# Benchmarking Weak Memory Models

Carl G. Ritson
University of Kent
C.G.Ritson@kent.ac.uk

Scott Owens
University of Kent
S.A.Owens@kent.ac.uk

## Abstract

To achieve good multi-core performance, modern microprocessors have weak memory models, rather than enforce sequential consistency. This gives the programmer a wide scope for choosing exactly how to implement various aspects of inter-thread communication through the system's shared memory. However, these choices come with both semantic and performance consequences, often in tension with each other. In this paper, we focus on the performance side, and define techniques for evaluating the impact of various choices in using weak memory models, such as where to put fences, and which fences to use. We make no attempt to judge certain strategies as best or most efficient, and instead provide the techniques that will allow the programmer to understand the performance implications when identifying and resolving any semantic/performance trade-offs. In particular, our technique supports the reasoned selection of macrobenchmarks to use in investigating trade-offs in using weak memory models. We demonstrate our technique on both synthetic benchmarks and real-world applications for the Linux Kernel and OpenJDK Hotspot Virtual Machine on the ARMv8 and POWERv7 architectures.

***Keywords*** memory models, concurrency, benchmarking, performance

## 1. Introduction

The complexity of weak memory consistency models (WMMs), as implemented in modern hardware (x86, ARM, POWER, etc.), makes the challenging task of writing correct and efficient concurrent programs even more challenging. Some of the difficulties are semantic: understanding which behaviours the WMM allows, which it forbids, and how to ensure that a program relies only on guarantees that the WMM actually provides. Other difficulties are related to efficiency: implementing algorithms that take advantage of enough of the performance optimisations that gave rise to the memory model's weakness. There is an inherent tension between these two difficulties because restricting weak behaviours (by inserting extra fences, dependencies, etc. into the program) can make the memory behaviour easier to understand, and make it easier to establish, formally or informally, that the program is correct. However, this might harm performance by, for example, blocking the processor's ability to execute out of order, or by increasing memory bus traffic.



*PPoPP '16* March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03. . . $15.00
DOI: http://dx.doi.org/10.1145/2851141.2851150

Although either extreme – defensive fence insertion without regard to performance, or very carefully optimised code that is extremely hard to reason about – will occasionally be important, in this paper, we investigate the space between. That is, we are interested in the trade-offs between semantic clarity and performance. There has been some work in this area from the semantic side [5, 27], but the performance side has not been well investigated, with the notable exception of Marino et. al. [22, 26] who advocate full sequential consistency (in §5 we relate our results here to theirs). Performance is our focus here; we establish techniques that will allow programmers to understand what the practical performance trade-offs are, and therefore make informed decisions on how much of the complexity of the WMM they must contend with. This is especially important with recent and upcoming additions to the POWER and ARM ISAs. For example, the ARMv8 offers both release/acquire loads/stores and a variety of DMB fences, and one often has a choice of which to use in a particular situation. Making the right choice requires an understanding of both the semantics and performance of each alternative.

For the most part, these decisions on how to interact with the WMM are the province of systems programmers: those implementing operating systems, virtual machines, compilers, and the like. Sometimes, they are implementing a formal language-level WMM on top of the hardware's WMM, as is the case in Java and C11/C++11 [4, 7, 21]. Other times, they are providing a more informal memory model abstraction (e.g., for the Linux kernel [23]), or establishing a certain correctness criteria (e.g., linearisability) for a particular algorithm or data structure (e.g., a lock-free stack or queue). In each case, they have WMM-related choices to make in establishing a *platform* that other programmers will use to write parallel programs, choices that affect the platform's efficiency and the semantic guarantees that it provides.

Systems programmers naturally want to use the most efficient fence that they are certain is correct. However, weaker (and potentially more efficient) options are, in general, more difficult to reason about and establish the correctness of, and it will only be worth the effort if the weaker option is a significant improvement over a more obviously correct default. The nature of WMMs complicate these choices.

- It is not always clear without benchmarking that one correct approach is more efficient than another. For example, although a POWER `lwsync` will generally be cheaper than a full `sync`, on newer ARMv8 machines, it is not a priori obvious how the performance of various load-acquire and store-release primitives compares with various barrier instructions, although sometimes they can be used for similar semantic purposes.
- The semantics of lightweight fences and dependency preservation in ARM and POWER [12, 14, 25] are subtle, and difficult to reason about.
- The semantics of the concurrency abstraction being implemented might not be settled. For example, in Java, requiring

a strong barrier after the initialisation of a final field makes the language's high-level semantics easier to program to, but could increase the run-time cost.

- The hardware's architecture might permit weak behaviours that current implementations do not exhibit. The programmer must decide whether to ensure correctness with respect to future hardware even though doing so might have performance implications on current hardware.
- When the WMM is underspecified, or poorly understood, the programmer might be more cautious, and accept a higher performance penalty from defensive programming.
- In some situations, performance can be sacrificed to support a more useful semantics. For example, extra fences could help implement a variant of the C11/C++11 memory model that forbids out-of-thin-air values [8].

Our goal here is to help the programmer measure when it is worth the effort: whether the choice affects performance enough on their target applications.

In this paper, we propose a methodology for evaluating benchmarks, and categorising them according to what qualitative aspects of WMM performance they stress. One significant part of our contribution, then, is in detailing how one can establish the usefulness of a benchmark in this space to facilitate the reasoned optimisation and regression testing of high-performance concurrent code. We cannot provide definitive answers because of the nature of WMMs: most WMM-relevant instructions do not have straightforward performance characteristics. They depend on the nature of the program, the surrounding program context, the other threads in the program, and potentially other applications running on the system.

This paper is mostly example driven. After a brief explanation of our terminology §2 and summary of our methodology §3, we present a series of benchmarks and their results under various implementation strategies for the OpenJDK's Hotspot VM running on multicore ARMv8 and POWER7 machines §4.2, as well as the Linux Kernel running on an ARMv8 machine §4.3. We focus on both of these pieces of system software due to their widespread applicability to hardware with WMMs. In particular, OpenJDK is the only open-source Java implementation targeting both ARMv8 and POWER architectures, and it is widely used in industry. Since our results focus on how to analyse the performance characteristics of a particular hardware WMM, with only a few concrete conclusion about these particular platforms, they should be broadly applicable to other settings where fine-grained control of ISA-level concurrency primitives (e.g., fences) is available. [1]

## 2. Background and terminology

Throughout the paper, we use *performance* to mean either the throughput of an application (time to process a unit of work), or response time of an application (under a given load) to serve a new request. Both of these measures of performance also have an inherent *stability* as determined by the spread of results from repeated testing. For response time in particular, the maximum value obtained by testing (worst case) is a key measure.

Benchmarks can be broadly divided in to microbenchmarks and macrobenchmarks. A *microbenchmark* is a program designed reproduce a small number of known behaviours to facilitate in vitro testing. Microbenchmarks are usually extracted or synthesized from known structures or patterns in applications. They provide results which are relatively easy to interpret with reference to the known structure of the benchmark. The disadvantage of microbenchmarks is that they remove complex interactions between the benchmark structure and other program code – the sort of interactions which make WMM performance challenging to evaluate.

Hence, we also include *macrobenchmarks* that reproduce whole application behaviour over a known set of input data or end-user interactions. Complex behaviour between program elements is retained; however, benchmark execution time is much longer than a microbenchmark and behaviours of interest may be conflated with many unrelated activities. Macrobenchmark results are broadly more relatable to real-world application behaviour, e.g. a 20% increase the performance of a macrobenchmark is much more likely to results in a 20% performance increase in similar programs.

