논문 2017-12-22

하이브리드 메모리 큐브 (HMC) 시스템의 고속 직렬 링크 (SerDes)를 위한 모델링 및 성능 분석

(Modeling and Analysis of High Speed Serial Links (SerDes) for Hybrid Memory Cube Systems)

전 동 익, 정 기 석*
(Dong-Ik Jeon, Ki-Seok Chung)

Abstract : Various 3D-stacked DRAMs have been proposed to overcome the memory wall problem. Hybrid Memory Cube (HMC) is a true 3D-stacked DRAM with stacked DRAM layers on top of a logic layer. The logic die is mainly used to implement a memory controller for HMC, and it is connected through a high speed serial link called SerDes with a host that is either a processor or another HMC. In HMC, the serial link is crucial for both performance and power consumption. Therefore, it is important that the link is configured properly so that the required performance should be satisfied while the power consumption is minimized. In this paper, we propose a HMC system model included the high speed serial link to estimate performance accurately. Since the link modeling strictly follows the link flow control mechanism defined in the HMC spec, the actual HMC performance can be estimated accurately with respect to each link configuration. Various simulations are conducted in order to deduce the correlation between the HMC performance and the link configuration with regard to memory utilization. It is confirmed that there is a strong correlation between the achievable maximum performance of HMC and the link configuration in terms of both bandwidth and latency. Therefore, it is possible to find the best link configuration when the required HMC performance is known in advance, and finding the best configuration will lead to significant power saving while the performance requirement is satisfied.

Keywords : Hybrid memory cube (HMC), High speed serial link, SerDes, HMC link modeling, Link analysis

Ⅰ. 서 론

일반적으로 컴퓨터 구조에서 메인 메모리로 널리 사용되고 있는 DRAM (dynamic random access memory)은 높은 성능과 대용량의 요구를 만족하기 위해 다양한 모습으로 발전하고 있다. 예를 들어, 그래픽 프로세서를 위한 GDDR SDRAM (graphics DDR SDRAM)과 저전력 모바일 시스템을 위한 LPDRAM (low-power DRAM), 그리고 매우 낮은 메모리 접근 대기 시간을 요구하는 시스템을 위한 RLDRAM (reduced-latency DRAM) 등처럼 다양한 모습의 DRAM이 출시되고 있다. 이런 노력에도 불구하고 DRAM에 요구하는 성능이 더 커지면서 최근에는 TSVs (through-silicon vias)를 이용한 3D 구조의 혁신적인 DRAM이 제안되고 있다 [1-4]. 대표적인 3D DRAM으로는 Wide I/O와 HBM (high bandwidth memory), HMC (hybrid memory cube)가 있다. Wide I/O는 삼성 전자에서 처음으로 제안한 3D DRAM으로 Wide I/O standard는 JEDEC에서 2011년에 발표되었다 [5].

*Corresponding Author (kchung@hanyang.ac.kr)
Received: Apr. 3 2017, Revised: July 11 2017, Accepted: July 12 2017.
D.I. Jeon, K.S. Chung: Hanyang University
※ 이 논문은 2015년도 정부 (교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (NRF-2015R1D1A1A09061079)

ⓒ IEMEK J. Embed. Sys. Appl. 2017 Aug. 12(4) 193-204
ISSN : 1975-5066
http://dx.doi.org/10.14372/IEMEK.2017.12.4.193

Wide I/O는 로직과 DRAM이 수직적으로 쌓여있으면서 128개의 넓은 I/O 채널을 가지고 있어 낮은 클럭 주파수에서도 높은 대역폭을 지원한다. 따라서 Wide I/O의 낮은 소비 전력 덕분에 모바일 임베디드 시스템에 각광받고 있다. 실제로 4 채널 Wide I/O는 200 MHz 클럭 주파수에서 하나의 I/O 핀 당 0.78 mW/Gbps의 전력이 소비되는 것을 확인하였다. 이는 LPDDR2 소비 전력의 8.96%로 Wide I/O의 특징을 잘 보여준다 [5]. 더구나 최근에는 Wide I/O 2가 개발되고 있어 Wide I/O에서 많은 부분을 개선하고 있다 [6]. HBM은 약간 독특한 구조의 3D DRAM으로써 대역폭을 극대화시킨 메모리이다. HBM은 AMD와 SK Hynix에서 2010년에 제안되었고 JEDEC에서 2013년에 표준으로 채택되었다 [7]. HBM은 Wide I/O와 다르게 인터포저를 이용한 2.5D 구조를 사용하고 있다. GPU (graphic processing unit)와 같이 높은 메모리 대역폭을 요구하는 장치들은 하나의 보드에서 많은 DRAM 칩들을 병렬적으로 구성하여 데이터 채널을 넓히지만, 더 많은 DRAM 칩을 병렬적으로 구성하기엔 물리적으로 한계가 있다. 따라서 HBM은 GPU와 메모리의 데이터 채널을 인터포저 레이어에서 넓혀 메모리 대역폭을 개선하였다. 반면에 2012년 Micron에서 제안한 HMC는 진정한 3D 구조를 가지고 있다 [8]. Wide I/O와 HBM과 다르게 HMC는 로직 다이위에 4개의 DRAM 다이를 적층한 구조이다. HMC는 이론적으로 로직 다이와 DRAM 다이 사이의 메모리 대역폭은 최대 320 GB/s를 가지는 것으로 알려져 있다. HMC는 적층된 로직 다이와 DRAM 다이를 기둥 형태로 수직적으로 나눈 vault로 구성되어 있다. 각각 vault의 로직 다이에는 기존 메모리 구조에서 메모리 request들을 스케줄링하고 DRAM 명령어로 변환하는 메모리 컨트롤러와 유사한 vault controller가 내장되어 있어 vault들은 독립적으로 동작할 수 있다. 따라서 많은 메모리 request들을 각각의 vault내에서 병렬적으로 처리할 수 있고 여러 개의 HMC들은 서로 로직 다이로 연결하여 전체 메모리 크기를 증가시킬 수 있다. 또한 HMC는 사칙 연산과 로직 연산을 로직 다이에서 지원하기 때문에 PIM (processing-in-memory) 구조로서 많은 연구가 진행 중에 있다 [9, 10].

