공유 메모리 기반 멀티 코어 시뮬레이터 병렬화

Parallelization of Shared Memory based Multicore Simulator

박현식 · 한환수
한국과학기술원 전산학과
hyunsik.park; hyunik.na; hwansoo.han@arcs.kaist.ac.kr

요 약

Multiple core designs have become commonplace in the processor market, and are hence a major focus in modern computer architecture research. Thus, for both product development and research, multiple core processor simulation environments are necessary. A well-known positive feedback property of computer design is that we use today's computers to design tomorrow's. Thus, with the emergence of chip multiprocessors, it is natural to re-examine simulation environments written to exploit parallelism.

In this paper we present a programming methodology for directly converting existing uniprocessor simulators into parallelized multiple-core simulators. Our method not only takes significantly less development effort compared to some prior used programming techniques, but also possesses advantages by retaining a modular and comprehensible programming structure. We demonstrate our case by applying this method to existing simulator (the SimpleScalar tool set) and developing simple kernel thread library package for simulated program. And also we demonstrate the upper limit of scalability when using relaxation of synchronization method. Our SimpleScalar-based framework achieves a parallel speedup of 1.44X on a dual-CPU quad-core (4-way) Xeon server. And also present better scalability when using relaxation of synchronization method.

1 서론

1.1 연구의 필요성

마이크로프로세서의 사용 주파수를 계속 올리는 방식으로는 성능 향상에 비해 전력 소모가 터무니없이 늘어난다. 또한 싱글 코어 칩으로는 클럭 속도 향상에 따른 발열량 문제가 있어 원하는 성능을 만족시키기 어려웠다. 전력 소모, 발열량을 초과하지 않으면서 높은 성능 요건을 만족시키기 위해서 디바이스의 주파수를 계속 올리는 대신 프로세서를 멀티 코어화 하는 방식으로 전환이 일어나고 있다.

집적 기술의 발달로 인해 하나의 칩 안에 필요한 여러 코어를 넣을 수 있게 되면서 더 적은 개발 비용으로 필요한 성능을 만족시킬 수 있게 되었을 뿐 아니라 기존보다 보드 크기와 전력 소모를 더 줄일 수 있게 되었다. 이러한 추세에 발맞추어 최근 몇 년 동안 멀티코어 프로세서가 시장에 많이 출시되고 있는 상황이다. 인텔과 AMD 모두 2005년 이후에 내놓은 CPU는 멀티 코어에 맞추어 개발하고 있다. 심지어 인텔은 CPU 하나에 최소 10개, 많

으면 수백 개의 코어를 집어넣는 'Many 코어' 기술까지 연구하고 있다. 지금까지 작동 속도를 높이는 데 무게를 두었던 x86 CPU 제조사들이 멀티 코어 시대를 맞아 CPU 아키텍처를 새롭게 바꾸고 있는 것이다. 그리고 퀄컴의 차세대 휴대 단말 분야 혹은 PMP 업체 등의 임베디드 단말 분야에도 듀얼 코어 기반의 제품을 곧 출시할 예정이다. [1] [14]

회사	CPU	용도
IBM	Power4, Power5, PowerPC 970MP	PC, 서버
HP	PA-RISC	PC, 서버
Sun Microsystems	UltraSPARC IV, UltraSPARC IV+, UltraSPARC TI	PC, 서버
AMD	Opteron, Athlon 64 X2	PC, 서버
Intel	Core Duo, Core 2 Duo, Xeon	PC, 서버
TI	OMAP2230	모바일
QualComm	MSM7600	모바일
MagicEye	MMSP2, MMSP2+	모바일(국내)

[표 I]Multi-core CPUs

그러나 이 기술을 제대로 활용하기 위해서는 잘 작성된 멀티 쓰레드 프로그램과 같이 소프트웨어 기술이 필수적이다. 이러한 추세는 프로그래밍의 패러다임에도 큰 변화를 가져다주고 있다. sequential한 프로그래밍 패러다임이 주류였던 과거 몇 십 년과는 달리 앞으로는 parallel 프로그래밍이 대세가 되어가고 있다. 이러한 환경은 sequential 프로그램에 익숙한 프로그래머에게 낯설고 접근하기 힘든 진입장벽을 가지고 있는데 그러한 문제를 해결하기 위하여 sequential 프로그램을 자동적으로 분할 시켜서 각각의 코어에 돌아가게 하는 parallel 컴파일러의 도움이 절실 시 되고 있다. 물론 이전에도 이러한 개념을 가진 parallel 컴파일러 기술은 존재 하였다. 예를 들어 OpenMP API를 사용하여 루프 수준의 병렬화와 함수를 대상으로 하는 쓰레드 수준의 병렬화가 가능하였다. OpenMP는 프로그램 개발자가 쉽게 병렬프로그램을 작성하게 도와주지만 완전한 자동화 방식은 아니므로 개발자의 추가적인 노력이 필요하다는 단점이 있다. 이외에 90년대 Fotran용 병렬화 컴파일러가 활발히 개발되었으며 그 중 Fotran과 C 프로그램을 자동으로 병렬화 해주는 Suif라는 컴파일러가 유명하지만 루프 레벨의 병렬화 위주이며 현재 널리 사용되고 있지 않다.

이러한 parallel compiler 제작에 사용되는 기술은 여러 가지가 있을 수 있다. 사실 parallel compiler는 오랫동안 연구되어온 분야로 프로그램 병렬화를 위한 데이터 의존성 분석, 코드 변환, 코드 분할, 프로세스간 통신비용과 동기화 문제 등과 관련된 수많은 연구 결과들이 이미 존재한다. 하지만 멀티코어 프로세서는 최근에야 개발된 신기술로 이전 병렬 컴파일러 기술들이 고려하지 못하는 점이나 이전 실험 환경과 실제로 맞지 않는 점 등이 많다. 병렬 컴파일러가 대상으로 삼는 아키텍처들은 이미 시장에 출시 된 것일 수도 있고 아니면 아직 시장에 출시되지 않은 다분히 새로운 기능을 가진 아키텍처 일 수도 있다. 따라서 최근에 혹은 가까운 미래에 시장을 선도하게 될 parallel architecture의 모습을 잘 파악하여 실험환경을 구성하는것 역시 중요한 일중에 하나다.

컴파일러를 제작하는데 있어서 실험 환경을 시뮬레이션을 이용하여 구성 하는 방법은 과거로부터 많이 사용되어 왔다. 시뮬레이터는 새로운 환경을 저렴하게 흉내 낼 수 있으며 공개가 안되었거나 혹은 새로운 이론을 구현함에 있어서 좋은 수단이 된다. 멀티 코어 아키텍처에 관한 실험 환경을 구성하는 일 역시 크게 다르지 않다. 아직 구현이 안 된 멀티 코어에 관한 feature를 사용하거나 test하는 컴파일러를 제작할 때 있어서 시뮬레이션의 방법을 사용하면 빠르고 더 저렴한 방법으로 멀티 코어 아키텍처를 흉내 내는 testbed를 갖출 수 있게 된다. 특히 scalability 같은 측면을 고려할 때는 시뮬레이션의 방법을 이용하여 그 한계점이 어디인가를 쉽게 알 수 있다. 이러한 한계점은 컴파일러의 성능을 비교하는데

도 사용되지만 아키텍처를 실제로 어떻게 구현해야 하는가에 있어서도 도움이 될 수 있다. 새로운 아키텍처를 개발 한때 여러 가지 팩터를 먼저 시뮬레이터로 실험해 보고 그 결과에 따라 아키텍처의 구조를 변화 시킬 수 있다. [22] [23]

그러므로 컴파일러 개발 시에는 이러한 시뮬레이터의 개발을 컴파일러의 개발보다 선행해서 혹은 병행해서 시작을 해야 한다. 시뮬레이터는 아키텍처 그 자체를 구현하는 것이기 때문에 아키텍처의 분석이 선행되어야 한다. 이러한 아키텍처의 특징에 관한 분석을 통하여 올바르게 동작하는 시뮬레이터를 얻을 수 있을 뿐만 아니라 더불어 병렬 컴파일러 제작에 있어서도 optimal하게 아키텍처를 이용할 수 있는 기회를 제공해줄 수 도 있다. [21]

1.2 연구의 목표

병렬 컴파일러 개발에 있어서도 소프트웨어 공학적인 측면을 생각해 본다면 빠른 test 결과의 산출이 중요함에는 두말할 여지가 없다. 하지만 실제 하드웨어로 available 하지 않은 부분을 시뮬레이션이라는 방법을 택해서 우회한 것이기 때문에 실제 하드웨어에서 제공해 주던 장점인 speed나 scalability측면을 시뮬레이터는 잘 보장 해 줄 수 없다. 이런 부분이 잘 보장 되어야 컴파일러의 성능을 측정하는 실험에서 빠르고 제대로 된 실험결과를 산출해 낼 수가 있다. 빠르고 정확한 피드백이 컴파일러 제작 기간을 단축시켜 주기 때문에 시뮬레이터의 speed와 scalability를 향상 시키는 것은 시뮬레이터 제작에 있어서 중요한 factor가 된다.

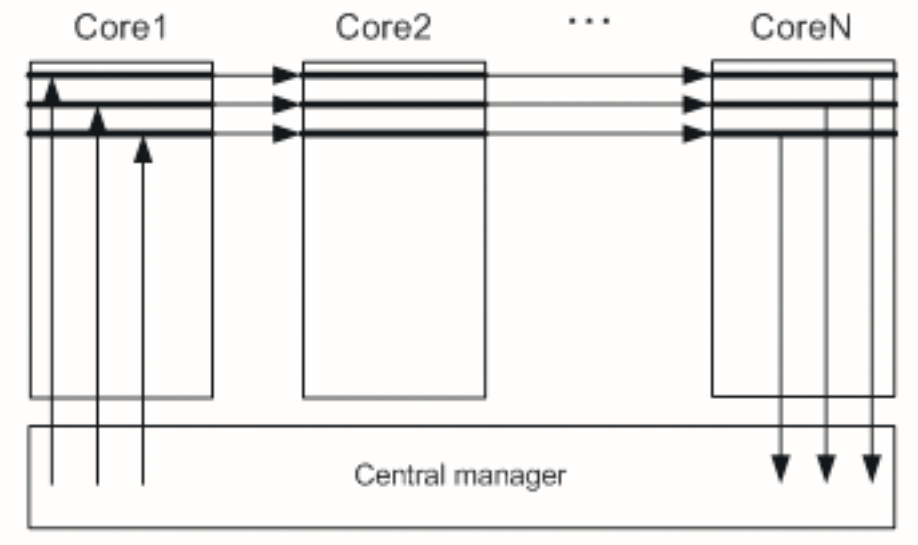
멀티코어 시뮬레이터의 속도가 전체 병렬 컴파일러 개발 기간에 영향을 끼치게 되리라는 것을 기본 전제로 본 연구의 목적은 다음과 같다. 일반적인 멀티코어 시뮬레이터의 구조에 여러 가지 기법을 적용하여 기존의 시뮬레이터보다 속도가 빠른 시뮬레이터를 개발한다. 또한 멀티코어 아키텍처의 특징상 코어의 개수가 many 코어 형태로 증가하게 될 때 쉽게 확장될 수 있는 멀티 코어 시뮬레이터의 구조를 개발한다. 그러기 위해서 우선 기존의 싱글 코어 시뮬레이터를 멀티코어 시뮬레이터로 바꾸는 것이 선행되어야 한다. 그러고 난 뒤 speed와 scalability를 향상 시킬 수 있는 기법을 이 연구에서는 제안하고 있다. 멀티코어에는 여러 가지 구조의 멀티코어가 있지만 본 연구에서는 공유 메모리 기반의 멀티코어를 그 대상으로 한다.

