Final Project Report

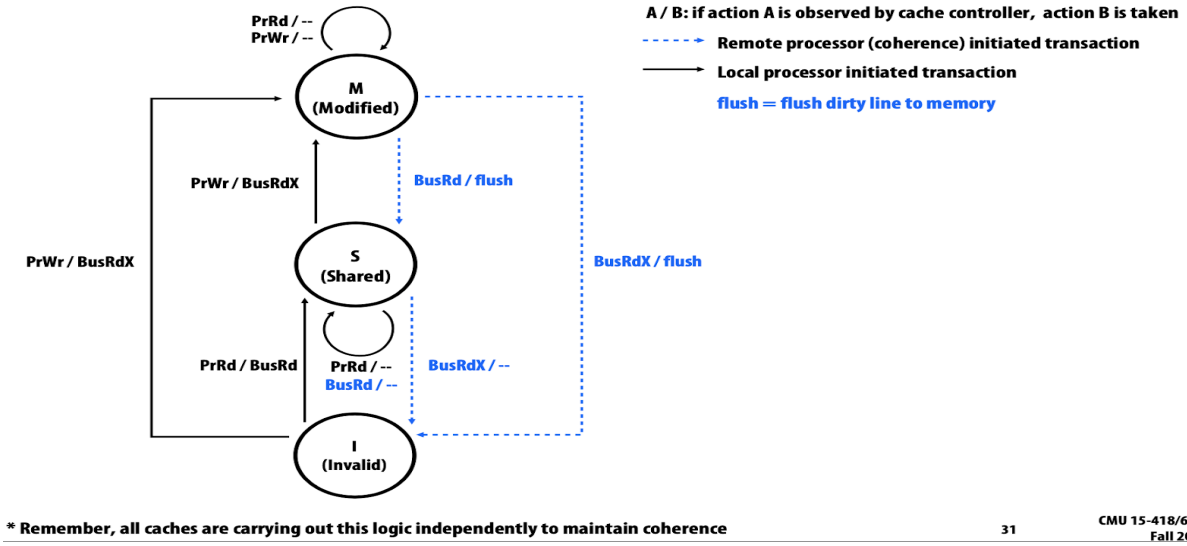
Tanay Asija(tasija), Xuan Peng(xuanpeng)

Summary

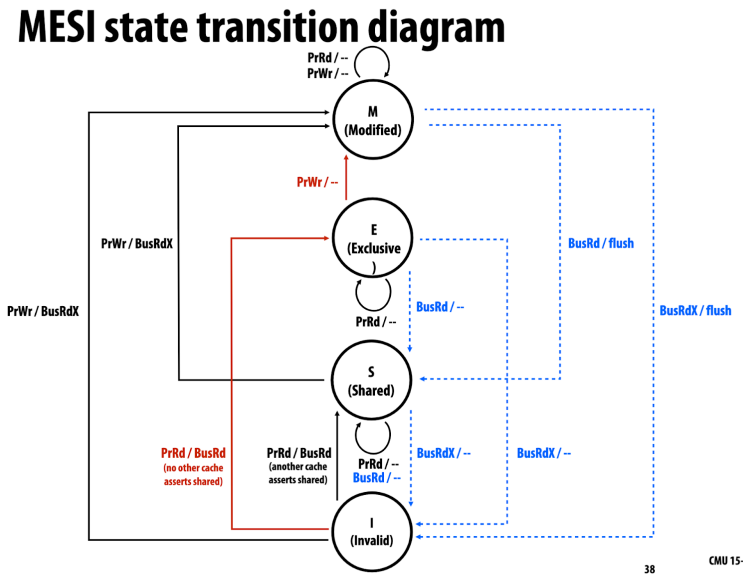
We have designed and implemented a multi-core non-blocking snoop-based trace-driven cache coherence simulator using the software primitives from the Structural Simulation Toolkit to allow for insightful performance studies by configuring the simulator with different cache block size, total cache size, various replacement policies, cache coherence protocols and arbitration policies in order to derive valuable results into the performance of different shared memory parallel programs with various access patterns, locality and sharing. The multithreaded parallel traces for the loads and stores made by each thread are collected using the PIN tool by executing the programs on the x86 Linux based GHC machines.

