POSTECH
포항공과대학교

성능버그가 없는 시스템 개발 및 운영기법 @ POSTECH

김장우
(and 김영석, 김지훈, 노한얼, 장한휘, 채동주)

2012년 1월 18일

포항공대 고성능 컴퓨팅 연구실

POSTECH

성공적인 (또는 이상적인) 컴퓨팅

"목표로 하는 일을
소프트웨어와 하드웨어가 서로 협력하여
(1) 최대한 빨리
(2) 최소의 비용으로
(3) 오류 없이
실행하는 것"

우리가 하고자 하는 것

"시스템 개발 및 운영을 위한

성능 모델링, 분석 및 예측 기법 제시"

이 기본 목표의 난이도는?
어렵다면 얼마나 어려울까?

성능 버그 (performance bug)?

= 시스템이 목표로 했던 성능을 얻어내지 못하는 것

성능버그가 생기는 대표적인 이유들:

- 잘못된 시뮬레이터에 의존한 개발
- 잘못된 워크로드에 의존한 개발
- 개발된 시스템의 잘못된 운영
- …

기능 버그 (Func. Bug) VS 성능 버그 (Perf. Bug)

- 시스템 개발자 (=시뮬레이터 개발자)가 걱정하는 오류

- 실행 버그 : 개발된 시스템이 예상대로의 기능을 수행할까?

  CPU 시뮬레이터: "CPU가 정해진 기능대로 동작하고 있는가?"
  오류의 예: 잘못된 계산 수행. 잘못된 메모리 접근.

- 성능 버그 : 개발된 시스템이 예상대로의 성능을 보여줄까?

  CPU 시뮬레이터: "CPU가 예측한 성능대로 동작하고 있는가?"
  오류의 예: 잘못된 분기명령어 예측확률, 잘못된 타이밍 구현

기능 버그 (Func. Bug) VS 성능 버그 (Perf. Bug)

- 실행버그는 'Golden Model (=Oracle)'과 비교해 쉽게 해결.

예) 이미 검증 된 명령어 에뮬레이션의 모델 결과 값(=kind of Oracle)

VS

새로 개발하고 있는 시스템의 시뮬레이터 결과 값

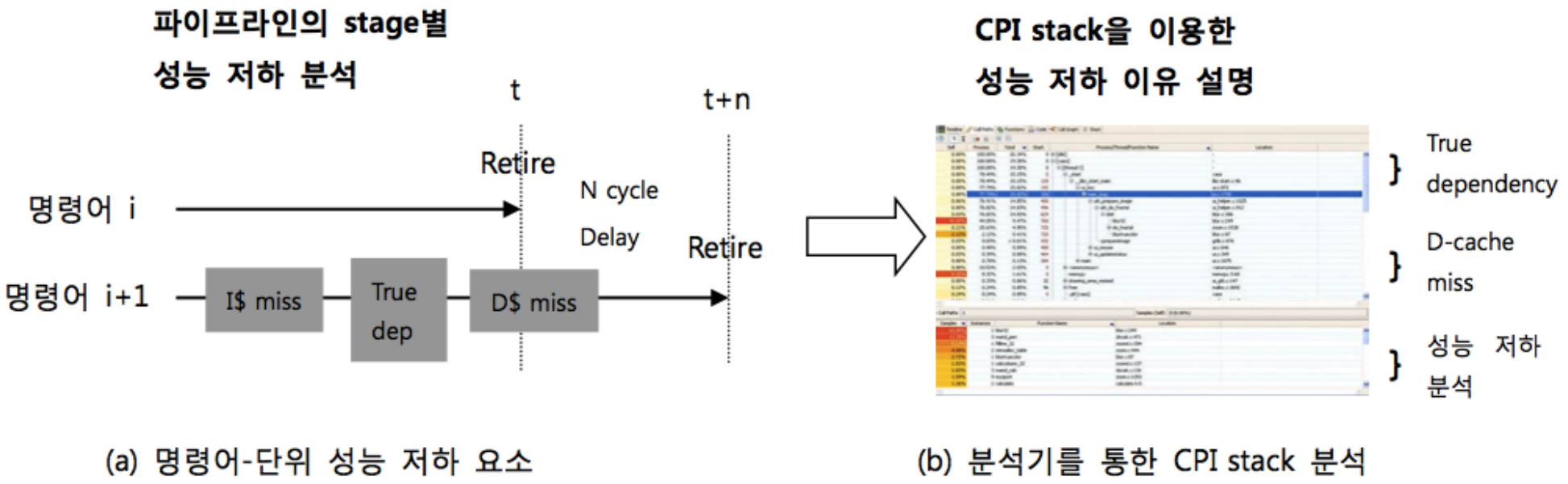
- 그러나 성능버그는 "제품 개발 완료" 까지는 평가하기 어려움.

비교할 대상이 없음

What we do @ POSTECH: #1

성능버그 최소화를 위한 CPU 시뮬레이션 기법

타이밍 트레이스를 이용한 시스템 성능 분석기 (1/3)



(a) 명령어-단위 성능 저하 요소

(b) 분석기를 통한 CPI stack 분석

타이밍 트레이스: 시뮬레이션을 통해 트레이스 (e.g., 명령어)가 시스템 (예: CPU) 에서 수행되어간 "시간요소"를 기록한 것

타이밍 트레이스를 이용한 시스템 성능 분석기 (2/3)

예: 3개의 연속된 명령어로 이뤄진 타이밍 트레이스

	afetch	fetch	dispatch	ready	issue	complete	commit	info[3]	dep[3]
inst i-1	104	104	111	120	120	121	125	0 0 0	2 30 30
inst i	104	113	120	121	121	122	126	1 0 2	30 0 0
inst i+1	124	374	381	381	383	385	389	5 0 0	0 2 29