2 멀티코어 시뮬레이터 병렬화 기법

2.1 순차적 멀티코어 시뮬레이터

현재의 멀티코어 시뮬레이션 방법은 어떠한 Parallelization 기법도 사용하지 않고 한 cycle당 모든 core를 시뮬레이션 하는 방법으로 진행 한다. 이 경우는 단순히 single core를 simulation하던 프로그램에서 각 코어 파트에 해당하는 함수를 단순 하게 duplicate시키고 각 core 함수를 하나의 manager에서 순차적으로 수행 시키는 방식으로 진행이 된다. 이것에 대한 그림은 Figure 1에 잘나 타나 있다. [18] [19] [20]

그림을 보면 첫 사이클에서 Central Manager가 Core1부터 CoreN까지 순차적으로 실행을 시킨다. 그러고 난 뒤 다시 Central Manager에게로 다시 control이 돌아오면 각 코어마다 공유하는 global 변수들을 update해주고 두 번째 사이클을 첫 번째 사이클과 비슷한 방식으로 진행한다. 즉 모든 core의 simulation이 one cycle by cycle로 synchronized 되어 진행 된다고 할 수 있다. 이런 방식으로 진행을 하면 각 코어 끼리 공유 하는 지점인 cache나 shared memory등에 access하는 루틴은 모든 cycle이 sync를 맞추어서 진행하기 때문에 문제가 없다. 다만 Central Manager가 Core를 실행 시키는 순서는 임의로 혹은 bus contention이 일어나는 방식대로 Core를 실행 시킬 수 있다. 이 경우에도 공유 리소스에서 발생할 수 있는 원래 machine semantic을 보장해 준다. 만약 8개짜리 멀티 코어 시뮬레이터에서 sequential 방



[그림 1] Sequential Multi-core Simulator Program

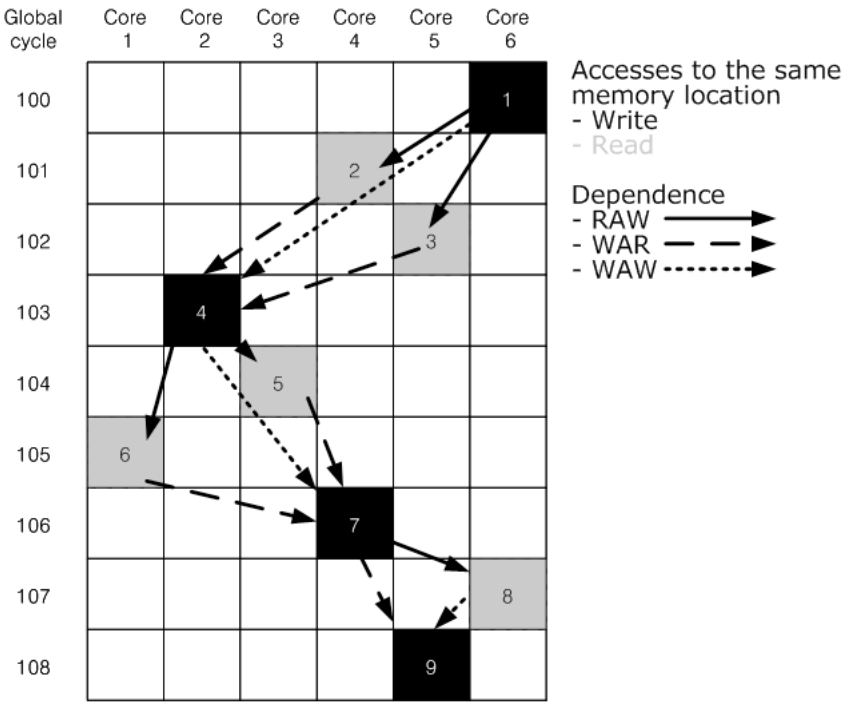
식으로 8개짜리 코어를 시뮬레이션 한다며 single 코어 시뮬레이션을 8번 반복하는 것인데, 이 8번의 반복은 실행시간을 8배 linear하게 증가 시켜준다. 코어의 개수가 점차적으로 증가하는 현재의 패러다임으로 봤을 때 이러한 시뮬레이션 시간의 linear한 증가는 scalability측면에서 시뮬레이션 반응성의 감소를 보여준다. [11]

2.2 쓰레드 수준의 병렬화

프로그램의 특징에 따라 병렬화의 granularity가 달라진다. 프로그램의 병렬성은 크게 데이터 수준의 병렬성과 쓰레드 수준의 병렬성, 파이프라인 수준의 병렬성 세 가지로 나눌 수 있다.데이터 수준의 병렬성은 큰 데이터 셋에 대해 서로 의존성이 없는 작업을 반복해서 수행할 때 각각의 태스크들이 서로 겹치지 않는 데이터 셋을 나누어 맡아 처리하는 것을 말한다. 쓰레드 수준의 병렬성은 서로 독립적인 데이터에 대해 독립적인 일을 하는 명령어들을 하나로 모아 태스크로 만들어 여러 태스크로 나누어 수행하는 것을 말한다. 이 때 보통 함수 단위로 태스크를 나누게 된다. 파이프라인 수준의 병렬성은 하나의 같은 데이터에 대해 일련의 작업을 수행할 때 각각의 스테이지별 하나의 태스크를 구성하며 병렬적으로 수행하기 위해 이전 스테이지의 결과가 들어오는 데이터를 FIFO방식의 큐를 사용하여 저장한 후 하나씩 꺼내어 현재 스테이지의 입력으로 사용하게 된다. 하지만 이런 병렬성들은 프로그램에 명확히 나타나있지 않은 경우가 많아 프로그램 분석을 통하거나 코드 변환을 통해야만 병렬성을 찾을 수가 있다. 앞의 두 병렬성은 서로간의 데이터 의존성이 없어야 적용 가능하지만 단 방향의 데이터 의존성만 존재하는 경우 마지막 파이프라인 수준의 병렬성을 찾을 수 있다. 시뮬레이터를 하나의 프로그램으로 본다면 역시 이러한 다양한 단계의 병렬성을 적용하여 프로그램을 병렬화 할 수 있다. 병렬화된 프로그램은 다시 하위의 멀티코어나 멀티프로세서 시스템의 도움을 받아서 더 빠른 실행이 가능할 수 있다.

이 연구에서 택한 병렬화 기법은 thread 수준의 병렬화 기법이다. 간단히 core를 하나의 태스크로 보았다. 서로 독립적인 데이터는 각 코어에서 사용하는 register file 등이 될 것이고, 독립적인 데이터에 대해 일을 하는 functional unit을 기본 적인 단위로 생각 하였다. 즉, 각 core를 시뮬레이션 하는 함수를 thread로 만들어 쓰레드 수준의 병렬성을 획득하려

했다. thread로 프로그램을 분할 하였을 때 thread간에 서로 독립적인 데이터에 대해서는 접근에 제약이 없다. 하지만 공유데이터 같은 경우에는 접근 시에는 동기화 문제를 생각해 주어야 한다. 즉, Cache, memory, bus 같은 공유자원을 잘 보호해서 race condition이 발생 하지 않게 보호 하는 것이 중요하다. 그래야만 원래 physical 하게 돌아가는 machine의 semantic을 어기지 않고 수행 할 수 있다. 이러한 thread로 나누는 과정과 각 thread간에 공유되는 변수의 동기화 문제는 결국 single threaded program을 multithreaded program으로 변형하는 문제에서 많이 접해 왔던 것들이다. [8] [12]



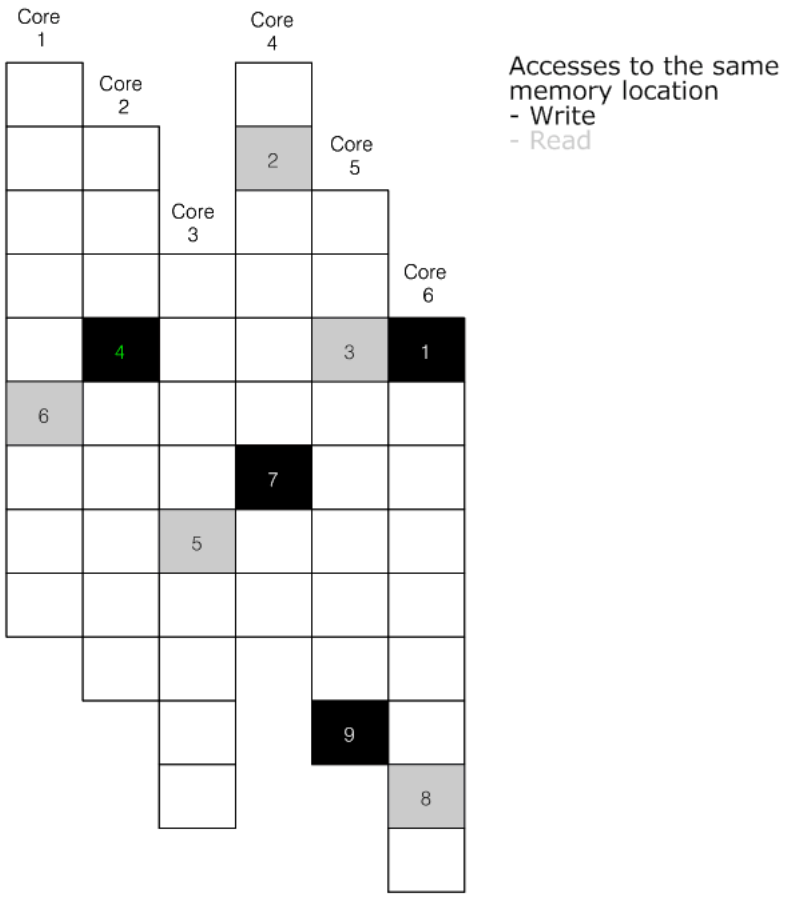
[그림 2] Access to shared memory in sequential Multicore simulator

Figure 2에 대한 그림은 sequential Multicore simulator에서 공유 자원인 shared memory에 접근할 때 모습을 나타내고 있다. 메모리 접근이 과한 프로그램을 인위적으로 만들어서 각각 코어가 shared memory에 read, write를 할때 dependency가 생긴다는 것을 알 수 있다. 우선 각 코어는 synchronized되어 돌아가기 때문에 global cycle이 존재 하고 그것에 따라 매 사이클 마다 core 간에 순서가 정해져서 그 순서대로 돌아간다. 여기서는 단순히 core의 번호가 적은 것이 먼저 실행된다고 생각한다. 각 코어에서 동시에 버스나 메모리 등에 접근한다면 번호가 적은 것이 먼저 자원을 차지하게 된다. 이 그림에서는 이러한 contention 사항은 고려하지 않았다.

cycle 100에서 core6이 write(1번 write)를 하고 이 write는 뒤의 cycle 101의 core4의 read(2번 read), cycle 102의 core5의 read(3번 read)에 영향을 준다. Figure 2를 보면 이러한 RAW (Read After Write) dependence가 실선 화살표로 표현되어 있다는 것을 알 수 있을 것이다. cycle 101의 core4의 read(2번 read), cycle 102의 core5의 read(3번 read)가 끝나고 난 뒤, cycle 103에서 core2가 write(4번 write)를 한다. core2의 write는 반드시 core4,5의

read가 끝나고 난 다음에 되어야 하며 Figure 2에서는 이것이 반점선의 WAR (Write After Read) dependence 로 표현되어 있다. cycle 100의 core6의 write와 cycle 103에서 core2의 write는 WAW의 (Write After Write) dependence가 존재 하는데 이 경우에는 1번 write와 2, 3번 read가 RAW dependence를 가지고 있고 2, 3번 read와 4번 write가 WAR dependence를 가지고 있기 때문에 hiding된다.

이런 식으로 각각 core가 shared memory에 접근 할때 생기는 dependence는 sequential한 core의 실행 순서가 보장이 됨으로 만족이 된다. 각 read, write간에 실행순서가 바뀌거나 하는 일은 발생하지 않으며 그러므로 simulated 되는 프로그램의 결과 역시 제대로 될 것이란 것을 보장해 준다. Sequential하게 각각 코어를 수행하는 시뮬레이터는 공유되는 어떠한 자원이라도 실제 physical level에서 일어나는 일들이 올바른 순서로 일어날 것임을 보장한다. 즉, 공유되는 bus, cache, memory 같은 것이 잘 보호 된다.



[그림 3] Access to shared memory in Multithreaded Multicore simulator

멀티쓰레드 버전의 멀티 코어 시뮬레이터의 각각 코어끼리 공유하는 자원에서 발생할 수 있는 동기화 문제가 Figure 3에 나와 있다. 각각의 코어는 시뮬레이터 프로그램에서 하나의 thread로 동작을 한다. 이런 경우에는 global cycle이란 것이 존재 하지 않고 각각 코어

마다 local cycle로 동작을 하게 된다. Global cycle은 없지만 각 local cycle이 일어나는 시점을 절대적인 시간 축에 놓고 봤을 때 local cycle의 형태가 Figure 3와 같다고 해보자. 이 그림에서 한 코어의 cycle 바운더리를 다른 것과 일치 시켰는데 이해를 쉽게 시키기 위한 것이고 실제로는 코어의 cycle 바운더리는 다른 것과 일치 하지 않을 수도 있다. 더 나아가 각 코어는 쓰레드로 구현이 됨으로 쓰레드가 block되는 상황에서는 코어의 실행이 멈출 수도 있다.

Figure 3에서 어떠한 동기화 문제가 발생하는지 살펴보면, 원래 physical level에서 존재하였던 dependence가 깨어지게 된다는 것을 알 수 있다. 구체적으로 예를 들어 core 4의 read(2번 read)와 core 5번의 read(3번 read)는 core 6의 write(1번 write)가 발생한 뒤 발생하여야 하는데 그렇지 못하다. core 4의 read는 3 cycle먼저 발생하므로 원래 머신에서는 있었던 RAW dependence가 깨어짐을 알 수 있다. core 5경우 동일한 cycle에 발생하지만 한 사이클을 시뮬레이션 할 때 먼저 메모리 시스템으로 access하는 코어가 5번 core가 될 경우 RAW dependence가 깨지게 된다. 이러한 dependence가 깨어지는 경우가 Figure 3의 하단부에서도 일어난다.

