토큰 코히런스 프로토콜을 위한 경서열 트렌지언트 요청 처리 방법

論 文
54D-10-5

New Transient Request with Loose Ordering for Token Coherence Protocol

朴 潤 慶† · 金 大 榮*
(Yun Kyung Park · Dae Young Kim)

Abstract - Token coherence protocol has many good reasons against snooping/directory-based protocol in terms of latency, bandwidth, and complexity. Token counting easily maintains correctness of the protocol without global ordering of request, which is basis of other dominant cache coherence protocols. But this lack of global ordering causes starvation which is not happening in snooping/directory-based protocols. Token coherence protocol solves this problem by providing an emergency mechanism called persistent request. It enforces other processors in the competition for accessing same shared memory block, to give up their tokens to feed a starving processor. However, as the number of processors grows in a system, the frequency of starvation occurrence increases. In other words, the situation where persistent request occurs becomes too frequent to be emergent. As the frequency of persistent requests increases, not only the cost of each persistent matters since it is based on broadcasting to all processors, but also the increased traffic of persistent requests will saturate the bandwidth of multiprocessor interconnection network. This paper proposes a new request mechanism that defines order of requests to reduce occurrence of persistent requests. This ordering mechanism has been designed to be decentralized since centralized mechanism in both snooping-based protocol and directory-based protocol is one of primary reasons why token coherence protocol has advantage in terms of latency and bandwidth against these two dominant protocols.

Key Words : Token Coherent Protocol, Transient Request, Persistent Request, Ordering

1. 장 개 요

멀티프로세서 시스템에서 모든 프로세서는 자신이 가진 캐쉬와 공유메모리의 일관성을 유지할 수 있도록 하기 위하여 캐쉬 코히런스 프로토콜을 사용한다. 현재 스누핑(snooping) 기반 또는 디렉토리 기반의 캐쉬 코히런스 프로토콜이 주로 사용되고 있다. 스누핑 기반 프로토콜[1]은 메모리 접근 요청을 시스템 버스 상에 브로드캐스팅하여 모든 프로세서들에게 알리는 방식으로 해당 메모리 블록의 가장 최근 데이터를 가지고 있는 프로세서가 요청에 대하여 응답 하거나 그렇지 않은 경우 공유 메모리로부터 데이터를 복사한다. 즉 모든 프로세서가 버스를 공유하여 브로드캐스트 되는 메시지를 같은 순서로 감지한다. 스누핑 기반 프로토콜은 구현이 쉽고 효율적이지만 모든 프로세서가 버스를 공유함으로써 버스의 대역폭에 의하여 성능이 제한되며 확장성에 심각한 문제를 갖는다. 디렉토리 기반 프로토콜의 경우[1] 각각의 메모리 블록에 대하여 읽기/쓰기가 허용되는 프로세서에 대한 정보를 디렉토리에 저장 관리한다. 따라서 모든 요청은 디렉토리에 모여진 후 처리된다. 각 요청의 처리 순서는 디렉토리 도착 순서에 의하여 결정된다. 디렉토리 기반 프로토콜은 모든 요청이 디렉토리 거쳐 처리됨으로써 스누핑 기반 프로토콜에 비하여 지연시간(latency)이 길다. 토큰 코히런스 프로토콜은[2]] 토큰을 이용하여 데이터 일관성을 유지하는 방식으로 지연 시간이나 확장성 제한에 따른 문제를 해결할 수 있다. 그러나 다른 프로토콜들과 달리 요청의 처리 순서를 정하지 않음으로써 스타베이션(stravation) 이 발생하게 되며 이 문제를 해결하기 위하여 퍼시스턴트 요청 (persistent request)을 사용한다. 토큰 코히런스 프로토콜에서 프로세서는 메모리 블록에 접근하기 위하여 메모리 접근 요청을 보내는데 이를 트렌지언트(transient) 요청이라 한다. 프로세서가 타임아웃 등을 이용하여 스타베이션 가능성을 감지하면 트렌지언트 요청보다 우선순위가 높은 퍼시스턴트 요청을 보내 스타베이션 발생을 방지한다. 퍼시스턴트 요청은 스누핑 기반 프로토콜의 경우와 같이 요청이 브로드캐스팅 되므로 처리 비용이 크고 확장성 면에서 문제가 발생하게 된다. 또한 퍼스트턴트 요청을 한 프로세서가 모든 토큰을 수집하여 작업을 종료할 때까지 해당 블록에 대한 접근이 중단 되므로 다른 퍼시스턴트 요청의 발생 가능성을 높인다. 프로세서 수가 증가함에 따라 스타베이션을 방지하기 위한 퍼시스턴트 요청도 함께 증가하게 되며, 전체 요청 중 퍼시스턴트 요청이 차지하는 비중이 증가하게 된다. 극단의 경우 모든 요청이 퍼시스턴트 요청이 되어 스누핑 기반 프로토콜을 사용하는 것과 동일한 상황이

