CS4223 Project 2
Cache Simulator

Justin Lu, Leon Thomm

November 18, 2022

This report summarizes the specifications, main implementation aspects, and simulation findings of a MESI and Dragon cache coherence simulator that was built as a project for *Multi-Core Architectures* at the *National University of Singapore*. The full code can be found on GitHub.

1 Specifications

processor behavior

- A processor executes instructions on every clock tick; if there's memory access it calls the cache and idles until it receives a response. (*)

bus behavior

- A cache communicates with other caches through the bus, and the bus is "owned" (locked) by a cache until it completely finished his transactions corresponding to one transition in the respective state diagram.
- Bus locking must be fair between the caches (request queue-style).
- The time for sending a bus transaction (address and transaction code) to other caches takes 2 cycles - just as long as sending only an address.
- All queued bus signals are always transmitted before a cache can lock the bus again - this ensures consistency.

cache behavior

- Any cache can always immediately respond to bus requests (*). If the cache gets asked to deliver a block which is pending eviction, the block is assumed to be invalid and cannot be delivered anymore.
- While the penalty for loading a cache line from memory is 100 cycles, the MESI cache implements Illinois and thus first tries to get the line from another cache, which adds to these 100 cycles if no one has it.
- The system always prefers cache-to-cache transfer.

A snooping cache can receive bus signals triggering three kinds of actions (and any combinations) 1. the cache responding / possibly providing some data 2. a transition in the data cache 3. bus signals as a result of (2)

We assume that action (1) can always be executed immediately, (notice the cache currently doesn't own the bus, because the cache who asked is owning it). However, the actual data cache might be busy resolving a request from the processor, so actions (2) and (3) must be buffered and handled later. Since the incoming bus signal should only have an effect on *future* operations in the cache (and not on the one currently running), it is safe for the cache to handle the bus signal immediately *after* completing the current processor request.

further system and timing specs

- If cache C_0 tries to cache a block b, the action of asking other caches whether they have b, and the action of sending the bus signal resulting from the following transition in C_0 are two separate steps which both require bus communication; i.e. we ignore the fact that the other caches could theoretically infer some of these bus transactions from C_0's asking.
- Arbitration policy: the processor/cache with lower id is preferred; e.g. if P_0 and P_1 want to write to the same block in the same cycle, P_0 will proceed and P_1 will have to adapt if necessary
- If multiple caches could deliver a line, there is no additional time needed to select one - the selection algorithm is expected to terminate in the same cycle.
- The memory is word-addressable not, byte-addressable.
- Memory is generally only updated on eviction/flushing. Dragon never explicitly updates memory, the memory is assumed to snoop *BusUpd* signals.
- The caches do not flush their dirty lines at the end of the simulation.
- The simulation ends when all processors entered state *Done* and and the bus is idling (finished sending any pending signals).

(*) as stated in the task description

key insights

1. The bus can only do one thing at a time, it serializes all requests.
2. All memory transfer goes through the bus.
3. There are two types of signals a cache can receive: *processor signals* and *bus signals.*
4. Each of these can be further divided into
   - those that can be answered right away
   - those that require communication with other caches or possibly memory

$\Longrightarrow$ Let C_0 be a cache processing a processor request on address a in block b requiring communication over the bus, and assume C_0 is owning the bus now and performing its operations. Let C_1 be another cache attempting a state transition on b as well. By owning the bus, C_0 ensures the following

- **either** C_1 is blocked from its transition (if it requires sending **and receiving** on bus) until C_0 is done and the system never enters an invalid state because the bus distributes the necessary signals before C_1 can lock it
- **or** C_1's transition does not require bidirectional communication on the bus (i.e. only sending a bus signal, e.g. MESI Invalid PrWrMiss) which might put the system into a temporarily invalid state (e.g. b in Modified in C_0 and C_1) but this inconsistency will be serially resolved once C_0 is done, for the same reason as above

From the perspective of some cache C_i, the bus can be:

