원격 디버깅을 이용한 명령어 단위 프로파일링 기법

김승균 김정원 이재진
서울대학교 컴퓨터공학부
seungkyun@aces.snu.ac.kr, jungwon@aces.snu.ac.kr, jlee@aces.snu.ac.kr

An Instruction Level Profiling Technique Using Remote Debugging

Seungkyun Kim Jungwon Kim Jaejin Lee
School of Computer Science and Engineering, Seoul National University

요 약

본 논문에서는 자원적 제약이 있는 임베디드 컴퓨팅 환경에서의 명령어 단위 프로파일 방법을 제시한다. 이전 많은 연구들은 일반적인 컴퓨팅 환경에서의 프로파일 방법에 대하여 연구되었으며, 특정한 목적 시스템에 한정된 경우가 많았다. 하지만 본 방법은 리눅스 상의 응용 프로그램에 대하여 오픈 소스인 GDB를 이용하여, 다양한 목적 시스템에 쉽게 적용 가능한 방법을 기술한다. 다른 한편으로 성능의 향상을 위하여, 기록 버퍼를 이용하여 호스트와 게스트 시스템 사이의 통신 부담을 줄여 처음 제시한 방법의 수배의 성능 향상을 얻을 수 있었다. 이외에 앞으로의 추가적인 최적화 기법들의 적용을 통한 성능 향상을 기대하고 있다.

1. 서 론

명령어 단위의 프로파일 기술은 컴퓨터 아키텍처의 최적화에 유용하게 사용될 수 있는 정보이다. 프로파일 기술은 명령분기 통계, 시스템 자원의 이용률, 자주 수행 되는 프로그램 지역, 특정 시점에서의 작업 집합들과 같은 정보들을 제공하여 준다. 이러한 정보들은 하드웨어 시스템의 최적화에 사용될 수 있으며, 특정 응용 프로그램에 대하여 성능을 향상시키기 위한 소프트웨어적인 최적화에 이용할 수 있다. 특히 얻어진 정보는 일반적으로 메모리 시스템의 최적화에 유용하게 사용된다.

일반적인 명령어 단위 프로파일은 시뮬레이터를 통하여 얻어지는 것이 보통이다. 하지만 이러한 방법은 다음과 같은 제약을 가진다. 우선 믿을 수 있고 완전한 시뮬레이터가 존재하여야 한다. 임베디드 시스템의 개발환경은 빠르고 지속적으로 변하기 때문에 그러한 시뮬레이터를 완성하기 어려우며, 특히 리눅스 상의 응용 프로그램에 대하여 프로파일 정보를 얻기 위해서는 프로세서 코어의 시뮬레이션만이 아니라 주변 장치에 대한 구현 또한 필요함으로 인하여 리눅스를 실행 가능한 시뮬레이터를 구현하기는 어려운 점이 있다.

본 논문에서는 시뮬레이터를 통한 프로파일 방법이 아닌, 실제의 임베디드 시스템 상에서 gdbserver를 이용하여 보드에서 직접 실행한 결과를 수집하는 방법을 사용하고 있다. 이를 위하여 디버거의 기본적인 기능인 브레이크 포인트를 이용한 single-stepping 방법으로 리눅스 상에서 수행 중인 응용 프로그램에 대한 명령어 단위 프로그램 수행 기록을 얻어 내는 방법에 대하여 설명 한다.

다음 2장에서는 이전에 사용된 프로파일 기법들에 대하여 장단점을 살펴보고, 3장에서는 GDB를 이용한 프로파일의 기본 원리에 대하여 소개한다. 4장에서는 앞의 기본적인 방법의 성능 향상을 위하여 사용한 방법에 대하여 설명하고 5장에서는 결과에 대하여 간략히 논의한다. 마지막으로 6장에서는 앞으로의 일들과 결론으로 마무리 짓는다.