II. 연구 배경 및 동기

HMC는 CPU와 DRAM 사이에 로직 다이가 있어 기존 메모리 구조와 다르다. 그림 1은 HMC

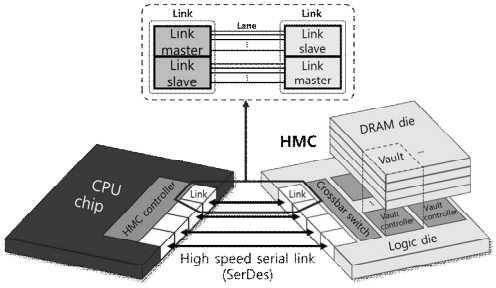


그림 1. HMC와 고속 직렬 링크의 구조
Fig. 1 The structure of HMC and high speed serial link

구조 및 CPU와 HMC 사이에 SerDes (serializer/ deserializer)라고 불리는 고속 직렬 링크 (high speed serial link)의 구조 모습이다. HMC의 고속 직렬 링크는 여러 개의 링크 (link)로 구성이 되어 있고 각각의 링크는 여러 개의 레인 (lane)으로 반대편 링크와 연결이 된다. 하나의 링크에서 데이터를 전송하게 되면 해당 데이터는 serializer를 통해 bit 스트링으로 변환되고 여러 개의 레인을 통해 전달된다. 레인은 단방향성을 가지고 있으며 한 번에 한 bit를 전송하다. 고속 직렬 링크는 클럭 신호와 데이터 신호를 독립적으로 사용하는 다른 일반적인 통신과 다르게, 레인에서 데이터 스트링사이에 클럭 신호를 끼워 넣어서 같이 보낸다. 따라서 HMC에 전원이 인가되면 보내는 곳과 받는 곳의 링크 클럭 동기화가 필요하다. 심지어 보내야할 데이터가 없더라도 링크는 클럭 동기화를 유지하기 위해 NULL 패킷을 보내야 한다. 결과적으로 고속 직렬 링크는 소비 전력이 매우 크고 심지어 HMC 전체 소비 전력에서도 가장 큰 비중을 차지하고 있다. 많은 논문들이 HMC 시스템에서 고속 직렬 링크 큰 소비 전력을 문제 삼고 있다 [11-14]. 특히 한 논문에서는 실험을 통해 HMC에서 고속 직렬 링크가 차지하는 소비 전력이 62%인 것을 보여줌으로써 저전력 링크 제어 방법의 중요성을 강조하고 있다 [15]. 또한 28 nm 공정으로 설계된 고속 직렬 링크 IP의 화이트 보고서에 따르면 종단 부분을 포함해서 4.3 mW/Gbps 만큼 전력을 소비하는 것을 알 수 있다 [16]. 따라서 링크의 속도가 높아질수록 소비 전력 또한 비례하므로 전력을 고려한 효율적인 링크 관리는 매우 중요하다.

HMC spec에 따르면 HMC는 링크의 개수, 전송 속도 그리고 링크의 전송 폭 등 다양한 링크 설정

을 제공하고 있다 [17]. 즉, 링크의 전송 속도를 낮추거나 부분적으로 활성화시켜 소비 전력을 줄일 수 있다. 그러나 링크는 CPU와 HMC사이에 메모리 request/response와 데이터를 전송하는 중요한 부분이기 때문에 부분적으로 링크를 활성화시키면 시스템 성능에 영향을 끼칠 수 있다. 따라서 시스템 성능 요구상황에 맞춰 적절한 링크 설정이 필수적이다. 링크로 보내지는 데이터들은 serializer에서 bit 스트링으로 변환되고 레인을 통해 한 bit씩 전송된다. 따라서 고속 직렬 링크는 데이터와 제어 정보를 같이 보내는 in-band 통신으로 모든 데이터들은 패킷으로 변환되어 전송된다. 이 패킷에는 데이터 전송의 시작점인 링크 마스터 (link master)와 전송 끝점인 링크 슬레이브 (link slave)의 현재 상황을 서로 파악하기 위한 다양한 링크 정보들이 포함되어 있다. HMC spec에서는 패킷의 프로토콜, 즉 사용되는 링크 정보와 정보들의 구성 및 배치들을 자체적으로 정의하고 있다 [17]. 이런 특징들을 가진 HMC를 정밀하게 모델링한 시뮬레이터가 없어 다양한 링크 설정에 따른 HMC 성능을 파악하기 위해서는 실제 HMC 실험용 보드에서 직접 실험을 하지 않는 한 매우 어렵다. 특히, 현재 연구되고 있는 많은 메모리들은 다른 특성을 가지고 있어 각각의 메모리 특성을 반영한 시뮬레이터 개발이 필요하다 [18]. 그러므로 이 논문에서는 HMC spec 2.1을 기준으로 HMC 내부 구조와 고속 직렬 링크의 동작을 모델링하여 HMC의 성능을 시뮬레이션 해보았다. 모델링된 HMC는 CasHMC (cycle-accurate simulator for hybrid memory cube)라는 이름으로 https://github.com/estwings57/CasHMC에 오픈 소스로 공개되어 있다 [19]. 이 논문의 나머지 부분에서는 모델링한 고속 직렬 링크의 동작 방법과 링크 설정에 따른 실험 결과를 보여줄 것이다.

Ⅲ. 고속 직렬 링크 (SerDes)의 구현 및 모델링

고속 직렬 링크는 단방향 in-band 통신이므로 보내는 쪽과 받는 쪽에서 통신을 제어할 로직이 필요하다. 통신을 보내는 쪽의 로직은 링크 마스터, 받는 쪽은 링크 슬레이브라고 부른다. 그림 2는 CPU (requester)와 HMC (responder)의 사이에서 데이터를 전송하는 고속 직렬 링크의 구조이다. 그림 2에서는 링크 마스터를 줄여 LM, 링크 슬레이브를 줄여 LS라고 표현하였다. Downstream 링크는

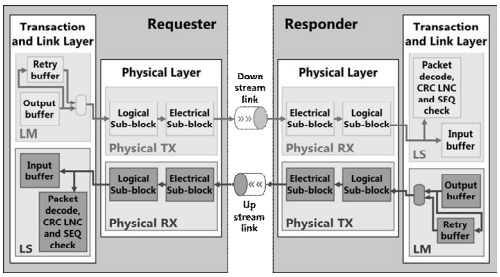
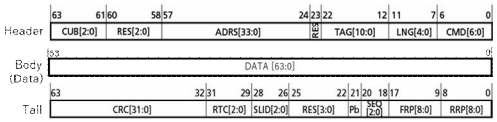
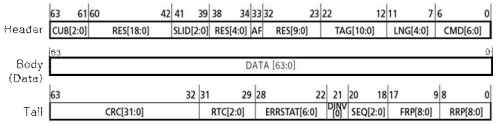


그림 2. 고속 직렬 링크의 내부 구조 [17]
Fig. 2 The internal structure of high speed serial link [17]



(a) memory request packet layout



(b) memory response packet layout

그림 3. HMC 패킷의 레이아웃 [17]
Fig. 3 HMC packet layout [17]