'시스템 구조'와 이 '실행 정보'를 조합하면

(1) 어떤 요소가 성능저하를 일으킨 요소인지 찾을 수 있고

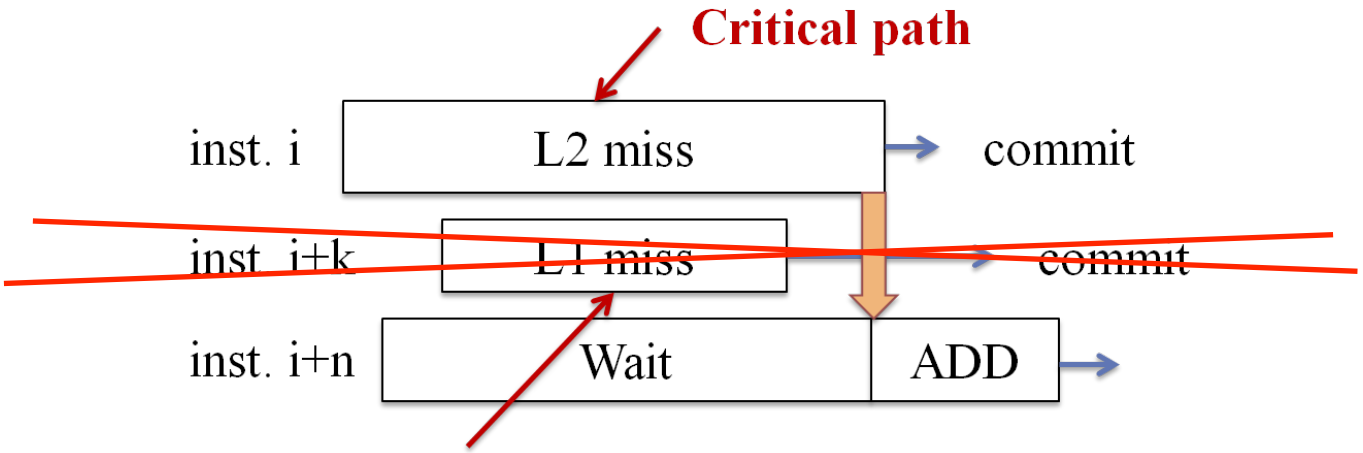
(2) 그 요소를 없앨 경우 **예측**되는 성능향상까지 가능해짐

타이밍 트레이스를 이용한 시스템 성능 분석기 (3/3)

'시스템구조'에 대한 지식이 있다면...

특정 명령어가 시스템 성능에 기여하는 정도 측정 가능

→ 성능 기여도 (performance criticality)

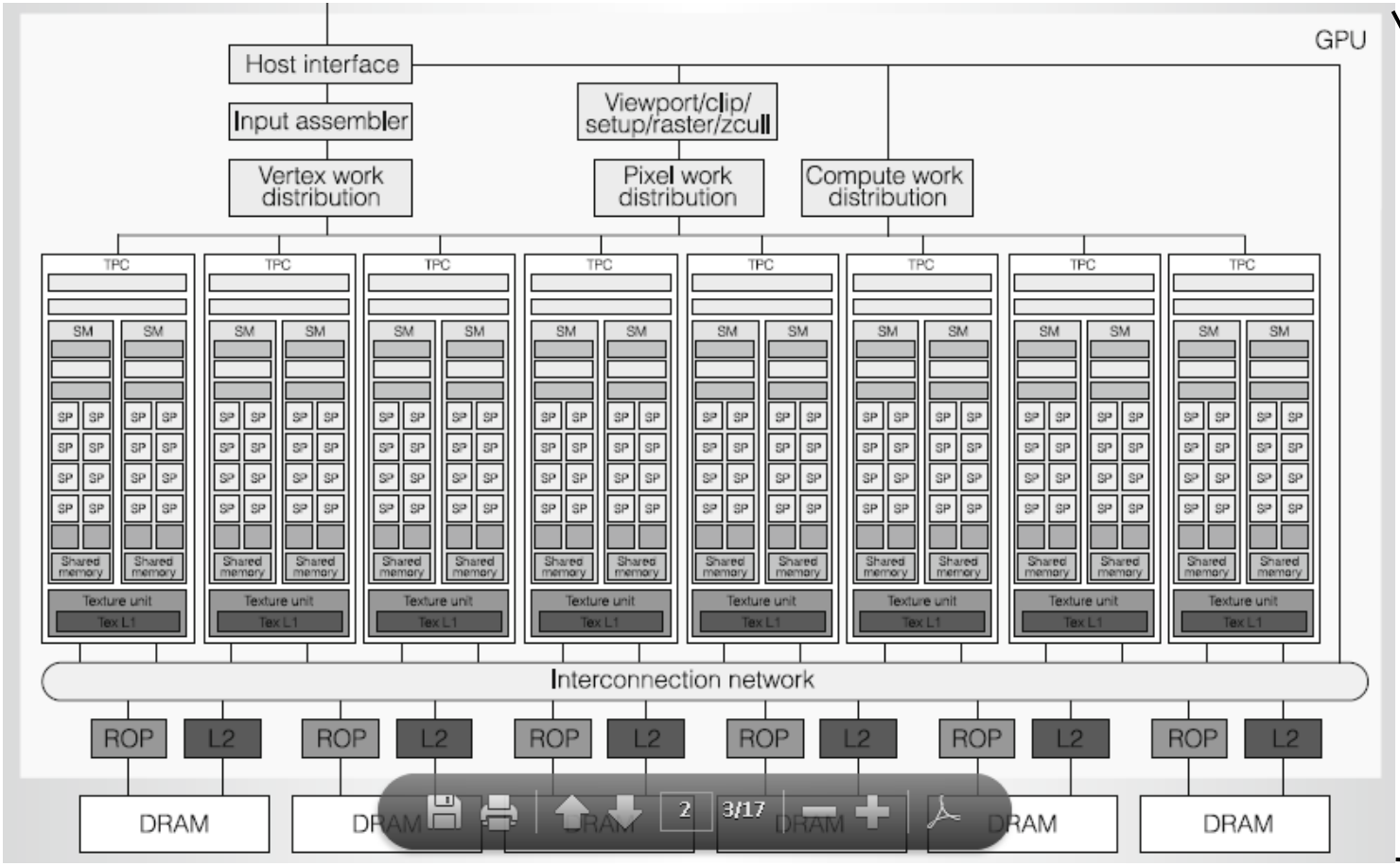


성능기여도가 없으면 무시해도 좋음

What we do @ POSTECH: #2

성능 버그 없는 GPU 프로그래밍 환경

GP-GPU (General-Purpose Graphic Procession Unit)



