협동 병렬 X-Match 데이타 압축 알고리즘
(The Cooperative Parallel X-Match Data Compression Algorithm)

윤 상 균 †
(Sang-Kyun Yun)

요 약 X-Match 알고리즘은 비교적 간단하여 하드웨어로 구현하는 데에 적합한 무손실 압축 알고리즘이다. X-Match 알고리즘은 사이클 당 32비트의 압축이 가능하므로 고속 압축에 적합하다. 그렇지만 버스 폭이 증가됨에 따라서 이에 맞추어서 압축 단위를 증가시킬 필요가 있게 되었다. 본 논문에서는 X-Match 알고리즘을 병렬로 수행하여 압축 속도를 2배 향상시키고 X-Match 알고리즘 거의 비슷한 압축률을 제공하는 협동 병렬 X-Match 알고리즘, 즉 X-MatchCP 알고리즘을 제안한다. 기존의 병렬 X-Match 알고리즘이 X-Match 알고리즘을 병렬로 수행할 때에 각자의 사전을 검색하는 데 비해서 X-MatchCP 알고리즘에서는 X-Match 알고리즘이 병렬로 수행되지만 전체 사전을 검색하여 매칭빈도를 높이도록 하였고 run-length 부호화도 두 워드에 대해서 한꺼번에 하는 방식으로 서로 협동하면서 동작한다. 메모리 데이타와 파일 자료를 사용한 시뮬레이션 결과 X-MatchCP 알고리즘은 같은 사전 크기의 X-Match 알고리즘과 거의 비슷한 압축률을 보였다. 그리고 X-MatchCP 알고리즘의 하드웨어 구현을 위한 전체적인 구조 설계를 Verilog 언어를 사용하여 수행하였다.
키워드 : X-Match 알고리즘, 병렬 압축 알고리즘, 압축 하드웨어

Abstract X-Match algorithm is a lossless compression algorithm suitable for hardware implementation owing to its simplicity. It can compress 32 bits per clock cycle and is suitable for real time compression. However, as the bus width increases 64-bit, the compression unit also need to increase. This paper proposes the cooperative parallel X-Match (X-MatchCP) algorithm, which improves the compression speed by performing the two X-Match algorithms in parallel. It searches the all dictionary for two words, combines the compression codes of two words generated by parallel X-Match compression and outputs the combined code while the previous parallel X-Match algorithm searches an individual dictionary. The compression ratio in X-MatchCP is almost the same as in X-Match. X-MatchCP algorithm is described and simulated by Verilog hardware description language.
Key words : X-Match algorithm, parallel compression algorithm, compression hardware

1. 서 론

프로세서 속도가 빨라지는 것에 비례하여 디스크의 접근 속도가 향상되지 못함에 따라서 서버급 컴퓨터들은 페이지 폴트율을 감소시켜서 디스크 접근을 줄이기 위해서 프로세서 속도에 비례하여 메모리의 크기도 함께 증가시키는 경향이 있다. 그 결과 메모리의 가격이 하락하는 최근의 추세에도 불구하고 메모리의 비용이 여전히 컴퓨터 전체 비용의 주된 요소가 되고 있다. 이러한 비용 부담을 줄이는 한가지 방법은 메모리에 데이타를 압축하여 저장하는 것이다.

메모리 데이타 중에서 즉시 사용되지 않는 부분을 압축하여 메모리에 저장하면 메모리에 더 많은 데이타를 저장할 수 있기 때문에 메모리의 유효 크기를 증가시킬 수 있게 된다. 이러한 압축 메모리의 사용은 동일한 기억용량을 적은 비용으로 제공할 수 있으며, 같은 비용으로 메모리의 기억용량을 증가시킬 수 있어서 메모리 크기 증가에 따른 성능 향상을 가져올 수 있다. 그렇지만 성능 향상을 위해서는 압축 및 복원에 소요되는 시간이 디스크 접근 속도에 비해서 훨씬 작아야 한다.

메모리 데이타에 대한 압축에 대해서 여러 연구자들이 연구를 수행하였다. Douglis[1]는 가상메모리 시스템에서 스왑 아웃되는 페이지들 중 일부를 디스크 대신에 메모리에 압축하여 저장함으로써 가상 메모리 시스템의 성능 향상을 시도하였다. 이 방법에서 압축 및 복원은

· 이 논문은 2002년도 연세대학교 학술연구비의 지원에 의하여 이루어진 것임
† 정 회 원 : 연세대학교 문리대학 정보기술학부 교수
skyun@dragon.yonsei.ac.kr
논문접수 : 2003년 5월 9일
심사완료 : 2003년 7월 31일

운영체제에 의해서 LZRW1 알고리즘[2]을 사용하여 소프트웨어로 구현되었는데 압축을 수행하는 데 소요되는 오버헤드 때문에 실제의 응용 프로그램에 대해서 그다지 성능 향상을 가져오지 못하였다. William과 Kaplan [3]은 LZ알고리즘보다 훨씬 간단하여 소프트웨어로 수행되는 압축 속도를 대폭 향상시키고 메모리 데이타 특성을 활용하여 어느 정도의 압축률을 제공하는 WK압축방법을 제시하였으며 이 압축 방법을 가상메모리 시스템에서의 압축 캐쉬에 적용하였다. 그리고 Roy 등[4]도 압축 메모리를 리눅스 시스템에서 디바이스 드라이버 형태로 소프트웨어로 구현하였으며 프로세서의 속도의 비약적인 발전에 힘입어서 어느 정도의 성능 향상을 가져왔다.

그렇지만 시스템의 성능 향상을 위해서는 메모리 데이타의 압축과 복원은 메모리 동작 속도로 이루어져야 하는 데 소프트웨어적으로는 이 속도로 동작하도록 구현하기가 어렵다. 이 때문에 압축 메모리를 구성하기 위해서는 하드웨어로 구현된 압축/복원기가 필요하며 메모리 동작 속도로 동작하는 하드웨어 압축기를 설계하기 위해서는 하드웨어로 구현하기에 적합하며 고속 동작이 가능한 비교적 간단한 압축 알고리즘이 필요하다.

압축 메모리를 위한 압축 하드웨어로는 X-Match와 MXT가 대표적이다. MXT(Memory Expansion Technology)[5]는 IBM이 압축메모리를 이용하여 메모리 유효 크기를 증가시키는 데 사용된 기술이다. MXT는 바이트 단위로 압축하는 LZ알고리즘을 압축속도를 높이기 위해서 4 바이트에 대해서 병렬로 압축을 수행하지만 사전은 공유하는 압축 알고리즘[6]을 사용하여 압축, 복원기를 구현하였다.

