목적 코드에서 LNOP 코드가 제거됨에 따른 SVLIW 구조의 성능 향상

정 보 윤[†] · 전 중 남[††] · 김 석 일[††]

요 약

SVLIW (Superscalar VLIW) 프로세서는 실시간에 긴 명령어를 스케줄하는 VLIW 프로세서의 일종으로 인출되어 실행될 긴 명령어가 사용할 자원과 앞서 인출되어 수행중인 긴 명령어가 사용하는 자원간에 충돌이 발생하면 인출하여 실행하려는 긴 명령어를 수행하지 않고 NOP으로만으로 구성된 긴 명령어(LNOP : Long NOP word)를 할당하여 긴 명령어간의 충돌로 인한 계산의 오류를 피한다. 따라서 SVLIW 프로세서에서는 목적 코드 내에서 LNOP을 제거할 수 있다. 본 논문에서는 목적 코드에서 LNOP이 제거됨에 따라 캐쉬 적중률이 얼마나 향상되는지를 분석하고 이로 인하여 예상되는 성능 향상을 연구하였다. 여러 가지의 벤치 마크 프로그램에 대한 모의 실험 결과, SVLIW 프로세서 구조는 기존의 VLIW 프로세서 구조에 비하여 성능이 5%이상 향상됨이 확인될 수 있었다.

Performance Improvement of SVLIW Architectures by Removing LNOPs from An Object Code

Boyun Jeong[†] · Joong-Nam Jeon[††] · Sukil Kim[††]

ABSTRACT

SVLIW (Superscalar VLIW) processor, a family of VLIW processors schedules very long instruction words at runtime. If a very long instruction word that is to be issued occurs data dependence relations and/or resource conflicts with those words that were under execution, a long NOP word is issued instead of the word until all the data dependence relations and/or resource conflicts have been resolved. Thus, LNOPs can be removed in object codes for SVLIW processors. In this paper, we measure an improvement of the cache hit ratio caused by removing LNOPs in the object code. We also analyze an improvement of the processor performance due to higher cache hit ratio of the processor. Benchmark tests promise that the performance of SVLIW processors is improved more than 5% compared with that of traditional VLIW processors.

1. 서 론

컴퓨터 기술의 발전에 따라 명령어 수준의 병렬성을 다룰 수 있는 프로세서 구조에 대한 연구가 광범위하게 진행되고 있다[1-6]. 그 중에서도 동적 스케쥴링(dynamic scheduling) 기법을 이용하는 슈퍼스칼라 프로세서와 정적 스케쥴링(static scheduling) 기법을 이용하는 VLIW 프로세서가 매우 성공적인 프로세서

※이 연구는 94-97년도 한국과학재단연구비 지원에 의해 수행되었음(과제 번호 : 94-1400-14-3)

† 정 회 원 : (주)한전정보 네트워크 S/W개발실
†† 종신회원 : 충북대학교 컴퓨터과학과 부교수

논문접수 : 1997년 1월 10일, 심사완료 : 1997년 8월 8일

구조로 간주되어 발전되어 왔다. 이들 프로세서는 하나의 프로세서 내에 다수개의 연산 처리기를 포함하고 있어 서로 독립적으로 실행할 수 있는 명령어들을 찾아내어 실행하는 공통된 특징이 있다.

슈퍼스칼라 프로세서[5, 6, 15]의 경우에는 동시에 실행할 수 있는 명령어들을 실시간에 찾아내므로 명령어간의 병렬성을 찾아내고 이를 빠르게 수행하기 위해서는 이들을 실시간에 수행할 복잡한 하드웨어가 필요하다. 그러나 명령어 수준의 병렬성을 프로세서가 추출하여 빠른 수행 방법을 결정할 수 있으므로 목적 코드를 생성하는 비용이 적은 특징이 있다. 이에 반하여 VLIW 프로세서의 경우에는 컴파일러가 명령어간의 자료 종속 관계나 자원 충돌을 방지할 수 있도록 긴 명령어로 구성된 목적 코드를 생성해주므로 프로세서는 단순히 생성된 목적 코드를 순서대로 실행하기만 하면 빠른 계산이 가능하다. 즉, VLIW 프로세서의 하드웨어 구성은 슈퍼스칼라 프로세서에 비하여 비교적 간단한 대신 컴파일러의 성능에 의하여 생성된 목적 코드에 의하여 VLIW 프로세서의 성능이 좌우되므로 성능이 우수한 컴파일러가 필요하다. 이러한 이유로 적은 비용으로 구현이 가능한 VLIW 프로세서를 효과적으로 사용할 수 있는 컴파일러를 개발하기 위한 연구가 광범위하게 수행되어 왔다[9-11].

VLIW 프로세서용 컴파일러는 목적 코드를 생성하면서 명령어간의 자료 종속 관계나 자원 충돌 여부를 분석하여 자료 종속 관계나 자원 충돌이 예상되면 명령어간에 이를 해소할 수 있는 수의 빈 명령어(NOP)로만 구성된 긴 명령어(LNOP)를 삽입하여 실행 시 이들 긴 명령어가 순차적으로 수행되는 과정에서 자료 종속 관계나 자원 충돌로 인한 계산의 착오를 방지하도록 한다. 따라서 VLIW 프로세서용으로 생성된 목적 코드에는 그만큼 NOP이 많이 포함될 수밖에 없다.

목적 코드에서 NOP이 차지하는 비중이 커지면 목적 코드가 메모리에 적재되었을 때, NOP으로 인하여 캐쉬나 메모리의 낭비가 초래된다. 이러한 문제를 방지하기 위하여 긴 명령어 내의 실제 실행되는 명령어들만 메모리에 적재하여 메모리의 낭비를 억제하는 가변 길이 명령어 기법이 개발되었다[4, 9, 12]. 가변 길이 명령어 기법은 목적 코드에서 NOP를 제거하여 메모리에서는 NOP를 포함하지 않도록 하되, 메모리에서 캐쉬로 목적 코드를 적재할 때 본래 상태의 긴 명령어를 구성한다. 따라서, 목적 코드 내에는 NOP이 포함된 원 상태의 긴 명령어로 복원하기 위해 필요한 정보를 저장하도록 한다. 그러나 이 기법을 이용하더라도 목적 코드 내에서는 NOP이 제거될 수 있으나 캐쉬에 적재되는 과정에서 NOP이 복원되므로 캐쉬의 이용률은 목적 코드에서 NOP을 제거하지 않았을 때의 결과와 비교하면 변화가 없다[12].

