응용 프로그램에 특화된 명령어를 통한 고정 소수점 오디오 코덱 최적화를 위한 ADL 기반 컴파일러 사용

안 민 욱[†] · 백 윤 흥[††] · 조 정 훈[†††]

요 약

빠른 디자인 공간 탐색 (Design space exploration)은 응용 프로그램의 동작을 구현하기 위한 임베디드 시스템을 디자인하는데 매우 중요하다. Time-to-market이 디자인의 주관심사가 되어감에 따라 ASIP(Application specific instruction-set processor)에 기반한 접근 방식이 디자인 방법론적으로 중요한 대안이 되고 있다. 이러한 접근 방식에서는 타깃 프로세서의 ISA(Instruction set architecture)를 코드 크기와 실행 속도 측면에서 응용 프로그램에 가장 적합하도록 변경한다. 본 논문의 목적은 우리의 새로운 재겨냥성 컴파일러를 소개하고, 많이 알려진 디지털 신호 처리용 응용 프로그램을 위한 ASIP 기반 디자인 공간 탐색에서 컴파일러가 어떻게 활용될 수 있는지 설명하고자 하는 것이다. 새롭게 개발된 재겨냥성 컴파일러는 이전의 재겨냥성 컴파일러의 기능을 제공할 뿐만 아니라 application 프로그램의 특징을 시각화하고 application 프로그램의 프로파일된 결과를 제공하므로 application의 성능을 증가시키기 위해 어떤 명령어들을 넣어야 하는지를 결정하는데 도움을 준다. 재겨냥성 컴파일러의 ADL (Architecture description language)를 이용하여 타깃 프로세서의 초기 RISC-style ISA을 기술하고, 컴파일러가 응용 프로그램을 위한 어셈블리 코드를 더 최적화할 수 있도록 응용 프로그램에 특화된 명령어를 ISA에 점진적으로 추가해 나간다. AC3 오디오 codec을 위한 실험 결과로부터 우리는 32%의 성능 증가와 20%의 프로그램 크기 감소를 얻을 수 있는 6개의 새로운 특화 명령어를 빠르게 찾을 수 있었다. 따라서 우리는 고성능의 재겨냥성 컴파일러는 특정 응용 프로그램을 위한 새로운 ASIP의 빠른 디자인을 하기 위한 중요한 핵심이라는 것을 확인할 수 있었다.

키워드 : 재겨냥성 컴파일러, 아키텍처 기술 언어, AC-3

Using a H/W ADL-based Compiler for Fixed-point Audio Codec Optimization thru Application Specific Instructions

Minwook Ahn[†] · Yunheung Paek[††] · Jeonghun Cho[†††]

ABSTRACT

Rapid design space exploration is crucial to customizing embedded system design for exploiting the application behavior. As the time-to-market becomes a key concern of the design, the approach based on an application specific instruction-set processor (ASIP) is considered more seriously as one alternative design methodology. In this approach, the instruction set architecture (ISA) for a target processor is frequently modified to best fit the application with regard to code size and speed. Two goals of this paper is to introduce our new retargetable compiler and how it has been used in ASIP-based design space exploration for a popular digital signal processing (DSP) application. Newly developed retargetable compiler provides not only the functionality of previous retargetable compilers but also visualizes the features of the application program and profiles it so that it can help architecture designers and application programmers to insert new application specific instructions into target architecture for performance increase. Given an initial RISC-style ISA for the target processor, we characterized the application code and incrementally updated the ISA with more application specific instructions to give the compiler a better chance to optimize assembly code for the application. We get 32% performance increase and 20% program size reduction using 6 audio codec specific instructions from retargetable compiler. Our experimental results manifest a glimpse of evidence that a highly retargetable compiler is essential to rapidly prototype a new ASIP for a specific application.

Key Words : Retargetable Compiler, Architecture Description Language, AC-3

1. 서 론

디지털 오디오는 디지털 신호 처리에서 중요한 분야 중 하나로 여겨진다. 과거 오디오 산업에서는 디지털 오디오 데이터 압축 기술이 주목 받았었다. 데이터 압축 기술은 오디오 신호를 방송하거나 복원 가능한 범위에서 정보를 줄여 좀 더 효율적으로 기록하는 것이 가능하다. 탁월한 성능을

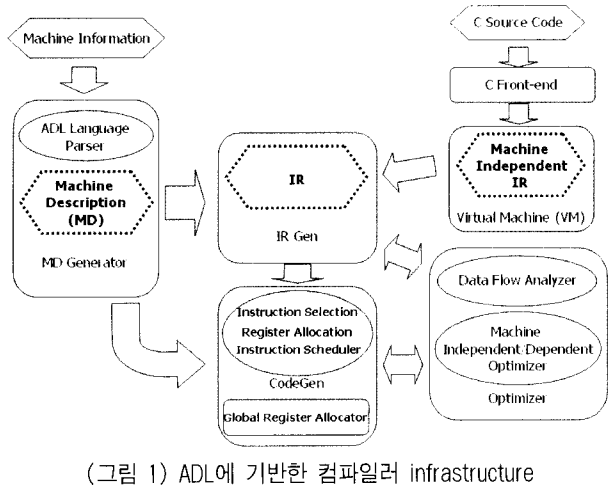
※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT 연구센터 지원사업 (IITA-2005-C1090-0502-0031), 한국학술진흥재단 젊은 과학자 연구활동 지원사업(D00191), 정보통신부 선도기반 기술개발 사업(A1100-0501-0004), 과학기술부 System2010 사업(M103BY010004-05B2501-00411), 정보통신부의 출연금 등으로 수행한 정보통신연구개발사업, 서울시 산학연 협력사업의 나노 IP/SoC설계기술혁신사업단의 지원으로 이루어졌습니다.

† 정 회 원 : 서울대학교 전기컴퓨터공학부 박사과정
†† 정 회 원 : 서울대학교 전기컴퓨터공학부 부교수
††† 정 회 원 : 경북대학교 전자전기컴퓨터학부 전임강사
논문접수 : 2006년 4월 20일, 심사완료 : 2006년 7월 2일

발휘하는 오디오 데이터 압축 기술 중 하나인 AC-3 는 유럽에서 ATSC(advanced television systems committee)에 의해 HDTV를 위한 오디오 서비스 부분 표준으로 채택되었다[2]. AC-3 압축 기술이 점점 중요해짐에 따라, 보다 많은 반도체 제작업체들은 최근 임베디드 시스템에서 많이 쓰이고 있는 system-on-a-chip(SoC) 형태로 AC-3를 구현하려 노력하고 있다.

AC-3가 포함된 오디오 코덱 알고리즘은 개선된 많은 기술들이 새로이 개발되고 발표되면서 계속적으로 향상되고 있다[2]. 따라서 집적회로를 통해 전체 알고리즘을 구현하는 것은 알고리즘이 변경되었을 경우에 기존의 알고리즘을 재활용할 수 없고 다시 제작해야 하므로 전체적인 time-to-market을 증가시키는 원인이 된다. 이 문제에 대한 대안으로는 ASIP들과 같이 시장에서 인기 있을 것이라 판단되는 임베디드 시스템의 맞춤형 디자인을 위한 다양한 programmable IP들을 고려해 볼 수 있다. ASIP이 AC-3 같은 고정된 응용 프로그램을 위해 디자인 될 때에는 응용프로그램의 성능을 증가시키기 위해서 어떤 아키텍처 디자인을 선택할 것인가에 대한 탐색(Design Space Exploration-DSE)이 필요하다. 성능에 대한 요구 사항을 만족시키는 디자인을 빠르게 찾기 위해서는 시뮬레이터나 컴파일러와 같은 ASIP 디자인 도구들은 필수적이다. 이는 다양한 하드웨어 구성에 대하여 커다란 응용 프로그램을 타깃하고 그리고 최적의 구성이 결정될 때까지 시뮬레이션을 수행하기 위해 필요하기 때문이다. 이 도구들을 자동으로 생성하기 위해 연구자들은 하드웨어 ADL(architecture description language) [16]라고 하는 몇몇 형식 언어를 제안하였다. 사용자들은 ADL을 사용하여 MD(machine description)를 작성함으로써 타깃 머신을 위한 도구들을 자동으로 얻을 수 있다.

본 연구에서 우리는 ADL에 기반을 둔 고성능 재겨냥성 컴파일러에 대하여 특히 관심을 가지고 있다. 다양한 이전의 연구들[7, 16, 10, 12, 16, 17]은 ADL 기반의 컴파일러가 각 하드웨어 구성의 개발을 빠르고 정확하게 할 수 있게 컴파일 된 코드를 제공함으로써 디자인 공간 탐색에서 없어서는 안될 중요한 부분임을 보여준다. ADL 기반 컴파일러의 중요성을 검증하기 위해서 우리는 최근에 AC-3 오디오 디코더를 동작시키기 위한 타깃 프로세서의 최적화된 ISA를 찾기 위해 ADL 기반의 컴파일러를 실험 대상으로 연구하였다. 불행하게도 대부분 존재하는 ADL기반의 컴파일러는 상업용이기 때문에 무료로 연구 목적에 사용할 수 없었다. 그래서 이번 연구를 위해 (그림 1)에서 보이는 바와 같이 ADL을 포함하고 있는 새로운 재겨냥성 컴파일러를 개발하였다. 다른 하드웨어 ADL과 마찬가지로 우리의 ADL도 시스템 수준의 아키텍처를 기술하기 위해 고 수준의 추상화(high-level abstraction)를 제공함과 동시에 아키텍처에 관한 저 수준의 상세함은 사용자에게 숨겼다.



(그림 1) ADL에 기반한 컴파일러 infrastructure

이 실험에서 우리는 타깃머신으로 단순한 명령어의 RISC 스타일 ISA를 가진 고정 소수점 프로세서(fixed-point processor)를 최초 아키텍처로 가정하였다. 그리고 초기 ISA를 ADL에 기술함으로써 컴파일러는 자동적으로 그 프로세서에 타겟팅 되고 결과적으로 AC-3 코드를 컴파일하기 위해 사용될 수 있다. 동시에 AC-3 코드 중 실행시간에 자주 사용되는 연산 패턴들의 그룹을 찾아내는 분석도 수행된다. 이러한 분석 결과로부터 발견된 패턴은 AC-3에 특화된 명령을 구분해 내는데 적용된다. 이렇게 찾아낸 명령어들은 처음 ISA에 계속해서 추가되고 새롭게 추가된 명령어에 의해 발생하는 성능 향상을 예측하기 위해 컴파일러를 재구성하게 된다.