X-Match[7-9]는 바이트 단위 대신에 32비트 워드 단위로 압축을 수행하여 압축 속도를 대폭 향상시켰다. 비교 단위가 커짐에 따라서 매칭 빈도가 급격하게 줄어들게 되지만 부분 매칭을 허용하여 이 문제를 해결하였다. X-Match 압축 알고리즘은 비교적 간단하여 하드웨어로 구현하기에 적합하며 매 사이클마다 32비트 워드에 대한 압축을 수행할 수 있으므로 고속 압축이 가능하다. 이 압축 알고리즘은 여러 종류의 FPGA를 사용하여 구현되었다[10-12]. X-Match 알고리즘은 원래 압축 메모리용으로 개발되었지만 다른 저장장치나 전송장치를 위한 실시간 압축을 위해서 사용할 수 있다. 그리고 반복되는 워드들에 대해서 run-length 부호화를 하는 기능을 X-Match 알고리즘에 추가한 X-MatchPRO 알고리즘[11,12]도 제시되었다.

앞에서 언급했듯이 X-Match 알고리즘의 가장 큰 장점은 고속 압축 동작이 가능하다는 것이다. 다른 하드웨어 압축기는 1바이트를 압축하는 데 1~2 사이클이 소요되지만 X-Match알고리즘을 사용한 하드웨어 압축기는 1사이클에 32비트, 즉 4 바이트를 압축할 수 있다[11]. 그렇지만 컴퓨터 시스템의 버스 폭이 64비트로 증가됨에 따라서 한 번에 64비트 씩 압축할 수 있는 하드웨어 압축기를 필요로 한다. 이러한 요구 조건을 지원하기 위해서 두 개의 워드에 대해서 X-Match 알고리즘을 병렬로 수행하여 압축 속도를 2배로 향상시키기 위한 기본 구조를 제시하고 이 구조의 동작을 SystemC로 모델링 한 연구가 수행되었다[13]. 그렇지만 이 연구는 사전의 전체 크기가 같은 경우에 압축에 참조되는 사전의 크기가 절반이 되어서 기존의 X-Match 알고리즘에 비해서 압축률이 저하될 수 있는 단점이 있다.

본 논문에서는 X-Match 알고리즘을 두 개의 워드에 대해서 병렬로 X-Match 알고리즘에 기반한 압축을 수행하여 한 번에 64비트를 압축할 수 있도록 하지만 압축률은 거의 차이가 없도록 하는 병렬 X-Match 알고리즘인 X-MatchCP 알고리즘을 제안한다. 이 방법에는 반복되는 워드에 대한 run-length 부호화 기능과 추가적인 압축률 개선 방법도 함께 제시된다.

본 논문의 구성은 다음과 같다. 2절에서는 관련 연구로서 X-Match 알고리즘에 대해서 소개하고 3절에서는 본 논문에서 제안하는 병렬 X-Match 알고리즘인 X-MatchCP를 소개한다. 4절에서는 기존의 X-Match 알고리즘과 제안된 X-MatchCP 알고리즘의 압축률을 비교하고 X-MatchCP 알고리즘에 대한 평가를 한다. 그리고 마지막 절에서 결론 및 향후 과제를 기술한다.

2. X-Match 압축 알고리즘

이 절에서는 본 논문에서 병렬화 대상이 되는 X-Match 알고리즘에 대해서 소개한다. X-Match 알고리즘[7,8]은 블록 내의 이전에 처리된 데이타들로 구성된 사전을 사용하며 현재 압축할 데이타와 사전에 저장된 각 데이타와의 비교를 통하여 압축을 수행한다. 이러한 비교는 CAM(content addressable memory)을 사용하여 하드웨어적으로 동시에 수행될 수 있다. X-Match 알고리즘에서의 압축은 고속 압축을 위해서 4바이트 워드 단위로 이루어지며 사전의 각 엔트리도 4바이트 크기이다. 그렇지만 비교 단위가 증가함에 따라서 사전의 엔트리와 매칭이 이루어지는 빈도는 줄어들게 된다.

X-Match 알고리즘의 대표적인 특징은 4바이트 단위의 비교로 인한 매칭 빈도의 감소를 보완하기 위해서 부분적인 매칭을 압축에 이용하는 것이다. 현재의 워드를 사전의 각 엔트리와 바이트 단위로 비교하여 가장 많은 수의 바이트가 매칭이 되는 엔트리를 찾는다. 4바이트 모두 매칭 되지 않더라도 2바이트 이상이 매칭 되면 4바이트 보다 작게 나타낼 수 있으므로 압축을 수행

하고 그렇지 않으면 압축을 수행하지 않고 원래의 워드로 나타낸다. 4바이트 전부 매칭되는 것을 부분 매칭과 구분하기 위해서 완전 매칭이라고 부른다. 단순히 매칭이라 함은 부분 매칭과 완전 매칭을 모두 포함한다.

압축을 하지 않을 때에는 "무압축"을 나타내는 1과 원래의 4바이트 워드로 나타낸다. 적어도 2바이트 이상이 일치할 때에는 다음과 같은 4개의 필드로 압축하여 나타낸다.

- 매칭 플래그: "압축"을 나타내는 0을 사용한다.
- 주소: 매칭된 사전 엔트리의 주소를 나타낸다.
- 매칭 코드: 4 바이트 중에서 어떤 바이트가 매칭되었는지를 나타낸다. 압축할 워드가 사전의 엔트리와 매칭되는 바이트 위치는 4비트 매칭 패턴 P로 나타낼 수 있다. 예를 들어서 P가 '1100'이면 상위 2바이트가 매칭이 됨을 나타낸다. 총 16개의 패턴이 존재하는 데 이 중에서 '1'이 2개 이상인 11개 패턴에 대해서 압축이 수행된다. 효율적인 압축을 위해서 11개의 매칭 패턴은 다양한 자료로부터 얻어진 빈도에 근거하여 Huffman 코드로 부호화한 매칭 코드로 나타낸다. H(P)는 매칭 패턴 P에 대한 Huffman 코드를 나타낸다.
- 매칭되지 않은 바이트: 압축할 워드 W에 대해서 매칭 패턴 P의 '0'인 부분에 대응되는 바이트들이 사전을 참조하여 얻을 수 없으므로 압축할 때에 함께 나타낸다. B(W,P)는 이러한 매칭되지 않은 바이트들을 나타낸다.