수백 개의 코어를 가진 GPU를 일반 병렬계산 목적으로 사용하는 것

프로그램 → 컴파일 → GPU → 프로그램 …

CUDA (Compute Unified Device Architecture)

```
// addArray: Adds two vectors with n elements
__global__ void addArray(int n, int *a, int *b, int *c) {
    __shared__ int shr_a[32], shr_b[32];
    int idx;

    idx = blockIdx.x * blockDim

    shr_a[idx] = a[idx];
    shr_b[idx] = b[idx];
    __syncthreads();

    c[idx] = shr_a[idx]   shr_b   ];
}
```

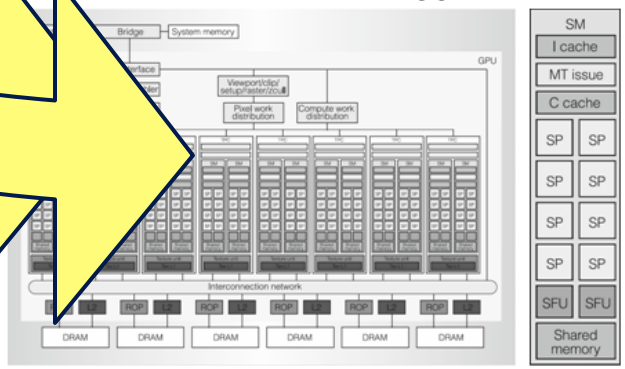
Maybe NOT

만족스러운 성능?

Perform c = a + b

컴파일

PTX ISA



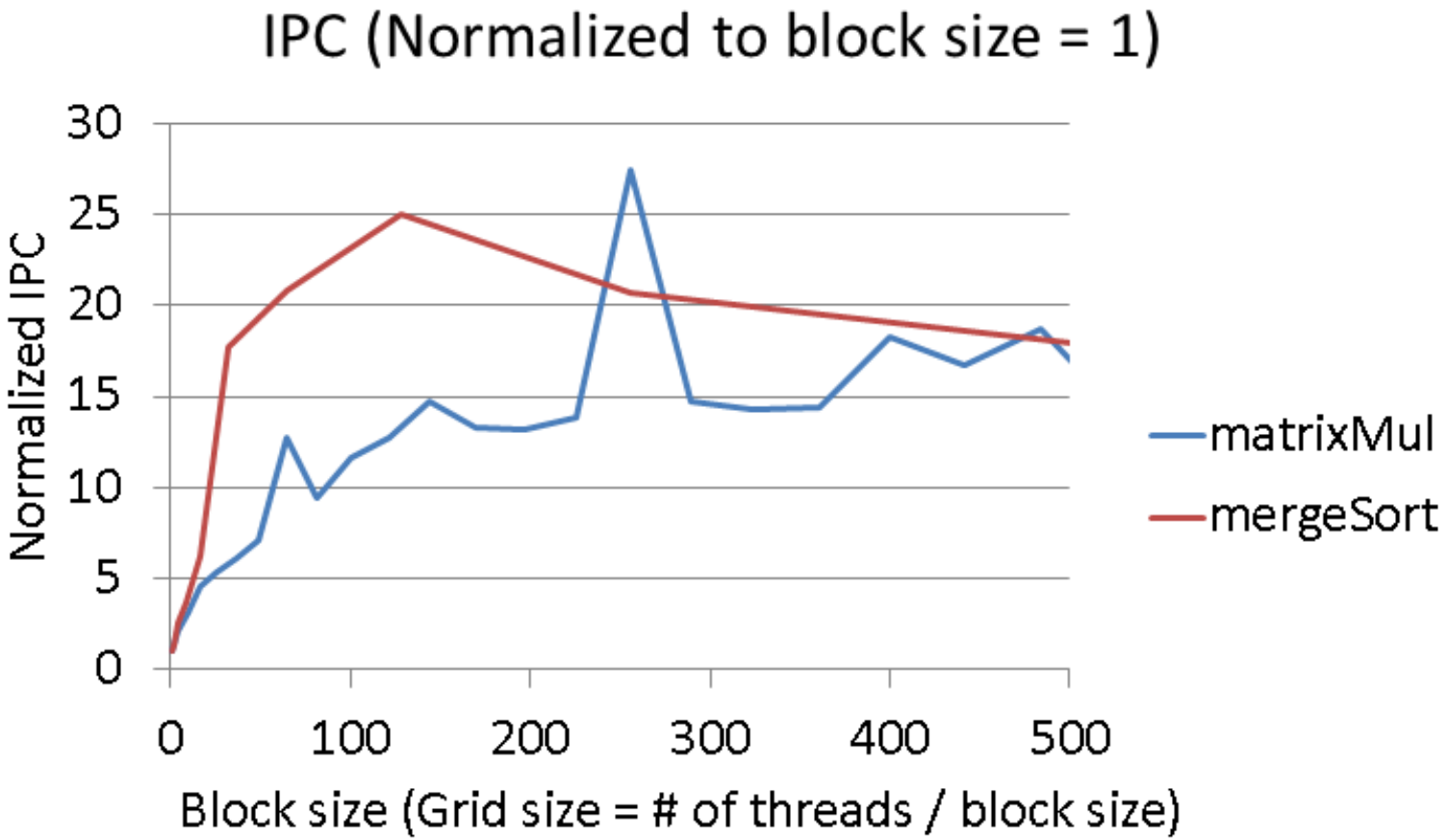
기존의 GPU 프로그래밍 환경의 문제

- **프로그래머**
  - GPU 구조에 대해 너무나 많은 것을 알아야 함

    (예: 코어 수, thread-core 할당 형태, 개인/공유 메모리, 네트워크 등)
  - 무수히 많은 trial-and-error 방식의 성능 최적화
  - 가상머신 위에선 기본적 성능평가 자체 불가능