Background

As covered in the course lectures, we studied multiple cache coherence protocols such as **MSI**, and **MESI**. MSI is the most basic cache coherence protocol that uses **Modified, Shared**, and **Invalid** to describe the state of a cache line, its state diagram is as follows:

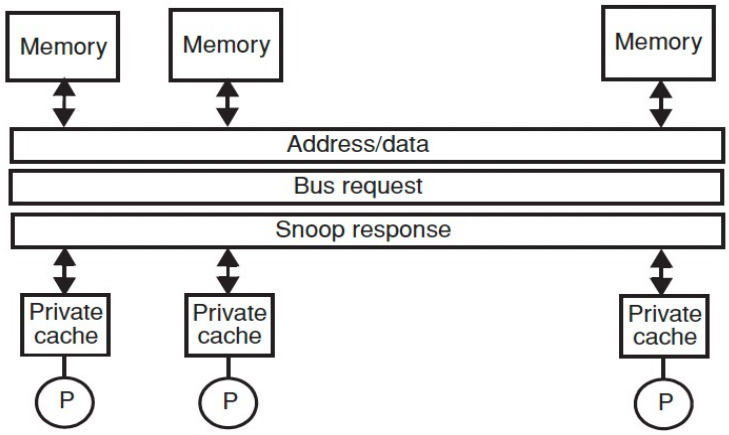


For MESI, one more “**Exclusive**” state is added in order to **reduce the coherence traffic** on the bus. And the state diagram for MESI is as follows:



By theory, the performance ranking of those protocols would be **MSI < MESI**. We want to verify this conclusion within our simulator and study the degree to which it helps in saving memory and bus traffic and consequently effective bandwidth utilization.

We also studied a couple of different cache implementation styles namely snooping-based cache implementation and directory-based cache implementation. We will implement the cache system in the **snoop-based** fashion. Below is the design for a snoop-based cache system:



Further, we get to know that there are some existing simulation frameworks that have been used and proven to be successful in a lot of use cases, such as **SST(Structural Simulation Toolkit)** and ZSim. We chose SST as the framework upon which we build our cache simulator. SST consists of 2 major components, namely SST-Core and SST-Elements. The SST-Elements are a set of computer hardware simulators that are officially-published or initially published by 3rd but then adopted by the official. Since we're building our own cache simulator as a new component, we won't be reusing the components in the SST-Components repository. And the SST-Core is a set of APIs and Specifications that provides the infrastructure that every component needs in order for them to function properly (e.g., clock, event communication channel, event handler, etc), and bring different components together to run coherently.

We're excited about applying what we have learned in the lectures into building a self-sufficient cache coherence simulator built upon SST, and then study the effect of the following on the performance of the multi-core cache system.

1. Different sizes of a cache line
2. Total cache size
3. Replacement policies
4. Cache coherence protocols,
5. Different access patterns, sharing and locality

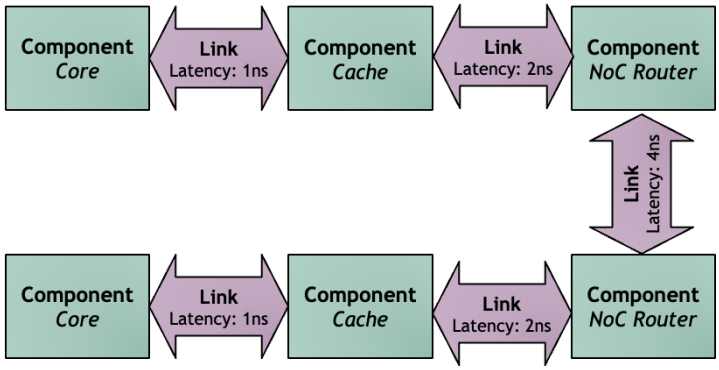
Approach

We have implemented our multi-core non-blocking snoop-based cache coherence simulator using the core APIs provided by the Structural Simulation Toolkit or SST. SST is an open source publicly available simulation framework that allows to define and build various hardware blocks using software primitives by defining the abstraction of a **Component** and a **Link.** We arrived at

the decision of using SST for our project after discussing it with Professor Skarlatos. Below is an example of using the SST primitives by defining multiple **Components,** each of which is an independent hardware block operating in parallel with all the other hardware blocks.

