Dynamic Detection and Mitigation of False Sharing with Intel Pin and LLVM

Brandon Kayes
University of Michigan
Ann Arbor, Michigan
bkayes@umich.edu

Daniel Hoekwater
University of Michigan
Ann Arbor, Michigan
dhoek@umich.edu

Thomas Smith
University of Michigan
Ann Arbor, Michigan
thsm@umich.edu

Tony Bai
University of Michigan
Ann Arbor, Michigan
bait@umich.edu

Abstract—**False sharing, a phenomenon in multi-threaded programs where two or more threads access different pieces of data that reside on the same cache line, can severely degrade the performance of concurrent programs.**

We introduce a novel dynamic false sharing detection and mitigation tool and detail its design and implementation. Our tool uses profile data generated from dynamic binary instrumentation tools combined with LLVM passes to transform code at compile-time; with these tools, we demonstrate and evaluate a method for generating an optimized binary which mitigates the performance penalties of false sharing.

Index Terms—**compilers, false sharing, Intel Pin, LLVM, parallelism, concurrency**

I. Introduction

False sharing is a phenomenon in multi-threaded code where memory accesses to different pieces of data by different CPU cores end up on the same cache line, leading to poor performance. A CPU that writes to a falsely shared cache line must invalidate all other CPUs' caches, and other CPUs will incur cache misses when they try to read data from anywhere in the falsely shared cache line. False sharing is an undesirable and theoretically unnecessary side effect in modern computing workloads which commonly make use of many CPU cores and are highly dependent on the speed of modern memory hierarchies.

Because it is not well-documented in language standards or in most education materials on concurrent programming, false sharing can thus be a silent killer of performance in multi-threaded programs. In this paper, we outline a profile-driven method for detecting and mitigating false sharing to reduce the unnecessary performance overhead associated with it. We introduce a novel dynamic false sharing detection and mitigation tool, which dynamically detects instances of false sharing and uses a profile-guided LLVM pass to prevent their occurrence.

In section II, we provide some necessary background on false sharing and manual methods to prevent it and diminish its effects. We then discuss the design and implementation of our dynamic detection and mitigation system in section III and section IV before evaluating its effectiveness on a variety of programs in section V. We discuss related and future work on false sharing in section VI and section VII. Finally, we summarize our findings in section VIII.

II. Background

In order to understand the motivation and implementation of our system, we first must understand caches and shared memory. Modern memory hierarchies make liberal use of caches to improve performance, and as a performance optimization, the smallest level of granularity that caches work with is an entire cache line, instead of caching individual bytes. This means that from the perspective of the cache, accessing a given byte in the cache line is equivalent to accessing any other byte in the cache line.

Additionally, there is a concept of private and shared caches in CPU design. A shared cache means that multiple cores on a machine share the cache, and therefore if multiple cores are accessing the same cache line, nothing special needs to happen since the data is already in the cache, and is shared amongst the relevant cores. This is in contrast to a private cache where each core has its own private version of the cache in order to improve memory access latency. With private caches, if multiple cores are accessing a given cache line (and at least one of the accesses is a write), then in order to maintain correctness and coherency, only the core which is writing to the cache line is allowed to have access to that line, and all the other caches have the line invalidated. When cores are accessing the exact same data (the same byte offset within the line), this is ideal because it is necessary for correctness. However, there is a performance penalty due to invalidation which is incurred even if the cores are accessing *different* data within the same cache line. This is an example of false sharing and is not ideal because this invalidation does not need to happen in order to guarantee correctness, but it is instead a side effect of the cache implementation.

Performance losses due to false sharing can be significant, and it is sometimes described as the silent killer of performance. [1] Manual efforts can be taken to reduce the possibility of false sharing, such as manually separating data and designing data structures in a way that moves previously falsely shared data out of the same cache line. This manual process is very time-intensive, isn't always portable, and requires careful reasoning about hardware details. This is something that error-prone humans aren't always great at, as demonstrated by previous work which shows that automatic detection and optimization of false sharing can beat traditional "hand-optimized" code in terms of performance. [1]

III. Design

To mitigate false sharing, we have designed a system to detect falsely-shared memory accesses and to transform the program into an optimized binary that avoids false sharing.

A. Detecting False Sharing

The first step of our pipeline is detection. Given a multi-threaded program, we want to (1) detect the memory accesses which are falsely shared and (2) match up these memory accesses with the names of global variables.

