Journal of The Institute of Electronics and Information Engineers Vol.53, NO.8, August 2016 http://dx.doi.org/10.5573/ieie.2016.53.8.059
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2016-53-8-7

Stream Processing에서 I/O데이터 일관성을 고려한 성능 최적화

(Performance Optimization Considering I/O Data Coherency in Stream Processing)

나 하 나*, 이 준 환**

(Hana Na and Joonwhan Yi©)

요 약

본 논문은 대량의 stream data를 처리하는 어플리케이션에서 하드웨어 가속기들이 접근하는 메모리가 non-cacheable에서 cacheable으로 변경됨에 따라 발생할 수 있는 데이터 일관성 문제를 고려하여 시스템 최적화를 진행하였다. 이를 위해 상위 수준 시뮬레이션을 통한 프로파일링 결과를 토대로 분석식을 만들어 활용하였다. 실험한 결과 여러 이미지 크기에서 메모리가 cacheable로 변경됨에 따라 평균 1.40배의 성능 향상을 보였다. 분석식의 주요 파라미터 최적화를 통해 최종적으로 3.88배의 성능 이득이 발생했으며, 항상 메모리가 cacheable인 경우의 성능이 항상 우월한 것은 아님을 확인할 수 있었다.

Abstract

Performance optimization of applications with massive stream data processing has been performed by considering I/O data coherency problem where a memory is shared between processors and hardware accelerators. A formula for performance analyses is derived based on profiling results of system-level simulations. Our experimental results show that overall performance was improved by 1.40 times on average for various image sizes. Also, further optimization has been performed based on the parameters appeared in the derived formula. The final performance gain was 3.88 times comparing to the original design and we can find that the performance of the design with cacheable shared memory is not always.

Keywords : Stream data, I/O data coherency, Optimization, Quantitative Analysis, Analysis formula

Ⅰ. 서 론

멀티미디어 어플리케이션이 증가함에 따라 규칙적으로 대량의 데이터를 접근해 처리하는 여러 stream processing 연구들이 진행된다[1~2]. 대량의 데이터를 효율적으로 처리하기 위해 멀티프로세서와 주변 하드웨어들이 사용된다. 그런데 이는 데이터 일관성 문제(data coherency problem)들을 유발할 수 있다.

먼저 멀티프로세서의 캐시들 간 데이터를 공유하기 때문에 데이터 일관성 문제가 발생할 수 있다. 이는 시스템의 잘못된 동작을 초래할 수 있으며 하드웨어의 경우 snooping과 directory-based방식으로 해결할 수 있다[3~5]. 소프트웨어의 경우 데이터가 stale한 시점을 컴파일러가 예측한 후, 무효화하는 명령어를 삽입해 데이터 일관성을 유지한다[6, 11, 14]. 이 문제는 하드웨어 가속기와 캐시 간 발생 가능한 주변장치 일관성 문제로 확장되었다. 만약 캐시와 하드웨어 가속기가 동일한 메모리를 접근하면, 임의의 데이터에 대한 여러 복사본들이 생겨 데이터 불일치 문제가 발생할 수 있다[7]. 하드웨어 가속기가 접근하는 메모리 영역과 캐시 정책에 따라 문제가 발생한다. 메모리 영역이 non-cacheable이거나 캐시 쓰기 정책이 write-through이라면, 일관성 문제는 없으나 wirte traffic이 크다[6]. 그러나 캐시 쓰기 정책이 write-back이라면 데이터가 쫓겨날 때에만 메모리에 반

* 한화탈레스(주) (Hanwha Thales Co. Ltd)
** 평생회원, 광운대학교 컴퓨터공학과
(Dept. of Computer Engineering, Kwangwoon University)
© Corresponding Author (E-mail : joonhwan.yi@kw.ac.kr)
※ 이 연구는 2013년 광운대학교 교내학술연구비 지원에 의해 연구되었음.
※ 이 논문은 2013년도 정부(교육부)의 재원으로 한국 재단의 지원을 받아 수행된 기연구사업임(No. NRF-2011-0025385)
Received ; September 4, 2015 Revised ; July 5, 2016
Accepted ; July 28, 2016

