Cache-Aware Virtual Page Management

by

Alexander Szlavik

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

With contemporary research focusing its attention primarily on benchmark-driven performance evaluation, studying fundamental memory characteristics has gone by the wayside. This thesis presents a systematic study of the expected performance characteristics for contemporary multi-core CPUs. These characteristics are the primary influence on benchmarking variability and need to be quantified if more accurate benchmark results are desired.

With the aid of a new, highly customizable, micro-benchmark suite, these CPU-specific attributes are evaluated and contrasted. The benchmark tool provides the framework for accurately measuring instruction throughput and integrates hardware performance counters to gain insight into machine-level caching performance. Additionally, the Linux operating system's impact on cache utilization is evaluated. With careful virtual memory management, cache-misses may be reduced, significantly contributing to benchmark result stability. Finally, a popular cache performance model, stack distance profile, is evaluated with respect to contemporary CPU architectures. While particularly popular in multi-core contention-aware scheduling projects, modern incarnations of the model fail to account for trends in CPU cache hardware, leading to measurable degrees of inaccuracy.

Acknowledgements

This work would not have been possible without the tireless efforts and insights provided by my supervisor Dr. Martin Karsten. Additionally, “Thank You” for providing me with tremendous opportunities through my studies.

I would also like to extend my graditude to my thesis readers, Dr. Kenneth Salem and Dr. Ashraf Aboulnaga.

Finally, my friends and familiy who supported me through a rough time, especially toward the completion of this thesis. Thank you for your understanding and encouraging words over this past year.

Dedication

This work is dedicated to my mother, Renate Szlavik. Hopefully this work finds its way into her collection of one-of-a-kind books.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern CPUs employ ever increasingly complex memory hierarchies in order to overcome the performance throttling issues of the “memory wall”. The memory wall problem characterizes the divergence in clock frequency of CPUs and main memory. In order to evaluate the performance impact of a new optimization technique or a new hardware component, much emphasis is given to large benchmark suites. These aggregate tests are designed to reflect real-world workloads, test many different types of applications, and form a common basis for comparison. However, these large benchmark suites are tailored to whole system performance analysis and are not well suited for more detailed analysis of memory performance characterstics. Perhaps even more importantly, research is focused on solving the problems associated with memory bottlenecks before presenting an inherent understanding of the variables at play. For example, questions such as “How well does a CPU cache mask the memory wall problem?” or “What can I do to improve my memory utilization?” are no longer a focus of modern research. The last time serious research efforts where dedicated to answering the “Why?” of memory performance issues dates back to the 1990s. Ideally, once a thorough understanding is established, the problem would be expressed mathematically, allowing conjectures for future generations of the same architecture.

This thesis presents an in-depth study of contemporary memory hierarchy performance characteristics. In order to profile and evaluate the impact of CPU cache hit rates as well as the influence of the virtual memory subsystem, a new microbenchmark suite has been implemented. Studies are performed using this custom benchmark utility, which is capable of stressing individual levels of the memory hierarchy. This characteristic provides the necessary data that is required when modelling a memory hierarchy. While the tool is designed with portability in mind, it does include features which are exclusively provided by the GNU Linux operating system (henceforth referred to as Linux). With the aid of performance monitoring hardware counters and a finely tuned execution engine, simulation grade accuracy is achieved. However, unlike simulation, the utility is run on real hardware measuring performance characteristics that map onto production software. Furthermore, this thesis work includes the capabilities for measuring simultaneously executing workloads on multi-processor or multi-core CPU systems.

An additional component of this work is the evaluation of mathematical models related to memory performance. Traditionally, the working set model is used as a first order approximation of an application's temporal locality behaviour. The model is coarsely grained, utilizing memory pages as its unit of measure. Stack Distance Profiles (SDP) are a refinement of the working set model and attempt to characterize temporal reuse at an arbitrary scale. The most common application of SDPs involve modelling cache utilization, predict shared CPU cache performance, and provide hints to the operating system scheduler with regards to multi-thread scheduling and placement. This work raises concerns with the SDP methodology when applied to cache level optimization and further discusses the impact of memory layout on cache performance.

Finally, a taxonomy of the memory hierarchy is presented with an emphasis on the importance of memory layout and with respect to CPU cache memory. This thesis elaborates on the shortcomings of virtual memory and re-exposes a performance throttling attribute of modern memory management. These attributes, which are inherent in the operation of virtual memory, are the primary cause of software performance non-determinism and as such are quantified and analyzed in this thesis.

Chapter 2

Background and Related Work

A cache is a dedicated architectural component which is designed to exploit the *locality of reference* principle [6]. In the most generic terms, a cache is a component which stores duplicate elements of data, usually with the purpose of reducing their access latencies. To achieve these optimization goals, caches are placed "closer" to the execution pipeline of the CPU when compared to main memory. Furthermore, caches are operated at the CPU's clock frequency, overcoming the limitations imposed by slower *random-access memory* (RAM). The increase in proximity provides for shorter data paths and a reduction in memory controller related circuitry, enabling CPU caches to satisfy memory requests more efficiently.

2.1 CPU Caches

Traditionally, CPUs are clocked separately from components beyond the motherboard's northbridge. A motherboard's northbridge provides access to main memory, as well as auxillary system components, such as USB and the PCI bus. There exist two reasons for this design choice: clock skew and slower components. Clock skew refers to the difference in a generated clock signal between two points on a wire. As frequencies of CPUs rise, the amount of time a signal remains set decreases symmetrically. Consequently, to maintain a coherent view of the generated clock signal, system componentes must be packed densly. Additionally, components such as main memory or the PCI bus are clocked at much lower rates than those of a CPU, leading to what is known as the memory wall problem [40]. Main memory is generally composed of dynamic random access memory (DRAM), which requires a relatively low clock frequency due to the capacitive memory storage technology

[18] [30]. Particularly due to the difference in clock speeds of the CPU and main memory, memory operations require many multiples of CPU cycles to complete. While not all of these cycles are wasted (thanks to complex out-of-order instruction issuing), hardware optimization techniques cannot mask all memory access delays.

CPU caches provide a probabilistic approach to mitigate the memory wall problem. Instead of paying the memory access cost on every access, there exists a chance that the current memory reference can be satisfied by the CPU cache. A memory request, which is served from cache is known as a cache hit, while failed cache look-ups are known as cache misses. Cache misses are further subdivided into compulsory, conflict and capacity misses. A compulsory cache miss occurs when a particular address is first referenced within an application. In this scenario, there is no chance (disregarding possible prefetching) that the data is cached. Capacity misses occur when a cache cannot hold all data referenced by the application. Finally, conflict misses are the only manageable miss type. A conflict miss is the result of replacing a cache line with another, when there is unused space available. Conflict misses are also the focus of this thesis and have received much attention since the advent of symmetric multi-processing CPU architectures [7] [12] [13] [26] [48].

2.1.1 Organization

Modern CPUs have complex memory hierarchies to mask the memory wall problem. A typical memory hierarchy is shown in Figure 2.1. The figure shows the CPU's registers at the lowest level, followed by the caching hierarchy and ending with main memory.

A CPU's register file is the smallest unit of memory storage and being directly attached to the execution pipeline, the fastest. Many instructions require data to be loaded into registers to perform the requested operation. Since there only exists a finite number of registers, their usage must be multiplexed during execution. Both hardware and software optimizations have been suggested and implemented to optimize the usage of CPU registers. Common techniques include register windowing [51], register colouring [14] and renaming [43].

The cache hierarchy begins above the register file and often contains 2 or 3 discrete components. The various levels of the cache are referred to by the prefix letter "L" followed by the distance from the pipeline. In general, the higher a cache component is placed in the cache hierarchy, the larger its capacity and access latency. L1 caches are further subdivided into data and instruction caches, as their access patterns greatly differ in practice. Instruction accesses have a large degree of locality, while data accesses often exhibit larger degrees of randomness. A cache which contains both data and instructions is referred to

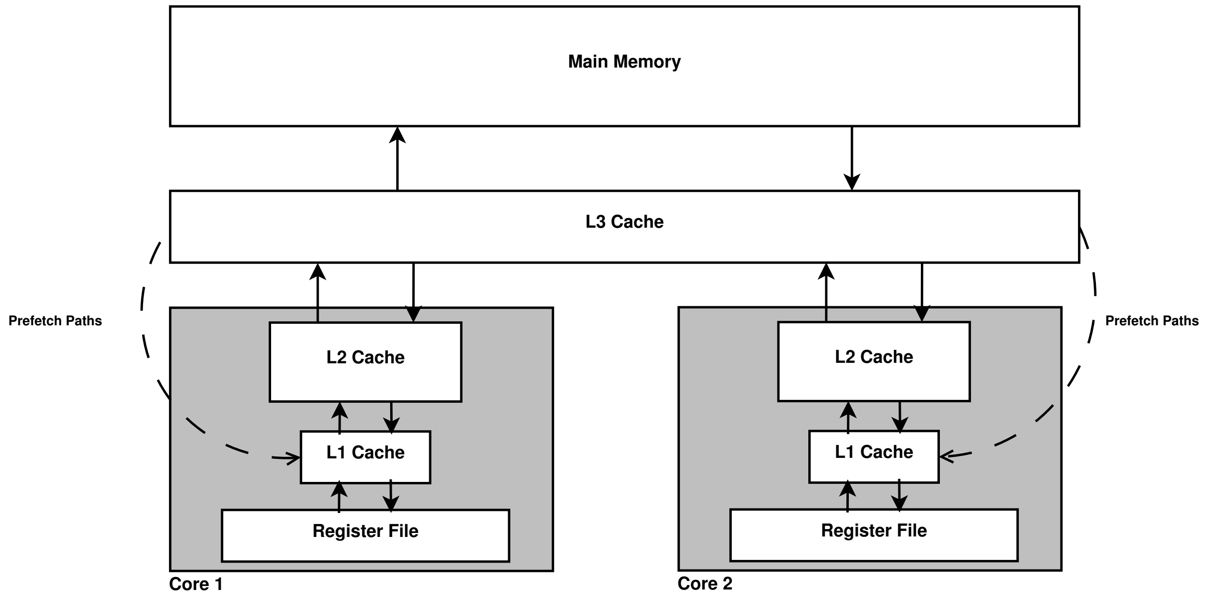


Figure 2.1: Common memory hierarchies

as a unified cache. L2 caches are mostly unified (true for current generation Intel, AMD, and PowerPC CPUs) and larger in size. On architectures that support multiple hardware threads or multiple cores, L2 caches may be shared in order to faciliate faster inter-core data transfers. Finally, L3 caches are a more recent addition to the caching hierarchy. This level is traditionally unified, mostly used in multi-core architectures and originally added as an off-chip component. With the advancement in transistor fabrication processes, L3 cache are now included directly on the CPUs die, greatly reducing their access latency.

CPU caches are generally much smaller when compared to the total available memory of modern computing devices (especially desktop and server style devices). Consequently, caches only store a subset of the addressable memory, requiring additional mechanisms to verify if the cache's contents contain a copy of a given memory address. In order to determine which pieces of cached data correspond to which memory addresses, CPU caches utilize the most significant bits of the memory address as a unique identifier. These bits are commonly referred to as the "Tag" bits and are used during cache lookup requests to determine if data related to a given Tag is available. Rather than store individual bytes of data, CPU caches store information in collections of bytes, referred to as "cache lines". This optimization appeals to the principle of spatial locality, hypothesizing that data located spatially close to the least recently referenced data item is more likely to be required in the next memory access. By caching multiple bytes within a single cache line,

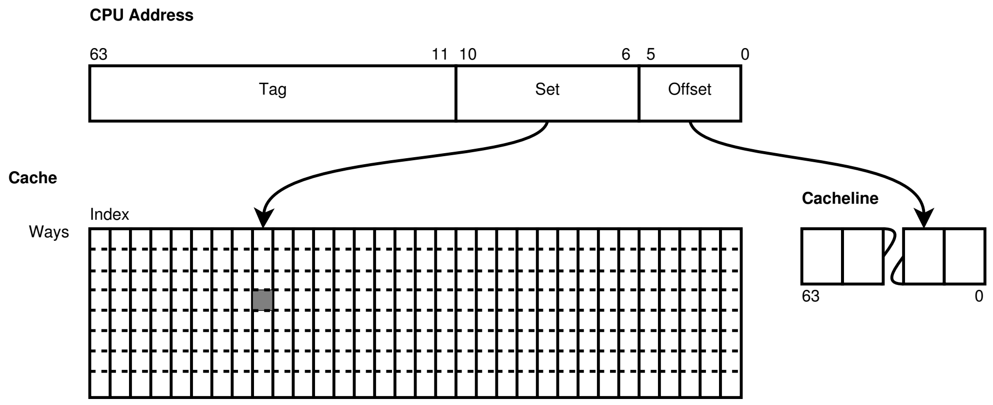


Figure 2.2: An 8 way set-associative cache of size 16 kB

this principle is honoured, providing equally fast memory access latencies for spatially close data.

Cache lines form the smallest storable unit within a CPU cache. To obtain a particular byte from a cache line, the least significant bits of the memory address are used. For example: A CPU cache utilizing 64 byte cache lines, requires the least 6 bits of address in order to select a single byte from the cache line. This is the case in the example depicted by Figure 2.2 and is visualized by the cache line blocks (each one byte in size) on the right hand side. The Figure depicts a 16 kB cache with 64 byte cache lines.

2.1.2 Associativity

Additionally, Figure 2.2 is organized in a set-associative manner. Associativity characterizes the degree of overlap within a cache. Since CPU caches are much smaller than the total physical memory available, a mapping of physical to cache memory must be created. Various methods exist, with the most prominent being direct mapping, set-associative and fully associative implementations. These mapping heuristics are a natural consequence of the equivalence classes formed by any group of bits within an address.

Direct Mapped A direct mapped cache has a many-to-one mapping between the machines address words and the CPU cache's index. That is to say, that every memory location has a fixed location within the CPU cache to which it may be mapped. This scenario is realized by using log_2(number of cache lines in cache) bits from the memory address. The cache depicted in Figure 2.2 provides 256 cache lines of 64 bytes each (16 kB cache), thus requiring 8 bits of the address ($log_2(256) = 8$). This resulting memory address division is represented in Figure 2.3 and details the “Tag”, “Line” and “Offset” bits. It is important to realize that two memory locations, seperated by exactly 16 kB, would map into the same cache line resulting in the reuse of a cache line by another memory location. The advantage of direct mapped caches is the look-up speed, which is strictly constant and thus excels on large caches. The downside of this approach is the limited utilization of the cache memory when the cache is small. In particular workloads which are laid out in a non-contiguous manner will suffer a greater number of conflict misses when compared to the other two approaches.

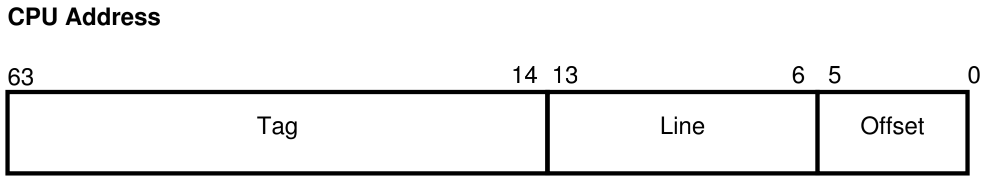


Figure 2.3: A direct mapped cache’s address compartmentalization

Fully Associative The other extreme comes in the form of a fully associative cache. In this setup, every address may map into any cache index, thus forming a many-to-many relationship. Such a cache would utilize an address line division as is shown in Figure 2.4, only requiring the “Tag” and “Offset” bits. The trade-offs are the complement of those in a direct mapped cache. On every memory access, all cache indexes must now be checked simultaneously for hits. This is of course exceedingly expensive on large caches and is only utilized on slower off-chip victim caches. Furthermore, non-direct mapped caches require cache line replacement heuristics. A fully associative cache must choose which cache entry to evict when a conflict or capacity miss is encountered. However, fully associative caches boast the best hit-rates, as an effort can be made to utilize more (or all) of the available cache space [18].

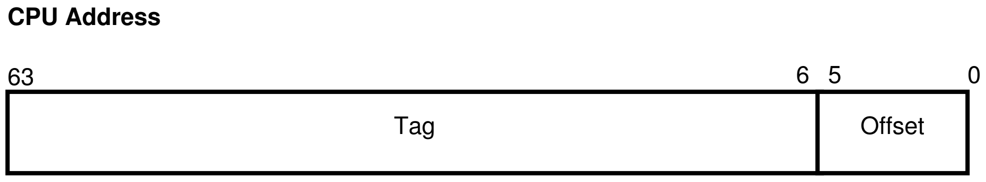


Figure 2.4: A fully associative cache’s address compartmentalization

Set Associative Finally, set associative caches strike a balance between direct mapped and fully associative caches. Congruence classes are formed by the "Set" bits of the address, which denote a set of cache lines. Within the set, a fully associative paradigm is assumed. In effect, an N-way set associative cache allows a particular cache line to be stored in one of N places. Figure 2.2 depicts an 8-way set associative cache, with a cache access to congruence class 8. Within the set, the 8 possible locations must be checked simultaneously as in a fully associative implementation.

Set associative caches are the predominant paradigm on contemporary multi-purpose CPUs, since their design reduces the conflict misses which are present in direct mapped caches [27]. Simultaneously, the cache is provided with some degree of temporal access pattern knowledge, which can be exploited when dealing with capacity misses. Depending on the layer of the cache, and thus its size, a varying degree of associativity is used to enhance the overall cache performance with respect to hit ratios. Empirical studies have shown that associativity degrees above a factor of 8 do not improve hit-rates of a cache and are rarely used at the L1 or L2 caching levels [18].

In summary, a CPU cache follows a variable sequence of steps in order to satisfy memory requests. First, depending on the degree of associativity used, the memory address provided to the cache is compartmentalized. Consider a set-associative cache implementation, as it is the most commonly found design at this time. Upon determining the distribution of bits among the Tag, Set and Offset, the Set bits are used to fetch a particular column of cache lines. Each stored cache line's Tag is then compared to the Tag bits of the given memory address. In case of a cache hit (matching Tags between one stored cache line and the given address' Tag bits) the cache line is fetched. Finally, the Offset bits are utilized to acquire the sought after byte from the cache. In the complementory scenario of a cache miss, the memory request is forwarded to the next level of memory (which could be another cache level or main memory).

2.1.3 Cache Sharing

Due to the increase in transistor density and the resulting increase in heat dissipation from contemporary CPU architectures, clock frequencies have reached an upper bound with current manufacturing technologies. To continue the technological advancement of modern CPUs, hardware manufacturers have introduced Chip-Multi-Processors (CMP). CMPs capitalize on increasing transistor density by duplicating the CPUs execution pipeline, registers and functional units to create hardware-level parallelism. In order to facilitate

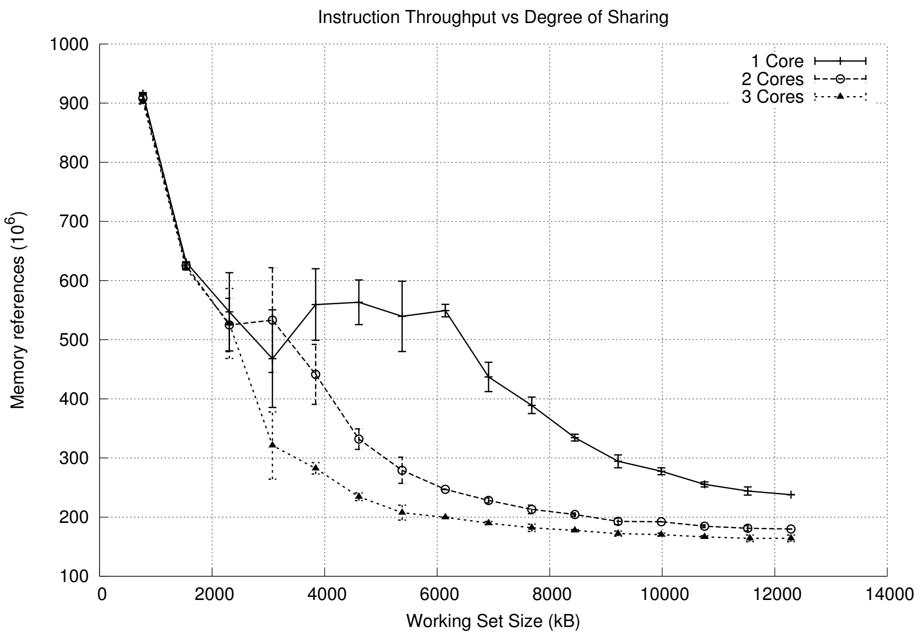


Figure 2.5: Instruction throughput under a shared L3 cache

accelerated communication and encourage thread-level parallelism in software, contemporary CMP architectures share levels of the high-speed cache memory. Sharing memory resources at the hardware level is not a new concept however. Symmetric-Multi-Processor (SMP) architectures provide multiple hardware threads of execution via discrete CPU units in a common motherboard. Distinct memory pools may be attached to each CPU, creating Non-Uniform-Memory-Access behaviour (NUMA). While each CPU has its own set of CPU caches and a local bank of main memory, they may also access each others' main memory.

With the advent of Chip-Multi-Processor CPUs, the level of shared memory is shifted from the main memory to the caching level of the CPU. A motivation behind sharing cache levels is the speed at which this memory is accessed. The shared memory may be exploited for communication amongst cores, or for faster data access among cooperating tasks. Additionally, indirectly cached instructions are also shared amongst CPU cores, providing accelerated execution for highly reused code paths (such as shared libraries).

