Whose Cache Line Is It Anyway?

Automated Detection of False Sharing

by

Mark Anthony Spear

B.S.E., Princeton University, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2015

Abstract

The abstraction of a cache is useful to hide the vast difference in speed of computer processors and main memory. For this abstraction to maintain correctness, concurrent access to memory by different processors has to be coordinated such that a consistent view of memory is maintained.

Cache coherency protocols are responsible for this coherency, but can have adverse implications for performance. The operational granularity of these protocols is a "cache line" (e.g. 64 bytes). Depending on the data contained in the cache line and the data's access patterns, the coherence can be superfluous and the performance implications severe: Consider the case where each byte within a cache line is exclusively read and written by specific cores and no coherence between cores should be necessary.

My collaborators and I developed a system which detects this phenomenon, known as false sharing, (my thesis), and present a system that can automatically rewrite programs as they are running to avoid the performance penalty. Some parallel programming benchmarks such as linear regression can see up to 3x-6x performance improvement.

Preface

This thesis presents work done in part of a larger system built in collaboration with Mihir Nanavati, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. It has been published as a paper entitled "Whose Cache Line is it Anyway: Operating System Support for Live Detection and Repair of False Sharing", in the proceedings of the 8th European Conference on Computer Systems (EuroSys 2013)[22].

Chapter 4 describes the architecture of the entire system and has been designed collectively with Mihir Nanavati, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. The implementation of the page granularity analysis (Section 4.2.2) belongs to Mihir Nanavati, the byte-level analyis (Section 4.2.3) to Shriram Rajagopalan and Mihir Nanavati, and the final diagnosis (Section 4.2.4) to the author and Mihir Nanavati.

The remapping technique described in Section 4.2.5 belongs to Nathan Taylor and Mihir Nanavati, and has also been published as Cachekata: Memory Hierarchy Optimization via Dynamic Binary Translation[26].

The integration of the performance counters (Section 4.2.1) into Xen and CCBench (Section 5.3) have been implemented in collaboration with Mihir Nanavati.

All other parts of this thesis, unless specified otherwise, are the work of the author.

Table of Contents

List of Tables

List of Figures

Glossary

EPT	Extended Page Tables
HITM	HIT-Modified
HVM	Hardware Virtual Machine
MA	Machine Address
MESIF	Modified Exclusive Shared Invalid Forward (A cache coherence protocol based on MESI)
MESI	Modified Exclusive Shared Invalid (A cache coherence protocol)
MIPS	Millions of instructions per second
MMU	Memory Management Unit
MSR	model-specific register
PA	Physical Address
PEBS	Precise Event Based Sampling
PMU	Performance Monitoring Unit
RDMSR	Read MSR
RDPMC	Read Performance Monitoring Counters
RFO	Read for ownership
VA	Virtual Address
WRMSR	Write MSR

Acknowledgements

This thesis would not have been possible without the guidance and patience of my supervisor, Andrew Warfield, and my collaborators and fellow students in the NSS lab. Over my time a UBC we have worked on many fun projects, both successes and failures, where the fun of the projects was only second to the fun of the scotch and beer after them.

For Kathleen, Dad, Mom, and Mike,
who have wanted this thesis done more than anyone.

Chapter 1

Introduction

Performance of individual processor cores have shown stagnant growth for a number of years, and the current trend toward faster computation is increasing the number of cores [12]. However, this architectural shift requires a simultaneous paradigm shift for programmers who are used to single-threaded programming. Programmers must now be aware of additional potential problems including scheduling, cache coherency, synchronization, communication, resource contention, and various other issues. Many of these issues have been around for years in larger distributed systems, but now they are being faced on individual machines.

CPUs try to hide some of the underlying distributed nature using mechanisms like cache coherence to pretend memory accesses are uniformly consistent. However, programs which are written obliviously to the underlying microarchitecture are prone to be caught by the mismatch between programming model and machine model for memory access performance. False sharing is an example of this class of problem: Ignorance of the hardware's "cache line" granularity for cache coherence can result in an application with unrelated data accessed by separate cores on the same cache line. The coherence protocol will be invalidating data in the various cores' caches, resulting in coherence misses instead of cache hits.

As we will discuss in Chapter 3, designing systems to detect false sharing is not new, but one would typically not want to use these systems due to the impracticality of their performance overhead.

We cannot rely on developers for performance testing for every microarchitecture (some of which have not even been developed yet). Nor can we rely on always having access to source to recompile/modify programs.

A system for detecting false sharing must be low-overhead so that we

can always keep it running, yet precise enough to find the exact pieces of data contributing to false sharing. A dynamic solution is necessary: Plastic. This thesis presents a false sharing detection system which uses a staged approach described in Chapter 4. This design achieves these goals by using some low-overhead advanced processor features (performance counters, event sampling) combined with sparing temporary use of high-overhead detection (page table manipulations and breakpoints) and analysis algorithms.

The performance of the end-to-end system is demonstrated on various workflows in Chapter 5. Our results show 3-6x speedup in workloads which exhibit significant amounts of false sharing.

Chapter 2

Background

The work presented in this thesis depends on a number of specifics about the x86 (and related) architectures: Cache coherence (and the related phenomenon of *false sharing*), performance counters, and virtualized memory address spaces implemented with page tables.

2.1 Cache Coherence and False Sharing

Multiple-core parallel computing systems have private per-core caches, but present a coherent view of memory to all execution units. In order to achieve a coherent view, cache coherency protocols such as Modified Exclusive Shared Invalid (MESI) and Modified Exclusive Shared Invalid Forward (MESIF) to provide a view which is easier for programmers to reason about.

These coherence protocols are optimized for the common case where most data that is shared is being read, potentially even read-only. Writes to non-exclusive cache lines can be particularly slow, as old data needs to be invalidated everywhere, and potentially repopulated.

In Figure 2.1, consider a single cache line of 8 DWORDS (64 bytes), C, with multiple accessors: Core A and Core B. If the same data in cache line C is being read and written by Core A and Core B, then we have an instance of *true sharing*. A lock or a reference counter are common examples of true sharing. By necessity, the cache line will be in each core's caches at various times. If the data being read from cache line C by Core A is disjoint from the data which was written by Core B (e.g. as in cache line C-2), and the data is only incidentally on the same cache line, then we have an instance of *false sharing*. It is a case of unfortunate luck that the granularity of

