컴파일방식 시뮬레이션 기법을 이용한 ASIP 어셈블리 시뮬레이터의 성능 향상

김호영*, 김탁곤**

Performance Improvement of ASIP Assembly Simulator Using Compiled Simulation Technique

Ho Young Kim, Tag Gon Kim

Abstract

This paper presents a retargetable compiled assembly simulation technique for fast ASIP(application specific instruction processor) simulation. Development of ASIP which satisfies design requirements in various fields of applications such as telecommunication, wireless network, etc. needs formal design methodology and high-performance relevant software environments such as compiler and simulator. In this paper, we employ the architecture description language(ADL) named HiXR^2 to automatically synthesize an instruction-level compiled assembly simulator. A compiled simulation has benefit of time efficiency to interpretive one because it performs instruction fetching and decoding at compile time. Especially, in case of assembly simulation, instruction decoding is usually a time-consuming job(string operation), so the compiled simulation of assembly simulation is more efficient than that of binary simulation. Performance improvement of the compiled assembly simulation based on HiXR^2 is exemplified with an ARM9 architecture and a CalmRISC32 architecture. As a result, the compiled simulation is about 150 times faster than interpretive one.

Key Words: ASIP, ADL, compiled simulation, assembly simulation

* 한국과학기술원 전자전산학과
** 한국과학기술원 전자전산학과 교수

1. 서론

반도체 기술의 발전에 따른 집적도 증가와 더불어, 디지털 통신 및 무선 통신 등의 새로운 응용분야의 출현은 하나의 칩 안에 점점 더 많은 기능을 요구하게 되었다. 이로 인하여, 프로세서 설계 방법은 메모리, 프로세서, 기능 유닛(functional unit), 시스템 버스 등의 시스템 구성 요소들을 분리하여 보드에 설계하던 방식에서 하나의 칩 안에 모든 시스템 구성 요소를 설계하는 이른바, 시스템칩(system on a chip) 설계 방식으로 바뀌게 되었다. 시스템칩에 따른 시스템 복잡도의 증가와 시장 선점에 대한 욕구는 시스템 설계의 효율적인 방법론을 필요로 하게 되었다.

프로세서는 시스템의 중추적인 역할을 하는 시스템의 가장 중요한 구성 요소이다. 프로세서는 용도에 따라서 범용 프로세서와 ASIP으로 나뉜다. ASIP(application specific instruction processor)은 특정한 응용분야 혹은 몇 개의 응용분야에 적합하도록 설계된 프로세서이다. 따라서 ASIP설계는 범용 프로세서와는 다르게 응용분야에 따라서 설계가 크게 달라지며 시장 선점 또한 매우 중요하다. 그럼에도 불구하고, 프로세서 설계 및 관련 개발 환경의 설계는 아직 자동화되지 못하고 각각을 직접 설계해야 하는 실정이다. 특히 주어진 설계 후보군들 중에서 사용자의 요구 사항을 만족시키는 설계를 찾는 설계 공간 탐색은 일반적으로 설계 및 평가가 반복적으로 이루어지기 때문에 바뀐 프로세서 구조에 따라서 개발 환경을 각각 만들어야 하는데, 각각의 구조에 따라 직접 프로그램을 개발하는 것은 시간도 많이 소모할 뿐만 아니라, 에러가 발생하기도 쉽다.

이에 따라 등장한 것이 바로 프로세서 구조 기술 언어인 ADL과 시뮬레이터 자동 합성 기법이다. ADL에서 프로세서 구조를 추상화 수준에 따라 필요한 부분만 기술한 후, 그 정보를 이용하여 시뮬레이터를 생성한다. 이러한 접근방법은 ADL 기술을 간단히 수정함으로서 쉽게 프로세서 구조를 변경하고, 그에 따른 시뮬레이터를 에러 없이 자동생성 시킬 수 있다는 장점을 갖는다.

ASIP설계는 넓은 설계 후보군을 가지고 있기 때문에 효율적인 설계 공간 탐색을 위해서는 계층적인 방법이 필요하다. 즉, 프로세서 구조를 추상화 수준에 따라서 몇 가지 수준으로 나누고 각각에 대해서 설계 공간 탐색을 수행한다면, 훨씬 효과적이다. 크게, 시간을 제외한 프로세서의 명령어(instruction) 수준의 행위만을 나타내는 부분과 파이프라인 구조의 시간정보까지 포함하는 부분으로 나눌 수 있다. 프로세서의 명령집합을 결정하고 컴파일러/시뮬레이터를 이용하여 주어진 응용 프로그램에 대하여 검증한 후, 파이프라인 구조를 결정하는 계층적인 방법은 설계의 효율을 돕는다.

