Cachekata: Memory Hierarchy Optimization via Dynamic Binary Translation

by

Nathan Bryan Taylor

Bsc., The University of Alberta, 2009

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

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

April 2013

© Nathan Bryan Taylor, 2013
Abstract

As hardware parallelism continues to increase, CPU caches can no longer be considered a transparent, hardware-level performance optimization. Adverse cache impact on performance is entirely workload-dependent and may depend on runtime factors. The operating system must begin to treat CPU caches like any other shared hardware resource to effectively support workloads on parallel hardware.

We present a binary translation system called Cachekata that provides a byte-granular memory remapping facility within the OS in an efficient manner. Cachekata is incorporated into a larger system, Plastic, which diagnoses and corrects instances of false sharing occurring within running applications. Our implementation is able to achieve a 3-6x performance improvement on known examples of false sharing in parallel benchmarks.
Preface

The work presented in this thesis is part of a larger system, built in collaboration with Mihir Nanavati, Mark Spear, 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).
# Table of Contents

Abstract ................................................................. ii

Preface ................................................................. iii

Table of Contents ...................................................... iv

List of Figures ........................................................ vi

Glossary ................................................................. vii

Acknowledgments ....................................................... viii

1 Introduction ........................................................ 1

2 Related Work ......................................................... 4
   2.1 Binary Translation systems ...................................... 4
   2.1.1 BT for sandboxing and protection ......................... 4
   2.1.2 BT for introspection .......................................... 5
   2.1.3 BT for just-in-time compilation ............................ 5
   2.2 BT for performance optimization .............................. 6
   2.2.1 Memory as bottleneck ....................................... 6

3 Design and Architecture ........................................... 7
   3.1 System Architecture ........................................... 8
   3.1.1 Xen ........................................................... 8
   3.1.2 Remap rules ................................................ 8
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>System architecture</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>False sharing in Phoenix’s linear regression benchmark.</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>False sharing of linear regression as a function of cores.</td>
<td>16</td>
</tr>
<tr>
<td>5.1</td>
<td>Byte-granular remapping allows data to be transparently isolated on separate cache lines.</td>
<td>18</td>
</tr>
<tr>
<td>5.2</td>
<td>Rewritelets for a single instruction code segment.</td>
<td>21</td>
</tr>
<tr>
<td>6.1</td>
<td>Linear regression running under Plastic.</td>
<td>28</td>
</tr>
<tr>
<td>6.2</td>
<td>Normalized performance of linear regression.</td>
<td>29</td>
</tr>
<tr>
<td>6.3</td>
<td>Normalized performance of benchmarks running natively and under Plastic.</td>
<td>30</td>
</tr>
</tbody>
</table>
Glossary

BT binary translation
CMOV conditional move
DRAM dynamic RAM
EPT extended page tables
HITM HIT-Modified
LBR last branch record
IR intermediary representation
JIT just-in-time compilation
MESI Modified Exclusive Shared Invalid protocol
P2M physical to machine translation
SRAM static RAM
VA virtual address space
VCPU virtual CPU
Acknowledgments

This work would not have been possible without the guidance of my supervisors, Andrew Warfield and Bill Aiello. Their unwavering support, willingness to indulge my academic dilettantism, and advice in matters scholastic, vocational, and humorous have been a great influence on me during my time in grad school. I would also like to acknowledge my fellow students in the NSS lab, too numerous to name here, with whom I’ve collaborated on lab projects and enjoyed many a pint of beer or glass of scotch.

Heinlein was right: specialization is for insects. I’m equally grateful for the constellation of friends I’ve made both through the Computer Science Graduate Students’ Association and also in the broader university community through the UBC Swing Kids, the UBC Dance Club, and the AMS Bike Kitchen. I am a better person for having known every one of you.
For my parents, Bryan, Karen, and Phyllis, who have supported me always, in all ways.
Chapter 1

Introduction

Cache rules everything around me — Wu-Tang Clan

The narrative of systems software has been one of weaving convenient lies to applications and users. Typically, such falsehoods serve to write the cheques that the underlying hardware can’t cash. Preemptive multitasking in a single-processor environment maintains the illusion of concurrency. Virtual memory and paging exposes a contiguous and fully-addressable memory space to every running process. And virtualization platforms allow the full software stack to be independently replicated across a single physical machine. Underneath the bottom of the software stack, though, is the computer’s physical hardware, and there exists here another set of lies that abstract away unpleasant realities that even the most sadistic software developer would prefer to ignore. There’s no better-known example of this than the modern multicore processor’s wide and deep memory hierarchy. Armed with a loose set of assumptions about locality of reference, on-chip caches exist to amortize the heavy cost of accessing main memory; all this takes place under the nose of the programmer, who need not know whether a memory access hits any cache in order to write correct programs.

However, that the memory model is so successful at being transparent means insufficient dues paid to the cache hierarchy by programmers result in a font of performance bottlenecks. Benchmarks timing accessing on-chip caches versus main memory[16,22] demonstrate the severity of cache misses; Hennessy and Patterson go so far as to change the unit of measurement from clock cycles to nanoseconds
when main memory is considered. The software/hardware disconnect is only exacerbated in the multicore era, where inter-core cache coherence protocols execute independently of the intentions of software developers, who themselves have a dim understanding of what the underlying hardware is doing.

There is much that compiler refinements and programmer education can do to help. Modern compilers are aware of laying memory out in a cache-friendly manner, and of course, there is no substitute for code refactoring by a memory-sensitive developer. However, the nontrivial manner in which hardware maps cache lines to sets makes inter-cache locality difficult to reason about at compile-time. Further, implementing a data structure that, in the general case, plays nicely within the cache hierarchy is a challenging feat that requires a large amount of person-hours and domain-specific expertise[20]. Lastly, unlike memory pages, the size and number of caches is variable between machines; there is no guarantee that assumptions about the system in which a binary is compiled will hold on the ones where the program is actually run. As a result, it behooves us to consider improving memory access patterns at runtime.

