Application에 최적의 ASIP 설계를 위한 효율적인 Architecture Exploration 방법

준회원 이 성 래*, 정회원 황 선 영*

An Efficient Architecture Exploration Method for Optimal ASIP Design

Sung-Rae Lee* *Associate Member*, Sun-Young Hwang* *Regular Member*

요 약

프로세서에 따라 수행 가능한 코드를 생성하는 retargetable 컴파일러와 성능 프로파일러는 어플리케이션에 최적화된 프로세서 디자인에 있어 필수적이다. 본 논문은 ADL (Architecture Description Language)에 기반한 architecture exploration 방법을 제시한다. 어플리케이션 프로그램에서 얻어낸 정보로부터 인스트럭션 합성과 프로세서 구조를 최적화 하였다. 어플리케이션에서 많이 사용되는 연산과 레지스터 사용에 대한 정보는 프로세서 최적화를 위해 사용되었다. 시스템의 효용성을 보이기 위해 JPEG 인코더에 대한 architecture exploration을 수행하였다. 제안된 방법을 사용해 설계된 ASIP은 초기 프로세서에 비해 약 1.97배의 성능을 가지는 것으로 측정되었다.

Key Words : ASIP, ADL, Retargetable 컴파일러, 프로파일, 아키텍처 exploration

ABSTRACT

Retargetable compiler which generates executable code for a target processor and performance profiler are required to design a processor optimized for a specific application. This paper presents an architecture exploration methodology based on ADL (Architecture Description Language). We synthesized instruction set and optimized processor structure using information extracted from application program. The information of operation sequences executed frequently and register usage are used for processor optimization. Architecture exploration has been performed for JPEG encoder to show the effectiveness of the system. The ASIP designed using the proposed method shows 1.97 times better performance.

Ⅰ. 서 론

반도체 제조 공정의 발전과 통신, 멀티미디어 기기의 사용이 증가하면서 임베디드 코어를 가지는 복잡한 시스템의 사용이 증가하고 있다. 임베디드 시스템 설계에 있어 칩 면적 및 소모전력 최소화, 특정 어플리케이션에 대한 실행성능 향상, 짧은 time-to-market의 만족은 갈수록 중요해지고 있다. 또한 스탠다드의 발전 및 소프트웨어 업그레이드에 대처 가능한 높은 programmability도 요구된다. 이런 측면에서 ASIC (Application Specific Integrated Circuits)은 특정 어플리케이션에 최적화된 성능을 낼 수 있으나 한정된 flexibility를 가지며 설계시간이 긴 단점이 있다. ASIP은 특정 어플리케이션만을 위한 인스트럭션 셋과 프로세서 구조를 가져 ASIC의 장점을 가지면서 높은 flexibility를 가진다. 하지

※ 본 논문은 2007년도 「서울시 산학연 협력 사업」의 「나노 IP/SoC 설계 기술 혁신 사업단」의 지원으로 이루어졌습니다.
* 서강대학교 전자공학과 CAD&ES 연구실 (hwang@sogang.ac.kr)
논문번호 : KICS2007-04-181, 접수일자 : 2007년 4월 17일, 최종논문접수일자 : 2007년 8월 30일

만 ASIP 설계간 어플리케이션의 분석을 통한 최적화된 인스트럭션 셋과 프로세서 구조의 선택은 많은 요소를 고려해야 하므로 상당한 시간과 엔지니어링 노력이 요구된다. ASIP 설계간 인스트럭션 변경에 적응 가능한 retargetable 컴파일러를 이용한 특정 어플리케이션의 분석으로 설계시간을 크게 줄일 수 있다[1][2][3]. 머쉰 기술 언어로부터 컴파일러 생성은 생성된 컴파일러를 이용한 architecture exploration의 용이성뿐만 아니라 하나의 기술로 프로세서 설계와 함께 컴파일러를 얻을 수 있어 전체적인 ASIP 설계 능력을 향상시킬 수 있다.

