리눅스 시스템을 위한 성능 분석

2010. 06. 24

국민대학교
연구원 이남승

목차

- 성능 분석의 필요성
- 리눅스 시스템의 성능 측정 기술
- 리눅스 기반 임베디드 시스템을 위한 본 연구 그룹의 성능 측정 기술
- 본 연구 그룹의 커뮤니티 소개

성능 분석의 필요성

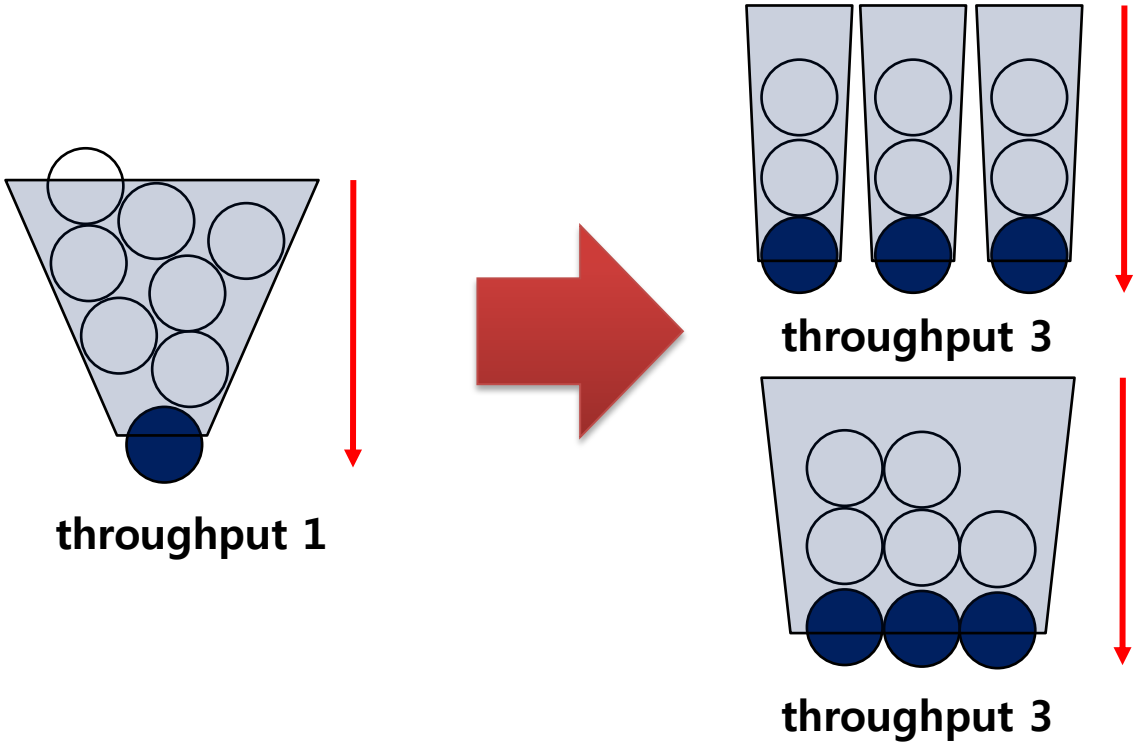
성능 분석의 필요성

fast enough

too slow

- **일반적인 컴퓨팅 환경(데스크톱 PC, 서버)**
  - 풍부한 하드웨어 자원이 제공되므로, 프로그램 작성시 성능적인 면은 고려대상이 되지 않는다.
- **임베디드 시스템**
  - 일반적인 컴퓨팅 환경에 비하여 매우 제한적인 하드웨어 자원 제공으로 인하여, 성능 적인 면이 반드시 고려되어야 한다.

성능 분석의 필요성



- **프로그램의 정확한 성능 병목 지점의 파악**
  - 프로그램의 성능을 향상
  - 개발 비용 절감
  - 생산성을 증가

성능 분석의 필요성

- **임베디드 시스템을 위한 성능 분석 도구의 필요성**
  - 복잡한 개발 과정
  - 비효율적인 성능 검증 프로세스

  >> 개발 비용 증가

리눅스 시스템의 성능 측정 기술

리눅스 시스템의 성능 측정 기술 – 프로파일링(1)

- **프로파일링(profiling)방법에 따른 분류**
  - **측정형 프로파일링**
    - 분석코드를 원하는 지점에 실제로 삽입하는 방법
    - **정적(Static) VS 동적(Dynamic)**
      - 정적 : 프로그램의 실행이전에 분석코드가 이미 실행파일에 포함됨
        - 소스코드 레벨(source code level)
      - 동적 : 프로그램의 실행 이후 동적으로 원하는 코드를 삽입
        - 리눅스의 Ptrace등의 기법을 사용
  - **샘플링형 프로파일링**
    - 분석코드의 삽입 없이 주기적으로 실행프로그램의 정보를 기록하는 방법
      - 문맥교환(context switch)이후 실행프로그램의 PC값을 기록

리눅스 시스템의 성능 측정 기술 – 프로파일링(2)

- **측정형 VS 샘플링형**

측정형	샘플링형
측정코드로 인해 실행루틴이 변경될 수 있다 측정코드 실행으로 인한 오버헤드가 존재한다 계산 결과 값이 정확하다	실행루틴의 변경이 없다 문맥교환 인한 오버헤드가 존재한다. 계산결과 값이 근사값이다

- **정적 측정형 VS 동적 측정형**

정적 측정형	동적 측정형
응용 프로그램이 수정이 필요하다 (소스코드 수정 or 재 컴파일) 성능측정 오버헤드가 작다	응용프로그램의 수정이 필요없다 성능측정 오버헤드가 크다

리눅스 시스템의 성능 측정 기술
- GNU gprof

- **사용자 함수 프로파일러**
  - 측정형 프로파일링
    - Gcc의 –pg 옵션을 이용하여 컴파일과정에서 각 함수들의 진입 진출지점에 측정코드를 삽입한다.
      - **프로그램의 재 컴파일이 필요함**
  - 사용자 함수의 성능분석
    - Flat 프로파일
      - 프로그램의 각 함수의 **소요시간** 및 **실행횟수**
    - 호출 그래프
      - 해당함수를 호출한 **상위함수**, **호출된 횟수**, 해당함수의 **서브루틴**