As mentioned in §1, we use the term *platform* to mean a system or library that implements some higher-level abstraction or facility on top of the hardware's WMM. The implementation of a platform requires decisions about where to put fences, which fences to use, and whether special instructions (e.g., release/acquire loads and stores) or other patterns (e.g., synthetic control-flow dependencies) should be used instead. For simplicity, we refer to a particular collection of these decisions as a *fencing strategy*, even though some of these things are not, strictly speaking, fences. We use the term *code path* to refer to a specific location in the platform's code where part of the fencing strategy is implemented.

We use the term *sensitivity* to describe the relative change in performance of a benchmark with respect to a change in the fencing strategy. For example, a benchmark which does not invoke memory barriers will show no sensitivity to changes in the fencing strategy, but a microbenchmark which only measures time taken by memory barriers will have absolute sensitivity to the memory barrier instruction sequence used.

## 3. Methodology

Our basic approach is to consider each benchmark as a black box that we run across various fencing strategies[2] for the underlying platform (e.g., the Linux kernel or the JVM), observing the resulting changes in performance. This can be applied in two complementary ways:

1. establishing the significance of a particular fencing choice for a platform by measuring sensitivity to changes across a number of benchmarks, and
2. establishing the sensitivity of a particular benchmark to the platform's fencing strategy by running it across a variety of choices.

In general, the sensitivity of a benchmark to a given code path that implements part of a platform's fencing strategy will be somehow related to the number of times that code path is invoked when running the benchmark. One could instrument a range of code paths with invocation counters to identify those that are most frequently executed by a given benchmark. In turn, this could be used to speculate which code paths a benchmark will be most sensitive to changes in. Although straightforward, this approach has key weaknesses: there is an inherent performance cost from counter instrumentation which might be hard to predict or unstable, counters may have subtle effects on the performance of the memory subsystem in multi-threaded programs, and although the count of invocations is indicative of sensitivity, it is not conclusive. For example, the code path might only be executed in a context that makes the operations contained within it fast relative to the same operations in a less frequent path that is executed in a worse context.

[1] We do not address x86 machines here because their WMM is much simpler semantically (TSO), and there are correspondingly fewer choices in how to use the model.

[2] The term 'fencing strategies' is defined in §2.

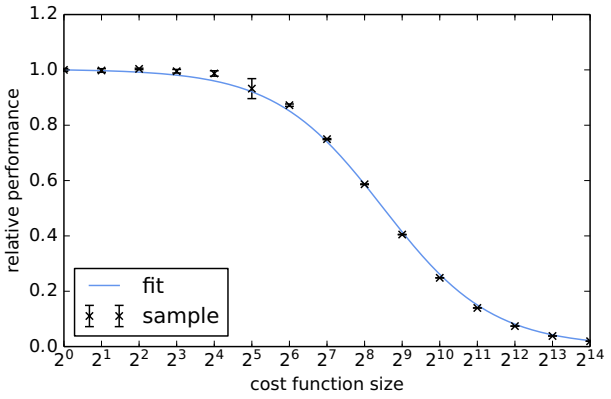


**Figure 1.** Example of fitting to compute sensitivity ($k = 0.00277 \pm 2.5\%$).

Because of the above concerns, we do not use counters. Instead, we use instruction sequences with known stable execution times that we call *cost functions*[3]. Unlike counters, cost functions do not need to do anything useful – they just take up a predictable amount of time. Therefore, we can implement them so that they perturb the system as little as possible: they do not need to access shared memory, and we can keep their code size very small to minimise effects on the instruction cache. Since the cost functions add a delay along a code path, we can use the relative change in benchmark performance when adding the cost function to observe how sensitive it is to that code path, relative to other code paths measured with the same cost function.

We first choose one or more of the most sensitive code paths to investigate. We then create an idealised model of the sensitivity of a given benchmark to each of these code paths by varying the execution time of the cost function, which is is essentially a spin loop with some means of controlling its run length. In the idealised model, we assume the normalised performance $p$ of the benchmark will follow the equation:

$$p = \frac{1}{(1-k)+ka} \tag{1}$$

Where $k$ is sensitivity of the benchmark to the given code path (which is a dimensionless ratio of execution times), and $a$ is the execution time of the cost function. Informally, this is saying that the overall performance $p$ is related to the change in cost $a$ times the sensitivity to that change $k$. [4]

By sampling the performance $p$ of the benchmark with increasing cost function sizes $a$, we compute the idealised sensitivity $k$ by curve fitting. For curve fitting we use the `curve_fit` non-linear least squares method from `scipy` [17] and report the estimated variance for all fits. Figure 1 shows an example of curve fitting to model sensitivity. If $k$ is comparatively low or variance is high, then the benchmark is not well suited to evaluating changes in the given code path. Sampling varied cost function activity also exposes complex behaviours which are not accurately modelled by simple sensitivity such as tipping point or phase behaviour in timing sensitive applications. These behaviours are interesting when they represent a class of applications, but must be well understood if used for benchmarking.

Once we have an empirical estimate of $k$, we can then relate changes in fencing strategies to each other on a consistent scale. We do this by changing the fencing strategy, then measuring $p$, and then computing $a$, which represents the cost of that change. Solving (1) for $a$ gives:

$$a = -\frac{(1-k)p - 1}{kp} \tag{2}$$

This can be used to directly compare the results of in vitro and in vivo testing, and hence is useful for exploring the whole system impact of small implementation changes.

For example, a microbenchmark (e.g. timing loop) can be used to establish relative costs of two implementation instruction sequences, $f$ and $g$. The relative impact on performance coupled with the sensitivity measure can be used to derive similar relative costs when tested in a macrobenchmark. If sequences do not have any complex effects then we would expect the two measures to agree, e.g. $g$ takes twice as long as $f$ when measured through micro or macro benchmarking. In general this will be true for the optimisation of computation intensive code; however, for memory intensive code, and particularly memory model implementation details, we expect to see a divergence between the results. This divergence results from the complex state of the memory hierarchy exercised by the macrobenchmarks, but not microbenchmarks. We have no particular expectation of what the divergence will be, only that the presence of divergence is interesting and indicates a benchmark is useful for testing a given code path.

## 4. Evaluation

We now turn to the exploration of our methodology (§3) on two platforms: the OpenJDK Hotspot virtual machine and the Linux Kernel. We establish the relevance of a range of benchmarks to testing fencing strategies in these two settings and present results from testing a range of implementation changes. We summarise the conclusions that we can draw from our experiments in §4.4.

### 4.1 Common Methodology

The experiments described in this paper were performed on an X-Gene 1 architecture 64-bit ARMv8 processor with 8-cores running at 2.4Ghz (with 16GiB of RAM). This processor has out-of-order execution, speculation, and observable weak memory behaviour. It approximates emerging commodity processors in both high-end mobile devices and high-density servers. While the ARMv8 architecture is likely to be more common in the future, we also provide results from a 12-core POWER7 system running at 3.7Ghz (with 128GiB of RAM). The POWER architecture also has well-documented weak memory behaviour [25] that is broadly similar to the ARM's, and so there are reasonable grounds for comparison.

Our evaluation technique makes extensive use of cost functions implemented by small instruction sequences injected into existing code. This cost function is a spin loop, and on ARM it is implemented as in Figure 2 and on POWER as in Figure 3. Note that in both cases we do not know the availability of registers to store the loop counter and hence must spill a register to the stack. This means the cost function will have a small impact on the memory subsystem. However, on ARMv8 OpenJDK a scratch register is available which allows the stack operations to be elided.

