임베디드 커널 추적 도구를 이용한 임베디드 시스템 성능 측정 기법

(A Scheme of Embedded System Performance Evaluations Using Embedded Kernel Trace Toolkit)

배 지 혜 † 윤 남 식 † 박 윤 용 ††
(Ji-Hye Bae) (Nam-Sik Yoon) (Yoon-Young Park)

요 약 임베디드 시스템은 정보, 교육, 산업, 서비스 등의 많은 분야에서 인간 중심의 서비스를 제공하고 있으며 이러한 임베디드 시스템의 관리, 제어, 통제 및 테스트를 위한 모니터링 프로그램이 다양하게 개발되고 있다. 모니터링을 하기 위해서는 여러 가지 커널 추적 도구를 사용하는데 본 논문에서는 기존의 커널 추적 도구와 비교하여 임베디드 시스템에 초점을 맞춘 커널 추적 도구인 ETT^{plus}를 제시하며, 임베디드 타겟과 호스트와의 추적 데이타 전송 방법을 제시하고자 한다. ETT^{plus}는 기존의 추적 도구가 가지고 있는 어려운 커널 패치 문제, 파일 시스템 사용 의존성 문제 등에 대한 해결 방안을 제공하고 있으며 ETT^{plus}를 이용하여 시스템 콜 수행 시간이나 네트워크 데이타 전송 시간과 같은 임베디드 시스템의 성능 측정 비교에 대한 설계 및 분석 결과를 제시한다.

키워드 : 임베디드 시스템, 모니터링, 커널 추적 도구

Abstract The Embedded system provides human-centric services in many fields of education, information, industry and service, and monitoring programs have been variously developed for managing, controlling and testing for these embedded systems. Currently, many kernel trace toolkits are being used for monitoring. These trace toolkits are so complicate that we present ETT^{plus}, our simple and explicit embedded kernel trace toolkit, for embedded systems and describe the transmission method for trace data between the embedded target system and the host system. ETT^{plus} provides the solution to solve the problems such as the difficult kernel patch and file system dependency in existing kernel trace toolkits like LTT. Furthermore, we present the experimental results about embedded system performance evaluations such as system call execute time or network data transmission time by using ETT^{plus}.

Key words : Embedded System, Monitoring, Kernel Trace Toolkit

· 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음(IITA-2007-C1090-0701-0020)

† 학생회원 : 선문대학교 컴퓨터정보학부
angdoo98@sunmoon.ac.kr
windsong@sunmoon.ac.kr
†† 정 회 원 : 선문대학교 컴퓨터정보학부 교수
yypark@sunmoon.ac.kr

논문접수 : 2007년 9월 6일
심사완료 : 2007년 11월 28일

: 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지: 컴퓨팅의 실제 및 레터 제13권 제7호(2007.12)
Copyright@2007 한국정보과학회

1. 서 론

엄청난 속도로 발전하는 반도체 기술과 컴퓨터 기술, 네트워크 기술은 1980년대부터 시작된 PC시대를 이끌어냈으며, 2000년대 들어서는 임베디드 시스템을 주축으로 하는 포스트(post) PC시대를 형성하게 되었다. 또한, 이동컴퓨팅(mobile computing)이 구현되면서 이전의 사람이 컴퓨터를 찾아가던 형식에서 탈피하여 사람이 지역이나 시간의 제한 없이 컴퓨터를 사용하고 싶을 때 그 즉시 서비스를 받을 수 있게 되었다.

이동컴퓨팅은 유비쿼터스(ubiquitous) 컴퓨팅, 편재형(pervasive) 컴퓨팅, 상황기반(context-based) 컴퓨팅 등의 이름으로 우리나라에서는 물론 세계 각국에서 매우 활발하게 연구되고 있다. 이동 컴퓨팅의 주요 구성요

소들이 바로 임베디드 시스템이다. 임베디드 시스템은 정보, 교육, 산업, 서비스 등의 많은 분야에서 인간 중심의 서비스를 제공하게 되고 이러한 맥락에서 21세기는 임베디드 시스템의 시대라고 많은 사람들이 예상하고 있다[1].

임베디드 시스템이란 마이크로프로세서 혹은 마이크로 컨트롤러를 내장하여 원래 제작자가 의도했던 특정 기능만을 수행하도록 제작된 장치를 말한다. 이러한 임베디드 시스템에서 사용되는 기본적인 커널을 임베디드 커널 이라하며 대표적으로 임베디드 리눅스 커널 등을 사용한다. 이러한 임베디드 커널의 분석 및 성능 평가를 위해 임베디드 시스템은 시험 및 상용화 단계에서 사용하기 위한 커널 추적 도구를 사용하고 있다.

커널 추적 도구는 디버깅, 테스팅 그리고 응용 프로그램들의 성능 평가 및 시스템 제어, 관리를 담당하는 시스템 모니터링을 위한 커널 분석 및 성능평가를 지원한다. 이러한 커널 추적 도구는 시스템의 상태 정보 또는 프로세스들의 정보를 동적으로 수집, 해석, 표시하기 때문에 임베디드 시스템을 관리하고 테스팅하기 위해서 필요하다[2].

그러나 기존의 커널 추적 도구를 사용하기 위해서는 커널 버전의 호환성 문제, 커널 패치의 어려움, 복잡한 데이타 분석방법 등의 문제점이 제기되고 있다. 이런 이유로, 본 논문에서는 임베디드 커널을 비교적 간단한 방법으로 직접 패치하여 원하는 커널 분석 정보만 추출해 내어 데이타를 GUI기반의 호스트 환경으로 가장 효율적인 방법으로 전송하는 구조를 제시하고자 한다. 본 논문에서 제시하고 있는 추적 도구인 ETI^{plus}는 기존의 추적 툴킷이 가지고 있는 어려운 커널 패치 문제, 파일 시스템 사용 의존성 문제 등에 대한 해결 방안을 제공하고 있다. 또한 ETI^{plus}를 이용하여 시스템 콜 실행 시간 측정과 네트워크 데이타 전송 측정과 같은 임베디드 시스템의 성능 평가에 관한 설계 및 측정결과를 제시하고자 한다.

2장에서는 현존하는 커널 추적 도구에 대해 기술하고자 하며 3장에서는 본 논문에서 제시하고자 하는 커널 추적 도구의 설계 및 구현 부분을 기술한다. 4장에서는 이러한 도구를 기반으로 임베디드 시스템 성능 측정 설계에 대해 설명하며 5장에서는 앞에서 제시한 성능 측정 설계를 이용하여 실제 임베디드 시스템의 성능 평가에 대해 기술한다. 마지막으로 6장과 7장에서는 최종적인 결론 및 향후 연구 과제 등을 기술한다.

2. 관련 연구

모든 리눅스 시스템에서는 많은 프로세스가 자원을 얻기 위해서 경쟁한다. 각 프로세스가 시스템 적재에 미치는 영향을 정량화하는 작업은 균형잡히고, 응답성이 좋은 시스템을 구성하는데 상당히 중요하다. 리눅스의 각 프로세스가 시스템에 미치는 영향을 분석하는 데는 몇 가지 기본적인 방법이 있는데 그 중에서 /proc 파일 시스템을 이용하는 방법과 LTT 및 LTTng를 이용하는 방법을 설명한다.

2.1 /proc 파일 시스템을 이용한 프로파일링

/proc 파일 시스템은 커널 내부의 데이타 구조와 시스템 전반에 대한 정보를 제공하는 가상 파일시스템이다. 이 정보 중 일부는 시스템의 작동 시간 동안 정적으로 유지되기도 하지만, 그 외 프로세스 실행 시간과 같은 몇몇 정보는 매 클럭 틱마다 커널에 의해 샘플링되어 동적으로 관리된다. /proc 디렉토리에서 정보를 얻어내는 전통적인 유틸리티로는 ps와 top같은 명령을 포함하는 procps 패키지를 들 수 있다. 현재 procps 패키지는 버전 두 개가 독립적으로 관리되고 있는데, 첫째는 릭 반 리엘(Rik van Riel)에 의해 관리되는 패키지로 http://surriel.com/procps/ 에서 얻을 수 있고, 둘째는 앨버트 카하란(Albert Cahalan)이 관리하는 패키지로 http://procps.sourceforge.net/ 에서 배포된다. 현재 두 가지 패키지 중 어떤 것을 공식 procps로 인정할 것인지에 대한 토론이 진행되고 있는 상황이긴 하지만, 두 패키지 모두 패키지에 포함된 Makefile이 크로스 플랫폼에서 사용되기에 적절치 않다는 문제가 있다.