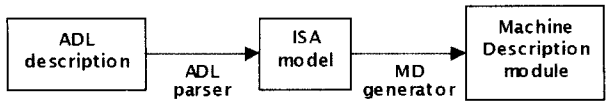
우리의 최근 연구를 설명하기 위해 2장에서는 우리가 개발한 컴파일러와 ADL을 소개한다. 3장에서는 AC-3 알고리즘에 대해 설명하고 4장에서는 응용 프로그램에 특화된 머신 명령어 탐색을 용이하게 하기 위해 고안된 두 개의 그래프에 기반한 자료 구조를 설명한다. 각각은 응용 프로그램 코드와 타깃 머신의 기능적 구조를 요약한다. 5장에서는 AC-3에 특화된 명령어의 코드 크기와 속도에서의 상당한 개선을 얻은 결과를 보여준다. 6장과 7장에서는 간단하게 우리의 연구성과를 다른 연구들과 연관성을 찾고 결론을 도출한다.

2. ADL 기반 컴파일 프레임워크

본 장에서 먼저 우리의 재겨냥성 컴파일러의 전체 구조에 대해 논의한다. 그리고 주어진 ISA가 어떻게 ADL로 표현되는지와 ISA에 맞는 컴파일러를 생성하기 위해 어떻게 ADL이 사용되는지를 보이기 위해 예제를 통해서 우리의 ADL을 설명한다.

2.1 컴파일러 개요

(그림 1)은 타깃 아키텍처의 ADL 명세(description)에 의해 수행될 수 있는 우리의 재겨냥성 컴파일러 구조를 나타낸다. ADL은 아키텍처의 구조적 정보와 수행적(behavioral) 정보를 명기함으로써 아키텍처의 특성을 기술한다. 현재에도 그 개념은 계속해서 확장되고 있지만 구조적 정보는 레지스터와 메모리 구조를 포함하고, 수행적 정보는 머신 명



(그림 2) 코드 생성을 위한 ADL 변환 과정

령어와 어드레싱 모드를 포함한다.

컴파일러는 MachineDescription(MD), VirtualMachine(VM), IRGen, CodeGen, GlobalRegAllocator, 그리고 Optimizer와 같은 몇 개의 C++ 모듈을 통해 구현된다. MD 모듈은 컴파일러에 필요한 머신 특성 정보를 전달하는 C++ 루틴의 집합이다. (그림 2)와 같이 ADL의 아키텍처 명세로부터 MD 모듈이 자동으로 생성되고, MD 모듈은 CodeGen 모듈의 입력이 된다. CodeGen 모듈은 MD 모듈을 명령어 선택(instruction selection), 레지스터 할당 그리고 명령어 스케줄링 단계의 머신 명령어 템플릿들의 집합으로 사용한다.

ISA 모델은 MD 루틴들에서 사용되는 ISA 템플릿을 구성하는 C++ 자료구조들의 집합이다. 이것을 구성하는 두 개의 주요한 컴포넌트는 resource와 operation이다. Resource 컴포넌트는 레지스터와 메모리 같은 저장 요소를 나타낸다. Operation 컴포넌트는 타깃 ISA를 추상화한다. 각각의 명령어, 어드레싱 모드에 관한 명세는 ISA 모델에서 명령어 템플릿으로 변환된다. 명령어 템플릿의 속성 가운데 액션 템플릿은 트리 모형의 레지스터 전달 리스트, 즉 RTL(register transfer level) 행동을 표현한다. CodeGen 모듈은 명령어 선택을 위해 이 트리 모형 템플릿을 바로 사용할 수 있다.

MD 생성기는 ISA 모델로부터 MD 모듈을 생성한다. 효과적인 코드 생성을 위해서는 다양한 매개 변수들이 필요하기 때문에 MD 발생기는 ISA 모델을 분석하고 레지스터 클래스와 레지스터 전달 그래프[4]와 같은 필수적인 정보를 추출한다. VM 모듈은 C front-end[1] 와 back-end 컴파일러간의 범용적인 인터페이스를 제공한다. VM은 ISA와 같은 어셈블리를 가진 가상 머신이다. 가상 어셈블리는 RTE(register transfer expression) 리스트의 조합으로 이루어진다. 각 RTE는 rvlaue 표현이 lvalue로 대입되는(**set lvalue rvalue**) 형태의 단일 명령어와 일치한다. 표현식에 있는 연산자는 그 타입에 따라 단항과 이항 연산자이고 피연산자(operand)는 심볼릭 레지스터나 메모리의 위치가 될 수 있다.

다음은 가상 어셈블리의 예이다.

```
L8: (set (SI: r2) (SI: 12(fp)))    // fp: frame pointer
    (set (SI: r3) (SI: 0(r2)))
    (set (SI: r4) (SI: 4(fp)))
    (set (SI: r2) (SI: 0(r4)))
    (set (SI: r2) (mult:SI (SI: r3) (SI: r2)))
    (set (SI: r3) (SI: 16(fp)))
    (set (SI: r3) (ss_plus:SI (SI: r3) (SI: r2)))
    (set (SI: 16(fp)) (SI: r3))
```

가상 어셈블리는 매우 직관적이다. 예를 들어, 첫 번째 줄의 의미는 'fp에 12를 더한 메모리의 값을 r2로 로드하라'이다. 모든 가능한 머신 독립적 최적화는 가상 어셈블리의 front-end에서 수행 가능하다.

사용자가 응용 프로그램 코드를 컴파일 할 때 먼저 가상 어셈블리가 생성된 후 CSE(common sub-expression elimination)와 CFA(control flow analysis)를 통해 그래프 구조의 중간 언어(intermediate representation, IR)로 변환된다. 그리고 CodeGen과 GlobalRegAllocator 모듈이 MD모듈의 루틴을 사용하여 IR을 타깃 코드로 생성한다.

IR은 계층적 그래프 구조를 가지고 있다. 각 기본 블록(Basic block) 노드는 상호의존적인 RTE 집합을 나타내는 트리와 DAG을 포함하는 forest이다. 같은 함수의 모든 기본 블록은 함수 노드를 나타내기 위해 CFG(control flow graph)를 형성한다. 마지막으로, 프로그램의 모든 함수 노드는 전체 프로그램을 나타내는 call graph를 형성한다. 전체적인 계층적 구조를 시각화하기 위해 IR과 통합된 graph visualization tool을 구현하였다. (그림 2)에서 보는 바와 같이 이 툴을 이용하여 소스 코드 분석과 특정 프로세스로 타겟팅된 컴파일러 모듈의 디버깅에 많은 도움을 얻을 수 있었다. IR은 동일한 인터페이스를 통해 모든 컴파일러 모듈들이 서로 결합한다. 예를 들어 Optimizer 모듈에 있는 reaching definition과 live range analysis와 같은 데이터 흐름 분석 기법들은 IR의 코드상에서 수행된다.

IR 계층의 최하위 레벨에서 RTE를 포함한 DAG를 Expression DAG(EDAG)라 한다. EDAG는 operator들과 그 operator들이 만들고 사용하는 value 사이의 데이터 의존성을 나타낸다. 이런 의미에서 EDAG의 노드는 크게 두 가지 타입, operator와 value로 구분된다. value 노드는 더 나가 symbolic variable, memory location, effective address, 그리고 constant의 네 가지로 나눠진다.

2.2 머신 서술을 위한 ADL

앞장에서 언급했듯이 ADL의 주 목적은 ISA의 정확성과 완벽성을 검증할 수 있게 타깃 프로세서를 서술하기 위한 정형화된 방법을 제공하는 것이다. 따라서 우리의 ADL은 엄격한 형식론(formalism)에 근거하여 정의되었다.

[정의] $ISADesc = <IS, AM, ST, R_{IA}, R_{AS}>$, *여기에서*
IS : 계층적으로 구성된 명령어 집합
AM : 타겟 아키텍처의 어드레싱 모드 집합
ST : 타겟 아키텍처의 저장장치 집합
$R_{IA} \subseteq IS \times AM^n$, *여기에서* $n > 0$
$R_{AS} \subseteq AM \times ST^n$, *여기에서* $n > 0$

BNF 표기법으로 서술된 ISADesc를 위한 ADL 문법의 중요한 부분은 부록 A에서 볼 수 있다. ADL은 primitives, storage, addressMode, 그리고 instruction와 같이 네 가지 섹션으로 구성된다. (그림 3)은 primitive 와 storage 섹션의 예를 보여준다. primitive 섹션은 기본적인 연산과 타입을 정의한다. 각 기본적인 연산은 타깃 머신의 더 이상 나눠질 수 없는 기본 동작(atomic behavior)을 상징한다. 예를 들면,

1) 이 실험에서는 컴파일러의 front end로 GCC를 사용하였다.

```
primitive {
    type { qi 8; hi 16; si 32; byte 8; halfword 16; word 32; }
    operation {
        ss_plus; ss_minus; mult; div; ··· mem_write; jump;
     }
}
storage  {
    memory qi dmem {
          latency 1 @[0x0000..0x9999];
    }
    register si reg[16];
    programCounter r15 PC;
    stackPointer r13 SPR;
}
```