The components are connected together using **Links**, which are SST primitives of a message queue with a backend implementation built on top of **OpenMPI**.

All SST components and development is done using **C++** with a flexible linkage of the defined component library into a **Python** based frontend to pass configuration parameters for carrying out performance studies.



We decided that we would build our own version of a cache coherence simulator on top of the core SST primitives to allow for -

1. First hand experience of using SST core primitives in order to get familiar with SST backend
2. Opportunity to extend the SST-Elements library by allowing for a trace-driven program collected from execution of real parallel programs instead of artificial generators or micro-benchmarks
3. Complete control over the abstractions and features we want to define, implement and study

In order to gather traces we used the PIN tool and extended the memory address tracer to allow for recording traces from a multithreaded shared memory parallel program. The output of the PIN tool is a single file containing interleaved memory loads and stores from different threads.

This file is parsed and split into a different file for each different thread which would directly be fed into the load store generator attached to the **Cache Component** that we design using SST. Here is an example of one of the trace files fed to a load-store generator.

```
threadId: 1, 0x7f36bd6c39fe: W 0x7f36bc872ef8
threadId: 1, 0x7f36bd631854: W 0x7f36bc872ef0
threadId: 1, 0x7f36bd631855: W 0x7f36bc872ee8
threadId: 1, 0x7f36bd63185d: W 0x7f36bc872e68
```

We describe our implementation below. We have 5 major components

1. Load store generator
2. Cache
3. Interconnect
4. Arbiter
5. Memory

The load store generator is fed the per-thread trace and issues the appropriate requests to the cache. The **caches are non-blocking** allowing for the load store generator to issue multiple outstanding requests up to a maximum limit, which can be configured in the simulator. The cache looks up the address in its cache and issues the appropriate request to the arbiter, in order to grab control of the bus and get back the snoop response if required. Multiple outstanding requests if aliasing to the cache line are queued up and do not generate additional bus traffic. However this depends on the load and store order. A load followed by a load will not generate any additional traffic, however a load followed by a store which is also a miss on the same cache line will generate an exclusive bus read request.

The interconnect is a generic, extremely simple broadcast based network, with all details of routing, flow control and topology abstracted out. We have an additional component for memory added for timing purposes in order to issue the request to memory and get back the response if

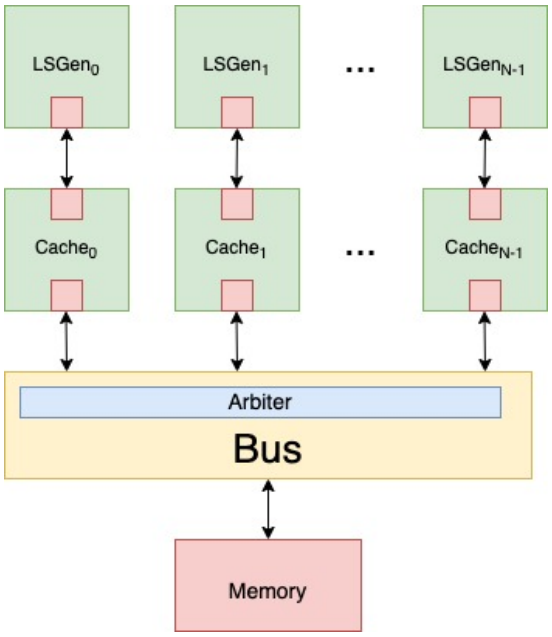
none of the caches have the requested cache line. We provide a highly configurable simulator in order to carry out meaningful performance studies by tuning the following -

1. Number of load store generators (up to 8 but easily extendable)
2. Cache block size
    a. This indicates the the size in bytes for a single cache block
    b. It is critical in deciding how much data is brought into the cache on accessing a virtual address
    c. This is also crucial in deciding how much data could possibly be shared among different threads running on different cores directly affecting the number of invalidations due to false or true sharing and subsequently the artifactual communication
3. Total cache size
4. Associativity
    a. The number of cache lines in a single set
5. Replacement policy
    a. The replacement policy followed during evicting a cache block in a set
    b. Can be one of
        i. round robin (RR)
        ii. least recently used (LRU)
        iii. Most recently used (MRU)
6. Cache coherence protocol
    a. Coherence protocol to be simulated by the caches. One of
        i. MSI
        ii. MESI
7. Arbitration policy
    a. Arbitration policy followed in granting caches access to the bus. One of

i. Round robin (RR)

ii. First in First Out (FIFO)

