Midterm Exam

Rusu Wu
11694764

Problem 1

(1)Halting run: start $\to u_0 \to u_1 \to u_2$

Initially, all the bags of B_{u_0},B_{u_1} and B_{u_2} are $\emptyset$. To begin with, the system would move from node u_0 to node u_1 and add a p into B_{u_1} dropping nothing. So far, B_{u_0} is still empty, and there is a p in B_{u_1}. Since there is a p in B_{u_1}, the label $\bar{p}|q$ is not vaild on u_1. So, u_1 only can move to u_2 and could not move back to u_0. When u_1 moves to u_2, B_{u_1} drops a p, and it becomes $\emptyset$. Also, B_{u_2} would add a q, and finally the system halts.

(2)Halting run: start $\to u_0 \to u_1 \to u_2$

Initially, all the bags of B_{u_0},B_{u_1} and B_{u_2} are $\emptyset$. Since u_0 doesn't have p, $\bar{p}|q$ is vaild on u_0. The system would move from node u_0 to node u_1 and add a q into B_{u_1} dropping nothing. So far, B_{u_0} is still empty, and there is a q in B_{u_1}. Since there is a p in B_{u_1}, the label $\bar{q}|p$ is not vaild on u_1. As label $q|p$ is vaild on u_1 ,u_1 can move to u_2 and could not move back to u_0. When u_1 moves to u_2, B_{u_1} drops a q, and it becomes $\emptyset$. Also, B_{u_2} would add a p, and finally the system halts.

Problem 2

Halting run 1: start $\to u_0 \to u_1 \to u_2$

Initially, all the bags of B_{u_0},B_{u_1} and B_{u_2} are $\emptyset$. The system would move from node u_0 to node u_1 and add a p into B_{u_1} dropping nothing. So far, B_{u_0} is still empty, and there is a p in B_{u_1}. Since there is a p in B_{u_1}, the label $p|\emptyset$ of edge $u_1 \to u_2$ is vaild on u_1, the system can move from u_1 to u_2. When it moves from u_1 to u_2, B_{u_1} drops a p, and it becomes $\emptyset$. Also, B_{u_2} adds nothing ,so its bag remains $\emptyset$. Finally the system halts.

Halting run 2: start $\to u_0 \to u_1 \to u_0 \to u_1 \to u_2$

Initially, all the bags of B_{u_0},B_{u_1} and B_{u_2} are $\emptyset$. The system would move from node u_0 to node u_1 and add a p into B_{u_1} dropping nothing. So far, B_{u_0} is still empty, and there is a p in B_{u_1}. Since there is a p in B_{u_1}, the label $p|q$ of edge $u_0 \leftarrow u_1$ is vaild on u_1. When the system moves from u_1 to u_0, B_{u_1} would drop a p and become $\emptyset$. Also, B_{u_0} would add a q. After that, the system moves from u_0 to u_1 adding one p into B_{u_1} and drops nothing from B_{u_0}. Since there is a p in B_{u_1}, the label $p|\emptyset$ of edge $u_1 \to u_2$is vaild on u_1, the system can move from u_1 to u_2. When it moves from u_1 to u_2, B_{u_1} drops a p, and it becomes $\emptyset$. Also, B_{u_2} add nothing remaining empty. Finally the system halts.

Halting run 3: start $\to u_0 \to u_1 \to u_0 \to u_1 \to u_0 \to u_1 \to u_2$

Initially, all the bags of B_{u_0},B_{u_1} and B_{u_2} are $\emptyset$. Firstly, the system would move from u_0 to node u_1 and add a p into B_{u_1} dropping nothing. So far, B_{u_0} is still empty, and there is a p in B_{u_1}. Since there is a p in B_{u_1}, the label $p|q$ of edge $u_0 \leftarrow u_1$ is vaild on u_1. When the system movess from u_1 to u_0, B_{u_1} would drop a p and become $\emptyset$. Also, B_{u_0} would add a q becoming $\{q\}$. After that, the system moves from u_0 to u_1 adding one p into B_{u_1} and drops nothing from B_{u_0}. Due to there is a p in B_{u_1}, the label $p|q$ of edge $u_0 \leftarrow u_1$ is still vaild on u_1. Then the system moves from u_1 to u_0. B_{u_1} would drop a p and become $\emptyset$. Also, B_{u_0} would add a q becoming $\{q, q\}$. After that, the system moves from u_0 to u_1 adding one p into B_{u_1} and drops nothing from B_{u_0}. Now, the system is at u_1, both the lebel to u_0 and u_2 are vaild on u_1. However, if the system moves back to u_0, it would add a q into its bag B_{u_0} changing it to $\{q, q, q\}$. Such that the total number of objects would exceed 2, which causes crash.Therefore, in order to have a halting run, the system should move from u_1 to u_2. When it moves from u_1 to u_2, B_{u_1} drops a p, and it becomes $\emptyset$. Also, B_{u_2} add nothing remaining empty. Finally the system halts.