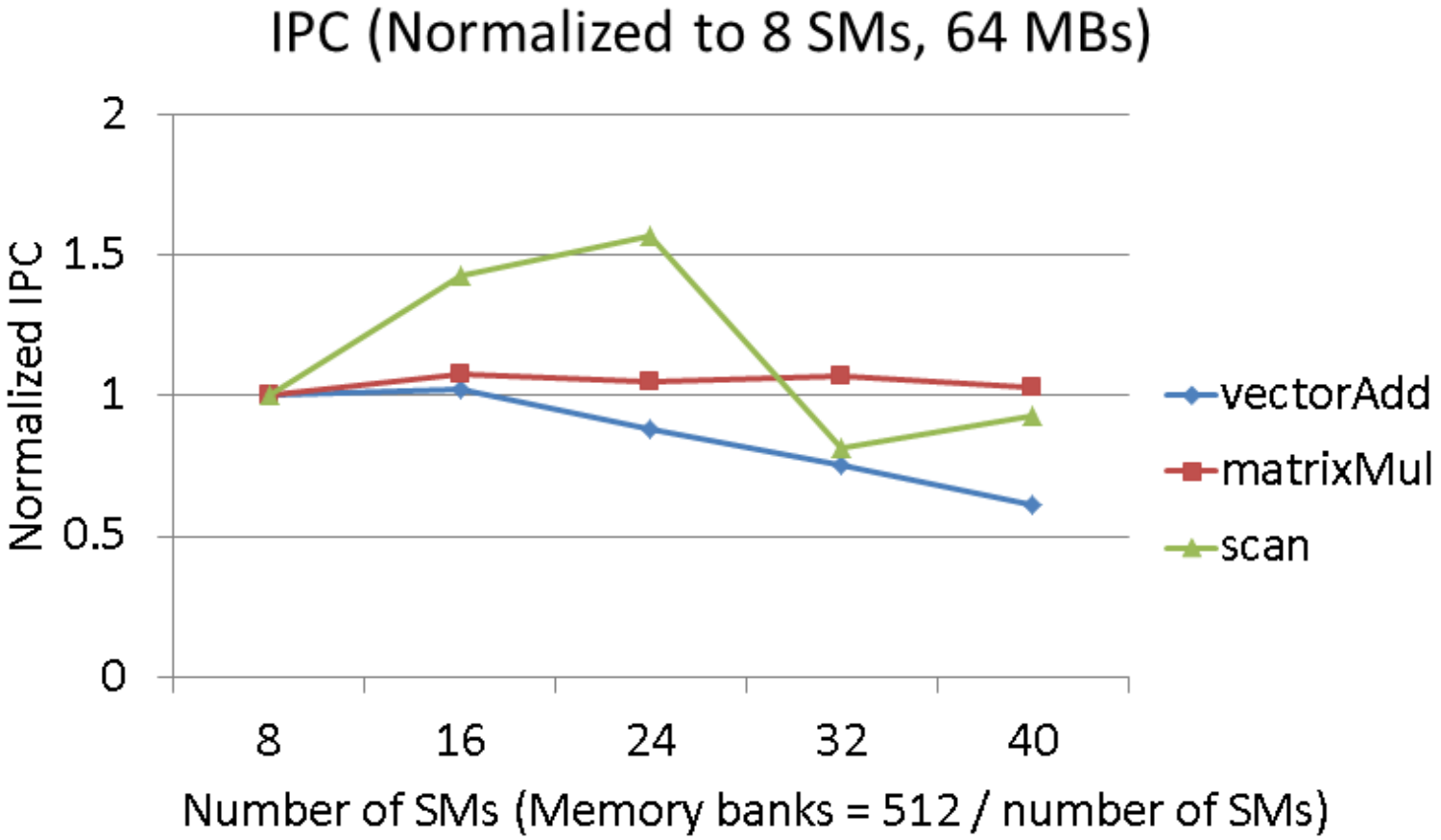
- **GPU 구조**
  - 플랫폼이 바뀌면 성능 최적화도 다시 시행되어야 함
  - 원본 코드접근이 불가능하면 재최적화 자체가 불가능

