REMIX: Online Detection and Repair of Cache Contention for the JVM

Ariel Eizenberg*, Shiliang Hu†, Gilles Pokam†, Joseph Devietti*

*University of Pennsylvania, USA †Intel Corporation, USA

arieleiz@seas.upenn.edu, {shiliang.hu, gilles.a.pokam}@intel.com, devietti@cis.upenn.edu

Abstract

As ever more computation shifts onto multicore architectures, it is increasingly critical to find effective ways of dealing with multithreaded performance bugs like true and false sharing. Previous approaches to fixing false sharing in unmanaged languages have employed highly-invasive runtime program modifications. We observe that managed language runtimes, with garbage collection and JIT code compilation, present unique opportunities to repair such bugs directly, mirroring the techniques used in manual repairs.

We present REMIX, a modified version of the Oracle HotSpot JVM which can detect cache contention bugs and repair false sharing at runtime. REMIX's detection mechanism leverages recent performance counter improvements on Intel platforms, which allow for precise, unobtrusive monitoring of cache contention at the hardware level. REMIX can detect and repair known false sharing issues in the LMAX Disruptor high-performance inter-thread messaging library and the Spring Reactor event-processing framework, automatically providing 1.5-2x speedups over unoptimized code and matching the performance of hand-optimization. REMIX also finds a new false sharing bug in SPECjvm2008, and uncovers a true sharing bug in the HotSpot JVM that, when fixed, improves the performance of three NAS Parallel Benchmarks by 7-25x. REMIX incurs no statistically-significant performance overhead on other benchmarks that do not exhibit cache contention, making REMIX practical for always-on use.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments; C.1.4 [*Processor Architectures*]: Parallel Architectures

Keywords false sharing, cache coherence, Java

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908090

1. Introduction

Multicore architectures drive an increasingly broad swath of computing devices, from servers to wristwatches, providing superior performance and energy efficiency to uniprocessor designs. However, multicores bring with them many well-known correctness and performance challenges. In this work we focus on an important class of multithreaded performance bugs called *cache contention bugs*. Cache contention bugs come in two varieties: *true sharing* and *false sharing* (see Section 2 for more details). Contention bugs are often difficult to discover because they arise due to subtle interactions between memory layout and cache coherence protocols. Such interactions are obscure or even invisible at the source code level. Significant cache contention bugs have been found in many software systems, including the Linux kernel [3, 6, 7], the MySQL database [38], the Boost C++ libraries [33], as well as several benchmark programs [27, 28, 34].

Previous systems that find and fix cache contention at runtime [27, 28, 34] have focused exclusively on false sharing in unmanaged C/C++ programs. Many existing profilers for managed languages [12] can identify some forms of true sharing like lock contention. However, code written in managed languages such as Java or C# can suffer from false sharing as well. Managed languages are very popular among today's developers, and false sharing bugs are arguably even harder to find and fix in managed code than in unmanaged code, as programmers have less control over memory layout in a managed language. False sharing can appear and disappear based on the opaque decisions of the garbage collector or the JIT compiler.

At the same time, managed runtimes afford much greater potential for repairing false sharing once it is identified. Previous work targeting unmanaged languages [27, 34] has had to resort to heavyweight virtual memory mechanisms to both find and fix false sharing. These mechanisms are relatively invasive and not well-suited for integration with a complex managed language runtime. In contrast, managed runtimes have the potential to fix false sharing the same way a programmer would – by directly adjusting memory alignment and padding – through the use of JIT compilation and garbage collection to modify the memory layout of live objects at runtime. By using efficient, natural repair mechanisms, a managed runtime system can potentially achieve the same performance as that of a source-code repair, but without the need for any programmer involvement.

In this paper we describe the REMIX system, a modified version of the Oracle HotSpot JVM that can automatically discover at runtime all forms of cache contention bugs and repair false sharing bugs in programs running on the JVM. To the best of our knowledge, REMIX is the first system to detect or repair false sharing in any managed language. REMIX relies on advanced performance counters available on recent Intel processors, which provide a uniform mechanism for detecting all forms of cache contention unobtrusively and with very low performance overhead. These performance counters provide unprecedented visibility into the behavior of application code and of the language runtime itself. Because REMIX integrates with the JVM, it natively supports all JVM languages, *e.g.*, Java, Scala and Clojure. REMIX is implemented in just 2500 lines of new code, modifying only 250 lines of existing JVM code, suggesting that it can be easily implemented in other managed runtimes.

This paper makes the following contributions:

- We demonstrate the use of commercially-available hardware performance counters to accurately identify both true and false sharing in large-scale Java and Scala workloads and within the JVM itself
- We propose new online repair techniques for false sharing enabled by managed runtime systems
- We present an implementation of the REMIX system in the Oracle HotSpot JVM
- We show that REMIX repairs false sharing issues as efficiently as hand-optimized code, while incurring essentially zero overhead for programs without cache contention.

2. Background

The fundamental source of cache contention is invalidation-based cache coherence protocols used in modern multiprocessor systems. Modern protocols enforce a *single-writer multiple-reader* (SWMR) invariant which means that a given cache line can, at any give time, either be writable by a single core or readable by multiple cores. Cache contention arises when two or more cores make repeated accesses to a given line, and at least one of those cores writes to the line. The coherence protocol serializes accesses to the line, resulting in substantial time and energy overhead.

Cache contention can arise in two varieties, depending on whether cores make contended accesses to the same bytes within a cache line (which is called *true sharing*) or to different bytes within the line (*false sharing*). True sharing often arises in the form of lock contention, though contention on volatile fields or regular data is also possible. False sharing is often invisible in source code, as it arises when two distinct and frequently-accessed objects end up being allocated in the same cache line. In managed languages, false sharing can appear and disappear as a copying garbage collector moves objects around in memory.

It is generally more difficult to repair true sharing than to repair false sharing. Fixing true sharing typically requires some form of application restructuring, *e.g.*, replacing a global counter with distributed local counters. False sharing, on the other hand, can be resolved by taking existing data structures and altering their alignment or adding padding. Such repairs do not constitute a semantic change to the program and are thus simpler to implement. Moreover, only a small amount of data can be subject to meaningful cache contention, as contention spread across a large region does not affect performance, so the amount of padding needed to repair false sharing is typically small.

Programmers in high-level languages like Java have little direct control over memory layout, as the layout of fields in memory is unspecified [25]. Padding must be carefully implemented to ensure field reordering does not undo the effects of padding (see Section 6.1 for an example). To help Java programmers fix false sharing issues declaratively, Java 8 adds support for a new field annotation `@sun.misc.Contended` [35]. Placing `@Contended` on a field instructs the JVM to add appropriate padding to the class definition to ensure that the field is isolated in its own cache line. Currently, 128B of padding are added before the first contended field, and another 128B after each contended field or field group (so multiple fields can be isolated together within a line). `@Contended` is, however, solely a repair mechanism, so programmers are responsible for both identifying where `@Contended` annotations are necessary and for not wasting space with overuse. Furthermore, `@Contended` cannot be added by the programmer to classes in standard libraries, or 3rd-party code to which the programmer has no source-code access. Another disadvantage of using `@Contended` is that memory usage is increased even if the program exhibits no cache contention at all, while REMIX pads data only when runtime conditions show it necessary. For this reason, REMIX is able to outperform even hand-tuned code in some cases (Section 6.1).

3. Detecting Cache Contention with Hardware Performance Counters

Because the underlying source of cache contention is the coherence protocol, gaining visibility into runtime coherence events is a direct and unified mechanism for detecting both true and false sharing. Recent Intel multiprocessors offer the ability to log detailed information about certain architectural events via the Precise Event-Based Sampling (PEBS) performance counter mechanism [9]. When an instruction i trig-

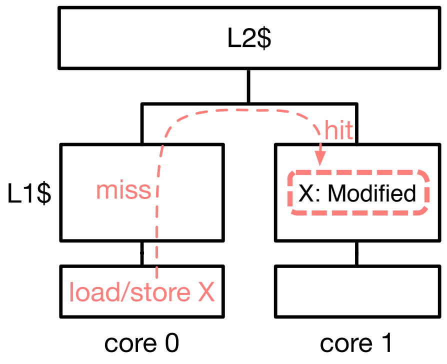

Figure 1: HITM events arise when one core's memory operation hits in a remote core's private cache, and the line in the remote cache is in the Modified state.

gers an event of interest, the PEBS mechanism generates a PEBS record which the hardware logs to an in-memory buffer. Each PEBS record contains i's PC, the memory address accessed by i (if i is a load or store), and the values of the general-purpose registers as of i's commit. When the buffer is full, an interrupt notifies the OS's PEBS driver to process the records and provide a new buffer for the hardware to use.

