논문 97-01-12

SVLIW 프로세서와 VLIW 프로세서의 명령어 캐싱에 따른 성능 분석

Performance Analysis of Caching Instructions on SVLIW Processor and VLIW Processor

池 承 泫*, 朴 魯 光*, 金 石 日**
(Sung-Hyun Ji, No-Kwang Park and Suk-Il Kim)

요 약

실시간에 VLIW 명령어를 스케줄링하는 SVLIW 프로세서 구조는 실행 중 LNOP(긴 NOP 명령어)를 삽입하여 자원 충돌이나 자료 종속 문제를 스스로 해결할 수 있다. 따라서 SVLIW 프로세서에서는 메모리나 캐시에 적재되는 목적 코드로부터 LNOP 명령어를 제거할 수 있다. 그러므로 SVLIW 프로세서에서는 같은 크기의 캐시를 가진 VLIW 프로세서에 비하여 프로그램의 실행 도중에 발생하는 캐시 미스의 발생 빈도가 적어진다. 캐시 미스가 적게 발생하면 결국 평균 메모리 참조 시간이 짧아지므로 프로그램을 수행하는데 걸리는 실행 사이클의 수가 적어지게 된다. 이러한 특징은 한편 명령어 파이프라인 단계를 늘림으로 인한 영향을 상쇄할 수 있기 때문에 전체적으로 성능을 향상시킬 수 있다. 본 논문에서는 두 가지 프로세서 구조에서 어떤 응용 프로그램을 수행할 때 소요되는 실행 사이클을 예측하는 모델을 확립하고 이를 비교하였다. 또한, 시뮬레이션 결과로부터 캐시 미스가 발생하였을 때 메모리를 참조하는데 걸리는 시간이 길어질수록 SVLIW 프로세서에서의 실행 사이클이 VLIW 프로세서의 경우에 비하여 짧아지는 것을 확인할 수 있었다.

Abstract

SVLIW processor architectures can resolve resource collisions and data dependencies between the instructions while scheduling VLIW instructions at run-time. As a result, long NOP word instructions can be removed from the object code produced for the processor. Thus, the occurrence of cache misses on the SVLIW processor would be lesser than that on the same cache size VLIW processor. Less frequent cache misses on the SVLIW processor would incur less frequent memory access, and thus, the total execution cycles to complete an application would be shortened compared with cases on the VLIW processor. Such a feature eventually compromises effects of longer instruction pipeline stages than those of the VLIW processor. In this paper, we formulate and compare two execution cycle models of the two architectures. A simulation results show that the longer memory access cycles when cache miss occurs, the total execution cycles of SVLIW processor would be shorter than those of VLIW processor. Keyword : Instruction level parallelism VLIW processor SVLIW processor Cache effectiveness Performance comparision

* 忠北大學校 電子計算學科
(Dept. of Computer Science, Chungbuk National Univ.)
** 忠北大學校 컴퓨터科學科
(Dept. of Computer Science, Chungbuk National Univ.)
接受日:1997年7月18日, 修正完了日:1997年10月13日

I. 서 론

근래에 들어와 여러 개의 연산기들로 구성되어 명령어 수준 병렬성을 처리할 수 있는 프로세서 구조와 병렬처리

기법에 관한 연구가 광범위하게 진행되고 있다[1-17]. 그 중에서 슈퍼스칼라 프로세서[1, 2, 11]는 병렬처리를 위한 별도의 프로그래밍 노력이 없이도 프로세서 자체가 일정한 길이의 명령어 스트림으로부터 병렬성을 추출하여 병렬처리를 하도록 하는 프로세서로 개발되었다. 따라서 슈퍼스칼라 프로세서용 컴파일러에서는 명령어간의 병렬성을 향상시키는 위주의 병렬처리 과정을 수행할 뿐, 명령어의 병렬성을 추출하는 기능이 강조되지 않는다. 그러나 실시간에 프로세서가 스스로 병렬성을 추출해서 실행 순서를 결정해야 하므로 프로세서의 구조가 매우 복잡하며, 실시간에 추출되는 병렬성의 최대 길이가 한번에 처리되는 명령어 스트림(윈도우 사이즈)의 길이에 좌우되므로 추출하는 명령어 병렬성의 길이가 작은 것이 문제가 되고 있다.

VLIW 프로세서 구조[1-14]는 컴파일러가 전체 코드로부터 명령어의 병렬성을 추출할 수 있기 때문에 추출 가능한 명령어 병렬성의 최대 길이가 매우 길어서 슈퍼스칼라 프로세서 구조의 경우에 비하여 빠른 계산이 가능하다. 그러나 이 구조는 컴파일러가 처리해야 하는 병렬처리의 과정이 매우 복잡하여 계산속도가 컴파일러의 성능에 따라 결정되는 문제점이 있다. 즉, VLIW 프로세서는 슈퍼스칼라 구조에 비하여 하드웨어의 구성이 간단하지만 목적코드를 생성하는 과정이 복잡해지므로 좋은 컴파일러를 만드는 것이 매우 어렵다.

VLIW 프로세서용 컴파일러는 목적코드를 생성하는 과정 중 프로세서 상에서 명령어간의 병렬성을 추출하고 이 병렬성을 토대로 긴 명령어(VLIW)를 구성한 다음, VLIW 명령어간의 자원 충돌 여부를 판단하여 LNOP(Long NOP word) 명령어를 하나씩 삽입하는 복잡한 과정을 수행해야 한다. 이 과정이 정확하게 수행되지 못하면 계산 결과가 달라지거나 실행사이클이 불필요하게 증가하게 된다. 뿐만 아니라 목적코드내에 많은 LNOP 명령어가 차지하고 있으므로 메모리에 적재될 목적코드내에 의미있는 연산이 수행될 명령어가 존재할 비율이 낮아지게 된다. 이러한 문제는 LNOP 명령어를 제거하도록 목적코드를 압축시키는 방법으로 해결할 수 있으나 이들 방법도 캐시에서는 원래의 목적코드 형태로 재구성되므로 LNOP 명령어가 목적코드내에 삽입되는 것을 해결할 수 없다[6,10].

캐시에 적재되는 명령어 중에서 LNOP 명령어를 제거하기 위해서는 실행 중 명령어간의 자료 종속성을 목적코드에 삽입하는 VFPE 구조[15]나 실행 중 명령어를 스케줄링하는 SVLIW 구조[16]와 같이 실행 중에 명령어 간의 자료 종속성과 명령어를 실행함에 따라 점유하게 되는 자원의 충돌을 검사하여 연산기가 자동적으로 LNOP 명령어를 수행하는 구조가 필요하다.