영돼 메모리와 캐시의 데이터가 일치하지 않을 수 있다. 이를 해결하기 위해 하드웨어의 경우 주변장치 데이터를 캐시에 바로 저장하거나 주변장치와 프로세서의 데이터를 따로 저장할 수 있다[7~9]. 소프트웨어의 경우 위처럼 컴파일러를 이용할 수 있으나 이 방법은 제한적일 수 있으며, 보수적 접근으로 인해 불필요한 무효화와 지역성 저하가 발생할 수 있다[6~11]. 또한 캐시 클린 또는 무효화 방법이 널리 사용될 수 있다[10, 15]. 클린은 dirty 데이터만을 메모리에 반영하는 것이며 무효화는 하드웨어가 메모리의 데이터를 수정했다면 해당 캐시의 데이터를 무효화하는 것이다[10, 15]. 하드웨어 방법은 성능 측면에서 효율적이지만 설계를 위한 비용, 시간, 면적이 추가되는 단점이 있다. 소프트웨어 방법은 상대적으로 구현 비용이 저렴하고 flexible한 장점이 있지만 일관성 유지를 위한 오버헤드가 추가되기 때문에 캐시 클린 방식이 효과적인지 파악하는 것이 중요하다.

본 논문은 데이터 일관성을 고려한 최적화를 위해 캐시와 DMAC가 포함되고 대량의 데이터를 다루는 영상처리 어플리케이션으로 분석을 진행했다. 캐시클린 방식이 성능 변화에 영향을 주는 파라미터들을 도출하고, ESL (electronic system level) 시뮬레이션을 통해 이들 파라미터들과 성능간의 관계를 분석식으로 도출하였다. 분석식을 통해 stream data의 양에 따른 성능개선 효과를 파악하고, 주요 파라미터 확인 및 추가 최적화가 가능함을 보인다. 여기서 도출된 분석식은 stream data가 저장되는 메모리를 프로파일링할 수 있다면 다른 시스템에도 적용 가능하다.

본 논문의 구성은 다음과 같다. Ⅱ장은 상위수준에서의 case study로 데이터 일관성 문제와 해결법에 대해 설명한다. Ⅲ장에서는 이미지 크기에 따른 실험 결과 및 분석을, Ⅳ장에서는 정량적 분석식과 활용법을 보인다. 마지막 Ⅴ장에서는 결론을 맺는다.

Ⅱ. 프랙탈 이미지 출력 시스템 및 일관성 문제

본 논문은 상위수준 기반 ESL에서 실험을 진행하였다. ESL은 GL(gate level)과 RTL(Register transfer level)보다 높은 추상화 계층으로 하드웨어 소프트웨어 통합 설계가 용이하다. 빠른 시뮬레이션 속도로 설계 초기에 프로파일링 결과를 토대로 병목현상의 원인을 파악해 최적화가 가능하다.