CPU에서 HMC로, upstream 링크는 HMC에서 CPU 방향으로의 데이터 전송을 뜻한다. 그림과 같이 하나의 트랜잭션 링크 레이어에는 링크 마스터와 링크 슬레이브가 존재한다. 같은 레이어에 있는 링크 마스터, 슬레이브를 각각 이웃 링크 슬레이브, 이웃 링크 마스터로 부른다. 링크 마스터는 보낼 패킷의 다양한 링크 정보들을 업데이트하고 링크 전송 상태에 따라서 링크 제어 패킷을 만들며 패킷의 전송을 제어한다. 모든 패킷은 8 bytes의 header와 8 bytes의 tail을 갖고 있으며 그림 3과 같이 HMC spec에 따라 사용되는 링크 정보와 배치가 정의되어 있다 [17]. 링크 슬레이브는 패킷을 전송받고 전송 오류를 확인하며 일부 링크 정보들을 패킷에서 추출하는 역할을 한다. 이 추출된 링크 정보들을 이웃 링크 마스터에서 새로운 링크 제어 패킷에 업데이트 되어 반대 방향으로 보내진다. HMC spec에서는 다음과 같이 NULL, PRET (retry pointer return), TRET (token return), IRTRY (INIT retry) 4개의 링크 제어 패킷을 정의하고 있다

[17]. 추출되는 링크 정보와 링크 제어 패킷, 링크의 동작 원리는 다음 장에서 자세히 다룰 것이다. 본 논문의 HMC 모델링은 HMC spec 2.1에서 정의하고 있는 링크 동작과 패킷의 정보들을 포함한 고속 직렬 링크의 모든 구조를 그대로 구현하였다.

1. 전방향 (forward) 링크 패킷 전송

고속 직렬 링크에서 전송되는 패킷들은 실제 메모리 request/response들이 변환되어 DRAM 메모리를 제어하는 패킷들과 링크 제어를 위한 링크 제어 패킷으로 나누어진다. 이 중에서 실제 메모리 request/response에서 변환된 패킷들의 전송 방향을 전방향 (forward)로 정의하였다. 전방향으로 패킷이 전송될 때 일부 링크 정보들이 추출되어 반대 방향으로 전달되는데, 이를 후방향 (backward)이라고 정의하였고 이에 대한 설명은 다음 장에서 이어나간다. 그림 4는 HMC 모델링의 고속 직렬 링크에서 전방향으로 패킷이 전송되는 순서도이다. 순서도에 따라 링크의 동작을 설명하기 위해 CPU에서 HMC로 메모리 read request가 발생했다고 가정해보자. 우선 CPU 쪽의 HMC controller에서 메모리 read request가 패킷으로 변환이 되고 이를 HMC로 보내기 위해 여러 개의 링크 중에서 적절한 링크를 선택하여 링크 마스터로 보내지게 된다. 고속 직렬 링크는 매우 빠른 속도로 데이터를 전송하기 때문에 통신상의 오류가 발생할 수 있다. 따라서 전송하려고 하는 패킷을 복사하여 retry buffer에 저장하고 오류가 발생하였을 경우 retry buffer에 저장된 패킷을 다시 전달하게 된다. 만약 retry buffer가 가득 찼을 경우 패킷은 buffer에 여유 공간이 생길 때 까지 기다렸다가 retry buffer에 자기 자신을 복사한 뒤 다음 순서로 넘어간다. Retry 포인터는 방금 retry buffer에 저장된 패킷이 어디에 위치하는지 알려주는 포인터이다. 따라서 패킷이 retry buffer에 저장되면 retry 포인터 값을 FRP에 업데이트 하게 된다. HMC의 고속 직렬 링크에서는 2가지의 오류 감지 기법을 사용한다. 첫 번째는 SEQ (sequence number)로 일반 패킷이 전송될 때 마다 링크 마스터와 링크 슬레이브에서 SEQ 값을 1 증가시켜 이를 비교하는 방법이다. 링크 마스터에서는 패킷 전송 준비가 끝났으면 링크 마스터의 SEQ 값을 1 증가시키고, 그 값을 패킷의 tail에 있는 SEQ에다가 업데이트 시킨다. 그리고 패킷 전송이 완료 되면 링크 슬레이브에서는 자신의 SEQ 값에서 1을 증가시키고, 그 값을 패킷 tail에 있는 SEQ 값과 비교하여 오류 발생 여부를 결정한다. 다른 오류 감지 기법으로는 CRC (cyclic redundancy check)가 있다. CRC는 오류 발생 여부를 판단하는 확인용 코드로써 패킷을 보내기 전 계산된 CRC 값과 전송이 끝난 후 계산된 CRC 값을 비교하여 오류를 감지한다. 링크 마스터는 패킷을 전송하기 전에 해당 패킷의 CRC 값을 계산하여 패킷 tail의 CRC에 값을 업데이트 하고 링크 슬레이브에서 새로운 CRC 값을 계산하여 패킷의 CRC 값과 비교한다. 따라서 링크 마스터에서는 SEQ와 CRC 값을 업데이트 하고 나서 패킷을 링크 슬레이브로 전송하다. 이 때 링크 마스터에서는 token counter 값을 패킷의 16 bytes 단위 크기만큼 감소시킨다. Token counter는 링크 슬레이브가 전송 받은 패킷을 저장할 수 있는 16 bytes 단위의 공간을 뜻한다. 패킷은 그림 3과 같이 body (데이터)가 없을 경우 header와 tail로만 구성되는 가장 작은 크기인 16 bytes 크기를 가지므로 token counter에서는 16 bytes 단위를 사용한다. 고속 직렬 링크는 단방향성을 가지고 한 번에 한 bit만 전송하는 단순한 구조로 되어 있기 때문에 링크 마스터는 링크 슬레이브가 전송 받을 수 있는 상태인지 스스로 파악해야 한다. 따라서 token counter 값과 전송할 패킷의 크기를 비교하여 링크로 전송 여부를 결정한다. 링크 슬레이브에서 패킷을 전달 받으면 앞서 말한 SEQ와 CRC 기법을 통하여 오류를 감지한다. 만약 오류가 발생한다면 이웃 링크 마스터는 error abort mode로 전환하여 retry buffer에 저장된 패킷들을 재전송하게 만든다. 이 부분에 대한 자세한 설명은 뒷장의 링크 재전송 부분에서 다룰 것이다. 전송 받은 패킷이 오류가 없다면 FRP 값을 추출하여 이웃 링크 마스터로 FRP 값을 전달하고 해당 패킷은 메모리로 전달되게 된다. 메모리로 전달된 패킷의 크

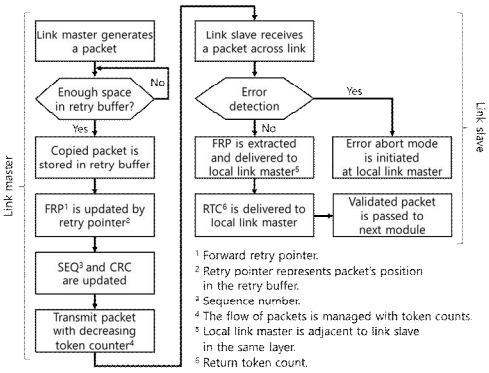


그림 4. 전방향 패킷 전송 순서도
Fig. 4 Packet transmission sequence flowchart in the direction of the forward link

기만큼 링크 슬레이브에 여유 공간이 생겼으므로 이를 알려주기 위해 이웃 링크 마스터로 패킷의 크기를 RTC (return token count) 값으로 전달한다. 패킷에서 추출된 FRP와 RTC의 처리는 다음 장에서 자세히 다룰 것이다.

