ILP 프로세서를 위한 성능측정 및 평가 시스템

이 상 정†

요 약

본 논문에서는 한 사이클에 여러 개의 명령들이 다중 이슈되어 명령어 수준에서 병렬처리되는 ILP 프로세서의 성능을 측정하고 평가하는 시스템을 개발한다. 개발되는 시스템은 C 컴파일러와 시뮬레이터로 구성된다. C 컴파일러는 C 소스 프로그램을 입력으로 받아 3-주소 코드형태의 중간언어를 생성한다. 생성된 중간언어는 ILP 프로세서의 환경 파라미터와 함께 시뮬레이터에 입력되어 시뮬레이션된 후 메모리 내용, 수행된 클럭 수 및 명령 트레이스, 수행된 명령들의 동적 빈도수, 분기명령의 예측률, profiling 정보 등을 생성한다. 개발된 성능측정 시스템의 동작 검증을 위하여 순차이슈 되어 정적으로 스케쥴링 되는 조건실행 방식의 성능과 분기처리 방식의 성능을 측정하여 분석한다.

A Performance Measurement and Evaluation System for ILP Processors

Sang-Jeong Lee†

ABSTRACT

In this paper, a performance measurement and evaluation system for ILP(Instruction Level Parallelism) processors which issue multiple instructions and execute them in parallel is developed. The system consists of a C compiler and a simulator. The compiler takes C source programs as an input and generates 3-address style intermediate code. Then the simulator accepts the intermediate code and simulates it. The results of simulation are the contents of memory before and after simulation, the number of executed clocks, the trace and the dynamic count of executed instructions, the prediction hit ratio and profiling information for each branch instruction.

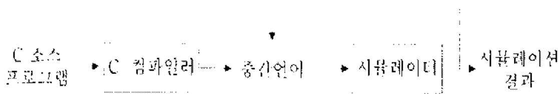
To verify and understand the behavior of the system, the performance of predicated execution and one of branch schemes is measured and its results are analyzed.

1. 서 론

최근 한 사이클에 여러 개의 명령을 이슈(multiple instructions issue)하여 명령어를 병렬 실행함으로써 명령어 수준에서 병렬성(Instruction Level Parallelism, ILP)을 이용하고 성능을 향상시키는 고성능 ILP 프로세서들이 개발되고 있다. ILP를 이용한 대표적인 아키텍처 방식으로는 슈퍼스칼라(superscalar), 슈퍼파이프라인(superpipeline), VLIW(Very Long Instruction Word) 아키텍처 방식이 있다[1-4]. 이들 아키텍처는 다중 이슈되는 명령어의 병렬처리를 위해 컴파일러가 정적으로 스케쥴(static schedule)하거나 또는 하드웨어 지원에 의해 동적 스케쥴(dynamic schedule) 된다. 명령 스케쥴의 효과는 이슈되는 명령어의 수와 분기명령의 처리 방식 등의 환경에 크게 좌우된다. 따라서 성능 대 가격 비를 극대화 시키면서 사용

※ 이 논문은 1997년도 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었음.
† 정 회 원 : 순천향대학교 공과대학 컴퓨터학부
논문접수 : 1998년 3월 12일, 심사완료 : 1998년 6월 25일

따라 응용목적에 부응하는 ILP 프로세서의 설계 사양을 결정하기 위해서는 컴파일러와 하드웨어의 다양한 설계 선택사항에 따라 성능을 측정하고 평가하는 작업이 필수적이다.



(그림 1) ILP 프로세서 성능측정 및 평가 시스템의 구성
(Fig. 1) Configuration of a Performance Measurement and Evaluation System for ILP processors

본 논문에서는 순차이슈(in-order issue)되고 컴파일러에 의해 정적으로 스케쥴되어 명령어 수준에서 병렬처리되는 ILP 프로세서의 성능을 측정하고 평가하는 시스템을 개발하고 제안한다. 개발된 시스템은 (그림 1)과 같이 크게 C 컴파일러와 시뮬레이터로 구성된다. 컴파일러는 성능을 테스트하기 위한 C 벤치마크 프로그램을 입력으로 받아 3-주소 코드형태의 중간언어를 생성한다. 생성된 중간언어는 설계 사양의 파라미터와 함께 시뮬레이터에 입력되어 시뮬레이션된 후 메모리 내용, 수행된 클럭 수 및 명령 트레이스,수행된 명령들의 동적 빈도 수, 분기명령의 예측율, 프로파일링(profiling) 정보 등을 생성한다. 시뮬레이션 결과에 따라 생성된 정보는 중간언어에 전달되어 수정할 수 있고, 다시 시뮬레이션을 수행하여 성능을 측정하고 분석할 수 있다. 개발된 성능 측정 시스템은 ILP 프로세서와 컴파일러를 설계할 때 명령어 수준의 상위 레벨에서 설계 사양을 선택하고 성능을 검증하는 수단으로 사용될 수 있다. 개발된 성능측정 시스템의 검증을 위하여 순차이슈되어 정적 스케쥴링되는 조건실행(predicated execution) 방식[14,21]과 분기처리 방식[18,20]의 성능을 측정하여 분석함으로써 타당성을 보인다.

2. 연구배경

이 장에서는 제안된 성능측정 및 평가 시스템을 기술하기에 앞서 ILP 프로세서의 기본 개념을 소개하고 기존의 ILP 프로세서 성능측정 시스템에 관한 연구를 기술한다.

2.1 ILP 프로세서

오늘날 대부분의 마이크로프로세서는 명령어의 수행 단계가 세분화되고, 한 명령의 수행이 끝나기 전에 다른 명령이 중첩실행 되도록 명령어들이 파이프라인 처리된다. 최근에는 단일명령 이슈 파이프라인 처리를 한층 발전시켜서 한 클럭 사이클에 여러 개의 명령을 이슈하고 명령어를 병렬 실행하여 명령어 수준에서 병렬성을 이용함으로써 성능을 향상시키는 고성능 프로세서들이 개발되고 있다. ILP를 이용한 대표적인 아키텍처 방식으로는 슈퍼스칼라, 슈퍼파이프라인, VLIW 아키텍처 방식이 있다. 이들 ILP 프로세서에서는 데이터 종속관계, 분기명령 및 프로세서의 구조적인 문제로 발생하는 파이프라인 해저드로 인해 지속적인 명령의 이슈와 파이프라인 처리가 방해되어 성능이 저하된다. 특히 분기명령은 대부분의 프로그램에서 명령어의 많은 부분을 구성하고 있기 때문에 프로세서의 성능을 저하시키는 주된 요인이 된다. 이와 같은 파이프라인 분기손실은 다중 이슈되는 명령의 수와 파이프라인 깊이가 커지는 추세에 있는 ILP 프로세서에서는 더욱 심각하기 때문에 대부분의 ILP 프로세서들이 이들 분기손실을 줄여서 성능저하를 최소화시키려는 시도를 하고 있다[10-14]. ILP 프로세서에서 분기명령으로 인한 성능의 제한을 극복하기 위한 방식으로 투기적 실행(speculative execution)과 조건실행(predicated execution) 방식 등이 있다. 투기적 실행[12]은 분기조건의 결과가 알려지기 전에 미리 수행하는 방식이고, 조건실행[14]은 보호실행(guarded execution)으로도 불리우며 조건분기 명령을 제거하고 이 분기명령과 제어 종속관계가 있는 분기 이후의 명령들에 대해 어떤 조건에 따라 조건적으로 실행되는 명령으로 변환하여 제어 종속관계를 데이터 종속관계로 바꾸어 주는 기법이다.

본 논문에서는 조건실행 및 분기처리 방식의 성능측정에 개발된 성능측정 및 평가 시스템을 적용하고 동작을 검증한다.

2.2 ILP 성능측정 시스템 관련 연구

본 논문의 ILP 프로세서의 성능측정 시스템과 관련된 연구로는 Illinois 대학의 IMPACT 컴파일러 시스템[14], HP 사의 HPL Pladoh 아키텍처[15], Wisconsin 대학의 SimpleScalar Tool Set[8,9] 등이 있다.

IMPACT 컴파일러 시스템은 ILP 프로세서에서 명령어 수준 병렬성을 이용하기 위한 투기적 실행과 조건

실행을 지원하는 최적화 컴파일러 기법의 연구를 위해 개발된 시스템이다. 컴파일러는 3종류의 중간언어인 Pcode, Hcode, Lcode에 근거하여 분할 구성된다. 각 중간언어에서는 각각 해당하는 컴파일러의 최적화 및 스케쥴링이 수행된다. 컴파일러에 의해 생성된 중간언어는 HP PA-RISC 아키텍처를 원형으로 확장된 IMPACT 아키텍처의 코드로 매핑되어 시뮬레이션이 수행된다.