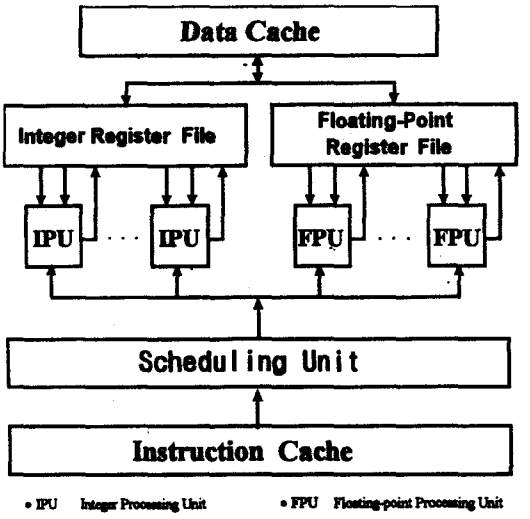
목적 코드에 NOP이 존재하므로 인하여 캐쉬의 이용률이 떨어지는 문제를 해결하기 위한 방법으로 VLIW 프로세서에 동적 스케쥴링 기법을 적용한 SVLIW 프로세서[13]가 본 연구팀에 의하여 제안되었다. SVLIW 프로세서 구조는 한 쌍의 긴 명령어에서 두 번째 긴 명령어를 수행하려고 할 때 먼저 할당되어 실행중인 긴 명령어와 두 번째 명령어간에 자원 충돌이나 자료 종속 관계가 존재하면 두 번째 긴 명령어를 수행하는 대신 LNOP을 수행하도록 하는 스케쥴링 유니트를 포함하고 있다. 따라서 SVLIW 프로세서용 목적 코드에는 VLIW 프로세서용 목적 코드에서와 같이 두 개의 긴 명령어간에 자원 충돌이나 자료 종속관계가 존재할 때 삽입된 LNOP이 불필요하다. 즉, 목적 코드에서 LNOP을 제거할 수 있으므로 SVLIW 프로세서의 캐쉬나 메모리에는 LNOP이 존재하지 않으므로 캐쉬의 이용률을 향상된다. 그러나 SVLIW 프로세서에서는 긴 명령어간의 자원 충돌과 자료 종속 관계를 검사하기 위한 명령어 사이클이 필요하므로 기존의 VLIW 프로세서에 비하여 명령어 사이클이 그만큼 늘어나서 전체적으로 실행 사이클이 늘어나는 단점이 있다. 따라서 SVLIW프로세서는 캐쉬의 이용률이 높아지는 대신 명령어 사이클이 늘어남으로 인하여 전체적으로 성능이 변화하게 된다.

VFPE프로세서[16]의 경우에도 SVLIW 프로세서와 같이 목적 프로그램으로부터 LNOP을 줄일 수 있으나 명령어간의 자료 종속성을 컴파일 시에 검출하여 컴파일러로 하여금 명령어의 실행 순서를 실행시 결정할 수 있도록 자료 종속성 정보를 목적 코드에 포함시켜야 한다. 뿐만 아니라 이 구조는 프로세서를 구성하는 연산기별로 서로 다른 명령어 그룹을 처리할 수 있도록 명확히 구분되어야 하므로 동일한 연산기를 여러개 배치할 수 없는 단점이 있다. 따라서 본

연구에서는 캐시의 영향을 평가하는 연구 범위에 VFPE프로세서를 포함시키지 않았다.

본 논문에서는 여러 가지 벤치마크 프로그램을 SVLIW 프로세서에서 실행시키는 경우에 캐쉬의 이용률의 향상 정도를 측정하고 캐쉬 이용률의 향상이 프로세서의 명령어 사이클의 늘어남으로 인한 영향을 얼마나 감소시키는지를 실험을 통하여 측정하였다.

본 논문의 구성은 다음과 같다. 제 2절에서는 LNOP 명령어를 실시간에 생성하여 스케쥴링 하는 SVLIW 프로세서를 소개하고 SVLIW 구조에서 목적 코드 내에서 LNOP이 제거되는 이유를 설명하였다. 또한, SVLIW 프로세서의 명령어 사이클이 5 사이클로 구성됨을 보였다. 제 3절에서는 SVLIW 프로세서용 컴파일러가 생성하는 목적 코드의 형태를 보였으며, 제 4절에서는 모의 실험 환경을 소개하고, 여러 가지 벤치 마크 프로그램을 시뮬레이터에서 수행시킨 결과를 가지고 VLIW 프로세서에 대한 SVLIW 프로세서의 성능 향상을 측정하였다. 마지막으로 제 5절에서는 본 논문의 결론과 향후 연구 방향을 기술하였다.



(그림 1) SVLIW 프로세서 구조
(Fig. 1) SVLIW processor architecture

2. SVLIW 프로세서

SVLIW 프로세서[13]는 그림 1과 같이 명령어 캐쉬로부터 긴 명령어를 인출하여 스케쥴러의 도움을 받아 연산 처리기에 긴 명령어를 구성하는 단위 명령어를 동시에 할당하여 연산을 개시한다. 이때 연산에 필요한 연산자는 레지스터 파일로부터 제공된다. 스케쥴링 유니트는 명령어 캐쉬로부터 인출된 긴 명령어를 분석하여 단위 명령어들이 요구하는 레지스터간의 자료 의존 관계나 파이프라인 실행 단계에서 자원의 충돌 여부를 검사하여 긴 명령어를 연산 처리기로 제공하여 실행하거나 빈 명령어를 수행하도록 한다. 따라서 SVLIW 프로세서의 명령어 사이클은 기존의 VLIW 프로세서에서 필요한 인출(Fetch), 분석(Decode), 실행(Execute), 쓰기(WriteBack)의 네 단계와 명령어를 스케쥴링 하는 단계로 구성될 수 있다.