프랙탈 이미지(fractal image) 출력 시스템은 프랙탈을 계산해 화면에 출력한다. 본 논문에서는 성능 프로

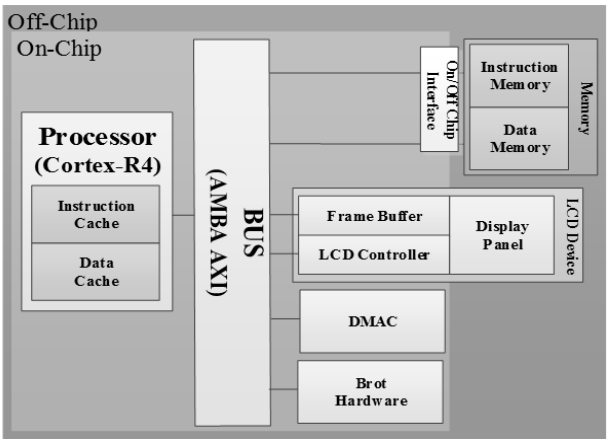

그림 1. 하드웨어 플랫폼
Fig. 1. Hardware platform.

```
int main(){
   InitLCD(); //Initialize LCD
   for(count=0; count<3; count++){
      Render(cx, cy, dx, dy);
   }
}
void Render(int cx, int cy, int dx, int dy){
   //Calculate fractal image
   for(y=SCRN_HEIGHT; y>=0; y--){
      for(x=0; x<SCRN_WIDTH; x++){
        //Brot hardware operation
        //Brot operation done check
        //Save calculation result into memory
           ...
        CacheClean(); //If memory is cacheable, do clean operation
        CopyResult();
      }
   }
}
void CopyResult(){//Perform data copy using DMAC
   //Interrupt enable
   //Provide three information for DMAC
   //SRC_ADDR,DEST_ADDR,DATA_SIZE
   //DMAC Operation start
   // Operation check
}
```

그림 3. 프로그램의 pseudocode
Fig. 2. Pseudocode of the program.

파일링을 위해 프랙탈 이미지 생성과 출력을 세 번 반복한다. 시스템은 cycle-accurate하게 Cortex-R4의 프로세서[12]로 C레벨에서 Carbon사의 SoC Designer[13]로 그림 1과 같이 구성하였다. 버스는 ARM의 고성능 버스 프로토콜인 AMBA AXI를[17], 각각 4KB의 명령어와 데이터 캐시를 사용하였고, 외부 메모리 딜레이는 40cycle로 설정하였다. 그림 2는 응용소프트웨어의 의사코드로, InitLCD함수가 register값을 설정해 LCD를 초기화 한 후, Render함수가 프랙탈 이미지를 계산해 메모리에 저장한다. CopyResult함수는 프랙탈을 LCD의 frame buffer로 복사해 화면에 출력한다.

표 1. 외부 메모리 접근 횟수와 사이클(Non-cacheable)
Table1. External memory access and total cycle.

이미지크기 / 항목	46KB	225KB	900KB
Memory Access	128,625	615,973	835,599
Total cycle	5,770,093	26,395,132	115,829,729

이 시스템은 빠른 계산과 데이터 복사를 위해 하드웨어들이 사용된다. 그림 1의 brot하드웨어는 Render함수에서 프랙탈을 계산하며, 계산된 이미지의 효율적인 전송을 위해 DMAC(direct memory access controller)를 사용하였다. DMAC는 대량 데이터 이동에 특화된 하드웨어로, 프로세서 개입 없이 메모리에 직접 접근해 데이터를 복사한다. 동작을 위해 source address, destination address, data size가 요구되며, 동작 과정은 [16]과 같다. DMAC는 CopyResult함수에서 프랙탈 이미지 결과를 LCD로 복사하는 데 사용된다.

그런데 DMAC와 캐시가 동일한 메모리 영역을 접근하면 두 가지 일관성 문제가 발생할 수 있다[10]. 먼저 DMAC가 source영역의 데이터를 destination으로 복사해 데이터가 수정되었지만, 캐시에는 변경되기 전의 데이터가 존재해 stale 데이터를 접근하는 것으로, 해당 어플리케이션에서는 발생하지 않는다. 다음으로 프로세서가 수정한 데이터를 메모리에 반영하기 전에 DMAC가 stale 데이터로 복사하는 경우이다. 이에 대한 해결책은 각각 앞에서 언급했던 무효화와 클린이 해당된다.

Brot하드웨어 결과가 non-cacheable영역에 저장되면 이미지가 클수록 외부 메모리 접근이 증가해 성능 열화가 발생한다. 표 1은 이미지 크기별 외부 메모리 접근에 따른 총 사이클이다. 이미지가 46KB에서 900KB로 증가되면서 메모리 접근이 128,625에서 835,599로 약 6.49배 증가하나 총 사이클은 5,770,093에서 115,829,729로 약 20.07배 증가했다. 반면에 cacheable일 경우 프로세서가 캐시에 저장된 이미지를 접근해 감소된 메모리 접근으로 성능이 향상되지만, 캐시가 write-back일 때 dirty데이터로 인해 데이터 불일치가 발생할 수 있다. 따라서 DMAC동작 전에 CacheClean함수로 프로세서와 메모리가 사용하는 데이터를 일치시킨 후, CopyResult 함수를 수행해야 한다.

