논문 13-38B-01-01

한국통신학회논문지 '13-01 Vol.38B No.01
http://dx.doi.org/10.7840/kics.2013.38B.1.1

효율적인 ASIP 설계를 위한 자동 인스트럭션 확장 시스템 구축

황 덕 호•, 황 선 영°

Construction of an Automatic Instruction-Set Extension System for Efficient ASIP Design

Deok-Ho Hwang•, Sun-Young Hwang°

요 약

본 논문은 어플리케이션에 최적화된 ASIP설계를 하기 위해 MDL을 기반으로 한 Retargetable 컴파일러를 이용한 자동 인스트럭션 확장 시스템을 제안한다. 제안된 시스템은 어플리케이션 프로그램으로부터 얻은 정보를 이용하여 확장 가능한 인스트럭션 후보를 모두 찾는다. 확장 인스트럭션 후보는 하드웨어 라이브러리를 통해 실제 구현 시의 특성에 대한 정보를 얻게 된다. 하드웨어 특성과 수행 속도 향상을 기반으로 주어진 제한 조건에 맞게 인스트럭션 셋을 선택하고 프로세서 구조를 최적화한다. 제안된 시스템의 효용성을 확인하기 위해 다양한 벤치마크 어플리케이션을 이용하여 자동 인스트럭션 확장 시스템을 수행하였다. 제안된 시스템은 기존의 ARM9TDMI의 프로세서로부터 최적화된 인스트럭션 셋과 프로세서 구조를 갖도록 하였다. 제안된 시스템에 의해 설계된 ASIP는 주어진 제한 조건에 따라 기존 프로세서와 비교하면 평균 33.5%의 수행 사이클이 감소하는 것으로 확인되지만, 프로세서의 면적은 증가하는 것으로 측정되었다.

Key Words : ASIP, MDL, Retargetable compiler, Architecture exploration, Instruction-set extension

ABSTRACT

This thesis proposes an automatic instruction extension system that utilizes retargetable compiler, based on MDL, to design an ASIP optimized for application. The proposed system uses information gathered from the application program to find all possible expandable instruction candidates. Expandable instruction candidates acquire the realization characteristics through hardware library. The system chooses instruction set and optimizes processor structure satisfying constraints on the bases of hardware characteristics and increase in execution speed. To confirm the efficiency of the proposed system, automatic instruction extension system was performed using various benchmark applications. The proposed system acquired optimized instruction set and processor structure, which are expanded from the commercial version of ARM9TDMI. Experimental results show that number of execution cycle has been reduced by 33.5% when compared to conventional version of ARM9TDMI, while area has been slightly increased.

Ⅰ. 서 론

멀티미디어 이동통신, 영상 압축, 암호 분야 등

※본 연구는 삼성전자(주)의 지원으로 수행되었으며 IDEC에서 제공한 CAD tool을 이용해 simulation을 수행 하였습니다.
• 주저자 : 서강대학교 전자공학과 CAD & ES 연구실, ejrgh119@sogang.ac.kr, 준회원
° 교신저자 : 서강대학교 전자공학과 CAD & ES 연구실, hwang@sogang.ac.kr, 종신회원
논문번호 : KICS2012-11-550, 접수일자 : 2012년 11월 26일, 최종논문접수일자 : 2012년 12월 11일

의 어플리케이션의 수행을 위한 임베디드 시스템의 사용이 증가함에 따라 어플리케이션에 대한 수요자의 요구를 만족하기 위한 시스템 반도체의 설계, 개발이 중요하게 되었다. 임베디드 시스템 설계에서 핵심 기능을 단일 칩에 집약하고 소모 전력을 최소화시키면서 어플리케이션에 대한 실행 성능 개선은 갈수록 중요해지고 있다. 또한, 수요자의 요구가 높아짐에 따라 시스템 설계 복잡도가 증가하여 time-to-market을 만족하게 하기 어렵게 되었다. 이런 측면에서 ASIC(Applic- ation Specific Integrated Circuits)은 특정 어플리케이션에 대해 최적화된 성능을 가진 프로세서의 설계는 가능하지만 한정된 flexibility를 가지므로 설계 복잡도가 증가할수록 time-to-market을 만족하게 하기 어렵게 된다. ASIP(Applic- ation Specific Instruction-set Processor)은 특정 어플리케이션에 적합한 인스트럭션 셋과 프로세서 구조를 가지므로 특정 어플리케이션에 최적화된 프로세서의 설계가 가능하고 높은 flexibility로 짧은 설계 기간에 사용자의 요구 사항을 만족시킬 수 있다[1]. MDL(Machine Description Language)을 이용한 retargetable 컴파일러 생성은 인스트럭션 변경에 적응 가능하므로 설계시간을 줄일 수 있고, 하나의 기술로부터 프로세서 설계와 컴파일러를 동시에 얻을 수 있어 ASIP 설계 능력을 향상할 수 있다. 임베디드 프로세서의 특성상 고유한 인스트럭션을 가지는 가변 인스트럭션 셋을 추출하는 기술은 어플리케이션의 성능을 가장 많이 향상 할 수 있는 방법이다. 확장 인스트럭션 설계를 위해서는 해당 어플리케이션의 소스 파일을 수행하여 자주 사용되는 지점의 코드 블록을 분석하여 새로운 확장 인스트럭션으로 정의하고 하드웨어 프로세서를 추가해야 한다. 이렇게 생성된 확장 인스트럭션은 입력 어플리케이션을 구동하는데 최적화되어 해당 ASIP의 성능을 최대화시킬 수 있다.