VFPE 프로세서 구조[15]는 목적코드에서 LNOP을 제거하는 대신 명령어간의 자료 종속성을 검출하여 컴파일러로 하여금 명령어의 실행 순서를 실행시 결정할 수 있도록 자료 종속성 정보를 목적코드에 포함시키도록 하며 하드웨어가 실행 중 명령어간의 자료 종속성 정보를 이용하여 실행 순서를 동적으로 결정한다. VFPE 구조를 위한 컴파일러의 컴파일 과정은 VLIW 구조용 컴파일러가 수행하는 대부분의 역할을 수행하므로 컴파일러의 구성도 VLIW 프로세서용에 비하여 크게 어렵지 않다. 뿐만 아니라 생성된 목적코드를 VFPE 프로세서에서 실행시키는 것이 동일한 연산기로 구성된 VLIW 프로세서에서 같은 작업을 실행시키는 것보다 우수한 것으로 보고되고 있다[15]. 다만 이 구조는 프로세서를 구성하는 연산기가 서로 다른 명령어를 처리할 수 있도록 명확히 구분되어야 하므로 프로세서를 구성하는 연산기의 수를 VLIW 프로세서와 같이 늘릴 수 없는 단점이 있다.

SVLIW 프로세서 구조[16,17]는 VLIW 프로세서 구조에 동적 스케줄러를 추가하여, VLIW용 컴파일러가 명령어간 자료 종속성 여부와 자원 충돌 여부를 검사하여 LNOP을 삽입하는 과정을 하드웨어가 담당하도록 하는 구조이다. 즉, SVLIW 프로세서에서는 VLIW 명령어를 연산기로 제공하기 직전에 명령어간의 자원 충돌 여부와 자료 종속성 존재 여부를 판단하여 자원 충돌이 예상되거나 자료 종속성이 존재할 경우에만 연산기로 LNOP을 제공하여 자동적으로 자원 충돌이나 자료 종속 문제를 해결하도록 한다. 따라서 SVLIW 구조는 VLIW 구조와 같이 프로세서를 구성하는 연산기의 배치를 자유롭게 설정할 수 있으면서도 LNOP 명령어를 목적코드로부터 제거할 수 있는 장점이 있다.

이상의 두 가지 프로세서 구조는 목적코드에서 LNOP이 제거될 수 있음으로 인하여 VLIW 프로세서용으로 생성된 목적코드의 길이에 비해 목적코드가 항상 짧아질 수 있다. 그러나 하드웨어 측면에서 보면 자원 충돌이나 자료 종속성을 검사하는 과정이 필요하므로 명령어 파이프라인 단계가 VLIW 프로세서에 비하여 늘어나게 된다. 따라서 이 구조에서는 어떤 목적코드를 수행하는 데 필요한 실행사이클이 늘어날 것이나, 목적코드의 길이가 짧아지므로 일정한 캐시에 포함되는 의미있는 명령어가 많아짐으로 인하여 평균 메모리참조 사이클이 짧아지게 된다. 즉, 전체적으로 어떤 응용 프로그램을

VLIW 프로세서와 VFPE 또는 SVLIW 프로세서에서 실행하였을 때 소요되는 실행사이클의 수는 단순히 목적 코드의 길이나 파이프라인 단계의 길이로 비교될 수 없다.

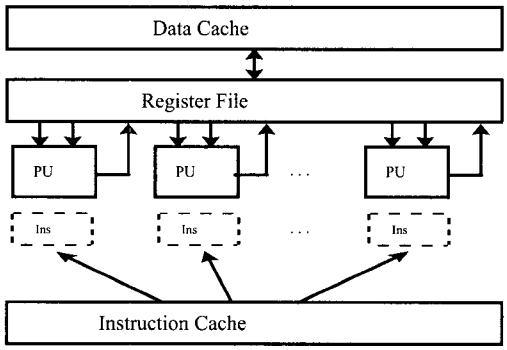
본 논문에서는 VLIW 프로세서와 실행 중 명령어 스케줄링이 가능한 프로세서 구조에서 여러 가지 벤치마크 프로그램을 실행시키면서 실행사이클을 측정하여 프로세서 구조의 성능을 비교하였다. 여기서 VFPE 구조는 VLIW 프로세서와 같이 다양한 연산기를 지닌 프로세서로 구성할 수 없으므로 본 논문에서는 비교의 대상에서 배재하고 SVLIW 구조를 중심으로 VLIW 구조와 성능을 비교하였다. 즉, 본 논문에서는 VLIW 프로세서와 동일한 연산기로 구성된 SVLIW 프로세서에서 동일한 응용 프로그램을 수행할 때 소요되는 실행사이클을 계산하는 모델을 수립하고 여러 가지 벤치마크에 대한 실험을 통하여 실제로 수행된 실행사이클을 측정하였다. 이를 위하여, 제 2절에서는 본 논문에서 비교할 대표적인 VLIW와 SVLIW 구조를 비교하고 설명하였다. 제 3절에서는 캐시의 크기와 메모리참조 사이클을 토대로 프로세서 구조별로 실행사이클을 모델화하고 이를 비교하였다. 그리고 제 4절에서는 여러 가지 벤치마크에 대하여 캐시의 크기와 메모리참조 사이클을 변화시키면서 같은 수의 연산기로 구성된 SVLIW 구조와 VLIW 구조에서 계산에 필요한 실행사이클을 비교하였다. 마지막으로 제 5절에서는 결론과 향후 연구방향을 기술하였다.

Ⅱ. VLIW 명령어를 수행하는프로세서 구조

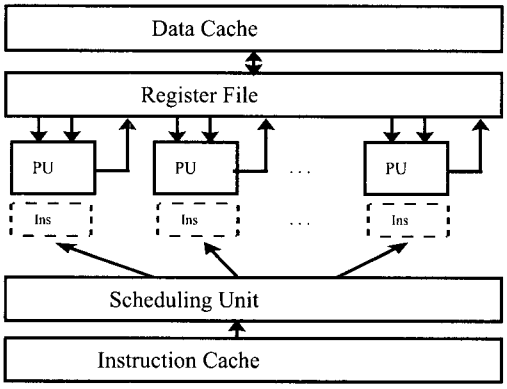
2-1. VLIW와 SVLIW 프로세서 구조

그림 1(a)는 대표적인 VLIW 프로세서 구조이다. VLIW 프로세서는 여러 개의 연산기들, 명령어 캐시와 데이터 캐시 및 레지스터 파일로 구성된다. 또한, 매 사이클마다 명령어 캐시로부터 단위 명령어들의 집합인 하나의 VLIW 명령어를 인출(fetch)한 후, 각각의 연산기에게 분배하여 작업을 수행토록 한다. 레지스터 파일은 임시 변수들을 저장할 때 사용될 뿐 아니라, 연산기들 사이의 자료 교환을 위한 빠른 통신 선로로도 사용된다