Modern Intel processors support many PEBS events. Of particular interest for us is the ability to track HITM coherence events, which arise when one core's memory request hits in a remote core's private cache, and the requested line is in the Modified state in the remote cache (HITM stands for Hit-Modified, Figure 1).

Despite their name, PEBS events are not fully precise. We have found through empirical testing on our experimental platform that PC in a PEBS record is more accurate than the data address. Performance counter accuracy tends to improve across processor generations, so we use a recent Haswell processor for our study (Section 5). The name of the HITM event we use implies that only load instructions generate PEBS records. We have found this not to be the case: stores that trigger HITM events do cause PEBS records to be generated, though at a much lower rate than for HITM-triggering loads.

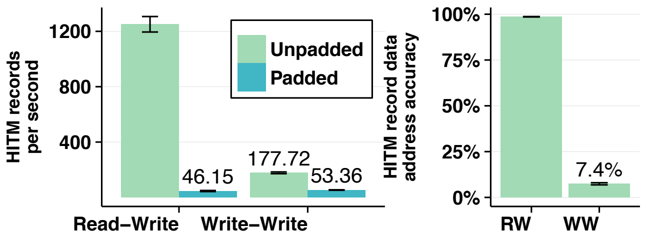

Figure 2: The rate at which HITM records are generated (left) and the accuracy of the data addresses in those records (right) for our microbenchmarks.

We developed some simple microbenchmarks to characterize the PEBS HITM mechanism on our 8-core Haswell machine. We wrote two microbenchmarks, exhibiting intense read-write and write-write false sharing with 8 threads accessing thread-private objects that get allocated together in the same cache line. We also developed padded versions of each microbenchmark that generate no false sharing. HITM events are sampled at a rate of 10,000 samples per second via the Linux perf facility.

The left side of Figure 2 shows the rate of HITM records received for the unpadded and padded versions of our microbenchmarks. Unpadded read-write benchmarks generate many more HITM records, providing a clear signal of the contention occurring at the hardware level. Adding padding removes the false sharing and the source of HITM records. In contrast to the read-write benchmarks, write-write false sharing sends a quieter signal, though the HITM rate still declines noticeably when padding is added.

The HITM records generated due to read-write contention have very accurate data addresses – 98% of the records refer to the heap objects where false sharing occurs. With write-write contention, however, only about 8% of the samples are accurate. Thus, it is more difficult to detect write-write contention than read-write contention via the PEBS HITM mechanism, though detecting both forms of contention is still feasible. We suspect the lower accuracy for write-write HITM events stems from the way hardware store buffers are implemented. Store buffers delay the completion of stores so that, by the time a HITM event occurs, the store instruction may have long since committed and the information needed for a precise HITM record is less likely to be available. Fortunately, our detection mechanism is forgiving of noisy input and requires only that a plurality of input data is correct to successfully locate the source of contention.

4. The REMIX System

The REMIX system relies on existing hardware and operating system support for the advanced PEBS performance counters available in recent Intel processors. As Figure 3 illustrates, REMIX uses the Linux perf API [32] to configure the hardware to record HITM events. perf support is a standard part of recent versions of Linux, and allows HITM events to be recorded entirely from userspace. Root permissions are not required for a process to monitor its own HITM events. The application code running atop the REMIX JVM is completely unmodified, which allows REMIX to work transparently with any JVM language. However, for simplicity, we refer to all application code as Java code in the remainder of this section.

HITM records received from the hardware enter the REMIX processing pipeline (Section 4.1), where they are classified as true or false sharing. If false sharing is detected, REMIX maps the contention to Java classes (Section 4.2)

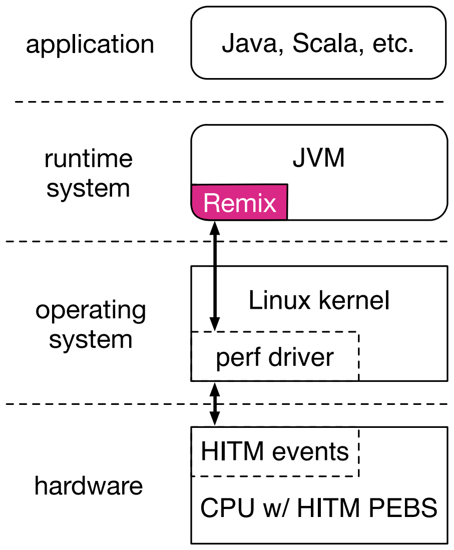

Figure 3: An overview of the REMIX system. HITM events flow from the processor through a kernel driver to the REMIX JVM extension. Note that OS and application code are unchanged.

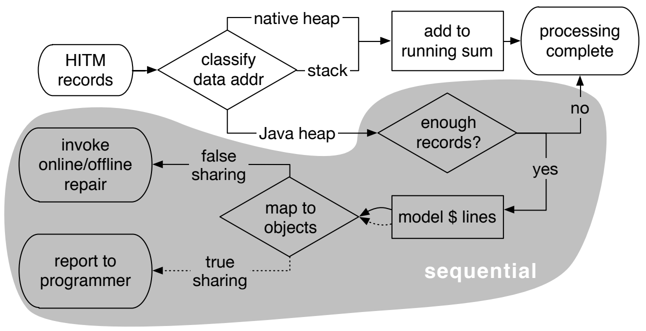

Figure 4: How REMIX detects true and false sharing. HITM record collection from perf and data address classification run in parallel on each Java thread. Subsequent stages (the shaded region) are performed sequentially after all Java threads are stopped.

and then either repairs the contention online as the program continues to run (Section 4.3) or generates a set of offline annotations for use with subsequent runs (Section 4.4).

4.1 HITM Record Collection

REMIX continually collects the HITM records generated by an application, processing those events periodically. Continual monitoring allows REMIX to detect contention that migrates over time or that only arises in certain phases of an application's execution.

REMIX uses the builtin Linux perf kernel support to gather HITM records from the hardware. REMIX allocates a small memory mapped region for each JVM thread executing Java code, and uses this region for asynchronous communication with the kernel's perf mechanism. The kernel logs HITM records into this region until they are processed by REMIX. While HITM records sometimes contain inaccuracies (Section 3), we found that incorrect HITM records usually contain NULL or nonsensical data addresses. We instruct perf to ignore HITM events from kernel memory so any address not mapped in the program's address space can be safely ignored.

To maintain efficiency, REMIX processes the per-thread HITM buffers only at stop-the-world (STW) events, such as biased lock revocation and STW garbage collection, that occur regularly regardless of the GC and that require all Java code execution in the JVM to pause and the stack layouts to be completely known. Piggy-backing on existing STW events minimizes REMIX's performance impact, but increases the latency of contention detection and repair. Each per-thread HITM buffer is read at each stop-the-world event, where it may be partially full. We use a 512KB buffer for each thread and never found a buffer completely full, suggesting that losing HITM records due to infrequent buffer reading is not a problem in practice.

When the JVM initiates a STW operation, the VM thread signals all the running Java threads to stop at their next safepoint. Before each thread stops, it classifies HITM records by data address as belonging to the JVM's collected heap, to the native heap or to thread stacks (Figure 4). Records for the native heap and stacks are accumulated in simple global counters – these counts can later be used by the programmer to diagnose cache contention within the JVM itself (see Section 6.2).

