What consistency guarentees do University of Edinbugh DICE machines uphold?

Orfeas Liossatos

October 11, 2022

This coursework will take as a subject the physical DICE machine `kirby`.

1 ISA, number of cores, and shared memory processor

The bash user command `lscpu` displays information about the system architecture. [1] On the DICE machine `kirby`, running `lscpu` prints, in particular:

```
Architecture: x86_64
...
CPU(s): 6
Thread(s) per core: 1
Core(s) per socket: 6
Sockets(s): 1
NUMA node(s): 1
...
Model name: Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz
...
L1d cache: 192 KiB
L1i cache: 192 KiB
L2 cache: 1.5 MiB
L3 cache: 9 MiB
NUMA node0 CPU(s): 0-5
```

Although `lscpu` outputs `x86_64` under `Architecture`, the system model is *Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz*, and therefore uses Intel's ISA renaming: *Intel64*, which, while remaining "code-compatible", has slight differences from the original x86_64 architecture, `AMD64`.[2] Furthermore, this model implements three instruction set extensions, *Intel® SSE4.1*, *Intel® SSE4.2*, and *Intel® AVX2* which are SIMD instruction sets, featuring for example `PCMPEQQ` which compares packed integers for equality, `PCMPESTRI` which performs packed comparison of string data with explicit lengths, and the double quadword shift-left logical instruction `PSLLDQ`; in each extension, respectively. [4] [5] [6] As for the number of cores, the machine `kirby` has `Socket(s)` ×

`Core(s) per socket` $\times$ `Thread(s) per core`$= 1 \times 6 \times 1 = 6$ logical cores. Furthermore, all the CPUs belong to a single *NUMA node*, which means that they are grouped together and communicate over single bus.[7] Since there is only one such node, this means that all the processors share the single large 9MiB L3 cache. [3] By definition, this makes `kirby` a shared memory multiprocessor.

2 Cache coherence

We determine whether cache coherence is enforced by `kirby`.

2.1 Requirements

Here, we work under the following definition of cache coherence.

Definition 1 *A processor enforces cache coherence if and only if*

1. *All writes are eventually visible to all other processors, and*
2. *The order in which writes appear in memory is the same for all processors.*

These necessary and sufficient conditions are *write-propagation* and *write-serialization*, respectively. [8]

2.2 Design

We test whether each condition holds separately, in the programs: `question1.c` and `question2.c`. As a general design note, compiler optimizations will be disabled for all C implementations of the pseudocode tests. This means that instruction reordering may only occur in hardware, as this is what we seek to investigate.

2.2.1 Propagation

Following the scientific method, we propose the following strategy for discovering the probability of the event H: "`kirby`'s multiprocessor enforces write-propagation".

1. We hold no prior belief as to whether write-propagation is enforced: $\mathcal{P}(H) = 0.5$.
2. We perform a multithreaded test under the hypothesis H, where a single thread performs a write to a memory location, and the other threads poll the same memory location for a change in value. The program stops successfully when all threads observe the changed value, or fails after a threshold time interval.
3. Record the outcome T, and repeat the above step for some number of iterations.
4. As a conservative estimate, we hold no belief as to the result of the outcome given that write-propagation isn't enforced: $\mathcal{P}(T|\bar{H}) = 0.5$; we compute $\mathcal{P}(T|H)$ as the proportion of successful tests.

5. We update $\mathcal{P}(H)$ to $\mathcal{P}(H|T)$ using Bayes' rule, and jump back to step 2 until $\mathcal{P}(H)$ passes a threshold probability value.

When all processors observe the changed value, we must slightly update the Bayesian prior towards the belief that the multiprocessor enforces write-propagation, since the probability that the multiprocessor doesn't enforce write-propagation decreases on each successful execution of the program. Indeed, if we lived in a world where write-propagation weren't enforced, the number of times the program would succeed before eventually failing could be modeled by a geometric distribution in the probability of a write always propagating to every other processor, which by definition would be less than 1. Conversely, when program fails, two possibilities present themselves. Either the time for the write to propagate would have surpassed the threshold value, or there exists a core which never would have observed the changed value. Since we are only interested in the latter case, we set a very large threshold time interval, such that the likelihood of the first case having occured is negligable, given that the program fails. Therefore, on failure, we must always tend towards the belief that the multiprocessor doesn't enforce write-propagation.