프로그래머의 GPU 할당형태 조절에 따른 성능변화



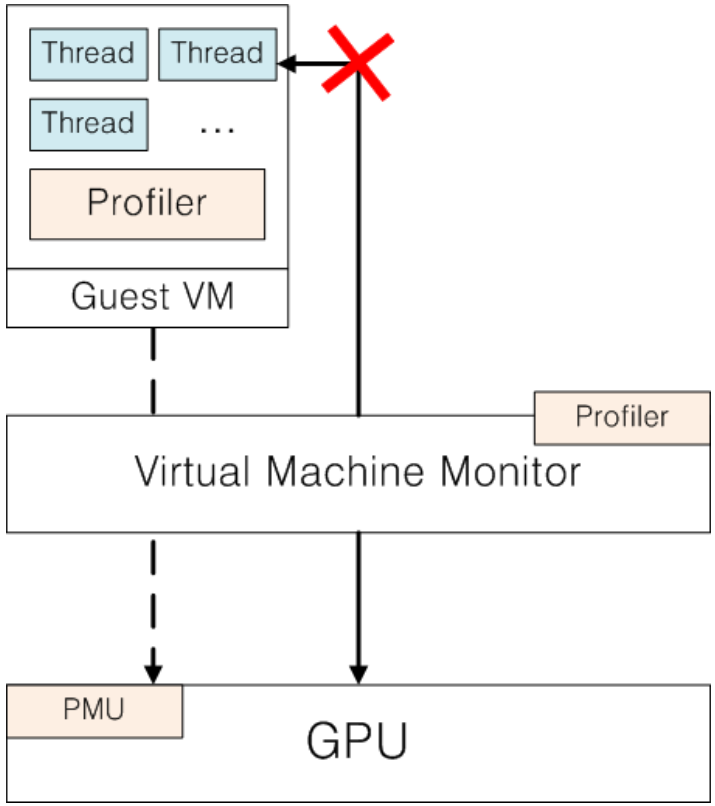
GPU로의 할당 형태에 따라 성능이 크게 변해서 "최적점"을 찾기 어려움

GPU 구조에 따른 성능변화 (예: GPU 코어 수 & 메모리 크기)



GPU 구조에 따라 성능이 다양하게 변하므로 "최적점"을 찾기 어려움

가상머신 환경에서 GPU 성능측정 어려움



게스트머신: 성능측정 유닛 (PMU) 접근이 어려움

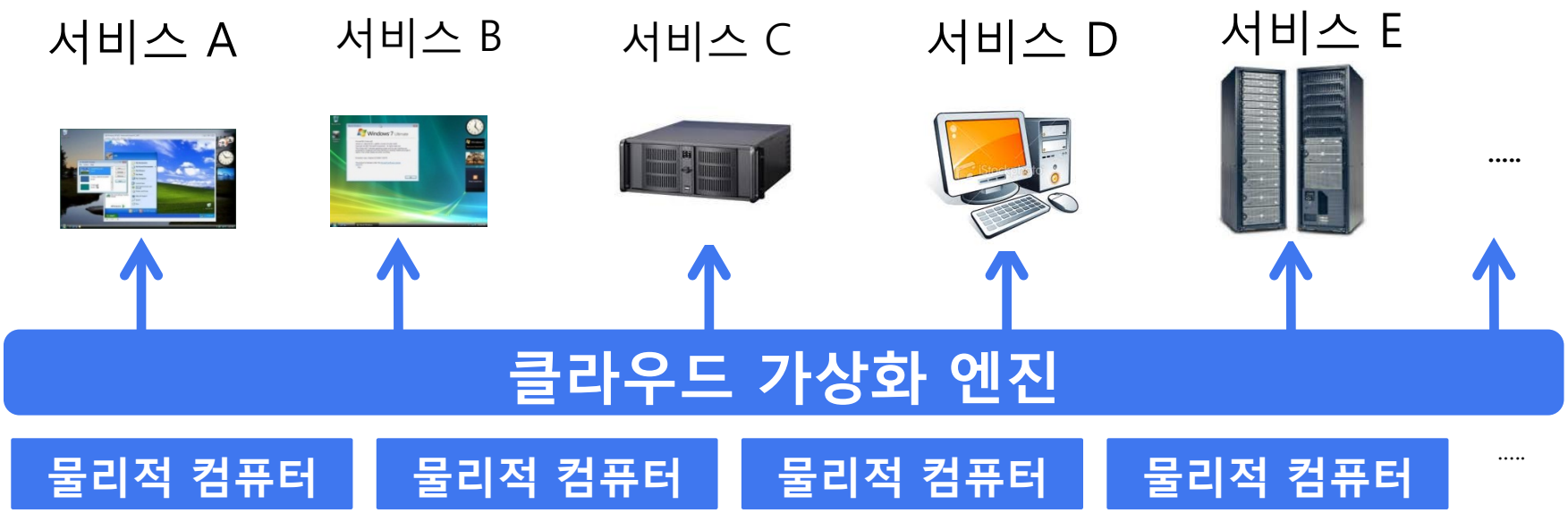
호스트머신: 게스트 머신의 내부구조를 보기 어려움

복수의 게스트머신를 위한 PMU 멀티플렉싱 필요

What we do @ POSTECH: #3

고성능 클라우드 컴퓨팅 운영기법

인프라스트럭쳐 클라우드 (IaaS) 컴퓨팅



데이터센터 @ POSTECH

가상화된 시스템의 사용자는 자신의 시스템을 온전한 독립 시스템으로 간주함

클라우드 컴퓨팅의 경쟁력을 결정하는 것?

