논문 2009-04-25

임베디드 환경에서 응용프로그램 시작의 가속 기법

(Acceleration Techniques of Application Startup for Embedded Systems)

박 은 병*, 이 용 준, 김 승 균, 이 재 진, 박 경 민
(Eun-Byung Park, Yong-Jun Lee, Seungkyun Kim, Jaejin Lee, Kyungmin Park)

Abstract : Due to digital convergence, mobile embedded systems need more functionalities and a fully fledged OS. Applications for such embedded systems are linked with many shared libraries available in the OS and access a large data set at launch time. This results in increased application launch time. In this paper, we propose two techniques for reducing the application launch time: lazy-loading and pinning. Lazy-loading defers loading shared libraries that are not used in the application at launch time, whereas pinning guarantees the residence of shared libraries and data used at launch time in the main memory.

Keywords : Embedded systems, Lazy-loading, Pinning, Application launching time

I. 서 론

모바일 시스템에 대한 다양한 기능적 요구로 인하여, 범용 운영체제인 Linux 기반 플랫폼의 탑재가 일반화되고 있다. 이에 따라 범용 운영체제 기반으로 작성된 다양한 라이브러리들을 모바일 환경에서 사용할 수 있게 되면서 쉽고 빠른 응용프로그램의 개발이 가능해졌다. 하지만 동적 링크하는 공유 라이브러리의 수가 증가하고 요구하는 데이터의 양이 많아짐에 따라 응용 프로그램의 시작 시간이 늘어나게 되었다. 모바일 임베디드 시스템에서 사용자 요청에 대한 응답시간은 매우 중요하므로, 이를 만족 시키고자 응용 프로그램의 시작 시간 단축에 대한 요구는 점점 커지고 있다

프로그램의 시작 시간에 영향을 미치는 요소는 크게 두 가지로 나뉜다. 첫째로, 프로그램이 실행될 때 참조하는 공유 라이브러리의 동적 링킹 과정이다. 사용자의 편의를 위해 GUI(Graphic User Interface) 환경의 프로그램이 일반적이며, 이는 많은 수의 공유 라이브러리 동적 링킹을 동반하게 된다. 둘째로 I/O 요청에 의한 시작 시간의 증가이다. 블록 디바이스로의 접근(Disk, Flash memory)은 메모리로의 접근보다 수 천 배 이상의 지연 시간이 요구 된다. 범용 운영체제에서는 이 둘 사이의 차이를 감소시키기 위해 이전에 참조하였던 데이터에 대해서는 커널 내부의 페이지 캐시에 유지하려고 노력한다. 하지만 메모리의 양이 크지 않은 임베디드 환경에서 페이지 캐시의 크기 역시 작으므로, 많은 양의 데이터를 유지하기 어렵다. 따라서 프로그램은 메모리의 양이 충분한 PC환경에서보다 더 많은 I/O를 요청하게 되며 시작 시간 역시 늘어나게 된다.

프로그램의 시작 시간을 단축하기 위한 다양한 연구가 있다. 널리 사용되는 임베디드 플랫폼인 구글 안드로이드(Android)[1]는 pre-fork process 기법을 사용해 프로그램의 동적링킹 과정과 초기화 과정의 시간을 단축한다. 하지만 Java로 작성된 프로그램에서만 유용하며, JVM(Java Virtual Machine)의 수정을 통해서만 가능하다. 이와 유사한 방법으로 fork-dlopen모델[2]에 대한 연구가 있다. 시작 시간의 단축에는 크게 기여하나 프로그램을 공유

* 교신저자(Corresponding Author)
논문접수 : 2009. 12. 13.,
수정·채택확정 : 2009. 12. 26.
박은병, 이용준, 김승균, 이재진 : 서울대학교 컴퓨터공학부
박경민 : 삼성전자
※ 본 연구는 교육과학기술부/한국과학재단 창의적연구진흥사업(매니코어프로그래밍연구단, 0421-20090025), 한국학술진흥재단 BK21 사업의 지원으로 수행되었음.

라이브러리 형태로 컴파일 해야 하는 단점이 있다. 이외에도 프로그램의 I/O 요청을 줄이기 위해 프로그램의 실행 전에 미리 데이터를 읽어 메모리에 적재하는 방법이 있다[3]. 하지만 이 방법은 사용자의 프로그램 사용 패턴을 분석, 예측 하며 정확한 예측이 따르지 않으면 불필요한 메모리의 사용을 야기한다.