이런 procps 패키지를 임베디드 시스템 환경에서 사용하는 방법은 그리 쉽지 않기 때문에, 대체적으로 임베디드 장비의 BusyBox에 포함된 ps를 사용하는 방법을 쓴다. 이 ps명령은 procps 패키지에 포함된 것만큼 자세한 프로세스 통계치를 출력해주지는 않지만, 타겟에서 수행되는 소프트웨어에 대한 간단한 정보를 얻기에는 충분하다.

```
#ps
PID  Uid    VmSize   Stat   Command
 1    0      820      S      init
 2    0               S      [keventd]
 3    0               S      [kswapd]
 4    0               S      [kreclaimd]
 5    0               S      [bdflush]
 6    0               S      [kupdated]
 7    0               S      [mtdblockd]
 8    0               S      [rpciod]
16    0      816      S      -sh
17    0      816      R      ps aux
```

그림 1. “ps” 명령어 실행화면

만약 이 정보만으로 부족한 경우에는 /proc 디렉토리에서 각 프로세스 항목을 직접 살펴봄으로써 원하는 정보를 얻을 수 있다.

2.2 LTT를 이용한 프로파일링

현재 가장 널리 사용되고 있는 리눅스의 주요 시스템

추적 유틸리티중의 하나는 리눅스 추적 툴킷(LTT, Linux Trace Toolkit)이다. 리눅스 추적 툴킷은 중요한 시스템 이벤트를 기록하고 실행중인 프로세스의 서브셋을 분석하는데 사용된다. strace와 같은 다른 추적 유틸리티가 ptrace() 시스템 콜을 기반으로 추적 메커니즘을 사용하는데 비해 LTT는 직접 커널을 패치하여 주요 커널 서브시스템의 작동을 중개하는 방식을 사용한다. 이 과정에서 생성된 데이타는 추적 서브시스템(trace subsystem)이 취합한 후, 추적 데몬(trace daemon)으로 보내져서 디스크에 기록된다. 이런 전 과정은 시스템의 작동이나 성능에 악영향을 주지 않도록 설계되었다. 실제 추적 구조가 시스템의 성능에 미치는 영향을 테스트한 결과, 추적 기능을 사용하지 않을 경우에는 성능 손실이 거의 없었고, 가장 심한 추적 부하를 가한 경우에도 성능 손실이 2.5% 이하였다[3,4].

그림 2의 그래프는 LTT에서 제공하는 시각화 툴 중 하나인 "Event Graph"에 대한 화면으로 프로세스 간의 상호 작용을 보여준다. 화면의 왼쪽 창에 출력된 것은 시스템을 추적하는 동안 실행되었던 프로세스 목록이며, 이 항목의 맨 마지막 줄에는 항상 리눅스 커널이 표시된다. 화면의 오른쪽 창에는 시스템의 동작을 나타낸 그래프가 표시된다. 이 그래프의 수평축은 시간의 흐름을 나타내며, 수직 축은 시스템의 상태 전이를 나타낸다.

LTT는 중요한 시스템 정보를 기록하므로 시스템 동작에 대한 매우 상세한 수준의 정보를 추출해 낼 수 있다. 리눅스 시스템에서 기본적으로 제공하는 /proc의 정보가 커널의 표본치에 의해 생성되는 것임에 반해, LTT의 정보는 커널과 프로세스의 이벤트에 대한 정확한 기록으로부터 만들어진다. LTT는 프로세스 별 통계 자료와 시스템 통계 자료의 두 가지 통계치를 제공하며, 해당 태스크가 CPU 시간을 얼마나 얻었는지(time running) 그리고 그 중 얼마를 실제 응용 프로그램 코드의 실행에 소비했는지(time executing process code)를 볼 수 있다. 또한 각 프로세스 마다 실행된 시스템 콜의 개수 및 전체 실행 시간 등에 대한 정보를 제공하고 있다[3,4].

이처럼 LTT에서는 응용 프로그램과 커널의 상호 작용에서 문제가 발생할 경우 상호 작용을 추적(tracing)함으로써 문제를 해결해주는 방법을 제공하고 있다. LTT에서 제공하는 많은 추적 정보들은 추적 대상이 되는 타겟의 성능에 대한 다양한 측면의 데이타를 얻어내는 작업으로 타겟의 잠재 성능을 최대한 끌어내기 위해 매우 중요하며 중요한 시스템 정보를 기록하므로 시스템 동작에 대한 매우 상세한 수준의 정보를 추출해 낼 수 있다.

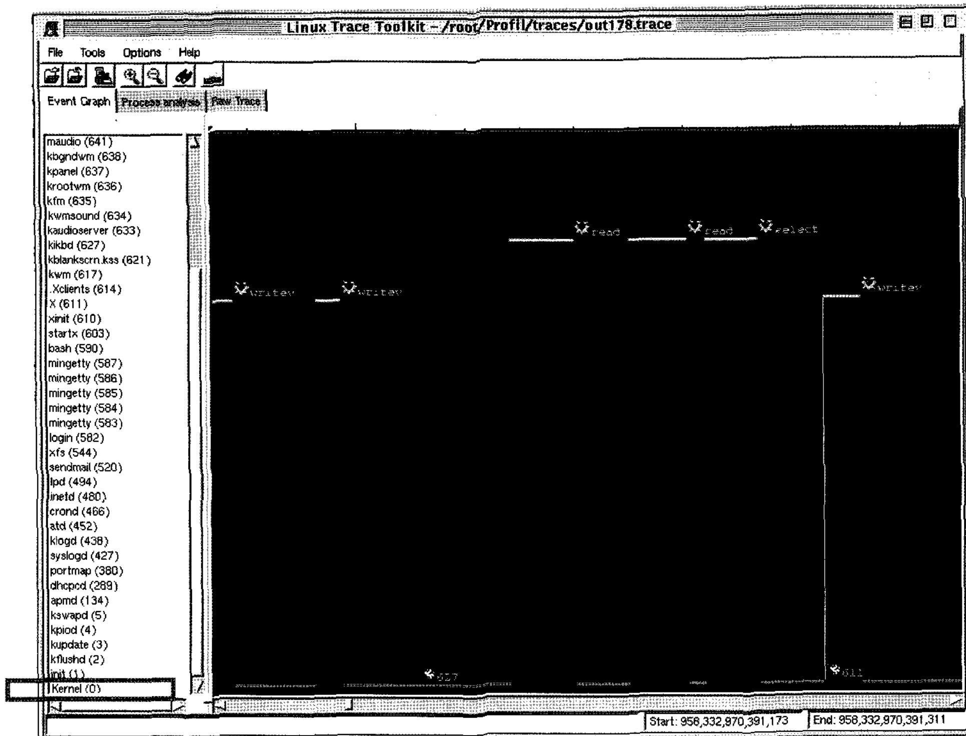


그림 2 LTT 추적 그래프의 예(Event Graph)

2.3 LTTng를 이용한 프로파일링

앞에서 소개한 LTT의 기능을 한층 더 강화하여 보다 많은 성능 평가 데이타를 추출하도록 새롭게 제시된 커널 추적 툴킷이 LTTng(LTT Next Generation)이다.

LTT의 재디자인이라고도 볼 수 있는 LTTng도 시스템의 작동에 많은 영향을 주지 않도록 설계 되었으며 독립적인 아키텍처로 구성되어 있다. 또한, LTTV라는 시각화 툴을 따로 제공하여 플러그인 기반의 모듈러 아키텍처를 지원하며 SMP Machine(Symmetric Multi-Processing)을 지원한다. 지원하는 이벤트 타입은 메타정보로 표현이 가능하며 한 번 추적을 할 때 마다 LTT에서의 비교적 적은 추적 양 한계에 비해 훨씬 많은 양의 추적을 할 수 있다[5,6].

그림 3은 LTTng의 LTTV(Linux Trace Toolkit Viewer)를 나타낸다. 크게 3부분으로 나뉠 수 있는데 현재 traceset에 대해 다중 통계치를 트리구조로 표현한 상단 왼쪽 부분과 각 프로세스마다 발생된 정보를 그래픽으로 나타내주는 상단 오른쪽의 그래프 부분, 그리고 각 이벤트 리스트에 대한 상세 정보를 나타내 주는 하단 부분이다[5,6].

그림 3에서 보면 LTT에서는 각각 따로 제공하는 기능인 “Event Graph”, “Process Analysis”, “Raw Trace”의 시각화 툴 기능을 LTTng에서는 하나의 화면으로 통합해서 제공하고 있는 것을 알 수 있다.

2.4 문제점 분석

LTT와 LTTng는 커널 추적을 하기 위한 대표적인 추적 도구이지만 이러한 커널 추적 도구의 몇 가지 문제점을 든다면 다음과 같다.