Ⅲ. 성능분석

성능은 DMAC가 복사하는 영역이 non-cacheable과 cacheable일 때의 총 사이클을 여러 이미지 크기에서 비교하였다. 표 2는 메모리가 cacheable으로 변경될 때, 수행 사이클과 메모리 접근의 이득과 두 경우의 데이터 캐시 미스율을 보인다. 네 이미지 크기에서 평균 1.40배의 사이클 이득이 발생하였다. 특히, 이미지가 2KB로 캐시보다 작은 경우에도 97,190사이클이 감소해 약 1.37배의 개선이 있다. 성능 향상은 프로세서에 의한 외부 메모리 접근이 여러 이미지 평균 약 1.55배 감소했기 때문이다. 또한 이미지가 커지더라도 개선된 성능 정도는 비슷하다. 표 2에서 메모리 영역이 변경되면서 캐시에 저장되는 메모리 영역이 커져 데이터 캐시 미스율은 평균 약 1.0337%만큼씩 비슷하게 증가함을 알 수 있다.

Ⅳ. 분석식 도출 및 활용

네 이미지 평균 1.40배의 사이클 개선이 있지만 최적

표 2. 메모리가 cacheable로 변경될 때의 데이터 캐시 미스율, 사이클, 메모리 접근 횟수 이득
Table2. Data cache miss ratio, and gain of cycle and memory access when memory is changed to cacheable.

구분	항목	2KB	46KB	225KB	900KB	Average
Function별 수행 사이클 이득	Render	108,663	1,624,654	7,746,460	32,808,038	10,571,953.75
	CopyResult	44	163	76	108	97.75
	FIQ Handler	2,876	39,464	188,888	738,807	242,508.75
	CacheClean	-14,440	-19,188	-19,188	-19,224	-18,010.00
	Etc	47	66	-516	-4	-101.75
	Total	97,190	1,645,159	7,915,720	33,527,725	10,796,448.50
	Gain	1.37	1.39	1.42	1.40	1.40
Memory Access 이득	Number of reduced accesses	2,898	46,960	225,542	279,902	138,825.50
	Gain	1.56	1.57	1.57	1.50	1.55
Data cache miss rate	Non-cacheable	0.0087%	0.0018%	0.0004%	0.0010%	0.0029%
	Cacheable	1.1558%	0.9511%	1.0667%	0.9731%	1.0366%
	Difference (Non-cacheable-Cacheable)	+1.1471%	+0.9493%	+1.0663%	+0.9721%	+1.0337%

화를 위해 시뮬레이션으로 캐시클린을 통한 성능 향상 요소들을 파악해 분석식을 만들었다. 주요 파라미터로 추가적으로 최적화를 진행하였으며, 캐시클린을 통한 성능 향상 원인을 알 수 있다. 상위수준 시뮬레이션이 필요한 이유는 주요 파라미터인 burst length를 계산하기 위해서이다. 이는 일련의 주소 데이터를 연속적으로 얼마나 전송하는지를 나타내며[17], 버스가 데이터를 효율적인 전송하는지에 대한 판단기준이 될 수 있다.

1. 분석식 도출

표 3은 분석식에 사용되는 변수들이며, 각 메모리 영역 설정은 아래첨자로 NC, C를 사용하였다. 예를 들어 메모리가 non-cacheable이고 버스를 통해 전송되는 데이터의 양은 Q_{NC}로 표현된다.

메모리가 non-cacheable일 때의 B와 Q는 표 3의 시뮬레이션으로 얻은 WA와 WD변수로 계산되며, 읽기일 때도 같은 방식이다. Cacheable인 경우에는 시뮬레이션을 하지 않아도 non-cacheable일 때와 비슷하게 B와 Q를 구할 수 있다. 이를 위해 캐시는 라인 단위로 동작하기 때문에 B_C는 word의 수이고, WA_C는 식 (1)과 같이 캐시 구조와 특성, 이미지 크기를 이용할 수 있다.

표 3. 분석식에서 데이터 획득 또는 계산되는 변수들
Table3. Variables need to be extracted or calculated.