To provide accurate comparisons of modified code to a base case we always inject a placeholder nop sequence into the base case. This keeps the binary image size equivalent between the base case and the test case, and minimises jitter, especially jitter introduced from changes in cache alignment.

[3] Cost functions are not implemented as functions to be called on the hardware, but injected directly inline. We use the word 'function' in the mathematical sense that they have a parameter controlling how much time ('cost') they use.

[4] We use $1/(1-k+ka)$ instead of $1/(1+ka)$ because we assume $a$ will never be 0 due to nops and untaken branches in our baseline. In practice $k$ is very small so both formulas would yield similar results.

```
1 stp x9, xzr, [sp, #-16]!
2 mov x9, N
3 subs x9, x9, #1
4 bne -4
5 ldp x9, xzr, [sp],#16
```

**Figure 2.** ARMv8 cost function, where `N` is replaced by a loop iteration count. In OpenJDK `x9` is a scratch register allowing for the removal of the first and last stack instructions.

```
1 std r11, -8, r1
2 li r11, N
3 addi r11, r11, -1
4 cmpwi cr7, r11, 0
5 bne cr7, -8
6 ld r11, -8, r1
```

**Figure 3.** POWER cost function, where `N` is replaced by a loop iteration count. This is only valid when comparison register `cr7` is unused, which is true for OpenJDK.

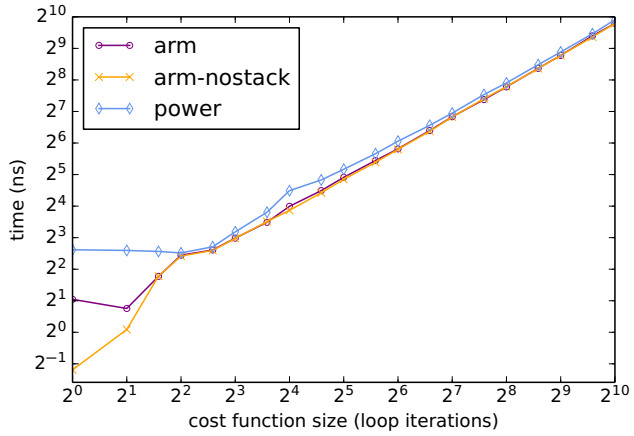


**Figure 4.** Time taken to execute respective cost functions.

Ideally our cost functions will require time to execute linearly proportional to the number of loop iterations; however, due to pipelining effects this is unlikely to be true for small numbers of loops. We establish through microbenchmarking the time taken (in nanoseconds) to execute the cost function for the values of `N` which will be used for investigation. These results are shown in Figure 4. As expected the relationship between cost function size and execution time becomes linear only when the number of loops is large. However we do not need to full understand this relationship, but simply apply the observed execution time of a given cost functions size to relate each data point to time taken executing the cost function.

Unless otherwise noted all reported results are geometric mean (reduce impact of outliers) from six or more samples measured after one or more warm-up runs of a given benchmark. All error bars represent a 95% confidence interval computed using the Student's t-distribution, which is appropriate for the small number of samples available. In the case of comparative results, errors are compounded as would be expected, i.e. comparative minimum is test case minimum divided by base case maximum.

### 4.2 OpenJDK Hotspot

In this section, we investigate the low-level instruction sequences used to implement Java's memory model within the OpenJDK Hotspot VM. Broadly speaking Java's memory model affects the behaviour of `volatile` varibles, low-level methods used by `java.util.concurrent`, and subtle interactions between the Java application and the VM's JIT compiler and garbage collector. Within OpenJDK, the Java Memory Model is enforced by elemental memory barriers `LoadLoad`, `LoadStore`, `StoreLoad` and `StoreStore` generated by the compiler. These are then assembled according to the WMM of the target hardware.

There also exist higher-level IR instructions representing combinations of the elemental barriers, namely `Volatile`, `Acquire`, `Release`, `LoadFence` and `StoreFence`. For example each volatile load is preceded by an invocation of the `Volatile` barrier and followed by `Acquire`. Conversely volatile stores are preceded by `Release` and followed by `Volatile`. This provides sequentially consistent behaviour for volatile variables. Various atomic APIs and internal usages generate combinations of barriers allowing more relaxed behaviour.

We focus on the fencing strategy of the currently under development JDK9 (Java 1.9). On POWER architecture, `Volatile` is equivalent to combined `LoadLoad`, `LoadStore`, `StoreLoad` and `StoreStore` barrier. `Acquire` and `LoadFence` produce a combined `LoadLoad` and `LoadStore` barrier. While `Release` and `StoreFence` produce a combined `LoadStore` and `StoreStore` barrier. Underlyingly `StoreLoad` becomes a `hwsync` instruction, while all other elemental barriers become `lwsync` instructions.

In JDK9's ARMv8 implementation the elemental memory barriers are compiled to `dmb ishld` for `LoadLoad` and `LoadStore`, `dmb ishst` for `StoreStore`, and `dmb ish` for `StoreLoad`. On JDK9, we can replace many barriers with appropriate uses of ARMv8's load-acquire/store-release instructions for affected memory locations (such as volatile accesses). Older JDK8 behaviour (with barrier instructions) can be enabled using the `UseBarriersForVolatile` runtime flag. We assess the JDK8 behaviour as the base case and experiment with JDK9 behaviour (load-acquire/store-release instructions) as an optimisation.

On both POWER and ARMv8 we modified the low-level assembler of the JIT compiler to change the barrier instruction sequence, inserting `nop` instructions or the cost functions shown in Figures 2 and 3. We measured the performance impact of inserting `nop` instructions into every elemental memory barrier, six instructions on POWER and three instructions on ARMv8. We found the peak drop in performance to be 4.5% for *h2* on ARM, with the mean drop for ARM being 1.9% and Power being 0.7%.

For benchmarks we draw on the DaCapo 9.12 suite of real-world applications widely used for testing JVM performance [6]. We filter the DaCapo suite to benchmarks determined by Kalibera el al to have notable concurrent behaviour [19]. This gives *h2* an in-memory database, *lusearch* a text search using the lucene search library, *sunflow* a ray-tracer, *tradebeans* and *tradesoap* which are client-server-database transaction processing benchmarks, and *xalan* an XML to HTML transformation benchmark. We supplement the DaCapo suite with a big data style benchmark based on Apache Spark (1.4.1), a multi-threaded map-reduce engine [2], emulating the methodology of Gidra et al [13] by running the GraphX RageRank implementation on 20 million nodes from the Stanford LiveJournal dataset [20]. This benchmark is referred to as *spark* in our results set.

For all benchmark runs we use a common set of JVM options. We disable the garbage-first collector which is the default on JDK9 and use the throughput-oriented collector from JDK8. This makes our results more broadly applicable to existing JDK8 installations. We limit the number of parallel collector threads to eight and

number of concurrent collector threads to four on ARMv8 and eight on POWER7. Default heap sizing is used for all DaCapo suite benchmarks. For *spark* the minimum and maximum heap is set at 12GiB and the work-load RDD is partitioned into 16 segments. Each benchmark is executed multiple times in the same JVM and the first two iterations discarded as warm-up (particularly for the JIT compiler).