As we discuss in Chapter 2, binary translation (BT) systems have been used in a wealth of areas, but it is natural to concede that dynamic recompilation for performance gains sounds counter-intuitive. This thesis argues that rather than emitting microoptimizations or other code transformations, it is more fruitful to consider BT as a mechanism to transform memory layout.

A principal obstacle to implementing BT-based systems is handling the innumerable corner cases and idiosyncrasies of modern processors, such as the baroque yet ubiquitous x86-64 architecture. While unacceptable in contexts such as a code sandboxing system, a BT system intended for performance improvement is free to ignore rewriting code segments where the code integrity is potentially compromised. As a result, despite making no claims of comprehensiveness and completeness with respect to its code analysis, we consider the system described in Chapter 3 to be capable of running production workloads.

While the described system is general and may be applied to any problem requiring memory remapping, Chapter 5 introduces the running example of mitigating false sharing in a collection of parallel synthetic and benchmark workloads. This chapter introduces the procedure by which x86 page protection is exploited to
generate and trap to specialized code blocks we call rewritelets, and describes in detail a rewritelet’s structure and invariant requirements.

The end-to-end performance of the system is demonstrated in Chapter 6, where a false sharing detection and mitigation system based on the earlier-presented rewriting system is described. In our results, we show a 3-6x speedup in our false sharing-exhibiting workloads.
Chapter 2

Related Work

2.1 Binary Translation systems

We emphasize that BT is in itself not a contribution of this work. Indeed, we leverage an existing BT library, DynamoRIO[12], heavily in this thesis, though, certainly, the academic and open source communities have provided alternative frameworks[18, 27, 31, 34] that we might have used. DynamoRIO was chosen because of its good performance, ease of integration into other systems, and its intermediary representation (IR) being tightly-coupled with the x86 platform.

BT systems, intended for either general use or as a library or plugin, or tuned for a specific intent, are prevalent in the literature already. Any of these systems could also be replaced with a full-system emulator[1, 7]; however, performance under these systems is bad enough that they are typically not considered appropriate for production settings.

2.1.1 BT for sandboxing and protection

BT is often used in a security context, providing runtime checks to ensure isolation for untrusted programs. Often such systems achieve their goal by restricting the program’s behaviour both within and outside its address space. It may interpose on entry points of system calls[18] to limit access to the outside world, and/or place stronger constraints on the integrity of the program’s control flow to enforce how the program modifies its own state[17, 26] (particularly the state of the BT system
itself). Often, deprecated or unused-by-userspace processor features are abused to better enforce the latter requirement[18, 42].

2.1.2 BT for introspection

Runtime introspection is a problem that BT lends itself well to. In such systems, instrumentation is compiled from a higher-level representation and woven into the executing program. Typically, introspection-based tools are used for bug-finding[13, 31, 34], although the slightly-contradictory notion of using BT for improving performance is not unheard of[36].

While certain tools expose instrumentation as a callback mechanism to execute on certain events such as a memory write[31], others treat instrumentation as a more general mechanism, iterating over streams of basic blocks[8, 34]. Owing to our particular focus on memory accesses, we are in spirit more aligned with the former than the latter. Most instrumentation systems operate on a particular userspace program. However, like our work, full-system instrumentation systems built atop virtualization platforms[13, 36] or within the OS[35] have been built as well.

2.1.3 BT for just-in-time compilation

BT can be considered as a mechanism for just-in-time compilation (JIT), although the literature often uses the former term to refer to translation between low-level representations, as opposed to JIT’s implication of a source-to-bytecode or bytecode-to-native transformation. Indeed, Aycock[5] identifies the earliest seminal JIT implementations as being for highly dynamic languages like Lisp[32] and APL[2], where the distinction between code and data blurs. The tradition continues to this day, where modern dynamic languages such as JavaScript[19] and Python[10] enjoy JIT implementations.

That said, JIT for static, traditionally-compiled languages came on the scene as early as the mid '70s[21], where the notion of optimizing hot code blocks was formalized. However, the runtime optimizations tended to be simple in nature. This sort of performance-based BT manifested itself later in an offline manner with feedback and profile-directed optimization. More recently, LLVM[27], a compiler
suite increasing in popularity, offers a JIT and runtime optimizer. Additionally, kernel probes have been JITted into kernelspace\[^{35}\].

### 2.2 BT for performance optimization

While BT has provided solutions to innumerable problems, some of which have been described above, applying it to performance issues seems counter-intuitive. While it’s true that JITs have shown performance improvement, this is typically versus the high cost of running an interpreted language. What’s worse, the complexity of making sophisticated optimizations does not lend itself well to BT; DynamoRIO \[^{12}\]’s optimizations – exchanging \textit{inc} for \textit{add} instructions – were arguably trivial. While there do exist exceptions, notably inlining function pointer calls or unrolling loops, performance improvements under JIT instrumentation is an unintended side effect \[^{18}\]. Indeed, the translation step itself can have sensitive performance side effects; jumping to a compiler module can maul the processor’s instruction cache, or, in certain systems, necessitate an expensive context switch.

#### 2.2.1 Memory as bottleneck

The amount of effort hardware designers have placed laying layers of cache between memory and the processor suggests the severity of the memory bottleneck problem. Indeed, one of the three oft-quoted guidelines for performance software is “don’t touch memory”\[^{1}\]. For the OS designer’s part, caches have long been considered the dominating factor in access costs, even over address spaces\[^{9}\]. And, for the application developer’s part, despite attempts at presenting a programmable cache to the program \[^{25}\], all this is handled automatically and is opaque.