위와 같이 현재 워드에 대한 압축을 수행함과 동시에 현재 워드를 사전의 맨 앞(0번지)에 저장하고 사전의 기존 값들은 하나씩 뒤로 저장하여 가장 최근에 압축이 수행된 워드들이 뒤이은 워드들의 압축에 참조될 수 있도록 한다. 사전의 이러한 동작은 쉬프트 레지스터를 사용하여 하드웨어로 구현할 수 있으며 사전의 외부 입력은 사전의 0번지에만 저장되므로 X-Match 압축기에서 사용하는 CAM을 단순하게 구성할 수 있다. 현재 워드가 사전의 맨 앞에 저장되는 대신에 마지막 주소에 저장되었던 워드는 버려진다. 현재 워드가 사전의 한 엔트리와 완전히 매칭 된다면 사전의 맨 앞부터 이 엔트리까지만 쉬프트시키고 매칭된 엔트리 이후의 워드들은 그대로 유지시켜서 매칭된 엔트리가 사전에 중복하여 저장되는 것을 방지한다. 이러한 사전 검색을 통한 워드의 압축과 사전의 갱신 과정은 입력 블록의 모든 워드에 대해서 압축이 수행될 때까지 반복된다.

X-Match 알고리즘은 그림 1과 같은 4단계의 회로로 구현할 수 있다. 1단계에서 사전을 검색하여 각 엔트리의 매칭 패턴을 얻고 2단계에서 가장 많은 바이트가 매칭된 엔트리와 주소와 매칭 패턴을 찾는다. 그리고 현재 워드를 사전에 저장할 수 있도록 하는 제어신호를 공급한다. 3단계에서 1, 2단계의 결과를 사용하여 앞에서 설명한 형식으로 현재 워드에 대한 압축 코드를 생성한다. 마지막 단계에서는 생성된 압축 코드가 일정한 크기 이상이 되면 일정한 크기를 외부로 출력하고 남은 코드를 다음에 생성할 코드와 합칠 수 있도록 한다. 각 단계의 수행에 1 클록 사이클이 소요된다고 할 때에 파이프라이닝을 이용하여 X-Match 알고리즘을 구현하면 매 사이클마다 압축 코드를 생성할 수 있기 때문에 고속 압축이 가능하게 된다.

그리고 X-MatchPRO 알고리즘[11,12]은 X-Match 알고리즘에 반복되는 워드에 대한 run-length 부호화 기능을 추가한 것이다. 가장 최근에 읽혀진 데이타가 사전의 0번지에 저장이 되므로 반복되는 워드가 있는 경우에는 0번지에서의 매칭이 반복되게 된다. 이러한 성질을 이용하여 반복되는 워드에 대해서 run-length 부호화를 할 수 있다. X-MatchPRO에서는 사전의 마지막 주소 A_{last}를 run-length 부호화를 나타내기 위해서 사용하며 run-length 부호화 결과는 압축 플래그 '0', 주소 A_{last}, 반복 길이의 세필드로 나타낸다. 이러한 run-length 부호화 코드 생성은 그림 1의 마지막 단계에서 이루어진다. 이 단계에서는 run-length 모드가 진행되는 동안에는 3단계에서 생성된 코드를 무시하며 run-length 모드가 끝날 때에 run-length 부호화에 의한 코드를 생성하여 출력을 하게 된다.

3. 병렬 X-Match 알고리즘

X-Match 알고리즘은 그림 2(a)와 같이 32비트 워드 단위로 압축을 수행한다. 그렇지만 컴퓨터 시스템의 버스 폭이 64비트로 증가됨에 따라서 버스 전송 속도에 맞추어서 압축을 수행하려면 한 번에 64비트 씩 압축할 수 있어야 한다. 이러한 압축 대역폭의 확장은 그림2(b)와 같이 2개의 X-Match 압축기를 사용하여 두 개의 32비트 워드를 병렬로 압축을 하도록 하여 구현할 수 있다. 이와 같은 방법은 [13]에서 시도되었다.

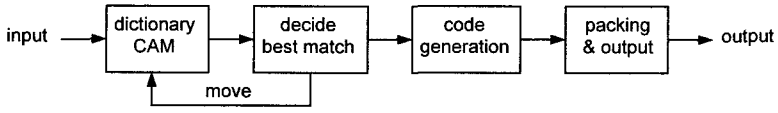

그림 1 X-Match 알고리즘의 구현의 4단계

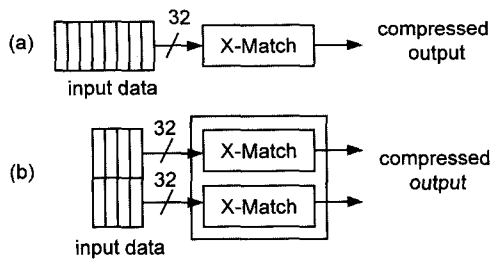

그림 2 X-Match 압축과 병렬 X-Match 압축

X-Match 알고리즘을 하드웨어로 구현할 때에 사전은 비교적 복잡한 CAM으로 구현되므로 하드웨어 복잡도 때문에 사전의 크기에 제약이 있게 된다. 그림 2의 두 가지 구현에서 압축기의 전체 사전의 크기가 서로 같은 경우를 비교하면 그림 2(b)의 병렬 압축기 내에 있는 두 개의 X-Match 압축기의 사전 크기는 그림 2(a)의 X-Match 압축기의 반이 되기 때문에 병렬 압축기의 압축 속도는 2배가 될 지라도 압축률은 대부분의 경우에 나빠지게 된다. 그리고 이러한 병렬 압축기에서 압축되는 두 출력은 압축 코드의 크기가 일정한 크기 이상이 될 때마다 서로 교차하여 출력을 하는 데 병렬로 수행하는 두 X-Match 압축기의 압축률에 차이가 있는 경우를 위해서 X-Match출력에 FIFO버퍼를 두어야 하며 압축된 데이타의 순서가 원래 입력의 데이타 순서와 달라지는 문제점이 존재한다.