본 연구에서는 프로그램의 시작 시간을 줄이기 위해 lazy-loading과 pinning 기법을 제안한다. Lazy-loading은 시작 시간에 참조하지 않는 공유 라이브러리의 로딩을 늦춤으로써 동적 링킹에 대한 시간을 감소시키고, Pinning은 시작 시간에 참조하는 공유 라이브러리 및 데이터의 메모리 상주를 보장함으로써 프로그램의 I/O 요청을 최소화 한다.

Ⅱ. 프로그램 프로파일링

본 장에서는 프로그램 프로파일링에 대하여 설명한다. 세 가지 측면에서 프로그램이 참조하는 공유 라이브러리 및 데이터를 추출 하였다. 이는 프로그램 최초 실행 시 한번으로 충분하며, 이후 프로그램 실행에는 미리 구축된 프로파일링 정보를 참조한다. 따라서 프로파일링에 요구되는 성능 저하는 미미하며 최초 프로파일링시 약간의 성능 저하는 용인 되어 질 수 있다.

1. 동적 링커 프로파일링

동적 링커(ld.so)는 Linux 기반의 모든 프로그램이 실행될 때 가장 먼저 수행되는 시스템 공유 라이브러리이다. 동적 링커는 프로그램이 참조하는 공유 라이브러리를 프로그램의 주소 공간에 사상(mapping)하고, 재배치(relocation)를 수행 한다. 공유 라이브러리는 프로세스 주소 공간에 사상되는 주소가 프로그램 실행 시에 결정되므로 동적 링커에 의한 재배치 과정이 필요하다. 이 과정에서 시작 시간의 향상을 위해 모든 심볼에 대한 재배치를 하지 않으며, 함수(procedure) 심볼에 대해서는 최초 호출 시에 재배치한다. 따라서 프로그램이 공유 라이브러리의 함수를 호출하면, 동적 링커로 제어권이 전달되고 동적 링커를 통해 프로그램이 시작 시간에 참조하는 라이브러리 및 함수, 함수에 대한 공유 라이브러리의 파일 내 위치, 그리고 함수를 포함하는 페이지를 알 수 있다.

2. Page-analyzer

프로그램이 공유 라이브러리의 함수를 호출하면, 해당 함수 내에서 또 다른 함수로의 호출을 동반한다. 이렇게 라이브러리 내부에서 호출되는 함수는 동적 링커를 거치지 않는다. 따라서 내부 함수 호출이 접근하는 페이지를 검출하기 위해 page-analyzer를 도입한다.

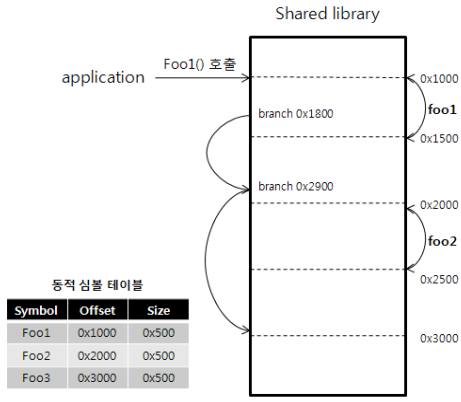

그림 1. Page-analyzer 분석 과정
Fig. 1. Process of page-analyzer

그림 1에서 프로그램은 라이브러리의 foo1 함수를 호출한다. Page-analyzer는 함수를 포함하는 공유 라이브러리를 역 어셈블 하고 foo1의 시작 주소인 0x1000부터 0x1500사이의 명령어를 검색하여, 함수 호출 및 분기를 야기하는 명령어를 찾는다. Branch 명령은 물론 add, sub, mov 등 PC(program counter)를 변경하는 모든 명령이 여기에 해당된다. branch 0x1800명령이 발견되며, 0x1800부터 다시 명령어를 검색한다. 내부 함수인 0x1800에 대한 심볼 정보는 라이브러리 배포 시에 파일크기를 줄이기 위해 제거 되므로 함수의 크기를 알 수 없다. Page-analyzer는 내부함수의 크기를 정하기 위해 동적 심볼 테이블의 심볼들을 경계로 삼는다. 따라서 0x1800부터 foo2의 주소인 0x2000까지 탐색하며 branch 0x2900명령이 발견되었다. 위의 과정을 통해 foo1()함수가 접근한 페이지는 0x1000, 0x2000으로 2 페이지에 해당된다.