Previous work\[^{11, 24}\] addresses detection of memory bottlenecks at runtime but typically does not focus on repair. We show that in addition to detection, low-overhead, online mitigation is also feasible using our generic remapper.

\[^{1}\] The other two are, helpfully, “don’t touch the disk” and “don’t do work”.
Chapter 3

Design and Architecture

Correcting memory bottlenecks during software development and testing is difficult, as pathological conditions may not manifest until well after deployment. There are examples of cache-line bottlenecks that manifest themselves at scales larger (but not much larger) than current x86 server architectures; false sharing in a parallel garbage collector has been observed on a 256-core Niagara server. The problem is furthered in the realm of managed languages: language runtimes make opaque to end application developers the manner in which memory is laid out; Java 7, for instance, is known to optimize out manually-added cache-line padding [39].

These observations led us to approach the memory hierarchy as an OS responsibility, similar to the manner in which virtual memory or process scheduling is managed by a central piece of software. Modifying the manner in which data is located or accessed in order to alleviate contention must yield a net performance gain, and should work on unmodified binaries without exploiting known compiler idioms or metadata such as debug information.

The following sections describe the architecture of Cachekata, our system, which allows byte-level regions of a program’s address space to be isolated or otherwise redirected elsewhere.
3.1 System Architecture

3.1.1 Xen

Xen is an enterprise virtualization platform that we use to provide a common hardware abstraction to guest virtual machines and to manage the physical hardware of the machine. Xen’s central core is a microkernel-based hypervisor that handles core functionality such as interrupt handling. The hypervisor defers high-level hardware duties such as device drivers and, in particular for our needs, interrupt handling policy, to a special control VM typically named Dom0.

Despite VM isolation being a core tenet of virtualization, Xen, like many other hypervisors, offers functionality for VMs to have their address space mapped into each others’. In particular, Dom0 may map the memory of any other VM using the xc_map_foreign_range hypercall, and force a modification to the guest’s page table permissions via real_xc_shadow_control.

3.1.2 Remap rules

Cachekata’s remapping engine runs in tandem with a rule generator, both of which run in Dom0. The rule generator monitors the runtime state of a guest VM and, based on the kind of optimization it seeks to make, emits remap rules to be consumed by the remapping engine. A remap rule is a structure indicating to where a block of guest virtual address space should be remapped. In the case of our false sharing-repair system, to be discussed in Chapter 5, the rule generator identifies cache lines it believes to be exhibiting false sharing and emits remap rules to the remap engine to split those cache lines up.

The set of remap rules given to Cachekata are tied to page fault registrations within Xen. During program execution, pages that have remap rules associated with have access permissions removed. On attempted access, control redirects first to Xen and then to the remapping engine. The engine will, in turn, modify the access to point to a location in a data pool as governed by whichever remap rule the faulting address falls within.
3.2 Design overview

Figure 3.1 shows the high-level architecture of our system, which is built as an extension to the Xen virtual machine monitor [6]. Building the system at the hypervisor level was a design choice and not a requirement; we simply required a mechanism by which hardware exceptions could trap to our software. In particular, our remapper tool registers as a fault handler for pages the system deems to be remapped through Xen. This allows mechanism to be built independently of any particular operating system. Additionally, certain kernel data structures have been shown to be sensitive with respect to their cache layout, making the choice of a hypervisor-level system appropriate for kernel-level remapping.

However, a hypervisor-level system does present certain challenges. An OS-level system would simplify some matters involving shared guest pages, which we elaborate on in Chapter 7. Additionally, when cast in the light of a memory interposition system, Cachekata fits nicely into certain extensible microkernel architectures; The Hurd[14], for instance, offers as first-class modules so-called “translators”, which act as pluggable libraries for interfacing with core components such as filesystems, processes, or memory. Finally, despite requiring tight coupling with the application to be rewritten, a similar system could even be im-
implemented entirely in userspace; indeed, early prototypes of this work were tested by linking workload binaries against our library and redirecting control flow via the UNIX SIGSEGV handler.

### 3.2.1 Code generation

Taking a trap to Xen on every access and emulating the faulting instruction with the remapped address would be prohibitively expensive, owing respectively to the heavy cost of two context switches first to Xen and then to the remapper tool, and the complexity of disassembling and reasoning about arbitrary binary code. We address these issues by translating the faulting instruction once and emitting it in a code cache, another region of memory we allocated within the process’ virtual address space (VA) space. In so doing, the modified accesses are done by executing native x86-64 code in the original program’s address space. The latter issue in particular is tackled by performing a partial reconstruction of the faulting instruction’s surrounding control flow into a self-contained gadget called a *rewritelet*. In so doing, hot code paths such as loops are preserved within the rewritelet. As a result, instructions need only be emitted the first time a faulting memory access is encountered, and a hardware fault need only be generated in the first iteration of a hot code segment. Rewritelets are described in detail in Section 5.3.

The code cache and data pool are allocated through an extension of the balloon driver, an in-guest memory management driver that is commonly installed in guest VMs.
Chapter 4

Memory and Sharing

While application developers tend to think of memory as a single, uniform “array” running through the addressbility of the computer, in fact it is a highly-dynamic, quickly-evolving system where memory access times may vary tremendously and physical locations of words may change underneath the feet of a running program.

4.1 The Modern Memory Model

4.1.1 Virtual memory

In order to support transparent multitasking and provide each process the illusion of a fully-addressable memory space, all modern computer systems provide a layer of indirection called virtual memory[15] between user programs and raw physical memory. Physical memory is sliced into pages, which are significant for us in that they are the finest granularity of memory whose access permissions can be tracked by hardware. Each running process contains an associated page table that maps pages’ virtual addresses to their corresponding location in physical memory. Associated with each page table entry is an present bit, indicating whether the given page is resident in RAM or has been swapped to disk, and a set of permission bits setting access control to words on that page.