The simulator has the following components, connected to each other as follows -



Result and Analysis

We report the following statistics at the end of every simulation -

1. Per cache hits
2. Per cache misses
3. Per cache hit rate
4. Per cache miss rate
5. Per cache invalidations due to coherency protocols
6. Per cache evictions
7. Total memory traffic
8. Total number of requests issued on the bus
9. Total number of snoop responses sent to the bus

We now describe the programs we run our performance studies with those workloads that represent different access patterns:

1. Global sum
    a. Threads compute the sum of an array with 128 values and threads are assigned chunks of array in blocked fashion
    b. All threads add the values to a single global variable
2. No-padding sum
    a. Threads compute the sum of an array with 128 values and threads are assigned chunks of array in blocked fashion
    b. All threads add the values that are assigned to its own entry in an array
3. Padding sum
    a. Threads compute the sum of an array with 128 values and threads are assigned chunks of array in blocked fashion
    b. All threads add the values that are assigned to its own entry in an array but the entries in the array are padded so that each entry falls in its own cache line
4. Ocean 16
    a. We run the computation with an access pattern similar to ocean where the value for a given index in a 2D plane depends on that value and all its neighbors
    b. The size of the 2D array is chosen to be 16 x 16
5. Ocean 32
    a. Same as Ocean 16 but the size of the 2D array is chosen to be 32 x 32
6. Ocean 64
    a. the size of the 2D array is chosen to be 64 x 64
7. Random
    a. Access a random element in the array and increment it

b. The random indices are generated before hand and partitioned across the threads

Note: All programs were run with 4 threads but our simulator is agnostic of the number of threads and cores and can be easily extended to multiple threads by simply changing the configuration provided in the test Python file.

We performed many studies on the relationship between metrics and different cache configurations & protocols. We list some interesting and important ones below:

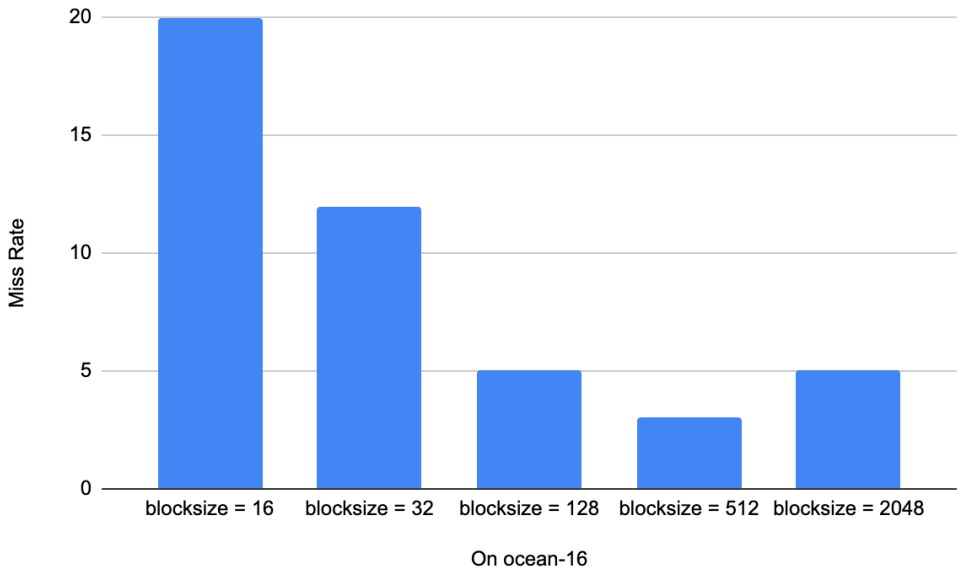
The comparison of bus traffic between MSI and MESI:

The implication behind this plot is straightforward. Recall what we learned on the lecture, the introduction of the “Exclusive” state in MESI protocol, is to avoid unnecessary bus traffic. Originally without the E state, for an cache line from I -> M (a ready-modify-write operation), it

will first send a BusRd/BusRdX on the bus so that the state becomes S. And then, for S to become M, it will send another BusUpgr on the bus. So there are altogether 2 bus broadcasts. But with the added E, for a ready-modify-write operation, we can just send a BusRdX on the bus and then bring the line to an E state. Since the E states signifies exclusive ownership of a cache line, later when we do the write operation, we don't need to send the BusUpgr anymore. So the total number of bus events is 1.