However, this layer of sharing introduces a new resource which is contented for. Time-multiplexed applications as well as spatially-multiplexed simultaneous threads contend for cache storage. While the access latency of the L3 cache is much lower then that of DRAM, the effective hit-rate on a shared cache is lower. This has a potentially large impact on run-time performance as is shown in Figure 2.5, which compares the memory instruction throughput of a microbenchmark application. The applied workload is a sequence of random memory read operations and is described in detail throughout Section 4. Figure 2.5 compares the decline in instruction throughput due to varying degrees of cache-sharing. Additionally, the Figure depicts 95% confidence intervals over 20 repeated trials for each data point shown. The utilized CPU cache provides 6 MB of L3 cache space, which is visually deducable due to the exponential decline in memory references beyond the 6 MB working set size while using only a single core. As the degree of sharing is increased to two cores, executing a memory access pattern drawn from the same distribution, the effective cache size is halved. This is made evident by the decline in memory reference instructions beyond the 3 MB working set size, indicating that the other 3 MB are consumed by the competing thread. As the microbenchmark is designed to be memory bound (it executes one memory reference every 6 complex instructions), cache misses introduce observable variation in the instruction throughput. Additionally, the problem continues to scale as the degree of sharing is increased to three cores. At this point, the effective cache size has been reduced to 2 MB.

Considering Amdahl's law [2], lowering the point of memory contention has profound performance impacts. In general, the principle of locality dictates that lower levels of the memory hierarchy are exercised more frequently. While this is workload dependent, some level of locality is naturally present in application software instruction streams. By shifting the point of contention (due to multiple threads of execution accessing the same cache memory) to the more frequently accessed levels of the memory hierarchy, more CPU cycles are potentially wasted on cache-misses. Avoiding cache conflicts in CMP architectures is thus an ongoing research topic [7] [48] [49].

2.2 Performance Counters

Hardware performance counters (HPCs), also called Performance Monitoring Units (PMUs), are architectural add-ons, providing a means of measuring the occurrences of predefined hardware events. Their implementation manifests as a set of control and counter registers, used to setup and read out hardware events respectively. Hardware events refer to the occurence of trackable hardware operations, such as cache misses or requests for memory translation. Modern CPUs implement a small set of HPCs per core and another small set for global, fixed events. The original usage case of HPCs comes at the design time of a new CPU chip, motherboard or other peripheral. Hardware designers utilize HPCs to characterize performance bottlenecks at run-time, on real hardware as opposed to simulations. Some of the most useful events measurable with a performance counter infrastructure are interconnect bandwidth utilization, memory performance and instruction throughput.

While hardware manufacturers have had access to these counters for some time, it is much more recent that operating system designers have started to utilize them. This is mainly due to hardware vendor's poorly documented HPC features and usage. Even today, no information is published (by vendors) related to HPC usage overhead. Application code profiling tools such as AMD's Code Analyst [19], OProfile [35] or Perf [20] rely on the availability of HPCs to perform in-depth software performance analysis. The usage of HPCs is now at a critical level, which has prompted the Linux kernel development community to include dedicated kernel level support and interfaces for performance counters (as of Linux 2.6.32). This support is an attempt at broadening the usability of performance counters and allow user space applications to leverage their utility. Most shared CPU cache optimizations are based on run-time performance counter information, such as cache miss-rates and instruction throughput per cycle. This thesis also makes heavy use of HPCs.

Hardware performance counters are still an emerging optimization tool. Particularly academics seek support for HPCs, as their usage allows hardware and software research to gain deeper insight into the underlying hardware performance, without requiring concrete knowledge of industry secrets or hardware implementation details. Much of the work done via simulation can now be performed on real hardware without losing information or including simplifying assumptions. Quite to the contrary, running experiments on real hardware allows researchers to utilize the plethora of existing software which would not be available on simulated hardware.

2.2.1 Multiplexing

At present, the support for hardware performance counters is still limited. Prominent CPU hardware vendors such as Intel, AMD, ARM, and IBM only allow for a small set of HPCs (on the order of 4-6 per core) [1]. These small numbers severely hinder accurate information acquisition. For example: IBM's Cycles per Instruction (CPI) breakdown is a systematic study of hardware performance under a given workload [22]. A complete CPI breakdown requires 16 events to be measured simultaneously, an unachievable task on any currently available architecture. To make matters more complicated, the available counter registers are not identical. Every counter register is only capable of measuring a subset of all available events (true for Power, Intel, ARM). Furthermore, some subset of events are mutually exclusive in the sense that only one of the events may be measured at any given time. For instance: On AMD CPUs of the family 0x10, only 1 north-bridge event may be measured at any given time [1].

In order to measure a larger number of events, as well as perform CPI breakdown type studies, simple round-robin multiplexing is used. Given a set of events, performance counters are allocated to a subset of the requested events. The performance counters are interrupted at a set frequency, the measured event occurrences are read and temporarily stored, and the counter's registers are reconfigured with a different set of performance events.

The resulting measurements are scaled up (via linear interpolation) to approximate their true value. While work has been done to make these approximations as accurate as possible, most performance counter libraries still utilize the basic round-robin mechanism to enable multiplexing [37]. The overhead and inaccuracies introduced by multiplexing should be of concern to anyone wanting to perform a detailed machine level code analysis.

2.3 Cache Performance Models

2.3.1 Working Set Model

The working set model describes the minimum amount of data that must be resident in memory to ensure efficient execution of an application [16]. The model is a fundamental effort in describing an operating system's paging behaviour and reasons about the operating characteristics of a virtual memory system [46]. Historically, the model has been used to allocate memory among processes, which are time-sharing the CPU. By counting the number of actively referenced virtual memory pages of a process during a given time quantum, the operating system infers the working set of this application. As the working set grows and shrinks, it is the operating system's responsibility to decide how many frames of a given process should remain in memory. For example: a process with a small workin set will likely need less frames present in main memory, compared to a process with a large working set.

Consider a time interval of length r and a set of memory pages W which are accessed during the time quantum. The size of the working set then is defined as the cardinality of W. More precisely, let $W(t, r)$ be the working set of an application at time t spanning time-interval r ($[t - r, t]$). This is represented in Figure 2.6. As time proceeds, the working set may contain a different set of pages depending on the amount of data actively used by a process.

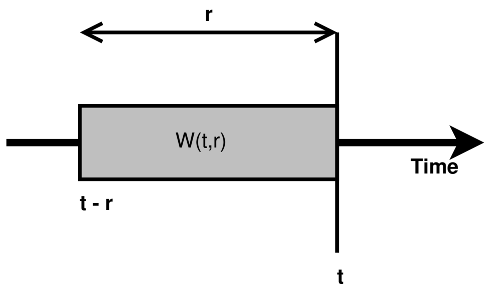


Figure 2.6: Visual representation of a working set W(t,r)

With the exponential increase in memory size over the last decade, the model has become less relevant as many workloads fit into main memory in their entirety. The concept of a working set is still used to describe an application's memory footprint as a whole.

Since the working set refers to the actively used set of pages for a given process, it also partially characterizes the demand on CPU caches. As the set of required pages increases, so does the probability of exceeding the cache's capacity or evicting another processes cache contents. The term *working set* will henceforth be used to describe sizes of workloads in experiments.

2.3.2 Stochastic Cache Model

In order to understand and validate the behaviour of any benchmark attempting to profile memory characteristics, it is worth gathering insight into the theoretical hit-rate characteristics of a CPU cache when subjected to no conflict misses. Access patterns play a crucial role in modeling the CPU cache hit-rate performance, since the memory access distribution determines which cachelines are exercised. Since memory access patterns are rarely known in advance, they are assumed to be drawn from well known distribution profiles. This approximation to the CPU's cache accesses results in stochastic models which characterize the caches utilization. Assuming a uniform access probability to every cache line, how does a workloads's size impact the cache hit-rates? As a working set changes in size, the actively referenced data may no longer fit into the current cache level, resulting in cache hits from higher levels of the cache hierarchy.

Consider the first cache level (L1) and a cache size of C_1. While the working set is less than or equal to the cache size, one would expect every cache access to be a hit. Assuming that no other time-multiplexed application executes instructions on the same cache, every memory reference which results in loading data into the L1 cache will remain, resulting in cache-hits for subsequent accesses to the same data. Once the working set exceeds C_1, the probability of a cache-hit follows a geometric series of the form $\frac{C_1}{x}$. That is to say that the probability of a cache hit is equal to the fraction of accessed data which fits into the L1 cache, namely C_1 out of x. Higher cache levels follow a similar pattern. The L2 cache is expected to incur no cache misses or hits while the working set is below size C_1. As the working set size increases and exceeds C_1, every miss in L1 should be served from L2 so long as the working set fits into C_2. Finally, once data no longer fits into L2, the probability of missing in L2 is the probability of missing in the L1 cache and missing in the L2 cache combined.

The following lists the equations used for Figure 2.9 and describe the stochastic behaviour of CPU cache behaviour under uniform random memory access patterns.
Let D be a discrete random variable which maps to the set L1 Hit, L2 Hit, L3 Hit. These event outcomes are mutually exclusive.
Let X be a parameter variable of working set size (in kB).
Let C_1,C_2,C_3 be L1 - L3 cache sizes respectively (in kB).
Let P(D=L1 hit | X=x) be the probability of a L1 cache hit, given a working set size of x.
Let P(D=L2 hit | X=x) be the probability of a L2 cache hit, given a working set size of x.
Let P(D=L3 hit | X=x) be the probability of a L3 cache hit, given a working set size of x.
Note that the L3 cache hit equation is derived in three steps, all of which are shown below.

$$P(\text{L1 hit}|X=x) = \begin{cases} 1 & \text{if } x \le C_1 \\ \frac{C_1}{x} & \text{if } x > C_1 \end{cases}$$

$$P(\text{L2 hit}|X=x) = \begin{cases} 0 & \text{if } x \le C_1 \\ 1 - \frac{C_1}{x} & \text{if } C_1 < x \le C_2 \\ (1 - \frac{C_1}{x}) * \frac{C_2}{x} & \text{if } x > C_2 \end{cases}$$

$$P(\text{L3 hit}|X=x) = \begin{cases} 0 & \text{if } x \le C_2 \\ P(\text{L1 miss AND L2 miss}) & \text{if } C_2 < x \le C_3 \\ P(\text{L1 miss AND L2 miss}) * \frac{C_3}{x} & \text{if } x > C_3 \end{cases}$$

$$P(\text{L3 hit}|X=x) = \begin{cases} 0 & \text{if } x \le C_2 \\ (1 - (P(\text{L1 hit}|X=x) + P(\text{L2 hit}|X=x))) & \text{if } C_2 < x \le C_3 \\ (1 - (P(\text{L1 hit}|X=x) + P(\text{L2 hit}|X=x))) * \frac{C_3}{x} & \text{if } x > C_3 \end{cases}$$

$$P(\text{L3 hit}|X=x) = \begin{cases} 0 & \text{if } x \le C_2 \\ (1 - (\frac{C_1}{x} + (1 - \frac{C_1}{x}) * \frac{C_2}{x})) & \text{if } C_2 < x \le C_3 \\ (1 - (\frac{C_1}{x} + (1 - \frac{C_1}{x}) * \frac{C_2}{x})) * \frac{C_3}{x} & \text{if } x > C_3 \end{cases}$$

Figures 2.7, 2.8 and 2.9 plot the theoretically expected cache behaviour. As the working set size increases and eclipses the boundaries of cache level A, the probability of a cache-hit within level A decreases at an inverse exponential rate. Meanwhile, a symmetric probability increase occurs in cache level A+1.

The model reflects the best case performance of a cache. This includes the assumption that the working set is evenly utilized by the application. While such an assumption is unrealistic for real applications, it is still possible to construct synthetic workloads which exhibit these memory access characteristics (Section 5).

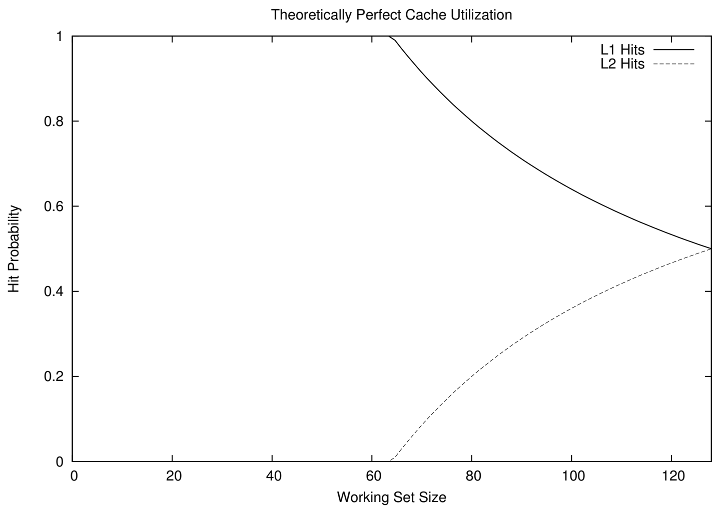


Figure 2.7: Theoretical cache hit rates over the L1 and L2 cache

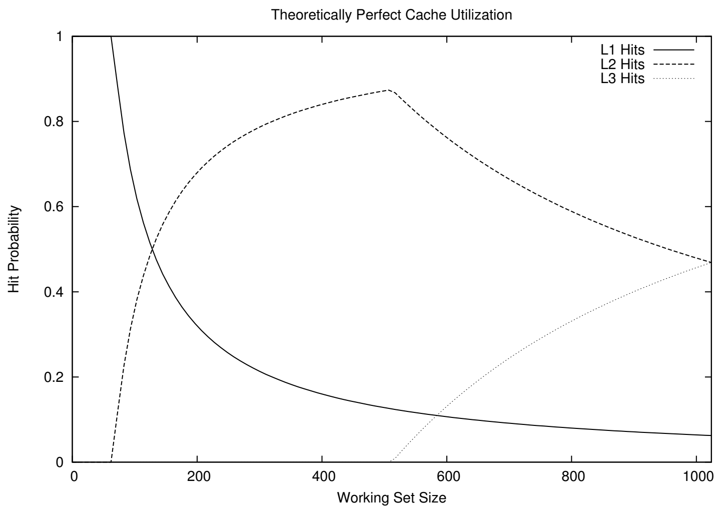


Figure 2.8: Theoretical cache hit rates over the L2 and L3 cache

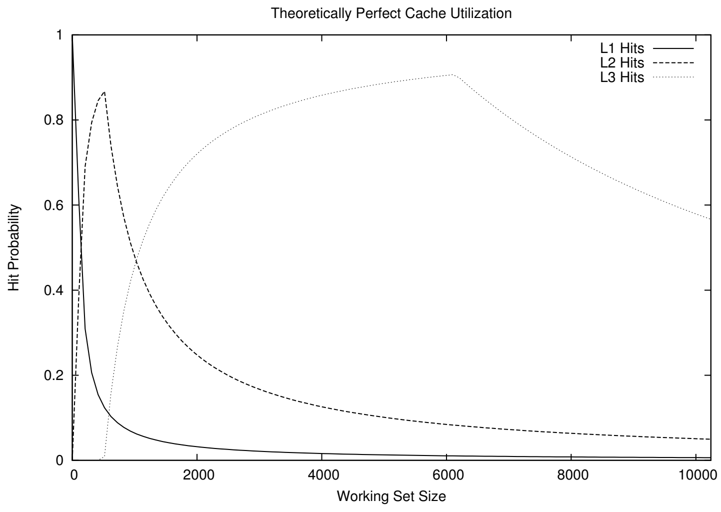


Figure 2.9: Theoretical hit rates over the L3 cache and main memory

2.3.3 Stack Distance Profile

Stack Distance Profiles (SDP) are a class of empirical models, aiming to capture the temporal data-reuse behaviour of an application [10] [11] [17] [39] [47]. In terms of CPU cache performance, SDP may be used to approximate cache hits and predict when an access will result in a cache miss. This insight has been used as the foundation for many cache aware scheduling optimizations [7] [34].

Recall that each cache set within a set-associative cache is itself, fully associative. That is, after determining which cache set a particular request is served from, every cache line within that set may contain the data. Additionally, associative caches require the use of replacement heuristics, in order to determine which stored cache line to evict in the event of a capacity miss. SDP models this behaviour by assuming the stack algorithm "Least Recently Used". Every cache line is placed in order of access recency, forming a stack containing the most recently used cache line at the top and least recently used cache line at the bottom. When determining which cache line to evict, the algorithm always chooses the least recently used cache line. Additionally, on each cache hit within a cache set, the accessed cache line is moved from its current position in the LRU stack to the top. In this manner, the temporal access order of the items within the cache is maintained.

In order to profile the temporal re-use behaviour of an application, SDP samples the distribution of LRU levels from which memory requests have been served. As such, SDP requires the inclusion of A+1 counters for every cache set, where A is the associativity of the cache in question. The A+1 counters, labelled $C_{\{n|1 \leq n \leq A+1\}}$, represent cache hits which where served from position n of the LRU stack (1 being the most recently and A being the least recently used). Cache accesses which result in a cache miss are counted by the special counter C_{A+1}. On every cache access, regardless of hit or miss, a counter is incremented, measuring the LRU level used.

A hypothetical histogram is shown in Figure 2.10. The graph demonstrates the principle of locality: the most recently used piece of data is the most likely to be re-used in the next memory access. This is evident from the large number of memory requests served from the top of the LRU stack, corresponding to counter C_1. A decline in hits is expected across the lower levels of the LRU stack, as data becomes less relevant to the computation at hand. Since the amount of data which exhibits temporal-locality properties is often small, the cache misses (C_{Miss}) make up a significant portion of the cache accesses. The figure depicts a balanced workload, where some temporal re-use exists. On the extremes, there exists applications with no temporal re-use such as streaming workloads, and tight loop operations which touch very little data and thus display a high degree of re-use.

While stack distance profiles grant insight into application locality behaviour, the model has significant shortcomings and is often unrealistic to obtain and use. The construct only models re-use behaviour for fully associative caches, and by extension a single cache set. It has yet to be shown how to extend the model across an entire cache without loss of

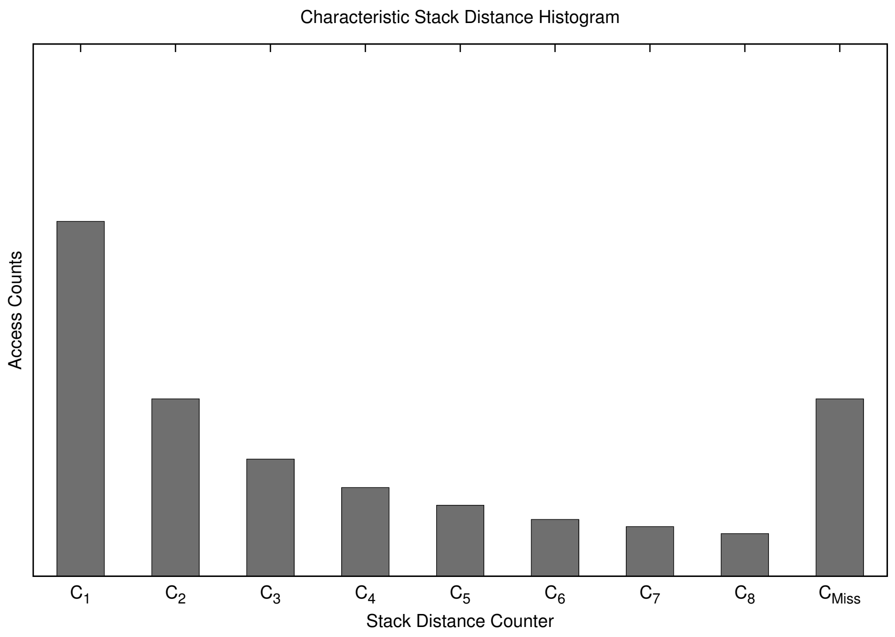


Figure 2.10: SDP displaying principle of spatial locality

generality. Since every cache set requires a separate SDP analysis, it is unclear how to interpret the impact of each cache set on the overall application miss-rate. Are all of the cache sets to be considered? Are only the cache sets which have been touched by the application to be evaluated? What about the re-use of cache lines by other applications? To make matters worse, it is assumed that an LRU replacement algorithm is employed within each cache set. Such an assumption is incorrect with respect to every general purpose CPU on the market. Implementing an LRU algorithm at run-time is prohibitively expensive for large associative caches and is commonly substituded with a pseudo LRU policy [23]. Ghasemzadeh H. et al. show a divergence of up to 17% from the true LRU profile, greatly reducing SDP's level of accuracy.

At the time of this writing, SDP has only been implemented in simulation. Methods for estimating the SDP exist and are based on commodity hardware counters. Xu et al. argue that a sufficiently accurate SDP may be generated by only considering cache miss-rates [12]. In their work, a "cach-hog" application is run simultaneously with a to-be-profiled application in a CMP environment. The "cache-hog" is designed to consume an exact amount of cache memory, leaving the remaining cache space to the tested application. By measuring the miss-rate of the tested application with varying levels of available CPU cache, the stack distance profile is estimated. However, the work does not consider the impact of memory layout and over-estimation of cache-misses. Additionally, these techniques still do not account for set-associative caches. SDP approximations generated by cache-partitioning schemes only apply to fully associative caches and by extension to a single cache set within a set-associative cache. This thesis analysis the degree of divergence between SDPs which are observed by hardware and those that are estimated in software, attributing the difference to the neglected impact of the virtual memory system.