명령어를 스케쥴링 하기 위해서는 우선 긴 명령어를 실행하기 위하여 필요한 레지스터가 준비되었는지를 알아야 한다. 또한 필요한 모든 레지스터들이 사용할 수 있다고 하더라도 현재 수행되고 있는 명령어가 사용하는 자원과 사용할 자원이 충돌을 일으키고 있지 않아야 한다. 따라서 스케쥴링 단계에서는 스케쥴링 하려는 긴 명령어가 사용할 레지스터의 자료 종속 관계를 검사하고 이용할 자원의 충돌을 검사하는 두 가지 절차를 수행하여야 한다.

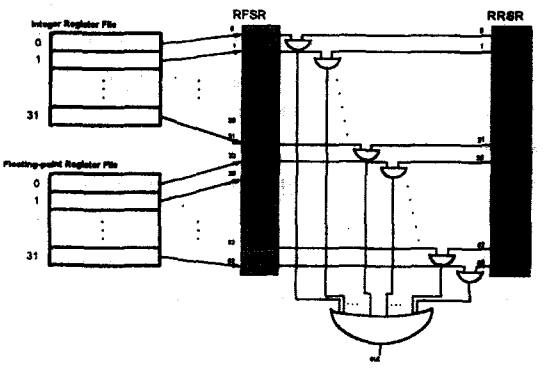
2.1 자료 의존성 분석

명령어를 스케쥴링 하기 위해서는 우선 긴 명령어를 실행하기 위하여 필요한 레지스터가 준비되었는지를 검사하기 위해서는 긴 명령어간의 레지스터별 자료 종속 관계 검사 장치가 필요하다. 또한, 각각의 연산 처리기를 구성하는 파이프라인에서의 충돌 여부를 검사하여야 한다. 검사 결과, 모든 레지스터의 사용이 가능하고 아무런 파이프라인 충돌이 발생하지 않는다면 긴 명령어를 구성하는 단위 명령어들을 해당 실행 유니트로 할당하여 수행을 시작토록 한다. 만일 필요한 레지스터가 준비되지 않았거나 파이프라인 충돌이 예상되면 긴 명령어 대신 LNOP를 스케줄하고 다음 사이클에서 다시 스케줄 가능 여부를 검사한다. 따라서 스케쥴링 유니트는 레지스터간의 자료 의존 관계를 분석하는 장치와 자원의 충돌을 검사하는 장치를 필요로 한다.

인출된 긴 명령어를 연산 처리기에서 실행을 시작하기 위해서는 긴 명령어가 필요로 하는 모든 레지스

터가 수행중인 명령어들에 의하여 사용되고 있지 않아야 한다. 이러한 과정은 레지스터 파일을 구성하는 모든 레지스터를 빠른 시간 내에 검사할 수 있어야 하므로 레지스터 별로 사용 여부를 알려주는 태그 비트(tag bit)를 붙이고 이 태그 비트를 이용하여 자료 의존성을 검사할 수 있다.

SVLIW 프로세서에서도 그림 2와 같이 레지스터 파일을 구성하는 각각의 레지스터에 대응하는 비트로 구성된 레지스터 파일 상태 레지스터 (RFSR: Register File Status Register)와 긴 명령어를 실행시키기 위하여 필요로 하는 레지스터에 대응하는 레지스터 요구 상태 레지스터(RRSR: Register Request Status Register)의 상태를 비교하여 사용중인 레지스터를 다시 요구하는지를 판단한다. 스케쥴링 유니트는 그림 2에 보인 바와 같이 RFSR과 RRSR을 비트별 AND 연산을 동시에 수행하고, 그 결과를 OR하여 만일 그 결과가 1이면 자료 종속 관계가 존재하므로 이번 사이클에서는 실행 유니트가 LNOP을 수행하도록 하고, 다음 사이클에서 다시 갱신된 RFSR과 RRSR을 이용하여 이 과정을 반복한다. 여기서 SVLIW프로세서를 구성하는 정수 및 실수용 레지스터 파일의 길이가 각각 32 단어이므로 RFSR과 RRSR이 각각 32 비트로 구성된다.

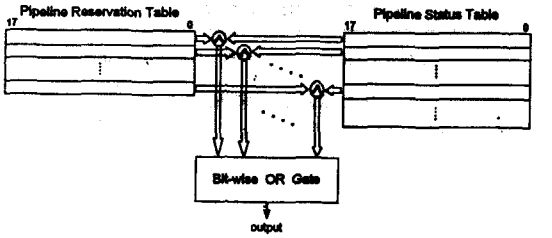


(그림 2) 긴 명령어간의 레지스터별 자료 종속 관계 검사
(Fig. 2) Data dependence check logic between a pair of VLIWs

2.2 자원 충돌 검사

스케쥴링 유니트가 수행하는 두 번째 검사는 실행시키려는 긴 명령어 내의 실수 명령어와 이미 실행 유니트에 할당되어 실행중인 실수 명령어들이 사용하는 파이프라인 자원의 충돌이 발생할지도 모르는 가능성을 검사하는 것이다. 정수 명령어의 경우에는 매 사이클마다 하나의 명령어가 실행될 수 있으므로 정수 처리기용 파이프라인에서는 충돌이 일어나지 않는다. 그러나 Mips 프로세서[14]에서 볼 수 있듯이 실수 처리기는 비정형적인 파이프라인 단계로 구성되므로 매 사이클마다 하나의 명령어를 할당하게되면 파이프라인에서의 충돌이 발생한다.