2.2.2 Propagation pseudocode

Since `kirby` supports one thread for each one of its six cores, we propose the following pseudocode for the test in step 2.

Algorithm 1: T_0

```
counter = 0;
x = 0;
MAKE_THREADS(1..5);
x = 1;
JOIN_THREADS(1..5);
while $counter \neq 5$ do
    if $TIME > THRESH$ then
        return $FAILURE$;
    end
end
return $SUCCESS$
```

Algorithm 2: $T_1 - T_5$

```
while $x \neq 1$ do
    if $TIME > THRESH$ then
        return;
    end
end
FETCH_&_ADD(counter,1);
return;
```

The main writer thread T_0 begins by resetting two variables in shared memory: `counter`, which will be atomically incremented by the polling threads T_1-T_5; and `x`, which will be later polled by T_1-T_5 in order to provide evidence for write-propagation. Then, the main thread spawns the five threads T_1-T_5, where it is assumed that they are distributed equally amongst the remaining cores. The new threads immediately begin to poll `x`.

If write-propagation were enforced, and `x`'s address were already in one of the cores' cachelines, then the cacheline would have been invalidated when T_0 wrote `x=0`. However, this relies on the assumption that the memory consistency model guarantees `W->W` program ordering, such that `MAKE_THREADS()` is issued after `x=0`. It is unknown at this point whether this guarantee is ensured. Thankfully, the overhead of creating a new thread (allocating memory, setting the instruction pointer to the correct instruction address, setting up interrupts) will make it so that the write `x=1` is likely to occur before the first read of `x!=0`, because the write `x=1` would have to sink past many instructions to be reordered with `x!=0`.

If write-propagation weren't enforced, then it could be that one of the reader thread's cachelines happens to contain `x=1`, and the line may not have been invalidated when the main thread wrote `x=0`. This means that the `counter` variable may reach 5 without write-propagation. Although we acknowledge this unlikely event, our strategy of repeating the experiment many times mitigates its impact on our results, given that an arbitrary read to `x` could possibly have returned any integer other than 1 with very high probability.

The main thread T_0 then waits for all other threads to finish: If a reader thread reads `x==1` before reaching the threshold time, then it atomically fetches and increments the counter variable, and closes; if the thread runs for too long, it closes without incrementing the counter. Finally, the main thread polls `counter` for the value 5, which is likely to be equivalent to the write `x=1` eventually becoming visible to all processors. Again, this relies either on the program ordering assumption `W->R` such that `counter!=1` isn't issued before `JOIN_THREADS()`, or that the implementation of `JOIN_THREADS()` introduces a memory fence before and after itself.

Note that the use of the atomic read-modify-write instruction `FETCH_&_ADD` may contribute directly to the belief that write-propagation is supported, because read-modify-write instructions are implemented using cache-line locking: the denial of cache protocol requests to the relevant cache line. Inspecting the assembly code with
`gcc -std=gnu99 -S question1.c -lpthread -O0`, we do in fact see the atomic instruction `lock addl $1, counter(&rip)`. This would therefore imply the existence of an invalidation-based cache-coherence protocol in `kirby`'s microprocessor architecture. However, a *high level* `FETCH_&_ADD` instruction may not in fact provide a low-level serialization garantee in general [9], so one needs to take care that `FETCH_&_ADD` isn't issued before `while(x!=1)`.

2.2.3 Propagation experiment

After 20 bayesian updates, with 500 test iterations each in order to determine $\mathcal{P}(T|H)$, we report a 99% probability of write propagation being enforced by the multiprocessor architecture. No fundamental changes to the program pseudocode were required in order to achieve this result. Thankfully, the gnu built-in function `__atomic_fetch_add` takes a parameter to specify the memory consistency model with respect to that instruction; we chose `__ATOMIC_RELEASE` in the hopes that a memory fence would set up to prevent `while(x!=1)` from sinking below the fetch-and-add instruction.[10] Interestingly, there is no memory fence instruction inserted into the assembly code; choosing `__ATOMIC_RELAXED` instead doesn't change the resulting behaviour at all. This may imply that the underlying memory consistency model doesn't allow `R->W` reordering, and the compiler somehow knows that no fence instructions would be needed, although more experiments will be required to confirm the result.