† 교신저자, 正會員 : 韓國電子通信硏究院 硏究員, 忠南大 情報通信工學科 碩士課程
E-mail : parkyk@etri.re.kr
* 正 會 員 : 忠南大 情報通信工學科 敎授 · 工博
接受日字 : 2005年 8月 2日
最終完了 : 2005年 9月 1日

발생하게 된다.

본 논문에서는 퍼시스턴트 요청의 수를 줄일 수 있도록 트렌지언트 요청 시 순서를 부여하는 방법에 대하여 기술하였다. 스누핑 기반 또는 디렉토리 기반 프로토콜과 같이 집중화된(centralized) 방식은 정확한 순서의 부여가 가능하지만 지연 시간이 길고 대역폭으로 인한 제한이 발생한다. 따라서 간단한 타임스탬프(timestamp) 카운터와 프로세서 아이덴티파이어(identifier)를 이용하여 경쟁 상태에 있는 프로세서들의 순서를 결정하도록 하였다. 새로운 트렌지언트 요청은 순서를 정확하게 부여하지는 않지만 퍼시스턴트 요청의 수를 줄일 수 있으므로, 프로세서의 수가 많은 시스템에서도 성능 저하 없이 토큰 토히런스 프로토콜을 사용할 수 있도록 한다. 2장에서는 토큰 코히런스 프로토콜에 대하여 기술하였으며 3장에서는 기존의 트렌지언트 요청에 순서를 부여하는 새로운 트렌지언트 요청 처리 방법에 대하여 설명하였다. 4장에서는 시뮬레이션을 이용하여 새로운 트렌지언트 요청 처리 방식의 성능을 평가하였고 5장에서는 향후 추가적인 연구가 필요한 항목에 대하여 기술하였다.

2. 장 토큰 코히런스 프로토콜

토큰 코히런스 프로토콜[3][4]은 시스템 초기화시 각 메모리 블록에 소유자 토큰을 포함하여 총 T개의 토큰을 할당하며 일관성 유지를 위하여 다음과 같은 규칙에 따라 토큰을 처리한다. 토큰은 필요시 프로세서 캐쉬에 저장 되거나, 코히런스 메시지에 포함되어 전달된다.

규칙 1. 각 메모리 블록은 총 T개의 토큰을 가지며, 그 중에는 하나는 소유자(owner) 토큰이 이어야한다.

규칙 2. 프로세서는 쓰기를 수행하기 위하여 해당 메모리 블록에 대한 T개의 토큰을 모두 가져야 한다.

규칙 3. 프로세서는 읽기를 수행하기 위하여 해당 메모리 블록의 토큰을 하나 이상 가져야 하며 유효한 데이터를 가져야 한다.

규칙 4. 코히런스 메시지가 소유자 토큰을 포함하고 있는 경우, 이 메시지는 반드시 유효한 데이터를 포함하여야 한다.

규칙 1부터 3은 여러 프로세서가 메모리 블록을 읽을 수 있지만 오직 하나의 프로세서만이 쓰기가 가능하도록 허용한다. 규칙 4는 모든 토큰을 가진 프로세서가 항상 유효한 데이터를 가질 수 있도록 한다.