본 논문에서 고려하는 SVLIW 프로세서의 실수 처리기는 파이프라인 단계를 비정형적으로 사용하는 것으로 간주하였기 때문에 연속된 실수 명령어의 실행은 파이프라인 단계에서의 충돌을 유발할 수 있다. 따라서 각각의 실수 처리기별로 파이프라인에서의 충돌을 검사하기 위하여 그림 3과 같이 각각의 실수 처리기별로 현재 사용중인 파이프라인 단계를 나타내는 PST(Pipeline Status Table)와 실수 처리기에 할당할 명령어가 사용할 파이프라인 단계 예약표인 PRT (Pipeline Reservation Table)를 비교하여 모든 실수 처리기에서 충돌이 발생하지 않을 때에만 긴 명령어를 실행하도록 허용하였다. 여기서 두 개의 실수 처리기로 프로세서를 구성하고 각 실수 처리기의 파이프라인 단계를 9 단계로 구성한다면 PRT와 PST의 필드는 그림 3과 같이 각각 18비트이며 그 길이는 각 명령어의 파이프라인 사이클 길이와 같다. 따라서 PST는 가장 긴 파이프라인 사이클 길이의 목록을 유지한다. 그림 3에서 논리회로는 PST의 각 행과 PRT의 각 행이 동시에 AND 연산을 수행하고, 그 결과를 OR 연산하여 총 18비트의 연산 결과를 얻고 그 결과를 18비트 논리합 연산을 통하여 PRT와 PST의 충돌 여부를 검사한다. 스케쥴링 유니트는 PRT와 PST를

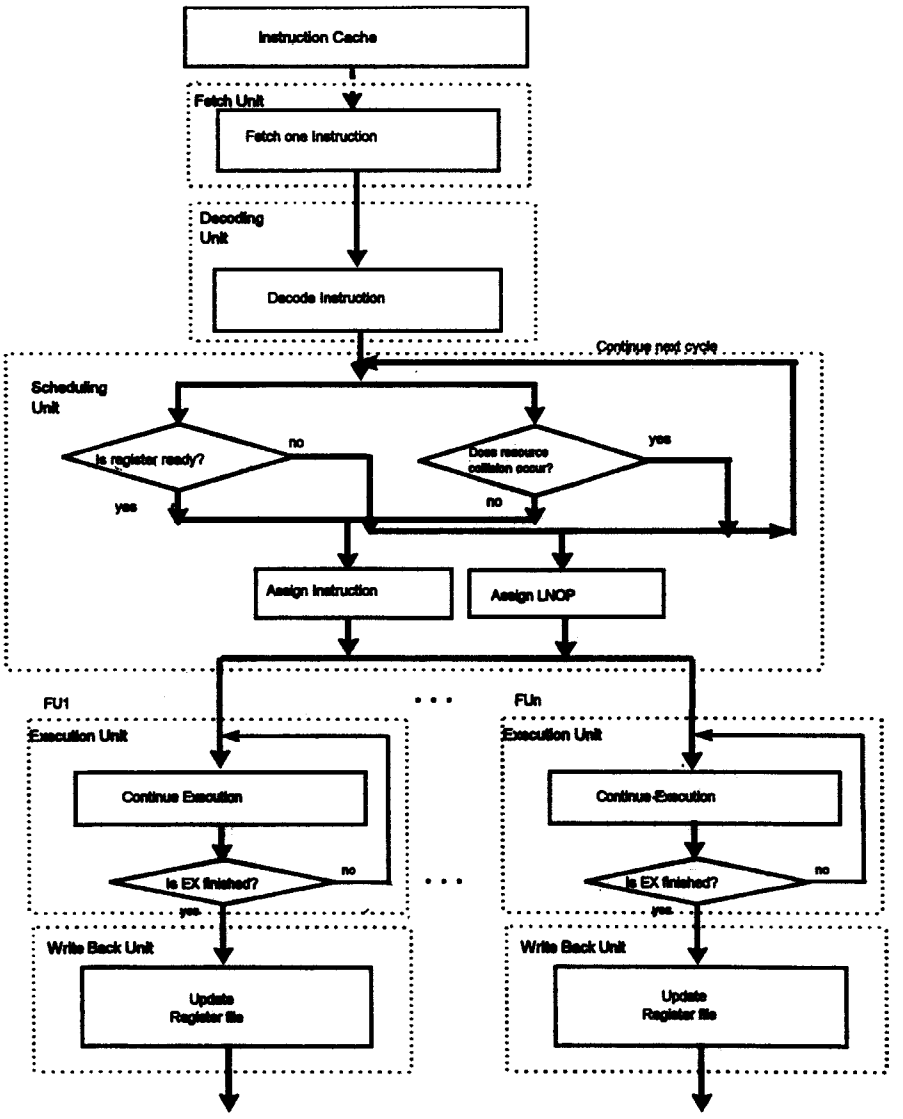


(그림 3) 파이프라인 충돌을 검사하는 논리회로
(Fig. 3) Logic for pipeline collision check

비교하여 충돌이 발생하지 않을 때에는 긴 명령어를 할당하고 충돌 시에는 LNOP을 할당한다.

2.3 명령어 실행 사이클

긴 명령어를 스케줄링 하려는 시점에서 필요로 하는 레지스터가 가용한지 여부를 검사하는 과정과 각 연산 처리기의 파이프라인에서의 충돌 가능성 여부는 연속성이 없으므로 병행적으로 검사될 수 있다. 또한, 긴 명령어 인출 단계에서 명령어 캐쉬로부터 컴파일러가 생성한 긴 명령어를 가져오고, 분석 단계에서 긴 명령어를 구성하는 각 단위 명령어들을 분석하고 각 단위 명령어들이 필요로 하는 비정형적인 파



(그림 4) SVLIW 프로세서의 명령어 실행 단계

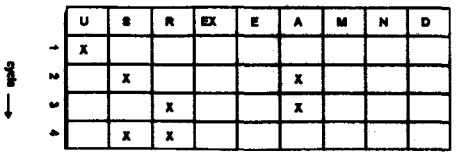
(Fig. 4) Pipeline stages on an SVLIW processor architecture

이프라인 단계나 사용할 레지스터에 관한 정보를 추출하도록 한다면 충분히 하나의 명령어 사이클에 자원 충돌 여부와 레지스터간의 자료 종속 관계를 확인할 수 있으므로 그림 4와 같이 SVLIW 프로세서의 명령어 실행 사이클을 구성할 수 있다.