#### 4.2.1 OpenJDK Results

Figure 5 shows the change in relative performance across all benchmarks as the size of the injected cost function is increased. Using curve fitting the relative sensitivity $k$ to the fencing strategy is computed for each benchmark. Both visually and with respect to the value of $k$ the *spark* benchmark is clearly the most sensitive to changes in the fencing strategy. On ARM this is followed by *xalan*; however, on Power *xalan* is unstable to the point of not being a reasonable benchmark. We suspect the instability of *xalan* on Power is related to the symmetric multithreading strategy of the CPU. On ARM *lusearch*, *tomcat* and *tradebeans* have significant instability, while on Power *tomcat*, *sunflow* and *xalan* are equally unstable. Overall *spark* is both stable and sensitive. With its general real-world applicability to big data workloads, *spark* is clearly a good benchmark to use when investigating memory model performance impact.

Having established the relevance of the *spark* benchmark we use it to investigate the sensitivity of specific elemental memory barriers. Figure 6 shows the results of injecting the cost function into each elemental memory barrier in turn. It should be noted that if a combination of barriers is requested, i.e. acquire or release, then a code path will appear in multiple results. On both ARM and Power the `StoreStore` barrier has the most impact, with Power being particularly sensitive. The breakdown of other barriers highlights implementation differences in OpenJDK between the two architectures. Clearly the developers of the ARM implementation are more defensive, adding more `LoadLoad` and `LoadStore` barriers than the Power developers who appear to rely more heavily on `StoreStore` and `StoreLoad`.

Given that *spark* is most sensitive to `StoreStore` on both ARM and Power, we investigate this further. On ARM we modified the generation of `StoreStore` from `dmb ishst` to `dmb ish`, we found this produced a statistically significant drop in performance of 0.7%. The mean relative performance of $0.99293$ combined with a sensitivity of $0.00884788$ suggests an increase in `StoreStore` execution time of 1.8ns. We were unable to determine a difference in execution time for `dmb ishst` and `dmb ish` using microbenchmarking; thus the figure of 1.8ns may be representative of their comparative executions times in practice. On Power we modified the generation of `StoreStore` from `lwsync` to `sync` (heavy-weight sync) and observed a performance drop of 12.5%. The mean performance of $0.87530$ and sensitivity of $0.01332662$ compute an increase in `StoreStore` execution time of 11.7ns (extra time over `lwsync`). Basic microbenchmarking of `sync` and `lwsync` determines their execution times to be 6.1ns and 18.9ns respectively, which is broadly consistent with our result. Furthermore the arithmetic mean of increases computed from other tests (excluding *xalan*) is 11.8ns. This suggests that on POWER7 the behaviour of these instructions is workload agnostic.

On ARM we performed a further measurement investigating the performance difference between memory barriers and load-acquire/store-release instruction implementations. Results are mixed, we observe performance increases with load-acquire/store-release of 2.9% in *xalan* and 3.0% in *sunflow*. We see no statistically significant change in *lusearch*, *tradebeans*, or *tradesoap*. The remaining benchmarks show performance drops: *h2* 0.3%, *spark* 0.5% and *tomcat* 1.7%. Given that *spark* and *xalan* appear to be stable and sensitive on ARM, the relative scale of performance increases to decreases favours load-acquire/store-release instructions.

Having established confidence in the *spark* benchmark we tested a patch pending approval which reduces the use of `dmb` instructions in parts of OpenJDKs locking code [15]. We found that this yields a 2.9% increase in *spark* performance when running with load-acquire/store-release instructions, but a 1% drop in performance when using memory barriers. This result hints at subtle interactions between load-acquire/store-release and `dmb` instructions which require further investigation.

### 4.3 Linux Kernel

The Linux Kernel memory model is not formally specified, but documented in the `memory-barriers.txt` file in the documentation directory of the sources [16]. The memory model makes few expectations about compiler behaviour and is enforced by the explicit use of barrier macros either directly or through larger concurrency frameworks such as RCU. These macros are platform specific and implemented in assembly in each architectures `include/asm/barriers.h` header file. These macros are the focus of our investigation of the Linux Kernel memory model performance. Additionally we incorporate testing of the `READ_ONCE` and `WRITE_ONCE` macros which prevent compiler or hardware speculation from duplicating reads or writes to memory locations. To facilitate this we use Linux Kernel 4.2 which incorporates these macros as a replacement for `ACCESS_ONCE` used for both reads and writes [23].

There is no established suite of benchmarks for investigating the impact of the Linux Kernel memory model. For our investigation we gathered a small set of candidate benchmarks mixing synthetic benchmarks and real-world software.

- *Kernel compilation*: a classic test of whole system performance is to perform a compilation of the Linux Kernel itself parallelized using the `-j` flag.
- *netperf* [18]: is a microbenchmark of network bandwidth, we use this to test TCP and UDP performance over the kernel loopback interface with 4096-byte packets.
- *ebizzy* [9]: is a synthetic benchmark which simulates common webserver workload, in particular this stresses the memory management subsystem.
- *OSM*: is an instance of the OpenStreetMap tile server stack which generates map tiles using geospatial data in a Postgres database. This can be driven directly to test tile generation rates (labelled *osm_tiles* in our results), or queried from a remote host through an Apache webserver to probe service response times (labelled *osm_stack* in our results).
- *lmbench* [24]: is a collection of Linux Kernel microbenchmarks which measure the time taken to execute a range of kernel system calls. We use a subset: `fcntl`, `proc_exec`, `proc_fork`, `select_100`, `sem`, `sig_catch`, `sig_install`, `syscall_fstat`, `syscall_null`, `syscall_open`, `syscall_read`, and `syscall_write`. To simplify presentation and downplay the role of any one microbenchmark, *lmbench* results are aggregated by an arithmetic mean (post comparison to the base case).
- *h2*, *spark*, *xalan*: are inherited from §4.2 as the most sensitive benchmarks from our evaluation of OpenJDK.

We compile the Linux Kernel with illegal, but uniquely identifiable, instruction sequences replacing all invocations of memory model macros. The kernel binary is then rewritten with an appropriate combination of `nop` and `dmb` instructions or cost function instructions as appropriate for each test. This maintains the binary