When a process attempts to access a virtual address from a page marked as absent or whose permission bits deny the access, a page fault interrupt is generated by
the hardware, and control flow redirects to the operating system. If the permissions are valid but the page is absent, the page is copied from disk back into physical memory and the program’s faulting instruction is re-executed. Otherwise, unless explicitly overridden by a signal handler, the process is terminated by the OS.

Virtualization adds another layer of memory indirection in that guest operating systems should not be allowed access to machine-physical memory. As a result, the hypervisor assigns and coordinates for each VM a guest-physical memory space, which the OS in turn multiplexes for its guest-virtual spaces. Without explicit hardware support to account for the separation between guest-physical and machine-physical memory, guest OSs cannot modify their own page table entries without trapping to the hypervisor, which would have to update the guest OS’s tables manually. Modern processors, however, feature hardware support for physical to machine translation (P2M) tables\(^1\). This decouples access permissions for a guest OS and the hypervisor; on a guest page table violation, control will trap to the former, and on a P2M violation, to the latter\(^{28}\).

### 4.1.2 The Processor Cache Hierarchy

The speed of accessing a word of memory from RAM has not been proportionate to the speed of modern processors. This is partially owing to the limited clock speed and bandwidth of the computer’s bus, but also to the fact that fast static RAM (SRAM) is too expensive to produce in quantities large enough to populate full system memory with. Therefore, RAM sticks consist of banks of dynamic RAM (DRAM), which are structurally simpler but require frequent access-stalling refreshes\(^{16}\).

Chip designers have attempted to have the best of both worlds by sticking small caches of SRAM on the processor die with the intention that as much of the current workload’s working set may remain resident there and be asynchronously flushed back to main memory at an appropriate time. For our purposes, we consider a cache to store 64-byte contiguous lines of data, which are indexed by a certain number of high-order bits in an address.

Similar to how software developers writing a hash table must balance complex-

\(^1\)On Intel processors, these are called extended page tables (EPT).
ity and speed, hardware designers must balance the speed of which the right cache line can be found, given an address, with the flexibility that will yield general-case good performance. A fully-associative cache can store any region of memory in any of its lines, but the logic required does not scale to caches much larger than TLBs[16] (which, on the Nehalem architecture, can store a mere 512 entries[22]). On the flip side, a direct-mapped cache simply uses the address’s relevant high-order bits as an index into a sequence of lines, but this only works well if such bits are evenly distributed. Set-associativity has become the happy medium, where each 64-byte region can be mapped into more than one cache line. For the Nehalem’s part, L1-data, L2, and L3 have lines that can be mapped to 4, 8, and 16 places, respectively [22].

We say that the processor has gained a cache hierarchy because, in this model, memory requests first go through at least one level of caches, and main memory is hit only in the event of a cache miss.

A cache on a parallel computing system may be shared between more than one core or local to exactly one. In multicore workloads with a high degree of spatial locality between threads, multiple copies of a cache line may find themselves in more than one core-private cache, as multiple threads may be accessing memory that is closely-spaced to each other. In a cache-coherent system, each core’s view of memory is forced to be consistent; as a result, writing to a shared cache lines will automatically propagate that change to other cores holding that line.

4.2 Cache coherence on the Intel Nehalem architecture

Intel’s cache coherence strategy has remained relatively unchanged since the release of the Nehalem micro-architecture in 2008. Coherence is maintained using MESIF, an extension to the common Modified Exclusive Shared Invalid protocol (MESI)[37] that allows for requests to be forwarded to a particular processor.

In MESI, there are four possible states: cache lines are marked as being exclusive to one cache or shared amongst multiple caches, as well as marked as having been modified or unmodified with respect to the version resident in main memory. MESI considers the “shared, modified” state invalid; if any cache’s copy of a cache line is modified, the line must only exist in a single cache. Multiple readers are
permitted to remain in the “shared” states, but on a write, all other readers are invalidated and the writer becomes exclusive owner of the cache line. A line is written back up the hierarchy only when it is evicted and only if its state tuple is “exclusive, modified”. An equivalent formulation from Hennessy and Patterson[22] merges the tuples into a set of state transitions, which makes constructing the protocol as a state machine more straightforward.

On Nehalem processors, the shared L3 cache acts as a directory[22] to maintain coherence among on-socket cores as well to service requests from other sockets. For local lines, the L3 snoops on main memory writebacks to maintain the freshness of its cached value. Matters become more complex when dealing with requests for remote lines; such requests are forwarded to the appropriate socket via the Quickpath Interconnect (QPI)[29]. Contention between sockets forces writes back to main memory, in contrast to on-socket contention, where it is sufficient to update the L3 cache; as a result, access time latencies of contentious memory vary significantly depending on the topology of the cores involved[33].

4.3 Overview of False Sharing
Cache coherency makes sense in the typical case[4] where a parallel workload accesses primarily large amounts of shared read-only data and smaller amounts of private, mutable state. Six out of Philip Collela’s seven dwarfs of parallel computing [4] fall into this category; out of the seven, only Fourier transforms require all-to-all sharing. In the typical scenario, however, the bulk of memory accesses will be to thread-local memory locations, which can be read and modified in private core caches.

A shared cache line where multiple cores are merely reading from the line suffers no performance degradation, as a read-only workload will never cause any line to become inconsistent with the others. However, when one or more cores become a writer to the line, then all other readers require that their caches become in sync with the writer’s. This access pattern results in coherence misses as the internal cache coherence protocol must negotiate between cores to preserve consistency.

True sharing occurs when concurrent accesses are to a shared word of memory, such as a lock or reference count. However, false sharing occurs when two
Despite different heap organizations and structure padding, both 32- and 64-bit binaries exhibit false sharing.