그림 1(b)에 보인 SVLIW 프로세서 구조는 명령어 캐시로부터 인출한 VLIW 명령어를 현재 연산기에서 수행 중인 VLIW 명령어들과 자료 종속성이 존재하는지 여부와 자원 충돌 여부를 검사하여 인출한 VLIW 명령어를 연산기에게 배분할 것인지를 결정하는 스케줄링 유니트를 가지고 있다. 스케줄링 유니트는 매 사이클마다 인출(fetch) 단계와 분석(decode) 단계를 거쳐 VLIW 명령어를 구성하는 단위 명령어가 수행되는 데 필요한 피연산자가 결정되면, VLIW 명령어와 이미 연산기에서 실행되고 있는 단위 명령어간에 어떠한 충돌이 발생하는 지를 검사한다. 만약 충돌이 발생하면 현재 VLIW 명령어는 모든 충돌이 해결될 때까지 스케줄링되지 않으며, 충돌이 발생하지 않으면 VLIW 명령어를 구성하는 단위 명령어는 지정된 연산기로 할당되어 실행을 수행한다.



(a) VLIW 구조



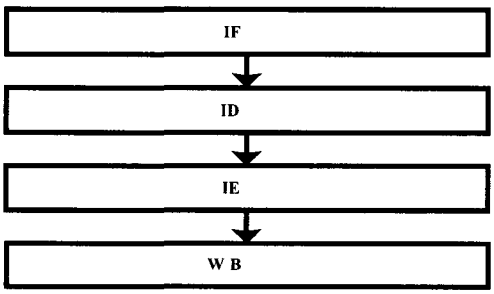
(b) SVLIW 구조

그림 1. VLIW 명령어를 수행하는 두 가지 프로세서 구조

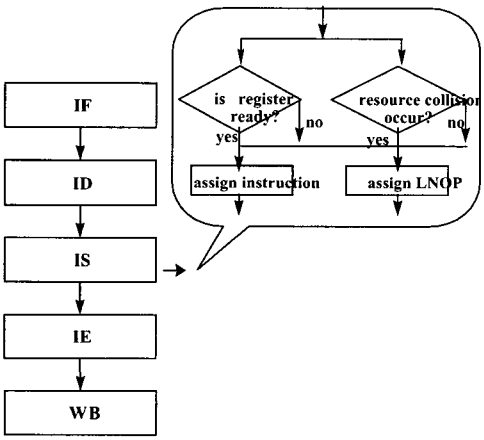
Fig. 1. Two processor architectures.

2-2. 명령어 파이프라인

전통적인 VLIW 구조의 명령어 파이프라인 단계는 그림 2(a)와 같이 IF (Instruction Fetch), ID (Instruction Decode), IE (Instruction Execute) 및 WB (Write Back)의 네 단계로 구성된다. IF 단계에서는 명령어 캐시로부터 하나의 VLIW 명령어를 가져온다. 이때 VLIW 명령어가 캐시에 존재하지 않으면 메모리를 참조하여 해당 VLIW 명령어를 인출한다. ID 단계는 IF 단계에서 인출된 VLIW 명령어를 분석하여 피연산자를 추출한다. ID 단계가 끝나면 각 단위 명령어는 IE 단계에서 실행되고 그 결과는 WB 단계에서 레지스터 파일에 저장된다.



(a) VLIW 구조



(b) SVLIW 구조

그림 2. 프로세서에서의 파이프라인 단계
Fig. 2. Instruction pipeline stages of two processors.

SVLIW 프로세서의 명령어 파이프라인 단계는 그림 2(b)와 같이 다섯 단계로 구성된다. IF 단계와 ID 단계는 VLIW 프로세서와 같이 VLIW 명령어를 인출하고, 각 단위 명령어의 피연산자를 추출한다. 이어서 IS 단계에서는 IE 단계에서 실행될 VLIW 명령어가 이미 IE 단계에서 실행 중인 VLIW 명령어들과의 파이프라인 단계에서의 자원 충돌 여부 및 자료 종속성 여부를 검사한다. 만약 자원 충돌이나 자료 종속성이 없으면, 스케줄링하려는 VLIW 명령어는 IE 단계로 전달되어 실행된다. 그렇지 않으면, 자료 종속성이나 자원 충돌 문제가 해소될 때까지 IE 단계로 LNOP 명령어를 전달하여 VLIW 명령어의 실행을 지연시킨다. IE 단계가 종료되면 WB 단계에서는 연산 결과를 레지스터 파일에 기록한다.

이상에서와 같이 VLIW 명령어 파이프라인 단계가 네 단계로 구성되는 데 비해, SVLIW 명령어 파이프라인 단계는 다섯 단계로 구성되므로, 동일한 작업을 수행하는 경우에 SVLIW 프로세서의 파이프라인 과정은 VLIW 프로세서의 경우에 비하여 더 많은 실행사이클이 필요하다. 그러나, SVLIW 프로세서는 동적 스케줄러를 내장함으로 인하여 VLIW 프로세서에서와 같이 목적코드내에 LNOP들을 삽입할 필요가 없다. 즉, 동일한 작업을 하는 목적코드의 길이를 비교하면 SVLIW 프로세서용 목적코드의 길이가 VLIW 프로세서용 목적코드의 길이에 비해 짧다. 따라서 SVLIW 프로세서에서는 이와 같이 목적코드 길이가 짧아지므로 일정한 캐시크기에 목적코드를 실행할 때 발생하는 캐시미스의 횟수가 적어진다. 캐시미스의 횟수가 줄어들면 그만큼 직접 메모리를 참조하는 횟수가 줄어들게 되므로 평균 메모리참조 사이클이 짧아지게 되어 전체적으로 어떤 작업을 VLIW 프로세서 상에서 수행할 때 요구되는 실행사이클의 수에 비하여 SVLIW 프로세서 상에서 같은 작업을 수행시킬 경우에 필요한 실행사이클의 수가 더 짧아지게 되어 파이프라인 단계가 늘어나는 역효과를 상쇄할 수 있게 된다. 그 가능성을 다음 절에서 분석하였다.

