Whose Cache Line Is It Anyway? Operating System Support for Live Detection and Repair of False Sharing

Mihir Nanavati Mark Spear Nathan Taylor Shriram Rajagopalan Dutch T. Meyer
William Aiello Andrew Warfield

Department of Computer Science, University of British Columbia
{mihirn, mspear, tnathan, rshriram, dmeyer, aiello, andy}@cs.ubc.ca

Abstract

As hardware parallelism continues to increase, CPU caches can no longer be considered as a transparent, hardware-level performance optimization. Cache impact on performance, in particular in the face of false sharing, is completely dependent on the software that is executing. To effectively support parallel workloads on cache coherent hardware, the operating system must begin to treat the CPU cache like other shared hardware resources, and manage it appropriately.

We demonstrate a prototype example of such support by describing *Plastic*[1], a software-based system that detects, diagnoses, and transparently repairs false sharing as it occurs in running applications. Plastic solves two challenging problems. First, it is capable of *rapid, low-overhead detection and diagnosis of false sharing* in unmodified, running applications. Second, it resolves identified instances of false sharing by providing a *sub-page granularity memory remapping* facility within the system. Our implementation is capable of identifying and repairing pathological false sharing in under one second of execution and achieves speedups of 3-6x on known examples of false sharing in parallel benchmarks.

1. Introduction

Cache contention on modern CPUs can lead to performance collapse. This collapse is entirely workload dependent and cannot currently be mitigated by the hardware providing the caching. As a result, there exist applications that perform terribly on modern CPUs because they contend for cache lines. Not only are existing systems unable to correct this, they are also unaware of the very existence of such contention.

In modern CPUs, the design of processor caches is complicated by two properties. First, rather than increasing frequencies, processors are becoming more parallel. Second, cache coherence is still broadly held as a necessary property of CPU implementations. Increasing parallelism means that there are more threads operating on memory at once, while coherence demands that all threads see a single, consistent view of that memory. Where concurrent accesses to the same cache line involve one or more writers, exclusive access is required and the resulting cache coherence protocol interactions necessitate expensive, synchronous notifications across multiple cores and even physical sockets in the system.

Avoiding cache line contention should be treated as a systems problem. The cache, after all, is a shared performance-critical resource and software layers such as the VMM, OS, and language runtime occupy a useful vantage point from which to mediate access. Unfortunately, identifying, understanding and resolving cache contention is a challenging task on modern CPUs. Once false sharing is identified, resolving it correctly requires a fine-grained remapping mechanism to "split" a cache line in a manner that allows concurrent threads to achieve non-contending access – a facility that is not provided by page-granularity MMU hardware.

The VMM-based prototype system described in the remainder of this paper achieves both of these goals. First, through a combination of hardware performance counters and memory virtualization, we present a *false sharing detection system* that is able to rapidly detect false sharing and identify the specific, relevant byte-level regions of data that are contending. Second, to resolve the contending accesses identified by our detector, we use both hardware page protection and binary instrumentation to introduce a *fine-grained memory remapper*. This facility extends conventional virtual memory support, which works at a page granularity, with a byte-level

[1] The title of our system is an analogy to the concept of neuroplasticity: the ability of the brain to adapt to changes in environment and behaviour over the course of its lifetime.

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

remapping facility. Using this interface, the system can elect to transparently move contending data structures in virtual memory into new locations in physical memory *while code actively executes* on the original virtual addresses.

We demonstrate that it is possible to detect and repair false sharing in a manner that works on existing hardware and applies to existing application binaries. Our detection system has sufficiently low overheads as to be deployed in both development and production environments, while the remapping engine transparently and efficiently redirects memory accesses, allowing data structures to be arbitrarily removed from the middle of a page and placed elsewhere in memory. The resulting system is capable of identifying and fixing false sharing in applications in under a second of execution, resulting in a significant speedup for concurrent workloads.

2. Cache Coherence and Scalability

Cache coherent systems are parallel computing systems which, despite the presence of private, per-core caches, present a single, unified view of memory to the entire system at any given point in time. The benefits of such consistent, shared memory, especially in parallel programming, come at a scalability cost to the extent that several highly-parallel architectures [19, 38] and OSes [2] have explored system design in the absence of cache coherence. Still, many computer architects [26] and systems designers [7] believe that existing systems can, in fact, continue to scale to much greater degrees of parallelism.

Cache Coherence and the x86: As a dominant example of general-purpose CPU design, Intel's x86 processor cache architecture has remained relatively unchanged since the release of the Nehalem microarchitecture in 2008. Coherency is maintained using MESIF, an extension to the popular MESI state protocol [33]. Each cache line has a state associated with it, while the inclusive L3 acts as a directory [17], both to maintain coherency amongst on-socket cores, as well as service requests from other sockets. Simultaneous reads are supported by allowing multiple copies of the same cache line to coexist in "Shared" state. Any write, however, causes the requesting core to become the owner of a cache line, which is put in "Exclusive" or "Modified" state in its L1, while all other copies are invalidated.

Subsequent requests from other cores are serviced by the on-socket L3, which checks the availability of the line. For local lines, the resulting flushes of modified data from private cores are snooped for modified values, which are then written back before completing the request. Requests for remote lines are forwarded to the appropriate socket via the Quickpath Interconnect (QPI) [23].

Accesses to modified cache lines force a write-back to a location accessible to the requesting core: contention amongst

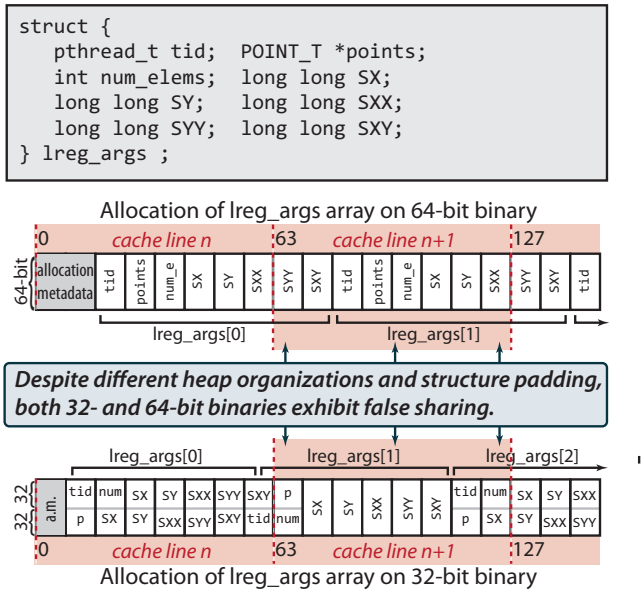

Figure 1. False sharing in the linear_regression benchmark.

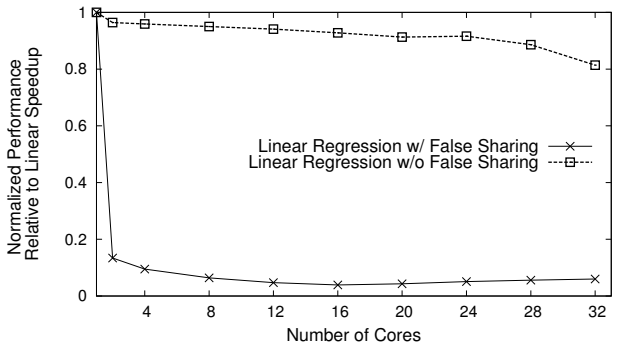

Figure 2. Effect of increased parallelism on performance.

on-socket cores updates the L3, while cross-socket cores are forced to write-back to main memory. As a result, latencies of accesses to contended memory vary significantly depending on the exact physical topology of the cores involved [28].

