가상 동기화 기법을 이용한 SystemC 통합시뮬레이션의 병렬 수행

(Parallel SystemC Cosimulation using Virtual Synchronization)

이 영 민 † 권 성 남 † 하 순 회 ††
(Youngmin Yi) (Seongnam Kwon) (Soonhoi Ha)

요 약 이 논문에서는 여러 개의 소프트웨어 혹은 하드웨어 컴포넌트가 존재하는 MPSoC(Multiprocessor-System-on-a-chip) 아키텍처를 빠르면서도 정확하게 통합시뮬레이션 하는 내용을 다룬다. 복잡한 시스템을 설계하기 위해서 MPSoC 아키텍처가 점점 일반화되고 있는데, 이러한 아키텍처를 통합시뮬레이션 할 때는 시뮬레이터의 개수가 증가하고 그에 따라 시뮬레이터들 간의 시간 동기화 비용도 증가하므로 전체적인 통합시뮬레이션 성능이 감소된다. 최근의 통합시뮬레이션 연구들에 의해서 등장한 SystemC 통합시뮬레이션 환경이 빠른 성능을 보이고 있으나, 시뮬레이터의 개수가 증가할수록 성능은 반비례한다. 본 논문에서는 효율적인 시간동기를 통해 통합시뮬레이션의 성능을 증가시키는 기법인 가상동기화 기법을 확장하여, (1) SystemC 커널을 수정하지 않고도 가상 동기화 기법을 적용한 SystemC 통합시뮬레이션을 수행할 수 있고, (2) 병렬적으로 가상동기화 기법을 수행할 수 있게 하였다. 이를 통해 SystemC 통합시뮬레이션의 병렬적인 수행이 가능해졌는데, 널리 알려진 상용 SystemC 통합시뮬레이션 도구인 MaxSim과 비교하였을 때, H.263 디코더 예제의 경우 11배 이상의 성능 증가를 얻었고 정확도는 5% 이내로 유지되었다.
키워드 : 하드웨어/소프트웨어 통합시뮬레이션, 가상동기화 기법, 병렬 시뮬레이션, MPSoC, SystemC

Abstract This paper concerns fast and time accurate HW/SW cosimulation for MPSoC (Multi-Processor System-on-chip) architecture where multiple software and/or hardware components exist. It is becoming more and more common to use MPSoC architecture to design complex embedded systems. In cosimulation of such architecture, as the number of the component simulators participating in the cosimulation increases, the time synchronization overhead among simulators increases, thereby resulting in low overall cosimulation performance. Although SystemC cosimulation frameworks show high cosimulation performance, it is in inverse proportion to the number of simulators. In this paper, we extend the novel technique, called virtual synchronization, which boosts cosimulation speed by reducing time synchronization overhead: (1) SystemC simulation is supported seamlessly in the virtual synchronization framework without requiring the modification on SystemC kernel (2) Parallel execution of component simulatorswith virtual synchronization is supported. We compared the performance and accuracyof the proposed parallel SystemC cosimulation framework with MaxSim, a well-known commercial SystemC cosimulation framework, and the proposed one showed 11 times faster performance for H.263 decoder example, while the accuracy was maintained below 5%.
Key words : HW/SW Cosimulation, Virtual synchronization, Distributed parallel cosimulation, MPSoC, SystemC

· 본 연구는 국가지정연구실 프로그램(번호 M1-0104-00-0015), 두뇌한국 21 프로젝트, SystemIC 2010 프로젝트, 정보통신선도기반기술개발사업, IT-SoC 핵심설계인력양성사업에 의해 지원되었으며, 이 연구를 위해 연구장비를 지원하고 공간을 제공한 서울대학교 컴퓨터연구소, 서울대학교 반도체공동연구소, 반도체설계 교육센터에 감사드립니다.
† 학생회원 : 서울대학교 전기.컴퓨터공학부
ymyi@iris.snu.ac.kr
ksn@iris.snu.ac.kr
†† 정 회 원 : 서울대학교 전기.컴퓨터공학부 교수
sha@iris.snu.ac.kr
논문접수 : 2006년 6월 15일
심사완료 : 2006년 10월 17일

1. 서 론

하드웨어/소프트웨어 통합시뮬레이션은 성공적인 하드웨어/소프트웨어 통합설계를 가능하게 해주는 핵심요소이다. 실제 프로토타이핑 보드가 제작되기 이전에 컴포넌트 시뮬레이터들을 통합시뮬레이션 함으로써 전체 시스템의 기능적 오류와 시간적 오류를 검출할 수 있고, 전체 시스템의 수행성능을 예측해볼 수 있다. 한편, 최

근 내장형 시스템의 설계복잡도가 지속적으로 증가하면서, 이에 대한효과적인 해결책으로서 멀티프로세서 시스템-온-칩(Multi-Processor System-on-chip) 구조가 보편화되고 있다. MPSoC 구조에서는 CPU 혹은 DSP와 같은 소프트웨어 컴포넌트들이 여러개 존재하며 또한 가속기 역할을 하는 하드웨어 컴포넌트(들)이 존재한다. 이러한 시스템 구조를 통합시뮬레이션 하기 위해서는 이전보다 많은 개수의 컴포넌트 시뮬레이터들을 수행시켜야 하고, 정확한 통합시뮬레이션을 위해서는 각 컴포넌트 시뮬레이터들 간에 시간 동기를 하면서 시뮬레이션을 진행해야 한다. 따라서, 일반적으로 전체통합시뮬레이션의 속도는 시뮬레이터의 개수가 증가함에 따라 감소된다. 통합시뮬레이션의 속도가 빨라야 더 넓은 설계공간을 탐색할 수 있기 때문에 통합시뮬레이션의 성능을 증가시키는 것은 하드웨어/소프트웨어 통합설계의 주요한 연구가 되어 왔으며, MPSoC 구조와 같이 시뮬레이터의 개수가 증가해도 빠른 통합시뮬레이션 성능을 유지하는 것이 이 논문의 주제이다.

최근의 SystemC 통합시뮬레이션 환경들[1-4]에서는 소프트웨어 컴포넌트는 명령어집합 시뮬레이터로 시뮬레이션하고, 이를 제외한 나머지 컴포넌트들은 SystemC로 모델링하여 시뮬레이션한다. 따라서 기존의 RTL (Register-Transfer-Level) 시뮬레이터가 포함된 통합시뮬레이션 환경에 비해 싸이클 정확도를 유지하면서도 매우 높은 성능을 내고 있다. 그러나 SystemC 통합시뮬레이션 환경에서도 기존의 전통적인 통합시뮬레이션 환경에서와 동일한 시간 동기화 기법을 사용하기 때문에, 시뮬레이터들 간의 시간 동기화에 따른 성능저하 문제가 여전히 존재한다. 그림 1은 대표적인 상용SystemC 통합시뮬레이션 도구인 MaxSim[1]에서 얻은 실측값으로서(Intel Dual Xeon 1.8GHz 서버), 이러한 성능저하 문제를 보여주고 있다. 소프트웨어 컴포넌트로 ARM 시뮬레이터를 사용하였고, 시뮬레이터의 개수를 증가시키면서 각 시뮬레이터에서는 인접한 시뮬레이터로부터 계산된 값을 넘겨받아 1씩 증가시켜 인접한 다른 시뮬레이터에 전달하는 간단한 예제를 수행시켰다. 그림 1에서 전체 통합시뮬레이션의 성능은 시뮬레이터의 개수에 반비례함을 확인할 수 있다. 일반적인 SystemC 시뮬레이션 환경에서와 마찬가지로 MaxSim에서도, 각각의 시뮬레이터들의 시간 동기를 위해 시뮬레이터들을 1 싸이클씩 번갈아 가며 순차적으로 진행시키기 때문에 시뮬레이터의 개수 N 이 증가할수록 통합시뮬레이션의 성능은 1/N으로 감소한다. 이 문제를 해결하기 위해서는 SystemC 통합시뮬레이션이 병렬적으로 수행될 수 있어야 하고, 병렬적으로 수행되는 시뮬레이터들 사이에 시간 동기가 효율적으로 이루어져야 한다. 만약 매 싸이클마다 시간 동기를 한다면 병렬적 수행에 의한 성능 개선보다 시간 동기 비용에 의한 성능 감소가 훨씬 더 크다. SystemC 통합시뮬레이션 환경에서는, 매 사이클 수행되는 시간 동기 비용을 줄이기 위해 각 시뮬레이터들이 별도의 프로세스가 아닌 함수 형태로 구현되어 전체 프레임워크가 하나의 프로세스로 구동된다. 이런 환경에서는 시뮬레이터들이 서로 다른 서버에 분산되어 수행될 수가 없고, 분산될 수 있다고 가정하여도 시뮬레이터 클록과 전역 클록이 매 사이클 동기화되어야 하기 때문에 병렬적으로 수행이 될 수 없다.