첫째, 커널 버전에 따라 각각 패치가 다른 커널 버전 호환성 문제가 있으며, 포팅이 필요한 임베디드 시스템에서는 장시간의 수동 패치 작업을 통해 커널 패치를 하여야 하므로 임베디드 OS에 적용하기가 상당히 어려운 점이 있다. 이를 위해 *ETT*^{plus}에서는 처음부터 임베디드 커널에 맞춘 패치를 제작하였으며 패치 부분이 복잡하지 않고 최소한 필요한 정보만을 추출하기 위해 간단한 패치 구조에 중점을 두었다.

둘째, 기존의 도구들은 기본적인 이벤트 정보를 활용 및 가공하여 많은 종류의 커널 정보가 제공되므로 사용자가 데이타를 분석할 때 가장 기본적인 성능평가 분석 정보만을 추출해 내기 어려운 문제점이 있다. *ETT*^{plus}에서는 프로세스 통계치를 제공하기 위한 최소한의 6가지 이벤트만을 구성하여 시각화 툴에서 이를 매우 간단하게 표현하고 있다.

셋째, 기존의 추적 도구들은 호스트 기반의 도구이며

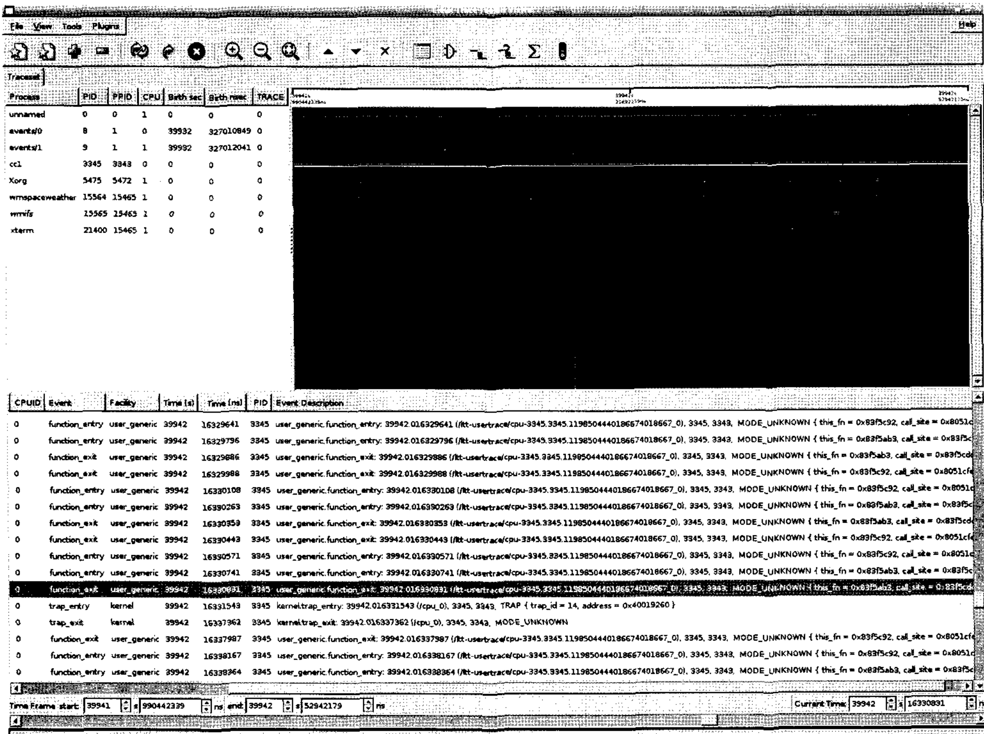


그림 3 LTTng 시각화 툴 뷰어

임베디드 용으로 제작된 도구가 따로 존재하지 않으므로 호스트와 타겟 사이의 통신 방법이 불명확하다. 또한, 기존의 시스템은 생성된 추적 데이타 파일을 저장하기 위해 타겟의 파일시스템이 반드시 존재하여야 한다. 이러한 문제점을 감안하여 ETT^{plus}에서는 추적 데이타 파일이 타겟의 파일시스템에 생성되지 않고 연결된 호스트 시스템으로 직접 추적 데이타를 전송하는 구조로 설계되어 있다.

마지막으로 기존의 커널 추적 도구들은 실시간성이 부족하다. 추적이 반드시 끝나야지 데이타 파일이 생성되는 구조를 가지고 있기 때문에 추적하는 동안에는 추적 내용을 알 수 없다는 문제가 있다. 이를 위해 본 논문에서는 추적이 완전히 끝나지 않아도 중간 추적 상황을 확인할 수 있도록 On-Line의 특성을 추가한 추적 시스템을 설계 중이다.

이와 같은 기존 도구들의 여러 문제점을 해결하기 위하여 본 논문에서는 간단하면서도 쉽게 핵심적인 커널 분석 정보만 추출해내는 커널 추적 도구를 제시하고자 한다.

3. 임베디드 커널 추적 도구 설계 및 구현

본 장에서는 본 논문에서 제공하는 임베디드 커널 추적 도구인 ETT^{plus}의 설계 및 구현에 대해 기술하고자 한다.

3.1 ETT^{plus} 설계

ETT^{plus}(Embedded kernel Trace Toolkit)는 본 논문에서 제시하고자 하는 커널 추적 도구이다. 다음 표 1은 과거 연구[7]에서 제시한 기법과 본 논문에서 제안한 기법의 비교를 한 것이다.

과거 연구[7]에서 제시되고 있는 ETT는 ETT^{plus}의 구버전으로 커널 추적 데이타 추출을 위해 임베디드 시스템과 호스트 사이에 구축된 NFS를 이용하였고 윈도우 기반의 단일 GUI 환경 툴을 제공하였으며, 본 논문에서 제시하는 ETT^{plus}는 이를 한층 업그레이드 하여 커널 추적 데이타 추출 과정에서 생성된 데이타를 임베디드 시스템의 파일 시스템을 사용하지 않고 직접 호스트 상으로 전송해주는 호스트/타겟 사이의 명확한 통신 방법을 제공한다. 또한, 커널 버전의 호환성 문제 및 어려운 수동 패치작업이 필요한 기존의 커널 추적 도구(예. LTT 또는 LTTng)와는 달리 ETT^{plus}에서는 꼭 필요한 정보만 추출해 내기 위해 최대한 커널 패치 작업을 최소화하였으며 이로 인해 추적으로 인한 간섭현상 또한 최소화된다. 호스트 시각화 툴은 윈도우 및 리눅스에서 호환되는 GTK+ 기반의 GUI 환경을 제공한다.

(1) ETT^{plus} 기본 구조

그림 4는 ETT^{plus}의 전체적인 구조를 나타내고 있다. 시스템은 타겟과 호스트 두 부분으로 나뉘며 실제로 추적데이타를 추출하고 전송하는 부분은 타겟 시스템에서 이루어지고 호스트 시스템에서는 GUI환경의 시각화 툴을 제공한다.

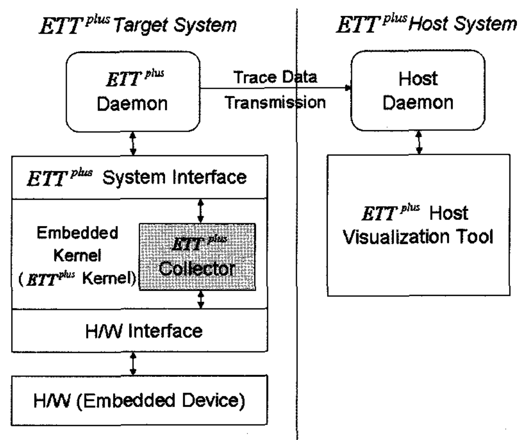


그림 4 ETT^{plus} 전체 구조

ETT^{plus} 콜렉터는 커널 내부에서 추적 데이타를 수집하고 커널 영역 메모리에 이를 저장하게 된다. 또한, 저장된 데이타를 타겟 유저 영역 메모리로 복사해주는 역할을 한다. 유저 영역의 ETT^{plus} 타겟 데몬은 ETT^{plus} 콜렉터로 데이타 복사 요청을 할 수 있으며 유저 영역 메모리에 복사된 추적 데이타를 호스트 데몬으로 전송해준다. 호스트 데몬이 ETT^{plus} 타겟 데몬으로부터 추적 데이타를 전송받으면 이를 호스트 상의 파일 시스템으로 저장하게 되고 최종적으로 ETT^{plus} 호스트 시각화 툴이 저장된 데이타 파일을 이용하여 사용자에게 커널 분석 정보를 GUI 환경으로 제공한다.