True and False Sharing: Cache coherent architectures optimize for parallel workloads that tend to have large amounts of shared read-only data and smaller amounts of private mutable state. Cache lines with multiple accessors, at least one of which is a writer, experience expensive *coherence misses* as the coherence protocol must negotiate between cores in order to preserve consistency. True sharing occurs when concurrent accesses are to a single, shared data structure, such as a lock or reference count. False sharing occurs when independent data structures happen to reside on a single cache line; here, the workload matches the assumption of shared reads and isolated writes, but the coarse granularity of isolation results in unnecessary contention.

The Phoenix [35] parallel benchmark suite's linear regression test is a popular example of false sharing [24, 40]. Figure 1 shows the `lreg_args` structure responsible for false sharing. An array of thread-indexed structures store intermediate per-thread state and are accessed in a tight loop causing a high degree of false sharing. Figure 2 compares the program's scalability against a version modified to eliminate false sharing. While the modified version scales nearly linearly, adding additional cores to the original version often makes it *slower*, even in terms of absolute time. Another access pattern causing false sharing, seen in the Linux kernel [7], has a single frequently updated field in a structure surrounded by read-mostly data.

Besides the workload, false sharing depends on many dynamic properties in a system. Figure 1 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, whereas the other produces an ideally sized 64-byte structure, but then misaligns it because of allocator metadata.

False sharing is still a problem in today's systems: False sharing has long been recognized and studied as a problem on shared memory systems [6]. While compiler support can help in some cases, it is far from universal.[2] Many instances of contention are properties of workload and simply cannot be inferred statically. As evidence, recent years have seen significant examples of false sharing in mature, production software. False sharing has been seen in the Java garbage collector on a 256-way Niagara server [11], within the Linux kernel [7], and in spinlock pools within the popular Boost library [10, 27]. Transactional memory relying on cache line invalidations to abort transactions [18] also performs poorly with false sharing [29]. These examples serve as the basis for CCBench, the microbenchmark suite discussed in Section 6. That false sharing occurs in mature software is an indication not of a lack of quality, but rather that workloads leading to contention are often not seen in development and testing.

3. Design and Architecture

The system described in this paper, called *Plastic*, provides system-level support to dynamically detect and mitigate persistent false sharing in unmodified application binaries. Plastic is a software implementation of a byte-granularity memory remapping mechanism. It allows any arbitrary byte range of virtual memory to be remapped from one physical location in memory to another, *while* the target is still running.

Specifically designed to mitigate false sharing, it determines the exact regions of virtual memory that currently exhibit contending accesses and then transparently remaps them to physical addresses on independent cache lines. Our approach is inspired by the sort of virtual-to-physical address remapping that is already possible with paging hardware, but refines it to a sufficiently fine granularity as to mitigate contention within a single cache line.

[2] For example, `gcc` fixes false sharing in the Phoenix linear regression benchmark (see Figure 1) at `-O2` and `-O3` optimization, while `clang` fails to even at the highest optimization level.

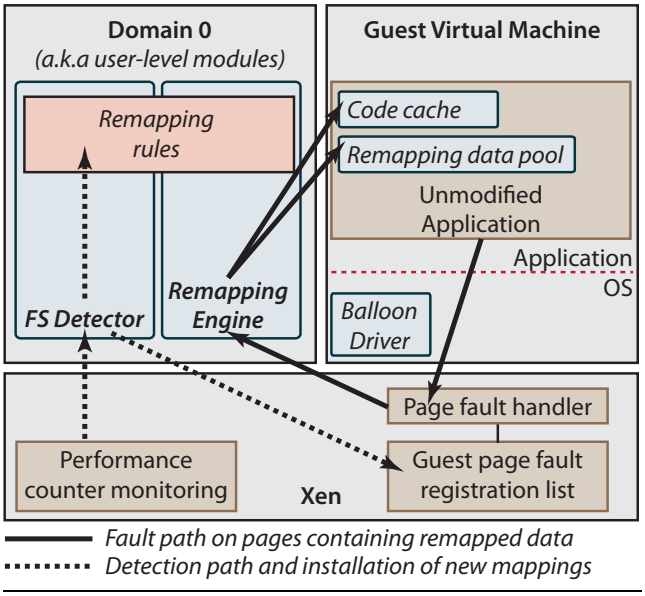

Figure 3. Plastic Architecture

Modern x86 hardware does not support remapping at this fine granularity. Plastic implements its own remapping system in software: provided with a set of byte granularity memory remapping rules, it applies them to a running system using live binary instrumentation. When false sharing is identified, Plastic creates a copy of the contending data on a non-contended cache line and uses dynamic instrumentation to redirect all accesses to that new location in memory.

Plastic is currently implemented on the Xen virtualization platform [1], where it takes advantage of memory interposition capabilities that are relatively easy to extend. It is important to emphasize that our approach is not specific to hypervisors: Plastic could be incorporated into an operating system with equal benefit. We use the term "operating system" in the title of this paper to emphasize the more general opportunity for this class of system support.

Processors mask access latencies using instruction pipelining and out-of-order execution, thereby significantly reducing the impact of false sharing when contending accesses are separated by even a few hundred instructions. Plastic targets applications where performance is materially impacted by false sharing; typically, long running applications with high-frequency, parallel accesses to memory. To be practically deployable, it must function with low overhead, and detect and mitigate false sharing quickly and efficiently without requiring any changes to existing applications.

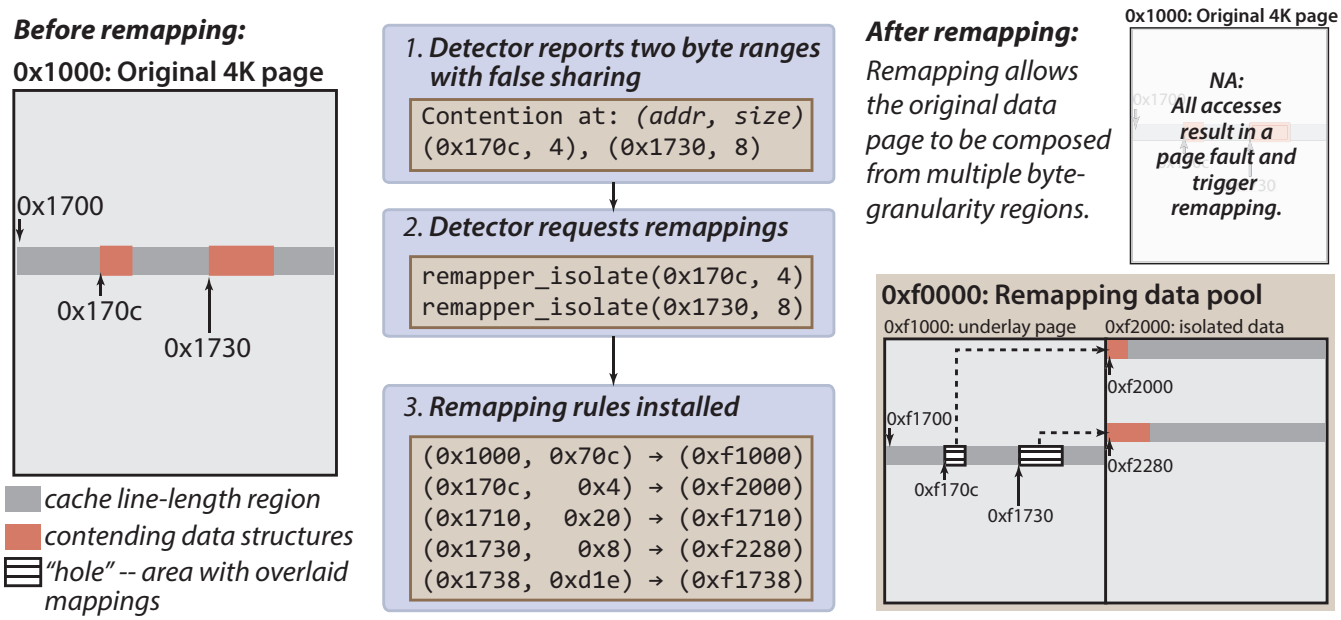

Figure 4. Byte-granularity remapping allows some data to be transparently isolated on separate cache lines.

