효율적이고 확장성 있는 다중-프로세서 시스템 시뮬레이터

김희경○, 박해우, 양회석, 하순회
서울대학교 전기컴퓨터공학부
{hkkim, starlet, hyang, sha}@iris.snu.ac.kr

Efficient and Extensible Multi-processor System Simulator

Heekyung Kim○, Hae-woo Park, Hoeseok Yang, Soonhoi Ha
School of EECS, Seoul National University, Korea

요 약

임베디드 시스템이 주목받으면서 개발상의 편의를 위해 시스템 시뮬레이터가 다양한 용도로 사용되고 있다. 시스템이 복잡해지고 소프트웨어의 규모가 커지면서 이러한 시스템 시뮬레이터들에 있어 그 성능은 매우 중요한 이슈가 되고 있는데, 본 논문에서는 공유 메모리를 사용하여 통신하는 다중 프로세서 시스템에서 동기화 횟수를 줄이는 방법을 제안하고 이를 기반으로 한 다중 프로세서 시스템 시뮬레이터를 개발하였다. 이 시뮬레이터는 프로세서 시뮬레이터의 내부를 크게 고치지 않고 공유 메모리 접근만을 가로채 동작이 가능하므로 쉽게 다양한 종류의 프로세서를 연결할 수 있는 확장성 역시 가지고 있다. 제안하는 동기화 기법과 개발된 시뮬레이터는 7개의 프로세서를 사용하여 동작하는 JPEG 인코더 예제의 구동을 통해 테스트되었으며, 이를 통해 인과율을 깨뜨리지 않고도 빠른 시뮬레이션이 가능함을 확인할 수 있었다.

1. 서론

프로세서 시뮬레이터는 오랜 시간 동안 연구, 개발되어 현재도 다양한 용도로 사용되고 있다. 이러한 프로세서 시뮬레이터들은 크게 두 가지 장점을 가지는데, 하나는 실제 프로세서가 없는 상황에서도 소프트웨어를 실행하고 디버깅할 수 있는 환경을 제공하여 개발 비용을 낮출 수 있다는 것이며, 다른 하나는 소프트웨어 실행에 있어 문제가 발생했을 때, 하드웨어가 잘못된 경우를 고려할 필요가 없다는 것이다.

이러한 프로세서 시뮬레이터들은 많은 경우 기능 검증용으로 사용되나, 임베디드 시스템이 주목을 받으면서 시간 검증 역시 프로세서 시뮬레이터들의 중요한 용도 중 하나가 되었다. ARM® Symbolic Debugger armsd™ [1]의 경우 내장되어 있는 ISS(Instruction Set Simulator)에서 시간 정보를 유지하고 있는 프로세서 시뮬레이터의 예이다.

근래에 들어서는 단순히 하나의 프로세서를 시뮬레이션하는 것이 아니라 여러 프로세서들과 컴포넌트들을 사용한 시스템 전체를 시뮬레이션하는 시스템 시뮬레이터들 역시 개발되고 있다. Mentor Graphics Seamless™, ARM® SoC Designer™ 등은 임베디드 시스템에서의 기능 및 시간 검증을 위해 개발된 도구들이다.

시스템이 복잡해지고 소프트웨어의 규모가 커지면서 시뮬레이터의 성능이 매우 중요해지고 있는데, 단일 프로세서 시뮬레이터의 경우 ARM 프로세서 시뮬레이터 기준으로 1M~10Mcycles/sec 정도의 성능이 나오지만, 이를 여럿 사용하여 다중 프로세서 시뮬레이터를 만드는 경우 동기화 문제로 인해 그 성능이 급격히 떨어지게 된다. 즉, 프로세서들끼리 공유하는 데이터들에 대한 접근 등을 시간 순서대로 정렬하는 데 드는 비용이 추가되어 전체 시뮬레이터의 성능이 감소하는 것이다.