ASIP 설계를 위한 architecture exploration 방식에는 configurable 프로세서 기반 방식[4], ADL 기반 방식[5][6][7]이 제안되었다. Configurable 프로세서 기반 방식은 Tensilica의 Xtensa 가 대표적으로 이미 정해진 기본 프로세서를 기반으로 인스트럭션의 확장과 파라미터의 수정을 통해 어플리케이션에 최적화된 프로세서를 설계하는 방식이다. 하지만 configurable 프로세서 기반 방식은 이미 정의된 기본 프로세서를 기반으로 ASIP을 설계하므로 flexibility가 떨어지는 단점이 있다. ADL 기반 방식은 ADL을 이용해 기술된 프로세서에 맞는 소프트웨어 툴체인을 자동생성하여 architecture exploration을 수행하는 방식이다. ADL 기반 방식은 초기 프로세서 모델로부터 시작하여 어플리케이션 프로그램의 multiple remapping을 통해 어플리케이션에 최적화된 프로세서를 만드는 iterative 방식이다[8]. ADL 기반 방식과 확장 가능한 프로세서 기반 방식모두 디자인 프로세스에 있어 특정 어플리케이션 프로그램을 디자인 하는 프로세서에 맵핑시켜주는 retargetable 컴파일러가 매우 중요하다. ASIP 설계간 어플리케이션의 분석을 통해 어플리케이션에 최적화된 프로세서 구조와 인스트럭션을 구성해야 한다. Retargetable 컴파일러는 상위레벨로 기술된 입력 어플리케이션을 타겟 프로세서 모델에 최적화된 하위레벨 코드로 변환시켜 시뮬레이션을 통한 동적 분석이 가능하게 한다. 또한 컴파일러의 중간 형태를 이용한 입력 어플리케이션 코드의 정적 분석을 통해 입력 어플리케이션의 특성을 분석할 수 있다. 이런 분석을 통해 특정 어플리케이션이 필요로 하는 operation sequence를 분석하여 어플리케이션에 맞는 인스트럭션 선택/합성과 프로세서 아키텍쳐의 구성이 가능하다. 참고문헌 [9]에서는 어플리케이션의 CDFG (Control Data Flow Graph) 에서 많이 발생하는 operation sequence에 대해 "bundling"을 통한 인스트럭션 선택 과 프로세서 데이터 패스의 최적화를 수행한다. 참고문헌 [10]에서는 컴파일러의 중간 형태의 그래프를 프로파일하여 적절한 computational pattern 템플릿을 생성 후 매칭함으로써 configurable의 코드생성을 수행한다. 이들에서는 여러 operation 노드에 하나의 패턴 템플릿을 맵핑시키는 방식을 통해 특정 어플리케이션에 대한 성능향상을 얻을 수 있으나 정적 분석을 통해 맵핑할 패턴을 결정하므로 최적의 결과를 얻을 수 없다. 참고문헌 [11]에서는 configurable 프로세서 기반의 ASIP 설계를 위해 어플리케이션의 자주 실행되는 코드 영역인 "hot-spot"의 분석을 기반으로 어플리케이션에 적절한 인스트럭션 셋과 프로세서 생성을 수행한다. 하지만 configurable 프로세서 기반의 ASIP 설계방식이므로 flexibility가 떨어진다.

본 연구에서는 SMDL[12]을 이용한 ADL 기반 방식의 architecture exploration을 수행한다. 입력 어플리케이션에 대해 프로파일이 가능한 시스템을 구축 하였으며 생성된 컴파일러가 생성한 코드의 시뮬레이션 및 프로파일을 통해 입력 어플리케이션에 최적화된 인스트럭션 셋과 프로세서 구조에 대한 분석을 수행한다. 또한 참고문헌 [13]에서 제시된 레지스터 파일 사이즈에 따른 성능 분석을 실제 어플리케이션에 적용하였다. II장에서는 SRCC (Sogang Retargetable Compiler Compiler) 시스템[14]의 개관을 보인다. III장에서는 제안하는 architecture exploration 시스템을 보인다. IV장에서는 실험결과를 보이며, V장에서는 결론 및 추후과제를 제시한다.

II. SRCC 시스템 개관

SRCC 시스템은 SMDL의 행위정보로부터 라이브러리 맵핑에 용이한 중간 형태를 만들고, 그 중간 형태와 동일한 라이브러리 중간 형태에 맵핑을 함으로써 lburg[15]의 instruction selector description을 생성한다. SMDL만을 입력으로 하는 seamless 컴파일러 생성을 위해 SRCC는 SMDL의 레지스터 파일 사용기술과 라이브러리 맵핑된 인스트럭션을 이용하여 컴파일러의 후위부 interface function을 생성한다. 생성된 코드 선택기와 interface function은 LCC[16][17]의 front-end와 결합하여 SMDL로 기술된 프로세서를 위한 컴파일러를 생성한다. 그림 1은 SRCC 시스템의 개관이다.

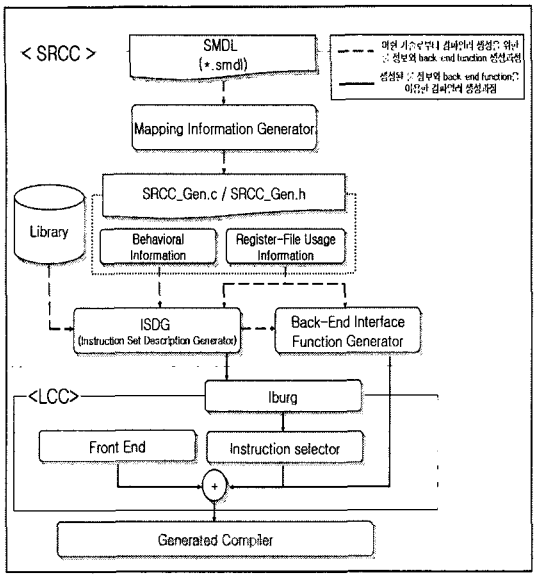

그림 1. SRCC 시스템 개관.

SRCC는 instruction selector description 생성에 있어 LISA와 같은 one-to-one, one-to-many, many-to-one 맵핑 방식을 사용한다[18][19]. 하지만 SRCC 시스템은 LISA와 달리 인스트럭션 행위기술의 라이브러리 맵핑을 통해 LISA에 비해 효율적으로 컴파일러를 자동 생성한다.

