임베디드 리눅스 시스템의 소프트웨어 계층 구조를 고려한 성능 분석 프레임워크

(A Performance Analysis Framework Considering the Hierarchy of Embedded Linux Systems Software Architecture)

곽 상 헌 † 이 남 승 † 이 호 림 † 임 성 수 ††
(Sangheon Kwak) (Namseung Lee) (Horim Lee) (Sung-Soo Lim)

요 약 최근 임베디드 시스템은 운영체제를 포함하는 복잡한 소프트웨어 계층 구조를 가지는 형태로 발전하고 있다. 이러한 소프트웨어의 성능을 분석하기 위해서는, 한 소프트웨어 계층에서의 성능 뿐 아니라 전체 소프트웨어 계층 구조를 모두 고려해야 한다.

본 논문에서는 리눅스 기반 임베디드 시스템의 모든 소프트웨어 계층 구조를 고려할 수 있는 성능 분석 도구를 설계하고 구현한 결과를 보인다. 제안하는 기법은 응용 프로그램이나 라이브러리에 대한 재컴파일 없이 모든 소프트웨어 계층의 성능 분석에 필요한 측정 정보를 수집한다. 이 기법을 통해 리눅스 기반 임베디드 시스템에서 응용 프로그램의 실행에 따라 발생하는 사용자 정의 함수, 미들웨어 라이브러리 함수, 커널의 시스템 호출, 커널 이벤트에 대한 다양한 성능 분석을 수행할 수 있다. 실험을 통해 본 연구를 통해 구현된 분석도구를 사용하여 실제 실행 경로 분석, 각 소프트웨어 계층의 함수나 이벤트의 소요시간 분석, 그리고 소프트웨어 계층간 실행 흐름 분석 결과를 확인할 수 있으며, 이를 통해 전체 소프트웨어 계층상의 성능 병목을 찾을 수 있음을 보인다.

키워드 : 소프트웨어 성능 분석, 성능 평가, 프로파일링, 소프트웨어 계층, 리눅스 기반 플랫폼

Abstract Recent embedded systems are being more complicated due to their hierarchical software architecture including operating systems. The performance of such complicated software architecture could not be well analyzed through separate analysis of each software layer; the combined effect and the interactions among the whole software layers should be considered.

In this paper, we show the design and implementation of a performance analysis framework that enables hierarchical analysis of performance of Linux-based embedded systems considering interactions among the software layers. By using the proposed framework, we can obtain useful run-time information about a hierarchical software structure which usually consists of user-defined function layer, library function layer, system call layer, and kernel events layer. Experimental results reveal that the proposed framework could accurately identify the performance bottlenecks with the corresponding software layers during executions of target applications through the accompanying sub-steps of the analysis: the actual execution paths, the execution time of each observed event in each software layer, and the control flows across the software layers.

Key words : Software Performance Analysis, Performance Evaluation, Profiling, Software layer, Linux Platform

· 본 연구는 2008년 국민대학교 교내연구비 지원에 의해 수행되었음
· 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음(NIPA-2010-C1090-1031-0009)

† 학생회원 : 국민대학교 컴퓨터공학과
hunnyk7@naver.com
mainly82@gmail.com
nerigo79@gmail.com
†† 종신회원 : 국민대학교 컴퓨터공학과 교수
sslim@gmail.com

논문접수 : 2010년 1월 4일
심사완료 : 2010년 4월 6일

정보과학회논문지 : 컴퓨팅의 실제 및 레터 제16권 제6호(2010.6)

1. 서 론

휴대폰, MP3, PMP 등 이동 단말은 멀티미디어, 웹 기반 서비스 등 다양한 기능을 제공하기 위해 운영체제를 포함하는 복잡한 소프트웨어 계층 구조를 가지는 형태로 발전하고 있다[1-3].

하나의 시스템을 이루는 소프트웨어 계층은 사용자 정의 함수 계층, 라이브러리 함수 계층, 시스템 호출 계층, 커널 이벤트 계층으로 구분할 수 있다. 응용 프로그램을 실행하기 위해서는 이 모든 계층에서 필요한 서비스를 수행해야 한다. 따라서, 하나의 작은 응용 프로그램을 실행하더라도 정확한 성능 병목지점 또는 병목원인을 분석하기 위해서는 모든 소프트웨어 계층 사이의 영향 및 소프트웨어 계층 구조를 넘나 드는 실행 흐름에 대한 분석이 필요하다.

그림 1은 성능 분석시 전체 소프트웨어 계층의 영향을 고려해야 하는 이유를 다른 리눅스 기반 시스템을 위한 성능 분석 도구들과 비교하여 설명한다. 현재, 리눅스 기반 시스템을 위한 성능 분석 도구로는 gprof[4], Ltrace, Strace[5]과 같이 단일 계층을 위한 성능분석도구부터 WindRiver사의 workbench[6], NEC의 Mevalet[7]와 같이 시스템 전반에 대한 성능분석을 수행하는 통합 분석도구등 다양한 종류의 성능분석도구들이 존재한다. 그림 1(a)는 응용 소프트웨어의 각 소프트웨어 계층을 넘나드는 실제 실행 흐름을 보인다. 그림 1(b), (c), (d)는 기존 성능 분석 도구가 분석할 수 있는 제한된 소프트웨어 계층을 보인다. 즉, 기존 성능분석 도구는 해당 계층의 함수나 이벤트의 발생 빈도 및 소요 시간에 대한 분석은 가능하지만, 다른 계층과의 상관관계를 분석하지 못한다. 또한, 기존의 통합성능분석도구들 역시 단일계층을 위한 성능분석도구들보다 많은 분석정보를 제공하지만, 모든 소프트웨어 계층을 포괄한 분석정보를 제공하지는 못한다.

본 논문에서는 리눅스 기반 임베디드 시스템에서 실행되는 응용 프로그램의 성능을 모든 소프트웨어 계층의 영향을 고려하여 분석할 수 있는 성능 분석 프레임워크를 제안하고 이를 구현한 성능 분석 도구를 통한 성능 분석 결과를 보인다. 제안하는 분석 기법은 분석 대상 소프트웨어의 소스코드에 측정용 코드를 삽입하여 동적으로 실행시 정보를 수집하는 측정형 기법과 실행 전 응용 소프트웨어의 구조 및 특성을 분석하는 정적 분석 기술을 이용하여[8] 모든 소프트웨어 계층 수준의 다양한 성능 분석을 수행한다. 제안하는 기법을 이용해 분석 가능한 요소를 나열하면 각 소프트웨어 계층의 함수 및 이벤트의 소요시간, 발생 빈도, 호출-피호출 관계 뿐 아니라 각 소프트웨어 계층간 실행 흐름, 정적 실행 가능 경로 및 동적 실제 실행 경로 등이 포함된다.