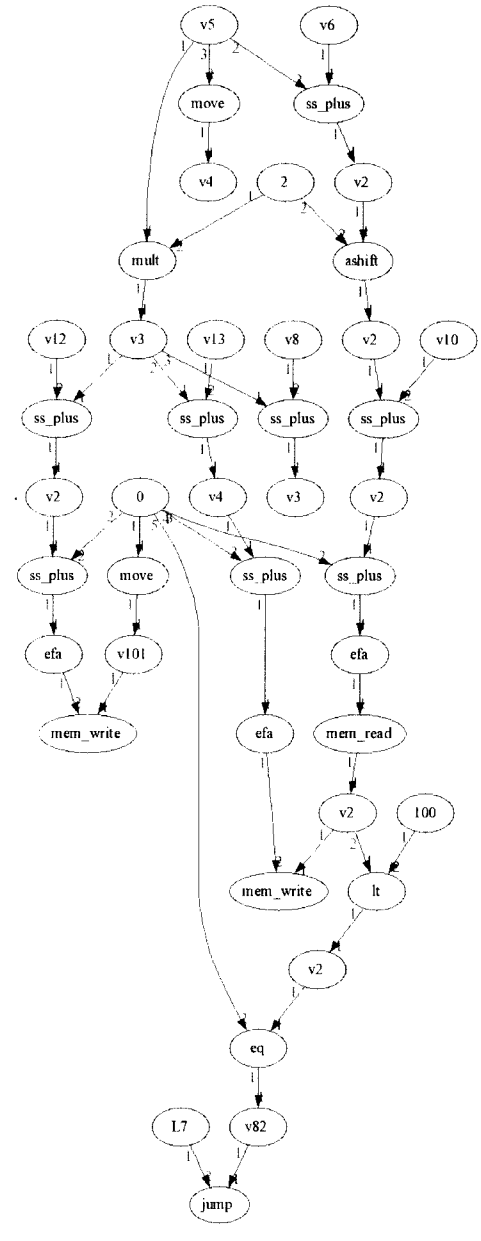
(그림 3) ADL에서 선언된 primitive 연산과 storage

```
addressModeSet   dataAddrModel : dataAddressMode {
      regAddrModel;      imm4AddrMode;
}
addressModeSet   dataAddrMode2 : dataAddressMode {
      regAddrMode2;      imm8AddrMode;
}
addressMode   regAddrModel : dataAddrModel {
      reg r3;       action {   r3;   }       syntax {  r3;  }
cost(0);
}
addressMode   imm4AddrMode : dataAddrModel {
      integer(4) int4;   action { int4; }    syntax { int4; }
cost(0);
}
addressMode   regAddrMode2 : dataAddrMode2 {
      reg r3;       action {   r3;   }       syntax {  r3;  }
cost(0);
}
addressMode   imm8AddrMode : dataAddrMode2 {
      integer(8) int8;    action {  int8; }   syntax { int8; }
 cost(0)
}instruction   addsl1: MultipleOps {
         reg  r1;
         dataAddrModel opn1;
         dataAddrMode2 opn2;
         action {   r1 = ashift(ss_plus(r1, opn1), opn2));   }
         syntax {   " addsl "::r1::", "::opn1::", "::opn2;   }
         cost(1);
}
instruction   addsl2 : MultipleOps {
         reg  r1, r2, r3;
         integer(8)  imm8;
         action {   r1 = ashift(ss_plus(r2, r3), imm8));   }
         syntax {   " addsl "::r1::", "::r2::", "::r3::", "::imm8;   }
         cost(1);
}
```

(그림 4) ADL로 기술된 두 개의 add-shift 명령어

(그림 3)의 primitive ss_minus 와 div는 하드웨어에서 뺄셈과 나눗셈을 상징한다. 타입 정보 또한 primitive 섹션에서 표현된다. 예를 들어, si 는 32-bit integer 타입을 표현한다. storage 섹션은 아키텍처의 추상 자원 구조(abstract resource structure)를 기술한다. 각 storage 구성요소들은 숫자와 모드, 두 개의 필드를 가지고 있다.

addressMode와 instruction 섹션은 타깃 ISA의 머신 명령어와 어드레싱 모드를 기술한다. 명령어와 어드레싱 모드 둘 다 ISADesc에서 operand type, action 그리고 syntax의 세가지 필드를 가진다. 이것들은 명령어의 의미를 외부적으로 명시하고 하드웨어의 세부적인 사항을 숨김으로써 아키텍처의 추상화된 동작을 나타낸다. 예를 들어, (그림 4)는 ADL에 기술된 add-shift-left 명령어를 보여준다.



(그림 5) Visualization of the IR for the kernel code of shortestpath

효과적인 top-down 디자인 방법론을 제공하기 위해서 이 섹션들의 각 서술은 계층적으로 정의되어 있다. 각 서술은 하위 레벨의 서술을 포함할 수 있다. 이 계층구조는 ISA를 더 쉽게 관리할 수 있게 하고, 명령어, 어드레싱 모드, storage섹션들을 독립적으로 기술할 수 있게끔 함으로써 5.3절에서 보게 될 MD 재사용의 극대화를 가능케 한다. primitive, storage 섹션에 정의된 구성요소들은 계층구조의 최하위에 위치하면서, 어드레싱 모드와 명령어 서술의 action 필드를 위한 기본 도구로서 역할을 담당한다. 다음은 레지스터 storage 타입과 primitive ss_plus로 정의된 변위(displacement) 어드레싱 모드의 서술 예이다.

action { efa = ss_plus(Rd, imm5); }

addressMode 섹션은 메모리접근 방식에 대해 정의한다.

load/store에 알맞은 메모리 참조를 위해서는 효과적으로 메모리에 접근할 수 있는 어드레싱 모드가 필요하다. 명령어도 동일한 방법으로 기술된다. 다음은 multiply and accumulate 명령어를 나타낸다.

action { rd = ss_plus(mult(rs, rm)); }

명령어가 기술될 때 그것의 오퍼랜드는 보통 storage 섹션에서 정한 레지스터나 메모리 타입이다. 그러나 사용자는 더 복잡한 어드레싱 모드를 정의할 수 있고, 그 정의된 어드레싱 모드를 명령어 서술에 사용할 수 있다. 예를 들어, (그림 4)의 어드레싱 모드 **dataAddrMode1**은 addressMode 섹션에 정의되어 있고, 명령어 **addsl1**의 서술에서 오퍼랜드로 사용된다.

3. 입력 어플리케이션: Fixed-point AC-3 코드

디지털 오디오와 HDTV 같은 제품에서는 전세계적으로 Dolby-developed AC 3나 Dolby Digital이 표준이다[22]. 이 기술은 사람의 귀에 들리지 않는 신호를 제거함으로써 음질의 훼손 없이 오디오 데이터 크기를 최소화한다. 또한 다양한 오디오 채널 포맷을 단일 low rate bit stream으로 변환 가능하다[21]. 최근에는 8채널 구성을 지원하여 종전의 모노 또는 스테레오에서 6채널 서라운드 포맷까지 가능하다.

3.1 AC-3 알고리즘의 연산 패턴

AC 3 알고리즘은 두 가지 부분으로 나누어진다. 하나는 encoder이고 다른 하나는 decoder이다. 우리는 이 연구에서 decoder 쪽에 더 관심을 두었다. 왜냐하면 encoder는 서버 측에서 실행되는 반면 decoder는 임베디드 프로세서가 있는 클라이언트 쪽에서 실행되기 때문이다. 게다가 encoder는 decoder의 역 과정이다. (그림 6)에서 보는 것과 같이 AC 3 프레임은 AC-3 decoder의 입력으로 들어간다. 각 프레임은 12개의 필드를 가지고 있는데 이것은 [2]에 명기되어 있다.

SYNC	CRC	SI	BSI	AB0	AB1	AB2	AB3	AB4	AB5	AUX	CRC

decoder는 encoded stream을 동기화해야 하고, 입력 프레임의 에러를 체크해야 하며, encoded spectral envelope와 quantized mantissa[4]와 같은 다양한 타입의 포맷을 식별해야 한다. 지수(exponent) 부분은 spectral envelope로부터 생성되고 가수(mantissa) 부분은 비트 할당에 의해 생기는 값으로부터 생성된다. 이 두 부분은 시간 도메인에서 PCM sample을 출력하기 위해 결합된다.

AC-3 알고리즘은 12단계의 기능 모듈들로 구성되어있으며 bit allocation, mantissa unpacking, inverse transformation[22], 이 세가지 주요 모듈이 실행 속도에 영향을 미친다. 이 모듈들은 (그림 6)의 굵은 선 박스 안에 표시되어 있다.

AC-3 bit stream에서 몇 개의 mantissa는 효과적인 데이터 전송을 위해 묶여진다. 데이터를 더 조밀하게 묶기 위해 몇몇의 mantissa는 하나의 전송 값으로 함께 그룹화 된다. bit allocation 루틴에서 각 그룹에 bit allocation 포인터를 계산하고 이를 이용해 decoder에서 다시 그룹을 나누고 각각의 mantissa 데이터로 복구 할 수 있다. mantissa를 다시 복구하는 과정에서 다음의 연산 패턴이 공통적으로 발견된다.

```
tr_coeff[k] = mant[k] >> exp[k];
tr_coeff[k] = quant_tab[mant_code[k]] >> exp[k];
```

그룹화된 mantissa를 나누는 연산은 다음이 대부분이다.

```
mant_code[a] = truncate (group_code / 25);
mant_code[b] = truncate ((group_code % 25) / 5 );
mant_code[c] = (group_code % 25) % 5;
```

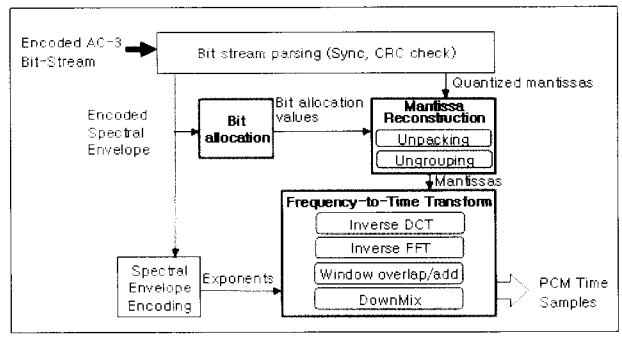
mantissa를 복구하는 과정에서 shift와 modulo 연산이 자주 사용되는 것을 볼 수 있다.

mantissa의 그룹이 나뉘고 복구된 후 각 encoded 채널의 주파수 계수 집합이 생성된다. inverse transform 모듈은 기본적으로 IFFT (Inverse fast fourier transform) 알고리즘을 사용하는데, 이는 계수를 주파수 도메인에서 시간 도메인으로 변환한다. 이 과정에서 다음 두 가지 중요한 루프 패턴이 있다.

```
for(k=0; k<N/4; k++)
Z[k]=(X[N/2-2*k-1]*xcosl[k]-
[2*k]*xsinl[k])+j*(X[2*k]*xcosl[k]+
X[N/2-2*k-1]*xsinl[k]);
for(n=0; n<N/4; n++) {
z[n] = 0 ;
for(k=0; k<N/4; k++)
z[n] += Z[k]*(cos(8*pi*k*n/N)+j* sin(8*pi*k*n/N));
}
```

3.2 고정 소수점 코드 변환

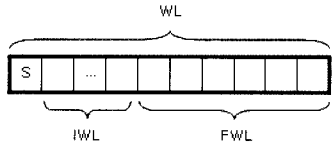
대부분의 임베디드 응용 프로그램과 같이 낮은 비용을 우선으로 하기 위해 AC-3 또한 고정 소수점 프로세서를 타깃으로 하여야 한다. 공교롭게도 Dolby Laboratories INC. 에서 제공하는 AC-3 코드는 부동 소수점 C로 되어있다. 부동 소수점을 고정 소수점 프로세서에서 수행하기 위해서는 소프트웨어 라이브러리를 통해 부동 소수점 프로세서로 에뮬레이션 해주어야 하는데 이는 많은 비용을 야기시킨다. 그



(그림 6) AC-3 디코더의 기능 모듈

래서 우리는 이 연구에서 원본 코드의 부동 소수점 데이터를 고정 소수점으로 변환했다.