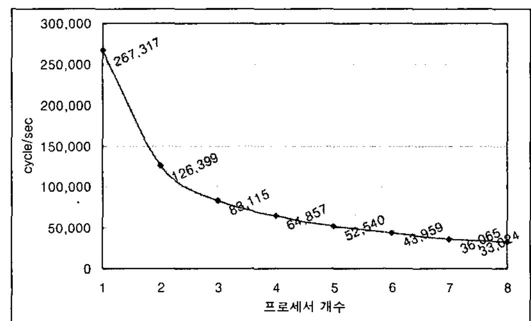


그림 1 프로세서 시뮬레이터 개수 증가에 따른 통합시뮬레이션 성능의 감소

한편, 최근에 제안된 바 있는 가상 동기화 기법[5]을 사용하면 전체 통합시뮬레이션 시간 중에서 시뮬레이터들간의 시간 동기화에 소요되는 시간의 비중이 거의 0에 가깝게 되어 통합시뮬레이션의 성능이 증가된다. 본 논문에서는 SystemC 통합시뮬레이션을 병렬적으로 수행하기 위해서, 가상 동기화 기법을 SystemC 시뮬레이터에 SystemC 커널의 수정 없이 적용하는 구조를 제안하고, 가상 동기화 기법을 이용하여 병렬적으로 시뮬레이터들을 수행시킬 수 있도록 가상 버퍼 기법을 제안한다. 즉, SystemC 시뮬레이터를 사용하면서도 병렬적인 수행이 가능하여 전체 통합시뮬레이션 성능이 증가하는데, 이것이 본 논문이 기여하는 바이다. 본 논문의 또 다른 주요 기여도는 가상 동기화 기법의 성능 개선을 수식적으로 분석하였다는 점이다.

본 논문의 구성은 다음과 같다. 다음 장에서는 주요 관련 연구들을 소개하고, 3장에서 시간 동기화 문제를 살펴보고 전통적인 시간 동기화 기법을 소개한다. 4장에서는 가상 동기화 기법을 설명하고 5장에서는 제안하는 확장된 가상동기화 기법을 설명한다. 그리고, 실험 결과들을 6장에서 정리하였고, 마지막으로 7장에서 결론을 맺는다.

2. 관련 연구들

상용의 SystemC 통합시뮬레이션 도구들을 포함하여 정확한 성능 검증을 지원하는 SystemC 통합시뮬레이션 환경들[1-3]에서 시뮬레이터들간의 동기는 매 사이클마다 시간 동기를 하는 락스텝(lock-step) 방법으로 이루어진다. 또한, RTL 하드웨어 시뮬레이터를 사용하여 속도가 매우 느리지만 그만큼 가장 자세하고 정확한 통합 시뮬레이션을 지원하는 상용 도구인 SeamlessCVE[6]에서도 락스텝 방법으로 시간 동기화를 한다.

시간동기화 비용을 줄임으로써 통합시뮬레이션의 성능을 높이고자 한 연구에는 낙관적 기법(optimistic approach)[7]과 최적화된 보수적 기법(optimized conservative approach)[8] 을 들 수 있다. Optimistic approach 에서는, 각 시뮬레이터가 매 사이클 전역 클록과 동기화하지 않고 자신의 클록을 기준으로 과거에 해당되는 이벤트가 도착하지 않을 것이라는 '낙관적' 가정을 하면서 시뮬레이션을 진행한다. 만약 이 가정이 실패할 경우 가장 최근의 체크포인트(checkpoint) 시점으로 되돌아가(rollback) 그 시점 이후 이루어진 모든 시뮬레이션을 취소하고 다시 시뮬레이션을 진행한다. 이 기법은 체크포인트 시점마다 백업을 해야 하는 추가적 비용이 들며, 무엇보다도 일반적인 시뮬레이터들이 롤백(rollback) 기능을 지원하지 않기 때문에 적용하기가 어렵다. 최적화된 보수적 기법(Optimized conservative approach)에서는 시스템 수준에서 다음에 발생할 이벤트를 예측함으로써 그 시점까지는 안전하게 시간 동기화를 하지 않고 진행하는 방법이다. 그러나 이러한 예측이 항상 가능한 것이 아니다. 한편, [4]에서는 시간동기화를 매 사이클마다 하지 않고 매 N 사이클마다 하는 방법을 제안하였는데, 그만큼 통합시뮬레이션의 정확도는 감소하게 된다.

설계탐색 과정에서 하나의 설계를 빨리 평가하기 위해서 주로 통합설계의 상위단계에서 사용하는 지연시간 표기 기법(delay annotation approach)은, 명령어집합 시뮬레이터를 사용하지 않고 하드웨어 IP및 통신 구조도 사이클 정확도로 기술하지 않기 때문에 매우 빠른 성능을 보여준다[9-12]. 이 방법에서는 응용 프로그램의 코드가 타겟 바이너리로 크로스컴파일 되어 타겟 시뮬레이터에서 수행되는 것이 아니라 예측된 지연시간(delay)을 호스트 코드에 삽입하여 호스트 바이너리로 수행한다. 하드웨어 IP도 소프트웨어 코드와 동일하게 SystemC 혹은 SpecC와 같은 TLM(Transaction-Level-Model) 언어로 기술되고 마찬가지로 지연시간이 삽입된다. 시뮬레이터의 추상도(abstraction level)를 높임으로써 정확도를 희생하고 성능을 증가시키는 이와 같은 접근법에서도 시간동기화 문제는 여전히 존재한다. 의미있는 수준의 정확도를 얻기 위해서는 락스텝 방법과 유사하게 가급적 자주 시간동기화를 해야 하는데, 이로 인해 통합시뮬레이션 성능이 떨어지게 된다. 이 문제를 해결하기 위해 [11]에서는, 시간동기화 함수인 consume()을 정의할 때 함수인자로 지연시간 값과 이 함수를 호출하는 모듈(시뮬레이터)이 받아들일 수 있는 인터럽트를 명시하게 함으로써, 만약 지연시간 중간에 인터럽트가 발생한다면 인터럽트를 처리하고 난 후, 원래 지연시간 값에서 인터럽트가 발생한 시점을 뺀 값으로 지연시간 값을 갱신한다. [12]에서도 이와 비슷한 역할을 하는 시간동기화 함수 inc_clock()을 정의하여, 시간동기화 함수가 불필요하게 자주 호출되지 않도록 하면서 정확도를 유지한다. 그러나 이러한 방법은 지연시간 표기 기법(delay annotation approach)에서만 가능한 것으로서 명령어집합 시뮬레이터를 사용하고 통신 구조 및 하드웨어 IP가 사이클 정확도로 기술되는 일반적인 통합시뮬레이션에서는 적용할 수 없다. 또한 지연시간 표기 기법 (delay annotation approach)의 정확도는 정적인 예측값인 지연시간의 정확도로 제한된다.

3. 시간 동기화 문제

기능적으로나 시간적으로 올바른 통합시뮬레이션 결과를 얻기 위해서는 인과성이 지켜져야 하는데, 이는 각 시뮬레이터 입장에서 볼 때 과거의 이벤트가 발생되어서는 안 된다는 것을 의미한다. 만약 컴포넌트가 인터럽트를 받아들이지 않는다고 가정한다면 그 컴포넌트 시뮬레이터는 시스템의 공유 자원에 접근할 때(즉, 다른 컴포넌트 시뮬레이터에 영향을 주거나 영향을 받는 시점)까지 다른 시뮬레이터들과 동기화를 하지 않고 시뮬레이션을 진행하여도 안전하다. 그러나 인터럽트가 존재한다면 언제 인터럽트가 발생할 지 모르므로 결국 매 사이클마다 외부 시뮬레이터들과 동기를 맞춰야 하며 이것이 락스텝 동기화 기법이다. 이 기법은 가장 직관적이며 구현이 쉽지만 매 사이클마다 동기를 하므로 비용이 가장 크다. 락스텝 동기화 기법을 사용했을 때의 통합시뮬레이션 시간을 정형적으로 분석한 내용이 정의 1[5]에 기술되어 있다.

정의 1. 락스텝(lock-step) 동기화 기법을 사용할 때의 통합시뮬레이션 시간

T: 전체 시뮬레이션 사이클

st_i: 시뮬레이터 i에서 한 사이클을 진행시키는데 필요한 시뮬레이션 시간

$sync$: 시간 동기화를 한 번 수행하는데 드는 시간

T_{trans}: 전체 통신 트랜잭션(읽기/쓰기)의 개수

st_{trans}: 통신 트랙잭션(읽기/쓰기) 하나를 시뮬레이션 하는 시간

$$\sum_{\forall i}\{T \times (st_i + sync)\} + T_{trans} \times st_{trans} \tag{1}$$

다음 장에서는 인터럽트가 존재함에도 매 사이클 다른 시뮬레이터들과 시간 동기를 하지 않으면서 정확도를 유지할 수 있는, 가상 동기화 기법에 대해 자세히 설명하겠다.