2. 관련 연구

우선 명령어 단위 프로파일의 의미에 대하여 살펴볼 필요가 있다. 완벽한 명령어 단위 프로파일이라고 한다면, 운영체제에서 수행되는 코드와 동시에 수행되고 있는 다른 프로세서들의 명령어들과 같은 CPU에서 수행되는 모든 명령어의 기록이라고 볼 수 있다. 이러한 완벽한 프로파일 기록을 만들어내기는 매우 어렵다. 이 논문에서는 특정 응용프로그램의 동작에 대하여 한정 지어 프로파일 기록을 저장하는 것에 범위를 한정 짓는다. 아래에서는 이전부터 사용하였던 single-stepping[1], instruction-inlining[2,3,4], hardware monitoring[5]과 프로세서 시뮬레이션[1,6]의 방법의 장단점에 대하여 살펴본다.

2.1. Hardware monitoring

일반적으로 Logic analyzer와 같은 하드웨어 장치 등을 통하면 프로세서에서 수행되는 명령을 직접 기록할 수 있다. 만약 계측 하드웨어 장치의 속도가 충분히 빠르고, 기록 버퍼의 크기가 크다면 완벽한 프로파일 결과를 저장할 수 있다는 장점이 있다.

하지만 특수한 하드웨어장치를 필요하며 가격 또한 적지 않다는 점이 단점이라 할 수 있다.

2.2. 프로세서 시뮬레이션

만약 프로그램이 시뮬레이터 상에서 완벽히 수행이 가능하다면, 모든 수집 가능한 정보들을 완벽하게 수집이 가능하다. 완벽한 프로파일 결과를 제공해줄 수 있지만 앞에서도 설명하였듯이 이를 위하여 시뮬레이터는 모든 하드웨어 구성요소들을 시뮬레이션 하여야 하며, 다양한 타깃 시스템을 위한 구현이 필요하다는 점과 같은 불리한 점이 있다.

2.3. Instruction-inlining

실행 이미지를 직접 변형하여, basic block의 시작 부분과 같이 얻고자 하는 정보가 있는 위치에 새로운 명령어를 추가하여 해당하는 부분이 수행될 때 마다 기록하여 원하는 정보를 추출하는 방식이다. 우선 소프트웨어 적으로 구현이 가능하기 때문에 다른 타깃 시스템에 영향을 받지 않는다는 장점이 있으며 프로그램의 실행에 있어서 부하 자체도 크지 않다. 반면 기록을 저장할 저장장치가 없거나 충분치 않다면 해결 방법을 찾아야 하며, 프로그램의 구조적 분석을 통하여 이전 프로그램의 수행에 영향을 미치지 않게 명령어를 삽입해야 한다는 어려운 점이 있다.

2.4. Single-stepping

디버깅 기능인 브레이크포인트[7,8] 명령어를 이용하여 각각의 명령을 수행하는 방식이다. 사용자의 응용프로그램에 대한 정보를 추출하여 내는 데에는 간편하고 좋은 방법이지만 커다란 단점이 있다. 매 명령어 또는 얻고자 하는 정보를 얻을 수 있는 위치마다 브레이크포인트 명령에 의한 인터럽트가 발생함으로 인하여 프로파일 수행 성능이 크게 느려진다. 하지만 우리는 single-stepping방법을 사용하고 있는데, 그 이유는 다음과 같다.

오픈소스 디버거인 GDB를 쉽게 이용할 수 있으며, 이미 여러 프로세서를 목표로 개발되어 있어, 이동성이 용이하다. 특히 gdbserver는 원격 디버깅을 지원하여, 호스트 컴퓨터에서 프로파일 기록을 저장할 수 있어 하드웨어 자원이 제한적인 임베디드 시스템에서의 사용이 적합하다.

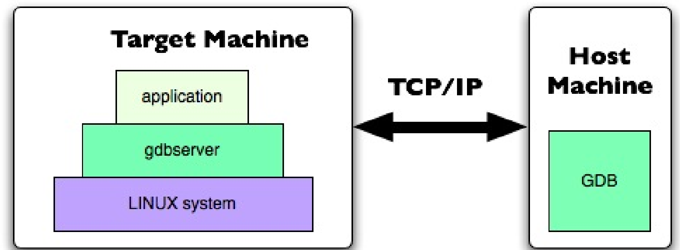

그림 2. 동작 개요

3. 기본 프로파일 시스템

프로파일의 시작은 GDB와 gdbserver의 기본기능을 이용하여 이들 간의 통신 내용을 파일에 기록하는 것으로부터 시작한다. GDB는 프로파일 기록을 저장할 호스트 컴퓨터에서 실행하고 타깃 보드에 맞게 수정된 gdbserver는 타깃 보드 상에서 응용프로그램을 실행시키고 호스트 컴퓨터와 통신을 할 수 있는 구조로 이루어져 있다. 현재 구현된 프로파일 시스템의 경우