본 연구에서는 ASIP 설계에서 MDL 기반의 설계 방식을 수행하기 위해 SMDL(Sogang Machine Description Langu- age) 시스템을 사용한다[2]. SMDL로 기술된 프로세서에 최적화된 컴파일러를 자동 생성해주는 SRCC(Sogang Retarge- table Compiler Compiler)를 통해 프로세서의 구조와 인스트럭션 셋 확장을 자동으로 구성하는 기법에 대해 제안한다. 컴파일러를 통해 타겟 머쉰 코드로 변환된 어플리케이션 코드를 이용하여 DFG(Data Flow Graph)를 구성한다. DFG를 서브 그래프로 나누어 확장 가능한 인스트럭션 후보를 추출한다. 추출된 인스트럭션 후보들은 하드웨어 특성화 과정을 거치면서 실제 구현될 하드웨어에 대한 정보를 포함하게 된다. 이후 필터링 과정을 거치면서 불필요하고 구현 불가능한 후보들을 필터링을 통해 제외하고 남은 후보들 중 사용자의 요구조건에 맞는 최적의 인스트럭션 셋을 하드웨어 정보를 이용하여 선택한다. 이후 SRCC를 업데이트하여 새로운 인스트럭션을 적용하고 하드웨어 프로세서를 추가한다.

본 논문의 구성은 다음과 같다. 2절에서는 관련 연구를 기술하며 3절에서는 구축된 자동 인스트럭션 확장 시스템의 개관과 각 과정에 대해 보인다. 4절에서는 실험 결과를 보이며, 5절에서 결론 및 추후 과제를 제시한다.

Ⅱ. 관련 연구

2.1. Background

ASIP 설계를 위하여 configurable 프로세서 기반의 설계 방식과 MDL 기반의 설계 방식이 제안되었다[3-6]. configurable 프로세서 기반의 설계 방식은 기본 프로세서와 인스트럭션 셋을 바탕으로 인스트럭션 확장과 파라미터의 수정을 통해 원하는 ASIP를 얻는 방식이다. Tensilica의 Xtensa가 대표적이며 프로세서와 소프트웨어 툴 셋은 쉽게 얻을 수 있지만, 기본 프로세서 모델을 바탕으로 설계자가 프로세서 모델의 파라미터만을 수정하여 ASIP를 설계하므로 제공되는 프로세서의 범주를 벗어나는 다양한 구조를 구현할 수 없는 단점이 있다. MDL 기반의 설계 방식은 설계자가 MDL을 통해 기술한 타겟 프로세서의 모델을 바탕으로 특정 어플리케이션에 최적화된 ASIP의 컴파일러, 시뮬레이터 등의 소프트웨어 툴 체인을 자동 생성하여 다양한 구조를 구현할 수 있고 MDL로부터의 소프트웨어 툴 체인의 자동 생성은 architecture exploration을 수행하여 설계 정확성과 일관성을 검증하고 최적화된 프로세서를 설계할 수 있게 한다[7]. 어플리케이션에 최적화된 ASIP설계를 위해서는 어플리케이션의 분석을 통해 프로세서의 구조와 인스트럭션 셋을 어플리케이션에 맞게 구성해야 한다. Retargetable 컴파일러는 상위레벨로 기술된 어플리케이션을 최적화된 타겟 프로세서의 하위레벨 코드로 변환시킨다. 컴파일러의 전처리 과정에 의해 변환된 중간 형태는 어플리케이션 코드의 정적 분석을 가능하게 하여 어플리케이션의 특성을 분석할 수 있다. 이러한 분석을 통해 해당 어플리케이션의 수행을 위한 최적화된 성

능을 가진 인스트럭션과 프로세서의 구성이 가능하다. 설계의 Time-to-market은 갈수록 중요해지고 있어 많은 비용과 시간이 있어야 하는 컴파일러의 설계를 자동화하는 시스템의 중요성이 주목받고 있다. retargetable 컴파일러는 ASIP를 설계할 때 인스트럭션 변경에 따른 컴파일러를 생성할 수 있으므로 설계 시간을 크게 줄일 수 있어 그 효용성이 증가하고 있다[8-10]. 어플리케이션에 최적화된 인스트럭션 셋을 자동으로 구성하는 기술 역시 매우 중요고 특히 어플리케이션의 성능을 가장 많이 향상할 수 있는 확장 인스트럭션 셋을 추출하는 기술은 임베디드 프로세서에서 갈수록 효용성이 증대되고 있다. 이전 연구에서 확장 인스트럭션을 추출하기 위해 근사 알고리즘 통하는 방식이 제시되었다[11]. 분석된 DFG상에서 MISO(Multiple Input Sin- gle Output) 형태의 확장 인스트럭션을 찾아내는 알고리즘이지만 이 알고리즘은 복잡도가 지수적으로 증가함에 따라 단순한 형태로만 존재하고 이러한 단순 근사 알고리즘은 정확한 확장 인스트럭션 셋을 추출하기에는 어려움을 보인다. 또한, 반복되는 패턴의 서브 그래프를 찾기 위해 템플릿을 생성하고 매칭하여 코드를 생성하는 방법이 제시되었다[12]. 특정 어플리케이션에 대해 성능 향상은 얻을 수 있지만, 기존에 구축된 템플릿에 근거함으로 flexibility가 떨어진다. 다른 방식으로 휴리스틱에 의존하여 중요하지 않은 인스트럭션 후보를 우선 제외해 최적의 인스트럭션을 선택하는 방법이 연구 되었다[13,14]. 이 방식에서 인스트럭션 후보의 비교를 위해 기준이 되는 하드웨어 정보를 FPGA에 기반을 둔 표준 셀 라이브러리를 사용해 미리 계산된 면적과 수행 시간을 사용한다. 하지만 FPGA로 구현 시의 최적화 과정을 고려하지 않은 단순 합으로 계산하므로 정확한 정보를 통해 비교가 이루어지지 않는다. DFG에서 가능한 모든 인스트럭션 후보를 Binary Tree Search를 이용하여 찾는 방법도 제시되었다[15]. 정확도 높은 확장 인스트럭션을 구현하지만 모든 인스트럭션을 후보에 대해 Binary Tree Search를 수행함으로써 많은 시간이 소요된다. 인스트럭션 후보를 찾는 방법에 대해서는 다른 여러 가지 접근 방식들이 제시되었다[16-18].

