논문 21-46-04-20 The Journal of Korean Institute of Communications and Information Sciences '21-04 Vol.46 No.04
https://doi.org/10.7840/kics.2021.46.4.757

실시간 임베디드 시스템 환경에서 RM 스케줄러를 이용한 태스크 수준의 메모리 결함 허용 기법

김 범 식•, 양 회 석°

Task-Level Memory Fault-Tolerance Technique Using an RM Scheduler on Real-Time Embedded Systems

Beomsik Kim•, Hoeseok Yang°

요 약

본 논문에서는 인공위성에서 사용되는 RTEMS/SPARC 환경에서 메모리에 발생하는 결함을 태스크 수준에서 소프트웨어적으로 감내하는 기법을 구현하고 이를 보고한다. 일반적인 시스템에서 하드웨어적으로 감내 불가능한 결함은 소프트웨어적으로도 처리하기 어려우므로 watchdog 타이머를 이용한 시스템 재부팅으로 처리한다. 이러한 방식은 긴 기능 중단을 초래하여 안정중심 시스템의 실시간 운용성을 저해한다. 본 논문에서는 실시간성을 유지하며 결함을 감내하기 위하여 SPARC(Scalable Processor ARChitecture) 기반 프로세서에 이식된 RTEMS(Real-Time Executive for Multiprocessor Systems) 운영체제의 RM(Rate-Monotonic) 스케줄러를 수정하고, 결함 주입, 결함 검출, 결함 정정 기법을 구현하여 태스크 수준에서 SRAM 메모리의 결함을 감내할 수 있음을 확인하였다. 또한, 비용분석을 위하여 구현기법에 따른 태스크의 최악실행시간과 CPU 이용률을 측정하고, 이를 분석하였다.

키워드 : 실시간 운영체제, 임베디드 시스템, 태스크 스케줄링, 신뢰성, 결함 허용
Key Words : Real-time operating systems, embedded system, task scheduling, reliability, fault tolerance

ABSTRACT

In this paper, we implement and report a software-based technique that tolerates memory faults at task level in the RTEMS/SPARC environment used in satellites. Since it is not trivial for software to handle the uncorrectable faults by hardware, it is common to handle them by means of rebooting using a watchdog timer. Such a rebooting approach causes suspensions of mission or functionality for a substantial amount of time, jeopardizing the real-time operation of the safety-critical system. In this work, we propose to tolerate memory faults in SRAM at task level by modifying the Rate-Monotonic (RM) scheduler of the RTEMS (Real-Time Executive for Multiprocessor Systems) operating system, ported to a SPARC(Scalable Processor ARChitecture)-based processor. We implemented fault injection, fault detection, and fault correction techniques in the RTEMS/SPARC environment. In addition, in order to show the cost effectiveness of the proposed implementation, the WCET (worst-case execution time) of the tasks and the CPU utilization have been measured and analyzed.

※ 본 연구는 과학기술정보통신부 및 정보통신기획평가원의 대학ICT연구센터지원사업의 연구결과로 수행되었음 (IITP-2021-2018-0-01424)
• First Author : Ajou University Department of Electrical and Computer Engineering, asd326541@ajou.ac.kr, 학생회원
° Corresponding Author : Ajou University Department of Electrical and Computer Engineering, hyang@ajou.ac.kr, 정회원
논문번호 : 202012-316-D-RE, Received December 14, 2020; Revised January 23, 2021; Accepted January 27, 2021

Ⅰ. 서 론

임베디드 시스템에 사용되는 마이크로프로세서의 트랜지스터 집적도가 높아지고, 동작 전압이 낮아지면서 소프트에러(soft-error)의 발생 가능성이 커지고 있다. (소프트에러는 single-event upsets(SEUs) 또는 일시적인 오류라고도 불린다) 소프트에러란 반도체에 저장된 비트값이 일시적으로 0에서 1로 또는 1에서 0으로 바뀌는 현상을 말하며, 주로 중성자인 우주 방사선(cosmic rays)이나 실리콘 및 패키지의 방사성 불순물에서 방출되는 알파 입자(alpha particle)에 의해 발생하는 것으로 알려져 있다[1].

높은 신뢰성이 요구되는 항공, 위성, 자동차, 그리고 의료기기 등의 안전중심(safety-critical) 시스템에서 소프트에러를 감내하지 못한다면 오작동으로 인해 재산 또는 인명 피해와 같은 심각한 결과를 초래할 수 있다. 따라서 기능 안전 표준(자동차 분야의 ISO-26262[2])에서는 소프트에러의 영향을 고려한 전자 시스템 설계를 요구한다.

