Atomics in Arm: Are they ruining your performance?

Ricardo Jesus and Michèle Weiland

EPCC, The University of Edinburgh

ISC 2022 AHUG Workshop

Outline

- Introduction
- The problem
- Benchmarking compare-and-swap operations
- Experiments and observations
- Conclusions

Introduction

- **Atomics** are indivisible operations guaranteed to run to completion without interference from other threads
- **Compare-and-swap (CAS)** is one of the most common and general atomic operations
  - Allows a thread to atomically compare a memory location with a given value and, if the values match, to write another value in that memory location
  - Used to implement locks, semaphores, concurrent data structures, etc
- **In AArch64** CAS operations can be implemented via LL-SC pairs of instructions or explicit CAS instructions (LSE, Armv8.1+)

The performance of LL-SC/CAS approaches can be fundamentally different on different micro-architectures

The PI_ATOMIC kernel

- The RAJA Performance Suite (RAJAPerf)[1] is a benchmark suite consisting of loop-based kernels extracted from HPC applications, other benchmark suites, etc
- PI_ATOMIC is one of RAJAPerf's kernels. It computes π via

$$\boxed{\pi = 4\arctan(1) = 4\int_0^1 \frac{1}{1+x^2}\,dx} \tag{1}$$

- Multiple implementations such as sequential, OpenMP, and RAJA
- Useful to assess the performance of OpenMP atomics in high contention

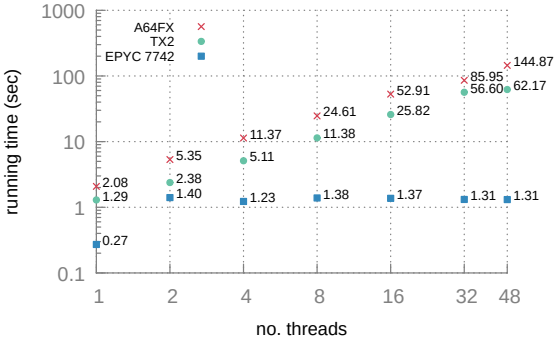
```
double *pi = ...;
#pragma omp parallel for
for(int i = 0; i < N; ++i) {
  double x = ((double)i + 0.5) * dx;
  #pragma omp atomic
  *pi += dx / (1.0 + x * x);
}
pi *= 4.0;
```

[1] https://github.com/LLNL/RAJAPerf

The problem: Poor performance on PI_ATOMIC

- We were using RAJAPerf to compare A64FX, ThunderX2, and AMD EPYC 7742 processors
- The PI_ATOMIC kernel stood out as the single worst-performing kernel on Arm
- **Fundamental difference** in the behaviours of the EPYC and the Arm-based CPUs
  - On the EPYC 7742, the running time **stays constant** for multi-threaded runs
  - On the A64FX/TX2, the running time **grows linearly** with the no. threads

- 50 repetitions of 1,000,000 iterations each (default parameters)
- Optimisation flags shown later



Inspecting PI_ATOMIC's assembly

- Assembly[1] is identical for x86 and AArch64 targets
- Move value from GP to FP register, operate, move result back and attempt CAS
- Unnecessary jump present in both targets

```
/*      C/C++      */
for(...) {
  #pragma omp atomic
  *x += 1.0;
}
```

```
/*                ASM (x86)                 */
.L2:
  vmovq   %rdx, %xmm2       // move x to XMM
  mov     %rdx, %rax          // create copy
  vaddsd  %xmm0, %xmm2, %xmm1       // x+1.0
  vmovq   %xmm1, %rcx    // move result back
  lock cmpxchg %rcx, (%rdi)  // attempt CAS

  jne     .L4               // retry if fail
  ...
.L4:
  mov     %rax, %rdx
  jmp     .L2
```

```
/*              ASM (A64)                   */
.L2:
  fmov    d1, x2  // move to FP register
  mov     x3, x2            // create copy
  fadd    d1, d0, d1     // compute x+1.0
  fmov    x4, d1       // move result back
  casal   x3, x4, [x0]     // attempt CAS
  cmp     x2, x3
  bne     .L4            // retry if fail
  ...
.L4:
  mov     x2, x3
  b       .L2
```

[1] Compiled with GCC 11.2 and `-O3 -fopenmp -march=znver2/-mcpu=a64fx`.