본 연구에서는 SMDL 시스템을 사용하여 자동 인스트럭션 확장 시스템을 구현함으로써 기존의 연구들에서와같이 기구축된 프로세서에서 FPGA를 사용하여 확장 인스트럭션을 수행하기 위한 하드웨어를 구현하지 않고 SMDL의 임베디드 코어 생성기를 통해 직접 하드웨어로 구현할 수 있도록 하였다. FPGA를 사용하지 않음으로써 불필요한 회로를 줄이고 실제로 구현될 하드웨어프로세서의 성능을 저장한 하드웨어 라이브러리를 바탕으로 인스트럭션 후보들을 비교하여 선택하므로 선택된 확장 인스트럭션의 신뢰도를 높이고 세분화된 제한 요건 설정이 가능하게 하였다. 또한 기존의 연구들에서 인스트럭션 선택을 위해 각 확장 인스트럭션에 대해 greedy 알고리즘을 사용하여 비교한 것에 반해 전체 확장 인스트럭션 셋에 simulated annealing을 수행하여 각 부분에 대한 비교가 아닌 전체 어플리케이션의 수행 성능을 비교하였으므로 기존의 greedy 알고리즘을 통한 비교보다 높은 성능을 갖는 확장 인스트럭션 셋을 구현하게 하였다.

2.2. SMDL 시스템 개관

SMDL 시스템은 프로세서에 관한 기술을 목표로 하여 만들어진 MDL로서, ASIP 설계 자동화의 한 부분을 구성한다. SMDL 기술은 ASIP 설계 자동화 시스템의 근간이 된다. 구축된 SMDL 시스템은 타겟 코어에 대한 합성 가능한 코드를 자동 생성하는 임베디드 코어 생성기(Embedded Core Generator), 인스트럭션 셋 시뮬레이터를 자동 생성하는 RISGen (Retargetable Instruction-Set Simulator Generator), 타겟 코어에 대한 머쉰 코드를 생성하는 retargetable 컴파일러인 SRCC (Sogang Retargetable C Compilier)로 이루어진다. SMDL 언어는 타겟 아키텍처의 인스트럭션에 대한 정보와 저장 모듈, 실행 유닛 등의 리소스 구조에 대한 정보와 파이프라인의 구조를 기술할 수 있는 C언어의 형태를 띠고 있는 머쉰 기술 언어이다. SMDL은 크게 structural description을 하는 내/외부 구조 기술부와 behavioral descr- iption을 하는 인스트럭션 셋 기술부로 나뉜다. 내부 구조 기술부에서는 타겟 프로세서의 내부 리소스들 및 아키텍처에 대하여 정의된다. 외부 구조 기술은 프로세서 이름 정의 섹션, 외부 포트 및 인터럽트에 대한 정의 섹션으로 나뉜다. SMDL의 사용자는 내/외부 기술을 바탕으로 하여 타겟 코어의 ISA(Instruction-Set Architecture)에 대해 structural de- scription을 지원한다. 인스트럭션 셋 기술부는 인스트럭션 bitwidth와 field를 정의하고 어드레싱 모드, 인스트럭션을 정의한다. 특히 인스트럭션의 행위 정보에 관한 기술은 C 언어와 유사한 syntax가 제공되어 상위수준에서의 효율적 기술이 가능하다.