3.1 Architecture

Figure 3 shows Plastic's architecture at a high level. The bulk of the system resides in a user space tool running on a modified version of Xen. The hypervisor serves two purposes. First, the machine's hardware performance counters are exposed to monitor coherence invalidations on individual processor cores. Second, page protection interfaces are used to interpose on memory accesses and to determine the exact byte ranges involved in false sharing. Plastic's execution involves two main responsibilities: detection and remapping.

Detecting False Sharing: Detecting false sharing on x86 is a challenging task, especially given the constraint of imposing low overhead on the system. Plastic takes advantage of hardware performance counters to detect memory contention by monitoring coherence invalidation events, which indicate multiple cores competing for exclusive ownership of a cache line. Proceeding from this observation, it performs a series of progressively more expensive refinements, but applies them to increasingly specific regions of execution. This approach, described in detail in Section 4, allows Plastic to quickly detect false sharing and refine the diagnosis down to specific byte-granularity regions of contended memory.

Remapping Cache Lines: On current x86 hardware, a page of memory – which is the smallest unit available to MMU hardware for remapping memory – contains 64 64-byte cache lines. Isolation requests from our false sharing detector are smaller regions *within* individual cache lines.

Plastic achieves fine-grained remappings as illustrated in Figure 4. Isolation requests from the detector specify small regions of memory that should be placed on an isolated cache line. The remapper responds to isolation requests by providing sufficient remapping rules to describe the entire page containing remapped data. The result is that the page becomes a composite of one or more small isolated data ranges that are mapped on top of an underlay page. All of these data components reside in the remapping data pool, and the original page of virtual address space is marked as "No Access", resulting in faults on any attempt to read or write data through that range of virtual memory.

To make remapping efficient, demand faults on access to the original page result in the analysis and modification of accessing code to interact directly with the remapped data. Guard conditions in this modified code redirect only references that interact with remapped data, and leave other references interacting with original addresses in memory. By directly altering the accessing code, Plastic avoids expensive faults on future accesses while efficiently isolating the contending data onto independent cache lines.

Plastic's remapper takes advantage of a design property that is almost never available to dynamic instrumentation systems: *the implementation need not be complete with regard to instruction set semantics*. Plastic is free to modify code wherever capable of improving performance, however, as our goal is strictly to improve performance in the system, the option always exists to do nothing. In cases where Plastic is unable to safely or efficiently remap data ranges, it restores the data to the original page by copying back the composite regions, invalidates all active remappings for that data, and allows execution to continue against the original location. This places both the code and data back in the original and

unmodified state. This observation, and the fact that page protection on the original data ensures that remapping covers all accesses to the remapped data from any code in the system, allow Plastic's design and implementation to pursue individual piece-wise optimizations based on workloads and access patterns that demonstrably benefit from remapping.

Plastic requires the guest OS to maintain both the code and data cache regions within the virtual address space of the application. This is the only requirement of Plastic on the guest and is achieved as an extension of the balloon driver, an in-guest memory management driver commonly installed in VMs. Under Windows, similar functionality can be obtained by the use of user-space AppInit DLLs.

4. Detection Methodology

Identifying false sharing typically requires a costly analysis of memory access patterns and to model those interactions within the cache hierarchy [16, 22, 40]. As shown in Figure 5, Plastic takes a multi-stage, sampling-based approach [9] to avoid these costs. Using a series of progressively more detailed (and consequentially higher-overhead) filters minimizes the impact of continuous detection and focuses the higher cost analysis on data likely to exhibit false sharing, based on information collected in earlier stages.

Starting from the left of the pipeline, we progressively refine the results, and begin by observing the presence of an abnormally large number of coherence invalidations using performance counters. We then isolate the pages where contention is occurring, before sampling memory accesses for short periods with emulation to find the precise regions of memory responsible for the contention.

4.1 Performance Counter Monitoring

Inputs: Running System
Outputs: Degree of contention in the system

Performance counters are special registers that store records of microarchitectural events, such as cache misses or branch prediction success rates, and are traditionally not used by the operating system and software. Acting as free running counters, they can be used to investigate the performance characteristics of the system with low overhead. Modern processors have events to count messages at every level of the cache hierarchy, including the invalidation messages sent from one core to another due to contended accesses.

While coherence invalidations indicate the presence of false or true sharing, it is hard to characterize its impact on performance solely using an absolute count. This is because the performance impact of every invalidation is not equal: invalidations caused by off-socket cores require data to be fetched from main memory and are much more expensive than those from on-socket cores. However, since invalidations essentially stall a core until the data is fetched, contending on-socket cores can issue more requests, and hence cause more invalidations, per second than off-socket cores. Additionally, out-of-order execution and pipelining allow processors to hide the latency of such invalidations with useful execution.

Rather than using invalidation counts in isolation, Plastic quantifies their effect on performance by calculating the number of invalidations per instruction executed. As suggested in Intel's performance manuals, values over a third of a percent signify potentially high degrees of contention [20].

Invalidations are counted using the `SNOOP_RESPONSE_HITM` event, which counts the number of snoop requests to a particular core that "hit" a modified value in a private cache that now needs to be written back to maintain coherency. Plastic uses per-guest virtualized counters within Xen to only count invalidations that occur during guest execution. While cache line granularity contention does not distinguish between true and false sharing, an absence of such contention signifies the lack of significant false sharing in the system.

4.2 Page Granularity Analysis

Inputs: Presence of high cache line contention
Outputs: Contended physical pages

Once cache line level contention is observed, true and false sharing are distinguished by determining the exact regions of memory accessed. Rather than instrumenting all memory accesses in the system, Plastic first determines contended pages by leveraging hardware page protection mechanisms.

As a hypervisor capable of running several unmodified guests simultaneously, Xen virtualizes memory and provides each guest the illusion of a contiguous address space. Hypervisor managed page tables perform an additional level of translation, either in software via shadow page tables or in hardware with Extended Page Tables (EPT) or Nested Page Tables (NPT) on Intel and AMD processors respectively.

Traditionally, Xen has a single set of hardware page tables per guest. Plastic extends these page tables to be per-core, each capable of having differing permissions for the same page. Using these tables to determine pages shared across cores is fairly trivial. Initially, all pages are set to "No Access" in all the private per-core page tables. Any subsequent access causes that core to fault, which promotes the permissions in its private page table and records the page and access-type in a per-core bitmap. Operating systems can perform similar analysis using per-thread page tables [3, 24].

Plastic periodically resets page permissions to determine the pages accessed by every core over several epochs. Since contended pages require at least one writer and one or more other readers or writers, the list of such pages for each sampling epoch is the intersection of the write bitmap of each core with the access bitmaps of all other cores. Contended

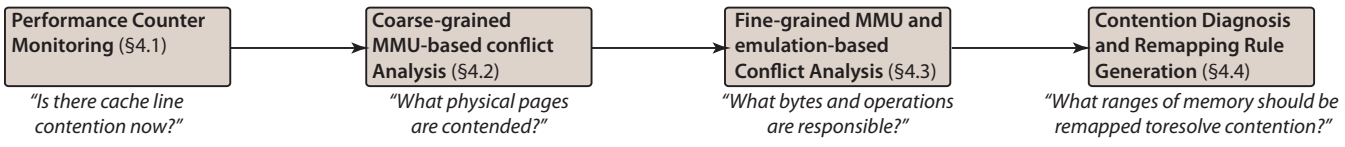

Figure 5. The stages of detecting and diagnosing cache contention.