One-to-one 맵핑은 단순히 하나의 서브젝트 트리에 하나의 인스트럭션이 맵핑되는 방식이며, one-to-many 맵핑은 하나의 서브젝트 트리에 여러개의 인스트럭션을 맵핑하는 방식이다. One-to-many 맵핑은 특정 서브젝트 트리에 one-to-one 맵핑되는 인스트럭션이 정의되지 않아도 다른 정의된 인스트럭션들의 조합이 서브젝트 트리에 맵핑 가능하게 함으로써 많이 사용되지 않는 특정 트리 패턴만을 위한 인스트럭션 사용을 줄일 수 있다. Many-to-one 맵핑은 ASIP 설계에 있어 매우 중요한 맵핑 방식으로 여러 서브젝트 트리에 하나의 인스트럭션을 맵핑하는 방식이다. Many-to-one 맵핑은 여러 서브젝트 트리를 하나의 인스트럭션으로 맵핑함으로써 코드사이즈를 줄이고 수행속도를 높이며, 이는 특정 어플리케이션을 위한 연산이 많은 ASIP의 특성상 many-to-one 맵핑으로 수행성능을 향상시킬 수 있다.

Ⅲ. Architecture Exploration 시스템

3.1 Architecture Exploration 흐름

본 논문에서는 C로 기술된 입력 어플리케이션을 타겟 프로세서로 수행하였을 시 성능 측정을 위해 SMDL 시스템을 이용한다. SMDL로 기술된 프로세서 모델은 SRCC에 의한 컴파일러 자동 생성을 수행하고 어플리케이션 프로그램은 자동 생성된 컴파일러에 의해 타겟 머쉰 코드로 변환된다. 생성된 코드는 SMDL 시뮬레이터에 의해 수행되고 프로파일 데이터를 생성한다. 프로파일 데이터와 어플리케이션 프로그램을 입력으로 받는 패턴 분석기는 어플리케이션 프로그램 수행시 많은 발생 빈도를 가지는 연산 패턴을 분석한다. 프로파일된 데이터와 연산 패턴의 분석 결과는 SMDL로 기술된 프로세서의 튜닝과 인스트럭션 선택과 합성을 수행한다. SMDL로 다시 기술된 프로세서 모델은 위 과정을 반복하여 어플리케이션에 최적화된 프로세서 모델을 만든다. 그림 2는 제안하는 architecture exploration 흐름의 개관이다.

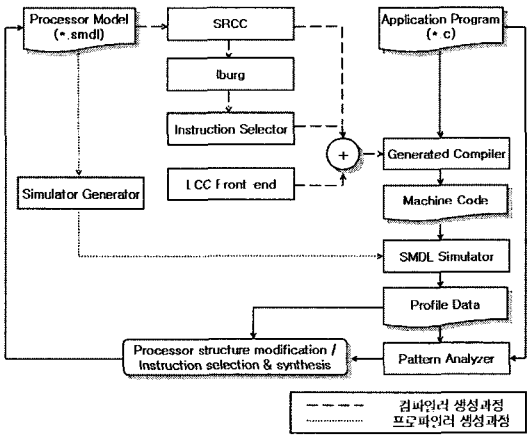

그림 2. Architecture Exploration 흐름의 개관.

3.2 프로파일 데이터

입력 어플리케이션에 대한 프로파일 요소는 표 1과 같다. 모든 프로파일 요소는 어플리케이션을 타겟 프로세서의 시뮬레이터 상에서 얻을 수 있다.

표 1. 시뮬레이션을 통한 프로파일 요소

프로파일 요소	설 명
인스트럭션 frequency	타겟 프로세서의 인스트럭션 수행 빈도
basic block frequency	컴파일된 어플리케이션 코드의 basic block 수행 빈도
conditional branch의 taken/not-taken ratio	조건 분기시 taken / not-taken 비율
어플리케이션 수행 싸이클 수	어플리케이션의 타겟 프로세서 상에서의 전체 수행 싸이클 수

인스트럭션 frequency와 어플리케이션 수행 싸이클 수는 SMDL 시뮬레이터로 프로파일 가능하다.

하지만 basic block frequency와 conditional branch taken/not-taken ratio는 LCC의 수정을 하고 컴파일러가 프로파일 데이터를 만드는 인스트럭션을 어플리케이션 코드에 삽입하여 시뮬레이션시 프로파일 데이터를 얻을 수 있도록 하였다. SRCC 시스템은 어플리케이션의 프로파일을 원할시 lburg의 코드 선택 룰에 프로파일을 위한 코드를 삽입한다. 프로파일을 위한 코드는 타겟 머쉰에 맞는 코드로 seamless하게 자동 생성된다. 그림 3은 basic block frequency와 conditional branch taken/not-taken의 프로파일을 위한 코드생성 룰의 의사코드를 보인다.

```
stmt: LABELV
    /* 레이블 출력 */
    "L.%A :\n
    /* 임의의 장소에 RX 레지스터 저장 */
    sw RX, PROF_REG_BACKUP\n
    /* 현재 basic block label frequency를 가지는 변수를
       RX에 load */
    lw RX, PROF_LABEL_%A\n
    /* 현재 basic block의 frequency 1증가 */
    addi RX, RX, 1\n
    /* 증가된 basic block frequency를 메모리에 저장 */
    sw RX, PROF_LABEL_%A\n
    /* RX 레지스터가 가졌던 값 로드 */
    lw RX, PROF_REG_BACKUP\n"
```