3. 시스템 콜 추적

동적 링커와 page-analyzer를 통해 프로세스 주소 공간에 사상된 라이브러리 및 데이터에 대하여 시작 시간에 참조하는 페이지를 추출 할 수 있었다. 하지만 이 외에도 read 시스템 콜을 사용하여 I/O요청을 하며, 이

역시 시작 시간에 큰 영향을 미친다. 프로그램이 호출하는 시스템 콜을 추적하기 위해 strace 디버깅 도구를 사용한다. Strace는 시스템 콜이 호출된 순서대로 기록을 남기고, 시스템 콜의 이름 뿐 만 아니라 매개변수, 리턴 값 등 I/O 요청 분석을 위한 유용한 정보를 제공한다. 또한 실행 중인 프로그램에 동적으로 적용이 가능하며, 다른 프로파일링 도구에 비해 프로그램의 성능 저하가 매우 작아 본 연구에 사용하기 적합하다. 프로그램이 실행되면 실행된 프로그램에 동적으로 strace를 적용하고 strace가 남긴 시스템 콜 추적 로그를 통해 open, read, close등을 분석하여 프로그램이 접근한 파일 경로, 접근한 파일 내의 위치 및 크기를 알 수 있다.

Ⅲ. 전체 시스템 구성

1. Lazy-loading

동적 링커는 프로그램이 참조하는 모든 라이브러리를 프로세스 주소 공간에 사상하는데, 이를 위해 open, read, mmap 등의 시스템 콜과 여러 번의 I/O요청을 동반한다. 따라서 참조하는 라이브러리의 수가 많아지면, 성능 저하가 따르게 되는데, 이점에 착안하여 시작 시간에 참조하지 않는 라이브러리는 프로세스 주소 공간에 사상하지 않으며, 최초 참조 시에 사상한다. 앞서 설명하였듯이, 공유 라이브러리 함수의 최초 호출 시에는 항상 동적 링커로 제어권이 전달되므로, 사상되지 않았던 공유 라이브러리에 대한 함수 호출시 해당 라이브러리의 사상을 동적 링커에서 수행할 수 있다.

2. Pinning

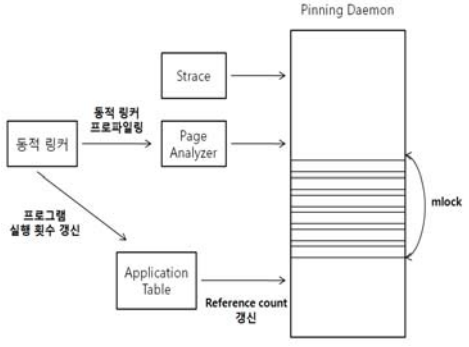

그림 2. Pinning daemon 구성도
Fig. 2. Structure of pinning daemon

동적 링커 프로파일링, page-analyzer, 시스템 콜 추적을 통해 프로그램이 시작 시간에 참조하는 라이브러리 및 데이터를 페이지 단위로 추출하였다. Linux에서는 프로세스의 특정 주소 공간에 사상된 페이지의 회수를 금지하는 mlock 시스템 콜을 제공한다. Pinning 기능을 수행하는 데몬 프로세스를 생성하고, 데몬 프로세스의 주소 공간에 프로파일링 결과로 산출된 페이지에 mlock 시스템 콜을 적용시킨다. 페이지의 메모리 상주가 보장되므로 해당 페이지를 참조하는 프로그램은 I/O 요청이 감소하여 시작 시간이 단축된다.

Pinning의 적용 대상 페이지를 선정함에 있어서 reference count를 정의한다. 자주 실행되는 프로그램일수록 사용자가 후에 또 실행할 가능성이 높으므로 각 프로그램 별 실행 횟수를 reference count에 반영한다. A 프로그램의 실행 횟수가 3, B 프로그램의 실행 횟수가 5라고 하자. A 프로그램이 참조한 페이지는 a,b 이며 B 프로그램이 참조한 페이지가 b,c이면 각 페이지에 대한 reference count는 a=3, b=8(3+5), c=5 이다. 모든 프로그램은 시작 전 동적 링킹 과정을 거치므로, 동적 링커에서 프로그램 실행 횟수를 application table에 기록하고, 주기적으로 깨어나는 pinning 데몬은 실행 횟수를 바탕으로 reference count를 갱신한다. Application table에는 누적된 프로그램의 실행횟수가 기록 되어 있다. 예전에 자주 실행되었던 프로그램이지만 현재 자주 실행되지 않는다면, 해당 프로그램의 실행횟수는 최신의 정보가 아니다. 따라서 일정 시간을 주기로 실행횟수를 감소시켜 프로그램의 실행 횟수가 현재 시스템의 상태를 반영하도록 하였다.