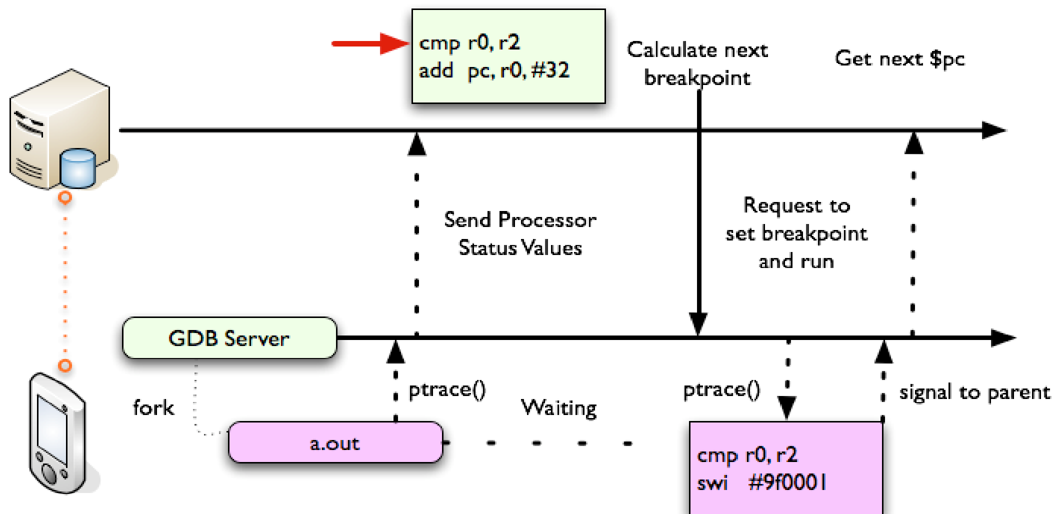

그림 1. stepi 명령어 수행 방식

ARM 프로세서에 대하여 구현되어 있는 상태이다. gdbserver의 경우 여러 프로세서에 대하여 구현코드가 존재하며, 코드의 크기 또한 크지 않아 다른 타깃에 대해서도 쉽게 확장 가능 하다.

우리의 프로파일 시스템에서는 수행하는 명령어의 주소뿐만 아니라 각 명령어를 분석하여, 해당하는 명령이 데이터를 저장 또는 불러오는 경우 그 대상이 되는 메모리의 주소 또한 결과로 제공하여 준다.

GDB의 기본 명령어인 stepi는 정확히 어셈블리 명령어 하나를 수행한다. Gdbserver와 GDB를 이용한 환경에서 stepi의 동작 원리는 다음과 같다.

0x8000	CMP r0, r2
0x8004	ADD pc, r0, #32

- 프로그램 레지스터 PC의 값이 0x8000이고 수행할 명령어들이 위의 표와 같다고 할 때 호스트 쪽 에서는 현재 수행할 정보들을 알기 위하여 gdbserver쪽에 현재의 PC 레지스터의 값과 그 값이 가리키고 있는 메모리의 값을 요청한다.

- 읽어온 메모리의 값을 분석하여 다음 수행 할 명령어 주소를 지정한다.
  - ■ 요청에 의해 값이 호스트에 전해지면, 현재 수행할 명령이 조건부 실행인지를 검사하고 조건이 있다면 CPU상태 레지스터의 값을 읽어와 비교하고, 조건에 의하여 실행되지 않는 명령이라면 다음 번지에 있는 명령을 읽어와 다시 다음 실행 명령의 위치를 검사한다.
  - ■ 명령 분기가 아니라면 다음 PC 레지스터 값에 해당하는 0x80004 번지에 특정 명령어 "SWI #9f0001"이라는 명령어를 적고, 프로그램을 다시 수행하도록 타깃 보드에 요청한다.
    - ◆ 명령이 LDR또는 STR와 같은 데이터 처리 명령이라면 명령의 대상 주소 값의 계산을 위해 특정 레지스터의 값을 요청한다.
  - ■ 다음 수행할 대상 명령이 PC 레지스터의 값을 변경하는 명령이라면, 명령어를 분석하여 대상 주소를 계산하거나 조건부 수행의 판단을 위하여 특정 레지스터의 값을 다시 gdbserver에 요청하여 읽어 들여야 한다.