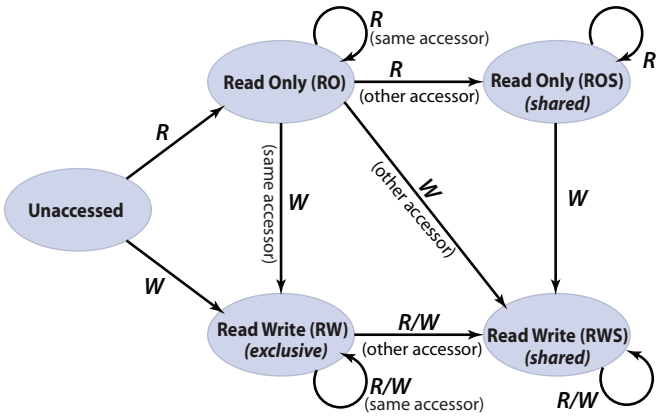

Figure 6. Sharing status of a byte in the access log.

pages, however, need not signify cache line contention since both thread migrations and non-overlapping heap objects on the same physical page could also be responsible.

4.3 Byte Level Access Analysis

Inputs: Contended physical pages
Outputs: Accessed bytes along with the identity of accessors

Analyzing memory accesses at a byte granularity requires recording every memory access. Plastic uses page table permissions to force a page fault on any access to a contended page; the fault handler restores these permissions before returning to the guest, while simultaneously setting up a single-step breakpoint to force a trap as soon as the instruction is retired. At this point, permissions are again reset, effectively forcing *every* memory access on that page to fault, where it is logged for further analysis.

Access patterns, delineated on a per-core basis, are not necessarily good indicators of sharing since migrating threads may access the same regions of memory from different cores. Distinguishing accesses at a thread level requires some knowledge of how threading libraries identify different threads; for instance, since most threading use the `fs(x86_64)` or `gs(x86)` segment registers to select the descriptor for the Thread Local Storage (TLS), Plastic simply logs the descriptor value as a thread identifier.

4.4 Remapping Rule Generation

Inputs: Byte level access log for contended pages
Outputs: Remapping rules for the page

Generating remapping rules involves identifying contended cache lines by classifying individual bytes according to the number of accessors and the access type. Plastic parses the entire byte-level access log and assigns one of the states in Figure 6 to each byte. Contended cache lines have multiple accessors with at least one writer (RW or RWS bytes).

During rule generation, memory regions within contended cache lines are grouped according to accessors. Multiple such groups are isolated from one another, while the bytes within the same group are remapped together. Groups with modified bytes (RW or RWS) are remapped to the isolated page, while the others are remapped to the underlay page.

4.5 Contention Verification and Adaptation

As a dynamic property, false sharing can evolve over time and requires constant monitoring and adaptation. Once the remapping rules are generated and sent to the remapping engine, the detection engine returns to monitoring performance counters for any contention in the system. Any detected contention triggers the entire detection pipeline which generates additional remapping rules as and when required.

Short lived false sharing may trigger detection and then subside before the actual remappings are applied. To avoid such scenarios, Plastic re-samples performance counters before remapping. While not guaranteeing that the previously detected false sharing exists, this ensures that some contention still exists before proceeding with the remappings.

5. Memory Remapping

Mitigating false sharing involves *transparently* and *safely* remapping *all* accesses to falsely shared regions of memory onto distinct cache line isolated locations. Plastic enforces such remappings with a combination of hardware-based page protection and dynamic binary instrumentation.

The remappings generated at the end of the detection pipeline only describe regions of contended memory without any mention of the corresponding accessors – in other words, they indicate what memory regions are to be remapped, but not where, in code, these remappings need to be applied. In order to detect all accessors, Plastic revokes access permission to contended pages and registers itself with the page fault handler for the entire lifetime of the remapping. Memory accesses are redirected using dynamic instrumentation by "correcting" the accessing instruction.

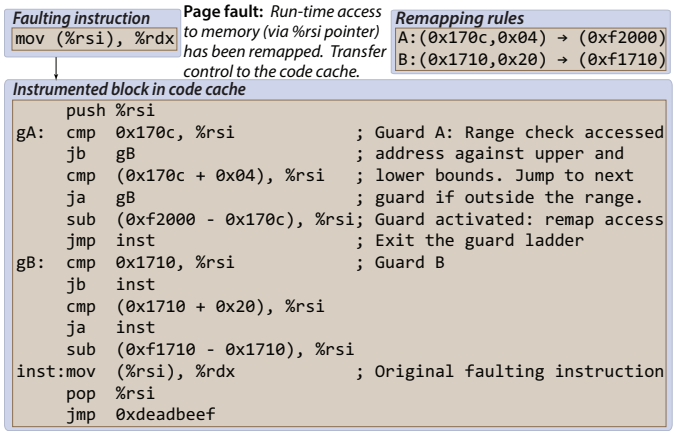

Figure 7. Guard Condition for a Single Instruction

Despite modifying the instructions executed, Plastic ensures that the semantics of the original code are preserved and the program remains safe at all times: instrumented instructions behave identically to the original with the exception of pointing to the remapped memory location.

5.1 Achieving Transparency

Plastic remaps arbitrary ranges of memory by redirecting memory accesses within a live, executing binary without any semantic knowledge of the program itself. While borrowing and adapting techniques from existing instrumentation and patching frameworks [4, 8, 25, 30, 31, 36], it does face two significant challenges. First, instrumenting a live binary cannot rely on any kind of load-time analysis; for example, several instrumentation frameworks [8, 25] generate a control flow graph and translate the entire program into basic blocks before executing it. Second, the kind of overheads acceptable in developer-facing diagnostic tools [30, 31] are not acceptable in performance-critical scenarios.

Plastic combines two techniques to apply the desired remappings: *fault driven redirection* and *guard conditions*. Rather than rewriting the original instruction stream, it maintains a separate code cache with instrumented instructions. The instrumentation, applied with the help of DynamoRIO's [8] disassembly library, modify the memory referenced by the instructions while Plastic redirects execution flow in a way that oscillates between the original code and the code cache.

Fault Driven Redirection: Plastic maintains a consistent view of remapped memory for the entire system by redirecting every accessor to an instrumented version in the code cache. Statically identifying all accessors to a region of memory, however, is complicated, especially in the case of an already executing binary. Furthermore, redirecting execution using branches [36] is not possible on a variable instruction size architecture like x86 because adding a `jmp` or `call` as a trampoline could overwrite subsequent instructions and leave the code in an inconsistent state.

Plastic avoids these issues by leaving the original code unchanged and redirecting execution via faults on the data path. By revoking access permissions to contended pages, it forces any access to trigger a page fault and maintains a mapping between the faulting instruction and the instrumented code. Plastic then acts as a centralized dispatcher and redirects execution to the code cache by updating the instruction pointer.

Redirecting execution via the fault path also has another benefit: unlike code trampolines, instrumentation is restricted to instances of an instruction that access contended data, while other instances remain unchanged. This prevents cases where all callers of a library function suffer from instrumentation overhead, even when required only by a single caller.

Guard Conditions: When copied to the code cache, faulting memory references are replaced by code that includes, in addition to the original instruction, instrumentation to appropriately modify the address referenced by the instruction. Figure 7 illustrates an example of one such code block.

As such instructions may access different addresses depending on the context under which executed, updating the reference address to a fixed, remapped location is insufficient. Instead, within the code cache, the instruction is preceded by a "guard condition", a set of checks similar to XFI [12].

Guard conditions verify that the memory referenced has been remapped and update the address according to the correct rule. References not matching existing rules fall through all the checks and execute the unmodified, original instruction, ensuring that program behaviour remains unchanged.