- owned: C_i can send bus requests
- foreign owned: C_i can respond to incoming bus requests - owner is responsible of preventing conflicts
- busy: the bus is not owned by anyone but busy sending signals
- free: C_i can try to acquire the bus lock in order to start sending requests

2 Implementation

2.a Approach 1 - Python Prototype

We first wrote a prototype in Python to implement most of the basic concepts and gain an understanding of the challenges. This prototype works mainly by determining the number of cycles any operation will take as early as possible, and optimizes by simulating multiple cycles at once when it is safe to do so. The source code can be found on github in the *cachesim-py* directory. Notice that this implementation does not yet fully follow the above specifications.

2.b Approach 2 - High Performance in Rust

In attempt to come up with a more scalable design, we spent *a lot* of time implementing a completely different solution in Rust. We chose Rust mainly for performance reasons and its design principles. Generally, a Rust implementation without any *unsafe* code compiling is an extremely good indicator for the quality and stability of the written code. It can still fail, but when it does it's much more clear why and where, and most of typical sources for errors are already eliminated by the rules enforced by the compiler. Implementing the cache simulator in Rust We saw as a challenge and opportunity to try get more familiar with the language.

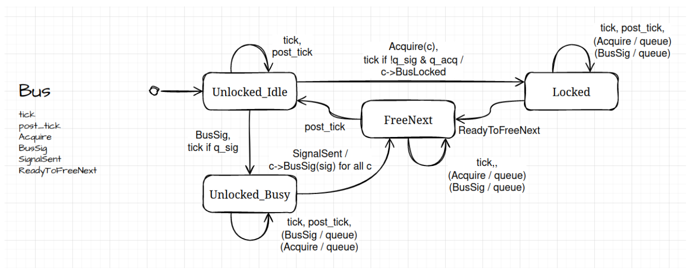
2.b.1 order of execution

Any simulator will have to step through a series of simulated cycles. A new cycles's rising edge (referred to as *tick* from now on) reaches processor, caches, and the bus (though this can slightly differ between implementations). A single-threaded simulator will inherently have an order of ticking (e.g. P_0, P_1, C_0, C_1, *bus*) causes a strong temptation to build the simulator with an exact order assumed. This, however, causes highly complex internal dependencies, makes therefore debugging really hard and the system generally un-scalable. The desire to have an implementation that is not dependent on the order of ticking and does not have any complex dependencies is what mainly inspired the design in the Rust implementation. After various failed attempts, we came up with a design that is based on to aspects: encoding every component as a *finite state machine*, and only indirect communication between components using a *delayed message queue* system.

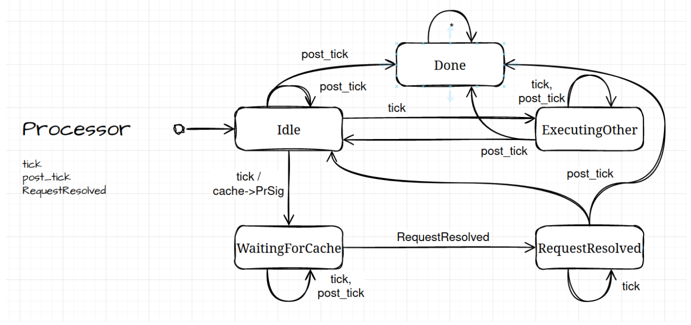
2.b.2 state machines

The trade-off of encoding all internal components as finite state machines without direct communication as opposed to different components calling each other's methods all over the place comes down to simplicity vs. code size. The Rust implementation is just below 2K lines of relatively dense code, but the modular design and strict separation of responsibilities makes it much more scalable - it could probably be reasonably easily extended by similar cache coherence protocols.