변수명	설 명	데이터 획득방법
MD	Memory delay	Simulation
SIZE	Image size	
WA	Write address횟수	
WD	Write data양이며 단위는 word	
B	평균 burst length($B=\frac{WD}{WA}$)	계산
Q	AXI로 전송되는 데이터 양 ($Q=WD=B*WA$)	

$$WA_C = \frac{SIZE}{Nmber of words * Nmber of ways} \quad (1)$$

결과가 저장되는 메모리 영역이 cacheable로 변경되면서 발생하는 성능 변화 요인은 두 가지이다. 먼저 캐시는 일관적인 burst length를 가져 메모리 접근이 감소한다. 또한 캐시의 데이터는 쫓겨나기 전까지 여러 번 사용될 수 있다. 데이터가 자주 사용될 때마다 버스가 메모리를 접근하는 횟수가 감소해 성능이 향상된다. 위의 요인들로 인해 같은 이미지 크기에서 프로그램이 동작하더라도 WA와 B는 다르기 때문에 Q_{NC}와 Q_C는 같지 않다. 그리고 같은 non-cacheable일 때여도 소프트웨어 최적화를 진행한다면 메모리를 접근하는 형태가 변경되기 때문에 Q_{NC}는 최적화 경우에 따라 달라진다.

식 (2)는 cacheable로 변경될 때의 사이클 단위의 성능 이득이며, 이 값이 CacheClean함수의 수행 사이클보다 커야만 성능 이득이 존재한다. 클린은 반복문으로 캐시 라인을 클린하는 명령어를 수행하는 것이기 때문에[10], 캐시의 구조 정보로 사이클을 구할 수 있다.

$$Gain = \frac{Q_{NC}}{B_{NC}}(MD+B_{NC}-1) - \frac{Q_C}{B_C}(MD+B_C-1) \quad (2)$$

식 (2)는 외부 메모리를 접근할 때마다의 사이클 비용의 차이를 의미한다. $(MD+B_{NC}-1)$와 $(MD+B_C-1)$처럼 표현되는 이유는 burst length가 B_{NC} 또는 B_C인 데이터를 외부 메모리에 읽거나 쓸 때, 첫 번째 데이터는 MD사이클이 소요되고 그 후 연속된 데이터들은 1사이클이 소요되기 때문이다. 또한 $\frac{Q_{NC}}{B_{NC}}$와 $\frac{Q_C}{B_C}$은 앞에서 언급된 두 가지 성능 변화 요인에 대한 데이터의 양을 비교해 캐시클린 효과의 원인을 분석하기 위해서이다. 그런데 Q_{NC}와 Q_C는 서로 다르기 때문에 이 양의 차이 Q_{NC}'를 식 (3)으로 정의하고, 이를 식 (2)에 대입하여 정리하면 식 (4)와 같다.

표 4. 실제 성능 이득과의 비교
Table4. Comparison between actual and expected gain. (단위 : 사이클)

이미지 크기	Average burst length		Expected gain					Actual gain	오차	
	NC	C	Burst(1)	Cache(2)	Clean	(1)÷(2)	Gain		Value	%
2KB	2.32	8	90,461	28,647	14,440	3.15	104,667	96,944	7,477	7.71%
11KB	2.35	8	363,738	73,902	18,221	4.92	418,419	389,193	30,226	7.77%
64KB	2.29	8	1,544,278	355,279	18,259	4.34	1,880,298	1,644,820	236,139	14.38%
225KB	2.29	8	7,397,462	1,382,999	18,221	5.34	8,761,239	7,915,831	846,519	10.96%
900KB	2.28	8	30,634,091	7,249,633	18,221	4.22	37,864,502	33,527,524	4,337,978	12.94%
평균	2.30	8	8,006,006.00	1,818,092.00	17,472.40	4.40	9,806,625.00	8,714,862.40	1,091,667.80	10.69%

$$Q_{NC}' = Q_{NC} - Q_C \quad (3)$$

$$\frac{Q_{NC}}{B_{NC}}(MD + B_{NC} - 1) - \frac{Q_{NC}}{B_C}(MD + B_C - 1) + \frac{Q_{NC}'}{B_C}(MD + B_C - 1) \quad (4)$$