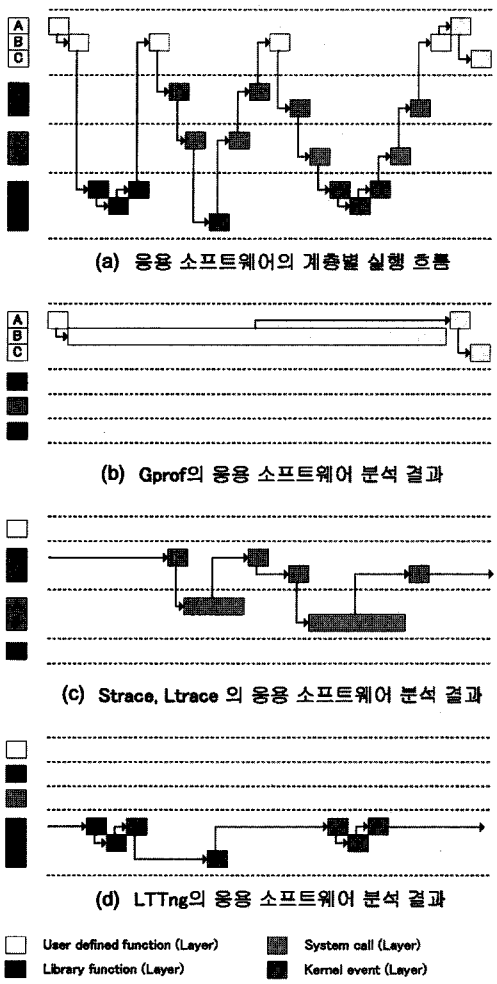


그림 1 응용 프로그램의 실제 실행 흐름과 기존 도구 별 성능 분석 결과 비교

본 논문의 구성은 다음과 같다. 2장에서는 성능분석을 위한 정보수집기법에 대하여 설명하고 3장에서는 기존의 성능분석도구들을 살펴본다. 4장에서는 본 연구에서 제안하는 성능분석 프레임워크의 구조 및 특징을 살펴본 후 5장에서 실험을 통하여 프레임워크에 대한 효용성을 확인하고, 6장에서는 결론을 맺고 향후 연구 과제에 대해서 검토한다.

2. 성능분석을 위한 정보수집 기법

2.1 측정형 기법

측정형 기법은 추적하고자 하는 지점에 추적을 위한 특수코드를 삽입하는 방법으로써 특수코드가 삽입된 실행파일을 동적으로 실행한 후, 특수코드가 생성하는 결

과를 이용하여 정보를 수집한다. 일반적으로 추적하고자 하는 루틴의 진입점과 진출점에 특수코드를 삽입하여 해당루틴의 호출횟수, 소요시간 등을 측정한다[8]. 측정형 기법의 경우 추적을 위하여 추가적인 코드가 삽입되므로, 추적을 위한 코드의 실행 및 이로 인한 캐쉬실패(cache miss)로 인하여 CPU 내에서 루틴의 실행순서를 변화 시킨다[8-10]. 이러한 점은 측정형 기법으로 수집한 정보의 정확성 떨어트린다.

측정형 기법은 특수코드를 삽입하는 단계에 따라 정적 측정형 기법과 동적 측정형 기법으로 나뉜다[11]. 정적 측정형 기법은 추적을 위한 특수 코드가 컴파일시점에 추가되는 되는데, 특수코드를 소스코드에 직접 추가하거나, 컴파일러에 의해 컴파일 단계에서 추가될 수 있다[11]. 정적 측정형 기법을 사용하는 대표적인 도구로서, 사용자 정의 함수를 추적하는Gprof, Kprof[12]등이 있다. 또한 LTTng[13]나 K42[14] 역시 커널 소스코드에 커널이벤트 추적을 위한 마커(marker)를 정적으로 삽입하여 시스템 호출 및 커널 이벤트의 발생 정보를 측정한다[12].

동적 측정형 기법은 실행 파일을 분석하여 추적을 원하는 부분의 메모리 주소에 중단점 및 분기와 같은 특수명령어를 삽입하는 방법이다. 일반적으로 추적을 원하는 지점에 중단점을 삽입한 후 실행시켜, 중단점이 실행될 때 발생하는 트랩을 처리하는 핸들러를 이용하여 정보를 수집한다. 동적 측정형 기법을 사용하는 대표적인 도구에는 리눅스 시스템의 성능이나 기능적 문제를 진단하는 SystemTap[15], 메모리 사용을 추적하는 valgrind[16], 대화형 디버거인 GDB[17], 동적으로 성능 평가 코드 등을 심어서 디버깅과 성능 평가를 하는 Dprobe[18], Dtrace[19], Pin[20], Paradyn[21] 등이 있다.

동적 측정형 기법에서 주로 사용하는 방법 중 하나가 리눅스의 ptrace이다. ptrace는 프로세스가 다른 프로세스의 시작과 멈춤, 메모리 및 레지스터에 대한 읽기와 쓰기처럼 다른 프로세스를 관리하는 것을 가능하게 해주는데[22], 이를 사용하는 대표적인 도구로는, 시스템 호출과 라이브러리 함수 호출을 추적하는 Strace, Ltrace 등이 있다.

정적 측정형 기법의 경우 소소코드 수준에서 특수코드를 삽입하기 때문에 성능분석을 위해서는 소스코드 또는 소스코드의 재컴파일이 필요하다는 단점을 가지지만, 동적 측정형 기법에 비해 측정을 위한 성능오버헤드가 작다. 동적 측정형 기법의 경우, 트랩을 처리하기 위한 핸들러로의 문맥교환(context switch)과 트랩처리 후 본래 명령어 복구, 실행, 중단점 재삽입과 같은 단계를 거쳐야 하기 때문에 정적 측정형 기법보다 성능오버헤드가 크다는 단점이 있다[11].

2.2 샘플링형 기법

샘플링 기법은, 데몬이나 성능 카운터를 이용하여 일정 주기마다 정보를 측정한다. 데몬을 통해 실행되고 있는 심볼의 정보를 추출하거나, PMU(Performance Countering Unit)을 이용해 명령어나 클럭 사이클과 같은 성능 관련 정보를 수집한다[8,9]. Oprofile[23]은 이 방법을 사용하여 사용자와 커널 영역의 인터럽트, 커널 모듈, 라이브러리, 응용의 명령어 측정 정보를 수집한다.

샘플링형 기법은 응용프로그램에 추가적인 코드의 삽입이 없기 때문에 측정형 기법의 문제점을 극복한다. 하지만 샘플링형 기법의 경우, 정보수집이 일정주기마다 일어나기 때문에 특정 루틴에 대한 정보를 놓치는 경우가 발생할 수 있다. 그렇기 때문에 샘플링형 기법을 이용하여 수집한 정보 역시 정확성이 떨어진다. 이러한 이유로 oprofile은 전체 소프트웨어 계층들의 성능분석이 가능하지만 각 함수 및 이벤트간 상관관계를 도출하지 못하는 한계를 가진다[8,9].

3. 기존의 리눅스 기반 성능 분석 도구

3.1 단일 계층을 위한 성능분석도구

프로그램의 실행관점에서 시스템을 계층화하면 다음과 같이 사용자 정의 함수 계층, 라이브러리 함수 계층, 시스템 호출 계층, 커널 이벤트 계층으로 구분할 수 있으며, 각 계층을 위한 대표적인 성능분석도구들이 존재한다.

gprof는 사용자 정의 함수를 분석하는 대표적인 리눅스 기반 성능분석 도구이다. gprof는 정적 측정형 기법을 이용하여 사용자 정의 함수에 대한 정보를 수집한다. gprof가 제공하는 정보로는 각 사용자 함수의 실행시간 및 호출횟수, 사용자 함수간의 호출관계가 있다[4]. 이외에도 사용자 정의 함수를 분석하는 성능분석도구로써, 프로그램의 코드 커버리지를 테스트하는 gcov[24], gprof에 그래픽 사용자 인터페이스가 추가된 kprof 등이 있다.