2.3.4 Circular Sequence Profile

Another method of gathering information about the locality behaviour of an application is provided by a circular sequence profile (CSP) [25]. CSPs were first formalized by Chandra et al. as an extension to stack distance profiling [17]. A circular sequence is defined as a sequence of distinct memory accesses, with the first and last access occurring on the same memory address. That is, every memory access to address x_i in a sequence of n accesses satisfies $\forall i, j, i \neq j : x_i \neq x_j$ if $i, j \neq 1, n$ and $x_i = x_j$ otherwise. In other words, a circular sequence profile measures the number of occurrences of d distinct memory accesses in n total accesses. Such a profile allows for the estimation of cache miss-rates. Consider a cache with associativity A and a circular sequence profile which has been implemented at the cache set level. The cache miss-rate may be estimated as the number of circular sequences which access $A + 1$ or more distinct cache lines within any number of memory accesses. Thus, a CSP contains information about the re-use distance of any particular main memory line of data.

In contrast to stack distance profiling, implementing the machinery required to obtain CSPs in hardware would be more difficult. While stack distance profiles measure the re-use distance of a cache line relative to the total number of memory accesses, a circular sequence profile describes the re-use distance for any n memory accesses. To gather this information in hardware, a counter must be added to each main memory line, as is the case for SDP and cache lines. Additionally, the inclusion of a table is required in order to keep track of each circular sequence. Once this information has been gathered, Chandra. et al. provide an inductive probability model, capable of predicting cache miss-rates within 5% of error. Furthermore, this work has been expanded to combine multiple CSPs and infer the combined cache miss-rate when two threads share the cache.

As circular sequence profiling is a refinement of the stack distance profiling methodology, it suffers from the same shortcomings while also introducing new issues. Just as is the case with SDP, CSPs do not account for set-associative caches. CSP assumes an even distribution of memory accesses to all cache sets, while simultaneously discounting the effects of virtual memory mappings. Given no assumptions about the virtual-to-physical memory layout an operating system provides, it is insufficient to claim that an application evenly distributes it's data acress the available CPU caches. This thesis corrects these assumptions and contratst the resulting memory re-use profiles to existing approaches.

Chapter 3

Motivation

3.1 Virtual Versus Physical Addressing

When hardware engineers design new cache memory hierarchies, a fundamental question that needs to be answered is whether to use a virtual or physical address to index and tag the caches. As a first approach one might naively choose to use virtual addresses, as application programs use virtual addresses directly and thus directly identify the sought after memory location with respect to the current address space. Furthermore, the compiler is able to perform locality aware optimizations, as the memory layout of the stack, data and code are known. However, the utilization of virtual addresses leads to multiple implementation issues. Since virtual addresses may be remapped to point to different physical frames, a cache may become inconsistent if a remapping occurs but the cache entry is not flushed [18]. Likewise, in a multi-programming environment, the same virtual address may reference two different physical frames, again requiring expensive cache flushes on every address space switch. With the advent of multi-CPU and multi-core CPU chips, the cache coherence protocols further complicate cache tagging as physical addresses are required [3].

By using physical indexing and tagging, modern CPUs are able to implement cache-coherence protocols. A physically tagged cache also solves the aliasing issue, where multiple identical virtual pages refer to different physical cache locations [30]. The physical tag serves as a unique identifier among cache lines which may have the same virtual address. However, using physical addresses requires the CPU's memory management unit (MMU) to translate every memory request prior to checking the cache. The involvement of the MMU introduces additional overhead to every memory request, which is not present in a purely virtual cache.

3.2 Cache Utilization

Besides the access latency to a cache, physical indexing and tagging introduces another, more subtle performance issue of cache utilization. user space applications strictly interact with virtual memory addresses, leaving the operating system in charge of managing the virtual-to-physical mappings on their behalf. In a set-associative cache, a subset of the memory address's bits are used to select a cache set to query. Since a modern CPU's cache implementation uses the bits of the physical memory address, the operating system is responsible for determining which cache sets are covered by a single page. The procedure of allocating a virtual page in a cache-aware manner has been well studied and partially forms the incentive behind page colouring (an optimization technique discussed in Section 3.3).

Kessler and Hill were among the first to study the performance impact of virtual page placement algorithms on physically indexed caches [32]. In their model, Kessler and Hill describe a cache to be partioned into a set of page sized bins. Each bin covers a subset of all cache sets within one cache level and holds as many frames as the degree of associativity. When the number of pages mapped into the bin exceed its limit, a conflict at the cache-level is more likely to occur, resulting in a cache conflict miss.

Oversubscribing to any particular page bin does not only under-utilize the CPU cache, but also impacts the cache hit-rate. In a cache with characteristics as shown in Figure 3.1, multiple frames are required to cover the cache. Figure 3.1 utilizes 4 kB frames, 64 byte cache lines and 256 cache sets. The 2 bits of overlap between the frame number and set is historically called the "bin-id" and identifies the page bin to which the frame is mapped to. Thus, the number of page bins is four for this cache, meaning that four frame allocations are required to cover each cache sets once. The increased likelihood of a cache miss is a result of the linear distribution of cache lines-to-cache sets within a frame. Since a single frame covers a subset of all available cache sets (given by the overlap in the frame offset bits and cache sets bits), each cache line within a page maps into a pre-determined range of

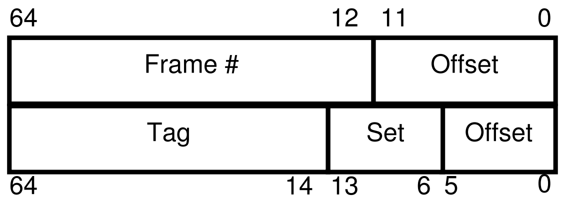


Figure 3.1: Addressline interpretation during paging and cache lookup

cache sets. As page bins become oversubscribed, so do the underlying cache sets. However, oversubscription to any cache set results in some data being evicted, requiring a cache-miss (capacity or conflict miss of cache set) if this evicted data is accessed again. If all physical frames are allocated in a contiguous manner, perfect cache coverage would be achieved.

The degree of cache-level conflicts due to paging may be formalized mathematically. Suppose an operating system performs uniform random frame allocations. That is to say, that on a request for memory, the operating system will select a frame which maps into any page sized bin with equal probability. Suppose that the cache is of size C, with page size S, associativity A and B number of page bins. The probability that n out of N pages are mapped to the same bin is given by a binomial distribution with success probability $p = \frac{1}{B}$.

$$P(\text{n pages in one bin}) = \binom{N}{n}(p)^n(1-p)^{N-n} \tag{3.1}$$

By applying the probability density function to all page bins uniformly, the number of page conflicts may be estimated as:

$$K_{avg} = B\sum_{u=A+1}^{N}(u-A)P(u) \tag{3.2}$$

The minimum number of conflicts (capacity misses), may be calculated by considering a perfect page to bin distribution. In a perfect distribution no page conflicts occur until all bins have been filled. The minimum number of conflicts is given by:

$$K_{min} = \max(0, N - \frac{C}{S})$$

Finally, the difference of $K_{avg} - K_{min}$ indicates how many conflicts above the minimum are expected to occur under a uniform random frame allocation. As the number and size of bins is finite (proportional to the cache size), the capacity of each bin will eventually be exceeded. The minimun number of bin-level conflicts is given by the theoretical distribution of frames to bins, such that each bin is even utilized. For example: In a system with eight bins, with each bin holding eight frames, it would require 64 frames of memory before any conflicts occur under an even bin utilization scheme. Figure 3.2 visualizes the difference $K_{avg} - K_{min}$, for a cache which has eight page bins and is eight way set-associative. The figure shows an increase in the expected bin-level frame conflicts until the memory allocation reaches the size of the cache. At this point the number of expected conflicts with respect to an even bin utilization decreases, as the minumum number of conflicts increases.

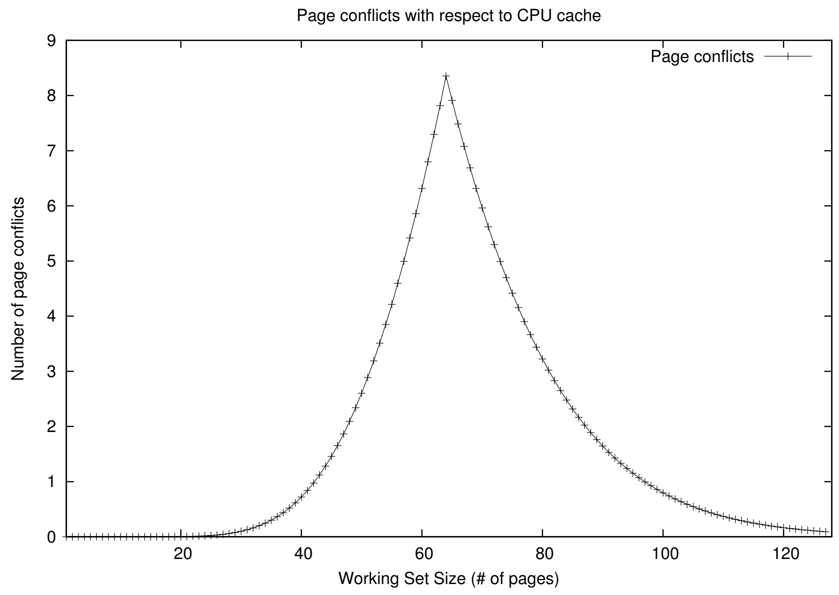


Figure 3.2: Expected page conflicts with respect to page bins

Note that a binomial distribution assumes that frames are drawn with replacement, which is not true. Once a frame is allocated, it is removed from the pool of potential memory allocations. Fortunatly, contemporary computer systems feature many gigabytes of main memory. The binomial distribution serves as an approximation to the actual hypergeometric distribution of single frame memory allocations. Additionally, this approximation significantly simplifies the computation of a theoretical memory allocation distribution.

In comparison, Figure 3.3 illustrates cache under-utilization due to sub-optimal page placement on Linux 3.3. Each data point in the figure indicates the number of missplaced pages which are causing cache conflict misses that could have been prevented. In the worst case, over 10 pages on average are missplaced, reducing the effective cache-size by over 600 kB and introducing unnecessary caching conflicts. The experiment allocates a pool of memory via the *malloc* interface provided by the standard C library. Upon allocating and referencing every cache line covered by the memory pool, the frame allocation of the operating system is queried to determine the frame distribution among cache-bins. This information is compared with a perfectly even distribution of frames and plotted in Figure 3.3, with 20 repeated experiments representing each datapoint. The plotted information includes 95 % confidence intervals.

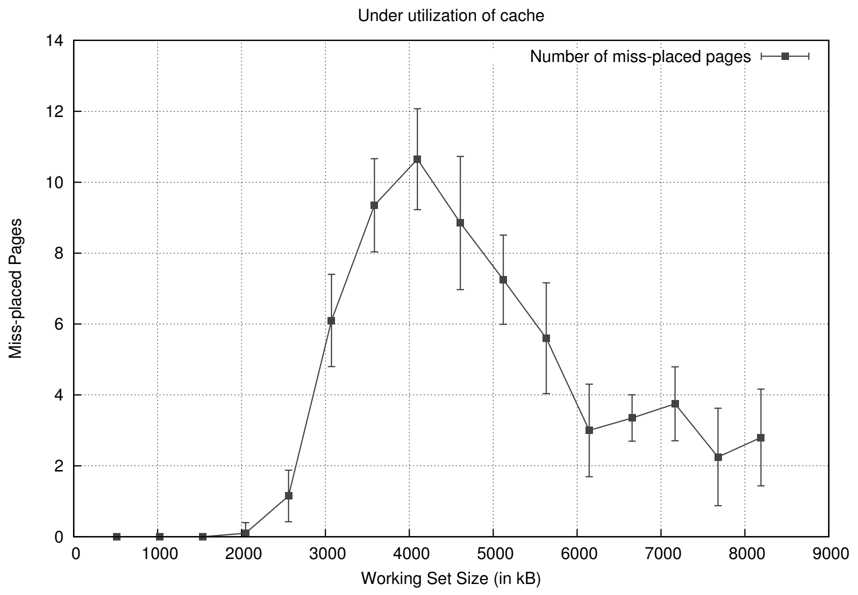


Figure 3.3: Under utilization of L3 cache on IBM Power 7, Linux 3.3

The results of the study indicate that the Linux operating system does not perform cache-aware frame allocation. At the critical working set size of 4 MB (which corresponds the L3 cache size in this experiment), the operating system allocates 10 frames to already filled cache-bins. Furthermore, this over-subscription of cache-bins results in the under-utilization of caches, meaning that each of the 10 frames could have been mapped into another cache-bin which is not at capacity.

3.3 Page Colouring

Page colouring was first implemented in the MIPS operating system(RISC/os) [50]. At the time, CPU caches were physically indexed and tagged leading to the binning effect described by Kessler et al. [32]. Figure 3.4 illustrates the discrepancy between virtual memory, physical frames and how these relate to the CPU cache. Consider a 7-page memory allocation in an application's virtual address space. Upon request for memory, the operating system will attempt to allocate physical memory frames and create the required page-table mappings. As the operating system allocates and frees frames, the physical address space becomes fragmented and contiguous allocation becomes increasingly unlikely. The 7-page memory allocation could result in a physical frame allotment as depicted in

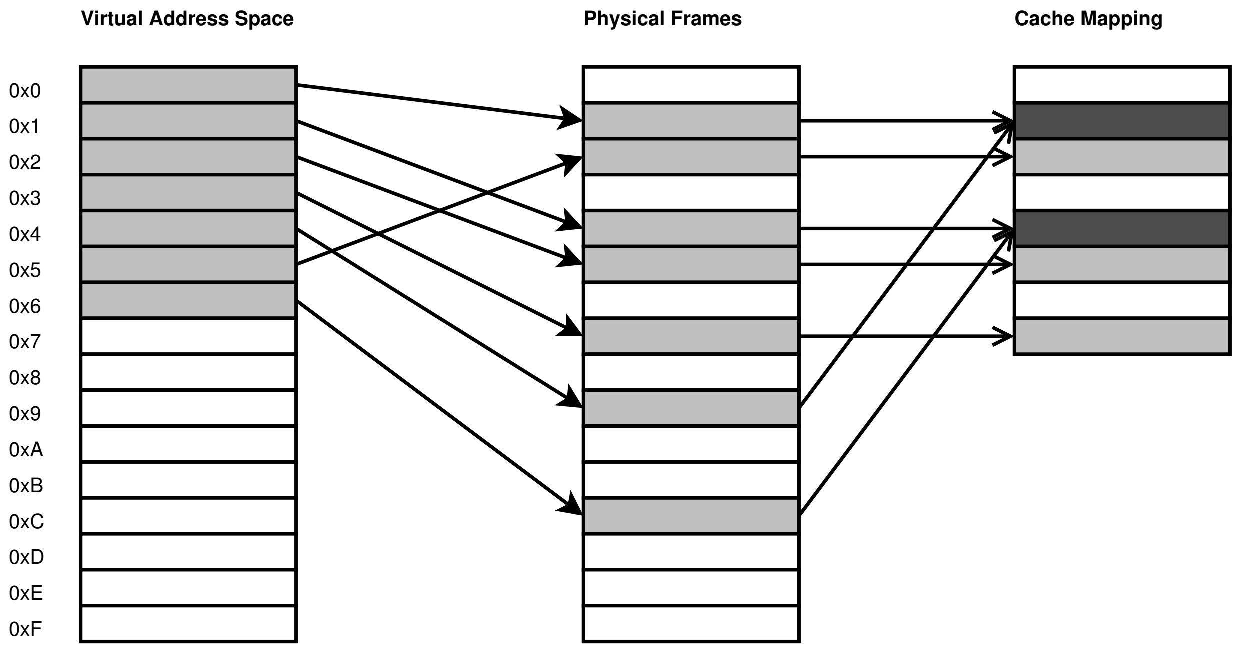


Figure 3.4: 7 virtual pages mapped to physical memory. The corresponding physically indexed cache utilization is visualized on the far right. The dark area corresponds to cache conflicts (at the page level)

Figure 3.4. The far right of the figure is the frame to cache mapping (bin mapping). The dark-shaded cells represent bins which have more than 1 page mapped to it, leading to caching conflicts for a direct mapped cache. Furthermore, the cache is under-utilized, as the two double mappings could have been placed in the empty page bins.

Page colouring assigns every bin a colour (a unique ID) and attempts to allocate memory of as many different colours as possible. In this way, the memory allocator spreads the virtual pages across the CPU's cache. The initial colouring is done at boot-time, allowing the operating system to allocate its own data-structures in a cache-aware manner. When a memory request is received, the memory allocator consults the current colour distribution and assigns the least frequently used active colour to the virtual address region (described as Bin Hopping by [32]).

Other allocation methods consider the frequency of usage of a particular colour and how many free items of a colour still exist. Such a metric accounts for the uneven distribution of available colours at boot-time. However, smarter colouring algorithms tend to incur higher overhead [36]. The consequences for depleting a colour are dire, as it involves re-colouring of existing pages. Depending on the attainable performance improvements, page re-colouring may involve deep-copying of frames or simply evicting existing frames.

This thesis utilizes the Bin Hopping approach in order to quantify the effects on memory instruction throughput due to cache-unaware frame allocation. However, unlike Kessler et al., the frame distribution is implemented entirely in user space, requiring no changes to the operating system.

3.3.1 Usage of Page Colouring

From it's initial inception in the 1980s to present day multi-core CPUs, page colouring has been used in a variety of performance optimization approaches. Today page colouring is most commonly used to partition CPU caches such as to give partitions of the cache to different system components and avoid cache pollution [4] [26] [29]. Cache pollution occurs when a hot piece of data is evicted and data with a low degree of reuse is inserted in its place. This results in sub-optimal utilization of the cache space, since caches provide the best performance improvements when data is frequently served from them. To combat cache pollution, the cache is partitioned into sections of high re-use and low-reuse. Page colouring allows whole pages of data to be carefully positioned within the cache, providing the mechanism with which pages are placed into various partitions. Some studies have shown how hot data may be identified online and offline and illustrated performance improvements of up to 50% for some average cases [26].

Others have successfully studied page colouring with the aim of reducing benchmark variability across multiple runs. Hocko and Kalibera study the impact of page colouring and bin hopping on a wide breadth of custom benchmarks and concluded that Linux's (kernel version 2.6.22.6) default page allocation scheme outperforms page colouring and bin hopping techniques when run sufficiently long enough [28]. The work further concludes that page colouring leads to the most predictable execution times and is thus well suited for time critical applications or operating systems. Performance is measured in terms of response time of a given benchmark, which is its completion time. However, the study does not attribute the gain in determinism to better cache utilization as the characteristics of its workload are far too complex and no hardware performance counters were utilized. The authors do not claim intimate knowledge of the workload, but ran C# based workloads with the hope of drawing a connection to real world applications. This thesis builds on these initial observations and draws a clear relation between instruction throughput variability versus memory layout.

Finally, page colouring may be used in contemporary multi-core CPU architectures to provide selective cache-sharing and to facilitate smarter simultaneous workload scheduling [4] [13]. A shared CPU cache has been shown to suffer excessive conflict misses when subjected to certain combinations of workloads [26]. Page colouring allows the operating system to partition competing applications with respect to the shared cache. By scaling the number of pages allocated to each application, it is possible to control an application's performance as a function of cache size. The microbenchmark suite implemented as part of this thesis is capable of mapping the worst case performance impacts to instruction throughput due to shared cache access by multiple threads of execution. Some preliminary results are shown in Section 2.1.3.

3.3.2 Implementations of Page Colouring

The impact of under-utilized CPU caches has been studied with industry standard benchmark suites via simulation and extensions to existing operating systems. Marko and Madison investigate the impact of a real page colouring implementation on Linux 2.0.33 [36]. The reported experiments showcase a 30% performance increase (measured in wall-clock time) of 12 X-Window based applications. This work is the first attempting to implement real software page colouring, including page re-colouring methods. However, the highly intrusive methodology utilized by Marko and Madison requires kernel re-compilation. The work presented as part of this thesis does not only implement a user space solution to measuring cache-utilization and its relation on instruction throughput, it also adds the additional support for hardware performance counters. Neither of these features are present in prior work.

Bahadur et al. study the effects of careful page placement on Linux [5]. The study contrasts bin hopping, Linux's slab allocation, and a random frame allocation method. The authors conclude that bin hopping does not significantly boost performance over the Linux slab allocator even though a 67% decrease in cache misses is observed. Furthermore, the authors claim that the slab allocator approximates physically contiguous allocations. While this is true for a system with low uptime, it is not true in general. Modern Linux (3.x) implementations actively seek to re-use as many fragmented frames before splitting a new block [8]. Furthermore, the cycles per instruction reduction data is outdated, since a significant reduction in last level cache misses greatly impacts a CPU's instruction throughput (see Chapter 5). The work further draws a relation to Amdahl's law [2], raising concerns of not optimizing the common case. Even though bin hopping realizes a large reduction in cache misses, only a negligible performance increase is observed. The authors attribute this discrepancy to the small impact on the available cache space when using bin hopping. A single virtual page covers the first level cache many times, thus failing to provide an opportunity for greater performance increases. However, contemporary CPU architectures provide much larger caches then those studied by Bahadur et al. and thus remove the constraints of Amdahl's law.

3.4 Hugepages and HugeTLB

A Translation Lookaside Buffer (TLB) is a CPU architectural component, serving as a cache to the Memory Management Unit (MMU). That is, the TLB stores a subset of a processes page table entries. On every memory access to an address provided by virtual memory, the MMU consults the TLB for a valid page-mapping. In the event of hitting in the TLB, the fetched virtual page to physical frame translation is passed to the MMU in order to resolve the memory request. Missing in the TLB on the other hand results in an expensive memory operation, requiring page table traversal by the hardware or software.