표 1 기존의 기법과 제안한 기법의 비교

	기존의 기법(ETT)	제안한 기법(ETT^{plus})
타겟/호스트 간의 데이타 전송방법	NFS 구축	소켓 통신
타겟 파일 시스템 이용 유무	이용	이용안함
데이타 저장 방법	파일시스템 이용	메모리 이용
커널 버전	2.4.X	2.6.X
시각화 툴	윈도우 기반 환경	GTK+ 기반 윈도우/리눅스 지원

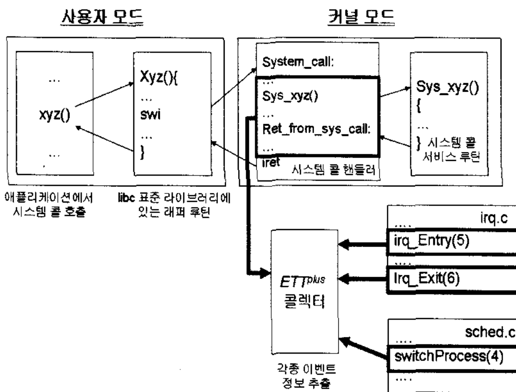


그림 5 ETT^{plus} 콜렉터 구조

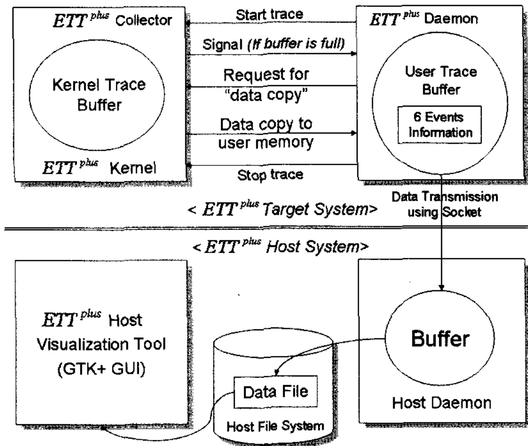


그림 6 커널 추적 데이타 흐름

ETT^{plus} 콜렉터는 실제 커널 추적 데이타를 수집하기 위한 부분으로 커널 패치 부분이라고 할 수 있다. 그림 5는 이러한 ETT^{plus} 콜렉터 구조에 대해 나타내고 있다 [8]. ETT^{plus} 콜렉터는 시스템 콜을 호출하는 애플리케이션과 해당하는 래퍼 루틴, 시스템 콜 핸들러, 시스템 콜 서비스 루틴과 같은 시스템 콜 정보를 추출하기 위한 부분과 인터럽트 발생 및 종료에 해당하는 루틴 및 프로세스 스케줄링 루틴에 대한 부분과 상호 작용을 한다. 커널 패치 파일 중 가장 기본적인 파일은 "linux/arch/arm/kernel/traps.c" 파일이며 실제로 ETT^{plus} 콜렉터의 역할을 담당한다. 이 외에도 몇 가지 추가된 커널 프리미티브 코드와 아키텍처 설정을 하기 위한 코드들이 있다. 2장에서 소개된 LTT에서는 리눅스 시스템 커널 버전마다 소스가 조금씩 다르므로 삽입되는 패치 코드 또한 제공하는 알맞은 버전으로 패치하여야 한다. 또한 LTT는 일반적인 리눅스 커널에 맞춘 패치 버전을 가지고 있으므로 임베디드 리눅스 커널에 맞게 패치하려면 방대한 양의 패치 파일들을 가지고 장시간의 커널 수정 작업을 해주어야 한다. 본 논문에서 제공하는 ETT^{plus} 패치는 임베디드 커널 2.6 버전에 맞춘 한계가 있지만 추가된 소스를 모듈화하여 쉽게 커널 소스에 붙여도 문제가 없도록 제작되었다.

(2) ETT^{plus} 추적 데이타 흐름

ETT^{plus} 콜렉터에 의해 생성된 커널 추적 데이타는 식별하기 위해 호스트 상의 GUI 환경으로 전달이 되어야 하는데, 다음 그림 6에서는 ETT^{plus} 시스템에서의 전반적인 추적 데이타 전송 흐름에 대한 그림을 나타내고 있다.

그림 6에서 보면, ETT^{plus} 타겟 데몬이 추적 시작을 요청하면 ETT^{plus} 커널이 ETT^{plus} 콜렉터를 동작시켜 추적을 시작한다. 이때 버퍼에 추적 데이타가 가득 차게 되면 ETT^{plus} 커널은 타겟 데몬으로 시그널을 보내게 된다. 시그널을 받은 ETT^{plus} 타겟 데몬은 커널 내부 메모리에 쓰여 있는 추적 데이타를 유저 영역으로 복사하기 위한 복사 요청을 보낸다.

요청을 받은 커널은 유저영역 메모리로 추적 데이타를 복사하게 되고 그 즉시 ETT^{plus} 타겟 데몬은 복사된 추적 데이타를 소켓 통신을 통하여 ETT^{plus} 호스트 데몬으로 전송하게 된다. 데이타를 받은 호스트 데몬은 호스트 머신상의 파일 시스템을 이용하여 데이타 파일을 생성하게 되고 이후에 받는 추적 데이타를 계속 파일에 저장하게 된다. 이러한 동작은 ETT^{plus} 타겟 데몬이 추적 종료를 요구할 때까지 반복적으로 일어나게 되며 최종적으로 호스트 상에서 생성된 데이타 파일을 이용하여 ETT^{plus} 시각화 툴이 GUI 방식으로 추적 데이타를 표현해 주게 된다. 다음의 표 2에는 본 논문에서 제안한

표 2 본 논문에서 사용한 중요 함수 및 플래그 정의

함수 및 플래그	세부설명	비고
trace_start()	추적 시작을 위한 시스템 콜	추가된 시스템콜
trace_stop()	추적 종료를 위한 시스템 콜	추가된 시스템콜
data_copy(addr,size,PID)	타겟 사용자 데몬이 타겟 커널로 추적 데이타 복사를 요청함	추가된 시스템콜
Trace_EnableFlag	값이 0이면 추적 불가, 1일 경우에만 추적 가능	추가된 플래그
copy_to_user(addr,size,PID)	커널영역에 쓰여진 버퍼를 요청한 사용자영역의 버퍼에 복사함	기존의 커널 함수
save_memory()	커널 버퍼가 가득 찼을 경우 다른 커널 버퍼로 전환하고 사용자 영역으로 시그널을 보내어 데이타를 복사하게 함	추가된 커널 함수

알고리즘 1. 타겟/호스트 간의 전체적인 추적 데이타 전송 과정

```
IF 추적 시작 이벤트 호출 THEN
1 trace_start() //타겟 사용자 영역
2 Trace_EnableFlag = 1 //타겟 커널 영역
3 이벤트 추적(Event 1, Event 2, ..., Event 6) //타겟 커널 영역
4 IF (커널추적버퍼 == FULL) THEN 8
5 ELSE
6 RETURN 3
7 END IF
8 타겟 커널 영역에서 사용자 영역으로 추적 데이타 복사 과정 //"알고리즘 2" 수행
9 데이타전송소켓(추적데이타) //타겟 사용자 영역
10 데이타수신소켓(추적데이타) //호스트 영역
11 파일쓰기(수신한 추적데이타) //호스트 영역
12 IF 호출(trace_stop()) THEN 16 //타겟 사용자 영역
13 ELSE
14 RETURN 3
15 END IF
16 Trace_EnableFlag = 0 //타겟 커널 영역
17 파일쓰기(마지막으로 수신한 추적데이타) //호스트 영역
18 파일닫기() //호스트 영역
```

추적 설계 방식 및 알고리즘 설명을 위하여 사용한 중요 함수 및 플래그의 정의를 나타내었다.

위의 함수 및 플래그의 정의를 바탕으로 타겟/호스트 간의 전체적인 추적 데이타 전송 과정을 위의 알고리즘 1과 같이 나타내었다.

위의 알고리즘 1에서, 호스트 상에서 사용자가 타겟에 대해 추적 시작 이벤트를 호출하게 되면 *trace_start()* 시스템 콜이 발생하여 *Trace_EnableFlag* 값을 1로 조정하게 되고 추적이 시작된다. 또한 커널 영역의 버퍼에 추적 데이타가 가득 차게 되면 사용자 영역의 버퍼로 데이타를 복사해 주는 과정이 일어나게 된다(알고리즘 2 수행).

이때, 사용자 영역의 데몬은 복사된 버퍼의 내용을 소켓통신을 통해 호스트 영역으로 데이타를 전송하게 되고 호스트에서는 이를 *파일쓰기()*를 이용하여 파일 시스템에 파일로 저장하게 된다. 사용자가 추적을 계속 원하면 반복 작업을 계속하게 되고 그렇지 않으면 *trace_stop()* 시스템 콜이 발생하여 *Trace_EnableFlag* 값을 0으로 조정하여 추적을 멈추게 된다. 추적종료 이벤트가 발생한 후에 쓰여지지 않고 남아있는 데이타들을 호스트에서는 모두 받아서 파일에 쓰고 *파일닫기()*를 하여 한번의 추적 시스템을 완성하게 된다.