비록 부동 소수점과 고정 소수점 코드는 같은 기능을 가지고 있지만 기능을 구현하는 하위 레벨의 연산 패턴은 다소 다르다. 이것을 보이기 위해 본 연구에서 어떻게 변환했는지를 설명하겠다. 다음은 word 길이(WL)를 가진 고정 소수점 포맷을 나타낸다.



고정 소수점 데이터 포맷은 변환을 위해 부호(sign), 정수(integer), 분수 비트(fractional bits)로 구성된다. 정수를 표현하기 위한 비트의 수를 IWL(integer word legth)라 하고, 분수를 표현하기 위한 비트의 수를 FWL(fractional word-length)라 한다. 그래서 WL = IWL + FWL + 1이 성립한다. 예를 들어 IWL이 2인 8bit 값, 01000100 이 있다면, 이진수로는 +10.00100, 십진수로는 2.125로 해석된다. IWL과 FWL에 따라 R(range)과 Q(quantizatio step)가 결정된다. R은 $-2^{IWL} \le R \le 2^{IWL}$, Q는 $Q = 2^{-FWL} = 2^{-(WL-1-IWL)}$ 이다. IWL의 크기 증가는 변수의 overflow를 막을 수 있지만, quantization 노이즈를 증가시킨다. 그래서 최적의 IWL 크기는 그 레인지(range)에 의해 결정되어야 한다. 각 프로그램 변수의 레인지를 계산하기 위해 우리는 시뮬레이션에 기반한 레인지 예측 방법 (simulation-based range estimation method)[13]을 적용하였다.

보통의 ANSI C 정수 곱셈에서 double precision product 결과의 하위 부분만을 저장할 때 고정 소수점 곱셈은 overflow를 막기 위해 그 상위 부분을 유지한다. (그림 7)에서와 같이 product 결과의 상위부분 곱하기와 쉬프트 연산 조합으로 얻을 수 있다. 결과적으로 원래의 AC-3 코드와는 다르게 많은 여분의 쉬프트 연산이 포함된 고정 소수점 버전의 코드가 생성된다.

```
void cifft(void) {
    double br, bi, cr, ci, dr, di, rtemp, itemp, ···;
    ···
    rtemp = br * cr- bi * ci;
    itemp = br * ci + bi * cr;
}
```

(a) a segment of floating-point code

```
void cifft(void) {
    long br, bi, cr, ci, dr, di, rtemp, itemp, ···;
    ···
    rtemp = (br * cr)>>15 - (bi * ci)>>15;
    itemp = (br * ci)>>15 + (bi * cr)>>15;
}
```

(b) translated to fixed-point data formats

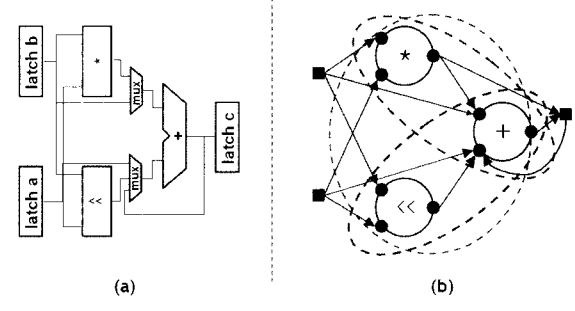
(그림 7) 부동 소수점을 고정 소수점으로 변환하는 예제

4. MOG and OCDFG

일반적으로 컴파일러는 타깃 머신의 ISA는 코드 생성 과정 동안에는 변하지 않는다고 가정한다. 따라서 컴파일러 작업은 정해진 ISA내에서 가장 이상적인 최적의 머신 명령어 시퀀스를 찾도록 제한되어 있으며 각각의 머신 명령어들은 소스 코드 내에서의 개별적인 연산들과 일치된다. 반면에 본 논문에서는 타깃 머신이 고정된 ISA가 아닌 응용 프로그램에 따라서 보다 나은 성능을 제공하는 유동적인 ISA를 갖는 타깃 머신을 가정한다. 주어진 응용 프로그램 코드 내에서 주어진 고정된 ISA에 대해 먼저 가장 최적화된 컴파일된 코드를 찾는다. 이렇게 찾은 코드의 성능이 만족스럽지 못할 경우에는 주어진 응용 프로그램에 대하여 특화된 코드 생성을 할 수 있는 추가적인 타당한 머신 명령어를 찾도록 한다.

타당한 머신 명령어의 정확한 의미는 조금 모호해 보인다. 따라서 본 장에서는 의미를 보다 명확히 하기 위해서 몇 가지의 용어를 더 소개하도록 한다. 먼저 machine operation graph(MOG)라고 불리는 directed graph을 정의한다. MOG는 아키텍처의 데이터 패스로부터 추출된 연산 동작(operational behavior)을 추상화한다. MOG에서의 각각의 노드는 데이터 패스의 연산 단위(operational unit)에 대응되고 두 개의 노드를 연결하는 하나의 에지(u, v)는 2개의 노드 u, v를 연결하는 의미이다. 모든 연산 노드는 각각의 연산 종류로 레이블 되어 있다. (그림 8) (a)는 3개의 연산 단위들이 서로 연결되어 있는 데이터 패스를 표현하고 있고 (b)는 이러한 데이터 패스를 간략하게 표현하고 있다. MOG에서의 각각의 노드는 연산 단위에 해당하는 연산 타입을 상징하고 우리는 그러한 노드를 연산 노드(operation node)라고 부른다.

데이터 패스의 연산 단위와 마찬가지로 각각의 연산 노드들은 inbound, outbound 노드를 통하여 이웃하고 있는 입출력 포트를 가지고 있다. 따라서, o(s, t)로 표현된 연산 노드는 각각의 입력 포트에 연결되어 있는 s 소스 오퍼랜드를 취하여 각각의 출력 포트에 연결되어 있는 t 목적 오퍼랜드로 변환해 주는 역할을 하는 함수를 의미한다. 예를 들어 8(b)에는 +(2,1), << (2, 1) 와 *(2, 1)으로 표현된 3개의 연산 노드가 존재한다. MOG에서 오퍼랜드는 다른 연산 노드이거나 레지스터와 메모리 같은 저장 요소일 수 있다. 8(b)



(그림 8) 아키텍처 예제와 MOG

에서 저장 요소는 검은색 사각형으로 표현되어 있다. MOG의 개념을 사용하여 이제 머신 명령어를 정의한다.

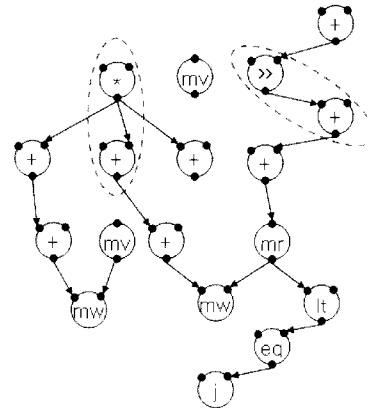
[정의 1] 주어진 machine operation graph M = (Vm, Em)이 있다고 가정할 때 머신 명령어는 다음과 같이 정의된다. I = (Vi, Ei) Vi < Vm 이고 Ei 역시 전체 Em의 부분집합이 되는 즉, 주어진 MOG의 connected sub graph I 가 머신 명령어가 된다. 만약 노드의 개수가 1일 경우에는 I는 단순하다고 하며 2개 이상일 경우에는 복합이라고 한다.

명령어 I가 하나 이상의 노드로 구성된 connected graph 이므로 <v1, v2,… vk>를 연결하는 적어도 하나 이상의 패스가 존재한다. 만약 k값이 simple path의 최대 길이라면 명령어 I를 k-복합 명령어라고 부른다. 단순 명령어는 정의에 따라서 1-복합 명령어이다. (그림 8) (b)에서 세 개의 단순 명령어 +(2, 1), *(2, 1), <<(2, 1) 그리고 점선으로 표시되어 있는 세 개의 2-복합 명령어가 존재한다. MOG는 단순히 연산 노드와 각각의 연산 노드간의 연결성 만을 표현하며 다른 정보들은 숨겨버린다. 예를 들면 8(a)에서의 feedback 데이터 패스 정보는 실제 MOG로 표현되어 있는 8(b) 그래프에서는 표현되지 않았다.

(그림 5)에서 표현되어 있는 응용 프로그램 코드의 IR 폼은 control data flow graph(CDFG) 형태로 표현되어 있다. CDFG의 각각의 기본 블럭들은 DAG (directed acyclic graph) 형태를 갖는 데이터 흐름 그래프의 형태로 주어진다. DAG D = (V_D,E_D) 에서 Ed는 데이터 흐름을 표현하며 노드 집합 Vd는 CDFG 연산을 위한 노드의 집합을 나타내는 Vo와 CDFG 연산에 의해서 생성되거나 사용되는 데이터 값의 합집합으로 구성되어 있다 (V_D = V_o ∪ V_d). 정리하면 타깃 머신의 아키텍처 정보, 즉 타깃 머신의 ISA를 표현하고 있는 것이 MOG이고 주어진 응용 프로그램 코드를 IR 수준으로 표현하였을 때에 나타나는 정보가 CDFG이다.

따라서 주어진 MOG M에 대하여 컴파일러의 코드 생성 과정은 M으로부터 추출한 머신 명령어들의 집합 {I_1,I_2,…,I_n}로 매핑하는 과정이라고 할 수 있다. 컴파일러에서 구현된 매핑 함수는 데이터 값 Vd를 아키텍처의 저장 요소로 그리고 연산 노드 Vo를 머신 명령어 I로 어떻게 매핑할 지에 대한 정보를 제공한다.

앞에서 언급한 MOG, CDFG에 관해서 다시 한번 언급하자면 기존의 컴파일러들은 고정된 MOG에서 결정된 ISA를 가정하였지만, 우리는 원래의 ISA에 추가적인 명령어들을 확장시킬 수 있다. 부가적인 명령어들은 아키텍처를 표현하는 MOG와 응용 프로그램 코드를 나타내는 CDFG 정보를 기반으로 하여 제한된다. 이러한 확장과정을 이용하기 위해서 우리는 기존의 복잡한 CDFG를 단순화 해야 할 필요가 있다. 왜냐하면 기존의 CDFG는 저장 타입과 같은 응용 프로그램 코드와 관련하여 불필요하게 자세한 정보들을 포함하고 있기 때문이다. 따라서 우리가 필요로 하는 연산간의 의존성 정보만을 간단하게 표현하는 operation CDFG(OCDFG)를 정의한다.