HPL Playdoh 아키텍처는 명령어 수준 병렬성 추출의 연구를 위해 명령어 수준에서 고안된 아키텍처이다. 이 아키텍처는 VLIW와 슈퍼스칼라 프로세서의 동작 특성을 반영하기 위해 RISC 형태의 다양한 세트의 명령어를 지원하는 아키텍처로 ILP 프로세서의 조건실행, 투기적 실행 및 스케쥴링의 성능 측정 수단으로 사용되고 있다.

Wisconsin 대학의 SimpleScalar Tool Set은 비순차 이슈(out-of-order issue)되어 수행되는 슈퍼스칼라 프로세서인 SimplScalar 아키텍처의 성능측정을 위해 개발된 시스템이다. SimpleScalar 아키텍처는 MIPS 아키텍처의 명령어 형태를 갖고 비순차 이슈로 동작하는 슈퍼스칼라 아키텍처이다. 시스템은 크게 컴파일러와 시뮬레이터로 구성된다. 컴파일러는 기존의 gcc 컴파일러를 사용하여 SimpleScalar 명령어 세트를 생성하였고, 시뮬레이터는 SimpleScalar 아키텍처의 명령어 세트를 시뮬레이션 하여 수행 사이클, 분기 예측률, 캐시 히트율 등의 수행정보를 출력한다.

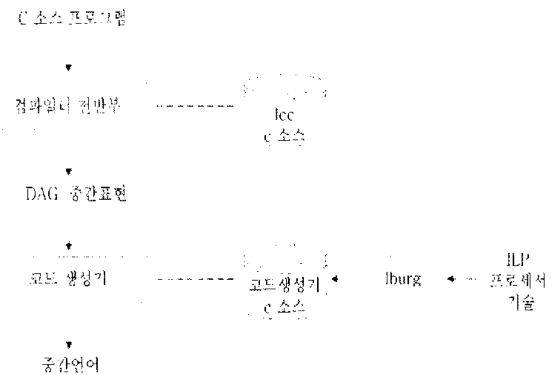
본 논문에서는 IMPACT 컴파일러 시스템, Playdoh 아키텍처와 같이 순차이슈되고 컴파일러에 의해 정적으로 스케쥴되어 명령어 수준에서 병렬처리되는 ILP 프로세서의 성능을 측정하는 시스템을 개발한다.

3. 컴파일러와 중간언어

3.1 컴파일러

연구 개발되는 C 컴파일러의 구성도는 (그림 2)와 같다. 컴파일러의 전반부(front-end)는 C 언어 소스 프로그램을 입력으로 받아 어휘분석 및 구문분석하는 과정이다. 개발된 컴파일러의 전반부는 C. Fraser와 D. Hanson이 개발한 retargetable C 컴파일러인 lcc [6]를 사용하였고, 코드 생성기(code generator)는 code-generator generator인 lburg[6]를 사용하여 구현하였다. lburg는 lcc의 중간표현인 DAG(Directed Acyclic Graph) 표현을 입력으로 받아 대상 머신의 코드를 생성하는 C 프로그램을 생성하는 코드 생성기 자동 생성 도구이다.

lcc는 ANSI C에 대한 Retargetable Compiler로 이미 그의 목적코드로 VAX, Motorola 68020, SPARC, MIPS R3000 등을 생성할 수 있게 구현되어 있다. lcc는 비록 다른 상용 컴파일러보다는 최적화된 코드를 생성하지는 않지만 그의 실행 속도는 다른 상용 컴파일러보다 2 배정도 빠르고 그 크기는 반 정도이다. lcc는 빠른 실행 속도를 위하여 전반부와 후반부와의 밀접한 관계를 유지하는 인터페이스를 사용한다. lcc는 중간언어로 DAG를 사용한다.



(그림 2) C 컴파일러 구성도
(Fig. 2) Configuration of the C Compiler

lcc의 전반부에서는 DAG를 생성하고 후반부에서는 이를 받아 목적 머신에 대한 코드를 생성한다. 즉, lcc에서는 DAG를 사용하여 전반부와 후반부를 서로 연결시킨다. 생성된 DAG는 전반부와 후반부 양쪽를 위한 데이터 구조이기 때문에 중간언어 수준에서 컴파일러 향상을 위한 어떤 추가의 단계를 삽입하기 위해서는 전반부와 후반부 양쪽에 대한 고려가 필요하다. 따라서 본 논문의 컴파일러는 이러한 점을 피하기 위해 lcc의 목적코드로 본 논문에서 정의한 중간언어가 생성되도록 lcc의 후반부를 수정하였다. 비록 이러한 일은 중간언어에서 실제 타겟머신의 코드로 변환시키는 매핑과정이 추가되지만 lcc와 같은 내부적인 표현을 갖는 중간언어보다 이해하기 쉽고 중간언어 생성단계까지의 전반부와 최적화 단계, 코드 생성단계를 포함한 후반부와의 완전한 분리를 취할 수 있다.

lcc의 전반부와 후반부의 인터페이스는 몇 개의 공유

데이터 구조, 19 개의 인터페이스 함수 그리고 36개의 연산자를 가지는 DAG로 구성된다. DAG 노드는 다음과 같은 구조체에 의해 정의된다.

```
typedef struct node *Node;
struct node {      /* DAG nodes: */
      short op;     /* 연산자 */
      short count;   /* 참조 횟수 */
      Symbol syms[3];  /* symbols */
      Node kids[2];  /* 오퍼랜드 */
      Node link;  /* forest에서의 다음 DAG */
      Xnode x;   /* 후반부를 위한 확장 */
```

kids는 오퍼랜드 노드를 가르킨다. 어떤 연산자는 오퍼랜드로 오퍼랜드 심볼테이블 엔트리를 가르킬수 있다. 이 경우 심볼테이블 엔트리를 가리키는 syms에 의해 지정된다. count는 이 노드가 다른 노드들에 의해 참조되는 횟수가 저장된다. link는 forest의 루트로부터 이어지는 다음 DAG에 대한 포인터를 가지고 있다.

〈표 1〉 Type Suffix
〈Table 1〉 Type Suffix

type suffix	type
C	char
S	short
I	int
U	unsigned
P	pointer type
F	float
D	double
B	structure
V	void

op 필드에는 노드의 연산자가 저장된다. 각 연산자의 마지막 문자는 〈표 1〉과 같은 type suffix를 가진다. 〈표 2〉는 DAG에 사용되는 연산자들의 리스트와 그에 따른 type suffix 그리고 kids의 개수, 사용되는 syms 개수를 나타낸다.

컴파일러의 후반부는 코드생성기 자동 생성기인 lburg로 구현된다. lburg는 전반부에서 생성된 DAG를 입력으로 받아 DAG의 패턴에 따른 목적코드를 생성하기 위해 일련의 규칙들을 입력받아 후반부의 코드 생성기의 C 소스 코드를 생성한다. lburg는 burg 언어라고 알려진 일련의 규칙을 읽어 트리패턴 매칭기를 생성한다. (그림 3)은 burg 문법에 대한 EBNF 표기법이다. term은 터미널(terminal) 심볼, nonterm은 논-터미널(non-terminal) 심볼을 나타낸다. 그리고, integer는 정수값을 나타낸다. 규칙들은 트리패턴으로 정의된다. 각각의 논-터미널 심볼은 하나의 트리를 나타낸다. lburg 기술은 nonterm에 각 DAG의 연산자에 해당하는 심볼을 기술하여 주고 tree 문법에 각 트리패턴에 따라 생성되는 코드를 정의한다. (그림 4)는 중간언어 생성기에 사용되는 lburg 기술의 일부 예이다. (그림 4)에서 보는 바와 같이 lburg기술에서는 심볼테이블과 오퍼랜드를 지정하기 위해 다음 〈표 3〉과 같은 템플리트가 사용된다. 템플리트들은 각 lburg 규칙들에 대한 동작을 지시한다. 예를 들면 %F는 함수의

