임베디드 시스템에서 명령어 기반의 자원 사용 분석 방법

조재황* · 정훈* · 신동하* · 손성훈*

*상명대학교

Instruction Level Resource Usage Analysis Method for Embedded Systems

Jae-hwang Cho* · Hun Jung* · Dong-ha Shin · Sung-hoon Son

*Sangmyung University

E-mail : xio@smu.ac.kr, powerloki@smu.ac.kr

요 약

최근 모바일 컴퓨터 및 임베디드 시스템이 대중화 되면서 전력, 공간, CPU 클럭, 메모리 등과 같은 자원을 효율적으로 사용하기 위한 연구가 많이 진행되고 있다. 기존의 임베디드 시스템 개발에서는 하드웨어 측면의 자원 사용에 대한 연구가 주를 이루어 졌으나 최근 임베디드 시스템에서 소프트웨어의 비중이 커짐에 따라 소프트웨어 측면에서의 자원 사용에 대한 연구가 필요하게 되었다. 본 연구에서는 임베디드 시스템의 자원 사용을 분석하는 새로운 방법인 "명령어 기반의 자원 사용 분석 방법(Instruction Level Resource Usage Analysis Method"을 제안하고 이를 "I-Debugger"라는 도구로 구현하였다. I-Debugger는 프로그램을 명령어 단위의 수행으로 제어하는 디버깅 층(Debugging Layer), 실시간으로 수행되는 명령어에 대한 데이터를 활용 가능한 정보로 변환하는 통계 층(Statistics Layer) 및 분석하고자 하는 응용에 적합하게 정보를 분석하는 분석 층(Analysis Layer)으로 구성된다. 본 연구에서 개발된 I-Debugger를 간단한 문제에 적용한 결과 자원 효율적인 임베디드 시스템 개발에 매우 유용하게 사용될 수 있음을 알 수 있었다.

ABSTRACT

As mobile computers and embedded systems are becoming popular recently, we need to study how to utilize the resources such as power, space, CPU clocks, and memory efficiently. In traditional embedded system development, we were interested in resource usage based on hardware but, as software is becoming more important, we need to study how to analyze the resource usage based on software. In this research, we propose a new method called "Instruction Level Resource Usage Analysis Method" and implement it as a resource usage analysis tool called "I-Debugger". I-Debugger is constructed on three layers: debugging layer which controls the execution of software on instruction level, statistic layer which gathers real-time data and convert to useful information, and analysis layer which generate useful information to specific applications. We have applied the debugger to some simple problem and found that our method is useful in developing resource efficient embedded systems.

키워드

임베디드 시스템, 자원, 명령어, 디버거

Ⅰ. 서 론

모바일 컴퓨터와 임베디드 시스템이 최근 많은 관심을 가지게 되면서 전력, 공간, CPU 클럭, 메모리 등 제한된 자원에 대한 효율적인 사용이 필요하게 되었고. 그에 따른 정확하고 효율적인 자원사용의 측정과 분석 방법이 요구되고 있다[1]. 전통적인 방법에서 측정과 분석 방법은 소프트웨어가 자원 소모에 주는 영향을 배제하고 하드웨어적인 측면에만 치우쳐 있다[1]. 이러한 하드웨어 측면의 측정이나 분석 방법은 소프트웨어에서 사용되는 자원에 대한 분석에 적용하기가 매우 어렵다. 임베디드 시스템에서 효율적인 자원의 사용은 크기, 무게, 전력 등 하드웨어적인 측면에서

뿐만 아니라 소프트웨어에 의해 시스템에서 사용되는 프로세서 사이클, 전원 그리고 메모리 등과 같은 제한된 자원 사용에 대한 분석이 필요하다. 또한 해당 응용에 대한 실시간성, 특정 자원에 대한 최대-최소 사용범위 그리고 요구되는 성능 등과 같은 응용분야의 특성에 대한 고려가 필요하다. 임베디드 시스템에서 중요한 구성 요소 중 하나인 소프트웨어는 그 비중이 점차 커지고 있으며 시스템 전반에 걸쳐 자원을 제어하고 있다[1]. 따라서 전통적인 하드웨어 측면에서의 자원 사용에 대한 측정 및 분석 방법에 반하여 소프트웨어에서의 자원 사용에 대한 방법이 요구된다[1]. 임베디드 시스템은 프로세서와 해당 프로세서에서 수행되는 특정 응용의 소프트웨어에 의해서 그 특성이 결정된다[2][3]. 소프트웨어에서 가장 기초가 되는 것은 프로세서에서 실행되는 개별적인 명령어 단위이다. 로직 게이트가 디지털 하드웨어 회로에서 계산의 기초 유닛이듯이 명령어는 소프트웨어의 기초 유닛으로 생각할 수 있다[1]. 프로세서에 매우 밀접한 특성과 정확하고 세밀한 측정과 분석을 하기 위한 방법으로 "명령어 기반의 임베디드 자원사용 분석 방법"이 고려된다.

명령어 기반에서 자원사용을 분석하는 방법으로는 타깃 시스템의 프로세서를 가상으로 구현하여 소프트웨어를 수행하는 시뮬레이터[4][5]를 이용한 방법, 명령어 수준의 프로파일러(Profiler) 도구를 사용하는 방법[6], 실제 타깃 시스템에서 수행 정보에 대한 로그를 남겨 분석하는 방법 등이 있다. 이러한 방법들은 "명령어 기반의 임베디드 자원사용 분석 방법"에서 매우 유용한 분석 도구들이다. 하지만 임베디드 시스템의 실시간성과 응용의 다양성 등을 고려할 때 몇 가지 요소들에 대한 제약이 있다. 첫째, 실시간으로 데이터를 측정 및 분석하기가 어렵다는 점이다. 대부분 실행시간이 아닌 로그를 통한 분석이나 가상으로 수행하는 방법을 사용하고 있다. 둘째는 실행점에 대한 제어가 매우 어렵거나 불가능하다는 점이다. 시뮬레이터와 프로파일러의 경우에는 어느 정도의 실행점에 대한 제어 방법을 제공하지만 실제 타깃 시스템에서의 가능한 모든 응용에 대한 제어와 상호작용을 기대하기는 힘들다. 마지막으로 측정된 데이터에 대한 정확성과 신뢰성이 떨어진다는 것이다. 시뮬레이터와 프로파일러는 실제 시스템에서 수행되는 것이 아니기 때문에 데이터에 대한 정확성과 신뢰성이 떨어진다. 기존의 시뮬레이터를 사용하여 명령어를 추적하고 분석하는 방법은 시뮬레이터를 어느 레벨에서 구현하는가에 따라 정확도의 차이가 존재한다[4]. 그리고 수행정보에 대한 로그를 남겨 분석 하는 방법의 경우 로그를 남기는 부분이 시스템에 영향을 미치지 않도록 구현하는 것이 매우 어렵다.

임베디드 시스템의 실시간성과 응용의 다양성, 정확하고 세밀한 데이터의 측정, 실제 타깃 시스템에서 수행 제어와 다양한 응용에 대한 상호작용의 제공과 같은 임베디드 소프트웨어의 자원 사용 분석의 특성을 적용하여 명령어 기반에서 효율적으로 자원사용을 측정 및 분석하기 위한 도구로 "I-Debugger(Instruction Measurement Debugger)"를 제안한다. "I-Debugger"는 프로그램을 코드기반으로 제어하기 위한 디버거(Debugger Layer)에 실시간으로 수행되는 명령어에 대한 데이터를 활용 가능한 정보로 가공하여 저장하는 통계 모듈(Statistics Layer)과, 분석하고자 하는 응용에 맞게 정보를 분석하여 원하는 형태의 결과를 출력하는 분석 모듈(Analysis Layer)을 추가하여 구현한다.

II. I-Debugger 분석 방법

I-Debugger는 [그림 I]과 같이 실제 타깃에서 수행되고 있는 소프트웨어의 제어와 명령어 정보를 제공해 주는 디버거층(Deubbger Layer)과 데이터를 활용 가능한 정보로 가공하여 저장하는 통계층(Statistics Layer) 그리고 통계층으로부터 전달 받은 정보를 응용에 맞게 분석 하여 원하는 결과로 출력하는 분석층(Analysis Layer)으로 이루어져 있다.

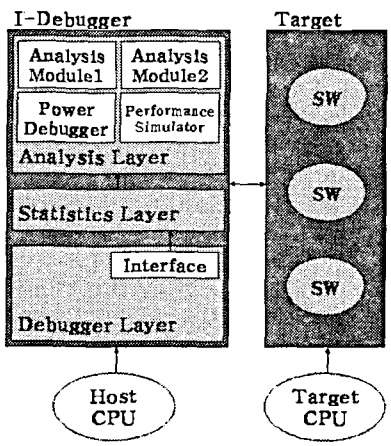

그림 1. I-Debugger의 구성도

I-Debugger의 디버거층에는 리얼뷰와 같은 하드웨어 디버거나 GDB와 같은 소프트웨어 디버거 등 여러 종류의 디버거를 적용 할 수 있다. 이번 연구에서는 디버거층에 GNU 디버거(GDB: GNU Debugger)를 사용하였고 출력층에는 수행 정보에 대한 실시간 출력 방식과 수행 시스템에 로그를 남기는 방식을 사용 하였다. 통계층의 경우에는 수행된 프로세서의 명령어의 개수를 누적하는 누산 방식의 구현을 사용하였다. 실험에 사용한 시

스템의 프로세서는 i686이다. 디버거층에 GNU 디버거를 선택한 이유는 첫째, 원격에서 실시간으로 수행 제어와 상호작용은 물론 명령어 및 코드 수준에서의 수행정보를 제공하고 둘째로 GNU 컴파일러에서 제공하는 C, C++, Java, Fotran 등의 다양한 언어로 개발된 소프트웨어와 x86, Arm, Mips 등의 CPU에 모두 적용이 가능하다는 점, 그리고 마지막으로 GPL(GNU Public License)을 따르기 때문에 비교적 소스 코드를 구하기가 쉽고 소스 코드의 변경과 변경된 코드의 배포가 자유롭다는 점 때문이다[7]. 실제 적용 사례를 "컴파일러의 최적화 정도에 따른 수행 명령어 측정 및 분석"과 "명령어 수준의 알고리즘 평가"를 통하여 소개한다.

III. I-Debugger 적용 사례

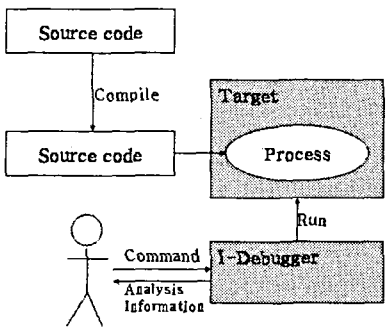

그림 2. I-Debugger의 사용 방법

"컴파일러의 최적화 정도에 따른 수행 명령어 수 측정 및 분석"은 컴파일러의 최적화 옵션을 다르게 주어 최적화 옵션이 어떤 변화를 가져다 주는지를 분석해 봄으로써 명령어 기반에서 소프트웨어 자원 사용에 대한 측정 및 분석이 실제로 어떻게 적용 될 수 있는지를 보여 준다. 분석 방법에서 사용된 컴파일러는 GNU C Compiler이고 아래의 소스 코드를 사용하여 재귀적으로 "팩토리얼 10"을 수행하는 코드를 최적화 옵션을 각각 "-O0"(최적화 하지 않음)와 "-O3"(높은 최적화)을 주어 컴파일하고 I-Debugger를 통해 분석에 필요한 데이터를 측정 및 분석한다. 실험은 [그림 2]에서와 같이 측정 또는 분석하고자 하는 대상이나 응용에 맞는 옵션을 주어 컴파일러를 통해 디버그 옵션을 포함한 실행 파일을 만들고 "I-Debugger"를 통해 타깃 시스템에서 수행 점을 제어하여 원하는 부분에서의 정보를 분석하는 방법으로 진행된다.

```
int fact(int n) {
    if (n > 1) return fact(n - 1) * n;
    else return 1;
}
```

표 1. I-Debugger를 이용한 컴파일러의 최적화 정도에 따른 수행 명령어 수 측정 결과

Instruction	Optimization0	Optimization3
add	12	0
sub	21	10
and	1	1
push	21	28
pop	0	1
mov	40	37
call	10	9
leave	10	17
ret	10	9
shr	1	0
shl	1	0
cmpl	10	9
jle	10	0
jg	0	9
jmp	9	0
dec	9	0
imul	9	8
movl	1	0
total	175	138

[표1]은 I-Debugger를 통해 컴파일러의 최적화 정도에 따른 명령어 수행에 대한 측정 결과이다. 주로 "add", "sub", "dec"와 같은 산술 연산의 명령어 수행이 눈에 띄게 줄어든 것을 볼 수 있다. 전체적으로도 수행 시에 사용된 명령어의 수가 줄어들었다. 컴파일러 최적화를 통한 명령어 수준의 변화에 대한 측정 데이터를 통해 분석된 컴파일러에서의 최적화 특성을 유추해 볼 수 있다. 아주 단순한 프로그램에서도 최적화에 대한 컴파일러의 특성이 명확히 나타나는 것은 매우 흥미로운 점이다.

"명령어 수준의 알고리즘 평가"는 같은 동작을 수행하는 두 알고리즘을 비교 및 평가하는 분석 방법에 I-Debugger를 적용한 사례이다. 이 사례에서는 팩토리얼에 대한 재귀 알고리즘과 순환 알고리즘에 대한 비교 평가를 한다. 먼저 재귀 알고리즘의 경우 앞서 소개한 사례에서 사용한 소스 코드를 사용 하였고 순환 알고리즘의 경우 아래의 소스 코드를 사용하여 평가하였다.

```
int fact(int n) {
    int nIndex, nFact;

    for (nIndex = n; nIndex >= 1; nIndex--)
        nFact *= nIndex;
    return nFact;
}
```

표 2-1. 컴파일러 최적화 수준 0의 순환과 재귀 알고리즘의 명령어 수준의 측정 결과

Instruction	recursion0	iteration0
add	12	3
sub	21	3
and	1	1
push	21	3
mov	40	26
call	10	1
leave	10	11
ret	10	1
shr	1	1
shl	1	1
cmpl	10	11
jle	10	11
jmp	9	10
dec	9	0
decl	0	10
imul	9	10
movl	1	1
total	175	104

표 2-2. 컴파일러 최적화 수준 3의 순환과 재귀 알고리즘의 명령어 수준의 측정 결과

Instruction	recursion3	iteration3
sub	10	2
and	1	1
push	28	1
pop	1	0
mov	37	2
call	9	0
leave	17	1
ret	9	0
cmpl	9	0
jg	9	10
dec	0	10
test	0	10
imul	8	0
total	138	37

[표2-1]은 컴파일러 최적화 수준 0에서의 순환과 재귀 알고리즘의 측정 결과이고 [표2-2]는 최적화 수준 3에서의 측정 결과이다. 명령어 수준의 측정 결과를 분석해 보면, 재귀 알고리즘에 비해 순환 알고리즘이 효율적인 것을 알 수 있다. 컴파일러의 최적화가 이루어진 재귀 알고리즘의 측정 결과보다도 최적화가 이루어지지 않은 순환 함수의 측정 결과가 전반적으로 낮은 것을 볼 수 있다. 그리고 무엇보다도 흥미로운 점은 순환 알고리즘이 컴파일러 최적화를 수행한 뒤의 결과이다. 이 결과를 통해 알고리즘이 컴파일러의 최적화에도 많은 영향을 미친다는 것을 알 수 있다.

IV. 결 론

본 논문에서는 임베디드 시스템에서 명령어 기반으로 하여 자원사용을 분석하는 방법으로써 I-Debugger를 제안하였다. 그리고 "컴파일러의 최적화 정도에 따른 수행 명령어 수 측정 및 분석"과 "명령어 수준의 알고리즘 평가"를 통해 I-Debugger가 실제로 적용된 사례를 보았다. 이와 같이 명령어 기반에서 자원사용을 분석하는 방법이 소프트웨어를 효율적으로 검증하는 도구가 될 수 있다.

이번 사례에서는 명령어 수에 대해서만 평가하였으나 CPU 사이클, 전력 소비량 측정 등을 쉽게 할 수 있다. 앞으로의 연구에서는 다양한 프로세서와 임베디드 시스템에 이 방법을 적용한다. 나아가서는 실제 응용에서의 측정 및 분석 방법을 제시하고 해당 응용의 개선과 최적화 방안을 연구한다.

참고문헌

[1] Vivek Tiwari, Sharad Malik, Andrew wolfe and Mike Tien-chien Lee, "Instruction Level Power Analysis and Optimization of Software", Journal of VLSI Signal Processing, 1-18, 1996.

[2] Vivek Tiwari, Sharad Malik and Andrew wolfe, "Power Analysis of Embedded Software A First Towards Software Power Minimization", 1994.

[3] Yau-Tsun Steven Li and Sharad Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", IEEE Transactions on Computer-Aided Design of Integrated Circuits and System, VOL 16, NO 12, December 1997.

[4] Akshaye Sama, M.Balakrishnan and J.F.M.Theeuwen, "Speeding up Power Estimation of Embedded Software", 2000.

[5] Robert F.Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", Technical Report UWCSSE 93-06-06.

[6] Kris Kaspersky, "Code Optimization Effective Memory Usage", A-LIST, 2003.

[7] Richard Stallman, Roland Pesch, Stan Shebs, et al, "Debugging with GDB", Free Software Foundation, 2005.