After the Java threads are stopped, REMIX aggregates the per-thread results. If the total number of collected HITM records falls below a predefined threshold, no further processing is performed on these HITM records. If the number of HITM records is sufficiently high, REMIX proceeds to differentiate between true and false sharing via cache line modeling. Cache lines are modeled with 64-bit bitmaps, with each bit signifying one byte of the 64-byte cache lines on our system. For each record, we find its corresponding cache line and set the bit for the first byte of the record's data address. Because of the type safety guarantees provided by the JVM, two overlapping memory accesses must have the same starting address, so it is safe to ignore the width of the memory access. After all records are processed, cache line bitmaps with only one bit set indicate true sharing. Cache lines with multiple bits set indicate one of two cases: a) false sharing or b) a mix of true and false sharing, with a) being far more common. It is possible to disambiguate these two cases with substantial extra bookkeeping to track how frequently each thread accesses each byte of a line. Moreover, we can easily distinguish these cases using REMIX's repair mechanism: once the falsely-shared fields are separated, any remaining contention can be clearly identified as true sharing. If high levels of true sharing are detected (we use a threshold of

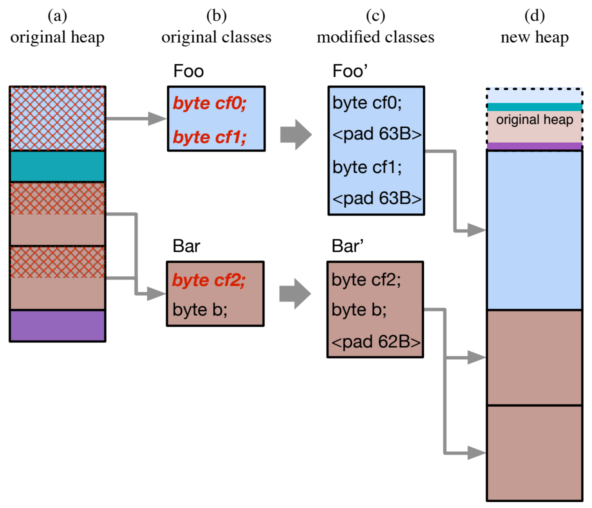

Figure 5: The REMIX detector maps contention (red cross-hatching) on raw heap addresses (a) to fields in corresponding Java classes (b, contended fields in italics). Then, the repair mechanism determines how to pad the class to isolate contended fields (c). Finally, the contended objects are padded as they are moved to the end of the heap (d).

1000 HITM records/second), REMIX stops collection until the next STW event to keep overheads low.

4.2 Online Contention Detection

To provide useful feedback to the programmer and the REMIX repair scheme, raw data addresses must be mapped to Java-level locations, *i.e.*, fields, array elements or object headers. REMIX is capable of identifying contention on any of these locations but for simplicity we only discuss classes and fields in this section. The Java heap consists of multiple generations, containing blocks of memory where each block represents one Java object. Each block begins with a metadata header containing marking information and a pointer to a special klass object which represents the Java Class of the object. Traversing the heap is accomplished by walking the blocks sequentially, with the block's size determined from layout information in the klass object, rather than any data encoded in the block itself.

Figure 5(a) shows an example heap, with contended addresses denoted by red cross-hatching. An instance of the Foo class (in blue) exhibits intra-object contention on two different fields of the same instance, while instances of the Bar class (in brown) suffer from inter-object contention on the same field in two different instances. To map contention onto fields, REMIX leverages heap indexing structures, *e.g.*, the BlockOffsetTable in the CMS and G1 collectors, if available. Some collectors do not have such indices, such as the ParNew GC used with REMIX's online repair mechanism, and in these cases REMIX traverses the entire Java heap, block by block. This traversal is inexpensive on modern hardware: in our experiments, a 16GB heap with 10M objects can be traversed in less than 100ms. As REMIX repairs false sharing the rate of HITM events drops and the need for traversal with it.

With explicit heap traversal, if the address of a HITM record resides within a block, the offset from the block start identifies which field is contended. During the traversal, REMIX builds a list of classes containing contention (Figure 5(b)). If the contended fields are identified as true sharing, REMIX reports the field names to the programmer for manual inspection. If false sharing is identified, then REMIX automatically initiates either online (Section 4.3) or offline (Section 4.4) false sharing repair.

4.3 Online False Sharing Repair

The REMIX online repair mechanism takes a straightforward but effective approach to false sharing repair, by adding padding between fields and to the end of objects. Fields are not reordered to avoid introducing additional false sharing. REMIX forces a separation of at least 64 bytes between any two contended fields. The mechanism ensures that there are 64 bytes between the last contended field and the end of the class. This approach ensures that no extra padding is introduced between fields that are already far enough apart, and increases class size by only a limited amount.

We first describe the challenges of modifying object layout at runtime (Section 4.3.1), and subsequently describe how REMIX addresses these issues (Sections 4.3.2-4.3.4).

4.3.1 Challenges of Runtime Object Modification

Changing the layout of just the contended instances of a class is impossible, as all instances of a class must be binary compatible. Moreover, modifications must respect inheritance. Figure 6 shows an example where REMIX discovers false sharing on a field a1 in an instance of class B. Since a1 is defined in class A, padding must be added not just to instances of B, but to any instance that contains field a1, *i.e.*, instances of both A and its subclasses like C. Assuming 64B cache lines and that there is no contention on the 16B object header, 48B of padding is necessary before a1. 62B of padding is necessary after a1, leveraging the uncontended field a2. This padding ensures that any instance of A (or its subclasses) will be insulated from false sharing no matter how it is allocated. As the memory layout at the bottom of Figure 6 shows, even if an instance of A is allocated directly between two other contended fields x and y, false sharing with x and y will be avoided. While the need for binary compatibility within a class hierarchy can in theory introduce lots of unnecessary padding, we find the space overhead to be minimal in practice (Section 6.5).

Changing a class's layout requires updating several internal HotSpot data structures. HotSpot stores **Java class fields** as an array of FieldInfos, containing information about the field, such as whether it is static or volatile, as well as its offset. REMIX modifies the class offset.

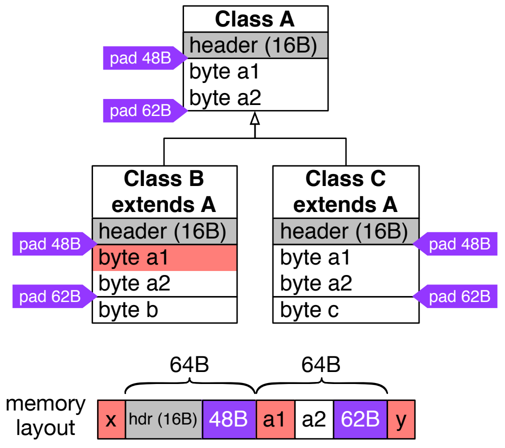

Figure 6: If false sharing is detected on a field *a1* in an instance of class *B*, padding *a1* affects the class *A* where *a1* is defined, and all of *A*'s subclasses. The memory layout on the bottom shows that, even if *a1* is surrounded by contended fields, it will be protected from false sharing.

Hotspot maintains an **OOPMap** (object pointer map) within each class definition. This table lists the field references inside the class, and is used by the GC to trace references through an instance of the class without iterating through the entire field list. The OOPMap must be updated to reflect how reference fields are moved due to padding.

Each Java class C is associated at compile-time with a **constant-pool** that contains the names of all classes, methods and fields used by C's methods. At runtime, names in this list are resolved and stored into a per-class cache, the **constant-pool cache**. This cache stores field offsets, so it has to be modified to reflect padding.

For **optimized methods**, HotSpot's JIT compiler inlines field offset information directly into generated native code. Furthermore, the JVM improves object allocation performance by using fast-path allocation for JIT-ed code that manipulates heap pointers directly, forgoing any explicit allocation function. Fast-path allocation is implemented by embedding the class instance's size into the generated code. Thus, even if no fields are moved and a class is only padded at the end, native code needs to be updated.

Finally, since instances are stored contiguously in memory, padding them requires relocating the instances to the end of the heap. The GC assumes the heap is contiguous, so the original blocks of these instances cannot remain empty. However, since a block's size is calculated from the class of the object within it, a block cannot remain an instance of the class as the class is now bigger than the block containing it, and traversing such a block would break the heap structure.