2.2.4 Write-serialization

We propose a similar strategy for discovering the probability of the event G: "`kirby`'s multiprocessor enforces write-serialization".

1. We hold no priors as to whether write-serialization is enforced: $\mathcal{P}(G) = 0.5$.
2. We perform a multithreaded test under the hypothesis G, where two threads concurrently write different values to the same memory location, then the other threads poll that memory location twice. If the polling threads read a different sequence of values from one another, then the program fails. Otherwise, the program succeeds after a threshold time interval.
3. Record the outcome T, and repeat the above step for some number of iterations.
4. As a conservative estimate, we hold no belief as to the result of the outcome given that write-serialization isn't enforced: $\mathcal{P}(T|\bar{G}) = 0.5$; we compute $\mathcal{P}(T|G)$ as the proportion of successful tests.
5. We update $\mathcal{P}(G)$ to $\mathcal{P}(G|T)$ using Bayes' rule, and jump back to step 2 until $\mathcal{P}(G)$ passes a threshold probability value.

Indeed, if it were the case that write-serialization were enforced, then by definition, all writes would be visible in the same order for every other thread. Then in the second step, the other threads' read operations would never result in an inconsistent order. If however, write-serialization were not enforced, then it may still be the case that the other threads' read operation result in an consistent order. Therefore we must repeat the experiment many times to verify the result. Conversely, when a single inconsistent ordering occurs, we must immediately throw out the belief that the multiprocessor enforces write-serialization.

2.2.5 Write-serialization pseudocode

We propose the following pseudocode for the test in step 2.

Algorithm 3: T_0

```
xs=[(0,0)..(0,0)];
order = 0;
x = 0;
MAKE_THREADS(1);
x = 1;
JOIN_THREADS(1);
obs = (0,0);
for $t$=0..3 do
    if $xs[t] = (1,2)$ or $xs[t] = (2,1)$ then
        if $obs = (0,0)$ then
            obs = xs[t];
        else if $obs \neq xs[t]$ then
            return $FAILURE$;
end
if $obs$ = (0,0) then
    return $INVALID$;
return $SUCCESS$
```

Algorithm 4: T_1

```
MAKE_THREADS(2..5);
x = 2;
JOIN_THREADS(2..5);
return;
```

Algorithm 5: $T_2 - T_5$

```
id = FETCH_&_ADD(order,1);
while $x \neq 1$ and $x \neq 2$ do
    nothing
end
r1 = x;
r2 = x;
xs[id] = (r1, r2);
return
```

The main thread T_0 begins by resetting variables in shared memory: `xs`, which records the order of reads from `x` for each reader thread $T_2 - T_5$; `id`, which allows each reader thread to record their results `(r1,r2)` in non-overlapping parts of the shared array `xs`; and `x`, which is written to by the writer threads T_0 and T_1 and subsequently read by the reader threads. Then, the main thread launches a thread with `MAKE_THREADS(1)` before attempting to write `x=1` concurrently with T_1's write `x=2`. Subsequently, with `JOIN_THREADS(1)`, the main thread waits for T_1 to wait for all $T_2 - T_5$ to write their observations to the shared memory array `xs`.

Finally, the main thread checks the shared memory array for the presence of both $(2,1)$ and $(1,2)$, which would immediately imply that write-serialization isn't enforced, returning $FAILURE$ if this is the case. Otherwise, if the shared memory array only contains identical pairs of $(1,1)$ and $(2,2)$, then the test contains no useful information and must be discarded with the $INVALID$ return value. The remaining cases: those where there there is at least one $(1,2)$ and no $(2,1)$ and vice-versa, constitute valid evidence towards the belief that write serialization is enforced.