**Figure 4.1:** False sharing in Phoenix’s linear regression benchmark.

independent words of memory happen to lie on the same cache line. As a result, despite the likely programmer intention of isolated, exclusive writes, unnecessary contention is created.

As described further in Section 6.1, the Phoenix[38] parallel benchmark suite’s linear regression test is a popular example of false sharing[30, 43]. Figure 4.1 shows the `lreg_args` structure that is responsible for false sharing. An array of these structures store per-thread state, where each element is accessed in a tight loop that stores intermediate values as the benchmark runs. The program suffers a very high degree of false sharing, actually performing worse in parallel than if it is run alone on a single core.

**Figure 4.2** compares the scalability of the benchmark against a version that has been hand-corrected to eliminate false sharing, by forcing each element of the array onto its own cache line. Without false sharing, the program scales nearly linearly. As the vast majority of the benchmark runs in parallel, this behaviour...
is expected. With false sharing, however, updates must be committed from cache 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[11]. The organization of the program is another: Figure 4.1 shows the same source file compiled as both 32-bit and 64-bit binaries. Despite identical source and cache organization, the nature of the 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 ends up misaligned because allocator metadata offsets the structure’s beginning word.

**Figure 4.2**: False sharing of linear regression as a function of cores.
Chapter 5

Remapping

Cachekata was designed as a general-purpose memory remapping mechanism. We have incorporated it into a larger system called Plastic, which dynamically detects and repairs instances of false sharing in parallel workloads. While outside the scope of this thesis, Plastic uses a combination of hardware performance counters and emulation to identify cache lines where false sharing is occurring. For the purposes of this chapter, we assume Plastic’s rule generator, after having observed a hypothetical program execute for a period of time, has emitted a series of remap rules that will break contentious memory accesses onto their own cache lines. This is shown visually in Figure 5.1 and made concrete in Chapter 6.

When a remap rule is added to the system, the P2M table entry that contains it has its access permissions revoked. In order for accesses to the process’ virtual address space to remain well-defined, additional implicit rules may be added to ensure the guest-physical page is fully-remapped.

5.1 Rewriting procedure

Since a remap rule only describes changes to memory access patterns, it indicates where but not when redirection is to occur. Therefore, the program’s execution proceeds normally until an invalid page is touched. As described in Section 4.1.1, hardware automatically redirects execution to Xen, which in turn begins executing Cachekata’s rewriting functionality.
By registering as a fault handler, Cachekata acts as a central dispatcher that redirects execution to rewritten code segments with memory accesses “corrected” according to the matched rule. Unlike other BT systems that analyzes the entire program’s control flow and translates each basic block when it begins executing[12, 13, 31, 34], we only rewrite and redirect execution for contentious memory accesses in a way that oscillates between the original and rewritten code.

While it would be desirable to overwrite the faulting instruction with a trampoline to the instrumented code to optimize out the hypervisor trap on future accesses, this is highly challenging for a number of reasons. First, the variable-length nature of x86 means the size of a jmp instruction may exceed the size of the faulting instruction. Further, overwriting the original code segment creates a race condition in a multithreaded workload; while we pause all virtual CPU (vCPU)s during rewriting and can inspect each of their program counters, at the hypervisor level we cannot determine conclusively whether other, context-switched out threads are executing instructions that we would be overwriting. In such a case we would leave the system in a potentially-inconsistent state. Or, worse, cause a vCPU to begin executing misaligned instructions, resulting in undefined behaviour. Cachekata avoids this by leaving the original code unchanged and taking a fault on the data path, rather than the code path.
Cachekata runs as a userspace tool outside Xen, and uses DynamoRIO[12]’s x86-64 disassembly library to decode the faulting program’s instructions and transform them into an IR. DynamoRIO was chosen for its robust support for the x86-64 instruction set and the overall tight coupling with it; as we discuss in Section 5.4, our claims of integrity and safety are dependent on knowing details of x86-64 in particular and not having an architecture-agnostic IR.

5.2 Control flow inference

One of our core tenets is that memory accesses acting as performance bottlenecks will occur frequently; for instance, consider testing a spinlock, executing a memcpy-like operation, or an access-heavy tight loop. Further, it behooves us to amortize the cost of redirecting control flow – requiring at least one context switch – by rewriting not merely the faulting instruction but the surrounding loop context. We seek to instrument the entirety of the loop and return to the original code segment when we have broken out of the loop.

Our goal, then, is to find the instruction corresponding to the top of the loop structure. In contrast to other instruction sets, x86 instructions are widely variable in length. Certain instructions, such as int3, only take up a single byte, while the architecture’s baroque addressing modes, immediate operands, and optional legacy prefixes can balloon a single instruction to 15 bytes. Further, instructions need not be aligned on any memory boundary. Therefore, while an instruction stream can be read forward, it is ambiguous as to what the previous instruction in a code segment is. Cachekata uses a combination of control flow analysis techniques and debug registers to determine addresses where previous instructions begin. From there, the partial control flow graph is stitched into a rewritelet structure, described in Section 5.3.

5.2.1 Last-branch record registers

The Intel Core2 series of processors introduced a series of registers, called the last branch record (LBR), that stores the n-most recent jump source and destinations executed on the CPU. Nehalem processors store the last 16 such branches, giving us a reasonably-lengthy history of previous addresses that correspond to the beginning
of instructions. We choose as our loop start instruction the backwards jump target most frequently observed in the set of 16 branches.

We observe that the typical idiom for compiler-emitted loops in gcc involves placing the conditional check at the epilogue of the loop segment and jumping back for the subsequent iteration. The loop header, as a result, contains merely the counter initialization and an unconditional jump to the epilogue. As a result, in such scenarios the LBR method is able to identify a loop block even when we fault on the first iteration.

