Group aware cache coherence

Chingyi Lin

ABSTRACT

The number of processor increases in current computing server. However, the typical MSI and MESI protocol only describe the global state for each cacheline. In this project, we augmented MSI protocol by adding a G- flag in its state. Furthermore, we modified the interconnection to leverage this new coherence protocol. In the end, we reduce 22% of scalibility cost of cache miss latency in PARSEC3.0's blackscholes program.

1. INTRODUCTION

With the emergence of multicore system, data movement becomes crucial and important. Programmers wrote various parallel programs to leverage the numerous cores in their calculation, but don't want to suffer from parallelization cost from cache coherence and data movement.

Cost of data movement becomes more and more expensive ss increased number of cores in current super computing server. Vast works focus on the interconnection between each cores. The general purpose of these works try to reduce the latency in data sharing or request sending.

These tricks lower the latency in cache coherence, but most of them do not solve the original question, preventing traffic on unwanted bus. Even some interconnection structure reduce the average transmission time, those requests still need to be transmitted to the central directory through the buses. This paper focuses on group aware cache coherence, which gives a hint of the information about data sharing.

The following are our main contribution.

1. Propose a extended MSI protocol to describe the coherence state in more detail.
2. Design an interconnection which leverage G-MSI.
3. We evaluate G-MSI on gem5 simulation and get 20% improvement in scalibility cost.

2. GROUP FLAG IN G-MSI

2.1 Group Flag

Group flag is a prefix of a MSI state. With "G-" flag, the state is changed from normal state to group state, which means that the cache coherence only need to be held inside the group only. Once the cacheline with group state send a cache coherence request to the directory, it contains the hint that the coherence behavior only needs to be perform inside this group.

2.2 Group states

We have two group states in this protocol, Group-Invalid (GI) and Group-Shared (GS). Group-Invalid suggests that the cacheline is invalid, but if there is a valid data, it must inside the group. Or simply to say, any cacheline of this address is invalid outside this group. Same as GS state, which guarantees that the cacheline is shared only if its processor is inside this group.

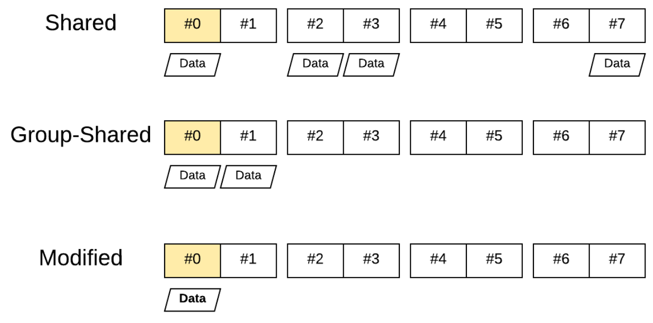

Figure 1: Example for states

In the figure 1, each two adjacent cores are a group. The first cases shows the example of typical shared-state cacheline. Because the data is shared among all the processors on the system, core #0 has responsibility to maintain the cache coherence and inform the others while updating this cacheline.

The second case shows the benefit of the group states. Though the cacheline is shared to other cores, these cores are a neighbor inside core #0's group. This case is common with a program only uses a subset of cores on the system. Its allocated local data is seldom accessed by other program in other core. With G-flag here, core #0 doesn't need to send request to cores outside its group. This trick saves both time and bandwidth on maintaining cache coherence.

2.2.1 No group-modified state

Group-Modified doesn't exist in this protocol. In typical MSI protocol, M state is unique among all the processor, which implies all the other processors are invalid. Based on this implication, G- prefix is a trivial flag for M state.

2.2.2 Group state is a proper subset of normal state

Each group state is a subset of its normal state. If a cacheline can be classified as GI or GS state, it's also valid to label them as I or S state. The G- flag contains the information outside the group. The lack of G- flag will make the transmission of requests/response more inefficient due to the conservative behavior from directory, but the result is still correct.

2.2.3 Group flag is a group-wise status

Instead of individual state in MSI protocol, group flag is applied on a whole group in G-MSI, which means that a cacheline with a group flag implies this cacheline is on group state in other cores in this group.

2.3 Special cases

With some extreme setting, group cache coherence protocol can be treated as other popular protocol.