〈표2〉 DAG 노드 연산자
〈Table 2〉 Operators of DAG nodes

syms	kids	연산자	type suffix	연산
1	0	ADDRF	P	파라미터의 주소
1	0	ADDRG	P	전역 변수의 주소
1	0	ADDRL	P	지역 변수의 주소
1	0	CNST	CSIUPFD	상수
	1	BCOM	U	bitwise complement
	1	CVC	IU	char로 타입 변환
	1	CVD	IF	double로 타입 변환
	1	CVF	D	float로 타입 변환
	1	CVI	CSUD	int로 타입 변환
	1	CVP	U	pointer로 타입 변환
	1	CVS	IU	short로 타입 변환
	1	CVU	CSIP	unsigned로 타입 변환
	1	INDIR	CSIPFDB	fetch
	1	NEG	IFD	부호 변환
	2	ADD	IUPFD	덧셈
	2	BAND	U	bitwise AND
	2	BOR	U	bitwise OR
	2	BXOR	U	birwise XOR
	2	DIV	IUFD	나눗셈
	2	LSH	IU	왼쪽으로 시프트
	2	MOD	IU	나머지 연산
	2	MUL	IUFD	곱셈
	2	RSH	IU	오른쪽으로 시프트
	2	SUB	IUPFD	뺄셈
2	2	ASGN	CSIPFDB	assignment
1	2	EQ	IUFD	같으면 분기
1	2	GE	IUFD	크거나 같으면 분기
1	2	GT	IUFD	크면 분기
1	2	LE	IUFD	작거나 같으면 분기
1	2	LT	IUFD	작으면 분기
1	2	NE	IUFD	같지 않으면 분기
2	2	ARG	IPFDB	인수
	1 2	CALL	IFDBV	함수 호출
	0 1	RET	IFDV	리턴
	1	JUMP	V	무조건 분기
1	0	LABEL	V	라벨 정의

프레임을 위해 할당된 프레임 크기를 출력하여 주고, %letter는 letter로 지정된 문자에서 'a'만큼의 값을 빼준 위치에 있는 심볼 테이블 엔트리 값을 출력한다. 예를 들면, %a는 p->syms['a'- 'a']->x.name이 되어 p->syms[0]->x.name의 값을 출력하여 준다. #은 템플리트에서 정의하기 힘든 코드들을 생성을 위해 emit() 함수를 호출하여 준다. (그림 5)는 (그림 4)의 lburg 표현에 대한 트리의 패턴 매칭 예이다.

```
grammar  → '%{' configuration '%}' { dcl } '%%' { rule }
           ['%%' C code]
dcl      → '%start' nonterm
         | '%term' { term '=' integer }
rule     → nonterm ':' tree  template [C expression] ;
tree     → term [ '(' tree [, tree ] ')' ]
         | nonterm
template → "{ any character except double quote }"
```

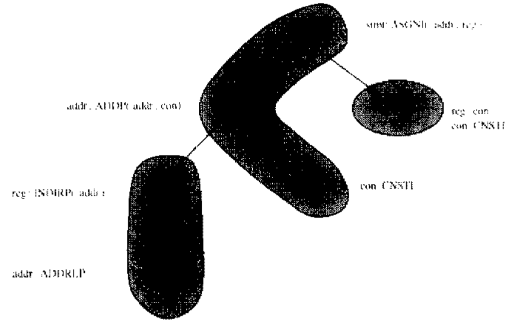
(그림 3) burg 문법규칙
(Fig. 3) Burg Grammar Rules

```
%start stmt
%term ADDI=309 ADDRLP=295 ASGNI=53
%term CNSTI=21 INDIRC=67
%%
con:  CNSTI              "%a"
addr: ADDRLP             "%a"
addr: ADDI(reg, con)     "ADD %c, %0, %1"
rc:   con                "%0"
rc:   reg                "ADD %c, %0, %1"
reg:  ADDI(reg, rc)      "ADD %c, %0, %1"
reg:  addr               "%0"
stmt: ASGNI(addr, reg)   "STR %0,%1"
```

(그림 4) lburg 기술의 예
(Fig. 4) An Example of lburg Description

〈표 3〉 템플리트
〈Table 3〉 Template

템플리트	출력
%%	'%' 출력
%F	프레임 크기
%숫자	숫자로 지정된 노드의 서브 트리
%letter	p->syms[letter - 'a']->x.name
임의의 문자	문자 자체
#	emit2() 함수 호출



(그림 5) 트리 패턴 매칭의 예
(Fig. 5) An Example of Tree Pattern Matching

3.2 중간언어

중간언어는 RISC 형태의 추상머신(abstract machine)에 대해 정의된 3 주소 코드의 명령어 형태를 가지면서 컴파일러가 지원하는 정적 정보(static information)를 표현할 수 있도록 머신 독립적으로 광범위하게 정의하였다. 중간언어의 모든 연산은 레지스터 상에서 이루어지고 메모리로의 접근은 오직 로드, 스토어 명령에 의해서만 이루어진다. 그리고 심볼에 대한 정보를 나타내기 위한 특별한 지시자들이 정의된다. 중간언어의 명령어는 그 형태에 따라 9 가지로 나누어진다. 데이터 연산 명령(DATAOP), 데이터 이동 명령(MOVOP), 비교 명령(CMPOP), 메모리 로드 명령(LDROP), 메모리 스토어 명령(STROP), 타입 변환 명령(CVOP), 분기 명령(JT, JF, JUMP), 스택 연산 명령(PUSH, POP), 그리고 그 이외의 명령들(NEGOP, CALL, RET, NOP)로 나누어진다. 다음은 각 명령어 형태에 따르는 오퍼랜드의 종류이다.

〈표 4〉에서 오퍼랜드 타입 이름 레지스터는 이름 그대로 레지스터를 의미한다. 여기서 레지스터는 레지스터 할당전의 무한개의 가상 레지스터 또는 레지스터 할당 후의 레지스터 번호일 수 있다. 가상 레지스터는 '$레지스터번호'의 형태를 띄거나 또는 소스 코드에서 사용되는 지역 변수 이름일 수 있다. 어드레스 타입은 메모리의 위치를 나타낸다. 어드레스 위치는 어드레스 위치를 정의한 라벨이거나, '*레지스터' 형태의 레지스터 간접 어드레싱 그리고 '라벨(레지스터)' 형태와 같이 라벨 + 레지스터의 어드레스를 나타내는 형태로 나타난다. 각 명령어에 대한 동작을 살펴보면, DATAOP는 오퍼랜드-2와 오퍼랜드-3에 대해 특정한 연산을 한 후 그 결과를 오퍼랜드-1에 저장하는 형태의 명령어로

〈표 4〉 중간언어의 명령어 형태에 따른 오퍼랜드 타입
〈Table 4〉 Operand Types for Instruction Classes of Intermediate Language

명령어의 형태	오퍼랜드-1	오퍼랜드-2	오퍼랜드-3
DATAOP	레지스터	레지스터	레지스터 / 상수
MOVOP	레지스터	레지스터 / 상수	없음
CMPOP	레지스터	레지스터	레지스터
LDROP	레지스터	어드레스	없음
STROP	레지스터	어드레스	없음
CVOP	레지스터	레지스터	없음
JT, JF	레지스터	목적 분기 라벨	없음
JUMP	목적 분기 라벨	없음	없음
PUSH, POP	레지스터	없음	없음
NEGOP	레지스터	레지스터	없음
CALL	함수 이름	파라미터 개수	없음
RET	없음	없음	없음

ADD, SUB, MUL, DIV, MOD, AND, XOR, OR, SHL, SHR 등의 명령이 있다. MOVOP는 오퍼랜드-2에서 오퍼랜드-1으로 데이터 이동 명령으로 MOV명령이 있다. CMPOP는 오퍼랜드-2와 오퍼랜드-3와 비교 연산을 한 후 각 비교 명령에 따른 결과를 오퍼랜드-1에 저장한다. 이에 대한 명령어로는 CMPEQ, CMPNE, CMPGE, CMPGT, CMPLE, CMPLT 등의 명령이 있다. LDROP와 STROP는 메모리 로드, 스토어 명령으로 LDR와 STR이 있다. CVOP는 데이터 변환명령이다. 오퍼랜드-2의 데이터는 지정된 명령대로 변환된 후 오퍼랜드-1에 저장한다. 명령어로는 CVCI, CVCU, CVSI, CVSU, CVIC, CVIS, CVIU, CVID, CVUI, CVUS, CVDF, CVDI, CVFD 등의 명령이 있다. 다음으로 분기명령 JT, JF, JUMP 명령이 있고 스택에 관계된 명령으로 PUSH, POP 그리고 그 이외의 명령으로 bitwise not명령인 NEG, 함수 호출 명령 CALL 그리고 리턴 명령 RET 등이 있다. 각각의 명령어에는 각 명령어들과 그의 오퍼랜드의 데이터 타입에 따라 type suffix가 붙을 수 있다. 예를 들면, 정수형의 데이터에 대한 덧셈 명령은 ADD이고, float형 덧셈 명령은 ADDF 그리고 double형 덧셈 명령은 ADDD이다.