(그림 9) (그림 5)의 IR로부터 생성된 OCDFG

[정의 2] 주어진 CDFG D = (VD,ED)에 대해서 operation CDFG Do = (Vo,Eo)라고 하자. 이때 Vo, Eo 와 VD, ED 의 관계를 살펴보면 Vo 는 VD 부분집합이고 Eo 는 Vo 를 연결하는 에지를 의미한다.

(그림 9)는 (그림 5)에서 표현한 IR 폼에 대한 OCDFG를 나타낸다. 기존의 CDFG에 비해서 코드 내에서 연산간의 의존성 관계를 보다 명확하게 표현해준다. 기존의 CDFG에서 메모리와 레지스터 같은 저장 정보를 과감하게 생략하고 연산간의 연결성만을 부각시킨다.

OCDFG에서의 각각의 노드는 기본적으로 MOG에서의 연산 노드와 동일하다. 따라서 두 개의 그래프에서 단일 노드 일치(single node matching)을 찾기는 간단하다. 단순하게 레이블과 I/O 포트의 개수만을 비교함으로써 일치 여부를 찾을 수 있다. 유사하게 OCDFG에서의 에지는 응용 프로그램 코드에서의 연산간의 의존성을 나타내며 MOG에서의 에지는 연산 단위간의 데이터 패스를 나타낸다. 따라서 OCDFG에서의 다중 노드 (multiple node)와 MOG에서의 k-복합 명령어

1. Build an MOG *M* from the target architecture.
2. Build an OCDFG *D* from the CDFG for the application.
3. Find all simple instructions from *M* and insert them to the base instruction set *S*.
4. Describe *S* in our ADL to target the compiler to *S*.
5. Compile the AC-3 code and measure the performance on the target processor with S.
6. Profile the performance results to make a set *O* of common operation patterns extracted from *D*.
7. Sort all operation patterns in O according to their importance at run time.
8. For each operation pattern *o* of *O* in the sorted order, do
   1) Check if there exists a composite instruction *i* in *M* which *o* matches according to Definition 3.
   2) If *i* is found, test if *i* is valid.
   3) If *i* is valid, insert *i* into *S* by describing *i* in the ADL.
9. Describe the augmented set *S* in the ADL to recompile the AC-3 code and measure the performance of the compiled code

(그림 10) 응용 프로그램에 특화된 명령어를 찾기 위한 절차

일치 문제는 물론 단일 노드 일치에 비해서는 다소 까다롭지만 그래프 패턴 일치 문제로 변환 가능하다. 이 변환 과정에서 연산 패턴이라는 용어를 사용하게 된다.

[정의 3] 주어진 OCDFG에서 연산 패턴의 정의는 OCDFG의 connected subgraph를 의미한다. 만약에 패턴 P 가 k-복합 명령어와 일치된다면 우리는 패턴 P를 k-matchable 패턴이라고 부른다.

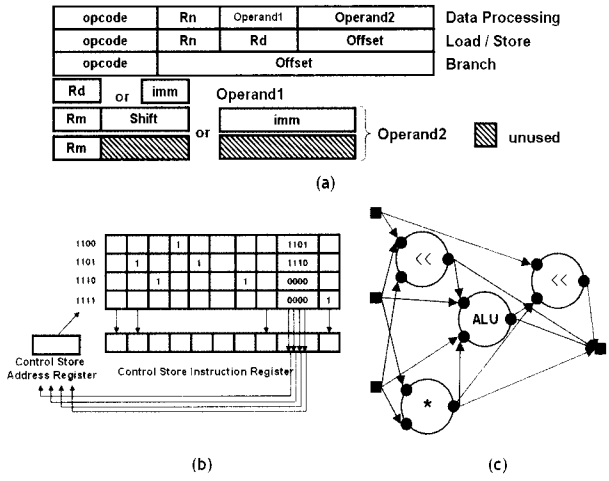
위의 정의에 의해서 (그림 9)에서 점선으로 표시된 2개의 패턴은 MOG에 있는 {*(2,1),+(2,1)}와 {>>(2,1),+(2,1)}의 2-복합 명령어와 각각 일치 되므로 (그림 8)에 있는 아키텍처에 대하여 2-matchable 패턴이다. 5장에서 우리는 MOG와 OCDFG를 가지고 타당한 머신 명령어를 정의하고 기본 ISA에 추가된 몇 개의 타당한 명령어를 통해 어떻게 컴파일 된 코드가 향상되는지를 보인다.

5. AC-3 특화 명령어 찾기

본 장에서는 AC-3 응용 프로그램 코드에 대해 주어진 타깃 아키텍처에 대해 가장 이상적인 명령어 집합을 찾기 위해서 사용된 ADL에 기반한 컴파일러에 대한 실험적인 연구 결과를 알리고 있다. 우리의 연구를 통해서 ADL을 기반으로 한 재겨냥성이 필수 불가결한 것임이 입증되었다. 왜냐하면 ADL은 기본 ISA에 새로운 명령어를 추가하고자 할 때에 매우 직관적이고 빠르게 새로운 명령어 집합에 대해서 재겨냥이 가능하도록 하며 또한 ADL에 의해 새롭게 retarget된 코드 생성기는 컴파일 된 코드에 대한 추가 명령어의 효과에 대한 빠르고 정확한 평가를 가능하게 해준다. (그림 10)의 과정은 우리의 컴파일러를 이용하여 주어진 타깃 아키텍처에 대해서 최적화된 ISA를 찾는 과정을 보여주고 있다. 단계 1, 2는 MOG와 OCFDG를 이용하는 과정으로 이미 4장에서 설명했고 나머지 과정에 대해서는 이어지는 절에서 설명하겠다.

5.1 타당한 명령어를 위한 물리적 제한

새로운 명령어를 앞장에서 언급한 MOG와 OCDFG를 이용하여 찾았을 경우 새로운 명령어가 주어진 타깃 아키텍처의 물리적 제한을 만족하는지를 증명함으로써 타당함을 테스트해야 한다. 이러한 물리적 제한은 데이터 패스 구성, 인코딩 계획, 제어 로직과 같은 타깃 아키텍처 디자인에 따라서 다양하게 변한다. 따라서 본 논문에서는 이러한 제약조건들을 일반화하지 않고 (그림 11)에서 보이는 AC-3 응용 프로그램에 대한 고정 소수점 프로세서에 초점을 맞추어 다루도록 하겠다. 먼저 기본 ISA는 앞에서 언급되었던 단순 명령어 집합 (정의 1 참조)만을 포함하도록 한다. 앞에서 언급되었던 단순 명령어는 타깃 머신의 데이터 패스에서 하나의 연산 단위에 대응되는 노드이므로 만약 단순 명령어를 포함하지 않을 경우에는 연산 단위가 없는 상황이기 때문에



(그림 11) 명령어 포맷, 타깃 프로세서의 control memory/store, MOG

반드시 단순 명령어는 포함해야만 한다.

타깃 프로세서는 크게 데이터 처리, 메모리 참조(load/store), 그리고 분기문의 세 종류의명령어 집합으로 구성되어 있다. 이 명령어들에 대한 어드레싱 모드는 register-indirect, displacement, absolute, immediate를 포함한다. 현재의 타깃 아키텍처에서는 모든 단순 데이터 처리 명령어가 수행 가능하고 데이터 패스에서는 몇 가지의 연산 단위가 직렬로 연결되어 있어서 나중에 필요할 경우에는 복합 명령어로 합성이 가능하도록 되어 있다.

명령어 추가를 쉽게 하기 위해 아키텍처에서는 임베디드 시스템에서 사용되는 프로그램 가능한 IP에서 쓰이는 micro-programmed control unit을 사용하고 있다. Control unit은 실제 머신 명령어를 수행하는 동안 데이터 패스에서 수행되는 실행 단계를 제어한다. 전체 명령어는 micro operation이라고 불리는 control signal들의 timed sequence에 의해서 수행되고 이러한 mice operation들은 펌웨어 혹은 시스템 메모리에 저장된다. 이러한 micro operation들을 저장하는 storage를 control store 또는 control memory라고 부른다. 명령어의 opcode는 이러한 control memory의 주소로서 사용되어 명령어에 대한 micro operation sequence가 control memory의 특정 위치에 저장된다. Control memory의 사용되지 않은 주소에 새로운 micro operation sequence를 기록하고 control memory address를 opcode로 정의한다.

4장에서 언급했듯이 복합 명령어는 MOG의 connected subgraph에서 얻어진다. 예를 들어 (그림 11) (b)를 살펴보면 *(2, 1), ALU(2, 1)의 2개의 노드를 갖는 subgraph는 하나의 2-복합 명령어를 이룬다. 이러한 복합 명령어에 대한 타당성은 명령어에 대한 micro operation sequence를 생성함으로써 확인 가능하다. 만약 타당한 micro operation sequence를 찾으면 복합 명령어는 타당하다고 판단하여 기본 ISA에 추가된다. 데이터 패스와 제어 로직에 대한 제한은 일치하는 패턴을 갖는 명령어를 찾는 동안 체크할 수 있다.

다른 제한으로 고려할 만한 것이 명령어 워드 길이이다. 우리의 타깃 머신의 경우에는 고정 워드 길이 명령어를 사

용하므로 MOG에서 얻어진 모든 복합 명령어가 타당한 것은 아니다. 따라서 명령어 I가 encoded되었다면 먼저 명령어 워드 길이 제한을 체크하기 위해서 명령어 오퍼랜드의 개수를 계산해야만 한다. Instruction word length = opcode + operand(operand 개수*operand bit 수)로 구성되므로 2가지 오퍼랜드의 요소를 살펴보도록 한다. 먼저 명령어에 대한 total operand bit은 MOG에서 구할 수 있다. MOG를 살펴보면 연산 노드에 대한 I/O 포트 정보를 포함하고 있다. (오퍼랜드 개수 정보는 포함하고 있다.) 포트는 크게 복합 명령어 내부에서만 사용되는 내부 포트와 외부의 다른 연산과 연결되어 있는 외부 포트로 구성되어 있다. 오퍼랜드의 개수는 외부 포트의 개수에 비례한다. 각각의 오퍼랜드에 대한 bit수와 오퍼랜드 개수는 encoding 제한을 결정짓는 중요한 요소이다 [15]. 만약 명령어의 오퍼랜드 개수가 너무 많아서 명령어 워드 길이 제한을 위반할 경우에는 각각의 오퍼랜드에 할당되는 bit수를 줄여서 전체 명령어 워드 길이를 맞추어야 한다. 이러한 경우는 long, complex 명령어를 사용하는 CISC구조에서 빈번하게 발생한다. 하지만, 우리의 논문에서는 오퍼랜드의 개수 증가로 인한 오퍼랜드 bit 감소는 고려하지 않는다. 왜냐하면 레지스터 오퍼랜드의 bit수를 줄임으로써 응용 프로그램 코드 내에서 더 많은 레지스터 스필 (register spill)이 발생하여 심각한 성능 저하가 초래되기 때문이다.