그러므로 압축기의 전체 사전의 크기가 같더라도 압축률은 비슷하게 유지할 수 있으며 두 32비트 워드 스트림의 압축률 차이로 인한 문제를 해결할 수 있도록 하는 방법이 필요하다. 이 절에서 이러한 조건을 만족시키는 병렬 X-Match 알고리즘인 X-MatchCP와 이 알고리즘의 구현 방법을 제시하고자 한다.

3.1 X-MatchCP 알고리즘

본 논문에서 제안되는 알고리즘은 그림 2(b)의 병렬 X-Match 압축기에서 입력 워드에 대한 사전 검색을 자기가 속한 사전 뿐 만 아니라 전체 사전을 대상으로 하도록 하여 매칭 빈도를 높이고 두 워드에 대해서 병렬로 X-Match 압축을 수행하여 얻은 압축 코드를 결합을 한 후에 마지막 단계로 보내어 출력하도록 함으로써 병렬로 수행하는 두 블록의 압축률 차이 때문에 필요했던 FIFO 버퍼가 더 이상 필요하지 않도록 수정하였다. 제안되는 알고리즘은 입력된 두 워드에 대해서 X-Match 압축 알고리즘을 병렬로 수행하지만 사전 검색을 할 때 사전을 공유하여 전체 사전에 대해서 두 워드에 대한 검색을 수행하며 병렬로 생성된 각 워드의 압축 코드를 결합하여 내보내는 방식으로 서로 협동하여 동작한다. 이 때문에 제안되는 알고리즘을 협동병렬(cooperative parallel) X-Match 알고리즘이라고 부르며 X-MatchCP 알고리즘이라고 나타낸다. 그림 3은 X-MatchCP 압축 알고리즘이 수행되는 전체 구성도를 나타낸다.

X-MatchCP 알고리즘의 기본적인 수행 과정은 다음과 같다.

1. 압축할 블록으로부터 한 번에 두 워드를 입력받아서 첫 번째 워드를 W0에, 두 번째 워드를 W1에 저장한다.

2. 입력받은 두 워드에 대해서 매칭 가능성을 높이기 위해서 전체 사전을 대상으로 동시에 검색한다.

3. 사전 검색 결과 첫째 워드 W0와 둘째 워드 W1에 대해서 가장 많은 바이트가 매칭된 사전 엔트리의 주소와 매칭 패턴을 각각 A0, P0와 A1, P1에 저장한다.

4. 첫째 워드의 검색 결과가 매칭이면 [0, A0, H(T0), B(W0,T0)]를 압축코드로 생성하고 그렇지 않으면 [1,W0]를 압축코드로 생성한다. 마찬가지로 둘째 워드의 검색 결과가 매칭이면 [0, A1, H(T1), B(W1,T1)]을, miss이면 [1,W1]을 압축코드로 생성한다.

5. 두 워드에 대한 압축코드를 서로 결합한다.

3.2 사전 갱신

X-MatchCP 알고리즘에서는 두 워드에 대한 사전 검

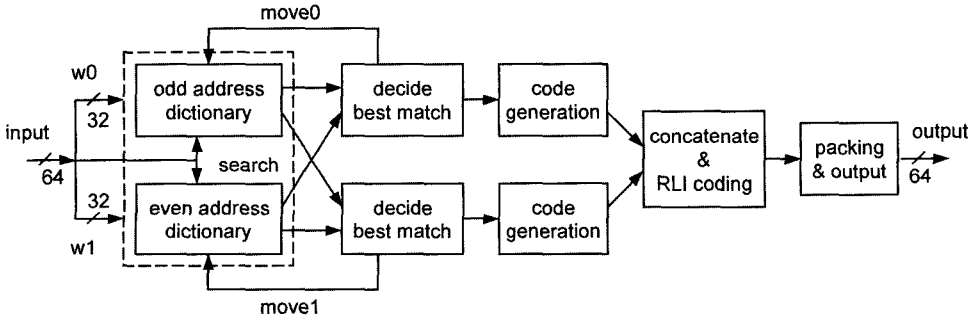

그림 3 X-MatchCP 압축 알고리즘의 구성도

색은 전체 사전을 대상으로 수행하지만 현재의 두 워드를 사전에 저장할 때에는 전체 사전을 짝수번지 사전과 홀수번지 사전으로 둘로 나누어서 첫째 워드는 홀수번지의 사전에, 둘째 워드는 짝수번지의 사전에 동시에 저장한다. 둘째 워드를 짝수 번지 사전에 저장하는 이유는 둘째 워드가 가장 최근에 사전에 입력되는 워드이므로 사전의 0번지에 저장되어야 하기 때문이다. 사전 내의 갱신도 짝수번지 사전과 홀수번지 사전이 동시에 독립적으로 이루어지며 X-Match 알고리즘에서와 같이 사전에 있는 기존의 엔트리는 자신이 속한 사전의 다음 위치로 이동한다.

그렇지만 이와 같이 홀수번지 사전과 짝수번지 사전에 독립적으로 저장됨에 따라서 두 개의 사전에 같은 내용이 저장될 수가 있으며 이 때문에 X-Match 알고리즘에 비해서 약간의 압축률 저하를 가져올 수 있다. 그림 4는 완전매칭이 발생하는 여러 가지 경우의 사전의 갱신 과정을 보여준다. 그림 4(a)는 입력워드 P가 자신이 저장될 사전에서 완전매칭이 발생한 경우로서 입력워드 P가 저장되는 사전은 완전매칭된 엔트리까지만 쉬프트된다. 그림 4(b)는 입력워드 D가 다른 쪽 사전에서 완전매칭이 발생한 경우로서 다른 쪽 사전은 완전매칭된 엔트리까지만 쉬프트된다. 그림 4(c)는 두 개의 입력 워드 K와 P가 한 쪽 사전에서 동시에 완전매칭된 경우이다. 이 경우의 사전 갱신은 서로 다른 사전에 데이타가 중복이 되는 것은 허용할 지라도 같은 사전에 데이타가 중복이 되는 것이 허용되지 않도록 하기 위해서 이 사전에 입력되는 워드 P와 완전매칭된 엔트리까지만 쉬프트가 되도록 한다. 이렇게 하면 나머지 완전매칭 엔트리 K는 사전에 남게 되어 다른 사전에 입력되는 워드 K와 중복이 되게 된다. 그림 4(d)는 두 워드가 동시에 양쪽 사전에서 완전매칭이 발생한 경우로서 그림 4(c)의 경우와 같이 자기 사전에 입력되는 워드와 완전매칭된 엔트리까지 쉬프트를 한다. 이 경우에 다른쪽 사전에서 완전매칭이 이루어진 엔트리는 그대로 남게 되어서 두 워드 모두 데이타가 중복된 상태가 유지된다.