중간언어 상에서 의사 명령어(pseudo operation)는 모두 '.'으로 시작한다. 의사 명령어들을 살펴보면 .START PROGRAM과 .END PROGRAM은 단순히 프로그램의 시작과 끝을 나타내고, .SEGMENT TEXT는 코드 세그먼트를 구별하고, .SEGMENT DATA는 데이터 세그먼트를 구별하기 위해 사용된다. 그리고 .GLOBAL은 파일의 범위를 벗어나 외부에서 참조되는 심볼이름을 정의한다. 함수들간의 구별을 쉽게 하기 위해서 함수의 시작과 끝을 나타내기 위한 .FUNC (함수이름)과 .ENDF (함수이름)이 사용된다. 다음으로 명령어 오퍼랜드로 오지 못하는 상수들을 위한 의사 명령어들이 있다. 이러한 상수은 리터럴 세그먼트나 데이터 세그먼트에 정의된다. 〈표 5〉는 메모리 내에 상수 정의를 위한 의사 명령어이다. (그림 6)은 프로그램에서 사용된 여러 가지 심볼에 대한 정보를 나타내기 위한 의사 명령어이다.

〈표 5〉 메모리 상수 정의 의사 명령어
〈Table 5〉 Pseudo Operations Defining Constants in Memory

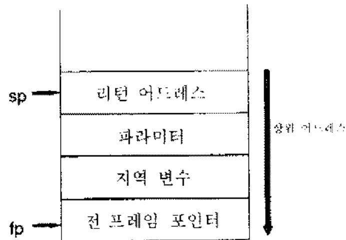
의사 명령어	의 미
.BYTE (상수값)	바이트 상수
.WORD (상수값)	워드 상수
.HWORD (상수값)	반 워드(half word) 상수
.DWORD (상수값)	더블 워드(double word) 상수
.SPACE (바이트단위의 크기)	명시한 크기만큼의 메모리 예약

* .global (name) (type) (scope) (ref) (size)
  의미 : 파일 범위를 가지는 심볼 정의.
  (name) : 메모리 위치에 대한 심볼 이름을 정의.
  (type) : 심볼의 타입을 명시하며 char, int, float, double, complex가 될 수 있다. complex 타입은 배열이나 구조체인 경우이다.
  (scope) : 심볼의 유효 범위를 나타낸다. 모두 GLOBAL이다.
  (ref) : 생성된 중간언어상에서 정적으로 참조되는 횟수.
  (size) : 심볼에 대한 크기 정보.
* .local (name) (type) (scope) (stack offset) (ref) (size)
  의미 : 함수 단위의 지역 범위를 가지는 심볼 정의.
  (name), (type), (ref), (size)는 위와 같은 의미를 가진다.
  (scope)는 모두 LOCAL이다.
  (stack offset)은 프레임(frame)상에서의 위치를 나타낸다.
* .parameter (name) (type) (scope) (stack offset) (ref) (size)
  의미 : 함수 호출자로부터 전달받는 인수 정보.
  scope가 PARAMETER임을 제외하고 위와 같은 의미를 가진다.

(그림 6) 심볼정의 의사 명령어
(Fig. 6) Pseudo Operations Defining a Symbol

중간언어의 생성 시 함수의 프레임(frame)은 (그림 7)과 같은 구조로 구성된다. 프레임 포인터(fp)는 현재

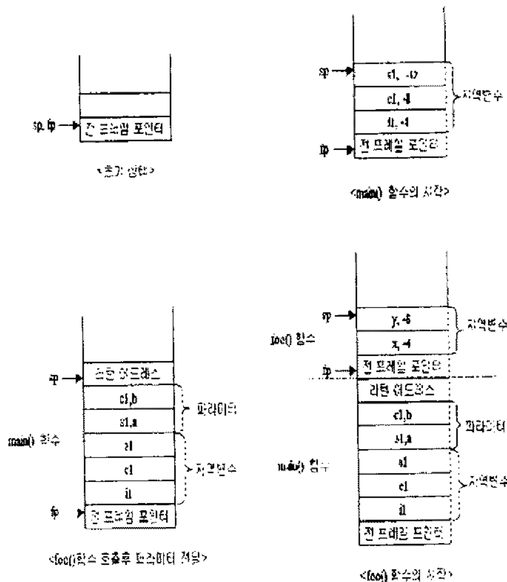
실행되고 있는 함수들을 위한 레코드 집합의 베이스를 가리키고 있다. 그리고 스택 포인터(sp)는 스택의 맨 위를 가리킨다. (그림 9)는 (그림 8)의 예제 C 소스에 대한 프레임 구조의 운영에 대한 예를 나타낸다.



(그림 7) 프레임 구조
(Fig. 7) Frame Construction

```
main() {                    int foo(int a, char b)
int i1;                     int x,y;
char c1;                    x = a;
short s1;                   y = b;
i1 = foo(s1,c1);            return x+y;
}                           }
```

(그림 8) C 소스 프로그램 예
(Fig. 8) An Example of a C Source Program



(그림 9) 프레임 운용의 예
(Fig. 9) An Example of Frame Operation

ILP 프로세서에서 표현될 수 있는 중간언어의 정적 정보로는 분기예측 방향, 조건실행 등이 있다. 정적 분기예측 정보는 명령어의 종류나 또는 미리 프로그램을 실행하여 얻어진 프로파일링 정보를 이용하여 컴파일러가 분기의 방향을 예측한 정보로써 예측된 분기방향이 맞았으면 아무런 분기손실 없이 지속적인 파이프라인 처리를 위해서 사용되는 정보이다. (그림 10)은 팩토리얼(factorial)를 계산하는 C 프로그램과 이를 컴파일하여 생성된 중간언어를 보여준다. 생성된 코드는 프로파일링을 이용하여 분기예측된 중간 코드이다.

```
int Result;
main()
{   int f=5;
    Result = factorial(f);
}

int factorial(int x)
{  if (x == 1)
        return 1;
   else
        return(factorial(x-1)*x);
}
```

(a)

```
.START PROGRAM
.GLOBL main
.SEGMENT TEXT
.FUNC main
.local f int LOCAL -4  2000  4
        PUSH    fp              ; save old sp
        MOV     fp,sp           ;new fp
        SUB     sp,sp,#4        ; sp <- sp - local_size
        MOV     f, #5                           ;integer
        PUSH    f                               ;integer
        CALL    factorial,1
        ADD     sp,sp,#4        ;sp <- sp + parameter
        STR     _ret,Result                     ;integer
L.1:
        ADD     sp,sp,#4        ; sp <- sp + local size
        POP     fp              ; restore old fp
        RET     ; return, pop return address
.ENDF main
.GLOBL factorial
.FUNC factorial
.parameter x int PARAMETER 8  2000  4
        ;function prolog
        PUSH    fp              ; save old sp
        MOV     fp,sp           ;new fp
        SUB     sp,sp,#0        ; sp <- sp - local_size
        CMPNE   $10,x, #1
      JT      $10,L.4 TA    ; Taken Prediction
        MOV     _ret, #1
        JUMP    L.3
L.4:
        SUB     $6,x, #1                        ;integer
        PUSH    $6                              ;integer
        CALL    factorial,1
        ADD     sp,sp,#4        ;sp <- sp + parameter
        MUL     $8,_ret,x
        MOV     _ret,$8
L.3:
        ADD     sp,sp,#0        ; sp <- sp + local size
        POP     fp              ; restore old fp
        RET     ; return, pop return address
.ENDF factorial
.SEGMENT DATA
.GLOBL Result
.global Result int GLOBAL 0  1000  4
Result:
.SPACE 4
.END PROGRAM
```

(b)

(그림 10) 팩토리얼 프로그램의 (a) C 소스 프로그램 (b) 생성된 중간언어
(Fig. 10) (a) A C Source Program (b) Generated Intermediate Code for a Factorial Program

4. 시뮬레이터

컴파일러에 의해 생성된 중간언어는 시뮬레이터에 입력되어 다음과 같은 기본 시뮬레이션과 정적 스케줄링 방식, 분기구조 및 분기예측 시뮬레이션을 수행한다.