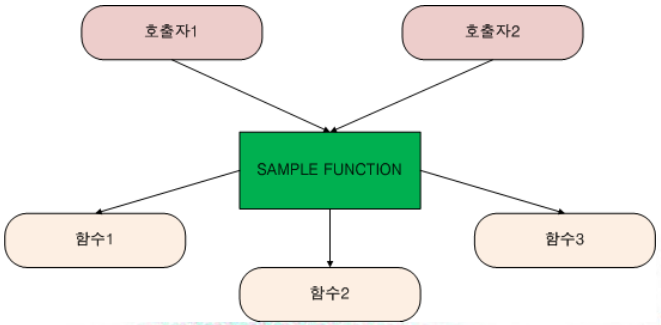
```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  first
  0.00      0.00     0.00        1     0.00     0.00  forth
  0.00      0.00     0.00        1     0.00     0.00  second
  0.00      0.00     0.00        1     0.00     0.00  third


                        Call graph


granularity: each sample hit covers 4 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00       1/1           main [10]
[1]      0.0    0.00    0.00       1         first [1]
-----------------------------------------------
                0.00    0.00       1/1           third [4]
[2]      0.0    0.00    0.00       1         forth [2]
-----------------------------------------------
                0.00    0.00       1/1           main [10]
[3]      0.0    0.00    0.00       1         second [3]
-----------------------------------------------
                0.00    0.00       1/1           main [10]
[4]      0.0    0.00    0.00       1         third [4]
                0.00    0.00       1/1           forth [2]
-----------------------------------------------
```

실행결과



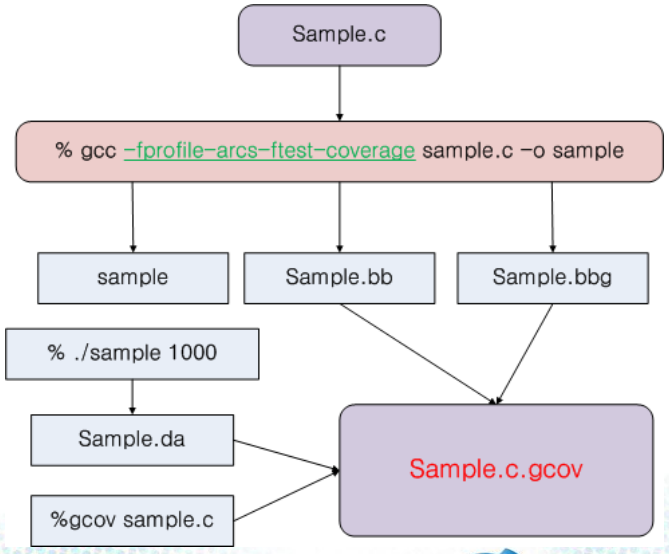
호출 그래프

리눅스 시스템의 성능 측정 기술
- Gcov

- **코드 커버리지(code coverage) 분석**
  - 프로그램 소스코드 단위로 죽은(실행되지 않는)코드를 분석함
    - **소스코드 라인단위로 실행시간과 실행횟수의 분석이 가능함**
  - 컴파일시 –fprofile-arcs-ftest-coverage 옵션필요
    - **프로그램의 재 컴파일이 필요함**

```
1          :  #include <stdio.h>
2          :  int main (void)
3          1  {
4          :    int i, total;
5          1    total = 0;
6         11    for (i = 0; i < 10; i++)
7         10      total += i;
8          1    if (total != 45)
9          0      printf ("Failure\n");
10         :    else
11         1      printf ("Success\n");
12         1    return 0;
13         :  }
```

실행예제



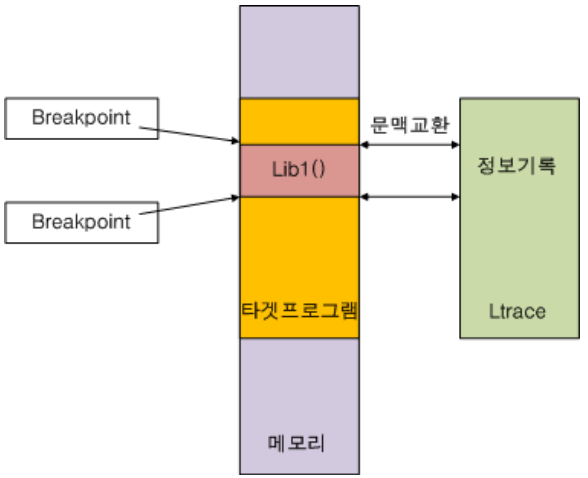
Gcov 출력생성 단계

리눅스 시스템의 성능 측정 기술
- Ltrace & Strace

- **Ltrace**
  - 사용자 프로그램이 사용하는 동적 라이브러리 추적
  - 시스템 콜의 추적도 가능함
  - 리눅스의 **Ptrace**를 이용하여 추적지점에 Breakpoint명령어를 삽입

- **Strace**
  - 시스템 호출 추적
  - Ptrace 기법 사용

- **동적 측정형 기법을 사용**
  - 소스코드의 수정 또는 재 컴파일이 필요 없음
  - **성능측정상의 오버헤드가 비교적 큼**
    - **정보 수집 및 기록을 위한 문맥교환**



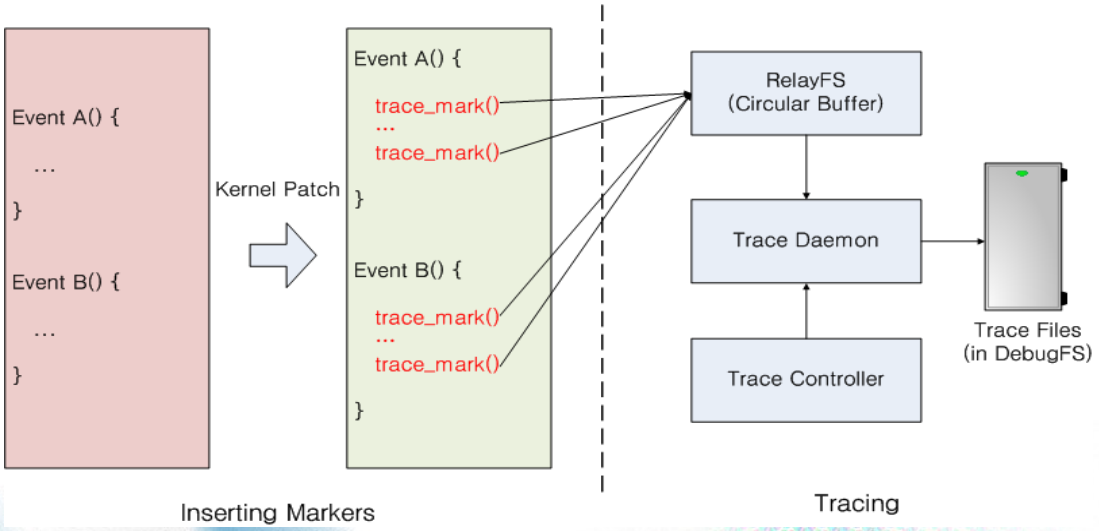
Ltrace 동작방법

```
open(".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
fcntl64(3, F_GETFD)                     = 0x1 (flags FD_CLOEXEC)
getdents64(3, /* 18 entries */, 4096)   = 496
getdents64(3, /* 0 entries */, 4096)    = 0
close(3)                                = 0
fstat64(1, {st_mode=S_IFIFO|0600, st_size=0, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f2c000
write(1, "autofs\nbackups\ncache\nflexlm\ngames"..., 86autofsA
```