Ⅲ. 프로세서 실행 모델

어떤 작업을 수행하는데 걸리는 실행사이클의 총 수는 1) 캐시미스(cache miss)시 메모리 참조에 필요한 사이클의 수, 명령어 실행시 필요한 사이클의 수, 파이프라인을 채우는데 필요한 사이클과 파이프라인을 비우는데

필요한 사이클의 수에 의하여 결정된다. 즉, 실행사이클의 총 수

$$T = T_m + T_x + T_p . \quad (1)$$

여기서 T_m은 캐시미스가 일어났을 때 메모리를 참조하는 데 필요한 사이클의 수이다. 따라서 한 번의 캐시미스가 발생했을 때 메모리참조에 필요한 사이클을 T_a라고 하면

$$T_m = T_a \times N_m .$$

여기서 N_m는 응용 프로그램이 실행되는 동안 발생하는 캐시미스의 총 횟수이다.

또한, 식 (1)에서 T_x는 캐시미스가 발생하지 않은 경우, 작업이 종료될 때까지 연산기에서 수행되는 명령어들의 실행 횟수이다. 만일 하나의 명령어가 하나의 사이클에 수행될 수 있다면 T_x는 명령어를 수행하는 데 걸리는 실행사이클이라고 할 수 있다. 본 논문에서는 하나의 명령어를 하나의 사이클에 실행하는 것으로 간주하였다. 또한, T_p는 실행동안 파이프라인을 채우거나 비우는 데 필요한 사이클 수의 합이다. 따라서,

$$T_p = 2(P-1) .$$

여기서 P는 명령어 파이프라인을 구성하는 단계의 수이다.

식 (1)에서 T_p는 프로세서 구조별로 일정하며 T_x 경우에는 실행하는 프로그램에 의해서 결정된다. 만약 분기명령어가 포함되지 않은 어떤 프로그램을 VLIW 프로세서와 SVLIW 프로세서에서 수행시키려고 하면 연산기가 수행하게 되는 명령어들에 의해서 소요되는 총 사이클의 길이 T_x 값은 동일하다. 만약 분기명령어가 포함된 응용 프로그램을 고려한다면, 명령어 파이프라인 수가 프로세서마다 서로 다른 T_x 값을 가지게 될 것이다. 예를 들어 그림 2(b)에서 보인 SVLIW 프로세서에서의 값 T_x^S는 하나의 분기명령어를 실행할 때마다 파이프라인을 비워야 하므로 세 사이클씩 증가되는 데 반해, 그림 2(a)에 보인 VLIW 프로세서에서 T_x^V는 두 사이클씩 증가된다. 즉, SVLIW 프로세서에서 분기명령어의 수행을 위해서는 VLIW 프로세서의 경우에 비해서 한 사이클이 더 필요하므로 그 차이는 수행된 분기명령어의 수에 의하여 결정된다.

$$T_x^V = T_x^S - N_b . \quad (2)$$

여기서 N_b는 작업이 종료될 때까지 실행되는 분기명령어의 총 수이다.

VLIW 프로세서에서 어떤 명령어를 인출하는 과정에서 캐시미스가 일어나면 메모리를 참조하여 명령어를 인출할 수 있을 때까지 파이프라인이 정지(stall)되어야 한다. 그러나 SVLIW 프로세서에서는 캐시미스로 인하여 메모리 참조를 해야 하는 경우에도 동적 스케줄러가 LNOP을 연산기로 제공할 수 있기 때문에 파이프라인은 수행 중인 작업을 계속 수행할 수 있다[17].

본 논문에서는 두 가지 프로세서 구조를 동일한 구조로 간주하여 비교하므로 SVLIW 프로세서의 경우에도 캐시미스가 일어날 때에는 파이프라인이 정지되는 것으로 가정하고 모델링하였다. 즉, 캐시미스가 일어나더라도 식 (2)의 관계가 그대로 유지되는 것으로 가정하였다.

식 (1)에서 첨자 V와 S를 이용하여 각각 VLIW 프로세서에서와 SVLIW 프로세서에서의 값을 표시하면, VLIW 프로세서에서 어떤 작업을 수행하는 데 필요한 실행사이클의 총 수

$$T^V = T_m^V + T_x^V + T_p^V . \quad (3)$$

여기서 T_m^V, T_x^V 및 T_p^V은 각각 VLIW 프로세서상에서 메모리 참조에 의하여 계산이 늦어진 사이클의 수, 명령어의 수행 횟수 및 VLIW프로세서의 파이프라인을 채우거나 비우는 데 걸린 사이클의 합이다. 따라서

$$T_m^V = T_a \times N_m^V \ , \ T_p^V = 2(P^V - 1) .$$

여기서 T_a는 일회의 메모리 참조에 걸리는 사이클의 수, N_m^V는 실행동안 발생하는 캐시미스의 횟수이고 P^V는 파이프라인 단계 값이다.

동일한 방법으로 SVLIW 프로세서상에서 요구되는 실행사이클의 총 수

$$T^S = T_m^S + T_x^S + T_p^S . \quad (4)$$

여기서 T_m^S, T_x^S 및 T_p^S은 각각 SVLIW 프로세서상에서 메모리 참조에 의하여 계산이 늦어진 사이클의 수, 명령어의 수행 횟수 및 SVLIW 프로세서의 파이프라인

을 채우거나 비우는 데 걸린 사이클의 합이다. 따라서

$$T_m^S = T_a \times N_m^S, \quad T_b^S = 2(P^S - 1).$$

여기서 N_m^S는 실행동안 발생하는 캐시미스의 횟수이고 P^S는 SVLIW 프로세서를 구성하는 명령어 파이프라인의 수이다.

식 (3)과 (4)로부터 그림 2에 보인바와 같이 $P^S = 5$ 와 $P^V = 4$ 이라고 가정하면,

$$T^V = T_a \times N_m^V + T_x^S - N_b + 6 \tag{5}$$

또는

$$T^S = T_a \times N_m^S + T_x^S + 8. \tag{5'}$$

식 (5) 및 (5')로부터 각각의 프로세서에서 실행되는 분기명령어의 갯수와 캐시미스의 횟수를 예측할 수 있다면, 각 프로세서에서 작업을 수행하기 위해서 필요한 실행사이클의 길이를 비교할 수 있다. 그런데 동일한 작업에 대해서는 결국 실행되는 분기명령어의 수가 같으므로 결국 두 프로세서 구조에서의 실행시간은 캐시미스의 횟수가 서로 다름에 의하여 영향을 받게 된다.