위의 알고리즘 2는 알고리즘 1의 8번 과정을 자세하게 나타낸 부분이다. 이 알고리즘은 타겟 커널 영역 메모리에서 타겟 사용자 영역 메모리로 데이타를 복사하는 과정이 핵심이다. 추적이 시작되면 타겟 커널은 커널

알고리즘 2. 타겟 커널/사용자 영역 간의 추적 데이타 복사 과정

```
IF (커널추적버퍼 == FULL) THEN
1 save_memory() { //타겟 커널 영역에서 사용자 영역으로 시그널 전송
      .......
      sys_kill(user_pid, SIGNAL)
  }
2 타겟 사용자 영역에서 시그널을 받음
3 시그널 루틴 함수() { //타겟 사용자 영역
      .......
      data_copy(user_addr, size, user_pid) //데이타 복사 요청
  }
4 copy_to_user(user_addr, size, user_pid) //타겟 커널 영역, 데이타 복사
```

메모리 내의 버퍼에 추적 데이타를 모두 기록하게 된다. 버퍼 크기는 한정되어 있으므로 버퍼가 가득차게 되면 *save_memory()* 커널 함수를 사용해 다른 버퍼로 교체를 하게 되고 시그널을 사용자 영역으로 보내게 된다. 사용자 데몬은 슬립 상태에 있다가 커널로부터 시그널을 받게 되면 시그널 루틴 함수 내의 *data_copy()* 시스템 콜을 호출하여 커널 영역의 버퍼를 사용자 영역의 버퍼로 복사하도록 요청을 한다. 이렇게 요청을 받으면 커널 영역에서는 *copy_to_user()*라는 커널 함수를 통해 커널 버퍼의 내용을 사용자 영역 버퍼로 복사하게 된다.

(3) ETT^{plus} 추적 데이타 이벤트

앞에서 제시한 바대로 추적시스템을 구동하게 되면 추적 데이타가 형성이 되고 타겟/호스트간의 통신을 통해 데이타 전송과정을 거쳐 최종적으로 호스트의 파일시스템에 파일로 저장이 된다. 추적 데이타는 ETT^{plus} 콜렉터에 의해 생기며 크게 총 6가지 추적 이벤트 타입을 기반으로 추적 데이타를 추출해 낸다. 각각의 추적 이벤트 타입에 대한 추적 데이타 정의는 다음 표와 같이 설명할 수 있다.

이벤트 1은 시스템 콜 호출 시 정보에 관한 것이고 이벤트 2는 이벤트 1에 해당하는 시스템 콜의 종료에 대한 정보이다. 이벤트 3은 추적하는 동안 발생한 모든 프로세스에 관한 정보이며 이벤트 4는 발생한 프로세스가 어떤 순서로 스케줄링 되어 처리되는지에 대한 정보이다. 이벤트 5는 추적하는 동안 발생한 인터럽트에 대한 정보이며 이벤트 6은 이벤트 5에 해당하는 인터럽트의 종료에 대한 정보이다.

2장에서 소개하였던 기존 커널 추적 도구들 중 대표적이라고 할 수 있는 LTT는 각 프로세스 수행 정보와 응용 프로그램이 호출한 시스템 콜에 대한 정보 및 인터럽트에 대한 정보를 중점적으로 제공하고 있다. 이와 같이 ETT^{plus} 시스템에서도 위에서 언급한 6가지 이벤트 데이타를 최종적으로 정보를 제공하기 위해 가장 기본적으로 추출된 데이타로 간주하며 이들 데이타를 가공 및 활용하여 ETT^{plus} 시각화 도구에서 각종 정보를 제공하고 있다. ETT^{plus} 시각화 도구를 통해 제공되는 최종 정보로는 각 프로세스 마다 시간 축에 따라 실행되는 "System Call Events", 이들 시스템 콜 각각에 대한 상세 정보를 나타내는 "System Call Information (PID, Syscall ID, Syscall name, Start time, End time, Execution time)", 추적 시작부터 종료까지 수행된 프로세스의 변화를 나타내주는 "Process Switching(PID, Process execution time)", 각 프로세스 마다 몇 개의 시스템 콜이 발생하였는지를 나타내는 "System call counts", 이들 시스템 콜 개수에 따른 총 수행 시간인 "Total execution time of system call", 전체 추적 시

표 3 추적 데이타 이벤트 타입

이벤트 번호	이벤트 타입	설명
1	System call entry	시스템 콜의 시작 부분 표시
2	System call exit	해당 시스템 콜의 종료 시간을 표시
3	Process	생성된 프로세스에 대한 상태 정보를 표시
4	Switching process	프로세스 스케줄링 표시
5	Irq entry	인터럽트 진입 부분에 대해 표시
6	Irq exit	인터럽트 종료에 대한 정보를 표시

간에 비례한 시스템 콜 수행 시간의 비율을 측정하는 "System call usage", 추적 동안에 발생한 인터럽트에 관한 정보를 제공하는 "Interrupt Event(Interrupt number, Execution time)", 발생한 인터럽트 개수를 나타내는 "Interrupt counts", 이들 인터럽트 개수에 따른 총 수행 시간인 "Total execution time of interrupts", 전체 추적 시간에 비례한 인터럽트 수행 시간의 비율을 측정하는 "Interrupt usage" 등이 있다. 이처럼 임베디드 커널의 각 프로세스가 시스템에 미치는 영향을 분석하는 데는 최소한 위와 같은 6가지의 이벤트를 바탕으로 이를 가공한 여러 정보를 통해 분석을 할 수 있다.

(4) 호스트/타겟 데이타 전송 방법

지금까지 본 논문에서 제시하는 추적 시스템의 전체적인 구조에 대해 전반적으로 설명하였다. 본 추적 시스템은 타겟/호스트 간의 추적시스템이며 타겟에서 발생한 추적데이타를 소켓통신을 통해 호스트 시스템으로 가져온 후 최종적으로 호스트 파일시스템을 이용한 파일 저장 방식을 제시하고 있다. 본 논문에서 제시한 전송 방법 이외에도 추적 데이타 전송 방법이 몇 가지가 있다. 임베디드 시스템에서 호스트와 타겟 시스템 사이에서의 몇 가지 데이타 전송 방법에 대한 분류를 하자면 크게 다음과 같이 3가지로 나눌 수 있다.

① NFS 파일 시스템 이용 : 과거에 구현된 *ETT* 시스템[7]으로 대용량 저장이 가능하다. 타겟 시스템이 반드시 호스트 시스템과 NFS로 연결되어 있어야 하며 비교적 쉬운 인터페이스에 속하지만 파일 시스템이 꼭 필요하다는 단점이 있다.

② 커널 내에서 직접 호스트로 전송 : 커널 내부에 데이타 전송 코드(소켓 통신)가 추가되어야 한다. 이중의 데이타 복사가 필요 없으며 사용자 수준에서 추적 데이타 전송시기 및 주기를 제어하기 위한 프리미티브가 필요하다. 데몬을 통한 데이타 전송의 중간단계가 필요 없지만 커널 패치가 그만큼 커지게 되고 사용자에 따라 시스템의 수정 및 최적화를 위해서는 반드시 커널작업을 해줘야 한다는 불편함이 있다.

③ ETT^{plus} 데몬 이용 : 본 논문에서 새롭게 제시된 ETT^{plus} 시스템으로 커널 영역에서 사용자 영역으로의 데이타 카피를 위한 커널 프리미티브(시스템 콜) 추가가 요구된다. ETT^{plus} 타겟 데몬에서 소켓 통신을 통해 호스트 데몬으로 전송하기 위해 최소한 1번의 추가 데이타 복사가 필요하며 사용자 수준에서 추적데이타 전송시기 및 주기를 제어할 수 있는 특징이 있다. 또한, 디버깅이 용이하며 파일 시스템이 필요 없다는 장점이 있으나 파일 시스템을 이용하는 것보다 다소 복잡한 시스템이다.

이와 같이 크게 3가지 방식의 데이타 전송 방법을 제시할 수 있으며, 본 논문에서의 ETT^{plus} 시스템에서는 세 번째 방식을 적용하고 있다.

3.2 ETT^{plus} 구현

본 장에서는 3장에서 설계된 ETT^{plus}에서 추출해 낸 실제 추적 정보를 이용하여 GUI 화면으로 표현한 부분에 대해 설명하고자 한다.

그림 7은 ETT^{plus} 그래픽 모드의 한 화면으로 프로세스 ID가 223인 프로세스 "ETT_start"에 대한 시스템 콜 정보를 그래프로 나타낸 화면이다. 위 그림에서 나타내 주고 있는 시스템 콜 정보는 시스템 콜 ID가 2번인 "fork_wrapper()"이며, 이 시스템 콜이 수행된 시간은 총 657usec이다.