- 해당 SWI명령은 OS에서 trap을 발생시키는 명령어로 그 지점에서 멈추고 대상 프로그램의 부모인 gdbserver를 깨운다. 깨워진 gdbserver는 SWI명령을 원래의 명령으로 수정하고, 다시 호스트의 GDB와 통신을 하여 현재 PC값과 같은 정보를 제공하며 맨 처음 동작을 반복 한다.

가령 0x8004번지의 명령의 경우 분기 명령인 B또는 BL이 아니지만, ADD명령의 결과를 프로그램 레지스터에 저장함으로써 동일한 효과를 얻게 되며, 해당 다음 명령의 주소를 계산하기 위하여 레지스터 r0의 값을 읽어와 다음 명령의 주소를 계산한다.

이러한 기본적인 동작을 이용하여, 응용 프로그램이 종료할 때까지 각 명령어를 하나씩 수행해나가며, 호스트 쪽에서는 수행하는 명령들의 주소와 메모리 참조 주소를 파일로 저장한다.

4. 성능의 향상을 위한 방법

우리가 제시한 방법은 이전의 일반적인 instruction-stepping과 크게 다르지 않지만, 서버 클라이언트 모델을 이용하여, 수행 결과를 호스트에

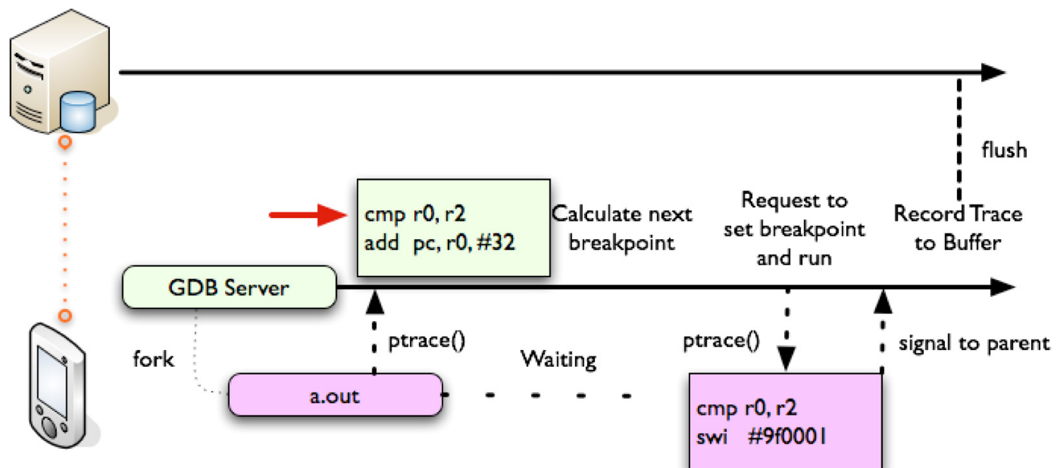

그림 3. 버퍼를 이용한 성능 향상

저장 가능하다는 장점이 있다. 이 방법의 가장 큰 걸림돌은 프로파일을 수행하는데 걸리는 부하이다. 현 방법에 있어 추가적 비용이 발생하는 부분은 크게 세 부분으로 나눌 수 있다.

- 호스트와 타깃 사이의 기록 전송에 따른 통신 비용
- “SWI”명령을 사용하여 브레이크포인트를 생성하여 프로그램이 멈춤으로 인하여, 예외 처리를 하기 위한 gdbserver와 대상 응용프로그램의 프로세스간 통신 비용
- 다음 번 수행될 명령의 주소를 계산하기 위하여, 명령어를 종류를 분류하고, 분기 대상 주소를 계산하는 비용

앞에서 설명하였듯이 GDB에서 *stepi*명령의 간단한 확장으로 명령어 단위의 수행 결과를 기록할 수 있었지만, 실제 적용 결과 초당 약 50개의 명령어를 수행하는 정도의 속도였다. 이 속도는 실제 프로그램을 프로파일 하기에 너무나 느려 속도 향상의 방법을 모색하였다.