효율적인 설계 공간 탐색을 위해서 ADL을 이용한 프로세서 타겟팅 뿐만 아니라, 높은 성능의 시뮬레이터를 합성하는 것 또한 중요하다.[1] 프로세서 구조가 바뀜에 따라서 각각의 프로세서에 대한 시뮬레이션이 반복되기 때문에 높은 성능의 시뮬레이터는 탐색 시간을 크게 줄일 수 있게 된다. 컴파일방식 시뮬레이션의 요지는 빈번히 반복되는 비효율적인 부분을 시뮬레이터를 합성하는 컴파일 타임에 한 번만 수행하고 시뮬레이션 런-타임에는 동적인 부분만을 수행할 수 있도록 하여 전체 시뮬레이션 수행시간을 단축시키는 기법이다.[2][3] 시뮬레이션 런-타임에 바뀌지 않는 정적인 부분의 비효율성에 따라서 컴파일방식의 시뮬레이션은 해석방식의 시뮬레이션에 비해 빨라지는 정도가 달라진다.

본 연구에서는 ADL을 이용하여 프로세서 명령어 수준에서 시뮬레이터를 합성한다. 보다 효율적인 설계를 위하여 명령어의 이진(binary) 이미지를 결정하지 않은 상태인 어셈블리 수준의 시뮬레이션을 수행한다. 이진 이미지를 결정하는 일 또한 설계 후보군의 탐색이 필요하므로 보다 이른 설계 단계에서의 검

증 후, 이진 이미지를 결정하는 것은 이진 이미지까지 한번에 탐색하여 결정하는 방법보다 효과적이다. HiXR^{2}라는 ADL과 컴파일방식의 어셈블리 시뮬레이터 합성기법은 유연성 있는 프로세서 타겟팅과 이른 설계 단계에서의 빠른 시뮬레이션을 통한 검증을 가능하게 하여, 효율적인 설계 공간 탐색을 돕는다.

본 논문은 다음과 같이 이루어져 있다. 2장에서는 관련 연구에 대하여 이야기 한다. 3장에서는 컴파일방식 시뮬레이션 기법과 해석방식 시뮬레이션 기법의 차이를 이야기하고 어셈블리 시뮬레이션에서 컴파일방식 시뮬레이션 기법의 효과에 대해 논한다. 4장에서는 제시된 ADL인 HiXR^{2}에 대하여 간략히 설명하고, 5장에서는 ARM9와 CalmRISC32 프로세서에 대한 컴파일방식 시뮬레이션 실험 결과를 해석방식과 비교하여 속도 향상을 보이고, 6장에서 결론 및 추후과제를 보인다.

2. 관련 연구

전통적인 프로세서 개발은 Verilog HDL과 VHDL등의 하드웨어 기술 언어(HDL)를 이용하여 RT(register transfer) 수준에서 이루어졌다. HDL 모델은 하드웨어 구조적인 정보를 전부 가지고 있기 때문에, 바로 하드웨어 합성이 가능하다는 장점이 있다. 그러나 이러한 모델로부터 프로세서 평가 및 검증을 위한 컴파일러/시뮬레이터 등을 개발하는 것은 HDL 모델의 복잡도로 인하여 매우 어렵다. 특히 RT 수준의 정보를 담고 있는 시뮬레이터는 시뮬레이션 속도가 매우 느릴 뿐만 아니라, 빠른 속도를 갖는 명령집합 시뮬레이터의 개발에 필요한 프로세서 명령어 등의 정보를 HDL 모델을 이용하여 알아내기는 매우 힘들다는 단점을 갖는다. 따라서 프로세서 구조를 바꾸어 가며 성능 평가 및 검증을 하는데 HDL은 적합하지 않다.