2.2.6 Serialization experiment

The result of the experiment was initially inconclusive, as the reader threads would never read two different values of `x`. Indeed, the overhead of creating four new threads in T_1 made the write `x=2` occur much later than `x=1`. This means that the scenario of some reader thread reading 1 before 2 would be very unlikely, invalidating the experiment. Instead, we would like it that each ordering would be equally likely. To remedy this, a barrier statement was added before `x=1` in T_0, `x=2` in T_1, and `while(x!=1 && x!=2)` in $T_2 - T_5$. Threads were allowed to advanced only once all other threads were waiting at the barrier, which allowed the writes `x=1` and `x=2` to occur concurrently.

The downstream effect of this change may have led to another unexpected result due to the fact that it is unknown at this point what reorderings are permissible within a thread. Indeed, if `r1=x` were allowed to be reordered with `r2=x`, then perhaps it would seem as if one of $T_2 - T_5$ observed writes in a different order in shared memory than the other threads. To prevent this from happening, a full memory fence statement was introduced between `r1=x` and `r2=x`. This has the additional knock-on effect of increasing the likelihood of `r1` and `r2` containing different values, which increases the value of the experiment.

It remains to discuss whether a possible reordering of `r2=x` and `xs[id]=(r1,r2)` could have also invalidated the experiment. Indeed, if `xs[id]=(r1,r2)` were executed first in some thread among $T_2 - T_5$, then no information could be gleaned with respect to the order of reads to `x` in that thread, as `r2` would be recorded as 0. In practice, however, this never occured. Perhaps this is due to the yet unknown memory consistency model, or due to the fact that, once compiled, the two instructions become four loads and two stores, in which case a reordering may have been too unlikely given the simplicity of the program.

In any case, after 20 bayesian updates, with 500 test iterations each in order to determine $\mathcal{P}(T|G)$, we report a 100% probability of write-serialization being enforced by the multiprocessor architecture.

2.3 Conclusion

We cannot directly compute the joint probability of write-propagation and write-serialization being enforced, because the second experiment ending depends on `x` being eventually propagated to the reader threads. However, since no observed tests in the second experiments stalled forever, we may approximate the joint probability as $\mathcal{P}(H)\mathcal{P}(G)$ regardless, and conclude that cache coherence is enforced with probability 99%.

3 Memory consistency

We determine `kirby`'s memory consistency model.

3.1 Requirements

In the discussion about the write-propagation experiment, it was mentioned that it may be the case that stores are not reordered with older loads by the processor. Therefore, the memory consistency models to explore are those that don't relax `R->W` ordering. These are *Total Store Ordering (TSO)* and *Partial Store Ordering (PSO).*

Definition 2 *A processor enforces TSO if and only if* `W->R` *ordering is relaxed.*

Definition 3 *A processor enforces PSO if and only if* `W->R` *and* `W->W` *orderings are relaxed.[11]*

Besides instruction reordering, there are other dimensions along which a memory consistency model may vary, and there may not be a name for said model. Therefore, we will also investigate whether `kirby`'s consistency model guarantees write-atomicity, and what synchronization instructions are exposed to the programmer. There may also be an additional level of complexity regarding instruction reordering being allowed or disallowed depending on whether there exists a data dependency between the two instructions. For simplicity, we will only investigate this possibility while evaluating our results by comparison with the ground-truth.

3.2 Design

We test reorderings in `question3.c` and `alt3.c`, and write-atomicity in `question4.c`.

3.2.1 Program reordering.

We propose similar Bayesian strategies as before, where successful test results allow us to be slightly more confident that a reordering is disallowed, but where a single negative test result forces us to throw out the hypothesis that the reordering is disallowed. For brevity, we describe only step two of the strategy.

Write-to-read reordering design We perform a multithreaded test, where two threads concurrently write a value to a shared memory location, each, then they attempt to read the other's memory location. If both threads read the initial values at each memory location, then the test returns in failure, otherwise it is successful.

Indeed, under the hypothesis that writes are allowed to sink below reads, for example when the processor contains a FIFO write-buffer, then it may be that both writes aren't issued until after each thread reads each other's shared memory location, returning the initial values.[12] Any other outcome is compatible with the hypothesis that writes are not allowed to sink below reads. Therefore, should we observe both initial values, we must immediate throw out the hypothesis that write-to-read reordering is disallowed. Conversely, observing any other outcome should slightly increase our belief in the hypothesis, following Bayes' Rule.