To aid with (1), we use and build upon Intel Pin [2], which is a publicly available dynamic binary instrumentation framework that enables the creation of dynamic program analysis tools. Specifically, we use the `pinatrace` and `dcache` tools within Intel Pin. `pinatrace` is used to output the memory accesses of a program, and `dcache` is an L1 cache simulator. Out of the box, both of these tools only work with single-threaded programs. However, for false sharing, we must analyze multi-threaded behavior of programs; thus, we modified both tools to be thread-safe and support multi-threaded workloads. Both tools were used for false sharing detection, and we will now explain how each was used.

1) `pinatrace`: The version of `pinatrace` that ships with Intel Pin instruments every load and store instruction, keeping track of the program counter, whether the access is a read or write, the memory address, size of the access, and the value of the memory at the address. We modified `pinatrace` to not only print out the memory accesses of a program, but also an identifier for which thread accessed the memory. We also modified the program such that it was thread-safe, enforcing linearizability of the memory accesses.

*2) **detect**:* This memory trace was then passed to a C++ utility, **detect**, which parsed the memory accesses and identified potential sources of false sharing based on criteria which are outlined in section IV. As the final step of this part of the detection phase, **detect** would output any interferences it found into a list of potential interferences. The list of interferences consists of a pair of addresses, along with a priority score derived from roughly how frequently those addresses conflicted with each other.

3) `mdcache`: Independently from `pinatrace`/**detect**, `dcache` was modified to support multi-core cache simulation, and was renamed to `mdcache`. We run `mdcache` on a program, and output cache hit-rate statistics, as well as a list of observed interferences, which also consisted of a pair of addresses, along with a priority score derived from roughly how frequently those addresses conflicted with each other in the cache simulation.

The `pinatrace`/**detect** tools and the `mdcache` tool were used for the same purpose of false sharing detection. However, both run independently, and the output from both are fed into the next stage of our system. The reason for having two independent systems for performing the same task of detection is because neither system by itself is as useful as both systems together. `pinatrace`/**detect** is used to output a "theoretical" list of interferences – it is not as precise, but is more likely to cover *all possible* interferences. `mdcache` is not as all-encompassing, but the interferences it detects are likely more serious. The outputs from both are combined and reconciled via the priority score – because of the nature of each system, `pinatrace`/**detect** outputs are weighted with lower priority, and `mdcache` outputs are weighted with higher priority. Together, the two systems come together to create a comprehensive detection system.

B. Fixing False Sharing

Once we have lists of interferences from `pinatrace`/**detect** and `mdcache`, the next step is to use an LLVM pass to fix the detected instances of false sharing. However, in order to do so, we must first match up the memory accesses in the interference list with the names of variables in the program, so that `LLVM Fix` knows which variables to target.

1) LLVM Globals Pass and MapAddr: To correspond interferences with variable names, we first use an `LLVM Globals` pass to output the addresses of global variables. With this information, another C++ utility called **MapAddr** is used to match the addresses of global variables with the interferences found from `pinatrace`/**detect** and `mdcache`. **MapAddr** outputs a list of the names of falsely shared variables, and passes this to an `LLVM Fix` pass for code transformation.

2) LLVM Fix Pass: Lastly, the output from **MapAddr** is provided as input to an `LLVM Fix` pass, which identifies variables which exhibit false sharing, and adds padding between them so that they are relocated onto different cache lines. If the variables are fields of a struct, we do a special transformation where a new struct is created with padding between the falsely shared members, and convert all accesses to the original struct in the program to access the new, modified struct. More details can be found in section IV.

After the `LLVM Fix` pass, an optimized binary is produced. For evaluation, we use timing information within the binary, and we also run `mdcache` one more time on the optimized binary to compare cache hit-rate statistics.

Figure 1 shows a diagram summarizing the design of our system.

IV. Implementation

We will now go into more detail about each component of our system.

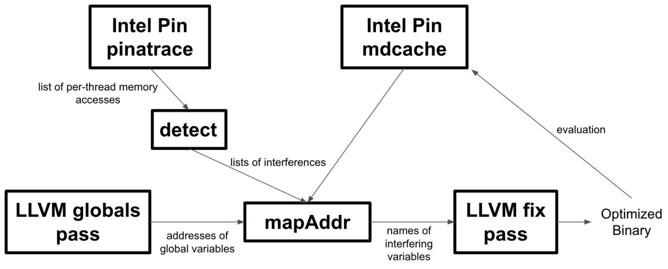

Fig. 1. **The components of our false-sharing detection and mitigation system.**

A. `pinatrace`

The first step in the detection process involves obtaining a profile of the memory accesses made by the target program. An existing tool called `pinatrace` does exactly this for single-threaded applications, so we extend this tool in a way that allows us to create a list of the per-thread memory accesses in the target. To enforce linearizability, we protected all trace logic and accesses to the trace output with a mutex, a synchronization primitive for enforcing mutual exclusion.