동일한 작업을 두 가지 프로세서 구조에서 수행할 때, 발생하는 캐시미스의 횟수는 실행되는 명령어의 길이에 따라 결정되는 것으로 간주할 수 있다. VLIW 프로세서용 목적코드에 LNOP 명령어가 균일하게 분포되어 있다고 가정하면 결국 VLIW 프로세서에서 명령어를 인출할 때 캐시미스가 발생할 가능성이 높다. 따라서 캐시미스의 횟수는 LNOP 명령어가 수행된 갯수에 비례해서 늘어난다고 가정하면 VLIW 프로세서에서의 캐시미스의 횟수를 식 (6)과 같이 가정할 수 있다. 즉,

$$N_m^V = N_m^S \times \frac{T_{LNOP} + T_h}{T_h}. \tag{6}$$

여기서 T_{LNOP}는 VLIW 프로세서용 목적코드내에 포함된 LNOP 명령어의 실행 횟수이며, T_h는 LNOP 명령어 이외의 명령어가 실행된 횟수이다. 즉, SVLIW 프로세서에서는 캐시에서 인출하는 명령어의 수가 T_h 개이며 VLIW 프로세서에서는 캐시에서 인출하는 명령어의 총 수가 $T_{LNOP} + T_h$ 개이다.

따라서 식 (5)와 (5')로부터 각각의 프로세서 구조에서 어떤 작업을 수행하는 데 필요한 실행사이클의 차이

$$\begin{aligned} \delta &= T^V - T^S \\ &= T_a(N_m^V - N_m^S) - N_b - 2 \\ &= T_a N_m^S \frac{T_{LNOP}}{T_h} - (N_b + 2). \end{aligned} \tag{7}$$

식 (7)에서 δ는 SVLIW 프로세서나 VLIW 프로세서에서 실행된 LNOP 명령어와 LNOP이 아닌 명령어의 비와 캐시미스의 횟수 및 메모리참조에 필요한 사이클 수에 의해 결정됨을 알 수 있다. 즉 δ가 양수이면 SVLIW 구조가 VLIW 구조에 비하여 δ 만큼 성능이 좋음을 의미한다. 만일 이 값이 음수이면 어떤 작업을 수행하는데 VLIW 구조가 SVLIW 구조에 비하여 적합함을 보여준다. 제 4절에서는 이러한 관찰을 토대로 동일한 연산기로 구성된 두 가지 프로세서 구조에서 여러 가지 벤치마크에 대한 δ 값을 측정하여 비교하였다.

Ⅳ. 실험 및 고찰

본 논문에서는 SVLIW 프로세서와 VLIW 프로세서 구조의 성능을 비교하기 위하여 VLIW 프로세서와 SVLIW 프로세서를 모의할 수 있는 시뮬레이션 시스템을 구현하였다. 시뮬레이션 시스템은 UNIX 환경의 SUN SPARC 10에서 C++로 구현되었다. 구현한 시뮬레이션 시스템은 VLIW 명령어를 생성하는 병렬화 어셈블러와 기존의 VLIW 프로세서와 SVLIW 프로세서를 모의할 수 있는 프로세서 부로 구성된다.

병렬화 어셈블러는 C 프로그램을 입력으로 받아 MIPS R3000 어셈블리 코드를 생성한 후, MIPS R3000 어셈블리 코드내의 매크로 명령어들을 확장하는 매크로 확장기(macro extractor), 매크로 확장기를 통과한 프로그램을 분석하여 병렬성이 있는 VLIW 명령어를 추출하는 병렬성 추출기(parallelism extrctor), 생성된 병렬코드로부터 연산 처리기의 종류와 수에 적합한 목적코드를 생성하는 목적코드 생성기(object code generator)의 순서대로 실행되어 원하는 목적코드를 생성한다. 이와같은 병렬화 어셈블러의

단점은 기본 블록(basic block)에의 병렬성만을 다루기 때문에 프로그램 전체의 병렬성을 추출하지 못하는 점이다. 그러나 SVLIW와 VLIW 구조에서 사용되는 VLIW 명령어의 구성이 동일하므로 두 구조의 성능 측정 결과는 전반적인 성능 차이를 충분히 보여줄 수 있을 것으로 판단된다.

기존의 VLIW 프로세서와 SVLIW 프로세서를 모의할 수 있는 프로세서 구조는 그림 1과 같다. 본 연구를 위한 성능평가의 일환으로 SVLIW 프로세서를, 캐시미스가 발생했을 경우 메모리참조 사이클동안 파이프라인이 정지(stall) 되는 VLIW 프로세서와 동일한 조건에서 비교하기 위해서, 캐시미스시에 메모리참조 사이클동안 파이프라인이 정지되는 경우와 파이프라인을 계속 진행되는 두 경우에 대한 실행사이클을 각각 구분하여 측정할 수 있는 시뮬레이터로 구현하였다. 실험에서는 정수 연산기 2 개로 구성된 SVLIW 프로세서와 VLIW 프로세서를 구성하여 비교하였다. 정수 연산기만으로 프로세서를 구성한 이유는 정수 연산이 한 사이클당 하나의 명령어를 처리하는 반면, 실수 연산은 실행 파이프라인에서의 수행과정이 복잡하므로 VLIW용 병렬화 어셈블러를 구현하는 과정에서 LNOP의 삽입을 정확히 예측하기 어렵기 때문에 컴파일 기법의 차이가 실험결과에 나타나는 것을 배제하기 위함이다. 또한 각 프로세서를 구성하는 연산기의 수를 2 개로 정한 것은 병렬화 어셈블러가 기본 블록에서의 병렬성만을 다루기 때문에 응용프로그램의 병렬성이 별로 크지 않을 것으로 판단했기 때문이다. 본 논문에서 선정한 벤치마크는 정수 연산만을 수행하는 프로그램들로 행렬곱셈 프로그램 MM1 및 MM2, Livermore Loop LL1 및 LL2, merge sort, fibonacci, binary search 등의 7가지이다.

표 1(a)에서 표 1(d)까지는 각각 메모리참조 사이클이 1, 2, 4, 8일 경우에 캐시의 크기를 변화시키면서 분기명령어의 실행 횟수(N_b), LNOP 명령어의 실행 횟수(T_{LNOP}), SVLIW 프로세서에서의 캐시미스의 횟수(N_m^S), VLIW 프로세서에서의 캐시미스의 횟수(N_m^V), SVLIW 프로세서에서 캐시미스가 발생했을 때 메모리참조 사이클동안 파이프라인이 정지되는 경우의 실행사이클의 수(T^S), SVLIW 프로세서에서 캐시미스가 발생했을 때 메모리참조 사이클동안 파이프라인이 계속 진행되는 경우의 실행사이클의 수(T^{SS}) 및 VLIW 프로세서에서의 실행사이클의 수(T^V)를 측정한 것이다.