4. 가상 동기화 기법

가상 동기화 기법의 핵심아이디어는 시뮬레이터들의 클록들끼리 시간 동기를 하지 않는 것이다. 대신, 각 시뮬레이터는 현재 발생한 이벤트가 가장 최근 이벤트로부터 얼마만큼의 시간이 지난 후에 발생된 것인지에 대한 상대적인 시간차이만을 저장하고 있다가 이 정보를 통합시뮬레이션 커널(혹은 통합시뮬레이션 엔진)에 전달한다. 통합시뮬레이션 커널은 각 시뮬레이터들로부터 받아들인 이벤트들의 상대적인 시간을 전역 클록으로 환산함으로써 전역클록을 유지한다. 이와 같은방식을 통해 각 시뮬레이터는 전역클록과 '가상적'으로 '동기화'된다. 각 시뮬레이터는, 수행되는 태스크가 시작하거나 종료될 때 혹은 데이타 동기가 필요할 때(서로 다른 태스크와 데이타를 주고 받을 때)만 통합시뮬레이션 커널과 통신을 하므로, 모든 시뮬레이터들끼리 매 사이클마다 시간 동기를 하는 전통적인 시간 동기화 기법에 비해서 성능이 증가한다.

4.1 트레이스 기반 가상동기화 기법

트레이스 기반 가상 동기화 기법에서는 시뮬레이터에서 발생하는 이벤트들을 트레이스 형태로 남겨서 보관하고 있다가 시뮬레이터가 통합 시뮬레이션 커널과 통신할 때 함께 넘겨준다. 트레이스는 메모리 트레이스와 비메모리 트레이스로 구분된다. 모든 메모리 트레이스는 [주소, 접근종류, 크기, 상대적인 시간]의 포맷으로 남겨지며, 태스크의 시작과 종료 등이 비메모리 트레이스에 해당된다. 이와 같이 시뮬레이터의 이벤트 정보들을 트레이스 형태로 모았다가 한꺼번에 통합시뮬레이션과 동기를 하면, 시간 동기를 최소화하면서도 시스템의 공유자원에 대한 경합을 반영할 수 있다. 그림 2는 트레이스 기반 가상 동기화 기법을 사용하는 통합시뮬레이션 환경을 보여주고 있다. 크게 두 부분으로 프레임워크를 구분할 수 있는데, 각 태스크별로 트레이스를 생성해내는 부분이 그림의 상단에 해당하며 각 태스크들의 트레이스를 순차적으로 정렬함으로써 전역클록을 진행시키는 부분이 그림의 하단에 해당한다. 이 때, 운영체제 및 통신 구조를 모델링하면서 정렬한다.

그림 2의 상단에 보면 소프트웨어 컴포넌트 시뮬레이터와 하드웨어 컴포넌트 시뮬레이터가 각각 통합시뮬레이션 커널인 백플레인에 연결되어 있다. 각 시뮬레이터에는 가상동기화 기법 프로토콜을 구현하고 트레이스를 생성할 수 있는 시뮬레이터 인터페이스가 구현된다. 시뮬레이터 인터페이스를 통하여 시뮬레이션 수행 중에 생성된 출력 데이타와 트레이스들을 백플레인에게 전달하고 입력 데이타들을 백플레인으로부터 전송받는다. 소프트웨어 컴포넌트 시뮬레이터에 타겟 운영체제가 적재되어 멀티태스킹으로 여러 태스크들이 수행될 경우, 가상 동기화 프레임워크에서는 적재된 운영체제의 스케줄러가 태스크 스케줄링을 하지 않고 대신 백플레인에 존재하는 운영체제 스케줄러 모델이 태스크 스케줄링을 한다. 왜냐하면 가상 동기화 기법에서 시뮬레이터들은 전역 클록을 유지하지 않고 오직 트레이스 생성을 위한 용도로만 클록을 사용하기 때문이다. 따라서, 전역클록을 참조할 수 있는 백플레인의 운영체제 스케줄링 모델이 태스크 스케줄링을 담당한다. 그림에서 표시된 OS API는 태스크들의 읽기/쓰기 블록을 지원하기 위한 쓰레드 라이브러리로서, 시뮬레이터 인터페이스로 간주할 수 있고 따라서 트레이스 생성의 대상이 아니다.

그림 2의 하단은 백플레인에서 모델링하고 있는 운영체제 스케줄링과 공유 메모리 및 통신 구조를 보여주고 있다. 백플레인은 시뮬레이터들로부터 트레이스를 전송받아 이를 해당 태스크 대변자(representative)에 일단 저장한다. 저장된 트레이스들은 운영체제 스케줄링과 공유 버스 경합을 모델링을 거쳐서 순차적으로 정렬된다. 만약 트레이스 정렬 중에 어떠한 태스크 대변자의 트레이스가 더 이상 존재하지 않는다면 트레이스 정렬을 일시 중지하고 다시 해당 태스크의 트레이스 생성을 시뮬레이터에게 요청한다. 트레이스를 정렬하는 알고리즘이 그림 3에 기술되어 있다.

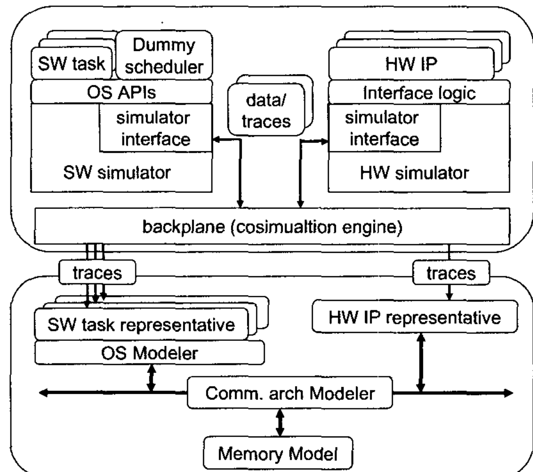


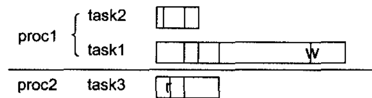
그림 2 트레이스 기반 가상동기화 기법을 이용한 통합 시뮬레이션 환경의 구조

```
1  while (1) {                 // for each trace
2     for (i=previously_paused_comp; i<numComp; i++) {
3        task = OS_Modeler( i )
4        if (task->trace == NULL) {
5           previously_paused_comp = i;
6           return;        // activate trace-obtaining part
7        }
8        if (task->trace->time < min_access_time)
9           min_access_time = task->trace_time;
10          min_task = task;
11       }
12    }
13    CommArch_Modeler( min_task->trace );
14 }
```

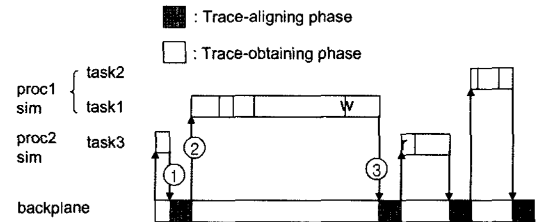
그림 3 트레이스 기반 가상동기화 기법의 트레이스 정렬 알고리즘

그림 3의 알고리즘은 모든 트레이스에 대해서 적용되는데, 알고리즘을 수행하면서 트레이스를 정렬하는 것 자체가 시스템의 전역 클록을 증가시키는 것에 해당한다. 알고리즘은 통합시뮬레이션이 종료하거나 어떤 태스크 대변자가 트레이스를 하나도 가지고 있지 않을 때까지 반복된다(line 6). 우선 가장 일찍 버스에 접근하는 태스크를 찾아낸다. 이를 위해 시스템이 있는 모든 컴포넌트들을 조사하는데(line 2), 이 때 해당 컴포넌트에 운영체제가 적재되어 있는 경우라면 운영체제의 스케줄링을 고려하여 조사한다(line 3). 가장 일찍 버스에 접근하는 태스크와 그 트레이스를 찾아냈다면, 통신 구조 모델러(line 13)는 이 트레이스의 정보를 바탕으로 메모리 접근 지연시간을 반영하고, 또한 현재 트레이스의 버스 접근시작 시각이 이전 트레이스의 버스 접근종료 시각보다 빠르다면(즉, 버스에 대한접근이 중첩된다면), 이러한 경합 지연시간을 반영한다.

