임베디드 시스템에서 가상 메모리 압축 시스템 설계

정 진 우† · 장 승 주††

요 약

임베디드 시스템은 일반 PC(Personal Computer)나 워크스테이션에 비해 느린 CPU와 작은 메모리 공간을 사용하고 있다. 따라서 임베디드 운영체제는 제한된 자원을 효과적으로 사용하도록 설계되어져야 한다. 그런데 임베디드 리눅스의 가상 메모리 관리 기법에서 페이지 폴트가 발생할 경우 스왑 디바이스로 페이지를 이동하는 과정에서 성능 저하가 발생한다. 본 논문에서는 가상 메모리 기법의 효율성을 높이며 메모리 공간의 효율성을 향상 시킬 수 있는 가상 메모리 압축 기법을 구현하였다. 가상 메모리 압축 기법은 임베디드 리눅스의 가상 메모리 관리 기법에서 스왑핑이 발생할 경우 스왑 디바이스로 이동하는 페이지들을 압축하여 이동시킴으로서 스왑핑에서 발생하는 성능저하를 감소시키며, 압축된 스왑 디바이스의 운영으로 메모리의 공간 효율성을 높일 수 있다. 또한 본 논문에서는 메모리 내의 소량의 데이터 압축에 적합한 알고리즘을 고안하여, 압축률의 효율성과 시스템 성능을 향상시키고자 하였다.

Design of Virtual Memory Compression System on the Embedded System

Jin-Woo Jeong† · Seung-Ju Jang††

ABSTRACT

The embedded system has less fast CPU and lower memory than PC (Personal Computer) or Workstation system. Therefore embedded operating system is designed to efficiently use the limited resource in the system. Virtual memory management of the embedded linux have a low efficiency when page fault is occurred to get a data from I/O device. Because a data is moving from the swap device to main memory. This paper suggests virtual memory compression algorithm for improving in virtual memory management and capacity of space. In this paper, we present a way to performance implement a virtual memory compression system that achieves significant improvement for the embedded system.

키워드 : 임베디드 시스템(Embedded System), 가상 메모리 시스템(Virtual Memory System), 가상 메모리 압축(Compression) 알고리즘

1. 서 론

운영체제에서 메모리 관리 서브시스템은 가장 중요한 부분 중 하나이며, 물리적인 메모리의 한계를 극복하기 위한 여러 기법들이 개발되었는데, 가상 메모리 기법이 성공적이다. 가상 메모리(virtual memory)는 메모리를 필요로 하는 프로세스 사이에 메모리를 공유하도록 하여 실제 메모리보다 더 많은 메모리를 가진 것처럼 보이도록 한다. 시스템은 메모리를 페이지(page)로 쪼개고 시스템이 실행되면서 이들 페이지를 보조기억장치로 스왑(swap)한다[11]. 임베디드 리눅스에서도 동일한 이와 같은 가상 메모리 시스템을 사용하고 있다.

가상 메모리 시스템에서의 페이지 폴트(fault)가 일어났을 때 느린 스왑 디바이스로 페이지들이 이동하기 때문에 성능 저하가 발생한다. 페이지 폴트가 일어나면 입출력 연산이 발생하며, 프로세스는 수행 상태에서 슬립 상태로 전이하여 CPU의 점유권을 상실하게 된다. 페이지 폴트가 발생한 페이지를 주기억 장치로 읽어 들이기 위해서 기존 페이지를 스왑 디바이스로 이동시킨다. 이런 페이지 교체 알고리즘으로 사용되는 예로는 대표적으로 LRU(Least Recently Used), LFU(Least Frequently Used) 알고리즘과 FIFO(First In First Out) 알고리즘 등이 있다[14, 15]. 그리고 이 페이지들이 다시 필요하면 스왑 디바이스에서 주기억장치로 읽어 들이게 된다. 이렇게 페이지 폴트가 일어났을 때, 이 페이지들을 스왑 디바이스로 이동시키는 과정에서 보조기억장치로의 입출력 연산으로 인한 속도저하와 성능 저하가 발생한다. 가상 메모리 압축 시스템은 주기억장치 내의 일정 영역에 할당된 압축 캐시 영역에 압축된 페이지를 저장하여 스왑 디바이스로 이동하는 시간과 횟수를 줄임으로서 페이지 폴

※ 본 논문은 2002년도 동의대학교 자체 학술연구비 지원을 받아 이루어졌음.
† 준 회 원 : 동의대학교 대학원 컴퓨터공학과
†† 정 회 원 : 동의대학교 컴퓨터공학과 교수
논문접수 : 2002년 9월 28일, 심사완료 : 2002년 11월 25일

트 응답시간을 최소화하여 전체적인 시스템의 성능을 높일 수 있다[6].

본 논문에서는 제안하는 CAMD(Compression Algorithm for in-Memory Data) 압축 알고리즘은 사전을 기반으로 한 LZ 계열의 압축 알고리즘이다. 압축 알고리즘에서 사용한 사전의 크기는 워드 단위 16개의 버퍼를 사용한다. CAMD 압축 알고리즘은 32비트 단위로 압축을 처리하며, 페이지 크기인 4K바이트 만큼 처리를 하면 압축 과정은 종료가 된다. 먼저 사전을 초기화하고 페이지의 데이터를 32 비트 단위로 처리를 한다. 그리고 32비트의 입력 값에 대한 패턴 분류를 한다. 입력 값이 0x0 혹은 0xffffffff인 경우와 그렇지 않은 경우로 분리한다. 또한, 압축을 할때 사용하는 사전으로 16엔트리의 해쉬 테이블을 사용한다. 본 논문에서 제안하는 CAMD 알고리즘은 적은 해쉬 공간을 이용하여 소량의 데이터를 사용하는 가상 메모리 공간 데이터를 다룬다. 모든 비트가 0 또는 모든 비트가 1인 데이터가 많이 발생하는 가상 메모리의 데이터 특성을 최대한 이용하여 최소한의 압축 데이터 길이를 갖도록 한다. 이런 데이터 이외에 일반적인 데이터들에 대해서는 적은 해쉬 테이블 엔트리(16엔트리)를 이용하여 압축 및 압축을 푸는데 따른 부담을 최소화시켰다.