5.2 Optimizing Performance

Trampolining to the code cache via page faults is slow and routinely hitting the fault path for every contended memory access is several orders of magnitude slower than simply allowing false sharing to exist. Plastic attempts to reduce faults by instrumenting entire code blocks rather than single instructions, whenever possible.

Almost all high-frequency, performance-impacting contended accesses, even calls or inlined accessors, are wrapped by loops. Instrumenting entire loops within the code cache amortizes the fault cost over several iterations. The primary exception to this case, i.e. high-frequency contended accesses in straight line code, is code that is called repeatedly through asynchronous event injection: syscalls, interrupts, and user-level equivalents such as signal handlers. Plastic can be extended to optimize this case by instrumenting the entire function and then rewriting function call sites [31].

Code blocks are instrumented by guarding memory references to ensure that they are correctly remapped. These code blocks terminate with a direct `jmp` back to the next instruction in the original code. Branch target offsets and `rip`-

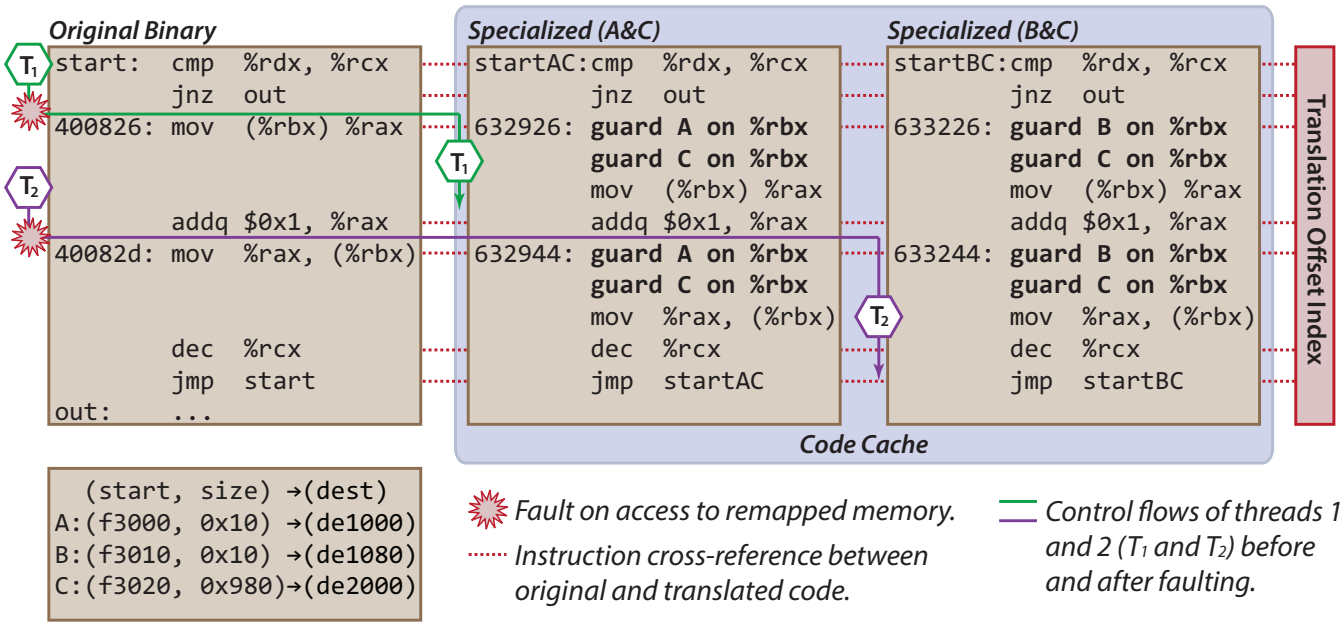

Figure 8. Control transfer on access fault from original binary, to specialized blocks in the code cache

relative accesses are corrected to account for both the code relocation and the added instrumentation instructions.

Identifying Code Blocks: To instrument an entire code block, rather than an individual instruction, Plastic faces the challenge of identifying block boundaries based only on an instruction pointer lying somewhere within the block. Fortunately, modern processors have the ability to track both source and destination addresses for recently taken branches using a facility called the Last Branch Record (LBR). Plastic identifies blocks by searching this LBR at the time of a fault for recent branches that move backwards in code and describe an address range containing the faulting instruction.

Despite the simplicity of this approach, it has proven to be very effective in practice: even with a typically 16-entry LBR history, Plastic is able to identify loop boundaries that effectively allows it to amortize the overhead of transferring execution to the code cache. Faulting instructions within hot code blocks cause the entire block to be instrumented while other accesses are instrumented on a per-instruction basis, preserving remappings for less-frequently accessors. In the future, we anticipate extending Plastic to periodically sample the LBR in the background to better detect nested loops and loop-embedded function calls.

Specialized Code Blocks: As implied by Figure 7, memory references within the code cache require a guard condition for *every* applied remapping. This approach, especially in the face of large numbers of mappings, is problematic: threads typically access only a small subset of the remapped data, similar to the example of adjacent private structures in Figure 1, and forcing execution through a long ladder of conditional guards increases the general-case overhead of remapping. Even worse, a small number of "straggler" threads may end up falling through *all* the guard conditions; the corresponding increase in overhead delays the entire program and loses the performance benefit for all other threads.

Plastic takes advantage of the locality of accesses within threads by *not* instrumenting blocks to contain a comprehensive list of guard conditions. Instead, individual thread-specific versions of a code block are generated to contain exactly the guards necessary to handle the remapping of data accessed by that specific thread. As a result, threads executing their specialized version evaluate as few conditionals as possible and fault on accesses when an appropriate guard is missing. This fault may then result in the generation of an alternate block containing the necessary additional guards.

Transferring Execution: While transferring execution from the original code to the code cache is trivial in the case of a single instruction, entire code blocks pose two problems to execution transfer. First, rather than transferring execution to the start of the instrumented block, it has to be transferred to the correct instruction within it. Second, for multi-threaded applications, Plastic must safely migrate all threads concurrently executing within that code block.

Plastic maintains mappings between the original code and the code cache using the Translation Offset Index. During page faults, it selects a specialized block with guards for the faulting address and transfers execution to the instruction corresponding to the faulting instruction in that block. The offset index allows faults on *any* instruction in the original binary to be appropriately redirected to the code cache. This

is important because at the time a remapping is applied, several worker threads may be executing different instructions within the code block being remapped.

Figure 8 illustrates this transfer for two threads, T_1 and T_2 that contend on independent, thread-specific data. T_1 accesses data described by rule A and triggers a page fault that generates a specialized version of the code block with guards for both rules A and C. The second guard is added because it describes the common case of accesses to the remainder of the original page. Meanwhile, T_2 continues executing the original code until it faults on a memory reference, in this case one associated with rule B; Plastic then generates a new specialized block for rules B and C, and transfers execution to it. In this regard, the combination of faults on data access and offset index matching between the original and specialized versions of code allow threads to be efficiently and safely redirected to the suitable specialized version.

5.3 Safety of Instrumented Code

Plastic guarantees that applied code transformations do not alter program functionality in any manner. To ensure this, it performs extremely simple transformations: every faulting instruction in the original code has an identical counterpart in the code cache. Instrumentations simply modify the referenced addresses and invariants that hold for the original execution also hold for the version within the code cache.

Plastic leverages the insight that, as a performance optimization, it can afford to be sound, but not complete, with respect to the instruction set. While systems providing strong security guarantees [13, 39] have to contend with several corner cases in x86, Plastic simply invalidates all the remappings when it encounters instructions it cannot safely redirect.

Invalidating remappings is simplified by the fact that the original code remains unchanged at all times. Data from the split pages is merged back to the original which is then unprotected. Execution transfer is mirrored from earlier – permission to the split pages is revoked and Plastic transfers accessors from the code cache back to the corresponding accessors in the original code. With both code and data restored to their original state, execution continues unhindered.