ETT^{plus} 시각화 툴은 추적 데이타 정보를 분석하는 기능 외에도 여러 가지 기능을 담고 있다. ETT^{plus} 시각화 툴이 제공하는 주요 기능으로는 다음과 같다.

- 추적 데이타 정보를 분석하는 기능(그래픽 모드, 텍스트 모드)
- 타겟에서 실행되었던 모든 프로세스가 차지하는 CPU 및 메모리 정보를 나타내주는 프로세스 상태 기능
- 로컬과 원격 시스템 사이의 자유로운 데이타 전송을 위한 FTP 기능
- 원격에 있는 시스템의 소스를 불러와서 로컬에서 컴파일하고 원격에서 바로 실행을 해주는 리모트 컴파일 기능
- 임베디드 시스템을 원격으로 제어하기 위한 리모트 쉘 기능
- 원격에 있는 파일 시스템을 제어할 수 있는 리모트 파일 시스템 기능

4. 임베디드 시스템 성능 측정 기법의 설계

본 장에서는 앞 장에서 설계 및 구현된 ETT^{plus} 시스템을 이용한 타겟 커널 시스템 콜 실행 시간 측정 설계 및 타겟/호스트 간의 네트워크 트래픽의 상관 관계 측정을 위한 설계 부분을 제시하고자 한다.

4.1 시스템 콜 실행 시간 측정 설계

본 논문에서 제시하고자 하는 시스템 콜 실행 시간 측정 설계는 크게 다음과 같이 3단계로 나누어진다.

단계 1. SYSCALL_ETT_OFF : ETT^{plus}를 동작하지 않은 상태에서 사용자 영역에서 시스템 콜을 발생시켜 이에 대한 실행 시간을 구한다. 이 방법은 사용자 모드에서 애플리케이션 프로그램을 이용하여 측정할 수 있다. 그림 8에서 ①+②+③의 시간을 구한 값이다.

단계 2. SYSCALL_ETT_ON : ETT^{plus}를 동작한 상태에서 나머지는 단계 1과 동일하게 적용하여 사용자 영역에서 시스템 콜을 발생시켜 이에 대한 실행 시간을 구한다. 그림 8에서 ①+②+③의 시간을 구한 값이다.

단계 3. ONLY_ETT : ETT^{plus}를 이용하여 수집한

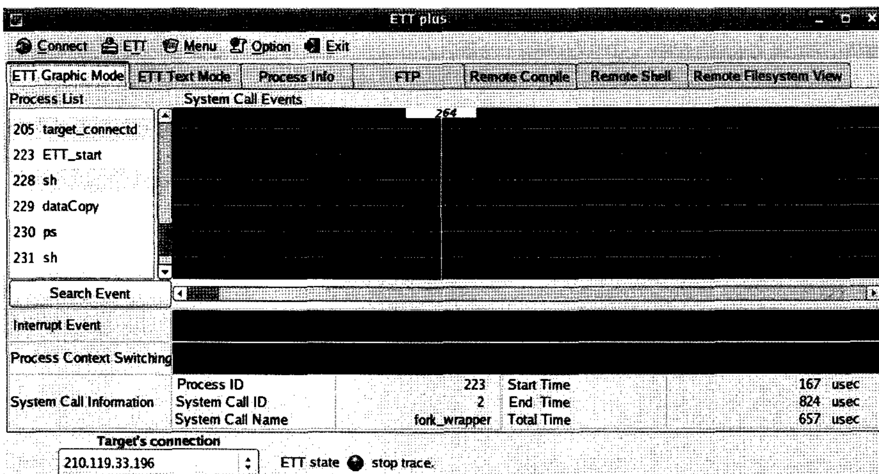


그림 7 ETT^{plus} 그래픽 모드

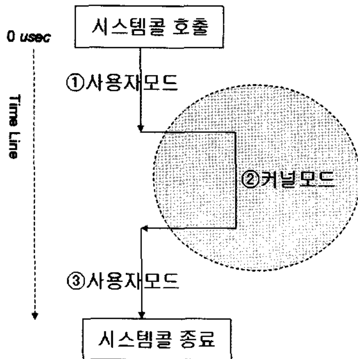


그림 8 시스템 콜 실행 영역 모드

커널 추적 데이타를 분석하여 커널 모드 내의 시스템 콜 실행 시간을 구한다. 그림 8에서 ②의 시간 값만을 구한 것이다.

실제로 보통의 임베디드 시스템은 패치 및 컴파일을 하지 않는 한 ETT^{plus}와 같은 커널 추적 도구를 사용할 수 없고 단계 3과 같이 커널 내에서만 측정한 시스템콜 실행시간을 얻는 것은 어렵다. 위의 설계대로 측정을 하게 된다면 다음과 같은 가정을 얻을 수 있다.

```
A : SYSCALL_ETT_OFF(단계 1 상태에서 측정한
    시스템 콜 실행시간, 단위 usec)
B : SYSCALL_ETT_ON(단계 2 상태에서 측정한
    시스템 콜 실행시간, 단위 usec)
C : ONLY_ETT(단계 3 상태에서 측정한 시스템 콜
    실행시간, 단위 usec)
```

가정 1) B > A > C

가정 2) 보정치 $r = \frac{(A-C)\times 100}{C}(\%)$

가정 3) $C = A-(A\times\frac{r}{100}), (r=2)$

가정 4) ETT^{plus} 오버헤드 $t = \frac{(B-A)\times 100}{A}(\%)$

시스템 콜 수행 시간은 가정 1과 같이 ETT^{plus}가 동작 중일 때 일반 사용자 애플리케이션으로 얻은 시스템 콜 수행 시간이 가장 길며, 그 다음으로 ETT^{plus}가 동작하지 않을 때 일반 사용자 애플리케이션으로 얻은 시스템 콜 수행 시간이다. ETT^{plus}를 이용하여 순수 커널 내부에서만 수행한 시스템 콜 시간이 가장 낮다. A와 C를 이용하면 보정치를 구할 수 있게 되는데, 예를 들어 A 값이 100*usec*이고 C값이 98*usec*이면 보정치 값은 2%가 된다. 즉, (가정 2)를 근거로 구한 보정치를 이용하여 동일한 하드웨어 구성 및 동일한 임베디드 리눅스 커널 버전인 환경에서는 사용자 모드에서 측정한 시스템 콜 실행시간에서 보정치 값을 빼주면 ETT^{plus}를 이용하지 않고도 실제 커널에서 수행한 시스템 콜 실행시간을 구할 수 있게 된다(가정 3). 본 논문에서는 보정치 값을 '2'로 가정하고자 한다. 또한, 일반 사용자 모드에서 측정한 시스템 콜을 ETT^{plus}가 각각 동작할 때와 안할 때를 비교하여 이에 대한 커널 추적 시스템의 오버헤드도 구할 수 있게 된다(가정 4).

4.2 네트워크 데이타 전송 시간 측정 설계

본 논문에서 제시하고자 하는 네트워크 데이타 전송 시간 측정 설계는 크게 다음과 같이 나눌 수 있다.

단계 1. NETWORK_ETT_OFF : ETT^{plus}를 동작하지 않은 상태에서 타겟/호스트 간의 데이타 전송에 대한 시간을 구한다.

단계 2. NETWORK_ETT_ON : ETT^{plus}를 동작한 상태에서 타겟/호스트 간의 데이타 전송에 대한 시간을 구한다.

위의 설계를 기반으로 ETT^{plus} 동작 상태에서 타겟/호스트 간의 추적 데이타를 전송하는 도중 일반 데이타를 전송할 때 어느 정도의 전송 오버헤드를 부여하는가를 측정할 수 있게 된다. 또한 이때 타겟에서 ETT^{plus} 설계에 따라 추적 데이타를 호스트로 전송하게 되면 이에 따르는 전송 오버헤드가 발생하게 되는데, 한번에 전송하는 추적 데이타의 크기를 각각 다르게 측정하여 이에 따른 전송 오버헤드를 비교 할 수 있게 된다.

4.3 기타

위에서 제시한 시스템 콜 실행 시간 측정 설계와 네트워크 데이타 전송 시간 측정 설계 이외에도 인터럽트에 관한 측정 설계 및 프로세스 스케줄링에 관한 측정 설계를 제시하고자 한다.

인터럽트에 관한 측정은 추적 시 발생하는 모든 인터럽트에 대한 종류별 빈도수 및 발생한 시간을 측정하여 발생한 인터럽트가 전체 시스템에 어느 정도의 영향을 미치는가를 측정하게 된다. 또한 ETT^{plus}는 추적 시작과 동시에 수행중인 프로세스의 스케줄링 정보를 알 수 있는데 각 프로세스의 스케줄링 빈도수 및 수행 시간 등을 측정하게 된다.