본 논문에서는 페이지 폴트가 일어났을 때 이들 페이지들을 스왑 디바이스로 이동시키지 않고 주기억장치 내의 압축된 캐시 영역을 할당하여 CAMD 알고리즘을 사용하여 압축된 페이지를 저장한다면 스왑 디바이스로 이동하는 시간과 횟수를 감소하여 페이지 폴트 응답시간을 줄이며 주기억장치에 저장되는 페이지들의 공간 활용도를 높일 수 있다. 또한 모바일 디바이스의 적은 저장 장치의 특성을 고려해서 압축된 스왑 디바이스를 사용하여 보조기억장치의 저장 공간 활용을 높일 수 있다. 본 논문에서는 압축된 캐시 영역을 설계하여 페이지 폴트가 일어났을 때 스왑 아웃되는 페이지를 압축하여 저장하는 가상 메모리 압축 시스템을 구현한다. 또한 가상 메모리 압축 시스템을 적용한 모바일 시스템은 기본적인 메모리 이외의 보조기억장치를 사용할 경우 압축된 페이지들의 사용으로 스와핑이 발생할 경우의 속도 향상과 메모리 공간의 효율성을 기대할 수 있다.

또한 메모리 내의 데이터는 일반적인 압축 알고리즘에서 다루는 데이터와는 다른 특징들을 가지고 있어서 메모리 내의 주소 값이나 배열 데이터와 같은 요소들을 고려하여 압축될 때의 효율성을 높였다. 그리고 압축을 수행 할 때 비트단위로 압축을 하며 사전을 이용하는 압축 기법을 사용하였다. 이렇게 본 논문에서는 메모리 내의 데이터에 적합한 압축 알고리즘을 고안한다.

그리고 본 논문의 구성은 2장에서 관련 연구를 살펴보고, 3장에서 가상 메모리 압축 시스템의 설계, 4장에서는 가상 메모리 압축 시스템의 구현에 대해서 설명한다. 5장에서는 실험 및 성능 평가를 설명보고 마지막으로 6장에서 결론을 내리도록 한다.

2. 관련 연구

가상 메모리 압축 시스템은 1993년에 Fred Douglis에 의해 이미 연구가 되었다. Douglis의 연구는 DECstation 5000/200 워크스테이션에 Sprite OS를 사용하여 가상 메모리 압축 시스템을 실험했다[4]. 그의 실험은 가상 메모리 시스템에서 페이지 폴트가 발생하여 스왑 아웃될 때 페이지들을 스왑 영역으로 이동시키는 과정에서의 부하를 줄이기 위한 것이며, 유동적인 크기의 원형 버퍼의 구조를 가진 압축된 캐시 영역을 만들고, 이 영역에 스왑 아웃되어 스왑 디바이스로 이동하는 페이지들을 저장하여 스왑 디바이스로의 입출력 횟수를 줄임으로써 시스템의 성능을 향상시키고자 하였다. 이 때 사용되는 압축 알고리즘으로 LZRW1을 사용하였다[10].

Douglis의 연구 결과 압축 캐시의 성능은 어플리케이션의 동작에 영향을 받으며, 압축과 입출력 연산의 상대적인 비용에 의존한다는 것을 증명하였다. 그리고 압축 캐시를 사용하지 않은 시스템보다 2~3배 정도의 속도 향상을 할 수 있다는 것을 증명하였다.

그러나 그의 연구는 특정 어플리케이션에서만 스왑 아웃될 때의 작업 부하를 감소시키거나 제거 할 수가 있었으며, 대부분의 모든 응용 프로그램에서는 적용되지 못했다.

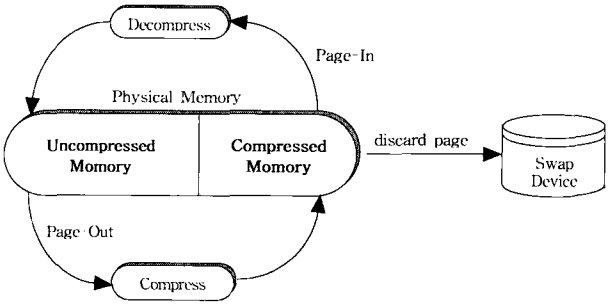
Douglis의 연구 이후 Doug Banks와 Mark Stemm은 포터블 디바이스 내에서의 압축된 메모리 시스템을 제안했는데 Douglis의 연구에서의 문제점으로 데스크탑이나 워크스테이션에서는 압축된 메모리 시스템에서의 시스템 성능향상은 기대할 수 없으므로 포터블 디바이스 내에서의 압축된 메모리 시스템을 고안하였다. Doug Banks와 Mark Stemm의 연구는 PDA(Personal Digital Assistant)나 랩탑과 같은 포터블 디바이스가 느린 통신 수단으로 연결되어 크기가 큰 응용 프로그램에 접근하는 경우이다. 그래서 그들의 실험은 느린 스왑 디바이스로의 접근하는 양을 줄여 시스템의 성능 향상을 증명하였다[1].

그러나 그들의 연구는 포터블 디바이스라는 한정된 영역에서 설계되었으며, 보조기억 장치가 로컬이 아닌 느린 통신 수단으로 연결된 디바이스에 접근하는 경우의 실험을 하였으므로, 일반적인 PC나 워크스테이션에서는 적용하기 힘들다는 문제점이 있다. 그리고 그들이 사용한 운영체제 역시 NACHOS 3.2 버전을 사용하여 진보된 메모리 관리 기법을 사용하는 운영체제와는 다른 실험적인 운영체제를 사용하여 보편적으로 사용되는 운영체제와는 이질성을 보이고 있다[1].