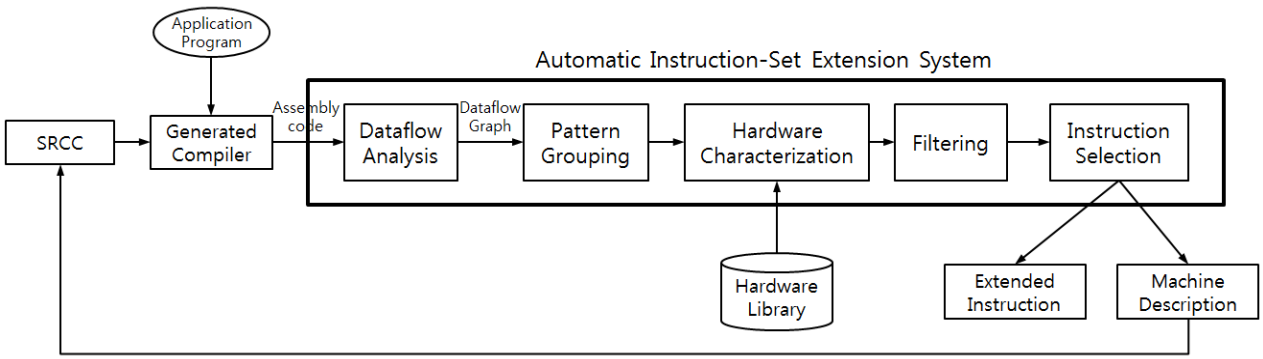

그림 1. 제안된 자동 인스트럭션 확장 시스템 개관
Fig. 1. Proposed Automatic Instruction-Set Extension System

2.3. SRCC 시스템 개관

SRCC 시스템은 SMDL로 기술된 타겟 프로세서의 행위정보로부터 전처리 과정을 통해 중간 형태를 만들고, 라이브러리의 행위정보를 같은 형태의 중간 형태로 변환한다. 변환된 SMDL의 중간 형태와 SRCC 라이브러리의 중간 형태를 맵핑함으로써 lburg[19]의 코드 선택기 기술을 생성한다. SMDL의 레지스터 파일 사용기술과 맵핑된 인스트럭션을 이용하여 컴파일러의 후위부 interface function을 생성하고 생성된 코드 선택기와 interface function은 LCC[20,21]의 front-end와 결합하여 SMDL로 기술된 타겟 프로세서를 위한 컴파일러를 생성한다. SRCC는 LISA와 같은 one-to-one, one-to-many, many-to-one 맵핑 방식을 사용한다[22,23]. SRCC는 행위기술 맵핑을 통해 라이브러리를 맵핑함으로써 컴파일러 자동 생성이 LISA와 비교하면 효율적으로 이루어진다. One-to-many 맵핑을 통해 one-to-one 맵핑되지 않는 인스트럭션을 정의된 인스트럭션의 조합을 통해 서브젝트 트리에 맵핑되어 사용 빈도수가 적은 특정 트리 패턴만을 위한 인스트럭션을 줄일 수 있다. Many-to-one 맵핑 방식은 여러 서브젝트 트리를 하나의 인스트럭션에 맵핑하는 방식으로 특정 연산의 반복이 많은 ASIP에서 코드 사이즈를 줄이고 수행속도를 높여 수행성능 향상에 큰 요인이 될 수 있다.

Ⅲ. 제안된 자동 인스트럭션 확장 시스템

본 논문은 주어진 제한 조건을 만족하면서 자동으로 어플리케이션에 최적화된 인스트럭션 셋을 구성하고 프로세서의 하드웨어를 추가하는 것을 목표로 한다. C로 기술된 어플리케이션은 컴파일러를 통해 타겟 머쉰 코드로 변환되고 변환된 타겟 머쉰 코드를 이용해 dataflow analysis를 수행한다. dataflow analysis는 확장 가능한 인스트럭션을 찾기 위해 타겟 머쉰 코드의 DFG를 형성한다. 형성된 DFG를 서브 그래프로 나누어 확장 인스트럭션 후보를 추출하고 추출된 확장 인스트럭션을 주어진 제한 조건에 맞게 필터링해 구현할 수 있도록 한다. 필터링 된 확장 인스트럭션 후보들을 하드웨어 라이브러리를 통해 실제 합성 시의 하드웨어 특성을 파악하고 하드웨어 특성 정보를 이용하여 주어진 제한 조건에 만족하도록 확장 인스트럭션을 선택한다. 그림 1은 제안된 자동 인스트럭션 확장 시스템 흐름의 개관을 보인다.

3.1. 패턴 그룹화(Pattern Grouping)

Dataflow analysis를 통해 형성된 DFG에서 확장 가능한 인스트럭션 후보를 추출한다. 확장 인스트럭션 후보 추출을 위해 형성된 DFG를 각 노드에서 depth-first로 서브 그래프를 나누는데 이때 종료지점은 노드에서 더 이상 엣지가 없을 경우, 엣지의 도착 노드가 load이거나 branch 인스트럭션인 경우

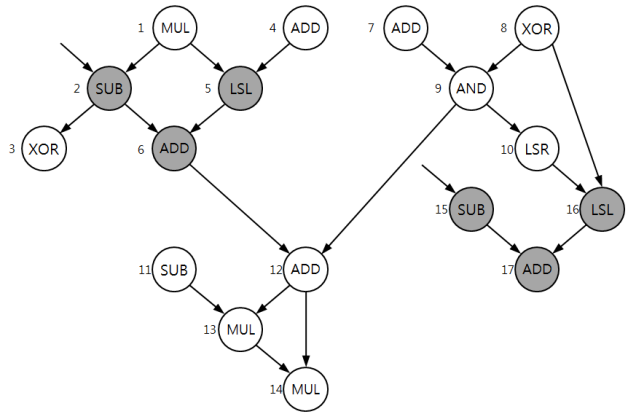

그림 2. 패턴 그룹화를 위한 DFG 예제
Fig. 2. Example of DFG for pattern grouping

이다. Load 인스트럭션의 경우 메모리에 접속해야 하므로 수행시간이 미리 정해져 있지 않아서 load 인스트럭션의 수행 결과가 나올 때까지 시간제한에 걸리지 않고 어떤 operation을 수행해야 할지 결정할 수 없다. Branch 인스트럭션의 경우 branch 뒤의 operation은 상황에 따라 수행되기 때문에 예측하는 것이 불가능하다.