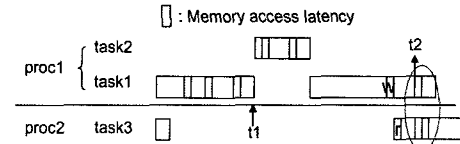
위에서 설명한 트레이스 기반 가상동기화 기법을 이용한 통합시뮬레이션 예제를 그림 4에 도시하였다. Task1과 task2가 proc1 프로세서에 매핑되어 있고, task3이 proc2에 매핑되어 있다고 가정하며, task2의 우선순위가 task1의 우선순위보다 높다고 가정한다. 그림 4의 (a)는 각 시뮬레이터에서 수행되는 태스크들의 수행길이와 트레이스들을 도시하고 있다. 그림에서 세로줄로 표시된 부분들이 각각 메모리 트레이스를 의미하며, 특별히 공유 메모리에 대한 쓰기/읽기는 'w', 'r'로 그림에 표기하였다. 이 예제에서는 task1과 task3이 데이타 의존성이 있어서 task1이 공유메모리에 데이타를 쓰고, 이를 task3이 읽어간다고 가정한다. 그림 4의 (b)는 각 태스크의 트레이스를 생성하는 단계와 이를 정렬하는 단계가 반복적으로 수행되면서 전체 통합시뮬레이션이 진행됨을 보여준다. 우선 proc2 시뮬레이터에서 task3이 수행되다가 task 1으로부터의 데이타가 존재하지 않아 읽



(a) 각 시뮬레이터에서 수행될 태스크의 수행 길이, 트레이스, 그리고 태스크간 데이타 의존성



(b) 트레이스 생성과 트레이스 정렬의 반복



(c) 운영체제 스케줄러 및 통신 구조가 반영되어 트레이스가 정렬된 모습

그림 4 트레이스 생성과 트레이스 정렬

기 블록된다. 가상동기화 프로토콜은 태스크의 수행이 완료되거나 읽기 혹은 쓰기 블록이 발생할 때 백플레인과 동기화를 수행하므로, proc2 시뮬레이터는 그 시점까지 생성한 트레이스를 백플레인에게 전달한다 (①). 트레이스 정렬단계에서는 그림 3의 알고리즘을 수행시키는데, task1과 task2에 대한 트레이스가 확보되지 않았으므로 트레이스를 정렬하지 못하고 task1에 대해 트레이스 생성을 요청한다(②). Task 1은 종료될 때까지 수행되는데, 공유메모리에 대한 쓰기를 포함한 Task1의 트레이스들이 백플레인에 전달되고(③), 백플레인이 트레이스를 정렬하고 데이타 동기를 맞추면서 task3은 읽기 블록이 해제된다. 마찬가지 방식으로 task2에 대한 트레이스도 생성을 한다. 그림 4의 (c)는 통합시뮬레이션이 끝나고 트레이스가 모두 정렬된 것을 도시하고 있다. 먼저 운영체제의 스케줄링을 반영하여 트레이스를 정렬한 것을 그림에서 보여주고 있다. t1시점에서 task2를 깨우는 인터럽트가 도착했다고 가정했을 때 task2가 task1을 선점(preemption)하여 수행되는 것이 모델링되었다. 또한, 각 트레이스마다 해당 메모리의 지연 시간을 반영하여 정렬한 것을 그림에서 확인할 수 있다. 그림 4(c)의 원 안에 표시된 부분은 proc1과 proc2가 동일한 버스에 연결되어 있다고 가정했을 때 task2와 task3의 버스에 대한 경합이 t2시점에 발생한 것을 도

시하고 있다. 경합으로 인해서 task3의 수행시간이 길어짐을 확인할 수 있다.

이와 같이 가상 동기화 기법이 적용되기 위해서는, 가상 동기화 프레임워크에서 태스크간 데이타 동기 시점을 인지할 수 있어야 한다. 일반적으로 태스크간 데이타 동기는 명시적으로 별도의 API를 통해서 발생하므로 이 API에서 시뮬레이터 인터페이스와 연동할 수 있도록 구현한다.

4.2 성능 및 정확도 분석

시간 동기화 비용을 제거한다는 장점외에도 가상 동기화 기법의 주요 장점 중 하나는 시뮬레이터가 휴지상태(idle state)에 있을 때 그에 대한불필요한 시뮬레이션을 제거할 수 있다는 것이다. 가상 동기화 기법에서는 시뮬레이터의 클록이 상대적인 시간 차이를 측정하는 용도로만 사용되기 때문에 클록의 절대적인 시각은 더 이상 의미가 없다. 따라서 이전 동기화 시점과 현재 이벤트가 발생한 시점 사이가 아무런 이벤트가 발생하지 않는 휴지 기간(idle duration)이었다면, 가상동기화 기법에서는 시뮬레이터의 클록을 현재 이벤트 발생 시점에 맞추기 위해 더 이상 무의미하게 시뮬레이터 클록을 진행시키지 않는다.

정의 2는 가상 동기화 기법으로 인한 통합시뮬레이션 성능 개선을 정형적으로 정의하고 있다. 정의 1에서 정의된 항목에서 추가로 시뮬레이터의 이용률(utilization)과 공유 메모리에 대한 통신 트랜잭션의 전체 개수를 정의한다.

정의 2. 가상 동기화 기법을 사용할 때의 통합시뮬레이션 시간

u_i : 시뮬레이터 i의 이용률

T : 전체 시뮬레이션 사이클

T^{sh}_{trans} : 공유 메모리에 대한 전체 통신 트랜잭션(읽기/쓰기)의 개수

st_i : 시뮬레이터 i에서 한 사이클을 진행시키는데 필요한 시뮬레이션 시간

st_{trans} : 통신 트랙잭션(읽기/쓰기) 하나를 시뮬레이션하는 시간

st_{trace} : 트레이스 한 개 생성시에 소요되는 시간

st_{eval} : 트레이스 한 개 정렬하는데 소요되는 시간

$$\sum_{\forall i}\{T \times u_i \times st_i\} + T^{sh}_{trans} \times (sync + st_{trans}) + T_{trans} \times (st_{trace} + st_{eval}) \quad (2)$$

식 (2)에서 볼 수 있듯이 통합시뮬레이션 되어야 하는 전체 사이클의 수가 $\sum_{\forall i}(1-u_i)$ 만큼 감소하며, 시간 동기화는 태스크간 데이타 동기(공유 메모리에 대한 읽기/쓰기)가 이루어질 경우에만 발생한다(T^{sh}_{trans}). 락스텝 동기화 기법을 사용할 때를 기준으로 가상 동기화 기법을 사용할 때의 통합시뮬레이션 시간 감소량을 명확히 하기 위해 식 (1)에서 식 (2)를 빼면 아래와 같은 식 (3)을 얻는다.

$$\sum_{\forall i}\{T \times (1-u_i) \times st_i\} + sync \times (T \times i - T^{sh}_{trans}) + st_{trans} \times (T_{trans} - T^{sh}_{trans}) - T_{trans} \times (st_{trace} + st_{eval}) \quad (3)$$

식 (3)에서 양수 항은 줄어든(개선된) 시간이며 음수 항은 가상동기화 기법에서 새롭게 추가된(오버헤드) 시간이다. 식 (3)을 살펴보면 세 가지 종류의 이득을 확인할 수 있다. (1) 첫 번째 항은 휴지 기간에 대한 시뮬레이션 제거로부터 얻는 이득이고 (2) 두 번째 항은 시간 동기화 시점의 개수가 줄어듦에 따른 이득이며 (3) 세 번째 항은 로컬 메모리 트랜잭션을 별도의 시뮬레이터로 시뮬레이션하지 않아도 되기 때문에 생긴 이득이다. 즉, 락스텝 동기화 기법을 사용할 경우에는 메모리 접근 지연시간 및 버스에 대한경합을 시뮬레이션 하기 위하여 로컬메모리를 포함한 모든 버스 접근을 별도의 하드웨어 시뮬레이터로 시뮬레이션 하였으나, 가상 동기화 기법에서는 이를 트레이스로 남기기만 하고 별도의 하드웨어 시뮬레이터로 매 트랜잭션 시뮬레이션하지 않기 때문에 생기는 이득이다. 반면에 네 번째 항목은 오버헤드를 의미하는데 전체 트레이스의 생성과 정렬에 드는 추가적인 시간을 의미한다. 6장의 실험에서 확인할 수 있듯이 이 시간은, 메모리 접근을 많이 하는 DivX 플레이어 응용의 경우에도 전체 시뮬레이션 시간의 2% 미만이다. 왜냐하면 트레이스는 파일 형태로 저장되어 전달되는 것이 아니라 메모리 버퍼에 저장되고 전송되므로 st_{trace}가 매우 작기 때문이다. 또한 트레이스 정렬은 그림 2에서 확인한 것과 같이 시뮬레이터가 아닌, 호스트 서버에서 수행되는 통합시뮬레이션 커널에서 이루어지기 때문에 st_{eval}도 매우 작다.