(a)

```
stmt : EQI4(reg,reg)
    /* 임의의 장소에 RX 레지스터 저장 */
    "sw RX, PROF_REG_BACKUP\n
    /* branch 횟수를 가지는 변수를 RX에 load */
    lw RX, PROF_BRANCH_CNT\n
    /* branch 횟수 1증가 */
    addi RX, RX, 1\n
    /* 증가된 branch 횟수를 메모리에 저장 */
    sw RX, PROF_BRANCH_CNT\n
    /* RX 레지스터가 가졌던 값 로드 */
    lw RX, PROF_REG_BACKUP\n
    /* conditional branch 코드 수행 */
    beq %0, %1, %a\n
    /* 임의의 장소에 RX 레지스터 저장 */
    sw RX, PROF_REG_BACKUP\n
    /* branch not-taken된 횟수를 가지는 변수를
    RX에 load */
    lw RX, PROF_NOT_TAKEN_CNT\n
    /* branch not-taken 횟수 1증가 */
    addi RX, RX, 1\n
    /* 증가된 branch not-taken 횟수를 메모리에 저장 */
    sw RX, PROF_NOT_TAKEN_CNT\n
    /* RX 레지스터가 가졌던 값 로드 */
    lw RX, PROF_REG_BACKUP\n"
```

(b)

그림 3. 프로파일을 위한 코드생성 룰의 의사코드.
(a) basic block frequency 프로파일을 위한 코드생성 룰,
(b) Conditional branch의 taken/not-taken ratio 프로파일을 위한 코드생성 룰.

Basic block frequency는 각 basic block label마다 basic block frequency를 갱신하는 인스트럭션을 삽입하고, conditional branch의 taken 횟수는 모든 conditional branch 수행 횟수와 not-taken된 횟수의 차로 구하였다.

3.3 패턴 분석기

패턴 분석기는 어플리케이션 프로그램을 LCC로 컴파일하였을 시 생성되는 LCC의 중간 형태와 어플리케이션의 basic block frequency를 이용해 어플리케이션에 많이 사용되는 연산 패턴을 분석한다. 인스트럭션 사이즈가 무한히 길 수 없어 많은 오퍼랜드를 가지는 인스트럭션을 만들기 어려우므로 패턴 분석기는 3개 이하의 오퍼레이션 노드를 가지는 패턴까지 분석하였다. 정적인 분석을 통해 많이 생기는 패턴이 어플리케이션의 실제 수행시 많이 발생한다고 할 수 없으므로 동적인 분석이 필요하다. 동적인 분석을 통해 얻어진 basic block frequency를 이용해 어떤 패턴의 발생시 그 패턴이 속한 basic block frequency만큼 패턴 발생 빈도에 더해주는 방식으로 패턴 분석을 수행하였다.

Ⅳ. 실험 결과

어플리케이션에 최적의 ASIP 디자인을 위한 architecture exploration 시스템의 검증을 위해 JPEG 인코더에 최적화된 인스트럭션과 프로세서 구조를 설계하였다. JPEG 인코더의 입력으로는 11.6 KB의 BMP 파일을 사용하였다. 초기 프로세서를 입력으로 시뮬레이션을 통해 기본적인 어플리케이션의 특성을 알아냈고 레지스터 파일 사이즈의 최적화를 수행하였다. 최적화된 레지스터 파일 사이즈와 어플리케이션의 패턴 분석을 통해 인스트럭션 셋과 프로세서 구조를 재구성하였다.

4.1 초기 프로세서

초기 프로세서는 MIPS[20][21]에 기반한 프로세서로 선택하고 이를 SMDL언어로 기술하였다. 초기 프로세서는 MIPS와 동일한 IF, ID, EX, MEM, WB의 파이프라인 스테이지를 가지며 나누기 연산은 restoring 알고리즘을 사용한 function을 호출하는 방식으로 구현하였다. 초기 프로세서의 인스트럭션으로 선택된 MIPS 인스트럭션 셋은 표 2와 같다.

표 2. 초기 프로세서의 인스트럭션.

인스트럭션 종류	인스트럭션 이름
load/store instruction	sw, sh, sb, lw, lh, lhu, lb, lbu
computational instruction	addiu, ori, lui, addu, subu, slt, sltu, and, or, xor, mul, srl, srlv, sra, srav, sll, sllv
jump and branch instruction	jr, jalr, bne, beq

Conditional branch는 slt, sltu, bne, beq의 조합으로 수행되며 ID 스테이지에서 branch target address를 계산하고 not-taken방식의 prediction을 채택하였다. 레지스터 파일의 사이즈의 증가에 따른 성능의 변화를 측정하기 위해 초기 프로세서는 11개의 32 비트 레지스터를 가지는 레지스터 파일을 가진다. 표 3은 초기프로세서의 레지스터 주소에 따른 usage를 나타낸다. Argument 전달을 위한 레지스터와 변수값을 저장하는 레지스터를 사용하지 않고 temporary 레지스터는 컴파일된 코드의 안정된 동작을 위한 최소 temporary 레지스터 수인 5개로 정하였다. 초기 프로세서의 컴파일러 생성후 C로 기술된 JPEG 인코더를 생성된 컴파일러로 컴파일한 후 시뮬레이션을 수행하였다.