그림 1은 프로세서 상태변환 및 토큰의 이동 과정을 보여준다. 모든 토큰을 가지고 있는 경우 프로세서의 상태는 M(modified) 이며 해당 메모리 블록에 대한 쓰기가 가능하다. 또한 해당 블록에 대한 읽기 요청이 있는 경우 하나의 토큰과 데이터를 보내주어야 한다. 데이터를 보내준 후 프로세서의 상태는 O(owned)로 변경되나 소유자 토큰을 가지고 있으므로 여전히 다른 프로세서들의 요청을 처리한다. 읽기를 요청한 프로세서는 하나의 토큰과 데이터를 수신하고 그 상태가 S(shared)로 변경된다. 아무런 토큰도 가지지 않은 상태는 I(invalid)이다. 프로세서는 쓰기 작업을 위하여 모든 토

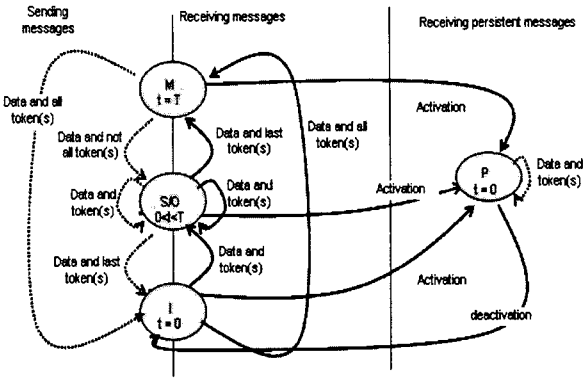

그림 1. 프로세서 상태 변환도

Fig 1. Process state transition diagram

큰을 수집하여야 하므로, 쓰기뿐만 아니라 읽기를 원하는 프로세서까지 경쟁에 참가하게 되는데, 경쟁으로 인하여 트렌지언트 요청이 실패한 경우, 프로세서는 정해진 횟수만큼 트렌지언트 요청을 반복하여 보냄으로써 메모리 접근을 재시도한다. 모든 시도가 실패한 경우 퍼시스턴트 요청을 보내 스타베이션 발생을 방지한다. 하나의 메모리 블록에 접근하고자하는 여러 프로세서 중 오직 하나의 프로세서만이 퍼시스턴트 요청을 보낼 수 있기 때문에 스타베이션 발생을 막을 수 있다. 각 프로세서들은 어느 프로세서가 퍼시스턴트 요청을 보냈는지를 기억하여 해당 메모리 블록의 토큰을 전달한다. 퍼시스턴트 요청을 한 프로세서가 필요한 모든 토큰을 수신하여 해당 메모리에 대한 작업을 수행한 후, 퍼시스턴트 요청을 해지한다. 퍼시스턴트 요청은 긴급 요청으로 항상 모든 프로세서에 전달되어야 하므로 요청을 모든 프로세서에게 브로드캐스트하며, 요청 처리시 해당 블록에 대한 모든 작업이 중지된다. 따라서 처리 비용이 크지만 스타베이션이 발생할 가능성이 있는 경우에만 사용되므로 프로세서의 수가 적은 시스템에서는 효율적으로 사용될 수 있다. 그러나 프로세서의 수가 증가함에 따라 퍼시스턴트 요청의 발생도 함께 증가하게 되어 효율성이 떨어지게 된다. 이러한 문제점을 해결하기 위하여 요청들 간의 처리 순서를 정할 수 있는 방법이 필요하며, 이 방법은 토큰 코히런스 프로토콜의 효율성을 유지할 수 있도록 브로드캐스트 되거나 집중화(centralized) 되어 처리되어서는 안된다.

3. 장 트렌지언트 요청 처리

토큰 코히런스 프로토콜 사용시 스타베이션이 발생하게 되는 이유 중 하나는 메모리 접근요청에 대한 순서를 부여하지 않기 때문이다. 새로운 트렌지언트 요청 방식에서는 경쟁이 발생하는 경우 타임스탬프와[5] 프로세서 아이덴티파이어를 이용하여 요청의 순서를 정의하도록 하였다. 각 프로세서들은 다른 프로세서로부터 메시지를 수신할 때 마다 그 값을 증가시키는 자신의 카운터를 가진다. 두 프로세서 간에 상호작용이 없었던 상태에서 경쟁이 발생하는 경우, 카운터 값의 비교를 통하여 부여한 순서가 실제 요청 순서를 나타내지는 못한다. 그러나 멀티프로세서 시스템에서 상호 작용의 수가 증가함에 따라 카운터를 이용한 순서 부여 방식이 스타베이