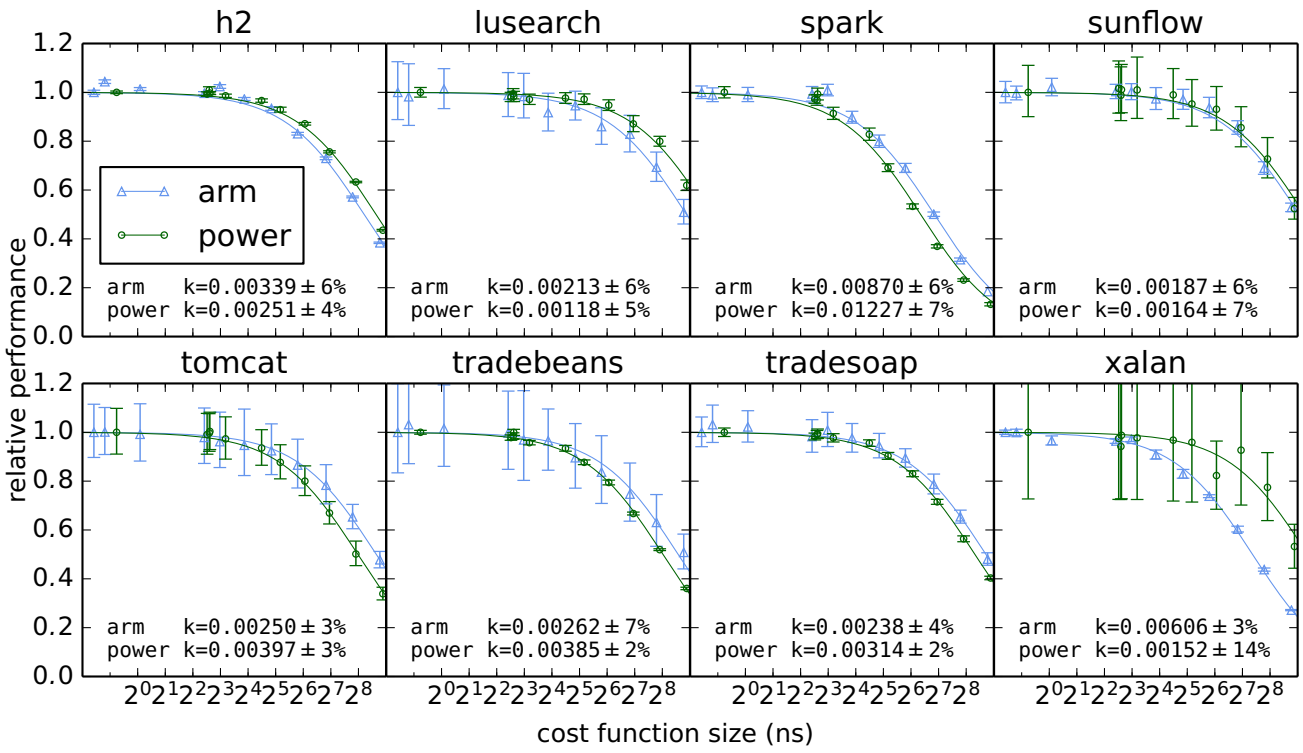


**Figure 5.** OpenJDK. Impact of increasing cost function size when injected into all memory barriers.

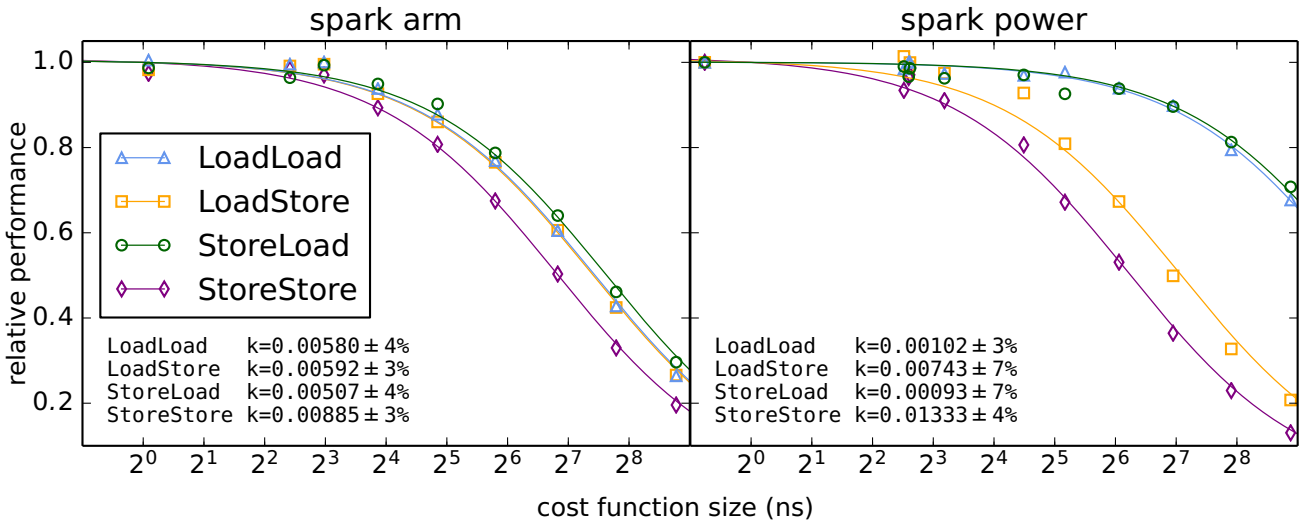


**Figure 6.** OpenJDK. Performance impact on *spark* benchmark of cost function when injected into specific elemental memory barriers.

size of all code section invariant regardless of the test. We observe an arithmetic mean 1.9% drop in performance impact across all benchmarks from introducing `nop` instructions into macros alongside their usual barrier instructions compared to an unmodified Linux Kernel 4.2. The largest drop in performance (6.6%) was observed in the *netperf* benchmarks. All further measurements in this section are compared to this modified kernel (default barriers with `nop` padding) as the base case.

#### 4.3.1 Linux Kernel Results

Compared to OpenJDK (§4.2) our investigation covers more code paths (14 compared to 4) and benchmarks (11 compared to 8). Therefore we start by probing sensitivity of benchmarks with respect to individual macros, rather than modelling general response of macros or benchmarks. This approach allows us to map the space more quickly, at the loss of our ability to reason about the general behaviour of a given benchmark. Expecting generally lower sensitivity to kernel behaviour, we inject a large cost function (1024

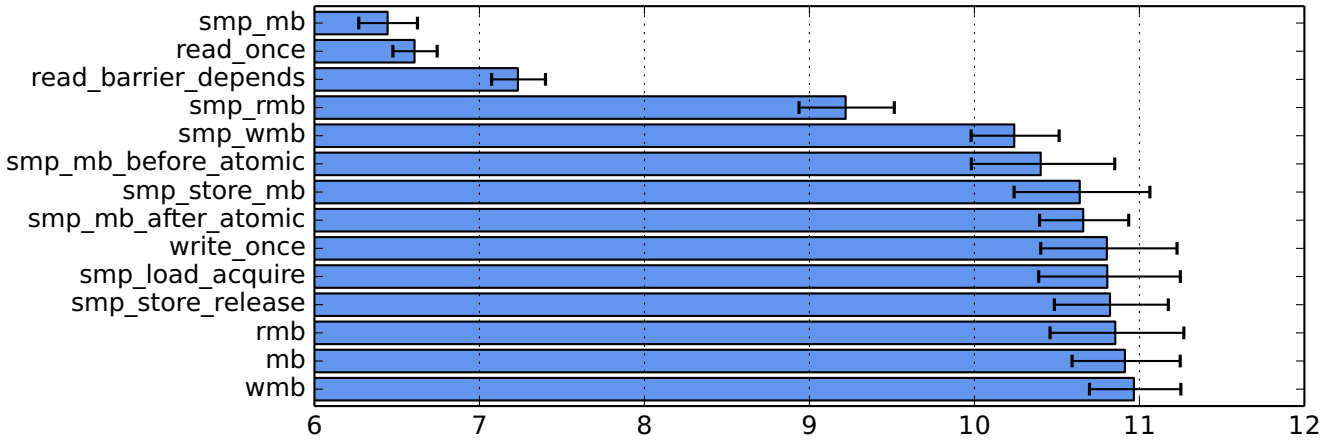


**Figure 7.** Linux. Sum of relative performance for all benchmarks aggregated with respect to memory model macro modified.