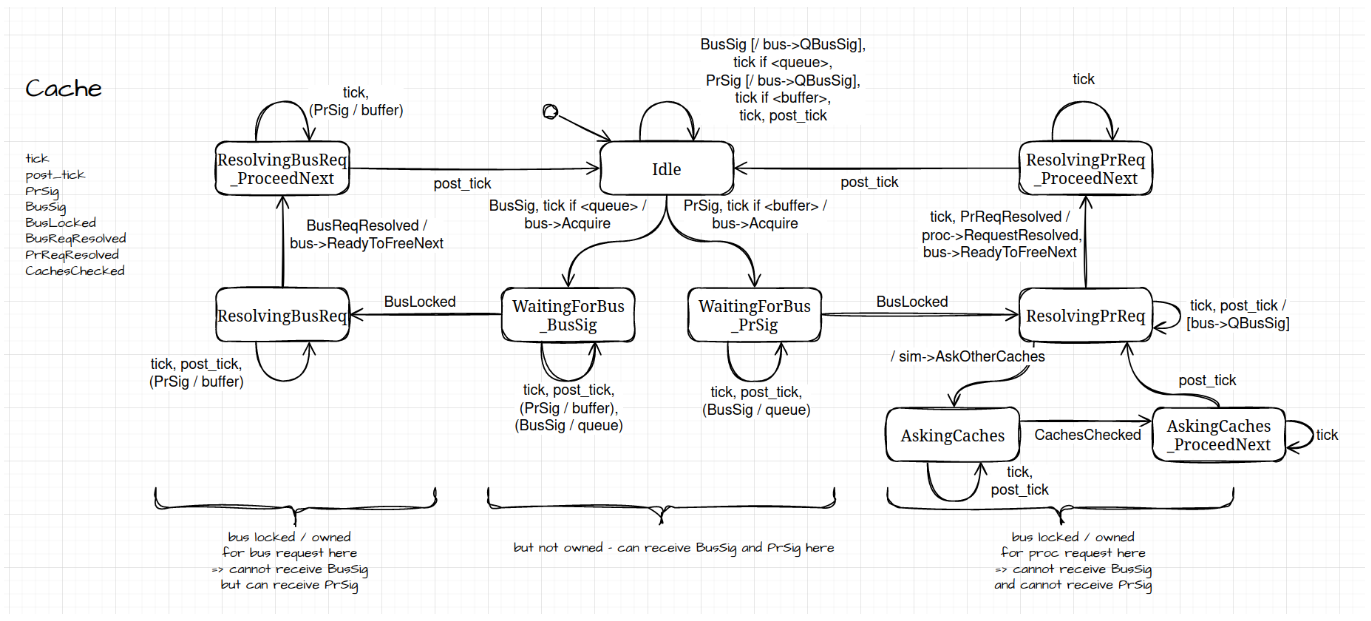
Important for the state diagrams are the key insights stated above, and specifically the two kinds of operations (requiring bus communication or not). Also, without an order of ticking specified, it is important when a cycle starts and when it ends (i.e. falling clock edge). On the rising edge components start to make progress, send signals to other components, process certain incoming signals, and make state transitions. The falling edge (*post_tick*) is used to make sure the component enters a new state for the next cycle, *after* all signals between components regarding the current cycle have been sent and processed (so no signals are allowed to be sent during *post_tick*). This greatly simplifies the design under parameterized timing constraints.


Bus
tick
post_tick
Acquire
BusSig
SignalSent
ReadyToFreeNext
Unlocked_Idle
Locked
FreeNext
Unlocked_Busy
tick, post_tick
Acquire(c), tick if !q_sig & q_acq / c->BusLocked
tick, post_tick, (Acquire / queue) (BusSig / queue)
post_tick
ReadyToFreeNext
BusSig, tick if q_sig
SignalSent / c->BusSig(sig) for all c
tick,, (Acquire / queue) (BusSig / queue)
tick, post_tick, (BusSig / queue) (Acquire / queue)