5. 성능 평가

본 장에서는 앞 장에서 설계된 ETT^{plus} 시스템을 이용한 타겟 커널 시스템 콜 실행 시간 측정 및 타겟/호스트 간의 네트워크 전송 트래픽의 상관관계에 관한 설계를 기반으로 실제의 성능 평가 부분을 제시하고자 한다.

본 논문에서의 성능 평가를 위한 실험 환경은 표 4와 같다. 통신 환경은 타겟과 호스트 간의 일대일 통신으로 구성하였고 네트워크 트래픽 분석을 위한 데이타 통신 방법은 소켓 통신 중 UDP통신을 이용하였다. 타겟 보

표 4 성능 평가를 위한 실험 환경

구성	설명
타겟 커널	임베디드 리눅스 2.6.11
타겟 보드	X-Hyper270A (하이버스 제작)
CPU	Intel Bulverde PXA 270 520MHz
메모리	64M
호스트 환경	페도라 Core 5, 크로스 컴파일러

드 환경은 임베디드 커널 2.6.11 버전이며 Intel Bulverde PXA270 520MHz의 CPU, 64M 메모리, JFFS2 파일 시스템 등으로 구성되어 있다. 호스트 시스템 환경은 페도라 Core 5이다. 호스트와 타겟은 이더넷 및 시리얼 통신으로 연결되어 있으며 타겟 개발을 위한 크로스 컴파일러가 호스트 상에 설치가 되어 있다.

5.1 시스템 콜 실행 시간 측정 성능 평가

그림 9에서 보여주고 있는 그림들은 각 시스템 콜 별로 앞의 4장의 시스템 콜 실행 시간 측정 설계 부분에서 제시한 3단계에 대한 실행 시간을 측정한 것이다. 시스템 콜 종류 중 “Syscall_1”, “Syscall_2”는 커널에 새로 등록한 시스템 콜이며 나머지는 커널에 내장되어 있는 시스템 콜을 테스트한 것이다.

그림 9에서 보듯이 일반 환경에서 시스템 콜 실행 시간을 측정한 것(A)보다 ETT^{plus}가 동작하는 환경에서의 시스템 콜 실행 시간(B)이 조금씩 올라감을 알 수 있다. 그러나 증가하는 시간간격이 크지 않으며 ETT^{plus}가 동작할 때의 시스템 콜 실행 시간 측정을 기반으로 가장 심한 추적 부하를 가한 경우에도 시스템 오버헤드는 전체 시스템 성능 중 2% 미만인 것으로 나타났다. 이는 전체 시스템에 최대 2.5%의 악영향을 미치는 LTT에 비해 상당한 성능 효과를 얻을 수 있게 된다. 결과적으로 ETT^{plus} 시스템이 동작하여도 전체적인 시스템 성능에는 큰 영향을 주지 않는다는 점을 알 수 있다.

표 5는 이와 같은 성능 평가를 위해 측정한 시스템 콜 중 몇 개를 예로 들어 수치로 표현한 것이다. 이 표

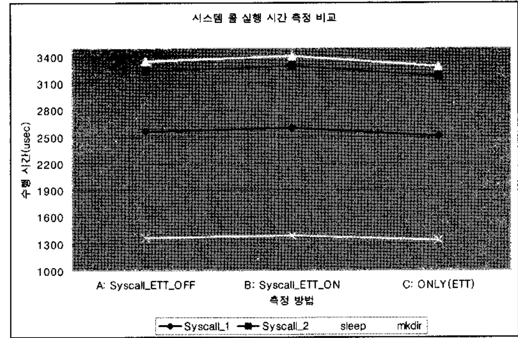


그림 9 각 시스템 콜 별로 측정한 실행 시간

를 통해 설계 부분에서 제시한 보정치 값 ‘2’를 증명하고 오버헤드는 2% 미만임을 알 수 있다. 물론 이들 측정치는 하드웨어 구성이 다르고 임베디드 리눅스 커널에 따라 시스템의 상태가 다르기 때문에 약간의 차이를 나타낼 수도 있다. 보정치 값에 대한 검증은 동일한 하드웨어 구성 및 동일한 임베디드 리눅스 커널 버전인 환경에서만 같은 값이 됨을 전제로 한다.

5.2 네트워크 데이타 전송 시간 측정 성능 평가

그림 10은 UDP 통신을 이용하여 ETT^{plus}를 사용하지 않는 경우와 사용하는 경우에 대한 네트워크 전송 시간 분석에 대한 결과이다. 사용된 데이타양은 100byte부터 시작하여 100byte씩 계속 누적되고 최고 2000byte를 끝으로 전송을 마치게 된다. 전송 시간은 데이타를 보내는 순간의 시간부터 데이타 전송 확인 메시지를 다시 받을 때까지의 시간을 측정하였으며 단위시간은 *usec*이다.

사실 ETT^{plus}를 사용하지 않는 경우와 사용하는 경우에 대한 네트워크 데이타 전송 시간은 거의 차이가 없고 다만 그림 10의 1100바이트 전송 시 나타나는 타겟/호스트간의 추적 데이타가 전송되는 순간에만 네트워크 데이타 전송 시간은 차이가 나게 된다. 타겟/호스트 간

표 5 시스템 콜 실행 시간 실제 측정 데이타 (시간 단위 : *usec*)

System Call	ETT_OFF(A)	ETT_ON(B)	ONLY_ETT(C)	보정치(%)	오버헤드(%)
Syscall_1	2556.2	2602.5	2508.6	1.9	1.81
Syscall_2	3238.8	3299.6	3176.9	1.95	1.82
sleep	3348.1	3402.4	3286.2	1.88	1.62
mkdir	1368.9	1395.1	1337	2.2	1.91
chmod	940.8	958.4	922.8	1.95	1.87
chdir	95.3	97	93.3	2.14	1.78
link	961.3	978.7	941.3	2.12	1.81
access	105	107	103	1.94	1.9
lstat	129.3	131.7	126.7	2.05	1.86
read	64.7	66	63.3	2.2	1.99
time	72.8	74	71.4	1.96	1.64

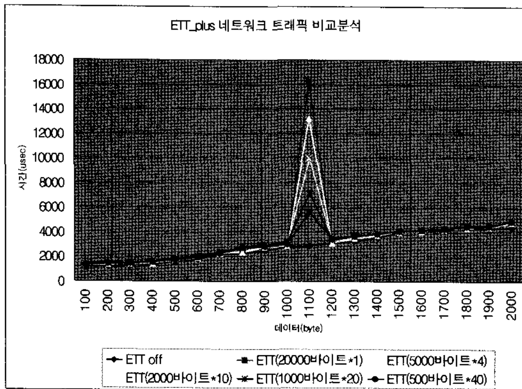


그림 10 타겟/호스트 네트워크 데이타 전송 시간 비교 분석

의 추적 데이타 전송은 기본적으로 20000바이트 단위로 이루어지는데 이때 네트워크 데이타 전송 시간은 가장 길게 측정이 된다. 그리하여 본 논문에서는 추적 데이타를 분할하여 전송함으로써 타겟/호스트 간에 발생하는 추적 데이타 전송 시 피크 타임의 과부하를 분산시키고 전체 네트워크 성능의 영향을 최소화하였다. 이러한 방법으로 ETT^{plus}를 사용하는 시스템에서도 네트워크 데이타 전송 트래픽에 관한 오버헤드를 줄일 수 있게 됨을 알 수 있다.

또한, 그림 10에서 보듯이 일반 환경(ETT off)보다는 ETT^{plus}가 동작하는 환경에서 네트워크 전송시간이 조금씩 올라감을 알 수 있다. 그러나 증가하는 시간간격이 크지 않으며 ETT^{plus}가 동작할 때 타겟/호스트 간의 추적 데이타 전송에 대한 오버헤드는 전체 네트워크 전송 시간 중 2.5% 미만인 것으로 나타났다. 결과적으로 ETT^{plus} 시스템이 동작하여도 전체적인 네트워크 데이타 전송에는 큰 영향을 주지 않는다는 점을 알 수 있다.

표 6은 ETT^{plus}와 LTT(LTTng)에 대한 주요 특징을 간략하게 비교하여 요약해 놓은 것이다. 2장 관련 연구에서도 언급했듯이 LTTng는 LTT의 업그레이드 버전으로 LTT에서 제공하는 정보 및 특징을 그대로 지원하며 확장 기능을 통해 도구 자체가 좀 더 무거워진 경향이 있다. 그러므로 임베디드 시스템에 초점을 맞춰 비교를 하기 위해서는 LTT가 좀더 적합하다고 판단이 된다. LTT(LTTng)는 주로 호스트 리눅스 시스템에 중점을 둔 도구인 반면 ETT^{plus}는 임베디드 타겟 시스템에 중점을 둔다. 그러므로, 임베디드 시스템 용 패치 또한 ETT^{plus}에서 좀더 용이하게 구성이 되어 있다. 또한 커널에서 추적된 데이타를 곧바로 로컬 파일 시스템에 저장하는 LTT와는 달리 ETT^{plus}는 추적 데이타를 로컬이 되는 타겟 파일 시스템에 저장하지 않고 이를 호스트 시스템으로 전송한 후 호스트 시스템 상의 파일에 최종적으로 저장되는 호스트/타겟 사이의 명확한 통신방법을 제시하고 있다.

