임베디드 시스템을 위한 효율적인 메모리 압축 기법

우장복° 최병창 서효중.
가톨릭대학교 컴퓨터공학과
{sofe4u°, hellocbc, hjsuh}@catholic.ac.kr

An Effective Memory Compression Scheme for Embedded System

JangBok Woo° ByeongChang Choi Hyo-Joong Suh
Dept. of Computer Science and Engineering, The Catholic University of Korea

요 약

최근 임베디스 시스템의 성능이 향상됨에 따라, 임베디드 시스템을 구성하는 CPU와 주변 장치들의 성능 격차를 해소하는 문제가 점차 중요해지고 있다. 그 중에서 시스템의 성능에 가장 큰 영향을 미치는 것이 CPU와 메모리간의 통신이다. 고성능 컴퓨터 시스템에서는 그동안 CPU와 메모리간의 성능 격차를 줄이기 위한 여러 가지 연구들이 활발하게 진행되었는데, 여러 가지 연구들 중에서 메모리를 압축하여 메모리의 기억공간을 효율적으로 확장하는 방법이 효과적으로 사용되고 있다. 임베디드 시스템에서도 이러한 기법을 적절하게 적용하면 메모리를 압축함으로써 동일 공간에 보다 많은 데이터를 저장할 수 있고, 버스를 이용하여 데이터를 전송할 때, 보다 많은 정보를 전송할 수 있게 된다. 또한, CPU와 메모리 간의 전송되는 정보의 크기를 줄일 수 있으므로 임베디드 시스템에서 전력소모의 대부분을 차지하고 있는 CPU와 메모리 간의 전력소모를 크게 줄일 수 있는 장점이 있다. 본 논문에서는 빈발 패턴 압축 기법을 적절하게 변형하여 임베디드 시스템을 위한 효율적인 메모리 압축 기법을 제시하고자 한다.

1. 서 론

최근 임베디드 시스템에 대한 관심이 높아지고, 해당 시스템의 성능이 증가함에 따라, 임베디드 시스템에서도 메모리를 관리하는 것이 중요한 이슈로 떠오르고 있다. 과거에는 메모리가 시스템이 필요로 하는 저장 공간을 충분히 제공했으나, CPU의 성능 향상으로 메모리 요구량이 매년 50~100%씩 증가하여 보다 많은 데이터를 저장하기 위한 충분한 대역폭과 신호 지연시간이 낮은 메모리 시스템이 요구되고 있다. 또한, CPU와 메모리의 성능 격차로 인해서 CPU가 메모리에서 데이터가 처리되는 것을 기다리는 일이 발생하게 되어 시스템의 성능이 저하되는 현상이 빈번하게 발생하고 있다[1]. 이러한 문제를 해결하기 위해서 입출력 버퍼의 개수를 늘리거나, 버스의 대역폭, 메모리의 크기, 캐쉬의 크기를 늘리는 방법 등이 제시되었는데, 해당 기법들은 하드웨어적으로 구조를 변경해야 하는 문제점이 있다. 이렇게 구조를 변경하지 않고도 시스템의 성능을 향상시키는 방법으로 메모리 압축 기법이 있다. 메모리 압축 기법은 메모리에 압축된 데이터를 저장함으로써 저장 공간과 대역폭의 증가 효과를 얻을 수 있으며, 전력소모도 줄어들게 된다[2]. 메모리 압축 기법의 종류에는 공통으로 등장하는 패턴(pattern)을 사전(dictionary)에 저장시켜놓고 사용하는 사전 기반 기법(Dictionary-Based Compression)과 데이터들의 일정 부분은 똑같이 유지되고 다른 일정 부분이 계속 변하는 경우에 사용하는 차이 기반 기법(Differential-Based Compression), 그리고 데이터가 사용되는 정도에 따라 가중치를 적용하여 압축을 하는 중요성 기반 기법(Significance-Based Compression) 등이 있다[3]. 그러나 이러한 기법들은 대부분 고성능 컴퓨터 시스템에서의 시스템 성능을 개선하기 위해서 고안되었으므로, 시스템 자원이 제한적인 임베디드 시스템 환경에 그대로 적용할 수 없다. 따라서 본 논문에서는 중요성 기반 기법의 일종인 빈발 패턴 압축 기법(Frequent Pattern Compression)을 임베디드 시스템의 특성에 맞게 적절히 변형하여 임베디드 시스템을 위한 효율적인 메모리 압축 기법을 제시하고자 한다.