타당성 테스트 과정을 (그림 11)의 예를 통해 설명한다. ISA에는 같은 연산 타입에 여러 어드레싱 모드를 갖는 명령어가 있다. +(2, 1)와 같은 단순 명령어를 생각해 보면 이 명령어는 다음과 같은 다양한 어드레싱 모드를 가질 수 있다.

```
add r1,r2       // r1 = r1 + r2    : two address format
add r1,r2,r3    // r1 = r2 + r3
add r1,r2,imm   // r1 = r2 + imm
add r,imm1,imm2       // r = imm1 + imm2
```

만약, MOG내에서 하나의 복합 명령어가 발견 되었다면, MOG에서는 storage type에 대한 정보가 없기 때문에 어드레싱 모드를 처음에는 알 수 없다. 따라서 타당성 테스트를 하는 과정에서 명령어가 이용할 수 있는 모든 가능한 어드레싱 모드를 찾아야만 한다. 임의의 어드레싱 모드가 복합 명령어에 대해서 배정되었을 때에 아키텍처적 한계로 제한될 수 있다. 예를 들어서 (그림 11) (c)에 있는 두 개의 노드 {+(2,1),≪(2,1)}[2]로 구성되어 있는 add-shift left 복합 명령어 (2-복합)의 경우에는 다음의 여섯 가지의 모드가 배정 가능하다.

```
addsl r1,r2,r3  // r1    (r1+r2) << r3
addsl r1,r2,imm       // r1 = (r1+r2) << imm
addsl r1,imm,r2       // r1 = (r1+imm) << r2
addsl r,imm1,imm2     // r = (r+imm1) << imm2
addsl r1,r2,r3,n      // r1 = (r2+r3) << n
addsl r1,imm,r2,n     // r1 = (imm+r2) << n
```

이 경우에 제한은 오퍼랜드의 개수이다. 마지막 두 개의 경우에 대해서 4개의 외부 오퍼랜드를 갖기 때문에 하나의 source, destination이 공유된다고 해도 기껏해야 3개의 오퍼랜드를 지원할 수 있으므로 타당하지 않게 된다.

다른 제한으로는 세 개의 노드 {≪(2,1),*(2,1),+(2,1)}[3]를 사용하는 *shift-left-mult-add* 명령어의 경우와 같은 오퍼랜드의 usage이다.

```
slmadd r1,r2,r3       // r1 = (r1*r2) + (r2<<r3)
slmadd r1,r2,imm      // r1 = (r1*r2) + (r2<<imm)
slmadd r1,imm,r2      // r1 = (r1*imm) + (imm<<r2)
slmadd r1,r2,r3,n     // r1 = (r2*r3) + (r3<<n)
slmadd r1,imm,r2,n    // r1 = (imm*r2) + (r2<<n)
```

* 연산의 두 번째 오퍼랜드와 << 연산의 첫 번째 오퍼랜드가 동일함을 주목하자. 이러한 제한은 데이터 처리 연산에 연결된 storage 요소의 제한된 개수 때문에 발생한다. 이 예제에서는 r1 = (r1 * r2) + (r3 << n) 연산을 수행할 수 있는 어드레싱 모드를 갖는 타당한 복합 명령어가 없다. 이는 위의 ***addsl*** 명령어에서와 같이 storage 요소의 개수에 대한 물리적 제한 때문이다.

5.2 AC-3를 위한 복합 명령어 선택

5.1절에서 살펴본 바와 같이 타당한 복합 명령어를 합성했을 경우에 실제 control memory에 저장되는 명령어의 개수는 실제 복합 명령어에 대한 다양한 어드레싱 모드 때문에 상당히 커지게 된다. 일반적으로 n개의 머신 명령어를 저장하기 위해서 필요한 메모리 공간은 대략 n * a * m이 된다. 여기서 a는 연산 타입에 대한 평균적인 어드레싱 모드의 개수고 m은 명령어 마다 micro operation의 개수다. 우리의 타깃 프로세서에서는 m * a값이 대략 20에서 30정도가 되므로 약간만 n값이 커지게 될 경우에도 control memory 공간이 상당히 크게 증가하게 된다. 이것은 가능한 한 적은 메모리 공간을 필요로 하는 임베디드 시스템에서는 상당히 치명적인 단점으로 지적된다. 게다가 8개의 data processing, load/store, branch만을 갖는 단순한 아키텍처에서도 상당수의 조합은 실제로는 불가능하지만 이론적으로 가능한 복합 명령어의 개수가 2^8만큼이 된다. 결론적으로 가능한 모든 조합을 찾기보다는 가능한 적은 명령어를 추가하면서 큰 성능의 향상을 가져올 수 있는 가장 최적의 솔루션을 찾아 제어 로직에 대한 하드웨어 비용이 급격하게 증가하는 것을 방지하는 것이 필요하다. 게다가 명령어 집합이 커지게 될 경우에는 반대로 작용할 수도 있다. 일반적으로 컴파일 시간은 전체 명령어 집합 크기에 비례한다. 왜냐하면 매우 큰 명령어 집합 후보 중에서 가장 최적의 명령어 시퀀스를 찾기 위해서는 보다 많은 시간이 소요되기 때문이다.

GPP(General purpose processor)와는 다르게 ASIP(application specific instruction set processor)은 반복적이고 불필요한 명령어 집합을 제거하여 명령어 집합의 크기를 줄임으로써

2) '≪'는 (그림 11) (c)의 두 개의 shift 연산 중에서 우측을 의미한다.

3) '≪'는 (그림 11) (c)의 두 개의 shift 연산 중에서 좌측을 의미한다.

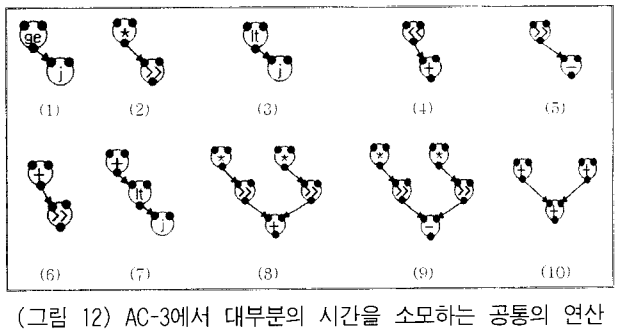
보다 나은 성능을 얻을 수 있다. 이러한 과정을 수행하기 위해서는 과연 주어진 응용 프로그램에 대해서 어떠한 명령어가 꼭 필요하고 어떠한 명령어가 불필요 한지를 판별할 수 있어야 한다. 따라서 목표로 하는 응용 프로그램에 분석과 프로파일링이 필요하다. 이렇게 해서 수집한 응용 프로그램에 특징적인 정보를 이용하여 불필요한 명령어 집합을 걸러낸다. 불필요한 명령어란 MOG의 명령어 집합 중에서 응용 프로그램 코드에 대한 OCDFG에서 나타나지 않는 non-matchable 패턴 명령어 혹은 비실현적인 제약조건을 갖는 명령어를 의미한다. 아래의 결과는 고정 소수점 AC-3에 대한 프로파일링 결과로 이러한 정보들이 AC-3 코드에 필요한 명령어를 찾기 위해서 어떻게 사용 되었는지 보여준다. 쓸모 있고 필요한 명령어만이 ISA에 추가된다.

다음의 표는 AC-3 디코더의 가장 중요한 5개의 루틴을 표시 하고 있다. 표의 숫자는 오디오 데이터의 한 프레임에 대한 디코더의 계산 시간에 대해서 각각의 루틴이 차지하고 있는 비중을 나타낸다.

function	unpmants	cifft	window_d	idctsc	Idctsc2	etc
time(%)	32.85	20.86	9.77	4.44	3.76	28.3

표를 살펴보면 5개의 중요한 루틴이 전체 수행시간의 3/4를 차지하였다. 복합 명령어를 통해서 수행시간을 줄이기 위해서, 앞에서 우리가 언급한 과정을 순차적으로 밟아 나갔다. 먼저 전체 소스 응용 프로그램 코드를 IR 폼으로 CDFG 포맷으로 변환한 뒤 CDFG를 OCDFG로 변환하여 (그림 12)와 같이 실행 시간에 가장 빈번하게 발생한 연산 패턴 10개를 찾았다. 10개의 패턴 중에서 2,4,5,6,8,9 의 6개의 패턴에서 포함된 shift 연산이 가장 빈번하게 발생하였다. 이는 부동 소수점을 고정 소수점으로 변환하기 때문이다. 예를 들어 연산 패턴 4는 부동 소수점의 fractional part에 대한 scaling을 하고 다른 수에 더하기를 하는 형태이다. 반면에 1,3,7 패턴은 전형적인 조건 분기 연산으로 조건 테스트와 점프 연산이 복합되어 있다.

이렇게 코드에 대한 연산 패턴을 찾은 다음에 어떤 패턴이 우리의 타깃 머신에서 matchable한지를 식별해야 한다. 이를 위해 먼저 (그림 11)에서 그려진 타깃 머신을 위한 MOG를 생성하고 이들 패턴과 일치 되는 MOG의 sub-graph를 찾는다. 만약 패턴이 일치하면 일치된 패턴이 5.1장에서 언급한 실제 타깃 아키텍처의 제한 조건을 만족하는지 타당성 조사를 한다. 타당성 조사를 통과한 명령어는 ISA에 새롭게 추가된다.



(그림 12) AC-3에서 대부분의 시간을 소모하는 공통의 연산 패턴

패턴 9는 패턴 2, 5를 포함하는 것에 주목하자. 만약에 패턴 2, 5가 항상 패턴 9내부에서만 일어난다면 패턴 2, 5에 대한 더 작은 명령어를 추가하는 대신에 하나의 복합 명령어 패턴 9를 추가하는 것이 더 낫다. 이와 같이 matchable한 패턴에 대해서도 패턴 높이에 따라서 우선순위를 둔다. 즉, 패턴 9의 높이는 3이고 패턴 2, 5의 높이는 2이므로 만약 패턴 9의 타당성이 입증된다면 우선적으로 ISA에 추가된다.

결과를 살펴보면 불행하게도 우리의 실험에서는 높이가 가장 큰 3개의 패턴이 모두 타깃 머신의 MOG와 일치되지 않았다. 패턴 10 역시 제외되었다. 따라서 10개의 가능한 패턴 중에서 타깃 머신의 MOG와 matchable한 6개의 패턴 만이 남게 되었다. 패턴 1, 2, 3, 4, 5, 6은 모두 높이도 2로 동일하므로 패턴의 빈도수에 따라서 우선순위를 설정하였다. 또한 이 여섯 개의 패턴은 모두 타깃 머신의 타당성 체크에서도 모두 타당한 패턴으로 나타났다.