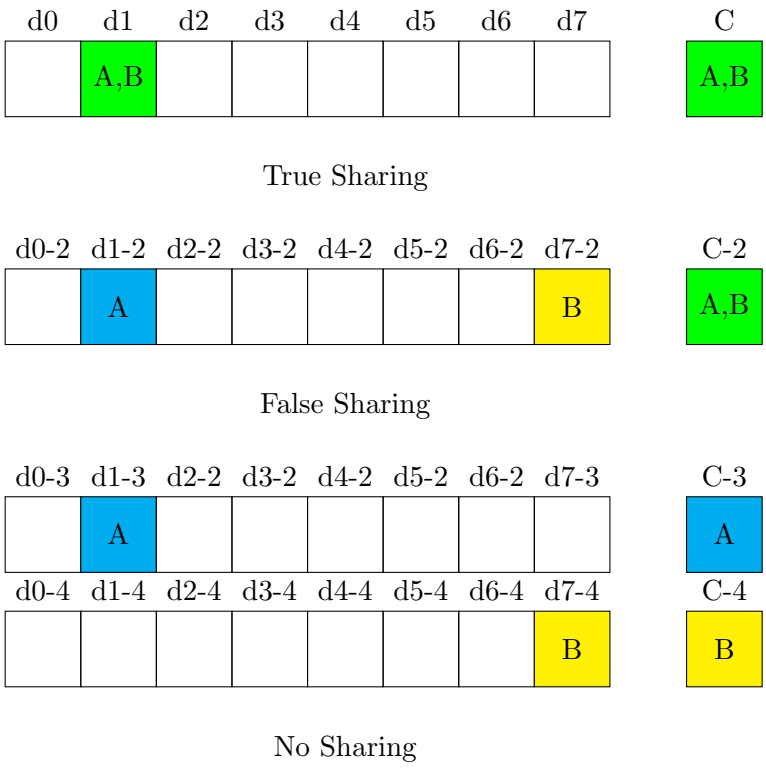

Figure 2.1: True, false, and no sharing of data: Only in the case of false sharing does a cacheline granularity view of accessors differ from the finer-grained view.

data isolation (a cache line) is too coarse for the actual coherency needs during a false sharing scenario. If the data were laid out different, e.g. in separate cache lines C-3 and C-4, then each line would be protected from invalidations due to the other core's execution.

In the absence of multiple cores and other code being executed, since core A continually reads its data in cache line C, one would expect it to remain in cache. However if Core B writes to cache line C, the data in Core A's cache becomes invalidated, and then Core A's reads would result in a *coherence miss*. This happens because the cores negotiate what data should be stored in their caches to preserve consistency. The cache coherence protocol will be performing invalidations and generating coherence misses, even though

the various cores don't need the rest of the contended data on the cache line. As we will see in Chapter 5, this can result in significant performance implications.

2.1.1 Cache Coherence on the x86 Architecture

The x86 processor cache architecture remains relatively unchanged since Intel's release of the Nehalem microarchitecture in 2008. In order to account for the increasing speed differential between CPUs and main memory, modern processors have multiple layers of caches. The Nehalem architecture has multiple cores per physical socket, each of which has a private L1 and L2 cache. All cores on the same socket share a larger common L3 cache. Between these different caches, replicated on all sockets in the system, and main memory, there is significant effort required to keep a coherent view of data at any given time.

The MESIF cache coherency protocol is responsible for maintaining consistency between caches on multiple cores. Each core's cache lines each have a state associated with them, and the shared L3 cache acts as a directory [10]. Simultaneous reads by multiple cores cause cache lines to exist in a "Shared" state. Any write requires exclusive ownership. "Modified" state represents exclusive ownership but with data which has not been written back to main memory. "Exclusive" is exclusive ownership, but the data in the cache is also in main memory. Entering either of these states mean the same cache line (if present) will become "Invalid" in other cores.

In order for a modified cache line to be read by another core, the data must propagate to a common location. For a local (on-socket) core's cache lines, the modified data will be updated in the packages shared L3 cache. However, a cross-socket read of modified data will require the data to be written back to main memory. Latencies of contentious memory access thus vary significantly depending on the topology of the cores involved [21].

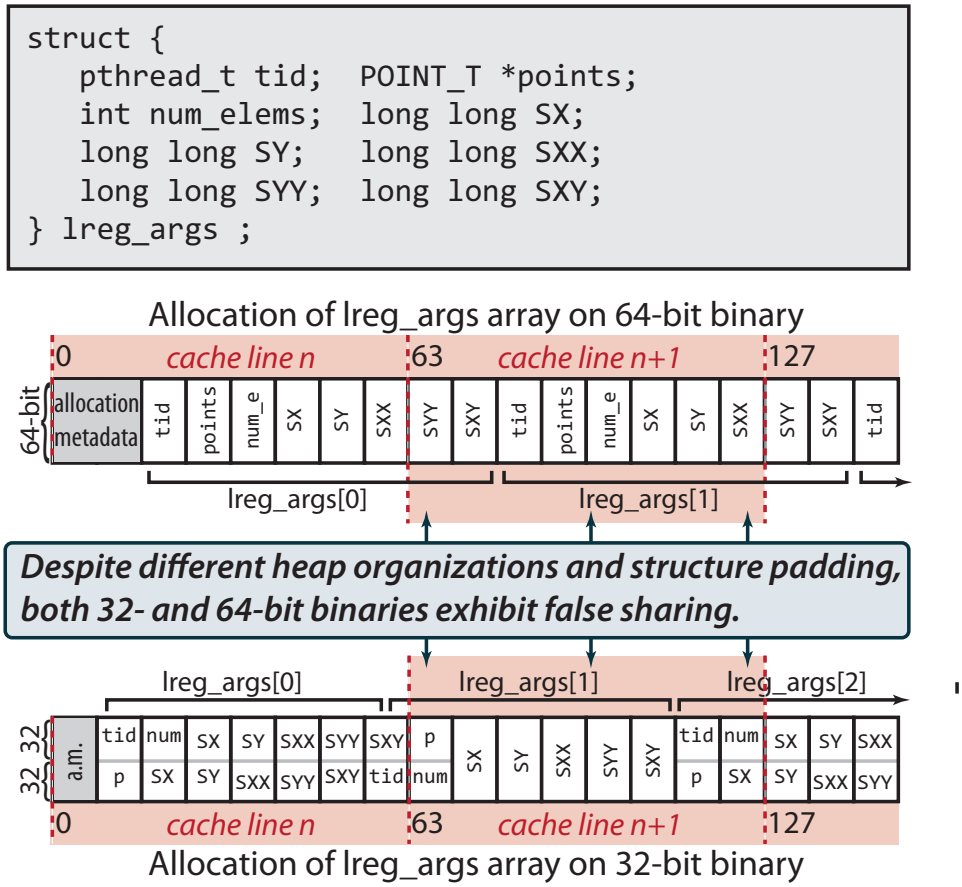

Figure 2.2: False sharing in Phoenix's linear_regression benchmark.

2.1.2 A Concrete Example of False Sharing

The "linear regression" test of the Phoenix [25] parallel benchmark suite's linear is a popular example of false sharing [16, 32]. Figure 2.2 shows the `lreg_args` structure which is responsible for false sharing. Each thread of the linear regression application stores its state in one of these structures, and they are all stored consecutively in memory in an array. The program has each thread continually updating these structures in a tight loop as the benchmark runs. Because the threads run independently on their own struct for the majority of execution time, there is no *true sharing* occuring.