Strace 실행결과

리눅스 시스템의 성능 측정 기술
- Linux trace toolkit(LTT)

- **리눅스 커널 전반에 관한 정보 제공**
  - 시스템 호출, 인터럽트, 트랩, 파일시스템, 타이머, IPC등
- **리눅스 커널 패치 필요**
  - 커널 내부의 각 이벤트 시작 및 종료 지점에 추적코드를 삽입함
- **사용자를 위한 그래픽 인터페이스 제공 - LTTV**



LTT 동작 구조

리눅스 시스템의 성능 측정 기술
- Oprofiler

- **리눅스용 시스템 프로파일러**
  - 2.2x, 2.4x, 2.6x 커널에 포함됨
    - 커널 빌드시 Oprofile기능 활성화
  - 응용프로그램 및 커널의 성능분석
  - **샘플링형 프로파일러**
    - 실행때마다 다른 결과를 얻을 수 있다.

리눅스 시스템의 성능 측정 기술
- 대화식 디버거(Debugger)

- **GDB : 소스코드 디버거**
  - 프로세스의 실행 및 수정, 내부 변수 값 모니터링 및 수정
  - 실행중인 프로세스 추적 가능
  - 정지점(breakpoint) 설정 가능 : 리눅스의 Ptrace사용
  - -g 옵션으로 컴파일
  - 원격 디버깅 제공 : **임베디드 시스템의 분석에 유용**
    - TCP/IP등으로 전송하여 다른 시스템에서 결과 분석
  - 그래픽 사용자 인터페이스 제공 : DDD

- **KGDB : 커널용 디버거**
  - 커널 내부에 포함 : 커널 빌드시 옵션 설정
  - 소스코드 레벨은 분석 불가능

리눅스 시스템의 성능 측정 기술
- 메모리 분석

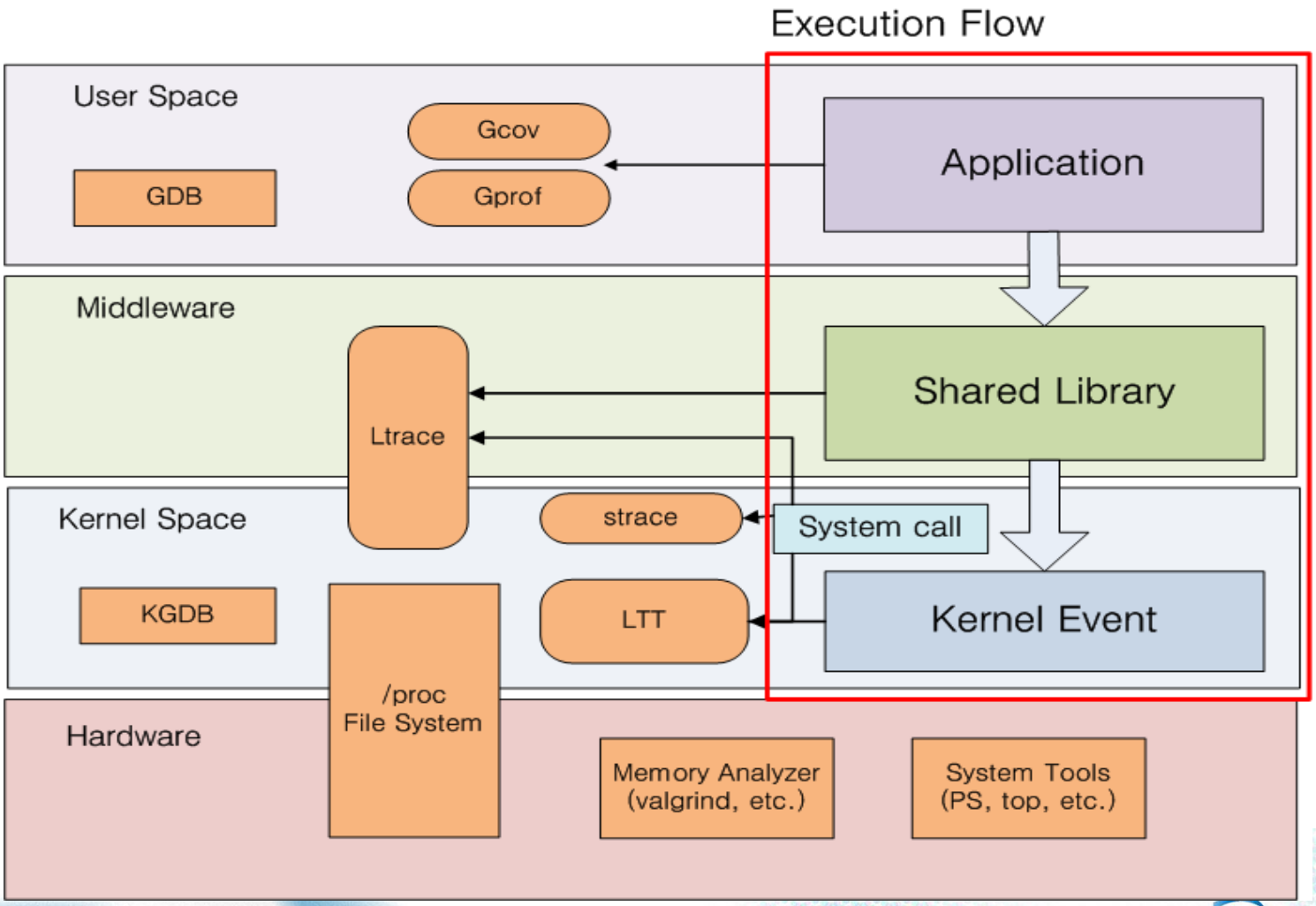
- **Memwatch**
  - 메모리 오류 감지
    - 이중 메모리 해제, 오류성 해제, 해제하지 않은 메모리, 오버플로우, 언더플로우
  - 컴파일 시 -DMEMWATCH & -DMW_STDIO 추가 필요
- **YAMD**
  - 동적 메모리 할당 오류 감지