2. 관련 연구

여러 압축 기법들 중에서 빈발하게 등장하는 값을 압축하여 시스템의 성능을 높이는 연구는 Yang과 Gupta에 의해서 체계적으로 정리되었다[4]. Yang과 Gupta는 SPECint95 벤치마크를 사용하여 서로 구분되는 작은 개수의 값들이 메모리에 접근할 때 빈번하게 사용되는 것과, 한 바이트 당 55%이상이 '0'값인 것을 발견하였다. 이러한 점에 착안하여 L1 캐쉬의 라인 하나하나에 압축되지 않은 데이터 한 라인을 연결하거나, 압축된 데이터 두 라인을 연결하는 방법으로 적어도 처리되는 데이터의 50% 이상을 압축된 데이터가 차지함으로써 적은 값이 빈발하게 나오는 직접 사상 캐쉬(Direct Mapped Cache)에서 최적의 성능을 보이는 FVC(Frequent Value Cache)라 불리는 빈발 값 중심의 캐쉬 디자인을 제시하였다. Almeldeen과 Wood는 FVC 기법에, 사용되는 정도에 따라 가중치를 부여하는 중요성 기반 압축 기법을 적용하여 FPC(Frequent Pattern Compression)이라 불리는 압축 기법을 제안하였다[3]. 이 기법에서 한 워드는 prefix라 불리는 3bit의 값과 데이터로 구성된다. 이 3bit의 prefix로 총 8가지의 서로 다른 패턴을 나타낼 수 있으며, 아래 표 1과 같이 압축되지 않은 값을 나타내는 1개의 패턴과 압축된 값을 나타내는 7개의 패턴으로 구성된다. 이 때, 7개의 패턴은 3bit의 prefix와 데이터가 함께 압축되어 전송되게 된다.

표 1 Frequent Pattern Encoding

Prefix	Pattern Encoded
000	Zero Run
001	4-bit sign-extended
010	One byte sign-extended
011	halfword sign-extended
100	halfword padded with a zero halfword
101	Two halfwords, each a byte sign-extended
110	word consisting of repeated bytes
111	Uncompressed word

위의 표 1에 8가지의 패턴이 나타나 있지만 모든 패턴이 자주 사용되는 것은 아니며, 음수의 경우는 사용되는 경우가 매우 드물다[1]. 또한, Benini와 Macii는 [5]에서 응용 프로그램의 실행 프로파일을 분석한 결과, 임베디드 시스템에서 작동하는 프로그램들은 전체 명령어 중에서 아주 적은 개수의 특정 명령어만을 주로 사용한다는 것을 알았으며, 그 중에서 256개의 명령어를 뽑아서 압축을 하는 시스템을 제안하였다.

3. 제안하는 기법

본 논문에서는 이러한 사실들에 주목하여 위의 8가지 패턴 중에서 spec2000벤치마크를 수행하였을 때[3], 표 2와 같이 사용 비율이 높게 나오는 패턴 3개와, 압축되지 않은 값을 나타내는 패턴 1개를 포함하는 2bit의 prefix로 표현되는 변형된 FPC 기법을 제안한다.

표 2 변형된 Frequent Pattern Encoding

Prefix	Pattern Encoded
00	Zero Run
01	halfword sign-extended
10	halfword padded with a zero halfword
11	Uncompressed word

아래 그림 1(a)는 일반적인 SimpleScalar 아키텍처의 레지스터 포맷으로 16bit의 annote와 opcode, 8bit의 rs, rt, rd, ru/shamt로 구성된다.

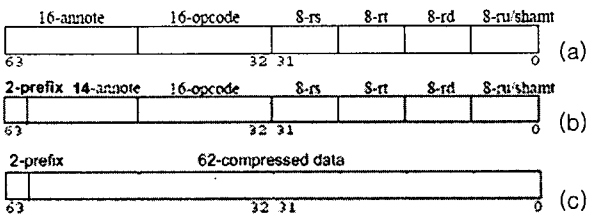

그림 1 SimpleScalar 아키텍처 레지스터 포맷

그림 1(b)와 1(c)는 본 논문에서 제안하는 메모리 압축 기법을 적용하기 위해서 SimpleScalar의 레지스터 포맷을 변형한 것으로써 내부에 2bit의 prefix를 포함한다. 그림 1(b)는 압축되지 않은 데이터를 위한 포맷으로 기존의 SimpleScalar 아키텍처 레지스터 포맷에서 16bit를 차지하던 annote를 14bit로 줄이고 나머지 2bit를 prefix에 할당하였다. 그림 1(c)는 압축된 데이터를 위한 포맷으로 2bit의 prefix와 62bit의 compressed data로 이루어진다. 본 논문에서 제안하는 기법의 시스템 구성도는 그림 2와 같으며, 이 기법에서 한 워드는 그림 1(b)나 그림 1(c)와 같이 구성된다.

