논문 22-2-02

IR 시뮬레이터
(IR Simulator)

최영규 · 안민욱 · 윤종희 · 김용주 · 김대호 · 백윤흥
서울대학교 전기컴퓨터 공학부
(ykchoi · mwahn · jhyoun · yjkim · dhkim@optimizer.snu.ac.kr, ypaek@snu.ac.kr)

요 약

컴파일러는 상위 언어로 작성된 프로그램을 프로세서를 위한 하위 언어로 번역하는 역할을 한다. 이때 컴파일러는 상위언어로 작성된 프로그램을 동일한 의미(semantic)를 가진 Intermediate Representation (IR)으로 바꾼 뒤 코드 생성 작업을 하게 된다. 컴파일러가 코드 생성 작업을 잘하기 위해서는 보다 많은 정보를 IR에서 가지고 있어야 한다. 예를 들어서 프로그램 상의 hot spot을 찾아내기 위해서는 Control Flow Analysis를 해서 얻은 basic block들의 실행 빈도를 알아야한다. 그러나 이러한 정보는 바이너리 코드를 얻어서 실행해보지 않고서는 알 수 없는 정보이다.

이 논문에서는 이러한 불편을 줄이고자 컴파일러의 IR을 직접 시뮬레이션해 볼 수 있는 IR 시뮬레이터에 대해서 논하고자 한다. 시뮬레이션을 통해서 얻은 정보는 프로그램을 최적화하는데 유용하게 사용되며 나아가 프로그램을 가속화하기 위한 새로운 명령어를 설계하는 등의 일에 유용하게 사용될 수 있다.

1. 관련 연구

1.1. 연구 배경

IR은 컴파일러가 상위 언어로 작성된 프로그램을 프로세서를 위한 하위 언어로 바꿀 때 그 중간 단계에서 프로그램을 표현하기 위한 수단이다. 컴파일러는 IR로 표현된 프로그램을 여러 가지 기법을 동원하여 분석하고 분석된 결과를 이용하여 다양한 최적화를 시도한다. 따라서 많은 IR은 간단하고 구조가 고치기 쉽게 디자인되고 확장성이 있게끔 되어있다. Application Specific Instruction set Processor(ASIP)에서 많이 사용되는 재겨냥성 컴파일러를 위해서 IR을 디자인 할 때에는 이러한 기본적인 기능이외에 좀 더 다른 것들을 고려하여야 한다.

우선 여러 가지 언어를 컴파일할 수 있는 frontend를 사용할 수 있도록 디자인되거나 좀 더 성능이 좋은 frontend 컴파일러를 가져다 사용하고 싶은 경우 IR은 자신의 correctness를 검증할 수 있는 수단을 가지고 있어야 한다. Open source 진영에서 많이 사용되는 GCC가 이와 같은 구조를 가지고 있는데 GCC가 C 언어 외에 C++이나 JAVA, ADA와 같은 여러 언어를 지원할 수 있도록 하기 위해서이다.[1)]

1) GCC 4.x에 와서 단일한 IR(GENERIC)을 사용하는 구조로 변경되었고 그 이전에는 언어에 따라서 여러 가지 IR이 존재하였다.

두 번째로 재겨냥성 컴파일러를 위한 IR은 프로세서 디자인에 도움이 될 수 있는 정보를 많이 줄 수 있어야 한다. 예를 들어서 프로그램의 hot spot을 알 수 있는 정보나 새로운 instruction을 디자인하는데 도움이 되는 instruction pattern정보와 같은 정보는 응용 프로그램의 성능을 높일 수 있는 새로운 instruction을 제안하는데 효율적으로 사용될 수 있다.

이러한 두 가지 조건을 만족하는 IR을 디자인하기 위해서 이 논문에서는 가상 머신(Virtual Machine)에 기반한 IR과 이를 직접 시뮬레이션해볼 수 있는 IR 시뮬레이터에 대해서 논하고자 한다.

논문의 구성은 다음과 같다. 2장에서는 관련된 연구에 대한 검토를 한다. 3장에서는 IR이 기반한 가상머신을 설명하고 4장에서는 IR을 시뮬레이션이 가능한 코드로 바꾸는 작업에 대해서 설명한다. 5장에서는 IR 시뮬레이터를 통해서 가능한 작업과 얻어지는 정보를 보여주고 6장에서 결론을 내도록 하겠다.

2. 관련 연구

본 논문의 연구와 가장 비슷한 연구로 자바 언어를 위한 가상머신이 있다. 자바 컴파일러는 자바 프로그램을 platform independent IR(intermediate representation)인 bytecode로 변환시킨다. 자바 가상 머신(JVM)은 이 bytecode를 실행시켜서 타겟에서 실행가능한 machine code를 생성하게 된다.

또한 Microsoft의 .NET Framework을 사용하는 code 개발 환경도 본 논문의 연구와 비슷하다. .NET을 통해 개발된 프로그램은 Microsoft Intermediate Language(MSIL) 혹은 Common Intermediate Language(CIL)의 형식으로 컴파일이 된다. 그리고 나중에 타겟이 되는 platform에서 실행이 될 때 다시 한 번 컴파일되어 machine language로 변환된다.

위의 두 가지 연구는 주로 IR을 통한 portability에 초점을 맞추었다면 본 연구는 IR을 통해 machine language 생성 시 성능 향상에 도움이 되는 정보를 얻고자 한 다는 점에서 차이가 있다.

3. 가상 머신(Virtual Machine, VM) IR (Intermediate Representation)

3.1. 가상 머신(VM) IR

가상 머신의 Instruction Set Architecture (ISA)는 특정 Architecture 편향된 부분을 배제한 Architecture independent minimal set으로 구성하였다. 가장 기본적으로 move, load, store operation이 있고, arithmatic, logical operation에 대해서도 정의되어 있다. 그리고 control flow operation의 경우 conditional branch, unconditional branch 그리고 call operation으로 이루어져 있다. data mode의 경우 fixed point data type의 경우에 SI mode를 1 word(32bit)으로 정하고, QI, HI, DI mode까지 지원하도록 구성하였다.

Mode	Meaning
QI	Quarter integer (word ¼)
HI	Half integer (word ½)
SI	Single integer (word)
DI	Double integer (word 2)

[그림 1] Data mode

Category	Operation
Control transfer	BEQ, BLE, BGE, BLT, BGT, JUMP
Function call	CALL, RETURN

Category	Binary OP(opcode src1 src2)
Arithmetic	ADD, SUB, MULT, DIV , ASHIFT
Logical	AND, IOR, XOR, LSHIFT

Category	Unary OP(opcode sr1)
Arithmetic	NOT, EXTEND
Logical	NEG

[그림 2] Operation category

가상 머신의 Instruction Set Format은 3-address code로 (set : DEST SRC)의 형태를 가진다. DEST와 SRC로는 각각의 instruction의 제약 조건에 따라서 레지스터, 메모리 주소, 산술연산의 결과 등이 들어가게 된다.

Inst.	Meaning	Virtual Machine IR
MOV	R[0] <- 1	(set (SI:R[0]) (const: SI 1))
LOAD	R[0] <- MEM[R[3] + 0]	(set (SI:R[0]) (SI:0(R[3])))
STORE	MEM[R[4] + 0] <- R[0]	(set (SI:0(R[4])) (SI:R[0]))
ADD	R[3] <- R[3] + R[0]	(set (SI:R[3]) (ss_plus:SI (SI: R[3]) (SI: R[0])))

[그림 3] Basic instructions

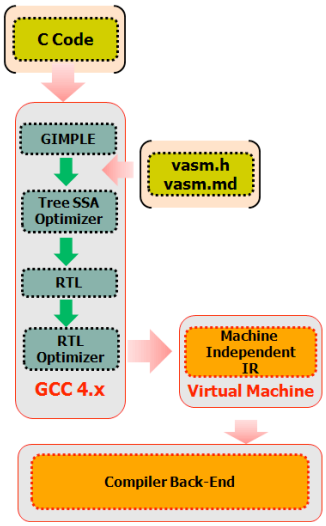
Inst.	Meaning	Virtual Machine IR
CALL	Function pin_down call	(call (function:pin_down))
JUMP	Unconditional branch to L2	(jump (label:L2))
BLE	Branch to L3 if (R[4] <= 16)	(jump (label:L3) (le:SI (SI:R[4]) (const: SI 16)))

[그림 4] Control flow instructions

3.2 가상 머신 코드 생성을 위한 GCC

GCC는 다양한 Cross Platform에서 동작할 수 있도록 macro expansion이라는 code generation 방법을 사용하고 있다. 이 때 GCC에게 제공해야 할 Target 정보는 두 가지가 있다. 하나는 register class, stack layout, calling convention등의 하드웨어의 구조에 관한 정보를 담은 Target description macro이고, 다른 하나는 명령어들의 종류와 동작에 관한 정보를 기술한 Machine description이다.

GCC에게 제공될 Target이 되는 가상 머신은 아래의 특징을 가진다. 32비트 머신으로 충분히 많은 general register를 할당하여 spill code가 생성되지 않도록 한다. Byte addressable memory를 가지며 addressing mode는 Base Offset addressing mode와 Absolute addressing mode를 지원한다. stack frame 관리를 위해 stack pointer register와 frame pointer register를 둔다. GCC Code generation의 Target을 가상 머신(Virtual Machine)으로 설정하고 porting



[그림 5] Virtual Machine IR

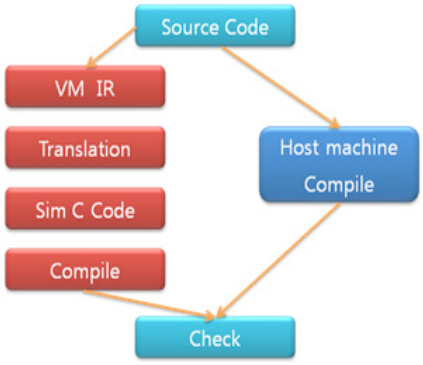
을 하여 Cross Compiler를 만들어 이를 이용해 가상 머신 IR(Intermediate Representation)을 생성한다. Compiler backend는 IR로 표현된 프로그램을 여러 가지 기법을 동원하여 분석하고 분석된 결과를 이용하여 다양한 최적화를 시도하여 최종 target machine code를 생성하게 된다.

4. 가상 머신(VM) IR Simulator

4.1. 기본 아이디어

가상 머신 IR code를 동일한 semantic을 가진 C code로 변환 시키면 host machine 상에서 컴파일 및 실행하여 simulation할 수 있다. 다시 말해, instrumentation 기법을 사용하여 VM IR의 instruction을 function으로, memory와 register는 global array로 치환하여 C code를 생성한다. 이와 같이 simulator 내부에 simulation을 수행하는 code와 semantic code를 함께 포함하고 있는 compiled simulation을 구현하였다.

이런 simulation을 통해서 VM IR의 정확성에 대한 검증이 가능하고 instruction count, Basic block(BB) frequency 등의 profiling data를 얻을 수 있다. 이런 profiling 정보는 응용 프로그램의 성능을 높일 수 있는 새로운 instruction을 제안하는데 큰 역할을 할 수 있을 것이다.



[그림 6] VM simulator

4.2. 가상 머신(VM) IR Simulator 구현

4.2.1. 레지스터 및 메모리 구현

가상 머신의 레지스터는 stack pointer, frame pointer, program counter, general register와 return value save register로 구성되어 있고, 이들을 C에서 integer global variable로 치환한다. 특히 general register의 경우 충분히 큰 size의 array를 잡아서 본래 무한개의 register를 갖는다는 가정에 부합하도록 구현하였다.

가상 머신의 메모리는 byte accessible하다는 가정에 부합하도록 char global array로 구현하였다. 또한 그 메모리 영역을 global data area, stack area, run-time heap 구분하여 가상 머신의 메모리 이미지를 구현하였다.

4.2.2. Instruction 구현

Instruction은 기본적으로 simulator code 내에서 function call로 치환된다. MOVE, LOAD와 STORE instruction의 경우는 기본적으로 memcpy()을 사용하여 구현하였다.

Arithmatic, Logical instruction 등은 [그림 8]에서와 같이 C 언어에서 제공되는 연산자들을 사용하여 구현하였다.

Branch instruction과 Call instruction은 각각이 target label과 target function pointer를 argument로 받는 함수를 만들어 처리하도록 하였다.

Library call의 경우는 semi-hosting 기법을 사용하여 처리하였다. 즉, library를 직접 구현하지 않고 host machine의 library에 연

결되도록 하여 결과를 확인할 수 있도록 하였다. 하지만, 이 경우 stack pointer, frame pointer의 정보를 관리하는 function prologue, epilogue 부분이 없기 때문에 semi-hosting 기법을 사용하기 전 후에 이 부분을 처리하도록 붙여주어 정상적인 처리가 가능하도록 하였다.

```
void VASM_MOVE_REG(void *dst, void *src, int size) {
    memcpy(dst, src, size);
}

void VASM_MOVE(void *dst, int src, int size, const ch
    memcpy(dst, &src, size);
}

void VASM_LOAD(int *dst, int src, int size) {
    memcpy(dst, &VASM_M[src], size);
}

void VASM_STORE(int dst, int *src, int size) {
    memcpy(&VASM_M[dst], src, size);
}
```

[그림 7] Move, load and store operaton

```
int VASM_OPER(enum VASM_OPER_TYPE oper, int src0, in
    int result;
    if(oper == VASM_SS_PLUS) {
        result = src0 + src1;
    } else if(oper == VASM_MULT) {
        result = src0 * src1;
    } else if(oper == VASM_AND) {
        result = src0 & src1;
    } else if(oper == VASM_IOR) {
        result = src0 | src1;
    } else if(oper == VASM_XOR) {
        result = src0 ^ src1;
    } else if(oper == VASM_SS_MINUS) {
        result = src0 - src1;
    } else if(oper == VASM_DIV) {
        result = src0 / src1;
    } else if(oper == VASM_MOD) {
        result = src0 % src1;
    } else if(oper == VASM_ASHIFT) {
        result = src0 << src1;
```

[그림 8] Aithmatic and logical operation

5. 실험 결과

5.1. Simulator code의 생성

Compiled simulation을 수행하기 위해서 아래의 [그림 10]과 같이 원래의 source code로부터 IR을 얻어 IR과 동일한 semantic의 c code를 생성한다. 이 외에도 simulation을 위한 code들이 생성되어 simulator가 동작하게 된다. 이렇게 생성된 simulator c code를 host machine 상에서 compile하여 수행을 하게 되면 IR의 correctness를 빠르고 정확하게 검사할 수 있게 된다.

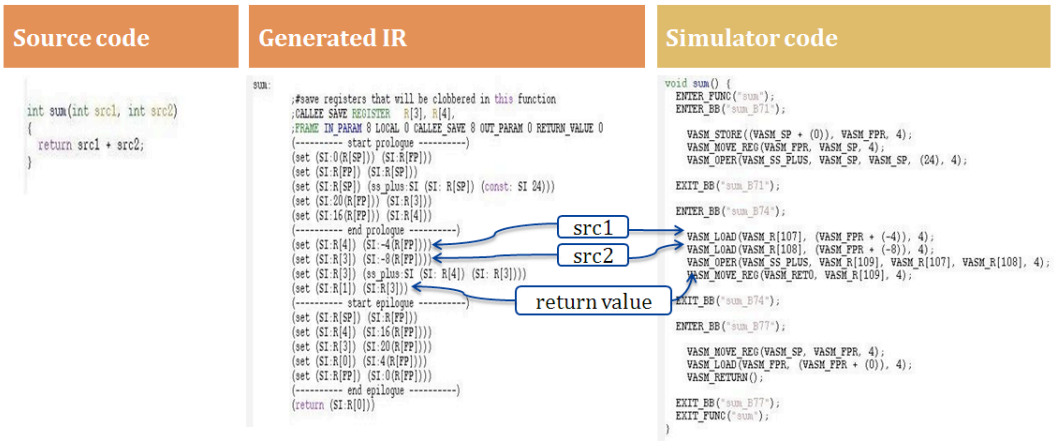
5.2. Profiling information

5.2.1. Instruction Count

Compiled simulation을 수행한 후 결과로서 total instruction count를 얻을 수 있다. GCC의 최적화 기술을 그대로 적용시킬 수 있으므로 최적화 단계에 따른 instruction count를 구하여 비교해 볼 수 있다. 다음의 [그림 11]은 fir, dijkstra_small, adpcm_decode benchmark에 대한 최적화 단계에 따른 total instruction count를 정리한 것이다.

Inst.	VM IR	Simulator C function
MOV	(set (SI:R[3]) (const: SI 1))	VASM_MOVE(VASM_R[3], (1), 4)
LOAD	(set (SI:R[0]) (SI:0(R[3])))	VASM_LOAD(VASM_R[0], (VASM_R[3] + (0)), 4)
STORE	(set (SI:0(R[4])) (SI:R[0]))	VASM_STORE((VASM_R[4] + (0)), VASM_R[0], 4)
ADD	(set (SI:R[3]) (ss_plus:SI (SI: R[3]) (SI: R[0])))	VASM_OPER(VASM_SS_PLUS, VASM_R[3], VASM_R[3], VASM_R[0], 4)
BLE	(jump (label:L3) (le:SI (SI:R[4])(const: SI 16)))	VASM_BRANCH(VASM_LE, VASM_R[4], (16), L8, 4);

[그림 9] Instruction to Function



[그림 10] Correctness check

Benchmark \ Opt. lv.	00	01	02
fir	1,219	523	520
dijkstra_small	4,560,213	2,209,002	2,192,689
adpcm_decode	11,967,070	6,038,754	5,676,305

[그림 11] Instruction count

5.2.2. Basic Block frequency

Simulation을 통해서 특정 basic block을 몇 번 수행하는가에 대한 정보를 수집할 수 있다. 이 profiling data를 통해서 프로그램의 hot spot을 알 수 있으며 이는 바로 프로세서의 성능을 높일 수 있는 정보로 활용될 수 있을 것이다.

```
BB Freq. : adpcm_decoder_B529 148
BB Freq. : adpcm_decoder_B532 148
BB Freq. : adpcm_decoder_B535 148
BB Freq. : adpcm_decoder_B538 73760
BB Freq. : adpcm_decoder_B541 140834
BB Freq. : adpcm_decoder_B547 147520
BB Freq. : adpcm_decoder_B550 28574
BB Freq. : adpcm_decoder_B553 147520
BB Freq. : adpcm_decoder_B556 58951
BB Freq. : adpcm_decoder_B559 147520
BB Freq. : adpcm_decoder_B562 65211
BB Freq. : adpcm_decoder_B565 147520
BB Freq. : adpcm_decoder_B568 70437
BB Freq. : adpcm_decoder_B571 147520
BB Freq. : adpcm_decoder_B577 147520
BB Freq. : adpcm_decoder_B580 147372
BB Freq. : adpcm_decoder_B583 73760
BB Freq. : adpcm_decoder_B586 6686
BB Freq. : adpcm_decoder_B589 147520
BB Freq. : adpcm_decoder_B595 148
BB Freq. : adpcm_decoder_B598 148
BB Freq. : adpcm_decoder_B601 77083
```

[그림 12] Basic block frequency

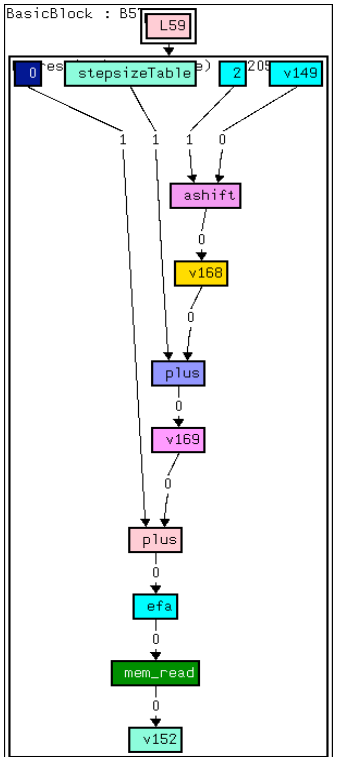
5.2.3. Pattern information

Basic block(BB) 내의 Instruction patten 정보를 수집할 수 있다. 이 정보와 Basic block frequency를 통해서 어떤 Instruction이 가장 많이 수행되는지 알아낼 수 있다. 실제로 가장 많은 수행회수를 보이는 BB 중의 하나인 adpcm_decoder_B577의 내부 Pattern information은 [그림 13]과 같다. 위와 같은 pattern이 주어졌을 때 응용 프로그램의 성능 향상을 위해서 도입할 수 있는 새로운 instruction에 대한 idea를 쉽사리 떠올릴 수 있을 것이다.

5.2.4. Semi-hosting 기법

Semi-hosting 기법을 통해서 host machine에서 지원하는 모든 library call을 사용할 수 있기 때문에 결과를 쉽게 출력해서 확인할 수 있을 뿐만 아니라 결과의 정확성에 대한 테스트도 쉽게 할 수 있다. [그림 14]는

dijkstra_small의 source code 중 일부이다. printf와 같이 host machine 상의 standard library에 속해 있는 function의 경우 semi-hosting 기법을 통해서 그대로 사용할 수 있다. 아래의 [그림 15]는 이 기법을 사용하여 생성된 simulator code이다.



[그림 13] Pattern information

```
printf("Shortest path is %d in cost. ", rgnNodes[chEnd].iDist);
printf("Path is: ");
print_path(rgnNodes, chEnd);
printf("      \n");
```

[그림 14] Source codes

```
    VASM_LOAD(VASM_R[288], (VASM_R[287] + (0)), 4);
    VASM_MOVE(VASM_R[289], (void *)dijkstra_small_c_LC2, 4, "dijkstra_small_c_LC2");
    VASM_MOVE_REG(VASM_ARG[0], VASM_R[289], 4);
    VASM_MOVE_REG(VASM_ARG[1], VASM_R[288], 4);
EXIT_BB("dijkstra_B707");
ENTER_BB("dijkstra_B710");
    vasm_call_wrapper("printf", printf, 2, 4, 4);
EXIT_BB("dijkstra_B710");
ENTER_BB("dijkstra_B713");
    VASM_MOVE(VASM_R[290], (void *)dijkstra_small_c_LC3, 4, "dijkstra_small_c_LC3");
    VASM_MOVE_REG(VASM_ARG[0], VASM_R[290], 4);
EXIT_BB("dijkstra_B713");
ENTER_BB("dijkstra_B716");
    vasm_call_wrapper("printf", printf, 1, 4);
EXIT_BB("dijkstra_B716");
```

[그림 15] Simulator code

```
Shortest path is 10 in cost. Path is:  0 16 5 9 1 11 23 15
Shortest path is 3 in cost. Path is:  1 11 23 27 16
Shortest path is 17 in cost. Path is:  2 5 8 0 17
Shortest path is 11 in cost. Path is:  3 16 5 9 1 11 23 27 10 21 18
Shortest path is 17 in cost. Path is:  4 26 19
Shortest path is 15 in cost. Path is:  5 9 1 11 23 27 10 14 20
Shortest path is 23 in cost. Path is:  6 9 1 11 23 27 10 21
Shortest path is 10 in cost. Path is:  7 26 19 15 22
Shortest path is 11 in cost. Path is:  8 0 16 5 9 1 11 23
Shortest path is 11 in cost. Path is:  9 1 11 23 27 16 5 8 24
Shortest path is 10 in cost. Path is:  10 21 27 16 5 9 25
Shortest path is 13 in cost. Path is:  11 23 27 16 5 9 1 7 26
Shortest path is 7 in cost. Path is:  12 16 5 9 1 11 23 27
Shortest path is 14 in cost. Path is:  13 11 23 27 10 28
Shortest path is 9 in cost. Path is:  14 12 16 5 29
Shortest path is 4 in cost. Path is:  15 28 0
Shortest path is 1 in cost. Path is:  16 5 9 1
Shortest path is 8 in cost. Path is:  17 19 2
Shortest path is 8 in cost. Path is:  18 3
Shortest path is 7 in cost. Path is:  19 5 4
```

[그림 16] Simulation result

simulator code를 컴파일하여 실행시키면 위와 같은 결과를 얻을 수 있는데 이것은 source code를 컴파일하여 실행시킨 출력과 동일하다. 이와 같이 결과를 바로 출력해서 확인할 수 있기 때문에 IR code의 correctness 검사가 자동적으로 이뤄지고, 복잡한 code의 디버깅 시간도 단축되는 효과가 있다.

6. 결 론

Application Specific Instruction set Processor(ASIP)에서 많이 사용되는 재겨냥성 컴파일러를 위해서 IR을 디자인 할 때에는 그 특성 때문에 여러 가지 고려해야 할 이슈들이 많이 있다. 그 중에서도 가장 중요한 것은 프로그램의 hot spot이나 instruction patter을 알 수 있는 정보들로서 이들을 통해서 프로그램의 성능을 향상 시킬 수 있는 새로운 instruction을 효과적으로 디자인할 수 있다. 하지만 이러한 정보들은 바이너리 코드를 얻어서 직접 실행해 보기 전까지는 얻을 수 없다. 두 번째 중요한 이슈는 여러 frontend를 가져다 사용하고 싶은 경우 IR의 correctness를 검증할 수 있는 수단이 있는가 여부이다.

이러한 조건을 만족하도록 본 논문에서는 가상 머신(Virtual Machine)을 기반으로 한 IR과 이를 직접 시뮬레이션할 수 있는 IR

Simulator를 제안하였다. 이를 위해 compiled simulation 방법을 사용하여 Simulator를 구현하여 host machine 상에서 컴파일하여 빠른 속도로 그 결과를 확인해 볼 수 있다. 먼저 Library를 semi-hosting 기법을 사용하여 host machine상의 library call로 연결되도록 하여 Simulation 결과를 source code와 완전히 동일하게 출력할 수 있어 IR code의 correctness를 결과 파일의 비교를 통해 빠르고 정확하게 확인 가능하였다.

또한 basic block(BB) frequency 정보를 얻음으로써 IR 상에서 어느 곳이 hot spot인지를 알아낼 수 있었다. pattern information은 어떤 pattern의 instruction들이 자주 나타나 수행되는지에 대해서 보여준다. 위의 BB frequency와 pattern information은 응용 프로그램의 성능을 높일 수 있는 새로운 instruction을 제안하는데 효율적으로 사용될 수 있을 것이다.

Acknowledgement

본 연구는 교육과학기술부/한국과학재단 우수연구센터육성사업(R11-2008-007-01001-0), 지식경제부 출연금으로 ETRI, SoC 산업진흥센터에서 수행한 ITSoC 핵심설계인력양성사업, 서울시 산학연 협력사업, 2008 년도 정보(교육과학기술부)의 재원으로 한국과학재단의 국가지정연구실사업(R0A-2008-20110-0), 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업(IITA-2008-C1090-0801-0020), 지식경제부 및 정보통신연구진흥원의 IT원천기술개발사업[과제관리번호:2006-S-006-02, 과제명: 유비쿼터스 단발용 부품/모듈]의 지원을 받아 수행 되었습니다.

참 고 문 헌

[1] James E. Smith, Ravi Nair, Virtual Machines Versatile Platforms for Systems and Processes, Morgan Kaufmann Publishers, ISBN 1-55860-910-5

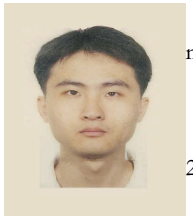
[2] 황승호, 이종열, “컴파일러의 개발”, IDEC

[3] Oliver Wahlen, Rainer Leupers, application specific compiler/architecture codesign:a case study, LCTES/SCOPES 02

[4] JongYeol Lee, In Cheol Park, Instruction set generation considering the IR of compilers, KAIST

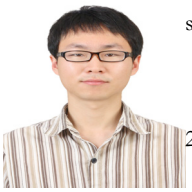
[5] Richard M. Stallman, GNU GCC compiler collection internals, www.gcc.gnu.org

최 영 규
hkkim@optimizer.snu.ac.kr
2002년 서울대학교
전기공학부(학사)
2008년~현재 서울대학교
전기컴퓨터공학부 석사과정
관심분야: 임베디드 소프트웨어, 컴파일러.



안 민 욱
mwahn@optimizer.snu.ac.kr
2003년 서울대학교
전기공학부(학사)
2003년~현재 서울대학교
전기컴퓨터공학부
석박사통합과정
관심분야: 임베디드 소프트웨어, 컴파일러.

윤 종 희
jhyoun@optimizer.snu.ac.kr
2003년 경북대학교
전기공학부(학사)
2003년~현재 서울대학교
전기컴퓨터공학부
석박사통합과정
관심분야: 임베디드 소프트웨어, 컴파일러.



김 용 주
shpark@optimizer.snu.ac.kr
2006년 서울대학교
전기공학부(학사)
2006년~현재 서울대학교
전기컴퓨터공학부
석박사통합과정
관심분야: 임베디드 소프트웨어, 임베디드 시스템
개발도구, 저전력 설계.

김 대 호
dhkim@optimizer.snu.ac.kr
2007년 서울대학교
전기공학부(학사)
2007년~현재 서울대학교
전기컴퓨터공학부 석사과정
관심분야: 임베디드 소프트웨어, 컴파일러.

백 윤 흥
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~2007년 서울대학교 전기컴퓨터공학부 부교수
2007년~현재 서울대학교 전기컴퓨터공학부 교수
관심분야: 임베디드 소프트웨어, 임베디드 시스템
개발도구, 컴파일러, MPSoC