표 3. 초기 프로세서의 레지스터 파일.

register name	usage
$0	constant 0
$1	reserved for assembler
$2	temporary
$3	temporary
$4	temporary
$5	temporary
$6	temporary
$7	stack pointer
$8	function address
$9	function return value
$10	return address

그림 4는 초기 프로세서로 JPEG 인코더를 수행할시 인스트럭션 frequency를 나타내며, 총 29개의 인스트럭션 중 6개의 인스트럭션이 사용되지 않는다. 초기 프로세서가 JPEG 인코더를 수행하기 위한 최소의 레지스터 파일 사이즈를 가지므로 레지스터

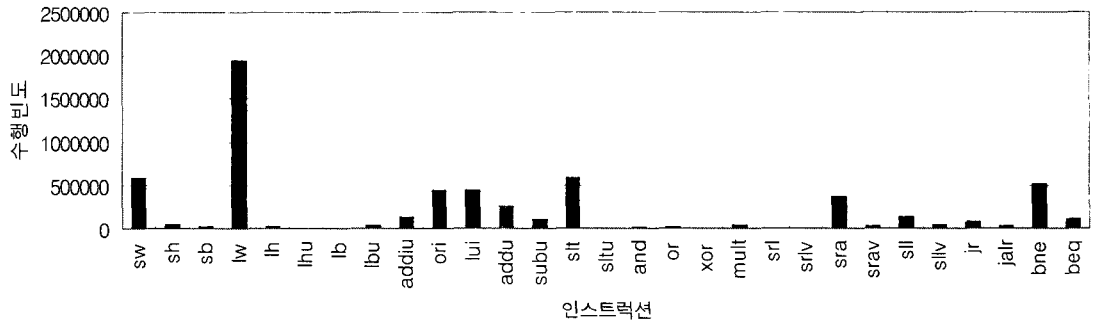

그림 4. 초기 프로세서의 인스트럭션 frequency.

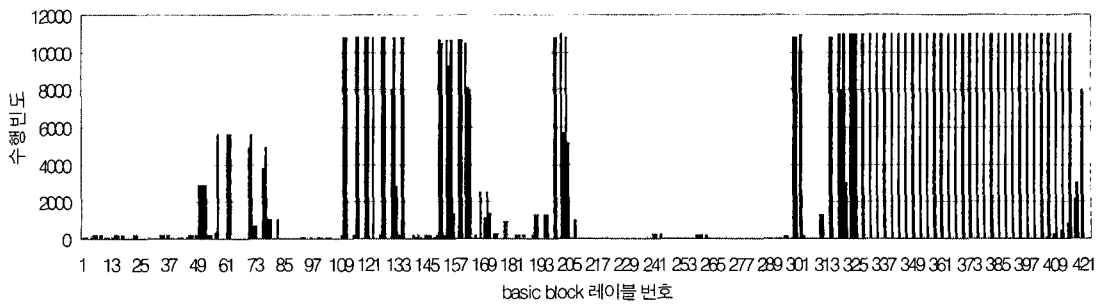

그림 5. JPEG 인코더의 수행시 basic block frequency.

spilling이 많이 일어나 sw, lw가 다른 인스트럭션에 비해 많이 수행된다. 그림 5는 JPEG 인코더의 수행시 basic block frequency를 나타낸다. 가로축은 basic block label이며 세로축은 수행 빈도를 나타낸다. Basic block frequency에서 수행 빈도가 10,000회 이상인 basic block은 전체 basic block에서 약 14.9%를 차지하는 것으로 나타났다. 10,000회 이상인 basic block들의 수행 빈도 합은 전체 basic block 수행 빈도의 81.6%를 차지하는 것으로 분석되었다. 조건 분기는 총 512,596번 taken되었으며 66,033번 not-taken되어 89 : 11의 비율로 taken 횟수가 not-taken 되는 횟수보다 많다.

4.2 레지스터 파일 사이즈 최적화

프로세서 optimization을 위해 JPEG 인코더에 맞는 레지스터 파일 사이즈를 구하였다. JPEG 인코더에 맞는 적절한 크기의 레지스터 파일 사이즈를 구함으로써 인스트럭션 필드에서 레지스터 파일 어드레스 필드 사이즈를 최적화하는데 사용하였다. 그림 6은 argument, temporary, variable 레지스터 수에 따른 JPEG 인코더의 총 수행 싸이클 수를 나타낸다.