션의 발생을 줄일 수 있다. 경쟁 상태에 있는 프로세서들의 카운터 값이 같은 경우 프로세서의 아이덴티파이어를 이용하여 순서를 정한다. 프로세서 아이덴티파이어는 각 프로세서 고유의 값이므로 모든 요청에 순서를 정하는 것이 가능하다. 그림 2는 각 경쟁 시 카운터 값의 증가 방법을 나타낸다.

```
P0: Processor which received this request
P1: Processor which sent this request

if (P0.counter = P1.counter)
{
  P0.counter := P1.counter + 1;
}
else if (P0.counter > P1.counter)
{
  P0.counter:= P0.counter + 1;
}
```

그림 2. 카운터 처리
Fig 2. Counter processing algorithm

그림 3은 요청 처리 알고리즘을 나타낸다. 프로세서 P0가 프로세서 P1 으로부터 트렌지언트 요청을 수신하여 경쟁이 발생한 경우, P0는 각 프로세서의 카운터 값과 프로세서 아이덴티파이어를 이용하여 처리 순서를 결정하여야 한다. 두 프로세서 중 경쟁에서 이긴 프로세서는 필요한 만큼의 토큰을 수집한 후 해당 메모리 블록에 대한 작업을 완료한 후 토큰과 데이터를 경쟁에서 진 프로세서에 전달할 수 있도록 자신의 *waiter* 변수에 상대 프로세서의 아이덴티파이어를 저장한다. 반면 경쟁에서 진 프로세서는 자신이 가진 토큰과 추후 수신하게 되는 토큰을 전달하기 위하여 자신의 *winner* 변수에 상대 프로세서의 아이덴티파이어를 저장한다.

경쟁에서 이긴 프로세서는 작업을 마친 후 자신이 가진 토큰과 데이터를 *waiter* 변수에 저장된 프로세서에 보내기 전에 *RESET* 메시지를 보내주어야 한다. *RESET* 메시지는 경쟁에서 진 프로세서의 *winner* 변수를 수정하여 경쟁에서 진 프로세서가 자신이 수신한 토큰을 자동으로 다시 *winner* 프로세서에 전달하는 것을 막아준다. 또한 경쟁에서 이긴 프로세서는 더 이상 사용되지 않을 자신의 *waiter* 변수도 수정한다.

```
P0: Processor which received this request
P1: Processor which sent this request

if (P0 issued a request for same block)
{
   if (P0.counter < P1.counter)
   {
      waiter := P1;
      // P0 is the winner.
   } else if (P0.counter > P1.counter) {
      winner := P1;
       // P1 is the winner
      P0 gives its token(s) to P1;
   } else if (P0.counter == P1.counter) {
      if (P0.id < P1.id)
     {
      waiter := P1;
      // P0 is the winner.
     } else if (P0.counter > P1.counter) {
      winner := P1;
      // P1 is the winner
       P0 gives its token(s) to P1;
   }
```