While simple to understand, straightforward to implement, and cheap to execute, the LBR method makes a great deal of assumptions about the code under analysis. For instance, if a different compiler used a different loop construct than the one described above, the LBR might not be populated with the correct loop beginning upon executing the first iteration. Therefore, we have implemented an additional, more general mechanism based around explicit control flow analysis.

5.2.2 Backwards jump analysis

Backwards jump analysis simply traces ahead from the faulting instruction until a jump is found. If the jump target is a lower address than the current rip, we recurse with the target, and if the jump is conditional, the subsequent instruction. This procedure continues until an already-discovered memory address or a non-local control flow instruction such as a ret is found.

When the jump analysis terminates, all local control flow changes (i.e., near jumps) have been enclosed into a control flow graph, where vertices represent single-entry, single-exit basic blocks. Since the graph generation is driven exactly by the possible execution paths of a code block, we know the control flow graph will be well-defined without needing to assume any compiler or language-specific idioms. However, it should be stated that the LBR approach was sufficient to reconstruct the rewritten code segments for all of our workloads discussed in Chapter 6.

5.3 Rewritelets

Once the code segments to be rewritten have been found by either of the above methods, they are copied into a rewritelet structure for each rewrite rule in the sys-
Figure 5.2: Rewritelets for a single instruction code segment.

A rewritelet is, briefly, a block-level optimization for a particular memory redirection rule: each memory access that falls within the rewritelet’s corresponding rule is redirected before the access occurs. Figure 5.2 shows how, on a hypothetical system with two installed remap rules, two rewritelets are generated for a faulting memory access. For the sake of clarity, the code segment copied into the rewritelet in this case is only one instruction in length, though in reality of course multiple instructions would be copied.

5.3.1 Structure

A rewritelet consists of the code segment to be rewritten, bookended by a prologue and epilogue. Non-relative jumps within the rewritelet need to be modified to not jump back to the original code segment, and relative jumps’ offsets need to be corrected to account for the newly-inserted set of guard conditions. Apart from that, however, the original instructions themselves are left unmodified. The rewritelet prologue simply saves the values of all scratch registers that we will need to use through the lifetime of the rewritelet’s execution. For example, in Figure 5.2 may modify the rsi register. Because we would like the register state to be left unmodi-
fied by us when we exit the rewritelet and rsi is not itself modified by the original code, its original value is stored and popped back in the epilogue, before jumping to the subsequent instruction in the original code segment.

5.3.2 Guard conditions

All faulting memory accesses are preceded by an guard condition, similar to the ones in XFI[17]. A guard condition checks whether the faulting address falls within exactly one remap rule’s range, and if it does, a modification to the pointer to point to the remapped region is made. Note that because rule ranges are known at code generation time, these values can be treated as constants and folded as immediates directly into the code. Additionally, in loops where accesses exhibit spatial locality – what we consider to be the common case – the CPU branch predictor will mark each guard condition as either strongly-taken or strongly-not-taken, making the overhead of executing the guard nearly free[41].

More than just the faulting one, guard condition is placed before every memory-touching instruction since we have no way of knowing a priori whether a particular instruction will ever potentially attempt to read or write memory remapped by this or any other rewrite rule. As a result, on every memory access attempt we have three potential scenarios:

1. The address does not fall under any rewrite rule: In this case, either the ja or jb branch will be taken, and the address will not be modified. The memory access will proceed as originally intended.

2. The address falls under the rewritelet’s rewrite rule: In this case, one may regard the address as being &src[i], for some offset into the remap rule. Execution will fall through both guard branches, and the register containing the address will be overwritten with &dst[i].

3. The address falls under another rewritelet’s rewrite rule: As in the first case, the address will remain unmodified and the original access will be attempted. However, this will result in an access violation (as the page to be accessed has its p2M permissions revoked) and a trap to Cachekata’s fault handler, which will redirect access to another rewritelet.
Case 3 could be simplified by, instead of having $n$ rewritelets, each with only one guard condition per memory-accessing instruction, having all faulting accesses redirect to a single instrumented code block, with each memory access preceded by $n$ guard conditions for all $n$ rewrite rules. This design, however, proved too inefficient for our purposes: constructing the set of guard conditions as an if/else ladder, obviously, scales linearly with the number of rewrite rules. Additionally, a binary search through an array of remap rules, even with most-recently-used rule caching, in practice showed no improvement with the number of remap rules used in our workload evaluation. Additionally, we found that a memory access would typically either touch a non-remapped region or the same region of remapped memory. Ultimately, while case 3 involves a slow page fault and rule lookup, we believe this case will only occur more than infrequently in pathological situations.

5.4 Integrity and Safety

Having strong guarantees about the nature of rewritelets is a top priority. Weaving rewritelets into normal execution must not interfere with the integrity of the system, and in order to generate rewritelets themselves, we must be able to manipulate the system from underneath its feet in a safe manner, particularly in a highly-parallel setting such as a scientific computing benchmark. By making no assumptions about previous state, a thread can be transferred over to a rewritelet at any point.

A key insight into building the system is that, as a performance optimization, Cachekata can afford to be sound but not complete. While existing systems aiming to provide strong security guarantees[18] must deal with the many corner cases in x86, we take an extremely conservative view of the kind of instrumentation we apply. We are able to reverse all remapping when we find instructions that are unsafe to instrument.

5.4.1 Atomnicity

Rewritelets must be generated atomically. Cachekata pauses all VCPUs while it is executing; as a result, no two VCPUs can race on generating rewritelets for the same fault concurrently. Additionally, because all redirection occurs through page
faults that trap to Cachekata, there is no need to worry about overwriting original
instructions that are being executed concurrently. By contrast, if we were writing
trampolines over the original code segment, we would need to ensure that no other
thread was executing the instructions we were changing underneath its feet. While
Cachekata can inspect the rip value of all vCPUs, it cannot account for context
switched-out tasks without OS modification.