- **Valgrind**
  - 모든 메모리 접근 및 malloc()/new()/free()/delete() 호출 추적
    - 초기화 되지 않은 메모리 사용
    - 해제된 메모리 읽기/쓰기
    - 배열 오버플로어, 언더플로어
    - 메모리 누수
    - 초기화 되지 않은 메모리 또는 정상적인 주소지정이 불가능한 메모리 넘기기
    - Malloc/new/new[] 와 free/delete/delete[] 불일치
  - 캐시 시뮬레이션
    - 소스코드의 행 단위로 캐시 실패횟수를 기록
  - 지원 Architecture : x86, PowerPC

리눅스 시스템의 성능 측정 기술
- 시스템 도구 & /proc 파일시스템

- **Ps**
  - 현재 시스템에서 **실행중인 프로세스**에 대한 **스냅샷 정보** 제공
- **Pstree**
  - 시스템의 **프로세스 트리**를 보여줌
- **Top**
  - 현재 리눅스 시스템에서 **실행중인 프로세스 정보**를 **동적으로** 보여줌
  - 부하 평균 , 메모리 사용률
- **Vmstat**
  - 시스템의 **리소스 정보**를 보여줌
    - CPU, I/O, 메모리
- **/proc 파일 시스템**
  - **메모리에 있는 시스템 정보의 반영**
    - 프로세서, CPU, 메모리, 파일시스템, 인터럽트, 파티션 등 시스템 전반적인 정보를 보여줌
    - 많은 유틸리티들이 /proc를 이용하여 정보 수집
  - 읽기전용 & 쓰기가능
    - 쓰기가능 정보의 경우 변경하면 바로 **시스템동작에 적용됨**

리눅스 시스템의 성능 측정 기술
- 계층별 성능분석 기술



리눅스 시스템의 성능 측정 기술
- 통합 성능분석 도구(1)

- 하나이상의 계층에 대한 성능분석 정보를 제공
  - Workbench (WindRiver)
  - Qplus/Esto (ETRI)
  - DevRocket (MontaVista)
  - Mevalet (NEC)

리눅스 시스템의 성능 측정 기술
- 통합 성능분석 도구(2)

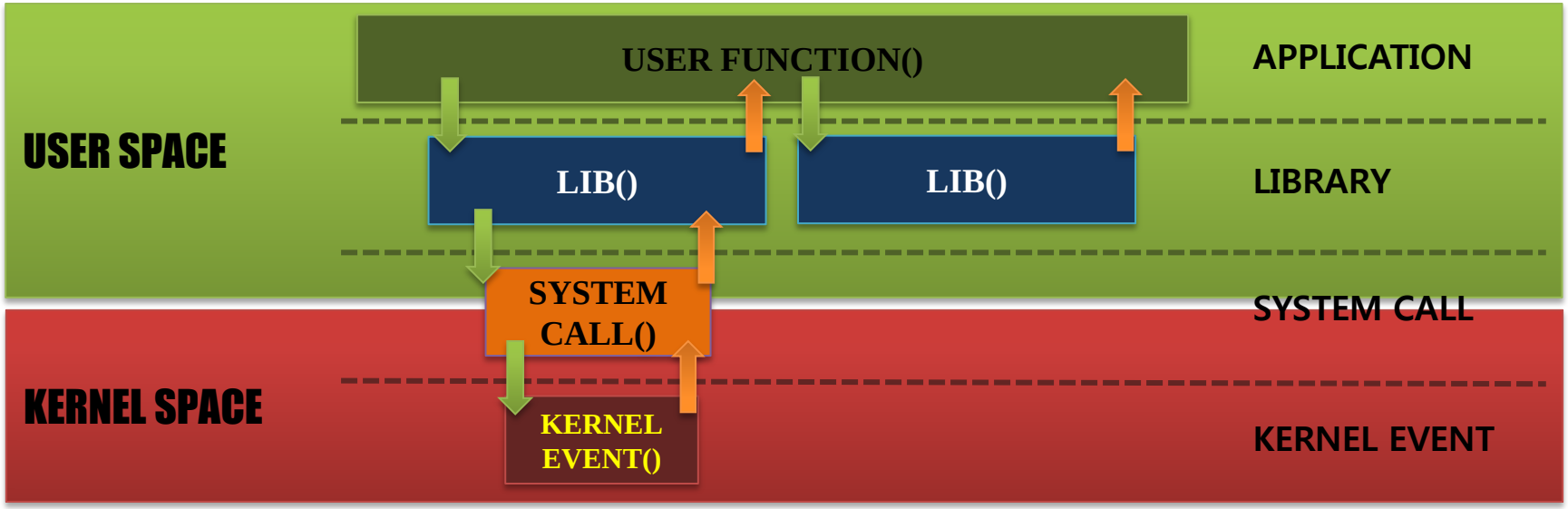
	Workbench	DevRocket	Mevalet	Qplus/Esto
개발 그룹	WindRiver	MontaVista	NEC	ETRI
프로세스	O	O	O	O
시스템 자원	O	O	X	O
사용자 정의함수	O	X	O	O
라이브러리함수	X	X	X	X
시스템 호출	X	O	O	X
커널 이벤트	X	O	O	X
사용자 함수 - 라이브러리	X	O	X	X
사용자 함수 - 시스템 호출	X	X	X	X
라이브러리 - 시스템 호출	X	X	X	X
시스템호출 - 커널 이벤트	X	X	O	X

리눅스 기반 임베디드 시스템을 위한

본 연구 그룹의 성능 측정 기술

본 연구 그룹의 성능 측정 기술

통합 성능 분석 도구의 필요성

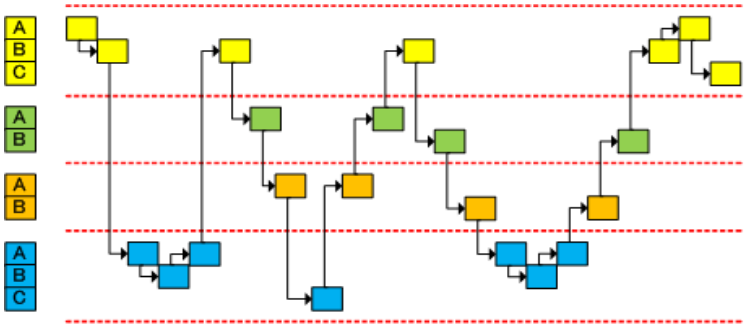


<리눅스의 계층적 소프트웨어 계층>

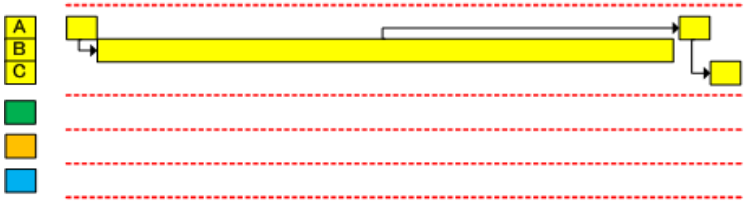
- **하나의 사용자 함수의 호출은 그 하위 계층 함수들의 호출을 야기한다.**
  - 하나의 사용자 함수의 호출은 전체 시스템의 성능에 영향을 미친다.
- **하나의 함수에 대한 정확한 성능 분석은 그 하위 계층 함수들에 대한 정확한 성능 분석이 되어야만 가능하다.**