표 1에서 δ는 $T^V - T^S$ 의 값을 계산한 것으로 δ가 양수이면 SVLIW 구조가 VLIW 구조에 비하여 δ 만큼 빨리 벤치마크 프로그램을 끝낼 수 있음을 의미한다. 만일 이 값이 음수이면 VLIW 구조가 벤치마크 프로그램을 수행하는 데 SVLIW 구조에 비하여 적합함을 나타낸다.

표 1에서 캐시미스가 발생했을 때 메모리참조에만 T_a 사이클이 걸리며, 그 명령어가 들어 있는 블록을 캐시에 적재하는 데 1 사이클이 걸리는 것으로 간주하여 명령어를 인출하는 데 T_a+1 사이클이 걸리는 것으로

표 1. 메모리참조 사이클의 변화에 따른 실행사이클
Table 1. Execution cycles on two processors.

(a) T_a = 1

benchmark	cache size	N_b	T_{LNOP}	T_h	N_m^S	N_m^V	T^{SS}	T^S	T^V	δ
LL1	128	170	844	5355	1462	1692	7330	9131	9419	288
	256				811	938	6849	7829	7911	82
	512				481	556	6679	7169	7147	-22
	1024				320	370	6519	6847	6775	-72
LL2	128	897	3293	7188	2393	3489	12873	15275	16568	1293
	256				1496	2181	11976	13481	13952	471
	512				1196	1743	11677	12881	13076	195
	1024				598	871	11079	11685	11332	-353
MM1	128	291	1459	14567	4013	4414	20186	24060	24569	509
	256				2226	2448	18448	20486	20637	151
	512				1377	1514	17599	18788	18769	-19
	1024				929	1022	17151	17892	17785	-107
MM2	128	267	1704	12166	3425	3904	17245	20728	21417	689
	256				2030	2313	15899	17938	18235	297
	512				1231	1403	15100	16340	16415	75
	1024				747	851	14617	15372	15311	-61
Merge	128	115	550	1441	410	566	2388	2819	3014	195
	256				249	344	2233	2497	2570	73
	512				106	146	2092	2211	2174	-37
	1024				95	131	2081	2189	2144	-45
Fibonacci	128	171	689	1659	543	768	2747	3442	3719	277
	256				342	484	2631	3049	3151	111
	512				246	348	2566	2848	2879	31
	1024				113	159	2461	2582	2501	-81
Binary	128	57	286	866	251	333	1347	1662	1767	105
	256				126	167	1278	1412	1435	23
	512				64	85	1216	1288	1271	-17
	1024				60	79	1212	1280	1259	-21

(b) T_a = 2

benchmark	cache size	N_b	T_{LNOP}^S	T_h^S	N_m^S	N_m^V	T^{SS}	T^S	T^V	δ
LL1	128	170	844	5355	1462	1692	8781	10593	11111	518
	256				811	938	7659	8649	8849	200
	512				481	556	7159	7693	7746	53
	1024				320	370	6839	7167	7145	-22
LL2	128	897	3293	7188	2393	3489	15265	17668	20057	2389
	256				1496	2181	13471	14977	16133	1156
	512				1196	1743	12873	14077	14819	742
	1024				598	871	11677	12283	12203	-80
MM1	128	291	1459	14567	4013	4414	24052	28073	28983	910
	256				2226	2448	20576	22712	23085	373
	512				1377	1514	18878	20165	20283	118
	1024				929	1022	17982	18821	18807	-14
MM2	128	267	1704	12166	3425	3904	20669	24153	25321	1168
	256				2030	2313	17928	19068	20548	580
	512				1231	1403	16330	17571	17828	247
	1024				747	851	15364	16119	16162	43
Merge	128	115	550	1441	410	566	2792	3229	3580	351
	256				249	344	2479	2746	2914	168
	512				106	146	2195	2317	2320	3
	1024				95	131	2181	2284	2275	-9
Fibonacci	128	171	689	1659	543	768	3206	3985	4487	502
	256				342	484	2914	3382	3635	253
	512				246	348	2784	3094	3227	133
	1024				113	159	2574	2695	2660	-35
Binary	128	57	286	866	251	333	1598	1913	2100	287
	256				126	167	1404	1538	1602	64
	512				64	85	1280	1352	1356	4
	1024				60	79	1272	1340	1338	-2

(c) T_a =4

bench mark	cache size	N_b	T^S_{LNOP}	T^S_h	N^S_m	N^V_m	T^{SS}	T^S	T^V	δ
LL1	128	170	844	5355	1462	1692	11693	13517	14495	978
	256				811	938	9279	10262	10725	463
	512				481	556	8119	8612	8815	203
	1024				320	370	7470	7807	7885	78
LL2	128	897	3293	7188	2393	3489	20050	22454	27035	4581
	256				1496	2181	16462	17969	20495	2526
	512				1196	1743	15265	16469	18305	1836
	1024				598	871	12873	13479	13945	466
MM1	128	291	1459	14567	4013	4414	31931	36099	37811	1712
	256				2226	2448	24930	27164	27981	817
	512				1377	1514	21534	22910	23311	392
	1024				930	1022	19741	20679	20851	172
MM2	128	267	1704	12166	3425	3804	27518	31003	33129	2126
	256				2030	2313	21987	24028	25174	1146
	512				1231	1403	18791	20033	20624	591
	1024				747	851	16858	17613	17864	251
Fibona	128	115	550	1441	410	566	3806	4049	4712	663
	256				249	344	2974	3244	3602	358
	512				106	146	2404	2529	2612	83
	1024				95	131	2371	2474	2537	63
Merge	128	171	689	1659	543	768	4208	5071	6023	952
	256				342	484	3539	4066	4603	537
	512				246	348	3248	3586	3923	337
	1024				113	159	2800	2921	2978	57
Binary	128	57	286	866	251	333	2100	2415	2766	351
	256				126	167	1656	1790	1936	146
	512				64	85	1408	1480	1526	46
	1024				60	79	1392	1460	1496	36