이러한 성능 감소 문제를 해결하기 위해서는 동기화에 드는 비용을 줄여야 하는데, 이를 위해서는 동기화 횟수를 줄이거나 동기화 한 번에 드는 비용을 줄이는 방법이 있다. 본 논문에서는 이 중 동기화 횟수를 줄이는 동기화 기법 및 이에 기반을 둔 다중 프로세서 시뮬레이터를 제안하였다. 이는 프로세서들이 상호 통신을 위해 공유 메모리를 사용한다는 가정 하에 동기화 횟수를 대폭 줄여 시뮬레이션 성능을 높인 것이다.

임베디드 시스템 개발에서 요구되는 시스템 시뮬레이터의 조건 중 다른 하나는 다양한 프로세서 시뮬레이터

를 지원할 수 있어야 한다는 것이다. 많은 프로세서 개발사들이 자사의 프로세서에 대한 시뮬레이터를 제공하고는 있으나, 대부분 외부로의 인터페이스에 대한 수정만 가능하게 할 뿐, 시뮬레이터 코어를 공개하고 있지는 않은데, 이는 다양한 프로세서를 채용한 임베디드 시스템 시뮬레이터를 만드는 데 어려움을 주는 요소이다.

본 논문에서 제안하는 시스템 시뮬레이터는 프로세서 시뮬레이터의 공유 메모리 접근만을 가로채 동작이 가능하므로 다양한 프로세서 시뮬레이터를 사용하도록 쉽게 확장 가능하다는 장점이 있다.

이어지는 본 논문의 내용은 다음과 같다. 2장에서는 관련된 연구들을 소개하고, 3장에서는 제안하는 동기화 기법을 소개하며, 4장에서는 이에 기반을 둔 다중 프로세서 시스템 시뮬레이터에 대해 설명한다. 5장에서는 실험을 통해 시스템 시뮬레이터의 동작을 검증하고, 6장에서는 본 논문의 결론을 맺고자 한다.

2. 관련 연구

다중 프로세서 시뮬레이션의 성능을 향상시키기 위한 방법은 크게 두 가지로 나누어볼 수 있다.

첫 번째는 보다 높은 추상화 수준에서 시뮬레이션을 수행하여 성능을 높이는 방법이다. SystemC[2] 등의 언어를 사용하여 TLM(Transaction level model) 수준에서 시뮬레이션을 수행하게 되면 RTL(Register transfer level) 수준에서보다 정확도를 희생하는 대신 빠른 시뮬레이션을 수행할 수 있다. 그 외에 실제로 시간을 계산하지 않고 예측한 지연 시간을 코드에 삽입하여 빠른 시간에 시뮬레이션을 가능하게 하는 지연 시간 표기 기법이 있다.[3][4][5][6]

두 번째는 동기화에 소요되는 시간을 줄여 시뮬레이션 성능을 높이는 것이다. 시뮬레이션이 실제와 같은 동작을 보이기 위해서는 각 프로세서들의 단위 동작들 사이에 시간 순서가 지켜져 인과율(causality)이 성립해야 하므로 프로세서 간 동기화는 필수적이다. 그러나 동기화를 위해서는 많은 시간 비용이 필요하기 때문에 시뮬레이션 성능 향상을 위해서는 이를 줄이는 것이 매우 중요하다. 동기화에 소요되는 시간 비용을 줄이는 방법에는 동기화 횟수를 줄이는 것과 동기화 한 번에 드는 시간을 줄이는 방법이 있는데, 후자보다는 전자가 성능 향상에 많은 도움이 되기 때문에 이를 위한 연구가 주로 이루어지고 있다.

일반적으로 사용되는 동기화 기법은 락-스텝 동기화 기법으로 이는 매 cycle마다 동기를 맞추는 것이다. Mentor Graphics Seamless™와 같이 정확도가 중요한 시뮬레이션 도구의 경우 이러한 동기화 기법을 사용한다. 이보다 성능을 높인 방법으로 낙관적 기법(optimistic approach)[7]이 있다. 이는 동기화를 맞추지 않아도 문제가 없을 것이라는 가정으로 각 프로세서 시뮬레이터들을 진행시키다가 문제가 발생하는 경우 마지막으로 기억했던 곳으로 돌아가 다시 시뮬레이션을 수행하는 것인데, 이는 시뮬레이터에서 특정 시점의 상태를 저장하는 기능과 해당 시점으로 되돌아가는 기능을 지원할 경우에만 사용이 가능하다는 문제가 있다.