Argument 레지스터는 function 호출시 parameter passing을 빠르게 수행되게 하는 레지스터로서, 컴파일러는 먼저 argument 레지스터에 전달하려는 데이터를 저장하고 레지스터를 모두 사용하면 메모리에 저장하게 된다. Argument 레지스터 크기에 따른 JPEG 인코더의 수행 싸이클 수를 보면 argument 레지스터가 2개까지 사용될 시 성능향상이 일어남을 알 수 있다. 이는 사용된 JPEG 인코더의 대부분의 function들이 2개를 넘는 argument를 가지지 않는다는 것을 말한다. Temporary 레지스터는 데이터 연산의 temporary 데이터나 common sub-expression elimination을 수행할 시 common sub-expression의 연산으로 생기는 데이터를 저장하는데 사용된다. Temporary 레지스터 크기에 따른 JPEG 인코더의 수행 싸이클 수는 큰 변화가 일어나지 않는다. 이는 common sub-expression이 hot-spot에서 많이 발견되지 않기 때문으로 분석된다. Variable 레지스터는 variable 데이터를 저장하는데 쓰이며, 이의 수에 따른 JPEG인코더의 수행 싸이클 수를 보면 약 3개에서 4개의 variable 레지스터를 사용하였을 시 성능향상이 크다는 것을 알 수 있다. 결국 초기 프로세서의 레지스터 파일에서 argument 레지스터 1개와 variable 레지스터 4개, 또는 argument 레지스터 2개와 variable 레지스터 3개를 추가하여 총 16개의

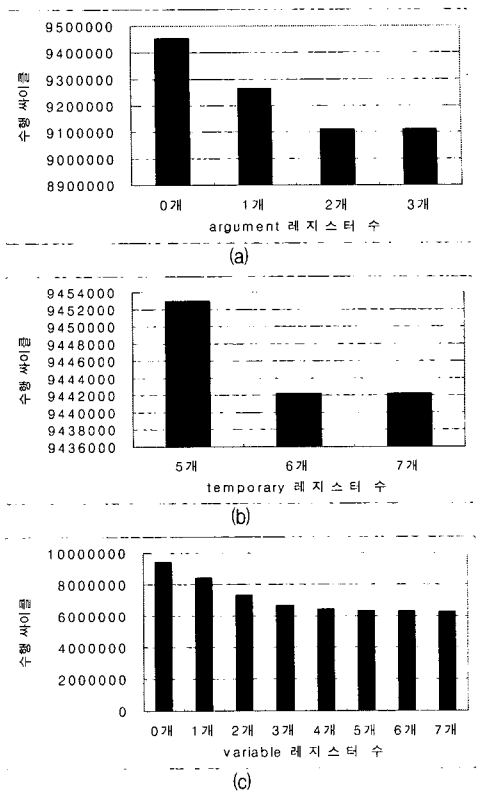

그림 6. 레지스터 파일 크기에 따른 수행 cycle.
(a) Argument 레지스터 수, (b) Temporary 레지스터 수, (c) Variable 레지스터 수에 따른 싸이클 수.

표 4. 최종적으로 선택된 레지스터 파일 usage.

register name	usage
$0	constant 0
$1	argument
$2	reserved for assembler
$3	temporary
$4	temporary
$5	temporary
$6	temporary
$7	temporary
$8	register variable
$9	register variable
$10	register variable
$11	register variable
$12	stack pointer
$13	function address
$14	function return value
$15	return address

레지스터 파일 크기로 JPEG에 최적화된 성능을 낼 수 있다. 이는 기존 MIPS의 레지스터 파일을 5 비트로 access하는데 비해 4 비트로 레지스터 파일을 access가능하게 할 수 있어 many-to-one 맵핑을 위해 필요로 하는 operand수로 인한 overhead를 감소

시킬 수 있다. 표 4는 최종적으로 선택된 16 비트 레지스터 파일의 usage를 보인다.

4.3 인스트럭션 선택/합성 및 프로세서 최적화

패턴 분석기를 통해 JPEG 인코더에서 많이 활용되는 패턴을 분석하였다. 표 5는 패턴 분석기가 출력한 패턴 중 패턴 frequency가 높고 인스트럭션 합성 가능한 일부 패턴을 보인다. 각각의 패턴은 LCC의 중간형태의 연산 패턴을 나타내며 각 연산 패턴에 맞는 오퍼레이션을 하는 인스트럭션을 many-to-one 맵핑을 통해 성능향상을 얻을 수 있다. 또한 many-to-one 맵핑된 인스트럭션 합성을 위해 데이터 패스의 재구성이 필요하다. 표 6은 많이 사용된 패턴들에 대해 many-to-one 맵핑을 위해 만든 인스트럭션을 보인다. Operation의 교환법칙이 성립하는 경우 여러 가지 패턴 셋이 하나의 인스트럭션에 맵핑될 수 있다. 그림 7은 새롭게 추가된 인스트럭션의 수행이 가능하도록 MIPS의 EX 스테이지를 재구성한 결과이다. MIPS는 여러 인스트럭션의 조합으로 분기 인스트럭션을 표현 하였지만 JPEG 인코더의 ASIP은 이를 사용하지 않았다. 거의 모든 분기가 taken되므로 ID 스테이지에서 branch target 주소를 계산하여 일단 분기하는 predict taken방식을 사용하였으며 분기 판단은 EX 스테이지의 ALU에서 이루어지도록 하였다. 이렇게 함으로써 기존 MIPS에 비해 분기가 not-taken 될 시 많은 싸이클 수의 손해를 보지만 not-taken될 확률이 크지 않고 여러 인스트럭션을 사용해 분기를 수행하지 않아 분기에 따른 overhead를 줄일 수 있다.