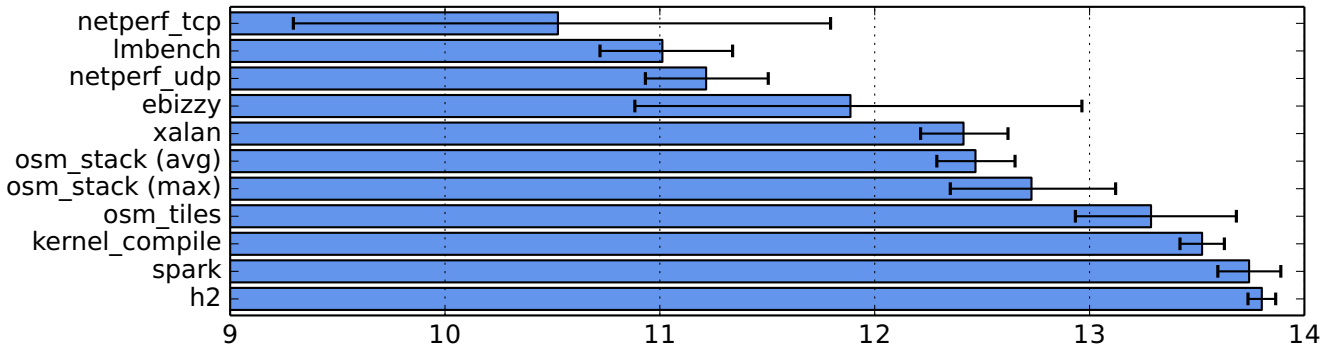


**Figure 8.** Linux. Sum of relative performance for benchmarks aggregated across all macro modifications.

loop iterations) into each macro in turn, and measure the relative performance impact on all benchmarks.

Our initial investigation produces 154 data points. Assuming all macros and benchmarks are equal we aggregate either by benchmark or macro to produce ranking of interest. Figure 7 shows aggregation across benchmarks for each macro, highlighting which macros have the biggest impact. It is clear that `smp_mb`, `read_once` and `read_barrier_depends` have the most impact. By default only `smp_mb` produces an instruction sequence (`dmb ish`), the others are simply compiler barriers.

Figure 8 shows the aggregation across macros for each benchmark, indicating which benchmarks are generally sensitive to changes in the macros under investigation. As might be expected, the synthetic and microbenchmarks, *netperf*, *ebizzy* and *lmbench*, are most sensitive with *osm_stack (avg)* and *xalan* being potential real-world applications with relatively high sensitivity. Some of most sensitive OpenJDK benchmarks, *h2* and *spark*, are almost completely insensitive to changes in the macros we are testing. We suspect that these benchmarks rely heavily on the Java Virtual Machine to coordinate their concurrency and thus have very few interactions with the kernel.

We chose to investigate further the `read_barrier_depends` macro. This macro is semantically interesting because it controls visible speculation across control dependencies, which can be aggressive on modern ARM hardware.[5] It is conceivable that with increasingly aggressive speculation on ARM processors that some control dependencies may be speculated in ways that lead to unintended consequences. The `read_barrier_depends` macro is used to enforce these dependencies as part of the `READ_ONCE_CTRL` macro (not investigated here). We chose to look at `read_barrier_depends` on the basis that `READ_ONCE_CTRL` has not yet been adopted into common usage and `read_barrier_depends` provides a superset of required control dependencies. It should be noted that this means our results may indicate a larger impact than justified, should the kernel need to adopt a fencing strategy for control dependencies.

Figure 9 shows the cost function sampling of `read_barrier_depends` with respect to the six previously identified most sensitive benchmarks. We find the sensitivity of real-world applications, *osm_stack* and *xalan*, to be very low. The synthetic benchmark, *ebizzy*, shows some sensitivity. While networking (*netperf*) benchmarks are particularly sensitive, the TCP benchmark has particularly poor stability. Visually both *netperf_udp* and *lmbench* show a trend toward a more linear relationship than that used by our model, this behaviour requires further investigation.

[5] This sort of speculation is also potentially relevant to the design of language-level WMMs that have no out-of-thin-air values.

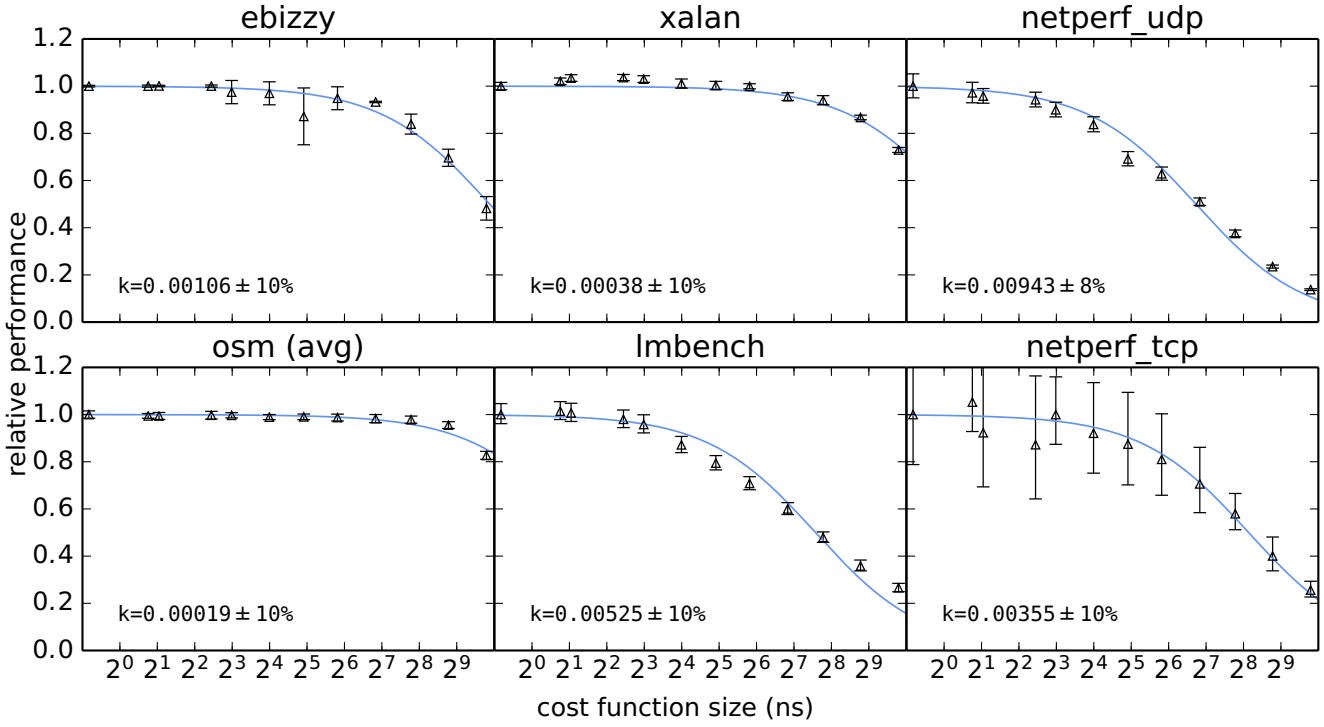


**Figure 9.** Linux. Sensitivity analysis of benchmarks with respect to `read_barrier_depends`.

We investigate a number of fencing strategies for `read_barrier_depends` with their relative performance shown in Figure 10. In the figure, *base case* is the kernel with default barriers and nop sequences. In *ctrl* and *ctrl+isb* `read_barrier_depends` contains an instruction sequence designed to introduce a true control dependency by testing the value of the last load against a constant (42) and conditionally branching over an impotent instruction. In the case of *ctrl+isb* the impotent instruction is an `isb`. For *dmb ishld* and *dmb ish* the implementation of `read_barrier_depends` is simply that instruction. Finally *la/sr* supplements the case of *dmb ishld* by adding `dmb ishld` to `READ_ONCE` and `dmb ishst` to `WRITE_ONCE`, with the intention of adding load-acquire/store-release semantics across all annotated reads and writes. Each of these test cases replicates a method for introducing ordering dependencies from the ARMv8 manual [3, B2.7.4].