□ 기본적인 시뮬레이션
- 수행 후 메모리 내용
- 수행된 클럭 수 및 명령 트레이스
- 사용된 명령들의 정적/동적 빈도 수

□ 정적 스케줄링 시뮬레이션
- 순차이슈 방식
- 조건실행 방식
- 투기적 실행 방식

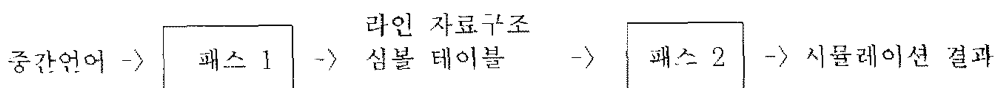
□ 분기구조 시뮬레이션
- 지연분기 방식
- 분기취소 방식
- 선택적 분기취소 방식

□ 분기예측 시뮬레이션
- 정적 분기예측 방식
- 분기예측률
- profiling 정보

시뮬레이션 시 이들 방식의 선택과 명령 이슈 수, 분

```
usage : ints [-options] filename
          log file(*.log), memory file(*.mem) are generated.
          -In,   Issue rate
                 n : 1(default) 2 4
          -Bn,   Branch scheme
                 n : 0, stalled branch scheme(default)
                     1, delayed branch scheme
                     2, squashing scheme
                     3, selective squashing scheme
          -Sn,   Delay Slot(Branch Penalty)
                 n : 1(default) 2 3 4
          - P    output branch profiling information in a file if 1-issue and stalled branch
                 scheme(default scheme)
          -D[n..] Debug options
                 n : 0, display for lexical analysis trace
                     1, display for line data structures
                     2, output for line data structures in a file *.dbg
                     3, display trace for line data structures
          -s     output symbol tables in a file *.sym
          -T[n..][r] output instruction traces
                 n : 0, display
                     1, output in *.tra
                 r : display resource states
```

(그림 11) 시뮬레이터 입력 파라미터
(Fig. 11) Simulater Input Parameters

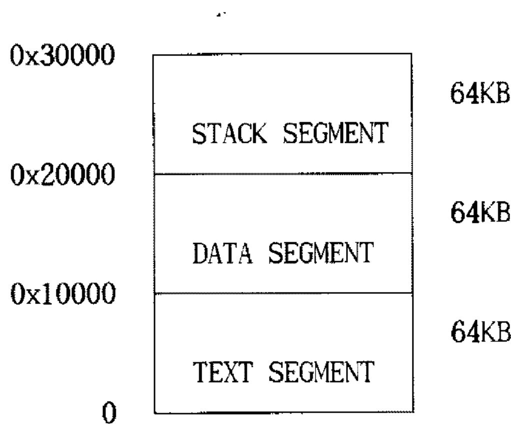


(그림 12) 시뮬레이터의 구성
(Fig. 12) Configuration of the Simulator

기손실 클럭 및 디버그 옵션 등의 파라미터가 (그림 11)과 같이 입력된다.

시뮬레이터는 (그림 12)과 같이 2 패스로 수행된다. 패스 1에서는 중간언어의 각 행을 읽어들여 해당 행의 명령 및 심볼에 대한 자료구조를 구축한다.

시뮬레이션 되는 메모리의 각 세그먼트는 (그림 13)과 같이 분할하여 사용한다. 소스 프로그램의 저장영역인 텍스트 세그먼트는 0 번지부터 시작하여 64KB의 영역이 할당된다. 데이터 세그먼트는 0x10000번지부터 시작하여 64KB가 할당되고, 마지막으로 스텍 세그먼트는 0x20000번지로부터 64KB의 영역이 할당된다. 심볼테이블은 해쉬테이블로 구성하였으며, scope(global, local)에 따라 독립적인 해쉬테이블을 운영한다. 위의 해쉬테이블에 대한 각 심볼의 해쉬 컨트롤은 심볼 데이터 자료구조에 저장된다.



(그림 13) 시뮬레이터의 메모리 영역
(Fig. 13) Memory Area of the Simulator

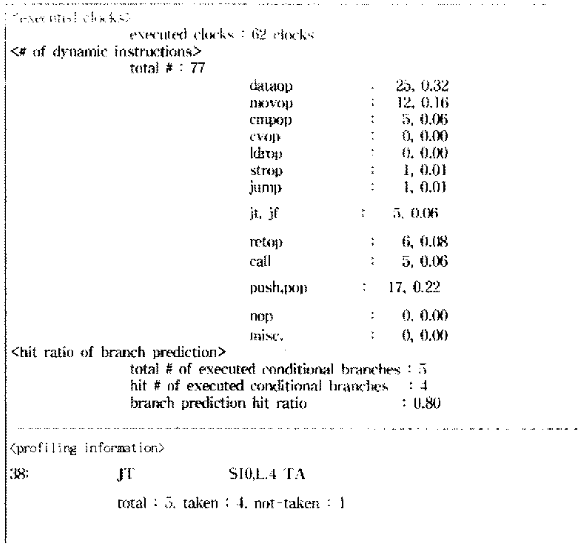
(그림 14)은 시뮬레이터의 패스 1에서 구성하는 라인 자료구조의 구성을 보인다. 시뮬레이터는 소스 중간언어를 한 라인씩 읽어들여 그 라인에 대한 모든 정보를 라인 자료구조에 저장한다. 패스 2에서는 패스 1에서 구축된 각 행의 자료구조를 읽어들여 파이프라인 스테이지 별로 구분하여 시뮬레이션을 수행한다. 동시에 다중 이슈되는 명령들은 명령 간의 데이터 종속 관계를 조사하여 종속관계가 없는 명령에 대해서만 동시에 실행하도록 스케쥴된다. 시뮬레이터에서 스케쥴은 각 명령들이 이슈 수에 따라 이슈와 수행 순서를 유지하면서 (in-order issue,in-order execution) 스케쥴 된다. 조건실행의 경우 각 파이프라인 단계에서 조건실행 기술자의 값으로 명령의 실행여부를 결정한다.

(그림 15)는 (그림 10)의 팩토리얼 예제 프로그램을 시뮬레이션하여 생성된 결과이다. (그림 16)은 (그림 10)의 팩토리얼 프로그램을 2-이슈,1-분기손실 클럭을 갖고 profiling 정보를 사용하여 분기의 방향을 예측하고 시뮬레이션한 후에 생성된 명령의 트레이스와 메모리,레지스터 내용을 보여주는 그림이다. (그림 16)에서 분기명령 (1)은 조건 분기문의 예측이 맞아서 아무런 분기 손실없이 인출된 명령을 처리하는 과정을 보여주며, 분기명령 (2)는 예측이 틀려서 이미 인출된 명령을 취소하고 올바른 분기 방향의 명령을 실행하는 것을 알 수 있다. (그림 16)의 (3)은 데이터 종속관계가 없는 명령들이 동시에 이슈되어 실행되는 것을 보여준다.

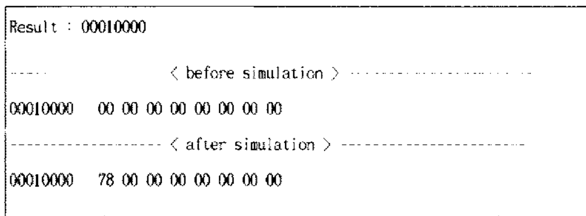
```
typedef struct LineS {
    int lt;                     /* line type */
    unsigned long no;           /* line number */
    int addr;               /* address value */
    Symbol *lb;             /* label name */
    struct {                /* operation code */
            int ot;
            int v;  /* opcode value */
         } op;

   union {                      /* operand 1 */
         Symbol *s;
         char *id;
         union {
         long i;
         double d;
         } v;
    } oprnd1;
   union {                      /* operand 2 */
         Symbol *s;
         Opr *o;
         Addr *a;
         int n;
    } oprnd2;
    union {                     /* operand 3 */
         Opr *o;
         Symbol *s;
         int pred;
    } oprnd3;
    int sq_bit;
    Symbol *predicate;
    char buf[MAX_BUFFER];
    struct LineS *next;
    struct LineS *prev;
} LineS;
```