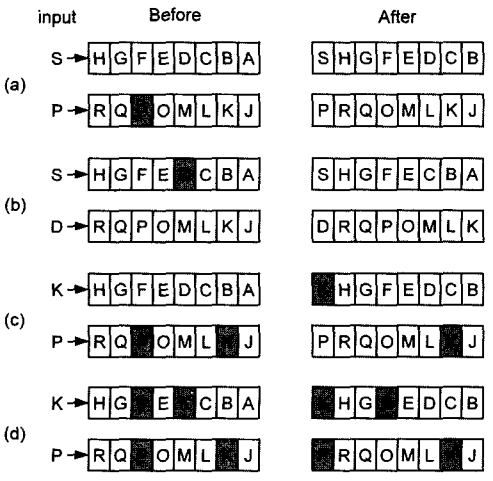

그림 4 여러 가지 사전 갱신의 예

X-MatchCP에서의 사전의 갱신 과정을 정리하면 다음과 같다. 여기서의 사전은 짝수 사전 또는 홀수 사전을 의미한다.

1. 사전에 완전매칭이 발생하지 않으면 사전의 마지막 엔트리까지 쉬프트 동작을 수행한다.
2. 사전에 1개의 완전 매칭이 있으면 완전 매칭된 엔트리까지 쉬프트 동작을 수행한다.
3. 사전에 2개의 완전 매칭이 있으면 이 사전에 입력되는 데이타와 완전 매칭된 엔트리까지 쉬프트 동작을 수행한다.

그리고 입력 워드는 자신이 속한 사전의 맨 앞에 저장된다. 이러한 방식으로 사전을 갱신하면 서로 다른 사전에 중복된 데이타가 존재할 수 있을 지라도 한 사전에는 중복된 데이타가 존재할 수 없다.

지금까지 소개한 X-MatchCP 알고리즘의 압축과정과 사전 갱신과정은 다음과 같은 의사 코드로 나타낼 수 있다.

```
Algorithm X-MatchCP
  clear dictionary D
  while (there are more data to be compressed) do
    read two words from input data stream
    W0 = first word, W1 = second word
    (A0, P0) = search(D, W0),
    (A1, P1) = search(D, W1)
    if (|P0| ≥ 2) then C0=[0, A0, H(T0), B(W0,T0)]
    else C0=[1,W0]
    if (|P1| ≥ 2) then C1=[0, A1, H(T1), B(W1,T1)]
    else C1=[1,W1]
    generate code C = [C0, C1]
    A_odd = fullmatch(D_odd, W0, W1)
    A_even = fullmatch(D_even, W1, W0)
    shift(D_odd, W0, A_odd), shift(D_even, W1, A_even)
  end
```

이 코드에서 search(D, W)는 사전 D에서 워드 W를 검색하여 가장 많이 매칭된 사전 엔트리의 주소 A와 매칭패턴 P를 반환하는 함수이다. |P|는 4비트 매칭패턴 P에 있는 1의 갯수이다. D_{even}은 사전 D의 짝수번지 엔트리의 집합이고 D_{odd}는 사전 D의 홀수번지 엔트리의 집합이다. fullmatch(D_k, W_i, W_j)는 사전 D_k에서 워드 Wi와 Wj를 검색하여 완전매칭되는 주소를 반환하는 함수인데 두 워드 모두 완전매칭되면 Wi와 완전매칭되는 주소를 반환하며 모두 완전매칭되지 않으면 사전의 마

지막 번지를 반환한다. 그리고 shift(D_k, W, A)는 사전 D_k의 처음부터 A번지까지 쉬프트를 하고 사전의 첫번째 엔트리에 워드 W를 저장하는 동작을 나타낸다. search와 fullmatch 함수는 알고리즘 표현의 편의를 위해서 분리하여 나타낸 것으로 실제 구현을 할 때에는 함께 구현된다.

3.3 run-length 부호화

X-MatchPRO 알고리즘은 X-Match 알고리즘에 반복되는 워드에 대한 run-length부호화 기능을 추가한 것이다. 그림 2(a)의 단일 X-MatchPRO 알고리즘을 적용할 때에 하나의 run-length 부호화 데이타로 표시되는 반복 워드 블록이 그림 2(b)와 같이 X-MatchPRO 알고리즘을 병렬로 수행한다면 이 워드 블록이 둘로 쪼개져서 2개의 run-length 부호화 데이타로 표시되므로 추가적인 압축률 저하를 가져온다. X-MatchCP 알고리즘에서는 이와 같은 병렬 X-Match 알고리즘의 단점을 해결하는 run-length 부호화 방법을 제공한다.

그림 5는 run-length 부호화로 표시되는 반복되는 워드를 나타낸다. 같은 워드가 반복되는 경우에는 처음과 마지막을 제외하면 첫째 워드와 둘째 워드는 서로 같다. 이 성질을 이용하면 반복되는 워드들을 하나의 run-length 데이타로 나타낼 수 있다.

첫째 워드 또는 두 워드 모두가 전체 사전의 0번지의 엔트리와 완전매칭이 되면 워드가 반복됨을 나타내며이를 이용하여 다음과 같이 run-length 부호화를 할 수 있다.

1. 압축을 시작할 때에는 보통압축 모드가 되며 앞에서 기술한 압축동작을 수행한다.

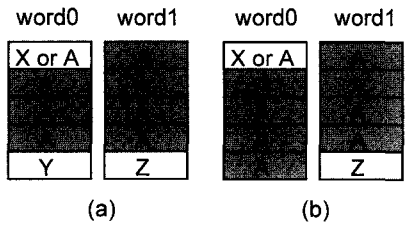

그림 5 run-length 부호화로 표시되는 반복 워드

2. 현재 압축되는 두 워드가 서로 같고 0번지의 엔트리와 완전매칭이 되면 압축 코드를 생성하는 것 대신에 run-length 부호화 모드로 전환하고 run-length를 2로 초기화한다.

3. run-length 부호화 모드에서 계속적으로 두 워드가 0번지의 엔트리와 완전 매칭이 되면 run-length를 2씩 증가시킨다. 그렇지 않으면 보통 압축 모드로 전환하며 이 때에 그림 5(b)와 같이 첫째 워드가 0번지의 엔트리와 완전매칭이 되면 이 워드까지 반복되는 것이므로 run-length를 1을 증가시킨다.