라이브러리 함수를 분석하는 대표적인 도구에는 ltrace가 있다. ltrace는 ptrace를 이용하는 동적 측정형 기법을 사용한다. ltrace는 프로그램에 의해 호출되는 라이브러리 함수들의 정보를 순차적으로 보여주는데 이 정보를 이용해서 각 라이브러리 함수들의 실행시간, 호출횟수, 매개변수 등을 파악할 수 있다[5].

또한 ltrace는 시스템 호출에 대한 정보도 보여준다. 그래서 ltrace를 이용하면 프로그램에서 호출되는 라이브러리 함수 및 시스템 호출의 정보들을 함께 확인할 수 있다[5]. 그 외에도 시스템 호출을 분석할 수 있는 도구로서, strace가 존재한다. strace는 ltrace와 같은 방법으로 동작하지만 시스템 호출에 관한 이벤트만을 처리한다[5].

LTTng(Linux Toolkit Tracer Next Generation)는 커널영역에서 발생하는 커널이벤트를 추적할 수 있는 대표적인 도구이다. LTTng는 커널이벤트 이외에도 커널에서 동작하고 있는 모든 프로세스들의 시스템 호출 역시 추적한다[13].

Oprofile은 샘플링형 기법을 사용하는 성능분석도구로서, 각 계층별 성능분석 정보를 모두 제공한다. 하지만 샘플링형 기법의 한계 때문에, 각 계층별 연관성을 도출하지 못하는 단점을 가진다[23].

표 1은 각각의 단일 계층을 위한 성능분석도구들이 제공하는 정보의 종류를 보여준다. 단일 계층을 위한 성능분석도구이기 때문에, 각각의 도구들이 보여주는 정보들은 특정 계층에 국한되어 있음을 알 수 있다.

3.2 통합 성능분석도구

단일 계층을 위한 성능분석도구들이 하나의 소프트웨어 계층에 대한 성능 분석 정보를 제공하는 반면, 통합성능분석도구는 하나이상의 계층에 대한 성능분석 정보를 제공한다.

WindRiver의 Workbench와 ETRI의Qplus/Esto[25], Montavista의 DevRocket[26]는 본래 통합개발도구로서, 내부적으로 다양한 디버깅 기능과 성능분석기능을 포함하고 있다. NEC의Mevalet은 성능분석도구로서, 운영체제 내부에 추적기를 포함시켜 운영체제에서 발생하는 시스템 호출 및 커널 이벤트 등에 관한 정보를 추적한다[7].

표 2는 각각의 통합분석도구들이 제공하는 정보의 종류를 보여준다. 몇몇 도구들은 소프트웨어 계층별 정보 이외에도 시스템의 자원사용 및 프로세스에 관한 정보를 제공한다.

3.3 기존 성능분석도구의 한계

표 1에서 보는 것과 같이 단일 계층을 위한 성능분석도구들은 시스템의 특정 하나의 계층에 대한 성능분석 결과만을 제공하기 때문에, 결과적으로 단일계층을 위한 성능분석도구들만을 이용하여 시스템의 정확한 병목지점 또는 병목원인을 찾는 것은 한계가 있다. 또한 각각의 단일 계층을 위한 성능분석도구들의 결과물을 취합하는 것으로는 계층간의 연관성을 파악할 수 없기 때문에 역시 문제해결에 한계가 있다.

통합성능분석도구들은 단일계층을 위한 성능분석도구들보다 많은 성능분석 정보를 제공한다. 하지만 표 2에서 볼 수 있듯이, 몇몇 통합성능분석도구들은 시스템 자원 및 프로세스에 관한 정보 등 다양한 정보를 보여주지만 이들 중에서도 시스템의 모든 소프트웨어 계층을 종합하여 결과를 보여주는 성능분석 도구는 없다.

결과적으로 시스템의 병목지점 및 병목원인을 정확하게 분석하는 것에는 한계가 있다.

표 1 개별 성능 분석 도구의 기능

	Gprof, Kprof	Strace, Ltrace	LTTng	Oprofile	제안하는 도구
사용자정의함수	O	×	×	O	O
라이브러리함수	×	O	×	O	O
시스템호출	×	O	O	O	O
커널이벤트	×	×	O	O	O
사용자함수-라이브러리	×	×	×	×	O
사용자함수-시스템호출	×	×	×	×	O
라이브러리-시스템호출	×	×	×	×	O
시스템호출-커널이벤트	×	×	O	×	O

표 2 통합 성능 분석 도구의 기능

	Work-bench	Dev-Rocket	Mev-alet	Qplus+ /Esto	제안하는 도구
상용화여부,개발그룹	Wind-River	Monta-vista	NEC	ETRI	KESL
프로세스별 상태 분석	O	O	O	O	×
자원 사용량 분석	O	O	×	O	×
사용자정의함수	O	×	O	O	O
라이브러리함수	×	×	×	×	O
시스템호출	×	O	O	×	O
커널이벤트	×	O	O	×	O
사용자함수-라이브러리	×	O	×	×	O
사용자함수-시스템호출	×	×	×	×	O
라이브러리-시스템호출	×	×	×	×	O
시스템호출-커널이벤트	×	×	O	×	O

4. 계층적 성능 분석 프레임워크

4.1 전체 시스템 구조

본 논문에서 제안하는 소프트웨어 계층별 성능 분석 프레임워크는 그림 2와 같은 구조를 가진다. 타겟 시스템에는 각 소프트웨어 계층의 성능을 측정하는 동적 사용자 영역 추적기(Dynamic Userspace Trace)와 동적 커널 영역 추적기(Dynamic Kernelspace Trace)가 있다. 각각의 추적기는 분석하는 타겟 시스템의 소프트웨어 계층을 크게 사용자 영역과 커널 영역으로 구분하여 성능을 측정한다. 각 영역을 기능에 따라 세부 계층으로

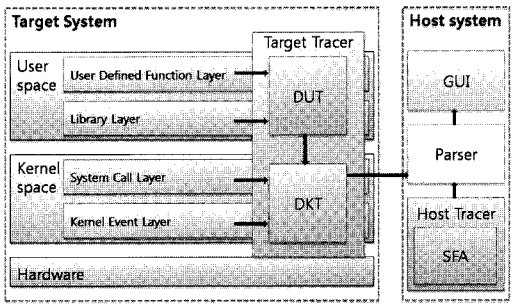


그림 2 시스템의 전체 구조

구분하면, 사용자 영역은 사용자 정의 함수 단위로 실행되는 사용자 정의 함수 계층, 사용자 영역에서 공유 라이브러리의 형태로 존재하는 미들웨어 함수들이 모여있는 라이브러리 계층으로 나뉜다. 그리고 커널 영역은 사용자 영역에서 커널 영역으로 진입하는 진입 지점인 시스템 호출 계층, 커널 내부의 이벤트로 이루어져 인터럽트나 이벤트를 처리하는 커널 이벤트 계층으로 나뉜다. 각 DUT와 DKT로부터 도출된 성능 분석 결과는 하나의 결과로 통합되어 호스트 시스템에 전달된다.