Contemporary general purpose operating systems still utilize the 4 kB default page size. There are two contributing factors for this decision: hardware does not support larger page sizes and legacy kernel components' reliance on 4 kB pages. The former reason is quickly diminishing, with TLBs now capable of storing multiple page sizes at the same time. Kernel and driver components dependence on fixed page sizes is a result of poor software engineering and should be considered an inadequate reason for such component dependence.

Linux's *Hugepages* (specifically called *HugeTLB*), provides the software components required to utilize the set of available virtual page sizes for a given CPU architecture. In particular, *Hugepages* provides a filesystem based solution for obtaining memory areas which are backed by larger virtual pages [24]. To use this component, the end-user is required to setup pools of memory from which to serve memory requests to *Hugepages*. Linux allows the end-user to allocate these large pages either manually, via the "mmap" systemcall interface, or by creating a filesystem entry under *HugeTLB's* mount point. Any data written to the file is backed by a larger virtual page, depending on the initial setup of the system. Recently, Linux implements a feature entitled "Transparent Hugepages", an operating system feature which detects large buffer pools and automatically allocates the underlying memory via HugeTLB [44].

The main motivation behind an increased page size is the reduction in activity in the TLB. Every miss in the TLB requires a costly memory operation to resolve. Misses may be attributed to working sets, which are simply too large to be covered by the TLB. An increase in page size, increases the amount of memory which the TLB is able to cover.

Additionally, larger virtual pages require fewer TLB level page replacements, as long as the working set does not greatly exceed the capacity of the TLB. These factors lead to fewer memory stalls, potentially improving instruction throughput of a CPU core.

A TLB supporting some form of *Hugepages* is commonly capable of translating more memory per TLB-entry. The AMD Opteron's TLB may hold 1024 4-kB page entries and 4 4-MB page entries, resulting in a factor of four more address space coverage. While the 4 kB pages provide a larger degree of flexibility, especially when dealing with small amounts of data, larger pages cover more memory with less overhead. The typical fragmentation issues are of course applicable and require the end-user to carefully choose when to use the *Hugepages* facilities. While the choice is generally workload dependent, there exists few end-user centric applications with memory footprints below several megabytes (considering the average MP3 requires approximately 3 MB).

A secondary benefit of increased page sizes is that each page covers a larger portion of the cache. This effectively reduces the number of bins to which any page may be mapped. For CPUs of the x86 based architectures family, a huge page (2 MB) is often as large or close to the size of the last level cache (L3), thus certainly covering the lower tiers (L1 and L2). This coverage may result in better CPU cache utilization and is explored in Chapter 4. No prior work has taken a systematic approach to understanding what the performance impacts are in isolation.

Chapter 4

Benchmark Evaluation

The primary drive behind the development of a custom micro-benchmark is the desire to intrinsically understand contemporary multi-core CPU cache memory behaviour. The canonical methodology for such a task is to construct a set of workloads, execute them on the CPU, and measure various performance characteristics. However, to draw any conclusions it is imperative to have an in-depth understanding of the utilized workload. While there exist industry standard benchmark suites (i.e., SPEC CPU) for measuring overall CPU performance on "realistic" workloads, drawing a 1-to-1 relation between their workloads and observed performance characteristics can be difficult. Tools such as SPEC CPU are large pieces of software attempting to model computational tasks at a variety of intensity levels. Covering a large breadth of workloads has the downside of additional complexities, which makes reasoning about performance characteristics difficult, if not impossible.

Academic efforts are often based on micro-benchmarks due to their fine-tailored nature and ease of modelling. Small custom benchmarks may be tailored to isolate performance characteristics or to find the best and worst cases of a workload in order to reason about the outlying or speciality cases. The downside to micro-benchmarks is the lack of breadth and real-world applicability.

The efforts described in this work are based on a custom micro-benchmark, titled *Cachetest*, tailored to study caching characteristics. The tool's objectives are as follows:

- Automatic memory access pattern generation.
- Utilize performance counters at low overhead.

- Measure the instruction throughput over a given access pattern.
- Allow for multiple buffer generating methods.
- Evaluate SDP algorithms and compare the resulting profiles to real run-time performance.
- Be portable and extensible.

These objectives along with a means to study the impact on CPU cache performance, due to the layout of memory-access-patterns, are the driving factors behind the development of *Cachetest*. In particular, no work exists that directly profiles the performance variation due to virtual memory allocation. *Cachetest* provides the means to perform machine-level profiling, so as to characterize and verify the expected hit-rates of a cache. Additionally, it is possible to allocate memory pools with a deterministic cache-aware distribution. *Cachetest*'s current workload generation methods deliberately do not reflect real world applications, but instead stress caches so as to determine an operating-system's virtual memory subsystem performance. In contrast, production software and industry standard benchmark suites stress the memory hierarchy as a whole and do not allow for specific targeting of CPU cache levels.

Additionally, the tool is used to generate the necessary data to perform stack distance profiling. SDP is used to model the re-use behaviour of cache lines but is affected by the memory distribution of the studied workload. *Cachetest* provides a translation layer allowing for the comparison of traditional fully-associative SDP analysis with actual contemporary set-associative implementations.

4.1 Metrics

Measuring application performance is done with a variety of metrics, often application specific. Some metrics are only valid in the context of an application, for example query result throughput of a database. A common metric across all computer applications is that of instruction throughput and is defined as the number of retired instructions per unit time ($\frac{\text{retired instructions}}{\text{time period } \Delta}$). As processor speeds vary, and with it the amount of time required to dispatch and retire a single instruction, it is more convenient to express the instruction throughput in terms of cycles spend during executions. In particular, the average number of cycles per retired instruction (CPI) is an architecture agnostic metric and is henceforth used to characterize workload performance.

CPU cache performance is directly related to CPI, as a cache's operating characteristics directly influence the performance of instructions with memory operands. Cache performance is measured in cycles spent to satisfy memory requests (CPI for memory bound instructions). However, this metric may be subdivided into its component types. The time taken for a memory request is approximated by the following equation:

$$\begin{aligned}\text{Access Time} =& P(\text{Cache Hit}) * \text{Cache Access Latency} \\ &+ P(\text{Cache Miss}) * \text{Main Memory Access Latency}\end{aligned} \tag{4.1}$$

Recall that caches provide a probabilistic approach to mitigating the memory wall problem (Section 2.1). The cache access time Equation (4.1) reflects these stochastic characteristics of caches. The importance of cache memory is evident when comparing the "Cache Access Latency" versus "Main Memory Access Latency". Figure 4.1 shows the access times of all levels of memory on an AMD Opteron 2427 CPU. Data for the plot is obtained from the "lmbench", open source, microbenchmark tool [41]. By measuring the time required to satisfy a predefined number of memory references, the microbenchmark calculates the mean access latency to the various levels of the memory hierarchy. This data has been translated to the mean memory access delay in terms of stalled cycles. The diagram indicates four distinct access time levels, those corresponding to L1, L2, L3 and DRAM accesses (respectively from left to right). For this CPU, access latencies are 3, 18, 43 and 135 cycles respectively. The figure also indicates the various cache level sizes: 64 kB, 512 kB and 6 MB respectively.

To measure the cache utilization of an application, miss-rate curves are used. The miss-rate is defined as the amount of cache misses relative to cache accesses. Generally speaking, high cache miss-rates are indicative of workloads exhibiting low degrees of locality, as the re-use distance of cache lines exceed the cache's size. Alternatively, shared caches may experience high miss-rates due to conflict misses that have been introduced by sharing.

Although the *Cachetest* application may be used to measure any hardware events, focus is given to cache hit-rates. To verify the theoretical cache behaviour model in Section 2.3.2 hit-rates of all cache levels are computed and contrasted to CPI.

4.2 Software Architecture

The *Cachetest* benchmark is divided into three components: *Execution Engine*, *Distribution Generator* and *Buffer Allocator*. The entire application is written in C++ and makes heavy use of object oriented design. The *Distribution Generator* and *Buffer Allocator* in

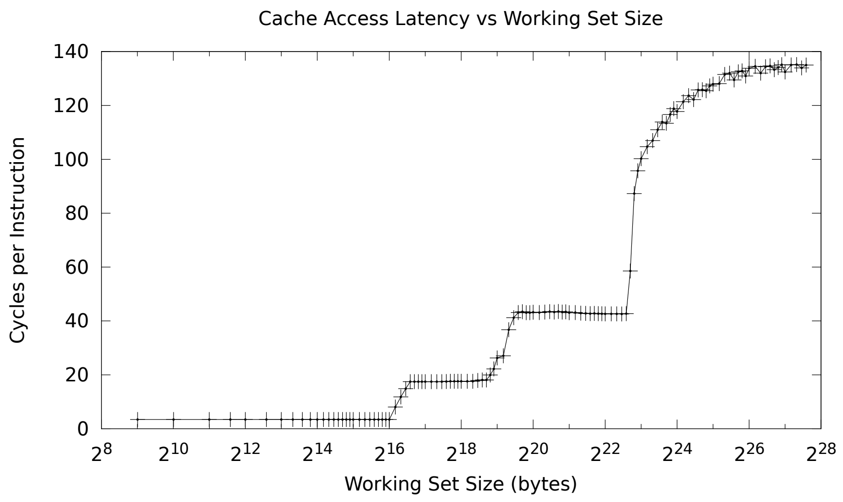


Figure 4.1: Cache access latency vs working set size

particular are designed in such a way as to allow new distribution and allocation methods to be rapidly inserted. The tool also makes use of mostly POSIX compliant system interfaces. While not requiring any additional third party software, some features require the Linux *Sysfs* and Process Filesystem (*procfs*) components [31] [42] in order to collect system information.

At run-time of the benchmark application, the three components work in tandem to configure the experiment and measure memory performance. Given a set of parameters, memory is allocated to serve as the benchmarking target. The task of allocating this memory pool is designated to the *Buffer Allocator* and may be performed in a variety of ways, to investigate the cache performance impacts introduced by virtual memory allocation. It is this buffer which is utilized by the *Execution Engine* during profiling.

Once a suitable buffer has been allocated, the memory access pattern is installed by the *Distribution Generator*. The user's selected distribution is pre-computed and stored inside the buffer to form an array of indexes. Each index refers to the next cache line which is to be accessed at run-time. The chain of memory locations is referred to as a pointer chasing algorithm and provides the basic method for repeatedly accessing a pre-determined memory pattern.

Finally, the initialized buffer is provided to the *Execution Engine* for profiling. The string of memory references is parsed while available PMU hardware counts hardware events and memory accesses. The *Execution Engine* forms the base component of *Cachetest*

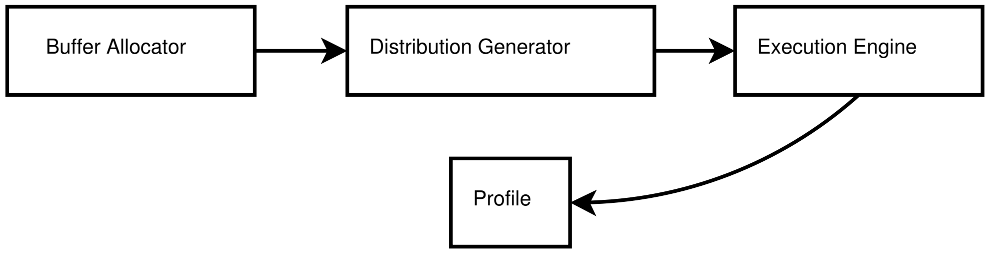


Figure 4.2: The components of *Cachetest*

by parsing option parameters, initializing the buffer and distribution classes and interacting with the PMUs.

4.2.1 Execution Engine

The heart of the benchmark tool is a tight loop of memory read operations over a pre-allocated integer array. Using the array, a pointer chasing algorithm is implemented by combining the base address and a given element within the array.

Algorithm 4.2.1 illustrates the pointer chasing approach used by the loop code. An array of indexes is setup prior to running the experiment. During execution, the next index is read from the array and stored in a register. This read operation is the only measured memory access within *Cachetest*. Additionally, a counter is incremented to reflect this memory access in software. The counter is stored in an available hardware register, so as to not influence the measured memory operations. The loop is executed a fixed number of iterations, or until a set amount of time expires.

Algorithm 4.2.1 *Execution Engine*'s tight loop

```
1: prevIdx = 0
2: while notDone do
3:     prevIdx = buffer[prevIdx]
4:     accessCount += 1
5: end while
```

In order to accurately count the number of memory references performed during execution, the code path in Algorithm 4.2.1 has been optimized to store all auxiliary data in available machine registers. As discussed in Section 2.2, modern hardware only provides a small set of programmable hardware event counters. To measure a greater amount of events, it is useful to infer as many statistics as possible. This reduces the number of events which must be scheduled on the PMU hardware and ultimately lead to more accurate results (increased granularity of time-mulitplexed PMUs). To implement this optimization, the number of read operations performed is counted by the *accessCount* variable, moving a counted hardware event (memory read operations) from the PMU to software. In order to assert high levels of accuracy, it is paramount that only one memory read operation occurs during each loop iteration. Mapping the tight loop's variables to registers is thus a requirement, as only memory accesses due to the pointer chasing algorithm should be measured.

The code path appears deceivingly trivial. However, on x86 based architectures, only four general purpose registers are available (excluding floating point registers, whose access time may be larger). While the more modern x86_64 architecture provides additional registers, their usage is avoided for backward compatibility of the tool and to allow the tool to run on custom operating system platforms (which may not provide the extra registers). The pseudo-code depicted in 4.2.1 already utilizes these four registers: The index register (*prevIdx*), the accessCount, the completion flag(*notDone*), and the buffer pointer used by the array.

The aforementioned variables are required for time-interval based measurements (the default). In this mode, the tight loop and PMU is executed for a fixed amount of time. Timing is accomplished by the operating system's *signal* interface and the POSIX *alarm* method. A signal based approach is used to mitigate pollution of possible PMU measurements, as PMUs are always configured to only measure userspace cycles. Signals are delivered by the kernel and are not measured by PMUs. However, this approach requires that the signal handler be able to manipulate a register-stored variable "notDone". To accomplish this, the fact that a signal delivery is performed synchronously to program execution is exploited. During the context switch to the alarm handler, all general purpose registers are backed up on the user-space application's stack. The signal handler's task is to locate the stack location of the "notDone" variable. Once the variable's contents are located on the stack, it is modified and set to false, thus releasing the main loop. Finally, upon switching back to the *Execution Engine*, registers are repopulated by the stack frame's backup, resulting in a changed exit condition.

An alternate operating mode is a fixed number of memory accesses. While this approach removes the possible signal delivery inaccuracies, it makes reasoning about the potential run-time length more difficult. When running large experiments, it is useful to estimate the running time of an experiment for scheduling purposes. Furthermore, a more severe issue arises when dealing with multiple concurrent benchmarks, as the run-time may vary greatly between competing workloads. In a concurrent experiment, the run-time of each thread/process should overlap 100% so as to simulate memory contention. If the experiment is set to execute over a set amount of memory accesses, and each memory access varies in duration due to variable contention, the desired concurrency effect may shift, invalidating results. A timer-based approach via signals allows multiple processes to be interrupted simultaneously, increasing the overall accuracy of the results.

4.2.2 Buffer Allocator

The *Buffer Allocator* is a distinct component of *Cachetest*, responsible for allocating the buffer array. Allocation may be performed via three methods: The default Linux *malloc* system call, a custom contiguous allocation or via Linux's HugeTLB implementation (HugeTLB is described in Section 3.4). The different allocation methods are necessary in order to study the impact of virtual page placement on CPU cache behaviour. All methods are completely implemented in userspace and only require a proper setup of Linux's Hugepages and access to the Process Filesystem (procfs).

The *malloc* method provides the most direct and conventional approach to allocating a buffer. *Malloc* is supplied by the standard C library (libc) and provides a user- and kernel-space hybrid solution to memory allocation. While malloc requests pages of memory from the operating system, it is libc's duty to manage sub-page allocations. When confronted with memory allocation, Linux's physical memory sub-system uses a slab allocator to serve the request. Slab allocation is aware of various dedicated memory pools, such as DMA (Direct Memory Access) or filesystem caches [8].

To manage fragmentation and allocation of each memory pool, the Slab allocator uses the Buddy algorithm. The algorithm represents its managed memory as a set of memory pools (free-lists), which may be split into smaller or merged into larger segments [33]. At boot time, while no memory is allocated from a particular pool of memory, the Buddy-Algorithm represents the pool of memory as a single segment. The size of this segment is by necessity a multiple of the operating system's virtual page size. Upon reception of a memory request, the algorithm traverses the free-lists in search for a minimum sized segment which satisfies the request. If the smallest available segment exceeds the requested memory allocation by a factor of 2 or more, the segment is split in half and tested for size again. This searching technique is applied recursively until a suitably sized segment has been found.

Allocations performed by the Buddy-Algorithm have the benefit of allocating large pools of physically contiguous memory, as long as the physical address space is not fragmented. However, as memory is claimed and freed, the size of smaller free-lists increase, leaving the physical address space fragmented. As the level of fragmentation rises, large memory requests become backed by non-physically contiguous memory. Malloc thus provides a sufficient and simple method of stressing Linux's page allocation system.

A custom contiguous memory allocation scheme is required if physically contiguous memory is desired. Physical contiguity (or a perfect cache-colouring) provides the best case CPU cache utilization, as has been discussed and experimentally validated in Section 3.3. However, it is not physically contiguous memory that is desired, but some physical memory frames which evenly utilize every cache set. Such an allocation is referred to as "cache-aware" and minimizes the number of cache conflicts due to poor page placement. In order to validate the hypothesis (cache-aware memory allocation is critical), a cache-aware memory allocation is required. Multiple approaches for this exist for Linux, most of which require fundamental changes to the operating system kernel. Page-Colouring is a solution at the memory allocation level, but costly to implement and not portable. For the purposes of this thesis, a portable userspace approach has been devised and implemented in *Cachetest* (portable so long as userspace has access to the page directory).

Algorithm 4.2.2 Custom Memory allocation

```
 1: alloced_buffer = 0
 2: current_cache_index = 0
 3: current_page = malloc all available memory
 4: while (alloced_buffer < required_buffer) do
 5:    frame = translate current_page to PFN via /proc/pagemap
 6:    if (frame % PAGES_IN_CACHE == current_cache_index) then
 7:       add current_page to memory pool
 8:       alloced_buffer += PAGE_SIZE
 9:       current_cache_index = (current_cache_index + 1) % PAGES_IN_CACHE
10:    else
11:       current_page += PAGE_SIZE
12:    end if
13: end while
```

Algorithm 4.2.2 provides a high level overview of the allocation scheme. To start, a very large request for memory is made via *malloc* (on the order of 512 MB). Performing such a large memory request attains the greatest probability of being backed by a large segment of the Buddy-Algorithm's allocation. However, if this large allocation does not yield physically contiguous memory, the Slab-Allocator attempts to fill the outstanding memory request with small pages to reduce physical memory fragmentation. These smaller page allocations result in potential cache overlap. Section 3.2 discusses the theoretical impact on the utilization and page conflict mapping with respect to random frame allocation. The model also serves as an indicator for the expected behaviour of the Buddy-Algorithm under heavy load.

Upon completion of the large memory allocation, Linux's Process Filesystem (*procfs*) is queried for the buffer's frame distribution. *Procfs* exports */proc/pagemap*, which allows userspace applications to query information about any virtual page. In particular this *procfs* node may be queried to determine a virtual page's physical frame number (PFN). As every pair of adjacent PFNs resemble physically adjacent frames of memory, it is possible to determine the distribution of physical frames with respect to the buffer. While the usage of *procfs* is itself not portable by definition, it only utilizes a single node. As long as other platforms export similar functionality, or a system call to obtain virtual page to physical frame numbers can be added, *Cachetest* will continue to operate with the same userspace allocation algorithm.

Algorithm 4.2.2 mitigates the cache issues by only using parts of a large buffer allocation. Currently, a simple linear matching algorithm is used to find a cache-wise contiguous subset of virtual pages belonging to the buffer array. The algorithm translates each virtual page number (VPN) to it's corresponding PFN through */proc/pagemap*. The resulting translation must satisfy the following congruence: PFN mod (# of pages covering cache) = current_cache_index. When a suitable page is found, it is added to the pool of pages which are to be used by the *Distribution Generator* and *Execution Engine*. The variable "current_cache_index" indicates which group of cachesets is to be covered by the next selected frame. Frame matching is performed in this way for as many pages as are required to cover the desired working set size.

Example: Consider an eight-way set-associative cache of size 4 MB with 64 byte cache lines and 4 kB memory pages (2^{12} bytes). The cache has a total of $\frac{\text{Cache size}}{Cachelinesize} = \frac{2^{22}}{2^6} = 2^{16}$ cache lines. Furthermore, there are $\frac{TotalCachelines}{Associativity} = \frac{2^{16}}{2^3} = 2^{13}$ cachesets, with each cacheset occupying $64 * 8 = 512$ bytes. Finally, it requires $2^{13} * 64$ bytes $= 2^{19}$ or 512 kB (or $\frac{1}{8}^{\text{th}}$ of 4 MB) of contiguous memory before any one cacheset is reused. Therefore, $\frac{2^{19}}{2^{12}} = 128$ physically contiguous pages are required to have one cache line of data placed into each cacheset. That is, every 128 contiguous pages form a quotient group over the cachesets. Suppose the first physical frame selected has a page frame number of 0x10. This frame is a member of the congruence class $x \mod 128 = 0\text{x}10$. The next selected frame must be a member of the congruence class $x \mod 128 = 0\text{x}11$, since such a frame would cover the cachesets immediately following those covered by frame 0x10. Using this algorithm for the allocation of all required pages, cache-wise physical contiguity is enforced, without requiring actual physical contiguity among the used frames. A large physical memory pool is required for this approach to work properly. The likelihood of finding physically contiguous frames decreases with the uptime of the benchmarked system but has not been an issue for any tested systems.