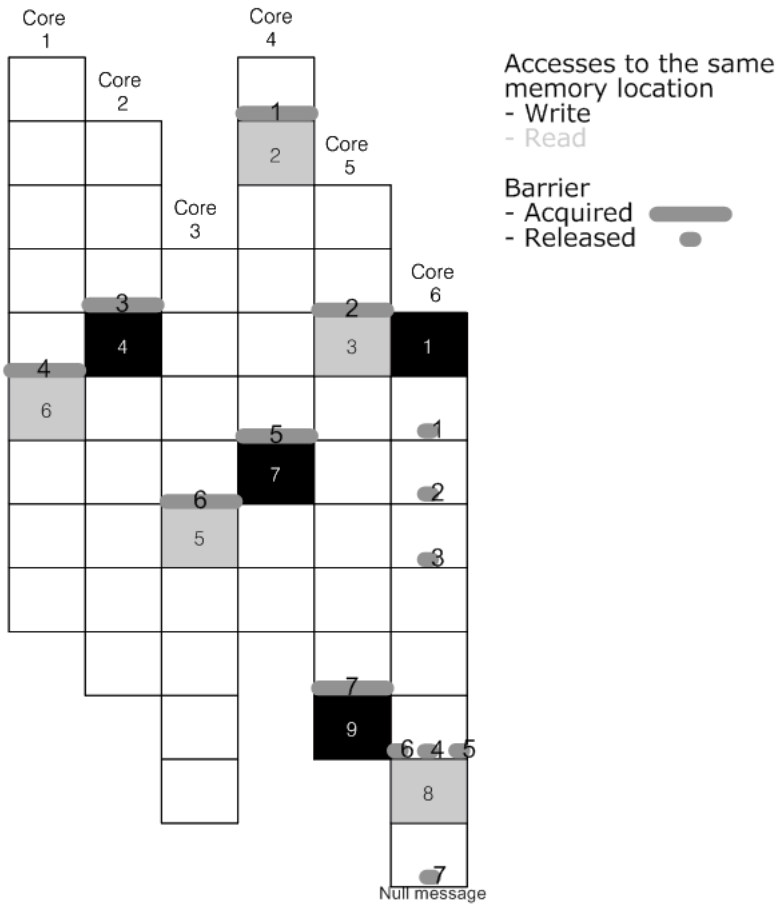
이렇게 각 코어를 thread로 mapping하여 시뮬레이터를 구동한 경우 physical level에서 발생한 dependence가 보장이 되지 않고 심지어 그로 인해 시뮬레이터 결과가 부정확 해질 수도 있다. 따라서 이런 공유되는 리소스는 항상 그 순서가 원래의 physical machine에서 지켜지는 방식대로 지켜져야 한다. 이것은 보통의 세마포어나 뮤텍스를 이용해서 한 번에 한 쓰레드만 공유 자원을 access하면서 보호하는 것 이상으로 엄격하게 공유 자원을 보호해야 된다는 이야기가 된다. 즉, 순서까지 정해진 동기화를 지켜 줘야 한다. 또한 단순히 write가 발생 하였을 때에만 동기화 처리를 해주어야 하는 것이 아니라 read가 발생 했을 때에도 동일하게 동기화 처리를 해주어야 한다. 한 코어에서 read가 발생 한 뒤, 다른 코어에서 실제로는 과거에 발생하여야 하는 write가 발생하는 경우도 있기 때문이다.

simulated 되는 프로그램에서 locking 기법을 사용하여 프로그램을 만들 었을때에는 locking기법 자체가 메모리 베리어 역할을 해주기 때문에, simulator에서 위와 같이 순서 까지 지켜주는 동기화를 시켜 줄 필요가 없을 것 같기도 하다. 하지만 simulated 되는 프로그램에서 발생하는 error상황의 semantic을 보장한다는 면에서 필요하다. 예를 들어 simulated 되는 프로그램에서 공유변수인데도 불구하고 동기화를 안 해서 발생하는 race condition이나 동기화 개체를 잘못 사용 했을 때의 dead lock semantic 역시 physical level에서 일어나는 그대로 보여 주는 것이 좀더 realistic한 시뮬레이터이므로 순서까지 지켜주는 동기화가 필요하다. [9] [10]

2.3 베리어 모델

Sequential하게 각각 코어를 수행하는 시뮬레이터는 공유되는 어떠한 자원이라도 실제 physical level에서 일어나는 일들이 올바른 순서로 일어날 것임을 보장한다. 코어를 thread에 mapping 시킨 multithreaded 버전의 멀티 코어 시뮬레이터는 physical level에서 일어난 dependence나 error semantic을 보장해주기 위해서 순서 까지 지켜주는 동기화가 필요하다고 했다. 순서까지 지켜주는 동기화 방식은 보통 다음의 Figure 4에서 나타난 방식대로 베리어 모델을 쓰는 경우가 많다. 본 논문에서도 베리어 모델을 사용하여 공유 자원들을 보호 했다. [17]

Figure 4에서 나타난 그림은 Figure 3에 베리어 모델을 덧붙인 그림이며 공유 자원 중에 특별히 shared memory 경우를 염두에 뒀다. 앞서 Figure 3에서 생긴 문제가 core의 local cycle이 서로 다르므로 한 core가 다른 core의 memory operation을 앞질러 가서 생기는 문제 였다. 이러한 앞지르기를 베리어를 통해서 막는다. 항상 memory operation이 일어 날 때는 다른 코어의 local cycle을 알아봐서 다른 코어가 자신이 실행 하려는 core를 통과 했는지 본다. 자신이 memory operation을 수행 하고 있고 가장 늦은 cycle이라면 바로 memory



[그림 4] Access to shared memory in Multithreaded Multicore simulator w/ barrier

operation을 처리한다. 그렇지 않고 자신보다 늦은 cycle의 core들이 있다면 우선 베리어에서 대기상태로 있는다. 다른 core들이 memory operation을 하면서 날려준 signal로 베리어에 갇혀 있던 core는 깨어나서 다시 자신이 가장 늦은 사이클인지 검사 하는 루틴으로 들어간다. 그러다가 어느 순간 자신이 가장 늦은 사이클이라는 조건이 충족되면 비로소 memory operation을 처리하게 된다. 즉, memory operation이 일어 날 때는 자신이 베리어에 들어가는 일과 베리어에서 대기 중인 다른 core를 깨우는 일을 모두 한다고 볼 수 있다. [15]

Figure 4를 기반으로 좀 더 자세히 설명하면 다음과 같다. 우선 절대적 시간 축에서 가장 빨리 도착하는 core4의 read(2번 read)가 자신이 가장 늦은 사이클을 진행하고 있는지 다른 코어의 local cycle을 검사 한다. 하지만 그렇지 않음을 깨닫고 베리어에서 잠든다(1번 베리어 acquired). 잠시 후, 비슷한 방식으로 core2의 read(4번 write)와 core5의 read(3번 read)가 다른 코어의 local cycle을 검사하고 베리어(2, 3번 베리어 acquired)에서 잠든다. 절대적 시간 축에서 가장 진행이 느린 core6이 cycle 2를 통과함으로 core4가 깨어날 조건은 충족되었다. 1번 베리어에서 잠들어 있던 core4가 깨어나는 시점은 Figure 4에서 1번 short bar(1번 베리어 released)로 표시 되었다. core4가 깨어나기 위해서는 다른 core가 memory operation을 하여서 시그널을 보내 주어야 하는데 이 그림에서는 core1이 read(6번 read) operation을 할

때 이다. 2번 베리어는 모든 코어가 3사이클이 지난 후에 release 될 수 있다. 역시 가장 진행이 느린 core6이 cycle 3을 통과 하고 난 뒤, core4의 write(7번 write)가 일어 날 때, core5가 2번 베리어에서 깨어나서(2번 short bar, 2번 베리어 released) memory operation을 진행한다. 3번 베리어도 1번, 2번 베리어가 release되는 비슷한 방식으로 release된다.

4번 베리어의 경우를 살펴보자. 4번 베리어는 모든 코어가 cycle 6을 지난 후에 비로소 해제 될 수 있다. 즉, 이것은 가장 진행이 느린 core6이 cycle 6을 통과하고 나면 해제 될 수 있다. 다만 core 6의 cycle 6을 통과 하는 지점에 4번 베리어가 해제 된다는 표시인 short bar가 없고 한 cycle 뒤의 cycle 7에 있는 것은 cycle 6에서 조건은 충족되었지만 memory operation을 진행하는 코어가 없기 때문이다. 그렇기 때문에 베리어 4, 5, 6은 모두 core 7이 memory operation을 하는 cycle 7이 되서야 해제 된다. 베리어가 해제 될 때도 4, 5, 6의 순서로 풀리는 것이 아니라 베리어에서 대기 하는 코어의 cycle수가 가장 작은 것부터 풀리기 때문에 6, 4, 5번의 순서로 베리어가 풀리게 된다.

7번 베리어는 어느 시점에 가서 해제가 되는지 알아보자. 가장 진행이 느린 core 6의 진행이 cycle 8을 지나고 난 시점에 7번 베리어가 해제 되어야 한다. 하지만 이대로 core 6에서 돌아가던 simulated 되는 프로그램이 종료했다고 가정해 보자. 모든 코어에서 돌아가던 simulated 되는 프로그램 역시 종료한 후 임으로 아무도 memory operation을 발생시키지 않고 따라서 core 5는 무한 대기에 빠지게 된다. 이러한 상황을 해결하기 위해서는 한 코어의 프로그램이 종료 했을 때 베리어에서 대기 하고 있는 다른 코어를 위해서 memory operation이 발생하지는 않았지만 가상적으로 signal을 전달할 필요가 있다. 이러한 메세지를 null message라고 하며 프로그램이 무한 대기에 빠지지 않게 하기 위한 중요한 방법이 된다.

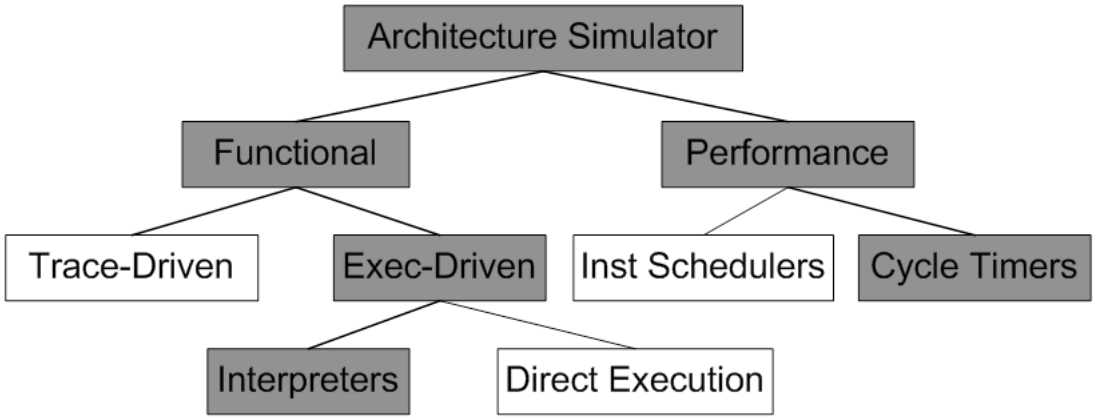
프로그램이 종료 했을 때만을 null message가 필요한 것은 아니다. 예를 들어 다음과 같은 극단적인 상황을 가정해 보자. core1에만 read(6번 read)가 있고 나머지 코어에서는 모두 memory operation이 사라 졌다고 해보자. core1번은 실행도중에 read를 만나게 되고 이윽고 베리어에서 대기하게 된다. 문제는 다른 어떤 코어도 memory operation이 없기 때문에 core1만 계속 해서 대기 하게 된다는 것이다. 이 경우 많은 시간이 지나고 난 뒤 베리어가 풀린다 하더라도 다른 core들이 core1을 기다리며 베리어에서 낭비하는 시간이 많아지게 됨으로 되도록이면 null message를 특정 시간 간격마다 한 번씩 뿌려 주는 것이 좋을 수 있다. [16]

위의 경우는 매우 극단적인 케이스를 가정한 것이다. 보통의 프로그램은 전체 수행의 30% 정도가 read, write등의 메모리 operation이다. 즉 10개의 instruction을 수행 시키면 그 중 3개는 메모리 operation이라는 말이 된다. 거의 3,4 사이클마다(1 instruction/cycle 가정) memory operation이 발생한다는 말인데 프로그램 종료 시점이외에 굳이 null message가 필요할까라는 의문이 들기도 한다. 그렇기 때문에 많은 시간동안 메모리 operation이 없을 경우에만 null message를 뿌려 주는 방법을 사용 할 수도 있다.

3 simple scalar

3.1 simple scalar의 개요

앞서 설명한 multithreaded 버전의 멀티 코어 시뮬레이터 제작을 위하여 simple scalar라는 아키텍처 시뮬레이터를 선정 하였다. [2] [3] 컴퓨팅 디바이스의 전체적인 행동 (machine, OS, 프로그램)을 흉내 내는데 그 목적이 있는 아키텍처 시뮬레이터는 system의 전체적인 설정과 simulated되는 프로그램을 입력으로 받아들여서 system output과 여러 가지 metric을 그 출력으로 내놓는다. 간단히 아키텍처 시뮬레이터의 특징을 구분하면 Figure 5 처럼 분류할 수 있다.



[그림 5] A Taxonomy of Architecture Simulator

Figure 5의 검게 음영이 진 부분이 simple scalar에서 구현한 부분이며 각각의 간단한 설명은 다음과 같다. functional simulator는 프로그래머가 프로그램 상에서 볼 수 있는 아키텍처 레벨의 기능을 구현하며 simulated되는 프로그램이 끝났을 때 machine의 state가 원래 physical level에서 돌린 것과 같은 결과를 가지게 하는데 초점을 맞춘다. 그에 비해 performance simulator는 micro architecture 레벨의 기능을 구현하며 functional simulator에서 보장하는 프로그램의 수행 결과에 추가하여 time 정보까지 physical machine과 비슷하게 하려 한다. 따라서 보통 functional simulator의 기능을 바탕으로 performance simulator를 만드는 경우가 대부분이며 따라서 보통 더 구현이 어려운 점이 있다. 또한 performance simulator는 더 기계에 가깝게 수행하도록 프로그램 되기 때문에 functional simulator의 속력보다 많이 느린 모습을 보여 주고 있다.

trace-based simulator는 먼저 프로그램을 실제 기계에 돌려서 나온 trace를 바탕으로 시뮬레이션을 하는 기법이다. 실제 cpu같은 functional unit을 구현하지 않고 register나 memory같은 machine state를 trace를 바탕으로 업데이트하기 때문에 비교적 구현하기가 쉽다. 그에 비해 execution-driven simulator는 run time에 program을 읽어서 functional unit을 이용하여 machine의 state를 업데이트 한다. 그러므로 trace-based simulator보다는 좀 더 구현이 어렵지만 실제 프로그램을 런타임에 수행시키므로 trace가 필요 없다는 장점이 있다. 다시 execution-driven simulator는 interpretation방식과 direct-execution 방식으로 구별되는데 interpretation 방식은 JVM처럼 하나의 byte code instruction을 여러 host machine 의 instruction 으로 바꾸어 동작하며 direct-execution 방식은 simulated program의 instruction을 instrument하여 host에서 직접 돌리는 방식을 취한다.

instruction scheduler는 execution graph를 이용하여 instruction을 micro 아키텍처 레벨의 자원에 mapping 시켜 주는 방식으로 동작한다. instruction은 한 번에 하나씩 in-order방식으로 graph에 mapping되며 따라서 좀더 physical machine을 잘 반영하지 못한다고 할 수 있다. 그에 비해 cycle-timer simulators는 각 사이클마다 micro architecture의 모습을 충실히 구현한다. 복수개의 functional unit과 pipelining, out-of-order issue, speculation같은 기법이 적용된 micro architecture 레벨의 시뮬레이터는 동시에 많은 instruction이 수행 중에 있을 수 있고 이것은 좀더 detail한 micro architecture state를 제공해 준다. 본 연구에서는 바로 performance simulator 중에서도 cycle-timer simulators를 대상으로 하였다.