식 (4)은 언급된 두 성능 향상 요인들을 구분할 수 있다. 앞의 두 항은 캐시의 일관된 burst length로 인한 Burst gain이며, 세 번째 항은 Cache gain으로 캐시가 데이터를 여러 번 사용해 Q_{NC}'만큼 메모리에 접근되지 않아 발생한 이득이다. 표 4는 실제 성능 이득과 분석식을 통한 성능 이득을 비교한 것이다. 실제 성능 이득과의 오차는 여러 이미지 평균 10.69%이며, Burst gain이 평균 4.40배 더 커 캐시클린의 원인임을 알 수 있다.

```
1.  for(y=SCRN_HEIGHT; y>=0; y--){
2.    for(x=0; x<SCRN_WIDTH; x+=16){
3.      //Brot hardware operation
4.      Col[0] =(int)*(nBROTBase+BROT_OFF_RES0);
5.       …
6.      //Calculate 16 RGB pixels
7.      pixelR[0] = (col[0])<<1; pixelG[0] = (col[0]*9)<<1; pixelB[0] = (col[0]*7)<<1;
8.       …
9.      //Grouping 4 pixels in a array with 12 elements
10.     Array[0] = (pixelR[1]<<24|pixelB[0]<<16|pixelG[0]<<8|pixelR[0]);
11.      …
12.     //Save result to the memory with memcpy function
13.     offset = ((y*SCRN_WIDTH+x)*3)>>2;
14.     memcpy((buff_result+offset), Array, sizeof(int)*12);
15.      ...
16.     CacheClean(); //If memory is cacheable, do clean operation
17.     CopyResult();
18.     }
19.   }
20. }
```

그림 3. Render함수의 코드 일부(unrolling factor=16)
Fig. 3. Partial code in Render function(unrolling factor=16).

표 5. Burst length의 비교
Table5. Comparison of burst length.

항목 / Unrolling factor	Unrolling	Memcpy w/ unrolling	Cacheable
4	3.2	3.6	8
8	3.0	4.1	8
16	2.7	5.6	8

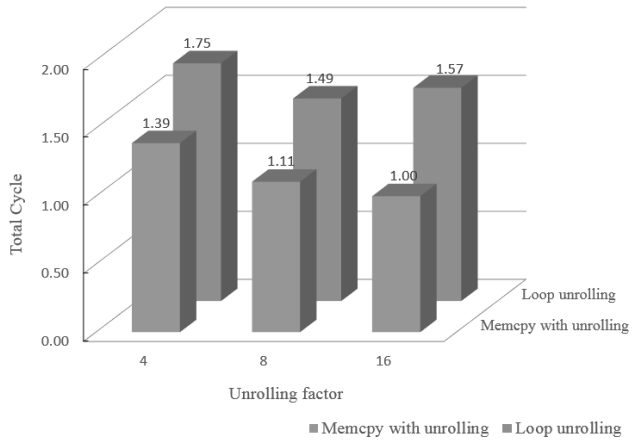

그림 4. Burst length 최적화를 통한 총 사이클 비교(Non-cacheable메모리)
Fig. 4. Comparison of total cycles after optimization of burst length(Non-cacheable memory).

2. Burst length최적화

Cacheable로 변경되면 캐시는 한꺼번에 메모리로부터 데이터를 가져오기 때문에 non-cacheable일 때의 burst length보다 증가된다. 따라서 non-cacheable일 경우의 최적화를 통해 burst length를 증가시켜 메모리 접근을 감소시키고, 이를 cacheable로 변경하면 성능이 개선될 수 있다. 이를 위해 두 방법을 사용했으며 이미지는 46KB로 진행하였다.

먼저 소프트웨어 최적화에 많이 사용되는 루프풀기(loop unrolling)를 사용하였다. 그림 3은 Unrolling factor가 16인 경우로, unrolling factor가 커질수록 루프 수행마다 계산되는 픽셀이 많아져 메모리 접근 횟수를 줄일 수 있다. 다음으로 계산된 각 1byte의 RGB픽셀들을 큰 데이터 타입 변수에 묶어 memcpy로 데이터를 저장하였다. 그림 3의 생성된 1byte의 RGB픽셀들을 4개씩 묶어(7~10줄), memcpy로 데이터를 한꺼번에 메모리에 저장하는 것이다(12~14줄). 이를 루프풀기와 함께 적용하면 생성된 픽셀들을 한꺼번에 메모리로 직접 복사해 burst length가 증가한다.