2. 후방향 (backward) 링크 패킷 전송

후방향 링크는 앞에서 설명한 메모리 request/response 패킷이 전달되는 방향의 반대 방향이다. 패킷에서 추출된 링크 정보들은 올바른 링크 전송을 제어하기 위해서 해당 패킷을 처음 만든 링크 마스터로 전달되어야 한다. 앞서 언급하였듯이 두 개의 링크 정보 (FRP와 RTC)들은 링크 슬레이브에서 추출되어 같은 레이어에 있는 이웃 링크 마스터로 전달이 된 뒤 후방향으로 전송된다. 그림 5와 그림 6은 FRP와 RTC가 HMC 모델링에서 후방향으로 전송되는 순서도이다. 첫 번째로 FRP는 패킷이 retry buffer에 저장된 위치를 의미한다. 만약 패킷이 성공적으로 전송이 되었다면 그림 5와 같이 retry buffer에 저장된 복사 패킷을 제거하기 위해 패킷이 만들어진 링크 마스터로 FRP를 돌려줘야 한다. 링크 슬레이브에서 FRP가 추출되어 이웃 링크 마스터로 전달되면 FRP는 RRP (return retry pointer)로 변환이 된다. 이 때 링크 마스터는 현재 전송을 기다리고 있는 패킷이 있는지 확인한다. 모든 패킷의 tail에는 RRP가 있어 전송을 기다리고 있는 패킷의 tail에 RRP 값을 업데이트하여 전달할 수 있다. 만약 전송될 패킷이 없다면 링크 마스터는 RRP 값을 전달할 PRET (retry pointer return) 링크 제어 패킷을 새롭게 만든다. PRET 패킷은 retry buffer에 저장하지 않기 때문에 FRP와 SEQ, CRC 값들을 업데이트 하지 않는다. 링크 슬레이브에서 RRP 값이 포함된 패킷을 전달받으면 RRP 값을 추출 한 뒤 이웃 링크 마스터로 그 값을 전달한다. 그리고 RRP가 가리키고 있는 retry buffer의 복사된 패킷을 제거하여 다른 패킷의 전송을 가능하게 한다. RRP 값이 PRET 패킷을 통해 전달되었다면 PRET 패킷 또한 목표를 다하였기 때문에 제거되게 된다.

다른 링크 정보인 RTC는 앞서 설명하였듯이 패킷이 오류 없이 전송을 완료하여 링크 슬레이브에서 메모리로 전달되는 패킷의 16 bytes 단위 크기이다. 따라서 그림 6과 같이 RTC 값은 패킷이 만들어진 링크 마스터로 링크 슬레이브의 여유 공간을 알려주기 위해 되돌아가야 한다. 즉, 링크 마스터의 감소된 token counter 값을 되돌려주는 역할

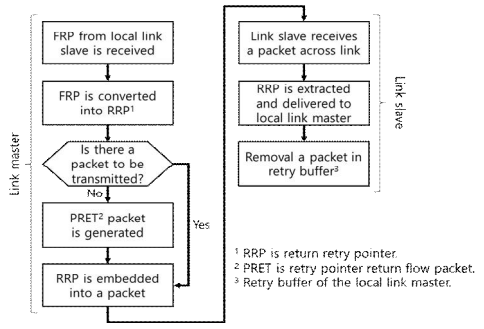


그림 5. 후방향 FRP 링크 정보 전송 순서도
Fig. 5 FRP transmission sequence flowchart in the direction of the backward link

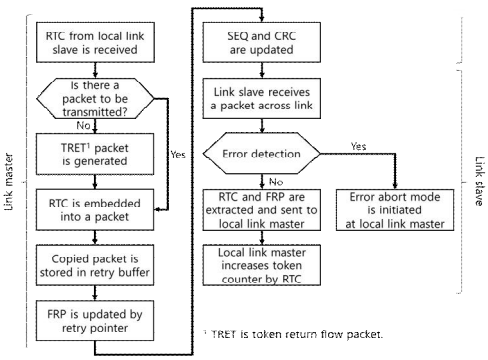


그림 6. 후방향 RTC 링크 정보 전송 순서도
Fig. 6 RTC transmission sequence flowchart in the direction of the backward link

을 한다. RTC 값이 링크 슬레이브에서 이웃 링크 마스터로 전달되면 RRP의 동작과 마찬가지로 RTC 값을 전달할 패킷이 있는지 확인한다. 만약 전송대기 중인 패킷이 없다면 TRET (token return) 링크 제어 패킷을 새롭게 만들어 RTC 값을 전달한다. TRET 패킷은 PRET 패킷과 다르게 retry buffer에 저장되므로 FRP와 SEQ, CRC 값들을 현재 상태에 따라 업데이트 된다. 링크 슬레이브가 RTC 값이 포함된 패킷을 전달받으면 SEQ와 CRC를 이용하여 전송 오류가 발생하였는지 확인한다. 오류가 발생하였으면 이웃 링크 마스터를 error abort mode로 동작시키고, 오류가 없다면 RTC와 FRP 값을 추출하여 이웃 링크 마스터로 전달한다. 이웃 링크 마스터는 전달받은 RTC 값만큼 token counter를 증가시키고 FRP는 RRP 값으로 변환시켜 다시 반대 방향으로 RRP 값을 전달한다.