Write-to-read reordering pseudocode We propose the following pseudocode for the test.

Algorithm 6: T_0

```
x = 0;
y = 0;
out = [0,0];
MAKE_THREADS(1..2);
JOIN_THREADS(1..2);
if $out[0] == 0$ and $out[1] == 0$ then
    return FAILURE
return SUCCESS
```

Algorithm 7: T_1

```
barrier();
x = 1;
out[0] = y;
return;
```

Algorithm 8: T_2

```
barrier();
y = 1;
out[1] = x;
return;
```

The main thread resets the shared memory x, y, and out before launching the competing threads T_1 and T_2, that attempt to set x and y, respectively. If two write-to-read reorderings occured: one between storing 1 in x and loading the value of y to a processor register, and the other between storing 1 in y and loading the value of x to a processor register, then the registers in both processors would contain the value 0. These would then be moved back to the shared memory array out, regardless of when x=1 and y=1 are finally issued. The first thread finally attempts to catch the result out==[0,0] after joining the two threads back.

It remains to discuss the possible effects of other reorderings on the values to be found in out. Indeed, if R->W reordering of the compiled assembly instructions from out[0]=y and out[1]=x were allowed, then it may be that an unknown register value could be written to out[0] before the register takes on the value of y. If such reorderings were to occur in both threads, then it may be that the test results in failure, and we would interpret the cause as a W->R reordering. Therefore, this test is not sufficient to determine whether W->R reorderings are the sole cause of a result such as out==[0,0], and must be corroborated with the R->W reordering test.

Write-to-write reordering design We perform a multithreaded test, where one thread writes two different values to the same shared memory location, and a second thread reads the memory location once, jumps a fence, and reads the memory location again, recording the read values. If the second value appears to have been read before the first, the test returns in failure, otherwise it is successful.

Indeed, under the hypothesis that writes are allowed to sink below writes, for example when the processor contains a non-FIFO write-buffer, then it may be that the second write instruction is issued before the first. [12] In this case, the read instructions will return the second value before the first. Furthermore, write-to-write reordering is the only explanation for the outcome because the read instructions are forced to be issued in order thanks to the full memory fence instruction. Therefore, should we observe the values out-of-order, then we must immediately throw out the hypothesis that write-to-write reordering is disallowed. Conversely, observing any other outcome should slightly increase our belief in the hypothesis, following Bayes' Rule.

Write-to-write reordering pseudocode We propose the following pseudocode for the test.

Algorithm 9: T_0

```
x = 0;
out = [0,0];
MAKE_THREADS(1..2);
JOIN_THREADS(1..2);
if $out[0] == 2$ and $out[1] == 1$ then
    return $FAILURE$
return $SUCCESS$
```

Algorithm 10: T_1

```
barrier();
x = 1;
x = 2;
return;
```

Algorithm 11: T_2

```
barrier();
out[0] = x;
mfence;
out[1] = x;
return;
```

The main thread resets the shared memory `x`, `y`, and `out` before launching the writer thread T_1 and the reader thread T_2. If a write-to-write reordering of `x=1` with `x=2` were to occur, then there exists an interleaving of instructions such that `out[0]==2` and `out[1]==1`: when both threads take turns issuing memory instructions on the shared memory bus. After joining the threads back, the main thread attempts to catch the result `out==[2,1]`, and returns in failure.

It remains to discuss whether other types of instruction reorderings could result in a negative test result as well. Here, the only other applicable reordering could be `R->W` reordering. Indeed, as in the write-to-read test, it may be that the processor register contains the value `r=2` before loading the value of `x`. Thus, a reordered store instruction `mov out[0], r` may end up leading to a failed test. Finally, we cannot say that `W->W` reorderings are the sole cause of a result such as `out==[2,1]`, and must be corroborated with the `R->W` reordering test.

Read-to-write reordering design We perform a simple singlethreaded test, where the value of one variable is assigned to another. If the final value of the variable doesn't change after the assignment, then the test returns in failure, otherwise it is successful.