3. 가상 메모리 압축 시스템의 설계

3.1 가상 메모리 압축 시스템의 구조

임베디드 리눅스의 메모리 관리 기법에서는 요구 페이징과 스왑핑 기법에 의해 페이지들이 더티 페이지(dirty page)가 되어 스와핑이 일어나면, 실행중인 프로세스는 입출력 연산이 발생하여 대기 상태로 전환되면서 많은 시간을 기다려야 한다. 이 때 시스템의 성능 저하가 발생하는데 가상 메모리 압축 시스템은 스왑 디바이스로 스와핑하는 페이지들을 주기억장치의 일부분에 압축하여 저장함으로써 스왑 디바이스로의 접근을 줄이는 방법으로 시스템의 전체적인 성능을 향상시킨다[3].



(그림 1) 가상 메모리 압축 시스템의 구조

(그림 1)에서는 가상 메모리 압축 시스템의 전체적인 구조를 나타내고 있는데 물리 메모리를 압축 메모리 영역과 비압축 메모리 영역으로 나누어서 스왑 디바이스로 이동하는 페이지들을 압축하여 압축된 영역에 저장을 하고 다시 페이지 아웃된 페이지들이 필요한 경우에 압축을 풀어서 비압축 영역으로 가져오게 된다. 이 때 압축 영역이 압축된 페이지들로 꽉 차게 되면 압축 메모리 영역을 늘리기 위하여 스왑 디바이스로 압축을 풀어서 보내게 된다[2].

가상 메모리 압축 시스템에서의 중요한 논점은 압축 알고리즘의 성능이라고 할 수 있는데 먼저 속도 면에서 압축을 하고 압축을 푸는 속도가 스왑 디바이스에 read/write 하는 속도 보다 빨라야 한다는 것이다. 초장기의 압축 알고리즘은 파일을 압축하는 목적으로 만들어 졌기 때문에 문자 단위의 압축이었다. 그리고 압축의 효율 또한 만족할 만한 수준이 되지 못해서 가상 메모리 압축 시스템에 적용하기 어려웠다. 그러나 이후에 많은 발전을 하였으며, 압축률과 속도 면에서 성능이 우수한 압축 알고리즘이 등장하고 있다[9, 13]. 그리고 압축 알고리즘의 선택에서 문자 단위의 데이터가 아닌 메모리 내의 데이터를 압축한다는 점에 대해서 고려를 하여야 한다.

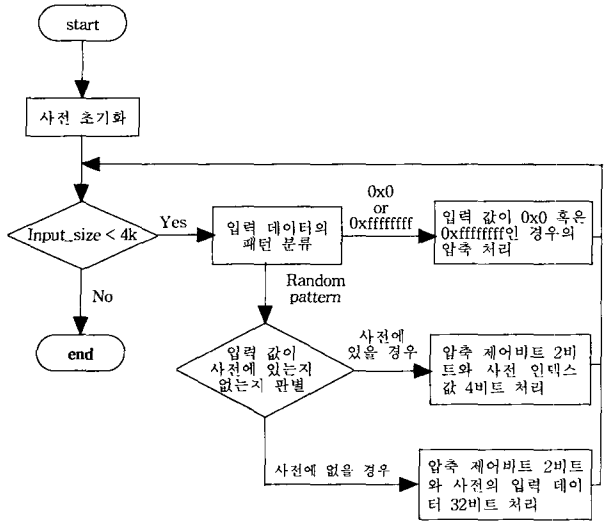
3.2 압축 알고리즘

압축 알고리즘에서 고려해야 할 사항은 압축율과 실행 속도, 그리고 메모리의 요구량을 우선적으로 생각해야 한다. 포터블 디바이스의 특성상 페이지의 크기가 일반 PC나 워크스테이션 시스템에 비해 작다는 것이다[5]. 리눅스의 경우 일반적으로 Alpha AXP 시스템은 8KB 페이지를, Intel x86 시스템은 4KB 페이지를 사용하고 있다. 그에 비해 임베디드 리눅스의 경우는 512B 혹은 1KB 페이지를 사용하고 있으므로 압축 알고리즘은 범용적인 알고리즘 보다는 적은 양의 압축에서 효율이 높고 압축 알고리즘으로 동작 원리가 단순한 알고리즘을 적용하여 페이지의 스왑 아웃되는 속도와 압축하는 속도의 차이를 감소시킨다면 시스템의 성능 향상을 기대할 수 있다. 그리고 포터블 디바이스의 경우 한정된 저장 공간을 사용하고 있다. PDA의 경우는 RAM (Random Access Memory)을 보조기억장치로 사용하는데 RAM에는 일반 응용 프로그램과 데이터, 그리고 스왑 디바이스 영역과 함께 사용이 된다. 따라서 압축된 스왑 디바이스의 사용으로 저장 공간을 늘린다면 적은 저장 공간을 응용 프로그램이나 데이터를 저장하는데 사용하여 시스템의 저장 용량을 향상시킬 수 있다.

압축 알고리즘의 종류에는 크게 무손실 데이터 압축과 손실 데이터 압축으로 나눌 수 있다. 무손실 데이터 압축의 종류에는 RLC(Running Length Coding), VLC(Variable Length Coding), Huffman coding, DBC(Dictionary Based Coding), LZ(Lempel-Ziv)-base와 같은 것들이 있다[12]. 본 논문에서는 LZ 기반의 알고리즘인 LZW 압축 알고리즘을 기반으로 하여 수정한다. LZ 기반 알고리즘은 압축할 자료를 코드화 하면서 기억장치 내에 문자열에 대한 색인표(Index Table)를 생성하여 한 문자열씩 확인하여 그 내용이 같은 문자열을 다시 만나면 그것을 미리 간직해둔 하나의 색인을 갖게 된다. 그렇게 중복되는 모든 부분들은 색인표를 가리키는 하나의 색인을 가지기 때문에 크기를 줄일 수 있다.