현재 단계의 방법에서의 주 성능 저하의 원인은 기록 전송에 따른 과도한 통신비용으로, <그림 1>에서 볼 수 있듯이 하나의 명령을 수행하기 위하여 다수의 서버/클라이언트 간 통신이 일어난다는 점이다. 또 다른 문제점으로는 매 명령마다 break-point를 설정하여 응용 프로그램과 gdbserver 프로세서 간의 통신이 발생하는 점을 들 수 있겠다. 우선은 성능의 향상을 위하여 서버/클라이언트 간의 통신 횟수를 줄이는 방법을 적용하였다.

이전의 방법에서 다음 번 수행할 명령어의 주소를 계산하고 그것을 지시하는 부분을 호스트 쪽에서 처리하였으나 이 부분을 타깃 보드로 이전하여 통신 횟수를 줄일 수 있었다. 또한 매 명령마다 호스트와의 통신이 아닌 프로파일의 결과를 임시로 저장하는 기록 버퍼를 두어 버퍼가 찰 때만 호스트 컴퓨터와 통신이 일어나는 방식을 사용하였다. <그림3>에서와 같이 단지 버퍼가 가득 차게 되었을 경우만 호스트와 통신이 일어나게 된다.

통신을 줄이기 위하여, 이전까지 호스트에서 하던 다음 수행하게 될 명령어 분석을 타깃 보드에서 함으로 인하여 그만큼의 성능 부담이 생겨나지만, 통신을 줄임으로써 얻는 이득이 훨씬 큼으로 성능 향상을 이룰 수 있었다.

마지막으로 gdbserver와 응용프로그램의 프로세서 간 통신비용에 대한 최적화는 이루어 지지 않았으며, 마지막 6장에서 다시 고려 하도록 하겠다.

5. 결과

이 장에서는 간단한 응용프로그램에 대한 프로파일 성능 결과를 보여 준다. 호스트 컴퓨터는 Linux를 사용하는 Pentium4를 사용하였고, 타깃 보드로는 210Mhz로 동작하는 ARM926EJ-S 개발 보드를 사용하였다. 두 머신의 연결은 100M Ethernet을 통하여 연결 하였다.

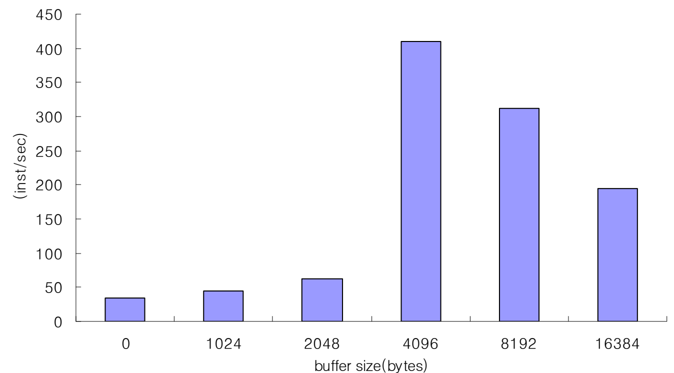

그림 4. 프로파일 성능

<그림4>의 결과에서 알 수 있듯이 버퍼를 사용하였을 경우 이전에 비하여 좋은 성능들을 나타내고 있으며 특히 4Kbyte일 때 성능이 가장 좋게 나타난다. 1Kbyte 또는 2Kbyte일 경우 원래 호스트 쪽에서 명령 분석을 하여, 다음 수행 명령어의 위치를 확정 지었던 동작을 타깃 보드 쪽에서 수행함으로써 생기는 부담이 생겨 커다란 성능 향상을 보이지 않으나 어느 정도 버퍼가 커짐에 따라 두 실험 컴퓨터 사이의 통신빈도가 줄어 들어 이러한 부담을 덮게 된다. 4Kbyte가 넘어가는 경우는 메모리 사용량의 증가로 인하여 응용프로그램의 수행에 방해가 된다.