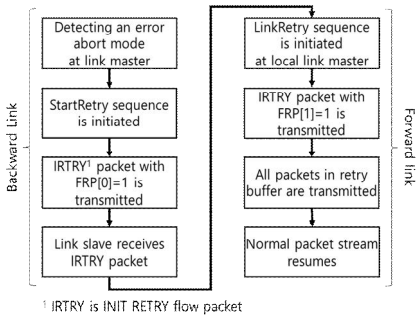


그림 7. 링크 재전송 순서도
Fig. 7 Link retry sequence flowchart

3. 링크 재전송

HMC의 고속 직렬 링크에서 하나의 레인은 최대 30 Gb/s로 매우 빠른 속도를 통해 통신을 한다. 따라서 링크에서는 전송 오류가 발생할 수밖에 없다. HMC에서는 SEQ와 CRC를 이용하여 오류를 감지하고 있다. 링크 슬레이브에서 오류가 감지되면 이웃 링크 마스터를 error abort mode로 동작시켜 링크의 재전송 단계로 들어가게 된다. 본 논문의 HMC 모델링은 BER (bit-error rate)을 이용하여 시뮬레이션 상의 링크 오류를 시뮬레이션 하였다. 그림 7은 HMC 모델링에서 오류가 발생하였을 때 동작하는 링크 재전송의 순서도이다. 링크 마스터가 error abort mode로 전환되면 오류가 발생한 패킷의 전송 방향과 반대 방향으로 링크의 오류를 알려주는 StartRetry 단계가 시작된다. StartRetry 단계에서는 일반 패킷의 전송을 잠깐 멈추고 FRP[0] 값이 1을 가지는 IRTRY (INIT retry) 패킷을 설정한 개수만큼 먼저 보내게 된다. 링크 슬레이브에서 FRP[0] 값이 1인 IRTRY 패킷을 받으면 이웃 링크 마스터를 LinkRetry 단계로 전환하여 오류가 생긴 패킷들을 재전송 하게 된다. 우선 LinkRetry 단계로 전환되면 일반 패킷의 전송을 모두 멈추고 FRP[1] 값이 1을 가지는 IRTRY 패킷을 설정한 개수만큼 보내게 된다. 그러고 난 뒤, retry buffer에 저장되어 있는 모든 패킷들을 저장된 순서대로 다시 보내게 된다. retry buffer에 있던 모든 패킷이 전송이 되면 이전에 멈춰있던 일반 패킷들을 다시 전송하게 된다.

IV. HMC 모델링의 정확성 및 필요성

HMC의 고속 직렬 링크는 모든 데이터를 패킷으로 변환하여 전송하기 때문에 링크 설정에 따라 이론적으로 계산된 성능은 실제 성능과 매우 큰 차이가 있다. 메모리 request/response 들이 8 bytes header와 8 bytes tail로 변환이 되고 전송되는 과정에서 여러 개의 링크 제어 패킷이 생성될 수 있기 때문에 단순한 수치만으로 링크의 성능을 예측하는 것은 불가능하다. 예를 들어 16 bytes 메모리 read request가 HMC로 발생했다고 가정해 보자. 실제 메모리에서 가져오는 데이터의 크기는 16 bytes임에도 불구하고 고속 직렬 링크로 전달되기 위해서는 최소 32 bytes header와 tail이 (메모리 request 패킷의 16 bytes header와 tail 그리고 메모리 response 패킷의 16 bytes header와 tail) 필요하다. 게다가 header와 tail로만 구성된 링크 제어 패킷 (NULL, PRET, TRET, IRTRY)또한 필요에 따라서 생성될 수도 있다. 따라서 고속 직렬 링크를 포함한 HMC의 성능을 파악하기 위해서는 실제 동작 환경에서 실험 및 시뮬레이션을 해야만 한다. 앞 장에서 언급한 고속 직렬 링크의 모든 동작을 본 논문의 HMC 모델링 (CasHMC)에 구현하였다. 또한 HMC spec에서 정의하고 있는 패킷 레이아웃과 패킷 전송 방법, 링크 설정 및 오류 감지 기법 등 모든 특징들을 그래도 모델링 하였다. 이 HMC 모델링은 C++로 구현되어 클럭 사이클 단위로 동작하는 사이클 정밀 시뮬레이터이기 때문에 하드웨어와 유사하게 동작한다. 또한 다른 CPU 시뮬레이터와 연동하여 실제 메모리 주소 값과 같은 메모리 request를 전달받고 사이클 단위로 서로 다른 시뮬레이터를 동기화되기 때문에 실제 환경의 동작을 시뮬레이션 할 수 있다. 더 자세한 동작 원리는 공개되어 있는 CasHMC 소스코드를 참고하면 된다. 따라서 논문에서 제안된 HMC 모델링은 실제 HMC 시스템 없이도 정확한 성능 결과를 얻을 수 있다.

고속 직렬 링크상의 전송되는 데이터들 중에서 실제 데이터가 아닌 패킷의 header와 tail의 비율을 파악하기 위해 SimpleScalar CPU 시뮬레이터 [20]와 논문의 CasHMC를 이용하여 SPEC CPU2006 벤치마크 [21]를 통해 시뮬레이션 하였다. 2개의 시뮬레이터를 연동하기 할 때 시뮬레이션의 시간을 줄이고자 SPEC CPU2006 벤치마크를 SimpleScalar 시뮬레이터에서 실행시키고 메모리 request 패턴을 trace 파일로 저장한 뒤, 메모리 trace 파일을 CasHMC에 입력하여 실험하였다. 그림 8은 9개의 SPEC CPU2006 벤치마크에 따라 HMC 대역폭과 고속 직렬 링크에서 주고받은 데이터

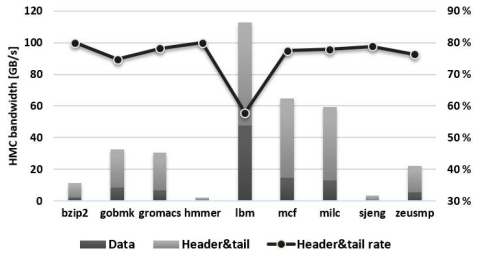


그림 8. HMC 대역폭과 header&tail의 비율
Fig. 8 HMC bandwidth and header&tail rate

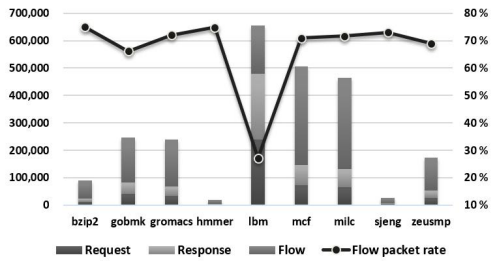


그림 9. 메모리 request와 response,
링크 제어 패킷의 개수
Fig. 9 The number of transmitted requests,
responses and flow packets

중에서 실제 메모리 데이터와 패킷의 header, tail의 비율의 그래프이다. 모든 벤치마크를 통틀어 패킷의 header와 tail의 비율은 평균적으로 약 75%이다. 그러나 lbm 벤치마크의 경우 58% 정도로 다른 벤치마크보다 현저히 낮은 비율을 보여주고 있다. 그림 9는 같은 벤치마크에서 시뮬레이션 동안 링크를 통해 전달된 메모리 request와 response, 링크 제어 패킷의 개수를 보여준다. lbm 벤치마크의 경우 그림 8의 패킷의 header와 tail의 비율과 같이 링크 제어 패킷의 수가 매우 낮은 것을 알 수 있다. 이것은 lbm 벤치마크가 그림과 같이 단위 시간당 매우 많은 메모리 request와 response를 발생시키는 벤치마크이기 때문이다. 만약 링크 마스터에 메모리 request/response 패킷이 전송대기 중이면 새로운 링크 제어 패킷을 생성할 필요 없이 대기 중인 패킷에 FRP나 RRP, RTC 값을 업데이트 할 수 있다. 따라서 링크에서 패킷의 전송량은 실행되는 벤치마크의 메모리 request/ response 빈도 특징에 따라서 달라진다. 이처럼 고속 직렬 링크의 성능은 실제 실행되는 프로그램 특징에 따라서 변하므로 실제 동작과 유사한 환경에서 시뮬레이션이 필요하다. 본 논문에서는 HMC spec에서 지원하는 다양한 링크 설정에서 시뮬레이션 하여 실행되는 프로그램의 특징 별로 최적의 링크 설정을 조사해보았다.