본 논문에서 설계한 압축 알고리즘은 사전을 기반으로 압축 알고리즘의 기법을 적용하였다. 압축 알고리즘에서 사용한 사전의 크기는 워드 단위 16개의 버퍼를 사용한다. (그림 2)에서는 압축 과정을 플로차트로 보여주는데, 먼저 입력 값이 0x0 혹은 0xffffffff인 경우와 그렇지 않은 경우로 분리한다. 0x0 혹은 0xffffffff인 경우의 압축은 입력 값을 아래와 같이 0x00000001과 마스크 연산을 하면 0x0인 경우는 0이 출력될 것이고 0xffffffff인 경우는 1인 출력이 된다. 따라서 입력 데이터가 0x0인 경우는 압축된 데이터로 00(0x0)이 출력되고 0xffffffff인 경우는 01(0x1)이 출력 된다. 그리고 압축된 데이터는 "Compressed_Buffer"에 0x0과 0xffffffff의 압축 결과인 00(0x00)혹은 01(0x01)을 나타내는 2비트만 출력으로 보낸다.

```
Compressed_Buffer = ( 입력 값 32비트 & 0x0000001 )
```



(그림 2) 압축 알고리즘의 플로차트

그리고 입력 패턴이 랜덤한 경우는 입력 값이 사전 데이터에 등록이 되어 있는지 없는지를 판별한다. 만약 입력 데이터가 사전에 등록이 되어있다면 압축 제어 비트 2비트의 출력 값 10(0x2)과 사전의 인덱스 값 4비트를 합한 6비트를 출력한다. 출력 형태는 아래와 같으며, 0x20에서 2는 압축 제어 비트 10(0x2)에 해당하며 0은 사전 인덱스 값이 더해질 4비트의 공간을 할당한 것이다. 그리고 사전 인덱스 값 4비트와 OR 연산을 하면 입력 값이 사전에 등록된 경우의 압축이 이루어진다.

```
Compressed_Buffer = ( 0x20 | 사전 인덱스 )
```

입력 값이 사전에 없는 값인 경우는 압축 제어 비트 11(0x3)과 입력 값 전체 32비트를 합한 34비트의 압축된 값을 출력으로 보낸다. 그리고 입력 값을 사전에 등록하면 된다. 사전에 등록할 때는 입력 값의 마지막 4비트의 값에 따라 사전 인덱스에 해당하는 위치에 등록을 한다. 이것은 사전의 전체 크기가 16개로 제한되어 있어서 입력 값의 마지막 4비트의 값이 최대 16까지 나타낼 수 있기 때문이다. 출력 형태는 아래와 같다. 이 경우 34비트의 출력을 보내기 때문에 2번의 출력으로 압축된 데이터를 보내야 한다. 첫 번째 출력은 압축 제어 비트 2비트를 보내고 두 번째 출력은 입력 값 32비트를 출력으로 보내는 것이다.

```
Compressed_Buffer = 0x3
Compressed_Buffer = 입력 값 32비트
```

전체적인 압축 과정은 위와 같이 동작을 한다. 제안한 압축 알고리즘에서 고려해야 할 사항은 사전 데이터의 운영에 관련된 내용인데 현재 압축 알고리즘에서 사용하는 사전 버퍼는 16개의 배열을 사용하고 있다. 처음에 설계한 방법은 16개의 배열을 큐처럼 사용하여 사전에 새로 등록되는 데이터와 사전 데이터의 검색에서 찾아지는 데이터는 배열의 앞쪽으로 보내서 가장 최근에 접근된 데이터들을 큐의 처음에 위치시킨다. 그리고 큐의 여유 공간이 없고, 사전에 새로운 데이터를 추가해야할 경우 큐의 끝에서 데이터가 소멸하게 된다. 이것은 LRU(Least Recently Used) 기법과 유사한 형태로서 사전 데이터의 검색에서 찾아지는 횟수를 늘려 압축률을 높이려고 했다. 그러나 이 방법은 속도 저하가 심해서 전체적인 성능의 저하로 나타났다. 왜냐하면 매번 사전 데이터를 검색할 때 순차검색을 통한 루프의 회수와 검색된 데이터를 큐의 앞쪽으로 이동시키는 과정에서 루프의 회수가 많아지는 문제로 압축하는데 소비되는 시간이 너무 길게 나타났으며 전체적으로 시스템의 성능 저하가 나타났다.

이러한 단점을 개선하여 설계한 방법은 압축률은 낮으나 속도 면에서 성능 향상을 기할 수 있다. 이 방법은 해시 기법을 사용한 사전 데이터의 접근을 시도하였다. 해시 함수는 입력 데이터를 사전 버퍼의 크기만큼 나눈 나머지 값을 사용하였으나 나머지 연산자의 수행 속도의 저하로 입력 데이터의 하위 4비트를 사전에 인덱스 값으로 사용하는 것으로 수정하였다.

압축을 푸는 과정은 압축하는 과정을 역으로 동작시키면 된다. 압축된 데이터의 형태는 아래와 같으며, 압축된 데이터가 어떤 형태의 데이터인지 나타내는 제어비트 2비트와 제어비트의 경우에 따른 데이터 비트로 구성되어져 있다.

압축 제어 비트	압축된 데이터

<표 1>에서는 압축된 데이터의 제어 비트를 나타내는데 00인 경우 출력 32 비트를 전부 0으로 채우고, 01인 경우는 1로 채우게 된다. 그리고 10인 경우는 사전에 등록된 데이터로 제어 비트 이후에 사전의 인덱스 값 4비트를 읽어서 사전에 있는 데이터를 출력으로 보낸다. 마지막으로 11인 경우에는 제어 비트 이후의 32비트를 출력으로 보내면 된다.