호스트 시스템에는 타겟 시스템에서 실행되는 응용 프로그램의 정적인 함수 호출관계를 분석하는 정적 호출 그래프 분석기(Static Call Graph Analyzer)가 있으며, 이 정적 분석 결과와 타겟 시스템에서 동적으로 측정한 결과를 조합하여 최종 분석 결과를 도출해주는 파서가 있다. 사용자는 호스트 시스템의 GUI 컴포넌트를 통해서 직관적인 형태의 성능 분석 결과를 확인할 수 있다.

4.2 동적 실행 추적기

본 프레임워크의 동적 실행 추적기는 응용 어플리케이션이 각 소프트웨어 계층을 거쳐 시스템 위에서 구동되는 동안 어플리케이션의 실행 흐름을 추적하고 성능을 분석하는 도구이다. 이 도구를 통해서 응용 어플리케이션의 구동간에 각 소프트웨어 계층의 어느 함수들이 호출 되는지 파악할 수 있고, 각 함수의 소요시간 및 호출 횟수에 대한 통계 수치 등을 파악할 수 있어서, 하나의 응용 어플리케이션의 구동으로 인하여 전체 시스템에 미치는 성능적인 영향을 파악할 수 있다.

동적 실행 추적기는 크게 동적 사용자 영역 추적기(DUT)와 동적 커널 영역 추적기(DKT)로 구분된다.

4.2.1 동적 사용자 영역 추적기(DUT)

DUT는 사용자 정의 함수와 라이브러리 함수를 모두 추적하기 위한 사용자 영역 추적기이다. 본 연구에서는 사용자 함수 추적을 위하여 사용자 함수를 추적기인 Utrace(User defined function Trace)를 개발하였다. 또한 라이브러리 함수를 추적하기 위하여 오픈 소스 도구인 Ltrace(Library function Trace)를 수정하여, Utrace와 통합시킴으로써, 사용자 정의 함수와 라이브러리 함수를 모두 추적할 수 있는 하나의 도구인 DUT를 개발하였다.

그림 3은 DUT 의 동작원리를 보여준다. DUT는 Ltrace와 Utrace를 연동한 것이기 때문에 DUT와 Ltrace의 동작원리는 거의 동일하다[27]. DUT의 동작 순서는 다음과 같다.

(1) DUT는 대상 응용 프로그램의 심볼 테이블을 분석하여 응용 프로그램 내에서 사용되는 사용자 정의 함수 및 라이브러리 함수들의 주소를 찾는다. (2) 찾아낸

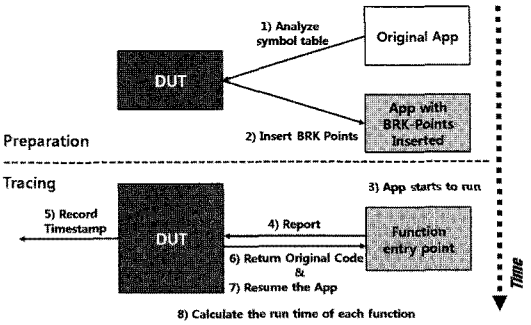


그림 3 동적 사용자 영역 추적기(DUT)의 동작원리

각 사용자 정의 함수 및 라이브러리 함수의 진입 진출 지점의 명령 코드를 Ptrace를 이용하여 중단점과 교체한다. (3) 대상 응용 프로그램을 구동시킨다. (4) 각 사용자 정의 함수 및 라이브러리 함수의 진입 및 진출 지점에 중단점이 걸려 있는 대상 응용 프로그램은 중단점에 걸리게 되면 실행을 멈추고 부모 프로세스인 DUT에게 보고한다. (5) 보고를 받은 DUT 는 타임 스탬프를 이용하여 사용자 정의 함수 및 라이브러리 함수로 진입 및 진출을 한 시점을 기록한다. (6) 기록 완료 후 DUT는 자식 프로세스의 해당 사용자 정의 함수 및 라이브러리 함수의 중단점을 원래 가지고 있던 명령 코드로 교체하여 대상 프로그램이 정상적으로 구동되도록 한다. (7) DUT는 멈춰 있는 대상 응용 프로그램을 다시 구동시킨다. (8) 대상 응용 프로그램의 수행 종료 후에 각 사용자 정의 함수 및 라이브러리 함수의 진입 및 진출 간의 시간 간격을 계산하여 각 함수가 소요한 시간을 계산한다.

DUT의 성능 측정 정보를 통해서 사용자 정의 함수와 공유 라이브러리의 함수의 호출 피호출 관계와 각 함수의 진입 및 진출 시점을 파악할 수 있고, 각 함수의 실행간 소요시간을 알아낼 수 있다.

4.2.2 동적 커널 영역 추적기(DKT)

DKT는 시스템 호출 계층과 커널 이벤트 계층의 정보를 수집하기 위하여, LTTng(Linux Trace Toolkit Next Generation)[28]를 수정 사용하였다. DKT를 이용하면 어플리케이션 구동간 발생하는 모든 시스템 호출 함수 및 커널 이벤트에 대한 성능을 파악할 수 있다.

사용자는 측정하고자 하는 커널 영역의 시스템 호출과 이벤트의 처리 부 소스 코드 안에 선택적으로 측정 코드를 삽입하여, 해당 시스템 호출 수행 시 및 커널 이벤트 처리 시 소요된 시간과 시스템 호출의 호출 횟수 및 커널 이벤트 발생 횟수 등에 대한 통계 수치 등을 알 수 있다. 그림 4와 같이 DKT의 동작은 크게 추적을 하기 위한 준비 과정과 추적 과정으로 나눌 수 있다. 추적을 위한 준비 과정을 위해서 본 연구에서는 커널에서

발생하는 각 이벤트에 대한 정보를 모두 분석하여 요구되는 이벤트에 대해서 선택적으로 추적이 가능하도록 분류시키고 자동화하였다. 이것은 커널 이벤트 추적 시 불필요한 이벤트에 대한 프로파일링으로 발생하는 성능 측정 오버헤드를 방지시켜준다. DKT의 추적 과정은 다음 같다. (1) DKT로 커널을 추적하기 위해서 측정이 필요한 위치에 마커(Marker)라는 측정용 코드를 삽입하고 커널을 생성한다. (2) 사용자 영역의 DKT컨트롤러를 이용하여 DKT 데몬을 실행시켜, (3) DKT 데몬이 DKT 를 실행 시키면 (4) DKT는 커널의 마커가 삽입된 이벤트 처리부가 실행될 때 마커에 의한 이벤트 호출 정보를 수집하여 RelayFS라는 환형 버퍼에 저장한다. (5) 이벤트 정보로 환형 버퍼가 가득 차게 되면, DKT가 디버그 파일시스템에 파일의 형태로 정보들을 저장하게 된다. (6) XML형태로 생성된 분석 결과는 호스트 시스템으로 전달되어, 결과 데이터에 대한 이해를 돕기 위해 본 연구에서 구현된 RA를 통해서 성능 분석 결과를 확인할 수 있다.