본 논문은 두 번째 방법에 초점을 둔 것으로, 동기화 횟수를 줄이는 방법을 사용한다. 동기화는 공유 메모리에 접근하는 경우에 수행하며, 프로세서가 자기 자신에게만 할당된 지역 메모리를 사용하는 경우에는 다른 프로세서와의 동기화 문제가 발생하지 않는다고 가정한다.

ARMⓇ SoC Designer™와 같은 시스템 시뮬레이터들은 근래에 많이 활용되고 있는 시스템 시뮬레이터이다. 이러한 시스템 시뮬레이터들은 충분한 프로세서 컴포넌트 라이브러리가 제공될 경우에 쉽게 활용될 수 있지만, 라이브러리로 제공되지 않는 프로세서 시뮬레이터를 붙여서 동작하는 것은 불가능하거나 큰 노력이 드는 것이 보통이다.

CATS[8] 시스템 시뮬레이터는 프로세서 시뮬레이터 SimpleScalar[9]를 수정해서 개발한 시스템 시뮬레이터인데 이 경우에도 프로세서 시뮬레이터의 내부를 크게 고쳐서 개발된 경우이다.

본 논문에서 제안하는 시스템 시뮬레이터는 프로세서 시뮬레이터의 수정을 하지 않거나, 외부 메모리 접근 부분 정도의 수정만으로 동작이 가능하기 때문에 확장성 면에서 이점을 가지고 있다고 할 수 있겠다.

3. 제안하는 동기화 기법

3.1. 시뮬레이션 대상 시스템의 구조와 그 명세

본 절에서는 동기화 기법을 소개하기에 앞서 시스템의 명세에 대해 설명한다.

시뮬레이션할 시스템의 컴포넌트로는 프로세서, 프로세서 별 지역 메모리, 공유 메모리가 있다. 각각의 컴포넌트는 다수 존재할 수 있으며, 공유 메모리의 경우 접근을 허용하는 프로세서를 지정할 수 있다. 본 논문에서 제안하는 시뮬레이터에서는 이러한 시스템 구성 정보를 XML 형식의 파일에서 읽어 동작을 수행하게 된다.

[그림 1]은 시뮬레이션할 시스템의 구조와 그 명세의 예이다. 이 시스템에서는 세 개의 ARM926EJ-S 프로세서가 존재하며 각각의 id를 index라는 번호로 가지게 된다. 각 프로세서들은 독립적인 지역 메모리들을 가지고 있고, 세 프로세서가 공유하는 하나의 공유 메모리가 존재한다. 본 명세에서 공유 메모리는 세 프로세서 모두 접근이 허용되어 있다. (XML 파일에 접근이 허용된 프로세서의 id들이 기술되어 있음을 볼 수 있다.) 여러 공유 메모리를 지정하기 위해서는 <sharedMem> </sharedMem> 태그를 두어 반복적으로 작성해주고 주소 범위 및 접근 허용 프로세서들을 기술해주면 된다.