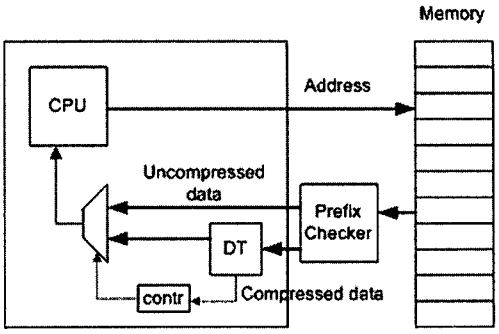

그림 2 CPU와 Memory간의 데이터 전송 구조

제안하는 시스템의 동작 과정을 살펴보면, CPU에서 메모리에 접근하여 데이터를 불러오는 경우 메모리에 저장되어 있는 값들은 압축된 상태이므로 prefix checker에서 prefix 값을 검사하여 해당 데이터의 압축 여부를 판별하게 된다. prefix의 값이 '11'인 경우 압축되지 않은 데이터 이므로 값을 CPU로 바로 넘겨주게 되며, prefix의 값이 '00'~'10'사이의 값인 경우 decompression table에서 압축 해제 과정을 거쳐서 CPU로 값을 넘겨주게 된다.

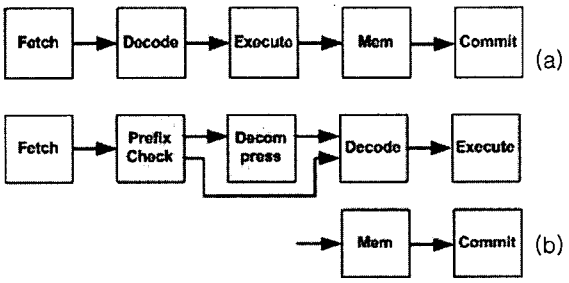

그림 3 SimpleScalar 파이프라인

위의 그림 3(a)는 메모리 압축 기법을 적용하지 않은 일반 시스템에서 Wirteback을 고려하지 않았을 때의 SimpleScalar 파이프라인을 나타낸다. Fetch, Decode, Execute, Mem, Commit의 총 5단계로 이루어져 있으며, 각 단계는 동일한 단위 시간에 수행된다[6]. 이 5단계의 과정의 수행 시간의 합이 CPU가 메모리에 접근하여 데이터를 한 번 읽어오는데 걸리는 시간이 된다. 이 시간을 T_m이라 하고, 어떤 한 응용 프로그램을 수행할 때 CPU가 메모리에 접근하는 회수가 k번이라고 가정하면, 프로그램 수행에 소요되는 총 시간 T_{org}는 식(1)과 같다

$$T_{org} = \sum_{i=1}^{k} T_{m_i} \quad (1)$$

이 때, T_{m_i}는 CPU가 메모리에 접근하여 데이터를 읽어오는 i번째 시간을 나타낸다. 그림 3(b)는 본 논문에서 제안하는 메모리 압축 기법을 적용했을 때의 SimpleScalar 파이프라인을 나타내며, 그림 3(a)에 두 단계가 추가되어 총 7단계로 이루어진다. Prefix Check

단계는 메모리에서 fetch된 데이터의 prefix 값을 확인하여 압축된 데이터가 아닌 경우 Decode단계로 진행하고, 압축된 데이터의 경우 Decompress단계로 진행한다. Decompress단계에서는 압축된 데이터의 압축을 해제한다. 이 때 걸리는 시간을 T_c, prefix 값을 확인하는데 걸리는 시간을 T_{px}라 하면, 본 논문에서 제안하는 압축 기법을 적용하여 어떤 한 응용 프로그램을 수행하는데 소요되는 총 시간 T_{var}는 식(2)와 같다.

$$T_{var} = \sum_{i=1}^{k}(T_{px_i} + T_{c_i} + T_{m_i}) \quad (2)$$

어떤 한 응용 프로그램을 SimpleScalar를 통해 시뮬레이션하여 위의 식 (1)과 (2)를 구하여 값을 비교하면 Prefix Check단계와 Decompress단계의 추가로 인해서 발생한 압축 해제 지연 시간이 클럭당 명령어 처리 시간(Instrunction Per Clock)에 미치는 영향을 알 수 있다. [3]에서 한 응용 프로그램을 수행했을 때 압축하지 않은 데이터의 비율이 평균 45%인 것을 고려하여, k가 100000이고 파이프라인의 각 단계가 수행되는 단위시간이 1IPC이며, CPU에서 필요한 데이터가 메모리에 모두 존재한다고 가정하여 식을 계산해보면,

$$T_{org} : T_{var} = 1 : 0.991 \quad (3)$$

식 (3)과 같은 결과가 나온다. 이 결과는 [3]에서 보였던 응용 프로그램을 수행했을 때 FPC 기법을 적용하지 않은 일반 시스템과 FPC 기법을 적용한 시스템의 평균 IPC의 비율인 0.995와 거의 유사한 것을 알 수 있다.