Indeed, under the hypothesis that reads are allowed to sink below writes, then the assembly instruction loading the value of the assigning variable to a processor register may be reordered with the instruction storing the value in the assigned variable. In this case, the assigned variable receives whatever value was stored in the register previously. If this value is not equal to the assigning variable, then we must immediate throw out the hypothesis that read-to-write reordering is disallowed. Conversely, observing any other outcome should slightly increase our belief in the hypothesis, following Bayes' Rule.

Read-to-write reordering pseudocode We propose the following pseudocode for the test.

```
Algorithm 12: $T_0$
x = 0;
y = 1;
mfence;
x = y;
mfence;
if $x == 0$ then
    return $FAILURE$
return $SUCCESS$
```

No other memory instruction reorderings may affect the result of the output, as these are prevented by `mfence` instructions. Indeed, if `W->R` reordering were allowed, and the first `mfence` instruction were absent, then it could be that loading `y` as the first part of the assignment `x = y` could occur before writing `y=1`, resulting in a final value of `x == 0`.

Read-to-read reordering design We perform a multithreaded test, where one thread writes a value to a shared memory location, jumps a fence, and then writes a different value to the same shared memory location. A second thread issues two reads to the shared memory location, recording the read values. If the second value appears to have been written before the first, the test returns in failure, otherwise it is successful.This may be allowed in hardware through dynamic scheduling.[12] Discussion of this reordering requires special attention.

Read-to-read reordering pseudocode We propose the following pseudocode for the test.

```
Algorithm 13: $T_0$
x = 0;
out = [0,0];
MAKE_THREADS(1..2);
JOIN_THREADS(1..2);
if $out[0] == 2$ and $out[1] == 1$ then
    return $FAILURE$
return $SUCCESS$
```

```
Algorithm 14: $T_1$
barrier();
x = 1;
mfence;
x = 2;
return;
```

```
Algorithm 15: $T_2$
barrier();
out[0] = x;
out[1] = x;
return;
```

In this case, however, special care must be taken in order to create a valid test where read-to-read reorderings would produce failing results. Indeed, our initial implementation of algorithms 14 and 15 in the above pseudocode yielded the following assembly code.

Algorithm 16: T_1

```
...
movl $1, x(%rip);
mfence;
movl $2, x(%rip);
...
ret;
```

Algorithm 17: T_2

```
...
movl x(%rip), %eax;
movl %eax, out(%rip);
movl x(%rip), %eax;
movl %eax, 4+out(%rip);
...
ret;
```

Here, even if we assume both `R->R` and `W->R` reorderings were allowed, there would be no sequence of instruction reorderings and interleavings which would result in a failed test. Therefore the test is invalid. The root of the problem is the reuse of the register `eax`. Indeed, the two load instructions here are completely identical, so if the second load instruction floats up and is issued before the first load, the last value written to `x` is the one that will be written to both elements of the array `out`. On the other hand, if a different intermediate register were used for the second load and store, then a reordering resulting in unexpected output is possible, again supposing that `W->R` ordering is also relaxed. This alternative program is in `alt3.c`.

Algorithm 18: T_1

```
...
movl $1, x(%rip);
mfence;
movl $2, x(%rip);
...
ret;
```

Algorithm 19: T_2

```
...
movl x(%rip), %eax;
movl %eax, out(%rip);
movl x(%rip), %ebx;
movl %ebx, 4+out(%rip);
...
ret;
```

3.2.2 Write-atomicity

We use the following working definition of write-atomicity.

Definition 4 *A processor enforces write-atomicity if and only if there is a single total order of all memory operations.[11]*

Again, a Bayesian strategy will be followed, where a successful test result allows to be more confident that write-atomicity is ensured, but negative test results force us to conclude that write-atomicity is not ensured.

Write-atomicity design We perform a multithreaded test, where two writer threads write a value to a shared memory location, each. Meanwhile, two reader threads attempt to read the shared memory locations, each in a different order. If it is the case that the writes appear to have been performed in a different order depending on which reader thread's perspective is taken, then writes are not atomic, and the hypothesis must be thrown out. Otherwise, we slightly move towards the belief that writes are atomic.

Write-atomicity pseudocode We propose the following pseudocode for the test.