4. run-length 부호화 모드에서 보통 압축 모드로 전환할 때에 run-length 부호화 데이타로 [0, A_{last}, run-length]를 생성하여 출력한다. 여기서 A_{last}은 run-length 부호화를 나타내기 위해서 지정된 주소로서 사전의 마지막 주소를 사용한다. 그리고 run-length 부호화가 종료된 직후의 입력인 현재의 두 워드에 대한 압축코드를 생성한다. 현재의 두 워드 중 첫째 워드가 0번지 엔트리와 완전매칭이 된 경우에는 둘째 워드에 대한 압축코드만 생성한다.

이와 같은 run-length 부호화는 연속적인 입력에 대한 정보를 필요로 하며 파이프라인 구조로 X-MatchCP 압축기를 구현하는 경우에 파이프라인의 이전 단계에서 다음 입력 워드에 대한 정보를 얻을 수 있으므로 run-length 부호화 구현이 간단해진다. 그림 6은 파이프라인의 각 단계에서의 run-length 부호화를 위한 신호들을 나타낸 것이다.

이 그림에서 both는 입력된 두 워드가 사전의 0번지의 엔트리와 동시에 매칭되는 지를 나타내며 first는 첫째 워드만 사전의 0번지의 엔트리와 매칭되는 지를 나타낸다. 이 두 값은 2단계인 decide best match 단계에서 얻을 수 있으며 파이프라이닝이 진행됨에 따라서 레지스터를 통해서 다음 단계로 넘겨진다. counter는 run-length 길이를 세기 위해서 사용되며 초기값은 0이다. 3단계인 code generator단계에서 bothC가 1이면 run-length 부호화 모드를 타나내므로 현재 counter값

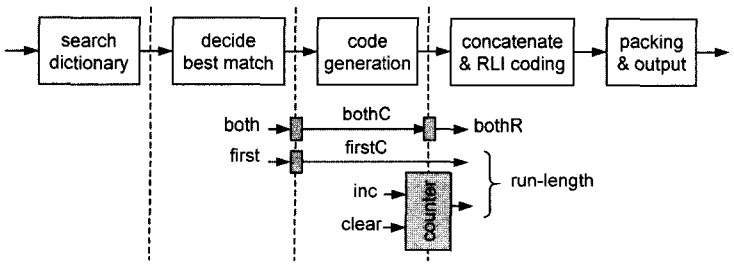

그림 6 파이프라인의 각 단계의 run-length 부호화 동작

을 증가시킨다. 그리고 bothC가 0이면 보통 압축 모드이므로 counter값을 0으로 만든다. counter의 증가 제어신호 inc를 bothC에, 클리어 제어신호 clear를 bothC를 반전(invert)시켜서 연결하면 counter의 이러한 동작이 수행된다. counter 값의 변화는 클럭에 동기가 되어 현재 사이클의 마지막 시점에서 이루어지며 다음 단계인 4단계에서 사용된다.

4단계인 concatenate & RLI coding 단계는 3단계에서 생성된 압축코드를 결합하고 출력하는 단계이다. 이 단계에서 bothR이 1이면 현재의 두 워드는 run-length 부호화를 하고 있다는 것을 뜻한다. 그리고 이 때 3단계의 bothC는 다음 두 워드의 run-length 부호화 상태를 나타내며 4단계에서의 코드 생성 동작에 영향을 미친다. 4단계에서의 run-length 부호화 동작은 다음과 같다.

1. bothR과 bothC가 동시에 1이면 run-length 부호화가 다음 워드까지 계속해서 진행 중인 상태이므로 압축코드를 출력하지 않는다.

2. bothR이 1이지만 bothC가 0이면 run-length 부호화가 현재 워드에서 종료됨을 나타내며 run-length 부호화 코드를 출력한다. run-length값은 counter값과 firstC를 결합하여 얻어진다. counter값이 1씩 증가 할 때마다 run-length는 2씩 증가하게 되므로 run-length는 counter값의 2배가 된다. 그리고 firstC가 1이면 다음 두 워드 중 첫째 워드까지 같은 워드가 반복됨을 뜻하므로 출력을 할 때에 run-length는 1이 더 증가해야 하는 데 firstC를 run-length의 최하위 비트로 사용하여 간단히 해결할 수 있다.

3.4 추가적인 개선 방법

압축하는 두 워드의 매칭 주소와 매칭 패턴이 서로 같으면 둘째 워드의 압축코드를 생성할 때에 같은 매칭 주소와 매칭 패턴을 사용하는 것 대신에 첫째 워드와 같음을 의미하는 표시와 함께 매칭되지 않은 바이트들만 나타내면 더 압축을 할 수 있게 된다.

두 워드 W0, W1의 매칭 주소와 매칭 패턴이 서로 같고 매칭주소와 매칭패턴이 각각 A와 P라고 하면 X-MatchCP 알고리즘에서는 둘째 워드에 대한 압축 코드를 [0, A, $H(P)$, $B(W1,P)$] 대신에 [0, A_{last}, $B(W1, P)$]로 생성한다. 여기서 A_{last}은 둘째 워드의 매칭 주소와 매칭 패턴이 첫째 워드와 같음을 나타내기 위해서 사용하는 주소로서 run-length 부호화 나타내기 위해서 사용된 주소와 같은 주소인 사전의 마지막 주소이다. run-length 부호화 코드는 항상 첫째 워드의 압축 코드 위치에 나타나며 매칭 주소와 매칭 패턴이 첫째 워드와 같음을 나타내는 압축코드는 항상 둘째 워드의 압축 코드 자리에 나타나므로 같은 주소를 사용할 지라도 두 경우는 구분할 수 있다. 그러므로 압축의 효율을 위해서 같은 주소를 사용한다. 이 동작은 코드 생성 단계인 3단계에서 수행한다.