Code Coverage: As discussed in Section 5.1, Plastic prevents stray accessors from modifying the original data by marking the page as "No Access" throughout the lifetime of the remapping. Guard conditions ensure that accesses to data lying outside the remapped region, even from within the code cache, do not get arbitrarily remapped.

Thread Safety: Plastic serializes handling page faults for the data page and the subsequent generation of instrumented code blocks. Such faults are infrequent, so serialization does not significantly affect performance, while simultaneously preventing race conditions due to concurrent accesses.

Leaked Pointers: Applications remain unaware of the remapped data ranges and as such all accesses to these ranges originate from within the code cache. Improperly restoring register state, however, may reveal these ranges to the original code. Such leaked pointers are dangerous: an application may inadvertently manipulate them and attempt to access undefined regions of memory. Plastic avoids leaked pointers by immediately restoring the original memory address after executing the faulting instruction.

Unsupported Instructions: Plastic invalidates existing remappings on encountering memory references that it cannot redirect. A list of several such instructions and the difficulties involved in their redirection follows. While workarounds for several of these instructions exist, they are currently not implemented in Plastic.

1. Repeat Prefixed Instructions: Instructions like `rep movs` and `rep cmps` may access several bytes of memory, spanning both remapped and non-remapped regions, using a single instruction. Plastic cannot detect the ranges of memory involved and redirect accesses just by modifying the parameters prior to execution. Instrumentation frameworks like Pin [25] explicitly convert such instructions into loops to overcome this issue.
2. Atomic Accesses: Memory accesses that straddle remap regions cannot safely be redirected without changing the instruction into a set of smaller granularity memory accesses. Examples of such scenarios include contiguous bytes updated by independent threads, causing them to be remapped to independent cache lines, that are later read by a single atomic 32 or 64-bit read.
3. `mov %rXx, (%rXx)`: Directly modifying instructions that write their address to their memory location lead to leaked pointers as remapped addresses get written back to memory. Such instructions can be redirected by using an extra register to hold the original address which is written to the remapped memory region.
4. Register Indirect Branches: Indirect branches, such as `jmp *%rXx` and `call *%rXx`, accessing function pointers tables located on protected pages need to be redirected to the remapped pointer table. Once the branch is taken, however, Plastic cannot restore the register to its original value. Such branches can be replaced with `rip`-relative branches referencing the remapped pointer table.

Lastly, there is the safety of the code cache itself to consider. The code cache must reside in the address space of the application and remain accessible at all times; its pages, however, are marked as read-only to prevent modification from within the application. While an application may attempt to jump into the middle of a block to avoid the guards, there is little value in this as it would only hinder the application itself.

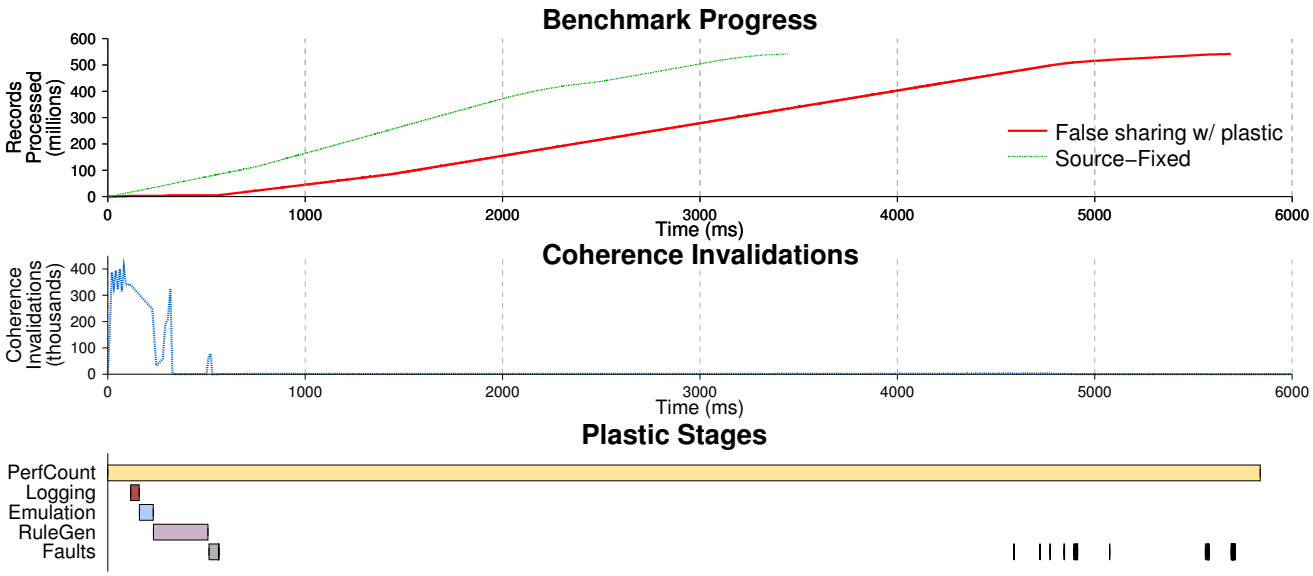

Figure 9. Linear regression running under Plastic.

6. 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, per-socket L3 cache. Plastic runs on Xen 4.2 with a Linux Dom0. All tests are run on an 8-core guest with virtual processors pinned to the corresponding physical core.

First, we describe the detailed execution of a false sharing workload under Plastic, followed by a discussion of the memory overhead of the system. We then assess the performance impact of remappings within a code block on other callers of the same code block. Finally, we evaluate Plastic's performance across a range of different workloads, compiled with `gcc 4.4.3` at the default optimization level. Reported performance results are the average of twenty runs.

6.1 Functioning of Plastic

Detection Times and Execution under Plastic: Figure 9 represents execution of the `linear_regression` workload discussed earlier running under Plastic and compares it with that of a source-fixed version of the same program. Along with total benchmark progress, coherence invalidations are also shown for the version with false sharing.

As the workload starts it immediately causes a significant number of coherence invalidations; correspondingly, its throughput is only a fraction of that of the source-fixed version. At around 125ms, the performance counters detect the presence of contention and activate the rest of the pipeline. At around 500ms, the remapping rules are synthesized and threads are migrated to the code cache by 600ms through a series of page faults. Execution remains within the code cache, as indicated by the absence of any further faults. Consequently, throughput rises and the benchmark progresses rapidly, while the coherence invalidations correspondingly drop to almost zero indicating the lack of contention.

A single thread aggregating results from remapped data is responsible for the page faults near the end of the execution. As these are non-high-frequency accesses, Plastic remaps them on a per-instruction basis, resulting in the high number of faults. Plastic continues to sample performance counters throughout execution for any further instances of contention; in this case, however, no other contention is detected.

Comparing the throughput of the Plastic-fixed and source-fixed versions helps precisely define the overhead of the extra instrumentation required for remapping. Once execution is transferred to the code cache, the throughput of the Plastic-fixed version is 110M/s compared to a throughput of 160M/s for the source-fixed version – a performance loss of 31%. The remaining difference in overall throughput is due to the false sharing before the remappings are applied.

Memory Overhead: Plastic trades memory for higher performance by requiring additional memory for the instrumented code and to pad and isolate contended cache lines. In practice we find that a reservation of 64 pages (256 KB) is sufficient for most remapping scenarios and does not significantly reduce the VA space available to applications.