B. *detect*

detect goes through the accesses and thread IDs outputted by `pinatrace`, and identifies potential instances of false sharing by examining pairs of accesses. For any given pair of accesses, **detect** marks it as an interference if the following criteria are met:

1) The accesses are from different threads
2) The accesses are the same cache line
3) The accesses access non-overlapping addresses
4) At least one of the accesses is a write

detect assigns a priority score to each interference according to how many times the interference was encountered in the `pinatrace` output. It outputs a list of all the interferences, with each interference consisting of a pair of addresses along with a priority score.

C. `mdcache`

Independently from the `pinatrace`/**detect** pipeline, we implemented a multi-core data cache simulator to evaluate the extent to which false sharing affects the program's execution and to identify data interferences that most contribute to false sharing. Inspired by the cache simulator CMP$im [3], we extended the existing Pin tool `dcache`, which tracks hitrate statistics of a single L1 data cache, to achieve three main goals: thread safety, evaluation of how common cache invalidations are, and identification of which potential interferences generate these cache invalidations in practice.

When a line in one cache is written to, it must invalidate that line in other caches so that they do not read stale data. To achieve this, we added a third type of access in addition to read and write accesses, ***invalidate***. When a cache processes a write, it makes an invalidate access to the L1 caches of the other cores. An invalidate access finds the cache line, if it is resident, and replaces it with a ***tombstone***. A tombstone, other than being different than a normal cache line, contains the destination address of the write access which invalidated it. If the cache finds a tombstone when searching for a cache line in a set, it examines the address of the request and, if it is not the same as the address of the invalidation (which would indicate true sharing), we log it as an interference. At the end of program execution, we ascribe a priority to each interference based on how many times it occurs.

Like in `pinatrace`, we made `mdcache` thread-safe by protecting accesses to each cache and to the global list of caches with a mutex. However, while this approach alone makes the program thread-safe for programs in which there is no invalidation, it introduces the potential for deadlock when handling invalidations. Consider caches for two threads, A and B, which both get a memory access to address `0xCAFE`. Both threads A and B process the write access to `0xCAFE` in their cache and go to invalidate the cache line upon which `0xCAFE` resides in the caches of the other cores. Thread A requests the mutex for thread B's cache, and thread B requests the mutex for thread A's cache, resulting in deadlock. To fix this problem, we define a global mutex governing writes to memory, such that only one thread can process a write at a time. This restriction enforces linearizability of writes without burdening the program with the performance overhead of contention for a global mutex that would be incurred by requiring mutal exclusion of every read and write access in the program. Critically, our cache simulator permits concurrent read accesses to the same data, only enforcing mutual exclusion for write accesses. Our tombstone implementation achieves the goals of evaluation of the incidence of false sharing as well as a generation of observed interferences with their priorities, and our concurrent implementation offers efficient thread-safety.

D. `LLVM Globals` *and* ***MapAddr***

As mentioned in section III, the `LLVM Globals` first goes through and outputs a list of global variable names along with their size and memory address. This information is used by **MapAddr** in conjunction with the lists of interferences outputted by the detection stage to generate the name and location in memory of all detected falsely shared global variables. The output from **MapAddr** can then be used as input to the `LLVM Fix` pass.

E. `LLVM Fix`

Finally, we run our `LLVM Fix` pass, which uses the output of **MapAddr** to determine which global variables to pad.

The fix pass mitigates inter-variable false sharing by aligning each conflicting variable to its own cache line. LLVM's `GlobalVariable::setAlignment` method gives us a convenient way to do this without rewriting instructions.

The fix pass also mitigates some cases of intra-variable false sharing for struct variables. If different members of the same struct exhibit false sharing, the fix pass attempts to add padding between those members. However, since the padding may invalidate certain uses of the struct, the pass does not add padding to structs that are used in arbitrary arithmetic, passed to functions, linked externally, or copied to other variables. The pass does fix any LLVM `getelementptr` instructions to use the new members offsets of the padded struct.

F. Link to Implementation

Our implementation is publicly available on GitHub and can be found at https://github.com/thomasebsmith/dynamic-false-sharing-mitigator.

V. Evaluation

To evaluate the effectiveness of our false sharing mitigation process, we used Pin to measure cache hit rate and the C function `clock_gettime` to measure the real time taken for a variety of benchmarks. We designed the benchmarks **basicGlobals** and **sharedStruct** to exhibit a large amount of false sharing. We designed the benchmarks **basicLocks** and **locks** to reflect real-world multi-threaded programs.

We compiled each benchmark once using Clang with the `-O3` compiler flag, and once using Clang with both the `-O3` compiler flag and the `-false-sharing-fix` compiler flag (the flag that runs our fix pass).