이러한 단점을 극복하기 위하여, 최근에는 명령어 수준의 프로세서 기술 언어들이 등장하였다. 이러한 연구들은 컴파일러/시뮬레이터에 필요한 정보들만 모아서 프로세서 기술을 간략화하고, 이 기술을 이용하여 컴파일러와 시뮬레이터를 합성해 내는 특징을 갖는다. nML[4]은 명령어를 기반 프로세서를 기술하도록 개발되어, HDL에 비해 매우 적은 정보로 프로세서의 컴파일러와 시뮬레이터를 만들어 낸다. 그러나 시뮬레이터는 해석방식으로 실제 하드웨어에 비하여 매우 느린 속도를 갖는다. nML이후, EXPRESSION[6], ISDL[7]과 같은 ADL이 개발되었다. 이러한 언어들은 nML을 확장하여 VLIW와 같은 좀더 복잡한 구조들을 표현할 수 있다. 그러나 nML과 마찬가지로 시뮬레이션 속도가 느리다는 단점을 갖는다. LISA[5]는 우리의 연구와 가장 유사하다. 다른 ADL과 다르게 컴파일 방식의 시뮬레이션을 수행하여 시뮬레이터의 성능을 개선하였다. 그러나 명령어의 이진 이미지까지 결정된 후, 시뮬레이션이 가능하기 때문에, 컴파일방식의 시뮬레이션 기법을 이용하였는데도 불구하고 성능향상이 수배 정도로 그리 높지 않다.[8]

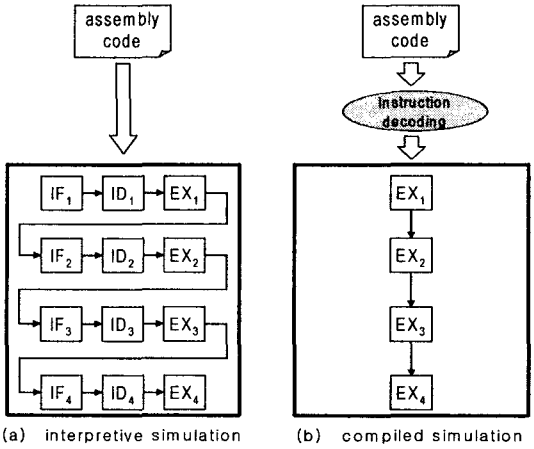
우리의 연구는 명령어 기반의 ADL을 이용하여, 컴파일 방식의 어셈블리 시뮬레이터를 생성한다. 이진 수준이 아닌 어셈블리 수준에서 합성함으로서 보다 이진 이미지를 결정하지 않고, 보다 이른 설계 단계에서 프로세서를 검증할 수 있다. 이른 설계 단계에서의 검증은 계층적인 설계 공간 탐색을 좀더 효율적으로 할 수 있게 해준다. 특히 이진 이미지는 전력 소모와 밀접한 관련이 있기 때문에 더 낮은 수준에서 하드웨어 구조를 정할 때 결정하는 편이 유리하다. 또한 어셈블리 시뮬레이션에 컴파일방식 시뮬레이션을 적용함으로서 많은 시간을 요구하는 문자연산 감소로 해석방식의 시뮬레이션 보다 매우 큰 성능향상을 얻는다.

3. 컴파일방식 프로세서 시뮬레이션

프로세서는 ASIC(application specific integrated circuite)과 같이 특정한 응용분야에 따라 정해진 일을 수행하도록 만들어진 하드웨어와는 다르게 로딩되는 응용프로그램에 따라서 유연성 있게 여러 가지 일을 하도록 설계 되었다. 응용프로그램을 이해하기 위하여 프로세서는 응용프로그램의 명령어들을 페치, 디코딩, 실행하는 과정을 반복한다.

컴파일방식 시뮬레이션의 목적은 시뮬레이션 수행시간을 줄이는 것이다. 해석방식의 명령어 수준의 프로세서 시뮬레이션은 하드웨어에서 일어나는 행위를 그대로 모사한다. 따라서 프로세서 시뮬레이터는 현재의 프로그램 카운터(PC) 값에 따라 응용 프로그램의 명령어를 페치하고(fetch), 해당 명령어를 디코딩하여 명령어의 의미를 파악하고 피연산자들을 저장소자 혹은 즉치값(immediate value)으로 페치한 후, 디코딩 결과에 따라 실행(execution)한다. 응용프로그램을 시뮬레이션하는 것은 결국 페치-디코딩-실행의 반복이 된다. 이러한 일련의 과정은 실제 프로세서에서는 전기적 신호에 따라서 순식간에 이루어지지만, 시뮬레이션에서는 훨씬 많은 시간을 필요로 한다. 컴파일방식 시뮬레이션은 시뮬레이션 해야 하는 일들 중 시뮬레이션 런-타임 때 변하지 않고 반복적으로 수행되는 정적인 부분을 시뮬레이션 런-타임 이전에 미리 한번만 수행하여, 시뮬레이션 런-타임 수행시간을 줄이는 테크닉이다.