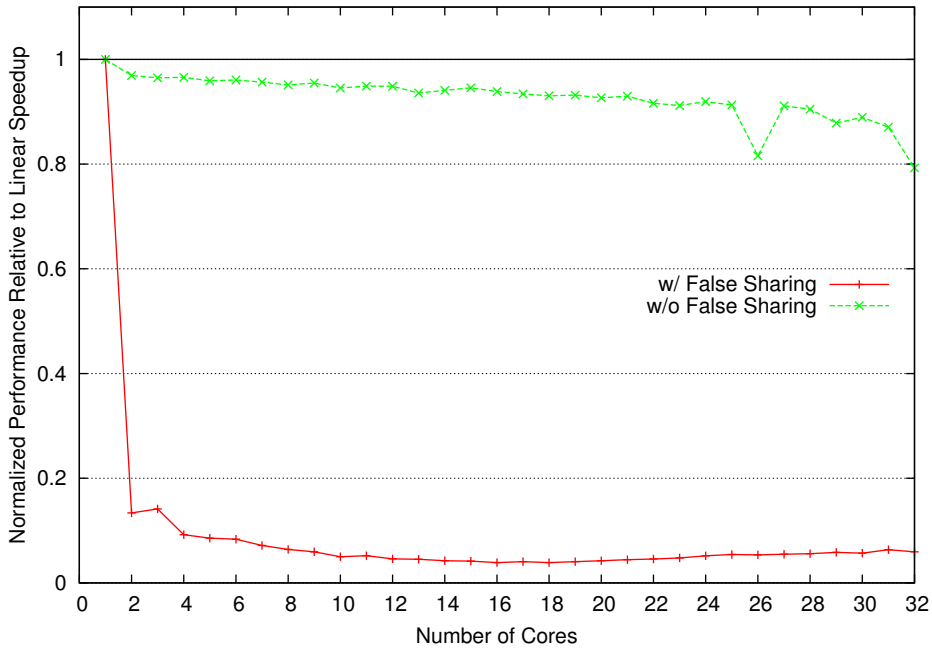

Figure 2.3: False sharing of lreg as a product of cores.

However, since the structures are packed onto the same cache line, they do exhibit *false sharing*. In fact, the false sharing is so severe that the program performs worse as it is given more cores to execute on. Figure 2.3 compares the program's scalability against a version that has the data aligned to eliminate false sharing. Without false sharing, the program scales nearly linearly. With false sharing, updates must be committed to memory before ownership can be transferred, reducing the performance of the cache hierarchy to that of main memory.

False sharing depends on many dynamic properties in a system. Workload is one: per-thread structures in the linear regression example receive continuous updates from all threads, while examples in the Linux kernel often feature a single variable with frequent updates surrounded by read-mostly data [7]. The organization of the program binary is another: Figure 2.2 also shows the same source file compiled as both 32-bit and 64-bit

binaries. Despite identical source and identical cache organization, the nature of false sharing is different: one case results in a 52-byte structure that tiles poorly across cache lines, where the other produces an ideally sized 64-byte structure, but then misaligns it because of allocator metadata. This is not a compiler or language specific issue: For example, Java 7 is known to optimize out manually-added cacheline padding [28] which was added by the programmer to try to avoid false sharing.

2.1.3 Significance

False sharing is a significant issue in cache coherent systems, and is one that will increase in significance in the future. Experimental hardware architectures [11, 27] and operating systems [3] have begun explorations in systems without cache coherence but the requirement of coherence for general purpose computing will likely not vanish any time soon. A scalability analysis of the Linux kernel [7] shows that current architecture will continue to work well with greater degrees of parallelism on existing models. Researchers have also shown that is it practical to scale cache coherence protocols to highly parallel architectures [19].

There are numerous examples of false sharing problems that are at larger scales than current x86 server architectures. David Dice has discussed false sharing issues with the Java garbage collector implementation on a 256-core Niagara server [8]. Similarly, the previously mentioned Linux kernel scalability investigation [7] revealed a number of cache contention issues relating to both true and false sharing.

2.2 Performance Counters

Hardware performance counters are implemented using extra special-purpose registers on the processor to track performance-impacting behaviours of the hardware. There are many kinds of events which can be counted, but the set of events can vary from one processor to another. Cross-platform use of performance counters can be difficult since the kinds of events one cares

about may be different from architecture to architecture. In recognition of this difficulty, Intel has created a number of "Architechtural" counters which behave consistently across microarchitectures and will continue to be supported on subsequent revisions [13].

There are only a few of these counter registers available, and the specific number varies by platform. For example Intel's Nehalem microarchitecture has 4 performance counters available. While few in number, the counters are programmable and thus a wide combination of different things can be monitored simultaneousy.

The Intel Nehalem PMU guide [31] discusses the specific details about how performance counters are configured via the Write MSR (WRMSR) instruction writing to specific control registers. The counters themselves are also implemented as model-specific registers (MSRs) and thus can be read with the Read MSR (RDMSR) instruction, but a faster instruction exists for reading performance counters: Read Performance Monitoring Counters (RDPMC). One of the most important factors about using performance counters is that the overhead of using them is negligible [6].

2.2.1 Precise Event Based Sampling

Precise Event Based Sampling (PEBS) is an extension to hardware profiling facilities available on Intel processors used to capture more information profiling workload behaviour.

PEBS-enabled performance counters can be configured to periodically sample information about the processor state when the counter overflows a preconfigured threshold. The machine state captured includes the values in various registers: flags, instruction pointer, general purpose registers, and some specialized fields which were added in Nehalem.

2.3 Page Tables

Another hardware technology leveraged by Plastic is hardware page table support for virtual memory.

All modern x86 CPUs support virtual memory, and hypervisors require virtualizing virtual memory [1].

A Memory Management Unit (MMU) is responsible for mapping a Virtual Address (VA) space (usually per-process) to a Physical Address (PA) space. Some processors have means of extending this mechanism to support virtualization in hardware: Extended Page Tables (EPT) is Intel's version of this technology. This exta level of indirection allows the hypervisor to control PA to Machine Address (MA) mappings on a per-virtual-machine level. For systems without hardware support for MMU virtualization, there is a software technique which can be used: Shadow Page Tables. Shadow page tables are under the control of the hypervisor, and are the composition of the VA-PA and PA-MA mappings.

In addition to just allowing for mapping one address to another, page table entries have associated permissions bits which we take advantage to carefully control access to memory during the detection pipeline in Plastic.

Chapter 3

Related Work

3.1 Memory Performance and False Sharing

Many memory allocators use private heaps, which helps avoid false sharing due to the potential performance implications [2]. Allocators can be susceptible to two kinds of false sharing: active false sharing (when objects allocated by different threads can wind up on the same cache line) and passive false sharing (when objects freed by a remote thread are immediately allocatable by the same remote thread). TCMalloc is susceptible to both kinds of allocator-generated false sharing [2]. Hoard is a memory allocator designed for scalability (and avoiding false sharing is an explicitly considered factor), though it is still subject to passive false sharing [2, 5].

3.2 Detecting False Sharing via Instrumentation

Pluto uses Valgrind to keep track of load and store events across different threads and performs worst-case analysis of possible false sharing, while imposing 2 orders of magnitude of overhead [9].

Sheriff [16] detects false sharing (with no false positives) and approximately 20% overhead on average. Their system makes assumptions about the usage of the pthread threading library which can break correctness guarantees if the assumptions are violated, e.g. if the application uses lock-free data structures.

Predator samples memory accesses using instrumented binaries and predicts false sharing on "virtual cache lines", which can detect potential false sharing even if it did not actually occur with the given object alignment and hardware cache configuration [17]. This approach has 6x performance

and 2x memory overhead. While predator can predict different-alignment false sharing, the theoretical predications may not be relevant for a given compiled binary/system.

All instrumentation-based approaches are quickly discounted for a general-purpose false sharing detection and mitigation system since modifying the source may not be an option.

3.3 Simulation

Cache simulation methods can provide data about behaviour in systems with various memory hierarchies and coherence algorithms. Cmp$im (a simulator which uses Pin [18]) can simulate workloads running at 4-10 Millions of instructions per second (MIPS), and is three orders of magnitude faster than more accurate simulators [15]. However, this is still quite slow, and thus unlike Plastic, Cmp$im [15] is only suitable for development and testing environments, and cannot reasonably run on general purpose systems.