〈표 1〉 제어 비트의 종류

제어 비트	00	01	10	11
제어 비트의 내용	전체 출력으로 0을 채움	전체 출력으로 1을 채움	사전에 등록된 데이터	사전에 등록되지 않은 데이터
제어 비트 이후에 읽을 비트 수	없음	없음	4	32

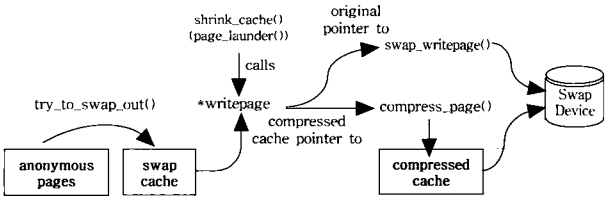
지금까지 압축 알고리즘의 설계에 대해서 살펴보았다. 압축 알고리즘의 설계에서 압축률과 수행 속도는 미묘한 관계로 서로 대칭되는 결과를 보여주고 있는데 압축률을 높게 설계할 경우는 코드의 복잡성으로 수행 속도가 떨어지는 문제점이 있고, 수행 속도를 높이기 위해서는 견고하지

못한 사전 데이터의 운영으로 압축률이 떨어지게 된다.

4. 압축 메모리 시스템의 구현

이 절에서는 제안한 압축 알고리즘을 시스템에 어떻게 구현하는지를 보여 준다. 압축 메모리 시스템의 동작은 크게 스왑아웃(Swap Out)과 스왑인(Swap In)으로 이루어지는데 기존 커널의 메모리 관리 기능을 수정 및 추가하여 압축 메모리 시스템의 기능을 설계한다.

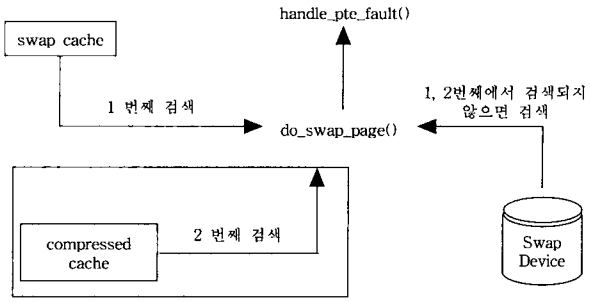
프로세스에 의해 맵핑되지 않는 페이지들을 어노니머스 페이지(anonymous page)라고 하며 이 어노니머스 페이지들은 더 이상 프로세스에 의해 사용되지 않는다. 그래서 이들 페이지는 나이가 들게 되고, 스왑 캐시 영역에서 스왑되어지게 된다. 이 때 어노니머스 페이지들을 스왑 캐시로 보내기 위해 페이지들을 걸러주는 일을 try_to_swap_out()이 처리를 한다[8]. 그리고 try_to_swap_out에 의해 스왑 캐시로 보내진 페이지들은 shrink_cache()에 의해 스왑 아웃이 된다[7].



(그림 3) 스왑 아웃의 과정

(그림 3)은 스왑 아웃(Swap Out)의 전체적인 과정을 보여주고 있다. 스왑 캐시로 들어오는 페이지들은 shrink_cache()의 writepage 호출에 의해 스왑 디바이스로 보내지게 되는데, writepage는 함수 포인터 구조로 되어 있고, 이 함수 포인터는 swap_writepage() 함수의 주소를 가리키고 있다. 스왑 디바이스로의 이동은 swap_writepage()에서 호출하는 rw_swap_page()에 의해서 각 디바이스에 따라 다르게 구현된 코드에 의해 처리가 된다. 여기서 writepage의 주소와 연결된 함수 포인터를 페이지를 압축하여 압축된 캐시로 보내는 compress_page() 함수의 주소로 연결하면 압축 메모리 시스템에서의 스왑 아웃 처리가 이루어지게 된다. 그리고 compress_page()에서 압축 캐시 영역의 공간이 부족하게 되면 기존 커널의 페이지 교체 알고리즘을 이용하여 폐기할 페이지들을 rw_swap_page()를 사용하여 스왑 디바이스로 압축을 풀어서 보내면 된다.

스왑인(Swap In) 과정은 매우 간단하게 동작을 하는데, 페이지 폴트가 발생하면 (그림 4)에서 보는 것과 같이 스왑 캐시에서 페이지를 찾게 되고 만약 스왑 캐시에 없다면 압축 캐시에서 찾는다. 마지막으로 압축 캐시에도 없으면 스왑 디바이스에서 찾게 된다.



(그림 4) Swap In의 과정

스왑인은 handle_pte_fault()가 처리를 하게 되는데, 페이지 폴트가 발생하면 호출이 되며 handle_pte_fault()는 다시 do_swap_page()를 호출한다. do_swap_page()는 첫 번째로 스왑 캐시 내의 페이지 테이블 엔트리를 검색하게 되고 이 때 page_launder()를 사용하게 된다. 만약 찾는 페이지가 발견이 되면 do_swap_page()는 찾은 페이지를 리턴하고 종료를 한다. 스왑 캐시에 찾는 페이지가 없다면 기존의 커널은 스왑 디바이스를 검색하게 되는데 스왑 디바이스를 검색하기 전에 압축 캐시를 검색하게 하여 압축 메모리 시스템을 적용한다. 페이지를 검색할 때 압축된 페이지를 풀어서 검색을 하도록 한다.