5.4.2 Completeness of coverage

By keeping the original data page protected for the entire lifetime of the remapping,
Cachekata does not need to statically identify all instructions that could access that
region of memory and instrument them. Instead, the page fault-based redirection
we use allows new instrumentations to be demand-faulted in whenever a new ac-
cess is attempted.

Conversely, while pointer values are checked and modified by guard conditions
before being dereferenced, but we do not allow these changes to propagate further
through the system. Leaked pointers could be modified and used to incorrectly
access remapped data structures without actually causing a fault, resulting in un-
deﬁned behaviour. To prevent this, Cachekata restores the original value of the
register to prevent the remapped address to be written back to memory.

Instructions that writes a faulting address, such as mov rax, (rax), are
hard to instrument directly because modifying the pointer (ie. the value stored in
rax) implicitly causes it to be leaked. Such instructions can be addressed by using
an extra register or stack push to store the original value.

Function pointers or vtables that are located on the protected page and accessed
via indirect jumps, such as jump *rax, still need to be modified to point to a
remapped location. However, as this instruction modiﬁes the program’s control
ﬂow, the instrumentation system has no way of restoring the value of rax on the
subsequent instruction (ie. after the jmp). A possible workaround is to translate
register-indirect jumps with a rip-relative one, although we have yet to encounter
a workload where such an instruction needs to be modiﬁed.
5.4.3 Register state

A more common concern is maintaining sane register state during rewritelet execution. As noted above, any scratch registers needed by the rewritelet are saved and popped before jumping out of the rewritelet. So long as the number of pushes and pops balances, the value of \( \text{rsp} \) will remain unchanged when we exit. However, since we are writing to addresses above the stack, if the program being rewritten has an otherwise-benign “return pointer to local variable” bug, this may trigger undefined behaviour.

As stated earlier, the guard condition before memory accesses will overwrite the \( rflags \) register. Even though by virtue of inserting test and conditional jump instructions we do not push and pop the \( rflags \) register. We consider it highly unlikely that \( rflags \) would be used without modification for multiple instructions or otherwise reused. There are, however, instructions that both access memory and rely on the state of \( rflags \), such as conditional move (CMOV). This, however, can be translated into an explicit jump and unconditional mov; additionally, while not officially deprecated, CMOV instructions are considered inefficient on out-of-order processors[40]. Such a transformation, while not implemented in our system, is straightforward to do with the existing tools at hand.

5.4.4 Rule-straddling accesses

A more serious set of issues involve memory accesses that involve reading or writing from more than one remap rule. For example, consider an 8-byte write to a memory region that has been remapped to two separate isolated 4-byte regions.

The x86 REP prefix is a common and particularly egregious case of this problem. The x86 architecture provides a provision by which common iterated operations such as memory moves can be “inlined” where the CPU’s underlying microcode is responsible for updating pointer values in the \( \text{rsi} \) and \( \text{rdi} \) registers and decrementing counter values in \( \text{rcx} \). Despite a potentially-hefty startup cost, throughout is greater than writing an explicit loop block, and therefore REP-prefixed instructions are widely used in memcpy/memset-like bulk copy functions. Performance improvements with REP-prefixed instructions also vary with pointer alignment [23], but “judicious” use of the REP prefix is recommended on modern pro-
cessors for string operations [3].

Since there is no way of single-stepping a REP-prefixed instruction, we can only regard the bulk copy as an indivisible operation. But, since the number of bytes that the instruction touches can’t be reasoned about during rewritelet-creation, it could come to pass that the instruction touches any number of remapped memory regions. As was the case with cmov instructions, any REP-prefixed instruction can be unrolled into an explicit loop at an unknown performance loss, based on the constraints described above.
Chapter 6

Evaluation

The mechanism implemented in Cachekata is general-purpose in nature and could be used in a variety of contexts beyond Plastic’s false sharing mitigation. Since we essentially expose a mechanism to invoke a callback on a byte-granular memory fault, in the limit, one could use this system even to instrument every memory access in a manner similar to that of Valgrind[34].

However, the evaluation presented for Cachekata, once again, orbits the false sharing mitigation problem. We have tested an end-to-end detection and repair system on a series of parallel benchmarks and synthetic workloads.

Plastic is evaluated on a dual-processor, 8-core Nehalem system with 32GB of memory. Each processor is a 4-core 64-bit Intel Xeon 5506, running at 2.13 GHz, with private per-core L1 and L2 caches, and a shared 8MB L3 cache. Plastic is built atop Xen 4.2, running a Linux 2.6.32.27 kernel as Dom0. We seek to understand the following questions:

- How is the execution of a program running under Plastic affected?
- What performance impact do redirecting execution to rewritelets incur?
- Does Plastic improve the performance of programs where false sharing occurs?
6.1 Workloads

The simplest and most artificial workload is called synthetic. It is designed as a pathological false sharing-exhibiting scenario. In it, each thread simply increments an element in a shared cache line-aligned integer array. Since all of the program’s work involves updating memory values that are independent and yet lie on the same cache line, we treat synthetic as a worst-case torture test. Additionally, its simplicity – its main loop is 10 lines of hand-written assembly – made it invaluable in debugging early versions of Plastic.

Additionally, we tested Plastic against the Phoenix[38] parallel benchmark suite. Phoenix has been reported by others to contain workloads that exhibit false sharing[30]. Of particular interest to us is Phoenix’s linear regression test, since it is a “real-world” workload that exhibits false sharing nearly to the degree of our synthetic one.