3.4 Sampling Event Counters

Intel Performance Tuning Utility [30] uses performance counters to identify the presence of cache line contention. However, it suffers from a high false positive rate as it does not attempt to distinguish if the contention is caused by true sharing, or if it actually is false sharing. It does not analyze the performance implications of specific data being falsely contended, nor integrate with a solution attempt to alleviate performance impacts.

3.5 Dynamic Analysis for Performance

Feedback driven optimization is a technique which can be used to inform compiler optimization decisions based on representative workloads of a running system [24]. An instrumented version of code is run to collect profiling information, which is used as an input for subsequent optimized recompilation. Such a system can not be used on generic applications, for which

source code may not be available. Plastic is a part of a larger system that (once it detects false sharing) remaps data and rewrites code for generic applications which exhibit false sharing [22, 26]. As shown in this thesis, these techniques for detecting and remapping falsely-shared data can lead to 3x-6x performance improvement in severe cases of false sharing.

Chapter 4

Design and Architecture of False Sharing Detection

4.1 Design Overview

Figure 4.1 shows the high-level architecture of the false sharing detection pipeline in Plastic. Precise detection of false sharing is expensive in terms of the performance hit on the running system. However, various more performant heuristics can be used to detect potential false sharing. Each phase of Plastic uses increasing expensive techniques for identifying (potential) false sharing, but on a narrower and narrower scope. This detection funnel is sufficiently lightweight on one end that it can continuously run without adverse performance impact, and is sufficiently precise to detect significant false sharing so that a dynamic binary rewriting system can significantly increase system performance. PEBS, however, is not integrated into this funnel due to platform limitations described in Section 4.2.6.

4.2 System Architecture

4.2.1 Performance Counters

As mentioned in Section 2.2, performance counters can track performance-impacting behaviour of hardware. However, these are still physical registers, and thus must be shared by all code executing on the system. This sharing includes multiple processes running under an operating system, and multiple virtual machines running under a hypervisor. These supervisor-level systems manage what is being counted, and do the appropriate saving and

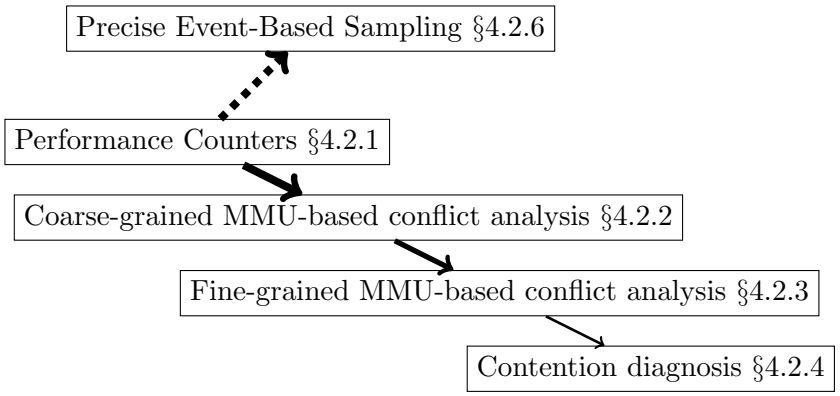

Figure 4.1: System Architecture

restoring of the register state when the currently running applications and VMs switch. Intel has published guidelines for sharing access to the Performance Monitoring Unit (PMU) [14], however for the purposes of Plastic we currently assume we are the only part of the system using the PMU.

Performance Counter Name	Description
SNOOP_RESPONSE. HITM	A cache snoop request to a particular core was a hit and the value was modified from what is in memory.
MEM_UNCORE_RETIRED. OTHER_CORE_L2_HITM	A HITM occurred and modified data was found in another core's cache.
EXT_SNOOP. ALL_AGENTS.HITM	A similar HITM counter for Intel Core 2 architectures.
INST_RETIRED. ANY	The number of instructions retired (completed).
L2_TRANSACTIONS. RFO	RFO requests for prefetches and demand misses.

Table 4.1: Performance counters useful for detecting false sharing.

A number of the important performance counters we keep track of are listed in Table 4.1. HITM counters are relevent to detecting cache line contention because they indicate the requested data was in a remote cache line, when the remote cache line is in a modified state. A high ratio of these

relative to the total number of executed instructions can indicate potential false sharing, but also occur when true sharing is occuring [29].

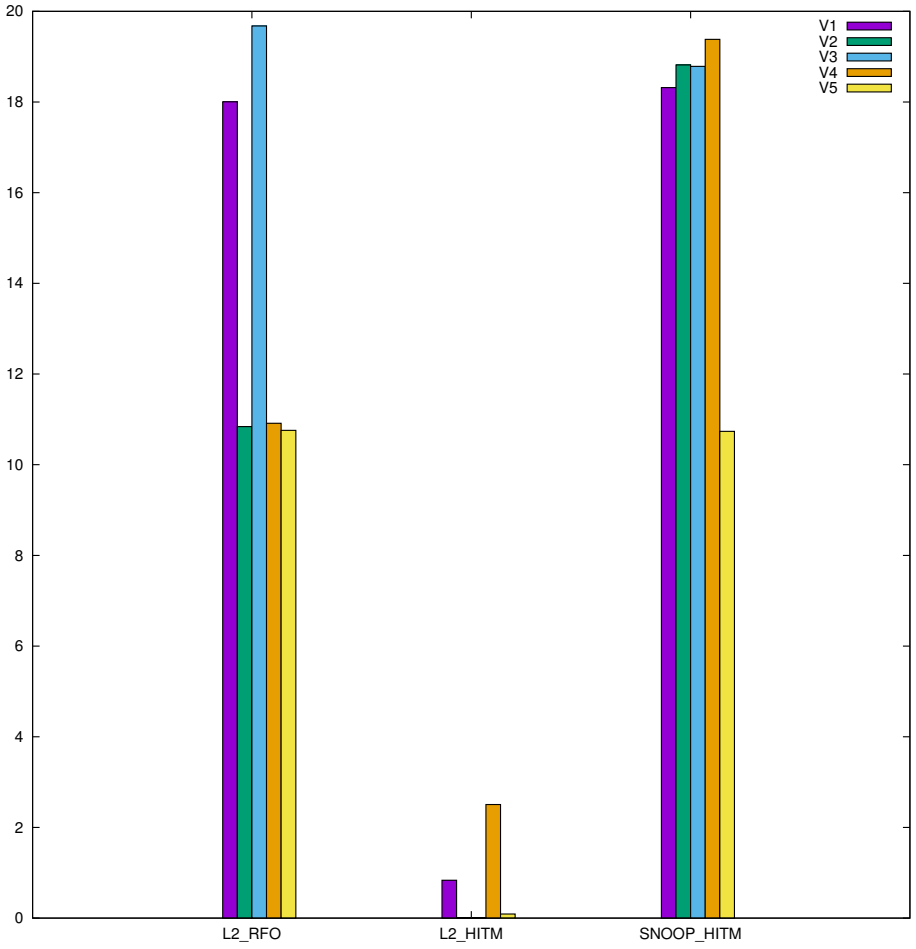

Figure 4.2: Performance counter values after running Listing A.1

In Figure 4.2 we show several false-sharing-related performance counters tested against 5 modes of false sharing generated by the code found in Listing A.1. `SNOOP_RESPONSE.HITM` responds quite well to all modes of false sharing, so we use it as our primary indicator of potential false sharing.