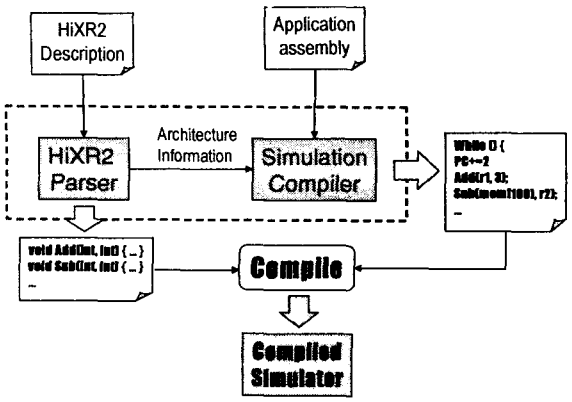
일반적으로 ASIP 응용 프로그램들은 코드 안에 많은 루프를 가지고 있다. 프로그램 코드 자체는 길지 않지만, 많은 루프로 인하여, 명령어들을 반복적으로 수행한다. 또한, 런-타임에 달라지는 것은 메모리나 레지스터 각각의 저장 소자의 값일 뿐이고, 명령어자체는 변하지 않는다. 따라서 런-타임에 반복되는 명령어를 계속하여 페치하고 디코딩 하는 것은 매우 시간 소모적인 일이다. 컴파일방식 시뮬레이션에서는 응용 프로그램의 명령어 페칭 및 디코딩을 런-타임 이전에 수행한다.



<그림 1> 해석방식과 컴파일방식 시뮬레이션 비교

<그림 1>은 해석방식 시뮬레이션과 컴파일방식 시뮬레이션을 간략히 비교한 그림이다. 컴파일방식 시뮬레이션에서는 시뮬레이션 문자 해독기(parser)에서 응용 프로그램 어셈블리를 받아들여 시뮬레이션 런-타임 이전에 페치 및 디코딩을 하고, 실행 부분만을 런-타임에 수행하도록 만든다. 그 결과로서 해석방식의 시뮬레이션에서는 런-타임에 계속해서 명령어 페치, 디코딩, 실행이 시뮬레이션 되는 반면, 컴파일방식에서는 실행 부분만이 시뮬레이션 되어 큰 성능 향상을 얻을 수 있다. 특히 명령어 워드가 이진수가 아닌 어셈블리로 이루어진 경우, 어셈블리 페치 및 디코딩을 위하여 문자(string)를 나누고 구분하는 문자 연산을 많이 하게 되는데, 문자 연산은 정수 연산에 비하여 일반적으로 훨씬 더 많은 시간을 소모한다. 따라서 어셈블리 시뮬레이션에서 컴파일방식의 시뮬레이션 기법을 사용하면, 이진 시뮬레이션 보다 높은 성능 향상을 얻을 수 있다.

컴파일방식 어셈블리 시뮬레이터 합성에 대한 전체 프레임웍은 <그림 2>와 같다.



<그림 2> 전체 프레임웍

HiXR^2 문자 해독기는 프로세서 명령어 정보 및 메모리, 레지스터 등의 저장소자 정보를 갖는 HiXR^2 기술을 받아들여 저장 소자를 데이터 구조를 만들고, 각 명령어의 행위 정보를 의미 있는 C/C++ 코드로 변환한다. 이와 동시에 시뮬레이션 컴파일러는 수행할 응용 프로그램의 어셈블리 정보와 HiXR^2 파서로부터 명령어 구조 정보를 입력으로 받아서 명령어 및 오퍼런드들의 페칭 및 디코딩을 미리 수행한다. 페칭 및 디코딩 결과는 호스트 컴퓨터(시뮬레이션을 수행하는 컴퓨터)에서 수행될 수 있는 형태의 C/C++ 코드로 변환되고, HiXR^{2}에서 생성된 명령어 행위 정보를 갖는 C/C++ 코드와 함께 컴파일 하여 실행 가능한 컴파일방식 시뮬레이터를 생성한다.