Benchmarking compare-and-swap operations

- Threads concurrently increment a counter `iters` times each using CAS operations
- Measure: **average time** and **average number of attempts** per increment
- A few parameters:
  - counter type: `uint64_t`, `double`
  - strategy: CAS, LL-SC
  - memory order: relaxed, acquire, release, acquire-release
- Kernels are implemented in assembly (next slide)

```cpp
#include <cstdint>

template <typename counter_t>
uint64_t kernel(uint64_t iters, counter_t* mem,
                bool weak, int memorder)
{
  uint64_t attempts = 0;
  do {
    counter_t expected, desired;
    do {
      attempts++;
      expected = *mem;
      desired = expected + 1;
    } while(!__atomic_compare_exchange(
                    mem, &expected, &desired,
                    weak, memorder, memorder));
  } while(--iters);
  return attempts;
}
```

Benchmark kernels

- From left to right: `double` kernel for x86, `uint64_t` kernel for AArch64 with CAS strategy, and `double` kernel for AArch64 with LL-SC strategy

```
kernel_dbl_x86:
  xor %r8d, %r8d
  vmovsd one, %xmm0
loop_dbl_x86:
  mov (%rsi), %rax
try_dbl_x86:
  inc %r8
  vmovq %rax, %xmm1
  vaddsd %xmm0, %xmm1, %xmm2
  vmovq %xmm2, %rcx
  lock cmpxchg %rcx, (%rsi)
  jne try_dbl_x86
  dec %rdi
  jne loop_dbl_x86
  mov %r8, %rax
  ret
one:
  .double 1.0
```

```
lea 0x1(%rax), %rcx
for uint64_t kernel
```

```
kernel_u64_cas:
  mov x9, 0

loop_u64_cas:
  ldr x2, [x1]
try_u64_cas:
  add x9, x9, 1

  add x3, x2, 1

  mov x4, x2
  cas x2, x3, [x1]
  cmp x2, x4
  bne try_u64_cas
  subs x0, x0, 1
  bne loop_u64_cas
  mov x0, x9
  ret
```

```
kernel_dbl_ldxr_stxr:
  mov x9, 0
  fmov d0, 1.0
loop_dbl_ldxr_stxr:
  ldxr x2, [x1]

  add x9, x9, 1
  fmov d1, x2
  fadd d1, d0, d1
  fmov x3, d1

  stxr w4, x3, [x1]

  cbnz w4, loop_dbl_ldxr_stxr
  subs x0, x0, 1
  bne loop_dbl_ldxr_stxr
  mov x0, x9
  ret
```

Experiments and observations

- All measurements taken in exclusive mode
- Each thread runs for 100,000 iterations
- Threads pinned to cores and placed close to each other (`OMP_PLACES=cores` and `OMP_PROC_BIND=close`)

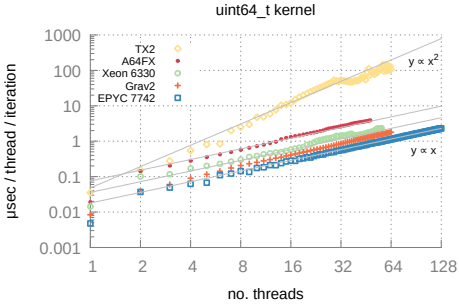
System	Processor	No. cores	Compiler	Flags
Fulhame	ThunderX2	2×32	GCC 10.1.0	-O3 -fopenmp -mcpu=native
Isambard 2	A64FX	48	GCC 11.1.0	-O3 -fopenmp -mcpu=native
C6g	Graviton2	64	GCC 10.3.1	-O3 -fopenmp -mcpu=native
C7g	Graviton3	64	GCC 10.3.1	-O3 -fopenmp -mcpu=native
ARCHER2	EPYC 7742	2×64	GCC 11.2.0	-O3 -fopenmp -march=native
ICX	Xeon Gold 6330	2×28	GCC 11.2.0	-O3 -fopenmp -march=native