In order to disambiguate the potential false sharing with true sharing,

when this high-HITM-per-instruction scenario is encountered we turn on more heavyweight instrumentation. This allows us to precisely determine if this truly is a case of false sharing (and if so: where the false sharing is occuring), without subjecting typical system execution to the slower detection stages.

4.2.2 Page Granularity Analysis

Xen is a virtualization platform that manages the physical hardware of machines and provides a common abstraction to guest virtual machines. It manages per-virtual-machine address spaces using mechanisms like shadow pages tables or EPT. In Plastic we extend this concept to per-core page tables, though operating systems can use similar techniques with per-thread page tables [4, 16]. Each of these per-core pages begins by being marked as *protected*. When data on the page is accessed, the permissions are updated to allow the access. The type of access (i.e., read vs. write) is recorded along with the page. Results and permissions are periodically reset, so that evaluation is done in a series of epochs. Since contention requires at least one writer and one other accessor, the output of this stage is the set of pages which are in the intersection of the write bitmap for one core and the access bitmap for all other cores. This contention could still be a case of *true sharing* at this point, so further analysis is required to differentiate the class of contention.

4.2.3 Byte Level Analysis

Byte level analysis is done by causing every relevent memory access to fault. The set of contended pages detected in Section 4.2.2 are configured to have the processor fault on every access. The fault handler logs the access, restores the appropriate permissions for the page in order for the access perform as expected, and sets up a single-step breakpoint. As soon as the memory-accessing instruction is retired, the permissions are again reconfigured to cause the next access to generate a fault. Thus, every access will generate a fault and the memory access will be logged.

Since sharing is more closely associated with different threads than different cores (e.g. a thread migrating from Core A to Core B is not really "sharing" data even if we detect accesses on two different cores) we distinguish threads by logging an appropriate segment register: fs(x86_64) or gs(x86).

The output of this phase is the set of accessed bytes, along with the identity of the accessors and the mode of access.

4.2.4 Diagnosis

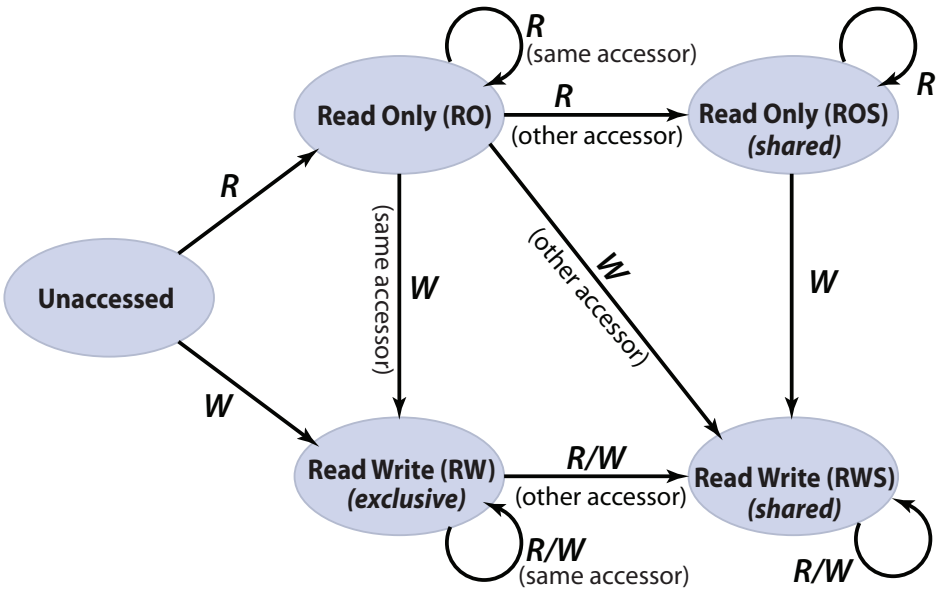

Figure 4.3: Sharing status of a byte in the access log.

Feeding the byte level access log into the state machine depicted in Figure 4.3 allows Plastic to make a determination of the sharing state for each byte. A contended cache line has at least one writer (cache lines with bytes in either the RW or RWS states), and multiple accessors. False sharing is happening if there are bytes with different accessors (as determined by the logged thread identifiers). Regions within the cache line are grouped ac-

cording to accessor, and groups with writes are relocated to their own cache lines on the isolated page by the remapping engine (Section 4.2.5).

4.2.5 Remapping

After determining that false sharing is occuring at a particular location, Plastic forwards that information to a remapping engine [23, 26]. This engine safely moves contended data onto separate cache lines, and rewrites executing code to refer to data in the new location. Figure 4.4 demonstrates the repositioning of data onto different cache lines. The performance benefits of removing false sharing in this way are demonstrated in the evaluation in Chapter 5.

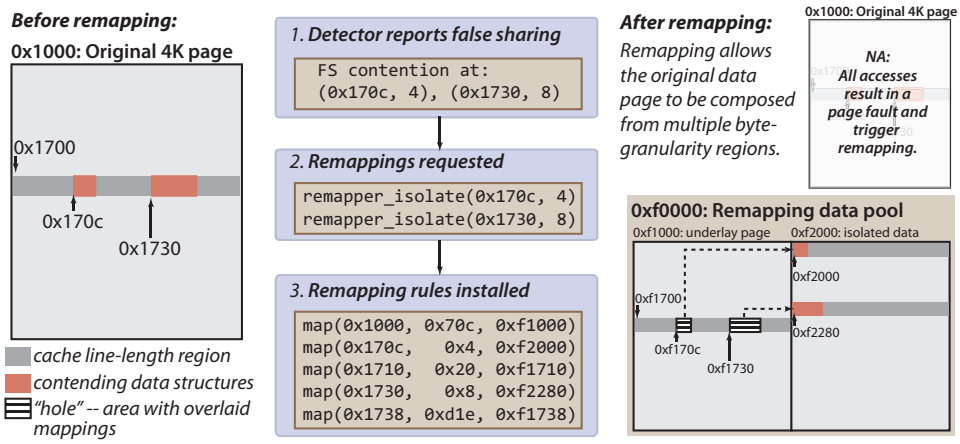

Figure 4.4: Independent pieces of data on a single cache line can cause false sharing. To avoid the negative performance impact, they can be remapped onto separate cache lines.

4.2.6 Precise Event Based Sampling

With the exception of PEBS, Plastic's detection proceeds in a stepwise manner. PEBS is exceptional in that, while it has the potential for helping with detection, the Nehalem-era implementation does not provide the additional data necessary to help identify false sharing.

For false sharing detection, the interesting information currently recorded in Nehalem would be the instruction pointer. However, this only really helps for inspecting the code, as we need to determine the location of the data being potentially falsely shared. It's insufficient to know some instructions which contribute to false sharing (especially since PEBS actually records the "at-retirement" state of the instruction and thus it records the "IP + 1" (where "1" is in reference to the subsequent instruction, rather than a literal byte offset). Consider the case where we believe data is contributing to false sharing. If we wish to remap the data to another location in the address space, we must ensure that all instructions which would ever attempt to access this data will be redirected to the correct location. Sampling instructions may give us an idea of the cost of remapping, but in itself cannot give an exhaustive list of places to remap.