4. X-MatchCP 알고리즘의 평가

X-MatchCP 압축 알고리즘은 X-MatchPRO 압축 알고리즘을 병렬로 동작하도록 한 것으로서 기존의 병렬 X-Match 알고리즘의 두 가지 문제점을 해결하였다. 사전 검색시에 전체 사전을 대상으로 검색을 함으로써 매칭 빈도의 저하를 줄이도록 하였고 두 워드에 대해서 병렬로 생성된 압축 코드를 결합하여 출력함으로서 병렬로 수행되는 두 워드 블록의 압축률 차이로 인한 처리 문제를 해결하였다. 그리고 run-length부호화도 두 워드 단위로 수행함으로써 기존 방법을 사용하여 병렬로 수행할 때에 두 개의 run-length부호화 코드가 생성되는 단점을 없앴다. 그리고 압축되는 두 워드가 같은 매칭 주소와 매칭 패턴을 보일 경우에 둘째 워드를 더 압축하여 표기하는 방법을 사용하여 압축률을 개선하려고 시도하였다.

X-MatchCP 알고리즘의 주된 평가는 속도와 압축률 두 측면에서 이루어질 수 있다. X-Match 알고리즘은 다른 압축 알고리즘에 비해서 압축률 측면에서는 다소 뒤떨어지지만 압축 속도 면에서는 가장 우위에 있다. LZ 알고리즘 또는 그 변형 알고리즘을 구현한 여러 하드웨어 압축기는 1바이트 압축에 1~2 사이클이 소요되지만 X-Match알고리즘은 1사이클이 32비트, 즉 4바이트를 압축할 수 있다. 이에 비해서 X-MatchCP 알고리즘은 1사이클에 64비트, 즉 8바이트를 압축함으로써 압축 속도를 X-Match보다 2배 향상시킨다. 그리고 1사이클의 길이는 파이프라이닝을 사용하여 구현하면 짧게 할 수 있다. 하드웨어 압축기의 클럭속도가 P MHz라고 하면 압축속도는 기존의 X-Match알고리즘이 4P MB/sec인데 비해서 X-MatchCP알고리즘은 8P MB/sec이다.

X-MatchCP 알고리즘의 압축률을 X-MatchPRO와 비교하기 위해서 시뮬레이션을 통해서 압축률을 산출하였다. 압축률에 대한 정의는 여러 가지가 있지만 여기서 압축률은 입력 데이타 길이에 대한 출력 데이타의 길이의 비를 백분율(percent)로 나타낸 것으로 정의한다. 압축률은 메모리 데이타와 파일 자료를 대상으로 산출하였다.

압축률 평가에 사용된 메모리 데이타는 가상 메모리 시스템에서의 압축 캐쉬 방법의 성능 평가[14]를 위해서 Kaplan에 의해서 생성된 메모리 트레이스(trace) 데이타[15]이다. 이 데이타는 프로그램이 수행되면서 메모리에서 페이징 아웃되어 압축 캐쉬에 저장되는 페이지의 내용을 저장한 것으로서 압축되는 메모리 내용의 동적인 상태를 잘 나타내고 있다. 메모리 데이타의 특성을

파악하고 압축률을 평가하기 위해서 많은 연구들에서 메모리의 스냅샷(snapshot) 데이타를 사용하였으나 이는 메모리의 정적인 상태를 나타낸 것이다. Kaplan에 의해서 생성된 데이타는 메모리 내용의 동적인 상태를 나타내므로 정적인 상태를 나타내는 메모리 스냅샷 데이타보다 실제의 상황을 더 잘 반영한다. 이 데이타는 페이지 크기가 4KB이고 리눅스 운영체제를 사용하는 Intel x86 시스템에서 8개의 프로그램을 수행시키면서 페이징 아웃되는 페이지의 내용을 수집한 것으로서 압축된 파일의 크기의 합이 1.7GB를 넘는 매우 방대한 양을 가지고 있다.

이 데이타를 사용하여 사전의 크기가 64워드이고 블록의 크기는 페이지 크기와 같은 4KB로 하여 X-MatchPRO와 X-MatchCP 알고리즘을 적용했을 때의 압축률을 구하였다. 결과는 그림 7에 나타나 있다. 이 그림에서 프로그램에 따라서 약간의 차이는 있지만 X-MatchCP 알고리즘은 X-MatchPRO와 거의 비슷한 압축률을 얻을 수 있음을 알 수 있다. 두 방법의 압축률의 차이는 최대 0.6%이다.

압축률 평가에 사용된 파일 자료는 Canterbury data set[16,17]이다. 이 자료는 컴퓨터 시스템에서 사용되는 텍스트 파일, 프로그램 소스, 웹 페이지, 엑셀 파일, 팩스 이미지 등 여러 가지 유형의 대표적인 파일들을 포함한다. 이 파일 자료에 대해서 두 알고리즘을 적용했을 때의 압축률은 그림 8에 나타나 있다. 이 그림에서도 두 방법의 압축률이 거의 비슷함을 알 수 있다. 두 방법의 압축률의 차이는 최대 0.4%이다.

X-MatchCP 알고리즘은 하드웨어 구현을 위한 압축 알고리즘이다. 이 알고리즘을 하드웨어로 구현했을 때의 동작을 검증하기 위해서 Verilog 하드웨어 기술언어를 사용하여 동작(behavior) 레벨의 설계를 하였으며 SynaptiCAD의 VeriLogger를 사용하여 컴파일을 하여 압축 하드웨어의 동작을 시뮬레이션을 하였다. Verilog 언어로 작성된 압축 하드웨어의 설계는 그림 3의 구조를 가지고 있으며 4단계의 파이프라이닝을 사용한 구조를 사용하였다. 그림 3에서 마지막 두 단계는 한 파이프라이닝 단계로 동작하는 것으로 설계를 하였다. 이 설계는 파이프라인을 사용한 압축기 구현의 전체적인 동작의 검증을 위한 동작 레벨의 설계이기 때문에 각 단계에서 소요되는 정확한 지연 시간에 대한 분석과 하드웨어의 구현 복잡도에 대한 분석은 수행되지 않았다. 이러한 분석은 X-MatchCP 알고리즘을 FPGA로 구현하는 계속적인 연구에서 다루어질 예정이다. 그리고 압축된 데이타를 복원하는 과정은 압축과정의 역과정으로 수행을 하는 데 압축에 비해서 간단하므로 언급을 하지 않았다.