6.2 Detection and Execution under Plastic

Figure 6.1 shows the end-to-end execution of the linear regression benchmark under Plastic. The top half of the figure shows HIT-Modified (HITM) values over time, which are Intel’s performance counter for coherency misses (ie. a cache
Figure 6.2: Normalized performance of linear regression.

miss where the line is present and modified in another core’s) as well as program throughput. The bottom half of the figure indicates which component of Plastic is currently active; the perf, pebs, logall, and emulation stages are part of the false sharing detection mechanism and not of interest to us. One can see that that around the 2000ms mark the first set of remap rules are faulted in. The effect of this is immediately observed, with the HITM count stabilizing, and throughput rising significantly until the end of the benchmark.

Figure 6.2 shows the performance of the linear regression benchmark from 1 to 8 cores, normalized against idealized speedup. Four versions of the program are shown: the original version with false sharing executing normally, the same unmodified version running under Plastic, and a version of the program where false sharing was fixed by hand. Additionally, as a strawman evaluation, we tested the performance degradation of modifying control flow by using a single rewritelet with an if/else ladder prior to all memory accesses.

With an increase in cores, performance of the unmodified version decreases exponentially. As the number of coherency misses increase to the point that cores are constantly stalling, the multi-threaded version is slower than a single-threaded version, even in terms of absolute runtime. The source-optimized version, on the
flip side, shows near-linear speedup. As expected, the if/else ladder scales badly, about as terribly as the original program. The version running under Plastic with no manual source corrections, however, yields a 3-6x performance improvement over the unmodified version.

Finally, it is important to evaluate the end-to-end impact of running workloads under Plastic. Figure 6.3 shows the performance of applications from the Phoenix benchmark suite, synthetic, and the streamcluster benchmark from the Parsec benchmark running on 2, 4, and 8 core configurations under Xen and Plastic. The performance shown is the average of five runs, normalized to the single-threaded performance of the same benchmark running under Xen without Plastic. While all workloads in the Phoenix and Parsec benchmarks were evaluated, we present a representative sample of results here for brevity.

Figure 6.3: Normalized performance of benchmarks running natively and under Plastic.
Programs exhibiting clear false sharing such as linear regression and synthetic enjoy a significant performance improvement under Plastic, especially as the number of cores increase. Those without false sharing execute normally with low overhead. string_match is a special case in that it probabilistically experiences false sharing, depending on how heap structures are laid out, but still routinely scales well as the cost of the coherence misses are masked in the computation.
Chapter 7

Discussion

One lesson that we have taken from this project is that real-world examples of memory problems like false sharing are much rarer than we expected. While there have been a large number of papers on the diagnosis and mitigation of false sharing in recent years, they almost all focus on Phoenix’s linear regression benchmark as an example. Further, in this example and others, compiler optimizations are often disabled, as 64-bit systems have a tendency to move working memory that happens to result in false sharing into registers during tight loops. Our focus on the linear regression benchmark is both educational and consistent with previous research in this area.

We feel that, despite the challenges we have faced in identifying a wider range of useful benchmarks, that this work stands as a contribution for two reasons. First, many real-world examples of false sharing that we are aware of only manifest once the number of cores scales to 48, 64, or beyond[11]. Our suspicion is that the “any-to-any” nature of cache coherence traffic will make this problem considerably more relevant over the next few years of processor hardware development. Second, the approaches developed for Cachekata’s fine-grained remapping are useful and applicable to other problems in OS research.
7.1 Future work

Despite our focus on false sharing correction in this thesis, Cachekata was intended to be a general-purpose remapping mechanism. While its transparent nature could in theory lend itself to other previously-discussed uses of full-system BT (in particular, security applications), a more comprehensive instruction decoder would have to be implemented to ensure completeness of coverage.

Possible work, of course, still remains even in the performance remapping space. Plastic used Cachekata as a memory splitter, but the mechanism for aggregating distantly-located but logically-close pieces of memory onto the same cache line would be a straightforward addition to the system. In so doing, it is possible that sparse lines could be consolidated, resulting in a more efficient use of the cache hierarchy.

This would, however, require a more dynamic policy for generating rewrite rules and, correspondingly, rewritelets. Currently, we indicate to where a region of VA memory should be remapped. Allocation of balloon driver memory when we wish to split memory is trivial assuming that the region of remapped memory is large enough to store each region on its own cache line. Consolidation is trickier, however, in that we have to specify a “closeness” metric and attempt to optimize for that metric.

Currently, our guard conditions simply redirect memory accesses. However, our instrumentation blocks could additionally have side-effects for profiling data, in a manner similar to Valgrind’s plugins[34].

We trust implicitly that the rules passed to Plastic will not exhibit the rewrite rule ping-ponging that will result in a slow path access on very many accesses. What to do if we find ourselves oscillating between rewritelets remains a mystery at this stage. One could certainly conceive of merging rewrite rules to correct for this behaviour as this effectively is an implicit “move a to be near b” problem.

Certain implementation features would have to be built to support such avenues of future research. In particular, rewrite rules and memory consolation requires discarding rules and rewritelets. Currently, we have no mechanism to free memory in the code cache or balloon driver, thus necessitating the implementation of a rewritelet garbage collector.
Chapter 8

Conclusion

Cachekata demonstrates that, by taking responsibility for the monitoring and management of CPU caches, an operating system can dynamically recover from workloads that exhibit pathological stresses on the memory hierarchy, as seen with false sharing. Cachekata demonstrated a sub-page granularity remapping facility that is sufficiently high-performance as to be incorporated into a larger system, Plastic, which improves throughput in cases of high-rate false sharing by 3-6x.

We argue that the age of the modern processor has heralded the end of the Little White Hardware Lie. It’s time for software developers to take note of the byzantine rites of maintaining cached state between cores and reflect on the subtle but serious implications for programmers interested in writing performant software. In particular, it is our belief that anticipating the intentions of running software and re-orienting them to better align with the realities of the underlying hardware is in line with the set of lies already exposed by systems software.
Bibliography


pages 1, 12, 13