5.3 실험 결과

실험에서 우리는 위에서 언급한 6개의 명령어를 하나씩 추가하며 컴파일된 코드에서의 영향을 코드 크기와 실행 속도 측면에서 측정을 하였다. 실험을 통해 ADL에 새로운 명령어를 서술하는 것은 5.1장에서 설명한 바와 같이 매우 직관적이라는 것을 알 수 있었다. 일반적으로 단순 명령어는 (그림 3)의 ADL primitive들로부터 직접적으로 기술할 수 있으며 addsl과 같은 대부분의 복합 명령어는 기존 primitive 혹은 이미 기술한 다른 종류의 복합 명령어에 기초하여 계층적으로 쉽게 합성할 수 있었다. 이런 계층구조는 우리의 ADL에 재사용성과 확장성을 가져다 주었다.

성능은 <표 1>과 <표 3>에서 보는 바와 같이 동적 명령 개수와 바이트로 된 코드 크기의 형태로 시뮬레이터에서 측정된다. 기본 성능은 단순 명령어를 사용하여 기술된 ADL을 통해 생성된 컴파일러를 사용하여 생성된 어셈블리 코드에 대해서 측정하였다. 그 후에 <표 2>에 따르면 패턴 1이 가장 높은 우선 순위를 가지고 있기 때문에 ADL에 복합 명령어로 기술한다. ADL에 추가된 정보를 기반으로 컴파일러는 그 프로세서에 맞게 타게팅하고 다시 컴파일된 코드의 성능을 측정한다. 이러한 실험 과정은 여섯 개의 명령어가 추가될 때까지 반복하며 모든 성능을 측정한다.

<표 1> 6개의 명령어가 차례로 ISA에 추가된 후의 동적 명령어 개수

Instr. / Func.	Base	(1)	(1,2)	(1,2,3)	(1,2,3,4)	(1,2,3,4,5)	(1,2,3,4,5,6)
unpmants	1116672	976320	965883	871519	785213	785213	785213
Cifft	709048	654908	610108	609944	609944	609944	609872
Window_d	332244	267732	249300	249300	249120	249120	249120
Idctsc2	150972	133820	114748	114748	110460	110460	110460
Idctsc	127902	110750	92318	92318	92318	92318	92318
Total	2436838	2143530	2032357	1937829	1847055	1847055	1846983

〈표 2〉 그림 12로부터 처음 여섯 개의 연산 패턴의 빈도(사이클 수로 표시)

Instr. / Func.	(1)	(2)	(3)	(4)	(5)	(6)
unpmants	140352	10437	94364	86306	0	0
Cifft	54140	44800	164	0	11200	72
window_d	64512	18432	0	180	4608	0
idctsc2	17152	19072	0	4288	5248	0
Idctsc	17152	18432	0	0	4608	0
Total	293308	111173	94528	90774	25664	72

〈표 3〉 여섯 개의 명령어가 차례로 추가된 후의 코드 크기의 감소 (바이트로 표기)

Instr. / Func.	Base	(1)	(1,2)	(1,2,3)	(1,2,3,4)	(1,2,3,4,5)	(1,2,3,4,5,6)
unpmants	4440	4216	4200	3768	3552	3552	3552
cifft	1700	1620	1588	1572	1572	1572	1568
idctsc	444	416	364	364	364	364	364
idctsc2	524	492	436	436	428	428	428
window_d	752	704	672	672	652	652	652
Total	**7860**	**7448**	**7260**	**6812**	**6568**	**6568**	**6564**

<표 1>에서 5개의 주요한 루틴의 소요시간을 비교해보면 6개의 복합 명령어를 추가하여 동적 명령어 개수는 약 32% 정도 감소하였다. 특히, 홀로 AC-3 디코더의 실행 시간의 1/3 정도를 차지하는 unpmants 루틴에 대해서는 40%정도의 향상이 있었다. 표 3에 따르면 코드 크기는 평균적으로 대략 20%를 약간 상회하도록 줄었다. 동적인 빈도와 코드의 크기가 아닌 실행 시간에 기반해서 AC-3 디코더로부터 다섯 개의 주요한 루틴을 선택했기 때문에 상대적으로 수행시간 측면에서 코드 크기보다 더 큰 효과를 나타내는 것을 알 수 있었다.

몇 가지 흥미로운 점은 표 2에 따르면 패턴 5의 경우 여러 루틴에서 빈번하게 발생함에도 불구하고 명령어를 추가하였을 경우에 표 1에 따르면 코드 크기의 감소는 발생하지 않는다. 이러한 결과는 패턴 2와 5는 모두 패턴 9의 부패턴으로서 2개의 패턴이 항상 같이 발생하기 때문에 패턴 2에 의한 효과가 이미 반영된 상태에서는 패턴 5에 의한 성능 향상은 기대하기 힘들다는데 있다. 따라서, 패턴 2와 5는 공통이므로 패턴 2가 발생한 후에는 패턴 5에 의한 개선효과가 없으므로 패턴 5를 ISA에 추가하지 않아도 된다. 이러한 성능 분석 결과를 토대로 최종 ADL 서술에서 패턴 5를 제거한다. 이러한 분석 결과로부터 단순히 연산의 빈도수만을 가지고 ISA를 디자인 할 경우 최적의 결과를 얻을 수 없으며 컴파일된 코드를 이용하여 실제 성능을 측정 해야만 정확한 정보를 얻을 수 있다. 이렇게 정확한 성능 측정을 위해 계속해서 변경되는 ISA에 맞는 컴파일된 코드를 생성하기 위해서는 ADL에 기반한 재겨냥성 컴파일러의 필요성이 다시 한번 강조된다.

종합하면 전체 AC-3 디코더의 동적 명령어 개수는 3399K에서 2576K로 대략 1/3정도가 줄어들었다. [21]에 따르면 AC-3 디코더는 하나의 오디오 프레임을 처리하기 위해 32msec가 필요하다. CPI (cycle per instruction)가 1이라고 가정하면 기본 ISA에 대한 최소한의 클럭 주파수는 106MHz (3399K/32msec)이다. 실제로는 CPI는 1보다 크므로 106MHz 이상의 클럭 주파수를 가져야 한다. 이러한 조건은 low-end 고정 소수점 프로세서에서는 불가능해 보인다. 하지만 AC-3 코드에 대하여 특화된 다섯 개의 복합 명령어를 추가하면 필요로 하는 주파수는 80MHz (2576K / 32msec)까지 낮출 수 있다.

6. 동기와 관련 연구

본 연구는 오디오 프로세서 디자인을 주로 다루는 업체와의 공동 연구과정에서 시작되었다. 공동 연구를 통해서 우리는 많은 임베디드 시스템 프로세서들이 자원 제약으로 인해서 심하게 제한되어 있음을 알게 되었다. 그러한 제한된 자원을 사용하여 우리가 원하는 성능을 얻기 위해서 이러한 프로세서에서는 실제 동작하는 소프트웨어가 프로세서의 다양한 복합 명령어와 어드레싱 모드들[20]을 효율적으로 사용하고 있다는 가정하에 디자인되어 있다. 수많은 프로세서 설계자들[5]은 많은 가능한 명령어들이 프로세서에서 수행되고 있는 소프트웨어 커널을 최적화 하는데 유용하다는 것을 주목하였다. 이러한 소프트웨어 커널은 *, +, 그리고 <<와 같은 단순 연산으로 구성된 복합 명령어들을 활용하여 수식들과 관련된 다양한 계산식에서 보다 효율적으로 처리 가능하다. 이러한 관찰 결과는 복합 명령어들이 보다 나은 코드 생성을 하는데 있어서 어느 정도 범위 내에서는 컴파일러에게 혜택을 준다는 사실을 확신시켜주었다. 결론적으로 보다 나은 코드 생성에 도움이 되는 복합 명령어를 찾기 위한 컴파일러 기술을 활용하기 위한 연구를 촉진시켜주었다. 이 논문에서 입증하였듯이 ADL에 기반한 컴파일러는 보다 빠른 컴파일러 재겨냥을 통해서 각각의 ISA 구성의 성능을 빠르게 평가함으로써 ISA 디자인 공간 탐색을 가능하게 하였다.

전통적인 컴파일러 타게팅 방법 [3][4][6][19]은 사용자에 의해서 일일이 손으로 머신에 특화된 매개변수들을 컴파일러에 입력해주고 코드 생성을 위해서 컴파일러 모듈을 조금 수정하는 방법이었다. 따라서 사용자가 컴파일러에 대한 많은 지식과 많은 시간을 소모하는 노력이 필요하였다. 최근에 들어서 프로세서 디자인에서 ASIP의 중요성이 커지면서 많은 연구자들이 주어진 아키텍처에 대해서 컴파일러의 상당 부분이 자동으로 구성되도록 컴파일러의 재겨냥성을 향상시키기 위한 컴파일러 프로젝트 연구를 시작하게 되었다. 이러한 모든 컴파일러들은 모두 하드웨어 ADL 바탕 위에서 제작되었다. 이러한 하드웨어 ADL은 프로세서 ISA의 동작 특징과 하드웨어 컴포넌트의 구조적 특징을 보다 효율적으로 이끌어 낸다[16].

주목할만한 하드웨어 ADL의 대표적인 예로 Chess/Check로 불리는 재겨냥성 EDA 툴 체인에서 사용되는 nML[17]을 들 수 있다. 또 다른 중요한 ADL로 LISA[23]가 있다. LISA

는 컴파일러를 위해서 디자인된 도구가 아니라 시뮬레이터 혹은 다른 관련 도구들을 위해서 디자인 되었었다. 이러한 한계를 극복하기 위해서 Cosy 컴파일러[1]가 LISATek 툴 체인으로 통합되었다. Cosy는 CGD라고 불리는 프로세서 모델링을 사용하여 제작된다. LISA와 CGD사이에는 의미론상으로 직접적인 관련성이 없기 때문에 새로운 프로세서에 컴파일러 재겨냥을 하기 위해서는 LISA 서술을 동일한 의미를 갖는 CGD 서술로 변환해야 한다. 또 다른 EDA(electronic design automation) 툴 체인으로 PEAS-III[11]이 있다. 물론 이 툴 체인 역시 Cosy를 컴파일러로 사용한다. Chess와 Cosy가 강력한 ADL을 기반으로 한 재겨냥성 컴파일러이지만 현재는 상품화 되어있기 때문에 공동의 연구를 위해서 자유롭게 사용할 수 없다.