Future work could use PEBS to determine whether to proceed to subsequent detection stages. Looking at samples of instructions that are accessing the candidate-false-shared data may be able to filter out un-rewritable code. i.e. even if this definitely is false shared data, we may not be able to do anything about it, so don't bother trying to make the precise determination.

The specialized fields relating to load latency recorded by the PEBS mechanism could be used in the future (Section 6.1).

Chapter 5

Evaluation

Plastic is evaluated on a dual socket, 8-core Nehalem system with 32 GB of memory. Each processor is a 4-core 64-bit Intel Xeon E5506 with private, per-core L1 and L2 caches and a shared L3. Plastic is implemented as an extension to Xen 4.2 and runs a Linux 2.6.32.27 kernel as Dom0. All tests are run inside an 8-core Ubuntu Hardware Virtual Machine (HVM) guest. First, we describe our selection of performance counters, then show the detailed execution of a false sharing workload under Plastic, followed by an evaluation of Plastic across a range of applications.

5.1 Performance Counter Selection

As the first phase of our detection pipeline, careful selection of events that are representative of potential false sharing is crucial to success. Candidate counters were selected by analyzing Intel's platform documentation [13, 30, 31] and evaluating behaviour under many workloads. The workloads contained known examples of false sharing and applications which did not exhibit false sharing. Candidate performance counters were selected on the basis of how effectively they discriminated between these two cases. Figure 5.1 shows an example counter which is only significantly incremented in applications which exhibit false sharing. (Many similar graphs are left out for space reasons.)

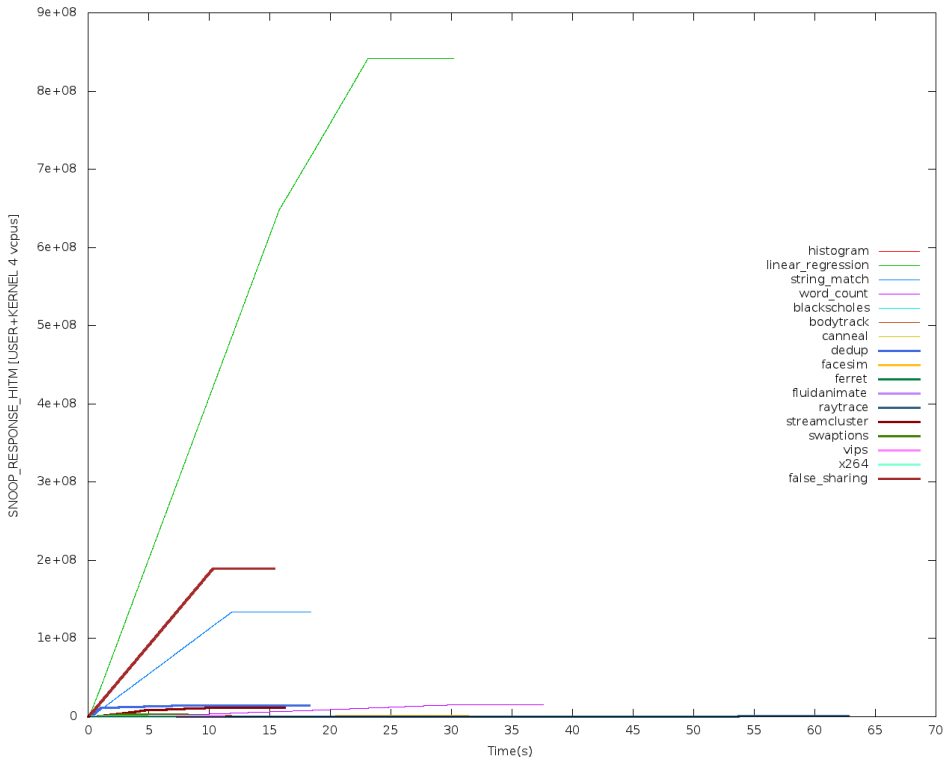

Figure 5.1: `SNOOP_RESPONSE.HITM` values over time while running various applications.

5.2 Detection Times and Execution Under Plastic

Figure 5.2 demonstrates the behaviour of an example workload which exhibits false sharing [1] running under Plastic. Because of the false sharing in the application, the workload immediately exhibits low performance due to the costly coherence invalidations. Plastic quickly identifies that false sharing is occurring (within 2 seconds). After the locations of false sharing are given to the rewrite engine of Plastic, the HITM rate significantly decreases and the throughput jumps significantly for the remainder of the program's

[1] Specifically, this is the linear regression benchmark from Phoenix.

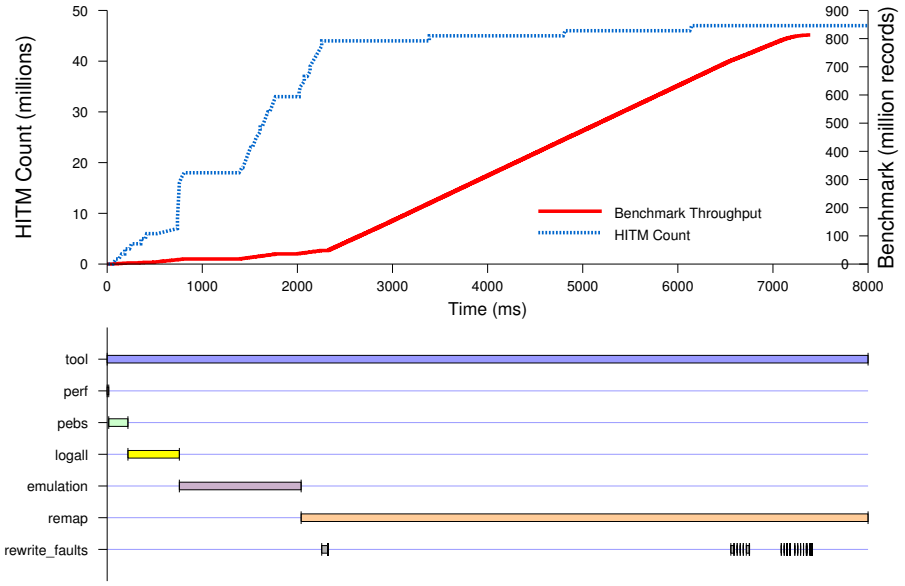

Figure 5.2: Linear regression running under Plastic.

execution.

5.3 CCBench

False sharing is a well-understood phenomenon from the perspective of hardware implementations and those programmers who actively think about hardware-level operation. Often its grave performance impact is mitigated during development due to careful profiling and testing. However, sometimes real workloads are beyond the scale of parallelism tested. CCBench is a microbenchmark suite designed to simulate similar troublesome workflows at a much smaller scale.

Table 5.1 taxonomizes different kinds of memory access patterns, observed in these real workloads, responsible for scalability bottlenecks and the performance of the corresponding microbenchmarks with and without Plastic.

Name	Description	Examples	Fixable?
`fs_independent`	Multiple accessors to independent variables (At least one writer)	Linear Regression in Phoenix [25] `spinlock_pool` in Boost [20] Bookeeping in the Java GC [8]	Yes
`fs_mixed`	Shared read-only data co-located with contended data	`net_device` struct in Linux [7]	Yes
`bitmask`	Bitmasks and flags	`page` struct in Linux [7]	No
`true_share`	Shared read-write data	Locks and global counters	No

Table 5.1: Microbenchmarks in CCBench. The fixable column denotes whether it can be fixed by simply remapping memory.