DKT로부터 얻는 성능 분석 결과에는 성능 분석을 하고자 하는 대상 응용 프로그램 외에도 다른 프로세스에 의한 시스템 호출이나 커널 이벤트가 발생한 것을 확인할 수 있다. 해당 응용 프로그램의 성능 분석 결과와 나머지를 구분하기 위해서 본 연구에서는 시스템 호출 및 커널 이벤트를 발생시킨 각 프로세스의 아이디(PID)를 출력할 수 있도록 마커를 수정하였다. 이로 인하여 각 프로세스의 성능 분석 결과를 구분할 수 있다.

DKT의 성능 분석 결과를 통해서 시스템 호출 함수와 커널 이벤트의 호출 피호출 관계와 각 함수의 진입 및 진출 시점을 알 수 있고, 이 정보를 통해서 각 함수의 실행간 소요시간을 알아낼 수 있다.

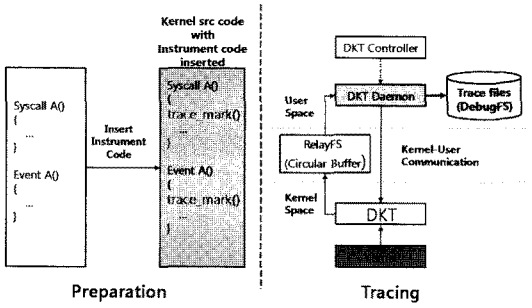


그림 4 동적 커널 영역 추적기(DKT)의 동작원리

4.3 DUT와 DKT의 연동

응용 어플리케이션 구동간의 실행 흐름을 사용자 함수부터 커널 이벤트까지 탑-다운(top-down) 방식의 하나의 흐름으로 파악하기 위해서는 DUT와 DKT로부터

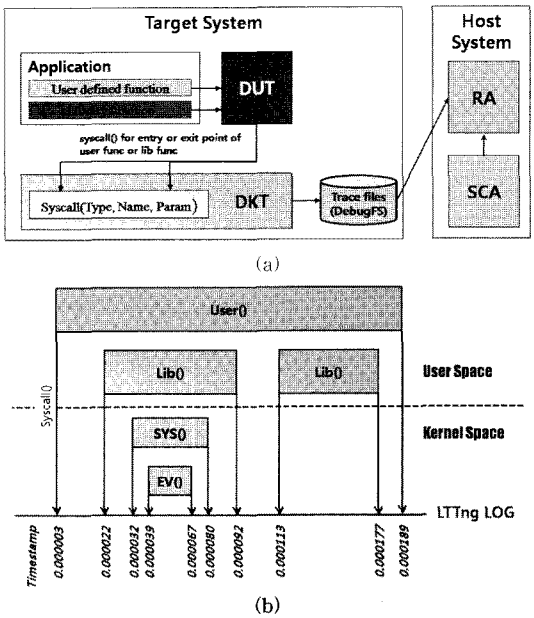


그림 5 DUT와 DKT의 연동원리

얻는 사용자 영역과 커널 영역에 대한 성능 분석 결과를 일관성 있고 각 계층과 계층간의 연관관계가 성립되도록 통합시켜야 한다.

이를 위해서 그림 5(a)와 같이 응용 프로그램의 성능 분석간에 각 계층의 성능 분석 도구들(DUT, DKT)을 동시에 구동 시켰다. 그리고 함수들의 호출, 피호출의 연관관계를 부여하기 위하여 DUT 통해서 사용자 영역 함수 호출 시점 기록시 본 연구에서 정의한 시스템 호출 함수를 호출하여 사용자 영역 함수들의 호출 시점을 DKT로 추적할 수 있도록 하였다. 이것은 그림 5(b)와 같이 DKT 를 통해서 얻는 성능 분석 결과 데이터만을 이용하여 소프트웨어 전 계층의 각 함수들의 호출, 피호출 관계와 각 함수들의 실행 소요 시간을 파악할 수 있게 한다.

4.4 정적 호출 그래프 분석기(SCA)

SCA는 분석대상 프로그램의 실행 가능 경로(executable path)를 트리구조로 보여준다.

SCA가 보여줄 수 있는 경로의 수준은 사용자 정의 함수와 라이브러리 함수 수준이며, 그림 6은 SCA가 실행파일로부터 실행 가능 경로 트리를 생성하는 과정을 보여준다. SCA는 리눅스 기본 유틸리티인 objdump, nm, readelf 유틸리티를 이용하는데, objdump로부터 실행프로그램의 어셈블리 코드를, nm으로부터 실행프로그램의 사용자 정의 함수 목록을, readelf로부터 실행프로그램의 라이브러리 함수 목록을 각각 생성한다. 그림 6과 같이 프로그램의 어셈블리코드 중 분기명령의 피연

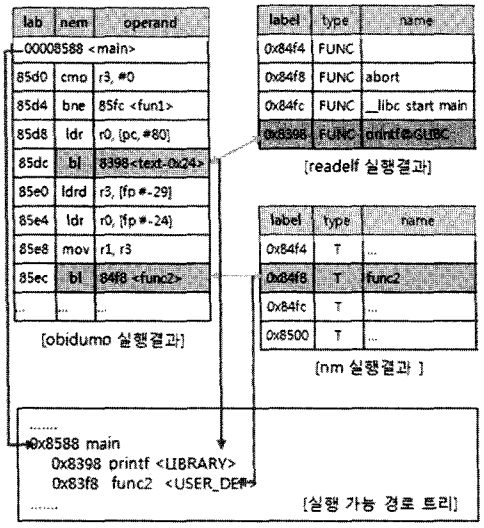


그림 6 정적 호출 그래프 분석기의 동작원리

산자들을 사용자정의함수 또는 라이브러리 함수 이름과 사상(Mapping)함으로써, 프로그램의 실행 가능 경로 트리를 생성한다.

SCA가 생성한 실행 가능 경로 트리는 동적 실행 추적기가 생성한 실제 실행 경로와 비교되어 소스코드 상에서 실제로 호출되지 않은 사용자 정의 함수들을 파악하게 해준다. 이는 소스코드상의 불필요한 부분을 찾는데 활용된다.

4.5 결과 분석기(RA)

RA는 각 성능분석 모듈들이 생성한 결과물들을 종합하여 보여주는 역할을 수행한다. RA는 그래픽 사용자 인터페이스(Graphic User Interface)와 파서(Parser)로 구성이 된다. 그래픽 사용자 인터페이스는 각각의 성능 분석모듈들이 생성한 결과물을 그래프, 트리, 테이블 등의 형태로 보여줌으로써, 사용자가 성능분석결과를 쉽게 이해하도록 돕는다. 파서는 타겟 시스템에 대한 성능 분석 결과는 소프트웨어 계층적으로 분류하고 GUI 컴포넌트가 인식할 수 있는 형태로 변환시켜준다.