3.2 simple scalar의 구조

Figure 6은 single core simple scalar tool set의 전체적인 block 구조도 이다. simulated 되는 프로그램을 static linking이 되도록 컴파일 하여 loader에게 전달하면 simulator의 interface 역할을 하는 loader는 simulated 되는 프로그램을 바탕으로 simple scalar simulator의 구동을 준비한다. 이후의 실행 flow를 추적하기 전에 먼저 simple scalar의 근간을 이루는 functional core, performance core, simulator core의 구조를 알아보기로 하자.

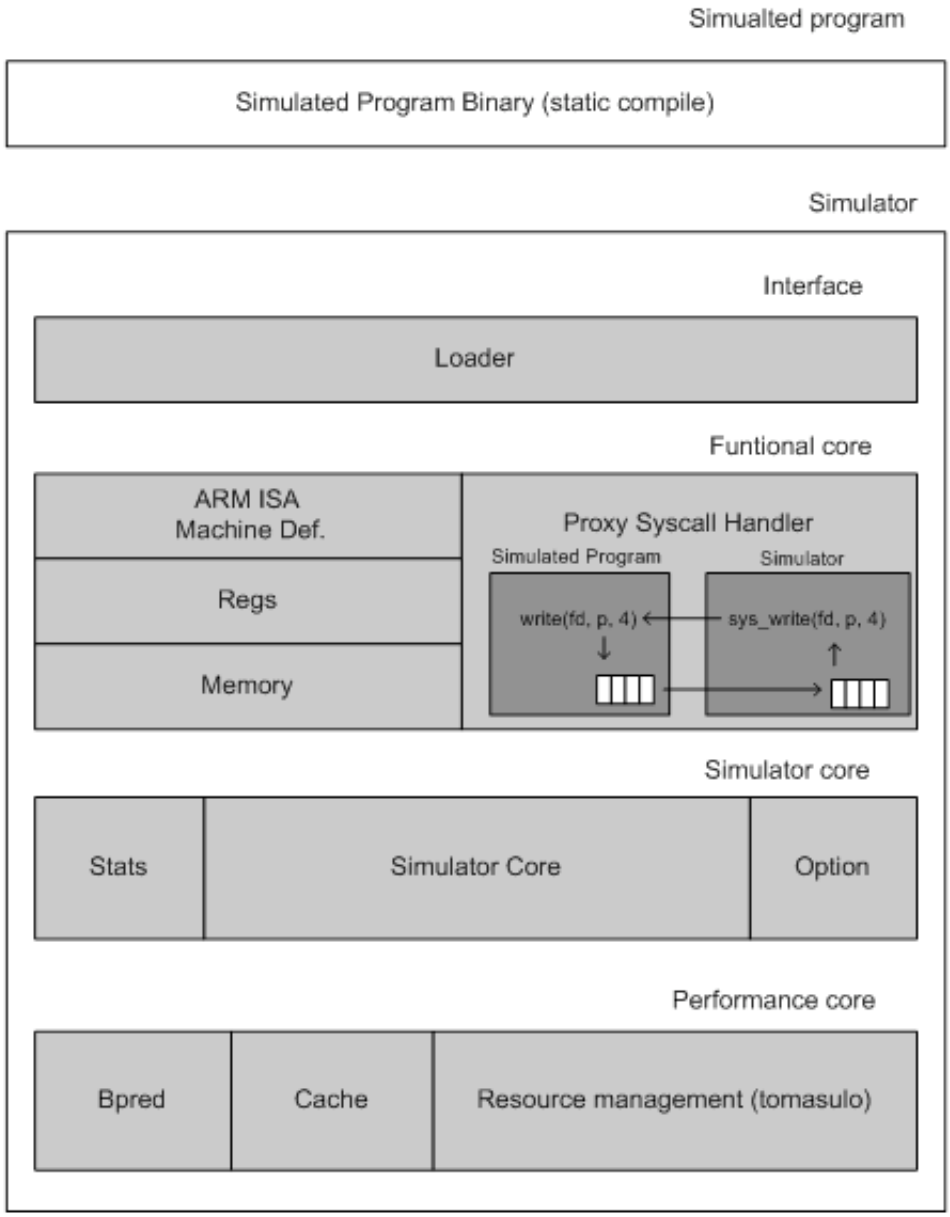
3.2.1 *functional core.* 시뮬레이터의 functional core는 functional simulator에서 필요로 한 architecture level의 기능을 구현하고 있다. Simple scalar는 pisa(potable isa), alpha, arm, ppc와 같은 Multiple ISA(Instruction Set Architecture)를 지원한다. 본 연구 에서는 arm ISA를 대상으로 한 functional core를 선택하였다. ISA만 ARM ISA를 따랐을 뿐이고 나머지 메모리나 캐시, micro architecture level의 구조는 simple scalar 특유의 구조를 사용한다. ARM ISA를 따랐기 때문에 simulated 되는 프로그램을 컴파일 할 때는 arm용 컴파일러를 사용하여 바이너리를 만들어야 한다.

Register set의 ISA구조에 맞게 따라 가야 함으로 보통의 arm 프로세서가 가지고 있는 형태를 유지 하였다. GPR (General Purpose Register)는 16개가 있고, 그 중 13번은 sp(stack pointer), 14번은 lr(link register), 15번은 pc(program counter)로 사용된다. 그리고 ARM ISA 내부 동작을 모니터링 하고 제어하기 위해 cpsr(current program status register)와 spsr(saved program status register)를 가지고 있다.

메모리 구조는 simple scalar가 멀티프로그램용 시뮬레이터는 아니지만 4GB virtual address space를 유지하고 있다. 간단한 Level 1 page table만을 구현했으며 각 page table entry는 linked list로 swap out 된 page들을 보관하고 있다. 실제 페이지가 요청될 때 호스트 machine에서 4KB 크기의 페이지를 할당하여 simulator에서 사용하는 구조다. virtual address space상에서 text 영역은 0x00400000부터 하위 주소(주소가 커지는 쪽)에 위치해 있으며 바로 아래에 data segment가 따라 붙고 그 하위에 동적 할당을 위한 heap이 시작된다. 0xC0000000부터 상위 주소(주소가 작아지는 쪽)로는 스택이 자라며 program을 위한 argument와 environment parameter를 simulator를 구동 시킨 쉘에서 받아와서 저장한다. 맨 하위 1GB는 가상적으로 커널의 사용을 염두에 두고 사용하지 않는다.

모든 OS의 feature가 simple scalar에 구현 되어 있는 것은 아니지만 간단한 system call은 proxy system call handler에 구현되어 있다. 실제로는 Ultrix Unix system call의 subset을 구현하였다. 시스템 콜을 처리하는 기본적인 알고리즘은 Figure 6의 proxy syscall handler part에 잘 나타나 있다. 먼저 simulated program에서 write(fd, p, 4) 라는 system call이 발생했다고 해보자. instruction이 system call(SWI)임을 알아챈 decode 로직은 proxy syscall handler logic을 호출한다. handler는 시스템 콜의 argument를 simulator의 memory에 복사하고 host에서 실제 시스템 콜 sys_write(fd, p, 4) 를 발생 시킨다. 그런 뒤 나타난 결과를 다시 simulated program memory에 복사함으로써 시스템 콜 처리를 완료한다. 만약 처리 할 수 없는 시스템 콜이 호출 된다면 handler는 그냥 무시해 버리거나 무시할 수 없는 경우 시뮬레이션을 멈추어 버린다. 이러한 구조로 인해 간단한 printf나 malloc같은 표준 c 함수들을 simulated program에서 사용할 수 있다. 다만 performance simulation시 시스템 콜 처리 부분은 시간 측정이 불가능 하다는 단점이 있다.

3.2.2 *performance core.* 시뮬레이터의 performance core는 performance simulator에서 필요로 한 micro architecture level의 기능을 구현하고 있다. 주로 정확한 시간을 측정하기 위한 구조들이 여기에 포함 되어 있다. 여기 있는 것은 약간 기능이 틀려도 데이터는 틀려지지 않고 다만 시간상의 측정의 오류만이 있었다는 것을 보장해야 한다. simple scalar에서는 multi level cache를 시뮬레이션 할 수 있다. 개별 L1, L2, TLB (각각 instruction과 data로 구별) cache의 구조를 설정할 수 있으며 전체적인 cache hierarchies를 여기서 조절 할 수 있



[그림 6] Simple Scalar Tool Set Structure

다. 모든 cache와 TLB 설정은 같은 포맷으로 가능 한데 set의 수와 associativity의 정도, set replacement 전략(LRU, FIFO, RANDOM)을 일렬로 나열한 것이 그 포맷이 된다.

branch predictor 역시 중요한 performance core의 한 파트가 된다. branch predictor는 몇 가지 카테고리에 따라 설정하는 방식이 완전 바뀌게 된다. 현재 시뮬레이터에 구현되어 있는 branch predictor의 카테고리는 perfect, bimod, 2-level adaptive predictor이다. 특히 branch predictor는 speculation을 일어나게 해주는 중요한 파트이기 때문에 time 정보에 크나큰 영향을 끼치게 한다.

이외에도 pipe lining, ooo(out-of-order) issue를 가능케 해주는 resource management 기능이 있다. resource management 기능은 복수개의 functional unit(integer ALU, integer multiplier/divider, Mem port)과 reservation station unit, reordering buffer, load store queue, create vector(register rename table) 등을 이용하여 tomasulo 알고리즘을 구현하였다.

3.2.3 *simulator core.* 시뮬레이터의 simulator core는 앞서 설명한 functional core와 performance core의 기능들을 이용해서 전체 시뮬레이터의 cycle당 처리해야 하는 일들의 루틴을 정하고 그것을 반복한다. functional core의 기능만 사용하면 simple scalar는 functional simulator의 모습으로 동작하고 performance core의 기능까지 사용하면 performance simulator의 모습으로 동작하는 것이다. 그래서 Figure 5에서 검게 음영이 진부분이 performance leaf에만 머물지 않고 다른 functional leaf 쪽으로도 존재하는 것이다. 현재 simple scalar에서 제공되는 simulator suite를 나열 하면 다음과 같다.

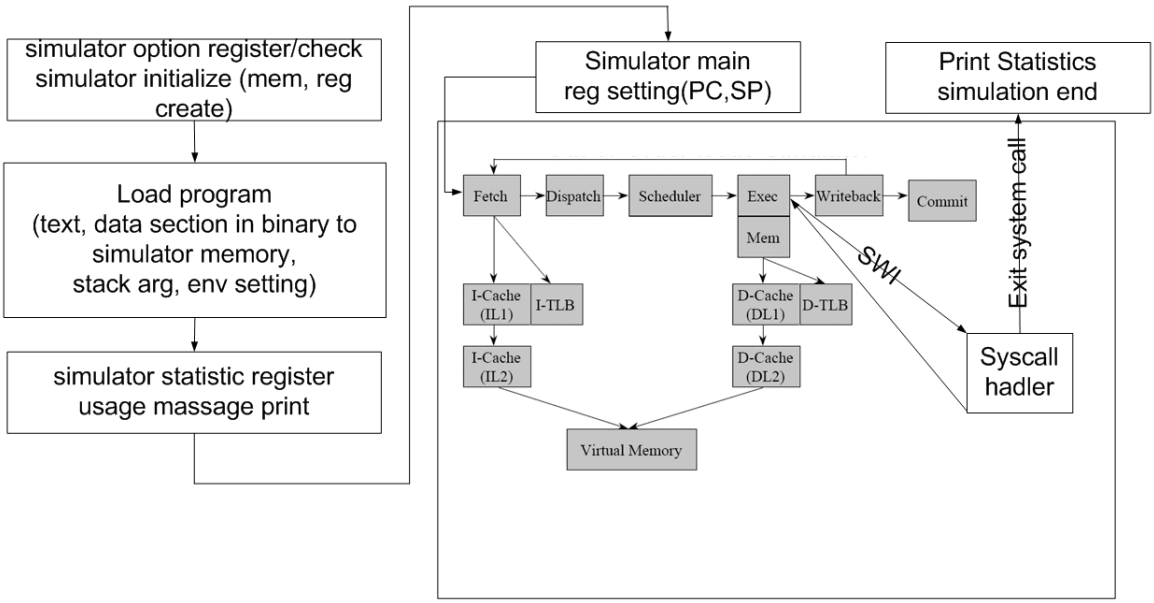
- Sim-fast: functional simulator
- Sim-safe: functional simulator, w/ option checks
- Sim-profile: functional simulator, lot of stats
- Sim-cache: performance simulator, cache stats
- Sim-outorder: performance simulator, ooo issue(functional units), branch prediction (misspeculation), cache(L1, L2, TLB)

Sim-Outorder가 multithreaded 버전의 멀티코어 시뮬레이터로 전환하려는 대상이 된다. 이외에도 simulator core에는 시뮬레이터 시스템의 옵션을 설정할 수 있는 option이라는 모듈과 시뮬레이터 시스템의 성능 측정을 위한 변수를 등록하는 stats라는 모듈이 존재한다. 이 모듈은 위의 모든 simulator suite에서 사용되는 범용 모듈이다.

