino, Yoshiki Tujihashi, Chikayoshi Morishima, Yasushi Hayakawa, Hiroyuki Nunogami, Takahiko Arakawa, and Hisanori Hamano. A low power SRAM using auto-backgate-controlled MT-CMOS. In *ISLPED '98: Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 293–298, New York, NY, USA, 1998. ACM Press.

[Par06] Jeff Parkhurst. From single core to multi-core to many core: are we ready for a new exponential? In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, pages 210–210, New York, NY, USA, 2006. ACM.

[Pat04] David A. Patterson. Latency lags bandwidth. *Communications of the ACM*, 47(10), October 2004.

[PK94] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[PKK09] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. Instruction-based reuse-distance prediction for effective cache management. In *SAMOS'09: Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation*, pages 49–58, Piscataway, NJ, USA, 2009. IEEE Press.

[PS93] C. H. Perleberg and A. J. Smith. Branch target buffer design and optimization. *IEEE Trans. Comput.*, 42(4):396–412, 1993.

[PSSK05] Salvador Petit, Julio Sahuquillo, Jose M. Such, and David Kaeli. Exploiting temporal locality in drowsy cache policies. In *CF '05: Proceedings of the 2nd Conference on Computing Frontiers*, pages 371–377, New York, NY, USA, 2005. ACM Press.

[Puz85] Thomas Roberts Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachusetts Amherst, 1985.

[PYF+00] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *ISLPED '00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 90–95, New York, NY, USA, 2000. ACM Press.

[QJP+07] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News*, 35(2):381–391, 2007.

[RBS96] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[RD96] J.A. Rivers and E.S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. *ICPP: International Conference on Parallel Processing*, 1:154, 1996.

[RKB+09] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 371–382, New York, NY, USA, 2009. ACM.

[RMS98] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS-VIII: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, New York, NY, USA, 1998. ACM Press.

[RTT+98] Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens. Utilizing reuse information in data cache management. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing*, pages 449–456, New York, NY, USA, 1998. ACM Press.

[SJ01] Premkishore Shivakumar and Norman P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Compaq WRL, August 2001.

[SJLW01] Srikanth T. Srinivasan, Roy Dz-ching Ju, Alvin R. Lebeck, and Chris Wilkerson. Locality vs. criticality. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 132–143. ACM Press, 2001.

[SLT02] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, Washington, DC, USA, 2002. IEEE Computer Society.

[Smi82] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3), September 1982.

[SV94] Dimitrios Stiliadis and Anujan Varma. Selective victim caching: A method to improve the performance of direct-mapped caches. Technical Report UCSC-CRL-93-41, University of California, Santa Cruz, October 1994.

[TFMP95] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[TH04] Masamichi Takagi and Kei Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 20–30, New York, NY, USA, 2004. ACM Press.

[TRS+99] Edward S. Tam, Jude A. Rivers, Vijayalakshmi Srinivasan, Gary S. Tyson, and Edward S. Davidson. Active management of data caches by exploiting reuse information. *IEEE Trans. Comput.*, 48(11):1244–1259, 1999.

[TTJ06] David Tarjan, Shyamkumar Thoziyoor, and Norman P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories Palo Alto, 2006.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2–4), 2005.

[TVTD01] Edward S. Tam, Stevan A. Vlaovic, Gary S. Tyson, and Edward S. Davidson. Allocation by conflict: A simple, effective multilateral cache management scheme. In *ICCD 2001: Proceedings of the 2001 International Conference on Computer Design*, pages 133–140, 2001.

[VHR+07] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98 –589, February 2007.

[VL99] Steven P. VanderWiel and David J. Lilja. A compiler-assisted data prefetch controller. In *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, page 372, Washington, DC, USA, 1999. IEEE Computer Society.

[VL00] Steven P. VanderWiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[VSP02] Sivakumar Velusamy, Karthik Sankaranarayanan, and Dharmesh Parikh. Adaptive cache decay using formal feedback control. In *WMPI '02: Proceedings of the 2002 Workshop on Memory Performance Issues*, May 2002.

[Wan04] Zhenlin Wang. *Cooperative Hardware/Software Caching for Next-Generation Memory Systems*. PhD thesis, University of Massachusetts, Amherst, February 2004.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[WPO+07] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C. Hoe, Derek Chiou, and Krste Asanovic. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, 2007.

[WSM+05] Joe Wetzel, Ed Silha, Cathy May, Brad Frey, Junichi Furukawa, and Giles Frazier. *PowerPC Virtual Environment Architecture*. International Business Machines Corporation, 2005.

[XIC09] Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. Combining thread level speculation helper threads and runahead execution. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 410–420. ACM, June 2009.

[YFPV02] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 151, Washington, DC, USA, 2002. IEEE Computer Society.

[YP92] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ISCA '92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, New York, NY, USA, 1992. ACM Press.

[ZTRC03] Huiyang Zhou, Mark C. Toburen, Eric Rotenberg, and Thomas M. Conte. Adaptive mode control: A static-power-efficient cache design. *Trans. on Embedded Computing Sys.*, 2(3):347–372, 2003.