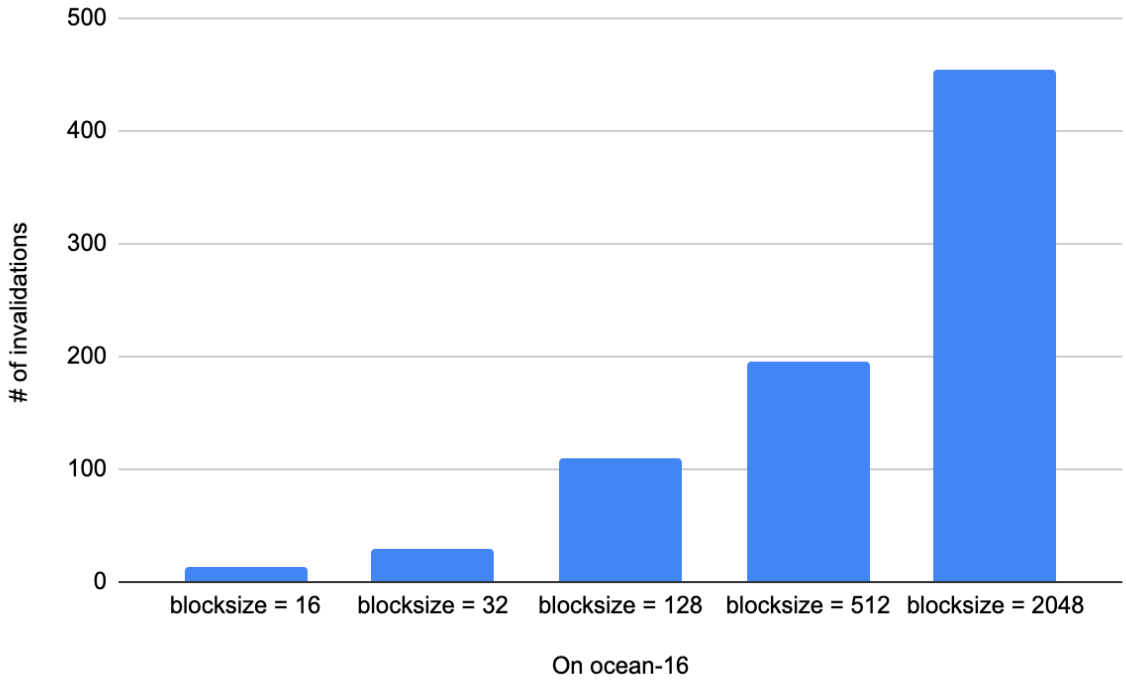
Depending on the access pattern, the bus traffic dropped by a different percentage. For example, in the padding sum workload, the bus traffic almost remains the same, because since each thread writes to its own cache line, there is very little sharing. This leads to the traffic being almost similar across MSI and MESI. However, the traffic in MESI is still slightly lower than MSI even for this case.

Cache miss rate with respect to the size of a cache line (The "U" shape we learned from written assignment):