4. HiXR^2 언어

HiXR^{2}는 프로세서의 컴파일러와 명령어 수준의 시뮬레이터를 자동 생성하기 위하여 고안된 언어이다. 프로세서의 동작과 명령어의 의미를 표현하기 위하여 HiXR^{2}는 저장 소자 정보를 나타내는 Storage, 명령어의 페치될 오퍼런들의 페치법을 나타내는 Addressing mode, 마지막으로 명령어가 갖는 오퍼런드들과 그 오퍼런드를 이용한 명령어의 행위 정보를 갖는 Instruction의 세 가지 섹션으로 나누어 기술한다.

4.1 Storage

Storage는 타겟팅 하고자 하는 프로세서의 메모리, 레지스터 파일, 상태 레지스터 등의 저장 소자들의 개수와 어셈블리 표현법을 기술한다. <그림 3>의 기술은 Storage를 기술하는 하나의 예제로서 1000개의 dmem[1]~dmem[1000]으로 표현되는 메모리 블럭과 r1~r16으로 표현되는 레지스터파일을 나타낸다.

```
Storage arm9x_Storage
{
        Memory dmem {
                nummemcells 1000
                type byte
        }

        RegisterFile GR {
                numofregs 16
                naming r
                type word
        }
}
```

<그림 3> Storage 기술

4.2 Addressing mode

명령어 어셈블리는 연산기호(mnemonic)와 오퍼런드(operand)로 이루어진다. 연산기호는 덧셈, 뺄셈 등의 연산의 행위를 나타내고, 오퍼런드들은 피연산자가 어떤 저장소자를 참조하는지 혹은 즉치값으로 사용되는지를 나타낸다. 이 섹션에서는 오퍼런드들의 의미 및 표현법을 규정한다. 이 정보를 이용하여 명령어 디코딩 시에 페치할 값을 구할 수 있다. <그림 4>는 어드레싱 모드 중 하나인 LSL_IMM 어드레싱 모드를 기술한 것이다. LSL_IMM은 즉치값 하나와 GR이라는 레지스터파일의 레지스터값 하나를 이용하여 명령어에서 사용될 피연산자의 값을 구한다. 예를 들어, "R1 ;

LSL #10"과 같은 오퍼런드는 레지스터 R1의 값을 왼쪽 논리이동(logical shift)을 즉치값 10만큼 한 후, 그 값을 명령어의 피연산자로 사용한다는 뜻이다.

```
AddrMode LSL_IMM : arm9x_AddrMode
{
        operands
        {
                2
                IMMEDIATE imm
                GR Rm
        }

        syntax {
                Rm ";""LSL " imm
        }

        action {
                return lshift(Rm,imm);
        }
}
```

<그림 4> LSL_IMM addressing mode 기술

하나의 명령어의 오퍼런들이 여러 개의 어드레싱 모드를 갖는 경우가 있을 수 있는데, 이것을 해결하기 위하여 화합(compound) 어드레싱 모드를 제공한다. 즉, ADD라는 명령어가 레지스터 다이렉트 값과 레지스터 다이렉트 값을 더할 수도 있지만, 레지스터 다이렉트 값과 즉치값을 더하는 경우도 있을 경우에 화합 어드렝싱 모드가 사용된다. 화합 어드레싱 모드에서는 하나의 오퍼런드가 가질 수 있는 어드레싱 모드를 나열한다. <그림 5>는 즉치값과 레지스터 다이렉트 모드를 갖는 화합 어드레싱 모드의 예시이다.

```
AddrMode REG_IMM : arm9x_AddrMode {

        Compound {
                IMMEDIATE
                REG
        }
}
```

<그림 5> Compound addressing mode

4.3 Instruction

Instruction은 연산기호 및 오퍼런드들의 의미, 그리고 오퍼런드들이 갖는 어드레싱 모드를 규정한다. <그림 6>은 "ADD" 명령어의 기술의 한 가지 예이다. 연산기호는 ADD를 사용하고, 각 오퍼런드의 어드레싱 모드는 레지스터 다이렉트 모드 두 개와 LSL_IMM 어드레싱 모드 한 개를 사용하여, 두 번째 레지스터 다이렉트 모드 값과 LSL_IMM 어드레싱 모드 값을 더하여 첫 번째 레지스터 다이렉트 모드의 레지스터에 넣는다. 이 기술에 따르면, ADD 명령어는 다음과 같은 형태의 어셈블리로 나타난다.