Bus



Processors


Cache
tick
post_tick
PrSig
BusSig
BusLocked
BusReqResolved
PrReqResolved
CachesChecked
ResolvingBusReq _ProceedNext
Idle
ResolvingPrReq _ProceedNext
ResolvingBusReq
WaitingForBus _BusSig
WaitingForBus _PrSig
ResolvingPrReq
AskingCaches
AskingCaches _ProceedNext
tick, (PrSig / buffer)
BusSig [/ bus->QBusSig], tick if <queue>, PrSig [/ bus->QBusSig], tick if <buffer>, tick, post_tick
tick
post_tick
post_tick
BusSig, tick if <queue> / bus->Acquire
PrSig, tick if <buffer> / bus->Acquire
BusReqResolved / bus->ReadyToFreeNext
tick, PrReqResolved / proc->RequestResolved, bus->ReadyToFreeNext
BusLocked
BusLocked
tick, post_tick / [bus->QBusSig]
tick, post_tick, (PrSig / buffer)
tick, post_tick, (PrSig / buffer), (BusSig / queue)
tick, post_tick, (BusSig / queue)
/ sim->AskOtherCaches
post_tick
CachesChecked
tick
tick, post_tick
bus locked / owned for bus request here => cannot receive BusSig but can receive PrSig
but not owned - can receive BusSig and PrSig here
bus locked / owned for proc request here => cannot receive BusSig and cannot receive PrSig

Caches

2.b.3 delayed queue

Building a system of communicating FSMs requires some communication channel. For this purpose, we implemented a new data-structure which is essentially a message queue with a discrete delay. Say a cache needs to send something to its processor, but this process is supposed to take two cycles. With a delayed queue interface, this is amazingly simple:

```
fn some_function() {
    <-- snip -->
    let send_proc = |&self_: Self, msg: ProcMsg,
    delay: i32| {
        self.tx.send(DelayedMsg {
            t: delay,
            msg: Msg::CacheToProc(proc_id,msg),
        }).unwrap();
    };
    // tx provides the interface for sending to
    the queue
    send_proc(self, ProcMsg::RequestResolved, 0);
}
```

The first fully functional implementation was based on a priority queue using binary heap, prioritized by the timestamps/delays of the messages. In an attempt to speed things up, we later re-implemented the same functionality based on a ring-buffered queue which is slower in the worst case, but faster on average than the binary heap. The ring buffer implementation turned out to be faster, especially when enabling the highest level of optimizations in *rustc*.

The amazing support for algebraic data types and pattern matching in Rust keeps the encoding of the state transitions simple. Below you can see how this is done by the example of a part of the message handler of the bus.

Bus MsgHandler

```
impl MsgHandler<BusMsg> for Bus<'_> {
    fn handle_msg(&mut self, msg: BusMsg) {
        match self.state {
            <-- see next -->
        }
    }
}
```

MsgHandler is the *trait* (comparable to an *interface* in Java) and *handle_msg* will be invoked by the simulator when there's a message in the queue with the current cycle as timestamp, addressed at the bus. Based on the bus's current state and the message, it can then trigger state transitions:

Bus MsgHandler continued

```
<-- snip -->
BusState::Unlocked_Idle => {
    match msg {
        BusMsg::Tick => {
            // transition
        },
        BusMsg::PostTick => (),
        BusMsg::Acquire(cache_id) => {
            // transition
        },
        BusMsg::BusSig(cache_id, sig) => {
            // transition
        },
        _ => panic!("Invalid bus state"),
    }
},
BusState::Unlocked_Busy => ...
<-- snip -->
```

2.b.4 optimizations and accessibility

The rich type system of Rust already enables quite substantial optimizations at compile time, which can be supported by using generic types etc. The re-implementation of the delayed queue for speed was mentioned in the previous section. It is also worth noting that turning on optimizations in the rust compiler (compiling for *release*, with level 3 optimizations) had a substantial impact on performance.

Further code-based optimizations focused on preventing unnecessary ticks of components in states where a *tick* message cannot cause any progress, which is fairly easy when all components are simply finite state machines and all communication goes through a central message queue.