그림 3. 접근 요청 처리
Fig 3. Access request processing algorithm

3.1 절 새로운 트렌지언트 요청 처리 예

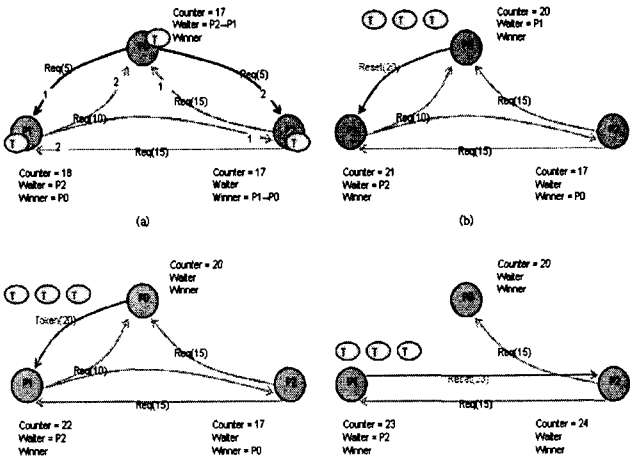

그림 4. 새로운 트렌지언트 요청 처리
Fig 4. New transient request prcessing

그림 4는 새로운 메모리 접근 요청 처리 방식의 처리 예이다. 프로세서 P0가 카운터 값이 5인 상태에서 특정 메모리 블록의 접근을 요청하였다. P1, P2 역시 해당 메모리 접근을 요청하였으며 요청시 카운터 값은 각각 10 과 15 이다. P0는 자신의 카운터 값이 5인 상태에서 요청을 보내고 P2로부터 Req(15), P1으로부터 Req(10)을 차례로 수신하였다. P0는 P2의 요청을 수신한 후 카운터를 16으로 증가시킨다. 수신한 P2의 카운터와 자신이 요청을 보낸 시점의 카운터를 비교하여 순서를 정한다. 자신의 요청을 보낸 시점의 카운터인 보다 수신한 요청의 카운터가 크므로 *waiter* 에 P2를 설정한다. P1으로부터 요청을 수신한 후 카운터를 17로 증가시킨다. P1의 카운터와 자신이 요청을 보낸 시점의 카운터와 *waiter*의 카운터를 비교하여 순서를 결정한다. 카운터 비교를 통하여 *waiter*가 P1으로 변경된다. 그림 4(a)는 이를 나타낸다. P0가는 토큰을 수신하여 해당 데이터에 대한 작업 수행을 마친 후, 자신의 *waiter* 값을 확인하여 대기하고 있는 프로세서가 P1임을 확인한다. P1에 토큰을 보내기 전에 *RESET* 메시지를 보내 P1 의 *winner*를 수정하도록 하여 자신이 보낸 토큰이 다시 되돌아오는 것을 막는다. 그림 4(b)에 이 과정을 나타내었다. 그림 4(c)는 P0가 P1에 토큰을 보내는 단계이다. 이때 P0는 자신의 *waiter* 값을 수정한다. P1은 자신에게 필요한 수의 토큰을 수신한 후 작업을 수행한다. P1가 P2간의 작업 처리 과정 역시 동일하다.

새로운 요청 처리 방식은 퍼시스턴트 요청의 발생하는 모든 경우를 제거하는 것이 아니라 퍼시스턴트 요청의 발생을 줄이는 방식이다. 예를 들어 프로세서 P3가 카운터 값 0을 가지고 있으며, 그림 3(c)의 상황에서 같은 메모리 블록에 접

근하고자 할 경우 P1의 *winner* 는 P3으로 수정될 것이다. 비록 P3이 P1이나 P2에 비하여 메모리 블록 접근 요청을 늦게 하였지만 카운터 값이 작아서 경쟁에서 이기게 된다. 이 경우 P1과 P2는 자신들이 요청한 토큰을 수신하는데 지연이 발생하며 스타베이션이 발생할 가능성이 생기게된다.

4. 장 실험 결과

토큰 코히런스 프로토콜은 모델링 언어인 SLICC 을 이용하여 구현하였다. SLICC 은 시뮬레이터인 Simics 및 메모리 타이밍 시뮬레이터인 Ruby에 대한 인터페이스를 제공한다. Simics는 4, 8, 16개 프로세서를 지원하며 각 프로세서, 메모리 및 네트워크 상태를 추적할 수 있다. 새로운 요청 처리 기능은 SLICC 으로 구현된 기존의 토큰 코히런스 프로토콜을 수정하여 구현하였다. 실험은 입출력 작업이 많은 gzip 을 이용하였다. 그림 5에 기존의 토큰 코히런스 프로토콜과 새로운 요청 처리 기능을 지원하는 토큰 코히런스 프로토콜을 사용하는 경우 발생하는 퍼시스턴트 요청의 수를 비교하였다. 4개의 프로세서를 갖는 경우 차이를 거의 찾아 볼 수 없으나, 8개 와 16개의 프로세서를 갖는 경우 20% 정도의 차이가 나는 것을 확인할 수 있다. 즉 프로세서 수가 증가함에 따라 퍼시스턴트 요청의 수가 기하급수적으로 증가하므로 이에 따라 성능의 차이도 더욱 커지게 된다. 그림 6은 4, 8, 16 프로세서 시스템에서 테스트 케이스를 수행하는 경우 발생하는 트렌지언트 요청을 처리하는데 소요되는 평균 CPU 싸이클을 나타낸다. 기존의 트렌지언트 요청을 처리하는데 소요된 CPU 싸이클을 기준으로 새로운 트렌지언트 요청 처리에 소요된 CPU 싸이클을 표시하였다. 프로세서의 수가 4, 16인 경우 기존의 토큰 코히런스 프로토콜이 새로운 요청 처리 방식을 이용한 토큰 코히런스 프로토콜 보다 높은 성능을 나타내었다. 그러나 기존의 토큰 코히런스 프로토콜이 이러한 사실로 부터 얻을 수 있는 성능상의 이점은 그림에서 보는 것과 같이 미미하다. 그림 6에서도 알 수 있듯이 프로세서의 수와 하나의 트렌지언트 요청 처리에 필요한 CPU 싸이클 간에 연관성이 없으며, 새로운 트렌지언트 요청 처리에 소요되는 CPU 싸이클과 기존의 방식과의 차이는 3%미만이다. 따라서 처리 비용이 높은 퍼시스턴트 요청을 줄일 수 있는 새로운 요청 처리 방식을 적용함으로써 효율성을 높일 수 있다.