또한 시스템 성능 오버헤드 면에서 LTT는 최대 2.5%미만으로 시스템에 악영향을 부여하지 않는다고 제시하고 있으며[3], ETT^{plus} 같은 경우는 위에서 테스트한 결과 이 오버헤드를 더 줄일 수 있게 되었다. 네트워크 패킷 전송에 따른 트래픽 오버헤드는 LTT 같은 경우 호스트 단일 시스템에서 동작하므로 트래픽 오버헤드 측정을 비교하기는 어려우며, 타겟/호스트 사이의 데이타 전송이 필요한 ETT^{plus}는 최대 2.5% 미만의 네트워크 트래픽 오버헤드를 나타낸다고 앞의 테스트에서 증명하고 있다.

6. 결 론

본 논문에서는 ETT^{plus}(Embedded kernel Trace Toolkit)를 이용한 커널 추적 데이타 추출 방법 및 추적 데이타 전송 방법에 대한 연구를 제시하고 있다. 또한, 이를 이용하여 임베디드 시스템의 성능 측정에 대한 설계 및 평가를 제시한다.

ETT^{plus}는 기존에 널리 사용되고 있는 커널 추적 도구인 LTT나 LTTng에 비하여 비교적 간단한 커널 패치 작업 및 추적에 대한 핵심 정보만을 분석할 수 있는 환경을 제공하고 있으며 커널 및 사용자 데몬 환경을 이용하여 추적 데이타를 전송하는 방식을 설계 및 구현하였다.

또한, 생성된 추적 데이타 파일이 타겟 파일 시스템에 저장이 되지 않고 이를 연결된 호스트로 전송하여 호스트상의 파일 시스템에 저장되는 방식을 적용하여 자원의 제약이 있는 임베디드 타겟의 파일 시스템 의존성 문제를 해결하였다.

표 6 LTT(LTTng)와 ETT^{plus}의 성능 평가 비교

	LTT(LTTng)	ETTplus
특징	호스트 리눅스 시스템 기반	타겟 임베디드 시스템 기반
데이타 저장 타입	파일시스템을 이용한 파일 저장	타겟 메모리 이용, 호스트 파일시스템 이용
시간 측정 방법	usec 단위(do_gettimeofday() 이용)	동일함
시스템 성능 손실(오버헤드)	최대 2.5%	최대 2% 미만
네트워크 오버헤드	-	최대 2.5% 미만

성능 측정의 결과 시스템 콜 실행 시간 측정을 통해 시스템 성능 오버헤드는 최대 2%임을 알 수 있으며 네트워크 데이타 전송에 대한 실험을 통해 ETT^{plus} 시스템이 작동하여도 전체적인 네트워크 전송 오버헤드는 최대 2.5% 미만의 결론을 얻을 수 있었다.

7. 향후 과제

본 논문에서 소개된 데이타 전송 방법은 추적 데이타 버퍼가 가득 찼을 경우에만 ETT^{plus} 타겟 데몬이 데이타 카피요청을 하여 버퍼의 크기만큼 복사해 오는 방식이었는데, ETT^{plus} 타겟 데몬에서 데이타 복사를 요청하는 주기적인 시간 간격을 설정하여 원하는 시간 마다 커널 버퍼의 내용을 사용자 영역 메모리로 복사하는 방식 또한 현재 설계 고려중에 있다.

이러한 방식의 장점으로는 커널 버퍼가 가득찰 때 까지 사용자 영역의 ETT^{plus} 타겟 데몬이 대기하지 않고 설정된 시간 간격에 따라서 버퍼의 내용을 호스트 데몬으로 전달함으로써 좀 더 실시간성에 가까운 시스템을 유지할 수 있다.

또한, 현재 ETT^{plus}에서는 기본적으로 6가지의 이벤트를 기반으로 추적 정보를 제공하고 있는데 앞으로 버전 업그레이드되는 과정에서 사용자의 필요에 따라 추가로 필요한 이벤트를 커널 패치를 통해 추출할 수 있도록 고려하고 있다.

참 고 문 헌

[1] J.H. Na, S.J. Kang, Y.I. Yoon, Y.Y. Park, S.B. Eun, H.N. Kim, *"Embedded System Programming,"* SciTech Media, 2004.

[2] Ji-Hye Bae, Yoon-Young Park, Jeong-Bae Lee, Sung-Hee Choi, Chae-Deok Lim, "A study on the Design of the Monitoring Architecture for Embedded Kernels based on LTT," *Proc. of 4th Asia Pacific International Symposium on Information Technology*, Gold Coast, Australia, pp. 68-71, Jan., 2005.

[3] Karim Yaghmour, *"Building Embedded Linux Systems,"* O'Reilly.

[4] Opersys Homepage, http://www.opersys.com/LTT

[5] LTTng & LTTV Homepage, http://ltt.polymtl.ca

[6] Mathieu Desnoyers, Michel R.Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," *Linux Symposium*, Ottawa, Canada, Jul., 2006.

[7] Ji-Hye Bae, Hee-Kuk Kang, John Y. Kim, Yoon-Young Park, "Monitoring Systems for Embedded Equipment in Ubiquitous Environments," *International Journal of Information Processing Systems (IJIPS)*, KIPS, Vol.2, No.1, pp. 58-65, Mar., 2006.

[8] Daniel P.Bovet, Marco Cesati, *"Understanding the Linux Kernel,"* 2nd Edition, O'Reilly.

[9] Nam-Sik Yoon, Ji-Hye Bae, Yoon-Young Park, Jin-Baek Kwon, "Kernel Tracing Toolkit for Embedded Systems," WSEAS Transactions on Computer Research, Issue 2, Vol.1, pp. 203-206, Dec., 2006.

[10] Karim Yaghmour and Michel R.Dagenais, "Measuring and Characterizing System Behavior using Kernel-Level Event Logging," Proc. of 2000 USENIX Annual Technical Conference, San Diego, California, USA, June 18-23, 2000.

[11] Ji-Hye Bae, Kyung-Oh Lee, Yoon-Young Park, "MONETA:An Embedded Monitoring System for Ubiquitous Network Environments," *IEEE Transactions on Consumer Electronics*, Vol.52, No.2, pp. 414-420, May, 2006.

[12] Yoon-Young Park, A Study on the Monitoring Model of Distributed Objects, *ETRI*, Korea, 2000.

[13] Q+ Esto Manual, Embedded Software Technology Center, *ETRI*, Korea, 2003.

[14] Mathieu Desnoyers, Michel R.Dagenais, "Low Disturbance Embedded System Tracing with Linux Trace Toolkit Next Generation," Embedded Linux Conference, 2006.

[15] Karim Yaghmour and Michel R.Dagenais, The Linux Trace Toolkit, *Linux Journal*, May, 2000.

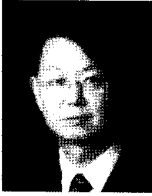
[16] Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, Michel Dagenais, "relayfs:An efficient unified approach for transmitting data from kernel to user space," In OLS(Ottawa Linux Symposium), pp. 519-531, July, 2003.

배 지 혜

1998년 3월~2003년 2월 선문대학교 컴퓨터정보학부 이학사. 2003년 3월~2005년 2월 선문대학교 전자계산학과 이학석사(임베디드시스템 전공). 2005년 3월~현재 선문대학교 컴퓨터정보학과 박사과정. 관심분야는 임베디드 시스템, 유비쿼터스 컴퓨팅, 센서 네트워크 등

윤 남 식

1999년 3월~2006년 2월 선문대학교 컴퓨터정보학부 이학사. 2006년 3월~현재 선문대학교 전자계산학과 석사과정. 관심분야는 임베디드 시스템, 리눅스 커널, 유비쿼터스 센서 네트워크 등



박 윤 용

1983년 3월~1985년 8월 서울대학교 계산통계학과(계산학전공) 이학석사. 1990년 3월~1994년 8월 서울대학교 계산통계학과(전산과학전공) 이학박사. 1985년 1월~1993년 2월 한국전자통신연구원 연구원. 1995년 1월~1996년 3월 한국전자통신연구원 초빙연구원. 1993년 3월~현재 선문대학교 컴퓨터정보학부 교수. 관심분야는 임베디드 시스템, 유비쿼터스 컴퓨팅, 센서 네트워크 등