The results from *ebizzy* contain too much variance to be statistically significant, with the exception of the *ctrl+isb* where performance drops several percent. The *osm_stack* results show a small, but statically significant drop of up to 1%, this result could be interpreted as reflective of the typical real-world impact of changing `read_barrier_depends`. Results from *xalan* seem to indicate that performance may be improved by adding `dmb ishld` instructions as both scenarios which add these instructions see a boost in performance. The trend between *netperf* results is almost identical for TCP and UDP; however, UDP shows much more subdued and stable changes. It is important to consider that UDP was determined to have higher sensitivity to `read_barrier_depends` than TCP, and thus should be considered more indicative. Given that *netperf* is a measure of bandwidth, it may be more appropriate to consider the peak values more important than mean values, in which case TCP and UDP benchmarks are almost the same in their results. Directly testing kernel call performance, *lmbench*, shows that all scenarios have an impact on system-call tests.

Computing $a$, the cost increase, for each test case against the *lmbench* microbenchmark suite gives the following values: *ctrl* 4.6ns, *ctrl+isb* 24.5ns, *dmb ishld* 10.7ns, *dmb ish* 11.0ns, and *la/sr* 21.7ns. The mean of these values computed from all other benchmarks are: *ctrl* 10.1ns, *ctrl+isb* 24.5ns, *dmb ishld* 1.8ns, *dmb ish* 10.7ns, and *la/sr* 15.9ns. The key discrepancies here are *ctrl* and *dmb ishld*. In the case of *ctrl* we speculate the effect on the branch prediction of the additional branch is more noticeable in macrobenchmarks. The *dmb ishld* results support it having complex behaviour, and not simply mapping to *dmb ish*. Finally the results for *ctrl+isb* match indicating that the behaviour of `isb` is broadly stable.

Overall, the introduction of `isb` instructions is unreasonable due to their effect on the processor pipeline; however, if ordering is required then `dmb ishld` or `dmb ish` represent the best case scenarios. The ordering guarantees of `dmb ishld` are also potentially stronger than those of a true control dependency (without `isb`) hence this is a particularly positive result. The potential performance benefits of `dmb ishld` require further investigation.

### 4.4 Reflections

In OpenJDK on POWER7 we observed a 12.5% decrease in performance for the *spark* macrobenchmark by changing a single barrier instruction in one memory model related code path. The *spark* benchmark is generally indicative of a class of real-world big data applications, making this a significant result. Applying the same modification in other widely accepted concurrent benchmarks yields much smaller changes, for example only 1.5% in *lusearch*. This order of magnitude difference between results could separate an acceptable implementation change and an unacceptable one, emphasising the importance of empirically establishing the sensi-

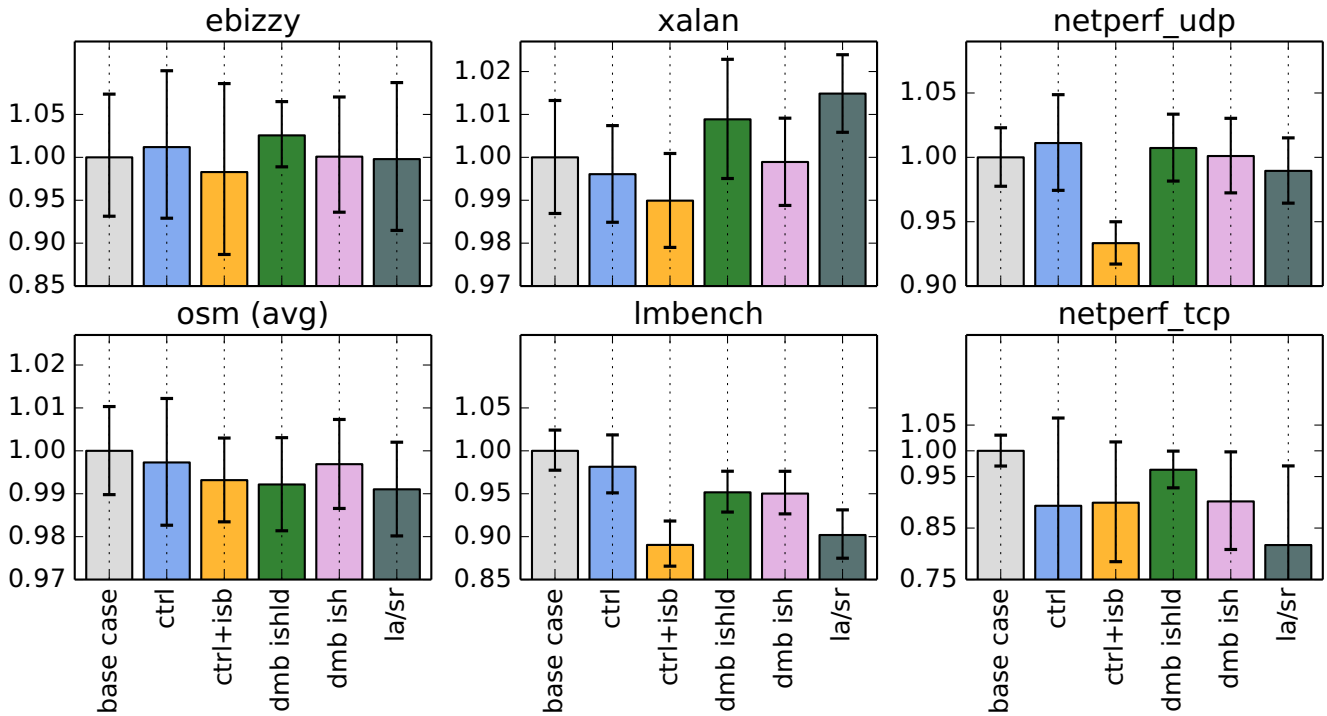


**Figure 10.** Linux. Relative performance for test implementations of `read_barrier_depends`.

tivity of benchmarks used for evaluation. On ARMv8, where the choice of barrier instructions is more subtle, we observe only small changes of a few percentages for a range of configurations; however, having established the sensitivity of our benchmarks we have a high degree of confidence in the relevance of the result.

In the case of POWER7, a microbenchmark measuring the time taken to execute a large number of `sync` and `lwsync` instructions would be able to establish a threefold difference in execution time between the two instructions. Our modelling approach and results allow us to verify this behaviour in practice. On our ARMv8 machine, a similar microbenchmark is not able to determine any difference between `dmb ish` variants; however, we are able to observe `dmb ishld` and `dmb ishst` to be faster `dmb ish`. The differences in measured behaviour of `dmb ish` variants between benchmarks of the Linux Kernel quantify the subtle behaviour of these instructions. With reference to Figure 10 for *dmb ishld* and *dmb ish*, the peak performance is almost identical, but the minimum performance indicates that `dmb ish` does substantially more work in many cases. Microbenchmarks such as *lmbench* are unable to expose these differences.

We find that high sensitivity benchmarks produce results which accurately calculate the change in cost of a code path. This fits with our model, i.e. large values of $k$ minimise errors in computed values of $a$. Our results also indicate that an aggregation (i.e. arithmetic mean) of lower sensitivity benchmarks may provide a similar result. Again, this fits our model by averaging to minimise errors.