6. 결론

이 논문에서 소개한 방식은 구현하기 간단하고 다양한 타깃에 적용 가능하며 운영체제 위에서 동작하는 응용 프로그램에 대하여 적용 가능하다는 장점이 있다. 하지만 상당히 느린 프로파일 속도를 가지고 있으며, 운영체제의 코드 자체에 대해서는 기록을 뽑아 낼 수 없다는 단점 또한 가지고 있다.

앞선 결과에서 볼 수 있듯이 두 실험 장비 사이의 네트워크 통신 부하는 버퍼를 사용함으로써 크게 사라짐을 알 수 있으나, 매 명령어에 대하여 브레이크포인트를 설정함으로써 gdbserver와 응용 프로그램 사이의 프로세서 통신이 일어난다는 점은 아직 해결하지 못한 문제로 남아있다. 이전의 single-stepping을 통한 프로파일 기법들은 이러한 문제를 basic block단위의 브레이크포인트 설정등과 같은 보다

큰 단위를 기준으로 브레이크포인트를 설정하는 빈도를 줄여 해결하고 있다. 이러한 해결책은 우리의 방법에도 쉽게 적용 가능한 형태이다. 제시된 방법에 의해 제공되는 명령어의 수행 기록과 데이터 접근에 대한 정보는 시스템 엔지니어 혹은 응용프로그램 개발자에게 유용하게 사용될 것이다.

우리는 보다 빠른 프로파일 방법을 위하여 instruction-inlining과 대상 프로그램의 부분적 전 분석을 통한 성능 향상을 기대하고 있으며, 기능적 시뮬레이션 기법의 단점인 하드웨어 장치의 구현의 어려움을 극복하기 위하여 본 기법을 통한 하드웨어 장치의 동작의 결과를 다시 시뮬레이션에 이용하는 등과 같은 이용 방법을 모색 중이다.

7. Acknowledgements

본 연구는 정보통신부 및 정보통신연구진흥원의 IT신성장동력핵심기술개발사업[2006-S-040-01, Flash Memory 기반 임베디드 멀티미디어 소프트웨어 기술개발]과 한국학술진흥재단의 BK21 사업의 일환으로 수행하였습니다. 또한 이 연구를 위해 연구장비를 지원하고 공간을 제공한 서울대학교 컴퓨터연구소에 감사드립니다.

참고 문헌

[1] Agarwal , R. L. Sites , M. Horowitz, ATUM: a new technique for capturing address traces using microcode, Proceedings of the 13th annual international symposium on Computer architecture, p.119-127, June 02-05, 1986.

[2] Chriss Stephens , Bryce Cogswell , John Heinlein , Gregory Palmer , John P. Shen, Instruction level profiling and evaluation of the IBM/6000, Proceedings of the 18th annual international symposium on Computer architecture, p.180-189, May 27-30, 1991.

[3] Anita Borg , R. E. Kessler , David W. Wall, Generation and analysis of very long address traces, Proceedings of the 17th annual international symposium on Computer Architecture, p.270-279, May 28-31, 1990.

[4] S. J. Eggers , David R. Keppel , Eric J. Koldinger , Henry M. Levy, Techniques for efficient inline tracing on a shared-memory multiprocessor, Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems, p.37-47, April 1990.

[5] Douglas W. Clark and Joel S. Emer, Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement, ACM Transactions on Computer Systems, 3(1):p.31-62, February 1985.

[6] Gurindar S. Sohi and Manoj Franklin, High-Bandwidth Data Memory Systems for Superscalar Processors, In Proc. of 4th Conference on Architecture Support for Programming Language and Operating Systems, p. 53-62 ACM,1991.

[7] Peter B. Kessler, Fast Breakpoints: Design and Implementation. SIGPLAN Conference on Programming Language Design and Implementation, p78-84 1990.

[8] Norman Ramsey, Correctness of trap-based breakpoint implementations, Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, p.15-24, January 16-19, 1994.

[9] J. Kelly Flanagan, Brent E. Nelson, James K. Archibald, and Knut Grimsrud, "BACH: BYU Address Collection Hardware, The Collection of Complete Traces." Proceedings of the Sixth International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, p. 128-137. 1992.

[10] Charlton D. Rose, J. Kelly Flanagan, Constructing instruction traces from cache-filtered address traces (CITCAT), ACM SIGARCH Computer Architecture News, v.24 n.5, p.1-8, Dec. 1996.