5.4 Shared Memory Benchmarks

In Figure 5.3 we see the impact of Plastic on a number of workloads in CCBench, Phoenix, and Parsec.

Programs exhibiting clear, pessimal false sharing, like *linear regression* and the synthetic workload, enjoy a significant performance improvement while running under Plastic. Those without false sharing show low overhead, and the overall performance is indistinguishable within the error bars.

5.5 Effect of Rewriting Optimizations

Figure 5.4 shows how the performance of the linear regression benchmark scales from 1 to 8 cores, normalized against idealized linear speedup. Four versions of the program are shown:

- a version with false sharing executing normally

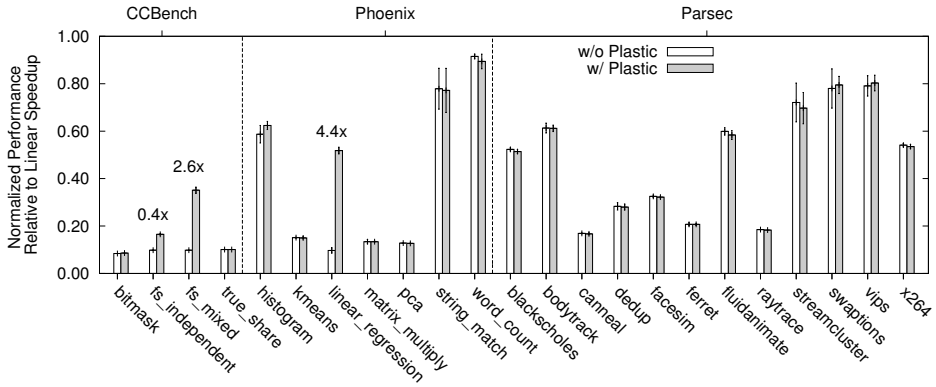

Figure 5.3: Performance of Phoenix, Parsec and CCBench suites running with Plastic.

- an unmodified version running under Plastic without optimized rewriting
- an unmodified version running under Plastic with optimized rewriting
- a version of the program where false sharing was fixed by hand

With an increase in cores, performance of the unmodified version decreases exponentially. As the number of coherency misses increases to the point that cores are constantly stalling, the multi-threaded version is slower than a single threaded version, even in terms of absolute run time. The source optimized version shows almost linear speedup. The version running unoptimized instrumentations does not suffer from false sharing, but has to perform enough comparisons that it is even slower than the version with false sharing. Using specialized rewriting under Plastic, yields a 3-6x performance improvement over the regular version.

An important takeaway is the observation that remapping can potentially have a negative impact on performance as well. We currently assume that the optimized instrumentation to remove false sharing is always desirable. Future work should take into account the rate of non-Plastic instructions retired as an indicator that Plastic should roll back the remappings and run with the original code and original data layout.

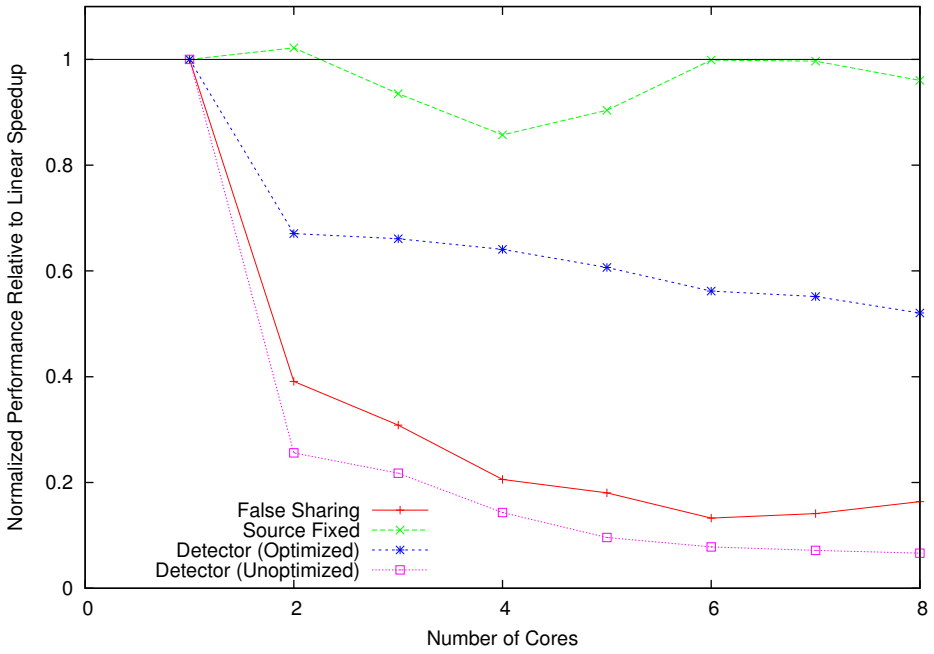

Figure 5.4: Normalized Performance of Linear Regression with and without Plastic.

Chapter 6

Discussion

Plastic is a functional proof of concept demonstrating that dramatic false sharing can be identified and fixed in a manner which does not have significant performance impact on workloads which do not exhibit false sharing. For benign workloads, the temporary performance impact only occurs during short detection phases (if at all, as most workloads tend not to even progress into the slower phases of detection).

Therefore it is feasible to consider the Plastic system as a desirable layer of system software, protecting applications from unexpected performance characteristics of the underlying hardware consistency model.

With additional performance counters, one can imagine other hardware "worst case scenarios" being detected and mitigated in similar ways. The utility of this system can be thought of like a post-compiler binary optimization step, similar to the HotSpot JVM's adaptive optimization, but generalized to x86 / x86_64 instructions rather than Java bytecode.

6.1 Future Work

Plastic is currently part of a system aimed at dynamically improving the behaviour of existing binaries at runtime. However, the detection pipeline could be integrated with debug symbols and development toolchains to allow developers to fix the problem at the source. This method of fixing false sharing would prove a larger performance win compared to the rewritten binary, and would be increasingly relevant for an extended detector which can identify other sources of poor performance which *cannot* be automatically mitigated.

If Plastic were to continually run alongside an arbitrary system, it should

be made to adhere to Intel's recommendations for sharing the PMU [14]. These recommendations include "Agents must be able to be configured and function without PerfMon resources" which would mean Plastic would be idle while the PMU is in use by other agents. Unfortunately running in the hypervisor is insufficient to guarantee that we always have access to the PMU since firmware may also use counters: When discussing not laying claim to an in-use counter: "This includes the VMM. The VMM should ensure any counter in-use by firmware is not disabled."

In post-Nehalem architectures when PEBS facilities can export memory locations corresponding to HITM events, future stages of the pipeline could be restricted to even fewer memory pages to track. This could decrease detection overhead even further. PEBS integration would also be useful for other cache analyses unrelated to false sharing by providing a gauge on how accessing some data takes many cycles to be loaded due to poor cache utilization. The remapping engine may be able to determine a more optimal arrangement of data so that a larger portion of the working set is available in cache.