5. 실 험

5.1 실험 환경

실험 대상 플랫폼으로 PXA270 기반의 임베디드 참조 플랫폼을 타겟 시스템으로 선정하였다. 그림 7은 실험대상 플래폼의 실제모습이다. 이 타겟 시스템에 제안하는 프레임워크를 탑재한 리눅스 커널 2.6.26.3 버전, 그리고 gtk+ 기반 스마트폰 소프트웨어 플랫폼인 GPE[29] 상에서 여러 응용에 대한 소프트웨어 계층 상에서의 성능 분석을 수행하였다. 호스트 시스템으로는 Ubuntu8.04-

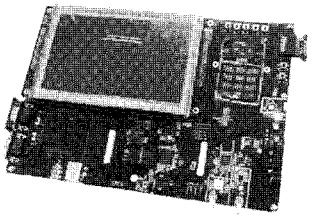


그림 7 PXA270 기반의 임베디드 참조 플랫폼

desktop을 운영체제로 사용하는 x86 PC에서 eclipse 기반의 GUI를 통해 성능 분석 결과를 확인할 수 있도록 하였다.

타겟 시스템의 루트 파일 시스템은 NOR 플래시 메모리에 GPE 파일 시스템을 jffs2 포맷으로 탑재하였으며, 유저 파일 시스템은 NAND 플래시 메모리를 jffs2, ubifs, yaffs2 포맷으로 탑재하여 사용하였다.

첫 번째 실험 대상 응용 프로그램으로는 벤치마크 툴에 대한 성능을 분석하기 위하여 Mibench[30]의 Djpeg을 선정, Jpeg 이미지 디코딩을 수행할 때의 성능을 분석하였다.

두 번째 실험 대상 응용 프로그램으로는, 첫 번째 실험 결과에서 나타난 성능 병목인 NAND 플래시 메모리의 성능을 분석하기 위하여, 쓰기 연산을 수행하는 응용 프로그램이 다양한 NAND 플래시 메모리 포맷에서 실행될 때의 성능을 분석하였다.

5.2 Mibench의 Djpeg에 대한 성능 분석

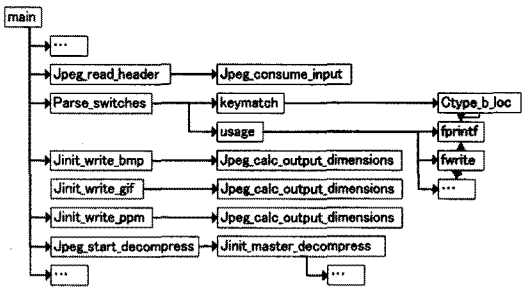
Djpeg을 이용하여 256x256 사이즈 jpeg 이미지를 ppm 이미지로 디코딩을 수행할 때의 소프트웨어 계층 구조를 고려한 성능 분석을 수행하였다. 사용자 정의 함수 및 라이브러리 함수에 대한 실행하기 전 실행 가능 경로 분석 및 실제 실행 경로 분석, 각 소프트웨어 계층별 함수에서의 소요시간, 소프트웨어 계층간 실행 흐름 분석을 통해 성능을 분석하였다.

또한, Jpeg 이미지 디코딩은 NAND 플래시 메모리를 jffs2 파일 시스템 포맷으로 사용한 유저 파일 시스템상에서 수행하였다.

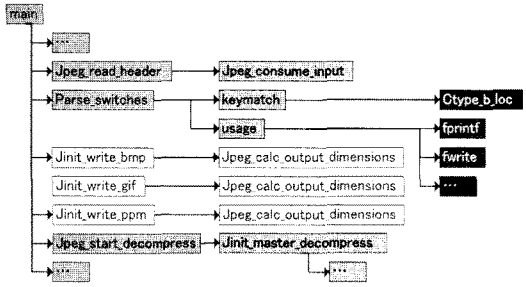
5.2.1 실행 가능 경로 대비 실제 실행 경로 분석

제안하는 정적 분석 기법과 실제 실행 시 동적으로 측정된 결과를 조합하여, 함수의 실행 가능 경로 대비 실제 실행 경로를 도출하였다. 사용자 정의 함수 및 라이브러리 함수에 대한 실행 경로를 트리의 형태로 나타낼 수 있다.

실험 결과, 그림 8과 같이 main 함수에서 실행 명령을 파싱하여, 여러 이미지 포맷 중 jpeg 이미지를 디코딩하는 함수들로 분기하며 실행되는 것을 확인할 수 있다.



(a) JPEG Decoder의 실행 가능 경로



(b) JPEG Decoder의 실제 실행 경로

실행 가능한 사용자 정의 함수 / 실행 가능한 라이브러리 함수
실행된 사용자 정의 함수 / 실행된 라이브러리 함수

그림 8 Jpeg Decoder의 실행 가능 경로 대비 실제 실행 경로 분석

5.2.2 소프트웨어 계층별 성능 분석

타겟 시스템의 각 소프트웨어 계층의 추적기를 통해 동적으로 측정한 결과를 호스트 시스템으로 전달 받고 변환하여, 각 소프트웨어 계층별 함수 및 이벤트별 소요 시간의 평균을 나타내었다.

분석 결과, 그림 9에서 알 수 있듯이 사용자 정의 함수에서는 디코딩 연산을 처리하는 decode_mcu, jpeg_huff_code 등의 함수에서의 소요시간만큼 디코딩된 이미지 값을 저장하기 위한 쓰기연산을 수행하는 jpeg_fill_bit_buffer, put_pixel_rows와 같은 함수의 소요시간이 큰 비중을 차지함을 알 수 있다.

마찬가지로, 라이브러리나 시스템 호출, 커널 이벤트의 경우에도 쓰기 요청의 처리부인 fwrite, write, fs_write와 같은 함수의 소요 시간이 큰 비중을 차지하는 것을 확인할 있다.

결론적으로, 현재 플랫폼에서 Jpeg 디코딩을 수행할 경우 디코딩을 수행하기 위한 연산의 소요 시간만큼 쓰기요청에 대한 처리의 소요 시간이 모든 소프트웨어 계층에 걸쳐 큰 비중을 차지하는 것을 확인할 수 있다.

5.2.3 소프트웨어 계층간 성능 분석

성능 병목을 확인할 수 있도록 하기 위해, 디코딩 된 값을 행 단위로 버퍼에 쓰는 부분인 put_pixel_rows 사용자 정의 함수의 두 번째 호출시의 실행 흐름을 그림 10