V. 실 험

1. 실험 환경

HMC는 기본적으로 수동적인 특징을 갖는 메모리 소자이기 때문에 HMC를 동작하게 하는 메모리 request들이 필요하다. 본 논문의 HMC 모델링은 3가지의 방법으로 메모리 request를 발생시킨다. 첫 번째는 다른 시뮬레이터와 연동하여 메모리 request를 전달받는 방법이고, 두 번째는 시뮬레이터에 내장된 임의의 메모리 request 생성기를 통해 request를 전달받는다. 마지막 세 번째 방법은 정해진 포맷으로 메모리 request들을 파일로 저장하여 읽어오는 방식이다. 이 실험에서는 메모리 request 데이터 크기와 메모리 utilization을 기준으로 다양한 특성을 가지는 메모리 request 파일들을 이용하여 세 번째 방법으로 실험을 진행하였다. 메모리 utilization은 하나의 CPU 클럭 사이클 당 몇 개의 메모리 request가 발생하였는지를 의미하는 것으로 메모리 utilization이 '0'이면 메모리 request가 없고 '1'이면 매 CPU 클럭 사이클 마다 메모리 request가 발생한 것을 뜻한다. 또한 HMC 모델링

표 1. 실험 환경
Table 1. Simulation Environment

Parameter	Value
Number of links	2, 4
Link lane speed (GB/s)	12.5, 15, 25, 28, 30
Link width	Full(16), Half(8), Quarter(4)
Link BER	10^{-12}
CPU clock period	0.5 ns
Memory density (total number of bank)	4 GB (256)
Address mapping	32-Byte max block
Request data size	128 bytes
Buffer (HMC cont / LM & LS / crossbar switch / DRAM command)	4 / 32 / 64 / 16

에서 사용한 DRAM timing 모델은 Gem5 시뮬레이터에서 정의한 HMC_2500_x32 모델을 사용하였고 더 자세한 실험 환경은 표 1에 정리되어 있다.

2. 실험 결과

그림 10은 4 링크, half-width, 25 GB/s로 링크가 설정되었을 때, 메모리 utilization에 따라서 HMC 대역폭과 메모리 request 대기 시간을 정리한 그래프이다. HMC 대역폭은 1초 동안 CPU와 HMC사이에 주고받은 데이터의 양을 뜻한다. 그림처럼 메모리 utilization이 증가할수록 HMC 대역폭이 증가하는 것을 알 수 있다. 그러나 메모리 utilization이 0.39 지점을 넘어서는 순간부터 HMC 대역폭이 더 이상 증가하지 않고 일정하게 유지되는 것을 알 수 있다. 이 지점을 지금부터 포화점이라고 부를 것이다. 이 포화점부터 HMC 대역폭이 증가하지 않는 이유는 메모리 request가 더 자주 발생하더라도 링크에서 패킷을 전송하는 속도가 한계에 도달하여 링크 마스터에서 패킷들이 전송을 대기 중이기 때문이다. 즉, 링크의 전송 속도가 메모리 request의 생성 속도를 따라가지 못하기 때문이다. 그래프의 다른 성능 요소인 메모리 request 대기 시간은 메모리 request가 발생한 시간부터 메모리에서 돌아오는 response가 CPU에 도착하는 시간까지 측정하여 평균화 하였다. 메모리 utilization이 낮을 때는 메모리 request가 적게 발생하여 패킷이 링크 마스터에서 기다리지 않고 바로 전송되기 때문에 request 대기 시간이 낮은 값을 일정하게 갖지만 포화점에서는 request 대기 시간이 급격히 증가한다. 포화점은 낮은 링크의 속도와 활용성 때문에 패킷들을 바로 보내지 못하는 것을 의미하므로 포화점 이후에는 링크 마스터에서 대기하는 시간 때문에 메모리 request 대기 시간이 급격히 증가한다. 그림에서는 포화점 이후에도 일정한 값을 유지하는데 이는 표 1과 같이 버퍼의 크기를 제한하여 시간이 측정되는 메모리 request의 개수를 제한하였기 때문이다. 버퍼의 크기 제한으로 생성되지 못한 메모리 request들은 여유 공간이 생길 때까지 대기하였다가 시뮬레이션 된다. 모든 메모리 request들은 파일로 저장되어 있어 라인 단위로 메모리 request를 생성하기 때문에 버퍼 크기 제한으로 중간에 소멸될 일은 없다. 이 실험의 결과로 4 링크, half-width, 25 GB/s로 링크가 설정되었을 때 포화점은 0.39이고 최대 85 GB/s 대역폭을 가지는 것을 확인하였다.

HMC spec에서는 다양한 링크 설정을 지원하고

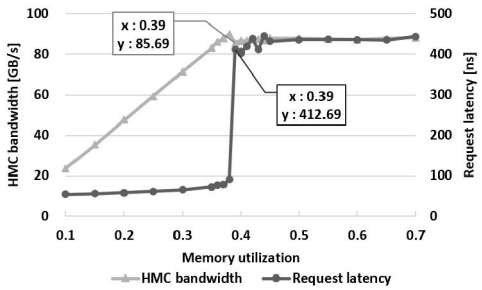
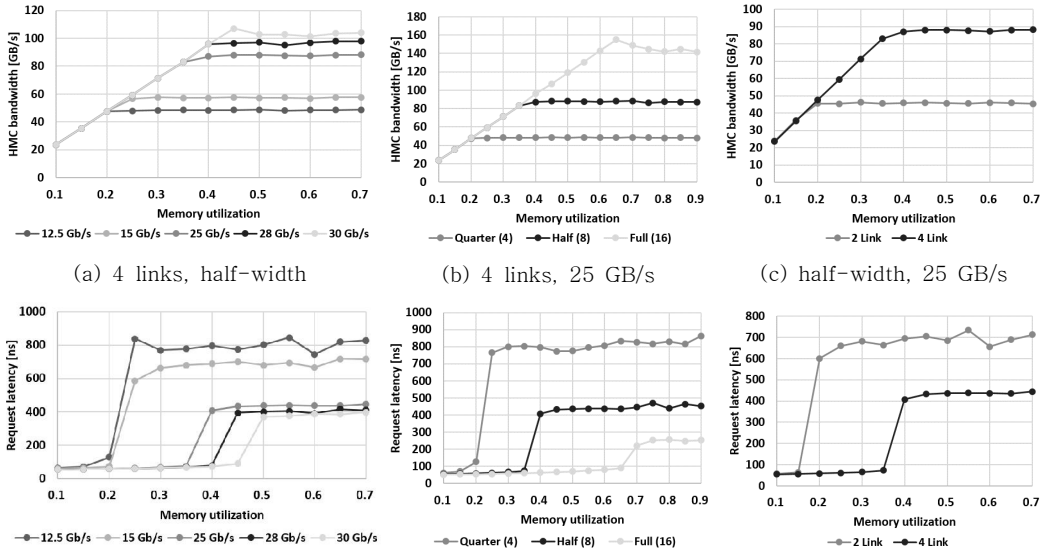