3.3 전체 프로그램 수행 구조

Loader가 simulated 되는 프로그램을 simulator의 입력으로 받아들이고 난 다음에 발생하는 execution flow가 Figure 7에 나타나 있다. 먼저 simulator 환경을 설정하기 위한 option이 등록 되고 옵션의 값이 이상한지 check하는 루틴이 돌게 된다. 여기서 이상한 점이 발견되면 시뮬레이터는 시작 되지 않고 사용자에게 피드백을 주어 어떤 option이 잘못되었는지 리포트 한다. option에 아무 이상이 없으면 simulator의 machine state를 initialize 하는 루틴으로 들어 와서 memory와 register file을 create하고 initialize 해준다. 그런 뒤 앞서 받았던 simulated program의 code segment를 text영역에 로딩하고 data 영역 역시 setting 한다. stack에 simulator가 구동 되던 쉘 환경과 프로그램으로 전달되는 argument를 로딩 한다. 그러고 나서 simulator에서 추출 해 내기를 원하는 통계수치를 변수로 등록을 하고 simulator main으로 점프한다. simulator main에서는 기본적으로 프로그램이 시작할 pc값과 sp값등이 설정되고 performance simulation을 위한 여러 가지 리소스들이 create되고 나서 initialize된다. 그런 뒤 main loop의 처음으로 가서 시뮬레이션을 시작한다. 이 main loop는 프로그램이 끝나기 전까지는 cycle마다 계속 해서 반복이 된다. 프로그램이 exit()

시스템 콜을 호출 하면 시스템 콜 handler에서 longjmp()함수를 이용해서 loop를 벗어난다. loop를 벗어난 후에는 사용자가 원하는 statistics를 출력하고 시뮬레이션을 종료한다.



[그림 7] Simple Scalar Execution Flow

ooo issue 구조를 구현한 sim-outorder의 main loop은 Figure 7의 진하게 음영으로 처리된 부분이다. 제일 먼저 instruction을 fetch해 오는 단계로 시작한다. 이 단계는 machine fetch bandwidth를 모델링 하였다. program counter값과 predictor state를 보고 instruction을 fetch해온다. branch execution unit에서 mis prediction이 발생 하였다면 이 단계에서 알게 되고 정상적으로 prediction이 된 주소를 알 때 까지 block 된다. fetch 된 instruction은 dispatch queue에 넣어진다. 그리고 다음 사이클에 access할 cache line을 위해서 line predictor를 probe하게 된다.

dispatch 단계에서는 fetch 단계에서 넘어온 instruction을 scheduler instruction에 넣는 일을 행하게 된다. machine decode, rename, allocate bandwidth를 모델링 하였다. dispatch queue에 들어가 있는 instruction을 가져와서 decode하고 pre-execute를 해본다. pre-execute 단계에서 data dependence를 optimize하고 branch prediction을 미리 확인해 본다. 만약 prediction이 발생하였다면 machine의 상태를 speculative state buffer에서 업데이트 하도록 처리한다. 다만 copy-on-write시점에서 speculative state buffer사용을 시작한다. reservation station 과 load store queue에 entry를 할당하여 instruction의 decoding상태를 저장한다. memory operation이라면 memory dependence checking을 이 시점에 함으로 효율을 좋게 한다.

scheduler 단계에서는 reservation station 과 load store queue의 entry를 확인 하고 실제 functional unit에 issue시킨다. 이 단계는 instruction의 functional unit으로의 issue를 모델링 하였다. 모든 register input이 준비가 된 instruction을 issue시킨다. 만약 모든 메모리 input(effective address)이 준비된 memory instruction인 경우에는 다음과 같은 방식으로 동작한다. 이전에 발생한 store가 있고 현재 store가 같은 address라면 이전 stroe는 무시하고 merge해 버린다. 그렇지 않으면 D-cache를 access하게 한다.

execute 단계에서는 scheduler에서 받은 instruction을 실행 시키는 일을 한다. functional unit과 D-cache issue와 execute latencies를 모델링 하였다. 우선 사용가능 한지 functional unit state와 access port state를 probe 해본다. 사용가능한 개체가 있다면 Issue bandwidth만큼 Issue를 시킨다. execution이 끝나면 instruction을 writeback 단계로 보낸다. 그리고 functional unit의 상태와 D-cache의 상태를 update한다. D-cache에 access할 때 hit가 되면 D-cache access latency만큼만 cycle을 기다리지만 D-cache나 D-TLB에 대하여 miss가 나면 miss latency까지 포함해서 미래의 issue에 대하여 stall하고 있는다.

writeback 단계에서는 commit단계로 instruction을 보내기 전에 mis-prediction등을 검사한다. 즉, writeback bandwidth, mis-predictions detection을 모델링 하였다. mis-prediction의 경우에 mis-prediction recovery sequence를 발동시킨다. mis-predicted branch인 경우에 제일 처음 잘못 prediction한 branch로 이전으로 machine state를 recover한다. reservation station을 recover 하고 rename table에 들어가 있던 잘못된 output dependence 역시 이때 recover한다.

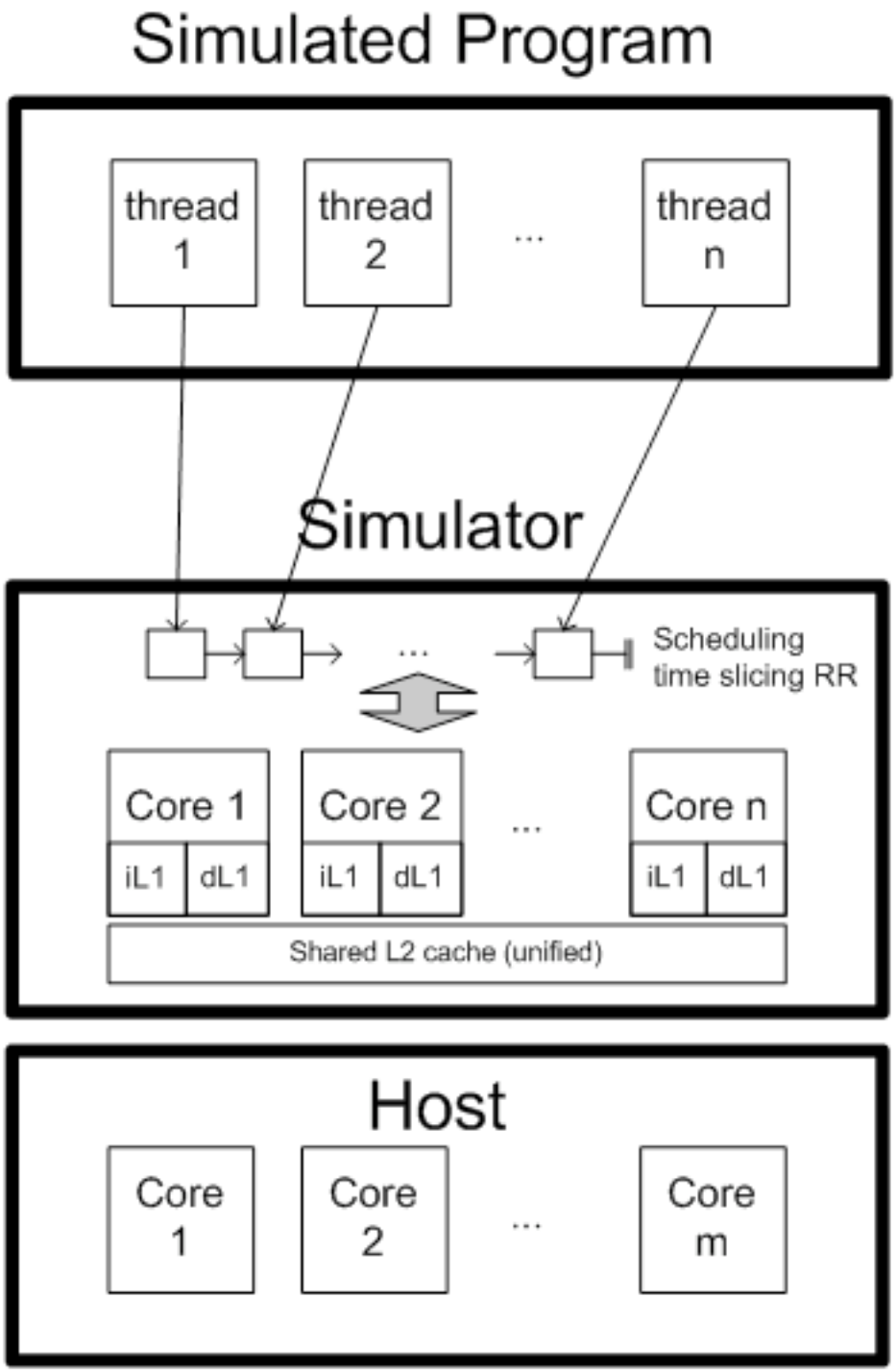
commit 단계에서는 instruction의 in-order retirement를 처리한다. 또한 store operation 경우에는 D-cache로 commit 하는 것을 처리한다. architected register file로 instruction commit 결과를 추가하고 그에 따라서 architected register file을 가리키던 rename table 역시 업데이트 한다. 그리고 다 쓴 resource들은 reclaim해서 뒤에 오는 instruction이 재활용해서 사용할 수 있게 한다.

4 simple scalar multicore버전의 개발

4.1 Core duplication

이전 까지 설명된 single core simulator(sim-outorder)를 multithreaded 버전의 멀티코어의 시뮬레이터로 만든 그림이 Figure 8이다. functional 코어의 일정 부분과 performance 코어의 일정 부분을 duplicate 하거나 하나의 task 단위로 만든 뒤, multithread화해서 multithreaded 버전의 멀티코어의 시뮬레이터를 만들었다. functional core던 performance core던 자료구조 부분, 즉 메모리에 할당되는 변수는 duplicate 대상이 되었다. 그리고 하나의 task(여기서는 함수라고 보자.)로 만들어서 multithread화해야 될 부분은 당연히 명령어의 집합이라고 할 수 있고 이러한 부분을 커다란 하나의 함수 루틴으로 만들어서 thread로 생성 시켰다. 이 함수 루틴을 앞으로 코어 thread라고 하겠다.

이전 Figure 6그림에서 functional core와 performance core를 보면 duplicate해야 하는 부분과 함수로 만들어서 multithread로 만들어야 할 부분이 각각 존재함을 알 수 있다. 우선 functional core를 먼저 살펴보면, register set의 경우는 duplicate해서 각 코어 마다 두어야 한다. 시뮬레이터 프로그램 안에서 register set은 하나의 structure로 구성되어 있다. 코어 thread마다 이런 structure가 필요하게 됨으로 각 코어 thread 별로 array 처럼 지역 공간으로 분리해 두는 방식을 택할 수 있다. 실제 구현은 thread local storage라는 gcc 3.3 버전 이후에 도입된 기술을 사용하였다. 이 기술은 모든 쓰레드가 동일한 이름의 변수를 사용해도 키워드 __thread만 변수 앞에 붙여주면 실제로는 각 쓰레드마다 독립적인 공간에 변수 값을 저장하게 해준다. 이미 만들어져 있는 single 쓰레드 프로그램을 병렬화 하고 싶을 때 유용하게 사용될 수 있는 기술로써 thread를 새로 생성했다고 해서 공유변수의 이름을 다 바꾸어 주거나 array화하지 않고 단순히 쓰레드별로 지역 변수처럼 사용할 수 있다. 실제로 __thread 키워드는 glibc에서 에러 값 errno에 사용되고 있다. 이는 쓰레드 마다 다른 에러 값을 저장하기 위해서이다. __thread는 gcc고유의 키워드 이며, 호환성 있게 사용하는 방법으로 pthread의 쓰레드 고유 데이터(TSD: Thread Specific Data)가 준비되어 있다. 그 예로 pthread_key_t 자료 형과 pthread_key_create 함수가 있다. tls는 사용에 제약이 있는데



[그림 8] Multi-threaded Multi-core Simple Scalar Tool Set Structure

리눅스를 예로 들면, TLS는 리눅스 커널 2.6 + gcc 3.3 + glibc 2.3 + NPTL하에서 활용 할 수 있다. 또한 아키텍처에 따라서도 구현방법이 크게 다르다. 예를 들면 x86에서는 TLS를 구현하기 위해 쓰레드 레지스터로 %gs 세그먼트 레지스터를 이용한다. 그 중에는 MIPS와 같이 여분의 쓰레드 레지스터가 남아 있지 않으므로 TLS를 구현할 수 없는 아키텍처도 있다. [13]

각 코어 thread에서 공유되는 ISA는 코어의 physical logic이라고 할 수 있다. ISA를 구현한 코드만 빌려 쓰는 것이기 때문에 각각의 코어 thread에서 공유하여도 문제가 생기지 않는다. Memory는 본 연구의 대상이 shared memory based multicore simulator이기 때문에 duplicate하지 않아도 된다. 만약 분산 메모리 환경이라면 각각의 코어는 자신의 로컬 메모리만 접근할 수 있으며 다른 메모리에 있는 데이터를 접근하기 위해서는 DMA(Direct Memory Access) 명령어를 사용하여 원하는 데이터를 자신의 로컬 메모리로 가져와야 한다. 이러한 경우에는 코어를 duplicate하는 작업에 memory를 duplicate하는 작업 역시 추가 되어야 한다. 시스템 콜 부분은 사실 자료구조가 아니라 명령어의 집합으로 보는 것이 맞다. 하지만 시스템 콜 부분에서 공유되는 변수가 있기 때문에 시스템 콜에 여러 코어 thread가 접근 할 때는 동기화가 필요하다. 다만 앞서 설명한 베리어 모델까지는 사용할 필요가 없고 mutex등을 이용하여 한 번에 한 코어 thread가 들어간다는 것 정도만 지켜 주면 된다. 실제 이러한 방식은 리눅스 커널 2.4처럼 reentrant 하지 못한 커널을 모델로 하여 시스템 콜 부분을 구현하였기 때문에 발생하는 문제이다. 리눅스 커널 2.6 처럼 커널안의 좀 더 세밀한 레벨에서 공유되는 변수를 보호하는 운영체제라면 여러 코어 thread가 동시에 시스템 콜을 사용하여도 문제가 없을 것이다.