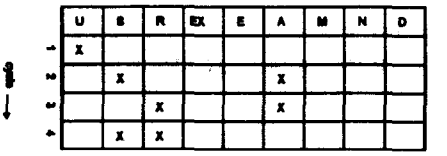
긴 명령어의 스케쥴링 과정이 종료되고 긴 명령어가 해당 연산 처리기로 할당되면 연산 처리기의 파이프라인 단계가 모두 종료되기 전에 할당된 명령어의 수행을 개시할 수 있다. 만일 종속관계나 자원 충돌이 발생하면 LNOP을 각 연산 처리기로 할당하여 다음 사이클에 긴 명령어의 할당 가능성을 다시 검사한다. 따라서 SVLIW 프로세서에서는 자원 충돌이나 자료 종속관계가 존재하는 긴 명령어 사이에서도 LNOP를 제거할 수 있으므로 목적 코드에서 LNOP을 제거할 수 있으며, 이로 인하여 캐쉬에 LNOP이 적재되는 것을 방지하여 캐쉬의 이용 효율을 높일 수 있게 된다.

3. SVLIW 프로세서용 코드 생성

SVLIW 프로세서용 목적 코드는 기존의 VLIW 프로세서용 컴파일러와 그 구조가 동일하며, 기존의 VLIW 프로세서용으로 생성된 목적 코드를 SVLIW 프로세서에서도 그대로 사용할 수 있다. 즉, VLIW 프로세서용으로 생성된 목적 코드에는 SVLIW의 스케쥴링 유니트가 자체적으로 발생시킬 LNOP이 이미 포함되어 있으므로 SVLIW 프로세서에서는 목적 코드의 긴 명령어를 절차에 따라 수행하기만 하면 된다. 비록 VLIW프로세서용으로 생성된 목적 코드가 SVLIW 프로세서용으로 사용이 가능하다고 해도 많은 LNOP이 포함되어 있으므로 인하여 SVLIW프로세서의 특징인 캐쉬의 효율을 높이지 못한다. 따라서 SVLIW 프로세서를 위해서는 목적 코드에서 LNOP을 제거해야 한다.



(a) ADD, SUBTRACT PRT



(b) CVT, ROUND PRT

(그림 5) 파이프라인 예약표
(Fig. 5) Pipeline Reservation Table

예를 들어, ADD, SUB, CONVERT 및 ROUND 명령어의 파이프라인 예약표가 각각 그림 5와 같다고 하고 실수 명령어 ADD와 CONVERT가 동시에 수행될 수 있는 첫 번째 긴 명령어이고, SUB와 ROUND가 곧 이어서 수행되는 한 쌍의 실수 명령어라고 하면, 두 번째 긴 명령어는 파이프라인의 충돌을 야기하므로 그림 6(a)와 같이 2개의 명령어 사이클이 지난 이후에 수행을 시작할 수 있다.

1	ADD	CONVERT
2	NOP	NOP
3	NOP	NOP
4	SUB	ROUND

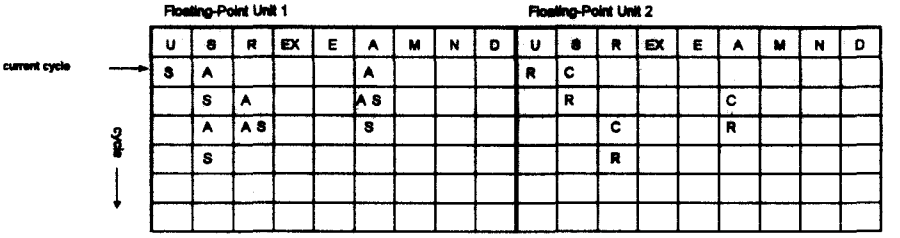
(a) VLIW용 목적 코드

1	ADD	CONVERT
2	SUB	ROUND

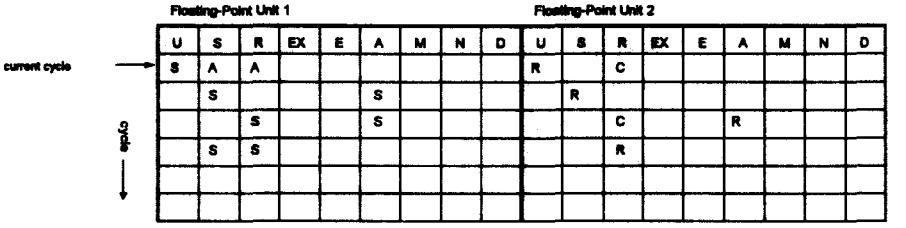
(b) SVLIW용 목적 코드

(그림 6) 프로세서 구조별 목적 코드
(Fig. 6) Object codes for each processor

SVLIW 프로세서용 코드는 그림 6(b)와 같이 두 개의 LNOP가 제거된다. 그림 6(b)의 목적 코드를 SVLIW 프로세서에서 실행시키는 경우, 우선 ADD와 CONVERT가 포함된 첫 번째 긴 명령어를 실행하고, 다음 사이클에서 SUB와 ROUND가 포함된 긴 명령어를 수행할 준비를 한다. 이때, 스케쥴링 유니트는 그림 7(a)에서 알 수 있듯이 PST의 네 번째 행과 PRT의 세 번째 행의 파이프라인 R 단계에서 충돌이 발생하므