Last but not least the buffer may be allocated with large virtual pages. The page size is a constant which the CPU must be aware of, as page sizes determine the characteristics of the address compartmentalization. Deciding on suitable page sizes is a well studied problem and highly workload dependent. General purpose CPUs and operating system often use 4-kB pages, while server or scientific applications tend to use larger page sizes (ie. 4 MB or 1 GB). Focused applications prefer larger page sizes since the often inherent determinism of the application may exploit spatial locality more efficiently.

Although hardware supports multiple page sizes, it is only recently that software support for variable virtual page sizes is embedded into operating systems. The Linux operating system provides “Hugepages” as a means of allocating larger pages. Hugepages mitigate paging overhead at the cost of potential fragmentation within the page. Additionally, the increase in page size provides larger segments of physically contiguous memory, allowing *Cachetest* to measure page allocation impact with an alternate approach.

An allocation which is backed by Hugepages is done via the “mmap” systemcall interface. Linux requires the inclusion of two option flags with *mmap*: MAP_HUGE and MAP_PRIVATE. As of this writing, Linux does not support shared Hugepages. Again, two approaches are studied with respect to page allocation under Linux Hugepages: Default allocation and custom contiguous. As is the case for the “malloc” based approach, default allocation only uses “mmap” without regard for potential cache overlap. The alternative algorithm for performing a custom allocation is identical to that in 4.2.2, with the exception of a larger PAGE_SIZE variable.

4.2.3 Distribution Generator

The third required component of the benchmark suite is a workload generator. This component is responsible for generating memory access patterns, matching the pattern with the identified buffer pool and installing the array used by the *Execution Engine*. While this component is written in a very extensible manner, only 2 distribution patterns have been studied to date: Uniform random and sequential. The current version also includes a Zipf distribution generator and a random harmonic series generator which have not been studied in great depth and are omitted for the remainder of this discussion. Providing a memory access pattern that may be reasoned about at a stochastic level and that provides insight into the the cache's and operating system's memory management performance is the dominating design goal for this component.

The memory access distribution plays a key factor in the effectiveness of CPU caches. Memory access patterns exhibiting a high degree of locality benefit greatly from fast CPU caches. Furthermore, the higher the degree of locality, the lower the impact of page placement on the overall CPI throughput. In terms of the working set model, if time is constant (the working set window is fixed) a high degree of locality will result in a reduction of the working set. This in turn reduces the negative impact of conflicting page placement, as fewer pages are allocated relative to the available cache.

Cachetest provides an extensible memory distribution infrastructure, which requires little effort to implement and use a new distribution. At its core, *Cachetest* uses an integer array to implement a low overhead memory access distribution. For any particular distribution, a sequence of array indexes is generated and stored in the memory pool allocated by the *Buffer Allocator* component. Each element of the array contains the index of another array element, which is to be accessed. These entries are pre-computed by the *Distribution Generator* for two reasons: Computing the next memory location at run-time (in the *Execution Engine*) introduces additional memory traffic and would severely skew the measured cache performance characteristics. Secondly, pre-computing and loading the memory reference pattern warms the CPU caches, resulting in fewer initial misses. Every access distribution must use the first array field as the final memory access, to create a circular access chain.

CPU caches use cache lines as the smallest unit of storage, typically on the order of 32, 64 or 128 bytes. As a result, it is possible to model some degree of locality with an integer array. *Cachetest* forces integer elements to be 4-bytes in size, allowing each cache line to hold $\frac{sizeof(cacheline)}{4}$ reference integers. An AMD Opteron based CPU uses 64 byte cache lines, allowing for 16 integers to be stored in a single cache line. Consequently, each cache line may be visited at most 16 times in one circular access.

While an integer array approach provides some degree of locality, it also limits access distributions that wish to utilize higher degrees of re-use. Exponential distributions (such as Zipf) exhibit extreme levels of locality, potentially leaving the buffer under-utilized. Although this issue is not addressed in the presented version of *Cachetest*, multiple optimizations approaches exist. By altering the size of the array elements, more elements may be stored within a single cache line. Alternatively the granularity of the distribution may be modified, so as to distribute memory accesses over sets of cache lines (rather then individual cache lines). While such an approach loses some accuracy, it provides integer factors of additional re-use.

The *Distribution Generator* is dependent on an additional layer of indirection introduced by the *Buffer Allocator*. As the buffer may be comprised of non-contiguous virtual pages, as is the case for the custom contiguous allocation scheme, a translation layer is needed to map array indexes to the correct virtual page. For example, consider a buffer array spanning 2 virtual pages. There exists $n = \frac{\text{page size}}{\text{sizeof(index)}}$ array indexes per page. If an access to array element $n + 1$ is made, the second page of the *Buffer Allocator*s pool of virtual pages must be accessed. The *Distribution Generator* accounts for these address shifts by suitably offsetting the array indexes at generation-time of the distribution. Offsetting the array's index implements the selective utilization of the available buffer space, and results in a new array index called the "mapped index".

Inserting an index into the integer array is common among all implemented distributions. Upon drawing the next index to access from the chosen distribution's number generator, the translation layer is queried for the mapped index, ϕ. At this point the previously accessed memory location, γ is loaded with the mapped index. That is, buffer[γ] = ϕ. Since the distribution generates cache line numbers, it is possible that a prior drawing already loaded an integer at γ. The cache line is used similarly to a hash-set by linearly searching the cache line for an available integer sized memory slot. Such a memory location is always available given the following terminating condition: If the cache line starting at buffer[ϕ] contains $n - 1$ ($n = \frac{sizeof(cacheline)}{sizeof(index)}$) elements (there is only 1 space left), then this location receives an entry which closes the access loop (buffer[ϕ] = First field in pointer chain).

The uniform random distribution generates a pointer chain, where each cache line sized block of memory is equally likely to be accessed next. Random numbers are drawn from a custom implementation of the Mersenne Twister pseudo-random number generator [38]. The initial cache line to be visited is stored in the distribution's object for bootstrapping the *Execution Engine*. While much software inherently exhibits some degree of locality, most re-use is a consequence of loops in program code. A random access pattern is not unrealistic in modern software architectures with respect to data. Since most dynamic tree data structures require random memory accesses for traversal.

In contrast, a sequential accesses pattern touches each cache line exactly once. A sequential pattern follows a deterministic access chain with regards to which cache set is accessed (as shown in Figure 4.3). CPU stride prefetching may become noticeable in such a scenario, as the stride-size is constant, resulting in hits on every prefetch of data. Alternatively, a variable stride size may be used to mitigate the effects of the hardware prefetching circuitry. The inclusion of a sequential distribution provides insight into access patterns exhibiting no spatial locality. Sequential data access is common in all types of applications, as the pattern is treated as a first-class citizen in programming languages.

Array's and linked lists are examples of data-structures which are often interacted with in a sequential manner.

A general approach to the entire benchmark tool is to leave as much of the system unaltered as possible. This includes hardware and kernel components, such as the CPU's prefetcher, memory hierarchy, or scheduling algorithms. The random distribution bypasses the CPU's stride prefetcher without needing to disable it. Results which are obtained in this manner, are still comparable to other memory distributions which may wish to measure the impact of the data prefetcher. Additionally, by not disabling hardware components, the tool continues to exercise these hardware paths and avoids the inclusion of artificial performance gains.

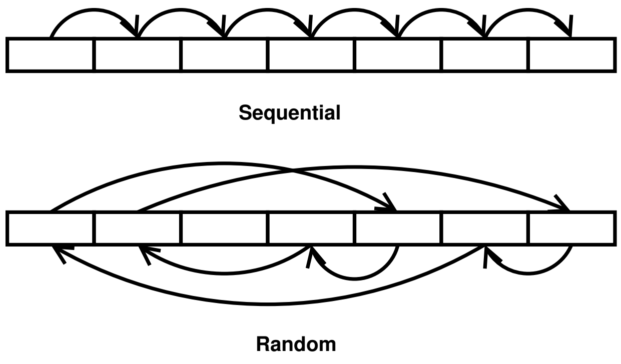


Figure 4.3: Example visual access patterns over cache lines

4.2.4 PMU Provider

Obtaining accurate performance counter readings is a non-trivial task in itself. Aside from the PMU multiplexing issues described in Section 2.2, reducing the code paths between virtually (higher level method) and physically (control register change) stopping a counter is challenging. Fortunately, hardware vendors provide the option to selectively count performance events in either privileged or unprivileged mode. By configuring the PMU to only count in unprivileged mode, a single *syscall* instruction is required to temporarily stop the counter. Once trapped into the kernel, the performance counter is disabled at the control register level.

In order to avoid third party library dependence, the *Cachetest* benchmark provides a custom interface to performance monitoring hardware. The interface is currently a wrapper to Linux's interface for the Machine Specific Registers, or MSR. Performance counters are controlled via these registers, a task which is accomplish-able from userspace. However, dedicated kernel level support for PMU hardware also exists in the form of *perf_events* [15]. Linux provides generic system call interfaces to *perf_events*, providing various levels of abstraction to the actual PMU components. *Cachetest* additionally implements a custom, low-overhead and single purpose wrapper to the system call interface. The wrapper accepts C++ standard STL containers for easy interaction between user and kernel components. While maintaining kernel and userspace separation, the provided *perf_events* wrapper requires nearly the minimum necessary overhead (a single function call). Lower overhead and further reductions in measured events can only be achieved by breaking down the *syscall* interface, which is outside the scope of this work. Currently, the wrapper library to Linux's *perf_events* accepts raw event codes as detailed by the manufacturer [1] as well as logical names provided by Perfmon2 [21].

Alternatively, *Cachetest*'s custom PMU interface may also be replaced with the user-level PMU provider PAPI [9]. PAPI (Performance Application Programming Interface), developed at University of Tennessee, is a feature rich performance monitoring hardware API. The software package provides an abstraction to performance counters as well as high precision timers and information about kernel level events. Additionally, PAPI provides diagnostic utilities to discover a CPU's measurable hardware events. With the aid of Perfmon2, PAPI may be configured with logical names rather than strict raw event codes. However, the usage of large libraries like PAPI may introduce an increase in measured overhead [15].

4.3 Validation and Results

In order to validate the accuracy and correctness of the benchmarking tool, experiments are performed on three CPU architectures. The goal is to match the theoretical cache performance curves from Section 2.3.2. For this, every experiment measures the amount of iterations the *Execution Engine* performs, as well as cache hit statistics for every cache level.

Additionally the *Cachetest* benchmark tool is used to investigate the impact of page placement on cache performance and instruction throughput.

4.3.1 Testbed Systems

For this study, three different platforms have been investigated to demonstrate the system-agnostic nature of the benchmark suite and its results. The set of systems is composed of an Intel Core i7 720qm, a Six-Core AMD Opteron 2427, and a Power 7 740 CPU. More detailed specifications and comparisons are listed in Table 4.1.

	Systems		
	AMD	Power 7	Intel
L1 Cache Size	64 kB	32 kB	32 kB
L1 Associativity	8	8	8
L2 Cache Size	512 kB	256 kB	256 kB
L2 Associativity	16	8	8
L3 Cache Size	6 MB	4 MB	6 MB
L3 Associativity	48	8	12
Default Page Size	4 kB	64 kB	4 kB
Huge Page Size	2 MB	16 MB	2 MB
TLB Coverage	4 MB	32 MB	4 MB

Table 4.1: Test-systems

All architectures run Linux kernel version 3.2.9 and isolate the given workloads with Linux CPU sets. Additionally, each test platform moves all existing processes to the CPU's boot core (generally labelled as core 0). This is done to remove potential inter-core-interrupt traffic and isolate shared caches on the Power 7 architecture.

The benchmark suite is run at the highest priority level and pinned to a single core in order to avoid process migration. Additionally, all modifiable hardware interrupts are re-routed to unused cores through Linux's *sysfs* environment [42]. Although not explicitly studied in this work, *Cachetest* does provide a multi-core feature whereby execution is synchronized via Linux's signal interface. This provides the functionality required to accurately measure simultaneously executing workloads, without requiring a user-level threading library.

Even with the aforementioned precautions, there still exists great potential for variability in the results. Interrupts and L3 cache pollution by other processes are inevitable on real hardware. For this reason every benchmark is run over a 10 second interval and repeated 20 times in order to reduce each experiment's variance.

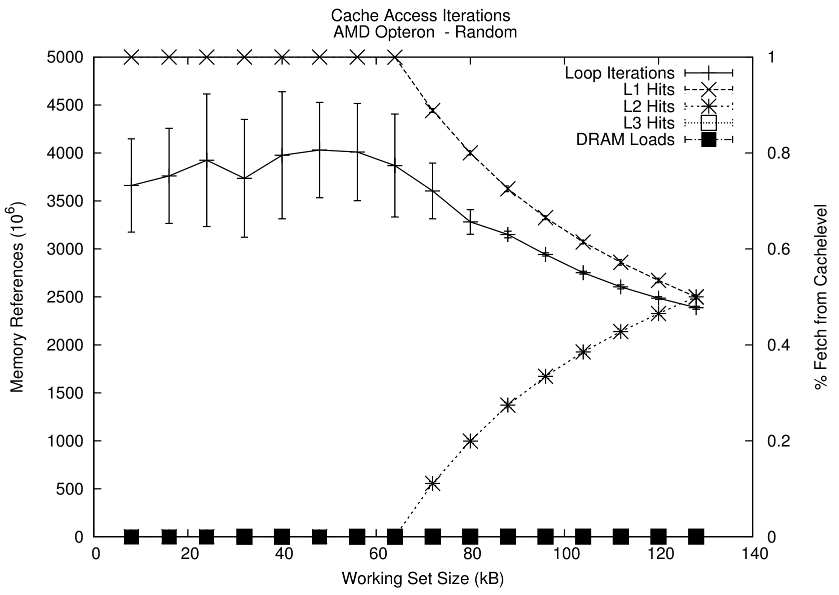


Figure 4.4: AMD *Cachetest's* L1 Results

4.3.2 Cache Hit Characteristics

To determine the accuracy of the cache utilization model and *Cachetest* application as a whole, a profile over each tested architecture's cache coverage is performed. For every level of the cache hierarchy, 16 working set sizes are evaluated and visualized in the following figures. Each figure plots the number of memory accesses performed on the left y-axis, while simultaneously showing a percentage breakdown of memory fetches from each cache level on the right y-axis. The x-axis represents one of the 16 working set sizes used to represent each experiment.

The following plots are gathered on the AMD Opteron architecture. Experiments performed on the other architectures show identically shaped cache hit rate curves and are available in the appendix of this thesis.

Beginning with a working set in the range of L1 cache memory (8 kB to 128 kB) and Figure 4.4, the cache hit-rate curves match those which have been theorized (Section 2.3.2). Initially, while the working set is smaller than the L1 cache size, 100% of memory fetches are performed by the L1 cache. Again, this makes intuitive sense, as the amount of memory used to perform the experiment fits entirely into the L1 cache. As the working set increases in size and topples the L1 cache boundary, L1 hits decrease symmetrically to an

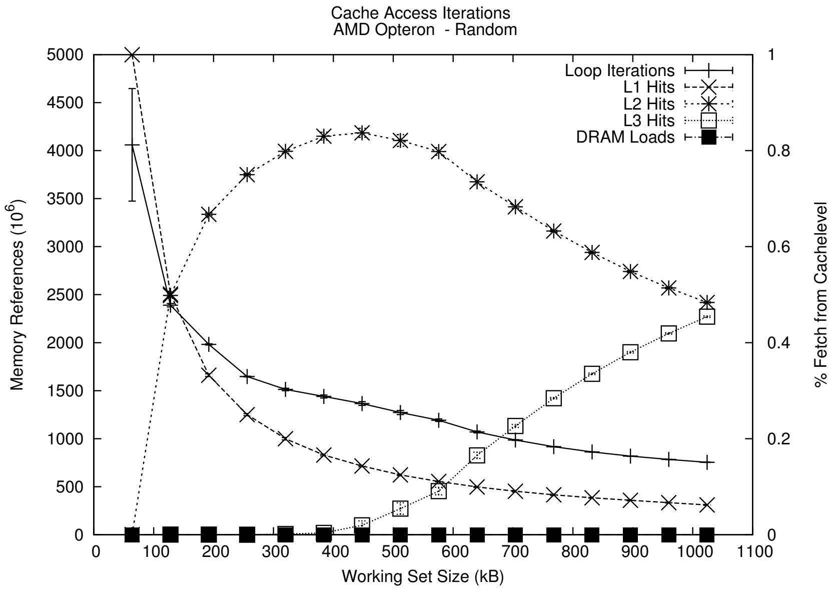


Figure 4.5: AMD *Cachetest's* L2 Results

observable increase in L2 hits. Due to the decrease in L1 fetches, the absolute instruction throughput decreases by a factor of 37% between working set sizes of 64 kB to 128 kB. As expected, L3 and DRAM do not serve any significant amount of memory requests, as the working set size fits into the L1 and L2 cache regions. The large variations in memory instruction throughput are a result of the the virtual memory layout with respect to the CPU cache. Although the entire working set fits into the L1 cache (for experiments one through eight), the memory layout may introduce additional cache misses, which are not a result of capacity.

Figure 4.5 visualizes the transition between all three cache levels. On the far left, a working set size of 64 kB still fits into the L1 cache and causes the instruction throughput to excel relative to all other depicted points. Consistent with Figure 4.4, a working set size of 128 kB (the second point) causes a 37% decrease in absolute instruction throughput relative to a working set that fits into L1.

The cache behaviour again matches that of the theoretical optimal utilization curves. As the working set sizes increases in size, the maximum L2 utilization marginally exceeds 80%. Once the L2's size is exceeded by the working set, capacity misses begin to dominate the cache's performance curves. The capacity misses are evident, as L3 hits increase symmetrically to the decline in L2 hits.

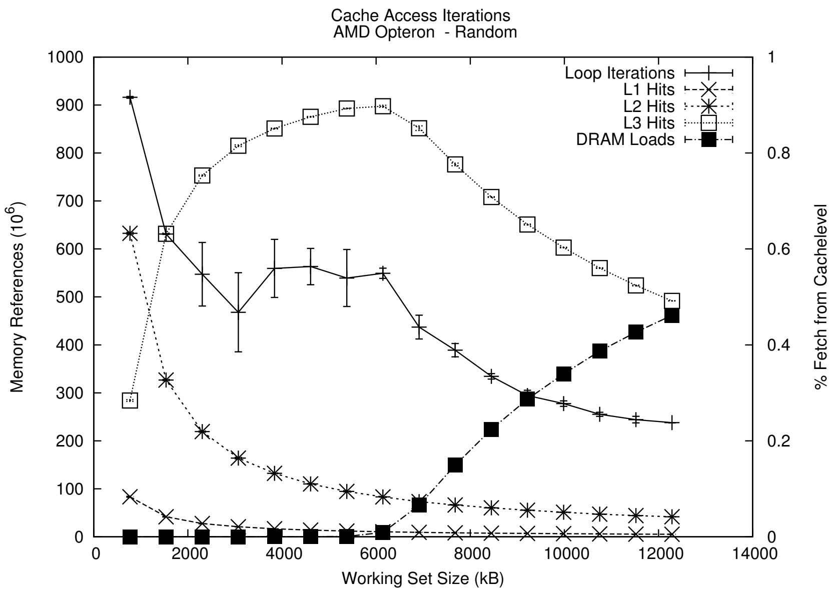


Figure 4.6: AMD *Cachetest's* L3 Results

It is important to note that a transition between any two cache levels does not negate the positive effects of the lower level. That is to say, that there always exists a finite chance of a previously referenced data being served from a lower cache level. The Figure 4.5 illustrates this behaviour via a non-zero L1 hit-rate curve.

Consistent with the measured memory latencies in Section 4.1, the performance impact due to misses in the L1 cache are an order of magnitude larger compared to L2 misses. Transitioning out of the L2 cache region produces a 50% drop in measured instruction throughput (evident between datapoints corresponding to 512 kB and 1024 kB).

Lastly, Figure 4.6 illustrates *Cachetest's* behaviour with a working set fitted to the L3 cache. As in the previous experiment, the effects of the previous cache level are visible at the far left of the plot. The cache hit-rate still follows the theoretical optimum, depicting an initial increase in L3 hit-rates, followed by the inverse exponential decline as the CPU's L3 cache size (6 MB) is exceeded. Likewise, main memory hits become more likely and a sharp decline in instruction throughput is observed.