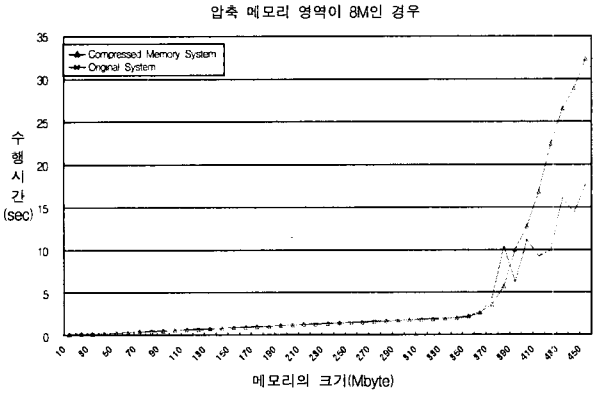
5. 실험 및 성능 평가

가상 압축 시스템의 실험은 임베디드 시스템에서 실험하기 전에 PC 시스템의 리눅스 소스에 적용하여 PC에서 가상 메모리 압축 시스템을 적용한 경우와 적용하지 않은 경우에 대해서 실험을 하였다. 메모리 관리 부분의 임베디디 리눅스 소스 코드와 일반 리눅스 소스 코드의 차이점은 없다. 따라서 일반 PC의 리눅스 시스템에 별다른 포팅 작업 없이 구현을 하였다.

그리고 시스템 테스트를 위해서 물리 메모리를 강제 할당 및 해제를 수행하는 "fillmem"이라는 프로그램을 만들었으며 물리 메모리의 부족으로 페이지 스왑핑이 일어나도록 하였다. "fillmem" 프로그램을 메모리의 크기별로 수행하여 메모리의 할당과 해제가 일어나면서 스왑핑이 일어날 때의 상태를 검사한다. 이 테스트에서 가상메모리에 성능 평가는 스왑이 발생할 경우 시스템의 수행 시간과 CPU 사용률을 측정하였다. 수행 시간의 측정은 메모리의 강제 할당과 해제를 수행하는 "fillmem" 프로그램으로 메모리의 양을 단계적으로 늘리면서 수행 시간을 측정하였다. "fillmem" 프로그램의 인자 값으로 메모리의 크기를 입력하여서 입력한 크기만큼의 메모리를 할당하는 실험을 하였다. 인자 값의 단위는 Mbyte이며, 메모리의 크기를 처음 10Mbyte부터 450Mbyte까지 늘리면서 전체 "fillmem" 과정이 끝나는 시점까지의 시간을 측정하여 그래프로 나타내었다.

압축 메모리 영역의 크기는 8Mbyte부터 128Mbyte까지 각각 같은 방법으로 측정을 하였다. 먼저 8Mbyte일 때의 실험결과는 (그림 5)과 같다. 이 실험 결과에서는 압축 메모리 기법을 적용한 시스템이 적용하지 않은 시스템보다 수행 시간이 더욱 좋지 않게 나타나고 있다. 그래프의 Y축이 수행 시간을 나타내는데 수치가 높을수록 성능이 좋지 않음을 나타낸다. 그리고 X축은 메모리의 양을 나타낸다. 여기서 메모리의 양이 10Mbyte에서 360Mbyte까지는 압축 메모리 시스템과 기존 시스템의 수행 시간이 같은 수치를 보이고 있다. 물리 메모리의 전체 크기가 384Mbyte이고 일부 데몬들에 의해 사용되는 물리 메모리의 양이 약 25Mbyte 가량 소모되어 가용 물리 메모리의 전체 크기는 360Mbyte 가량 된다. 그래서 물리 메모리 보다 더 많은 양의 메모리를 할당하는 370Mbyte 시점에서 압축 메모리 시스템과 기존 시스템과의 수행 시간이 달라지는 것을 알 수가 있다.

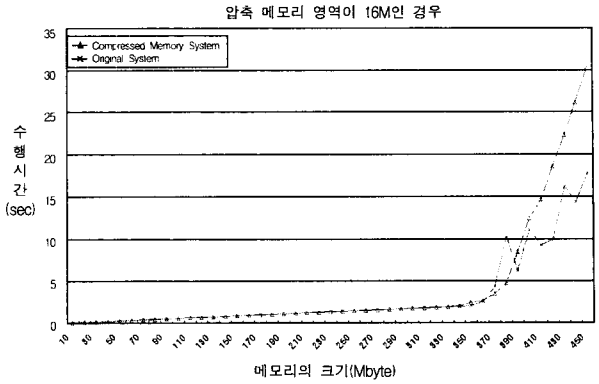
그리고 압축 메모리 시스템의 성능이 떨어지는 이유는 압축 메모리 영역이 페이지 폴트 되어 스왑 아웃되는 페이지들의 양을 전부 수용하지 못하여 압축 메모리 영역에서 다시 스왑 디바이스로 이동하는 페이지와 폐기되는 페이지들이 늘어나면서 "fillmem" 프로그램의 수행 속도가 저하되는 현상을 보이고 있다.



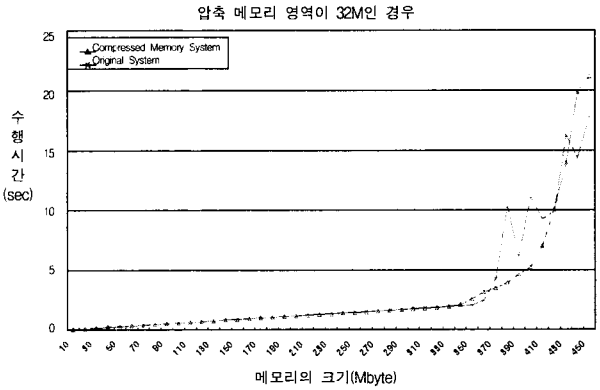
(그림 5) 압축 메모리 영역의 크기가 8M인 경우의 수행 시간 측정

(그림 5)의 기존 시스템의 그래프의 변화를 보면 특이한 현상이 보이고 있는데 390, 410, 430Mbyte에서 수행 속도가 큰 폭으로 늘어났다가 줄어드는 현상을 보이고 있는데 이것은 스왑 디바이스로의 접근이 많아지는 시점에서 스왑핑되거나 폐기되는 페이지의 양이 더욱더 많아지면서 주기억장치의 메모리 공간을 늘리는 작업이 활발히 일어나는 것에서 수행 시간이 늘어나는 현상이다.