본 연구 그룹의 성능 측정 기술

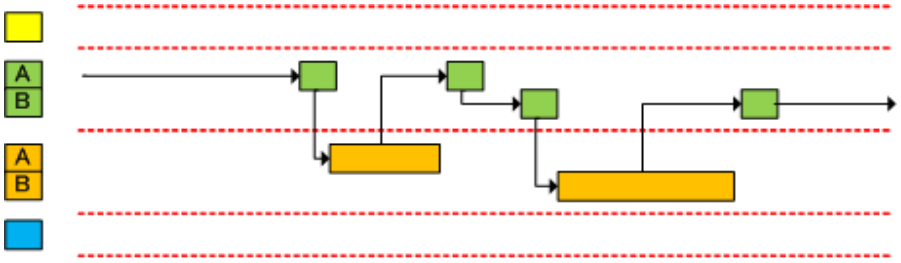
관련 연구(오픈소스 기반의 성능 분석 도구)



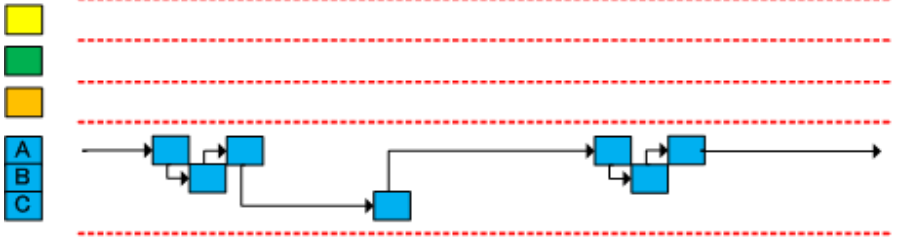
<가> 응용 소프트웨어의 계층별 실행 흐름



<나> Gprof의 응용 소프트웨어 분석 결과



<다> Strace, Ltrace 의 응용 소프트웨어 분석 결과



<라> LTTng의 응용 소프트웨어 분석 결과

User defined function (Layer)
Library function (Layer)
System call (Layer)
Kernel event (Layer)

- 한 소프트웨어 계층만을 고려하여 정확한 분석이 가능한가?

본 연구 그룹의 성능 측정 기술
관련 연구(오픈소스 기반의 성능 분석 도구)

	GDB	Gprof	Strace	Ltrace	Lttng	Oprofile	Ps, top	본 도구
주목적	디버깅	사용자함수 추적	시스템호출 추적	라이브러리 함수 추적	커널이벤트 추적	커널이벤트 추적	프로세스의 상태 확인	계층간 실행흐름분석
설치를 위한 커널의 설정 및 패치	X	X	X	X	O	O	X	O
사용자정의 함수 분석	X	O	X	X	X	O	X	O
라이브러리 함수 분석	X	X	X	O	X	O	X	O
시스템 호출 분석	X	O	O	O	X	O	X	O
커널 이벤트 분석	X	X	X	X	O	O	X	O
사용자 영역 – 라이브러리 상관 관계 분석	X	X	X	X	X	O	X	O
사용자 함수 – 시스템 호출 상관 관계 분석	X	X	X	X	X	O	X	O
라이브러리 – 시스템호출 상관 관계 분석	X	X	X	X	X	O	X	O
시스템 호출 – 커널 이벤트 상관 관계 분석	X	X	X	X	X	O	X	O

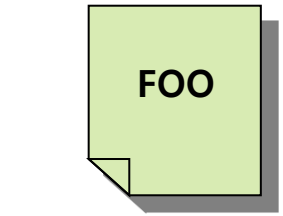
본 연구 그룹의 성능 측정 기술

관련 연구(오픈소스 기반의 성능 분석 도구)

- 관련 기술들의 공통적인 특징
  - 특정 계층에 특화된 성능 분석만을 함
- 관련 기술들의 공통적인 맹점
  - 전체 시스템의 관점에서의 성능을 고려하지 않음