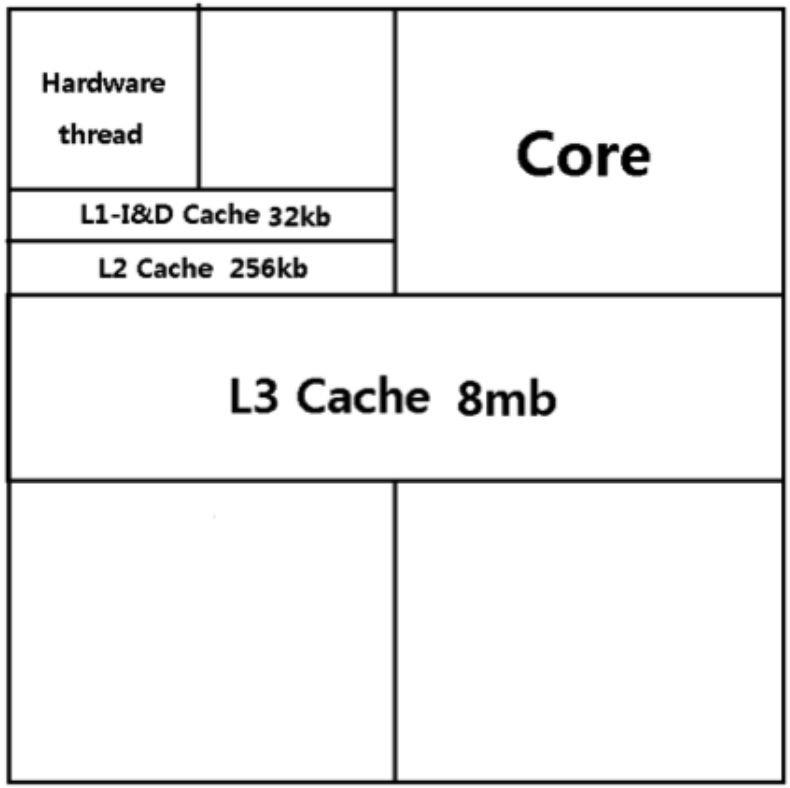
(1) 데이터센터 가상화를 통해

(2) 소프트웨어/하드웨어를 서비스로 제공하고

(3)“실시간 관리”를 통해

(4)“서비스의 가격대 성능비, 안정성”을 극대화시킴

클라우드 환경에서 성능저하 및 성능평가 어려움



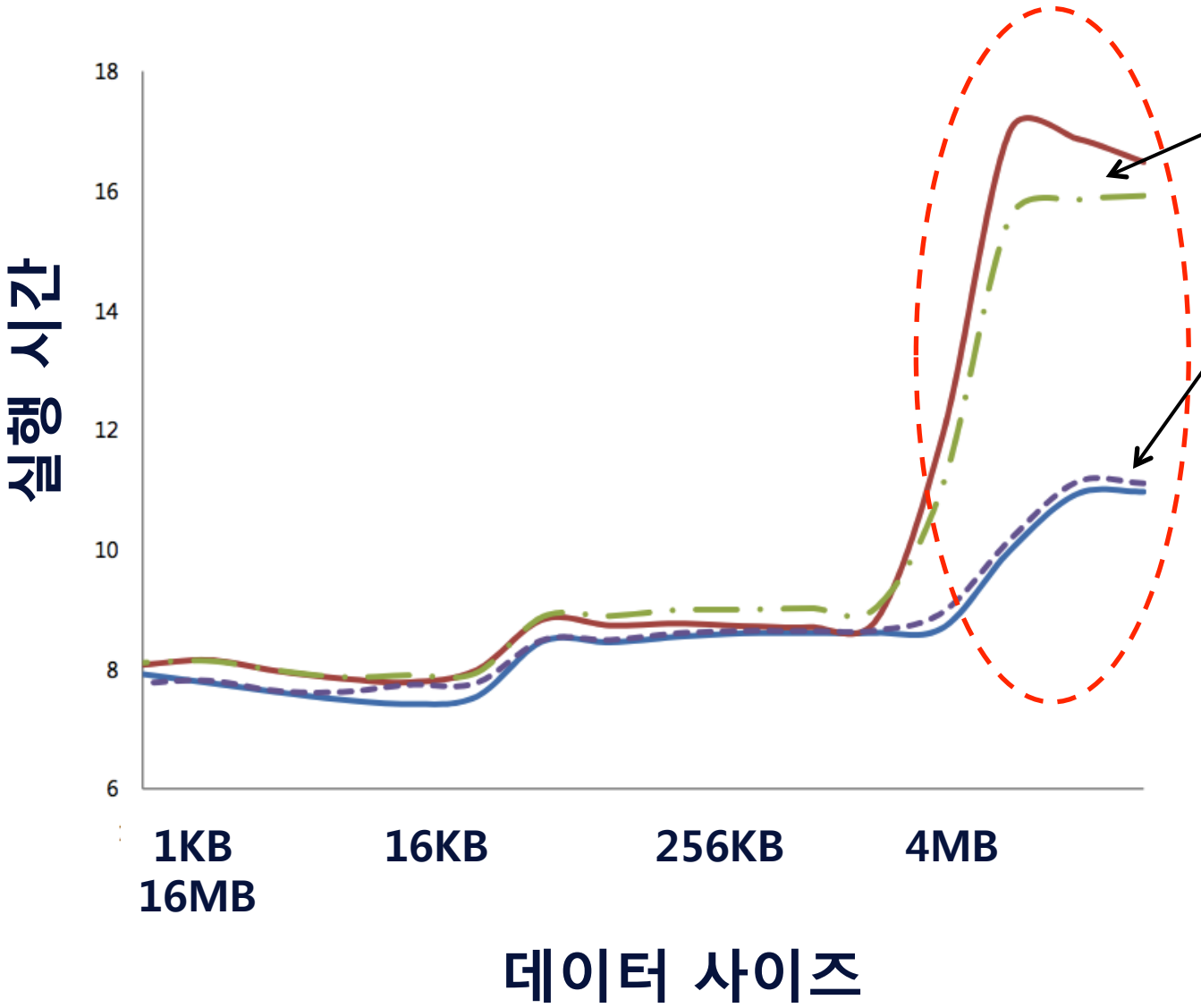
가상화된 멀티코어/멀티노드 시스템은 진정한 "멀티"가 아님

예) 캐쉬, 네트워크, 하이퍼쓰레딩

가상화된 시스템은 기존의 성능분석 툴을 사용하거나 신뢰할 수 없음