ADL 기반의 architecture exploration 방식은 전체적인 프로세서 구조를 어플리케이션에 최적화되고 유연하게 바꿀 수 있다는 점에서 configurable 프로세서 기반의 architecture exploration 방식보다 우수하다.

표 5. 패턴 및 패턴 frequency.

번호	패 턴	패턴 frequency
1	LTI4(RSHI4(reg,con1),reg)	329,520
2	ADDP4(LSHI4(reg,con1),reg)	122,544
3	RSHI4(SUBI4(reg,reg),con1)	21,466
4	RSHI4(ADDI4(reg,reg),con1)	16,390
5	GEI4(reg,NEGI4(reg))	10,816
6	RSHI4(MULI4(con2,reg),con1)	10,816
7	SUBI4(reg,MULI4(con2,reg))	8,112
8	ADDI4(reg,MULI4(con2,reg))	8,112
9	RSHI4(ADDI4(reg,con1),con1)	6,216
10	ADDI4(MULI4(con1,reg),reg)	5,408
11	ADDI4(MULI4(con2,reg),reg)	2,704
(reg: register 데이터, con1: 1-byte 상수, con2: 2-byte 상수)		

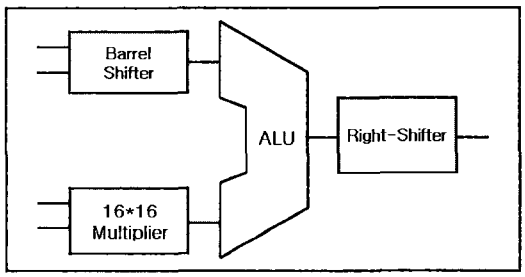

그림 7. 제안된 ASIP의 EX 스테이지 데이터 패스 구조.

4.4 최종 결과

초기 프로세서, 레지스터 파일 크기를 최적화한 프로세서, 인스트럭션 선택과 합성과 프로세서 구조 최적화한 최종 ASIP 프로세서의 성능을 비교하기 위해 서로 다른 BMP 파일 A, B, C를 입력으로 하는 JPEG인코더의 시뮬레이션을 수행하였다. 시뮬레이션 결과 표 7의 결과를 얻었다. 최종적으로 설계된 ASIP은 초기 프로세서에 비해 평균 1.97배, 레

표 6. 합성된 인스트럭션 셋.

번호	인스트럭션	설 명	맵핑되는 패턴 번호
1	sra-blt rs1, rs2, shamt, offset	if((rs1>>shamt) < rs2) PC = PC + (offset << 2)	1
2	sll-add rd, rs1, rs2, shamt	rd = (rs1 << shamt) + rs2	2
3	sub-sra rd, rs1, rs2, shamt	rd = (rs1 - rs2) >> shamt	3
4	add-sra rd, rs1, rs2, shamt	rd = (rs1 + rs2) >> shamt	4
5	addi-sra rd, rs1, shamt, imm	rd = (rs1 + imm) >> shamt	9
6	neg-bge rs1, rs2, offset	if(rs1 >= -rs2) PC = PC + (offset << 2)	5
7	muli-sra rd, rs1, shamt, imm	rd = (rs1 * imm) >> shamt	6
8	muli-sub rd, rs1, rs2, imm	rd = rs1 - (rs2 * imm)	7
9	muli-add rd, rs1, rs2, imm	rd = rs1 + (rs2 * imm)	8, 10, 11

지스터 파일 크기를 최적화한 프로세서에 비해 평균 1.29배의 성능 향상을 얻었다.

표 7. 프로세서별 JPEG 인코더 수행성능. (싸이클 수)

데이터	초기 프로세서	레지스터 파일 크기 최적화한 프로세서	최종 ASIP
A	9,453,014	6,335,103	4,608,482
B	8,917,920	5,651,228	4,411,565
C	6,635,496	4,399,221	3,570,030

Ⅵ. 결론 및 추후과제

본 논문은 ADL 기반의 architecture exploration 방법에 대해 기술하였다. SRCC는 타겟 프로세서에 맞는 컴파일러의 자동 생성으로 architecture exploration을 빠르게 진행되도록 하며 다양한 맵핑 방식의 지원으로 어플리케이션에 최적화된 인스트럭션 셋을 결정할 수 있도록 한다. 패턴 분석기의 설계 및 시뮬레이션을 통해 입력 어플리케이션에 대한 프로파일이 가능하도록 하였으며 이를 이용해 다양한 관점에서 어플리케이션의 특성을 분석함으로써 전체적인 architecture exploration 과정이 체계적이고 빠르게 진행되었다. 본 논문에서 보인 방법을 이용하여 JPEG 인코더에 최적화된 ASIP을 설계하였으며, 설계결과 JPEG 인코더를 위한 인스트럭션 선택과 합성을 통해 초기 프로세서에 비해 약 1.97배의 수행 성능 향상을 얻었다.