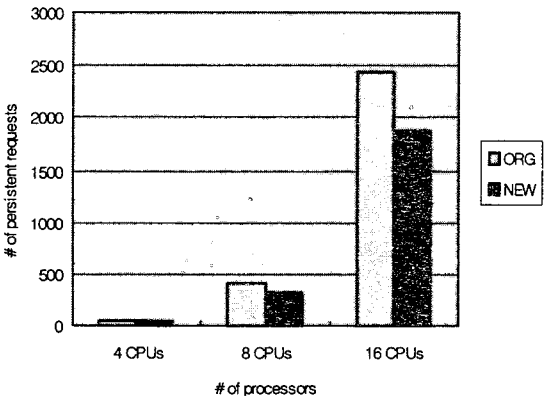

그림 5. 퍼시스턴트 요청 수
Fig 5. Number of persistent requests

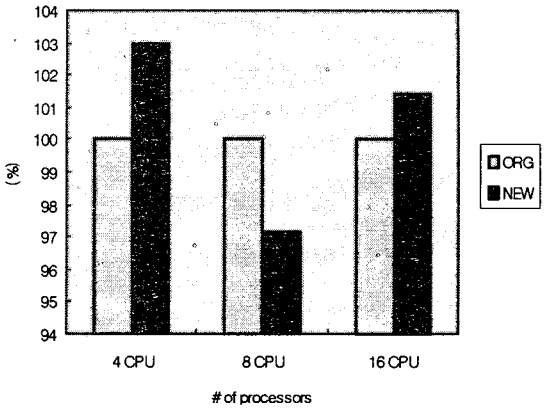

그림 6. 트렌지언트 요청 처리 CPU 싸이클
Fig 6. Number of cycles performed with transient request

그림 7에서 프로세서의 수가 증가함에 따라 프로세서를 연결하는 네트워크가 점차 퍼시스턴트 요청으로 포화되어 감을 알 수 있다. 8개의 프로세서 시스템에서 기존의 토큰 코히런스 프로토콜을 사용하는 경우 전체 약 3.83%가 퍼시스턴트 요청이었으며, 새로운 요청 처리 방식을 지원하는 토큰 코히런스 프로토콜을 사용한 경우 전체 요청 중 2.7% 만이 퍼시스턴트 요청임을 알 수 있다. 16 프로세서 시스템인 경우 기존의 프로토콜을 사용한 경우 전체 요청 중 퍼시스턴트 요청이 5.63%를 차지하였으며, 새로운 요청 처리 방식을 적용한 경우 퍼시스턴트 요청이 차지하는 비율이 3.67%로 감소하였다. 본 논문에서는 16 프로세서 시스템까지 실험하였으나, 프로세서의 수가 증가함에 따라 전체 요청 중 퍼시스턴트 요청이 차지하는 비율을 더욱 감소 시켜 시스템 성능을 향상시킬 수 있다.

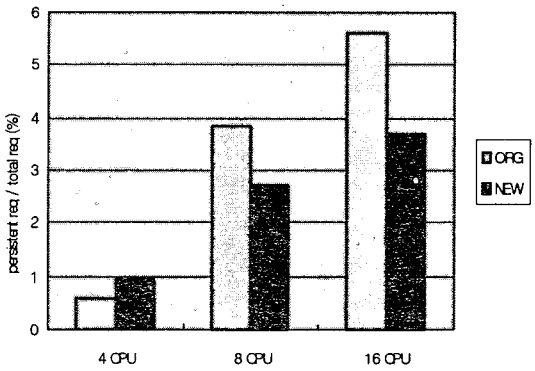

그림 7. 퍼시스턴트 요청 비율
Fig 7. Percentage of persistent requests

그림 8에서는 기존의 토큰 코히런스 프로토콜과 새로운 토큰 코히런스 프로토콜을 사용한 경우의 응답시간을 비교하였다. 기존의 토큰 코히런스 프로토콜을 사용한 경우의 응답시간을 기준으로 새로운 토큰 코히런스 프로토콜을 사용한 경우의 응답 시간을 비교하여 나타내었다. 응답시간 면에 있어서 성능 향상은 크지 않았다. 그러나 그림 5에서 나타난 것과 같이 퍼시스턴트 요청의 수가 많아질수록 시스템의 응답시간도