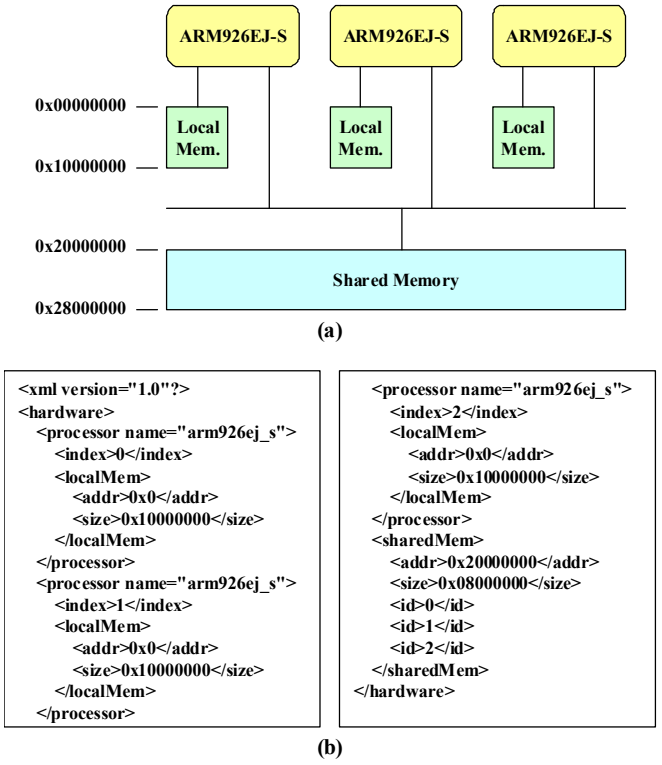

그림 1. 시뮬레이션할 시스템 구조의 예와 그 명세

3.2. 동기화 및 시간 보정

제안하는 동기화 알고리즘에서는 각 프로세서 시뮬레이터들이 공유 메모리에 접근하는 시점에서 해당 프로세서 시뮬레이터를 멈추고 동기화를 수행한다. 즉, 각 프로세서 시뮬레이터는 자신의 지역 메모리에서 데이터를 읽고 쓰는 중에는 동작을 멈추지 않고 시뮬레이션을 진행할 수 있다.

각 프로세서 시뮬레이터들은 공유 메모리 접근 시 시뮬레이션 커널에 요청을 보내고 요청이 처리될 때까지 기다린다. 시뮬레이션 커널은 요청이 들어오면 바로 처리하는 것이 아니라, 모든 프로세서 시뮬레이터들의 요청들을 보아 가장 먼저 처리되어야 할 요청부터 처리하고 해당 프로세서 시뮬레이터가 진행할 수 있도록 해준다.

이러한 과정에서 순서가 지켜지지 않는 일이 없도록 하기 위해서, 시뮬레이션 커널은 모든 프로세서 시뮬레이터들이 공유 메모리에 접근하거나 종료될 때까지 기다린다. 모든 프로세서 시뮬레이터가 공유 메모리에 접근한 경우 그 중 가장 진행이 늦은 프로세서 시뮬레이터를 선택하면 이 시뮬레이터가 접근한 시점 이전에 다른 프로세서가 요청한 공유 메모리 접근은 없다는 것이 보장되므로 해당 프로세서 시뮬레이터의 요청을 처리하고 진행시킬 수 있다. 어떤 프로세서 시뮬레이터가 종료한 경우 이 프로세서 시뮬레이터는 향후 공유 메모리 접근이 없을 것으로 보고 진행할 수 있다.