그림 10. HMC 대역폭과 메모리 request 대기 시간 실험 결과
Fig. 10 HMC bandwidth and memory request latency simulation result

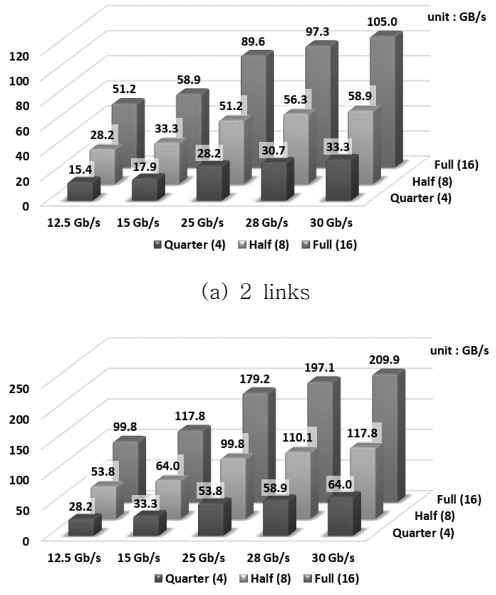
있다. 각각의 링크 설정에 따라 단위 시간에 전송할 수 있는 패킷의 양이 정해져 있으므로 HMC 대역폭과 메모리 request 대기 시간과 같은 HMC 성능에도 크게 영향을 미친다. 즉, 어떤 링크 설정이 최대로 지원할 수 있는 HMC의 성능을 알 수 있다면 시스템의 동작 환경에 따라 적절한 링크를 설정할 수 있다. 이런 걱정 없이 링크를 최대한으로 설정하고 사용할 수 있지만 그러면 앞에서 설명하였듯이 링크의 큰 소비 전력 문제를 피할 수 없다. 따라서 활성화 되어 있는 링크의 개수를 줄이거나 링크의 속도를 줄여 적절한 링크 설정을 선택함으로써 링크의 불필요한 소비 전력을 줄일 수 있다. 링크 설정에 따라서 HMC의 성능을 파악하기 위해 그림 11과 같이 다양한 실험을 수행하였다. 그림 11은 링크 설정 별로 메모리 utilization에 따라서 HMC 대역폭과 메모리 request 대기 시간을 측정한 결과이다. 모든 링크 설정은 HMC spec 2.1에 정의된 설정만 사용하였다. 그림 11 (a), (b), (c) 에서처럼 각 링크 설정에 따라 최대로 가질 수 있는 HMC 대역폭이 정해져 있다. CPU에서 많은 메모리 request가 발생하거나 메모리에서 많은 데이터들이 메모리 response로 전달이 되더라도 CPU와 HMC를 연결하는 링크 설정에 따라 성능 bottleneck이 발생하기 때문이다. 그림 11 (d), (e), (f) 는 링크 설정에 따른 메모리 request 대기 시간을 보여주고 있다. 앞에서 설명하였듯이 메모리 request 대기 시간의 포화점과 HMC 대역폭의 포화점이 서로 일치하는 것을 볼 수 있다. 따라서 각 링크 설정에 따라 포화점 부분이 최대 성능이라고 할 수 있다. 그림 11의 결과를 토대로 그림 12와 같이 각 링크의 최대 HMC 대역폭을 정리하였다. 그림 12는 활성화된



(a) 4 links, half-width (b) 4 links, 25 GB/s (c) half-width, 25 GB/s

(d) 4 links, half-width (e) 4 links, 25 GB/s (f) half-width, 25 GB/s

그림 11. 링크 설정별로 메모리 utilization에 따라서 HMC 대역폭과 메모리 request 대기 시간 실험 결과
Fig. 11 HMC bandwidth and memory request latency simulation result with respect to memory utilization in accordance with link configuration



(a) 2 links

(b) 4 links

그림 12. 모든 링크 설정에 따른 최대 HMC 대역폭
Fig. 12 The maximum HMC bandwidth in accordance with all link configurations

링크의 개수가 2와 4일 때 최대 HMC 대역폭을 보여준다. 이 정보를 이용하여 시스템에서 요구하는 메모리 대역폭을 미리 파악하면 링크를 적절하게 설정할 수 있다. 예를 들어, 시스템에서 요구하는 최대 대역폭이 30 GB/s라고 하였을 때 링크 2, quarter-width, 28 GB/s가 최적의 링크 설정이 된다. 만약 이보다 안 좋게 설정을 하면 링크에서 성능 bottleneck이 걸리기 때문에 전체 시스템의 성능이 저하된다. 반대로 링크를 최적의 설정보다 좋게 설정하면 특별한 성능 이득 없이 링크에서 불필요하게 많은 전력이 소비되게 된다. 사용자 입장에서는 시스템의 최대 메모리 대역폭을 정확히 알기 쉽지 않지만 대략적인 수치로도 처음 HMC 시스템을 설정할 때 링크 설정의 최대 성능 정보를 이용한 다면 적절하게 시스템을 운영할 수 있을 것이다. 따라서 본 논문의 HMC 모델링은 링크의 최적화된 설정을 알려줄 뿐만 아니라 HMC 시스템 동작과 성능을 미리 파악할 수 있도록 큰 도움을 줄 것이다.