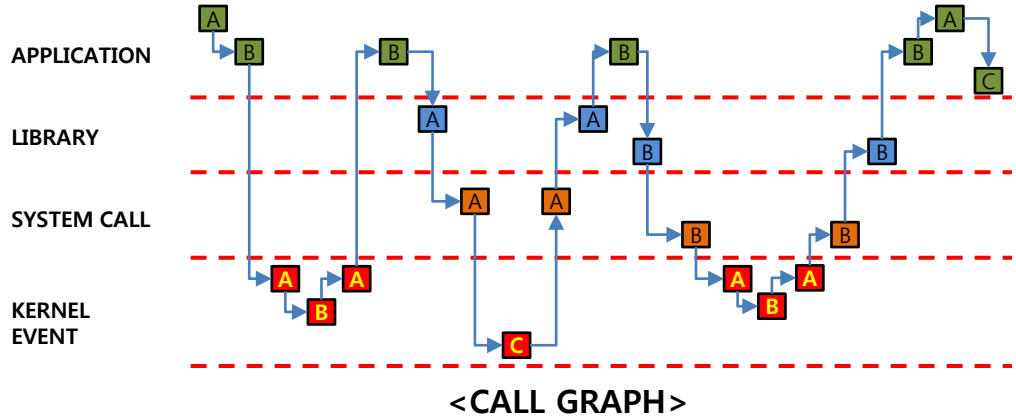
본 연구 그룹의 성능 측정 기술

연구 목표

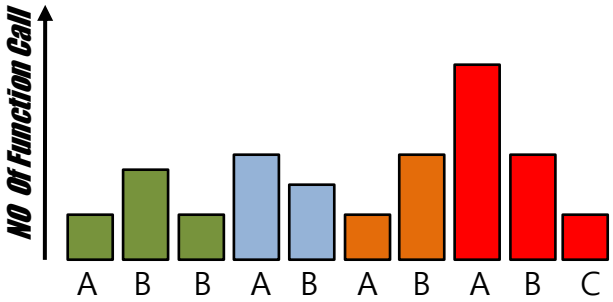


<APPLICATION 'FOO'>

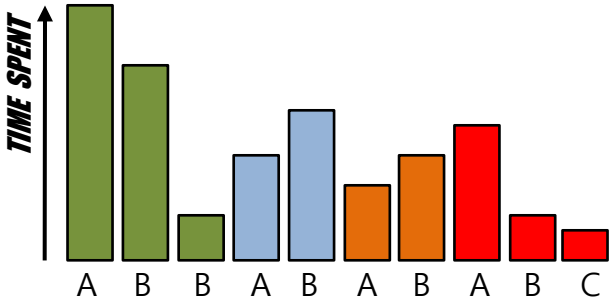
성능분석



<CALL GRAPH>



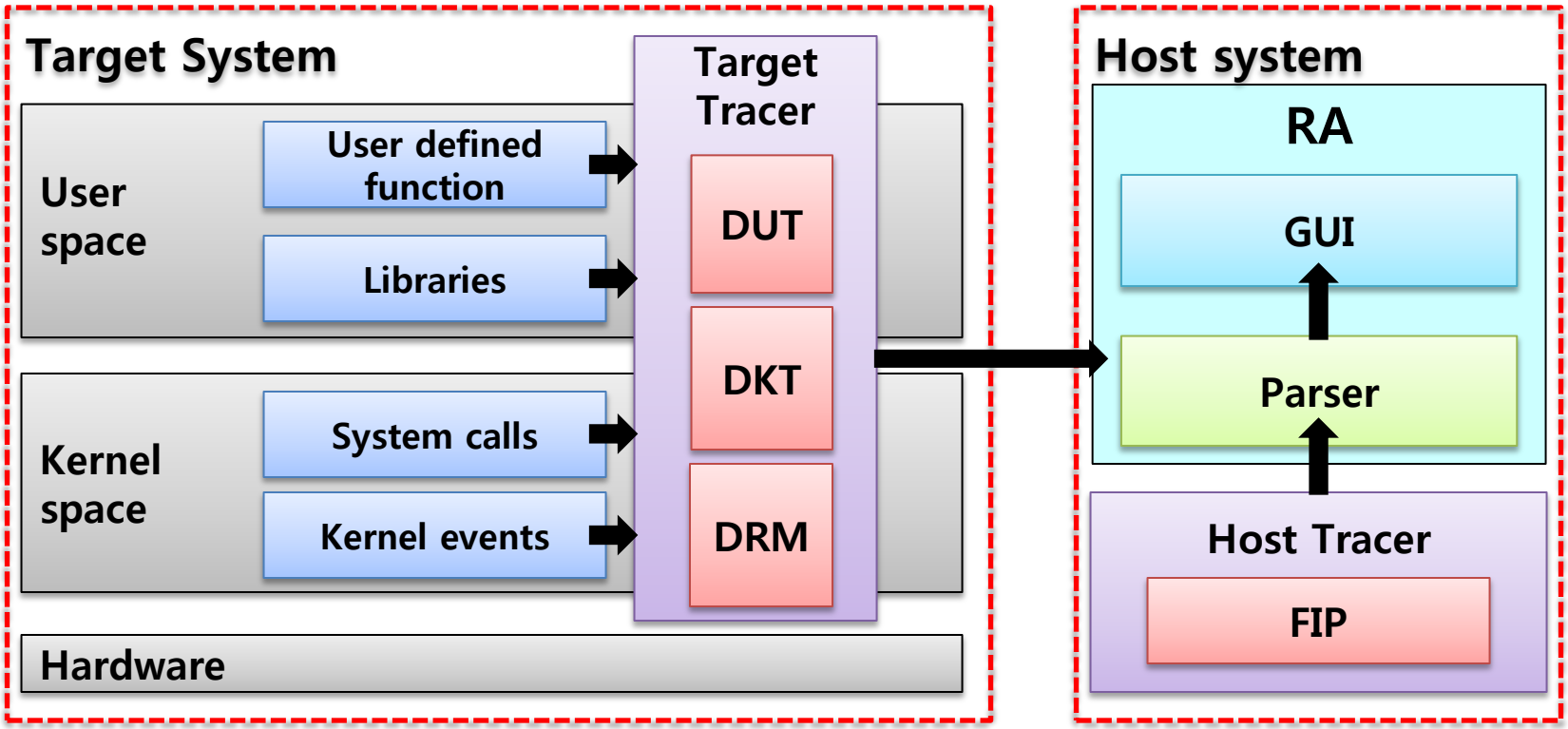
<함수들의 호출 횟수>



<함수들의 소요시간>

- 체계적인 성능 분석으로 성능상의 병목지점을 직관적으로 도출하는 통합 성능 분석 도구 구현

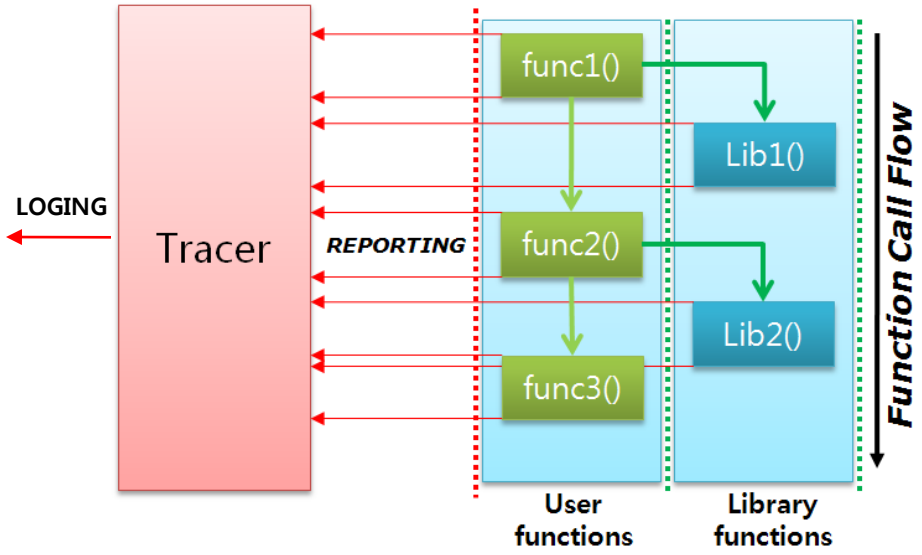
본 연구 그룹의 성능 측정 기술
시스템 구조



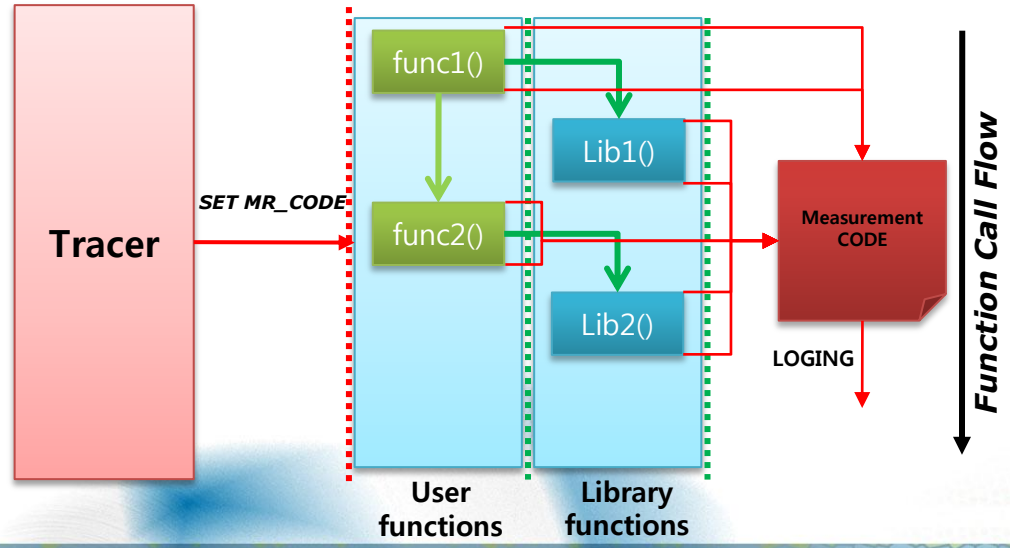
- **호스트 시스템(Host System)**
  - Host Tracer : 어플리케이션의 심볼 테이블을 이용하여 함수에 대한 정보(함수의 이름, 주소 등)를 추출
  - Parser, GUI : 분석 결과들을 연동하여 사용자에게 제공