예) 시스템 프로파일링 서포트

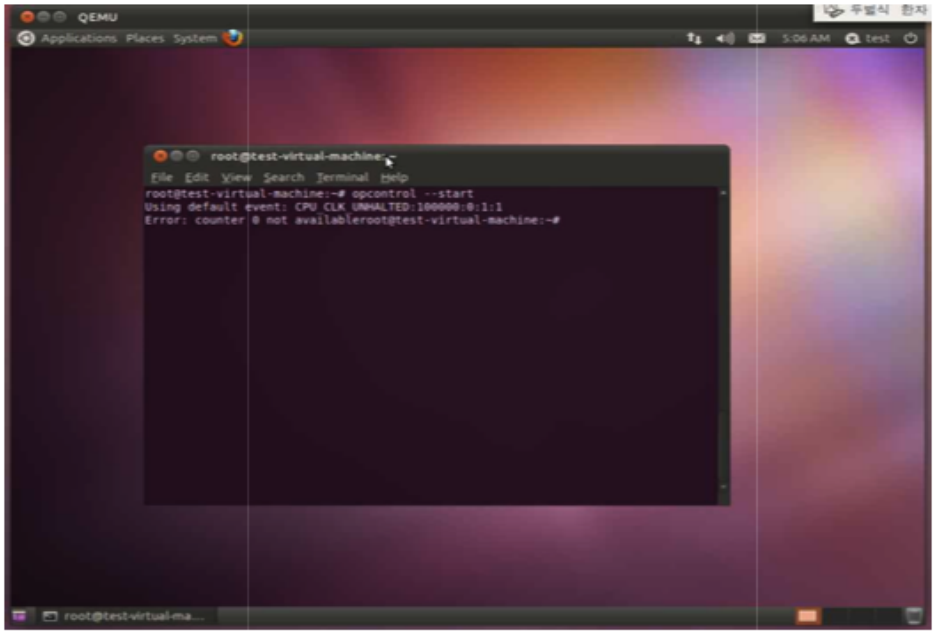
성능저하: 2개 프로그램을 2개 코어에 돌리면..



사용자의 머릿속의
시스템 != 실제로
부여 받은 시스템

심각한 QoS 위협

성능평가: 가상화 시스템은 성능분석이 어려움

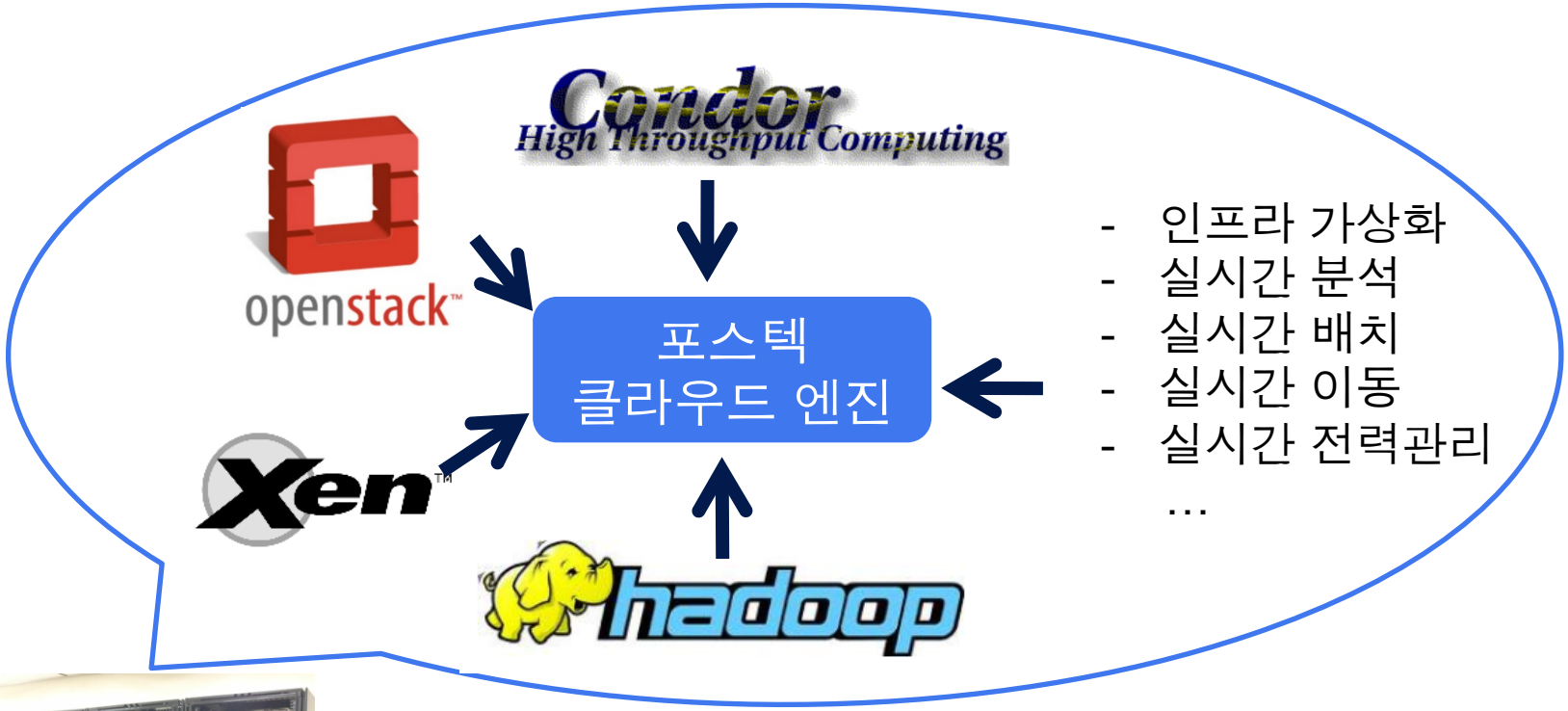


samples	%	
45249	98.6892	kvm_intel
364	0.7939	vmlinux
36	0.0785	r600_dri.so
22	0.0480	kvm