표 5는 루프풀기만 적용했을 경우와 루프풀기와 함께 memcpy를v 사용할 때의 burst length를 메모리가 cacheable인 경우와 함께 비교한 것이다. 루프풀기만 적용하면 unrolling factor가 커질수록 길어진 코드로 인해 명령어 캐시 미스가 증가해 burst length가 감소된다.

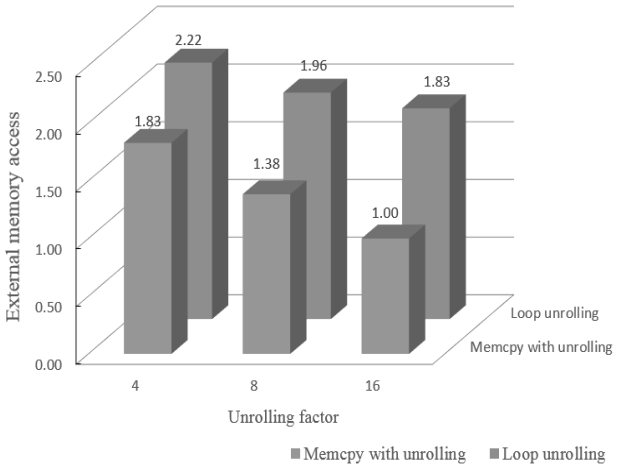

그림 5. Burst length최적화를 통한 외부 메모리 접근 비교(Non-cacheable메모리)
Fig. 5. Comparison of external memory after optimization of burst length(Non-cacheable memory).

표 6. 최적화 기법의 성능 이득
Table6. Performance gain of optimization.

Unrolling factor	Cacheable Gain	Optimization Gain
4	1.03	2.94
8	1.04	3.68
16	1.04	3.88
20	1.02	3.36
40	0.98	2.31
80	0.96	1.60

루프풀기와 함께 memcpy를 사용하면 unrolling factor가 16일 경우 5.6까지 증가된다. 이는 같은 데이터의 양을 처리하기 위한 메모리 접근수가 감소해 cacheable일 때의 일관된 burst length에 근접해지는 것이다. 그림 4와 그림 5는 메모리가 non-cacheable일 경우의 총 사이클과 외부 메모리 접근 횟수를 보인다. 외부 메모리 접근 횟수가 최대 2.22배의 이득이 발생해 최대 1.75배의 총 사이클 이득이 있음을 알 수 있다.

그렇지만 non-cacheable영역의 burst length를 최적화 한 후, cacheable로 변경해도 성능이 항상 우월하진 않다. 표 6은 메모리가 cacheable로 변경될 때의 성능 이득과 최적화를 진행할 때 발생된 성능 이득을 더 큰 unrolling factor까지 보여준다. unrolling factor가 20일 때까지 cacheable일 때의 성능이 좋지만 이 이후에는 성능이 오히려 퇴화된다. 이는 cacheable영역의 증가로 데이터 캐시 미스가 증가했기 때문이다. 예로 unrolling factor가 80일 때 데이터 캐시 미스는 cacheable로 변경되면서 0.038%에서 1.058%로 약 27.78배 증가해 메모리 접근 횟수는 22,892에서 26,838으로 약 1.17배의 증가했다. 최종적으로 최적화를 통해 최대의 성능 이득은 3.88배임을 확인할 수 있었다.