- **타겟 시스템(Target system)**
  - DUT(Dynamic Userspace Trace), DKT(Dynamic Kernelspace Trace) : 동적으로 사용자, 커널 영역을 추적
  - DRM(Dynamic Resource Monitor) : CPU, 메모리등의 자원 사용양상을 분석

본 연구 그룹의 성능 측정 기술

동적 사용자 영역 추적기



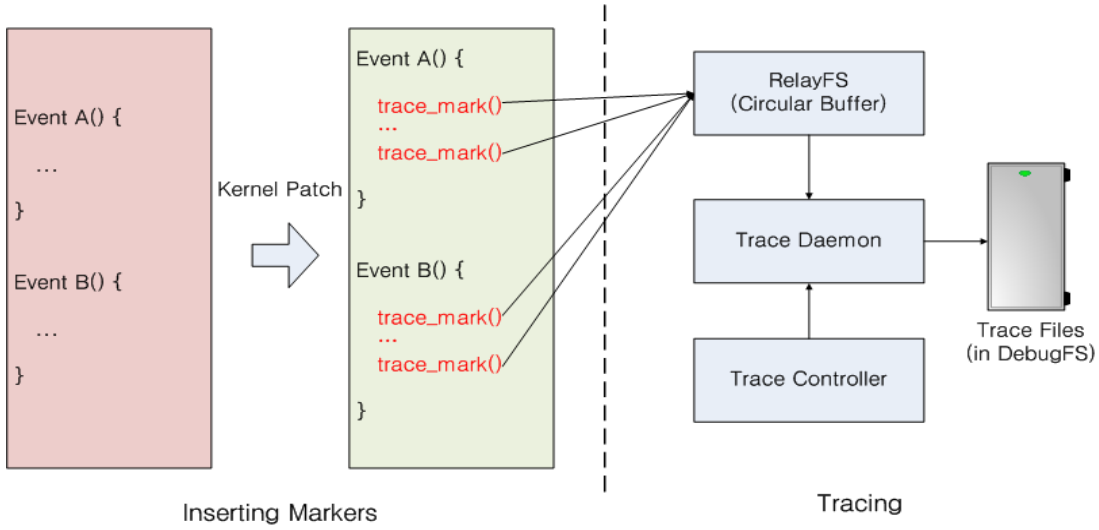
- 기존 연구
  - 바이너리 분석 후 중단점 삽입으로 응용 추적(Ptrace system call 사용)
    - 함수 진입, 진출 점을 추적
  - 단점
    - 성능 추적 간에 잦은 문맥 교환으로 큰 오버헤드 발생



- 본 연도 연구
  - 소스 수정 없이, 바이너리 분석 후 런타임에 Measurement Code(성능 추적 코드) 삽입(Ptrace system call 사용)
    - 함수 진입, 진출 점을 추적
  - 장점
    - 어플리케이션 수행간 문맥 교환이 발생하지 않음(오버헤드 문제 개선)