4.3.2 Class Modification

The list of classes to be modified is determined (Section 4.3.1) by traversing the entire loaded class list. For each class that needs modification, a *modification plan* is created, *e.g.*, everything after offset 16 has to be moved 40 bytes forward. Examples of padded classes are shown in Figure 5(c), note that the modification plan uses uncontended fields as padding where possible. A pointer to this plan is stored in the klass object. Note that the class size data structure is not yet modified at this stage, as it will break heap traversal causing mismatching block sizes. REMIX uses the class list traversal to fix up the constant pool. Each visited class has its constant pool cache examined for field information. For each class C to be modified, REMIX generates a *size-preserving* class C_{sp}, a dummy class exactly the same size as C was before modification. C_{sp} is used for the empty blocks left behind by modified instances, ensuring that the block size calculation does not produce any mismatches (Section 4.3.3).

4.3.3 Instance Modification

REMIX's instance modification requires a full heap traversal, but is optimized to only pad live instances of modified classes, as padding and relocating dead instances is unnecessary. Typically, only a small fraction of the heap requires processing, making REMIX's heap traversal much cheaper than a full GC which processes all live objects.

REMIX begins by marking all live objects on the heap. This is implemented by invoking the first phase of the mark-and-sweep GC, processing all strong roots in the system. Special care has to be taken with special reference objects, such as soft, weak, final and phantom references, to ensure that the instances referenced by these objects are marked as well. Luckily, the mark-and-sweep collector has a built-in mechanism for controlling the lifetime of soft references, and REMIX extends this mechanism to ensure all other reference types are considered as active during this marking phase, without modifying the regular GC's operation.

REMIX then traverses the allocated heap space to find instances of a modified class C. Found instances that are not GC marked are modified to point to the size-preserving class C_{sp}, and otherwise left alone.[1] Live instances are expanded, using the modification plan, to a new location at the end of the heap (Figure 5(d)). Since traversal requires that the class size field is not yet updated, special care is taken to ensure that heap traversal does not traverse any expanded objects, as at this point an expanded object's size mismatches its class size. Forwarding pointers are stored in the old location as during a regular GC.

Next, the 3rd phase of the mark-and-sweep algorithm is invoked as-is, tracing all strong roots and adjusting all pointers to forwarded objects.

The JVM implements a *deoptimize* capability to deoptimize active native Java stack frames and return them to interpreted evaluation. This feature is required to support aggressive optimization techniques such as Class Hierarchy

[1] ”Dead” classes that have finalizers are not considered dead by this pass, as the earlier heap trace marks finalizer heap references as live.

Analysis, as old frames are invalidated when new classes are loaded. REMIX extends HotSpot's existing on-stack replacement mechanism, which supports changes in method bodies, to also support changes in class definitions, similarly to the Jvolve system for dynamic software updates [43]. REMIX uses this facility to deoptimize any frame that accesses modified classes or classes with a modified constant pool cache.

Finally, the system modifies the size field of all modified classes, and performs housekeeping required due to the use of the GC mechanism such as restoring object markings, locks and other data structures.

While runtime object modification is complicated, the whole process usually runs just once or twice in a program's execution, and takes no more than 250ms even for large programs. Because REMIX can leverage the heavily-optimized JIT and GC machinery of modern language VMs, runtime object modification is viable for real-world applications.

4.3.4 Supporting sun.misc.Unsafe

sun.misc.Unsafe is used in several Java standard library classes (especially those for concurrent and atomic operations) and in other heavily-optimized libraries. One usage pattern involves getting the offset of a field f via Unsafe.objectFieldOffset(), storing the offset in a static field, and later using Unsafe operations to access f at the recorded offsets. Applied indiscriminately, REMIX's object modification can invalidate these pre-recorded offsets and cause memory corruption.

REMIX fully supports code that uses sun.misc.Unsafe. For safety, REMIX will not pad objects that have been accessed directly with Unsafe.objectFieldOffset() and warns the user if this restricts REMIX's repair facility. For the common case of Unsafe usage that does not touch falsely-shared objects, REMIX requires no code changes. For example, the LMAX Disruptor MultiProducerSequencer code (see Section 5) uses Unsafe in such a way and is supported without any source changes.

To support code that uses Unsafe.objectFieldOffset(), we implement a REMIX-aware sun.misc.Unsafe library with a slightly modified interface. Instead of returning the offset of a field directly, client code asks our modified sun.misc.Unsafe library to write the field offset to a particular static field on the client's behalf. Note that direct memory access using the offset is unaltered and is just as fast as with the original sun.misc.Unsafe.

Our library records the mapping between field offset and static field, updating the static fields accordingly if offsets change due to object modification. Modifying existing sun.misc.Unsafe usage for this new interface is a one-line change for each field accessed in this way.

4.4 Offline False Sharing Repair

In some domains, *e.g.*, with real-time applications, the extra complexity and variability of an online repair mechanism may not be desirable. To address such use cases REMIX implements an offline false sharing repair scheme. With this approach, REMIX uses HITM records to identify contended fields, and these contended fields are written to a profiling log. On subsequent application runs, the profiling log is used to pad classes with contended fields at class load time via the same padding algorithm used in the online repair scheme (Section 4.3). REMIX's offline repair operates entirely on bytecode and does not require access to program source code, unlike Java 8's @Contended annotation.

REMIX's offline repair does not require source code access, as it operates on compiled class files. When REMIX is used for offline repair, additional avenues for optimization are available, as performance collection is not required at regular intervals but can be limited to JVM operations that move objects, such as heap compaction and object promotion, and to the end of the application's execution.

Because REMIX's offline repair is less-coupled to the garbage collector implementation, we have integrated offline repair with HotSpot's concurrent G1 [13] collector. The G1 collector supports fast mapping from contended addresses to fields, accelerating REMIX's detection phase.

4.5 Limitations

The current REMIX system has a number of limitations. REMIX's online repair mechanism is simplified in our prototype by supporting only the stop-the-world ParNew GC. Extending REMIX to support concurrent GC presents a significant but tractable engineering effort. Both the CMS and the G1 GCs support locking the heap to enable foreground VM operations, and this can directly support REMIX's detection phase. Additionally, REMIX needs to be adapted to the CMS/G1 heap structures, mainly to support moving objects while padding them.

REMIX does not employ all possible false sharing repair strategies. REMIX does not reorder fields as reordering could create additional contention and possibly hurt performance by modifying the JVM's carefully-selected field order. REMIX does not address inter-object false sharing solely through reordering heap objects. Moving heap objects introduces extensive coupling with the design of the GC, as future collections must be informed to preserve the appropriate inter-object spacing. This also requires tracking contention on a per-instance, rather than per-class, basis which increases memory usage substantially. Once a class is selected for padding, REMIX will not later un-pad the class to see if contention returns. For our workloads the space overhead of padding is trivial (Section 6.5); reclaiming that space is not worth the cost of doing so and the risk of having to re-pad if contention arises again.

REMIX can detect false sharing between static fields and array elements, but does not yet repair these instances of contention as they were very uncommon among our benchmarks. Supporting static field repair would be a trivial extension. Repairing array false sharing is trickier as padding array elements can rapidly increase space usage.

5. Experimental Setup

We evaluate REMIX on an 8-core Haswell system with 32GB of memory and 64B cache lines. The processor is a 64-bit Intel Core i7-5960X running at 3.0 GHz, with HyperThreading disabled, TurboBoost enabled, and a max turbo frequency of 3.5 GHz. The operating system is Ubuntu 14.04.2 LTS with a custom-built, Linux 4.1.0 kernel with perf support enabled using the parameters listed in Table 1, which also lists the specific PEBS events we use. The REMIX JVM is based on OpenJDK8 build 132, compiled with gcc-4.9.2. The REMIX source code is available from `https://github.com/upenn-acg/remix`. All benchmarks were executed on the JVM with the -XX:+UseParNewGC to specify the parallel stop-the-world garbage collector, and -XX:-EnableContended to disable the operation of @sun.misc.Contended. The JDK8 HotSpot JVM was run in server mode (which includes the C2 JIT compiler), the only mode available in the 64-bit version.