가상 동기화 기법을 사용한 통합시뮬레이션의 정확도는 통합시뮬레이션 커널에서 모델링하는 실시간 운영체제 스케줄러 모델과 통신 구조 모델의 정확도에 영향을 받는다. 가상동기화 기법에서 실시간 운영체제를 시뮬레이터에 직접 적재하지 않고 모델링을 하기 때문에 얻는 성능 개선 폭과 정확도 손실 폭에 대해서는 [14]에 자세히 기술되어 있다. [14]에 보면 프로세서에서 캐쉬가 사용되지 않은 경우에는 0.1%의 오차를 보여주었고 캐쉬가 사용된 경우 최대 7%의 오차를 보여주었다. 이와 마찬가지로, 사이클 정확도를 가지는 별도의 시뮬레이터로 메모리 트랜잭션들을 시뮬레이션 하지 않고 통합시뮬레

이션 커널의 통신 구조 모델에서 모델링함에 따라 정확도가 손실될 수 있는데, 동적인 특성을 많이 반영해야 하고 소프트웨어 컴포넌트에서 수행되어야 하는 운영체제에 비해서 통신 구조는 비교적 정적이고 모델링이 용이하므로 오차가 적다. 6.3절의 실험 결과를 보면 전체 오차가 5% 미만임을 확인할 수 있다.

4.3 활용 범위와 구현의 확장성

앞 절에서 살펴본 바와 같이 트레이스 기반의 가상동기화 기법을 사용하면 통합시뮬레이션 의 성능을 현저하게 개선시킬 수 있기 때문에 검증 목적의 통합시뮬레이션뿐만 아니라 설계 공간탐색 목적으로도 활용될 수 있다. 통합시뮬레이션 커널에서 수행되는 운영체제 모델과 통신구조 모델의 파라미터를 바꿔가면서 시스템의 성능이 어떻게 영향을 받는지 확인할 수 있다.

가상 동기화 기법을 적용하기 위해서는, 트레이스 기반 가상동기화 기법에 필수적인 트레이스 생성 기능 및 가상 동기화 프로토콜이 시뮬레이터에서 지원되어야 한다. 일단, 시뮬레이터에 가상 동기화 프로토콜만 구현되면 시뮬레이터가 가정하는 플랫폼(호스트 서버의 운영체제)에 독립적이므로 확장성이 우수하다.

5. 확장된 가상 동기화 기법

5.1 SystemC 시뮬레이터의 지원

이전의 가상동기화 기법에서는 하드웨어 시뮬레이터로 RTL 시뮬레이터인 ModelSim을 사용하였다. RTL 시뮬레이터의 경우 수 GHz의 호스트 서버에서 시뮬레이션을 수행할 때 초당 약 1K 사이클의 매우 낮은 성능을 보이기 때문에, 전체 통합시뮬레이션 성능의 병목이 되었다. 즉, 식 (2)에서 전체 통합시뮬레이션 시간의 대부분은 첫 번째 항목에서 기여하는데 RTL 시뮬레이터의 st_i가 매우 크기 때문에 병목이 발생한다. 이것이 RTL 시뮬레이터를 가장 많이 쓰이는 TLM 시뮬레이터인 SysetmC 시뮬레이터로 대체한 주요 동기이다. 4.3절에서 밝힌 것과 같이 가상동기화 기법에서 요구되는 구체적인 기능을 아래에 열거하였다.

1) 백플레인(통합시뮬레이션 커널)과 통신을 하기 위해 소켓을 연결할 수 있어야 한다.
2) [주소, 접근종류, 크기, 시간]의 포맷으로 구성된 트레이스를 생성할 수 있어야 한다.
3) 백플레인으로부터 입력 데이타를 전달받고, 출력 데이타 및 생성된 트레이스를 백플레인에게 전달한다.
4) 시간 차이를 계산하기 위해서 시뮬레이터의 로컬 클록을 참조할 수 있어야 한다.

일반적으로 소프트웨어 컴포넌트 시뮬레이터의 경우에서는, 사용자가 memory mapped I/O로 통신하는 주변장치를 시뮬레이터에서 모델링할 수 있도록 하기 위해 사용자가 직접 구현하는 callback 함수나 메모리 모델이 주어진다. 프로세서와 같은 소프트웨어 컴포넌트에서 매번 메모리를 접근할 때 이러한 함수가 호출되므로, 위에서 열거한 기능들을 이 함수에 구현하면 된다. 하드웨어 컴포넌트 시뮬레이터의 경우, HDL로 기술한 RTL 시뮬레이터에서는 C 언어로 기술할 수 있는 FLI 인터페이스가 제공하므로 이 인터페이스를 사용하여 위의 기능을 구현하면 된다.

그림 5는 일반적인 SystemC 통합시뮬레이션 프레임워크의 구조와 SystemC를 하드웨어 시뮬레이터로 사용할 때의 가상 동기화 기법 프레임워크의 구조를 도시하고 있다. 그림 5(a)가 명령어집합 시뮬레이터를 포함하는 일반적인 SystemC 통합시뮬레이션 환경의 구조인데, SystemC 커널이 직접 시뮬레이터(혹은 시뮬레이션 모델)들을스케줄하고 수행시키는 통합시뮬레이션 커널 역할을 수행한다. 그림 5(b)는 본 논문에서 제안하는 SystemC 시뮬레이터를 포함한 가상 동기화 기법프레임워크이다. 이 구조에서 SystemC 시뮬레이터는, 하나의 통합시뮬레이션 환경이 아닌 하드웨어 시뮬레이션만 담당하는 컴포넌트 시뮬레이터로 활용된다. 제안하는 구조의 핵심적인 내용은, 가상 동기화 기법에 필요한 기능들을 수행하는 시뮬레이터 인터페이스를 SystemC커널과 버스라이브러리 사이에 구현한다는 점이다. 이러한 설계의 가장 큰 이점은 SystemC 커널과 HW IP를 모두 수정하지 않으면서 가상 동기화 기법을 적용할 수 있다는 점이다. 이렇게 설계할 때 필요한 가정은, HW IP가 버스 인터페이스를 통하여 시스템의 다른 컴포넌트와 통신한다는 점이다.

본 논문에서 사용된 SystemC 시뮬레이션 환경은 Dynalith[13] SystemC 환경으로서 OSCI에서 제공하는 SysmteC 표준 라이브러리 외에 AMBA 버스 모델이 구현되어 있다. AMBA는 ARM 프로세서를 위한 버스 프로토콜로서 일반적인 버스의 특성을 지니고 있기 때문에, 본 논문에서 기술하고 있는 가상동기화 기법을 위한 인터페이스 구현은 일반적인 다른 버스에서도 구현될 수 있다.

가상 동기화 기법을 적용하기 위한 시뮬레이터 인터페이스는 하나의 클래스로 구현되어 있고 SystemC 커널 위에서 하나의 개체로 존재한다. 그림 6은 기존에 정의된 AMBA 마스터 인터페이스가 가상 동기화를 위한 시뮬레이터 인터페이스로 어떻게 재정의되는가를 보여주고 있다.

그림 6에서 myCosimInterf는 시뮬레이터 인터페이스 개체로서, 하드웨어 IP가 버스 인터페이스를 통하여 데이타를 읽거나 쓰려고 접근하면 시뮬레이터 인터페이스의 읽기나 쓰기가 호출된다. 그림 5(a)에서와 달리 제안

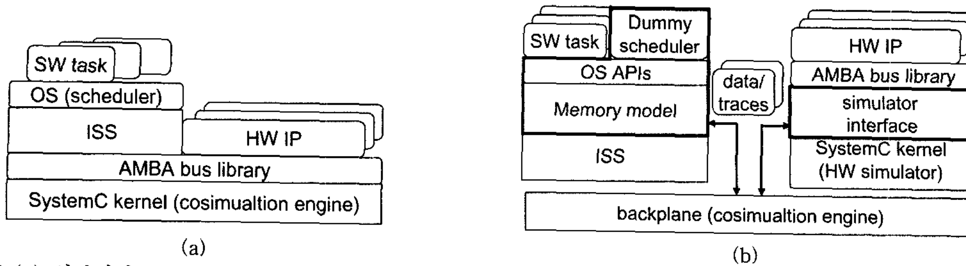


그림 5 (a) 일반적인 SystemC 통합시뮬레이 환경, (b) SystemC 시뮬레이터가 포함된 가상 동기화 기법을 이용한 통합시뮬레이션 환경

```
void ahbmst_read(const unsigned addr, unsigned& data) {
    myCosimInterf->readData(addr, data, 1);
}
void ahbmst_write(const unsigned addr, const unsigned data) {
    myCosimInterf->writeData(addr, data, 1);
}
void cosimInterf::readData(const unsigned addr, unsigned
    data[], const int len) {
    //1. Synchronize with the backplane
    //2. Read data from the memory model
    //3. Generate trace
}
void cosimIntef::writeData(const unsigned addr, const unsigned
    data[], const int len) {
    //1. Synchronize with the backplane
    //2. Write data to the memory model
    //3. Generate trace
}
virtual void ahbslv_read(const unsigned addr, unsigned& val)=0;
virtual void ahbslv_write(const unsigned addr, const unsigned
    val)=0;
```

그림 6 SystemC 시뮬레이터를 위한 가상 동기화 기법 인터페이스 정의

하는 그림 5(b)의 구조에서는 버스 및 메모리 에 대한 SystemC 모델이 더 이상 사용되지 않기 때문에 시뮬레이터 인터페이스 객체가 공유 메모리 모델을 포함한다. 만약 주어진 주소에 해당하는 데이타가 현재 시뮬레이터 인터페이스 내의 공유 메모리 모델에 존재하지 않는다면 백플레인과 통신을 하여 데이타 동기를 맞춘다. 이 시점에서 그 동안 작성되었던 트레이스도 함께 전송하므로 시간 동기가 데이타 동기 시점마다 이루어진다. 이때 트레이스의 시간값을 계산하기 위하여 sc_time()을 참조한다.