(그림 14) 라인 자료구조
(Fig. 14) Line Data Structure



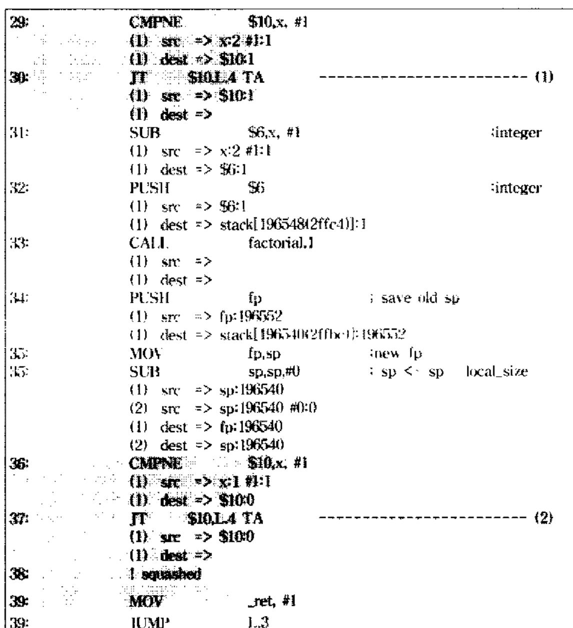
(a)

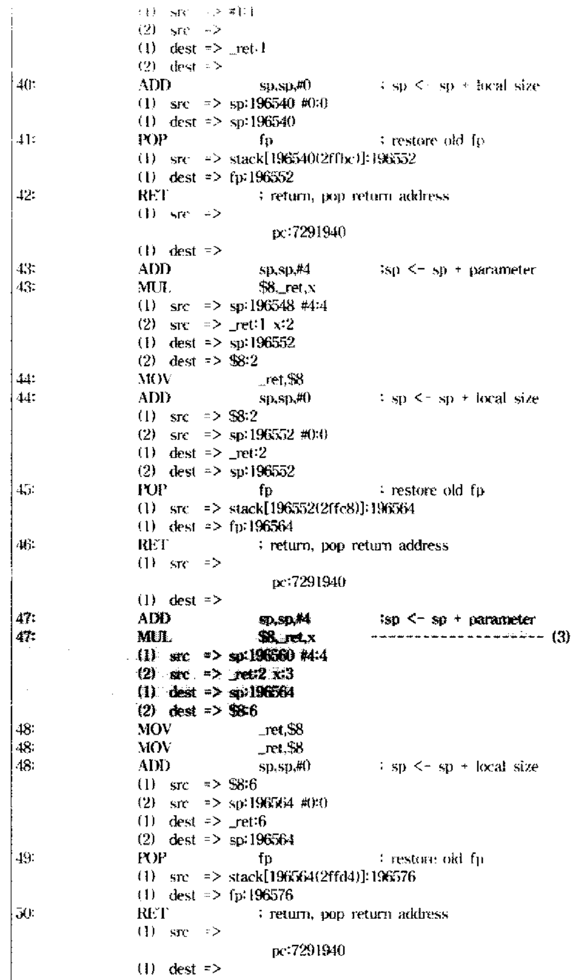


(b)

(그림 15) (그림 10) 팩토리얼 프로그램의 (a) 시뮬레이션 결과 정보 (b) 메모리 덤프 내용

(Fig. 15) (a) The Information of Simulation Results (b) The Contents of Memory Dump for the Factorial Program in (Fig.10)





(그림 16) (그림 10) 팩토리얼 프로그램의 트레이스 생성 예

(Fig. 16) An Example of Trace Generation for the Factorial Program in (Fig. 10)

5. 성능 측정 적용 예

개발된 성능측정 시스템의 검증을 위하여 순차이슈되어 정적 스케쥴링되는 조건실행 방식[14,21]과 분기처리 방식의 성능을 측정하여 분석함으로써 타당성을 보인다. <표 6>은 성능 측정을 위해 사용된 벤치마크 프로그램을 보여주며, <표 7>은 각 벤치마크 프로그램을 수행시켜 동적으로 실행된 명령의 수와 각 명령 형태의 비율을 보여주는 표이다.

<표 7>에서 dataop는 데이터 연산 명령, movop는 데이터 이동 명령, branch는 분기명령, cmpop는 비교명령, memory는 메모리 참조 명령을 나타내며 etc.는

〈표 6〉 벤치마크 프로그램
〈Table 6〉 Benchmark Programs

내용구분		내 용 설 명
시뮬레이션 프로그램	mxmi	최대값,최소값 구하기(max-minmum) 프로그램
	bish	이진탐색(binary search) 프로그램
	qsrt	빠른정렬(quick sort) 프로그램
	fibn	피보나치 수열 구하기(fibonacci sequence) 프로그램
	wcnt	문장내의 문자, 단어 및 줄의 수 세기(word counter) 프로그램

〈표 7〉 벤치마크 프로그램의 동적 명령어 수 및 비율
〈Table 7〉 Dynamic Instruction Count and Ratio for Becnhmark Programs

[illegible]	dataop		movop		branch		cmpop		memory		etc		총 명령어 수
	수	비율	수	비율	수	비율	수	비율	수	비율	수	비율	
mxmi	33	0.22	13	0.09	31	0.21	27	0.18	45	0.30	2	0.01	151
bish	20	0.26	12	0.15	16	0.20	12	0.15	16	0.20	2	0.03	78
qsrt	157	0.24	99	0.15	152	0.23	113	0.17	92	0.14	38	0.06	651
fibn	23	0.20	36	0.32	17	0.16	12	0.11	20	0.18	5	0.04	113
wcnt	244	0.12	365	0.17	464	0.22	422	0.20	354	0.17	2	0.01	2109
평균비율	0.21		0.18		0.20		0.16		0.20		0.03		

이외의 기타 명령을 의미한다. 〈표 7〉에 나타난 바와 같이 데이터 연산 명령을 제외하고는 분기명령과 메모리 참조 명령의 비율이 평균 20% 정도로 수행되어 이들 명령의 효율적인 처리가 성능에 큰 영향을 미침을 알 수 있다.

5.1 조건실행 방식 성능 측정

조건실행은 조건에 따라 조건적으로 실행되는 조건실행 명령(predicated instruction)을 사용하여 분기명령을 제거하는 기법으로 조건실행 명령은 일반 명령에 조건실행 기술자(predicate specifier)를 첨가한 명령이다. 조건실행은 조건실행 명령의 조건실행 기술자의 조건을 평가하고, 만약 조건이 참이면 그 명령을 실행하고, 그렇지 않으면 그 명령을 NOP로 처리한다. 예를들어 다음과 같은 ADD 명령의 경우,

ADD dest, src1, src2 IF p_cond

만약 p_cond의 조건이 참이면 src1과 src2를 더해서 dest에 저장하고, 만약 p_cond가 거짓이면 ADD 명령은 NOP로 처리되어 프로그램 실행에 아무 영향을

〈표 9〉 조건실행 시뮬레이션 모드에 따른 측정값
〈Table 9〉 Performance Measurements of Each Simulation Mode for Predicated Execution

모드	I			II			III			IV		
이슈 / PG	1	2	4	1	2	4	1	2	4	1	2	4
mxmi	181 (1.00)	160 (1.13)	159 (1.14)	161 (1.12)	148 (1.22)	146 (1.24)	150 (1.21)	142 (1.27)	141 (1.28)	142 (1.27)	138 (1.31)	131 (1.38)
bish	93 (1.00)	80 (1.16)	79 (1.18)	86 (1.08)	72 (1.29)	71 (1.31)	80 (1.16)	68 (1.37)	67 (1.39)	71 (1.31)	67 (1.39)	66 (1.41)
qsrt	784 (1.00)	712 (1.10)	698 (1.12)	621 (1.26)	594 (1.31)	592 (1.32)	614 (1.28)	592 (1.32)	590 (1.33)	593 (1.32)	578 (1.35)	576 (1.36)
fibn	127 (1.00)	102 (1.25)	100 (1.27)	114 (1.11)	92 (1.38)	91 (1.39)	108 (1.18)	98 (1.30)	87 (1.46)	94 (1.35)	82 (1.55)	81 (1.56)
wcnt	2120 (1.00)	1802 (1.17)	1800 (1.18)	1980 (1.07)	1790 (1.18)	1788 (1.19)	1953 (1.09)	1711 (1.23)	1708 (1.24)	1910 (1.11)	1708 (1.24)	1701 (1.25)
평균	(1.00)	(1.16)	(1.18)	(1.13)	(1.28)	(1.29)	(1.18)	(1.30)	(1.34)	(1.27)	(1.37)	(1.39)

주지 않는다. 즉 조건실행은 조건 분기명령을 이용하지 않고도 조건적으로 실행되는 프로그램의 구조를 표현할 수 있으며, 조건실행 기술자는 올바른 결과만이 프로세서의 상태에 영향을 주도록 한다.