(d) T_a = 8

bench mark	cache size	N_b	T^S_{LNOP}	T^S_h	N^S_m	N^V_m	T^{SS}	T^S	T^V	δ
LL1	128	170	844	5355	1462	1692	17537	19365	21263	1898
	256				811	938	12519	13506	14477	971
	512				481	556	10039	10536	11039	503
	1024				320	370	8759	9087	9365	278
LL2	128	897	3293	7188	2393	3489	29622	32026	40201	8965
	256				1496	2181	22446	23953	29219	5266
	512				1196	1743	20049	21253	25277	4024
	1024				598	871	15265	15871	17429	1558
MM1	128	291	1459	14567	4013	4414	47983	52151	55467	3316
	256				2226	2448	33834	36068	37773	1705
	512				1377	1514	27042	28427	29367	940
	1024				930	1022	23458	24395	24939	544
MM2	128	267	1704	12166	3425	3804	41218	44703	48745	4042
	256				2030	2313	30107	32148	34426	2278
	512				1231	1403	23715	24957	26236	1279
	1024				747	851	19846	20601	21268	667
Fibona	128	115	550	1441	410	566	5246	5689	6976	1287
	256				249	344	3970	4240	4978	738
	512				106	146	2828	2953	3196	243
	1024				95	131	2751	2854	3061	207
Merge	128	171	689	1659	543	768	6380	7245	9035	1852
	256				342	484	4907	5434	6539	1105
	512				246	348	4231	4570	5315	745
	1024				113	159	3152	3373	3614	241
Binary	128	57	286	866	251	333	3104	3413	4098	679
	256				126	167	2160	2294	2604	310
	512				64	85	1664	1736	1866	130
	1024				60	79	1632	1700	1812	112

가정하였다. 표 1에서 캐시의 크기가 작을 때에는 T_a의 값에 관계없이 캐시의 크기가 큰 경우에 비하여 δ 값이 큰 것을 볼 수 있다. 이것은 캐시의 크기가 작을수록 VLIW 프로세서에서의 캐시미스 발생률이 SVLIW 프로세서의 경우보다 훨씬 크기 때문이다.

같은 크기의 캐시로 구성된 프로세서를 비교하면 T_a의 값이 작을수록 δ 값이 적어지는 것을 볼 수 있다. 특히 캐시의 크기가 크고 메모리참조 사이클 T_a(=1) 값이 적을수록 VLIW 프로세서의 경우가 SVLIW 프로세서의 경우보다 실행사이클이 적은 것을 알 수 있다. 그 이유는 캐시미스의 영향이 적어서 파이프라인 단계가 길어진 영향이 그대로 반영되기 때문으로 분석된다. 실험에서 선정된 벤치마크들을 이용할 경우 T_a 값이 1 보다 클때는 실험에 사용된 벤치마크의 경우에는 δ 값이 항상 양수인 것을 보여주고 있다.

그림 3은 캐시크기가 512 바이트일 때 T_a 값의 변화에 따른 δ 값을 나타낸 것으로 T_a 값이 클수록 SVLIW 구조가 VLIW 구조에 비하여 유리한 것을 보여주고 있다. 캐시 크기가 작을수록 분명하게 나타나는 이러한 경향은 표 1로부터도 확인할 수 있다.

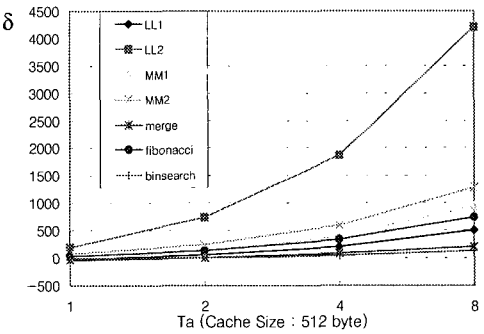


그림 3. 메모리참조 사이클의 변화에 따른 실행사이클 비교

Fig. 3. Execution cycles varying memory access time.

표 1에서 T^{SS}는 SVLIW 프로세서에서 캐시미스가 발생하였을 경우에도 명령어 파이프라인이 정상적으로 진행되는 경우에 실행사이클을 측정한 것이다. VLIW 구조에서는 실행 중 LNOP 명령어를 스스로 발생하지 못하므로 명령어 파이프라인의 작업을 중지시켜야 하나 SVLIW 구조에서는 하드웨어가 LNOP 명령어를 발생시킬 수 있다. 실험결과 T^{SS}가 T^S에 비하여 항상 작은 값을 가지는 것으로 측정되었다. 그림 4는 캐시 크기가 512 바이트일 때 (T^{SS}- T^V)를 도시한 것으로 그림 3과 같이 T_a가 클수록, SVLIW 구조가 VLIW 구조에 비하여 성능이 우수하며 T_a가 작은 경우에도 SVLIW 구조가 VLIW 구조에 비하여 성능이 좋은 것을 알 수 있다. 이상의 실험으로부터 캐시가 성능에 미치는 영향은 VLIW 구조에 비하여 SVLIW 구조가 더 큰 것을 알 수 있다.

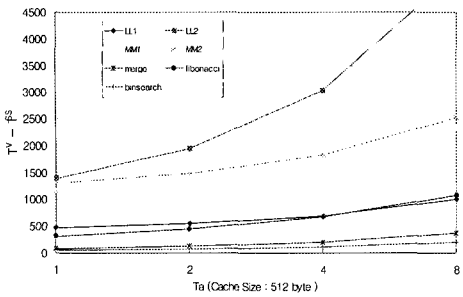


그림 4. 메모리참조 사이클의 변화에 따른 실행 사이클 비교

Fig. 4. Execution cycles varying memory access time.

V. 결 론

본 논문에서는 실시간에 VLIW 명령어를 스케줄링하는 SVLIW 프로세서 구조의 명령어 캐싱에 따르는 효과를 기존 VLIW 프로세서 구조와 비교·분석하였다. VLIW 구조와 슈퍼스칼라 구조의 장점을 살린 SVLIW 구조는 실행시간 내에 LNOP을 삽입하여 자동적으로 자원 충돌이나 자료 종속 문제를 해결할 수 있다. 이러한 SVLIW 구조의 특성 때문에, 동일한 크기의 캐시내에 LNOP이 제거된 명령어들을 갖는 SVLIW 구조는 다수의 LNOP을 함께 포함하는 VLIW 구조보다 캐쉬이용 효율이 높은 특징이 있다. 즉, SVLIW 구조에서의 캐시미스의 발생 빈도는 VLIW 프로세서의 캐시미스의 발생 빈도에 비하여 적으므로 SVLIW 구조에서 어떤 작업을 수행하는 데 소요되는 실행사이클은 VLIW 프로세서에서 같은 작업을 수행하는 데 걸리는 실행사이클보다 더 적게 된다.