그림 2의 예제에서 노드 12에서의 서브 그래프는 노드 12-13, 노드 12-13-14, 노드 12-14로 나누어진다. 또한, 노드 1-2-3과 노드 6-7-8은 같은 오퍼레이션은 갖는 같은 패턴이므로 하나의 패턴으로 구현할 수 있다. 따라서 같은 패턴을 찾아 하나의 패턴으로 매칭하는 과정이 요구된다. 그림 2의 예제에서 노드 2-5-6과 같은 패턴을 찾기 위해 모든 SUB 오퍼레이션 노드를 찾는다. 예제에서 노드 2, 11, 15에 해당하고 이 노드들에서 부분적으로 비교를 수행한다. 부분적으로 노드 2-6, 노드 2-3, 노드 11-13, 노드 15-17을 비교하여 같은 패턴을 찾는다. 노드 2-6과 노드 15-17가 같은 것을 확인하고 이후 비교 결과가 확인 가능할 때까지 나머지 노드들에서도 비교를 수행한다. 같은 패턴을 찾는 작업을 모든 패턴에 대해 수행하여 하나의 패턴에 매칭한다.

3.2. 하드웨어 특성화(Hardware Characterization)

추출된 패턴은 하드웨어 라이브러리를 통해 각 패턴의 하드웨어 특성을 얻을 수 있다. 각 하드웨어 특성은 이후 필터링과 인스트럭션 선택 과정에서 각 패턴을 비교하기 위한 기준이 된다. 하드웨어 라이브러리는 각 오퍼레이션을 수행하는 하드웨어 셀의 수행 시간, 면적, I/O 포맷, 파워 특성을 포함하고 이 정보를 이용하여 각 패턴의 하드웨어 특성을 분석한다. 수행 시간은 각 패턴의 가장 긴 패스에서의 셀 수행 시간의 합으로 나타낼 수 있으며 이는 클럭 사이클과 연동하여 각 패턴이 수행되는 사이클 수를 나타낼 수 있다. 면적은 각 패턴을 하나의 인스트럭션으로 합성하기 위해 하드웨어를 추가해야 하므로 패턴 내의 각 하드웨어 셀의 면적의 합으로 나타낼 수 있다. 이때 각 하드웨어 셀 간의 inter-connect가 추가로 발생하고 패턴마다 컨트롤 로직과 디코드 로직이 추가로 발생하며 레지스터 파일의 사이즈가 달라진다. 하지만 레지스터 파일의 사이즈와 컨트롤 로직과 디코드 로직은 하나의 패턴이 아닌 모든 패턴에 의해 변화하고 각 패턴이 끼치는 영향을 나누어 측정하기 불가능하므로 제외

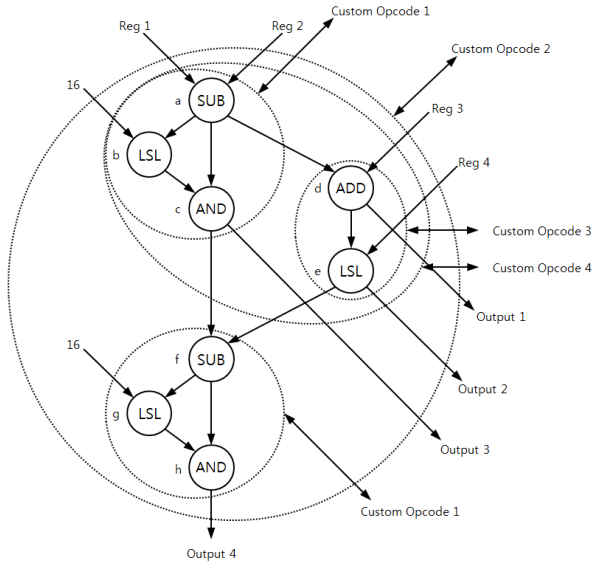

그림 3. 필터링을 위한 DFG 예제
Fig. 3. Example of DFG for filtering

된다.

3.3. 필터링

필터링 과정을 통해 생성된 불필요한 패턴들은 모두 제거된다. 제거되는 패턴에는 레지스터 파일의 포트 제한 조건을 만족하지 않는 경우, 합성된 인스트럭션이 수행 속도 감소로 이어지지 않는 경우, 인스트럭션이 사이즈가 너무 길어지는 경우를 포함한다. 인스트럭션 합성 때문에 critical path는 변화될 수 있지만, 이는 매 합성 시마다 critical path를 찾아야 하는 과정이 포함되므로 최초의 critical path에 대해서만 필터링을 수행하였다. 그림 3의 예제의 DFG에서 critical path는 노드 a-b-c-f-g-h의 path이다. 여기서 노드 d-e를 합성하는 경우는 시스템 수행 속도에 영향을 미치지 않으므로 제외된다. 이 경우 노드 a-b-c을 합성하는 경우가 가장 이상적이다. 노드 a-b-c의 합성으로 생긴 확장 인스트럭션은 노드 f-g-h에서 반복되므로 각 패턴을 한 사이클에 수행할 수 있게 한다. 전체 DFG를 하나의 인스트럭션으로 합성하는 custom opcode 3의 경우 4개의 입력과 4개의 출력이 필요하므로 레지스터 파일이 8개의 포트가 있어야 하게 된다. 또한, 8개의 오퍼랜드를 가져 인스트럭션 사이즈가 너무 길어지므로 제외된다.