이제 performance 코어에서 duplicate할 부분과 thread화 시킬 부분을 살펴보자. branch predictor는 각 코어 thread 별로 존재해야 하는 자료구조다. 현재 코어가 실행 하고 있는 instruction의 스트림을 잘 예측하여 branch가 나왔을 때 taken할지 not taken할지 결정 하는 것이므로 다분히 각 코어의 현재 상황에 종속적일 수밖에 없다. branch predictor는 predictor부분과 BTB(Branch Target Buffer)로 구성이 되는데 두 모듈 역시 structure로 되어 있으므로, TLS를 써서 구현을 하였다. 다만 mis-speculation이후의 roll back과정에서 이미 진행한 다른 코어의 메모리 operation의 semantic을 깨는 경우가 생길 수 있다. 베리어 모델에 이러한 제약 사항까지 더해지면 operation이 더 빈번하게 speculative machine state에서 진행되고 완벽히 mis-speculation이 없다는 보장 하에서만 original machine state에 commit하는 상당히 큰 오버헤드가 발생한다. 그래서 branch predictor를 이용은 하되 wrong path로 mis-speculation하는 행동은 막아 두었다. 실제로 상용 멀티코어 칩을 봐도 이러한 복잡도의 증가와, 각 코어를 duplicate하는데 소비되는 transistor로 인하여 branch predictor 구조를 간략히 하거나 생략하는 경우가 많다.

리소스 매니지먼트 코드는 tomasulo 알고리즘을 충실히 구현하고 있다. 자세한 것은 앞선 3.3 절에서 설명했던 Execution Flow을 보면 된다. 각 코어 안에서 ooo issue가 일어나는 것은 전체적인 multithread 멀티코어 시뮬레이터의 수행 구조이다. 따라서 이 부분을 하나의 함수로 만들어서 multithread화하면 쉽게 앞서 말한 multithread 멀티코어 시뮬레이터를 구현할 수 있다. 실제로 Execution Flow는 하나의 simulator main함수에서 각기 다른 pipeline stage 함수를 부르는 것 이다. 그래서 구현은 단순하게 이 simulator main함수를 pthread_create함수를 이용해서 multithread화하는 방식으로 진행 했다. 이 부분이 바로 위에서 설명한 코어 thread이다. Figure 8 에서 코어 thread 부분은 Core1, Core2 ... CoreN으로 표시되는 block이다. simulated program에서 만들어진 thread가 work queue 방식으로 queue에 매달려있으면 코어 thread들이 그 queue에서 job(simulated program의 thread)을 하나씩 꺼 내와서 정해진 시간동안 처리하고 다시 work queue의 맨 끝에 넣는 방식으로 멀티코어 시뮬레이터가 동작한다. 물론 work queue를 어떻게 매니지먼트 하는가도 성능에 영향을 끼칠 수 있는 factor이다. I/O등으로 thread가 block이 된 경우에는 따로 I/O queue를 만들어서 달수도 있고 한번 할당한 코어에는 cache affinity 측면을 고려하여 다시 할당 될

확률을 높이게 하는 등의 optimization 기법을 생각해 볼 수 있다. pthread 구현과 비슷해 질수록 성능은 좋아 질것이다.

캐시는 실제로 앞서 설명한 베리어 메모리 모델을 사용해야 하는 단 하나의 공유 지점이다. Figure 8에서 공유 지점은 shared L2 unified cache로 나오고 있다. 하지만 private L1 data cache에 cache coherence protocol이 돌고 있고 항상 cache는 inclusion property를 만족하기 때문에 모든 공유가 L1 data cache에서 일어난다고 말할 수 있다. 실제 베리어 모델의 구현은 Figure 9에 있는 코드를 이용하였다. 모든 memory 접근 operation이 있을 때는 먼저 L1 cache부터 뒤지게 된다. 여기에 있으면 바로 써버리고 miss가 나면 하위 메모리 시스템에서 가져오게 되지만 결국 L1 cache에 쓰게 된다. 이러한 L1 cache를 access하는 함수에서 L_synchronize 함수를 먼저 거치게 함으로 베리어 모델은 달성 된다. 베리어는 앞서 2.3절에서 설명한 알고리즘을 pthread_mutex_lock과 unlock, pthread_cond_wait와 broadcast로 구현을 하였다. 공유 변수인 l1_sync array에 접근해야 함으로 먼저 lock 잡고 자신의 local cycle을 다른 코어가 볼 수 있게 업데이트 한다. 그런 뒤 자신이 가장 느린 cycle을 진행하고 있다는 조건이 만족 된 코어가 있을 수도 있으니 다른 코어를 깨우기 위해서 signal을 broadcast 한다. 그런 뒤 무한 loop를 돌면서 자신이 가장 느린 cycle을 진행하고 있는지 검사를 하고 그렇지 않다면 조건 변수에서 잠든다. 잠에서 깬 어느 순간 조건이 만족한다면 베리어에서 나와서 memory operation을 진행 한다. 실제로 무한 대기의 가능성이 존재함으로 각 코어는 원래 load나 store가 일어나는 비율인 5사이클 마다 signal을 날려준다. 이것이 바로 Null message이다. 또한 현재 코어가 simulated 하는 쓰레드가 종료 하였을 때도 signal을 날려준다.

```
void L1_synchronize() {
    bool_t check_again = FALSE;
    pthread_mutex_lock( &l1sync_cond_mutex );
    l1_sync[core_id] = sim_cycle;
    pthread_cond_broadcast( &l1sync_cond );
    do {
        check_again = FALSE;
        for (int i = 0; i < num_cores; i += 1) {
            if (i != core_id && l1_sync[i] < sim_cycle) {
                    check_again = TRUE;
                    pthread_cond_wait( &l1sync_cond, &l1sync_cond_mutex );
                    break;
            }
        }
    }
    while (check_again);
    pthread_mutex_unlock( &l1sync_cond_mutex );
}
```

[그림 9] L1 Data Cache Synchronized w/ Barrier

L1 instruction 캐시의 경우에는 read only이기 때문에 cache coreherence protocol이 돌 필요가 없다. 그러므로 공유지점이 되지 않고 L1 data cache와는 달리 메모리 베리어 모델이 필요 없다. DTLB, ITLB 의 경우에도 duplicate가 필요하다. 현재 수행하는 메모리의 working set은 각 코어마다 많이 차이가 날것이다. TLB를 공유하면 capacity의 부족으로 인한 tlb miss penalty가 커질 것이고 그렇게 되면 physical level을 잘 반영 할 수 없을 것이다. 거기다가 공유되는 지점에 필요한 동기화 문제역시 큰 성능저하를 불러 올 수 있다. 따라서 TLB는 duplicate를 한다. 그렇지만 실제 원본인 memory page table은 하나를 공유한다. 그렇기 때문에 다른 코어가 빨리 혹은 늦게 진행함으로 써 생기는 원래

physical level과는 다른 page table의 정보가 부정확한 cycle정보를 생성 할 수 있다.(잘못된 결과가 나오지는 않는다.) TLB의 접근은 항상 data 캐시나 instruction 캐시를 접근하는 memory operation이 있어야만 발생한다. DTLB가 같은 경우에는 L1 data cache를 접근하려던 memory operation이 베리어를 통해 순서까지 맞춘 동기화를 보장하기 때문에 앞서 말한 physical level과는 다른 page table의 정보가 발생하지 않는다. 하지만 ITLB의 경우에는 L1 instruction cache를 접근하려던 memory operation이 read뿐만 임으로 동기화를 맞추지 않는다는 점 때문에 physical level과는 다른 page table의 정보가 발생할 수 있다. 이것을 막으려면 ITLB에 베리어를 치던지 instruction cache에 베리어를 쳐야 하는데 매 사이클 read만 일어나는 instruction 캐시에 베리어를 친다는 것은 너무 성능 저하가 발생할 수도 있을 것 같아서 막았다. 그렇다고 마찬가지로 매 사이클 access가 되는 ITLB에 베리어를 친다는 것 역시 큰 성능 저하가 우려되기 때문에 약간은 physical level의 semantic과는 다르지만 ITLB와 instruction 캐시 모두에 베리어를 치지 않았다. 약간은 틀린 사이클 정보가 나온다는 것이 한계가 될 수 있지만 최악의 상황은 피할 수 있었다. [4]

4.2 multi-threading feature를 더하기

multithreaded 멀티코어 시뮬레이터는 완성이 되었다. 이제 simulated 되는 프로그램을 어떠한 구조를 가져야 하는지 이야기 해보자. Single threaded application은 당연히 멀티코어 환경을 이용하지 못한다. 멀티 프로그램은 멀티 코어 시뮬레이터를 이용할 수는 있지만 각 코어간의 캐시나 TLB사이에 sharing이 적을 뿐더러 일어난다 할지라도 false sharing이 대부분 일 것이다. 그렇기 때문에 프로그램을 병렬 컴파일러로 컴파일 되어 나온 바이너리의 성능을 멀티코어 환경 위에서 측정해보고 컴파일러 제작에 피드백을 주자라는 원래의 시뮬레이터 개발 목적에 부합하지 않는다.

위의 사항을 고려하면 multi-threaded 프로그램이 필요하다. 각 threaded가 서로 다른 멀티코어를 점유하면서 실행되어야 한다. 그렇기 때문에 user level의 멀티 쓰레드 프로그램은 적합하지 않다. ULT(user level thread) 프로그램은 runtime이 따로 프로그램에 하단에 붙는 방식으로 돌아간다. OS가 일정한 time quantum을 현재 프로세스에 할당하면 runtime이 다시 그 time quantum을 쪼개서 thread에게 할당 하는 방식으로 구현이 된다. thread는 OS와 machine이 어떤 구조를 가지고 있는 알지 못하고 반대로 OS와 machine도 현재 이 프로세스가 어떠한 방식으로 동작하는지 모른다. 중개자 역할을 하는 runtime이 OS와 machine의 구조를 감추어 버리기 때문에 thread는 하위에 멀티코어가 있는지도 모르고 더욱이 사용할 수도 없다. glibc에 있는 setjmp(), longjmp()라는 continuation passing style의 함수를 사용하면 ULT를 쉽게 구현할 수 있으나 우리가 원하는 feature를 나타내기 위해서는 KLT(kernel level thread)가 필요하다. KLT는 ULT와 반대로 OS와 machine의 구조가 어떻게 되는지 알고 있으면 따라서 하위 구조가 멀티코어 라면 그 구조를 이용하도록 노력할 것이다. 즉, 커널 안에 프로세스처럼 쓰레드가 생성이 되고 실행이 되는 방식으로 동작을 한다. 본 연구에서 제작한 multithreaded 버전의 simulator를 테스트 해보기 위해서 다음과 같은 KLT를 구현하였다. [5] [7]

커널 안에서 쓰레드가 구현 된다고는 하지만 현재 우리의 시뮬레이터는 커널이나 OS의 개념을 가지고 있지 않다. 다만 커널의 subset이라고 하는 syscall handler가 있으면 이것과 코어 thread에 간단한 KLT를 구현하였다. 대표적인 KLT인 pthread를 최대한 벤치마킹 하였으며 아래에 보이는 함수들의 set을 구현하였다.

– int thread_create(void (*funcptr)(), void *arg);
  thread create 함수는 thread로 실행 시킬 함수를 가리키는 함수 포인터와 thread에 인자로 전달할 void 포인터를 인자로 받는다. ARM에서 함수 호출시 인자를 r0부터 r3까지 담는 것과 비슷한 방식으로 함수 포인터와 void 포인터를 inline assembly를 이용해서 레지스터 r0와 r1에 담고 나서 SWI(system call instruction)를 호출한다. syscall 핸들러에서

는 r0와 r1에 담겨진 함수 포인터와 인자포인터를 바탕으로 work queue에 새로운 item을 단다. 하나의 item은 simulated program의 하나의 thread에 mapping이 되며 다음과 같은 정보를 가지고 있다. [6]
thread id는 thread 별로 고유한 값이며 처음 thread가 생성이 될 때, 즉 item의 새로 만들어 질 때 주어진다. 또한 item을 위한 register set이 할당이 된다. 왜냐하면 현재 쓰레드가 코어 thread상에서 돌다가 work queue로 yield되었을 때, register set의 정보를 유지하고 있어야 하기 때문이다. 그래야 다음번 새로운 코어 thread에 할당되었을 때 코어 thread의 register set을 현재 자신이 가지고 있는 register set 으로 덮어씌울 수 있고 그래야만 제대로 된 수행이 보장 받기 때문이다. stack base, SP(register set의 stack pointer)는 각 thread 별로 고유한 stack의 처음과 끝을 가리키는 수치로써 처음에는 각 thread별로 virtual address space상의 고유한 위치를 부여 받는다. 그리고 PC(register set의 program counter)와 r0(register set의 r0)는 아까 전에 전달된 함수 포인터 인자와 void 포인터 인자 값을 갖게 된다. 이 같은 변수 세팅 과정이 끝이 나면 work queue의 맨 끝에 지금 만들어진 item이 붙는다. 그러고 이 item이 core thread을 점유하면 그때부터 thread가 실행되는 것이다.
Thread scheduling은 특별히 함수가 없고 item이 core thread에 할당되고 나서 정해진 시간이 지나면 다시 work queue로 돌아가는 라운드 로빈 방식을 선택하였다.