Impact of Remappings on Other Callers: A simple microbenchmark helps verify the assertion in Section 5.1 that callers of code blocks not referencing remapped data are unaffected: a function executes referencing non-contended

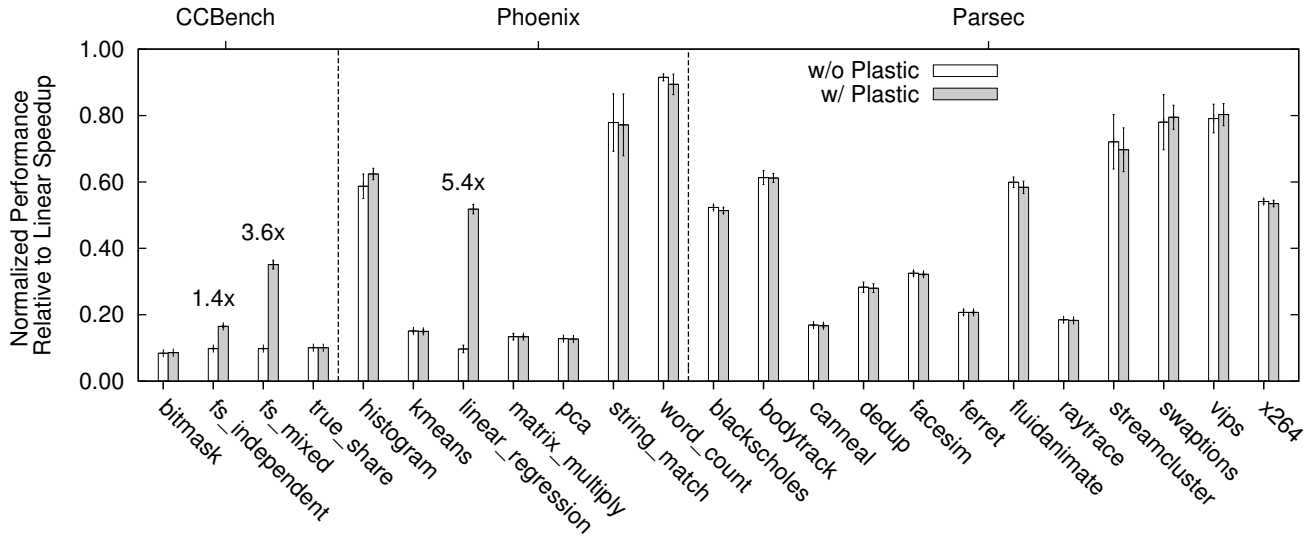

Figure 10. Performance of Phoenix, Parsec and CCBench suites running with Plastic.

memory followed by an execution referencing contended memory that triggers remapping. Finally, the original execution is repeated. Non-coherence misses are ignored since data resides on a single cache line. The last execution has under 1% overhead compared to the first, demonstrating the negligible impact on callers not accessing remapped data.

6.2 Performance Analysis

Plastic is evaluated by comparing the performance of several workloads running under Xen, normalized against their single threaded performance, with and without Plastic. While this data, shown in Figure 10, ignores the virtualization overhead, we find it to average 3% over the benchmarks.

Plastic samples contended pages for multiple 2ms epochs, followed by 250ms of emulation. Stage lengths can be varied if desired and, like in any sampling, represent a trade-off between the speed and accuracy of detection. Nevertheless, we believe they are sufficient for most high-frequency false sharing and are, in practice, able to accurately detect the exact regions of false sharing in all the workloads evaluated.

The CCBench: Performance artifacts due to memory contention in real workloads, like those in Section 2, often only manifest themselves at a not-yet-common degree of scale. To replicate these effects with fewer cores, we have developed *CCBench*, a suite of microbenchmarks that model their memory access patterns on real workloads, but exacerbate their effects by contending at higher frequencies.

Table 1 briefly describes the different microbenchmarks and the real workloads they emulate, while their performance is seen in Figure 10. Including both true and false sharing workloads allows us to measure both the performance improvement when Plastic is able to "fix" the problem as well as the impact on performance when it is unable to do so.

`fs_independent` is a classical example of false sharing with multiple readers and writers accessing independent values in a global array. It is modeled after spinlocks in a lock pool [27] or the bytes used to represent the dirtied status of pages during Java garbage collection [11]. Isolating the sets of values accessed by a thread onto independent cache lines reduces execution time from 18.4s to 5.1s, a speedup of 3.6x.

`fs_mixed` involves false sharing between a shared read-only data range and a shared read-write range, with accessors equally distributed between both data ranges. Read-mostly spinlocks in the `net_device` structure in Linux contend with the transmission and receive queues in a similar manner [7]. Remapping splits these data ranges and reduces execution time from 18.7s to 11s, a modest performance improvement of 40%. Comparing the performance of accessors now shows a bimodal distribution: accessors to the read-only range are contention-free and display a speedup of 6.6x, but overall program performance is limited by accessors to the read-write range still suffering from some contention.

`bitmask` models bitmasks like the flags in the `page` structure in the Linux kernel. A combination of read-only and read-write flags leads to false sharing *within* a single byte, alleviated only by splitting the flags into discrete versions [7]. From a byte-level perspective, however, bitmasks represent true sharing and do not benefit from remapping.

`true_share` simulates lock contention with concurrent reads and writes to the same memory location.

`bitmask` and `true_share` represent pathological scenarios for Plastic: high degrees of true sharing that are impossible to distinguish from false sharing without emulation. De-

Name	Description	Examples	Fixable?
fs_independent	Multiple accessors to independent variables (At least one writer)	Linear Regression in Phoenix [35] spinlock_pool in Boost [27] Bookeeping in the Java GC [11]	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 1. Microbenchmarks in CCBench. The Fixable column denotes whether it can be fixed by simply remapping memory.

spite this, due to the sampled nature of emulation, both show little overhead and perform within 1% of normal execution.

Shared Memory Benchmarks: Plastic's effectiveness with real-world benchmarks is evaluated against several applications from the Phoenix [35] and PARSEC [5] benchmark suites. Both of these suites are specifically designed for shared memory workloads and are representative of applications from several different domains.

Plastic fixes significant amounts of false sharing in linear_regression, showing a speedup of 5.4x compared to normal execution. For the remaining workloads, Plastic imposes an average of 3% overhead, demonstrating that it does not adversely impact workloads without false sharing.

As a live detector, Plastic prioritizes performance over completeness and focuses on detecting high-impact false sharing. While this does lead to some instances of false sharing remaining undetected, we quantify the degree of false sharing in these instances by comparing against other detectors. Sheriff [24] detects false sharing in streamcluster, swaptions, histogram, string_match, and word_count in addition to linear_regression.

Out of these, streamcluster, swaptions, and word_count show less than 5% degradation due to false sharing and do not have enough contention to trigger the detection pipeline. False sharing in string_match is caused due to the heap allocator. Not only is this not observed on our system, but in practice it scales well up to 8 cores.

In contrast, while histogram definitely suffers from false sharing, the performance impact is found to be only around 25% when compared against a source-fixed version. This does not warrant remapping because, unlike the case in linear_regression, fs_independent, and fs_mixed, the benefit of mitigating false sharing no longer masks the remapping overhead discussed in Section 6.1.

The benchmark results highlight the differences in approach taken to false sharing detection and mitigation by Sheriff [24] and Plastic. Sheriff forces threads within a process to operate on private pages, set up using copy-on-write semantics, and merges them at synchronization points. False sharing is avoided simply by batching updates to contended memory regions. linear_regression has little synchronization and exhibits a 9x speedup. In contrast, Plastic has a low throughput detection phase prior to applying the remappings and shows a speedup of 5.4x. By avoiding expensive page copy operations, Plastic does an excellent job of uniformly imposing low overhead on workloads that do not exhibit false sharing. In contrast, Sheriff shows significant overheads in programs with frequent locking such as fluidanimate and canneal.

We find that Plastic can quickly and accurately detect and correct false sharing with low overhead. In cases where false sharing exists, but imposes only a small overhead, it is able to correctly value its potential to improve performance and do no harm. At the same time, Plastic can significantly improve the execution of workloads where false sharing would otherwise impose a crippling scalability limitation.