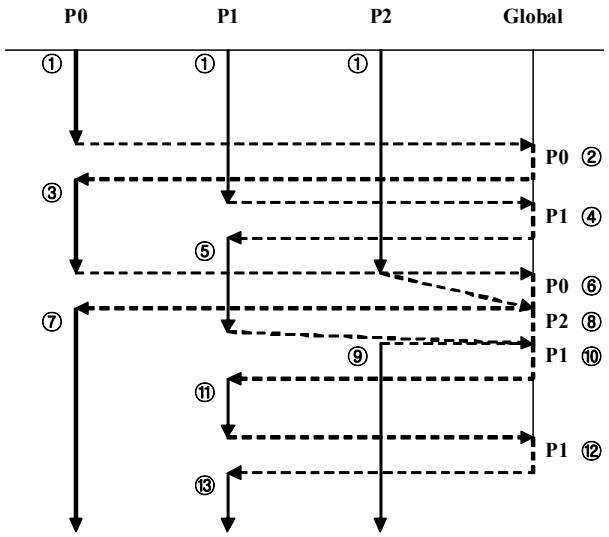

그림 2. 프로세서 시뮬레이터 간 동기화의 예

[그림 2]는 프로세서 시뮬레이터 간 동기화의 예이다. 먼저 ①모든 프로세서 시뮬레이터들이 공유 메모리에 접근할 때까지 각각 진행하도록 한다. 모두 공유 메모리 요청을 진행한 경우 가장 먼저 접근한 ②P0의 요청을 처리한 다음 ③P0가 다음 공유 메모리 접근할 때까지 진행할 수 있도록 한다. P0 진행 후 공유 메모리 접근을 하게 되면 가장 진행이 늦은 ④P1의 요청을 처리하고 ⑤진행할 수 있게 한다. 이후 P0, P2의 요청이 동시에 들어오는데, 이런 경우 앞 번호의 프로세서에 우선순위가 있다고 보고 ⑥P0의 요청을 먼저 처리하고 진행한 뒤 ⑧P2의 요청을 처리, ⑨진행하게 된다. 그 사이에 들어온 ⑩P1의 요청은 그 이후에 처리되며 ⑪⑫⑬그 이후의 진행도 앞서와 유사하다.

시뮬레이션 커널에서는 시간 보정을 위해 몇 가지 정보를 기록해두는데, 한 가지는 각 프로세서가 공유 메모리 접근으로 인해 지연된 시간이며, 한 가지는 각 공유 메모리 별로 마지막 처리를 완료한 시간이다. 각 프로세서 시뮬레이터는 자신이나 다른 프로세서 시뮬레이터가 메모리 접근으로 인해 얼마나 지연되었는지, 어떤 공유 메모리에 접근할 때 다른 프로세서의 해당 메모리 접근으로 지연되는 일이 있는지 알 수 없으므로, 이를 시뮬레이션 커널이 관리하는 것이다.

```
; selected_iss = 가장 느리게 진행된 ISS
; mem = 접근되는 공유 메모리
; iss_cycles = 해당 ISS가 진행한 시간 (메모리 접근 지연 시간 제외)
; delayed_cycles = 해당 ISS가 메모리 접근으로 지연된 시간
; mem_last_cycles = 해당 공유 메모리가 마지막 처리를 완료한 시간

if ((iss_cycles[selected_iss] + delayed_cycles[selected_iss])
      >= mem_last_cycles[mem]) {

   delayed_cycles[selected_iss]
         += memory_access_time(mem, access_type);
   mem_last_cycles[mem]
         = iss_cycles[selected_iss] + delayed_cycles[selected_iss];

} else {

   mem_last_cycles[mem]
         += memory_access_time(mem, access_type);
   delayed_cycles[selected_iss]
         = mem_last_cycles[mem] - iss_cycles[selected_iss];

}
```

그림 3. 시뮬레이션 커널의 시간 보정 알고리즘

[그림 3]은 시간 보정 알고리즘을 나타낸 코드인데, if 문을 만족하는 경우는 앞서 있던 공유 메모리 접근이 모두 처리된 뒤에 해당 공유 메모리 접근을 요청한 경우이며, 만족하지 않는 경우는 앞서 들어온 공유 메모리 접근이 처리되기 전에 공유 메모리 접근을 새로 요청한 경우이다. 전자의 경우 단순히 메모리에 읽고 쓰는 지연 시간을 늘리는 것으로 충분하지만 후자의 경우 앞서 들어온 요청들의 처리가 완료된 시점까지 충돌로 인해 추가 지연되는 것으로 보는 처리가 필요하여 두 가지를 나누어 작성하였다.

3.3. 동기화 횟수 분석

제안한 방법은 락-스텝 동기화 기법에 비해 동기화 횟수를 줄여 동기화에 드는 비용을 줄일 수 있다. 먼저, 락-스텝 동기화 기법을 이용할 때 걸리는 시뮬레이션 시간을 정리하면 식 (1)과 같다.[10]

$$\sum_{\forall i}\{T_{\max}\times(st_i+sync)\}+T_{trans}\times st_{trans} \quad \text{... (1)}$$