보다 많이 단축된 것을 알 수 있다. 따라서 보다 많은 프로세서를 가진 시스템에 적용하는 경우 응답시간은 더욱 단축될 수 있으며 특히 퍼시스턴트 요청 증가로 인하여 프로세서 간의 네트워크가 포화 상태에 빠지는 경우를 줄일 수 있다.

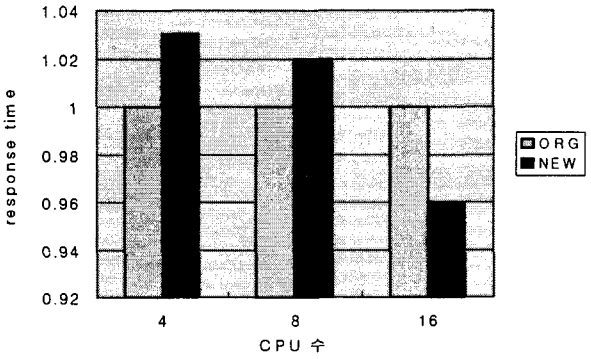

그림 8. 응답 시간 비율
Fig 8. Ratio of response time

5. 결 론

토큰 코히런스 프로토콜을 사용하는 경우 프로세서의 수가 증가함에 따라 브로드캐스트 되는 퍼시스턴트 요청의 수도 함께 증가하여 확장성에 문제가 발생하게 된다. 본 논문에서는 확장성을 향상 시킬 수 있도록 퍼시스턴트 요청의 수를 줄일 수 있는 새로운 트렌지언트 요청 처리 방식을 제시하였다. 추후 카운터 값이 보다 정확한 요청 순서를 나타낼 수 있도록 하는 순서 부여 방법에 대한 연구가 필요하다. 보다 정확한 순서 부여를 통하여 스타베이션이 발생하는 횟수를 줄일 수 있을 것이다. 또한 퍼시스턴트 요청이 발생하게 되는 경우에 대한 보다 자세한 연구를 수행하여 스타베이션의 발생 이유 및 발생 빈도를 파악함으로써 퍼시스턴트 요청의 발생을 줄일 수 있을 것이다. 효율성을 보다 높이기 위하여 프로세서의 수에 상관없이 퍼시스턴트 요청의 발생을 일정 수준 아래로 유지 할 수 있는 방법에 대한 연구도 필요하다.

참 고 문 헌

[1] Mark S. Papmarcos and Janak H. Patel, A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories, ISCA Volume 12, Issue 3, pp. 348-354, 1984.

[2] Milo M. K. Martin et al, Token Coherence: A New Framework for Shared-Memory Multiprocessors, IEEE Micro Vol. 23, No. 6, pp. 108-116, Nov/Dec 2003.

[3] Milo M. K. Martin et al, Token Coherence: Decoupling Performance and Correctness, ISCA Volume 31 Issue 2, pp. 182 -193, 2003

[4] Milo M.K. Martin, Token Coherence, University of Wisconsin-Madison, 2003

[5] George Coulouris et al, Distributed Systems: Concepts and Design 3rdEd., Chapter 10 Time and Global States, Addison-Wesley, 2001.

저 자 소 개

박 윤 경 (朴 潤 慶)

1987년 고려대학교 수학교육과 졸업. 1987~현재 한국전자통신연구원 연구원. 2003년~현재 충남대학교 정보통신공학과 석사과정

Tel : 042-860-4954
Fax : 042-860-5545
E-mail : parkyk@etri.re.kr

김 대 영 (金 大 榮)

1975년 2월 서울대학교 전자공학과 학사. 1977년 2월 한국과학기술원 통신공학과 석사. 1983년 2월 한국과학기술원 통신공학과 박사. 1983년~현재 충남대학교 공과대학 교수

Tel : 042-821 - 6862
Fax : (042) 823 - 5586
E-mail : dykim@cnu.ac.kr