More architectural performance counters would be necessary for cross-microarchitecture compatability of any system that would rely on them. While detection/analysis during development could reasonably run only on a restricted set of platforms, widespread availability of counters would serve two goals: 1) Being able to deploy a system like Plastic on end-users systems, and 2) Since development and deployment environments are different, they could be running into different sets of hardware-inflicted performance issues.

Chapter 7

Conclusion

Plastic demonstrates that an operating system (or hypervisor) can take responsibility for intelligent resource management of shared hardware resources being used inefficiently by software. Software can generate pathological use cases for caches with significant negative performance implications, by being unaware of low level implementation details like hardware cache line size. Plastic shows that low overhead detection of hardware issues can be combined with a larger system [22] in order to dynamically fix these cases, resulting in 3x-6x improved throughput.

This work intelligently uses pieces of data exported by the hardware, and can be used to motivate exposing more counters and more information about data or code which is causing hardware to press up against the inefficient parts of its design. In the case of Plastic we show that some classes of issues can be automatically alleviated, though in general feedback could be directed to software developers to work around these hardware limitations when possible.

Bibliography

[1] Ole Agesen. Software and hardware techniques for x86 virtualization. *Palo Alto CA, VMware Inc*, 2009. → pages 10

[2] Martin Aigner, Christoph M Kirsch, Michael Lippautz, and Ana Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. *arXiv preprint arXiv:1503.09006*, 2015. → pages 11

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, 2009. → pages 8

[4] T. Bergan, N. Hunt, L. Ceze, and S.D. Gribble. Deterministic process groups in dos. In *OSDI*, 2010. → pages 17

[5] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000. → pages 11

[6] Georgios Bitzes and Andrzej Nowak. The overhead of profiling using pmu hardware counters. 2014. → pages 9

[7] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *OSDI*, 2010. → pages 7, 8, 24

[8] David Dice. False sharing induced by card table marking. *David Dice's Weblog*, Feb 2011. → pages 8, 24

[9] Stephan M. Gunther and Josef Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *WBIA*, 2009. → pages 11

[10] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. → pages 5

[11] John Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, Devon Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010. → pages 8

[12] Wei Huang, Karthick Rajamani, Mircea R Stan, and Kevin Skadron. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro*, (4):16–29, 2011. → pages 1

[13] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual*. 043 edition, 2012. → pages 9, 21

[14] Peggy Irelan and Shihjong Kuo. Performance monitoring unit sharing guide. `https://software.intel.com/sites/default/files/ea/95/30388`. → pages 15, 28

[15] Aamer Jaleel, Robert S. Cohn, Chi keung Luk, and Bruce Jacob. CMP-Sim: A pin-based on-the-fly multi-core cache simulator. In *MOBS*, 2008. → pages 12

[16] Tongping Liu and Emery D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 3–18, New York, NY, USA, 2011. ACM. → pages 6, 11, 17

[17] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. Predator: Predictive false sharing detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 3–14, New York, NY, USA, 2014. ACM. → pages 11

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, 2005. → pages 12

[19] Milo M K Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *To appear, CACM*, 2012. → pages 8

[20] mcmcc. false sharing in boost::detail::spinlock_pool? `http://stackoverflow.com/questions/11037655/false-sharing-in-boostdetailspinlock-pool`, June 2012. → pages 24

[21] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *PACT*, 2009. → pages 5

[22] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T Meyer, William Aiello, and Andrew Warfield. Whose cache line is it anyway?: operating system support for live detection and repair of false sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 141–154. ACM, 2013. → pages iii, 13, 29

[23] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. Whose cache line is it anyway?: Operating system support for live detection and repair of false sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 141–154, New York, NY, USA, 2013. ACM. → pages 19

[24] Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. Feedback-directed optimizations in gcc with estimated edge profiles from hardware event sampling. In *Proceedings of GCC Summit 2008*, pages 87–102, 2008. → pages 12

[25] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society. → pages 6, 24

[26] Nathan Bryan Taylor. Cachekata: memory hierarchy optimization via dynamic binary translation. 2013. → pages iii, 13, 19

[27] C. Thacker. Beehive: A many-core computer for fpgas (v5). *Unpublished Manuscript*, Jan 2010. → pages 8

[28] Martin Thompson. Re: kernel + gcc 4.1 = several problems. `http://ondioline.org/mail/cmov-a-bad-idea-on-out-of-order-cpus`. → pages 8

[29] Avoiding and identifying false sharing among threads. `http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/`. → pages 16

[30] Intel performance tuning utility. `http://software.intel.com/en-us/articles/intel-performance-tuning-utility/`. → pages 12, 21

[31] Intel microarchitecture codename nehalem performance monitoring unit programming guide (nehalem core pmu). `https://software.intel.com/sites/default/files/76/87/30320`. → pages 9, 21

[32] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong,

and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *VEE*, 2011. → pages 6

Appendix A

Selected Code Listings

Listing A.1: Selected variations of code triggering false sharing

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE 1
#endif
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <pthread.h>

unsigned long *globalvar;
int quit = 0;
unsigned long iterations = (1UL << 32);

void *worker(void *t)
{
 long id = (long)t;
 unsigned long i = iterations;
 int reader = id & 1;
 int off = id / 2;
 unsigned long localvar = 0;
 unsigned char *ptr = (char *)(globalvar + id);

 while(i--)
```

```
 {
#if V1
    //simple increment
   globalvar[id]++;
#elif V2
   // increment via temporary stack variable
   localvar = globalvar[id];
   localvar++;
   globalvar[id] = localvar;
#elif V3
   //update part of 8 byte word
   ptr[id]++;
#elif V4
   //alternating reads & writes
   (i & 1) ? (localvar = globalvar[id]) : (++globalvar[id]);
#elif V5
   //always read or always write
   reader ? (localvar = globalvar[off]) : (++globalvar[off]);
#endif
 }
}

#define PIN_THREADS 1
int main(int argc, char *argv[])
{
   pthread_t *threads;
   pthread_attr_t *attr;
   cpu_set_t cpuset;
   int i, nthreads;

   nthreads = sysconf(_SC_NPROCESSORS_ONLN);
   if (nthreads < 0) {
     perror("failed to get number of online processors");
```

```
    return -1;
  }
  iterations /= nthreads;

  posix_memalign((void *)&globalvar, 4096, 4096);
  memset(globalvar, 0, 4096);
  threads = malloc(sizeof(pthread_t) * nthreads);
  attr = malloc(sizeof(pthread_attr_t) * nthreads);

#if PIN_THREADS
  //pin main thread to cpu0
  CPU_ZERO(&cpuset);
  CPU_SET(0, &cpuset);
  sched_setaffinity(0, sizeof(cpuset), &cpuset);

  //pin threads
  for (i = 0; i < nthreads; i++) {
    CPU_ZERO(&cpuset);
    CPU_SET(i, &cpuset);
    pthread_attr_init(&attr[i]);
    pthread_attr_setaffinity_np(&attr[i], sizeof(cpuset), &cpuset);
  }

  for (i = 0; i < nthreads; i++)
    pthread_create (&threads[i], &attr[i], worker, (void *)i);
#else
  for (i = 0; i < nthreads; i++)
    pthread_create (&threads[i], NULL, worker, (void *)i);
#endif

  for (i = 0; i < nthreads; i++)
    pthread_join (threads[i], NULL);
```

```
    free(globalvar);
    free(threads);
    free(attr);
    return 0;
}
```