2.3.1 All processors are a group

Assume all of the processors are in a single group. This protocol will be equivalent to typical MSI protocol. Since we don't have processors outside the group, there is no concept of "group" here. In other words, the G-prefix here is trivial. Under this setting, GS is same as S state, also GI can be seen as I here. After these state merge, we only left three state, M, S and I. Furthermore, the transition is same as MSI protocol.

2.3.2 Each process is a group

Another special case is in opposite. We take every processor as a group, which means that each action should be taken to a processor outside the group. This case is similar to the MESI protocol. Because the GS state here claims that only the processors inside its group might have the data. Under this group structure, this claim is equivalent to "This cache is the unique processor with this cacheline", which is similar with the definition of "Exclusive state" in MESI protocol. (But the behavior are different.)

2.4 State transition

As we have more states in G-MSI, the behavior of these states should be well-defined.

Intuitively, G-flag should not affect any behavior in MSI. In figure 2, all the behavior of these group state are same with its normal state. GI can be promoted to GS with issued load, and GS is changed to M with issued store.

2.4.1 Defining invalidated modified-state cacheline

The only exception happens on the invalidation of M state. To prevent ambiguity, M cannot switch to both I and GI on the same event. In here, we define the invalidated cacheline goes to I state. This definition is not precise enough, because everyone will be invalid once the M-state cacheline is invalidated, implying that cacheline should be in GI state. But for simpler implementation, we define these invalidated cachelines into I state. This modification is valid based on the property in 2.2.2.

2.4.2 Reduction/Expansion

Once a cacheline is owned by a group only, states in this group will be "reduced", or in other words, given G-flag. This situation comes from two event

1. Cache asks a data and get from memory.
2. Someone outside the group is invalidated and no one outside the group has the data.

In opposite, when the data is shared to another core outside the group, its state will be "expanded" into normal state. It will only happen on someone outside the group requiring shared or modified state for the cacheline.

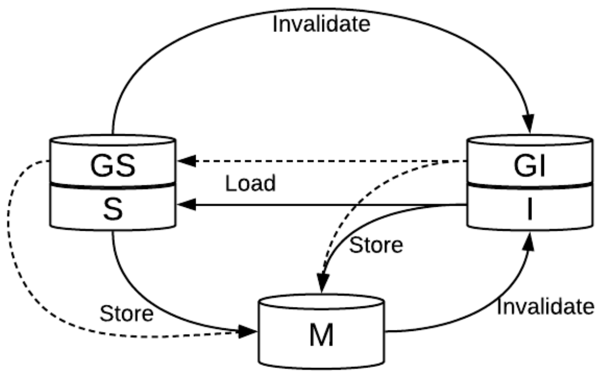

Figure 2: State transition

3. EXTENDED INTERCONNECTION

To leverage this protocol, we also proposed an interconnection to improve the efficiency of G-MSI.

3.1 Baseline interconnection

The baseline interconnection consists of simple, global cache coherence bus, which connects directory and all the L1 cache on the system. Even the detailed state is known, the reduced requests/responses still need to be transmitted on the global bus.

3.2 Group bus

To distinguish the transmission of reduced requests and the normal, an external local bus is necessary. In G-MSI, we augmented ”group buses”

The group buses are a set of buses, in which each of them connects the directory and all processors of a group. Due to the less number of components it connects to, the transmission time of these buses will be shorter than global bus. This bus has the following properties

3.2.1 Uni-directional from directory

Though the group buses connect directory to the cores inside the group, preventing unfair comparison, cores cannot send a request to the directory through this faster bus.

3.2.2 Communication between cores inside the group

A core only needs to send its request to its neighbor inside the group, if the cacheline is with G-flag. Internal buses can accelerate this smaller domain transaction. But once a receiver is outside the group, its request still needs to go through the global bus.