소프트에러를 감지, 감내하기 위해 하드웨어와 소프트웨어를 이용한 다양한 보호, 강화 기법들이 적용되고 있다. 가장 일반적인 방법 중 하나는 삼중화(TMR, Triple Modular Redundancy)이다[3-5]. 삼중화에 의해 보호되는 회로는 원래 회로의 복사본 3개와 majority voter로 구성된다. 3개의 복사본 중 2개가 올바르게 작동한다면 시스템은 3개의 복사본 중 하나에서 발생하는 오류를 감내할 수 있다. 하지만 두 개 이상의 복사본에서 오류가 발생한다면 삼중화만으로 오류를 감내할 수 없기 때문에 field programmable gate array(FPGA)에서 configuration 메모리를 주기적으로 새로 고침하는 메모리 스크러빙이 함께 적용되기도 한다[6,7]. 자동차 도메인에서는 electronic control units(ECUs) 수준에서 삼중화와 majority voter를 적용하거나 microcontroller unit(MCU) 수준에서 lockstep 멀티 코어와 watchdog timer를 이용한 오류 감내 시스템이 고안되고 있다[8, 9]. 또한 메모리에 해밍부호, Bose-Chaudhuri-Hocquenghem(BCH)부호, 리드-솔로몬 부호와 같은 에러 정정 코드(ECC, Error-Correction Code)를 적용하기도 한다[10]. 하드웨어를 이용한 보호기법들은 추가적인 하드웨어 오버헤드가 필요하다는 단점이 있다.

소프트웨어를 이용한 강화 방법은 재실행(re-execution), replication, 그리고 checkpointing이 대표적이며 이들은 추가적인 연산시간 오버헤드가 요구된다[11-13].

소프트웨어 운영체제의 태스크 수준에서 소프트에러를 감지, 감내하기 위한 대부분의 연구는 이론적인 모델링 위주이며 실제 산업에서 사용되는 하드웨어와 실시간운영체제 환경에서 직접 구현하고, 검증하는 실용적인 사례는 거의 없다. 또한, 고칠 수 없는 결함은 소프트웨어적으로 유연하게 처리하기보다 watchdog timer를 이용한 시스템 재부팅으로 감내하는 것이 일반적이다[14]. 이때 watchdog timer를 이용한 시스템 재부팅은 시간 오버헤드가 굉장히 크기 때문에 실시간 제약 조건이 존재하는 안전중심 시스템에서 예기치 않은 결과로 이어질 수 있다.

[15]에서는 소프트에러의 발생확률이 굉장히 낮고, 주기적인 제어 태스크가 소프트에러를 어느 정도 감내할 수 있다는 특성으로부터 RM(Rate-Monotonic) 스케줄러, (m,k)-패턴, 그리고 (m,k)-제약을 이용하여 모든 태스크에 보호 기법을 적용하지 않으면서 제어 성능의 큰 저하 없이 시스템 이용률을 줄일 수 있는 FTS(Fault Tolerant Scheduler) 기법을 제안했다. [16]에서는 유럽 우주국(ESA, European Space Agency)과 [15]의 FTS를 인공위성에서 주로 사용되는 RTEMS(Real-Time Executive for Multiprocessor Systems) 실시간 운영체제에서 RM 스케줄러를 수정하고, 위성용 프로세서에서 구현하고자했다. 하지만 수정한 RTEMS 커널과 FTS 알고리즘에 많은 오류가 존재했고, 가장 중요한 태스크 수준에서의 결함 주입, 결함 검출, 그리고 결함 정정을 구현 및 검증하지 못했다.

본 논문에서는 [15,6]를 기반으로 실제 인공위성에서 사용되는 RTEMS 실시간 운영체제와 SPARC (Scalable Processor ARChitecture) 계열의 위성용 프로세서 개발환경에서 RM 스케줄러 커널을 수정하고, 태스크 수준에서 결함 주입, 결함 검출, 그리고 결함 감내 기법을 구현하였다. 실험결과 태스크 수준에서 SRAM 메모리에 주입한 결함을 감내할 수 있었고, 하드웨어와 소프트웨어를 이용한 결함 검출, 결함 정정 기법에 따른 태스크들의 최악실행시간을 측정하고, CPU 이용률을 분석했다.

Ⅱ. 본 론

2.1. 하드웨어 아키텍처

본 논문에서는 향후 국내 인공위성에 탑재될 가능성이 높은 Cobham Gaisler의 GR712RC 개발 보드를 사용했다[17]. GR712RC는 열악한 우주 환경에서 높은 신뢰성을 제공하기 위해 여러 하드웨어 보호 및 방사

선 내성 패키징으로 설계되었다[10]. 180nm CMOS 공정의 듀얼 코어 32-비트 LEON3-FT SPARC V8 프로세서 기반으로 동작 주파수는 80MHz이고, 동작 전압은 1.8V이다. 또한 부동 소수점 연산을 위한 IEEE-754 부동 소수점 장치(floating-point unit, FPU)를 가지고 있다[10]. 본 논문에서는 싱글 코어(CPU0)만을 사용하고, 다른 코어(CPU1)는 power-down 모드로 설정했으며 디버깅과 모니터링에는 LEON 프로세서를 위한 GRMON 디버거[18]를 사용했다.