Instruction throughput declines by approximately 50% between working set sizes of 6 MB and 12 MB, as a result of fetching data from main memory as opposed to the CPU's cache. This is again consistent with the data access latency data described previously. The conducted experiment only plots working sets of up to 1.5 times the size of the measured

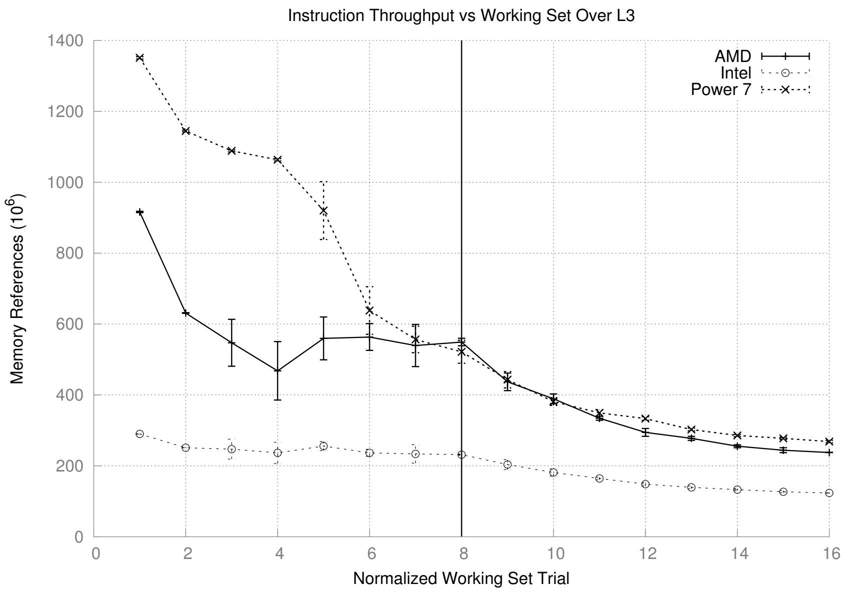


Figure 4.7: Instruction throughput comparison over L3 Cache

cache level. This has been done for readability of the plots, as the main memory latency only slowly dominates the L3's performance-improving impact.

4.3.3 Page Placement Impact

Every figure in the previous section perfectly describes the cache hit rate behaviour for a uniform random workload. The presented data only utilized a default, non-cache aware, memory allocation scheme. Additionally, each plot exhibits a large degree of variability in instruction throughput for working set sizes that fit into a given cache level. Measured CPI variability is particularly high for working set sizes close to the cache's boundary.

Figure 4.7 compares the instruction throughput of the three CPU architectures with respect to memory reference throughput, while utilizing workloads which fit into each CPU's L3 cache. Although every architecture executes the same number of experiments, each CPU has a different cache size. Every experiment run spans 16 evenly distributed working set sizes in the range of ± 0.5(cachesize). Consequently, the x-axis has been normalized to reflect each of the 16 performed experiments within the given range. Experiment eight is highlighted as the last working set size which fits into each CPU's L3 cache.

Despite the fact that each CPU's memory access latencies and core clock frequencies

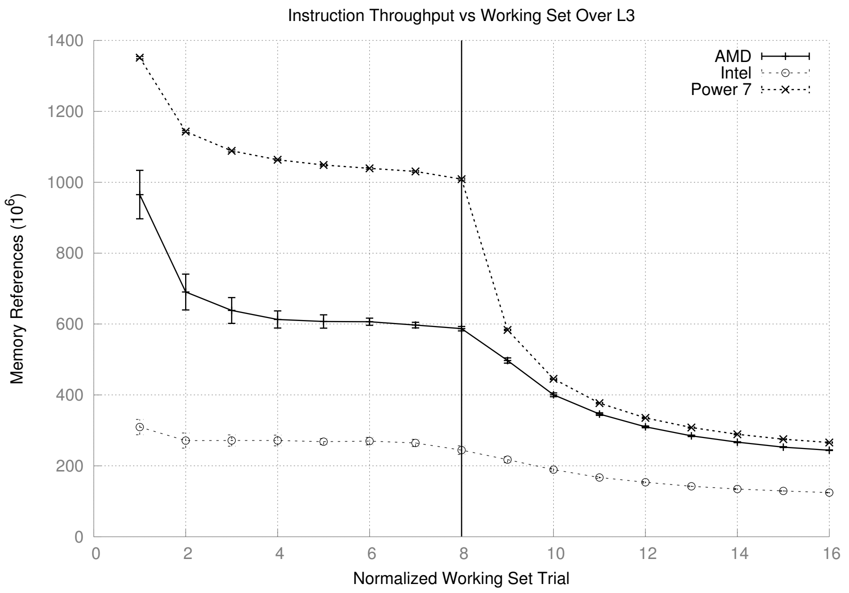


Figure 4.8: Instruction throughput comparison over L3 Cache with Perfect Page Colouring

differ, all three experiments exhibit significantly more memory instruction throughput variation for workloads which fit into the cache (experiments one through 8) as compared to workloads that exceed the caches capacity. Beyond the eighth experiment, each measured experiment stabilizes and follows a smoothly declining curve. Note that, although the variability of the Core i7 results do not appear to be as significant as those on Power 7 and AMD, they represent the same percentage deviation. Due to scaling of the y-axis, the standard deviation bars are not as pronounced. The appendix includes a full-scale representation of each architecture's experimental trials.

The observed variability is spatially consistent with the conjectured page-conflict mappings of Section 3.2. Recall that the random frame allocation model predicts a peak conflict rate when the working set is equal to the cache size. Each conflicting page mapping results in the over-subscription of CPU cache sets, which subsequently introduce unnecessary cache misses and memory traffic.

Smarter memory allocation could avoid these conflict misses and provide greater runtime performance. To validate the impact of page placement on the caching and CPI performance of the *Cachetest* benchmark, the custom contiguous memory allocation is used and contrasted to the default allocation scheme. The cache-aware memory allocation scheme ensures even cache set utilization and avoids page mapping conflicts until the working set exceeds the cache size. Figure 4.8 showcases these experiments and validates the hypothesis. By performing cache-aware virtual memory allocations, the *Cachetest* benchmark is able to stabilize it's memory instruction throughput. This is evident in the significant reduction of variable instruction throughput, as well as an overall absolute increase in instruction throughput. The Power 7 architecture particularly improves instruction throughput performance due to page colouring. With a best case performance improvement of 66 %, the Power 7 architecture demonstrates the significance of cache-aware memory allocation. Additionally, all architectures are now able to maintain their best case instruction throughput for working sets up to their respective maximum cache sizes. In the case of Power 7 an additional 2 MB (50 % of total cache space) of cache memory are effectively utilized.

4.3.4 Page Size Impact

The CPU's page size further complicates the performance characteristics of CPU caches. Recall that the page size and cache associativity determine the number of required pages to cover each cache set once. An increase in page size consequently reduces this number, while simultaneously reducing the probability of cache conflicts due to page placement.

Additionally, larger page sizes reduce the amount of TLB entries required to cover the same amount of memory. This reduction in TLB granularity has the additional effect of requiring less cache memory to store evicted entries, as well as require fewer TLB misses to serve memory requests which exceed the TLB's coverage. With regards to instruction throughput, these factors manifest as additional cache contention and pipeline stalls respectively.

Figures 4.9 and 4.10 elaborate on these points by visualizing and contrasting the instruction throughput on all three architectures when using Linux's Hugepages memory allocation method. Figure 4.9 reflects similar measured performance when compared to the custom contiguous buffer allocation scheme in Figure 4.8. This is no coincidence, as both approaches decrease the potential for cache conflict misses due to over-subscribing cachesets. Moreover, a reduction in TLB overhead is observed. Figure 4.10 depicts the measured standard deviation of all three experiment approaches on AMD. While the default memory allocation scheme dominates the plot with large non-determinism, both cache-aware allocation schemes give increasing predictability. By removing cache conflict misses induced by poor page placement and a reduction in TLB overhead, the allocation method based on Linux's Hugepages results in the most deterministic execution profile. Interestingly, the custom contiguous memory allocator undergoes a rapid reduction of variance

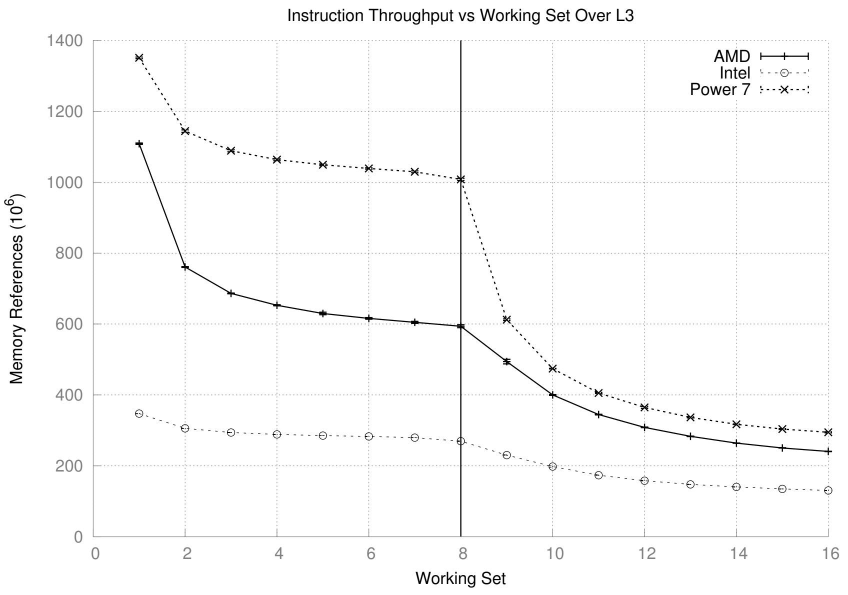


Figure 4.9: Instruction throughput comparison over L3 Cache with Huge Pages

as the working set size increases. This is attributed to AMD Opteron's set associative TLB implementation, which although capable of covering 4 MB of addressable memory, causes potentially unnecessary TLB conflicts. The analysis of this thesis could be applied to study the performance impact of set associative TLB's on instruction throughput and forms an intriguing topic for future study. Just as is the case for set associative caches, as the working set size increases and eclipses the size of TLB, the probability of hitting in the TLB decreases at an inverse rate.

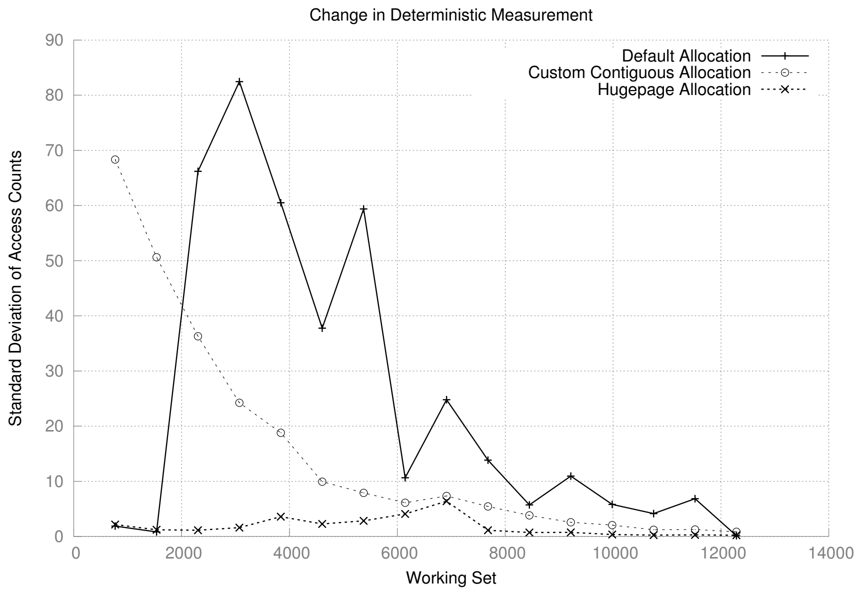


Figure 4.10: Comparison of Variability in CPI on AMD

Chapter 5

Impact On Models and Existing Systems

Having validated the performance impact due to frame allocation, it remains to be shown how to model the impact of page conflicts on CPU cache behaviour. Ideally such a model describes the cache hit/miss-rate of every cache level due to page placement.

The popular stack distance profile model is examined with a focus on uneven-utilization and under-utilization of CPU caches. Contemporary hardware does not provide the means to accurately measure SDPs, prompting researchers to create approximations based on existing hardware performance counters. However, these methods fail due to various over-simplifying assumptions, such as a perfect LRU replacement policy and even cache set utilization.

The impact of page sizes on the determinism and performance of CPU caches is evident. Linux's Hugepages are an extreme form of larger than default page sizes on modern operating systems. Power 7 has recognized the performance limiting implications of page sizes which may be too small for contemporary large CPU caches by increasing the default virtual page size to 64 kB (compared to 4 kB).

5.1 Virtual Page Sizes

An operating system's virtual page size forms the smallest physically contiguous memory allocation unit. As such, its size dictates the coverage and conflict potential related to the CPU cache. Although last level cache (LLC) sizes have steadily increased over the past decade, the default virtual page size of many operating systems has remained constant at 4 kB. This ever growing gap has subtle performance impacts on workloads that utilize the last level cache.

Consider a CPU cache with size C in bytes, associativity A, line size L and page size P in bytes. The size of a cache set is given as $S = AL$, with a total of $T = \frac{C}{S}$ cache sets over the entire cache. A virtual page covers $\frac{P}{L}$ cache lines, where each cache line maps into a distinct and adjacent cache set as long as $\frac{P}{L} \leq T$. Additionally, the number of "bins" as described by Kessler et al. is given by:

$$B = \frac{T}{\frac{P}{L}} = \frac{TL}{P} = \frac{CL}{SP} = \frac{C}{AP} \tag{5.1}$$

The number of bins in a cache describes the degree of potential cache under-utilization. Fewer bins result in a reduction of unused cache space, while simultaneously increasing the effects of sub-optimal frame allocation strategies. As the probability of over-subscribing to any cache-bin increases, so does the probability of page placement induced cache misses (as dictated by Equation 5.2).

The issue of under-utilizing the cache could be addressed by increasing the page size or the cache's associativity. Either alteration decreases the amount of available bins and reduces the observed cache-miss issues. Unfortunately, increasing the associativity of large LLCs is not feasible when compared to contemporary cache sizes. Space and latency requirements on CPU dies prohibit the inclusion of the required circuitry. CPU caches are designed to be fast, requiring short critical paths for lookup and tag matching. This requirement forms an upper bound on the number of matches which may be performed efficiently each cycle. Additionally, the rapid increase in cache sizes would require an equally rapid increase in associativity.

Increasing page sizes however is an attainable goal, with support for variable page sizes already present in contemporary CPUs. Power 7 based architectures already utilize 64 kB pages, allowing the entire L1 (data and instruction) cache to be covered. Thus, workloads which fit into a single page have the benefit of encountering no caching conflicts at the L1 level. Furthermore, data may be easily placed in the cache by only utilizing virtual addresses, as the underlying frame directly maps to the L1 cache sets. This has the attractive benefit of avoiding cache conflicts with careful data placement in the virtual address space.

Linux's Hugepages serve as an experimental verification of this hypothesis. By using extremely large page sizes, the amount of bins is reduced to a single one. The experiment shows decreased variability as well as improved CPI measurements due to a reduction in conflict page mappings. As long as a given workload effectively utilizes the memory provided by a single virtual page, internal fragmentation and wasted space are insignificant. A thorough study of contemporary working set sizes is required to determine the optimal page size for a given workload. Ideally, an operating system would provide a system such as Linux's "Transparent Hugepages", to provide variable sized virtual pages for a given working set.

5.2 Theoretical Model

Section 3.2 presents evidence of the potential issues associated with naive frame allocation methods, while Section 3.4 discusses the theoretical potential impacts. Additionally, the issue of naive page placement may be formalized mathematically as follows. Let each page be mapped into a "bin", which covers a subset of contiguous cache sets.

For the uniform random access pattern, the probability of a cache miss based on the operating system's frame allocation may be estimated as follows:
Let A be the cache's degree of associativity.
Let $T(x)$ be the number of pages mapped to bin x.
Let $E(x) = \max(0, T(x) - A)$, the number of capacity-exceeding pages in bin x.
The probability of a cache-miss is

$$P_{\text{Miss}}(x) = \left(\frac{E(x)}{E(x) + A}\right) * \left(\frac{\text{Pages in bin x}}{\text{Total Pages}}\right) \tag{5.2}$$

While there exist fewer pages in the bin than the cache's associativity, no conflict misses will occur ($E(x) = 0$). This case is trivial, as sufficient space exists within the cache to warrant additional data to be placed. However, as the bin's capacity is exceeded ($E(x) > 0$), a given memory request's probability of missing in the cache increases with $\frac{E(x)}{\text{E(x) + A}}$. Again, this result is easy to see when considering the probability of accessing a page that is not present in the CPU cache. As the number of pages increases, so does the probability of accessing a non-cached page.

In order to generalize the probability model to the entire cache, an access to bin x is weighted by the fractional amount of pages in x. As a first order approximation, the fraction of total pages mapped into the bin is used as a hotness indicator. Unfortunately this model does not account for inter-page locality behaviour. Repeated accesses to a single page are not reflected in the weighting, which makes this model specific to the uniform random memory access pattern.

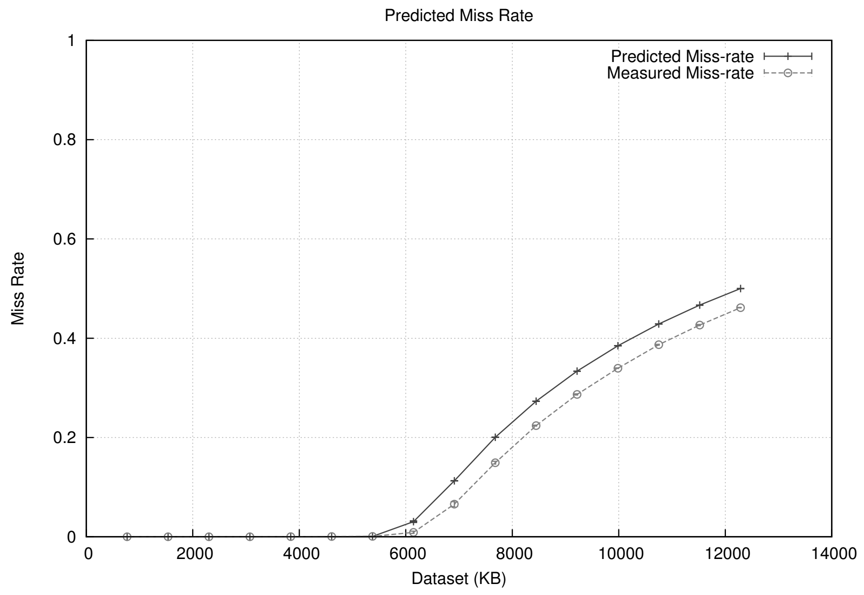


Figure 5.1: Predicted Miss rate of L3 random workload on AMD

Figure 5.1 illustrates the model visually and uses the same test run that is shown in Figure 4.6. Note that the theoretical model requires the virtual-to-physical frame allocation as input. The approach is able to predict the cache miss rate within 5% error on AMD. This large error is attributed to the high associativity of the AMD Opteron CPU. When increasing a CPU cache's associativity without also increasing its overall size, the model cannot cope with the decrease in page-to-slot mapping granularity. As the associativity increases, a single page covers more cache sets, but the number of cache sets decreases. This results in a loss of information, as fewer contention points are available.

In comparison, the model holds up for the cases of Intel and Power 7 (low associativity). Figures 5.2 and 5.3 show statistical significance among the model and measured results. Each plot depicts a 95% confidence interval of the modelled result. As the Linux operating system does not implement any form of page colouring, many pages begin to overlap in the cache starting at the 2 MB mark. As predicted by the random memory allocation distribution, the capacity of many bins is exceeded prior to exceeding the cache's capacity. This results in the under-utilization of the cache and subsequent increase in cache misses.

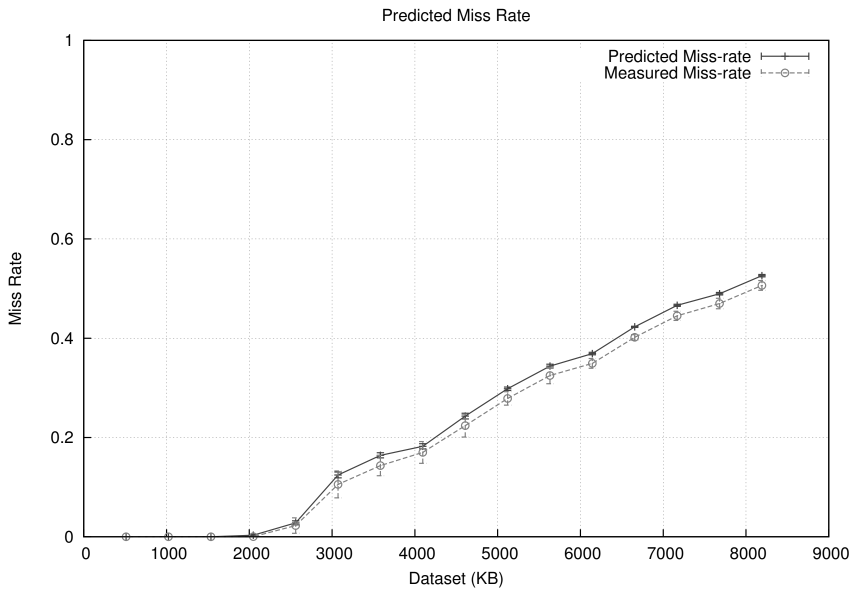


Figure 5.2: Predicted Miss rate of L3 random workload on Power 7

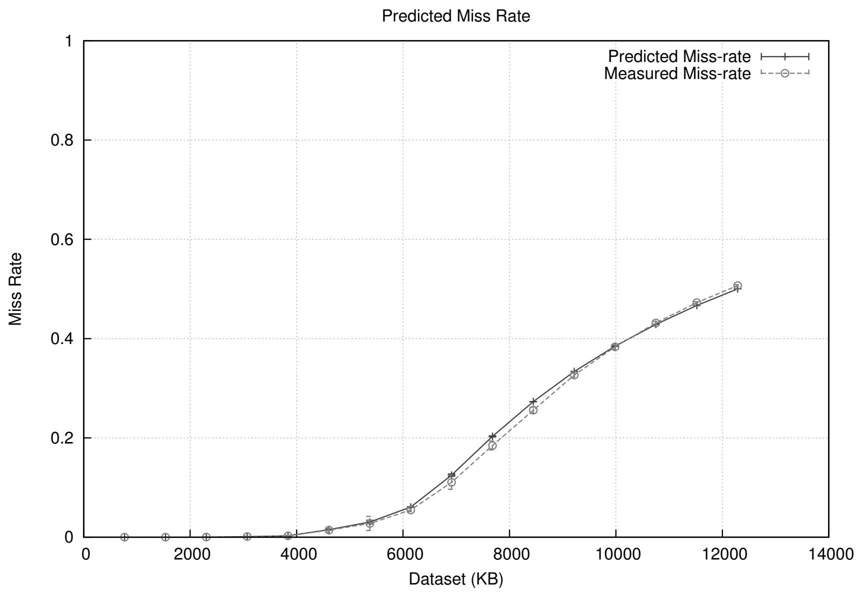


Figure 5.3: Predicted Miss rate of L3 random workload on Intel Core i7

Applicability of the model is limited, as it assumes a uniform random distribution of memory accesses over the working set. Currently, there exists no notion of locality within the the uniform random trials. If the model is to be extended to include inter-page memory references, data from stack distance profiles would need to be included. While all mechanisms for such an investigation are in place, due to time constraints this analysis is left as future work.