T_{max} : 시뮬레이션 cycle 중 가장 긴 cycle
st_i : 시뮬레이터 I에서 한 cycle을 진행하는 데 필요한 시뮬레이션 시간
sync : 동기화를 한 번 수행하는 데 드는 시간
T_{trans} : 공유 메모리에 접근하는 트랜잭션의 개수
st_{trans} : 트랜잭션 하나를 시뮬레이션 하는 시간

이에 비해 제안하는 동기화 기법을 사용할 때 소요되는 시간은 식 (2)와 같다.

$$\sum_{\forall i}\{T_i\times st_i\}+T_{trans}\times(st_{trans}+sync) \quad \text{... (2)}$$

Ti : 시뮬레이터 i의 cycle

두 식을 비교하면 동기화 횟수가 줄어들고, 이로 인해 성능 향상을 얻을 수 있음을 알 수 있다. 식에서 알 수 있듯이 제안하는 동기화 기법의 효과는 전체 실행 cycle에 비해 공유 메모리에 접근하는 트랜잭션 수가 적을수록 더 크게 나타난다.

4. 시스템 시뮬레이터의 구조

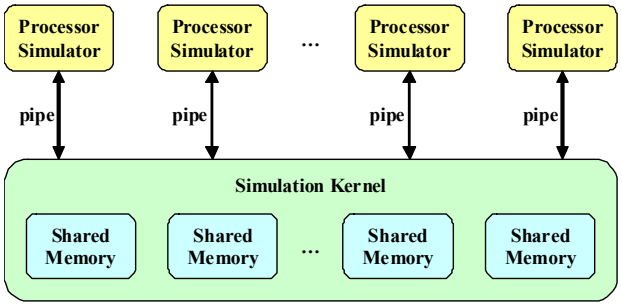

그림 4. 전체 시스템 시뮬레이터 구조의 개략

제안하는 동기화 기법에 기반을 둔 시스템 시뮬레이터는 대략 [그림 4]와 같은 구조를 지닌다. 하나의 시뮬레이션 커널이 여러 개의 프로세서 시뮬레이터와 연결되어 있으며, 파이프로 통신을 수행하게 된다. 공유 메모리에 대한 각종 정보 역시 시뮬레이션 커널이 가지고 있어 동기화에 사용하게 된다.

4.1. 프로세서 시뮬레이터

각 프로세서 시뮬레이터들은 각기 지역 메모리를 가지

며 공유 메모리를 사용하는 경우에만 파이프를 통해 시뮬레이션 커널에 접근 요청을 보낸다.

많은 프로세서 시뮬레이터들이 버스나 메모리 접근에 대해서 외부로의 인터페이스를 제공하기 때문에, 공유 메모리 접근 인터페이스가 외부에 제공되는 경우 프로세서 시뮬레이터 부분은 거의 수정하지 않고도 사용할 수 있다.

프로세서 시뮬레이터는 공유 메모리에 접근 할 때 공유 메모리 접근 명령과 관련 정보들을 시스템 시뮬레이터로 보내고 결과를 기다리며, 결과가 돌아오면 진행을 계속하게 된다.

4.2. 시뮬레이션 커널

시뮬레이션 커널은 파이프를 통해 각 프로세서 시뮬레이터에 제어 명령을 보내거나 프로세서 시뮬레이터가 보낸 요청에 대한 응답을 보낸다.

시뮬레이션 커널이 하는 가장 중요한 역할은 3.2. 절에서 살펴보았듯이 각 프로세서의 지역 시각을 시스템 내의 광역 시각으로 변환하여 처리 순서를 배열하고, 인과율을 유지하는 것이다. 이는 메모리 접근 시간에 대한 연산이나 공유 메모리 동시 접근으로 인한 충돌 고려 역시 포함하고 있다. [그림 5]는 시뮬레이션 커널의 동작을 간략하게 나타낸 것이다.

```
while (there is any processor alive)
{
   wait for a memory access request or a termination message
      from the current running processor simulator

   if (a termination message has come)
      remove the processor simulator from the processor list

   find the earliest request R which is not handled
   transact the request R
   compensate the timing information

   give the result to the simulator S which requested R
      and let S continue to run
}
```

그림 5. 시뮬레이션 커널의 동작

시뮬레이션 커널은 프로세서 시뮬레이터로부터 공유 메모리 접근 요청이 들어오면 이에 대한 결과를 바로 보내주는 것이 아니라 일단은 대기하고 3.2. 절에서 설명했듯이 모든 동기화 시뮬레이터가 요청을 보내거나 종료한 상황이 왔을 때 가장 진행이 늦은 프로세서 시뮬레이터의 요청을 처리하게 된다.