하드웨어 ADL을 기반으로 한 연구 목적으로 사용 가능한 컴파일러들이 존재한다. 대표적인 예로 UCIrvine에서 개발된 EXPRESSION이라고 불리는 ADL위에 제작된 EXPRESS [16]를 들 수 있다. EXPRESS는 ILP(instruction level parallelism)을 이용하여 C 프로그램에서 최적화된 타깃 코드를 생성해 낸다. EXPRESS는 EXPRESSION을 사용하는 시뮬레이터 SIMPRESS와 함께 개발되어 ASIP 디자인을 위한 연구목적의 재겨냥성 툴 체인을 형성하고 있다. AVIV[8]는 MIT에서 개발된 ISDL이라고 불리는 ADL을 기반으로 한 연구목적의 재겨냥성 컴파일러로서 코드 생성기는 명령어 선택, 레지스터 할당, 그리고 스케줄링을 고려하여 코드의 질을 향상시키고 있다. 다른 ADL로 [18]이 있다.

우리가 새롭게 개발한 재겨냥성 컴파일러는 다른 재겨냥성 컴파일러와 마찬가지로 application의 성능을 높이기 위해서 고안된 새로운 특화 명령어를 추가하거나 기존의 명령어를 쉽게 제거할 수 있는 기능을 가지고 있다. 또한 타겟 머신의 레지스터 수와 종류에 대한 기술을 달리 함으로써 레지스터 할당기를 조절하여 필요한 성능을 얻을 수 있는 최적의 파라미터를 결정할 수 있도록 되어있다. 이러한 디자인 공간 탐색을 빠르고 쉽게 반복하기 위해서 우리는 컴파일러에 IR단계에서 프로그램의 특징을 시각화하고 성능을 미리 프로파일할 수 있는 tool을 포함시켰다. IR단계에서 시각화를 하면 성능 개선을 위한 명령어를 고안할 때 컴파일러가 이를 자동으로 만들 수 있는 컴파일러 friendly한 명령어들을 고안할 수 있다. 이를 이용하면 프로그램의 특징을 알아내고 성능에 있어서 병목지점을 쉽게 찾아 이를 개선할 수 있는 새로운 명령어들을 디자인하는 과정을 쉽고 빠르게 반복할 수 있다.

7. 결 론

6장에서 언급되었던 모든 ADL에 기반한 컴파일러들은 모두 서로 다른 아키텍처모델을 가정하고 있으며 서로 다른 개발 플랫폼을 가지고 있다. 특정 응용 프로그램에 대한 ASIP 디자인에 관한 연구들이 있지만 그들은 모두 다양한 제약 조건과 요구되는 성능을 가지는 응용 프로그램뿐만 아니라 다른 아키텍처 플랫폼과 소프트웨어 환경에서 이루어졌다.

이번 논문의 목적은 기본 ISA에 포함시킬 응용 프로그램에 특화된 명령어를 연구하고 코드생성에서 그것을 활용함으로써 ASIP의 성능을 향상시키기 위해 우리의 재겨냥성 컴파일러를 사용하고자 하였다. AC-3를 타깃 응용 프로그램으로 선택한 것은 DSP에서 가장 중요한 응용 프로그램중의 하나이기 때문이다.

우리의 논문에서 언급된 2개의 그래프 구조(MOG, OCDFG)는 타깃 아키텍처와 응용 프로그램의 operational behavior를 추상화하여 타당한 복합 명령어를 보다 쉽게 찾게 하기 위해서 소개되었다. 우리의 실험을 근거로 할 때 재겨냥된 코드에서 얻어진 성능 결과는 복합 머신 명령어들이 AC-3 코덱 응용 프로그램에 매우 적합함을 명확하게 보여주었다. 우리는 AC-3 특화 명령어들을 사용함으로써 코드 크기와 속도 측면에서 실질적인 성능 향상을 확인할 수 있었다.

서론에서 언급한 것처럼 현재 대부분의 재겨냥성 컴파일러는 이미 상용화되어 그 원본을 구해서 연구 목적으로 사용하기 어렵다. 우리는 새롭게 재겨냥성 컴파일러를 만듦으로써 앞으로 이를 이용한 하드웨어 소프트웨어 동시 디자인에 대한 연구에 발판을 마련했다고 생각한다.

참 고 문 헌

[1] Associated Compiler Experts, Inc. http://www/ace/nl.

[2] ATSC Standard: Digital Audio Compression (AC-3), Revision A, In *Advanced Television Systems Committee*, 20 August 2001.

[3] A. Appel, J. Davidson and N. Ramsey, The Zephyr Compiler Infrastructure. *Technical Report at the University of Virginia*, 1998.

[4] G. Araujo and S. Malik. Code generation for Fixed-point DSPs, In *ACM Transactions on Design Automation of Electronic Systems*, Vol.3, No.2, pp.136-161, April, 1998.

[5] S. Bilavarn, E. Debes, P. Vandergheynst and J. Diguet. Processor Enhancements for Media Streaming Applications , In *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, Vol.41, No.2, 2005.

[6] C. W. Fraser, et. al. BURG: fast optimal instruction selection and tree parsing, In *ACM SIGPLAN Notices*, pp.68-76, 1992.

[7] G. Hadjiyiannis, S. Hanano, and S. Devada. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the 34th Design Automation Conference*, pp. 299-302, 1997

[8] S. Hanono, S. Devadas, Instruction Selection, Resource Allocation, and Scheduling in the AVIV retargetable code generator, 35th Design Automation Conference(DAC), 1998.

[9] J. Hennessy and D. Patterson. Computer Architecture: A

Quantitative Approach, Morgan Kaufman Publishers, 2003.

[10] M. Hohenauer. A Methodology and Tool Suite for C compiler generation from ADL Processor Models, In *Design Automation and Test in Europe Conference & Exhibition*, Vol.2, 2004.

[11] M. Itoh, et. al. PEAS-III: An ASIP Design Environment. In *Proceedings of the Int. Conf. on Computer Design*, 2000.

[12] S. Jung and Y. Paek. The Very Portable Optimizer for Digital Signal Processors, In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp.84-92, 2001.

[13] K. Kim, J. Kang and W. Sung. AUTOSCALER For C: An Optimizing Floating-Point to Integer C Program Converter For Fixed-Point Digital Signal Processors, *IEEE Transactions on Circuits Systems*, Vol.47, No.9, Sep., 2000.

[14] P. Lapsely, et al. DSP Processor Fundamentals, Architectures and Features, In *IEEE Press*, 1997.

[15] J. Lee, K. Choi and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs, In *Proceedings of the IEEE/ACM international conference on Computer-aided design*, San Jose, 2002.

[16] P. Mishra and N. Dutt. Architecture Description Languages for Programmable Embedded Systems, In *IEE Proceedings on Computers and Digital Techniques*, 2005.

[17] J. Praet, D. Lanneer, W. Geurts and G. Goossens. Processor modeling and code selection for retargetable compilation, ACM Transactions on Design Automation of Electronic Systems, Vol.6, No.3, pp.277-307, July, 2001.

[18] H. Scharwaechter, et. al. ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study, In *ACM Transac. on Embedded Computing Systems*, Vol.2, No.3, 2001.

[19] R. Stallman, Using and Porting GNU CC. Free Software Foundations, Feb. 1998.

[20] J. Staunstrup and W. Wolf. Hardware/Software Co-Design: Principles and Practice, Kluwer Academic Publishers, 1997.

[21] S. Vernon. Design and implementation of AC-3 coders, In *IEEE Transactions on Consumer Electronics*, 1995.

[22] Steve Vernon. Dolby Digital: Audio Coding for Digital Television and Storage Applications, In *AES 17th International conference on High Quality Audio Coding*, September 1999, Florence, Italy.

[23] Coware, Inc. http://www.coware.com

부록 A : ADL의 BNF 표현

```
ISADesc   :          primitiveDesc
|          ISATopDesc ISAStorageDesc
|          ISAAddressModeDescListInstructionDescList
ISAAddressModeDescListInstructionDescList :
|          ISAAddressModeDescListInstructionDescList ISAAddressModeDescListInstructionListElement
|          ISAAddressModeDescListInstructionListElement
ISAAddressModeDescListInstructionListElement: ISAAddressModeSetDesc
| ISAInstructionSetDesc
| ISAAddressModeDesc
| ISAInstructionDesc
ISAAddressModeSetDesc : addressModeSetDecl
addressModeSetDecl : ADDRESSMODESET ID LBRACE addressModeSetList RBRACE
| ADDRESSMODESET ID COLON ID LBRACE addressModeSetList RBRACE
addressModeSetList  :
addressModeSetList addressModeSetElement
| addressModeSetElement
addressModeSetElement: ID SEMICOLON
ISAAddressModeDesc: ADDRESSMODE ID COLON ID LBRACE addressModeBodyDesc RBRACE
ISAInstructionSetDesc : instructionSetDecl
instructionSetDecl:
INSTRUCTIONSET ID LBRACE instructionSetList RBRACE
| INSTRUCTIONSET ID COLON ID LBRACE instructionSetList RBRACE
| INSTRUCTIONSET ID LBRACE RBRACE
| INSTRUCTIONSET ID COLON ID LBRACE RBRACE
instructionSetList: instructionSetList instructionSubSet
| instructionSubSet
instructionSubSet: ID SEMICOLON
ISAInstructionDesc: INSTRUCTION ID COLON ID LBRACE instructionBodyDesc RBRACE
```

안 민 욱

e-mail : mwahn@compiler.snu.ac.kr
2003년 서울대학교 전기공학부(학사)
2003년~현재 서울대학교 전기컴퓨터 공학부 박사과정
관심분야 : 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, 컴퓨터 시스템 설계

백 윤 흥

e-mail : ypaek@snu.ac.kr
1988년 서울대학교 컴퓨터공학과(학사)
1990년 서울대학교 컴퓨터공학과(석사)
1997년 UIUC 전산과학(박사)
1997년~1999년 NJIT 조교수
1999년~2003년 KAIST 전자전산학과 부교수
2003년~현재 서울대학교 전기컴퓨터공학부 부교수
관심분야 : 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, MPSoC

조 정 훈

e-mail : jcho@ee.knu.ac.kr
1996년 KAIST 전기및전자공학과(학사)
1998년 KAIST 전자전산학과(석사)
2003년 KAIST 전자전산학과(박사)
2003년~2005년 하이닉스 반도체 선임연구원
2005년~현재 경북대학교 전자전기컴퓨터학부 전임강사
관심분야 : 임베디드 소프트웨어, 임베디드 시스템 개발 도구, 컴파일러, 전력 최적화