V. 결 론

하이브리드 메모리 큐브 (HMC)는 여러 개의 HMC를 연결하는 chaining 기법과 기본적인 연산이

내장된 로직 다이의 특성으로 서버와 PIM (processing-in-memory) 시스템에서 매우 촉망받고 있는 3D 메모리이다. HMC는 기존 메모리 혹은 다른 3D 메모리와 다르게 고속 직렬 링크 (SerDes)로 CPU와 연결하여 빠른 속도로 데이터를 주고받는다. 그러나 클럭을 데이터 사이에 끼워 넣어 통신하는 고속 직렬 링크의 특징 때문에 전송할 데이터가 없더라고 링크들은 항상 연결을 유지해야 한다. 이로 인해 HMC 시스템의 전력에서 절반 이상이 고속 직렬 링크에 소비되는 문제점이 발생하고 있다. 고속 직렬 링크는 전송 속도 혹은 활성화된 링크의 개수에 비례한다. 따라서 시스템에서 요구하는 성능에 따라서 링크를 적절히 설정하는 것이 매우 중요하다. 본 논문에서는 고속 직렬 링크를 포함한 HMC를 HMC spec에서 정의하고 있는 동작 및 구조를 그대로 모델링하여 실제 실행 환경에서 다양한 실험을 해보았다. 실험 결과 각각의 링크 설정에서 최대 성능을 가지는 포화점을 발견하였고 이를 토대로 최대 HMC 대역폭을 링크 설정 별로 조사하였다. 사용자는 HMC를 설정할 때 이 실험 결과를 이용해서 적절한 링크를 설정 할 수 있다. 즉, 낮은 링크 설정으로 인한 성능 bottleneck과 높은 설정으로 인한 불필요한 소비 전력 사이의 적정점을 찾는데 도움을 줄 수 있다. 이 실험 결과는 앞으로 HMC 링크 관리 연구에도 기여할 것으로 기대된다. 또한 논문의 HMC 모델링을 통해 다양한 벤치마크 프로그램이나 다른 시뮬레이터와 연동으로 HMC 시스템을 분석하는데 큰 도움을 줄 것이다.

References

[1] G.H. Loh, “3D-Stacked Memory Architectures for Multi-Core Processors,” Proceedings of International Symposium on Computer Architecture, pp. 453-464, 2008.

[2] U. Kang, H.J. Chung, S. Heo, D.H. Park, H. Lee, J.H. Kim, S.H. Ahn, S.H. Cha, J. Ahn, D. Kwon, J.W. Lee, H.S. Joo, W.S. Kim, D.H. Jang, N.S. Kim, J.H. Choi, T.G. Chung, J.H. Yoo, J.S. Choi, C. Kim, and Y.H. Jun, “8 Gb 3-D DDR3 DRAM Using Through-Silicon-Via Technology,” IEEE Journal of Solid-State Circuits, Vol. 45, No. 1, pp. 111-119, 2010.

[3] D.H. Woo, N.H. Seong, D.L. Lewis, and H.H.S. Lee, “An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth,” Proceedings of International Symposium on High-Performance Computer Architecture, pp. 1-12, 2010.

[4] C. Nimmagadda, D. Lisk, and R. Radojcic, “3D Stacking: Where the Rubber Meets the Road,” Proceedings of IEEE International Conference on IC Design & Technology, pp. 1-3, 2012.

[5] J.S. Kim, C.S. Oh, H. Lee, D. Lee, H.R. Hwang, S. Hwang, B. Na, J. Moon, J.G. Kim, H. Park, J.W. Ryu, K. Park, S.K. Kang, S.Y. Kim, H. Kim, J.M. Bang, H. Cho, M. Jang, C. Han, J.B. Lee, K. Kyung, J.S. Choi, and Y.H. Jun, “A 1.2V 12.8GB/s 2Gb mobile Wide-I/O DRAM with 4×128 I/Os using TSV-based stacking,” Proceedings of IEEE International Solid-State Circuits Conference, pp. 496-498, 2011.

[6] JEDEC standard, “Wide I/O 2 (WideIO2) - JESD229-2,” Jedec Solid State Technology Association, 2014.

[7] JEDEC standard, “High Bandwidth Memory (HBM) DRAM - JESD235,” Jedec Solid State Technology Association, 2013.

[8] J. Jeddeloh and B. Keeth, “Hybrid Memory Cube new DRAM Architecture Increases Density and Performance,” Proceedings of Symposium on VLSI Technology, pp. 87-88, 2012.

[9] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture,” Proceedings of International Symposium on Computer Architecture, pp. 336-348, 2015.

[10] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S.aibal Mukhopadhyay, “Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory,” Proceedings of International Symposium on Computer Architecture, pp. 380-392, 2016.

[11] P. Rosenfeld, “Performance Exploration of the Hybrid Memory Cube,” University of Maryland, 2014.

[12] S.H. Pugsley, J. Jestes, R. Balasubramonian,

V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads,” IEEE Micro, Vol. 34, pp. 380-392, 2014.

[13] D. Resnick, “Opportunities to Upgrade Main Memory,” Proceedings of International Symposium on Memory Systems, pp. 55-59, 2015.

[14] S.H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads,” Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, 2014.

[15] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing,” Proceedings of International Symposium on Computer Architecture, pp. 105-117, 2015.

[16] Analog Bits, “28nm SERDES Product Brief,” Analog Bits Inc. white paper, 2011.

[17] Hybrid Memory Cube Consortium, “Hybrid Memory Cube Specification 2.1,” 2014.

[18] Y. Joo, M. Kim, I. Han, and S. Lim, “Cache Simulator Design for Optimizing Write Operations of Nonvolatile Memory Based Caches,” IEMEK J. Embed. Sys. Appl., Vol. 11, No. 2, pp. 87-95, 2016 (in Korean).

[19] D.I. Jeon and K.S. Chung, “CasHMC: A Cycle-accurate Simulator for Hybrid Memory Cube,” IEEE Computer Architecture Letters, 2016.

[20] T. Austin, E. Larson, D. Ernst, “SimpleScalar: An Infrastructure for Computer System Modeling,” IEEE Computer, Vol. 35, No. 2, pp. 59-67, 2002.

[21] J.L. Henning, “SPEC CPU2006 Benchmark Descriptions,” ACM SIGARCH Computer Architecture News, Vol. 34, pp. 1-17, 2006.

Dong-Ik Jeon (전 동 익)

He is received his B.S. in Electronics & Communication Engineering from Hanyang University, Ansan, Korea in 2012, and he is currently working toward a Ph.D. in Electronics and Computer Engineering from Hanyang University, Seoul, Korea. His research interests include the memory controller, DRAM memory architecture, and hybrid memory cube (HMC).

Email: estwingz@naver.com

Ki-Seok Chung (정 기 석)

He is received his B.S. in Computer Engineering from Seoul National University, Seoul, Korea in 1989, and his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1998. He was a Senior R&D Engineer at Synopsys, Inc. in Mountain View, CA from 1998 to 2000, and was a Staff Engineer at Intel Corp. in Santa Clara, CA from 2000 to 2001. He also worked as an Assistant Professor at Hongik University, Seoul, Korea from 2001 to 2004. Since 2004, he has been a professor at Hanyang University, Seoul, Korea. His research interests include low power embedded system design, multi-core architecture, image processing, SoC-platform based verification and system software for MPSoC.

Email: kchung@hanyang.ac.kr