Fence Placement for Legacy Data-Race-Free Programs via Synchronization Read Detection

Andrew J. McPherson
University of Edinburgh
ajmcpherson@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Susmit Sarkar
University of St. Andrews
ss265@st-andrews.ac.uk

Marcelo Cintra
Intel
marcelo.cintra@intel.com

Abstract

Fence placement is required to ensure legacy parallel programs operate correctly on relaxed architectures. The challenge is to place as few fences as possible without comprising correctness. By identifying necessary conditions for a read to be an acquire we improve upon the state of the art for legacy DRF programs by up to 2.64x.

Categories and Subject Descriptors B.3 [*Hardware*]: Memory Systems; D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Fence Placement, Relaxed Memory Models

1. Introduction

Modern parallel architectures employ relaxed memory models for performance reasons. Legacy programs written assuming Sequentially Consistent (SC) operation require fences to be placed to ensure correctness on such architectures.

The starting point of understanding the required placement of fences is the *Delay-set analysis* of Shasha and Snir [8]. They observed that to ensure SC, ordering all pairs of accesses is unnecessary. Only conflicting pairs of accesses (the delay sets) that can lead to SC violations need to be ordered – where conflicting accesses are two accesses to the same address, at least one of which is a write. The memory orderings produced are then subject to *fence minimization* [5], which seeks to minimize the number of fences used to enforce the orderings.

Scalability issues and a reliance on alias analysis, mean that in practice conservative approximations are used. Notably the Pensieve project [3, 9], which uses thread escape analysis to determine potential acquires and releases.

We take a fresh look at fence placement. Our point of departure is that we do not seek to enforce SC for the general case; instead we seek to ensure data race freedom. More specifically, we insert sufficient fences to ensure that those memory accesses that are race free[1] in the SC world continue to be race free in the relaxed world. To put it succinctly, we guarantee SC behavior for race free accesses.

Our approach is based on the realization that SC (which strongly orders all accesses) is not an end in itself to programmers; rather it is enough for programmers to have SC semantics only for synchronization accesses (that are used to guard other data accesses from racing). Therefore, it suffices if we identify such operations and provide SC semantics for only those operations.

Although we do not promise SC in general, it is important to note that our approach guarantees SC for well synchronized programs i.e., programs that do not contain data races. More formally, we preserve only SC-allowed behaviour for the class of programs whose behavior is characterized by values returned by only those reads that are race-free under SC.

Our approach is similar in spirit to DRF (data-race-free) programming models, which form the basis of recent concurrent programming language models, such as the C11 concurrency model [1, 2] and the Java Memory Model specification [6]. This is a programming model which gives semantics to only DRF programs: programs in which synchronization operations are correctly labelled and the program is well synchronized using those operations. In return for this discipline the system (hardware + compiler) guarantees SC. However, legacy programs lack the distinction between data and synchronization. Our approach automatically discovers synchronization operations for such legacy program.

2. Our Approach

We aim to conservatively identify synchronization operations. If we can be relatively precise, we can prune unnecessary orderings found by more traditional approaches. Existing fence minimization techniques can then be applied.

We have identified two signatures and proven [7], that at least one must be fulfilled for a read to be an acquire:

- Control acquire: a read feeds its value to a predicate tested for in a branch in its forward slice.
- Address acquire: a read provides the address value for the subsequent data access that the read (acquire) protects.

[1] A memory access is said to be race-free if in all legal SC executions, it is ordered with its conflicting accesses in each execution, following Gharachorloo [4].

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA.

ACM 978-1-4503-3205-7/15/02.
http://dx.doi.org/10.1145/

While it is possible for an acquire to meet the address signature and not the control signature, we find that in practice those that meet the address signature also meet the control signature. To reinforce this point we performed an empirical study of 10 common synchronization primitives, the results of which are presented as Table 1. These primitives represent common patterns used in synchronization, indeed some underpin programs we examine later in Section 3.

	Acquires		
	Addr	Ctrl	Pure Addr
Chase Lev WSQ	✓	✓	✘
Cilk-5 WSQ	✘	✓	✘
CLH Lock	✓	✓	✘
Dekker	✘	✓	✘
Dijkstra	✘	✓	✘
Lamport	✘	✓	✘
MCS Lock	✓	✓	✘
Michael Scott LFQ	✓	✓	✘
Peterson	✘	✓	✘
Szymanski	✘	✓	✘

Table 1. Breakdown of the types of acquires found in common synchronization kernels.

