Development of a simulation software for multiprocessor cache coherence protocols on top of *SystemC*

Andrea Sottoriva, 17th December 2006
Grid Hardware Infrastructure course
Universiteit van Amsterdam

Introduction

The objective of this project was to develop a simulation software for cache coherence protocols mainly focused on the *snooping* protocols family. I will describe my result across three main steps corresponding to the simulation of three different systems:

- a single processor cache
- a multiprocessor cache with *valid/invalid* snooping protocol
- a multiprocessor cache with *MESI* snooping protocol

For each step I'll provide a quick definition of the problem, the solution adopted and some validation experiments. Furthermore experiments performed using trace files will also be discussed. The chief issue I had to investigate was the performance impact for a family of protocols widely used for multiprocessors systems based on *Unifor Memory Access* architectures.

The scenario I will describe for the second and third part of my analysys involves multiple CPUs with a local cache each. The CPU+Cache blocks share a common main memory through a common bus on which specific snooping hardware acts.

The components of the system actually simulated are caches, the snooping hardware, the common bus and, partially, the CPUs (only for cache requests generation). In the entire software no real data are simulated, everywhere we deal only with addresses. Finally, the simulation framework used is *SystemC* (`http://www.systemc.org`).

Single processor cache

Before to discuss the snooping protocols we first explore the simulation of a single cache, supposing a mono processor system. In this case I implemented a cache with the following tunable parameters:

- size
- word length
- cache line length
- associativity
- memory request delay

and the following fixed characteristics:

- LRU (Less Recent Used) replacement policy
- Write-back policy using dirty bits

The behavior of such system is the classical behavior of a common cache, so we'll not go further with the details and we'll discuss directly the experiments performed.

Validation

In this simple case the validation process is quite straightforward: generate few regular requests from the CPU and check the correctness of the cache behavior. An example could be to generate incremental requests on a small 64 bytes cache, checking the regularity of cache misses or writes-to-memory operations. Actually I performed this experiments for each single configurable parameter of the cache ensuring the simulation would get me the expected results.

Trace files

Once the implementation is assumed to be correct, we can go on with the trace files examination. The addresses in the trace file I used are generated from the following code (suppose the matrix stored in memory by rows):

```
int A[N][N];
int X[N];
int Y[N];

int i, j, s;

for(i = 0; i < N; i++) {
    s = 0;

    for(j=0; j < N; j++) {
       s += A[i][j] * X[j];
    }

    Y[i] = s;
 }
```

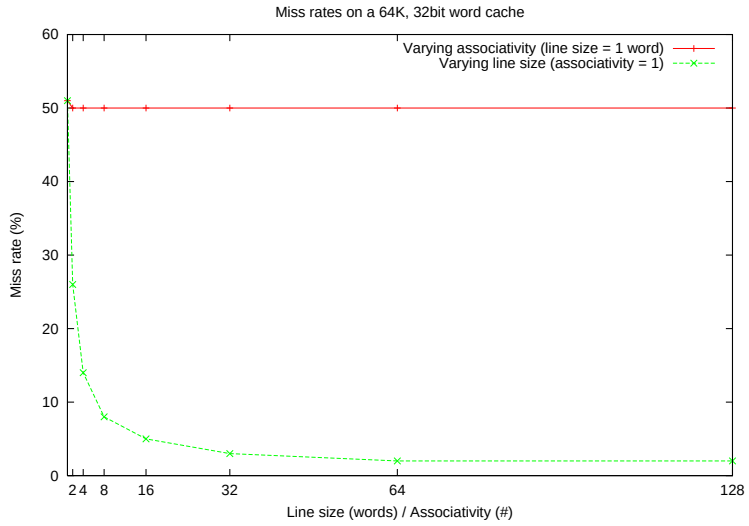
For all the following experiments I used a 64K cache with configurable associativity and line size and I studied the trend of two main measures of performance:

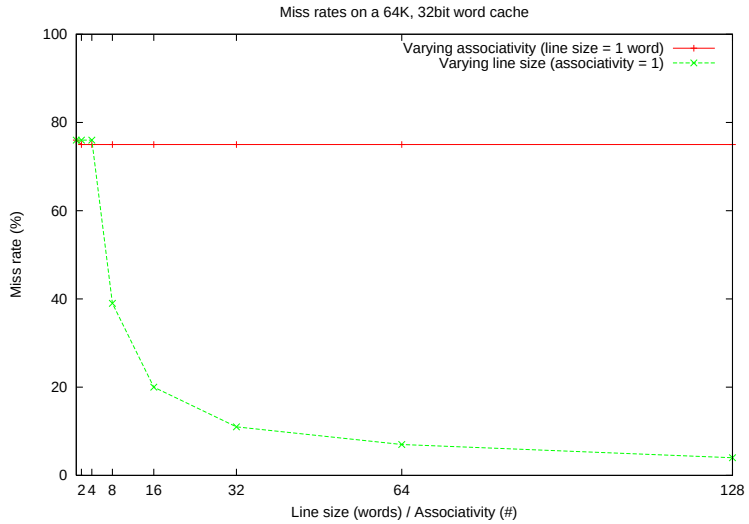
- *Read miss rate*: the percentage of cache reads that result on a cache-miss operation, involving a memory reading.
- *Write miss rate*: some of the write operation can result on a memory access if the underlying cache line is dirty, this rate can measure how many cache writings will result on the total amount of memory requests.

The total execution time is, from the cache point of view, strongly dependent from the memory accesses and then from the previous two measures. This is

the reason why the most important performance indicators for a cache are the miss rates.

While the first measure is mostly influences by the decribed configurable parameters such as the cache size, line size and associativity (depending on the structure of the requests) the second measure depends also from the used replacement policy. Further on the document this value will become more important for snooping protocols where the write policy adopted (back/through/around) is very significant. Here below are shown the miss rates while changing the cache line size or the associativity level:





As we expected, due to the structure of the code, the system does gain much performance on raising the cache line size for example. On the other hand the

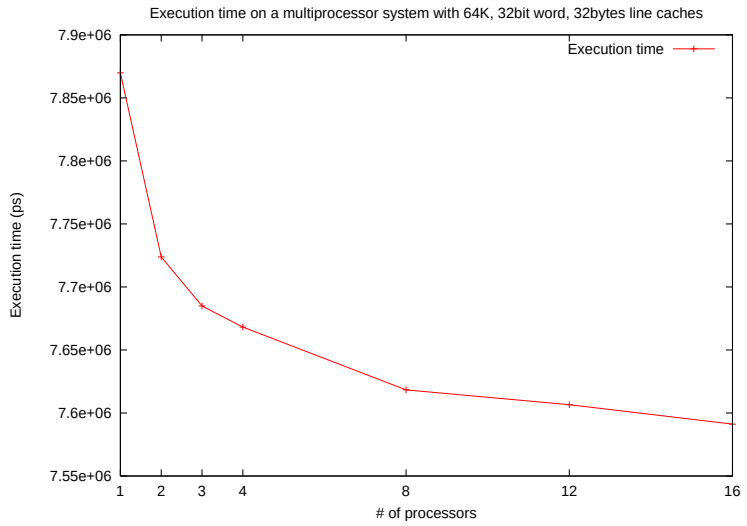
associativity factor is not relevand cause to the sequential accesses to memory.