7. Related Work

Several existing systems study the cache subsystem and detect false sharing in existing application workloads. Plastic also shares similarities in detection mechanisms with race detectors and other systems designed to diagnose memory contention issues apart from false sharing.

False Sharing: Sheriff [24] shares a similar goal to Plastic in transparently detecting and fixing false sharing in production environments. It splits threads into separate, independent processes, each of which has private page tables. Changes to memory are localized to a private copy of modified pages which are merged together at synchronization points. False sharing is detected by identifying interleaved writes at a cache line granularity, while mitigation simply reduces the frequency of accesses to contended cache lines.

Sheriff makes assumptions about the use of the pthread API for synchronization and may break correctness if these assumptions are violated; for instance, in the case of lock-free data structures. Similarly, locks and other synchronization primitives located on the same page as contended structures may prevent the successful mitigation of false sharing.

Zhao, *et al.* [40] use memory shadowing to track ownership of cache lines and analyze thread-access patterns with-

out full cache simulation. Using DynamoRIO [8] for instrumentation, they help detect cache contention with around 5x overhead, and makes no attempt to mitigate the problem.

DProf [34] is a data profiler that associates access costs with data rather than instructions. It helps developers identify memory regions frequently experiencing high access costs, including due to true and false sharing, but does not automatically distinguish between the causes or help mitigate them.

Intel Performance Tuning Utility [21] uses performance counter monitoring to identify contention, but does not distinguish between true and false sharing. Unlike Plastic, it neither attempts to analyze the performance effects of this contention, nor does it attempt to fix them in any way.

Several approaches for detecting cache contention model the cache in software [16, 22, 37], and are clearly intended for development use only. Pluto [16], a Valgrind based instrumentation engine, imposes around two orders of magnitude overhead, while Cmp$im [22], which uses Pin [25] to simulate the entire memory hierarchy and coherence algorithms in software, runs at 4-10 MIPS.

Race Detection: Like detecting false sharing, race detection requires instrumenting every memory access to log the ordering of accesses to shared memory. Aikido [32] uses hypervisor-based, per-thread shadow page tables to identify contended pages for further analysis, in a manner similar to Plastic. Greathouse, *et al.* [14] use coherence invalidations as a trigger for more heavy-weight, software-based analysis to identify actual data races. As a race detector, however, they treat any such invalidation as suspicious and perform further analysis, without analyzing its performance impact.

8. Discussion and Future Work

In order to maximize performance, Plastic leverages hardware facilities whenever possible, only resorting to software interposition when the required features are not exposed by the hardware. Processors, however, are constantly evolving and some additional features may further reduce overhead.

Performance counters operating in sampling mode, called Precise Event Based Sampling (PEBS) or Instruction Based Sampling (IBS) on Intel and AMD processors respectively, store processor state on sampled occurrences of selected microarchitectural events. Sampling coherence invalidations stores processor state when contended memory accesses occur. Unfortunately, on Nehalem processors this does *not* include the memory address accessed. Recording this address, as proposed for future architectures, would allow Plastic to identify contended memory regions directly in hardware.

Plastic relies on page-granularity protection to detect access to contended memory, unfortunately resulting in faults for all accesses to that page. Hardware watchpoints overcome this by detecting accesses at byte-granularity, but are extremely limited in number. An unlimited number of watchpoints [15] would help significantly reduce the number of faults.

Language runtimes that manage memory for their applications independently of the OS could suffer unnecessary performance degradation due to remapping. Plastic invalidates applied remappings to maintain program safety, but does not currently extend this facility to monitor the performance impact of the remapping and, if necessary, restore execution to the original code. Plastic also cannot be disabled on a per-application basis, but could be extended to provide this facility with support from the guest OS.

Lastly, dynamically fixing false sharing should be a last resort only for when statically fixing the problem in source is not possible. Plastic's detection engine could be extended with debug symbols to provide developers source-level reports of the contending data structures.

9. Conclusion

Plastic demonstrates that, by taking responsibility for monitoring and managing coherence misses in its caches, a system can dynamically recover from workloads that exhibit pathological false sharing. In order to achieve this, the system tackled two challenging problems. First, the aggregation and integration of a variety of monitoring and diagnosis techniques, including hardware performance counters, shadow paging, and instruction emulation to quickly and precisely identify false sharing with low overhead. Second, the system demonstrated a sub-page granularity remapping facility that is sufficiently high-performance as to show a speedup of 3-6x in cases of high-rate false sharing.

10. Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Gilles Muller, for their valuable feedback. Geoffrey Lefebvre provided much valuable advice regarding the design, while Mike Kozuch and Babu Pillai at Intel Labs, Pittsburgh patiently permitted us to experiment with their multi-core machine. Finally, our thanks to Tim Deegan, Steve Hand, Malte Schwarzkopf, Tim Harris, and several members of the systems lab at UBC for comments and suggestions at various stages of this work.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[3] T. Bergan, N. Hunt, L. Ceze, and S. Gribble. Deterministic process groups in dos. In *OSDI*, 2010.

[4] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *PASTE*, 2011.

[5] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Workshop on Modeling, Benchmarking and Simulation*, 2009.

[6] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *SEDMS*, 1993.

[7] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *OSDI*, 2010.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, 2003.

[9] M. Burrows, U. Erlingsson, S.-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. In *ASPLOS*, 2000.

[10] B. Dawes, D. Abrahams, and R. Rivera. Boost C++ libraries. *http://www.boost.org*, 2009.

[11] D. Dice. False sharing induced by card table marking, February 2011. URL `https://blogs.oracle.com/dave/entry/false_sharing_induced_by_card`.

[12] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *OSDI*, 2006.

[13] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC*, 2008.

[14] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. In *ISCA*, 2011.

[15] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A case for unlimited watchpoints. In *ASPLOS*, 2012.

[16] S. M. Gunther and J. Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *WBIA*, 2009.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 5 edition, 2011.

[18] M. Herlihy and J. Moss. System for achieving atomic non-sequential multi-word operations in shared memory, June 27 1995. US Patent 5,428,761.

[19] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, and et al. *A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS*. IEEE, 2010.

[20] Intel. Avoiding and identifying false sharing among threads, November 2011. URL `http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/`.

[21] Intel. Intel performance tuning utility, October 2012. URL `http://software.intel.com/en-us/articles/intel-performance-tuning-utility/`.

[22] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob. CMPSim: A pin-based on-the-fly multi-core cache simulator. In *MOBS*, 2008.

[23] D. Levinthal. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors, 2008.

[24] T. Liu and E. D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *OOPSLA*, 2011.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[26] M. Martin, M. Hill, and D. Sorin. Why on-chip cache coherence is here to stay. *CACM*, 55(7):78–89, 2012.

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

[28] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *PACT*, 2009.

[29] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[30] R. J. Moore. A universal dynamic trace for linux and other operating systems. In *USENIX ATC*, 2001.

[31] M. Olszewski, K. Mierle, A. Czajkowski, and A. D. Brown. JIT instrumentation: a novel approach to dynamically instrument operating systems. In *EuroSys*, 2007.

[32] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. P. Amarasinghe. Aikido: Accelerating shared data dynamic analyses. In *ASPLOS*, 2012.

[33] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA*, 1984.

[34] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *EuroSys*, 2010.

[35] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA*, 2007.

[36] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI*, 1999.

[37] J. Tao and W. Karl. CacheIn: A toolset for comprehensive cache inspection. In *International Conference on Computational Science*, 2005.

[38] C. Thacker. *Beehive: A many-core computer for FPGAs (v5)*. MSR Silicon Valley, Jan 2010. URL `http://projects.csail.mit.edu/beehive/BeehiveV5.pdf`.

[39] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, 2009.

[40] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *VEE*, 2011.