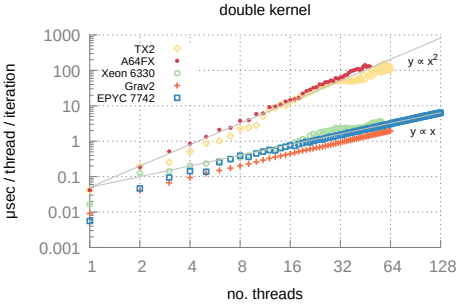
- Only show results for `cas` and `ldxr+stxr` are shown (i.e. relaxed memory order)
- Same trends with other memory orders (correlation coefficient below)

memory order	A64FX		Graviton2		ThunderX2	
	uint64_t	double	uint64_t	double	uint64_t	double
casa	0.9999	0.9993	1.0000	1.0000	0.9998	0.9998
casl	1.0000	0.9989	1.0000	1.0000	0.9999	0.9998
casal	0.9999	0.9989	1.0000	1.0000	0.9999	0.9997
ldaxr+stxr	1.0000	1.0000	0.9636	0.9559	0.9997	0.9994
ldxr+stlxr	0.9999	1.0000	0.9621	0.9380	0.9989	0.9987
ldaxr+stlxr	0.9999	1.0000	0.9638	0.9633	0.9983	0.9983

Performance of CAS instructions (I)

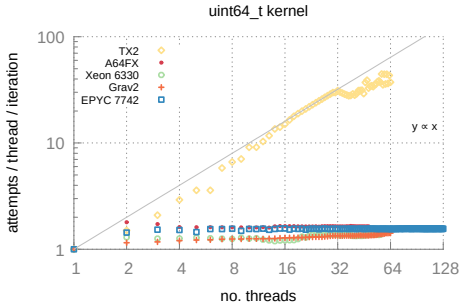
- Two types of behaviour in terms of average time per increment
  - **Linear scaling** on Graviton2, x86 CPUs, and A64FX (on the `uint64_t` kernel)
  - **Quadratic scaling** on ThunderX2 and A64FX (on the `double` kernel)
- Behaviour exhibited by the `double` kernel matches that seen in `PI_ATOMIC`

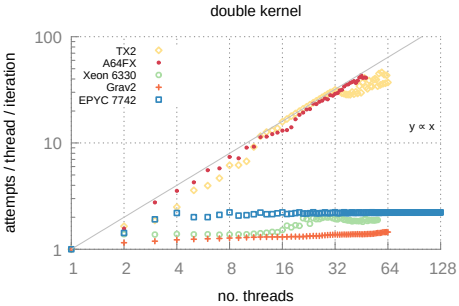




Performance of CAS instructions (II)

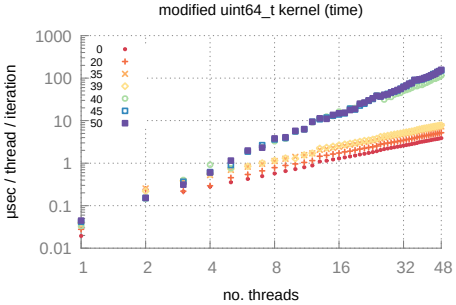
- Analogous trends in terms of average number of attempts per increment
  - **Constant** on Graviton2, x86 CPUs, and A64FX (on the `uint64_t` kernel)
  - **Linear scaling** on ThunderX2 and A64FX (on the `double` kernel)
- Explains the slowdown
- Suggests a lack of built-in backoff mechanisms in the A64FX and ThunderX2