## 5. Related Work

There is not a large body of work evaluating memory model or fencing strategy performance. In part this is because aggressively weak hardware, especially multicore ARM processors, has only recently become commonplace. Work on formalising these WMMs and thus determining valid fencing strategies is still ongoing [12, 14]. Furthermore the diversity of synchronisation instructions in the ARMv8 and POWER8 ISAs are only just being utilised by current operating systems and programming languages.

A notable evaluation of memory model performance by Marino et al. advocates the use of SC semantics [22, 26]. They demonstrate by careful modification of the LLVM compiler that it is possible to preserve SC semantics with a maximum slowdown of 34%. The CPU performance oriented benchmark suite used is appropriate for evaluating compiler output, but does not specifically target concurrent behaviour. Importantly, the authors claims on slowdown are with respect to x86 TSO while our experiments focus on weaker models. Our evaluation of fencing strategies for read dependencies on ARM (see §4.3.1) gives some indication that it may be possible to support an SC execution strategy on ARM within Marino's upper performance bound; however, their finding of a mean slowdown of 3.8% is unlikely to be replicated. If an SC compatible fencing strategy was proposed for use in the Linux Kernel then our work would provide a good starting point for evaluating the performance implications.

In their work on the Buffer Memory Model (BMM) for Java, Demange et al. modified the Fiji VM to implement an alternative memory model for which a range of compiler re-orderings could be verified [11]. The BMM was designed to have an efficient mapping to TSO architectures and the authors observed their memory model changes to be largely performance agnostic on x86. The authors used the DaCapo 9.12 suite of benchmarks for measurements as we have done in this work; however, the subset chosen omits most of the benchmarks with concurrent behaviour as established by Kalibera [19]. In their conclusion, the authors postulate that their memory model changes will have a large impact on more relaxed architectures (such as Power), but cannot speculate on the size of this impact.

Alglave et al. used a range of source and binary analysis techniques to search for execution patterns relating to known memory model fragments (litmus test shapes) in 17000 packages from the Debian Linux distribution [1]. While the authors approach is not specifically concerned with performance, a by-product of their search is detecting software which may be affected by changes in memory model or fencing strategy. These results are complementary to our approach and could be used to propose new candidate benchmarks for in-depth evaluation using our methods.

Our performance evaluation technique is similar to the Causal Profiling approach of Curtsinger and Berger [10]. Causal Profiling uses empirical experimentation to establish the potential impact of speeding up a code path. A given code path is virtually sped up by slowing down all other threads executing concurrently with it. This provides an estimate of the whole program impact of optimising a given code path. Conversely our approach slows down only the code path under evaluation in a manner agnostic to threading. The assumption is that the model our technique creates will be used primarily to compare a range of implementations which negatively affect the performance of the application under test. Furthermore, our thread agnostic method is considerably less invasive making it more practical to apply to low-level system programming environments such as operating system kernels.

## 6. Conclusion

We developed and applied techniques for selecting and modelling benchmarks demonstrating

- a fixed-size cost function across a range of benchmarks and code paths to establish sensitivity rankings (§4.3.1), and
- a variable-size cost function to model the sensitivity of a given benchmark to a specific code path (§4.2.1 and §4.3.1).

These techniques are generally applicable wherever a code path can be annotated with a cost function, but are most relevant for evaluating changes with complex effects on the processor pipeline and memory subsystem which are difficult to replicate in microbenchmarks. While in this paper we focused on the comparison of barrier implementations on a single instance (processor design) of each hardware architecture, the techniques demonstrated could also be used for comparison of barrier costs between different implementations of an architecture. This is particularly true where sensitivity rankings can be used to establish *real world* use cases sensitive to a given code path.

An obvious extension to the work in this paper is to explore the annotation of code paths related to compiler optimisations. This is particularly relevant to language level WMMs where compiler optimisations may add, remove or reorder memory accesses. With the JVM JIT compiler this could be accomplished by adding a dedicated cost function IR node which is added to code paths where a given optimisation occurs or would occur. These IR nodes could then be assembled with or without cost function instructions. The process of iterating the cost function could also be encapsulated in the VM, potentially yielding a turnkey evaluation system. Within the context of JVM it is also worth exploring the relationship of our measurements to existing sampling techniques used for JIT optimisation. However a JIT compiler is not required, similar modifications could be made to a C11 compiler such as GCC. Binary rewriting techniques may also be applicable for exploring fencing strategies in already compiled code, e.g. C11 atomics.

## Acknowledgments

This work was funded by EPSRC grants EP/K040561/1 and EP/M017176/1.

## References

[1] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. doi:10.1145/2627752.

[2] Apache Software Foundation. Apache Spark: lightning-fast cluster computing, 2014. URL `http://spark.apache.org`.

[3] ARM Limited. *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. ARM Limited, 2015.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL '11: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, 2011. doi:10.1145/1926385.1926394.

[5] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015. Proceedings*, pages 283–307, 2015. doi:10.1007/978-3-662-46669-8_12.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006. doi:10.1145/1167473.1167488.

[7] H. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, 2008. doi:10.1145/1375581.1375591.

[8] H. Boehm and B. Demsky. Outlawing ghosts: avoiding out-of-thin-air results. In *MSPC '14: Proceedings of the workshop on Memory Systems Performance and Correctness*, pages 7:1–7:6, 2014. doi:10.1145/2618128.2618134.

[9] R. R. Branco and V. Henson. ebizzy benchmark, 2008. URL `http://ebizzy.sourceforge.net`.

[10] C. Curtsinger and E. D. Berger. Coz: finding code that counts with causal profiling. In *SOSP '15: Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015. doi:10.1145/2815400.2815409.

[11] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: a buffered memory model for Java. In *POPL '13: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 329–342, 2013. doi:10.1145/2429069.2429110.

[12] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL '16: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[13] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: a garbage collector for big data on big NUMA machines. In *ASPLOS '15: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 661–673, 2015. doi:10.1145/2694344.2694361.

[14] K. E. Gray, G. Kerneis, D. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *MICRO-48: Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015. doi:10.1145/2830772.2830775.

[15] A. Haley. OpenJDK RFR: 8135187: DMB elimination in AArch64 C2 synchronization implementation, 2015. URL `http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-September/018899.html`.

[16] D. Howells and P. E. McKenney. Linux kernel memory barriers, 2015. URL `https://www.kernel.org/doc/Documentation/memory-barriers.txt`.

[17] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

[18] R. Jones. Netperf benchmark, 2015. URL `http://www.netperf.org`.

[19] T. Kalibera, M. Mole, R. E. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *OOPSLA '12: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 335–354, 2012. doi:10.1145/2384616.2384641.

[20] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, 2014. URL `http://snap.stanford.edu/data`.

[21] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, 2005. doi:10.1145/1040305.1040336.

[22] D. Marino, A. Singh, T. D. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an SC-preserving compiler. In *PLDI '11: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–210, 2011. doi:10.1145/1993498.1993522.

[23] P. E. McKenney. Linux-kernel memory model. Technical report, ISO IEC JTC1/SC22/WG21, 2015. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4444.html`.

[24] L. McVoy and C. Staelin. LMbench - tools for performance analysis, 2012. URL `http://www.bitmover.com/lmbench/`.

[25] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI '11: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 175–186, 2011. doi:10.1145/1993498.1993520.

[26] A. Singh, S. Narayanasamy, D. Marino, T. D. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *39th International Symposium on Computer Architecture, ISCA 2012*, pages 524–535, 2012. doi:10.1109/ISCA.2012.6237045.

[27] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL '15: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220, 2015. doi:10.1145/2676726.2676995.