Table 1: REMIX configuration parameters

PEBS HITM events
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM
MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_HITM
Linux kernel configuration
CONFIG_PERF_EVENTS=y
CONFIG_HAVE_PERF_EVENTS=y
CONFIG_PERF_EVENTS_INTEL_UNCORE=y

We evaluate REMIX on a wide range of Java and Scala workloads. We run the LMAX Disruptor Java inter-thread messaging library [46], a high-performance lock-free library with 7K LoC that has been extensively hand-optimized to avoid cache contention. Disruptor is used by the LMAX electronic foreign exchange market in London [29], and has also been adopted by many open source projects including the Apache Log4J [17] logging library. We use Disruptor to evaluate REMIX's accuracy and performance by removing the benchmark's padding that was added to avoid false sharing, to see if REMIX can find and fix the contention. We describe our Disruptor experiments more fully in Section 6.1.

We also evaluate REMIX on the Spring Reactor [37] framework, a large-scale library for developing asynchronous JVM applications that is part of the Spring Framework [42] and that builds upon Disruptor's RingBuffer implementation. Reactor spans 46K lines of code across 600 classes. Similarly to Disruptor, we used Reactor's own benchmark to evaluate two versions of the Reactor framework, the original padded version, and an unpadded version. We examined the benchmark's memory layout and determined it is very representative of what might be expected of large applications using Reactor.

We also run a collection of more conventional benchmarks. The DaCapo 2006 [2] and 9.12 Bach Java benchmarks, and the ScalaBench [40] Scala benchmarks, were run

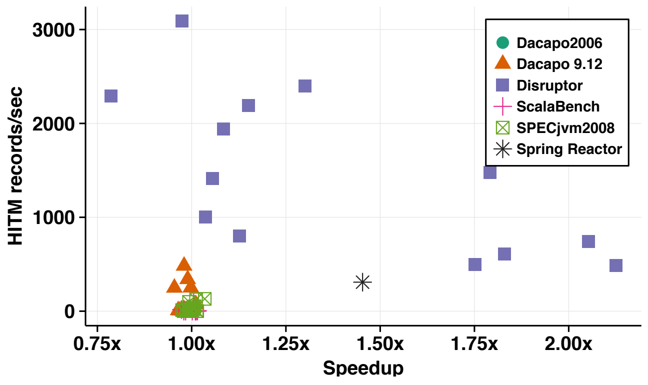

Figure 7: A benchmark's speedup with REMIX (x-axis) plotted against its rate of HITM records (y-axis).

with the large input, using 15 trials for each measurement where each trial involves 5 iterations to avoid cold-start effects. We exclude the DaCapo benchmarks (batik, eclipse) that did not run with JDK8, as well as those that did not run with the large data set (fop, luindex). The SPECjvm2008 Java benchmark suite [11] was run with the default setting of 2 minutes warmup and 4 minutes test. The Java versions of the NAS Parallel Benchmarks 3.0 [18] are run with the Class A input.

Error bars on all graphs specify ± one standard deviation. All benchmarks, except where specified otherwise, were run with a sampling rate of 100 samples per second. The threshold for detecting contention was set to 20 HITM records per second on a field, or 50 records per second on a class.

6. Evaluation

Figure 7 shows, for all 57 of our workloads, their speedup with REMIX (x-axis) against the average rate of HITM records per second they experience (y-axis). There is a strong positive correlation between speedup with REMIX and HITM rate. The vast majority of the workloads experience low HITM rates and negligible performance impact with REMIX. The unpadded Disruptor workloads (purple squares in Figure 7), however, exhibit high rates of HITM records and correspondingly large speedups with REMIX. We focus first on the Disruptor workloads (Section 6.1) which exhibit false sharing, then the NAS Parallel Benchmarks which reveal a significant true sharing bug in the JVM (Section 6.2), workloads without significant contention (Section 6.3) and some brief experiments with different Java and Scala actor libraries (Section 6.4). Finally, we present profiling information about REMIX's space and time usage in Section 6.5.

6.1 LMAX Disruptor and Spring Reactor

Disruptor is a high-performance inter-thread Java messaging library used by the LMAX electronic foreign exchange. Disruptor's code is extensively hand-optimized to maximize

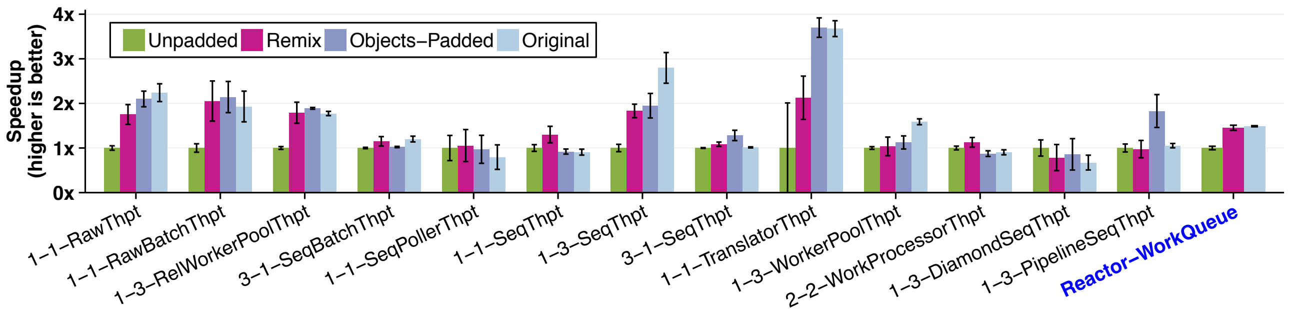

Figure 8: LMAX Disruptor and Spring Reactor speedup, normalized to *Unpadded*.

concurrent performance on modern CPUs, and corrects false sharing by implementing both object and array padding. We evaluate several versions of Disruptor. The first is an *Unpadded* version. Disruptor's padding is implemented using a handcrafted class hierarchy to mitigate JVM field reordering. For example, in the com.lmax.disruptor.Sequence class, a single volatile long is padded using a four-level class hierarchy, as depicted in Listing 1. The *Unpadded* code on which REMIX operates is much simpler and more natural, as shown in Listing 2.

Listing 1: Disruptor Sequence.java

```
class LhsPadding {
    protected long a, b, c, d, e, f, g;
}
class Value extends LhsPadding {
    protected volatile long value;
}
class RhsPadding extends Value {
    protected long i, j, k, l, m, n, o;
}
public class Sequence extends RhsPadding {
    // actual work
}
```

Listing 2: Unpadded Sequence.java

```
public class Sequence {
    protected volatile long value;
    // actual work
}
```

We also evaluate the *Original* version of Disruptor, which has all the original padding in place, and an *Objects-Padded* version that restores padding on objects but not arrays. As REMIX does not currently address false sharing on arrays, *Objects-Padded* represents the performance we expect from REMIX. The *Original* version shows the maximum performance potential of the code.

We ran 20 of Disruptor's performance tests, with 25 runs of each. The results, normalized to the *Unpadded* version, are summarized in Figure 8. We obtained significance p-values between 10^{-7} and 10^{-11} with the unpaired t-test with a two-sided alternative hypothesis. For brevity, seven tests with no statistical difference between the four versions are not shown. As expected, REMIX's performance is usually on par with the *Objects-Padded* version, offering substantial speedups over the *Unpadded* version without any additional code complexity or programmer involvement. REMIX is slower to a statistically significant degree on just two tests: 1-3-PipelineSeqThpt and 1-1-TranslatorThpt. For 1-3-PipelineSeqThpt, *Objects-Padded* performs better than all other versions, including *Original*, for unknown reasons. For 1-1-TranslatorThpt, REMIX's repair is activated on only about half the runs, which could be addressed with better threshold sensitivity. When REMIX's repair is activated, 1-1-TranslatorThpt performs as well as *Objects-Padded*.

REMIX sometimes even outperforms the *Original* Disruptor code, as in 1-1-SeqThp and 2-2-WorkProcessorThpt, where the *Original* version is unnecessarily padded and REMIX's adaptive repair approach provides better cache locality by avoiding unnecessary padding.