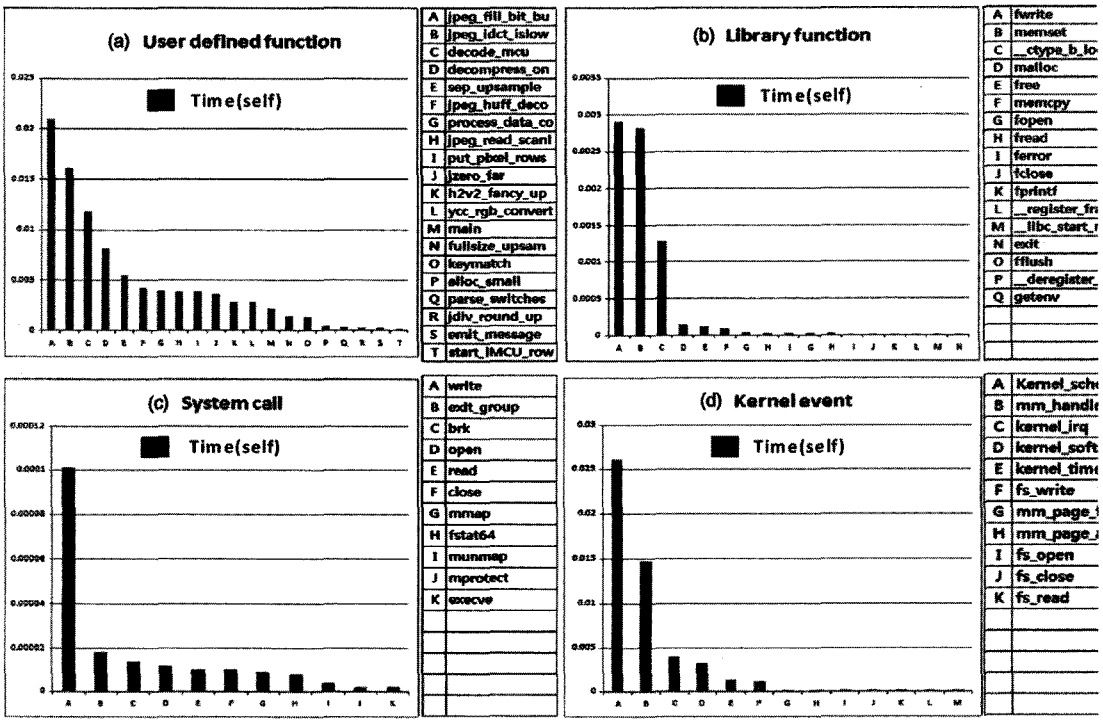


그림 9 DJpeg의 소프트웨어 계층별 성능 분석

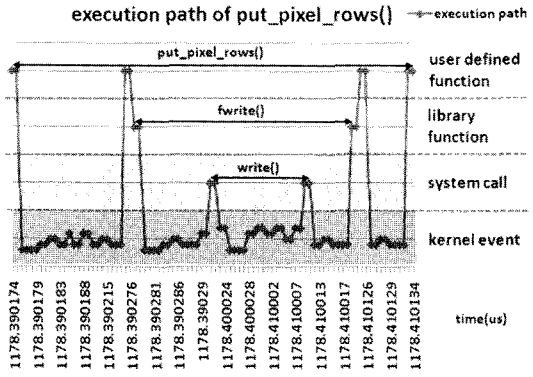


그림 10 put_pixel_rows 함수의 소프트웨어 계층간 실행 흐름 분석

과 같이 나타내었다. 실험 결과, 쓰기연산을 수행하는 fwrite 라이브러리 함수나 write 시스템 호출이 많은 차지하는 것을 확인할 수 있다.

5.3 NAND 플래시 메모리의 쓰기 성능 분석

이전 실험에서 쓰기 연산에 대한 지연이 성능 병목을 야기하였으므로, 이 문제를 개선할 수 있도록 NAND 플래시 메모리의 다양한 파일 시스템 포맷에 대한 쓰기 연산의 성능을 비교하였다.

NAND 플래시 메모리에서 사용 가능한 파일 시스템 포맷 중 Jffs2, ubifs, yaffs2 에 대한 성능을 분석하였다. 즉, 쓰기연산을 반복적으로 수행하는 같은 응용 프로그램의 쓰기 요청에 대한 각 파일 시스템 포맷의 처리 방법에 따라 어떻게 다른 실행 흐름과 성능 차이를 보이는지를 실험을 통해 확인하였다.

응용 프로그램은 fopen, fprintf, fclose 와 같은 라이브러리 함수를 통해 파일 시스템에 접근하여 쓰기 연산을 수행하도록 작성하였다.

응용 프로그램이 각각의 세 파일 시스템에서 실행 될 때, 모두 fprintf 함수를 통해 쓰기 요청된 데이터를 버퍼에 저장하고 있다가, 버퍼가 찼을 때나 close 요청으로 쓰기 연산이 끝났을 때 실제 물리 메모리에 쓰기 연산을 수행한다. 즉, 이 응용 프로그램의 경우 fclose 라이브러리 함수가 호출되는 시점에 파일 시스템에 write, close 시스템 호출을 수행한다.

5.3.1 소프트웨어 계층별 성능 분석

실험 결과, 각각 파일 시스템 기법에 따른 각 라이브러리 함수의 요청에 대한 수행 시간이 그림 11과 같이 다른 것을 확인할 수 있다. 즉, 같은 요청에 대한 커널 내부의 시스템 호출이나 이벤트 처리부의 다른 처리 방법에 따라서 큰 성능 차이를 보인다. Jffs2에 비하여, NAND 플래시를 위한 저널링을 수행한 yaffs2가 쓰기 성능이 좋으며, B+ 트리 인덱싱이나 TNC(Tree Node Cache)를 통해 적은 양의 SRAM을 사용하는 ubifs는 약 5배 빠른 쓰기 성능을 보이는 것을 확인할 수 있다[31,32].

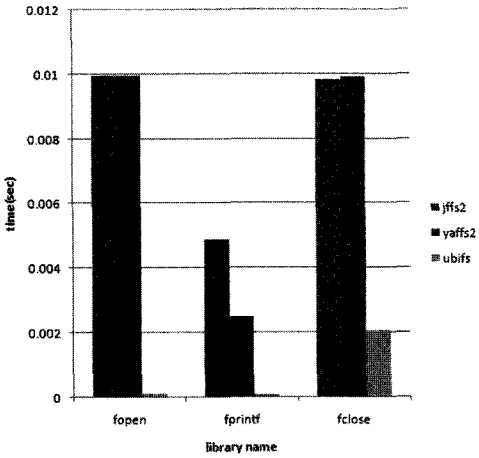


그림 11 라이브러리 함수의 평균 소요 시간

5.3.2 소프트웨어 계층간 성능 분석

이 응용 프로그램을 실행할 경우 실제 write 시스템 호출이 처리되는 부분은 fclose 라이브러리 함수가 호출된 시점이기 때문에, fclose 라이브러리 함수에 대한 각 파일 시스템의 실행 흐름과 응답 시간이 다른 것을 그림 12에서 확인할 수 있다.

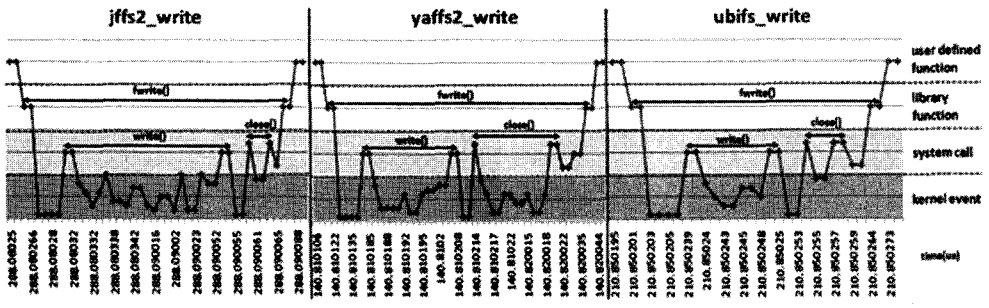


그림 12 NAND 플래시 메모리의 파일 시스템 형태에 따른 fclose 함수의 실행 흐름 분석

6. 결론 및 향후 연구

본 논문에서는 복잡한 계층 구조를 가지는 리눅스 기반 임베디드 시스템의 모든 소프트웨어 계층을 고려하여 성능을 분석할 수 있는 소프트웨어 계층적 성능 분석 프레임워크를 제안하였다.