Simulator Optimizations

```
match procs[proc_id as usize].state {
    // some optional optimizations
    ProcState::WaitingForCache | ProcState::Done
    => continue,
    _ => send_msg(Msg::ToProc(proc_id, Tick)),
};
```

Furthermore, it is worth noting that this system is - by design - easily parallelizable using multi-threading. However, because of the central message queue, all threads should run on the same core (in order to share caches) when running the simulation.

We also integrated some improvements over a naive MESI implementation, which include the Illinois behavior of preferring cache-to-cache transfer when possible, and the ability to continue and unlock the bus even when bus signals are pending transmission (the bus itself holds a queue for these).

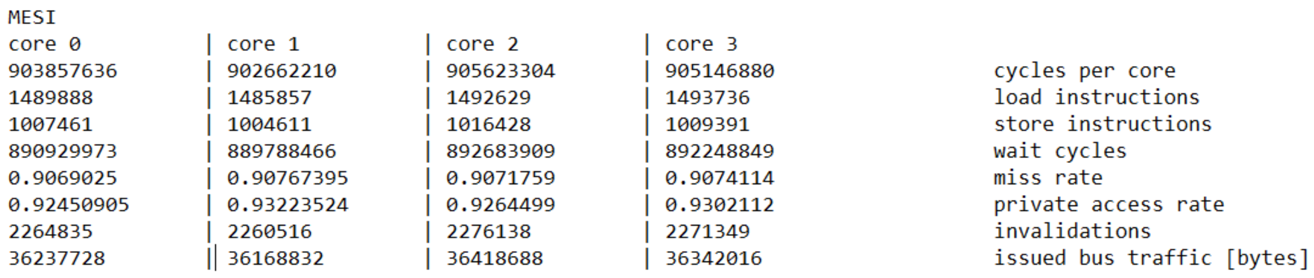
We also used the opportunity to do some WebAssembly (*WASM*) benchmarking. We could compile the simulator to WASM architectures without any code changes. The execution time of the WASI version was only about twice that of the native one, which is quite impressive.

3 Analysis

3.a MESI

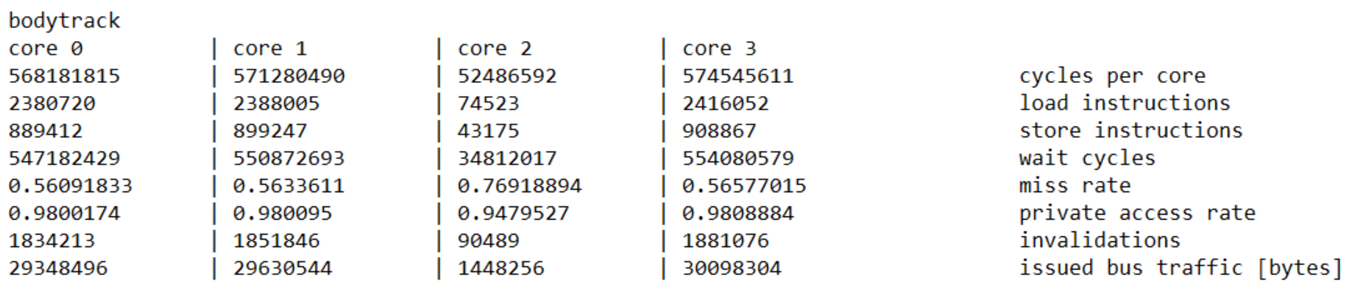
Benchmark results for MESI protocol (1 KiB Cache, direct map, block size 16):

Blackscholes:



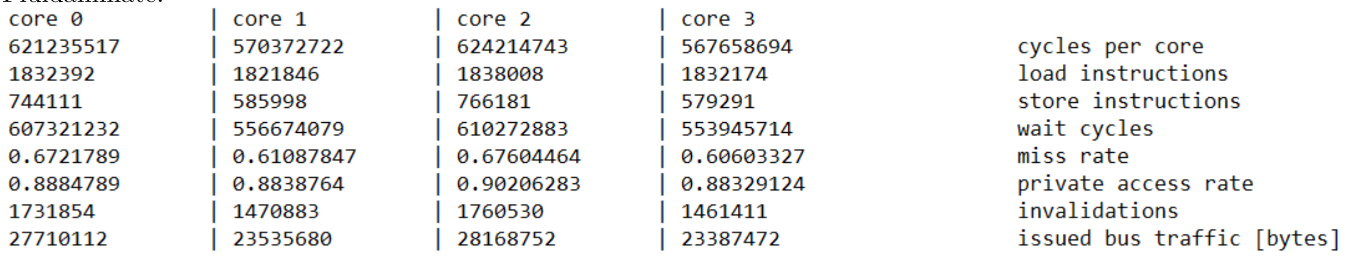
```
MESI
core 0          | core 1          | core 2          | core 3
903857636       | 902662210       | 905623304       | 905146880               cycles per core
1489888         | 1485857         | 1492629         | 1493736                 load instructions
1007461         | 1004611         | 1016428         | 1009391                 store instructions
890929973       | 889788466       | 892683909       | 892248849               wait cycles
0.9069025       | 0.90767395      | 0.9071759       | 0.9074114               miss rate
0.92450905      | 0.93223524      | 0.9264499       | 0.9302112               private access rate
2264835         | 2260516         | 2276138         | 2271349                 invalidations
36237728        || 36168832       | 36418688        | 36342016                issued bus traffic [bytes]
```

Bodytrack:



```
bodytrack
core 0       | core 1       | core 2       | core 3
568181815    | 571280490    | 52486592     | 574545611              cycles per core
2380720      | 2388005      | 74523        | 2416052                load instructions
889412       | 899247       | 43175        | 908867                 store instructions
547182429    | 550872693    | 34812017     | 554080579              wait cycles
0.56091833   | 0.5633611    | 0.76918894   | 0.56577015             miss rate
0.9800174    | 0.980095     | 0.9479527    | 0.9808884              private access rate
1834213      | 1851846      | 90489        | 1881076                invalidations
29348496     | 29630544     | 1448256      | 30098304               issued bus traffic [bytes]
```

Fluidanimate:

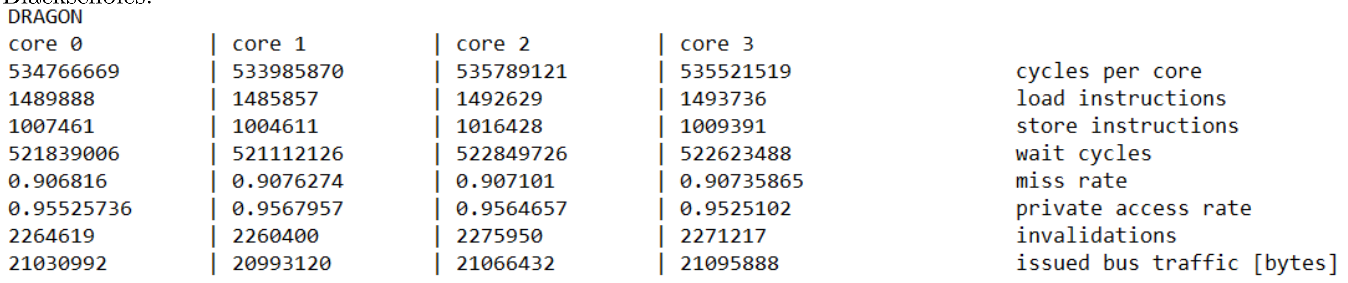


```
core 0       | core 1       | core 2       | core 3
621235517    | 570372722    | 624214743    | 567658694              cycles per core
1832392      | 1821846      | 1838008      | 1832174                load instructions
744111       | 585998       | 766181       | 579291                 store instructions
607321232    | 556674079    | 610272883    | 553945714              wait cycles
0.6721789    | 0.61087847   | 0.67604464   | 0.60603327             miss rate
0.8884789    | 0.8838764    | 0.90206283   | 0.88329124             private access rate
1731854      | 1470883      | 1760530      | 1461411                invalidations
27710112     | 23535680     | 28168752     | 23387472               issued bus traffic [bytes]
```