Problem 3 The runing of bounded Cpts516-system can be seen as the switching of many configurations. Hence, We can treat $\alpha \to \beta$ as an edge from node α to β, and each configuration is treated as a node of bounded Cpts516-system. Then we obtain a graph G that is similiar to a bounded Cpts516-system. Since a bounded Cpts516-system has limited numbers of configurations, we can list out all the possible configurations of a bounded Cpts516-system as the input w of a Turing Machine. Hence, w is given since we only have many distinct configurations finitely. Such that, we can check whether α can reach β by one step. And also we can check wheter the input can be recognized by a Turinng Machine. Therefore, if there is an algorism that can check whether the initial configuration of the bounded Cpts516-system with certain input w can reach an accepting configuration, we say that the system is decidable.

```
Construct a TM $M$
input $w$
check $w$ is the following farmat
  $u_0\#u_1\#u_2\#....\#u_n\#$
  for some n, and each is a $u_n$ configuration of TM M
Check $u_0$ is the initial configuration of M over input $w$.
Check $u_n$ is an accepting configuration.
Check each $u_i$ can reach $u_i$ by a one step transistion in M.($u_i + 1$ is the resulting configuration from $u_i$ after running one instruction in TM)
```

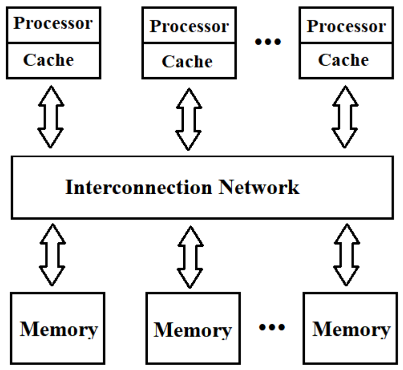
Once all Check out. M says yes on w, otherwice says no on w. Such that we can say that a bounded Cpts516-system's halt is decidable.

Problem 4

‘

Implementing Cpts516-system to Achieve Cache Coherence

In order to pursue various properties such as high performance on speed, low power consumption and low cost, many modern computers adapt to the computer architcture that has mutiple processors sharing one memory in hardware. We call the architecture as shared-memory multiprocessors architecture. Shared-memory multiprocessors architecture usually has many processors associated with mutiple levels of private cache. Also, these private caches can communicate with each other or visit the shared memory equally through interconnection network[1]. The figure below is an example of shared-memory multiprocessors system which is consisted by mutiple processors, private caches, and interconnection network between cache and memory.



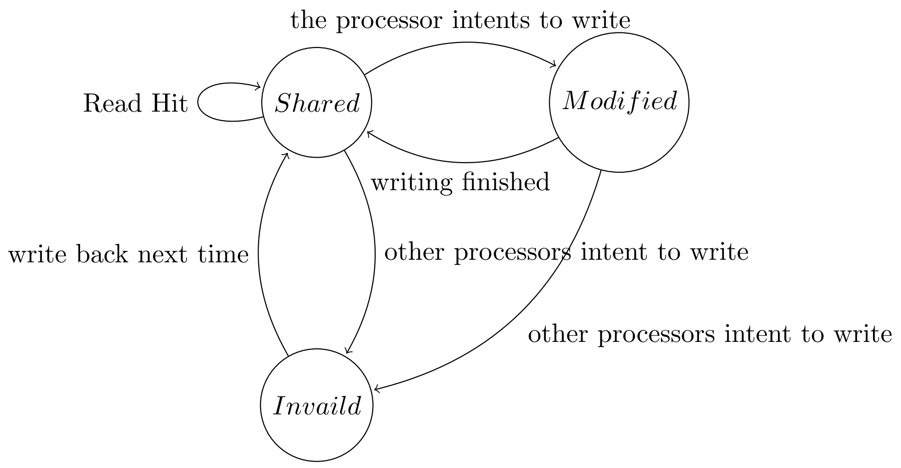
Using large, multi-level caches can reduce the bandwidth requirements of the memory.When a certain content is cached, its location information would be recorded in the cache. In this way, the visit time will be shortened when the next time this location is visited. Correspondingly, the required memory bandwidth is also reduced because there is no need to directly access the memory[2].