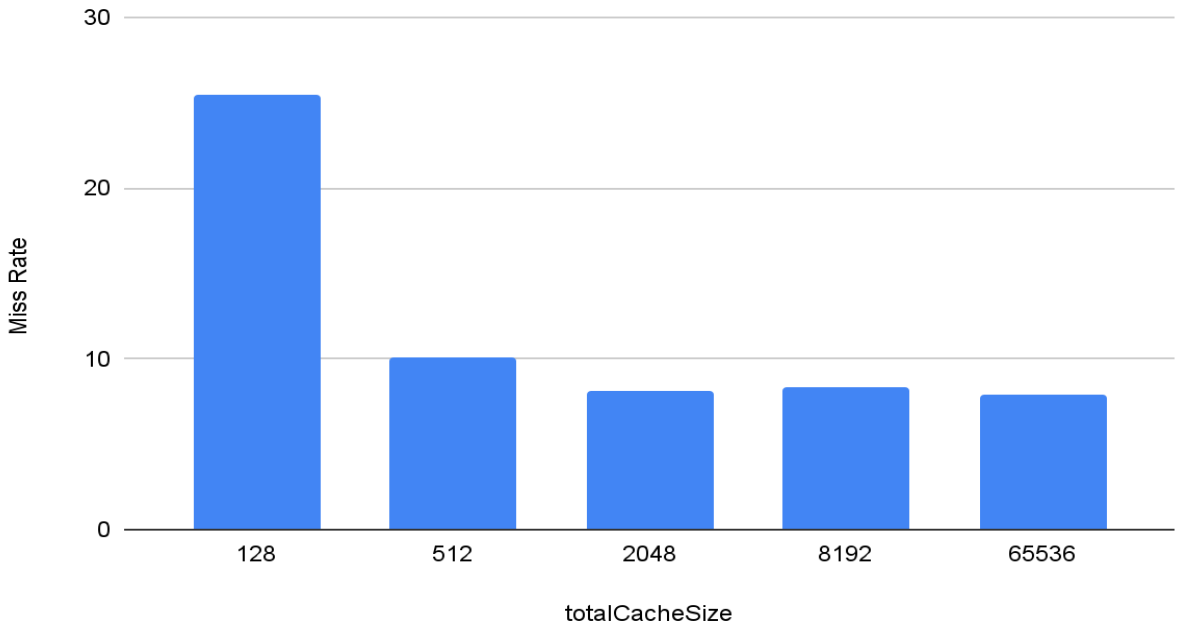
When the block size is very small, cold miss will be high. As we increased the block size, cold miss gradually dropped. However, when the block size is too high, false sharing could happen, which makes the miss rate increase again. The plot we get from the experiment proves this point.

Number of invalidations with respect to the block size:



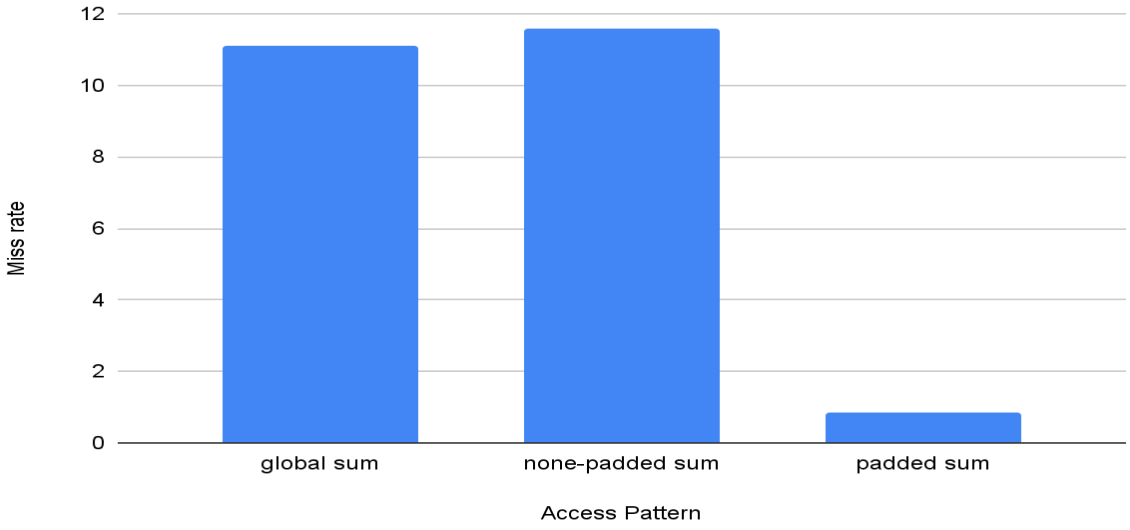
This plot is pretty interesting since it shows the number of invalidations that take place as the cache block size is increased with the rest of the cache parameters kept fixed. It is quite intuitively understood which validates the correctness of our implementation. As the cache block size increases the probability of a cache line getting shared between two cores increases, which means the number of invalidations should increase as the cache block size increases. This plot also supports the “U” shaped curve shown above, where the increase in cache block size leads to better spatial locality but it is in direct competition with the number of invalidations going up due to false sharing as seen here clearly.