The rightmost bar of Figure 8 shows the results for the Spring Reactor framework. Because of its complexity (46K LoC), Reactor demonstrates REMIX's ability to detect and repair false sharing in a complex application. We run 30 iterations of Reactor's WorkQueue benchmark, which creates thousands of communicating producers and consumers to stress the framework. While Reactor has been hand-optimized to avoid false sharing, REMIX running on unoptimized code is able to match this level of performance completely automatically.

6.2 The NAS Parallel Benchmark Suite

Using REMIX with the NAS benchmarks revealed intense true sharing for the BT, LU and SP workloads, with most HITM records coming from one of the JVM's internal heaps. Further investigation revealed the true sharing originated from software performance counters maintained by the HotSpot JVM. HotSpot decides whether to compile methods by tracking their usage through two counters - an invocation counter and a backedge counter. These counters are shared across all threads, and thus suffer true-sharing when updated at high frequency. This is not normally a problem, as high update rates will push HotSpot to compile the function, thus solving the cache contention.

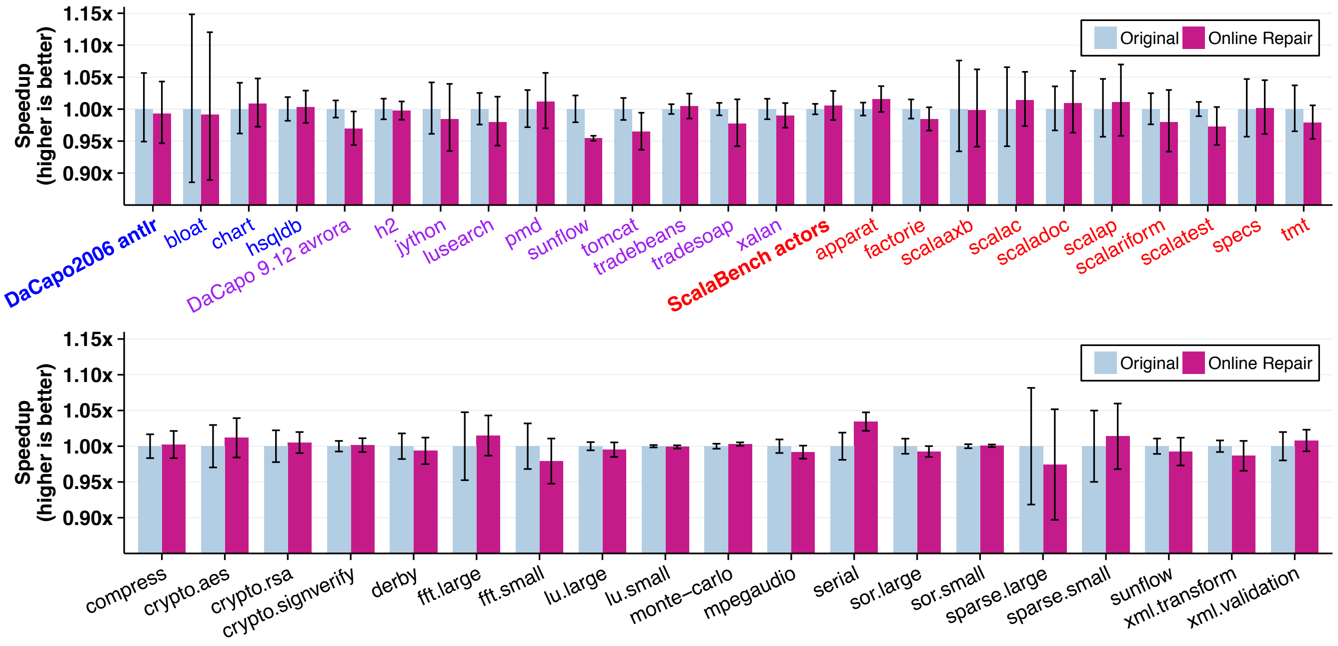

Figure 9: REMIX speedup for DaCapo and ScalaBench (top) and SPECjvm2008 (bottom).

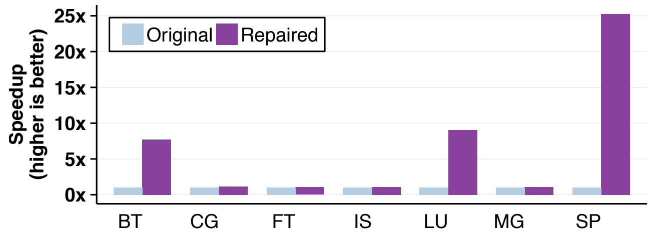

Figure 10: Speedup due to REMIX detecting true sharing on interpreted method profiling counters and triggering compilation for large methods.

However, the JVM has a limit on the size of functions it attempts to compile, and the functions in these three NAS workloads are beyond that limit so they never get compiled and the true sharing persists.

We augmented the REMIX version of Hotspot with a simple fix that detects contention on these HotSpot performance counters and enables compilation for the function they profile. With this feature in place, the runtime of the BT, SP and LU applications improves by 7-25x without impacting the other workloads, as shown in Figure 10.

6.3 Benchmarks without Significant Contention

Across the DaCapo, ScalaBench and SPECjvm2008 suites (Figure 9), most benchmarks do not show any statistically significant performance improvement or degradation with REMIX, using the Kolmogorov-Smirnov test with a 99% confidence interval. They serve chiefly to show the REMIX does not impair the performance of applications that have no or little cache contention.

In DaCapo 9.12's sunflow benchmark, REMIX detects a mix of true sharing and false sharing in the class org.sunflow.math.Matrix4. The contention is mostly true sharing, so REMIX's false sharing repairs end up being fruitless. sunflow runs for about 12 seconds on our experimental machine so REMIX's overhead results in a mild 5% slowdown. The SPECjvm2008 version of sunflow uses the same code as the DaCapo 9.12 version, but with a larger input that better amortizes REMIX's fixed costs so that REMIX has no appreciable slowdown. In DaCapo 9.12's h2 benchmark, moderate levels of true sharing were detected on several fields of the org.h2.engine.Database class and the org.h2.util.SmallLRUCache. The lusearch benchmark runs for less than a second on our machine, and so spends a significant portion of its time in the JVM interpreter where it experiences some performance counter true sharing similar to the NAS workloads (Section 6.2).

ScalaBench's actors benchmark exhibits a very high rate of true sharing on both the scala.actors.threadpool.LinkedBlockingQueue and java.util.concurrent.LinkedBlockingQueue classes.

While most of SPECJvm2008's benchmarks, as seen in Figure 9, have no cache contention and are not affected by REMIX, the *serial* benchmark is useful for evaluating REMIX. REMIX detects and repairs minor false sharing in the serial benchmark, which serializes and deserializes primitives and objects in a producer-consumer scenario. To this end, the benchmark allocates several OutputStream objects in a loop, one for each thread, which are laid out by the

JVM consecutively in memory. The classes themselves are small, containing two fields each, and thus are prone to false sharing. REMIX detects this false sharing and repairs it but, because the benchmark does not emphasize stream performance, the overall impact is just a 5% speedup.

6.4 The Savina Actor Benchmark Suite

The Savina benchmark suite [23] was designed to evaluate the performance of actor libraries across 30 benchmarks. Actor libraries are designed to provide a high-level abstraction for developing concurrent applications, hiding, by design, details such as cache coherency. REMIX was evaluated with eight of these libraries: Akka [51], FuncJava [19], Gpars [45], Habanero [22], Lift [50], Scalaz [21], and Scala actors. All are written in Scala except for FuncJava which is written in Java. REMIX uncovered multiple false and true sharing bugs in many of the libraries, as well as a few in the benchmarks themselves.

In the astar and cigsmok benchmarks, REMIX discovers true sharing on the seed of java.util.Random. The benchmarks use Math.random, a function with known thread contention [36]. Replacing Math.random with a per-thread instance of java.util.Random yields a 28x speedup.