한편, 그림 6의 하단에서 볼 수 있듯이, 슬레이브 인터페이스는 pure virtual 함수로 정의되어 IP에서 각각 구현된다. 앞서 설명했듯이 버스에 대한 시뮬레이션 모델을 시뮬레이터 인터페이스가 가로채 대신 모델링해야 하므로, 만약 버스에 연결된 어떤 마스터가 어떤 IP를 슬레이브 인터페이스로 접근한다면, 시뮬레이터 인터페이스는 이 IP의 슬레이브 인터페이스를 호출해주어야 한다. 따라서 시뮬레이터 인터페이스 개체는 슬레이브 인터페이스를 가지는 각 IP 개체에 대한 포인터를 초기화 과정에서 획득해야만 하며 이 때문에 하나의 버스에 연결되는 슬레이브 IP의 개수에 대한 최대값을 제한해야 한다. 그러나 이러한 제약은 실제 버스 설계에서도 존재하는 것이므로 본 논문에서 제안하는 인터페이스 설계의 단점이라 할 수 없다.

결과적으로 제안하는 가상동기화 기법을 위한 SystemC 시뮬레이터 인터페이스는 SystemC의 스케줄러나 커널 및 하드웨어 IP를 전혀 수정하지 않아도 되고 단지버스 트랜잭션 API가 재정의되기만 하면 된다.

5.2 가상 동기화 기법의 병렬적 수행

분산된 환경에서 병렬적으로 시뮬레이터들을 수행시킬 때 시간 동기를 최소로 하지 않는 한 성능 개선보다는 성능 감소를 초래하기가 쉽다. 가상 동기화 기법을 사용하면 시간 동기화 비용이 무시할만한 수준으로 제거되기 때문에 분산 서버나 혹은 SMP서버에서 병렬적으로 시뮬레이터들을 수행시켜 성능을 추가로 개선하는 것이가능하다. 본 절에서는 가상 동기화 기법을 이용한 병렬 통합 시뮬레이션 기법을 설명한다.

병렬적인 시뮬레이션을 통하여 통합 시뮬레이션의 성능 개선을 이루기 위해서는, 시간 동기화에 대한 문제가 해결되어야 할 뿐만 아니라 시뮬레이터들간에 데이타 동기의 필요성 즉, 데이타 의존성이 낮고 병렬성 자체가 높아야 한다. 일반적으로 MPSoC와 같은 구조에서 태스크를 매핑할 때는 상호 데이타 의존성이 없는 독립적인 태스크들을 서로 다른 프로세서로 매핑하고, 만약 태스크들간에 상호 데이타 의존성이 존재한다면 이들을 파이프라인 형태로 수행시킴으로써 프로세서의 이용률(utilization)을 높인다. 그림 7(a)에 보면, 프로세서1의 태스크가 데이타 생성을 하고 프로세서 2의 태스크가 이를 소비하는 과정이 파이프라인 형태로 병렬 진행되는 것을 도시하고 있다. 그러나 이와 같은 파이프라인

형태의 병렬 수행을 병렬적으로 시뮬레이션 하고자 할 경우 그림 7(b)에서 볼 수 있는 바와 같이 데이타 의존성에 의하여 시뮬레이터들이 블록되어 병렬성이 존재하지 않는다(병렬성이 존재하지 않아 시뮬레이터가 수행되지 못하는 부분을 O표로 표시). 그림 7(b)는 기존의 가상 동기화 기법을 적용하였을 때의 시뮬레이터들의 수행순서를 도시하고 있는데, 통합시뮬레이션 커널인 백플레인이 시뮬레이터들에게 수행 명령을 내리고 수행 응답을 기다리는 과정의 반복이 함께 도시되어 있다. 기존의 가상 동기화 기법에서는 데이타의 쓰기/읽기 블록이 발생하거나 태스크의 종료 시에만 시간 동기가 이루어지므로, 시뮬레이터 1의 태스크는 쓰기 블록이 발생할 때(①)까지 수행되며 시뮬레이터 2의 태스크가 수행되고 데이타가 소비되어 버퍼가 비어지고 나면 쓰기 블록이 해제되어 비로소 수행이 가능하다. 그러나 버퍼가 비어진 것을 시뮬레이터1이 알게 되는 시점은 시뮬레이터 2에서 두 번째 읽기 블록이 발생하여 백플레인과 동기를 하고 난 후(②)이므로 시뮬레이터들이 병렬적으로 수행될 수가 없다. 그림 7(b)에서 발생하는 시뮬레이터의 직렬적인 수행 문제는, 서로 다른 프로세서에 매핑된 태스크 간에 데이타 의존성이 있을 때 그 데이타 버퍼에 쌓인 현재 데이타의 크기가 얼마인지 갱신이 바로 되지 않기 때문에 발생한다. 그러나, 이를 해결하기 위해 시간 동기를 매 사이클 수행하거나 너무 자주 수행하게 되면 병렬 시뮬레이션으로 인한 성능 증가를 기대하기 어려워진다.

제안하는 가상 동기화 기법을 이용한 병렬 시뮬레이션은 데이타 의존성이 있는 버퍼의 크기를 가상적으로 늘려줌으로써 병렬성을 높인다(그림 7(c)). 즉, 버퍼에 데이타를 생성하는 프로세서가 파이프라인의 m단에 해당하고 버퍼의 데이타를 소비하는 프로세서가 파이프라인의 n단에 해당한다면, 실제 버퍼의 크기가 k일 때 버퍼의 크기를 k+(n-m)으로 가상적으로 늘려서 설정한다. 이렇게 설정하면, 늘어난 가상 버퍼로 인해 쓰기 블록이 발생하지 않기 때문에 버퍼의 데이타를 소비하는 시뮬레이터와 버퍼의 데이타를 생성하는 시뮬레이터가 병렬적으로 수행이 된다.

그림 7(c)에서 시뮬레이터1과 시뮬레이터 2 사이의

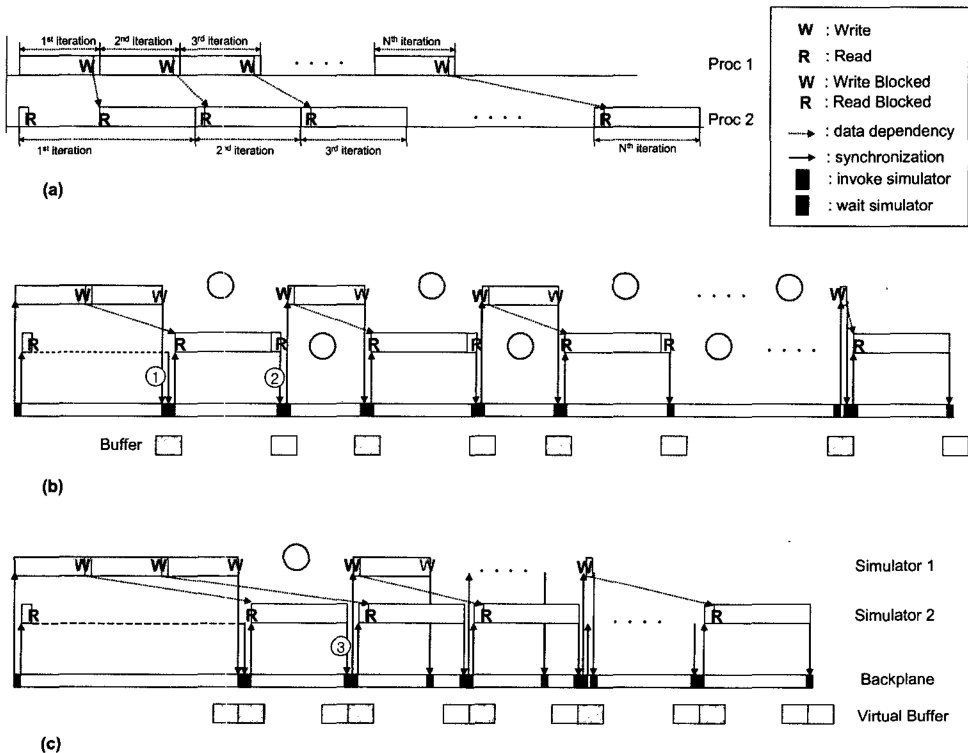


그림 7 (a) 2개의 프로세서를 사용한 파이프라인 형식의 병렬 수행
(b) 가상 버퍼를 사용하지 않았을 때의 가상 동기화 기법에 의한 시뮬레이터 수행 순서
(c) 가상 버퍼를 이용한 병렬성 획득 및 시뮬레이터 수행 순서

버퍼의 원래 크기가 1이고 두 프로세서 시뮬레이터가 연이은 파이프라인 단계이므로, 가상 버퍼를 1+1=2의 크기로 설정한다. 그림에서 보듯이, 초기 단계 이후 매 단계에서 시뮬레이터1과 시뮬레이터 2가 서로 블록됨이 없이 병렬적으로 수행이 된다. 이 때 추가로 요구되는 가상 동기화 프로토콜이 존재한다. 파이프라인의 첫째 단에 해당하는 프로세서를 제외한 나머지 프로세서 시뮬레이터들은 파이프라인 단계의 반복(iteration)이 끝날 때마다 시간 동기를 수행하여야 한다(③). 그림 7(c)에서 시뮬레이터 2가 읽기 블록을 당하지 않았음에도 매 반복(iteration)이 끝날 때마다 동기를 수행하고 있음을 확인할 수 있다.