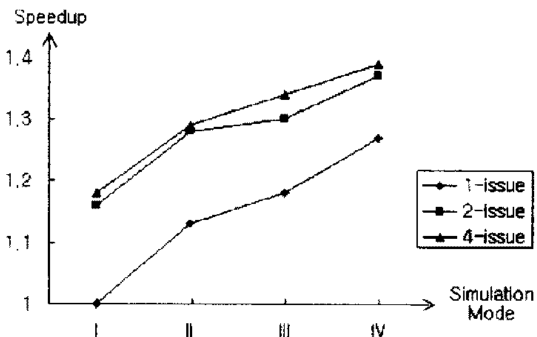
본 논문에서는 논문 [21]에서 제안된 조건실행(predicated execution) 방식을 본 논문의 시스템에 적용하여 측정하고 분석한 내용을 소개한다. 각 방식의 비교 측정을 위하여 적용된 시뮬레이션 모드는 〈표 8〉에 표시 되었다.

〈표 8〉 조건실행 시뮬레이션 모드
〈Table 8〉 Simulation Mode of Predicated Execution

시뮬레이션 모드	I	스케쥴을 하지 않은 일반적인 코드
	II	국소 스케쥴링(local scheduling) 코드
	III	광역 스케쥴링(global scheduling) 코드
	IV	조건실행 코드

시뮬레이션 모드 I은 정적 스케쥴하지 않고 C 소스 프로그램을 컴파일한 결과 그대로를 시뮬레이션 하였고 모드 II에서는 기본블럭 단위로 국소 스케쥴(local schedule)한 코드에 대한 시뮬레이션 모드이다. 스케쥴링 알고리듬으로는 리스트 스케쥴링 알고리듬[1]을 적용하였다. 리스트 스케쥴링 알고리듬은 먼저 각 명령의 오퍼랜드를 분석하여 데이터 종속관계를 나타내는 DDG(Data Dependent Graph)를 구성한다. DDG 구성 후에 각 DDG 상의 노드에서 부모노드가 없는 노드부터 스케쥴 대상으로 선정하여 스케쥴한다. 모드 III은 기본블럭 단위를 넘어서 광역 스케쥴링한 코드의 시뮬레이션 모드이고, 슈퍼블럭 스케쥴링 알고리듬[14]을 적용하였다. 슈퍼블럭은 하나의 진입점과 여러 개의 출구를 갖고 중간 진입점(side entrance)이 없는 트레이스이다. 이 슈퍼블럭은 여러개 기본블럭을 결합하여 스케쥴 대상이 되는 프로그램의 영역이 기본블럭 보다 커짐으로써 명령들의 병렬처리 가능성이 향상된다. 구성된 슈퍼블럭에 대해 리스트 스케쥴링 알고리듬이 적용된다. 모드 IV는 if-변환으로 분기문을 제거하고 조건실행을 지원하는 명령으로 프로그램을 변환 한 후 슈퍼블럭 스케쥴링 알고리듬을 적용하여 스케쥴된 코드를 시뮬레이션한 모드이다.

〈표 9〉는 명령의 이슈 수와 시뮬레이션 모드에 따라 각 벤치마크에 대해 시뮬레이션한 후 측정된 클럭 수를 보여주는 표이다. 〈표 9〉에서 괄호 안의 값은 각 벤치마크 프로그램에 대해 시뮬레이션 모드 I, 1-이슈의 클럭 수를 해당 유형의 클럭 수로 나누어서 구한 성능향상(speedup) 값이다. (그림 17)은 시뮬레이션 모드에 따라 성능이 향상되는 정도를 보여주는 그래프이다. 〈표 9〉와 (그림 17)에서 나타난 바와 같이 1-이슈에 대해서는 조건실행을 적용한 모드 IV가 모드 I, II, III 보다 각각 27%, 14%, 9%로 성능이 향상되었고, 2-이슈에 대해서는 21%, 9%, 7% 정도 성능이 향상되었다. 또한 4-이슈에 대해서는 21%, 10%, 5% 정도 성능이 향상되었음을 알 수 있었다. 이와같은 실험 결과 조건실행이 명령어 수준 병렬성을 효과적으로 이용하여 ILP 프로세서의 성능을 향상시킬 수 있음을 확인하였다.



(그림 17) 조건실행 시뮬레이션 모드에 따른 성능향상
(Fig. 17) Speedup of Simulation Modes for Predicated Execution

5.2 분기처리 방식 성능측정

분기처리 방식의 성능은 논문 [18,20]에서 적용하여 측정하고 분석한 내용을 소개한다. 본 논문에서는 무조건 파이프라인 정지(pipeline stall), 지연분기(delayed branch), 동적 분기취소과 선택적 분기 취소 방식 등의 분기처리 방식 등에 대하여 성능을 비교한다.

무조건 파이프라인 정지는 분기명령의 분기조건과 타겟주소가 결정될 때까지 분기손실 클럭 만큼 파이프라인을 정지시키는 방식이다. 지연분기 방식은 하드웨어에 의한 파이프라인 정지를 사용하지 않고 컴파일러에 의해 분기손실을 줄이는 방식이다. 즉, 이 방식은 분기명령의 조건 및 타겟주소가 결정될 때 까지 지연되

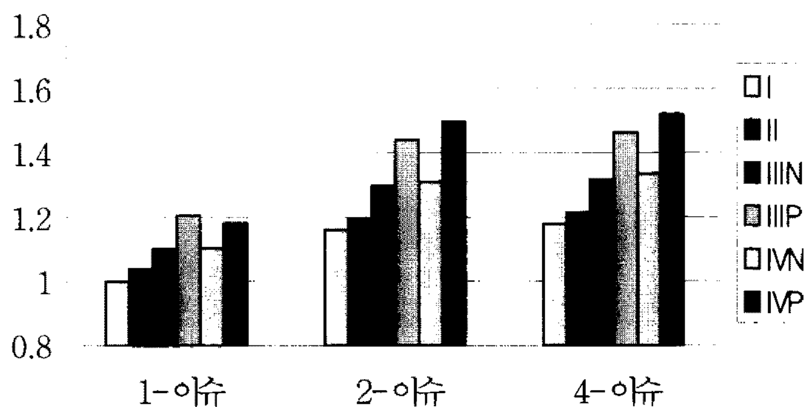
어야 하는 클럭 사이클 만큼의 지연슬롯(delay slot)에 NOP(no operation)을 삽입한다. 컴파일러는 NOP에 의한 사이클 낭비를 줄이기 위해 분기명령과 데이타 종속관계가 없는 명령어를 가능한 한 많이 NOP 대신에 삽입하는 스케쥴을 하여 분기손실을 줄이는 방식이다. 분기손실을 줄이는 다른 방식으로 미리 분기의 방향을 예측하여 실행하는 동적 분기취소방식이 있다. 이 방식의 분기예측은 분기명령의 과거 수행정보인 분기조건, 분기 타겟주소등을 수집하여 BTB(branch target buffer)에 저장한다. 다시 해당 분기명령을 페치할 때 이미 BTB에 저장된 분기예측과 타겟주소 정보를 이용하여 분기명령을 수행한다. 분기조건이 판정되어 예측된 분기방향이 맞았으면 아무런 분기손실 없이 지속적인 파이프라인처리가 이루어지며, 분기예측이 틀렸을 경우에는 이미 인출되어 처리된 명령어들을 하드웨어에 의해 동적으로 취소하고 올바른 경로의 명령어들을 다시 인출하여 실행한다. 선택적 취소 분기방식은 지연분기 방식과 분기취소 방식의 이점을 결합하여 정적 스케쥴링에 의한 분기손실을 줄이는 방식이다. 컴파일러가 분기명령과 데이타 종속관계가 없는 안전한 명령을 지연슬롯에 스케쥴한 후 채우지 못한 남은 지연슬롯에 NOP로 채우지 않고 분기명령의 방향을 예측하여 예측된 방향의 불안전한 명령을 채우고 실행한다. 분기예측이 맞는 경우에는 분기손실 없이 실행하고 분기예측이 틀렸을 경우에는 불안전한 명령만을 취소한다. 이들 각 분기처리 방식에 대해서 분기예측은 무조건 분기실패 예측(predict not-taken)과 profiling 방식[11]을 적용하여 성능을 측정하였다. profiling 방식은 사전에 임의의 데이터를 사용하여 프로그램을 실행하여 분기의 방향에 대한 정보를 구한 후에 이 정보에 근거하여 분기의 방향을 예측하는 방식이다. 〈표 10〉은 성능측정에 적용된 각 분기처리 방식의 시뮬레이션 모드를 나타낸다.