ADD R1, R2, R3; LSL #10

이 명령어의 의미는 'R3의 값을 10만큼 왼쪽 논리이동을 한 값과 R2의 값을 더하여 R1에 넣는다.'이다.

```
Instruction add : ALU
{
        mnemonic "ADD"

        operands
        {
                3
                REG Rd : DST
                REG Rs : SRC
                LSL_IMM Opmd2 : SRC
        }

        action
        {
                Rd = add(Rs, Opmd2);
        }
}
```

<그림 6> ADD instruction 기술

Instruction에서는 기술의 편의성을 위해서 계층적인 기술 방법을 제공한다. 몇 개의 명령어를 묶어서 하나의 집합으로 만들고 다시 그 집합들을 묶어서 하나의 집합으로 만들 수 있다. <그림 7>과 같이 add, sub 등의 산술 명령어들을 ArithmeticOPs 라는 명령어 집합으

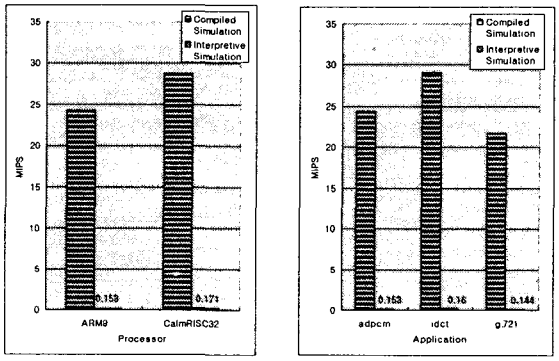
로 묶어 표현할 수 있다.

```
Instruction_Set ArithmeticOPs : arm9x_InstructionSet {
        Subset {
                add
                sub
                subs
                subges
                rsb
                mul
                mla
                cmp
                cmn
                and
                eor
                orr
        }
}
```

<그림 7> Instruction의 계층적인 기술

5. 실험 결과

컴파일방식 시뮬레이션의 성능 향상을 측정하기 위하여, 프로세서 구조와 응용 프로그램을 바꾸어 보며 실험을 하였다. 시뮬레이션을 수행한 호스트 컴퓨터는 Athlon 2100+ (1.72GHz), 512MB이고, Windows XP를 기반으로 실험하였다. 사용된 프로세서는 ARM사의 ARM9 프로세서와 삼성에서 개발된 CalmRISC32 프로세서이고, 사용된 응용 프로그램은 통신과 멀티미디어에 주로 사용되는 MediaBench[9]이다. <그림 8> (a)는 MediaBench 중 한가지인 ADPCM 응용 프로그램을 프로세서 구조를 바꾸어 가며 실험을 한 결과이고, <그림 8> (b)는 ARM9 프로세서 구조에 대하여 응용 프로그램을 ADPCM, g.721, idct(inverse discrete cosine transform)으로 바꾸어 가며 실험을 한 결과이다. 실험결과는 각각의 경우에 대하여 컴파일방식 어셈블리 시뮬레이션은 해석방식 어셈블리 시뮬레이션에 비해 약 150배 이상의 성능향상이 있었다. 이러한 성능향상은 어셈블리 시뮬레이션의 문자 연산이 없어지면서 가능해진 결과로, 앞의 관련 연구[8]의 이진 시뮬레이션의 결과와는 다른 높은 성능향상이다.