- int thread_exit();
- int thread_join(int thread_id);
  위의 두 함수는 쓰레드간에 동기를 맞추기 위해서 구현되었다. thread join함수는 스케줄링 타임에 thread id에 해당되는 특정 item이 끝났는지 확인을 하고 끝나지 않았다면 자신의 item이 core thread에 할당 되지 않게 하였다. 즉 thread id에 해당하는 item이 끝날 때 까지 현재 item을 기다리게 하였다. exit는 그런 과정을 모든 item에 대하여 확장한 함수다. 모든 item이 끝나기 전까지 자신의 item은 실행을 하지 않고 기다리게 된다. 이 함수는 모든 thread가 끝나고 자신이 마지막으로 무언가를 처리할일이 있을때 유용하다.
- int mutex_init(pthread_mutex_t *mutex);쓰레드 간에
- int mutex_lock(pthread_mutex_t *mutexm);
- int mutex_unlock(pthread_mutex_t *mutex);
- int mutex_destroy(pthread_mutex_t *mutex);
- int cond_init(pthread_cond_t *cond);
- int cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- int cond_signal(pthread_cond_t *cond);
- int cond_broadcast(pthread_cond_t *cond);
- int cond_destroy(pthread_cond_t *cond); 모든 mutex와 conditional variable에 관한 함수는 현재 simulated되는 프로그램에서의 자신의 주소를 syscall handler로 보낸다. syscall handler에서는 이 주소를 simulator에서의 주소로 바꾸고 그 주소를 이용하여 바로 pthread mutex와 pthread condition variable 함수를 호출한다. 즉 program의 동기화 개체를 simulator에서의 동기화 개체로 mapping시켜서 구현을 하였다. 그렇기 때문에 simulated program에서 바로 pthread 타입의 변수를 사용한 것이다.

위에서 구현한 KLT함수를 이용하여 간단한 multi thread 프로그램을 만들 수 있으나 spec 벤치 마크 같은 대용량의 프로그램을 multithread로 만드는 것은 무리가 있어 보인다. 좀더 pthread 가깝게 구현을 해야지 그런 대용량의 프로그램에도 잘 적용 할 수 있을 것이고 현재는 간단한 multithreaded 프로그램을 만들어서 simulator에서 돌려보고 simulator의 성능을 측정해 보았다.

4.3 upper limit of transactional memory

앞선 multithreaded 멀티코어 simulator에서 사용되는 동기화 개체는 다음과 같다.

(1) work queue에서 item(simulated program의 thread)을 추가/제거할 때 사용하는 동기화 개체

(2) L1 data cache에 대한 메모리 베리어 동기화 개체

(3) 시스템 콜을 호출 할 때 사용되는 동기화 개체

(4) simulated 되는 프로그램 안에 내제하고 있는 동기화 개체

simulator를 하나의 멀티 threaded 프로그램으로 봤을 때 이같이 많은 동기화 개체가 존재 하면 동기화 오버헤드로 인하여 각각 코어를 sequential 하게 시뮬레이션 하는 것보다 더 많은 시간이 걸릴지도 모른다. 병렬화 하는 것이 항상 좋은 결과만을 보여주는 것은 아니다. 이렇게 동기화 개체가 많을 경우 사용할 수 있는 한 가지 방법은 optimistic 하게 공유되는 변수에 동기화 개체를 사용하지 않고 나중에 공유되는 변수에 race condition이 있었다는 것을 알게 될 때, 다시 그 부분을 roll back 해서 수행하는 방식이 있다. 이렇게 매 메모리 operation마다 optimistic한 방식을 적용하는 기법을 transactional memory기법이라고 한다. 이전의 multithread 멀티코어 시뮬레이터의 성능 향상정도와 transactional memory 기법을 더 적용 할 때 생기는 이득의 upper bound를 비교하여 앞으로 시뮬레이터의 성능 향상을 위하여 이러한 기법을 적용해도 되겠는지를 더 알아보았다.

나열한 동기화 개체 중에 가장 자주 그리고 큰 동기화 오버헤드를 부여 하는 것이 L1 data cache에 대한 메모리 베리어 동기화 개체 이다. 또한 다른 동기화 개체는 반드시 필요하거나(1, 4번) 시뮬레이터의 대대적인 수정이 필요한 것이라서(3번) 이것에 대해서 새로운 기법을 적용하는 것이 가장 합당할 것이다. 이 베리어를 완전 제거 하여 최대로 얻을 수 있는 수치는 분명 시뮬레이터가 어떠한 기법을 쓰던 얻을 수 있는 가장 큰 성능 향상을 보여 줄 것이다. 물론 시뮬레이션 결과값이 틀릴 수 있으나 최대치를 알아보기 위한 것이다. [24]

또한 transactional memory기법의 ideal case일때 결과를 산출해봐서 이 방식이 더 try해 볼 만한 방식인지 알아보자. transactional memory 기법의 ideal case는 베리어를 완전 제거하는 것이 아니라 특정 공유되는 주소에 관해서만 베리어 동작을 하게 만들어 transactional memory에서 발생할 수 있는 roll back 현상을 연출하는 것이다. 특정 공유되는 주소만이 transactional memory에서 violation이 일어나게 만드는 지점이기 때문에 이 부분만을 베리어 처리하여 transactional memory기법을 적용 했을 때 upper limit를 알아보자. 물론 특정 공유되는 주소에서만 베리어를 치기 위하여 simulated 되는 프로그램에서 자기가 공유하는 global 변수의 주소를 KLT에서 썼던 simulator에 system call 방식으로 등록하는 별도의 함수를 더 구현하였고 이 등록 하는 오버헤드는 transactional memory를 썼다면 자연적으로 없어야 할 파트에 속하기 때문에 시뮬레이션 시간에서는 제외시켰다.

5 실험

5.1 실험 환경

간단한 멀티 쓰레드 프로그램을 앞서 만든 kernel level thread library를 이용하여 arm-linux-gcc로 컴파일 한 것을 simulated program으로 사용하였다. 멀티 쓰레드 프로그램은 8개의 쓰레드로 구성되어있으며 이것은 멀티코어 시뮬레이터의 코어수가 8개임을 감안해서 만든 수치이다. 보통 멀티코어 개수 이상의 쓰레드는 만들지 않는다. 그리고 아래의 모든 simulated 프로그램이 global data에 access할 때는 동기화를 한 뒤 access를 한다.

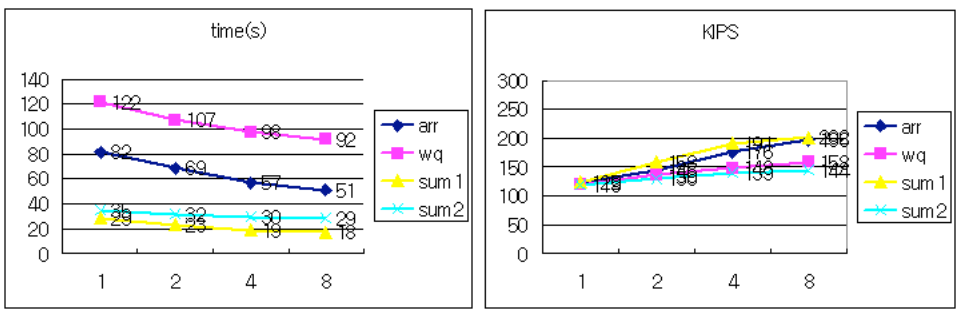
– array: global array변수를 8개의 쓰레드가 구역을 나누어서 array에 담겨져 있는 값을 local 변수에 더하는 프로그램이다. 마지막에 모든 쓰레드에서 더한 local 변수 값을 global변수에 합친다.
– work queue: global queue가 있고 8개의 쓰레드가 queue의 item을 하나씩 가져와서 item에 담겨져 있는 값을 local 변수에 더하는 프로그램이다. 마지막에 모든 쓰레드에서 더한 local 변수 값을 한 global 변수에 합친다.
– sum1: local 변수에 반복해서 특정 데이터를 더하고 마지막에 모든 쓰레드에서 더한 local 변수 값을 한 global 변수에 합치는 프로그램이다.
– sum2: global 변수에 반복해서 특정 데이터를 더하는 프로그램이다.

Simulated program	feature
array	global array, sum to the local variable, last sync
work queue	global queue, sum to the local variable, last sync
sum1	local data, last sync
sum2	global data, always sync

[표 II]Experimentation Methodology, Simulated Program Configuration

simulator의 설정을 나타내는 표가 Table III 에 나와 있다
시뮬레이터가 돌아가는 host machine의 configuration이 Table IV 에 나와 있다. quad core 칩이 2개가 있으며 실험 시에는 simulated program의 thread가 8개, simulator의 코어가 8개로 제한되어 있는 상태에서 scalability가 보장되는 측면을 보여주기 위하여 코어를 1에서 8개 일 때까지 나누어 놓고 실험하였다.

5.2 실험 결과



[그림 10] Multi-threaded Multi-core Simple Scalar w/ barrier

Table V과 Figure 10는 barrier가 있는 버전의 시뮬레이터에서 simulated program을 돌렸을 때 나온 결과 수치를 표현한 것이다. host machine의 코어를 1, 2, 4, 8개 사용하였을 때 simulated 프로그램의 elapsed time과 그에 따른 KIPS(Kilo Instruction per Second)를 표시하고 있다. 그리고 improvement는 host core가 1개 일 때의 KIPS를 1.0으로 봤을 때 상대적인 수치를 나타내고 있다. 프로그램 별로 서로 많이 차이가 나는 elapsed time을 시뮬레이터 성능의 척도로 삼기에는 무리가 있다. 실제로 Table V을 보면 sum1의 경우 8개 core에서 시뮬레이션 했을 때가 1개의 core에서 시뮬레이션 했을 때 보다 11초 빨라졌음을 알 수

[표 III]Experimentation Methodology, Simulator Configuration

Simulator (gcc-3.4, TLS, pthread lib)	Value
multi-core count	8
instruction fetch queue size (in insts)	4
instruction decode B/W (insts/cycle)	4
instruction issue B/W (insts/cycle)	4
instruction commit B/W (insts/cycle)	4
branch predictor type	bimod
bimodal predictor config (table size)	2048
BTB config (numsets, associativity)	512 4
speculative predictors update (default non-spec)	non-spec
extra branch mis-prediction latency	no speculation across basic block
run pipeline with in-order issue	flase
issue instructions down wrong execution paths	false
register update unit (RUU) size	16
load/store queue (LSQ) size	8
l1 data cache config	dl1:128:128:2:l
l1 data cache hit latency (in cycles)	1
l2 data cache config	ul2:16384:128:4:l
l2 data cache hit latency (in cycles)	9
l1 inst cache config	il1:256:128:2:l
l1 inst cache hit latency (in cycles)	1
flush caches on system calls	false
memory access latency (first chunk, inter chunk)	18 2
memory access bus width (in bytes)	8
inst TLB config	itlb:16:4096:4:l
data TLB config	dtlb:32:4096:4:l
inst/data TLB miss latency (in cycles)	30
total number of integer ALU's available	4
total number of integer multiplier/dividers available	1
total number of memory system ports available	2

[표 IV]Experimentation Methodology, Host Machine Configuration

Host machine configuration
2×x Intel zeon quad core, 2.33GHz
8M L2 unified cache (16-way, Write Back)
128K L1 data cache (8-way, Write Back)

있다. 그에 비해 work queue는 30초나 빨라졌다. work queue에서 더 시뮬레이터의 성능이 좋아 진 것으로 알 수 있으나 시간이 줄어든 비율을 보면 sum1의 경우에는 0.62가 나왔으나 work queue의 경우에는 0.75가 나와 실제로는 sum1이 성능향상이 더 큰 것을 볼 수 있다. 이러한 이유 때문에 비교를 위한 KIPS같은 normalized된 수치가 필요하다. KIPS는 얼마나 시뮬레이터가 단위 시간당 많은 수의 인스트럭션을 처리했는가를 표현해준다. 즉, 이 값이 클수록 시뮬레이터의 성능은 좋다고 할 수 있다.