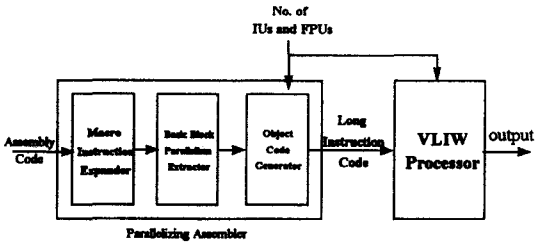
(a) Collision Case in PST



(b) No Collision Case in PST

(그림 7) 파이프라인 단계의 충돌 검사
(Fig. 7) Collision check throughout pipeline stages

로 스케쥴링 유니트는 LNOP을 실행 단계로 진입시켜 사이클을 진행시키고, 다음 사이클에서 긴 명령어를 시작할 수 있는지를 검사한다. 다음 사이클에서는 PST가 한 행씩 쉬프트 된다. 그러나 PST와 비교하면 마찬가지로 파이프라인 단계에서 충돌이 일어나므로 다시 한번 LNOP을 실행 단계로 제공하고 다음 사이클을 기다린다. 세 번째 사이클에서 비로소 스케쥴링 유니트는 그림 7(b)에 보인바와 같이 충돌이 발생하지 않으므로 긴 명령어를 실행 단계로 제공하여 비로소 실행이 시작된다. 따라서 SVLIW 프로세서에서는 LNOP을 목적 코드에서 제거하여도 수행 결과가 제거하지 않았을 때와 동일한 결과를 얻을 수 있다.



(그림 8) 시뮬레이션 시스템
(Fig. 8) Simulation system

4. 실험 및 고찰

4.1 실험 환경

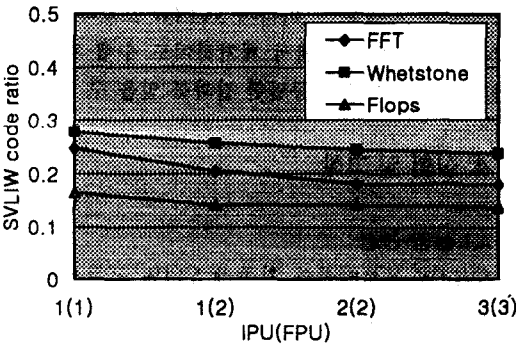
SVLIW 프로세서와 기존의 VLIW 프로세서의 성능을 비교하기 위하여 여러 가지 프로세서를 모의할 수 있는 시뮬레이션 시스템을 UNIX 환경의 SUN SPARC 10에서 C++로 구현하였다. 구현한 시스템은 그림 8과 같이 긴 명령어를 구성하는 병렬화 어셈블러(parallelizing assembler)와 여러 가지 프로세서 구조를 모의하는 프로세서 부로 구성된다.

병렬화 어셈블러는 어셈블리 프로그램을 읽어들여 기본 블록을 추출하고 기본 블록 내의 병렬성을 분석하여 각각의 구조에 적합한 긴 명령어로 구성된 목적 코드를 발생한다. 프로세서 부는 프로세서의 형식, 프로세서를 구성하는 연산 처리기의 종류와 수를 입력받아 원하는 구조의 프로세서를 구성할 수 있다. 프

로세서 부는 정수 처리기, 실수 처리기, 명령어 캐쉬, 데이터 캐쉬, 정수형 레지스터 파일 및 실수형 레지스터 파일 등 6개의 모듈과 긴 명령어간의 자원 충돌이나 자료 종속 관계를 검사하는 스케쥴링 유니트로 구성된다.

4.2 목적 코드 길이 비교

SVLIW 프로세서용 목적 코드는 기존의 VLIW 프로세서용 목적 코드에 비하여 LNOP이 제거되므로 목적 코드의 길이가 감소된다. 따라서 우선 SVLIW와 기존의 VLIW 프로세서 구조별로 각각의 구조를 위한 병렬화 어셈블러가 생성한 목적 코드의 길이를 비교하여 보았다. 이 실험을 위해서는 디지털 신호처리용으로 사용되는 FFT, Flops 및 Whetstone 등의 세 가지 벤치 마크 프로그램을 선정하였다. 이들 벤치 마크 프로그램은 실수 및 정수 연산이 충분히 포함된 과학 계산용 프로그램으로 정수 유니트와 실수 유니트로 구성된 SVLIW 프로세서를 구성하는 유니트들을 충분히 이용할 수 있다.



(그림 9) 목적 코드의 길이 비교
(Fig. 9) Length of two object codes

그림 9는 각각의 프로세서를 구성하는 정수 처리기(IPU)와 실수 처리기(FPU)의 수를 변화시켰을 때 생성되는 목적 코드의 길이를 비교한 것이다. 그림 9에서 가로축의 1(1), 1(2), 2(2) 및 3(3)은 정수 처리기가 각각 1, 1, 2, 3개이며, 실수 처리기가 각각 1, 2, 2, 3개인 경우에 기존의 VLIW 프로세서용 목적 코드의 길이를 기준으로 SVLIW 프로세서용으로 생성된 목적 코드의 비율을 측정한 것이다. 그림 9에서 그 비율은 정수 및 실수 처리기가 각각 한 개씩 일 때, 14%내지 29%수준이며, VLIW 프로세서용 목적 코드 내에서 LNOP이 차지하는 비율이 71%내지 86%에 달하는 것을 보여준다. 세 가지 벤치 마크 중에서도 Flops의 경우에 그 비율이 14%로 가장 작았다. 그 이유는 Flops 벤치 마크의 경우에 실수 연산이 차지하는 비율이 높아 자원의 빈번한 충돌이 발생하여 VLIW 프로세서용 목적 코드 내에 LNOP이 차지하는 비율이 크기 때문으로 분석된다.

또한, 프로세서를 구성하는 정수 및 실수 처리기의 수가 증가하여도 목적 코드의 길이가 급격히 변화하지 않는 것을 알 수 있었다. 그 이유는 프로세서를 구성하는 정수 및 실수 처리기의 수가 늘어나면 하나의 긴 명령어에 포함되는 단위 명령어가 늘어나므로 전체적으로 목적 코드의 길이가 짧아져야 하나, 본 논문에서 사용한 벤치 마크 프로그램 내에 자료 종속 관계나 자원의 충돌이 빈번하게 발생하기 때문에 긴 명령어를 구성할 명령어 수준의 병렬성이 낮기 때문인 것으로 판단된다. 또한, 본 실험에서 사용한 명령어 수준의 병렬성을 추출하는 역할을 하는 병렬화 어셈블러의 기능이 기본 블록(basic block) 내에서의 병렬성 추출로 한정되어 있기 때문인 것으로 분석된다.