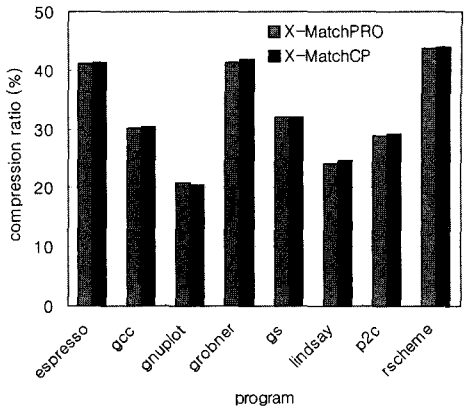

그림 7 프로그램 수행시의 메모리 데이타에 대한 압축률 비교

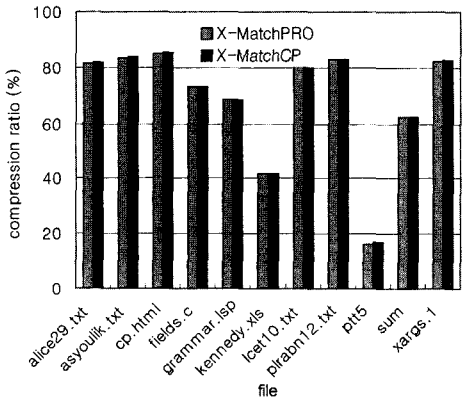

그림 8 canterbury data set에 대한 압축률 비교

5. 결론 및 향후 과제

본 논문에서는 하드웨어로 구현하는 데 적합한 압축 알고리즘인 X-Match 알고리즘을 병렬로 수행하여 압축 속도를 2배로 증가시키면서 X-Match 알고리즘과 거의 비슷한 압축률을 보이는 X-MatchCP 알고리즘을 제시하였다. X-MatchCP 알고리즘은 두 32비트 워드에 대해서 병렬로 X-Match알고리즘을 수행하면서 사전을 검색할 때에 전체 사전을 함께 사용하고 병렬로 생성된 압축코드를 결합하여 출력하고 run-length 부호화도 두 워드에 대해서 한꺼번에 수행하는 방식으로 동작한다. 이렇게 병렬로 수행하는 X-Match 알고리즘이 서로 협동하여 동작하므로 협동 병렬 X-Match, 즉 X-MatchCP 알고리즘이라고 부른다.

제안된 X-MatchCP 알고리즘에 대한 압축률을 메모리 데이타와 파일 자료를 사용하여 산출한 결과 X-MatchPRO 알고리즘을 사용한 경우와 압축률이 큰 차

이가 없음을 보여서 제안된 알고리즘의 유용성을 보였고 이 알고리즘의 하드웨어 구현을 위한 전체적인 설계를 Verilog 하드웨어 기술언어를 사용하여 수행하고 동작 레벨의 시뮬레이션을 함으로써 하드웨어 구현에 대한 타당성을 검증하였다.

앞으로의 과제는 FPGA를 목표 하드웨어로 하여 X-MatchCP 알고리즘을 설계 및 구현을 하고 알고리즘의 하드웨어 구현의 복잡도에 대해서 분석을 하는 것과 압축된 내용을 복원하는 하드웨어와 실제의 시스템이 사용을 할 수 있도록 압축 하드웨어의 입력, 출력 인터페이스에 대한 설계를 수행하는 것이다.

참 고 문 헌

[1] F. Douglis, "The compression cache: using on-line compression to extend physical memory," *Proc. Winter USENIX Conf.*, pp. 519-529, Jan. 1993.

[2] R. Williams, "An extremely fast ZIV-Lempel data compression algorithm," *Proc. Data Compression Conf.*, pp. 362-371, Apr. 1991.

[3] P. Wilson, S. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," *Proc. USENIX Annual Conf.*, pp. 101-116. June 1999.

[4] S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," *Proc. 15th. Parallel and Distr. Processing Symp.*, pp. 630-636, Apr. 2001.

[5] R. Tremaine, T. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy, "Pinnacle: IBM MXT in a Memory Controller Chip," *IEEE Micro*, vol. 21, no. 2, pp. 56-68, Mar/Apr, 2001.

[6] P. Franaszek, J. Robinson, and J. Thomas, "Parallel compression with cooperative dictionary construction," *Proc. Data Compression Conf.*, pp. 200-209, Apr. 1996.

[7] M. Kjelso, M. Gooch, CH, and S. Jones, "Design and performance of a main memory hardware data compressor," *Proc. 22nd EuroMicro Conf.*, pp. 423-430, Sep. 1996.

[8] S. Jones, "Partial-matching lossless data compression hardware," *IEE Proc. Computer Digital Tech.*, vol. 147, no. 5, pp. 329-334, Sep. 2000.

[9] S. Jones and J. Nunez, "Data compression having improved compressed speed," International patent application no. WO0156169, Aug. 2001.

[10] J. Nunez, C. Feregrino, S. Bateman, and S. Jones, "The X-MatchLITE FPGA-based data compressor," *Proc. 25th EUROMICRO Conf.*, pp. 126-132, Sep. 1999.

[11] J. Nunez and S. Jones, "The X-MatchPRO 100 MB/sec FPGA-based lossless data compressor," *Proc. Design, Automation and Test in Europe*, pp. 139-142, Mar. 2000.

[12] J. Nunez, C. Feregrino, S. Jones, and S. Bateman, "X-MatchPRO: A ProASIC-based 200 Mbytes/s full-duplex lossless data compressor," *Proc. 11th Inter. Conf. FPL, Lecture Notes in Computer Science*, Springer, pp. 613-617, Aug. 2001.

[13] J. Nunez and S. Jones, "Loseless data compression programmable hardware for high-speed data networks," *Proc. IEEE Inter. Conf. Field-Programmable Technology (FPT)*, Hong Kong, pp. 290-293, Dec. 2002.

[14] P. Wilson, S. Kaplan, and Y. Samaragdakis, "The case for compressed caching in virtual memory systems," *Proc. USENIX Conf.* pp. 101-116, June 1999.

[15] S. Kaplan, *Compressed caching in virtual memory systems*, http://www.cs.amherst.edu/~sfkaplan/research/compressed-caching

[16] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," *Proc. Data Compression Conf.*, pp. 201-210, 1997.

[17] *The canterbury corpus*, http://corpus.canterbury.ac.nz/

윤 상 균

1984년 서울대학교 전자공학과 공학사 1986년 한국과학기술원 전기및전자공학과 공학석사. 1995년 한국과학기술원 전기및전자공학과 공학박사. 1984년~1990년 현대전자(주). 1992년~2001년 서원대학교 전자계산학과 부교수. 1998년 University of Michigan, visiting scholar. 2001년~현재 연세대학교 문리대학 정보기술학부 부교수. 관심분야는 컴퓨터 시스템, 컴퓨터 하드웨어, 컴퓨터 네트워크