5.3 Stack Distance Profiling

The tool chain implemented as part of this work includes scripts capable of generating stack distance profiles. For portability and ease of extensibility, all of the SDP calculations are performed with Python 2.x scripts. A memory reference string in the form of accessed cache sets (or cache lines) is required as input, although parsing of an arbitrary virtual or physical address trace is also supported.

The script computes the stack distance profile for every cache set, as well as the weighted average of all cache sets combined. As all explored literature is unclear on how to generalize a stack distance profile across an entire cache, a weighted average is chosen. The weight of each cache set's SDP is based on the fraction of total memory references to that cache set, giving hot cache sets more weight. However, this also implies that cache misses in hot cache sets are weighted heavier, accounting for conflict misses generated by sub-optimal page placement.

In order to contrast non virtual memory aware methods of approximating stack distance profiles, with those that are, *Cachetset* allows for the exportation of its memory access patterns. While this is not currently possible in hardware, the presented application suite queries the operating system for virtual page to physical frame mappings. This provides the necessary information to compute a SDP as it would be measured by hardware. Deriving the real SDP is accomplished by creating a virtual-to-physical cache set transfer function. The required translation information is provided by the contents of the processes' page tables.

For an access to a given virtual address, the page table of the benchmarked process is consulted. The access is translated to a physical memory request, which subsequently may be translated into an access to a cache set. Once the physical cache set has been determined, a true LRU replacement algorithm is assumed and simulated within the cache set. Although the shortcomings of this assumption has been discussed with regards to existing literature, the focus of this experiment is to measure the degree of inaccuracy of existing SDP approximation approaches. For this reason, the same replacement policy, as that found in contemporary literature, is used and contrasted.

5.3.1 Page Placement Impact on Stack Distance Profiling

Stack distance profiles are calculated with the knowledge of memory access patterns and how these patterns translate to the physical cache sets. Virtual page placement directly affects which cache set's data will be cached and thus may introduce variability into SDP calculations. In particular, any optimization based on stack distance profiles may become inaccurate with time or repeated executions.

Consider a compile-time and SDP-based optimization approach for multi-core aware scheduling. In such a scenario, the compiler generates stack distance profiles which represent the approximate memory re-use behaviour of the application. However, assumptions must be made about the memory layout which will be present at run-time. While some efforts have shown how to reduce the impact of page placement on stack distance calculations [10], the employed statistical methods only extend to two-way set associative caches. Contemporary optimization approaches ignore the virtual-to-physical page mapping impact and assume the physical placement of data to mirror that of the virtual address space.

Additionally, run-time optimizations based on miss-rate curves are in peril. Many run-time optimizations which approximate stack distance profiles via cache miss-rates do not consider the potential for non-deterministic variability due to virtual page placement.

To measure the degree of variability and error due to memory layout assumptions, *Cachetest*'s generated memory reference string is evaluated with respect to SDP. First, a stack distance profile is generated assuming a perfect memory allocation (physically contiguous memory allocations). Secondly, the same trace is translated to physical cache set accesses and re-evaluated using SDP. Figure 5.5 plots the fractional difference between the approaches, $\frac{PhysicallyAwareMissRate}{NaiveMissRate}$, on Power 7 (note the logscale on the y-axis). Each bar represents the median of the 20 measurements and each errorbar indicates the maximum observed value.

The figure confirms the previously discussed results, with large discrepancies visible for working set sizes that lie within the cache. Due to the extreme sway in error, the median is used rather than the mean, with error bars indicating the maximum observed error. The observed minimum error is omitted, as it is approximately 1.0 for every run. Interestingly, these results remain constant, across a system reboot which disqualifies claims of the slab allocation algorithm providing sufficient physical contiguity.

With regards to the studied Power 7 architecture, the experiment indicates a worst case SDP miss-rate indicator of 65 times that of the theoretical optimum. The variation and scale of these miss-rates lead to the unpredictable performance behaviour observed in Section 4.3.2 and outline a working set region which is particularly susceptible to miss placed pages. All architectures present the largest deviation from the theoretical optimum for working set sizes which form the cirtical sizes at which last level caches are most effective. These sizes correspond to working sets which would maximally utilze the studied L3 caches and thus reduce their effictiveness in mitigating the memory access latencies.

Figure 5.4 repeats the analysis for the AMD Opteron CPU architecture. Again, the CPU's high degree of associativity is evident in the observed results. The increase in associativity dictates the number of pages which may be placed into the same bin before an overflow occurs. Additionally, there exists a factor of two difference between the number of cache sets on AMD versus Power 7. AMD, utilizing the fewest cache sets of all three architectures, provides the most accurate stack distance profile.

Finally, the observed analysis of Intel's Core i7 differ by a factor of 10 when compared to Power 7, although their cache's associativities are comprable. This large difference arises due to the increase in page size, which is a factor of 16 larger on Power 7 (this is also the case for AMD). As the page size increases, the probability of performing a bad page allocation, relative to the CPU cache, decreases. However, the increase in page size causes more cache sets to be in conflict if the corresponding page is mapped to an over-subscribed bin.

While it is theoretically possible to express the expected cache-miss rate from Equations 3.1 and 5.2, the solution involves application of the incomplete beta function which subsequently results in a recurrence relation. As solving this relation is out of the scope of this work, it is left as future work.

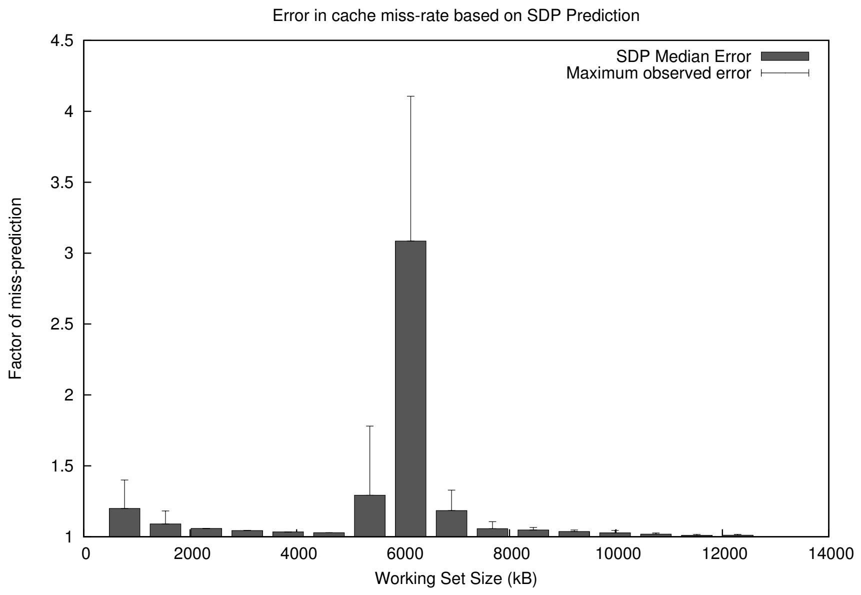


Figure 5.4: Error in cache miss-rate prediction due to uneven cache utilization on AMD

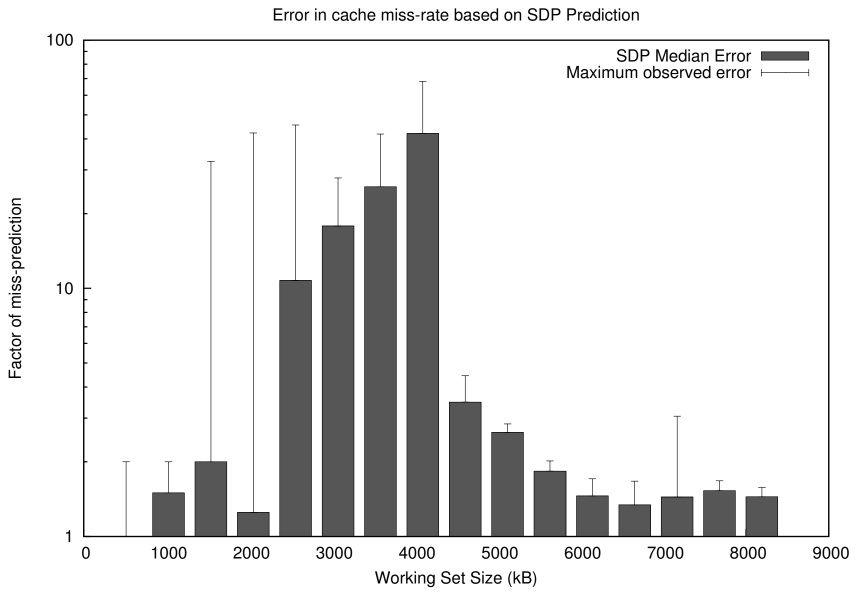


Figure 5.5: Error in cache miss-rate prediction due to uneven cache utilization on Power 7

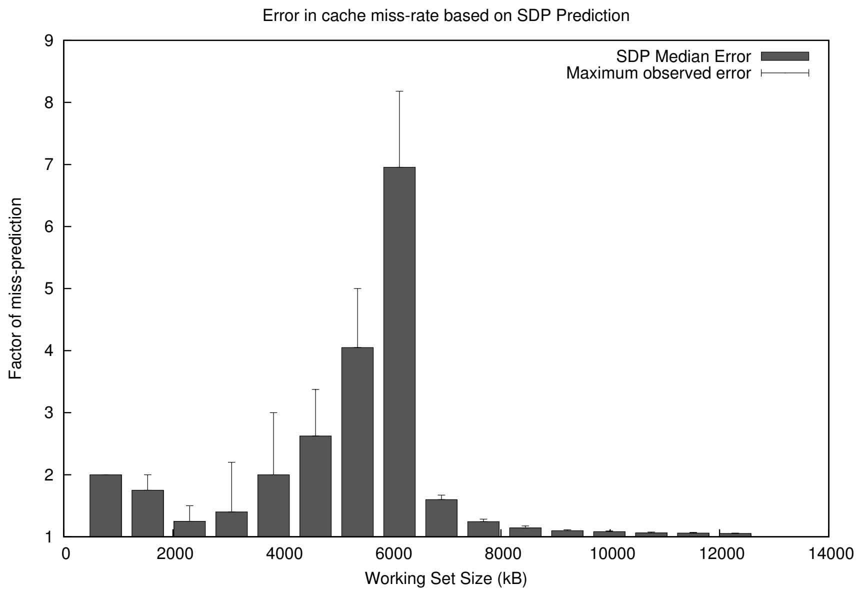


Figure 5.6: Error in cache miss-rate prediction due to uneven cache utilization on Intel

Chapter 6

Conclusion and Future Work

Today's systems research is heavily driven by benchmark results. Conceptually, a standardized benchmark provides an equal ground for comparison and evaluation of overall system performance. Benchmark suites such as SPEC CPU, SPLASH 2 and PARSEC are large software packages designed to stress various aspects of hardware and software performance. Utilizing these benchmark suites (and others of their kin) provide a coarse-grained view of whole system performance. While the most recent offerings of these tools provide great breadth and depth, their usage comes at the expense of insight. Very few users know the inner workings of these benchmark suites and often accept their usage as the panacea to software performance comparison.

The work presented in this thesis takes a systematic approach to understanding contemporary virtual memory performance characteristics. With the aid of a small and manageable benchmark tool, titled *Cachetest*, theoretical cache performance is validated. *Cachetest* provides the infrastructure for machine-level profiling of custom memory access patterns. Profiles are generated as a combination of instruction throughput and hardware performance counter information.

Experimentation and code inspection indicate that the Linux operating system does not provide cache-aware memory allocation or mechanisms for attaining greater run-time performance determinism. With measured instruction throughput degradations of up to 50% for non cache-aware memory allocation, a more careful approach to memory allocation should be considered. Additionally, run-time variability with regards to CPU cache utilization has been observed. These variations lead to unpredictable application performance and directly influences other benchmarking techniques. A reduction in execution performance variability leads to a reduction in required trials for experimental software performance evaluation.

The importance and predictability of the performance impact on CPU caches is modelled and validated experimentally. By inspecting the frame allocation of the operating system it is possible to accurately predict the impact on CPU cache behaviour as long as some assumptions are made about the applications memory access patterns. Common locality metrics are re-evaluated, removing the assumption of a perfect cache-aware memory allocation. While this work does not show absolute prediction errors in existing work, it is now clear that the memory layout plays a large role for the accuracy of all models.

However, it is also demonstrated that there exists a critical working set size with respect to the studied cache size at which the impact of page placement is the greatest. As long as the working set size of a particular workload fits within a cache-level, cache-aware page placement is critical to optimally utilize the cache's available space. For working set sizes exceeding a cache's capacity, page placement becomes less critical as memory latencies are dominated by expensive main memory data fetches. With respect to contemporary memory hierarchies, the largest overall performance sway is observed to occur in the L1 cache. A miss in this cache requires upward of 20 times the cycles in latency when compared to misses in other cache-levels and hence benefits the most from careful virtual page placement.

Finally, this work also re-exposes the need for variable page sizes on general purpose operating systems and their underlying hardware. Depending on the characteristics of a CPU's cache in combination with a given workload, variable page sizes yield greater runtime performance predictability as well as potentially fewer page conflicts. Increasing the default page size to cover the first level cache eliminates unpredictable benchmark execution runs. This reduces the requirement for many repeated measurements as execution profiles become more deterministic. This area provides opportunity for a more in-depth analysis, where various page sizes could be simulated with the *Cachetest* benchmark suite.

With this work, the fundamental properties of contemporary CPU caches and virtual memory's influence on these properties is re-examined. Insight attained from this work should provide the reader with reasons and examples of *why* fundamental operating system mechanics may skew benchmarking results.

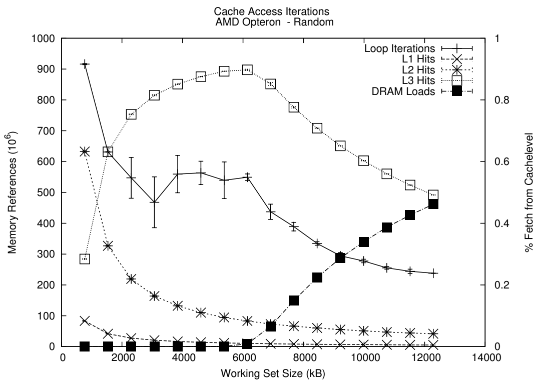
APPENDICES

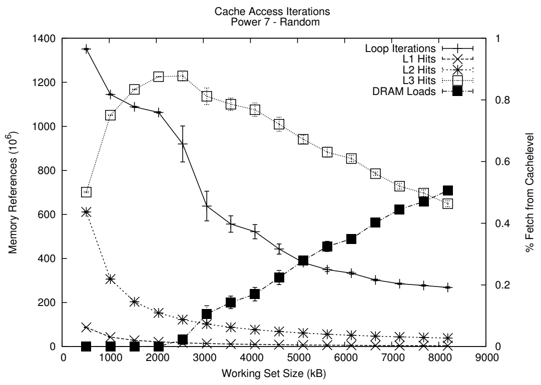
Appendix A

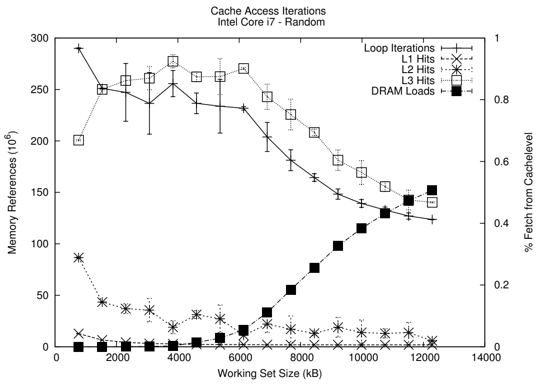
L3 Measurements

A.1 Naive Allocation Scheme

The figure shows the output from the *Cachetest* benchmarking suite for different machines, depicting all hit-rates among the cache's levels, as well as the absolute instruction throughput. The respective L3 cache sizes for AMD Opteron, Power 7 and Intel Core i7 are 6 MB, 4MB and 6 MB respectively. These values are also deductible from the various hit-rate curves, as well as the declining instruction throughputs.

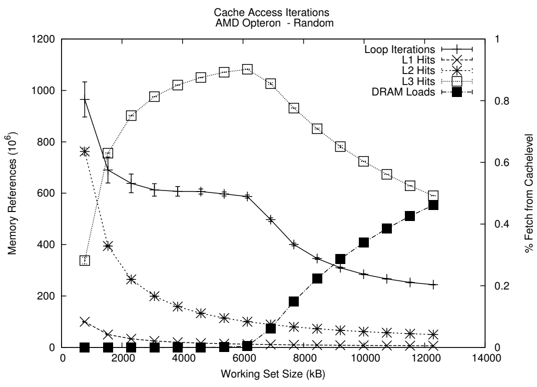

Cache Access Iterations
AMD Opteron - Random
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)

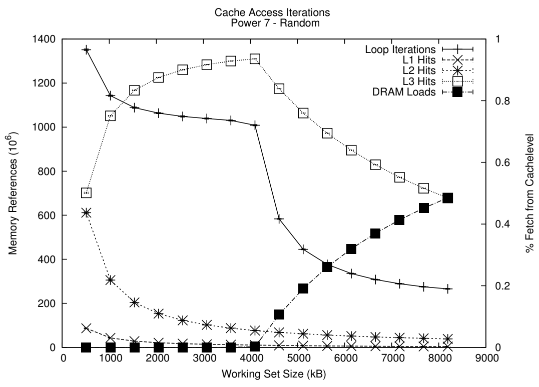

Cache Access Iterations
Power 7 - Random
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)

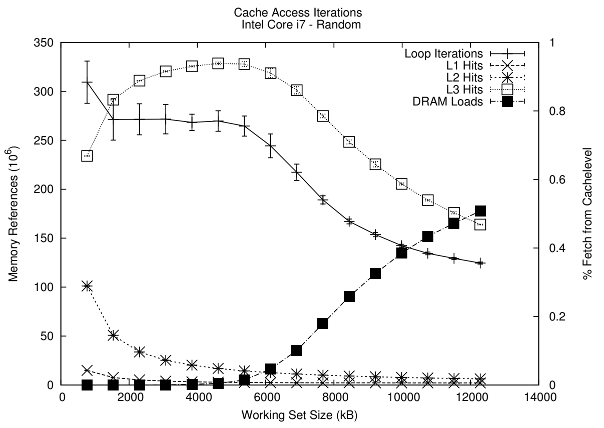

Cache Access Iterations
Intel Core i7 - Random
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)

A.2 Perfect Page-Colouring

Changing the buffer allocation strategy to perform perfect page colouring, results in a much smoother and deterministic instruction throughput curve. Additionally, the CPU's cache hit-rates follow the expected theoretical hit-rate curves. Some variation is still present however. This is attributed to TLB overhead and misses (shown in Section).

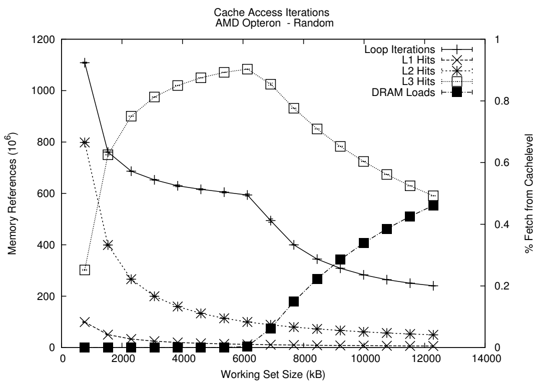


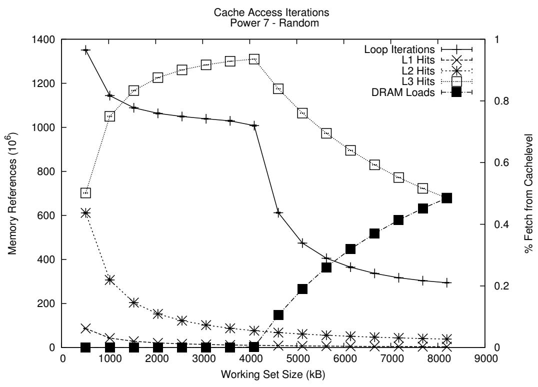


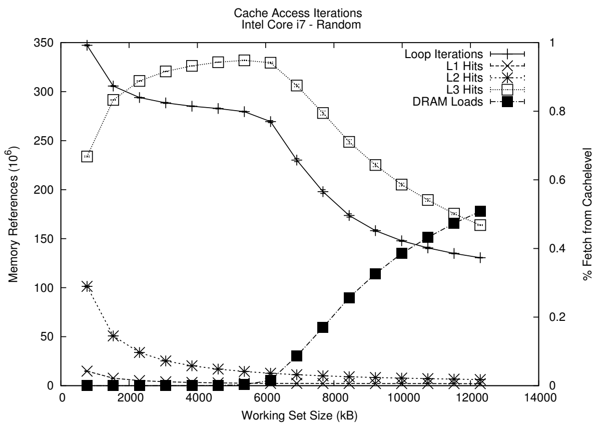


A.3 Hugepages Random

The buffer strategy utilizes Linux's Hugepages, which results in a perfect page-colouring as well as a reduction in TLB misses. As fewer TLB entries are required to cover the buffer, fewer potential misses are incurred.





