신속한 ASIP 성능 평가를 위한 재적응성을 갖는 컴파일러/시뮬레이터 프레임웍

오세종* · 김호영* · 김탁곤*

Retargetable Compiler/Simulator Framework for Rapid Evaluation of ASIP

Se Jong Oh · Ho Young Kim · Tag Gon Kim

Abstract

이 논문은 빠른 ASIP(application specific instruction processor) 평가를 위한 재적응성을 가진 컴파일러/시뮬레이터 환경에 대해 이야기 한다. ASIP의 성능은 하드웨어 구조뿐만 아니라, 수행되는 응용 소프트웨어에 영향을 받기 때문에, 높은 성능의 ASIP 개발을 위해서는 컴파일러 및 시뮬레이터의 개발이 선행되어야 한다. 그러나 다양한 ASIP 구조에 따라 적합한 고성능의 컴파일러/시뮬레이터를 만드는 일은 매우 시간 소모적인 일이 될 뿐만 아니라, 오류가 발생하기도 쉽다. 본 논문에서는 HiXR2라는 ADL(architecture description language)을 이용하여 명령어 구조를 기술하고 이를 바탕으로 컴파일러와 시뮬레이터를 자동 생성하였다. HiXR2의 재적응성 및 생성된 컴파일러/시뮬레이터의 정확성을 검증하기 위하여 ARM9 프로세서와 CalmRISC32 프로세서 구조를 각각 기술하고, 각각에 대하여 응용프로그램 코드를 컴파일 및 시뮬레이션 하는 예제를 보였다.

1. 서론

ASIP 개발은 군사, 상업적으로 사용되는 응용 프로그램의 핵심이 되는 중요한 기술이다. 뿐만 아니라, 시스템칩(SOC)과 같은 ASIP을 이용한 시스템 집적을 위해서는 이른 설계 단계에서 ASIP을 다른 시스템 요소들과 함께 시뮬레이션 하고 검증하는 일이 필요하다. 그러나 ASIP은 다른 하드웨어 요소와는 달리 응용프로그램에 따라 다른 일을 수행하기 때문에 ASIP 시뮬레이션을 위해서는 응용프로그램을 머신코드로 만들어주는 컴파일러가 필요하다. 따라서 컴파일러의 성능에 따라 ASIP 시뮬레이션 결과가 달라질 수 있게 되고, 결국 설계 단계에서 정확한 ASIP 성능평가를 위해서는 높은 수준의 코드 최적화가 가능한 컴파일러가 필요하다. 이와 더불어 시스템의 빠른 성능평가를 위해서는 높은 성능의 시뮬레이터가 필요하다. 따라서 컴파일러와 시뮬레이터의 성능은 시스템 개발에 큰 영향을 끼친다.[1]

정해진 프로세서 구조에 국한하여 컴파일러와 시뮬레이터를 개발하는 것은 시스템 집적 시 프로세서의 변화에 따라 각각의 컴파일러/시뮬레이터를 개발해야 할 뿐만 아니라, 에러가 발생하기도 쉽기 때문에 매우 시간 소모적인 일이 된다. 이에 따라 등장한 것이 바로 프로세서 구조 기술 언어인 ADL과 컴파일러/시뮬레이터 자동

* 한국과학기술원 전자전산학과 전기 및 전자공학전공

합성 기술이다.[2][3] ADL에서 프로세서 구조를 필요한 부분만 추상화 하여 기술한 후, 그 정보를 이용하여 시뮬레이터를 생성한다. 이러한 접근은 ADL 기술을 간단히 수정함으로서 쉽게 프로세서 구조를 변경하고, 그에 따라 컴파일러와 시뮬레이터를 에러 없이 자동생성 시킬 수 있다는 장점이 있다. 그러나 하나의 프로세서에 최적화 되지 않은 형태이므로 직접 설계한 컴파일러/시뮬레이터에 비해 성능이 떨어질 수 있다는 단점을 갖는다.

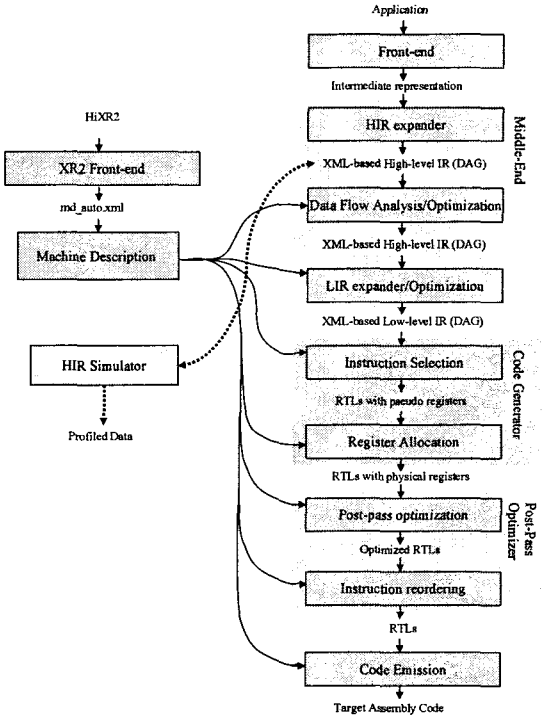
본 논문에서는 HiXR^{2}라는 ADL과 컴파일러/시뮬레이터 자동생성 기술을 보인다. 뿐만 아니라, 고성능의 컴파일러/시뮬레이터를 얻기 위하여 사용한 몇 가지 기법에 대하여 이야기 한다.

2장에서는 재적응 가능한 컴파일러의 구조에 대하여 설명하고, 3장에서는 이러한 컴파일러를 어떻게 생성할 수 있는지 설명한다. 4장에서는 고성능 시뮬레이터 자동생성 기법에 대하여 설명하고, 5장에서 ARM9와 CalmRISC32 프로세서 구조로 재적응성 및 컴파일러/시뮬레이터 자동생성의 정확성을 보인다. 마지막으로, 6장에서 결론 및 추후과제를 보인다.

Ⅱ. 컴파일러의 구조

컴파일러의 대상 아키텍처(target architecture)를 변경하기 위해서는 크게 세 가지 방법이 존재한다. 사용자에 의하여 대상 아키텍처에 대한 정보를 제공하는 방법과, 개발자가 컴파일러에서 아키텍처에 종속적인 부분의 소스코드를 수정하는 방법, 마지막으로 몇 개의 아키텍처에 대한 컴파일러를 만들고 나서 사용자 선택에 따라 코드를 생성하는 것이다. 현재 사용되고 있는 많은 컴파일러는 개발자에 의하여 수정되는 방식을 가지고 있으며, 소수의 컴파일러[6][7]만이 사용자가 제공하는 아키텍처 정보에 의해 대상 아키텍처를 위한 코드를 생성한다. 그러나 이러한 컴파일러는 대상 아키텍처가 상당히 고정되어 있거나 어느 정도 일정한 구조를 가진 아키텍처에만 코드를 생성할 수 있다.

이 논문에서 소개하는 CaT[10] 컴파일러는 사용자가 아키텍처의 정보를 제공하여 원하는 아키텍처에 대한 코드를 생성할 수 있는 컴파일러이다. 또한, 최대한 많은 아키텍처를 효율적으로 지원할 수 있도록 아키텍처 기술 언어 외에 중간 표현(Intermediate Representation)의 생성에서도 사용자가 개입할 수 있는 기술언어를 제공하고 있다. 마지막으로 최적화 컴파일러 개발을 위하여 컴파일러 개발자가 쉽게 컴파일러를 이해하고 발전시킬 수 있도록 모든 부분이 객체지향으로 설계되어 있다.



〈그림 1〉 CaT 컴파일러 구조

CaT 컴파일러의 구조는 그림 1과 같으며, 크게 프론트엔드(front-end), 미들엔드(middle-end), 및 백엔드(back-end)로 나뉘어 있다. 프론트엔드에서는 상위수준 언어(high level language)의

파싱을 담당하며, 미들엔드에서는 중간 표현에서 최적화 및 분석과정을 수행하며 상위 수준 언어에 종속적인 프론트엔드와 아키텍처에 종속적인 백엔드가 서로 영향을 받지 않도록 다리 역할을 한다. 백엔드에서는 코드 생성 및 아키텍처 종속적인 최적화 과정을 수행한다. 프론트엔드의 상위수준 언어는 원하는 언어를 중간 표현으로 변환하는 변환기를 작성함으로써 여러 언어를 사용할 수 있으며, 현재는 C 언어를 사용하기 위하여 LCC[8]를 프론트엔드로 사용하고 있다. 미들엔드는 크게 상위수준 중간 표현(High IR)과 하위수준 중간표현(Low IR)으로 나뉜다. 상위 수준 중간 표현은 상위 수준 언어나 아키텍처에 비종속적인 표현으로 아키텍처에 비종속적인 코드 분석이나 최적화를 수행할 수 있다. 또한 상위 수준 언어나 아키텍처에 관계없이 프로그램의 프로파일을 얻을 수 있도록 시뮬레이터를 가지고 있다. 반면에 하위수준 중간표현은 추상화 수준이 더 낮아져 아키텍처 정보에 종속적인 형태를 가지고 있다. 백엔드에서는 하위수준 중간표현을 받아 코드 생성 알고리즘을 수행한 후 RTL(Register Transfer List)[9]들을 생성하게 된다. RTL들을 바탕으로 아키텍처 종속적인 최적화 알고리즘들을 수행할 수 있다.

미들엔드의 일부분과 백엔드는 아키텍처에 종속적이므로 사용자가 제공하는 아키텍처 정보에 따라 자동적으로 변경되어야 한다. 사용자가 제공해야하는 정보와 컴파일러 재적응에 대한 자세한 내용은 다음 장에서 설명한다.

Ⅲ. 컴파일러 생성

원하는 대상 아키텍처에 대하여 컴파일러를 생성하기 위하여 사용자가 기술해야할 부분은 두 가지로 나누어진다. 첫 번째는 아키텍처의 명령어 집합을 기술하는 $HiXR^2$이고, 두 번째는 상위수준 중간표현에서 하위수준 중간표현으로 변환하는 변환 규칙에 대한 기술이다.

$HiXR^2$의 명령어 집합 기술은 아키텍처에서 지원하는 기본 연산자(micro operator)의 기술, 기억 장치(storage)에 대한 기술 및 이들을 조합하여 각 명령어들을 기술하는 부분으로 나누어진다. 각 명령어는 RTL로 표현되며, RTL은 아키텍처에서 동시에 수행되는 RT(Register Transfer)들로 구성된다.

$$\text{Instruction} \equiv \text{RTL} \equiv \{RT_1, RT_2, RT_3\}$$

각 RT는 lvalue = rvalue 형태의 표현식이며, rvalue의 계산 결과가 lvalue에서 지정한 기억 장치로 저장된다. rvalue에서의 연산자(operator)로 기술된 기본 연산자들을 사용할 수 있으며, 피연산자로는 레지스터, 메모리, 상수 등을 사용할 수 있다.

컴파일러의 명령어 선택기(instruction selector)에서는 자동적으로 명령어를 생성하기 위하여, 먼저 매칭(matching)과정을 통해서 하위 수준 중간 표현을 표현할 수 있는 모든 RT들의 조합을 찾고, 최적의 명령어를 선택하기 위한 알고리즘을 통해 필요한 RT들만 남긴 다음, RT들을 모아서 RTL들을 형성한다. 이러한 RTL들을 가지고 데이터 종속관계에 따라 스케줄링(scheduling)하고 레지스터 할당(register allocation)을 하고 나면 대상 아키텍처에 대한 명령어들이 생성된다.

상위 수준에서 하위 수준으로의 중간 표현 변환은 컴파일러만이 가지는 기술 언어를 통해 변환 규칙을 기술할 수 있다. 이러한 기술을 하는 이유는 상위 수준 표현이 아키텍처 종속적인 하위 수준으로 변환하면서 자동적으로 변환하기 어려운 부분이나 함수의 호출 약속(calling convention)에 대한 기술을 하기 위함이다. 변환 규칙은 상위수준 중간 표현의 각 연산자에 대하여 해당 연산자를 $HiXR^2$에 기술된 기본 연산자

들의 조합으로 표현하는 방법을 정의하며, 필요에 따라서 특정 기억장치를 사용하여 연산하도록 기술할 수 있다.

변환 알고리즘은 상위수준 중간표현을 후진트리검색(post-order traverse)으로 각 노드(node)를 탐색하면서 변환 규칙에 따라 하위 수준 중간표현의 표현식을 생성하고 이를 스택에 저장한다. 종단(terminal)이 아닌 노드에서는 자식 노드들에서 스택에 저장한 하위수준 표현식과 현재 노드의 변환 규칙을 이용하여 새로운 하위수준 중간표현을 만든다. 그림 2는 상위수준 중간표현의 add32 연산자에 대한 변환규칙의 간략한 예를 보여주고 있으며, 32-bit 더하기 연산을 하는 상위수준 중간표현을 실제 아키텍처에 존재하는 16-bit 더하기 연산으로 풀어서 하위수준 중간표현을 생성하고 있다.

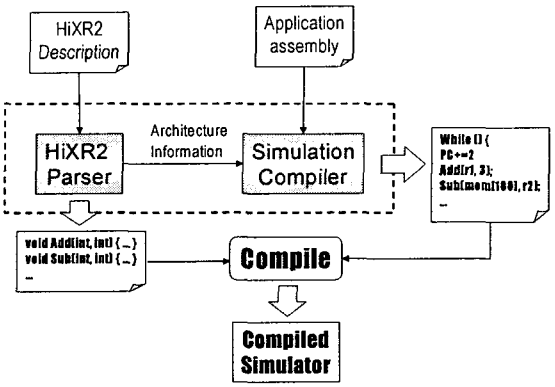
```
at_add32
 push(createOp("micro_add16",pop(),pop()));
 push(createOp("micro_addc16",pop(),pop()));
```

〈그림 2〉 add32에 대한 변환 규칙의 예

대부분의 상위수준의 연산자는 비슷한 기본연산자로 표현될 수 있지만, 함수 호출 약속에서는 두 표현의 형태가 상당히 다를 수 있다. 이는 함수 호출 약속의 형태와 사용하는 명령어가 아키텍처마다 상당히 다르기 때문이며, 사용자는 상위수준의 함수호출 연산자들에 대하여 실제 아키텍처에서 함수 호출에 사용하는 기본 연산자들로 표현해야한다.

Ⅳ. 시뮬레이터 생성

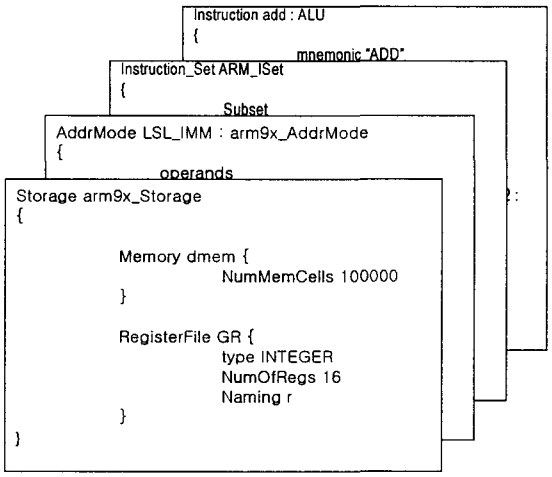
재적응성을 갖는 시뮬레이터 생성은 하드웨어 구조적인 부분과 소프트웨어적인 부분 두 가지로 나누어서 이루어진다. HiXR^2 파서는 인스트럭션 구조정보를 갖는 HiXR^2 기술을 받아들여 각 인스트럭션의 구조정보를 의미 있는 C 코드로 변환한다. 시뮬레이션 컴파일러는 수행할 응용 프로그램의 어셈블리와 HiXR^2 파서로부터 인스트럭션 구조 정보를 이용하여 인스트럭션 및 오퍼런드들의 페칭 및 디코딩을 수행한다. 페칭 및 디코딩 결과는 호스트 컴퓨터(시뮬레이션을 수행하는 컴퓨터)에서 돌릴 수 있는 형태의 C 코드로 변환되고, HiXR^{2}에서 생성된 C 코드와 함께 컴파일 하여 실행 가능한 고성능의 컴파일 방식 시뮬레이터를 생성한다.[4]



〈그림 3〉 시뮬레이터 생성

Ⅴ. 실험결과

컴파일러/시뮬레이터의 재적응성 및 성능평가를 보이기 위하여 ARM사의 ARM9프로세서 구조와 삼성전자에서 개발된 CalmRISC32 프로세서 구조를 각각 HiXR$_2$로 기술하였다. CalmRISC32는 전체 기술 크기가 34.5KB이고 ARM9는 38.1KB로 명령어 집합을 기술하였다. (그림 4) 두 개의 기술을 이용하여 각 프로세서 구조에 맞는 컴파일러와 시뮬레이터를 자동 생성하였고, 컴파일러와 시뮬레이터의 정확성 및 성능을 검증하기 위하여 멀티미디어 관련 벤치마크 프로그램인 MediaBench[5] 응용 프로그램을 사용하였다.



〈그림 4〉 ARM9 프로세서 기술

Ⅵ. 결론 및 추후과제

본 논문에서는 신속한 ASIP 성능 평가를 하기 위한 컴파일러/시뮬레이터 자동 생성 기법을 나타내었다. HiXR^{2}를 기반으로 한 간단한 기술을 통하여 프로세서 구조를 바꿀 수 있는 재적응성을 가질 뿐만 아니라, 컴파일러와 시뮬레이터를 자동 생성하여 각 ASIP 구조를 이른 설계 단계에서 빠르게 평가해 볼 수 있었다. 재적응성 및 컴파일러/시뮬레이터 자동 생성 기법은 프로세서 설계 공간 탐색이나 응용 프로그램의 코드 검증 시에 매우 유용하게 쓰일 수 있다. 본 연구의 정확성 및 성능 검증은 널리 사용되는 ARM9 프로세서와 삼성에서 개발된 CalmRISC32 프로세서를 이용하여 하였다.

추후 계획 사항으로는 HiXR^{2}의 표현력을 확장하여, 인스터럭션 수준의 병렬성을 나타내고 그에 따라 VLIW와 Super Scalar 머신의 컴파일러/시뮬레이터 자동 생성을 가능하도록 하는 것이다. 또한 ADL로부터 Verilog나 VHDL 등으로 이루어진 RTL 수준의 하드웨어기술로 합성해 내는 일이 필요하다. 컴파일러에서도 이종 레지스터(heterogeneous register)를 가진 아키텍처에서도 성능 평가에 충분한 코드 생성을 할 수 있도록 이종 레지스터를 위한 레지스터 전역 할당 알고리즘 (global register allocation)등이 추가 되어야 한다. 또한 VLIW 아키텍처를 비롯한 SIMD 명령어 지원을 위하여 명령어 수준 병렬성을 찾기 위한 알고리즘도 추가되어야 한다.

Acknowledgements

본 연구는 SystemIC2010 사업의 일환으로 삼성전자(주)에서 지원하는 "저전력 32bit 멀티미디어용 CPU Core 기술개발"의 위탁과제로 수행되었음.

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. 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

[3] 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

[4] 김호영, 김탁곤, "컴파일 된 시뮬레이션 기법을 이용한 ASIP 시뮬레이터의 성능 향상", 한국 시뮬레이션 학회 추계 학술대회 2002

[5] C. Lee, M. Portkonjak, W. H. Mangione-Smith "MediaBench: a tool for evaluating and synthesizing multimedia and communi-

cations systems", in *Proc. IEEE/ACM International Symposium on*, pp. 330-335, 1997

[6] Target Compiler Technology, Chess/Checker : A retargetable too-suite for embedded processors, January, 2002

[7] Tensilica, Xtensa Application Specific Microprocessor Solutions, 2000

[8] C. Fraser and D. Hanson. A Retargetable C Compiler : Design and Implementation. Addison-Wesley Publishing Company, New York, 1995.

[9] A. Appel, J. Davidson, and N. Ramsey. The Zephyr Compiler Infrastructure. Technical Report at http://www.cs.virgnia.edu/zephyr, University of Virginia, 1998

[10] http://cat.kaist.ac.kr, KAIST, 2003