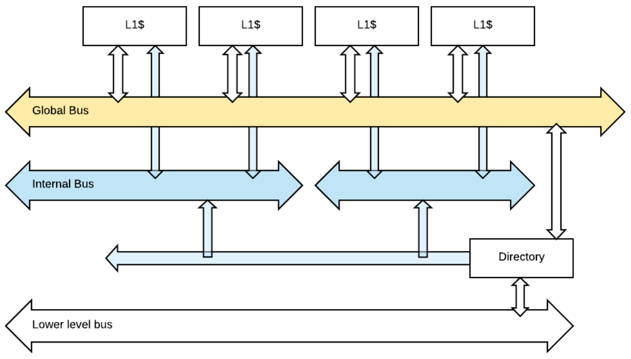

Figure 3: Interconnection for G-MSI

3.3 Transmission time assumption

To benefit from internal bus, we make an assumption for transmission time. Generally, the number of component connected to bus is positively correlating to the transmission time. To make it simple, we take the transmission time function t(x)=kx, where x is the number of component on the bus, and k is a constant. In short, the transmission time is linearly proportional to the number of component.

4. EVALUATION

4.1 Experiment setup

4.1.1 Benchmark

The parallel program can be categorized into two main kind. One is using the fixed number of thread. To get balance between the performance of program and resource utilization, we usually specify a fixed number of thread. But in some high performance cases, they will fully utilize every processor on the system.

For parallel program benchmark, we use Blackscholes in PARSEC3.0, which tries to solve Black-Scholes Partial Differential Equation.

4.1.2 Parameter

To configure the system, we have the following parameters.

1. nCore: Number of core in the whole system.
2. nIntras: Number of core in a group.
3. nThread: Number of thread used in the program

There are two major kind of methodology in grouping processors. Fix-number grouping doesn’t change the number as nCore increases. But in dynamic grouping, nIntra is proportional to nCore, which implies that the number of group is fixed.

For transmission time modeling, only two parameters should be determined.

1. cyclePerCore: The ratio between transmission cycle and nIntra, which should be a constant based on our assumption.
2. memLatency: The latency for memory access. It should be larger than the longest transmission path. In short, memLatency $>$ nCore * cyclePerCore

4.2 Fix-threaded program

We evaluate the program using two and four threads separately. Each of them are running on 25 and 20 many-core system with different configuration respectively. The configuration includes

- nCore: From 2^1 to 2^5, excluding the nCore greater than nThread due to resource insufficiency.
- nIntra: With fix-number grouping (1,2,4) and dynamic grouping ($\frac{1}{2}$nCore, nCore)

4.2.1 Execution time

nCore	nIntra=1	nIntra=2	half-MSI	MSI
2	1678.027	1678.027	1678.027	1678.027
4	1710.398	1710.398	1710.398	1710.398
8	1770.688	1770.688	1770.688	1770.688
16	1892.475	1892.475	1892.475	1892.475
32	2136.828	2136.826	2136.828	2136.828

Table 1: Execution time of 2-threaded program (unit:ms)

nCore	nIntra=1	nIntra=2	half-MSI	MSI
4	799.244	799.244	799.244	799.244
8	831.780	831.794	831.697	831.730
16	896.650	896.692	896.667	896.672
32	1026.723	1026.741	1026.721	1026.753

Table 2: Execution time of 4-threaded program (unit:ms)

From the table 1 and 2, the scale-up of system slow down the program greatly. It means the increased coherence transmission time degrades the performance.

Though the execution time is affected by nCore, each setting of nIntra seems to have little difference. In 2-threaded, even with 32-core system, the maximum difference is only 2 microseconds in a 2-sec program. The 4-threaded has 30 microseconds but still too small.

We cannot see the great benefit of G-MSI from the execution cycle result. In section 4.2.2, we will compare the other statistical data in these simulation.

4.2.2 Cache miss penalty

G-MSI is targeted on the request transmission. This transmission occurs only on cache miss in any core. Thus, cache miss penalty should decrease with the presence of G-MSI. Figure 4 and 5 show the cache miss penalty of all the systems. The steep slope suggests the miss penalty is sensitive to the scale of system. It turns out that when nIntra is equal to nThread, the system will get the minimum cache miss penalty. This makes sense because larger nIntra waste time on transmission of group buses, and the smaller nIntra might use some time-consuming global transmission between cores.