3.4. 인스트럭션 선택(Instruction Selection)

필터링 후 실제 구현할 확장 인스트럭션을 선택하는 과정을 거친다. 서로 다른 확장 인스트럭션 후

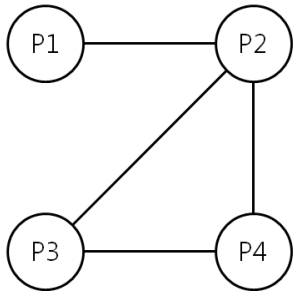

그림 4. Conflict graph 예제
Fig. 4. Example of Conflict graph

보가 같은 노드를 포함하고 있을 수 있다. 같은 노드를 포함하여 겹치게 된 확장 인스트럭션을 동시에 사용하는 것은 바이너리 맵핑 과정을 매우 복잡하게 만들 뿐 아니라 그로 인한 성능 향상이 매우 작으므로 확장 인스트럭션 후보들을 conflict graph를 사용하여 겹치지 않도록 하여 확장 인스트럭션 셋들을 구성한다.

그림 4는 그림 3의 예제의 확장 인스트럭션 후보에 대한 conflict graph를 나타낸다. 각 custom opcode 1, 2, 3, 4는 패턴 P1, P2, P3, P4에 맵핑되었다. 위의 conflict graph를 통해 P1-P3, P1-P4, P2의 확장 인스트럭션 셋들을 얻을 수 있다.

각 확장 인스트럭션 셋들 중 최저 비용을 갖는 확장 인스트럭션 셋을 찾기 위해 사용자가 요구하는 비용 함수로 simulated annealing을 수행한다. 하드웨어 특성화 과정에서 추출한 각 패턴의 하드웨어 특성을 바탕으로 수행 시간, 면적, I/O 포맷, 파워 특성을 이용해 비용 함수를 결정한다. 비용 함수의 설정에 따라 사용자가 원하는 설계를 할 수 있다. 예를 들어 프로세서를 설계할 때 주어진 면적 내에서 최소의 수행 시간이 나오도록 설계하거나, 파워 소모와 수행 시간을 최소화시키는 설계가 가능하다. 본 연구에서는 제한된 확장 인스트럭션 사이즈 내에서 최대의 속도 향상을 위해 critical path에서의 확장 인스트럭션으로 인한 사이클 감소를 측정하여 최대치가 되도록 비용함수를 설정하였다. 비용 함수가 결정되면 초기 어플리케이션 코드를 기반으로 확장 인스트럭션 셋을 선택적으로 적용하여 비용을 평가하고 더 낮은 비용의 해가 발견되면 새로운 해로 선택한다. 이후 더 이상 낮은 해가 발견되지 않을 때까지 반복적으로 수행하여 최적의 확장 인스트럭션 셋을 찾는다. 볼츠만 확률값에 따라 높은 비용을 가질 수도 있지만, 내부의 루프가 반복됨에 따라 제어변수 T가 낮아지면서 새로운 확장 인스트럭션으로 이루어진 해의 이동은 점점 비용이 낮아지는 방향으로 수렴하게 된다. 제어변수 T가 충분히 낮은 값에 이를 때 확률적으로 최저에 가까운 낮은 비용의 상태에 머물게 되고 그때의 해를 최종 확장 인스트럭션 셋으로 선택한다.

3.5. 출력

인스트럭션이 선택되면 새로운 확장 인스트럭션으로 정의하고 하드웨어 프로세서를 추가하기 위해 machine description과 확장 인스트럭션의 하드웨어 형태가 출력되며, 확장 인스트럭션의 하드웨어 형태는 이후 HDL(Hardware Description Language)로 실제 하드웨어를 구현하기 위해 사용된다. 하드웨어 중간 형태는 SMDL의 임베디드 코어 생성기의 인스트럭션 정보를 업데이트하여 확장 인스트럭션이 적용된 ASIP을 구현하고 생성된 machine description은 구현된 하드웨어에서 동작할 수 있도록 SRCC를 업데이트하여 새로운 확장 인스트럭션을 적용하도록 컴파일러를 수정한다. 생성된 컴파일러는 확장 인스트럭션이 적용되면서 어플리케이션의 DFG를 변화시켜 critical path가 바뀔 수도 있다. 사용자가 원하는 성능이 구현될 때까지 반복적으로 인스트럭션 확장 시스템을 적용 시켜 최적화된 인스트럭션 셋을 찾는다.