이러한 기법은 가상 동기화 기법을 이용하기 때문에 가능하다. 가상 동기화 기법에서는 트레이스 생성과 트레이스 정렬이 구분되므로 트레이스 생성할 때는 비순차적(out-of-order)으로 수행되어도 무관하기 때문이다. 트레이스 정렬 단계에서 가상 버퍼에 대한 트레이스 정렬을 할 때는 가상 버퍼가 연관되어 있는 원래의 버퍼에 대한 접근으로 주소를 환산하여 정렬한다. 이렇게 하면, 가상 버퍼 없이 순차적으로 통합시뮬레이션 했을 때와 비교했을 때 동일한 정확도를 유지하게 된다.

6. 실 험

6.1 실험 환경 및 설정

현재 가상동기화 기법은 PeaCE[15]라는 통합설계 환경에 구현되어 있다. 실험에 사용된 예제는 H.263 비디오 디코더로서, 이 중에서 IDCT(Inverse Discrete Cosine Transform)는 하드웨어로 매핑하였고, IDCT를 제외한 부분은 3 개의 ARM926ej-s 코어에 매핑하였다. 가정하는 플랫폼과 구체적인 매핑 결과를 그림 8과 표 1의 첫 번째, 두번째 열에 정리하였다. 그림 8에서 보듯이 각 프로세서들은 로컬 버스와 로컬 메모리를 가지며, 프로세서간 통신을 위한 공유 메모리가 존재하는데 이는 글로벌 버스를 통해서 접근된다. 글로벌 버스에는 IDCT 하드웨어 IP들이 연결되어 있고, 또한 벡터 인터럽트 컨트롤러(Vectored Interrupt Controller)가 있어서IDCT의 입력을 생성하는 프로세서와 IDCT의 출력을 소비하는 프로세서에 각각 연결되어, IDCT의 입력을 소비한 직 후, IDCT의 출력을 생성한 직후 각 프로세서들에게 인터럽트를 생성한다.

표 1에서 H263Reader라는 블록은 avi file을 읽어서 한 프레임씩 디코더에게 전달하는 블록으로서 프레임 단위로 수행된다. 나머지 블록들은 매크로블록 단위로 수행되는데, 본 실험에서는 QCIF포맷의 입력을 디코딩하므로 한 프레임 당 99개의 매크로블록이 수행된다. 각 매크로 블록에 대해서 ARM(1)에서는 VLD, DeQuant, InvZigzag이 수행되고, IDCT가 하드웨어로 수행되며, ARM(2)에서는 MotionCompensation이 수행된다(참고로, DisplayFrame는 99개의 매크로블록 디코딩이 완료된 후 결과를 출력하는, 프레임 단위로 수행되는 블록이

표 1 H.263 디코더 예제의 분할 및 매핑 정보, 각 시뮬레이션 시간의 비중

Processor Element Name	Algorithm Block Name	Simulation Time Portion (%)
ARM(0)	H263Reader	0.4
ARM(1)	VLD	50.4
	DeQuantizer	
	InvZigZag, etc.	
IDCT(Y)	IDCT_Y	1.2
IDCT(U)	IDCT_U	
IDCT(V)	IDCT_V	
ARM(2)	DisplayFrame	24.0
	MotionCompensation, etc.	24.0

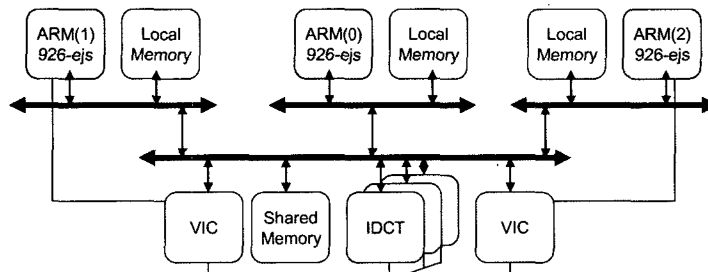


그림 8 실험에서 설정한 플랫폼의 도시도

다). 이와 같이 프로세서2개와 1개의 하드웨어 IP가 파이프라인 형태로 병렬적으로 수행되어 99개의 매크로 블록을 디코딩하는 예제이다. 이와 같이 설정하여 H.263 디코더에서 QCIF 포맷3 프레임(I, P, P)을 디코딩하였다.

제안하는 통합시뮬레이션 환경에서 SystemC시뮬레이션은 Dynalith SystemC 환경[13]으로 수행하며 소프트웨어 코드는 ARM사의 ARMulator로 수행한다. 6.2절과 6.3절의 모든 시뮬레이터는 Intel Dual-Xeon 1.8 GHz의 동일한 서버에서 수행하였으며, 타겟 코드는 arm-elf-gcc 3.4.5 (O3 option)으로 빌드하였다.

6.2 SystemC 시뮬레이터에 대한 가상동기화 기법 적용

SystemC 시뮬레이터에 가상동기화 기법을 적용했을 때 기존 RTL 하드웨어 시뮬레이터로 가상 동기화 기법을 구성했던 것에 비해 통합시뮬레이션의 성능이 얼마나 개선되었는지를 표 2에서 확인할 수 있다. IDCT가 RTL로 기술되어 ModelSim에서 수행되었을 때의 전체 시뮬레이션 시간은 127초이고, 표에서 별도로 보여주고 있지는 않지만, 이 시간 중에서 ModelSim에서 수행된 시간은 109초로서 86%에 해당한다. 즉, RTL 하드웨어 시뮬레이터를 사용할 경우 하드웨어 시뮬레이터에서 걸리는 시간이 전체 통합시뮬레이션의 병목현상임을 확인할 수 있다. 하드웨어 시뮬레이터를 본 논문에서 제안한 것과 같이 SystemC 시뮬레이터로 구성하여 가상 동기화 기법을 이용한 통합시뮬레이션을 수행하면, 하드웨어 시뮬레이션에서 걸리는 시간은 불과 0.18초가 되어 전체적으로 약 7.5배의 성능 개선을 얻을 수 있다. 하드웨어로 구성하는(매핑하는) 블록이 많으면 많을수록 SystemC 시뮬레이터를 사용함으로써 얻는 성능 개선도는 증가할 것이다. 이 실험에서 RTL 시뮬레이터를 사용했을 때와 SystemC 시뮬레이터를 사용했을 때의 소프트웨어 코드는 동일하나 RTL IDCT와 SystemC IDCT의 모델링 오차에 의해 전체적으로 2%의 오차가 발생하였다.

6.1절에서 설명하였듯이 파이프라인 형태의 병렬 디코딩에 참여하고 있는 프로세서는 2개로서 각각 IDCT 이전과 IDCT 이후(MotionCompensation)의 매크로블록 디코딩을 담당하는데, 이 두 프로세서 시뮬레이터에서 걸리는 시뮬레이션 시간이 각각 7.82, 7.47초로서 전체 시뮬레이션 시간의 45.5%, 43.4%를 차지한다. 즉 SystemC 하드웨어 시뮬레이터를 사용함으로써, 이제는 프로세서 시뮬레이터가 새로운 병목현상이 되었음을 알 수 있다.

6.3 통합시뮬레이션 성능 및 정확도 비교

본 논문에서 제안하는 확장된 가상 동기화 기법의 성능과 정확도를 널리 알려진 상용 TLM 통합시뮬레이션 환경인 MaxSim[1]과 비교한 결과를 표 3에 정리하였다. 파일 읽기를 수행하는 H263Reader 블록이 매핑된 태스크만 armcc로 컴파일하였는데, 이는 MaxSim에서 파일 읽기 등의 I/O 모델링은 armcc로 컴파일했을 때만 지원되기 때문이다. 그러나 이 태스크가 전체시뮬레이션 사이클에서 차지하는 비중은 1.0%(55,148 cycle)밖에 안 되고, arm-elf-gcc 3.4.5 버전의 경우 O3 옵션으로 컴파일했을 때 본 예제의 경우 armcc와 비교하여 -5.5% 밖에 차이가 나지 않으므로, H263Reader 블록에 의한 오차는 무시할 수 있다.