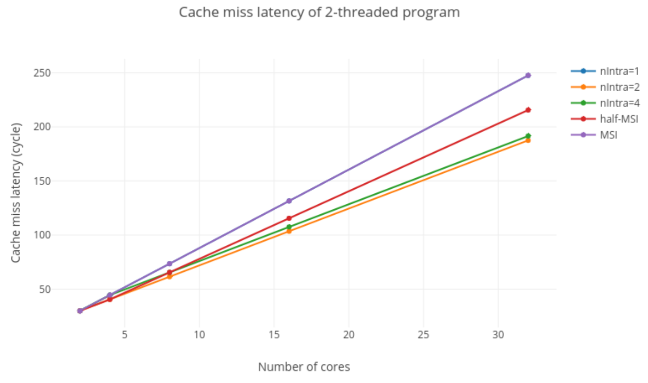

Figure 4: Cache miss penalty for 2-threaded program

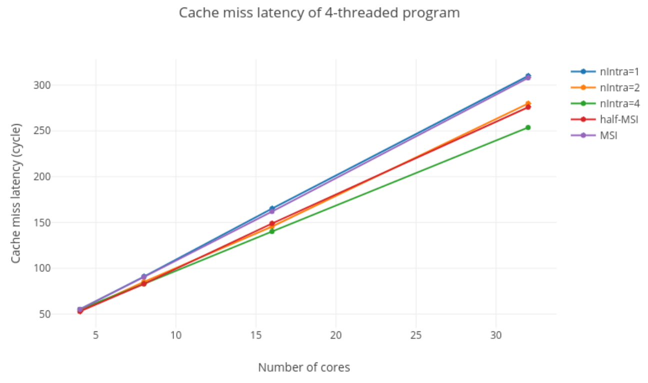

Figure 5: Cache miss penalty for 4-threaded program

4.2.3 Scalability cost

Scalability cost is used to evaluate the additional cost with the scale-up of the system. The scalability cost from m-core system to n-core system with configuration A is defined as

$$scal\_cost_{configA}(m,n) = \frac{cost(n) - cost(m)}{n-m}$$

From the simulation, we can assume the scalability cost is independent to m,n. The definition can be simplified into

$$\begin{aligned} scal\_cost_{configA} &= scal\_cost_{configA}(m,n) \\ &= \frac{cost(n) - cost(m)}{n-m}, \forall m,n \end{aligned}$$

Under this definition, we know that G-MSI saves 22% in scalibility cost of cache miss latency.

4.3 Full-threaded program

Full-threaded program evaluates the full performance of the system. Each program will use all the cores on its system.

4.3.1 Execution time

Same as section 4.2.1, the execution time doesn't improve much. But here, we can see an interesting point. The 32-core program is slower than 16-core, which implies that adding core or parallelism doesn't always work.

nCore	nIntra=1	nIntra=2	half-MSI	MSI
2	1678.027	1678.027	1678.027	1678.027
4	799.244	799.244	799.244	799.244
8	558.789	558.811	558.809	558.797
16	495.988	495.967	495.975	495.955
32	519.868	519.876	519.894	519.880

Table 3: Execution time of full-threaded program (unit:ms)

4.3.2 Cache miss penalty

In cache miss penalty, everyone is greater than MSI. But unlike fix-threaded program, there's no one configuration dominates every cases.

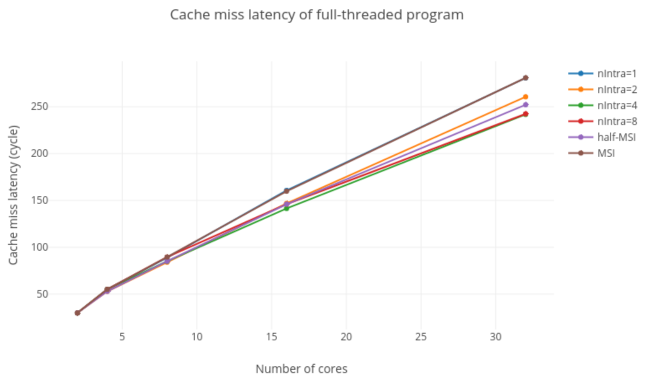

Figure 6: Cache miss penalty for full-threaded program

5. DISCUSSION

5.1 The un-improved execution time

From the section 4, the execution time is not improved much compared to the miss latency. After we dig into the statistic, we found that the blackscholes testbench has little data sharing rate. Each blackscholes program has about 1.8 million misses, but even there are 32 cores, only three thousand misses come from other caches.