As shown in Table I, we were able to achieve cache hit rate increases of up to 34.51% in contrived benchmarks and up to 5.15% in lock-related benchmarks.

Benchmark	**Original Hit Rate**	**Hit Rate With Fix**
basicGlobals	71.38%	99.90%
sharedStruct	65.39%	99.90%
basicLocks	93.25%	99.40%
locks	97.51%	98.04%

TABLE I
L1 Data Cache Load/Store Hit Rates by Benchmark

As shown in Table II, we similarly saw up to 1.32 times faster execution for the **basicGlobals** and **sharedStruct** benchmarks, and up to 1.27 times faster execution for the **basicLocks** and **locks** benchmarks.

Benchmark	**Original Time (ms)**	**Time With Fix (ms)**
basicGlobals	3.88	2.96
sharedStruct	5.30	3.99
basicLocks	213.2	167.8
locks	4.43	3.89

TABLE II
Exeuction Time by Benchmark

VI. Related Work

Our paper builds on previous related work from Jeremiassen and Eggers [1] in which they use static methods to detect false sharing. They implement 3 transformations to remove false sharing automatically: indirection, group and transpose, and padding. Our work builds upon the work of Jermiassen and Eggers by providing a method for profiling a program to obtain empirical data to detect false sharing and assign a priority to different instances, enabling a data-driven decision of which instances to suppress via padding.

VII. Future Work

In this section we discuss future work which could improve our implementation and make this optimization more widely applicable.

A. Performance

We have tuned our entire framework to have acceptable performance on examples which have on the order of millions of memory accesses. This is great for showing non-trivial benchmark examples, but still falls over when trying to scale to real production workloads. The largest bottleneck in our system is currently associated with obtaining and manipulating the memory trace for our program. When obtaining the memory trace, there is a significant performance overhead associated with running the instrumented program, because every memory access needs to be logged in a file. In future work, we would like to investigate using hardware support to reduce the run-time cost of memory trace collection. Additionally, the trace collection and processing fails to scale because it currently logs every single memory access. As a second future work, we would like to investigate using deduplication or priority-based heuristics when collecting the memory traces so that we could gain an empirical picture of the false-sharing that exists within a program without necessarily sampling every single memory access.

B. Additional Transformations

To reduce complexity, our system only reduces false sharing by applying a padding transformation on global variables. Previous works have demonstrated other transformations which reduce false sharing by introducing indirection or rearranging elements. [1] Future work could extend our `LLVM Fix` pass to incorporate these additional transformations, which would be beneficial since they don't increase the binary size and memory usage as much as the padding transformation does.

Additionally, it would be interesting to investigate how to apply these transformations to fstack and heap addresses instead of just global variables. We think that stack variables generally would not be too difficult to transform because their location is statically known within their stack frames, and pointer aliasing is a relatively rare problem. However, the heap is much more difficult, because without making modifications to the underlying allocator, it would be nearly impossible to determine whether a given memory access is going to access a transformed section of memory. It would be interesting to investigate if modifications to the memory allocator would enable more heap transformations to be performed.

Finally, future work could extend our `LLVM Fix` pass to support padding within a broader range of struct types. Currently, the pass is limited to structs that are only accessed using LLVM `getelementptr` instructions, and padding is limited to top-level struct members. Future work could extend this to support passing padded structs to functions and padding arbitrarily nested struct members.

VIII. Conclusion

This paper proposes a profile-driven approach to identify and mitigate false sharing in concurrent programs. It introduces a tool for doing so using dynamic binary and byte code instrumentation, in coordination with a false sharing detection heuristic and a final LLVM optimization pass.

Our approach is flexible, easy to use, and presents the opportunity for substantial program speedup. We evaluated our tool on several realistic benchmarks, achieving a significant increase in cache hit rate and in runtime performance for every benchmark. Our experience in developing this tool shows that dynamic analysis of concurrent programs has considerable potential for performance improvements over the lifetime of a program for a pretty reasonable compile-time cost.

IX. Acknowledgements

We would like to acknowledge Professor Scott Mahlke for helpful discussions and guidance during the course of this project.

References

[1] T. Jeremiassen, S. Eggers, “Reducing false sharing on shared memory multiprocessors through compile time data transformations,”, PPOP, 1995

[2] “Pin 3.21,” Pin - A Binary Instrumentation Tool. Intel, 28-Oct-2021. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[3] A. Jaleel, R.S. Cohn, C.K. Luk, B. Jacob, “CMP$im: A Binary Instrumentation Approach to Modeling Memory Behavior of Workloads on CMPs“