Cache miss rate with respect to the total cache size:



This can make sense very easily. The bigger a cache is, the more data it can hold. No harm will be brought to the cache performance in any way as the cache size gets increased.

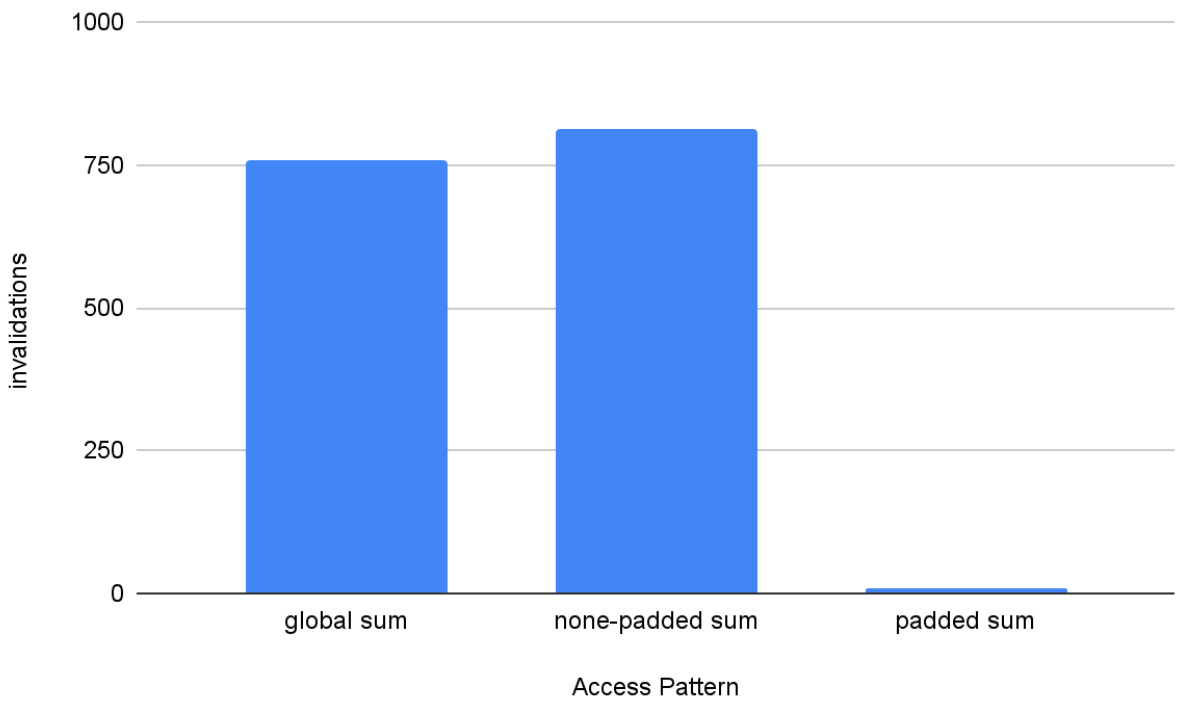
Cache miss rate with respect to different access patterns:



In the global sum and non-padded sum workloads, they're both adding their own portion of an array to a sum variable. The difference is, the sum variable is a global one across threads, whereas the non-padded sum maintains different sum variables for different threads by using a int array. However, since the array itself is short (4 in our benchmark, so the total size is 16), these individual integers of sum still fit in the same cache line. That's why the miss rate is still high.

In the padded sum case, we added padding to each int variable (i.e., both the sum variable, and the int element in the array) to make each int occupy exactly a whole cache block. We can see that the miss rate dropped dramatically because now every thread has its own working set of cache blocks, and they don't overlap each other.

Number of invalidations with respect to different access patterns:



The reason is the same as what we mentioned in previous analysis. There is no shared cache line between processors, so no invalidation operation will be triggered in this case.

References

1. SST Docs
2. How to add a new SST component
3. SST quickStart instructions
4. CMU 15-418/618 course slides
5. Intel Pintool
6. SST simple element examples

Labor Division & Credit Distribution

All the work is evenly distributed between the 2 group members. So each gets 50% of the credit.