Less data exchange and intensive data reuse is awesome as a programmer because data sharing is one of the most expensive part in a parallel program. However, G-MSI cannot get a huge benefit on these low-sharing-rate program. In next section, we wrote a "bad" program with lots of data exchange to help our evaluation.

5.2 Self-written intensive data sharing program

To see the advantage in G-MSI, we wrote another program by ourselves. In this program, data is intensively shared between cores. This program is simple.

```
void thread( int *c, int tid, int threads ){
    for( int j=0; j<N; j++ ){
        for( int i=tid; i<N; i+=threads ){
            c[i]++;
        }
    }
}
```

This function is forked into every thread, which increment an element of an array. Since the interleaving access makes the array shared to all cores, each cacheline will be shared and switched among cores repeatedly.

5.2.1 Analysis

Data sharing rate is a good metric to measure how intensively the data shares between cores. Here, we define the data sharing rate as the ratio of getting data from other caches to getting data from directories on miss. In other words, data sharing rate is the probability that the data will come from other caches on miss.

	2	4	8	16	32
blackscholes	8.5e-6	6.1e-5	2.0e-4	6.3e-4	2.2e-3
canneal	5.3e-4	2.3e-3			
self-written	0	0.48	0.44	0.47	0.46

Table 4: Data sharing rate for some programs

Table 4 shows the data sharing rate of some program. It turns out that this self-written program has a relatively large data sharing rate. With this intensive data sharing, this program is suitable as a test program for our G-MSI.

5.2.2 *Execution time*

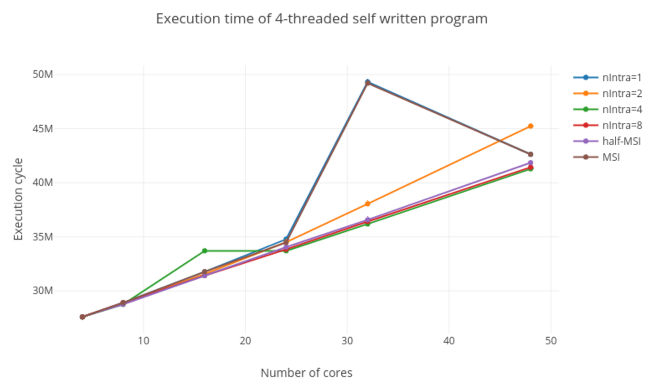

Figure 7: Execution time for 4-threaded self-written program

Figure 7 shows the execution time of our self-written program, in 48-core system, we can save 8% to 10% of scalibility cost, but to be noticed is, a poor grouping will worsen the performance. For example, nIntra=2 takes longer than original MSI protocol here. So, a good grouping strategy is a necessity in G-MSI.

5.2.3 *Cache miss penalty*

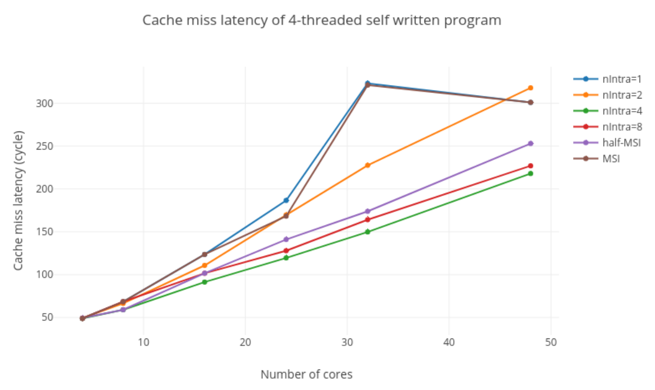

Figure 8: Cache miss penaltyfor 4-threaded self-written program

Same as section 4, the nIntra=4 still get the lowest cache miss penalty. Furthermore, it saves more than 30% scalability cost

6. CONCLUSION

G-MSI extended the original MSI protocol with more detailed data sharing information. With the modified interconnection, it can transmit the request in a more efficient way. Due to the lower data sharing rate of blackscholes benchmark, though G-MSI saves 22% of scalibility cost, it doesn't improve the execution time greatly. But from the experiment in discussion, we found that with a higher data sharing rate, G-MSI can save the scalibility cost in exection time from the awareness of the group.