3.b Dragon

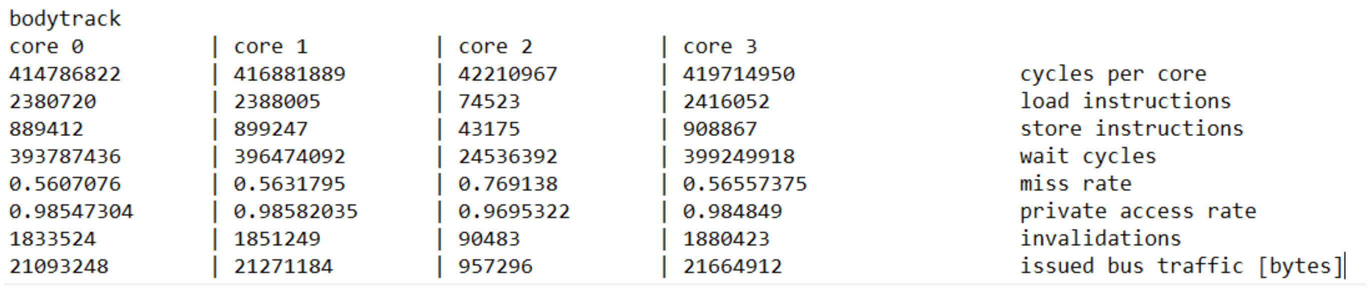
Benchmark results for Dragon protocol (1 KiB Cache, direct map, block size 16):

Blackscholes:



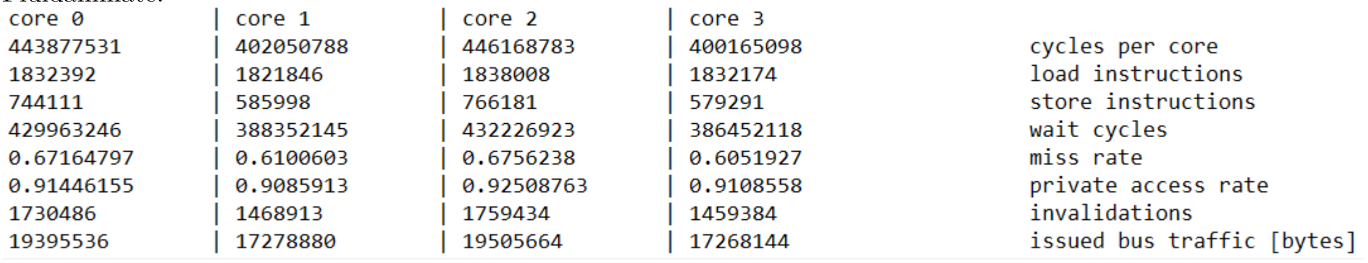
```
DRAGON
core 0       | core 1       | core 2       | core 3
534766669    | 533985870    | 535789121    | 535521519              cycles per core
1489888      | 1485857      | 1492629      | 1493736                load instructions
1007461      | 1004611      | 1016428      | 1009391                store instructions
521839006    | 521112126    | 522849726    | 522623488              wait cycles
0.906816     | 0.9076274    | 0.907101     | 0.90735865             miss rate
0.95525736   | 0.9567957    | 0.9564657    | 0.9525102              private access rate
2264619      | 2260400      | 2275950      | 2271217                invalidations
21030992     | 20993120     | 21066432     | 21095888               issued bus traffic [bytes]
```