5. 실험

제안한 동기화 기법과 이에 기반을 둔 다중 프로세서 시스템 시뮬레이터를 테스트하기 위해 본 장에서는 JPEG 인코더를 예제로 사용하였다. JPEG 인코더는 [그림 6]과 같이 7개의 단계를 거쳐 비트맵 파일을 JPEG 파일로 만들어내게 된다.

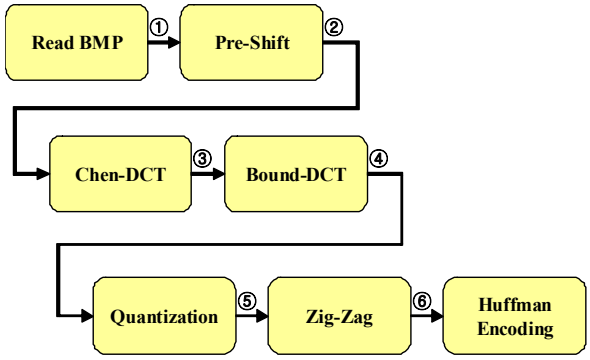

그림 6. JPEG 인코더의 블록 다이어그램

실험에서는 이 7개의 단계를 각각 하나의 프로세서에 할당하고, 공유 메모리를 두어 이들 사이에서 통신이 일어날 수 있도록 하였는데, [그림 6]의 ①②③④⑤⑥은 각 단계 사이의 버퍼를 나타낸 것으로 여기서는 ①②③을 하나의 공유 메모리에, ④⑤⑥을 하나의 공유 메모리에 배정하였다.

```
<xml version="1.0"?>
<hardware>
  <processor name="arm926ej_s">
    <index>0</index>
    <localMem>
      <addr>0x0</addr>
      <size>0x10000000</size>
    </localMem>
  </processor>
  <processor name="arm926ej_s">
    <index>1</index>
    <localMem>
      <addr>0x0</addr>
      <size>0x10000000</size>
    </localMem>
  </processor>
  <processor name="arm926ej_s">
    <index>2</index>
    <localMem>
      <addr>0x0</addr>
      <size>0x10000000</size>
    </localMem>
  </processor>
                :
```

```
  <processor name="arm926ej_s">
    <index>6</index>
    <localMem>
      <addr>0x0</addr>
      <size>0x10000000</size>
    </localMem>
  </processor>
  <sharedMem>
    <addr>0x20000000</addr>
    <size>0x00030000</size>
    <id>0</id>
    <id>1</id>
    <id>2</id>
    <id>3</id>
  </sharedMem>
  <sharedMem>
    <addr>0x20030000</addr>
    <size>0x00030000</size>
    <id>3</id>
    <id>4</id>
    <id>5</id>
    <id>6</id>
  </sharedMem>
</hardware>
```

그림 7. JPEG 인코더 예제 실험에 사용된 시스템 구조 명세

[그림 7]은 JPEG 인코더 예제 실험에 사용된 시스템 구조 명세이다. 총 7개의 프로세서(0번~6번)와 2개의 공유 메모리가 명세되어 있음을 알 수 있다.