In the radixsort benchmark using the Scalaz framework, REMIX detects false sharing on the head field of the scalaz.concurrent.Actor class, as well as on java.util.concurrent.atomic.AtomicBoolean. REMIX repairs the false sharing for a speedup of about 6%. REMIX also detects significant true sharing on the ctl field of the java.util.concurrent.ForkJoinPool system class. This system class is the source of HITM records for many of the actor frameworks, suggesting it as a target for future optimization.

6.5 Profiling REMIX

In this section we present REMIX profiling information, directly measuring its space and time costs. For brevity, we focus on the benchmarks where REMIX's repair mechanism is actually invoked: the Disruptor workloads, Spring Reactor, SPECjvm2008's serial benchmark and DaCapo 9.12's sunflow benchmark. In other benchmarks, REMIX does only a small amount of work with no appreciable overhead.

Table 2 shows the number of classes loaded, the time spent in processing the class hierarchy (Section 4.3), and the size of the heap. All measurements are taken at the time when REMIX's repair begins. The data show that even with over a thousand classes loaded and non-trivial heap sizes, REMIX's repair mechanism runs very quickly.

The top graph of Figure 11 quantifies the extra heap space used by REMIX for objects that are subject to runtime modification. Each bar shows the space occupied by the original, unmodified objects (light green) and the additional space used for padding in the modified objects (teal). In most cases, less than 2KB of heap space total is used, and never more than 17KB. This shows that, even though REMIX may modify an entire class hierarchy to implement padding, there

Suite Benchmark	Classes Loaded	Processing (ms)	Heap Size (MB)
DaCapo			
sunflow	1,879	1.47	853
Disruptor			
1-1-RawThpt	736	0.55	42
1-1-RawBatchThpt	736	1.51	42
1-1-SeqPollerThpt	793	0.75	43
1-1-SeqThpt	902	0.49	26
1-1-TranslatorThpt	824	1.26	60
1-3-DiamondSeqThpt	905	0.663	83.9
1-3-PipelineSeqThpt	905	0.579	75.9
1-3-RelWorkerPoolThpt	803	1.09	84
1-3-SeqThpt	905	0.696	75.5
1-3-WorkerPoolThpt	863	0.70	84
2-2-WorkProcessorThpt	887	0.476	76.5
3-1-SeqBatchThpt	813	1.12	76
3-1-SeqThpt	813	1.19	77
Spring Reactor			
WorkQueue	1,617	0.748	41
SpecJVM			
serial	1,498	1.21	1,603

Table 2: The size of the class hierarchy, time spent processing the class hierarchy, and the heap size, at the time when REMIX's repair begins.

are typically very few instances of the affected objects in the heap which helps make REMIX's repair operation fast. REMIX's adaptive approach can target padding to just the classes that need it on a given execution, avoiding the bloat that can result from statically-implemented manual padding.

In the middle graph of Figure 11, we show how much time REMIX spends in each of its three main processing stages. Collection time (red segments) is time spent receiving HITM records from the kernel and classifying the contention as true or false sharing (Section 4.1). Collection is done in parallel by each thread, and we present the average time spent per thread. Detection time (yellow segments) is time spent mapping from HITM records to JVM heap objects and classes (Section 4.2), and repair time (green segments) is time spent modifying objects with padding (Section 4.3). The total amount of time spent in the REMIX system is very small across our benchmarks, on the order of a few hundred milliseconds, and never more than 1 second. Repair time in particular is low because REMIX modifies only a small number of objects to mitigate false sharing (Figure 11, top). Moreover, REMIX's work is a one-time cost: once false sharing is repaired, there is little collection work to do and no detection or repair, so programs running with REMIX incur essentially no extra overhead.

To characterize how long it takes REMIX to repair false sharing in our workloads, we measure the elapsed time from JVM launch until the repair operation is complete. The bottom graph of Figure 11 presents the results, shown as a percentage of the application's total runtime. We can relate

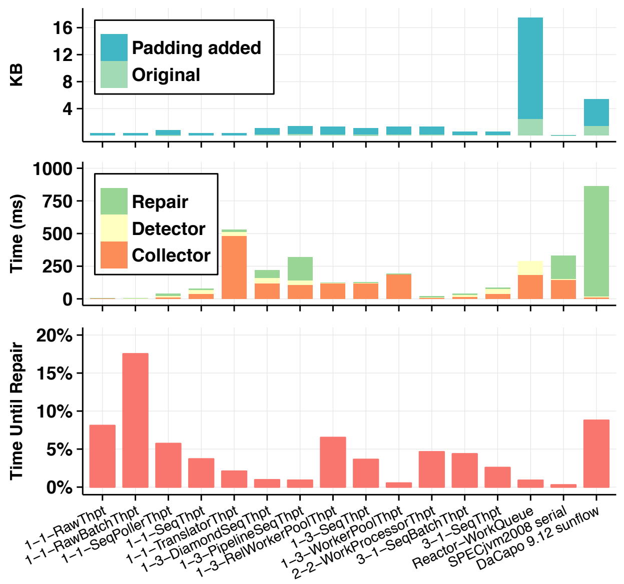

Figure 11: REMIX profiling results.
Top graph: The space occupied by objects that REMIX targets for runtime modification, broken down into the space used by the original objects and the padding added by REMIX.
Middle graph: Where REMIX spends its time, broken down between collecting HITM records (red), detecting contention and mapping it to objects and classes (yellow), and repairing the contention by modifying objects (green).
Bottom graph: The proportion of an execution that elapses before REMIX's repair mechanism is complete.

the time until repair to the potential speedups obtainable by REMIX via Amdahl's Law: if REMIX completes its repair operations quickly, a large fraction of the execution will be accelerated by the removal of false sharing. The bottom graph of Figure 11 shows that this is indeed the case, with most benchmarks repaired within the first 5% of their execution. Once REMIX's repairs are in place, an application can run at native speed with similar efficiency to a hand-implemented fix. As Section 4.1 observes, REMIX's detection mechanism piggybacks on existing STW events for simplicity but could initiate its own events once sufficient HITM records are collected, reducing REMIX's time until repair even further.

7. Related Work

Online false sharing detection and repair schemes for unmanaged code, like Sheriff [27], Plastic [34] and LASER [31], are the most closely-related work to REMIX. Both Sheriff and Plastic rely on invasive changes to the runtime environment. Sheriff executes threads as isolated processes, tracking memory updates and sharing them when synchronization occurs. Sheriff's execution model can result in non-termination for some programs with ad-hoc synchronization. Furthermore, Sheriff's diff-based method for detecting memory updates can result in a phenomenon similar to *word tearing* that produces out-of-thin-air values in the presence of data races, which makes Sheriff incapable of providing the safety guarantees needed for the Java memory consistency model ([30], Section 4.6). Plastic provides a sub-page-granularity virtual memory remapping facility via dynamic binary instrumentation and custom OS or hypervisor support. Thus, neither Sheriff nor Plastic are well-suited for integration with a high-level language runtime. LASER uses the same performance counters leveraged by REMIX, and characterizes their precision on a 4-core Haswell platform. While LASER adopts a more compatible approach to online false sharing repair, its performance is far from the efficiency of manual repairs, unlike REMIX.

Several other projects provide **false sharing detection**, and report contention to the programmer for manual repair. [39] relies on full-system simulation. Pluto, [26] and the Dynamic Cache Contention Detection scheme [53] use dynamic binary instrumentation to instrument all shared memory accesses. The Predator scheme [28] relies on compiler instrumentation for the same purpose. Such heavyweight instrumentation provides clear visibility of the program's memory access behavior. Predator can even predict the presence of false sharing on alternate machines with larger or smaller cache lines. The runtime cost of such heavy instrumentation is prohibitive, however, and typically slows execution by an order of magnitude. Notably distinct from these software-based approaches is Intel's VTune profiler [8], which can monitor PEBS HITM events just as REMIX does. However, VTune is a performance profiler and cannot automatically repair the contention it detects.

Java 8 provides a new **false sharing annotation**, `@Contended` [35], which can be added to fields subject to false sharing. `@Contended` tells the JVM to pad annotated fields so that they always reside in their own cache line. `@Contended` annotations must be manually added by the programmer, and we are not aware of any tools that provide guidance on where `@Contended` annotations should be placed. Using `@Contended` requires access to source code, which could prevent programmers from fixing false sharing in library code. `@Contended` can be applied only to defined fields, with no mechanism provided to annotate synthetically-injected fields such as the $parentthis$ reference for inner classes. Using REMIX, in contrast, requires no access to source code, can operate on any field in any class loaded into the JVM, and requires no programmer effort.