본 연구 그룹의 성능 측정 기술

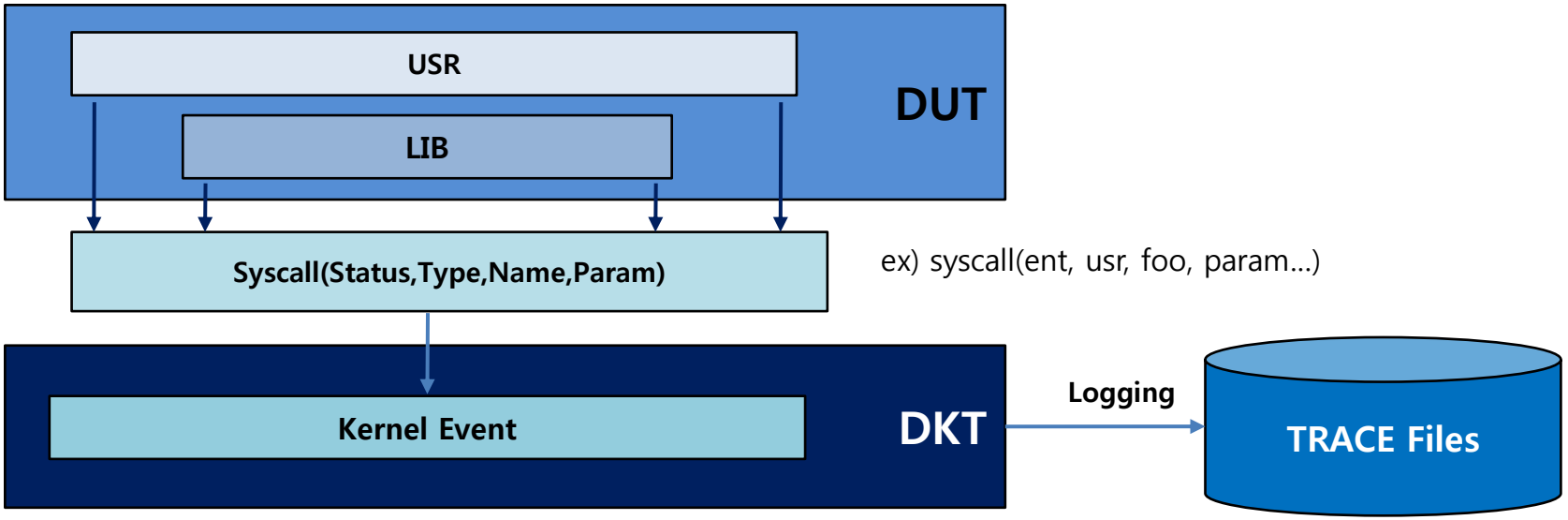
동적 커널 영역 추적기



- 측정 코드 삽입을 통한 시스템 호출, 커널 이벤트를 추적
  - 리눅스 커널 추적기 LTTng 이용
    - 시스템 호출과 커널 이벤트의 핸들러에 측정 코드인 마커 삽입
    - 측정 코드 삽입 후 실행 시 핸들러의 마커가 추적 정보를 기록

본 연구 그룹의 성능 측정 기술

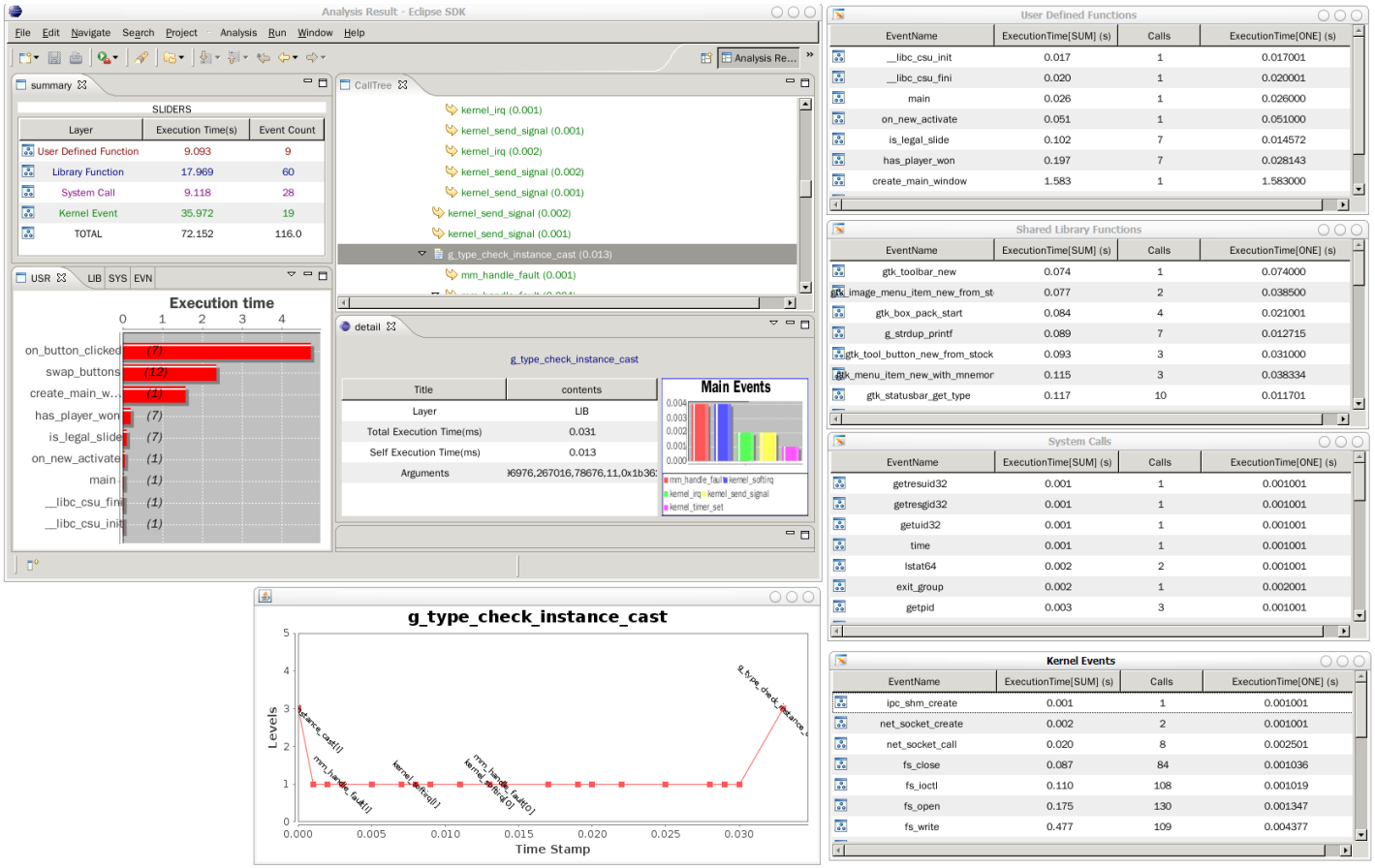
사용자 영역 추적기와 커널 영역 추적기의 연동



1. 각 사용자 영역의 함수의 진입, 진출 지점에서 특정 system call 을 호출
2. 커널 이벤트를 발생
3. DKT 에서 호출 정보를 수집

본 연구 그룹의 성능 측정 기술

Graphic User Interface



- 직관적인 분석 결과 출력으로 사용자가 쉽게 성능상의 병목 지점을 발견할 수 있도록 유도

본 연구 그룹의 커뮤니티 소개

- **주소 : http://sourceforge.net/projects/ipa-tool/**
- **개발인원 : 9명**
- **V0.2 released, v0.3 개발 진행중**
  - 사용자 영역 추적기 최적화, GUI 보완
- **프로그램 소개 및 사용자 가이드 문서 등록**
- **순위 : 10,820 (2010. 6 현재)**
- **다운로드 횟수 : 290**

감사합니다.

Q & A