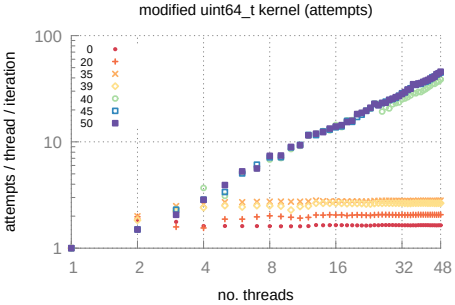




Effect of loop-body size on the A64FX

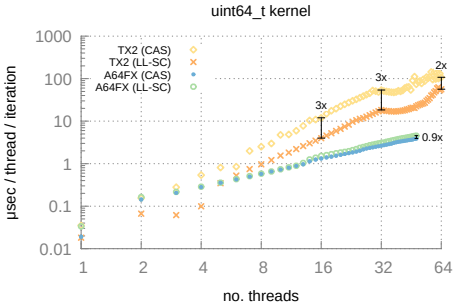
- Tested by inserting `add` instructions to the body of the integer kernel ("additional latency of loop body")
- **Linear behaviour** if number of instructions inserted ≤ 39
- Sharp transition to **quadratic behaviour** if ≥ 40 instructions inserted
  $\Rightarrow$ Matches `double` kernel

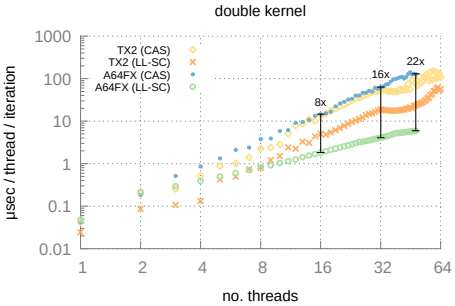




Performance of LL-SC on A64FX and ThunderX2

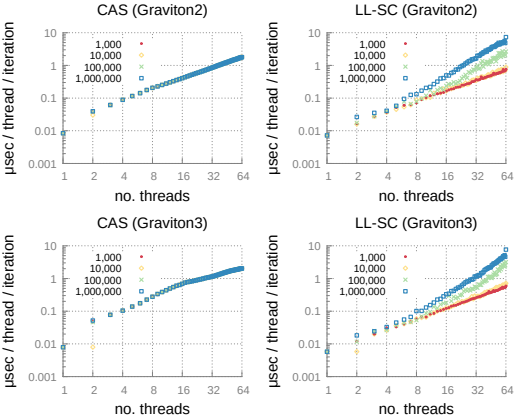
- Flat 2–3x speedup on ThunderX2 over CAS
- The **double LL-SC kernel exhibits linear scaling on the A64FX**
  - For high thread counts ⇒ speedup of more than 20x vs. CAS





Performance of LL-SC on Graviton2 and Graviton3

- With LL-SC, average time per increment <u>increases</u> as the number of increments performed increases
- This contrasts with CAS, which is **invariant** to the number of increments per thread
- For a relatively small no. increments (<100,000), LL-SC approach can still bring significant speedups (2–3x) over CAS



Conclusions

- The newer LSE CAS instructions do not perform optimally on either the A64FX or ThunderX2
  - They can cause very significant slowdows whereby the average time per successful CAS operation **quadruples** as the no. threads attempting the operation doubles
  - On the A64FX this behaviour arises when the time to prepare the CAS operation rises above a critical value around the 40 clock cycles mark
- Despite being older, the LL-SC instructions can provide significant speedups when compared to the newer CAS instructions on all Arm-based CPUs we tested
  - Flat 2–3x speedups on ThunderX2 and on Graviton2/3 (for low iteration counts)
  - On the A64FX, **LL-SC reduces** the quadratic scaling seen with CAS instructions **to the expected linear scaling**
    $\Rightarrow$ For high thread counts, this results in speedups of over 20x
- We are currently working with vendors to understand the reasons leading to these behaviours and to develop optimised atomic primitives for their microarchitectures

Questions?