게스트에서는 호스트의 퍼포먼스 카운터 접근 불가

호스트에서는 게스트 시스템이 큰 프로세스 하나로 보임

POSTECH 클라우드 엔진: 공개소프트웨어 기반



POSTECH 데이터센터 **(100+** 노드**)**

Tools	Function
OpenStack	공개 클라우드 엔진
Condor	공개 워크로드 분산시스템
Hadoop	공개 파일 분산시스템
Xen/KVM	공개 가상화 플랫폼

대표적 적용기술: (1) 클라우드 성능평가 기술

- **노드레벨: 게스트머신과 호스트머신 협력기법**
  - 게스트머신 : 호스트머신에게 자신의 정보 제공
    (메모리 이미지, 프로세스 ID 등)
  - 호스트머신 : 게스트에게 성능평가 유닛 (PMU) 접근 허용
    복수의 게스트를 구분하기 위한 멀티플렉싱 허용

- **시스템레벨: 자원충돌 모니터링**
  - 위에서 개발된 기술을 이용하여 가상환경내 자원충돌 확인
  - 필요에 따른 자원재할당 시행 (예: 가상머신 이동)

대표적 적용기술: (2) 가상머신 고속이동기술

- **이동이 필요한 이유**
  - 사용자에게 약속한 성능 보장/ 자원 효율성 및 안정성 증대

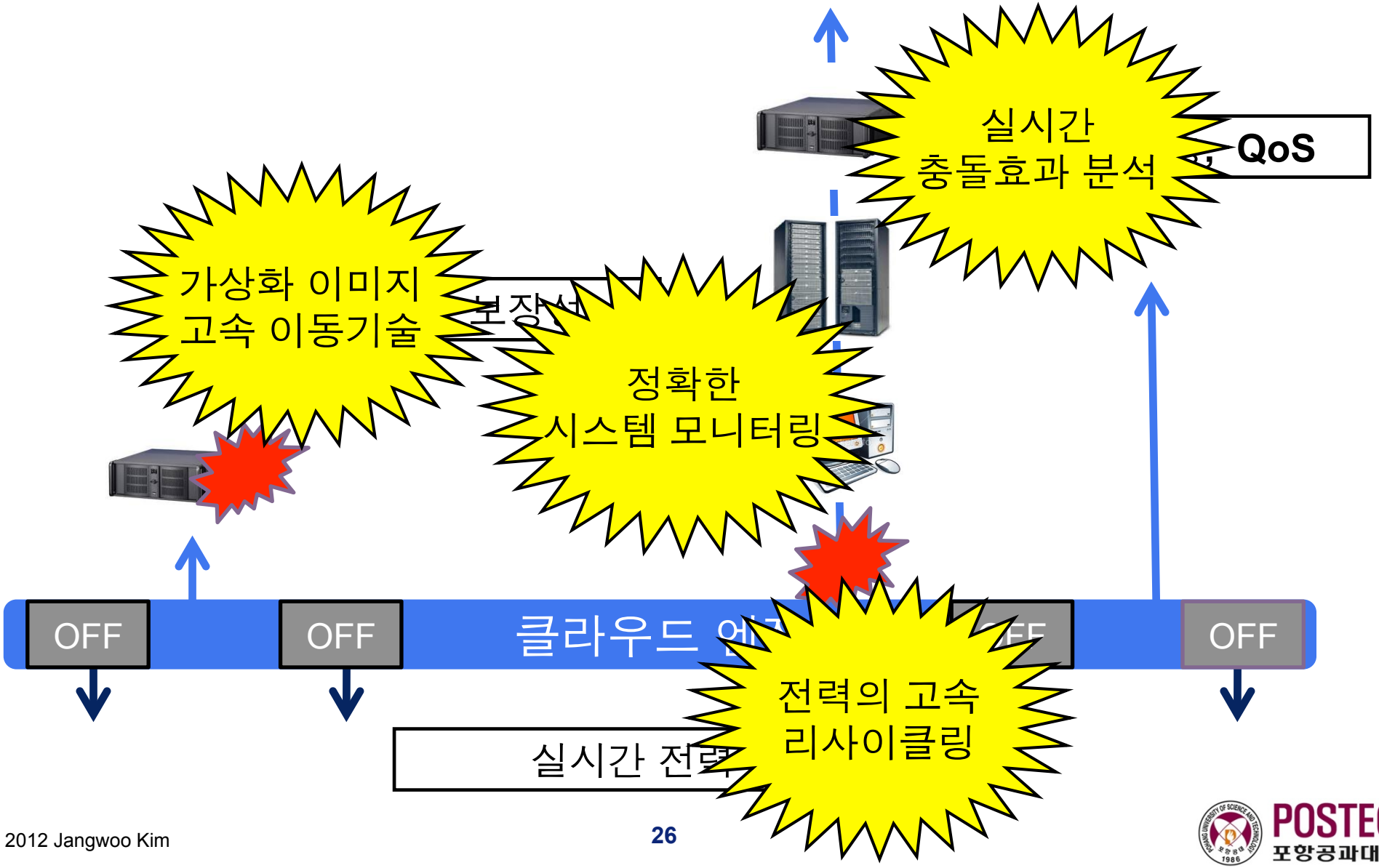
- **"고속" 이동기술**
  - **사용자는 자신의 컴퓨터의 물리적 이동을 느낄 수 없어야 함**
  - 이동시켜야 할 최소량 : 물리적 메모리 + 프로그램 컨텍스트

    (만약 네트워크를 통해 옮겨야 할 메모리가 24GB라면?)
  - 기존 방식: 사전이동 VS 사후이동

    사전: 대부분의 메모리가 이동될 때 까지 이동을 미룸. 이동실패 가능.

    사후: 즉시 이동 허용, 그러나 없는 메모리는 기존의 노드에서 가져와야 함.

POSTECH 클라우드 엔진 : 실시간 최적화



감사합니다.

Jangwoo Kim
e-mail: jangwoo@postech.ac.kr
http://www.postech.ac.kr/~jangwoo