컴파일 시뮬레이션에 의한 빠른 어셈블리 코드 검증환경의 개발

안 재 우　금 기 일　성 원 용
서울대학교 전기공학부

Development of A Fast Assembly Code Verification Environment with New Compiled Simulation Techniques

Jae-Woo Ahn　Ki-Il Kum　Wonyong Sung
School of Electrical Engineering, Seoul National University

요 약

컴파일러 또는 프로그래머에 의해 생성된 어셈블리 코드를 짧은 시간에 검증할 수 있는 컴파일.시뮬레이션 환경을 개발하였다. 기존에 알려진 컴파일 시뮬레이션의 기법외에 점진적 재컴파일 (Incremental Recompilation), C-어셈블리 통합 시뮬레이션 (C-Assembly Cosimulation) 등의 기법을 추가로 고안·구현함으로써, 코드의 검증에 소모되는 시간을 대폭 줄일 수 있었다. 개발된 컴파일 시뮬레이션 환경은 현재 연구중인 VLIW 스케쥴러를 위한 실험 환경의 일부로 구현되고 있으며, SPARC 코드를 변형한 ESPARC 코드 및 ESPARC 코드를 병렬화하여 얻은 VLIW 코드를 검증할 수 있다. SPEC 정수 벤치마크의 일부 및 UNIX 유틸리티의 ESPARC 코드에 대한 시뮬레이션 결과를 보임으로써, 개발된 환경의 효율성을 검증한다.

1. 서 론

프로그래머블 프로세서의 설계과정 중, 가장 상위 수준에 해당하는 것이 명령어 세트 구조 (Instruction Set Architecture, 이하 ISA) 와 컴파일러의 설계이다. ISA 는 하드웨어와 소프트웨어 사이의 인터페이스 역할을 하므로, 효율적인 ISA 와 그 효율성을 충분히 활용하는 컴파일러를 설계하는 것은 전체 시스템의 성능에 결정적인 영향을 미친다. 이러한 상황에서 ISA 및 컴파일러를 위한 효율적인 설계환경을 구축하는 것은 최종 성능 뿐 아니라 빠른 time-to-market 능력을 위해서도 매우 의미있는 일이다.

일반적으로 프로세서의 하드웨어 설계와 컴파일러의 설계는 병렬적으로 이루어지는 경우가 많다. 이것은 전체 개발 시간의 단축을 위한 것이기도 하지만, 컴파일러의 개발 과정에서 하드웨어 구조를 조정할 수 있는 많은 정보를 얻을 수 있기 때문이기도 하다. 따라서, 컴파일러 개발자는 생성된 어셈블리 코드의 정확성 및 성능을 프로세서가 존재하지 않는 상태에서 검증해야만 한다. 이런 경우 어셈블리 코드의 검증은 주로 ISA 시뮬레이터에 의존한다. 기존의 ISA 시뮬레이터는 대부분 인터프리터 방식을 쓰는데, 명령어의 해석 및 명령어의 수행에 따른 파이프라인의 진행 상황등을 수행시간에 모두 표현하기때문에 매우 느리다는 단점을 지닌다. 이에 비해, 컴파일 시뮬레이션 기법은 그러한 작업을 수행시간이 아닌 시뮬레이터 생성시간에 정적으로 표현하므로 시뮬레이션 시간이 상대적으로 짧다. 본 연구에서는 기존의 연구결과를 바탕으로, '점진적 재컴파일' 에 의해 기존 환경에 존재하는 비효율성을 개선할 뿐만 아니라, 'C-어셈블리 통합 시뮬레이션' 에 의해 전체 프로그램중 일부분의 모듈만을 매우 빠르게 시뮬레이션할 수 있는 환경을 개발하였다.

본 논문의 구성은 다음과 같다. 2장에서는 어셈블리 코드 검증을 위한 컴파일 시뮬레이션의 전체적인 구현방법과 문제점에 대해서 알아보고, 3장과 4장에서는 그와 같은 문제점을 해결하기 위해 본 연구에서 개발한 기법들에 대해 소개한다. 5장에서는 새로운 기법들을 적용한 실험의 결과를 제시하고, 6장에서 결론을 맺기로 한다.

2. 기존의 컴파일 시뮬레이션 기법

컴파일러 또는 프로그래머가 생성한 어셈블리 코드는 각 명령어의 수행에 따른 프로세서의 상태 (레지스터 값, 메모리 내용, 플래그 값, ... 등) 변화를 호스트에서 수행 가능한 동등한 코드로 표현함으로써 짧은 시간에 정확성 및 성능을 검증할 수 있다. 이러한 방법을 컴파일 시뮬레이션

기법이라고 하는데 원래는 회로 설계의 검증 방법으로 처음 제안되었으나[1] 최근에는 ISA 시뮬레이션에 도입되어 그 성능이 확인되고 있다 [2, 3, 4].

컴파일 시뮬레이션에 의한 ISA 시뮬레이션의 과정은 대체로 그림 1 과 같으며, 크게 세가지의 단계 (pass-1,2,3) 를 거쳐 시뮬레이션을 수행한다.

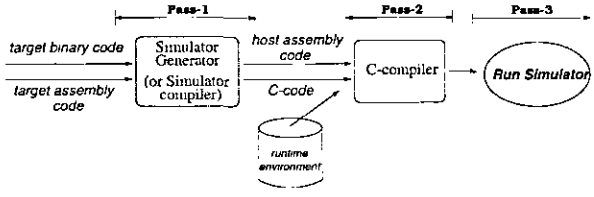


그림 1: 컴파일 시뮬레이션의 진행 과정

그림 1 에 나타나 있듯이, 컴파일 시뮬레이션을 위해서는 시뮬레이션하고자 하는 프로그램의 목적 코드 또는 어셈블리 코드를 해석하는 '시뮬레이터 생성기 (Simulator Generator) [1] ' 가 있어야 한다. 본 연구에서는, 어셈블리 코드를 입력으로 받아서, 그것과 의미적으로 동등하게 수행될 C-코드를 생성하는 시뮬레이터 생성기 'SimGen' 을 제작하였다. 그림 1에서처럼, 시뮬레이터 생성기의 출력으로 시뮬레이션이 수행될 호스트의 코드를 직접 생성하는 방식도 있지만[4], 이식성 및 디버깅의 편의성을 고려하여 C-코드를 생성하는 방식을 택하였다. 생성된 C-코드는 C-컴파일러에 의해 컴파일된 후, 미리 만들어 놓은 수행시간 환경 (주로 메모리 관련 모듈) 과 링크되어 하나의 독립적인 시뮬레이터를 구성한다.

컴파일 시뮬레이션이 빠른 이유는, 인터프리터 방식의 경우 시뮬레이션 시간에 이루어질 많은 일들을 실제 시뮬레이션이 이루어지기 이전 (pass-1,2) 에 수행하기 때문이다. 그러나, 시뮬레이션만을 수행하는 pass-3의 시간이 줄어든다고 하더라도, 그것을 준비하기 위한 pass-1,2 의 시간이 부담이 된다면 문제가 아닐 수 없다. 본 연구의 초기 결과에 의하면, pass-1 의 시간은 무시할 만한 것이지만 pass-2 의 시간은 상당한 것이어서, 어떤 벤치마크에 대해서는 pass-3 보다 pass-2 에서 더 많은 시간을 소모하는 경우도 발생한다. 이러한 문제는 본 연구뿐 아니라 다른 연구에서도 이미 알려진 것으로서 이를 극복하기 위한 연구가 진행되고 있다 [3]. 3장에서 소개할 '점진적 재컴파일' 기법은 이러한 문제점에 대한 해결책을 제시한다.

[1] 이것을 '시뮬레이터 컴파일러' 라고도 한다.

3. 점진적 재컴파일 (Incremental Recompilation)

기존의 컴파일 시뮬레이션 방법은, 입력 코드의 일부분만 수정되더라도 수정되지 않은 부분에 대한 시뮬레이터 코드를 모두 재생성하므로, 그 때마다 전체 시뮬레이터 코드를 재컴파일할 수 밖에 없다. 즉, 입력 코드의 수정된 부분과는 상관없이, 언제나 일정한 시간을 시뮬레이터 생성을 위한 재컴파일에 소모해야 하므로 그림 1 에서 pass-2 의 시간이 줄지 않는 것이다.

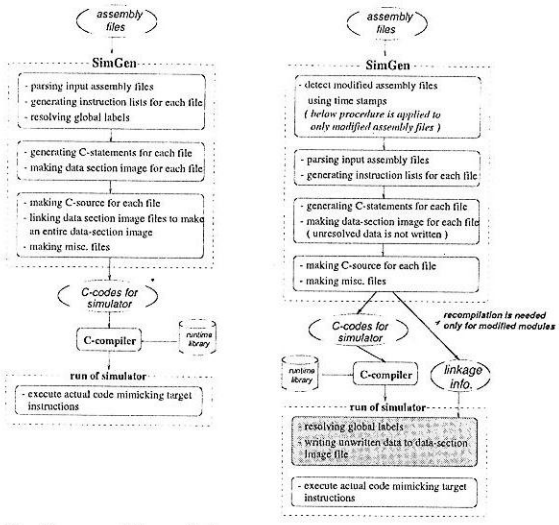
컴파일러의 최적화 과정 및 프로그래머의 노력은 전체 프로그램에서 가장 많이 수행되는 특정 프로시져들에 집중되는 경향이 있으므로, 새롭게 생성된 코드는 전체 프로그램의 일부분인 경우가 많다. 이런 경우, 일부분의 수정된 코드를 검증하기 위해서 전체 모듈에 대한 시뮬레이터 C-코드를 다시 생성하여 재컴파일하여야 하는 것은 매우 비효율적이다. 이것은 프로그램의 각 모듈에 대하여 정의되는 시뮬레이터의 변수값들, 이를테면, 레이블값, 데이터 영역의 내용 등이 모듈간에 상호 의존적이기때문에 발생하는 문제이다.

본 논문에서는 시뮬레이터 생성시점에서 모듈간의 상호영향을 배제함으로써 재컴파일의 부담을 크게 완화시키는 '점진적 재컴파일 (Incremental Recompilation)' 기법을 제안한다. '점진적 재컴파일' 방법은, 수정된 모듈에 대해서만 시뮬레이터의 C-코드를 다시 생성함으로써, 재컴파일에 소모되는 시간을 획기적으로 단축시킨다 (그림 2).

이 방법에 의한 컴파일 시뮬레이션은 수행 시간에 동적으로 라이브러리를 연결하는 동적 연결 (dynamic linking) 과 유사하게 구현된다. 즉, 각 모듈간에 상호 영향을 주고받는 시뮬레이터의 변수값을 시뮬레이터 생성시간에 절대적으로 결정하지 않고, 실제 시뮬레이션의 수행 시간에 각 모듈의 정보를 참조하여 결정하도록 하는 것이다. 그림 2 를 보면, '점진적 재컴파일'에 의한 시뮬레이션의 경우, 시뮬레이션 생성시간에 이루어지던 일들 중 모듈간의 상호 참조를 해야 하는 것들이 시뮬레이션의 수행시간으로 옮겨져 있음을 알 수 있다.

4. C-어셈블리 통합 시뮬레이션 (C-Assembly Cosimulation)

일반적인 ISA 시뮬레이터는 검증하고자 하는 프로그램을 구성하는 모든 모듈에 대하여 시뮬레이션을 수행한다. 즉, 관심이 없는 모듈에 대해서도 속도가 느린 시뮬레이션 루틴을 사용하여 프로그램의 수행을 표현하는 것이다. 이에 반해, 컴파일러 개발자나 프로그래머는 특정 모듈이나 프로시져에 대해 생성된 어셈블리 코드에 대하여 그 코드가 의미적으로 정확한 것인지를 빨리 확인하고 싶어한다. 이런 상황에서, 이미 검증이 끝났거나, 또는 관심이 없는 모듈에 대해서까지 시뮬레이션함으로써 관심의 대상이 되는 모듈 또는 프로시져의 결과만을 빨리 얻을 수가 없는 것은



(a) without Incremental Recompilation (b) with Incremental Recompilation

그림 2: 점진적 재컴파일을 쓰는 경우 기존 방법과의 비교

비효율적이다.

본 논문에서 제안하는 'C-어셈블리 통합 시뮬레이션' 은 지금까지의 컴파일 시뮬레이션 기법을 변용하여 위와 같은 상황에서 매우 빠른 코드 검증의 방법을 제공한다. 'C-어셈블리 통합 시뮬레이션' 은 관심의 대상이 되는 모듈에 대해서만 시뮬레이터 코드를 생성하고, 나머지 모듈은 원래의 C-코드에 대한 목적 코드를 그대로 이용하는 새로운 컴파일 시뮬레이션 방법이다. 검증하고자 하는 프로그램이 src1.c, src2.c, src3.c 의 세가지 모듈로 구성되어 있고, 이 중 src3.c 의 어셈블리 코드에 대해서만 검증을 하고 싶을때, 그림 3 과 같은 과정을 거쳐 C-어셈블리 통합 시뮬레이션을 수행할 수 있다.

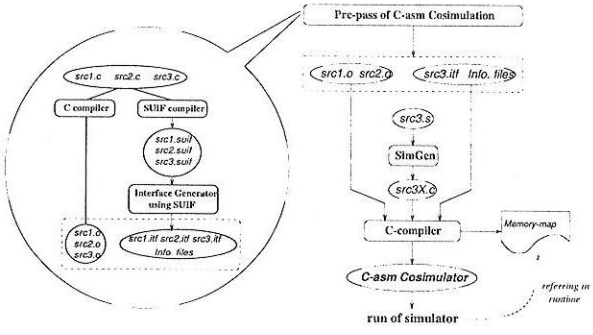


그림 3: C-어셈블리 통합 시뮬레이션 과정

원래의 C-코드로 표현된 모듈과 시뮬레이터의 C-코드로 표현된 모듈 [2] 이 함께 수행되기 위해서는 원래 프로그램 모듈들의 상호 참조를 C-모듈과 시뮬레이터 모듈 사이에서도 동등하게 구현해야 하며, 이것을 위해 다음의 사항을 고려해야 한다.

- C-모듈에서 정의한 전역 변수에 대한 시뮬레이터 모듈에서의 참조
- 시뮬레이터 모듈에서 정의한 전역 변수에 대한 C-모듈에서의 참조
- 시뮬레이터 모듈에서 정의한 전역 함수에 대한 C-모듈에서의 호출
- 함수 호출에 따른 스택의 상호 참조

이러한 경우를 모두 구현하기 위해, C-모듈과 시뮬레이터 모듈이 같은 메모리 공간을 쓰도록 하고 [3], 그 때 생기는 문제들을 아래와 같이 해결하였다.

4.1 전역 변수의 상호 참조

시뮬레이터 모듈에서는 C-모듈에서 정의한 전역 변수를 그 전역 변수에 대한 레이블값을 이용하여 참조한다. 따라서, 시뮬레이터 모듈에서 메모리를 참조할 때 쓰는 레이블 값들이 해당 전역 변수의 주소값과 일치해야 한다. 이것을 위해 시뮬레이터의 각 모듈을 링크한 후 출력된 메모리의 배치상태 (memory map) [4] 를 참조하여 시뮬레이션 수행이전에 전역 변수의 주소값을 미리 알아낸다 (그림 3). 이 값을 시뮬레이션의 수행 전단부에서 해당하는 레이블의 값에 대입해 줌으로써, 시뮬레이터 모듈에서 참조하는 전역 변수에 대한 메모리의 내용이 C-모듈에서의 그것과 일치하도록 할 수 있다.

시뮬레이터 모듈에서 정의하는 전역 변수를 C-모듈에서 참조할 수 있도록 하기 위해서는, 시뮬레이터 모듈에 해당하는 원래의 소스 코드에서와 똑같이 전역 변수를 정의하도록 하였다. 이것은 4.2절에서 설명할 인터페이스 코드를 이용하여 구현하였다.

4.2 전역 함수의 상호 호출

시뮬레이터 모듈에서 정의하는 전역 함수를 C-모듈에서 호출하는 경우, 시뮬레이터 모듈의 C 코드에는 원래의 소스 코드에 있었던 함수가 존재하지 않으므로 문제가 발생한다 [5]. 이 문제는 원래 존재했던 함수를 대신하는 인터페이스 함수를 만듦으로써 해결할 수 있다. C-모듈에서 시뮬레이터 모듈의 함수를 호출하면, 해당 함수에 대한 인터페이스 함수가 수행된다. 인터페이스 함수는 입력 인자를 호출 규약에 따라 프로세서 모델에 전달하고, 프로그램 카운터가 피호출 함수를 가리키도록 한 뒤, 자신이 속한 모

[2] 표현의 편의를 위해, C-코드로 표현된 모듈을 'C-모듈' 이라고 하고, 시뮬레이터의 C-코드로 표현된 모듈을 '시뮬레이터 모듈' 이라고 부르기로 한다.

[3] C-어셈블리 통합 시뮬레이션이 아닌 경우, 시뮬레이터에서 표현하는 모든 메모리 주소값들은 목적 프로세서에 대한 가상 메모리 값이며, 메모리를 참조할 때마다 이 값들을 호스트의 주소값으로 변환한다.

[4] 링커의 옵션으로 출력할 수 있다.

[5] 시뮬레이터 모듈에는 그 함수를 구성하는 명령어들이 C-코드 형태로 나열되어 있을 뿐이다.

들에 대한 '모듈 함수[6] '를 호출한다.

인터페이스 함수 및 4.1 에서 설명한 전역 변수의 정의가 포함되어 있는 코드를 '인터페이스 코드'라고 부른다. 인터페이스 코드는 원래의 소스 코드에서 함수의 내용을 적절히 변환시킨 것으로서, SUIF (Stanford Univ. Intermediate Format) 컴파일러 시스템의 소스 변환 기능을 이용하여 구현하였다. 인터페이스 코드 생성은 검증하고자 하는 프로그램에 대해 C-어셈블리 통합 시뮬레이션의 pre-pass 에서 한 번만 수행하면 충분하다 (그림 3).

4.3 호출자 스택 (Caller Stack) 의 참조

SPARC ISA 의 호출규약에 의하면, 입력 인자의 개수가 6개를 초과하거나, structure 와 같은 집합 데이터 (aggregate data) 를 리턴하는 함수를 호출할 때, 피호출 함수에서 호출자의 스택을 참조하도록 되어 있다[5]. 이런 상황에서, 호출자 (caller) 가 C-모듈에 존재하고, 피호출자 (callee) 가 시뮬레이터 모듈에 존재할 때 문제가 발생한다. 시뮬레이터 모듈의 함수는 자신의 스택을 동적 메모리에 명시적으로 할당하므로 시뮬레이터의 코드에서 직접 접근이 가능하지만, 호출자의 스택은 그렇지 못하기 때문이다. 이러한 문제는 목적 ISA 에 매우 의존적인 것으로서, 본 연구에서는 SPARC 어셈블리 코드를 시뮬레이터의 C-코드에 삽입함으로써 해결하였다.

5. 실험 결과

'점진적 재컴파일' 과 'C-어셈블리 통합 시뮬레이션' 은 기존의 컴파일 시뮬레이션 환경에 추가적으로 구현되었다. SPEC 정수 벤치마크와 몇 가지의 UNIX 유틸리티를 이용하여 시뮬레이션함으로써, 추가된 기법들의 효용성을 검증하였으며 본 장에서 그 결과의 일부를 제시한다.

다음의 표 1 에서는 각 벤치마크에 대하여 수행이 가장 많이 되는 모듈만 수정되었다고 가정했을때, '점진적 재컴파일' 기법을 쓰는 경우와 그렇지 않은 경우의 pass-2 시간을 비교하였고, 표 2 에서는 수행이 가장 많이 되는 모듈만을 시뮬레이터 모듈로 만들어 C-어셈블리 통합 시뮬레이션을 수행했을때의 시뮬레이션 시간 (pass-3) 을 전체 모듈을 모두 시뮬레이션한 결과와 비교하였다[7].

6. 결론 및 향후 연구 계획

본 연구에서 구현한 새로운 컴파일 시뮬레이션 기법은 기존의 방법으로는 이를 수 없었던 빠른 어셈블리 코드 검증의 경로를 제공한다. 향후 보다 유연한 시뮬레이션 환경의 구현을 위해 프로세서의 기술 (machine description) 과 임의의 ISA 에 대한 컴파일 시뮬레이션 환경의 구축에 관한 연구가 이루어져야 할 것이다.

[6] 현재의 구현은 하나의 시뮬레이터 모듈에 대해 하나의 C-함수를 정의하는데, 이것을 모듈 함수라고 부른다.

[7] 실험은 128MB 의 주메모리를 갖는 143MHz UltraSPARC-I 에서 수행되었으며, 표에 제시한 시간값들은 사용자 시간 (*user time*) 을 의미한다.

표 1: 점진적 재컴파일에 의한 Pass-2 시간의 단축

Pass-2A: 점진적 재컴파일을 쓰지 않는 경우
Pass-2B: 점진적 재컴파일을 쓰는 경우
(*Pass-2B* 는 가장 수행이 많이 되는 모듈만 수정되었다고 가정했을때의 결과이다.)

Benchmark	Pass-2A (sec)	Pass-2B (sec)
li	461.6	20.1
eqntott	217.1	8.2
espresso	578.0	38.1
yacc	769.2	54.0

표 2: C-어셈블리 통합 시뮬레이션에 의한 Pass-3 시간의 단축

Pass-3A : C-어셈블리 통합 시뮬레이션을 쓰지 않는 경우
Pass-3B : C-어셈블리 통합 시뮬레이션을 쓰는 경우
(*Pass-3B* 는 가장 수행이 많이 되는 모듈만을 시뮬레이터 모듈로 만들었을때의 결과이다.)

Benchmark	Input	Pass-3A (sec)	Pass-3B (sec)
li	9 queens	2852.2	416.6
eqntott	int_pri_3.eqn	439.6	325.1
espresso	bca.in	212.8	62.9
yacc	C grammar	2.74	0.35

참고 문헌

[1] Zeev Barzilai, J. Lawrence Carter, Barry K. Rosen, and Joseph D. Rutledge, "HSS - A High-Speed Simulator," *IEEE Transactions on Computer Aided Design*, vol. CAD-6, no. 4, pp. 601-617, July 1987.

[2] S. Park, S. Shim, and S. Moon, "Evaluation of Scheduling Techniques on a SPARC-Based VLIW Testbed," to appear in *Proceedings of the 30th Annual International Symposium on Microarchitecture (Micro-30)*, Dec. 1997.

[3] Vojin Zivojnovic, Steven Tjiang, and Heinrich Meyr, "Compiled Simulation of Programmable DSP Architecture," in *1995 IEEE Workshop on VLSI Signal Processing*, 1995, pp. 187-196.

[4] J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, B. Hall, R. Miranda, S. K. Chen, and A. Polyak, "Architecture, Compiler and Simulation of a Tree-based VLIW Processor," Research Report RC 20495, IBM Research Division, T.J. Watson Research Center, July 1996.

[5] SPARC International Inc., *The SPARC Architecture Manual – Version 8*, Prentice Hall, 1992.