Bodytrack:



```
bodytrack
core 0       | core 1       | core 2       | core 3
414786822    | 416881889    | 42210967     | 419714950              cycles per core
2380720      | 2388005      | 74523        | 2416052                load instructions
889412       | 899247       | 43175        | 908867                 store instructions
393787436    | 396474092    | 24536392     | 399249918              wait cycles
0.5607076    | 0.5631795    | 0.769138     | 0.56557375             miss rate
0.98547304   | 0.98582035   | 0.9695322    | 0.984849               private access rate
1833524      | 1851249      | 90483        | 1880423                invalidations
21093248     | 21271184     | 957296       | 21664912               issued bus traffic [bytes]
```

Fluidanimate:



```
core 0       | core 1       | core 2       | core 3
443877531    | 402050788    | 446168783    | 400165098              cycles per core
1832392      | 1821846      | 1838008      | 1832174                load instructions
744111       | 585998       | 766181       | 579291                 store instructions
429963246    | 388352145    | 432226923    | 386452118              wait cycles
0.67164797   | 0.6100603    | 0.6756238    | 0.6051927              miss rate
0.91446155   | 0.9085913    | 0.92508763   | 0.9108558              private access rate
1730486      | 1468913      | 1759434      | 1459384                invalidations
19395536     | 17278880     | 19505664     | 17268144               issued bus traffic [bytes]
```

Overall, we notice very similar results when comparing the MESI and Dragon protocols for all three of the benchmarks, running with a 1K direct mapped cache with block size of 16.

From running the three different benchmarks with a 1K direct mapped cache with a block size of 16, we observe that on average, the MESI protocol has nearly double the cycles per core compared to the DRAGON protocol. Both the MESI and Dragon protocols have similar miss rates, and similar proportions of their cycles spent on wait cycles. Furthermore, both the MESI and Dragon protocols have similar miss rates and private access rates. Likewise, both experience similar amounts of invalidations. However, the MESI protocol has a higher amount of bus traffic compared to the Dragon protocol.

The increase in the number of execution cycles can be explained by the high cost of invalidations in the MESI protocol. With the MESI protocol, when a cache block is written to, all other caches must discard their versions of the block, whereas in the Dragon protocol, there are bus update signals that will update the data in other caches. Another factor for the increased cycles is due to our implementation of the Illinois MESI protocol, where caches can share data between one another. Cache misses require a search of all other processors' caches to see if they have the block being searched for.

The increase in the volume of bus traffic in bytes in the MESI protocol can be explained by the fact that we implemented the Illinois-MESI protocol. With this protocol, we can share data between caches, so the bus will frequently transfer cache blocks between the caches of different CPUs, which adds a substantial amount of overhead in traffic.

From observing the results of the Dragon and Illinois-MESI protocol across these three different benchmarks, the Dragon protocol would be preferred for each benchmark because of the lower cycle counts, as well as the lower volume of bus traffic. Using the Dragon protocol would lead to a more efficient execution with respect to these three benchmarks.

3.c Fluidanimate on MESI

cache size	**assiciativity**	**block size**	**avg. miss rate**
512	1	16	0.74105008
512	2	16	0.733073148
512	3	16	0.74515101
512	4	16	0.72208518
1024	1	16	0.6412578
1024	2	16	0.60319607
1024	3	16	0.594221075
1024	4	16	0.611503995
2048	1	16	0.5248601
2048	2	16	0.491306805
2048	3	16	0.467433068
2048	4	16	0.47193364
4096	1	16	0.446913793
4096	2	16	0.39347735
4096	3	16	0.37981965
4096	4	16	0.372726315

For both the MESI and Dragon protocols, as we increase the size of our cache, the average miss rate will go down. Likewise, if we increase the associativity of our cache, the average miss rate will also go down. When we increase the size of our cache, there are more spaces for blocks to be placed in. When we increase the associativity of our cache, a block of memory can be placed in more possible locations. Both of these contribute to more items being stored in our cache, reducing the miss rate.