Ⅳ. 실험 결과

제안된 자동 인스트럭션 확장 시스템의 검증을 위해 4가지 벤치마크 어플리케이션 adpcm decoder, g721 encoder, jpeg decoder, blowfish에 대해 최적화된 인스트럭션 셋과 프로세서 구조를 설계했다. adpcm decoder, g721 encoder, jpeg decoder는 MediaBench[24]에서 인용하고 jpeg decoder는 Mibench[25]에서 인용하였다. 초기 프로세서는 ARM9TDMI[26]에 기반을 둔 프로세서로 설정하고 이를 SMDL을 통해 구현해 벤치마크 어플리케이션을 수행하고 자동 인스트럭션 확장 시스템을 적용해 어플리케이션 분석을 통한 최적화된 인스트럭션 셋과 하드웨어 프로세서를 재구성하였다.

표 1, 2는 초기 프로세서에서 수행한 벤치마크 어플리케이션에 대해 제안된 시스템이 적용된 최종 프로세서의 수행 사이클 감소와 면적의 증가를 보인다. 면적의 증가 수치는 공정 과정에 따라 다른 수치를 나타내게 되므로 90nm 공정을 통한 ARM9TDMI를 자동 인스트럭션 확장 시스템을 적

표 1. 프로세서의 수행 성능 (합성되는 오퍼레이션이 3개 이하일 때)
Table 1. Processor performance (when less then three operations are synthesized)

Benchmark Function	Number of Patte-rns	Number of Initial Processor Execution Cycles	Number of Final Processor Execution Cycles	Execution Cycles Comparison (%)	Chip Area Compari-son (%)
adpcm decoder	47	6,124,744	3,780,706	- 36.7	4.6
g721 encode	69	258,536,428	175,875,121	- 29.6	12.3
jpeg decoder	64	1,356,089	821,872	- 37.1	26.3
blowfish	16	10,641,422	6,865,434	- 30.6	5.9
평 균				- 33.5	12.3

표 2. 프로세서의 수행 성능 (합성되는 오퍼레이션이 5개 이하일 때)
Table 2. Processor performance (when less then five operations are synthesized)

Benchmark Function	Number of Patterns	Number of Initial Processor Execution Cycles	Number of Final Processor Execution Cycles	Execution Cycles Comparison (%)	Chip Area Compar-ison (%)
adpcm decoder	102	6,124,744	2,698,125	- 54.3	8.9
g721 encode	213	258,536,428	165,728,479	- 32.9	24.1
jpeg decoder	222	1,356,089	774,908	- 41.5	65.2
blowfish	57	10,641,422	5,347,448	- 49.2	11.0
평 균				- 44.5	27.3

용했을 때의 추가되는 면적을 표시하였다. 표 1은 레지스터 파일의 포트 제한과 인스트럭션 사이즈 제한을 만족하기 위해 필터링 과정에서 서브 그래프의 오퍼레이션 개수를 3개 이하로 제한한 경우이다. 합성되는 오퍼레이션의 개수를 3개 이하로 제한했을 때는 평균 33.5%로 수행 사이클이 감소하였고 12.3%의 면적 증가가 있었다.

표 2는 확장 인스트럭션 사이즈가 커짐에 따른 수행 성능을 비교하기 위해 합성되는 오퍼레이션 개수를 5개 이하로 증가시켰을 때의 수행 성능을 보인다. 합성되는 오퍼레이션의 개수를 5개 이하로 제한했을 때는 평균 44.5%로 수행 사이클이 감소하였고 추가 되는 면적은 평균 27.3%로 확인할 수 있다.

매칭 되는 패턴의 개수는 확장 인스트럭션 사이즈 제한이 완화됨에 따라 증가하였지만, 레지스터 파일의 입출력 제한이 2-input, 1-output에서 4-input, 2-output으로 증가하였다. 전체 칩 면적은 합성되는 오퍼레이션의 개수가 3개 이하일 때에 비해 5개 이하일 때 증가하였다. 확장 인스트럭션의 사이즈 증가는 프로세서 속도의 향상을 가져오지만 각 확장 인스트럭션의 가용성이 떨어지게 되고 전체 칩 면적의 증가를 가져온다. 실험 결과에서 확장 인스트럭션의 사이즈를 증가시켰을 때 수행 속도는 평균 11% 향상하지만, 전체 칩 면적이 평균 15% 증가되고 본 연구에서는 고려하지 않은 레지스터 파일의 사이즈, control/decode logic, inter-connect 회로의 사이즈가 증가하므로 실제 구현된 프로세서에서는 칩 면적 차이는 더 크게 나타나게 된다.

V. 결론 및 추후과제

본 논문은 MDL기반의 프로세서 설계 방식을 통해 프로세서를 설계한다. ASIP 설계에서 필수적인 retagetable 컴파일러를 생성하는 SRCC를 통해 확장 인스트럭션을 자동으로 결정하는 시스템을 제시한다. 어플리케이션의 성능을 최대로 높일 수 있도록 확장 인스트럭션을 자동으로 결정하여 어플리케이션에 최적화된 확장 인스트럭션 셋을 찾는 방법을 제시하였다. 컴파일된 어플리케이션의 DFG에서 정적인 분석을 통해 자주 사용되거나 병목이 되는 지점의 코드를 분석하여 새로운 확장 인스트럭션으로 정의한다. 정의된 확장 인스트럭션을 수행하기 위한 하드웨어 프로세서를 추가하고 컴파일러를 수정하여 차후 반복적인 수행이 가능하도록 한다. ARM9TDMI를 기반으로 둔 프로세서에서 벤치마크 어플리케이션을 통해 새로운 프로세서를 생성하여 제시된 시스템의 효용성에 대해 검증하였다. 생성된 프로세서는 평균 33.5% 수행 사이클 감소를 보였다.