There is a rich literature on extending garbage collection to **optimize heap layout** in managed languages, typically focusing on Java. For example, work on object inlining and object fusion [14–16, 24, 47–49] emulates the layout of nested structs in Java, placing objects consecutively on the heap to optimize address calculation and locality. Other systems optimize the heap for cache locality [5, 20, 41] by packing frequently-accessed objects together. These layout optimizations are complementary to REMIX, as REMIX can discover when these prior optimizations backfire by creating cache contention instead of locality.

Other have used **performance counters in managed runtimes** to insert memory prefetch instructions [1], understand performance behavior [44], and to inform decisions about when to optimize methods [4]. None of these systems focus on cache contention, and thus they are complementary to REMIX.

Finally, many academic and commercial tools exist to **profile lock contention** in both managed and unmanaged code. For Java, tools like VisualVM [10] and YourKit [52] are popular performance profiling tools that can also identify lock contention. Such tools work by instrumenting synchronization operations, and thus they must be aware of the synchronization library used by a program. In contrast, REMIX's performance counter approach sees the fundamental contention that exists at the hardware level, and is thus completely synchronization-library agnostic and can detect contention in both application code and within the JVM itself.

8. Conclusion

We have presented the REMIX system, the first system to detect all forms of cache contention, and automatically repair false sharing, within a managed language runtime. REMIX is implemented in the HotSpot JVM, and leverages OS and architecture support for advanced hardware performance counters to track the HITM events that are the underlying cause of cache contention at the coherence protocol level. These HITM performance counters provide an integrated view of all of the cache contention occurring in the system, allowing REMIX to monitor true and false sharing both within JVM applications and also within the language runtime itself. Using REMIX, we discover true and false sharing bugs in several Java and Scala workloads and also a significant performance bug in HotSpot itself. We show that REMIX is able to repair false sharing using the same padding techniques that a programmer would, allowing REMIX to achieve the same efficiency as manual fixes but without any programmer effort. REMIX has negligible performance impact on applications that do not trigger cache contention, thus making REMIX suitable for always-on detection and repair of contention in production environments.

Acknowledgments

We thank Liang Luo for early discussions about this work. This work was supported by NSF grant SHF-1525296. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. Prefetch Injection Based on Hardware Monitoring and Object Metadata. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 267–276, 2004.

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, 2006.

[3] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[4] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. Using HPM-sampling to Drive Dynamic Compilation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 553–568, 2007.

[5] Trishul M. Chilimbi and James R. Larus. Using Generational Garbage Collection to Implement Cache-conscious Data Placement. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 37–48, 1998.

[6] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, 2012.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, 2013.

[8] Intel Corporation. Avoiding and Identifying False Sharing Among Threads. https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads, 2011.

[9] Intel Corporation. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, 6 2015.

[10] Oracle Corporation. VisualVM: All-in-One Java Troubleshooting Tool. https://visualvm.java.net/, 2015.

[11] Standard Performance Evaluation Corporation. SPECjvm2008. http://www.spec.org/jvm2008/, 2008.

[12] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously Measuring Critical Section Pressure with the Free-lunch Profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 291–307, 2014.

[13] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, 2004.

[14] Julian Dolby. Automatic Inline Allocation of Objects. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 7–17, 1997.

[15] Julian Dolby and Andrew Chien. An Automatic Object Inlining Optimization and Its Evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 345–357, 2000.

[16] Julian Dolby and Andrew A. Chien. An Evaluation of Automatic Object Inline Allocation Techniques. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 1–20, 1998.

[17] Apache Software Foundation. Apache Log4j 2 website. http://logging.apache.org/log4j/2.x/, 2015.

[18] Michael A. Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan. Implementation of the NAS Parallel Benchmarks in Java. Technical Report NAS-02-009, NASA Advanced Supercomputing Division, 2002.

[19] functionaljava.org. functionaljava: A Library for Functional Programming in Java. functionaljava.org, 2010.

[20] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The Garbage Collection Advantage: Improving Program Locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 69–80, 2004.

[21] L. Hupel and typelevel.org. scalaz: Functional programming for Scala. http://typelevel.org/projects/scalaz/, 2010.

[22] Shams Imam and Vivek Sarkar. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 75–86, 2014.

[23] Shams M. Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors, Agents & Decentralized Control*, AGERE! '14, pages 67–80, 2014.

[24] Ondrej Lhoták and Laurie Hendren. Run-time Evaluation of Opportunities for Object Inlining in Java. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, JGI '02, pages 175–184, 2002.

[25] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*, chapter 4.4 The class File Format. Oracle Corporation, 2015.

[26] C.-L. Liu. False Sharing Analysis for Multithreaded Programs. Master's thesis, National Chung Cheng University, 7 2009.

[27] Tongping Liu and Emery D. Berger. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 3–18, 2011.

[28] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. PREDATOR: Predictive False Sharing Detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 3–14, 2014.

[29] LMAX. LMAX Disruptor — Open Source — LMAX Exchange. https://www.lmax.com/disruptor, 2015.

[30] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. Efficient Deterministic Multithreading Without Global Barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 287–300, 2014.

[31] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris Newburn, and Joseph Devietti. LASER: Light, Accurate Sharing dEtection and Repair. In *Proceedings of the 2016 IEEE 22nd International Symposium on High Performance Computer Architecture*, HPCA '16, 2016.

[32] Linux Programmer's Manual. *perf_event_open(2) Linux Programmer's Manual*, 2015.

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

[34] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. Whose Cache Line is It Anyway?: Operating System Support for Live Detection and Repair of False Sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 141–154, 2013.

[35] Scott Oaks. *Java Performance: The Definitive Guide*. O'Reilly Media, 3rd edition, April 2014. Page 266.

[36] Oracle. Java 7 SE API documentation: java.util.Random. http://docs.oracle.com/javase/7/docs/api/java/util/Random.html, 2014.

[37] Reactor Project. Spring Reactor. http://projectreactor.io/, 2015.

[38] Mikael Ronstrom. MySQL team increases scalability by >50% for Sysbench OLTP RO in MySQL 5.6 labs release april 2012. http://mikaelronstrom.blogspot.com/2012/04/mysql-team-increases-scalability-by-50.html, April 2012.

[39] Martin Schindewolf. Analysis of Cache Misses Using SIMICS. Master's thesis, Institute for Computing Systems Architecture, University of Edinburgh, 2007.

[40] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '11, pages 657–676, 2011.

[41] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 13–25, 2002.

[42] Spring.io. Spring.io website. https://spring.io/, 2015.

[43] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 1–12, 2009.

[44] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using Hardware Performance Monitors to Understand the Behavior of Java Applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 5–5, 2004.

[45] The GPars team. The GPars Project - Reference Documentation. http://www.gpars.org/guide/, 2014.

[46] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. http://disruptor.googlecode.com/files/Disruptor-1.0.pdf, 5 2011.

[47] Christian Wimmer and Hanspeter Mössenböck. Automatic Feedback-directed Object Inlining in the Java Hotspot Virtual Machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 12–21, 2007.

[48] Christian Wimmer and Hanspeter Mössenböck. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 14–23, 2008.

[49] Christian Wimmer and Hanspeter Mössenbösck. Automatic Feedback-directed Object Fusing. *ACM Trans. Archit. Code Optim.*, 7(2):7:1–7:35, October 2010.

[50] LLC. WorldWide Conferencing. Lift Framework - LiftActor. http://liftweb.net/, 2014.

[51] Derek Wyatt. Akka Concurrency - Building reliable software in a multi- core world. Technical report, Artima Incorporation, 2013.

[52] YourKit. YourKit Java Profiler - .NET Profiler. https://www.yourkit.com/, 2015.

[53] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic Cache Contention Detection in Multi-threaded Applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 27–38, 2011.