V. 결 론

본 논문은 대량의 stream 데이터를 처리하는 시스템에서 DMAC와 캐시가 일으키는 데이터 일관성 문제를 고려한 최적화를 보여준다. 이를 위해 상위수준 실험의 프로파일링을 통해 분석식을 만들었다. 이 분석식으로 성능에 영향을 주는 요소인 burst length를 최적화하여 메모리 접근 횟수를 감소시켰다. 실험 결과, 캐시클린을 사용할 경우 메모리가 non-cacheable인 경우보다 여러 이미지 크기 평균 약 1.40배의 사이클의 이득을 가졌다. Burst length와 메모리 관점에서 최적화를 진행한 결과, cacheable로 변경됨에 따라 최대 3.88배의 성능 이득이 있었다. 또한 실험을 통해 경우에 따라 cacheable일 때의 성능이 항상 우월한 것은 아님을 확인할 수 있었다.

REFERENCES

[1] D.Kudithipudi, S.Petko, E.B.John, “Caches for Multimedia Workloads:Power and Energy Tradeoffs”, IEEE Transaction, vol.10, pp. 1013-1021, 2008.
[2 Zheng Fang, C.Venkatramani, R.Wagle, K.Schwan, “Cache Topology Aware Mapping of Stream Processing Applications onto CMPs”, In ICDCS, pp. 52-61, 2013.
[3] A.Dash, Petrov,P., “Energy-Efficient Cache Coherence for Embedded Multi-Processor Systems through Application-Driven Snoop Filtering”, In Proc. of 9th EUROMICRO Conference on DSD, pp. 79-82, 2006.
[4] D.Chaiken, C.Fields, K.Kurihara, A.Agrawl, “Directory-Based Cache Coherence in Large-scale Multiprocessors”, IEEE Computer, pp. 49-58, June 1990.
[5] J.Archibald, Jean-Loup Bear, “Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model”, ACM TOCS, vol. 4, pp. 273-298, Nov. 1986
[6] H.Cheong, A.V.Veidenbaum, “A Version Control Approach to Cache Coherence”, In Proc. of 3rd Intl. conference on supercomputing, pp. 322-330, 1989.
[7] Thomas B.Berg, “Maintaining I/O Data Coherence in Embedded Multicore Systems”, IEEE Micro, pp. 10-19, May, 2009.
[8] Dan Tang, Yungang Bao, Weiwu Hu, Mingyu Chen, “DMA Cache: Using On-Chip Storage to Architecturally Separate I/O Data from CPU data for Improving I/O Performance”, In Proc. of 16th Intl. Symposium on HPCA, pp. 1-12, Jan. 2010.
[9] R.Huggahalli, R.Iyer, S.Tetrick, “Direct Cache Access for High Bandwidth Network I/O“, In Proc. of 32nd Intl. Symposium on Computer Architecture, pp. 50-59, 2005.
[10] ARM, ARMv7-R Architecture Reference Manual
[11] Zucker,R.N, Beat,Jean-Loup, “Software versus hardware coherence: performance versus cost”, In Proc. of Intl. Conference, Jan. 1994.
[12] ARM, Cortex-R4 and Cortex-R4F Technical Reference Manual
[13] Carbon: http://www.carbondesignsystems.com
[14] Ashby,T.J., Diaz.P., Cintra,M., “Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters”, IEEE

Trans. Computers, pp. 472-483, 2011.
[15] A.Sloss,D.Symes,C.Wright, “ARM System Developer’s Guide”, Morgan Kaufmann, 2004.
[16] Hana Na, Changwon Choi, Joonwhan Yi, “Mass Data Transfer Using DMAC along with Cache Flush”, IEIE, pp. 71-74, June 2014
[17] ARM, AMBA AXI Protocol

저 자 소 개

나 하 나(학생회원)
2014년 광운대학교 컴퓨터공학과 학사졸업.
2014년～2016년 광운대학교 컴퓨터공학과 석사졸업.
2016년～현재 한화탈레스 연구원.
<주관심분야 : 상위수준 설계, 저전력 설계>

이 준 환(평생회원)
1991년 연세대학교 전자공학과 학사졸업.
1998년 Univ. of Michigan, EECS 석사졸업.
2002년 Univ. of Michigan, EECS 박사졸업.
1991년～1995년 삼성전자 시스템LSI 연구원
2003년～2008년 삼성전자 통신연구소 수석연구원.
2008년～현재 광운대학교 컴퓨터공학과 부교수.
<주관심분야 : SoC 구조설계, 저전력 설계, 반도체 설계, Computer Vision>