4.3 캐쉬 적중률 비교

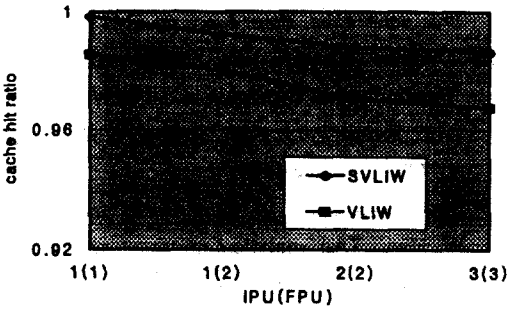
그림 10은 명령어 캐쉬의 크기를 4K 바이트로 설정하였을 때, 세 가지 벤치 마크 별로 프로세서 구조별로 목적 코드를 실행시킬 때 캐쉬의 적중률을 비교한 것이다. 그림에서 알 수 있듯이 SVLIW 프로세서에서의 캐쉬 적중률이 기존의 VLIW 프로세서에서의 캐쉬 적중률보다 향상된 것을 알 수 있다. 그 이유는 SVLIW 프로세서용 목적 코드 내에 LNOP이 제거되었기 때문에 목적 코드 내에서 LNOP이 차지하는 영역을 줄여서 일정한 크기의 캐쉬 블록 내에 저장되는 유효한 코드의 길이가 VLIW 프로세서용으로 생성된 목적 코드에 비하여 길어서 프로그램의 실행에 따라 코드가 메모리와 캐쉬간에 이동하는 횟수를 줄일 수 있기 때문이다. 또한, 그림 10에서는 프로세서를 구성하는 연산 처리기의 수가 증가함에 따라 캐쉬 적중률이 떨어지는 경향을 보이는데, 이는 벤치 마크 프로그램 내에 명령어 수준의 병렬성이 떨어져 하나의 긴 명령어를 구성할 때 빈 명령어가 많이 삽입되기 때문

이다.

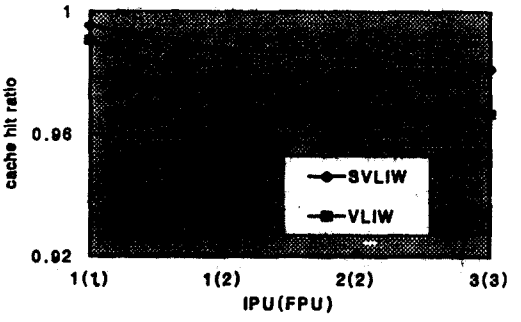
캐쉬의 적중률이 증가함에 따라 캐쉬 내의 블록들에 대한 반복 사용이 증가하므로 메모리로부터 캐쉬로의 블록 이동 횟수가 적어진다. 따라서, 메모리로부터 캐쉬로의 블록 대체에 소요되는 지연 시간이 줄어들어 전체 실행 시간을 단축할 수 있다.

4.4 실행 시간 비교

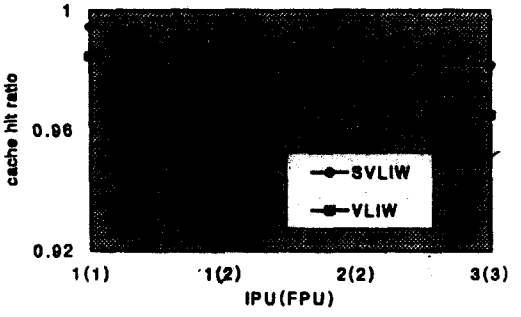
그림 11은 각 프로세서용으로 생성된 목적 코드를 해당하는 프로세서에서 실행시킬 때, 수행된 명령어 사이클을 측정한 것이다. 여기서 캐쉬 적중이 되지 않았을 때 다시 메모리를 참조하는데 걸리는 시간이 4 사이클이라고 가정하였다. 실험 결과, 벤치 마크 프



(a) FFT

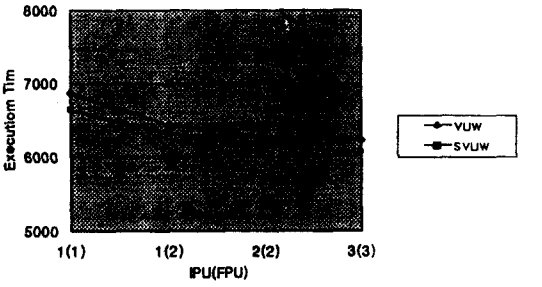


(b) Flops

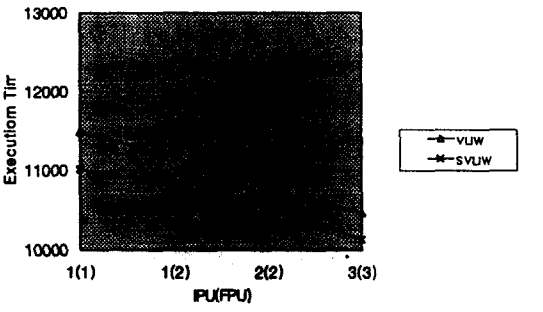


(c) Whetstone

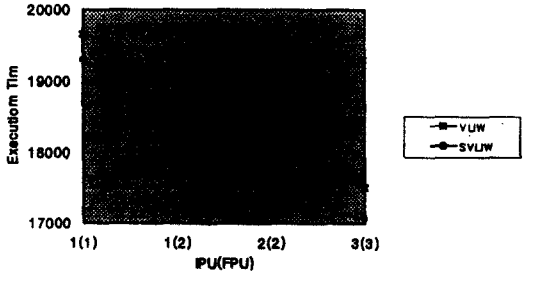
(그림 10) 캐쉬 적중률의 비교
(Fig. 10) Cache hit ratio



(a) FFT



(b) Flops



(c) Whetstone

(그림 11) 벤치 마크별 실행 시간
(Fig. 11) Execution times for benchmark programs

로그램의 길이가 길수록 기존의 VLIW 프로세서에 비해 SVLIW 프로세서에서의 실행 시간이 현저히 줄어드는 것을 알 수 있는데, 이는 프로그램의 길이가 길수록 캐쉬 적중이 되지 않아 캐쉬 블록의 대체 횟수가 상대적으로 늘어나기 때문인 것으로 분석된다.