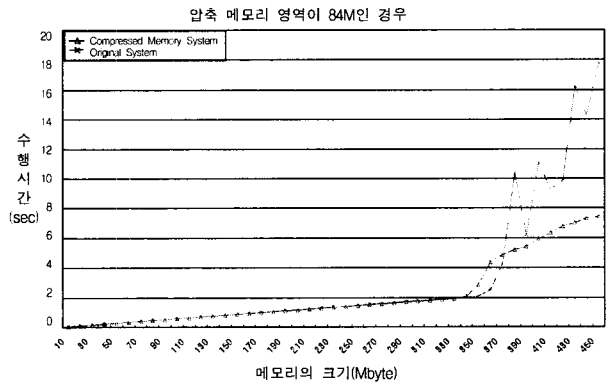
다음 실험으로 압축 메모리 영역의 크기를 16Mbyte와 32Mbyte로 수정하고 시스템의 재부팅 후 "fillmem" 실험을 같은 환경에서 같은 조건으로 수행한다. 실험 결과는 (그림 6)와 (그림 7)과 같다.



(그림 6) 압축 메모리 영역의 크기가 16M인 경우의 수행 시간 측정



(그림 7) 압축 메모리 영역의 크기가 32M인 경우의 시간 측정



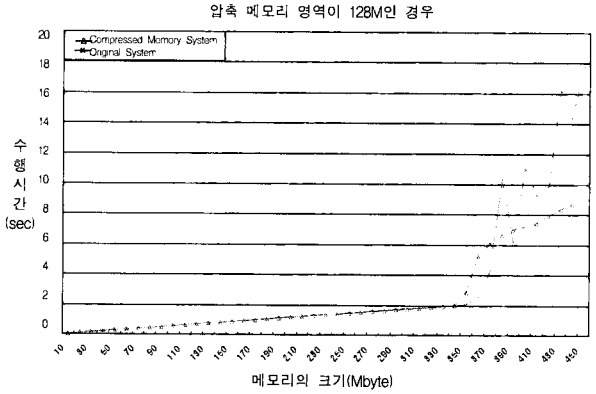
(그림 8) 압축 메모리 영역의 크기가 64M인 경우 수행 시간 측정

(그림 8)은 압축 메모리 영역을 64Mbyte로 늘려서 측정한 실험 결과이다. 실험 방법은 앞에서 실시한 방법과 동일하게 수행하였으며 실험 결과 기존 시스템 보다 수행 시간이 좋게 나타나는 것을 볼 수가 있다.

압축 메모리의 수행 시간이 최적인 압축 메모리의 크기는 각 시스템의 물리 메모리의 양과 시스템에서 동작중인 프로세스에 의해 사용되는 메모리의 양에 따라 달라질 수 있다. 데몬 프로세스들이 사용하는 메모리의 양에 따라서 달라질 수가 있다. 따라서 현재 시스템의 메모리 상태에 따

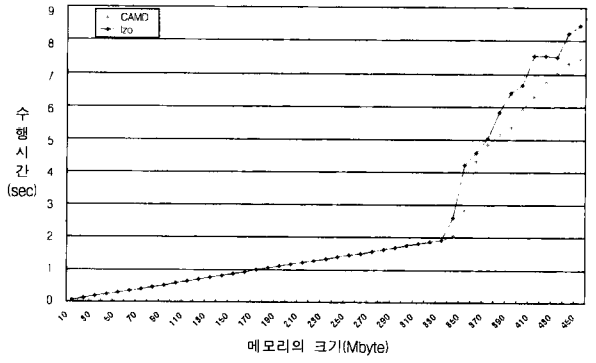
라 유동적으로 압축 메모리의 크기를 변경할 수 있는 기능이 추가된다면 좀더 최적의 압축 메모리 관리 기법이 될 수 있을 것이다.

이 실험에서는 전체 물리 메모리의 1/6의 메모리를 압축 메모리 영역으로 할당할 경우에 최적의 수행 시간을 보이고 있다. (그림 9)를 보게 되면 압축 메모리의 크기가 128Mbyte인 경우의 실험 결과를 보여주고 있는데 최대 수행 시간이 8초를 넘는 것을 볼 수가 있다. (그림 8)와 비교해 보면 (그림 8)의 최대 수행 시간이 8초를 넘지 않는데 비해 (그림 9)의 그래프에서는 최대 수행 시간이 8초를 넘는 것을 확인할 수 있다. 따라서 128Mbyte의 압축 메모리 공간이 64Mbyte의 메모리 공간보다 수행이 더욱 좋지 않게 나타나는 것을 볼 수 있다.



(그림 9) 압축 메모리 영역의 크기가 128M인 경우의 수행 시간 측정

(그림 10)은 LZ 계열의 압축 알고리즘인 LZO 압축 알고리즘과 본 논문에서 제시하는 CAMD 압축 알고리즘과의 수행 속도를 비교한 그래프이다. CAMD알고리즘이 LZO 알고리즘에 비하여 성능이 우수함을 알 수 있다.



(그림 10) CAMD알고리즘과 LZO알고리즘과의 성능 비교

이상의 실험에서 압축 메모리 영역의 크기에 따라 수행 시간의 성능이 다르게 나타나는 것을 확인할 수 있다. 압축 메모리 영역의 크기가 8, 16, 32Mbyte인 경우는 압축된 페이지를 저장할 수 있는 공간이 충분하지 못하기 때문에 시스템의 수행 시간이 압축 메모리를 사용하지 않은 시스템보다 좋지 못하게 나타난다. 그리고 압축 영역이 64Mbyte일 때 가장 수행 시간이 좋게 나타나며 64Mbyte 보다 큰 경우는 압축 메모리 공간을 할당하기 위해 주기억장치의 메모리의 공간을 많이 소모하여 주기억장치에 할당할 수 있는 공간 부족으로 64Mbyte인 경우 보다 수행 시간이 좋지 못하게 나타나는 것을 확인할 수 있었다.