Multiprocessor caches with *Valid/Invalid* snooping protocol

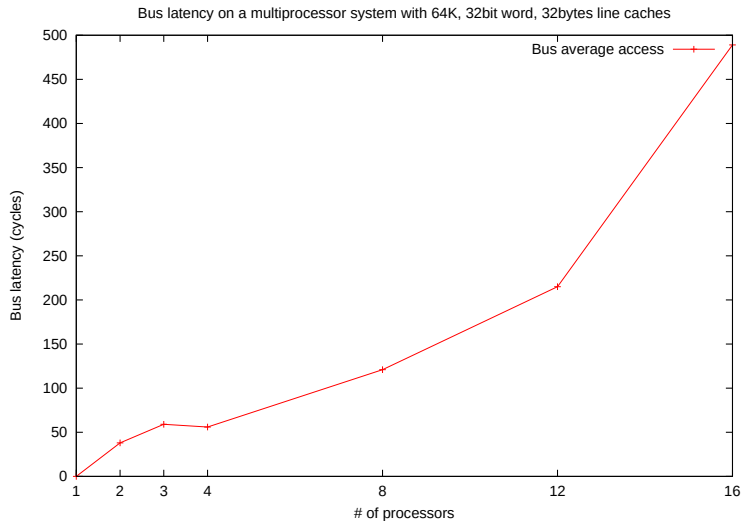
The valid/invalid snooping protocol is based on a very simple concept: use write-through policy and invalidate a datum whenever a write operation is snooped from the bus. Thus the behavior of our cache is the same regarding all the issues but the writing operations that will perform a direct access to memory. Moreover the dirty bit is no longer needed given we never have any *non-fresh* data on our cache. About the snooping, it will be the snooping hardware that will catch the address of the writings from the bus and change the data state, if it is present in the local cache.

To simulate this new system we need to define a new entity that will simulate the behavior of the bus: *BusMem*. This entity is a single instance of a normal class, shared between all the cache modules. It permits lock/unlock operations and it maintains the information about address and operation performed on the bus, besides the actual bus owner. The access to memory needs a mandatory passage through the common bus, for this reason the structure that simulate the contendence for the bus between the processors involves also the simulation of the main memory delay. Both for reading and writing the bus must first be locked together wih the memory. In this scenario is interesting to see how the total execution time fall increasing the number of processors.

Although here I'll now show the miss rate trend, from the experiments performed I found it remains stable on increasing scalability. This is what I expect given the intrinsic scalability of the code that doesn't compromise the performance due to the invalidation of snooped data. In particular in the examined case the execution time up to 16 processors seems to scale more than linearly using our benchmark code:



Here we take a look on the average latency caused by the race condition on the bus, it has clearly at least a linear correlation with the number of processors:



Although implementations of the valid/invalid protocol with write-back exist in literature, the solution I developed is based on the write-through policy: this influence much the execution time due to the frequent memory accesses for writings, even supposing a miss-rate close to 0%.

Multiprocessor caches with *MESI* snooping protocol

In the MESI snooping protocol we avoid to adopt a strict write-through policy by adding complexity to the cache state. The idea is to discriminate between critical events. In this case the cache state can have the following values:

- *Invalid*:

  The cache line doesn't hold any valid data. Writings to a cache line in this state result in a write-around to memory without the loading of the datum into the cache.

- *Shared*:

  The datum is shared between caches around the network, the value is the same of the main memory. Writings to a cache line in this state result to a write-through changing the state to *Exclusive*. Moreover, as described by the graph below all the other caches that snoop the writing will invalidate their datum instance.

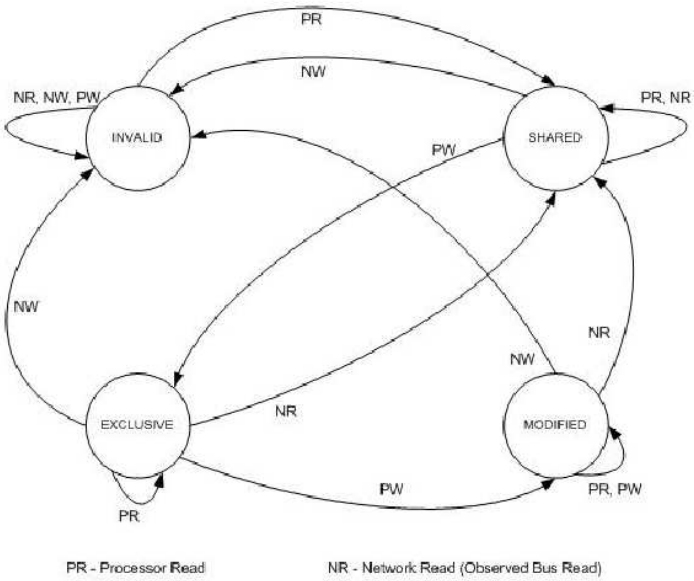
- *Exclusive*:

The datum is the only copy in the network, its value is the same of the main memory. Writings here result on a state changing to modified with a write-back policy. In this case we can exploit the locality of data and the assumption the overlapped usege of them is minimum, to avoid the maximum number of writings to memory as possible. Supposing after the invalidation from the previous *Shared* state, nobody else will try to access to the data for a reasonable amount of time.

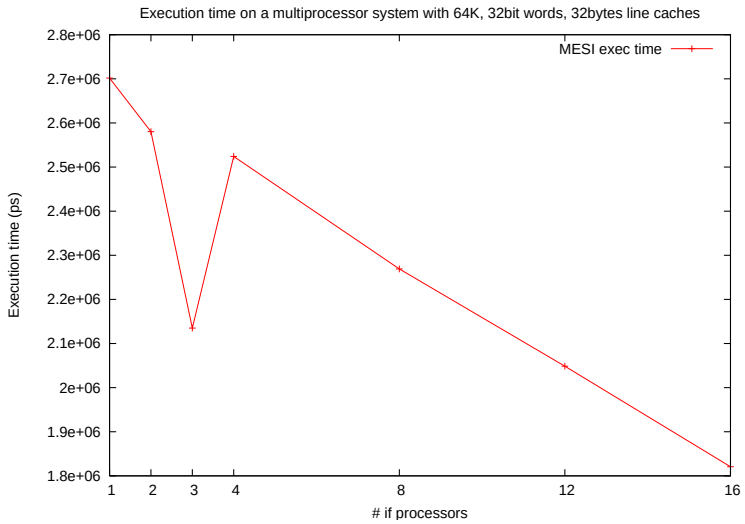
- *Modified*:

  The cache line holds the most recent and correct datum, the datum into memory is incorrect. Here a write-back policy is applied as well as in the *Exclusive* state. On snooping of a read from this address by the network a on-the-fly datum correction is performed: the reading operation requested is switched to a writing operation, putting on the bus the correct datum present on the local cache and let the requester to wait for a memory reading that in fact it's a memory writing. In the end of the transacion in the bus will be available the *fresh* datum, coherent respect to the memory, all the cache lines on the network containing the datum will be set to *Shared*.

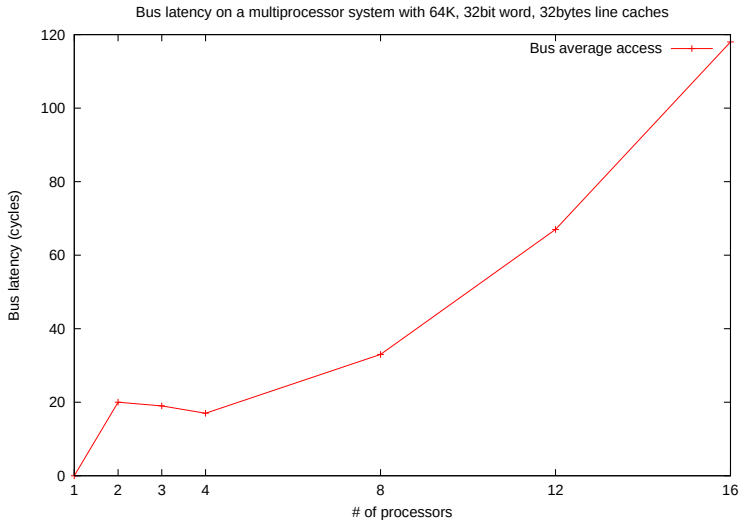
The following graph decribes the protocol as a finite state machine:



Here I performed the same experiments I performed for the valid/invalid protocol to study the scalability of the protocol:



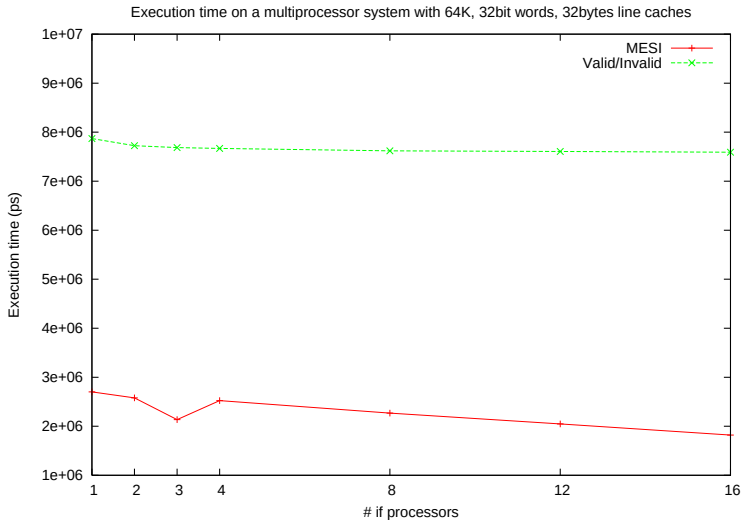
The trend is linear as it was for the valid/invalid protocol, but with a significant performance increasing (futher in the document we will compare the two results).



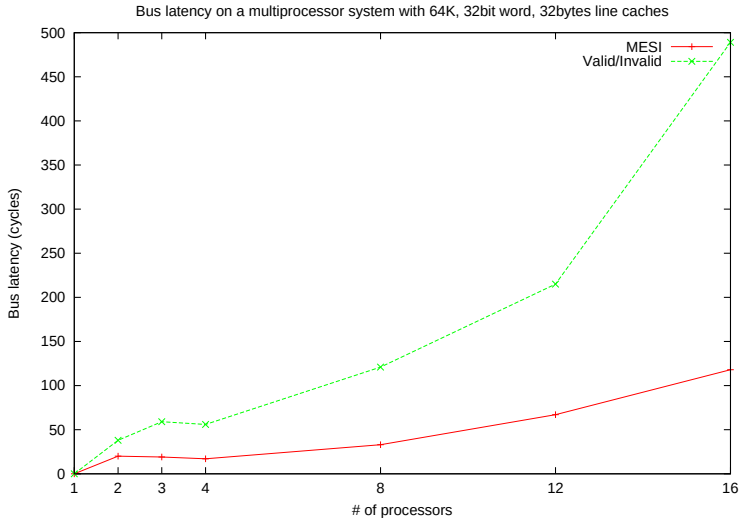
The average bus latency is similar to the previous snooping protocol with a more than linear trend.

Performance comparison

As we've seen in the previous section the two snooping protocol scale in the same way increasing the number of processors. Here the comparisons between them show that we gain a considerable amount of performance with the MESI protocol (as we obviously expected) obtaining an increase of about 300% on the performance independently by the number of processors:



I got the same results for the bus latency: the performances are increased independently from the scale by a factor of about 500%:



Conclusion

In this document we investigated how to realize and measure the performance of multiprocessor cache systems. We compared two main cache coherence algorithm of the family of the *snooping* protocols and we've shown how the performances rise with a multiple state cache-line protocol such as the MESI. Summaring, the objective of a snooping protocol is to maintain the global cache coherence minimizing the non-necessary memory accesses. Concluding our discussion appears also clear how much is important the software optimization, in this case we used a very well scalable piece of code. The optimization from a compiler point of view of the code remains however very tricky. In particular for cache systems we need to deal only with memory accesses, a kind of instructions that cannot be easily optimized in compilation time.