본 논문에서는 이것을 확인하기 위하여, VLIW 구조와 SVLIW 구조에서 어떤 작업을 수행할 경우 필요한 실행사이클을 계산하는 모델을 수립하였고, 이를 평가하기 위하여 VLIW 구조와 SVLIW 구조를 모의할 수 있는 시뮬레이션 시스템을 구현하였다. 실험에서는 캐시의 크기와 메모리참조 사이클을 변화시키면서 VLIW 구조와 SVLIW 구조에서 작업을 종료하는 데 걸리는 실행사이클 차이를 측정하였다. 성능평가를 위하여 사용된 벤치마크 프로그램은 2개의 행렬곱셈 프로그램과 2개의 Livermore loop, merge sort, fibonacci, binary search 등의 7 가지이다.

시뮬레이션을 통하여, SVLIW 프로세서가 VLIW 프로세서에 비하여 한단계 늘어난 파이프라인 단계를 갖음에도 불구하고 더 짧은 목적코드를 갖기 때문에 더 적은 캐시미스의 횟수와 더 짧은 평균 메모리참조 사이클을 얻음을 알 수 있었다. 이러한 사실은 어떤 작업을 VLIW 프로세서상에서 수행할 때 요구되는 실행사이클의 수에 비하여 SVLIW 프로세서 상에서 같은 작업을 수행시킬 경우에 필요한 실행사이클의 수가 더 짧아지게 되어 파이프라인 단계가 늘어나는 역효과를 상쇄할 수 있음을 증명하였다. 즉, 캐시미스가 발생하였을 때 메모리참조에 필요한 사이클이 클수록 SVLIW 구조가 VLIW 구조에 비하여 크게 유리한 것을 확인할 수 있었다.

앞으로는 실수연산기를 포함한 경우에 대한 연구를 계속할 예정이며 특히 병렬화 어셈블러를 병렬화 컴파일러로 확장하여 명령어간의 병렬성을 프로그램 전체로부터 추출하였을 때도 동일한 결과를 얻을 수 있는 지에 대한 연구를 계속할 것이다.

참 고 문 헌

[1] Shyh-Kwei Chen, W. Kent Fuchs and Wen-MeiW. Hwn, "An analytical approach to scheduling code for superscalar and VLIW architectures," *Proc. Inter. Conf. Para. Pro.*, pp. I258-I292, 1994.

[2] Soo-Mook Moon, Kemal Ebcioglu, "On Performance and efficiency of VLIW and superscalar," *Proc. Inter. Conf. Para. Pro.*, pp. II 283-287, 1993.

[3] Arthur Abnous, Roni Potasman and Alex Nicolau, "A Percolation based VLIW architecture," *Proc. Inter. Conf. Para. Pro.*, pp. I144-I148, 1991.

[4] Arthur Abnous and Nader Bagherzadeh, "Pipelining and bypassing in a VLIW processor," *Trans. Para. Dist. Sys.*, Vol. 5, No. 6, pp. 658-664, June 1994.

[5] Andren Capitanio, Nikil Dutt and Alexadru Nicolau, "Partitioning of variables for multiple-register-file VLIW architectures," *Proc. Inter. Conf. Para. Pro.*, pp. I298-I301, 1994.

[6] Thomas M. Conte and Sumedh W. Sathaye, "Dynamic Rescheduling: A technique for object code compatibility in VLIW architecture," *Proc.*

28th Inter. Symp. Micro., 1995.

[7] Rau B. R., "Dynamically scheduled VLIW processors," *Proc. 26th Inter. Symp. Micro.*, pp. 80-90, 1993.

[8] Robert P. Colwell, Robert P. Nix, and etc "A VLIW architecture for a trace scheduling compiler," *Proc. Trans. Comp.*, Vol. 37, No. 8, pp. 967-979, August 1988.

[9] Chriss Stephens, Bryce Cogswell, and etc, "Instruction level profiling and evaluation of the IBM RS/6000," *Proc. Inter. Symp. Comp. Arch*, pp. 180-189, 1991.

[10] Joseph A. Fisher, "Trace Scheduling: A technique for global microcode compaction," *Trans. Comp.*, Vol. C-30, No. 7, pp. 478-490, July 1981.

[11] Kai Hwang, *Advanced Computer Architecture*, McGraw-Hill, 1993.

[12] Joseph A. Fisher, "The VLIW Machine: multiprocessor for compiling scientific code," *Proc. Inter. Conf. Para. Pro.*, pp. 45-53, July 1984.

[13] *MIPS R4000 Microprocessor User's* Manual, MIPS Computer Systems, Inc., 1991.

[14] Alexandru Nicolau, Joseph A. Fisher, "Measuring the parallelism available for VLIW architectures," *Trans. Comp.*, Vol. C-33, No. 11, pp. 968-976, Nov. 1984.

[15] Shusuke Okamoto and Masahiro Sowa, "Hybrid Processor based on VLIW and PN-Superscalar," *PDPTA'96 International Conference*, pp. 623-632, 1996.

[16] Boyoun Jeong, Joongnam Jeon and Sukil Kim, "Design of VLIW Architectures minimizing Dynamic Resource Collisions," *Journal of KISS*, Vol. 24, No. 4, pp. 357-368, 1997. 4.

[17] No-Kwang Park, Sung-Hyun Jee and Sukil Kim, An *Efficient Cache Design for Dymanic Instruction Scheduling Processors*, Tech. Rept, CBUCS97-4, Chungbuk National University, 1997. 7.

저 자 소 개

池 承 泫(會員申請中)

1993년 충북대학교 전자계산학과 이학사. 1995년 충북대학교 전자계산학과 이학석사. 1996년~현재 충북대학교 전자계산학과 박사과정. 관심분야는 병렬처리 컴퓨터구조, 병렬처리 알고리즘 등

朴 魯 光(會員申請中)

1997년 한국기술교육대학교 정보통신 공학과 공학사. 1997년~현재 충북대학교 전자계산학과 석사과정. 관심분야는 병렬처리 컴퓨터구조, 병렬처리 알고리즘 등

金 石 日(正會員)

1975년 서울대학교 전기공학과 공학사. 1984년 연세대학교 전자계산학과 공학석사. 1989년 North Carolina Univ. 전기 및 컴퓨터공학 박사. 1975년~1990년 국방과학연구소 연구원, 선임연구원. 1990년~현재 충북대학교 컴퓨터과학과 부교수. 관심분야는 병렬처리 컴퓨터구조, 병렬처리 알고리즘, Supercomputing Advanced factory automation 등