Computers that apply shared-memory multiprocessors architecture usually support the caching of shared data and dedicated data. Shared data means that the date can be cache or used by all the processors, while dedicated date only can be used by an exact processor. As I say above, using caches can speed up the operation time. However, caching shared data will bring a problem called cache coherence. When different processors' cache have the same copy of a certain memory loction, if one of the processors changes the value of this copy, then conflicts will occur; the copy of other processors's cache is no longer updated. So, if the next time there is a read on the copy, wrong results would be gennerated. Therefoere, caches should be supposed to be transparent. It need to be ensured that the cached contents are undated. This is what we called cache coherence. In order to achieve cache coherence, some systems apply software approaches to avoid the shared data's brock being cached by more than two caches through restricting or prohibiting task migration. Besides, some other systems implement cache coherence protocols to ensure cache coherence; by using cache coherence protocals, all the private caches can have a copy of a certain shared data.[3]

The key to achieve cache coherence of protocols is to track the shared states of data blocks. So far, there are two kinds of cache coherence protocols that are normally used. One is directory-based protocols, and another kind is snooping protocols. Directory-based protocols keep track of the status of physical blocks by recoring its states in to a same directory. Whereas, systems that adapt to snooping protocols has snopopers in caches. If a cache has a copy of physical block, the snopoper can track the states of the block. In short, Directory-based protocols use a directory to recode states, while snooping protocols snoop states through bus such as the bus between caches and memory.[2]

MSI 3-State cache coherence protocol (MSI protocol) is a simple snooping protocol. This simple protocol has three states as its name says: Modified, Shared and Invaild. Modified(M)means that the cache line is under modifying, and other caches can not reach the cache line and copy it (at the state of Modified, the cache line is exclusive.) Shared(S) refers to the state that the cache line can be copied by other caches. Finally, Invaild(I) is the state that the content of the cache line with exact address tag is no longer updated because other caches have changed its content.

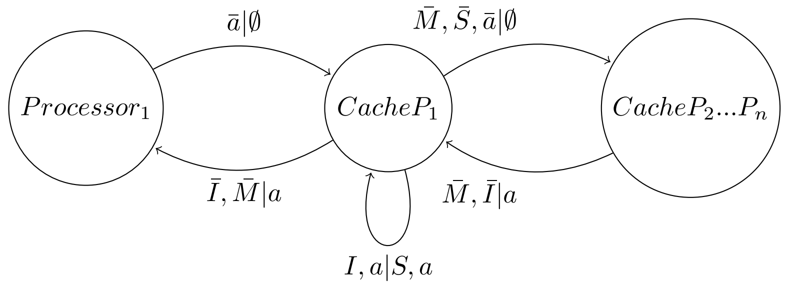
My approach of addressing cache coherency with Cpts516-system is referenced from MSI Protocol. In each cache line in cache, I defined that there are state bits storing the current state. Also, I defined 3 basic state including Modifyed, Shared and Invaild to record the states of a cashe line as the MSI protocol. The graph below shows the states and state transitions of my cache coherence protocol.


the processor intents to write
Read Hit
$Shared$
$Modified$
writing finished
write back next time
other processors intent to write
other processors intent to write
$Invaild$

Normally, there are four cases in cache/memory operation including Read Hit, Read Miss, Write Hit and Write Miss.

(1)Read Hit: The associated cache of the processor currently has the cache line, and the cache line is in the S state. Hence, the processor directly read to the cache line, and the cache line keep the state as S during the process.

(2)Read Miss: ①The private cache of the processor currently has the cache line but in the I state, ②or the private cache of the processor currentely does not have the cash line. In these cases, the cache line should first search the required content from other processors'caches. If the searching fail, it secondly load form memory. Below I take ① as an example.



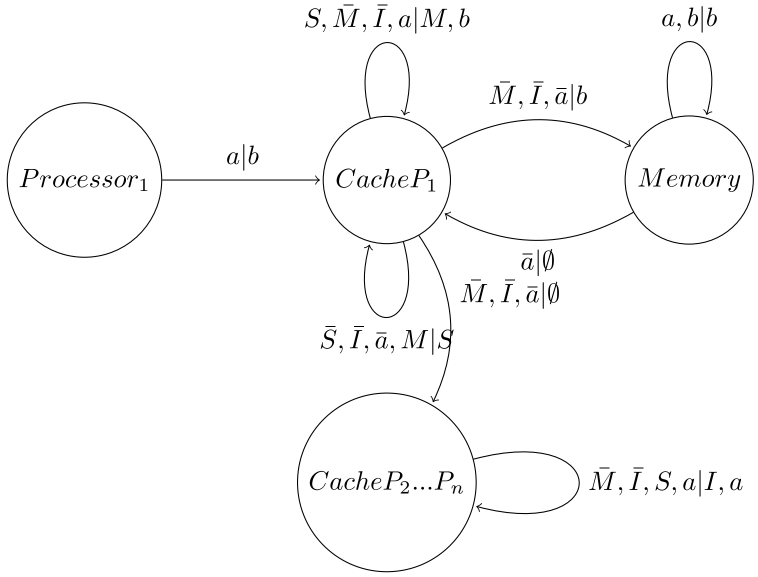
Initially, the a cache line's state had been set to be I due to other processor's writing on the cache line. In this case, if $processor_1$ requires a from its cache, there is a Read Miss happening $(aisinvaildnow)$. $Processor_1$'s cache would firstly send a requirement bus of a to the other processors' private caches, searching that whether there is a cashline with exact address tag in S state. After other caches recive the requirement, once they find that they have the required cache line in