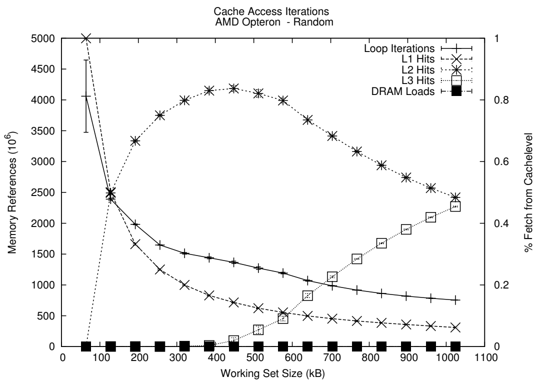


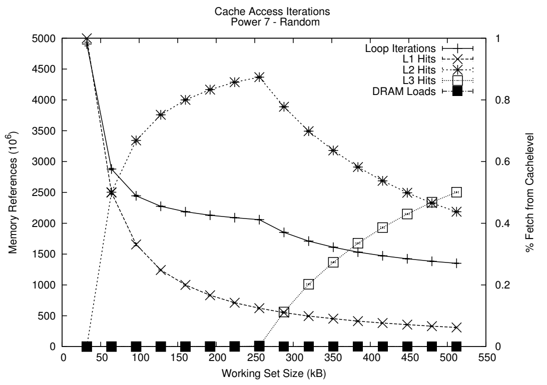
Appendix B

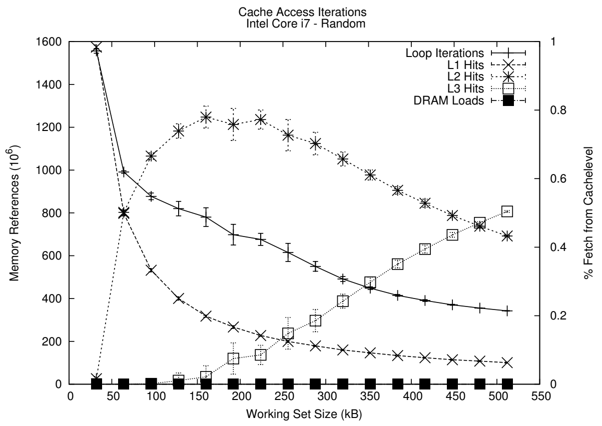
L2 Measurements

B.1 Naive Allocation Scheme

The figures showcase *Cachetest's* output when performing measurements over the L2 cache range. Power 7's larger page size results in a complete covering of the cache with much fewer pages, thus avoiding the TLB's potential negative impact on the instruction throughput. Additionally, the increased page size results in better cache utilization.

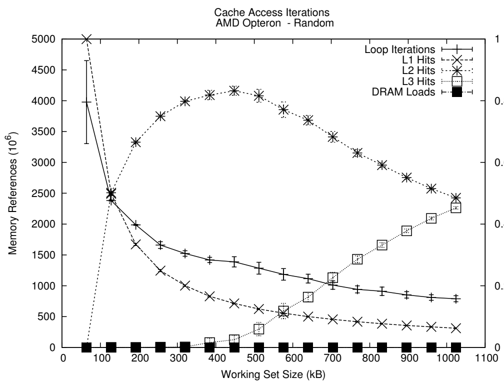

Cache Access Iterations
AMD Opteron - Random
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads

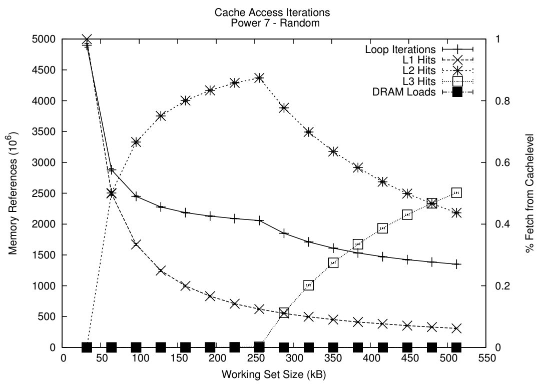

Cache Access Iterations
Power 7 - Random
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads

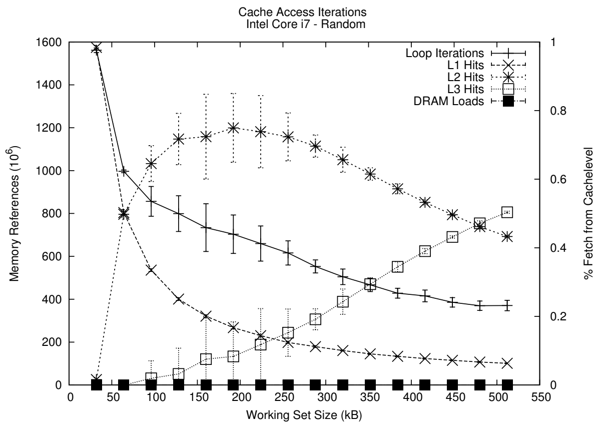

Cache Access Iterations
Intel Core i7 - Random
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads

B.2 Perfect Page-Colouring

Changing the buffer allocation strategy to perform perfect page colouring, results in a much smoother and deterministic instruction throughput curve. Additionally, the CPU's cache hit-rates follow the expected theoretical hit-rate curves. Some variation is still present however. This is attributed to TLB overhead and misses.

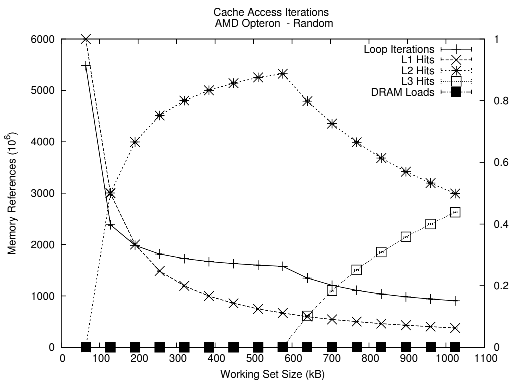


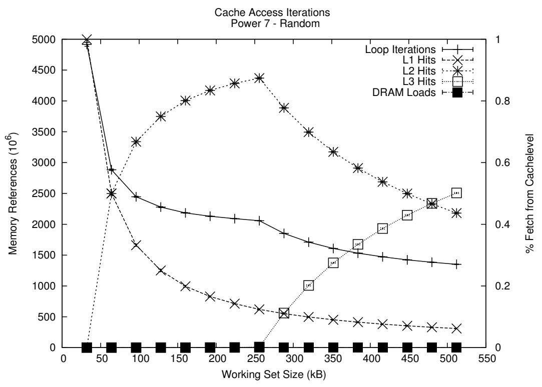


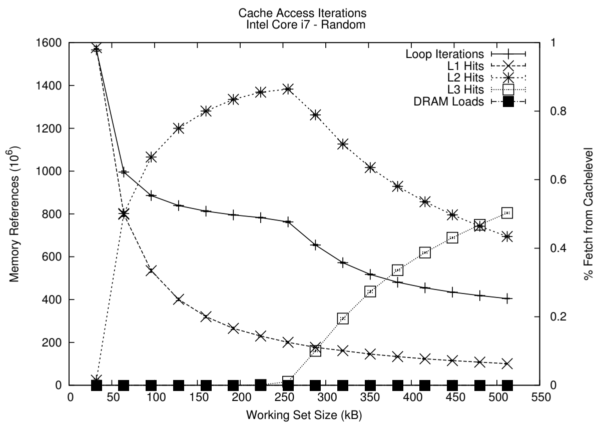


B.3 Hugepages Random

The buffer strategy utilizes Linux's Hugepages, which results in a perfect page-colouring as well as a reduction in TLB misses. As fewer TLB entries are required to cover the buffer, fewer potential misses are incurred. This is particularly evident on the Core i7 architecture, where TLB misses are relatively more expensive.





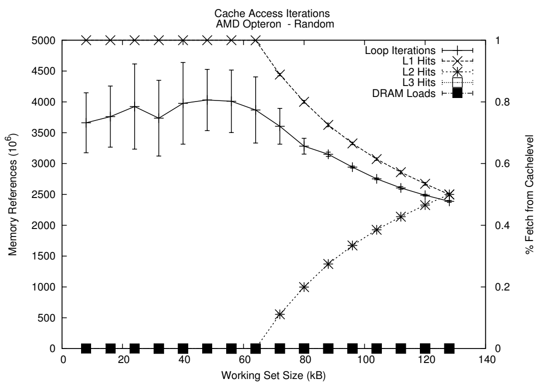


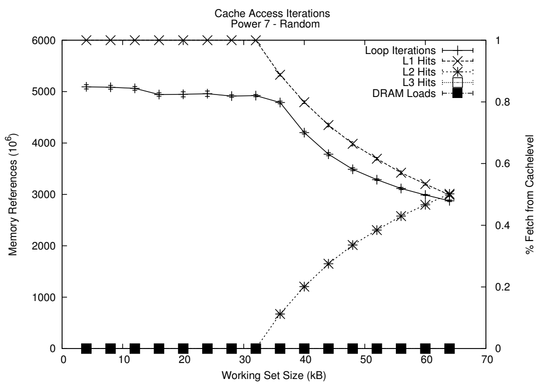
Appendix C

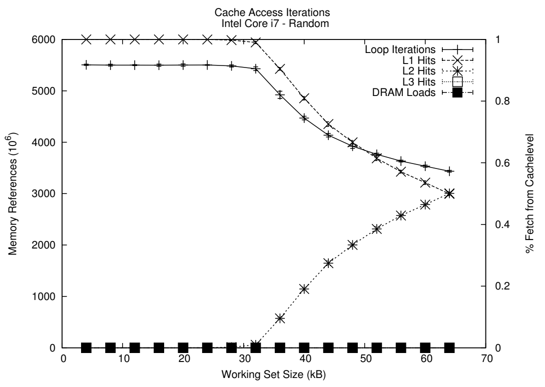
L1 Measurements

C.1 Naive Allocation Scheme

Measurements at the L1 level show the greatest variation in instruction throughput. This is a result of the large penalty associated with a miss in the L1 cache. Virtual memory overhead introduced by TLB misses is also amplified and visible in the variable performance results of the AMD Opteron.

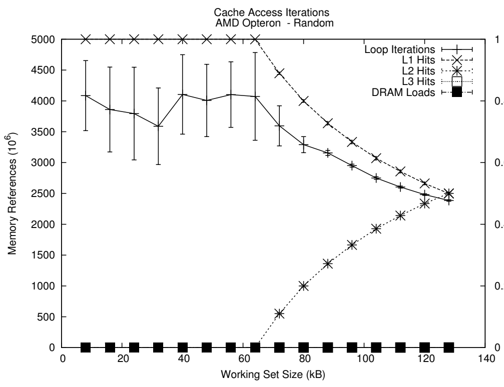

Cache Access Iterations
AMD Opteron - Random
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)

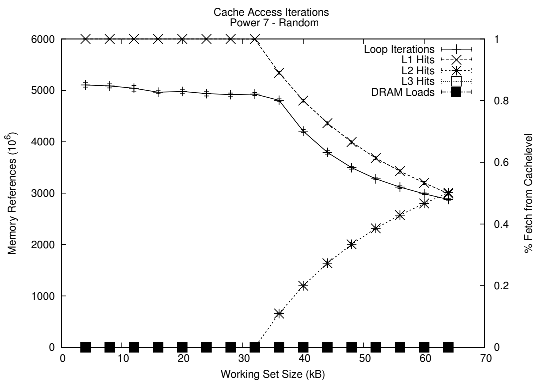

Cache Access Iterations
Power 7 - Random
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)

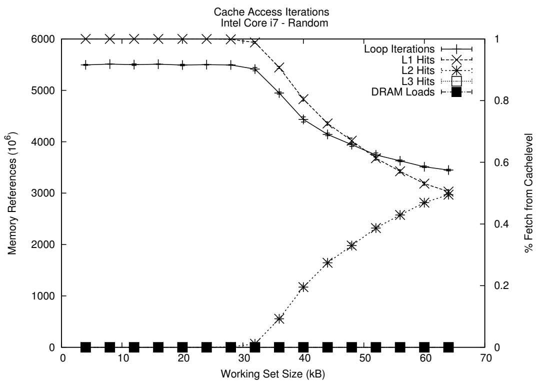

Cache Access Iterations
Intel Core i7 - Random
Loop Iterations
L1 Hits
L2 Hits
L3 Hits
DRAM Loads
Memory References (10^6)
% Fetch from Cachelevel
Working Set Size (kB)

C.2 Perfect Page-Colouring

While no significant reduction in instruction throughput is observed for either architecture, the Opteron CPU still yields large variations in CPI. This is attributed to the CPU's TLB management and is shown to be corrected when utilizing Hugepages.

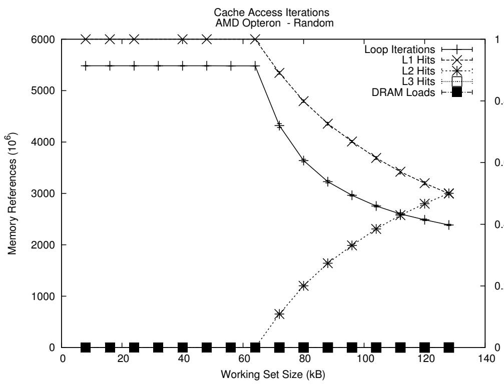


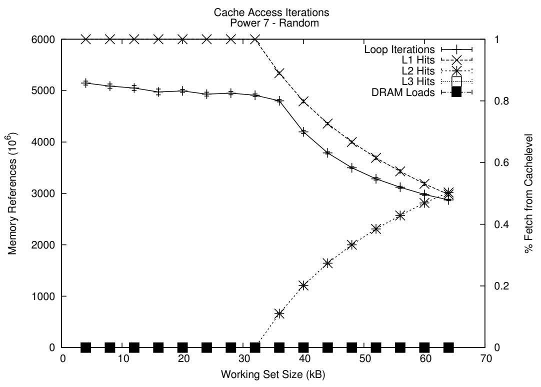


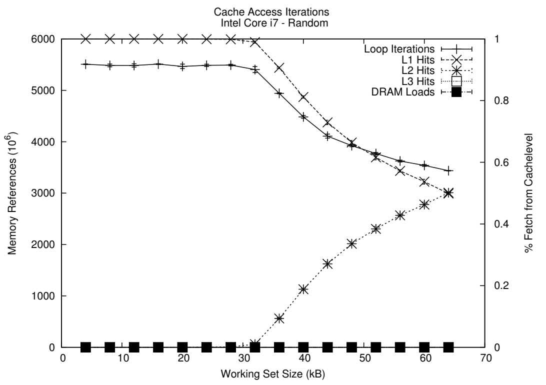


C.3 Hugepages Random

Finally, the buffer strategy utilizes Linux's Hugepages, which results in perfect page colouring as well as a reduction in TLB misses. The AMD Opteron CPU yields the greatest instruction throughput increase.







Bibliography

[1] AMD Corporation. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the 1967 American Federation of Information Processing Societies conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] J. Archibald and J. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transaction on Computer Systems*, 4(4):273–298, September 1986.

[4] R. Azimi, D. Tam, L. Soares, and M. Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS*, 43(2):56–65, April 2009.

[5] S. Bahadur, V. Kalyanakrishnan, and J. Westall. An empirical study of the effects of careful page placement in linux. In *Proceedings of the 36th annual Southeast regional conference*, ACM-SE 36, pages 241–250, New York, NY, USA, 1998. ACM.

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

[7] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *Transaction of the ACM*, 28(4):8:1–8:45, December 2010.

[8] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly Media, Incorporated, 2005.

[9] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[10] C. CaBcaval and D. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 150–159, New York, NY, USA, 2003. ACM.

[11] C. Cascaval, L. DeRose, D. Padua, and D. Reed. Compile-time based performance prediction. In *Languages and Compilers for Parallel Computing*, volume 1863 of *Lecture Notes in Computer Science*, pages 365–379. Springer Berlin / Heidelberg, 2000.

[12] X. Chi, C. Xi, R. Dick, and Z. Mao. Cache contention and application performance prediction for multi-core systems. In *2010 IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS '10, pages 76–86, March 2010.

[13] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[14] K. Cooper and A. Dasgupta. Tailoring graph-coloring register allocation for runtime compilation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 39–49, Washington, DC, USA, 2006. IEEE Computer Society.

[15] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. *SIGARCH*, 39(3):353–364, June 2011.

[16] P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64 – 84, January 1980.

[17] C. Dhruba, G. Fei, K. Seongbeom, and S. Yan. Predicting inter-thread cache contention on a chip multi-processor architecture. In *11th International Symposium on High-Performance Computer Architecture*, pages 340 – 351, February 2005.

[18] U. Drepper. What every prograammer should know about memory. `http://people.redhat.com/drepper/cpumemory.pdf`, 2007.

[19] P. Drongowski. An introduction to analysis and optimization with amd codeanalyst. 2008.

[20] S. Eranian. http://perf.wiki.kernel.org.

[21] S. Eranian. The perfmon2 project. `http://perfmon2.sourceforge.net`.

[22] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A performance counter architecture for computing accurate cpi components. *SIGOPS*, 40(5):175–184, October 2006.

[23] H. Ghasemzadeh, S. Mazrouee, and M. Kakoee. Modified pseudo LRU replacement algorithm. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems*, ECBS 2006, pages 376–382, 2006.

[24] M. Gorman and P. Healy. Performance characteristics of explicit superpage support. In *Proceedings of the 2010 international conference on Computer Architecture*, ISCA'10, pages 293–310, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. *SIGMETRICS*, 34(1):228–239, June 2006.

[26] R. Gupta and C. Chi. Improving instruction cache behavior by reducing cache pollution. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 82–91, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[27] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computer Systems*, 38(12):1612–1630, December 1989.

[28] M. Hocko and T. Kalibera. Reducing performance non-determinism via cache-aware page allocation strategies. WOSP/SIPEW, pages 223–234, New York, NY, USA, 2010. ACM.

[29] T. Huang, Q. Zhong, X. Guan, X. Wang, X. Cheng, and K. Wang. Reducing last level cache pollution through OS-level software-controlled region-based partitioning. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1779–1784, New York, NY, USA, 2012. ACM.

[30] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann, 2008.

[31] T. Jones. http://www.ibm.com/developerworks/library/l-proc/index.html.

[32] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *Transaction of the ACM*, 10(4):338–359, November 1992.

[33] K. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, October 1965.

[34] I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi. Modeling of cache access behavior based on Zipf's law. In *Proceedings of the 9th workshop on Memory performance: Dealing with Applications, systems and architecture*, MEDEA '08, pages 9–15, New York, NY, USA, 2008. ACM.

[35] J. Levon and P. Elie. Oprofile. *A system-wide profiler for Linux systems. Homepage: http://oprofile.sourceforge.net*, 2007.

[36] M. Marko and A. W. Madison. Cache conflict resolution through detection, analysis and dynamic remapping of active pages. In *Proceedings of the 38th annual on Southeast regional conference*, ACM-SE 38, pages 60–66, New York, NY, USA, 2000. ACM.

[37] W. Mathurm. Improved estimation for software multiplexing of performance counters. *MASCOTS, IEEE Computer Society*, 2005:23–34, 2005.

[38] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Model Computer Simulation*, 8(1):3–30, January 1998.

[39] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78 –117, April 1970.

[40] S. A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, pages 162–174, New York, NY, USA, 2004. ACM.

[41] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[42] P. Mochel and M. Murphy. Linux kernel documentation, sysfs: http://kernel.org/doc/documentation/filesystems/sysfs.txt.

[43] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th annual international symposium on Microarchitecture*, MICRO 26, pages 202–213, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[44] J. Navarro, Iyer S., P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *SIGOPS Operating Systems Review*, 36(SI):89–104, December 2002.

[45] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 155–164, New York, NY, USA, 1999. ACM.

[46] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts.* Wiley Publishing, 8th edition, 2008.

[47] A. J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, 4(2):121–130, March 1978.

[48] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 258–269, Washington, DC, USA, 2008. IEEE Computer Society.

[49] D. Tam, R. Azimi, L. Soares, and M. Stumm. Rapidmrc: approximating L2 miss rate curves on commodity systems for online optimizations. *SIGPLAN*, 44(3):121–132, March 2009.

[50] G. Taylor, P. Davies, and M. Farmwald. The TLB slice, a low-cost high-speed address translation mechanism. *SIGARCH*, 18(3a):355–363, May 1990.

[51] D. Wall. Register windows vs. register allocation. *SIGPLAN*, 39(4):270–282, April 2004.