가상동기화 기법을 이용한 통합시뮬레이션 환경은 가상 버퍼를 사용한 병렬 통합시뮬레이션 기법을 적용한 것과 아닌 것으로 구분하여 수행하였다. 표 3에서 확인할 수 있듯이, 병렬 통합시뮬레이션을 하지 않더라도 SystemC 시뮬레이터에 가상 동기화 기법을 적용하여 통합시뮬레이션 했을 때(VS_serial+SystemC) 약300 Kcycle/sec로서 MaxSim에 비해서 8배 이상의 성능 증가를 얻을 수 있다. 성능 증가의 요인은 4.2절의 식 (3)에서 정리하였듯이, 시간 동기화 비용이 현격히 줄어들고 휴지기간을 시뮬레이션 하지 않으며 로컬 메모리에 대해서 별도의 시뮬레이터로 수행하지 않아도 되기 때문이다. 반면에 정확도는 5%의 적은 오차를 보이는데, 이 오차는 4.2절에서 설명한 가상 동기화 기법에서의 모델링 오차와 서로 다른 통합시뮬레이션 환경에 따른 타겟 코드의 미세한 차이에 의한 오차에서 기인한다.

가상 버퍼를 이용한 병렬통합시뮬레이션 기법을 추가로 적용했을 때는(VS_parallel+SystemC) 약 400 Kcycle/sec로서 MaxSim에 비해서 11배 이상의 성능 증가를

표 2 SystemC 시뮬레이터를 활용한 가상동기화 기법에 의한 성능 증가

	Simulated cycles	Simulation time	cycles per second	Error	Speedup
VS_with_ModelSim	5,202,621	126.65 sec	41,079 cycle/sec	0%	1.00
VS_with_SystemC	5,329,886	17.20 sec	309,877 cycle/sec	2%	7.54

표 3 제안하는 통합시뮬레이션 기법들과 MaxSim™의 통합시뮬레이션 성능, 정확도 비교

	Simulated cycles	Simulation time	Performance	Error	Speedup
MaxSim 6.0	5,053,686 cycles	142.57 sec	35,447 cycle/sec	0%	1.00
VS_serial+SystemC	5,329,886 cycles	17.20 sec	309,877 cycle/sec	5%	8.74
VS_parallel+SystemC	5,261980 cycles	13.24 sec	397,430 cycle/sec	4%	11.21

얻었다. 가상 동기화 기법에서 병렬 통합시뮬레이션 기법을 사용하지 않은 것에 비해서 28%의 성능 개선이 이루어졌다. 병렬 통합시뮬레이션에 의한 추가 성능 개선은 대상이 되는 예제의 병렬성에 따라 달라지는데, 본 실험에서 설정한 비디오 디코더의 병렬성은 약 32%정도이다. 표 1의 셋째 열은 본 실험에서 설정한 비디오 디코더를 하나의 서버에서 순차적으로 통합시뮬레이션 했을 때 각 시뮬레이터에서 걸린 시간의 비중을 보여주고 있다. 6.1절에서 설명했듯이 파이프라인 형태의 병렬 디코딩에 참여하고 있는 주체는 2개의 ARM 프로세서와 IDCT인데, H263Reader뿐 아니라 DisplayFrame이라는 블록도 프레임당 한번씩 수행되는 블록이므로 병렬화될 수가 없다. 따라서 각 시뮬레이터에서 걸린 수행 비중(%)을 기준으로 계산한 전체 성능 증가의 상한선은, 100/{max(50.4, 48.0)+1.2+0.4}이 아니라 100/{max(50.4, 24.0)+1.2+0.4+24.0} = 1.32이 된다.

SystemC를 사용하면 프로세서 시뮬레이터의 성능이 전체 통합시뮬레이션 성능의 병목이 되기 때문에 MPSoC 구조처럼 프로세서의 개수가 많을수록 병목이 심해진다(그림 1). 따라서 병렬 수행에 참여하는 프로세서 시뮬레이터의 개수가 많을수록 가상 동기화 기법을 이용한 병렬 통합시뮬레이션에 의한 이득은 그만큼 커진다.

7. 결 론

이 논문에서는 여러 개의 소프트웨어 혹은 하드웨어 컴포넌트가 존재하는 MPSoC(Multi-Processor System-on-chip) 아키텍처를 빠르면서도 정확하게 통합시뮬레이션 하는 내용을 다룬다. 기존에 제안된 트레이스 기반의 가상 동기화 기법을 기반으로 하여 SystemC 통합시뮬레이션을 병렬적으로 수행할 수 있는 기법을 제안하였다.

먼저 SystemC 시뮬레이터에 가상 동기화 기법 적용하여 기존에 병목이 되었던 RTL 시뮬레이터를 대체하고 통합시뮬레이션의 성능을 개선시켰는데, 제안된 방법은 SystemC 커널 코드나 하드웨어 IP 코드에 별도의 수정이 필요 없다는 장점이 있다.

SystemC 시뮬레이터로 대체하여도 MPSoC 구조와 같이 프로세서 시뮬레이터의 개수가 증가하면 통합시뮬레이션 성능의 병목이 되는데, 이를 해결하기 위해서는 가상 동기화 기법을 이용한 병렬 시뮬레이션 기법을 제안하였다.

H.263 비디오 디코더 예제를 대상으로 널리 알려져 있는 상용 SystemC 통합시뮬레이션 환경인 MaxSim과 성능과 정확도를 비교했을 때, 제안하는 기법은 11배 이상의 성능 증가를 이룬 반면 5%의 오차만을 보였다.

참 고 문 헌

[1] MaxSim, http://www.arm.com/products/DevTools/MaxSim.html

[2] ConvergenSC, http://www.coware.com/products/convergensc.php

[3] Luca Benini, Davide Bertozzi, Davide Bruni, Nicola Drago, Franco Fummi, Massimo Poncino, "SystemC Cosimulation and Emulation of Multiprocessor SoC Designs," *Computer,* v.36 n.4, pp.53-59, April 2003.

[4] Luca Formaggio, Franco Fummi, Graziano Pravadelli, "A timing-accurate HW/SW co-simulation of an ISS with SystemC," *In Proc. Hardware/Software Codesign and System Synthesis*, September 2004.

[5] Dohyung Kim, Youngmin Yi, Soonhoi Ha, "Trace-driven HW/SW Cosimulation Using Virtual Synchronization Technique," *In Proc. Design Automation Conf.*, June 2005.

[6] SeamlessCVE, http://www.mentor.com/products/fv/hwsw_coverification/seamless/

[7] Sungjoo Yoo, Kiyoung Choi, "Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model," *In Proc. 6th Int'l Workshop on Hardware/Software Co-Design*, March 1998.

[8] Wonyong Sung, Soonhoi Ha, "Efficient and Flexible Cosimulation Environment for DSP Applications," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Special Issue on VLSI Design and CAD algorithms*, Vol.E81-A, No. 12, pp. 2605-2611, December. 1998.

[9] 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*, March 2002.

[10] AndreasGerstlauer, Haobo Yu and Daniel Gajski, "RTOS Modeling for System-Level Design," *In Proc. Design Automation and Test in Europe*, March 2003.

[11] 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 Conf.*, January 2004.

[12] Zhengting He, Aloysius Mok, Cheng Peng, "Timed RTOS modeling for Embedded System Design," *In Proc. Real Time and Embedded Technology and Application Symposium*, March 2005.

[13] Dynalith, http://www.dynalith.com

[14] Youngmin Yi, Dohyung Kim, 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, October 2003.

[15] Soonhoi Ha, Choonseung Lee, Youngmin Yi, Seongnam Kwon, Young-Pyo Joo, "Hardware-software Codesign of Multimedia Embedded Systems : the PeaCE Approach," *In Proc. Embedded and Real-Time Computing Systems and Applications*, pp.207-214, August 2006.

이 영 민

2000년 2월 서울대학교 컴퓨터공학과 학사. 2000년 3월~현재 서울대학교 전기컴퓨터공학부 석박사 통합과정. 관심분야는 하드웨어-소프트웨어 통합설계, 하드웨어-소프트웨어 통합시뮬레이션, MPSoC 내장형 소프트웨어 설계

권 성 남

2002년 2월 서울대학교 컴퓨터공학과 학사. 2002년 3월~현재 서울대학교 전기컴퓨터공학부 석박사 통합과정. 관심분야는 하드웨어-소프트웨어 통합설계, 성능 예측, MPSoC 내장형 소프트웨어 설계

하 순 회

정보과학회논문지 : 시스템 및 이론
제 33 권 제 9 호 참조