Table V에서는 1개 코어를 사용하였을 때보다 8개 코어를 사용하였을 때 전체적으로 44% 의 향상을 보여주고 있다. array같은 경우에는 global 변수를 access하긴 하지만 구역을 정해서 access하기 때문에 동기화가 필요 없고 sum1 같은 경우에는 로컬 변수만을 주로 access하기 때문에 동기화가 필요 없다. 그렇기 때문에 매번 동기화를 해야 되는 work

[표 V]Multi-threaded Multi-core Simple Scalar w/ barrier

	host core	1	2	3	4
array	time(s)	82	69	57	51
	KIPS	122	145	176	196
	improvement	1.000000000	1.18852459	1.442622951	1.606557377
work queue	time(s)	122	107	98	92
	KIPS	119	136	148	158
	improvement	1.000000000	1.142857143	1.243697479	1.327731092
sum1	time(s)	29	23	19	18
	KIPS	125	158	191	202
	improvement	1.000000000	1.264	1.528	1.616
sum2	time(s)	35	32	30	29
	KIPS	119	130	139	144
	improvement	1.000000000	1.092436975	1.168067227	1.210084034
	avg. KIPS	121.25	142.25	163.5	175
	avg. improvement	1.000000000	1.173195876	1.348453608	1.443298969

queue나 sum2 보다 좀 더 성능이 좋게 나온다. sum2 같은 경우에는 매우 빈번하게 동기화를 진행하기 때문에 21% 의 향상 밖에 보질 못했다. Figure 10의 KIPS 그래프를 보면 전체적으로 증가치가 둔화 되는 것을 볼 수 있다. 이 같은 경우에는 더 많은 host core를 달아도 증가치가 제자리를 머물거나 오히려 감소할 수도 있다고 생각 된다. scalability 보장 측면에서 보면 그래프가 직선 형태를 가지게 되는 것이 바람직하나 많은 동기화 오버헤드로 인하여 이것이 완벽하게는 달성 되지 못하고 있는 것이다.

이런 문제점을 해결할 방안으로는 4.3절에 transactional memory라는 것을 제안 하였다. 우선 transactional memory기법이외에 어떤 기법을 사용하던 최대 upper limit를 알고 싶어 베리어를 완전 제거 했을 때 결과 수치를 테이블과 그림으로 표현하였다. Table VI과 Figure 11에 잘 표현되어 있으며 이 결과를 보면 전체적인 성능 향상이 71% 정도에 해당한다. 모든 simulated program에서 성능 향상을 볼 수 있었다. 특히 그래프를 보면 어느 정도 linear하게 증가하고 있음을 알 수 있는데 이것은 scalability 측면에서 더 많은 host core를 달았을 때 더 큰 성능 향상이 보장된다는 것을 암시하고 있다. 그러나 여전히 simulated program이 성능 향상 정도가 array, sum1과 work queue, sum2의 두 그룹으로 분리 된다는 것을 볼 수 있는데 이것은 simulated program에 내제되어 있는 동기화 오버헤드를 따른 수치이기 때문이다.

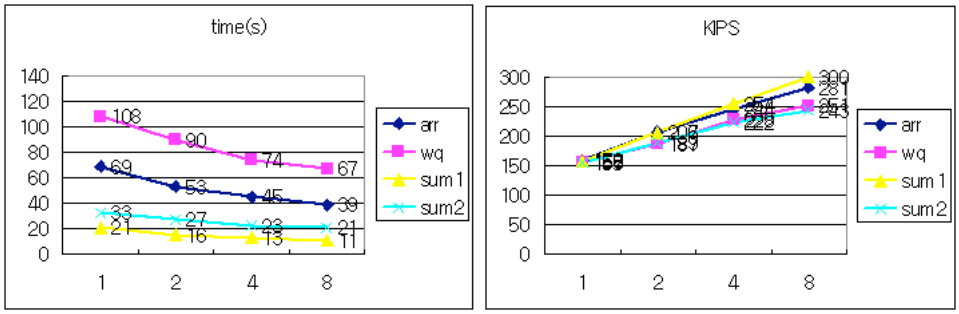
그에 이어 transactional memory의 upper limit를 특정 공유 변수에 대한 베리어로 연출했을 때 결과 수치가 Table VII과 Figure 12이다. 전체적인 성능 향상이 56% 정도에 해당하며 더 중요한 것은 그래프가 베리어를 완전 제거했을 때만큼은 아니지만 어느 정도 linear하다는 것이다. 즉, scalability 측면에서 더 많은 코어를 달아도 성능이 좋아짐을 예상하게 한다. 물론 실제 transactional memory를 구현한 것을 사용하면 이것보다는 수치가 떨어지겠지만 그래도 transactional memory 기법의 적용시 성능 향상의 희망을 보여 주고 있다.

6 결론 및 향후 연구 과제

현재의 processor기술은 발열, 전력 문제로 인해서 더 이상 clock수를 높이는 쪽으로 진행되지 않고 칩 안에 들어가는 core의 수를 늘리는 쪽으로 진행 되고 있다. 지금까지 sequential한 프로그램 패러다임에 익숙한 프로그래머는 이런 parallel한 환경이 낯설고 당황스러울 텐데, 그런 문제를 풀기 위해 parallel 컴파일러의 도움이 절실시 되고 있다. parallel 컴파일러의 개발에 큰 도움이 될 수 있는 것이 아키텍처 시뮬레이터이다.

[표 VI]Multi-threaded Multi-core Simple Scalar w/o barrier

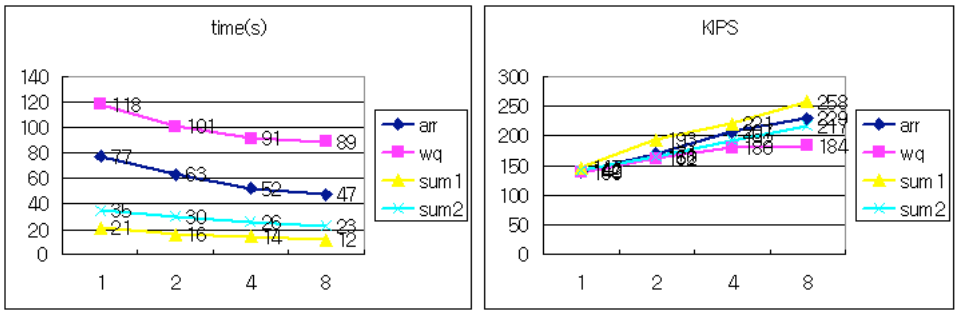
	host core	1	2	3	4
array	time(s)	69	53	45	39
	KIPS	159	207	244	281
	improvement	1.000000000	1.301886792	1.534591195	1.767295597
work queue	time(s)	108	90	74	67
	KIPS	156	187	228	251
	improvement	1.000000000	1.198717949	1.461538462	1.608974359
sum1	time(s)	21	16	13	11
	KIPS	157	206	254	300
	improvement	1.000000000	1.312101911	1.617834395	1.910828025
sum2	time(s)	33	27	23	21
	KIPS	155	189	222	243
	improvement	1.000000000	1.219354839	1.432258065	1.567741935
	avg. KIPS	156.75	197.25	237	268.75
	avg. improvement	1.000000000	1.258373206	1.511961722	1.714513557



[그림 11] Multi-threaded Multi-core Simple Scalar w/o barrier

[표 VII]Multi-threaded Multi-core Simple Scalar w/ barrier on shared variable

	host core	1	2	3	4
array	time(s)	77	63	52	47
	KIPS	140	171	207	229
	improvement	1.000000000	1.221428571	1.478571429	1.635714286
work queue	time(s)	118	101	91	89
	KIPS	139	162	180	184
	improvement	1.000000000	1.165467626	1.294964029	1.323741007
sum1	time(s)	21	16	14	12
	KIPS	147	193	221	258
	improvement	1.000000000	1.31292517	1.503401361	1.755102041
sum2	time(s)	35	30	26	23
	KIPS	142	166	192	217
	improvement	1.000000000	1.169014085	1.352112676	1.528169014
	avg. KIPS	142	173	200	222
	avg. improvement	1.000000000	1.218309859	1.408450704	1.563380282



[그림 12] Multi-threaded Multi-core Simple Scalar w/ barrier on shared variable

아키텍처 시뮬레이션은 해당 아키텍처가 존재하지 않거나 아키텍처 파라미터 값을 바꾸었을 때 어떤 결과가 생기는지를 알아보는데 아주 유용한 기법이다. 현재 제공되고 있는 멀티코어 아키텍처 시뮬레이터는 하나의 글로벌 타이머를 두고 모든 프로세서에서 이 타이머에 대해 이벤트를 발생시키고 타이머 순서대로 처리하는 방식으로 동기화를 맞추어 준다. 하지만 이 방식은 하나의 프로세서에서 모든 멀티코어를 시뮬레이션 해야 된다는 단점이 있다. 따라서 기존 방식 대신에 각 코어를 하나의 쓰레드에 할당시켜 여러 개의 쓰레드에서 동시에 코어를 시뮬레이션 함으로써 성능을 개선시키는 연구를 본 논문에서 수행하였다. 또한 각 코어 간에 데이터 공유가 발생할 시 서로간의 동기화를 맞춰주어야 하는 기법을 구현하였으며, 실험 환경을 구성하기 위하여 simulated program을 멀티 쓰레드 프로그램화 시켜주는 간단한 Kernel Level Thread library를 구현하였다. 이 모든 것을 simple scalar라는 아키텍처 시뮬레이터 상에서 처리하였다.

실험 결과를 분석하면 1개 코어를 사용하였을 때보다 8개 코어를 사용하였을 때 전체적으로 44% 의 향상을 보여주고 있다. scalability 보장 측면에서 더 시도할 기법으로 transactional memory 기법을 선정하였으며 이것의 upper limit를 측정해 본 결과, 전체적인 성능 향상이 56% 정도에 해당하였고 그래프 증가치가 좀더 linear해 지는 것을 볼 수 있었다.

참고문헌

[1] 이광용, 박호준, 김동한, 강동욱, 김재명, 박승민 *멀티코어 기술 및 산업 동향* 주간기술동향 **1295** (2007).

[2] D. Burger, T. Austin, and S. Bennett *Evaluating Future Microprocessors: the SimpleScalar Tool Set* Technical Report 1308, Computer Sciences Department, University of Wisconsin (1996).

[3] Todd M. Autsin *SimpleScalar Hacker's Guide* (1997).

[4] *TLBs, Paging-Structured Caches, and Their Invalidation* Application Note (2007).

[5] Youil Kim, Hwansoo Han, Jaeho Lee, Sunja Kim *Developing WIPI Runtime Engine for VM-less Mobile Terminals* (2003).

[6] David Seal *ARM Architecture ReferenceManual* Addison-Wesley (2000).

[7] John Haskins, Jr. *Light-weight Thread Library for Linux* http://www.cs.virginia.edu/~jwh6q/lwt-web

[8] James Donald, Margaret Martonosi *An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation* IEEE Computer Society (2006).

[9] K. Barr et al. *Simulating a Chip Multiprocessor with a Symmetric Multiprocessor* Proc. of the Boston Area Architecture Workshop (2005).

[10] M. Chidester and A. George *Parallel Simulation of Chip-Multiprocessor Architectures* ACM Transactions on Modeling and Computer Simulation vol. 12, no. 3**176?200** (2002).

[11] D. Nicol and R. Fujimoto *Parallel Simulation Today* Annals of Operations Research no. 53 **249?285** (1994).

[12] D. Penry et al. *Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multiprocessors* Proc. of the 12th Intl. Symp. on High-Performance Computer Architecture (2006).

[13] *Thread-local Storage* http://en.wikipedia.org/wiki/Thread_Local_Storage (2006).

[14] L. Barroso et al. *A scalable architecture based on single-chip multiprocessing* Proc. of the 27th Intl. Symp. on Computer Architecture (2000).

[15] K. Chandy and J. Misra. *Distributed simulation: A case study in design and verification of distributed programs* IEEE Transactions on Software Engineering, 5(5) **440-452** (1979).

[16] R. DeVries *Reducing null messages in Misra's distributed discrete event simulation method* IEEE Transactions on Software Engineering, 16(1) **82-91** (1990).

[17] M. Durbhakula, V. Pai, and S. Adve *Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors* Proc. of the 5th Intl. Symp. on High-Performance Computer Architecture (1999).

[18] B. Falsafi and D. Wood *Modeling cost/performance of a parallel computer simulator* ACM Transactions on Modeling and Computer Simulation, 7(1) **104-130** (1997).

[19] R. Fujimoto *Parallel and Distributed Simulation Systems* John Wiley & Sons, Inc. (2000).

[20] A. George, R. Fogarty, J. Markwell, and M. Miars *An Integrated Simulation Environment for parallel and distributed system prototyping* Simulation, 75(5) **283-294** (1999).

[21] L. Hammond, B. Nayfe, and K. Olukotun *A single-chip multiprocessor* IEEE Computer, 30(9) **79-85** (1997).

[22] J. Hennessy and D. Patterson *Computer Architecture: A Quantitative Approach* Morgan Kaufmann (1996).

[23] Kent Milfeld et al. *Effective Use of Multi-Core Commodity Systems in HPC* (2006).

[24] Robert Ennals *Cache Sensitive Software Transactional Memory* Intel Research Cambridge (2006).

박 현 식

– 2001.3-2006.2 아주대학교 미디어학부 (B.S.)
– 2006.3-2008.2 한국과학기술원 전산학과 (M.S.)

<관심분야> 병렬 컴퓨팅, 병렬 아키텍쳐 및 시뮬레이션

한 환 수

– 1993 서울대학교 컴퓨터 공학과 (B.S.)
– 1995 서울대학교 컴퓨터 공학과 (M.S.)
– 2001 Computer Science, University of Maryland (Ph.D.)
– 2001.3-2002.12 Sr. Engineer, Intel Architecture Group, Intel
– 2003.1-2007.8 한국과학기술원 전산학과 조교수
– 2007.9-present 한국과학기술원 전산학과 부교수

<관심분야> 최적화 컴파일러, 병렬 컴퓨팅, 임베디드 컴퓨팅, Trustworthy 컴퓨팅