추후과제는 architecture exploration을 더욱 가속화시킬 수 있는 scheduler에 기반한 성능 측정기의 개발이 요구된다.

참 고 문 헌

[1] P. Marwedel, "Code Generation for Embedded Processors : An Introduction", in *Code Generation for Embedded Processors*, P. Marwedel and G. Goosens, ed., pp. 14-31, Kluwer Academic Publishers, 1995.

[2] C. Liem, *Retargetable Compilers for Embedded Core Processors*, Kluwer Academic Publishers, 1997.

[3] R. Leupers, "Compiler Design Issues for Embedded Processors", IEEE Design & Test of Computers, Vol. 19, No. 4, pp. 51-58, July-Aug. 2002.

[4] R. Gonzalez, "Xtensa : A Configurable and Extensible Processor", IEEE Micro, Vol. 20, No. 2, pp. 60-70, March-April 2000.

[5] A. Hoffmann et al, "A Novel Methodology for the Design of Application-Specific Instruction Set Processors(ASIPs) Using a Machine Description Language", IEEE Trans. Computer-Aided Design, Vol. 20, No. 11, pp. 1338-1354, Nov. 2001.

[6] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer, "The MIMOLA Language Version 4.1", Technical report, University of Dortmund, 1994.

[7] A. Fauth, J. Van Praet, and M. Freericks, "Describing Instructions Set Processors Using nML", in Proc. European Design & Test Conf., Paris (France), pp. 503-507, Mar. 1995.

[8] R. Leupers et al, "Retargetable Compilers and Architecture Exploration for Embedded Processors", IEE Proc. Computers and Digital Techniques, Vol. 152, No. 2, pp. 209-223, Mar. 2005.

[9] J. Van Praet, G. Goosens, D. Lanneer, and H. De Man, "Instruction Set Definition and Instruction Selection for ASIPs", in Proc. Int. Symp. High-Level Synthesis, pp. 11-16, May 1994.

[10] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems", in Proc. Int. Conf. CAD 2001, pp. 127-130, Nov. 2001.

[11] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey, "A Design Flow for Configurable Embedded Processors Based on Optimized Instruction Set Extension Synthesis, in Proc. Design Automation & Test in Europe, Munichi, Germany, Mar. 2006.

[12] 조재범, 유용호, 황선영, "임베디드 프로세서 코어 자동생성 시스템의 구축", 한국통신학회논문지, 30권 6A호, pp. 526-534, 2005년 6월.

[13] L. Wehmeyer, M. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan, "Analysis of the Influence of Register File Size on Energy Consumption, Code Size, and Execution Time",

IEEE Trans. Computer-Aided Design, Vol. 20, No. 11, pp. 1329-1337, Nov. 2001.

[14] 이성래, 황선영, "머쉰 행위기술로부터 Retargetable 컴파일러 생성시스템 구축", 한국통신학회논문지, 32권 5호, pp. 286-294, 2007년 5월.

[15] C. Fraser, R. Henry, and T. Proebsting, "BURG - Fast Optimal Instruction Selection and Tree Parsing", ACM SIGPLAN Notices, Vol. 27, No. 4, pp. 68-76, April 1992.

[16] C. Fraser and D. Hanson, *A Retargetable C Compiler : Dsign and Implementation*, Ben-jamin/Cummings, 1995.

[17] C. Fraser and D. Hanson, "The lcc 4.x Code-Generation Interface", Microsoft-Research, 2003.

[18] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun, "C Compiler Retargeting Based on Instruction Semantics Models", in Proc. Design Automation and Test in Europe, pp. 1150-1155, 2005.

[19] J. Ceng, W. Sheng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun, "Modeling Instruction Semantics in ADL Processor Descriptions for C Compiler Retargeting". in Proc. Int. Workshop Systems, Architectures, Modeling, and Simulation (SAMOS), 2004.

[20] G. Kane and J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.

[21] D. Patterson and J. Hennessy, *Computer Organization & Design*, Morgan Kaufmann, 1992.

이 성 래 (Sung-Rae Lee) 준회원

2006년 7월 서강대학교 전자공학과 졸업

2006년 8월~현재 서강대학교 전자공학과 석사과정

<관심분야> 고성능 프로세서 및 optimizing compiler 설계, ASIP design methodology, multimedia application용 하드웨어 설계 등.

황 선 영 (Sun-Young Hwang) 정회원

1976년 2월 서울대학교 전자공학과 졸업

1978년 2월 한국과학원 전기 및 전자공학과 공학석사 취득

1986년 10월 미국 Stanford 대학교 전자공학 박사학위 취득

1976년~1981년 삼성 반도체(주) 연구원, 팀장

1986년~1989년 Stanford대학 Center for Integrated Systems 연구소 책임 연구원 및 Fairchild Semiconductor, Palo Alto Research Center 기술자문

1989년~1992년 삼성전자(주) 반도체 기술자문

2002년 4월~2004년 3월 서강대학교 정보통신대학원장

1989년 3월~현재 서강대학교 전자공학과 교수

<관심분야> SoC 설계 및 framework 구성, CAD 시스템, Embedded System, DSP System 설계 등