임베디드 운영체제의 스케줄링 프리미티브를 고려한 정적 최악실행시간 분석도구

(Static Worst-Case Execution Time Analysis Tool for Scheduling Primitives about Embedded OS)

박 현 희 [†] 양 승 민 [††] 최 용 훈 [†††]
(Hyeon Hui Park) (Seung Min Yang) (Yong Hoon Choi)

요 약 임베디드 운영체제에서의 실시간성 지원은 현대 임베디드 시스템에서 추가사항이 아니라 필수 사항이다. 이러한 임베디드 운영체제가 사용되는 시스템의 실시간성 지원을 충족하기 위해서는 시스템 내 실시간성이 요구되는 태스크들의 스케줄링 가능성 여부가 중요하며, 이를 시스템 수행 전 검증해야 할 필요가 있다. 스케줄링 가능성 분석에서 핵심적인 부분 중의 하나는 태스크의 최악실행시간을 구하는 것이다. 기존의 최악실행시간 분석도구들은 일반적인 응용 태스크 즉, 응용 프로그램의 최악실행시간을 위주로 분석하였기 때문에 응용 프로그램들이 운영체제에 의해 스케줄링시 영향을 받는 운영체제의 스케줄링 관련 프리미티브들(스케줄러, 인터럽트 서비스 루틴등)에 대한 고려는 전혀 하지 않고 있다.

본 논문에서는 임베디드 운영체제 중에 널리 사용하고 있는 임베디드 리눅스가 사용되는 임베디드 시스템에서의 스케줄링 관련 프리미티브들을 고려하는 최악실행시간 분석 도구를 설계하고 구현한다. 이 분석도구는 일반적인 응용 프로그램 뿐만 아니라 임베디드 리눅스 커널내의 스케줄링에 영향을 미치는 관련 프리미티브들의 최악실행시간을 분석하여 스케줄링 분석의 정확성을 더욱 더 높인다. 이 도구는 현재 임베디드 환경에서 통합개발환경으로 제작된 이클립스(Eclipse)의 플러그인 형태로 개발되어 어떠한 플랫폼에서도 동작 가능하고 사용자가 사용하기에 편리한 인터페이스 및 기능을 제공할 수 있도록 구현한다.

키워드 : 임베디드 시스템, 최악실행시간, 최악실행시간 분석, 스케줄링 분석

Abstract Real-time support of embedded OS is not optional, but essential in contemporary embedded systems. In order to achieve these system's real-time property, it is crucial that schedulability analysis for tasks having its property have been accomplished before system execution. Acquiring *Worst-Case Execution Time(WCET)* of task is a core part of schedulability analysis. Because traditional WCET tools analyze only its estimation of application task(i.e. program), it is not considered that application tasks are affected by scheduling primitives(e.g. scheduler, interrupt service routine, etc.) of OS when it schedules them.

In this paper, we design and implement WCET analysis tool which deliberates on scheduling primitives of system using embedded Linux widely used in embedded OSes. This tool can estimate either WCET of normal application programs or corresponding primitives which have an influence on schduling property in embedded Linux kernel. Therefore, precision of estimation about schedulability analysis is improved. We develop this tool as Eclipse's plug-in to work properly in any platform and support convenient interface or functionality for user.

Key words : Embedded System, Worse-Case Execution Time(WCET), WCET Analysis, Schedulability Analysis

· 본 연구는 한국전자통신연구원 위탁연구과제(2004-S-058)의 지원으로 수행하였습니다.

† 학생회원 : 숭실대학교 컴퓨터학과
darklight@realtime.ssu.ac.kr
†† 종신회원 : 숭실대학교 컴퓨터학부 교수
smyang@ssu.ac.kr
††† 비 회 원 : 한국전자통신연구원 임베디드S/W연구단 연구원
yonghoon@etri.re.kr

논문접수 : 2007년 9월 6일
심사완료 : 2007년 10월 22일

1. 서 론

현재 사용되고 있는 임베디드 운영체제(커널)에서의 실시간성 지원은 이제 추가사항이 아니라 필수 요구사항이 되었다. 임베디드 운영체제가 사용되는 시스템은 논리적인 정확성뿐만 아니라 시간제약 사항에 대한 요구사항을 보장하도록 설계되고 구현된다. 이러한 임베디드 운영체제가 사용되는 시스템의 실시간성 지원을 충족하기 위해서는 시스템 내 실시간성이 요구되는 태스크들의 스케줄링 가능성 여부가 중요하며, 이를 시스템 수행 전 검증해야 할 필요가 있다. 스케줄링 가능성 여부를 판단하기 위한 스케줄링 가능성 분석에서 핵심적인 부분 중의 하나는 태스크의 최악실행시간(Worst-Case Execution Time, WCET)을 구하는 것이고 이를 시스템 수행 전에 검증하기 위해서 정적인 최악실행시간 분석을 사용한다.

이런 이유로 기존의 많은 대학 및 연구소는 최악실행시간 분석에 관한 연구를 진행해왔으며, 여러 최악실행시간 분석 도구들이 구현되었다. 그러나 기존의 도구들은 일반적인 응용 태스크 즉, 응용 프로그램의 최악실행시간을 위주로 분석하였기 때문에 응용 프로그램들이 운영체제에 의해 스케줄링시 영향을 받는 스케줄링 관련 프리미티브(스케줄러, 인터럽트 서비스 루틴등)에 대한 고려는 전혀 하지 않고 있다.

이러한 문제를 해결하고자 본 논문에서는 임베디드 운영체제 중에 널리 사용하고 있는 임베디드 리눅스가 사용되는 임베디드 시스템에서의 스케줄링 관련 프리미티브를 고려하는 최악실행시간 분석 도구를 설계하고 구현한다. 이 분석도구는 일반적인 응용 프로그램 뿐만 아니라 임베디드 리눅스 커널내의 스케줄링에 영향을 미치는 관련 프리미티브의 최악실행시간을 분석하여 스케줄링 분석의 정확성을 더욱 더 높인다. 이 도구는 현재 임베디드 환경에서 통합개발환경으로 제작된 이클립스(Eclipse)[1]의 플러그인 형태로 개발되어 어떠한 플랫폼에서도 동작 가능하고 사용자가 사용하기에 편리한 인터페이스 및 기능을 제공할 수 있도록 구현한다.

본 논문의 구성은 다음과 같다. 2장에서 최악실행시간 분석에 대한 정의와 최악실행시간 분석방법 중 운영체제 프리미티브에 대한 기존 연구들을 알아본다. 3장에서는 스케줄링 프리미티브를 고려한 최악실행시간 분석도구의 설계 및 구현에 대하여 명시하고, 4장에서는 이에 대한 성능 평가를 보여준다. 마지막으로 5장에서 결론 및 향후 연구 방향을 논한다.

2. 관련 연구

2.1 최악실행시간 분석

임베디드 시스템에서 시스템을 구성하는 태스크들의 시간적 정확성을 검증하기 위해서는 우선적으로 태스크들에 대한 최대의 실행시간을 파악해야 한다. 여기서, 태스크들에 대한 실행시간의 상위 한계 값을 구하는 것을 최악실행시간 분석이라고 한다. 태스크의 실행시간은 프로세서가 태스크를 실행하는데 걸리는 시간으로 정의한다[2].

최악실행시간 분석의 목적은 시스템이 수행되기 전에 프로그램의 최악실행시간 정보를 계산하여 미리 제공하는 것이다. 이를 통해 시스템의 최악실행상태를 분석하고 시간적 정확성을 검증한다. 또한 최악실행시간 분석의 목표는 정확한 최악실행시간의 계산이 아니라 과대측정(over estimation) 범위를 가능한 최대로 작게 하는 것이고, 계산된 실행시간의 범위는 신뢰할 수 있어야 하므로 과소측정(under estimation)되지 않아야 한다.

최악실행시간 분석은 일반적으로 두 부분으로 나뉜다. 하나는 상위 수준의 흐름 분석 단계이며, 다른 하나는 하위레벨 분석 단계이다. 상위 수준의 흐름 분석 단계는 프로그램의 제어 흐름 그래프(control flow graph, CFG)를 생성하고, 생성된 그래프에서 가장 긴 실행경로를 찾는 과정이다. 하위레벨 분석 단계는 상위 수준 흐름 분석의 결과 값인 최악실행경로를 따라 프로그램 수행시간에 영향을 미치는 하드웨어 영향, 즉 캐시나 파이프라인, 분기예측과 같은 것을 고려하여 명령어 레벨에서 실행 사이클을 계산하는 단계이다. 그림 1은 이러한 개념을 나타낸다.

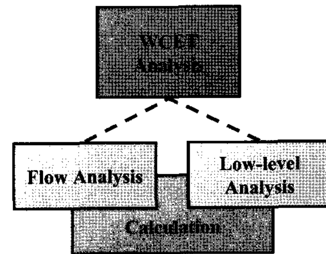

그림 1 상위수준 흐름분석과 하위레벨 분석

2.2 운영체제 프리미티브의 최악실행시간 분석

Colin과 Puaut[3]은 실시간 운영체제인 RTEMS[4]를 HEPTANE[5]이라는 WCET 분석기를 이용하여 분석하였다. 이 연구에서는 실시간 커널인 RTEMS의 커널 함수의 최악실행시간을 예측하여 정적 분석을 하는 접근 방식을 채택하였다. 태스크 관리와 동기화를 구현하는데 필수적인 RTEMS 커널의 12가지 주요 커널 함수를 분석하였다. 그러나 단일 파일 안에서 각각의 커널 함수를 분석하기 위해서는 단계적으로 모든 파일이 통

합되어야 하는 단점이 있다.

Carlsson과 Engblom외[6]는 ARM9TDMI 아키텍처 기반에서 동작하는 OSE[7] 운영체제에서 인터럽트가 금지되는 지역에 대한 최악실행시간 분석에 대해 연구하였다. 인터럽트가 금지된 지역(disable interrupt region, DI region)이라고 부르는 특정 코드 영역에 대한 최악실행시간을 분석함으로써 전체 시스템에서 스케줄링 가능성을 분석하는데 목적을 두었다. Sandell과 Ermedahl외[8]은 ARM7TDMI를 기반으로 OSE 운영체제의 코드에 대한 최악 실행시간을 분석하였다. 이 연구는 WCET 측정의 정확성 보다 현재 WCET 분석방법을 실제 소스코드에 적용시킬 때, 발생하는 어려운 점들을 조사하는데 목적을 가지고 있다. 위에서 언급한 연구들은 모두 커널 내에서 스케줄링 가능성 분석에 영향을 끼치는 스케줄링 프리미티브들(스케줄러, 인터럽트 서비스 루틴 등)에 대한 분석은 이루어지지 않았다.

3. 설계 및 구현

본 논문에서 제안하는 최악실행시간 분석도구는 특정 프로세서를 대상으로 하고 특정 스케줄링 분석기의 연동을 목적으로 설계되었으나, 각각의 모듈들이 독립적으로 동작하도록 설계되었기 때문에 다른 프로세서 혹은 다른 스케줄링 분석기와의 연동 또한 가능하다. 이 분석도구는 기본적으로 C로 작성된 소스코드를 분석하여 기본블록(basic block)으로 구성되는 제어 흐름 그래프(Control Flow Graph, CFG)를 생성한다. 기본블록이란 분기(branch)나 순환(loop)이 포함되지 않은, 시작과 끝이 분명한 제어흐름을 가진 연속된 문장들(statements)의 순서열이다[9]. 따라서 CFG의 각 노드는 기본블록이 된다. 이 정보들을 바탕으로 최악실행시간을 계산하는 분석도구이다. 다음 그림 2는 최악실행시간 분석도구의 전체 구성을 추상화 한 것이다.

분석도구는 크게 두 부분으로, 흐름 분석기와 실행시간 분석기로 구성된다. 흐름 분석기는 프로그램 소스 코드로부터 구문 트리를 생성하고, 이 트리를 기본으로 상위 수준 기본블록을 생성하며, 프로그램 경로 그래프를 생성한다. 또한 프로그램 소스 코드의 오브젝트파일(obj)을 입력받아 하위 수준 기본블록으로 나눈다. 이후, 프로그램 경로 그래프와 하위 수준 기본블록을 실행시간 분석기로 전달한다.

실행시간 분석기는 흐름 분석기가 생성한 기본블록과 흐름 정보를 입력으로 받은 후 명령어를 분석한다. 명령어 분석 시 계산되는 최악실행시간은 하드웨어 특성(캐시, 파이프라인, 분기 예측)을 고려한다. 분석결과로서 표시되는 최악실행시간의 단위는 실제 프로세서의 실행 사이클이다. 최악실행시간 분석 시, 분석도구에서 파악하기 어려운 분기의 선택이나, 반복 순환의 횟수등과 같은 사용자가 입력 가능한 정보를 처리하기 위하여 Annotation 기법을 사용한다. 이를 위해 힌트(Hint) 정보라는 구조체가 존재하고 사용자가 이를 정의할 수 있다. 이는 다른 스케줄링 분석도구와의 연동에서 사용될 수 있다.

3.1 흐름 분석기

흐름 분석기는 프로그램 소스 코드를 입력으로 받아, 코드 내의 기본블록들과 각 기본블록 간의 관계에 대한 정보가 포함되어 있는 제어 흐름 그래프를 생성한다. 생성된 CFG 파일에 대한 기본적인 파싱이 이루어지고, 이 파일을 분석하여 기본블록에 대한 정보를 수집한다.

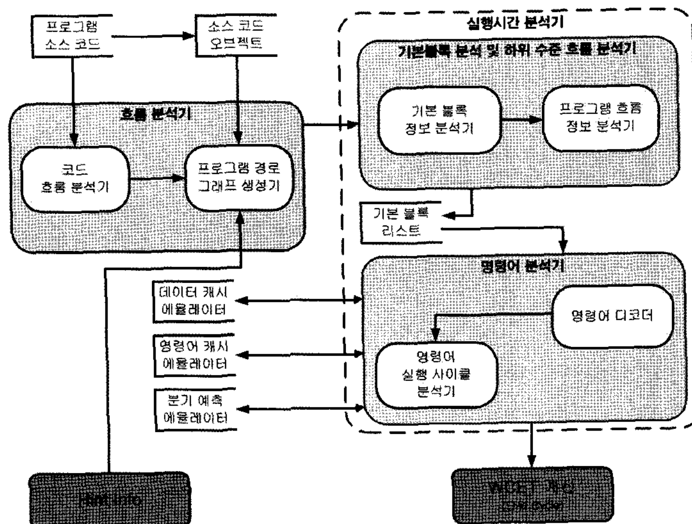

그림 2 최악실행시간 분석도구의 구성도

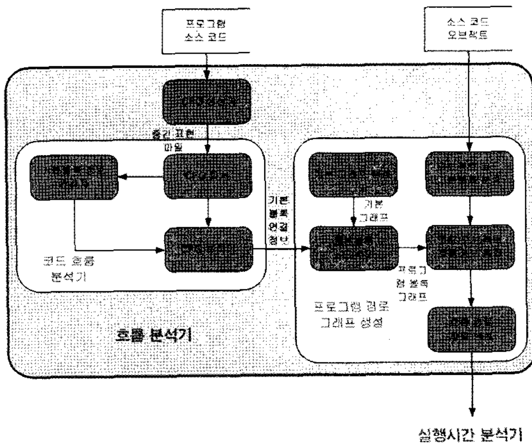

그림 3 흐름 분석기

또한 코드의 오브젝트 파일 기본블록 정보와 코드 상의 기본블록 정보를 동기화시킨다. 분석된 기본블록에 대한 정보를 바탕으로 흐름 경로 그래프를 생성하고 이 그래프를 이용하여 최악경로를 구한다. 흐름 분석기는 크게 코드 흐름 분석기와 프로그램 경로 그래프 생성기로 구성된다. 그림 3은 이러한 흐름 분석기의 전체 구조를 나타낸다.

3.1.1 코드 흐름 분석기

코드 흐름 분석기는 CFG 파일 생성기와 CFG 파서, CFG 분석기로 이루어져 있다. CFG 파일 생성기는 프로그램 소스 코드를 입력받아 CFG 파일을 생성한다. CFG 파일은 상위 수준 프로그램 언어(즉, C언어)에서 기본블록으로 나누고 각 분기 시에 어떤 기본블록으로 이동하는지에 대한 정보가 포함되어 있으며, 각각의 기본블록은 3-주소 코드(3-address code)[10] 형식으로 구성되어 있는 파일이다. 이 CFG 파일은 GNU GCC 4.0의 C 컴파일러에서 지원되는 것으로, CFG 파일 생성기는 이 GCC C 컴파일러를 이용하여 CFG 파일을 생성한다. CFG 파서는 생성된 CFG 파일에 대한 파싱을 담당한다. 파싱 작업이 수행되는 동안 기본블록 번호, 기본블록이 위치한 소스라인번호, 분기될 기본블록 번호들, 분기를 결정하기 위해 사용되는 조건식 등, 기본블록에 대한 여러 기본 정보를 수집하고 최종적으로 CFG 파일에 대한 추상 구문 트리를 생성한다. 이런 정보의 수집은 기본블록 정보 관리자가 수행한다. CFG 파서는 Java의 파서 생성기인 Javacc[11]로 구현된다.

CFG 분석기는 파싱의 결과로 나온 추상 구문 트리 정보를 입력으로 받아 추상 구문 트리내의 노드들을 분석하고 기본블록 정보 관리자에서 수집한 기본블록 정보를 기반으로 기본블록 연결 정보를 생성한다. 이 기본블록 연결 정보에는 파싱되고 분석된 모든 기본블록에 대한 정보가 포함되어 있다. 특히, 사용자에게 분기 또는 반복되는 루프에 대한 정보를 주기위한 힌트 정보를 분석한다. 코드 흐름 분석기는 일반적인 응용 프로그램뿐만 아니라, 임베디드 리눅스 커널의 스케줄링 프리미티브들(스케줄러, 인터럽트 서비스 루틴등)에 대한 코드 흐름 분석이 가능한 것이 장점이다. 이와 관련하여 자세한 것은 3.3절에서 다룬다.

3.1.2 프로그램 경로 생성기

프로그램 경로 생성기는 크게 오브젝트 코드의 기본블록 분석을 수행하고 코드 흐름 분석기에서 받은 기본블록 정보를 이용하여 코드와 오브젝트의 흐름정보를 통합한다. 특히 이 과정에서 최악경로(Worst-Case Control Path)를 구해낸다. 오브젝트 코드 기본블록 분석기는 오브젝트 코드의 기본블록을 구분한다. 코드/오브젝트 흐름정보 통합기는 기본블록 노드가 추가된 프로그램 제어 경로 그래프와 오브젝트 기본블록 분석 정보를 사용하여 프로그램 제어 경로 내의 각 노드와 오브젝트 기본블록 정보를 통합한다. 하나의 기본블록 노드는 하나의 오브젝트 기본블록 정보를 참조한다. 흐름 경로 파일 생성기는 추상 기본블록 트리와 오브젝트 파일 정보를 이용하여 프로그램 제어 경로 그래프를 생성하고 이를 이용하여 최악경로를 구한다. 흐름 경로 파일은 최악실행시간 엔진에서 사용되는 파일로서, 프로그램 흐름에 대해 각 기본블록이 수행되는 순서가 기록되어 있다. 최악실행시간 엔진에서 이 파일을 입력받아 프로그램 전체의 최악실행시간을 계산한다.

3.2 실행시간 분석기

실행시간 분석기는 오브젝트 기본블록 파일과 흐름 경로 파일을 입력으로 받는다. 기본블록 정보 파일은 오브젝트 덤프 유틸리티로부터 생성된 덤프 파일에서 기본블록을 추출하고, 흐름 경로 파일은 흐름 분석기로부터 받는다. 실행시간 분석기는 크게 기본블록 분석 및 하위수준 흐름 분석기와 명령어 분석기로 구성된다. 기본블록 분석 및 하위수준 흐름 분석기는 코드 흐름 분석기와 비슷한 역할을 하나, 코드 흐름 분석기가 상위 수준 프로그램 언어를 주 대상으로 하는 반면, 기본블록 분석 및 하위수준 흐름 분석기는 오브젝트 파일내의 어셈블리 언어에 대한 기본블록 및 흐름 분석을 수행한다는 차이점이 있다. 이 때 흐름 분석은 코드 흐름 분석기에서 입력받은 흐름 경로 정보를 기반으로 한다.

명령어 분석기는 실제 명령어 사이클을 분석하는 부분으로, 명령어/데이타 캐시와 파이프라인, 분기 예측을 고려하여 전체 사이클을 계산한다. 명령어 분석기는 명령어 디코더와 명령어 실행 사이클 분석기로 구성된다. 명령어 디코더는 캐시나 파이프라인, 분기 예측에서 명령어에 따른 분석을 적용하기 위해서 수행하며, 명령어

의 타입과 피연산자(operand)를 분류한다. 본 논문에서 구현한 실행시간 분석기는 ARM계열 프로세서인 XScale[12]을 기반으로 한다. 그러므로 분류의 기준은 ARM 계열 명령어 분류 기준에 따른다.

명령어 실행 사이클 분석기에서는 캐시 분석을 위해 명령어와 데이타 캐시 시뮬레이션 방법을 사용한다. 이 방법은 기본블록의 명령어를 수행하면서 발생하는 캐시 적중과 캐시 미스를 측정하고, 캐시 미스가 발생 시에 메모리로 접근하는데 걸리는 패널티 사이클을 부여한다. 또한 명령어 분석기는 파이프라인 분석을 위해 명령어 디코더에서 분류한 명령어의 종류에 따라 수행하는데 걸리는 최소 사이클의 개수와 레지스터 의존성에 의해 발생할 수 있는 지연 사이클의 존재 여부를 판단, 명령어 사이클의 총 개수를 계산한다.

분기 명령어가 발생했을 때를 위한 분기 예측 에뮬레이터를 이용하여 분기 주소를 결정하고 그 주소를 다음 실행하는 기본블록의 첫 명령어 주소와 비교한다. 이 두 값이 같을 경우 분기 예측 성공이 되고, 다를 경우 분기 예측 실패가 된다. 분기 예측 실패가 발생하였을 경우, 패널티 사이클을 적용한다. 이 패널티 사이클은 기존에 실행된 명령어를 파이프라인에서 비우는데 걸리는 시간이다.

3.3 CFG파서(CFGParser)

코드 흐름 분석이 가능하려면 입력되는 프로그램 소스 코드의 파싱(parsing)이 먼저 우선된다. 본 논문에서 파싱의 사용은 일반적인 구문 분석(syntax analysis)을 위함이 아닌, 중간 코드(intermediate code)를 생성하여 기본블록을 추출하는데 목적이 있다. 중간코드가 필요한 이유는 일반적인 프로그램 언어(C 같은)에서 기본블록을 추출하기 위해서는 한 문장 내에 존재할 수 있는 분기나 순환을 분리해야 되기 때문이다. 이는 컴파일러의 구현에서 사용되는 일반적인 방식이다. 물론 컴파일러는 이를 여러 다른 플랫폼상의 기계어로 변환하려는 목적으로 사용한다.

응용프로그램에 대한 소스 코드의 경우는 보통 표준 문법을 따르기 때문에 파싱을 위한 파서를 생성하고 코드 흐름 정보를 추출하는 것은 그다지 어려운 일이 아니다. 그러나 커널과 같은 시스템 소프트웨어에 대한 소스 코드의 흐름 정보 추출은 쉽지 않은데 그 이유는 다음과 같다. 첫째, 커널 코드는 수많은 전처리 문장(예, 매크로나 #define으로 선언된 문장들)이 포함되어 있기 때문에 코드를 파싱하는데 어려움이 따른다. 둘째, 여러 가지 자료구조가 복잡하게 얽혀 있기 때문에, 선언된 변수에 대한 분석이 힘들다. 셋째, 커널 코드가 표준 문법 외에 특정 컴파일러의 확장문법을 사용하는 경우가 있다. 특히, 리눅스 커널은 성능 최적화를 위해 표준 C 문법에 GCC Extension[13]이라는 확장문법을 사용하고 있다. 따라서 소스 코드를 직접 파싱하는 기존의 방법과는 다른 접근 방법이 요구된다.

3.3.1 CFG 파일

본 논문의 흐름 분석기의 대상은 임베디드 리눅스 상에서 동작하는 응용 프로그램들과 리눅스 자신의 커널 코드이다. 위에서 언급한 커널 코드의 흐름 분석의 어려움을 해결하기 위해서 리눅스 커널을 컴파일하는 기본 컴파일러인 GCC 컴파일러의 기능을 사용한다. GCC 4.0에서 지원하는 CFG dump 기능은 GCC 컴파일러 자신이 사용하는 CFG 정보를 출력하는 기능이다. 컴파일 시에, "-fdump-tree-cfg-lineno"과 같은 옵션을 포함하여 소스 코드를 컴파일하면, 이진 실행파일 이외에 디버깅 정보로써, CFG 파일이 생성된다. 그림 4는 CFG 파일 생성의 예를 보여준다.

```
 ...
>gcc -c foo.c -fdump-tree-cfg-lineno
 ...
```

그림 4 CFG 파일의 생성

그림 4의 작업이 수행되면, "foo.c.t13.cfg"라는 CFG 파일이 생성된다. 이 CFG 파일은 3-주소 코드(3-address code)로 구성되어 있으며, 모든 코드는 전처리 과정이 되어있다. 또한 코드의 함수별로 기본블록이 나누어져 있으며, 라인정보와 기본블록 연결정보도 가지고 있다. 3-주소 코드의 일반적인 형식은 다음과 같다. 본 논문의 최악실행시간 분석도구는 이 3-주소 코드의 기반을 둔다.

- x = y *op* z : 일반적인 할당문(assignment statement)을 가리킨다. 여기서 *op*는 이진 산술 또는 논리 연산자이다.
- x = *op* y : *op*가 단항연산자일 경우이다. 단항 연산자로는 마이너스(-), 논리 부정(!), 시프트 연산자 등을 포함한다.
- x = y : y 값이 바로 x에 할당되는 형식의 복사 문장이다.
- *goto* L : 무조건 분기, 레이블 L 다음의 3개 주소 문장이 실행되는 부분이 된다.
- *if* x *relop* y *goto* L : 조건 분기, x 와 y에 대한 관계 연산자 등에 적용되며 x가 y에 대한 관계 *relop*를 만족하면 레이블 L에 위치한 문장이 실행되고 그렇지 않으면 바로 다음의 문장이 실행된다.
- *func()* : 일반적인 함수가 호출될 때의 문장이다.

이 CFG 파일의 장점은 커널 코드에서 사용되는 복잡한 전처리문장들 혹은 GCC 문법 확장(extension)들이

이미 처리되어 있다는 것이다. 따라서 일반적인 소스 코드의 컴파일 시에 컴파일 옵션이나 매크로의 선언에 대한 고려가 필요 없다. 예를 들어, 리눅스 커널 컴파일 전에 수행되는 "*make config*"의 결과에 따라 실제 커널 코드가 수행되는 부분과 수행되지 않는 부분을 구별해야하는 작업은 CFG 파일이 생성될 때 컴파일러에 의해서 수행되므로, 커널 코드 자체를 파싱하기 위해 부가적인 노력을 기울일 필요가 없다.

3.3.2 CFG 파일의 형식

그림 5는 리눅스 커널 코드중 kernel/sched.c내의 스케줄러 핵심 함수인 schedule()함수에 대한 CFG파일의 일부분을 나타낸 것이다.

```
  # BLOCK 26, starting at line 2730
  # PRED: 25 (true)
<L45>:;
  [kernel/sched.c : 2730] D.16099 = rq->nr_uninterruptible;
  [kernel/sched.c : 2730] D.16100 = D.16099 + 1;
  [kernel/sched.c : 2730] rq->nr_uninterruptible = D.16100;
  # SUCC: 27 (fallthru)
...
...
  # BLOCK 28, starting at line 2735
  # PRED: 17 (false) 18 (false) 24 (fallthru) 27 (fallthru)
<L47>:;
  [kernel/sched.c : 2736] D.16101 = rq->nr_running;
  [kernel/sched.c : 2736] D.16102 = D.16101 == 0;
  [kernel/sched.c : 2736] if ([kernel/sched.c : 2736] D.16103 != 0)
goto go_idle; else goto <L55>;
  # SUCC: 29 (true) 31 (false)
...
...
```

그림 5 sched.c.t13.cfg (schedule()함수) 파일

각 기본블록은 "# BLOCK"으로 시작하며, 그 다음의 숫자는 기본블록의 번호를 표시한다. "starting at line"은 기본블록이 커널 소스에서 위치하는 라인번호를 나타낸다. "# PRED"와 "# SUCC"는 기본블록의 predecessor와 successor를 표시한다. 즉 "# PRED" 다음에 나타나는 숫자들은 현재 기본블록이 수행되기 전의 앞선 기본블록을 표시하는 것이며, "# SUCC" 다음의 숫자들은 현재 기본블록이 수행되고 나서 그 다음 수행될 기본블록의 숫자를 표시한다. "<L55>"는 3-주소 코드 형식에서의 레이블이다. predecessor와 successor가 표시될 때 각 기본블록이 분기된 조건에 따라서 구별될 수 있는데, 그 조건은 다음을 따른다.

- (true) : 만일 분기가 존재할 때 분기되는 조건의 식이 참일 때
- (false) : 만일 분기가 존재할 때 분기되는 조건의 식이 거짓일 때
- (fallthru) : 분기가 없이 다음 기본블록으로 이동시

예를 들어, 그림 5의 예에서 기본블록 28번에 대한 successor는 "# SUCC: 29 (true) 31 (false)"이다. 이 의미는 기본블록 28번이 마지막에 분기할 때 분기조건이 참이면 29번 기본블록으로, 거짓이면 31번 기본블록으로 분기하라는 것이다. predecessor 리스트의 경우는 여러 개의 기본블록이 포함될 수 있으나, successor 리스트의 경우는 "(true) (false)"나 "(fallthru)"의 경우가 존재한다.

3.3.3 기본블록 정보 파일

CFG 파일 파서는 이전에 설명한 것과 같은 구조를 가진 CFG 파일을 파싱하여 CFG 파일에 대한 추상 구문 트리(Abstract Syntax Tree, AST)를 생성한다. Java용 파서 생성기인 Javacc와 JJtree를 이용하여 CFG 파일 파서 클래스인 CParser(CFGParser)를 생성한다. CFG 파일 파서에 의해 파싱된 각 기본블록의 정보를 ASTFunctionDefinition과 ASTBlock 같은 기본블록 정보 관리 클래스를 통하여 얻는다. 다음 그림 6은 기본블록 정보 클래스들의 구성 개념을 보여준다.

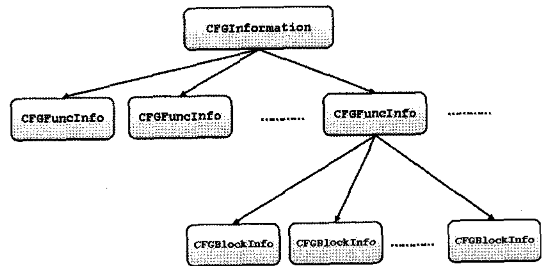

그림 6 기본블록 정보 클래스의 구조

분석 대상이 되는 하나의 커널 코드 파일에 대한 정보는 CFGInformation 클래스가, 파일 내의 각 함수에 대한 정보는 CFGFuncInfo 클래스가 저장하고 있다. 함수를 구성하는 기본블록에 대한 정보는 CFGBlockInfo 클래스가 가지고 있다. 따라서 CFGInformation 클래스는 CFGFuncInfo 클래스의 리스트를 포함하고 있고, CFGFuncInfo 클래스는 여러 개의 CFGBlockInfo 클래스로 구성된다.

3.4 상위 레벨과 하위 레벨의 기본블록 매핑

일반 응용프로그램 코드의 경우 흐름 분석기에 의해 분석된 상위 흐름 경로 정보와 오브젝트 덤프 파일의 하위 흐름 경로 정보는 1:1 매핑이 바로 이루어진다. 그러나 커널 코드의 경우는 1:1 매핑이 이루어지지 않는 경우가 있는데, 성능의 최적화 혹은 구현 기법으로(예를 들어, nop를 구현하기 위한) 기본적인 할당문이 없는 기본블록이 존재하기도 한다. 이런 경우, 오브젝트 덤프 파일 내에 의미 없는 분기문이 추가되기 때문에, 이를 해결하기 위해 하위 수준 흐름 분석기에서 상위 기본블록 정보와 1:1 매핑이 되도록 기본블록에 대한 유효성 검사와 동시에 상위 레벨의 기본블록 정보와의 매핑을

수행한다.

하위 흐름 경로를 분석하기 전에, 오브젝트 덤프 파일에 대한 유효 분기 테이블(valid branch table)을 작성한다. 오브젝트 덤프 파일내에 존재하는 분기문들에 대해서, 분기 테이블의 엔트리에는 분기문의 유효 속성값이 들어간다. 이 속성값은 분기 대상이 되는 주소가 기본블록을 나누는 기준이 되는 주소인지에 따라 달라진다. 그림 7은 이러한 유효 분기 테이블의 자료구조를 보여준다. 그림 7에서 bool 형으로 정의된 *realbranch*가 유효 속성값을 표현한 것이다.

이 유효 분기 테이블을 이용하여 다음의 과정으로 하위 기본블록 정보를 생성한다. (1)현재 명령어가 존재하는 섹션(컴파일러에 의해 정의된 섹션)을 찾는다. (2)현재 명령어가 분기 명령어 인지 검사한다. (3)현재 명령어가 분기 명령어일 경우 현재 명령어의 주소가 분기 테이블 항목에 있는지 검사하여, 존재한다면 현재 명령의 이전을 기본블록으로 나눈다. (4)현재 명령어가 bl, blx가 아닌 경우 현재 명령어 다음도 기본블록을 나눈다. (5) 현재 명령어의 주소가 분기 테이블 항목에 등록되어 있는지 검사하여, 해당 항목이 존재한다면 기본 블록을 나눈다. 그림 8은 이런 과정의 예를 그림으로 표현한 것이다.

```
typedef struct __BranchTableElement
{
        UINT32                          address;
        char                            instruction[8];
        bool                            realbranch;
        struct __BranchTableElement     *next;
} BranchTableElement;

typedef struct __BranchTable
{
        struct __BranchTableElement             *element;
        struct __BranchTable                    *next;
} BranchTable;

BranchTable *headOfBranchTable = NULL;
```

그림 7 섹션별 분기 주소를 저장하기 위한 자료구조

3.5 스케줄링 프리미티브의 최악실행시간 분석

그림 9는 커널 코드로 구성되어 있는 스케줄링 프리미티브(kernel/sched.c의 Schedule()함수)의 최악실행시간을 계산하는 과정의 예를 나타낸다.

크게 왼쪽 부분이 커널코드 흐름 분석 부분이고, 오른쪽 부분이 실행시간 분석 부분이다. 그림 8과 같은 매핑 과정이 이루어진 CFG 기본블록 정보를 바탕으로 미리 정해진 흐름분석 방법에 따라 상위레벨 흐름의 최악수행경로를 계산하여 실행시간 분석기에 전달한다. 실행시간 분석기에서는 해당 경로 정보를 이용하여 오브젝트 코드 기본블록의 최악실행시간을 계산한다.

4. 동작환경 및 평가

4.1 동작환경

최악실행시간 분석도구는 INTEL XScale 내장 프로세서를 대상으로 하고, XScale용 GNU C/C++ 크로스 컴파일러에 의해 생성된 프로그램에 대하여 분석하며, INTEL x86 호환기종에서 동작한다. 다양한 플랫폼 상에서의 동작을 지원하기 위해서 JVM(java virtual machine) 상에서 동작하는 통합개발환경인 이클립스의

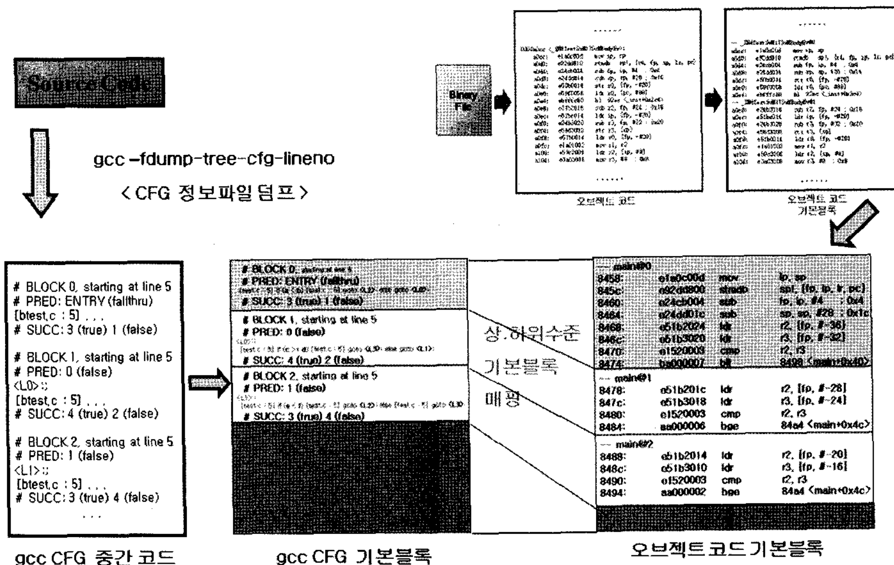

그림 8 상,하위 기본블록 정보의 매핑

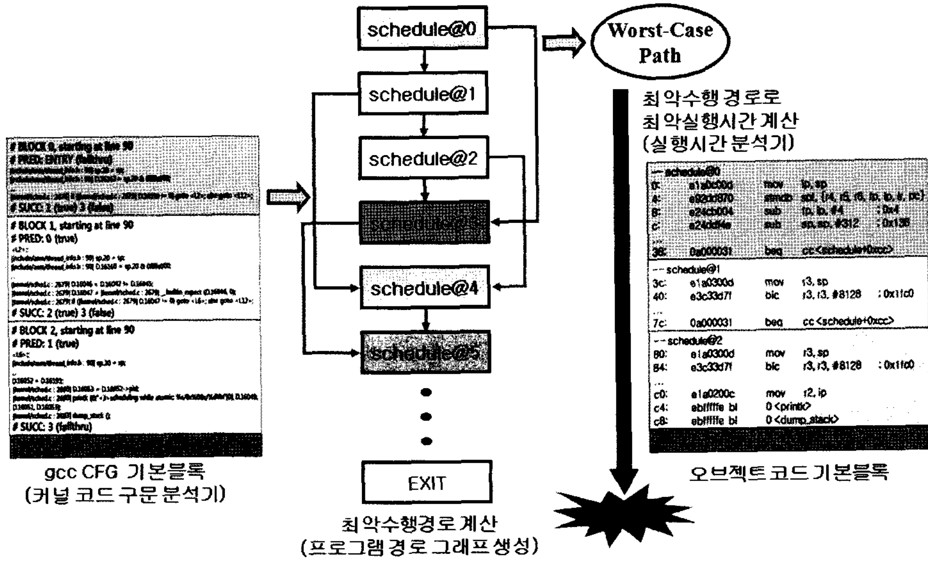

그림 9 Schedule()의 최악실행시간 계산 과정

플러그인으로 제공된다.

상위 수준의 흐름 분석기는 플러그인 내에 존재하기 위해 Java로, 하위 실행 시간 분석기의 구현은 C로 구현된다. 흐름 분석기와 실행시간 분석기간의 연동을 위해, JNI(Java Native Interface)를 사용한다. 그림 10은 특정 스케줄링 분석도구와 연동된 최악실행시간 분석도구의 실행화면이다.

4.2 평가

본 논문에서 구현한 커널 코드 최악실행시간 분석 도구의 성능을 측정하기 위해 일부 커널 코드와 테스트 프로그램의 실제 수행시간과 본 연구에서 구현한 분석 도구로 측정한 WCET 값 간의 오차를 비교하는 방식으

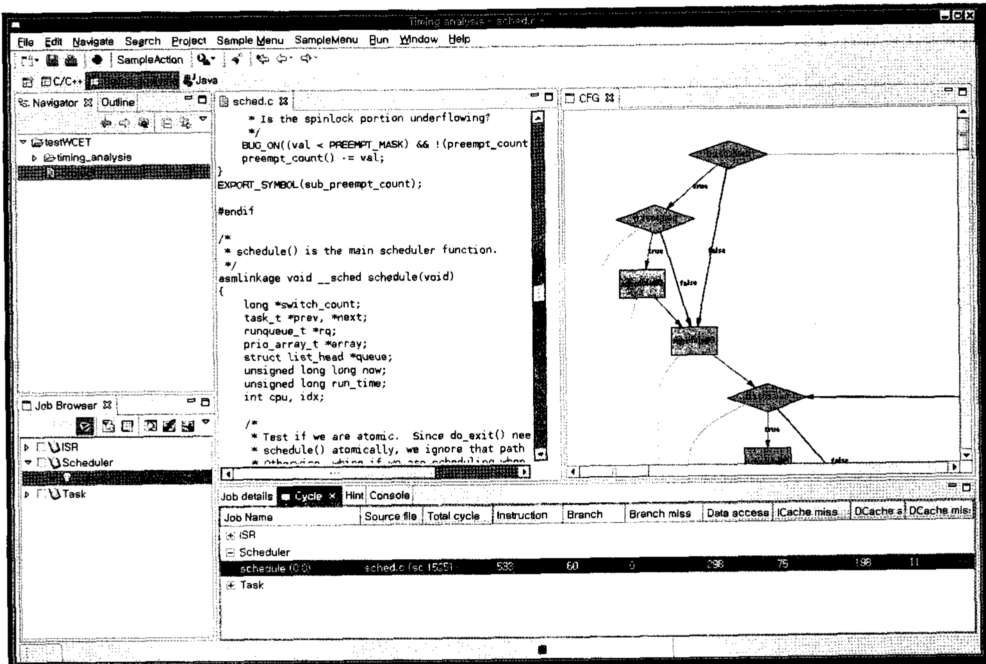

그림 10 최악실행시간 분석도구 이클립스 플러그인의 실행화면

```
asmlinkage void schedule(void)
{       ... 중략 ...
①       if (!current->active_mm) BUG();
need_resched_back:
        prev = current;
        this_cpu = prev->processor;

②       if (unlikely(in_interrupt())) {
                printk("Scheduling in interrupt\n");
                BUG();
        }
        ... 중략 ...
        /* move an exhausted RR process to be last.. */
③       if (unlikely(prev->policy == SCHED_RR))
                if (!prev->counter) {
                        prev->counter = NICE_TO_TICKS(prev->nice);
                        move_last_runqueue(prev);
                }
        ... 중략 ...
        prepare_to_switch();
        {
                struct mm_struct *mm = next->mm;
                struct mm_struct *oldmm = prev->active_mm;
                if (!mm) {
④                       if (next->active_mm) BUG();
                        next->active_mm = oldmm;
                        atomic_inc(&oldmm->mm_count);
                        enter_lazy_tlb(oldmm, next, this_cpu);
                } else {
⑤                       if (next->active_mm != mm) BUG();
                        switch_mm(oldmm, mm, next, this_cpu);
                }
        ... 중략 ...
}
```

그림 11 리눅스 커널(2.4.18)의 schedule() 함수

로 성능 평가를 진행한다. 다음은 성능 평가 환경을 나타낸다.

- Host : Intel x86 System
- Host OS : GNU/Debian Linux
- Cross Compiler : GNU C Compiler 4.0 (Target: XScale)
- Objdump Utility (Target: XScale)
- Target OS : Embedded Linux Kernel 2.4.18

성능 평가에 사용할 커널 코드는 스케줄링 프리미티브 중 하나인 리눅스 커널 2.4.18 스케줄러(kernel/sched.c)의 schedule() 함수이며, 응용 프로그램으로는 선형 탐색 함수 테스트 프로그램 코드를 사용한다. 코드의 실제 수행시간 측정은 XScale의 모니터링 Coprocessor를 사용하여 측정한다. 이것은 XScale용 리눅스 커널 2.4.18 버전에 리눅스 모듈로 테스트 프로그램 수행 코드를 삽입하여 측정하였다.

그림 11은 리눅스 커널의 schedule() 함수의 일부분이고, 그림 12는 선형 탐색 프로그램이다.

```
int linear_search(int array[100], int value)
{
    int i;

    for (i=0; i<100; i++) {
        if(array[i]==value)
            return i;
    }
}

int main() {
    int a[100], i;

    for (i=0; i<100; i++)
        a[i] = i+1;

    i = linear_search(a, 50);

    return 0;
}
```

그림 12 선형 탐색 프로그램

4.3 평가 결과

커널의 schedule() 함수와 선형 탐색 프로그램에 대하여 표 1은 본 논문에서 구현한 최악실행시간 분석도구에 의한 분석결과이며, 표 2는 XScale 모니터링으로 측정한 수행시간 값이다.

각 프로그램에 대한 과대 측정 비율은 다음과 같은 식을 사용하여 계산했으며 이를 표 3에서 보여준다.

- $OER = (\frac{WTE}{RE} - 1) \times 100$

표 1 최악실행시간 분석기에 의한 분석 결과

프로그램	힌트 반영 여부	결과 값 (Clock Cycle)
schedue()	반영 하지 않음	15,522
	분기힌트 반영 (도달 불가능한 흐름 제거)	12,985
linear_search	반영 하지 않음	1,530
	반복횟수 50 반영	780

표 2 XScale 성능 모니터링 측정 결과

프로그램	평균 실행시간(Clock Cycle)
schedue()	11,348
linear_search	443

표 3 최악실행시간 분석기의 과대측정 비율

프로그램	힌트 반영 여부	최악실행시간 분석기 분석 결과 (WTE)	XScale 측정 결과 (RE)	과대 측정 비율 (OER)(%)
schedule()	반영 하지 않음	15,522	11,348	36.79
	분기힌트 반영 (도달 불가능한 흐름 제거)	12,985		14.43
linear_search	반영 하지 않음	1,530	443	245.37
	반복횟수 50 반영	780		76.07

*OER*은 과대 측정 비율을 나타내며, *WTE*는 최악실행시간 분석기가 계산한 분석결과(단위 Cycle)를, *RE*는 *XScale*에서 측정한 결과(단위 Cycle)를 의미한다.

4.4 평가 분석

분석 결과에 의하면, 커널의 schedule() 함수의 경우 아무런 힌트를 주지 않고, 최악실행시간을 분석했을 때, 36.79%의 과대 측정 비율을 보이지만, 도달 불가능한 흐름을 제거하는 분기 힌트를 반영하였을 경우에 과대 측정 비율을 14.43%까지 줄일 수 있었다. 기본적으로 아무런 힌트를 주지 않을 경우에 모든 흐름들 가운데 가장 긴 경로를 최악실행 시간에 이용한다. 때문에 현재의 시스템 설정 상태에 따라 수행하지 않는 흐름까지 반영해서 계산하기 때문에 과대 측정을 유발하게 된다. 위에서 말한 도달 불가능한 흐름은 다음과 같다.

그림 11에서 ①의 위치를 보면 current->active_mm이 null일 경우 오류가 발생하게 되는데, active_mm은 cpu에 의해 제어중인 사용자 주소 공간을 말하며, 이 주소가 존재하지 않을 경우 오류 상태에 빠지게 되는 것이다. 실제의 수행에서 이런 경우는 시스템이 오작동을 일으키지 않는 이상 발생하지 않기 때문에 일반적으로 도달 불가능한 흐름이므로 제외해야 한다. 따라서 if 조건문 내부의 BUG(); 문장 쪽이 아니라 바로 다음 문장으로 분기하도록 힌트를 반영 한다. 그림 11의 ②는 인터럽트 중에 schedule() 요청이 발생했을 경우 스케줄러가 동작하지 않게 처리를 하는 코드이다. 이 경우도 도달 불가능한 흐름이라 간주하고, if 조건문 내부의 문장 쪽이 아니라 바로 다음 문장으로 분기하도록 힌트를 반영 한다.

그림 11의 ③은 스케줄링 정책이 SCHED_RR 일 경우, if 조건문 내부의 문장을 수행하는 코드이다. SCHED_RR은 연성 실시간 프로세스를 위해 지원하는 스케줄링 정책의 하나이다. 성능 평가에 쓰인 커널에는 이 스케줄링 정책을 설정하지 않았다. 따라서 해당 조건문 내부의 문장은 도달 불가능한 흐름이므로 제외하기 위해 if 조건문 내부 문장이 아니라 바로 다음 문장으로 분기하도록 힌트를 반영한다. 마지막으로 그림 11의 ④와 ⑤는 ①과 같은 형태를 가지므로, 같은 방식으로 분기 힌트를 반영한다. 이와 같이 도달 불가능한 문장으로의 수행 흐름을 제거하는 분기 힌트를 반영하여 과대 측정을 줄일 수 있다.

선형 탐색 프로그램의 경우, 그림 12의 코드에서 100개의 배열에 1~100까지의 숫자를 순차적으로 저장하고, 선형 검색 방법으로 50을 찾는 행동을 한다. 아무런 힌트를 주지 않을 경우 배열의 개수 n이 100개이므로 n번 즉 100번 반복하는 것으로 인지하여 245%의 과대 측정을 하게 된다. 하지만 본 논문에서 구현한 분석기는 반복 횟수를 사용자 입력으로 받을 수 있기 때문에, 사용자가 코드를 보고 숫자 50이 배열의 50번째에 위치하기 때문에, 50번 반복한다는 힌트를 주게 되면 과대 측정 비율은 76%로 현저하게 줄어들게 된다. 변수에 의해 결정되는 반복문의 경우, 사용자 입력에 의해 결정하는 기능을 지원하면서 이 부분에 대한 과대 측정을 줄일 수 있게 되었다.

성능 평가에 사용한 schedule() 함수와 선형 탐색 함수 두 프로그램에 적절한 힌트를 반영한 경우에도 14%~76% 정도의 오차를 보여준다. 그 이유는 하위 레벨 분석 단계에서 Clock Cycle을 계산할 때에 분석 당시 시스템의 모든 레지스터와 캐쉬, 파이프라인 상태를 모르기 때문에, 이 모든 값들을 초기화하고, 분석을 진행하기 때문이다. 캐쉬 미스 한 번에도 수십에서 수백 Clock Cycle의 차이가 발생하므로 이런 오차가 생기는 것이다. 이 부분을 개선하기 위해서는 분석 대상 시스템의 레지스터와 캐쉬, 파이프라인 상태 등의 정보를 알려주는 OS 프로파일러 같은 도구가 필요할 것이다.

5. 결론 및 향후 연구 방향

컴퓨터 기술의 급속한 발전과 함께, 우리는 언제 어디서나 컴퓨터 환경을 접할 수 있게 되었다. 이러한 환경에서 임베디드 시스템의 실시간성 지원에 대한 요구가 증가하고 있다. 이를 충족하기 위한 방법 중 하나는 임베디드 운영체제에서 태스크들의 스케줄링이 가능한지 불가능한지를 판단하는 스케줄링 가능성을 분석하는 것이다. 스케줄링 가능성 분석을 위해서는 각 태스크의 최악실행시간 정보가 필수적이다. 그래서 기존의 많은 대학 및 연구소는 최악실행시간 분석에 관한 연구 및 도구를 구현해 왔으나, 임베디드 운영체제의 스케줄링에 영향을 끼치는 요소에 대한 고려는 이루어지지 않았다. 본 논문에서 구현한 최악실행시간 분석도구는 다음과 같은 장점을 가지고 있다. 첫째, 본 최악실행시간 분석도구는 특정 임베디드 프로세서를 대상으로 하고 있으며, 이러한 특정 프로세서에 대한 기존의 도구는 존재하지 않는다. 둘째, 기존의 도구들이 일반 응용 프로그램의 최악실행시간을 위주로 분석하였기 때문에 응용 프로그램 수행 중에 커널 모드로 진입하였을 때의 상황은 반영할 수 없었으나, 본 최악실행시간 분석도구는 이를 반영하여 좀 더 세밀한 스케줄링 분석이 가능하다. 셋째, 분기 정보와 반복문의 반복 횟수 등의 힌트를 사용자가 직접 입력하여 반영할 수 있는 기능을 지원하기 때문에 정확한 최악실행시간 분석이 이루어진다.

향후 연구 방향으로는 다음과 같다. 현재 분석 대상 파일에서 다른 함수의 호출이 발생할 때 이 함수에 대한 최악실행시간을 계산하기 위해, 호출함수테이블을 사

용한다. 이 호출함수테이블에는 분석 대상이 호출하는 함수들에 대한 정보와 최악실행시간 값이 포함된다. 따라서 분석 대상의 최악실행시간을 계산하기 위해 해당 호출함수테이블내의 모든 함수들에 대한 최악실행시간 값을 미리 계산해야 하는 단점이 존재한다. 이를 해결하는 좀 더 지능적인 호출함수에 대한 처리가 요구된다. 또한 좀 더 정확한 최악실행시간을 위해 오브젝트 덤프 파일내의 디버그 정보들을 사용하는 연구가 필요하다.

참 고 문 헌

[1] Eclipse Project, http://www.eclipse.org/

[2] P.Puschener and A.Burns, "A review of worst-case execution-time analysis, Real-Time Systems," Guest Editorial 18(2-3), pp. 115-128, May 2000.

[3] A.Colin, I. Puaut, "A modular and retargetable framework for tree-based wcet analysis," In Proc. 13th Euromicro Conference of Real-Time Systems (ECRTS' 01), June 2001.

[4] RTEMS(Real-Time Executive for Multiprocessor Systems), home page: http://www.rtems.com/

[5] A.Colin, I. Puaut, "A modular and retargetable framework for tree-based wcet analysis," In Proc. 13th Euromicro Conference of Real-Time Systems (ECRTS' 01), June 2001.

[6] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, B. Lisper, "Worst-Case Execution Time Analysis of Disable Interrupt Regions in an Commercial Real-Time Operating System," In Proc. 2nd International Workshop on Real-Time Tools (RTTOOLS' 2002), 2002.

[7] Enea, Enea Embedded Technology, homepage: http://www.enea.com

[8] D. Sandell, A. Ermedahl, J. Gustafsson, B. Lisper, "Static Timing Analysis of Real-Time Operating System Code," 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA' 04), Cyprus, 2004.

[9] Alfred V.Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers - Principles, Techniques, and Tools," pp. 528-529, Addison-Wesley, 1988.

[10] Alfred V.Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers - Principles, Techniques, and Tools," p466, Addison-Wesley, 1988.

[11] Javacc Home, https://javacc.dev.java.net/

[12] Intel PXA27x Processor Family Developer's Manual, Intel, 2004.

[13] GNU GCC Extension, http://gcc.gnu.org/extensions.html

박 현 희

2000년 숭실대학교 컴퓨터학부 졸업(학사). 2003년 숭실대학교 대학원 컴퓨터학과 졸업(석사). 2004년~현재 숭실대학교 대학원 컴퓨터학과 박사과정. 관심분야는 실시간 시스템, 내장 시스템, 리눅스 커널

양 승 민

1978년 서울대학교 공과대학 전자공학과 학사. 1978년~1981년 삼성전자㈜ 연구원. 1983년 미국Univ. of South Florida 전산학 석사. 1986년 미국Univ. of South Florida 전산학 박사. 1987년 미국Univ. of South Florida 조교수. 1988년~1993년 미국 Univ. of Texas at Arlington 조교수. 1993년~현재 숭실대학교 컴퓨터학부 교수 겸 (주)엠스톤 대표이사. 관심분야는 Real-Time System, Operating System, Fault-Tolerant System

최 용 훈

2002년 고려대학교 컴퓨터학 학사. 2003년 카네기멜론대학교 소프트웨어공학 석사. 2004년~현재 한국전자통신연구원 임베디드S/W연구단 S/W개발도구연구팀 연구원. 관심분야는 소프트웨어 아키텍처, 실시간 임베디드 시스템, 사용자 경험 연구