5. 결 론

본 논문에서는 명령어 수준의 병렬성을 다루는 SVLIW 프로세서에서 성능을 벤치 마크 프로그램을 이용하여 측정하였다. SVLIW 프로세서는 기존의 VLIW 프로세서에 실시간 명령어 스케쥴링 유니트를 추가한 VLIW 프로세서로 기존의 VLIW 프로세서용으로 생성된 목적 코드에서 볼 수 있는 자원 충돌 혹은 레지스터간의 자료 종속 관계가 존재하는 긴 명령어간에 삽입된 NOP로 채워진 긴 명령어(LNOP)를 제거할 수 있는 장점이 있다. 즉, SVLIW 프로세서의 스케쥴링 유니트는 인출된 긴 명령어들을 실행하기 전에 이미 실행중인 긴 명령어들과 자료 종속 관계나 자원 충돌을 일으키는지 미리 검사하여 어떤 충돌이 발생하게 되면 이것이 해소될 때까지 LNOP을 실시간에 생성하여 스케줄한다. 따라서 SVLIW 프로세서는 기존의 VLIW 프로세서에 비해 스케쥴링 유니트의 추가로 인한 구조의 복잡도가 높아지지만, 매 사이클마다 긴 명령어의 실행 가능성을 타진하기 때문에 긴 명령어를 생성하는 컴파일러의 예측 정확성에 대한 부담을 줄일 수 있다.

목적 코드로부터 LNOP을 제거하게되면 결국 일정한 크기의 캐쉬 내에 존재하는 실행 명령어가 차지하는 비율이 높아져 캐쉬 적중률을 높이며, 결국은 프로그램의 실행에 따른 캐쉬 내의 블록 대체 횟수를 줄여 메모리로부터 캐쉬로의 블록 이동에 소요되는 횟수를 감소시켜 전체적으로 프로그램의 실행 시간이 감소됨을 보여주었다. 이것으로부터 SVLIW 프로세서 구조는 기존의 VLIW 프로세서 구조에 실시간 명령어 스케쥴링 유니트를 포함시키므로 인하여 비록 명령어 실행 사이클은 늘어났으나 목적 코드 내에서 LNOP을 제거함으로 인하여 성능 향상을 이룩할 수 있는 구조임을 알 수 있었다.

앞으로는 SVLIW 프로세서용 컴파일러를 구현하여 프로그램 전체의 병렬성을 추출하는 경우에도 본 논문에서와 같이 기본 블록 내에서만 병렬성을 추출한 경우에 비하여 성능이 향상될 수 있는지에 대한 연구를 계속할 것이다.

참 고 문 헌

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

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

[3] Shyh-Kwei Chen, W. Kent Fuchs and Wen-Mei W. Hwu, "An analytical approach to scheduling code for superscalar and VLIW architectures," *Proc. Inter. Conf. Para. Proc.*, pp. I285-I292, 1994.

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

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

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

[7] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journ. Research Development*, Vol. 11, pp. 25-33, Jan. 1967.

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

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

[10] Monica Lam, Software Pipelining: *An effective scheduling technique for VLIW*, PhD thesis, Carnegie Mellon, May 1987.

[11] Albert Y. Zomaya, Selim G. Akl, and etc., *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996.

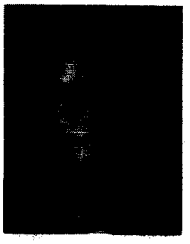
[12] Thomas M. Conte and Sumedh W. Sathaye, "Dynamic Rescheduling: A technique for object code compatibility in VLIW architecture", *Proc. 28th Inter. Symp. Micro.*, 1995.

[13] Boyoun Jeong, Joongnam Jeon and Sukil Kim, "A design of a VLIW architecture minimizing dynamic resource collisions," *Journ. KISS*, Vol. 24, No. 4, pp. 357-368, 1997. 4.

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

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

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



정 보 윤

1995년 충북대학교 컴퓨터과학과 학사

1997년 충북대학교 전자계산학과 석사

1997년~현재 (주)한전정보 네트워크 S/W개발실 근무.

관심분야: 병렬처리 컴퓨터 구조, 병렬 처리 알고리즘, 시스템 통합 프로그래밍 등임

전 중 남

1981년 연세대학교 전자공학과 학사.

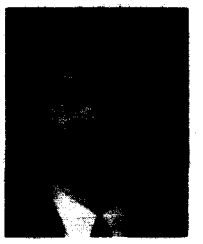
1985년 연세대학교 전자공학과 석사.

1990년 연세대학교 전자공학과 박사.

1990년~현재 충북대학교 컴퓨터과학과 부교수

1996년~현재 미국 Texas A&M 방문교수

관심분야: 컴퓨터 구조, 병렬 처리 알고리즘, 공장자동화 등임



김 석 일

1975년 서울대학교 전기공학과 학사학위 취득

1975년~1990년 국방과학연구소 선임연구원으로 근무

1985년~1989년 미국 North Carolina State University 에서 공학박사 취득

1990년~현재 충북대학교 컴퓨터과학과 부교수로 재직중.

관심분야: 병렬처리 컴퓨터 구조, 슈퍼컴퓨팅, 병렬처리언어, 이기종분산처리, 시각장애인 사용자 인터페이스 등임