기존의 많은 단일 소프트웨어 계층 성능 분석 도구나 다른 상용 혹은 오픈 소스 기반의 통합 성능 분석도구들이 할 수 없었던 모든 소프트웨어 계층을 고려한 성능 분석을 제안한 프레임워크를 통해 사용자 정의 함수 계층부터 커널 이벤트 계층까지 계층적으로 분석할 수 있다.

본 프레임워크는 동적 측정형 기법과 정적 측정형 기법 그리고 정적 호출 관계 분석 기술을 이용하여, 응용 프로그램의 실행 가능 경로 중 실제 실행 경로를 추적하고, 실제 실행 경로의 사용자 정의 함수, 라이브러리 함수, 시스템 호출, 커널 이벤트 계층별 호출 관계를 분석하였다.

이러한 장점을 가지는 반면, 본 연구는 몇 가지 단점을 가지고 있다. 첫째로, 성능분석을 위한 성능 오버헤드가 크다. 특히 DUT의 경우 ptrace 기법을 사용하는 Ltrace의 동작구조를 그대로 취하기 때문에 다른 부분에 비해 큰 성능 오버헤드가 발생한다[33,34]. 둘째로, 사용자 정의 함수나 라이브러리 함수 중 함수 포인터를 사용한 경우는 추적이 불가능하다. 정적으로 실행 파일을 분석할 때 대상 심볼의 주소만 가지고 있기 때문에 해당 함수를 분석할 수 없다.

향후 연구 과제는 이러한 한계를 개선하는 방향으로 진행할 것이다. 먼저, 추적 상의 오버헤드를 줄여야 한다. 특히 DUT의 오버헤드를 줄이기 위하여, 성능분석 간에 사용자 정의 함수나 라이브러리 함수의 추적 깊이나 영역을 선택할 수 있도록 구현하고, 오버헤드가 큰 Ptrace 기법 대신 Pin[20]등에 적용된 좀더 가벼운 동적 측정형 기법을 적용할 예정이다. 또한 성능 카운터를 이용하여 함수의 발생을 추후 분석하는 형태로 함수 포인터를 추적하지 못하는 현재의 문제점을 개선할 수 있을 것이다.

마지막으로, 시스템 자원에 대한 성능 분석 기능을 추가할 계획이다. Valgrind와 같이 메모리 에뮬레이션 기술을 통해 메모리 사용량을 분석하고, Proc파일 시스템과 top등의 도구들을 활용하여 CPU 사용률을 분석한다. 또한 시스템 자원 사용에 대한 분석 정보들을 응용 프로그램의 실행흐름과 매칭하여 특정 시점의 시스템 자원 상태를 분석할 수 있도록 연구할 것이다.

참 고 문 헌

[1] Ed Burnette, Hello, Android, second edition, Pragmatic Bookshelf, 2009.

[2] David Lefty Schlesinger, Architectural Principles and Overview of the LiMo Platform, OSCON, 2008.

[3] S. Lim, Feature Story: Android Platform, Embedded World 2009. 8, pp.52-58.

[4] Graham, S., Kessler, P., and McKusick, M. gprof: A Call Graph Execution Profiler. In Proceedings of ACM SIGPLAN Symposium on Compiler Construction. ACM, 1982.

[5] Strace, Ltrace: http://www.gnu.org/software/libc/resources.html

[6] http://www.windriver.com/products/workbench/

[7] K. Kantou et al, Performance Measurement/Analysis Tool "mevalet," *NEC technical journal*, vol.2, no.2, 2007.

[8] Steve Best, Linux Debugging and Performance Tuning: Tips and Techniques, Prentice Hall, 2005.

[9] Mark Wilding, Self-Service Linux: Determining Problems and Finding Solutions, Prentice Hall, 2005.

[10] Raj Jain, The Art of Computer Systems Performance Analysis Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley & Sons, Inc., 1991.

[11] N. Nethercote, Dynamic Binary Analysis and Instrumentation. PhD Thesis, University of Cambridge, Mass., USA, 2006.

[12] http://kprof.sourceforge.net/

[13] http://lttng.org/

[14] Orran Krieger et al, K42: building a complete operating system, ACM SIGOPS Operating Systems Review archive, vol.40, Issue 4 (October 2006).

[15] FC Eigler, R Hat, Problem solving with systemtap, Proceedings of the Ottawa Linux Symposium, 2006.

[16] http://valgrind.org/

[17] http://www.gnu.org/software/gdb/

[18] Richard Moore, A universal dynamic trace for Linux and other operating systems, In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference June 2001.

[19] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal, Dynamic Instrumentation of Production Systems, In Proceedings of the 2004 USENIX Annual Technical Conference.

[20] Chi-Keung Luk Robert Cohn Robert Muth Harish Patil Artur Klauser Geoff Lowney StevenWallace Vijay Janapa Reddi Kim Hazelwood, Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[21] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. The

Paradyn ParallelPerformance Measurement Tools. IEEE Computer (28) 11, (Nov., 1995).
[22] Pradeep Padala, Playing with ptrace, Part I, Linux Journal 2002.
[23] J. Levon and P. Elie. Oprofile: A system profiler for linux. http://oprofile.sf.net, September 2004.
[24] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[25] http://www.qplus.or.kr/
[26] http://www.mvista.com/product_detail_devrocket.php
[27] R.R. Branco, Ltrace Internals, Ottawa Linux Symposium, 2007.
[28] M. Desnoyer, M.R. Dagenais, The LTTng Tracer: A low impact performance and behavior monitor for GNU/Linux, Ottawa Linux Symposium, 2007.
[29] http://gpe.handhelds.org.
[30] M.R. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite, In IEEE 4th Workshop on Workload Characterization, Dec. 2001.
[31] http://en.wikipedia.org/wiki/YAFFS
[32] http://www.linux-mtd.infradead.org/doc/ubifs.html
[33] J.Keniston, A.Mavinakayanahalli, P. Panchamukhi, Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps, Ottawa Linux Symposium, 2007.
[34] Francis M. David, Jeffrey C. Carlyle, Roy H. Campbell, Context Switch Overheads for Linux on ARM Platforms, Proceedings of the 2007 workshop on Experimental computer science.

곽 상 헌
2008년 국민대학교 컴퓨터공학(학사). 2010년 국민대학교 컴퓨터공학(석사). 관심분야는 시스템 성능분석, 메모리 프로파일링

이 남 승
2009년 국민대학교 컴퓨터공학(학사). 2010년~현재 국민대학교 컴퓨터공학과 석사과정 재학중. 관심분야는 시스템 성능분석, 가상화, 신뢰성 컴퓨팅

이 호 림
2009년 국민대학교 컴퓨터공학(학사). 2010년~현재 국민대학교 컴퓨터공학과 석사과정 재학중. 관심분야는 시스템 성능분석, 가상화, 신뢰성 컴퓨팅

임 성 수
1993년 서울대학교 컴퓨터공학(학사). 1995년 서울대학교 컴퓨터공학(석사). 2002년 서울대학교 컴퓨터공학(박사). 2001년~2004년 팜팜테크(주) 기술총괄이사. 2004년~현재 국민대학교 컴퓨터공학과 부교수. 관심분야는 신뢰성 컴퓨팅, 가상화, 실시간 시스템, 고성능 임베디드 시스템