S state, they would send back the exact cache line to $processor_1'scache$. Since the private cache of $processor_1$ has the updated a cache line now, it would switch its state from I to S and send the required cache to $processor_1$.

If the other processors' private caches could not find the exact cahce line that is in S state, Processor 1's cache shoud load a from memory. Same with above, once processor 1's cache loads a from memory, it would change its I state to S state.

Case ② is similiar to case ①,but the original state is $\emptyset$ (This is because there is no such cache line in its private cache).Same as case ①, the cache would ask other processor's cache to search. If there is a in S state, send it to $processor_1$'s cache. $Processor1$'s cache would evict one cache line's content and clear its state as $\emptyset$. Then, $processor_1$'s cache change the state from $\emptyset$ to S. If the other processors' private caches could not find the exact cahce line that is in S state, $Processor_1$'s cache shoud load a from memory. Same with above, once $processor_1$'s cache load a from memory, it evicts one cache line's content and clear its state as $\emptyset$. In the same time,it will change its $\emptyset$ state to S state and store the content into the cache line.

(3)Write Hit: The processor intents to write or modify the content of a memory with exact address tag, and its private cache currently has the cache line. Hence, the processor firstly changes the content of the its private cache as well as tells other processors' caches that the content of the cache line is no longer vaild. Also, it need to help the memory modify the content in the certain bolck. Below is the operation process modoling by a bounded Cpts516-system.


$S, \bar{M}, \bar{I}, a|M, b$
$a, b|b$
$\bar{M}, \bar{I}, \bar{a}|b$
$Processor_1$
$a|b$
$CacheP_1$
$Memory$
$\bar{a}|\emptyset$
$\bar{M}, \bar{I}, \bar{a}|\emptyset$
$\bar{S}, \bar{I}, \bar{a}, M|S$
$CacheP_2 ... P_n$
$\bar{M}, \bar{I}, S, a|I, a$

$Processor_1$ intents to replace b with a, and once its private cache recives the instruction, it would switch its current state from S to M (Note that a is initially in S state in the cache line). Once a has been written by b, the cache has three options: one is to switch it state back to S, or to inform other processors' caches that a is no longer vaild(inform them that their state of the cache line with exact address tag should be switched to I), or to help memory to repalce a with b. These three options need to follow time order; it means that the cache should firstly change its state of M back to S, then inform other processor's caches and finally to modify memory. Since there is no such constraint in time when we use System-S to model, there can be one of the limitations of the modeling process.

(4)Write Miss: The processor tend to write the content of a memory that has exact address tag, but its private cache doesn't have the exact cache line. The process is similar with Write Hit, and the sole diference is that the private cache should firstly evict one cache line and clear its state as $\emptyset$.Then, it write the blank cache line with certain content and tag it with the certain address.

Moreover, it should change the state from $\emptyset$ to M at the same time. Once it finishes writing, switch the state to S. After that, the following process is similiar with Write Hit: Fristly informs other caches of the invaility of the cache line; secondly send a instruction to memory and help menmory change the content.

In conclusion, by implementing the bounded Cpts516-system, I developed an approach to snoopy the cache coherence for the system of shared-memory multiprocessors. In this approach, the key is to snoopy the current states of cache lines, which references from MSI protocol. However, there are two abvious weaknesses within the approach. One is that the opprocah is referenced from MSI protocol, so it follows all the limitations of MSI protocol(for example, when there is a write miss, it generate two bus transactions including read miss and write miss, which increases burden to the bus). Another one is that the approach does not have any ordering constraint on time in some process such as the case of Write Hit.

References

[1]P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, 1990.

[2]J. L. Hennessy and D. A. Patterson, *Computer architecture A quantitative approach.* Cambridge: Morgan Kaufmann, 2019.

[3]J. Archibald and J.-L. Baer, "Cache coherence protocols: evaluation using a multiprocessor simulation model," *ACM Transactions on Computer Systems*, vol. 4, no. 4, pp. 273–298, 1986.