〈표 10〉 분기처리 방식 시뮬레이션 모드
〈Table 10〉 Simulation Mode of Branch Schemes

기호	시뮬레이션 모드	분기예측방식
I	무조건 파이프라인 정지 방식	X
II	지연분기 방식	X
III_N	동적 분기취소 방식	predict not-taken
III_P	동적 분기취소 방식	profiling
IV_N	선택적 취소 분기 방식	predict not-taken
IV_P	선택적 취소 분기 방식	profiling

(그림 18)은 2개의 분기손실((braqnch penalty) 클럭을 갖는 경우 〈표 10〉의 각 방식에 대한 성능을 보여주는 그림이다. (그림 18)에서 분기예측은 같은 분기예측 조건하에서 각 분기취소 효과에 다른 성능을 비교하기 위해서 동적 분기취소 방식에서도 무조건 분기실패 예측과 profiling 방식의 분기예측을 적용하여 시뮬레이션 하였다. 성능향상은 1-이슈,무조건 파이프라인 정지 방식의 실행 클럭 수를 각 방식의 실행 클럭 수로 나눈 상대값이다.

(그림 18)에 나타난 바와 같이 하드웨어에 의한 동



(그림 18) 분기처리 방식에 대한 성능향상
(Fig. 18) Speedup for Branch Schemes

적 분기취소 방식의 성능이 가장 우수함을 알 수 있다. 그러나 부가 하드웨어에 의존하지 않고 컴파일러의 스케쥴에 의한 선택적 취소 분기 방식도 이 동적 분기취소 방식 보다 평균 3% 정도 밖에 성능 차이가 나지 않음을 알 수 있다. 이 차이는 선택적 취소 분기방식이 지연슬롯에 스케쥴할 수 있는 명령들의 영역이 함수 내로 국한되어 지연슬롯이 증가할수록 늘어난 지연슬롯을 채울수 있는 명령들이 부족한 경우가 발생하는데 반하여 동적 분기취소 방식은 예측이 맞았을 경우 분기명령 이후에도 함수의 영역을 넘어서 지속적으로 실행됨으로 인한 성능의 차이이다. 또한 동적 분기취소 방식이 지연분기 방식 보다 평균 11%(III_N), 28%(III_P) 정도 성능이 향상되었는데 이는 지연분기 방식이 데이타 종속관계가 없는 안전한 명령으로 지연슬롯을 충분히 채우지 못하고 있음을 나타낸다. 결론적으로 컴파일러에 의해 효과적인 정적 스케쥴링을 하는 경우 적은 하드웨어를 가지면서 분기손실에 의한 성능저하는 동적방식과 비슷한 수준을 유지할 수 있음을 알 수 있다.

6. 결 론

본 논문에서는 한 사이클에 여러 개의 명령들이 다중이슈(multiple issue)되어 명령어 수준에서 병렬처리되는 ILP 프로세서의 성능을 측정하고 평가하는 도구를 연구 개발하여 제안하였다. 개발된 시스템은 크게 C 컴파일러와 시뮬레이터로 구성하였다. 컴파일러는 성능을 테스트하기 위한 C 벤치마크 프로그램을 입력으로 받아 3-주소 코드형태의 중간언어를 생성하였다. 생성된 중간언어는 설계 사양의 파라미터와 함께 시뮬레이터에 입력되어 시뮬레이션된 후 메모리 내용, 수행된 클럭 수 및 명령 트레이스,수행된 명령들의 동적 빈도 수, 분기명령의 예측률, profiling 정보 등을 출력하였다. 개발된 성능측정 시스템의 검증을 위하여 정적 스케쥴링되는 조건실행 및 분기처리 방식에 대해 벤치마크 프로그램을 컴파일하고 시뮬레이션을 하여 성능측정을 수행한 결과 본 논문에 제안된 성능 측정도구가 ILP 프로세서를 위한 효과적인 성능 측정 및 검증 수단으로 사용될 수 있음을 확인하였다.

따라서 연구 개발된 성능 측정 및 평가 시스템을 ILP 프로세서의 명령어 수준 상위 레벨에서 성능을 검증하고 평가하는 수단으로 사용하면 ILP 프로세서 및 이를 위한 최적화 컴파일러 연구를 위한 기초 연구자료를 수집하고 설계 사양을 결정하는 도구로서의 활용이 기대된다.

앞으로 보다 대규모의 벤치마크 프로그램의 컴파일을 위해서 C 컴파일러의 라이브러리를 개발하여 연결하고, 시뮬레이터가 비순차 이슈(out-of-order issue)를 지원하도록 보완할 예정이다.

참 고 문 헌

[1] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, 1986.

[2] J. Hennessy and D. Patterson, *Computer Architecture a Quantitative Approach*, Mor-gan Kaufmann Publishers Inc., 1990.

[3] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[4] J.Smith and G.Sohi, "The Microarchitecture of Superscalar Processors", Proceedings of the IEEE, Vol.83 No.12, pp.1609-1624, Dec. 1995.

[5] W.Hwu, R.Hank, D.Gallagher, S.Mahlke, D.Lavery, G.Haab,J.Gyllenhall and D.August, "Compiler Technology for Future Microprocessors", Proceedings of the IEEE, VOL.83, No.12, 1995.

[6] C.Fraser and D.Hanson, *A Retargetable C Compiler : Design and Implementation*, The Benjamin/Cummings Publishing Company, Inc., 1995.

[7] *Introduction to Shade*, Sun Microsystems Laboratories, Inc., 1992.

[8] D.Burger,T.Austin and S.Bennett, "Evaluating Future Microprocessors: the SimpleScalar Tool Set", Technical Report 1308, Com-puter Science department, University of Wisconsin, Madison, 1996

[9] D.Burger and T.Austin, "The SimpleScalar Tool Set, Version 2.0", Univ. of Wisconsin-Madison, Computer Science Department Technical Report #1342, -June, 1997.

[10] S. McFarling and J. Hennessy, "Reducing the Cost of Branches",Proc. of the 13th An

-nual Symposium on Computer Architecture, pp. 396-403, 1986.

[11] W. Hwu,T. Conte and P. Chang, "Comparing Software and Hardware Schemes of Reducing the Cost of Branches", Proc. of the 16th Annual Symposium on Computer Architect-ure, pp. 396-403, 1986.

[12] T. Yeh and Y. Patt, "Two-Level Adaptive Training Branch Prediction", Proc. of the 24th Annual International Symposium on Microarchitecture, pp.51-61,Nov.1991.

[13] C. Su and A. Despain, "Branch With Masked Squashing in Superpipelined Processors", Proc. of the 21th Annual Symposium on Computer Architecture, pp. 130-403, 1994.

[14] S.Mahlke, "Exploiting Instruction Level Parallelism in the Presence of Conditional Branches", Ph.D dissertation, Department of Electrical and Computer Engineering, University of Illinois,Urbana, IL, 1996

[15] V.Kathail, M. Schlansker and B. Rau, "HPL PlayDoh Architecture Specification: Version 1.0", HP Laboratories Technical Report HPL-93-80, Feb. 1994.

[16] Free Software Foundation, *GNU FLEX/BISON Manual*,1995.

[17] 정회목,문수묵, "고성능 마이크로 프로세서를 위한 최적화 컴파일러 동향", 한국정보 과학회지,제14권 제7호, pp.27-35, 1996.7.

[18] 황은석,최준기,유병강,이상정, "슈퍼스칼라 프로세서를 위한 선택적 취소 분기 방식" 대한전자공학회 추계학술대회 논문집 제18권 제2호, pp.381-384, 1995.

[19] 이상정, *마이콤 컴파일러의 최적화 기법에 관한 연구*, 삼성전자주식회사 위탁연구과제 최종보고서, 1996.

[20] 심현규,김유신,이상정, "ILP 프로세서를 위한 성능 측정 도구의 개발", 한국정보과학회 '97봄학술발표회논문집(A),제24권 1호 p.3-6,1997.

[21] 유병강,이상정, "ILP 프로세서를 위한 조건실행 지원 스케줄링 알고리듬", 한국정보 처리학회논문지 제5권 제1호, pp.202-214, 1998.

이 상 정

1983년 한양대학교 전자공학과 (공학사)

1985년 2월 한양대학교 대학원 전자공학과(공학석사)

1985년 8월 한양대학교 대학원 전자공학과(공학박사)

1988년~현재 순천향대학교 컴퓨터학부 교수

관심 분야 : 프로세서설계, 최적화 컴파일러 설계, 마이크로프로세서 응용