Ⅳ. 실험 및 평가

1. 실험 환경

실험에서 사용된 평가 보드는 TI사의 beagle board rev C2이다. OMAP3530 processor가 탑재되었고 600MHz의 ARM Cortex-A8코어를 사용하며, 256MB LPDDR RAM의 메모리를 가진다. 또한 Nokia에서 주도하는 Linux기반 오픈소스 임베디드 플랫폼인 Maemo 5.0 alpha를 사용하며, Maemo에 존재하는 10개의 다양한 프로그램에 대해 실험하였다.

2. 시작 시간 측정 방법

사용자가 실행시킨 프로그램의 윈도우 창이 화

면에 보여 지고, 사용자가 원하는 명령을 프로그램에게 보낼 수 있는 시점까지를 시작 시간이라고 정의한다. Maemo 플랫폼의 프로그램은 기본적으로 GTK 라이브러리 기반으로 작성되는데, 정확히 이 시점까지의 시간을 측정 할 수 있는 idle callback 함수를 제공한다[4]. 10개 중 2개의 프로그램(hex-a-hop, barrage)은 idle callback을 사용할 수 없어서, 프로그램 메뉴가 화면에 나타난 직후에 시간을 측정 하도록 소스를 수정하였다.

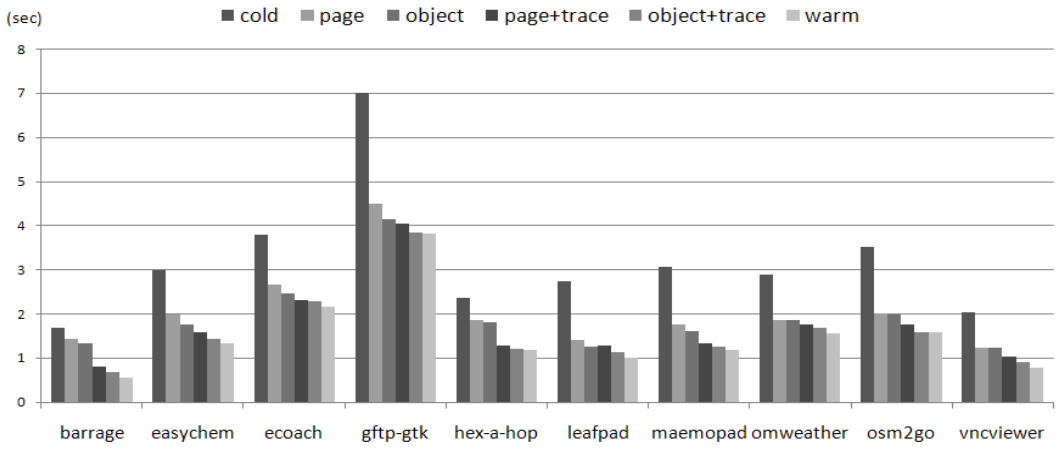

그림 3. Pinning 적용 후 시작 시간 비교

Fig. 3. Comparison of launching time after applying pinning

커널 내부의 페이지 캐시에 프로그램이 참조하는 페이지의 적재 유무에 따라 시작 시간이 큰 차이를 보이므로 warm cache와 cold cache 두 경우로 나누어 측정한다. warm cache는 프로그램을 여러 번 수행시켜 프로그램이 참조하는 페이지를 페이지 캐시에 적재시킨 후에 시작 시간을 측정한다. 이와 반대로 cold cache는 페이지 캐시에 있는 페이지들을 drop후 측정하는데 linux에서는 이를 위한 인터페이스를 제공 한다(echo 1 > /proc/sys/vm/drop_caches)[3].

3. 측정 결과

그림 3는 cold cache와 warm cache를 기준 으로 pinning을 통한 시작 시간의 단축을 각 항목 별로 비교한다. 각 프로그램이 참조한 페이지 및 데이터를 pinning 후에, 페이지 캐시를 drop한다. Pinning을 하여 페이지 회수를 방지 하였으므로 drop후에도 여전히 페이지 캐시에 유지된다. pinning을 하는 대상에 따라 page, object, page + object, page + trace, object + trace의 총 다섯 가지 항목으로 나눈다. Page는 page-analyzer의 분석을 통해 검출된 page, object는 page-analyzer를 사용하지 않고 프로그램이 참조하는 모든 공유 라이브러리 및 데이터를 object 단위로 pinning한다. Trace는 시스템 콜 추적 후에 검출된 page이며 이를 각각 page와 object항목에 추가하여 실험한다.