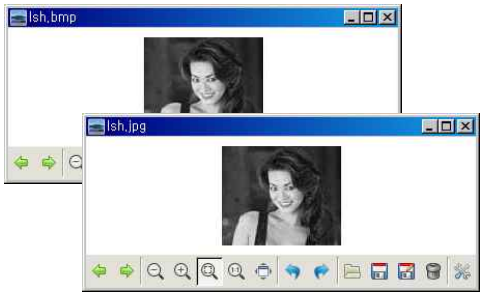

그림 8. JPEG 인코더의 수행 결과

각각의 블록에 해당하는 7개의 프로그램을 작성하여 본 시스템 시뮬레이터에서 동작시킨 결과 [그림 8]과 같이 정상적인 인코딩 결과를 얻을 수 있었다. 총 시뮬레이션 cycle이 2179353 cycle임에 비해 동기화 횟수는 176653회로, 매 cycle 동기화를 진행하는 락-스텝 동기화를 사용했을 경우(2179353회)에 비해 월등히 적은 횟수의 동기화를 수행했음을 알 수 있었다.

6. 결론 및 향후 과제

본 논문에서는 다중 프로세서 시스템 시뮬레이션을 수행함에 있어 동기화 횟수를 줄여 성능을 높이는 방법을 제안하고 이에 기반을 둔 다중 프로세서 시스템 시뮬레이터를 개발하였다. 시뮬레이션 커널은 프로세서 시뮬레이터와 공유 메모리 인터페이스만으로 연결되기 때문에 다양한 프로세서 시뮬레이터들을 다수 사용하여 동작할 수 있는 확장성 있는 구조를 갖추고 있다.

제안한 다중 프로세서 시스템 시뮬레이터는 7개의 프로세서를 사용한 시스템 구조 하에서 JPEG 인코더 예제를 구동하여 테스트한 결과 인과율을 해치지 않고 잘 동작함을 확인할 수 있었다.

본 논문에서 제안한 방법은 공유 메모리만을 프로세서 간 통신에 사용하는 경우를 전제하고 있는데, 현재 프로세서들 위에서 동작하는 소프트웨어 간의 동기화를 위해 하드웨어 락(lock)과 같은 요소들 역시 시스템 시뮬레이터에 추가한 상태이며, 향후 다른 방식의 통신을 고려한 시뮬레이터의 지원에 대해서도 연구를 진행할 예정이다.

▶ 감사의 글

본 연구는 BK21 프로젝트, 교육과학기술부 도약연구지원사업(R17-2007-086-01001-0)에 의해 지원되었습니다. 또한 서울대학교 컴퓨터신기술연구소와 IDEC은 본 연구에 필요한 기자재들을 지원해주었습니다.

▶ 참고 문헌 및 자료

[1] armsd. ARM RealView Development Suite. http://www.arm.com

[2] SystemC. Open SystemC Initiative. http://www.systemc.org

[3] Sungjoo Yoo, Gabriela Nicolescu, Lovic Gauthier, and Ahmed Jerraya, “Automatic Generation of Fast Timed Simulation Models for Operating Systems in SoC Design”, In Proc. Design Automation and Test in Europe, Mar. 2002.

[4] Andreas Gerstlauer, Haobo Yu and Daniel Gajski, “RTOS Modeling for System-Level Design”, In Proc. Design Automation and Test in Europe, Mar. 2003.

[5] Zhengting He, Aloysius Mok, Cheng Peng, “Timed RTOS modeling for Embedded System Design”, In Proc. Real Time and Embedded Technology and Application Symposium, Mar. 2005.

[6] Aimen Bouchhima, Sungjoo Yoo, and Ahmed Jerraya, “Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model”, In Proc. Asia South Pacific Design Automation Conference, Jan. 2004.

[7] Sungjoo Yoo and Kiyoung Choi, “Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model”, In Proc. 6th International Workshop on Hardware/Software Co-Design, Mar. 1998.

[8] Dohyung Kim, Soonhoi Ha, and Rajesh Gupta, “CATS: cycle accurate transaction-driven simulation with multiple processor simulators”, In Proc. Design Automation and Test in Europe, Apr. 1997.

[9] SimpleScalar tools. SimpleScalar LLC. http://www.simplescalar.com

[10] Youngmin Yi, Dohyung Kim, and Soonhoi Ha, "Virtual Synchronization Technique with OS modeling for Fast and Time-accurate Cosimulation," In Proc. Hardware/Software Codesign and System Synthesis, pp.1-6, Oct. 2003.