(a) 프로세서 변화에 따른 결과 (b) 응용 프로그램 변화에 따른 결과

<그림 8> 실험 결과

6. 결론

본 논문에서는 ASIP 시뮬레이션을 효율적으로 하기 위한 컴파일방식 어셈블리 시뮬레이터 자동 생성 기법을 나타내었다. HiXR^{2}를 기반으로 간단한 기술 변경을 통하여 프로세서 구조를 쉽게 바꿀 수 있는 재적응성을 가질 뿐만 아니라, 매우 빠른 시뮬레이션 속도를 갖기 때문에, 프로세서 설계 공간 탐색이나 응용 프로그램의 코드 검증 시에 매우 유용하게 쓰일 수 있다. 뿐만 아니라, 이진 수준이 아닌, 어셈블리 수준의 시뮬레이션을 지원함으로서 이진 이미지가 결정되지 않은 이른 설계 단계에서 프로세서 검증이 가능하다는 장점을 갖는다. 특히 어셈블리 시뮬레이션에 적용된 컴파일방식 시뮬레이션 기법은 많은 시간을 소요하는 문자 연산을 시뮬레이션 런-타임에 제거하여 높은 시뮬레이션 성능 향상을 얻을 수 있었다. 본 연구의 정확성 및 성능 검증은 널리 사용되는 ARM9 프로세서 구조와 삼성에서 개발된 RISC 프로세서인 CalmRISC32 프로세서 구조를 이용하여 증명하였고, 응용 프로그램은 멀티미디어에 많이 사용되는 MediaBench가 쓰였다. 컴파일방식 어셈블리

시뮬레이션은 해석방식 어셈블리 시뮬레이션에 비하여 150배가 넘는 속도 향상이 있었다.

추후 계획 사항으로는 HiXR^{2}의 표현력을 확장하여, 명령어 수준의 병렬성을 나타내고 그에 따라 VLIW와 Super Scalar 머신의 컴파일방식 시뮬레이션을 수행할 수 있도록 하는 것이다. 또한, 현재의 명령어 수준의 시뮬레이션에서 파이프라인 구조 정보까지 컴파일방식 시뮬레이션 할 수 있도록 확장하는 일이 필요하다.

참고문헌

[1] M. Hartoog, J. Rowson, et al., "Generation of software tools from porcessor descriptions for hardware /software codesign," in Proc. of the Design Automation Conference (DAC), Jun. 1997.

[2] V. Zivojnovic, S. Tjiang, and H. Meyr, "Compiled simulation of programmable DSP architectures," in Proc. of IEEE Workshop on VLSI Signal Processing, pp. 187-196, Oct. 1995

[3] S. Pees, V. Zivojnovic, A. Ropers, and H. Meyr, "Fast Simulation of the TI TMS 320C54x DSP," in Proc. Int. Conf. on Signal Processing Application and Technology, pp. 995-999, Sep. 1997.

[4] A.Fauth; J.Van Praet: M. Freericks: Describing instruction set processors using nML. Proceedings of European Design and Test Conference, pp. 503-507, Paris France, 1995.

[5] A. Halambi, P. Grun, et al., "EXPRESSION: A language for architecture expoloration through compiler/simulator retargetability," in Proc. of the Conference on Design, Automation & Test in Europe, Mar. 1999

[6] G. Hadjiyiannis et al.: ISDL: An instruction set description language for retargetability. In Proc. DAC, 1997

[7] V. Zivojnovic, S. Pees, and H. Meyr, "LISA - machine description language and generic machine model for HW/SW co-design," in Proceedings of the IEEE Workshop on VLSI Signal Processing, Oct. 1996

[8] R. Leupers, J. Elste, B. Landwehr "Generation of Interpretive and Compild Instruction Set Simulators", in Proc. of the Asia & South Pacific Design Automation Conference(ASP-DAC), 1999

[9] C. Lee, M. Portkonjak, W. H. Mangione-Smith "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems", in Proc. IEEE/ACM International Symposium on, pp. 330-335, 1997

저자소개

김호영

1999 한국과학기술원 전기 및 전자공학과 학사

2001 한국과학기술원 전자전산학과 석사

2001 ~ 현재 한국과학기술원 전자전산과 박사과정

관심분야: 시스템 모델링 시뮬레이션, 프로세서 기술언어 및 시뮬레이션, 프로세서 성능 예측

김탁곤

1988 아리조나대학교 전자/컴퓨터공학과 공학박사

1980 ~ 1983 국립수산대학(현: 부경대학교) 통신공학과 전임강사

1987 ~ 1989 아리조나 환경연구소 연구 엔지니어

1989 ~ 1991 캔사스 대학교 전자전산학과, 조교수

1991 ~ 1998 한국과학기술원 전기 및 전자공학과 조/부교수

1998 ~ 현재 한국과학기술원 전자전산학과 교수

관심분야: 모델링/시뮬레이션 방법론 및 환경 개발, 시뮬레이션 기반 시스템 분석.