추후과제는 어플리케이션의 성능을 더욱 향상시키기 위해 어플리케이션을 정적인 분석뿐 아니라 동적인 분석을 포함하여 사용 빈도가 높은 패턴을 추출해야 하고 시스템을 적용해 인스트럭션이 합성되면서 변화하는 critical path에 적응 가능한 시스템 개발이 요구된다.

Reference

[1] M. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies : survey and issues," in *Proc. IEEE/ACM Int. Conf.* VLSI Design. (VLSI 2001), pp. 76-81, Bangalore, India, Jan. 2001.

[2] J. B. Cho, Y. H. Yoo, and S. Y. Hwang, "Construction of an automatic generation system of embedded processor cores," *J. KICS*, vol. 30, no. 6A, pp. 526-534, Jun. 2005.

[3] R. Gonzalez, "Xtensa : a configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60-70, Mar./Apr. 2000.

[4] A. Hoffmann, Tim Kogel, Achim Nohl, Gunnar Braun, Oliver Schliebusch, Oliver Wahlen, Andreas Wieferink, and Heinrich Meyr, "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.

[5] 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.

[6] A. Fauth, J. Van Praet, and M. Freericks, "Describing instructions set processors using nML," i*n Proc. European Design & Test Conf.*, pp. 503-507, Paris, France, Mar. 1995.

[7] R. Leupers, M. Hohenauer, J. Ceng, H. Scharwaechter, H. Meyr, G. Ascheid and G. Braun, "Retargetable compilers and architecture exploration for embedded processors," *IEEE Proc. Computers and Digital Techniques*, vol. 152, no. 2, pp. 209-223, Mar. 2005.

[8] P. Marwedel and G. Goosens, *Code Generation for Embedded Processors*, Kluwer Academic Publishers, pp. 14-31, 1995.

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

[10] R. Leupers, "Compiler design issues for embedded processors," *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 51-58, Jul./Aug. 2002.

[11] A. Alippi, "Determining the optimum extended instruction set architecture for application -specific reconfigurable VLIW CPUs," in *Proc. IEEE Int. Workshop on Rapid System Prototyping*, pp. 25-27 Jun. 2001.

[12] R. Kastner, A. Kaplan, S. Memik, and E. Bozorgzadeh, "Instruction generation for hybrid reconfigurable systems," *ACM Trans. Des. Autom. Embed. Syst.*, vol. 7, no. 4, pp. 605-627, Oct. 2002

[13] N. Clark, H. Zhong, and S. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transac. Comput.*, vol. 54, no. 10, pp. 1258-1270, Oct. 2005

[14] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Trans. Comp. Aid. D.*, vol. 23 no. 2 pp. 216-228. Feb. 2004.

[15] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. Comp. Aid. D.*, vol. 25, no. 7, pp. 1209-1229, Jul. 2006.

[16] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 69-78, Sep. 2004,

[17] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, pp. 183-189, Feb. 2004,

[18] X. Chen, D. Maskell, and Y. Sun, "Fast identification of custom instructions for extensible processors," *IEEE Trans. Comp. Aid. D.*, vol. 26, no. 2, pp. 359-368. Feb. 2007

[19] 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, Apr. 1992.

[20] C. Fraser and D. Hanson, *A Retargetable C Compiler : Design and Implementation*,

Ben-jamin/Cummings, 1995.

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

[22] 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, March. 2005.

[23] 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”. *J. VLSI Sig. Proc.*, vol. 43, no. 2-3, pp. 235-246, Jun, 2006

[24] C. Lee, M. Potkonjak, and W. Mangione-Smith, “MediaBench: a tool for evaluating and synthesizing multimedia and communications systems,” in *Proc. IEEE/ACM Int. Symp. on Microarchitecture*, pp. 330-335, Dec 1997.

[25] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, and T. Mudge, R. Brown, “MiBench: a free, commercially representative embedded benchmark suite,” i*n Proc. IEEE Int. Workshop on Workload Characterization*, pp. 3-14, Dec. 2001

[26] ARM, ARM922T Technical Reference Manual (rev 0), 2001.

황 덕 호 (Deok-Ho Hwang)

2011년 2월 서강대학교 전자공학과 학사

2011년~현재 서강대학교 전자공학과 CAD & ES 연구실 석사과정

<관심분야> ASIP Design, Embedded System 설계

황 선 영 (Sun-Young Hwang)

1976년 2월 서울대학교 전자공학과 학사

1978년 2월 한국과학원 전기 및 전자공학과 공학석사

1986년 10월 미국 Stanford 대학 전자공학 박사

1976년~1981년 삼성반도체(주)연구원, 팀장

1986~1989년 Stanford 대학 Center for Integrated System 연구소 책임연구원 및 Fairchild Semiconductor Palo Alto Research Center 기술자문

1989~1992년 삼성전자(주) 반도체 기술 자문

1989년 3월~현재 서강대학교 전자공학과 교수

<관심분야> SoC 설계 및 framework 구성, CAD 시스템, Computer Architecture 및 DSP System Design 등