Cold cache대비 page + trace 항목에서 40 ~ 55%, object + trace 항목에서 40 ~ 60%의 성능 향상을 관찰 할 수 있었다. 또한 위의 실험에서 측정한 10개의 프로그램에 대해 pinning시 요구되는 메모리 양은 page(20.3MB), object(26.2MB), page + trace(30.8MB), object + trace(38.3MB)이다. page단위 pinning에서, 적은 메모리 양으로 object 단위와 대동소이한 시작 시간을 보인다.

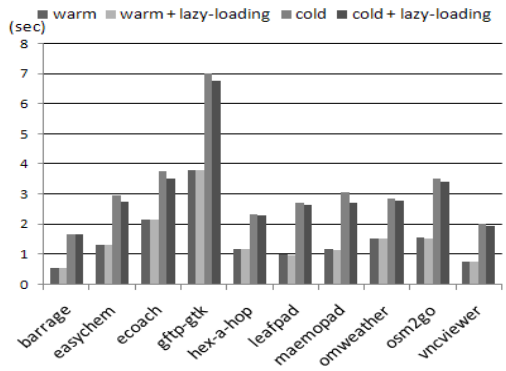

그림 4. Lazy-loading 적용 후 시작 시간 측정 결과

Fig. 4. Launching time of lazy-loaded application

그림 4은 lazy-loading이 적용된 프로그램의 시작 시간을 각 4개의 항목으로 나누어서 측정한 결과이다. warm + lazy-loading은 warm cache에서 lazy-loading 적용 시에 측정 결과 이며, cold + lazy-loading은 cold cache에서 lazy-loading 적용 시에 측정 결과이다. warm cache에서는 성능 향상이 크지 않지만, cold cache 에서는 최대 10.7%(maemopad) 시작 시간 단축을 보였다.

V. 결 론

본 연구에서 프로그램의 시작 시간 단축을 위해 lazy-loading와 pinning기법을 개발 하였다. 메모리 양이 적은 임베디드 환경에서 빈번한 페이지 캐시 회수로 인해 프로그램의 시작 시간이 증가 하는데, 두 기법은 이런 상황에서 만족할 만한 시작 시간 단축을 보여 주었다. 또한 특정 플랫폼에 종속적이지 않으므로 임베디드 Linux기반의 어떠한 환경에서도 쉽게 적용이 가능하다.

참고문헌

[1] http://code.google.com/android.

[2] C. Jung, D. Woo, K. Kim, S. Lim, "Performance characterization of prelinking and preloading for embedded systems", Proc. of 7th ACM & IEEE International Conference on Embedded Software, pp. 213-220, Salzburg, Austria, 2007.

[3] B. Esfabd, "Preload: An adaptive prefetching daemon", MS Thesis University of Toronto, 2006.

[4] L. Colitti, "Analyzing and improving GNOME startup time", Proc. of 5th System Administration and Network Engineering Conference, Delft, The Netherlands, 2006.

저 자 소 개

박 은 병

2009년 경희대학교 컴퓨터공학부 학사. 현재, 서울대학교 컴퓨터공학부 석사과정. 관심분야: 운영체제, 임베디드 소프트웨어, 가상화.

Email: eunbyung@aces.snu.ac.kr

이 용 준

2009년 서울대학교 컴퓨터공학부 학사. 현재, 서울대학교 컴퓨터공학부 석사 과정. 관심분야: 멀티코어 시스템, 임베디드 소프트웨어.

Email: yongjun@aces.snu.ac.kr

김 승 균

2005년 서울대학교 컴퓨터공학부 학사. 현재, 서울대학교 컴퓨터공학부 석/박사 통합과정. 관심분야: 컴파일러 post-pass 최적화, 멀티코어 환경에서의 디버깅 기법.

Email: seungkyun@aces.snu.ac.kr

Email: shunlee@dgist.ac.kr

이 재 진

1991년 서울대학교 물리학과 학사.
1995년 Stanford Univ. Computer Science 석사.
1999년 Univ. of Illinois at Urbana- Champaign, Computer Science 박사.
2000~2002 Michigan State Univ. Computer Science and Engineering 조교수.
현재, 서울대학교 컴퓨터공학부 부교수.
관심분야: Compilers, Computer Architecture, High Performance Computing.
Email: jlee@cse.snu.ac.kr

박 경 민

2003년 서울대학교 컴퓨터공학부 석사.
현재, 삼성전자 재직중.
관심분야: Flash memory, 파일시스템, 메모리.

Email: kyungmin.park@samsung.com