수행 시간 측정 실험으로 주기억장치의 여유 메모리 공간보다 더 많은 메모리의 할당을 요구할 경우에 효율성을 얻을 수 있다는 것을 알 수 있는데, 이것은 위의 그래프들에서 공통적으로 주기억장치의 크기보다 크게 메모리를 할당하고 해제하는 시점에서, 시스템에 스왑이 발생하고, 스왑이 더욱더 많이 발생할수록 시스템의 수행 시간을 많이 소모한다는 것을 알 수 있었다. 이렇게 가상 메모리 압축 시스템을 적용할 경우 시스템의 주기억장치와 스왑 디바이스와의 접근에서 발생하는 지연 시간을 감소시켜 수행 시간의 효율성을 얻을 수 있었다.

6. 결 론

본 논문에서는 임베디드 시스템에서 스왑 아웃되는 페이지들을 압축하여 주기억장치 내의 압축 캐시 영역에 저장하는 메모리 관리 기법인 가상 메모리 압축 시스템을 설계함으로써 스왑 아웃되는 시간을 줄이고 스왑 영역의 사용량을 줄임으로써 전체적인 시스템의 메모리 공간을 절약하여 시스템 성능 향상을 높일 수 있는 시스템을 설계하였다.

본 논문에서 제시하는 가상 메모리 압축 알고리즘은 주기억장치를 압축되지 않은 영역과 압축된 캐시 영역으로 나누어서 페이지 폴트가 발생했을 때 느린 스왑 디바이스로 이동하는 페이지들을 압축된 메모리 영역에 저장한다. 그리고 이후에 이 페이지들이 프로세스에 의해 필요로 될 때 느린 스왑 디바이스가 아닌 압축된 메모리 영역에서 데이터를 가지고 옴으로써 보조기억장치로의 접근 횟수를 줄일 수 있도록 설계하였으며 압축된 스왑 영역을 사용함으로써 보조기억장치의 저장 공간을 늘리는 효과를 낼 수 있다.

본 논문의 실험에서 가상 메모리 압축 알고리즘을 적용한 시스템과 적용하지 않은 시스템과의 성능 비교에서 주기억장치의 가용 공간보다 더 많은 메모리를 요구하는 작업에서 수행 시간의 효율성을 얻을 수 있었으며 이것은 시스템에서 스와핑이 발생할 때의 지연시간을 최소화하여 전체적인 수행 시간의 향상을 얻을 수 있었다. 본 논문에서 제시한 압축 알고리즘은 전체 비트가 0과 1로 구성된 데이터를 적은 비트로 압축하고 사전에 등록된 데이터와 사전에 등록되지 않은 데이터들을 제어 비트에 의해 처리되도록 구현한다. 그리고 사전의 운영은 해쉬를 사용하여 구현하였다.

앞으로의 개선 사항으로는 좀더 메모리 내의 데이터에 적합하고 효율적인 압축 알고리즘의 개발과 주기억장치의 상태에 따라 유동적으로 압축 메모리 영역의 크기가 변경될 수 있는 시스템을 개발하는 것이다.

참 고 문 헌

[1] Dong Banks, Mark Stemm, "Investigating Virtual Memory Compression on Portable Architectures," 1995.

[1] Caroline Benveniste, "Cache-Memory Interfaces in Compressed Memory System," IEEE, 2001.

[2] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," Proceedings of the Usenix, 1999.

[3] Fred Douglis, "The compression Cache : Using on-line compression to extend physical memory," USENIX Proceedings, Winter 1993.

[4] S. Kwong and Y. F. Ho, "A statitical Lempel-Ziv Compression Algorithm for Personal Digital Assistant(PDA)," IEEE, 2001.

[5] Ian McDonald, "The use of a Compressed Cache in an Operating System supporting Self-Paging," September, 1999.

[6] Sumit Roy, Raj Kumar, Milos Prvulovic, "Improving System Performance with Compressed Memory," IEEE, 2001.

[7] David A Rusling, "The Linux Kernel," January, 1999.

[8] M. S. Pinho, W. A Finamore, "Using arithmetic code to improve performance of Lempel-Ziv encoders," Electronic Letters, Aug., 2000.

[9] Ross N. Williams, "An Extremely Fast Ziv-Lempel Data Compression Algorithm," Data Compression conference, 1991.

[10] Maurice J. Bach, "The Design of The Unix Operating System." Prentice-Hall International Editions.

[11] Khalid Sayood, "Introduction to Data Compression Second Edition," Morgan Kaufmann, 2000.

[12] Data Compression Reference Center http://www.rasip.fer.hr/research/compress/algorithms/index.htm.

[13] Ross N. Williams, "An Extremely Fast Ziv-Lempel Data Compression Algorithm," Data Compression conference, 1991.

[14] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel", O'relly.

[15] Abraham Silberschatz, Greg Gane, Petter Baer Galvin "Operating System Concept," 5th Edition.

정 진 우

e-mail : jjw0323@dreamwiz.com

2001년 동의대학교 컴퓨터공학과(학사)

2001년~현재 동의대학교 컴퓨터공학과 석사과정

관심분야 : 운영체제, 소프트웨어 공학, 객체지향 디자인 패턴

장 승 주

e-mail : sjjang@dongeui.ac.kr

1985년 부산대학교 계산통계학과(전산학) 학사

1991년 부산대학교 계산통계학과(전산학) 석사

1996년 부산대학교 컴퓨터공학과 박사

1987년~1996년 한국전자통신연구원 시스템 S/W연구실

1993년~1996년 부산대학교 시간강사

2000년~2002년 University of Missouri at Kansas City, visiting professor

1996년~현재 동의대학교 컴퓨터공학과 부교수

관심분야 : 운영체제, 분산시스템, Active Network, 시스템 보안