Algorithm 20: T_0
```
x = 0; y = 0;
order = 0;
out = [0,0];
MAKE_THREADS(1..4);
JOIN_THREADS(1..4);
if $out[0] == 2$ and $out[1] == 1$ then
    return FAILURE
return SUCCESS
```

Algorithm 21: T_1
```
barrier();
x = 1;
return;
```

Algorithm 22: T_2
```
barrier();
y = 1;
return;
```

Algorithm 23: T_3
```
id = FETCH_&_ADD(order, 1);
barrier();
r1 = x;
r2 = y;
out[id] = (r1<<1)+r2;
return;
```

Algorithm 24: T_4
```
id = FETCH_&_ADD(order, 1);
barrier();
r1 = y;
r2 = x;
out[id] = (r1<<1)+r2;
return;
```

3.3 Experiment

After 20 Bayes updates with 500 iterations each, we report the following conclusive results. First, `R->W` reordering is disallowed with 99% probability, which corroborates the validity of the tests for `W->W` and `W->R` reordering, which are disallowed with probability 99% probability and 0% probability, respectively. Finally, `R->R` reordering is also disallowed with 99% probability. Furthermore, write-atomicity is also enforced with 99% probability. Only the read-to-read experiment was redesigned, although this was only because the initial naïve test was invalid due to a compiler particularity, as previously explained.

3.4 Results.

From our experiments, we deduce that the memory consistency model is very likely to be TSO. As we have seen, the processor also supports a number of memory fence instructions. It remains to compare this result with the true memory consistency model of the system, in order to evaluate the quality of our tests in post. The Intel® 64 Architecture Memory Ordering white paper informs us that, indeed, only `W->R` program ordering is relaxed, but not when the write and read instructions are to the same location. This reveals an inadequacy in our test. Furthermore, the memory consistency model further specifies reordering behaviour with respect to locked instructions. For example, "In a multiprocessor system, locked instructions have a total order", and "Loads and stores are not reordered with locked instructions".

References

[1] C. Qian, Z. Karel, C. Heiko. *LSCPU.* (2021) https://www.man7.org/linux/man-pages/man1/lscpu.1.html. Retrieved 12th March 2022.

[2] S. Wasson. *64-bit computing in theory and practice.* (2005). https://techreport.com/review/8131/64-bit-computing-in-theory-and-practice/. Retrieved 12th March 2022.

[3] Intel. *Intel® Core^{TM} i5-8500 Processor.* https://www.intel.com/content/www/us/en/products/sku/129939/intel-core-i58500-processor-9m-cache-up-to-4-10-ghz/specifications.html. Retrieved 12th March 2022.

[4] Oracle. *SSE4.1 Instructions.* x86 Assembly Language Reference Manual. https://docs.oracle.com/cd/E36784_01/html/E36859/gntbu.html. Retrieved 13th March 2022.

[5] Oracle. *SSE4.2 Instructions.* x86 Assembly Language Reference Manual. https://docs.oracle.com/cd/E36784_01/html/E36859/gntaj.html. Retrieved 13th March 2022.

[6] Oracle. *AVX2 Instructions.* x86 Assembly Language Reference Manual. https://docs.oracle.com/cd/E36784_01/html/E36859/gntae.html. Retrieved 13th March 2022.

[7] Microsoft. *NUMA Support.* (2021). https://docs.microsoft.com/en-us/windows/win32/procthread/numa-support. Retrieved 13th March 2022.

[8] V. Nagarajan. (2022). *Lecture 4: Coherence Protocols.* [Lecture Notes]

[9] curiousguy. *Stack overflow discussion.* https://stackoverflow.com/questions/51086258/is-stdatomicfetch-add-a-serializing-operation-on-x86-64. Retrieved 14th March 2022.

[10] gnu. *Built-in functions for memory model aware atomic operations.* https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/_005f_005fatomic-Builtins.html. Retrieved 25th March 2022.

[11] V. Nagarajan. (2022). *Lecture 5: Memory Consistency.* [Lecture Notes]

[12] S. Biswas. (2020). *CS636: Memory Consistency Models.* [Lecture Notes]. https://www.cse.iitk.ac.in/users/swarnendu/courses/spring2021-cs636/memory-models.pdf

[13] Intel. *Intel® 64 Architecture Memory Ordering White Paper.* Revision 1.0. (2007). https://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf