인공위성 소프트웨어 타이밍 분석

이종인° 최종욱 이재승 강수연
한국항공우주연구원
{jilee°, jwchoi, jslee, sykang}@kari.re.kr

Timing Analysis for Satellite Flight Software

Jong-In Lee° Jong-Wook Jae-Seung Lee Soo-Yeon Kang
Satellite Electronics Dept., Korea Aerospace Research Institute

요 약

인공위성 탑재 소프트웨어는 정해진 시간 내에 필요한 작업을 수행하여야 하는 실시간 내장형 소프트웨어로 타이밍 분석이 중요하다. 기존의 인공위성소프트웨어 개발 시 적용되는 타이밍 분석기법은 개발자의 수작업에 의존하여 많은 시간과 노력이 요구되며 정확성에 문제가 있을 수 있는 단점이 있었다. 본 논문에서는 위성소프트에어의 타이밍 분석에 적용 가능한 최장 실행시간 (Worst Case Execution Time, WCET) 기법을 조사하고 보다 정확한 (tight) WCET를 구하기 위해 입력 데이터를 고려한 WCET 분석 방안을 제안한다.

1. 서 론

인공위성 탑재소프트웨어 (flight software)는 궤도상에서 운영 중의 오류 발생을 줄이기 위해 예정된 작업을 정해진 시간 내에 순서대로 수행하는 동기방식으로 설계되는 경우가 많으며 (synchronous, deterministic scheduling) 지상 명령의 수신 처리와 같은 비 동기적인 이벤트도 처리하여야 한다. 또한 발사 후 예측치 못한 하드웨어 또는 소프트웨어의 오류에 기인한 소프트웨어 수정 (code patch)을 위해서는 여분의 메모리 영역이 확보되어 있어야 한다. 이를 위해 위성소프트웨어 개발 시 주요 단계별로 소프트웨어 타이밍 및 사이징을 (software timing and sizing)을 정기적인 추정 (estimation) 또는 실측 (measurement)을 통하여 분석하는 것이 중요하며 위성발사 시에 완성된 소프트웨어의 크기와 수행시간이 margin을 고려한 메모리 및 CPU 이용률 (memory/throughput usage) 요구사항을 만족하여야 한다. 그러나 실제로 소프트웨어를 수행시켜 정확한 WCET를 측정하는 것은 어려우며 대부분의 경우 예측치는 실현가능하지 않은 경우까지 포함하게 되어 과대평가 (overestimation)되는 경우가 많다. 과대평가된 CPU 이용률 (throughput)이 요구사항에 정의된 값을 초과할 경우 이를 해소하기 위한 소프트웨어 알고리즘의 변경이 필요하고 이에 따른 전체 시스템의 성능 저하가 초래될 수 있으므로 정확한 타이밍 분석은 매우 중요하다. 본 논문에서는 기존의 위성제어소프트웨어의 타이밍 분석 방법을 소개하고 문제점을 살펴본 후 개선된 WCET 분석방법을 제시하고자 한다. 2절에서는 기존의 위성제어 소프트웨어 타이밍 분석 방식을 소개하고, 3절에서는 일반적으로 실시간 내장형 소프트웨어의 타이밍 분석에 사용되는 WCET 방식을 알아보고, 4절에서는 위성제어소프트웨어의 타이밍 분석에 적용할 수 있는 자동적인 WCET 분석방안을 제시한 후, 5절에서는 결론 및 향후 연구방향을 제시한다.

2. 위성제어소프트웨어의 타이밍 분석방식

위성제어소프트웨어의 타이밍 분석을 위해 일반적으로 사용되는 방법은 개발자가 모듈별로 가능한 프로그램 실행 path를 파악하고 각 path를 지나도록 test case (input data)를 찾아내어 프로세서 시뮬레이터 등의 도구를 이용하여 실제로 path를 따라서 수행되는 코드의 instruction cycles 수를 합하여 계산한다[1]. 계산된 path중 가장 수행시간이 긴 path를 해당 모듈의 WCET로 선택한다. (그림 1 참조)

```
 1 2 3 4 5 6 7 8 9 A
 | | | | | | | | | |  0162:
 | | | | | | | | | |  0163: /*****************************************************************
 | | | | | | | | | |  0164: *  Processes CODA messages from ECU (only when CODA data is received)   *
 | | | | | | | | | |  0165: *****************************************************************/
 \ \ \ \ \ \ \ \ \ \  0166:
 | | | | | | | | | |  0167:                                    /* check CODA data is received    */
 |v|v|v|v|v| | | | |  0168:    if (KFS_ecu_coda_count != 0) {
 |v|v|v|v|v| | | | |  0169:       KFS_ecu_coda_count = 0,
 |v|v|v|v|v| | | | |  0171:       if(KFS_ecu_coda.coda_crc ==
 | | | | | | | | | |  0172:             util_crc_calc ((unsigned char *) &KFS_ecu_coda.as_in,
 \ \ \ \ \ \ \ \ \ \  0173:                KFS_ECU_CODA_LEN-2, accum)) {
 | | | | | | | | | |  0174:                                    /* copy ECU coda from CDA to CODA  */
 |v|v|v|v| | | | | |  0177:          OBC_coda.tcvl_timid = KFS_ecu_coda.tcvl_timid,
 |v|v|v|v| | | | | |  0179:          OBC_coda.tcvl_aggr = KFS_ecu_coda.tcvl_aggr;
 | | | | | | | | | |  0182:                                    /* copy set entry into CODA page   */
 |v|v|v|v| | | | | |  0183:          if((KFS_ecu_coda.error_count > 0) &&
 \ \ \ \ \ \ \ \ \ \  0184:             (KFS_ecu_coda.error_count < TFS_ERROR_TBL_LEN)) {
 | | | | | | | | | |  0185:                                    /* copy error count into CODA page */
 |v|v| | | | | | | |  0186:             MMD_coda_page.ecu_error_count = KFS_ecu_coda.error_count,
 \ \ \ \ \ \ \ \ \ \  0187:                                    /* copy set entry into CODA page   */
 |v|v| | | | | | | |  0188:             if((KFS_ecu_coda.set_index >= 0) &&
 | | | | | | | | | |  0189:                (KFS_ecu_coda.set_index < KFS_ecu_coda.error_count))
 |v| | | | | | | | |  0190:                MMD_coda_page.ecu_set_error[KFS_ecu_coda.set_index] =
 | | | | | | | | | |  0191:                   KFS_ecu_coda.ecu_err_info,
 | | | | | | | | | |  0192:          }
 \ \ \ \ \ \ \ \ \ \  0193:       }
 | | | | | | | | | |  0194:       else {
 | | | |v| | | | | |  0195:          util_error_handler (ERR_EXE_ECU_CODA_CRC, "").
 \ \ \ \ \ \ \ \ \ \  0196:       }
 | | | | | | | | | |  0197:    }
 | | | | | | | | | |  0198:
```

그림 1. 위성소프트웨어 모듈의 Timing Estimation 사례

상위 프로그램의 경우도 마찬가지로 가장 수행시간이 긴 path를 지나는 test case를 찾아서 수행시간을 계산하며 bottom-up 방식으로 자신이 호출하는 하위 모듈의 WCET를 모두 합해 계산한다. 그러나 이러한 개발자에 의한 수동적인 방법은 많은 시간과 노력이 소요되며 모듈간의 데이터 의존관계 (data dependency)를 고려하여야 하므로 오류가 발생할 여지가 많아 결과의 정확도가 떨어지는 단점이 있다.

3. WCET 분석방법 및 문제점

WCET 분석방식은 실제로 프로그램을 실행시키지 않고 정적 분석 (static analysis)을 통하여 최장 수행시간을 미리 예측하는 방식이다[2,3,4]. WCET 분석방법은 그림 2와 같이 프로그램 흐름분석, 저 수준 분석 및 계산의 단계로 이루어진다.

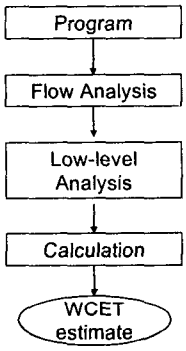

그림 2. WCET 분석단계

프로그램 흐름 분석은 loop의 최대 수행횟수, 입력변수의 의존성, 함수 호출, 등의 프로그램의 동적인 행위에 대한 분석으로 사용자의 annotation 또는 자동적인 분석을 통해 이루어지며 일반적으로 Control Flow Graph (CFG)로 나타낸다. 저 수준 분석은 프로그램 흐름 분석에서 추출된 각 기본 블록(basic block) 들의 수행시간을 실제 하드웨어 환경에서의 object code level에서의 수행 시간을 계산하는 것이다. Cache, pipeline 등의 프로세서의 특성을 고려하여야 한다. 계산은 프로그램 흐름분석 및 저 수준 분석결과를 토대로 WCET를 구하는 단계로 tree-based 기법, path-based 기법, Implicit Path Enumeration Technique (IPET), 등이 있다.

3.1 Tree-based 방식

프로그램 코드를 분석하여 control flow tree를 작성한 후 각 node의 수행시간을 그림 3과 같이 bottom-up 방식으로 단말 노드에서 root node로 단계적으로 계산하는 방식이다[5,6].

*if node*의 경우, 자신의 수행시간에 child nodes 중 가장 큰 수행시간을 더하며, loop의 경우 child nodes의 수행시간을 모두 더한 시간에 자신의 수행시간을 더한 후, 그 결과에 loop bound를 곱하게 된다.

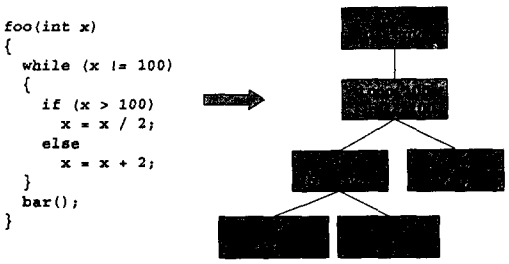

그림 3. Tree-Based 방식

3.2 Path-based 방식

프로그램 소스코드를 분석하여 CFG로 변환한 후 graph search algorithm을 사용하여 가장 수행 시간이 긴 path를 구하는 방법이다[7]. 이 방식은 tree-based 방식보다 tight한 WCET를 구할 수 있다.

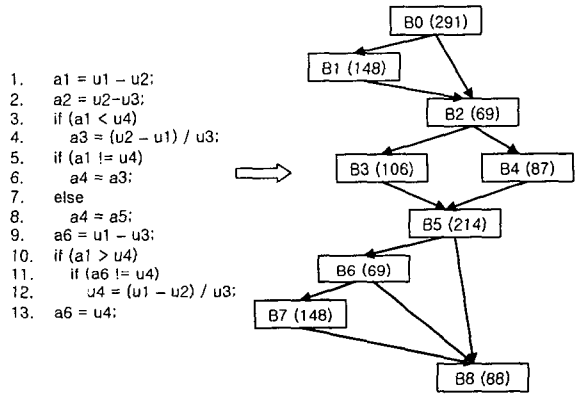

그림 4. Path-Based 방식

3.3 IPET 방식

프로그램의 CFG에서 도출되거나 사용자가 제공한 제약조건 (structural, functional constraint)을 이용하여 모든 제약조건을 만족하는 objective function내의 각 블록의 실행횟수와 수행시간의 곱을 더한 최대값에 의해 계산된다[8,9,10].

$$WCET = MAX(\sum_{i}^{N} t_i x_i)$$

$$where\ x_i\ is\ the\ execution\ count\ of\ basic\ block\ B_i$$

$$t_i\ is\ the\ execution\ time\ of\ basic\ block\ B_i$$

IPET에서 각 블록의 수행횟수는 Integer Linear Program (ILP) 또는 Satisfiability solving 기법을 이용하여 구한다. IPET 방식은 모든 경로를 explicit하게 탐색하지 않아 다른 방법에 비해 powerful 하나 path를 implicit하게 다루어 어떤 block이 몇 번 수행되었나는 알 수 있으나 정확한 경로는 알 수 없는 차이점이 있다. 그림 5는 IPET에서 CFG로부터 구조적인 제약조건 (structural constraints)을 구

하는 것을 보여준다.

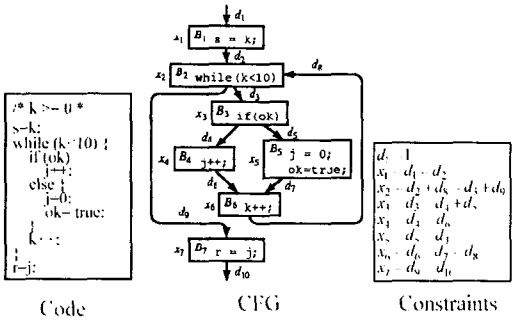

그림 5. IPET 방식의 구조적 제약조건

4. 제안된 WCET 분석방법

그러나 위에서 제시한 WCET 분석방법들은 입력 데이터에 따라 최장수행시간이 달라질 수 있는 경우에는 프로그램이 실행될 때 까지는 알 수 없으며 각 path 사이의 functional constraints를 자동으로 추출하기 어려운 단점이 있다.

이러한 문제점을 해결하기 위해 본 논문에서는 제안하는 WCET 분석 방법은 그림 6과 같이 Path-based WCET 분석방식과 자동 테스트 데이터 생성기법을 결합하는 것이다.

Path-based WCET 기법은 프로그램의 CFG로부터 최장실행시간 경로를 찾아낸다. 자동 테스트 데이터 생성은 선택된 최장실행시간 경로에 대하여 path predicate를 찾고 이를 이용하여 입력변수 (input variables)에 대하여 해를 찾는 것이다. 찾은 해는 가장 긴 수행시간 경로를 따라가기 위해 필요한 입력변수를 기술하는 등식/부등식의 시스템으로 표시된다.

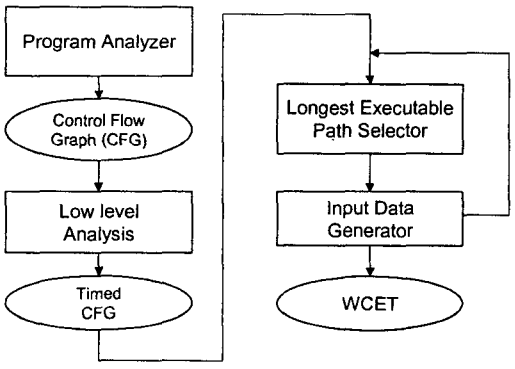

그림 6. 새로운 WCET 분석방법

5. 결 론

실시간으로 위성을 제어하는 위성탑재소프트웨어의 타이밍 분석에 사용되는 기존의 방법을 소개하고 문제점을 살펴본 후 최장실행시간을 찾는 세 가지 WCET 분석기법들을 (tree-based, path-based, IPET) 살펴보았다.

WCET 분석기법은 계산된 WCET 값이 실제로 가능한 최장실행시간을 과소평가 않아야 하고 (safeness) 가능한 한 실제 최장실행시간에 근접하여야한다 (tightness). 또한 사용자의 수동적 개입을 줄여 자동화될 수 있어야 하며 기존의 코드에 수정 없이 적용할 수 있어야 한다. 또한 WCET분석기법은 입력 데이터에 따라 WCET가 달라 질 수 있으므로 더욱 tight한 WCET를 구하기 위해서는 입력 데이터를 고려하여야 한다. 본 논문에서는 입력 데이터를 고려한 최장실행시간 분석기법을 제안하였다.

향후 loop bound, 함수호출, data dependency 등의 functional constraints를 소스코드로부터 자동으로 추출해 내고 predicate abstraction 등의 abstraction 기법을 이용하여 실제 복잡한 프로그램의 타이밍 분석에 적용할 수 있는 WCET 분석기법에 대한 연구를 진행할 계획이며 궁극적으로는 이러한 연구결과를 이용하여 특정한 하드웨어 구조 (processor architecture)에 맞는 WCET 도구를 개발하고 이를 일부 위성제어소프트웨어의 타이밍 분석에 적용하여 적합성 여부를 검증하는 것을 목표로 하고 있다.

참고문헌

1. KOMPSAT-2 Flight Software Timing & Sizing Report, KARI, 2003
2. Alan C. Shaw, "Reasoning About Times in High-Level Language Software," IEEE TSE Vol. 15, No. 7, July 1989
3. C. Y. Park, Alan C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," IEEE Computer, May 1991
4. Peter P. Puschner, Ch. Koza, "Calculating the Maximum Execution Time of Real-Time Programs, J. Real-Time Systems," Vol. 1, No. 2, Sept. 1989, pp. 159-176
5. P. Puschner, Ch. Koza, "Calculating the Maximum Execution Time of Real-Time Programs," The Journal of Real-Time Systems, Vol. 1, No. 1, 1989, pp.159-176
6. S.S. Lim, Y.H. Bae, et al., "An Accurate Worst-Case Timing Analysis for RISC Processors," IEEE TSE, Vol. 21, No. 7, July 1995, pp.593-604
7. F. Stappert, P. Altenbernd, "Complete Worst-Case Execution Time Analysis of Straight-Line Hard Real-Time Programs," Journal of Systems Architecture, 46(4), 2000, pp. 339-355
8. Peter P. Puschner, Anton V. Schedl, "Computing Maximum Task Execution Times - A Graph-Based Approach," Proc. Of IEEE Real-Time Systems Symposium, Vol. 13, Kluwer Academic Publishers, 1997, pp. 67-91
9. Y. S. Li, S Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," Proc ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems, pp. 95-105, La Jolla, CA, USA, June 1995
10. Jakob Engblom, Andreas Ermedahl, "Modeling Complex Flows for Worst-Case Execution Time Analysis," Proc. IEEE Real-Time Systems Symposium, Nov. 2000
11. 코드기반 다목적실용위성 2호 소프트웨어 검증기법연구 보고서, 한국과학기술원, 2003