4. 성능 평가

위에서 계산을 통해서 얻은 식 (3)을 이용하여 [3]에서 얻은 결과와 비교하여 성능을 예상하여 평가한다.

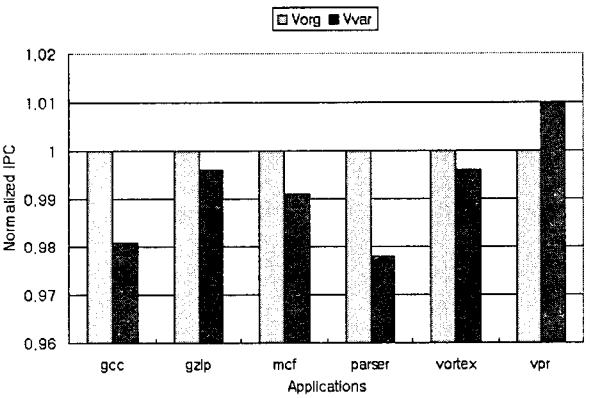

그림 4 성능 예상 비교 그래프

위의 그림 4에서는 제안하는 시스템의 압축 해제 지연 시간으로 인해서 응용 프로그램에 따라 IPC 값이 줄어드는 것을 예상할 수 있다. 아래 그림 5는 기존의 FPC 기법과 제안하는 변형된 FPC 기법의 압축률을 예상하여 비교한 그래프이다. 기존의 FPC 기법과 비교하여 압축률의 차이가 거의 없는 것을 예상할 수 있다.

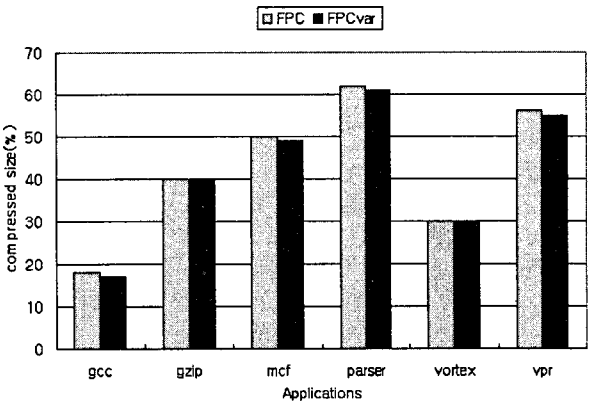

그림 5 압축률 예상 비교 그래프

제안한 기법을 위해서 SimpleScalar 아키텍처 포맷을 변환하여 적용이 진행 중이며, 차후 연구 결과를 제시할 예정이다.

5. 결론 및 향후 과제

본 논문에서는 빈발하게 등장하는 값을 압축 하는 기법에, 프로그램 내에서 사용되는 정도에 따라서 가중치를 부여하여 압축하는 기법을 적용하여 고안된 FPC 가법을 적절히 변형하여 임베디드 시스템을 위한 효율적인 메모리 압축 기법을 제안하였다. 향후 연구과제는 기존의 SimpleScalar 아키텍처를 수정하여 제안한 시스템의 구현을 완료하고, 벤치마크 프로그램 등을 이용하여 시스템의 압축률, prefix의 값에 따른 성능 비교와 에너지 소모 비교 등을 통해서 제안한 시스템의 성능을 검증하는 것이다.

6. 참고 문헌

[1] Magnus Ekman and Per Stenstrom, "A Robust Main-Memory Compression Scheme", International Symposium on computer Architecture(ISCA-32), pages 74-85, 2005

[2] J. Liu, N.R. Mahapatra, K. Sundaresan, S. Dangeti, and B.V. Venkatrao, "Memory system compression and its benefits", Proc. 15th Annual IEEE International ASIC/SOC Conference(ASIC/SOC 2002), papes 41-45, 2002.

[3] Alameldeen A. R. and Wood D. A, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches", Technical Report 1500, 2004

[4] Zhang Y., Yang J. and Gupta R. "Frequent Value Locality and Value-centric Data Cache Design", Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 150-159, 2000

[5] L. Benini, A. Macii, E. Macii, M. Poncino, "Minimizing memory access energy in embedded systems by selective instruction compression", IEEE TVLIS, vol. 10, 2002

[6] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0", Computer Architecture News 25(3), 1997.