Having identified these signatures, we are able to build on the Pensieve model. After identifying a conservative subset of reads as acquires using thread escape analysis, as in Pensieve, we use our signatures to prune the set, without compromising its conservative nature. These sets, and analysis of the Control Flow Graph, identifies a reduced set of orderings. A fence minimization algorithm is then used to place fences and compiler directives. For example on x86 only orderings of the form $w \rightarrow r_{acq}$ require a full fence, with compiler directives (which have no presence in the final binary) used to prevent incorrect reordering by the compiler.

3. Results

We implemented our analysis and the Pensieve model as an intraprocedural analysis pass in LLVM 3.4.1. Using a set of lock-free programs and the SPLASH-2 [10] benchmarks, we compare both the **Fast** (control acquires only) and **Safe** (control and address acquires) variants of our approach with an implementation of Pensieve using locally-optimized fence minimization (as in Fang et al. [3]). To establish a baseline we also compare against a (minimal) manual fence placement. All programs were compiled using $O2$ optimizations.

The percentage of shared reads that are marked acquires by each variant of our approach is presented as Figure 1. As we can see, the Fast analysis greatly reduces the number of potential acquires. In the best case (*Water-NSquared*), only 7% are potentially acquires. On average[2] we see 18% of the reads marked as acquires.

To examine the impact of reducing the number of fences, we executed the programs having applied Pensieve, both variants of our approach and normalize these against manual fence placement. Each of the experiments was repeated 100 times and averages taken. The results are presented as Figure 2.

On average we see that Pensieve is 1.94x slower than the baseline, with our Fast approach being only 1.44x slower than the baseline. The Safe approach is 1.69x slower than the baseline. In other words, on average, our Fast approach results in a 30% speedup over Pensieve, while the Safe approach results in executions 14% faster than Pensieve.

[2] Geometric mean is used for all normalized results.

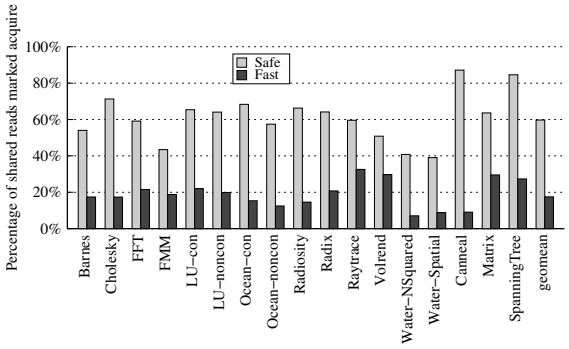

Figure 1. Static percentage of potentially thread-escaping reads that our analysis marks as an acquire.

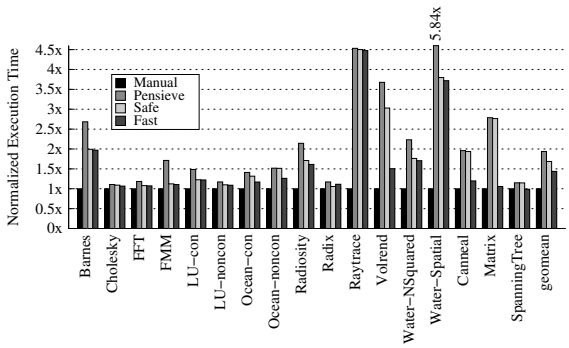

Figure 2. Execution time with fences placed using Pensieve, Safe, Fast and manual fence placement.

References

[1] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.

[2] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[3] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*, pages 285–294, 2003.

[4] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, 1995.

[5] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput.*, 50(8):824–833, 2001.

[6] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *POPL*, pages 378–391, New York, NY, USA, 2005. ACM.

[7] A. J. McPherson, V. Nagarajan, S. Sarkar, and M. Cintra. Fence placement for legacy data-race-free programs via synchronization read detection. *Technical Report,* University of Edinburgh, 2014.

[8] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[9] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent java programs. In *PPoPP*, pages 2–13, New York, NY, USA, 2005. ACM.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.