2.2 결함 허용 스케줄링

멀티 태스크 셋은 $W=\{\tau_1,\tau_2,...,\tau_N\}$와 같이 정의되며 태스크 τ_n는 $(wcet_n,p_n)$의 튜플로 정의되고, $wcet_n$은 최악실행시간(WCET, worst-case execution time) 그리고 p_n은 주기를 의미한다. 태스크의 마감시간(deadline)은 주기와 같다.

태스크 연산은 결함 검출, 결함 정정의 적용 여부에 따라서 세 가지 버전으로 구분된다. Basic 태스크($\tau_{n,B}$)는 연산만 수행하고, Detection 태스크($\tau_{n,D}$)는 연산 후 결함을 검출하고, Correction 태스크($\tau_{n,C}$)는 연산 후 결함을 정정하며, 각각의 최악실행시간은 $wcet_{n,B}$, $wcet_{n,D}$, 그리고 $wcet_{n,C}$으로 표기되며 이를 비교하면 $wcet_{n,C}>wcet_{n,D}>wcet_{n,B}$와 같다.

(m,k)-패턴은 태스크 스케줄러가 어떤 버전의 태스크를 실행할지의 계획을 의미하며 태스크 τ_n의 (m,k)-패턴은 다음과 같다. $\Phi_n=(\phi_{n,0},\phi_{n,1},...,\phi_{n,(k-1)})$ 이때 ϕ는 0 또는 1이고, $\sum_{i=0}^{k-1}\phi_{n,i}=m$이다. (m,k)-패턴에 따르면 k번의 연속적인 태스크 실행 중에서 m번의 강화된 태스크를 실행하며, 즉 $\phi=0$에서는 기본적인 연산 태스크(τ_B)의 실행 그리고 $\phi=1$에서 강화된 태스크(τ_D 또는 τ_C)의 실행을 의미한다.

패턴에는 잘 알려진 E-패턴과 R-패턴의 두 가지 유형이 있다[19,20]. E-패턴은 "0"과 "1"이 균등하게 분포되는데, 만약 $(4,8)$-패턴이라면 $(0,1,0,1,0,1,0,1)$이다. 반면에 R-패턴은 "0"과 "1"이 각각 그룹화되어 $(4,8)$-패턴에서 $(0,0,0,0,1,1,1,1)$과 같다.

(m,k)-패턴에 따른 태스크 스케줄링 방법은 정적(Static) 기법과 동적(Dynamic) 기법 두 가지로 나눌 수 있다. 정적 기법은 미리 정해진 (m,k)-패턴을 그대로 따르며 태스크를 스케줄링하고, 이때 SRE(Static Reliable Execution)는 $\phi=0$에서 τ_B, $\phi=1$에서 τ_C를 실행하며 SDR(Static Detection and Recovery)은 $\phi=0$에서 τ_B, $\phi=1$에서 τ_D를 실행하고, τ_D에서 결함이 감지되면 τ_C를 이어서 실행한다. (m,k)-패턴이 $(0,1,0,1,1)$ 일 때 SRE와 SDR에서의 태스크 스케줄링 간트 차트를 그림 1에서 확인할 수 있다.

동적 기법은 정적 기법과 다르게 미리 정해진 (m,k)-패턴을 경우에 따라 수정하여 태스크를 실행한다. 이때 동적 기법의 DRE(Dynamic Reliable Execution)는 $\phi=0$에서 τ_D, $\phi=1$에서 τ_C를 실행하며 DDR(Dynamic Detection and Recovery)은 $\phi=0$과 $\phi=1$에서 모두 τ_D를 실행하며, $\phi=1$에서 실행한 τ_D가 결함을 감지하면 τ_C를 이어서 실행한다.

(m,k)-제약은 태스크의 연속적인 k번 실행 중에서 m번을 결함 없이 실행해야하는 제약이다. 이때 제어 태스크는 제어 성능의 큰 저하 없이 결함을 어느 정도 감내할 수 있다는 특성을 바탕으로 (m,k)-제약을 위반하지 않는다면 (m,k)-패턴에 따른 태스크 스케줄링을 최대한 늦춰서 오버헤드가 가장 큰 τ_C의 실행을 최소화하고, CPU 이용률을 낮추게된다[15]. 본 논문에서는 미리 정해진 (m,k)-패턴을 따라 태스크를 스케줄링하는 정적기법 중 SDR 기법만을 적용하였다.

2.3 하드웨어 결함 감지 및 정정

GR712RC는 그림 2에 도시된 것과 같이 하드웨어

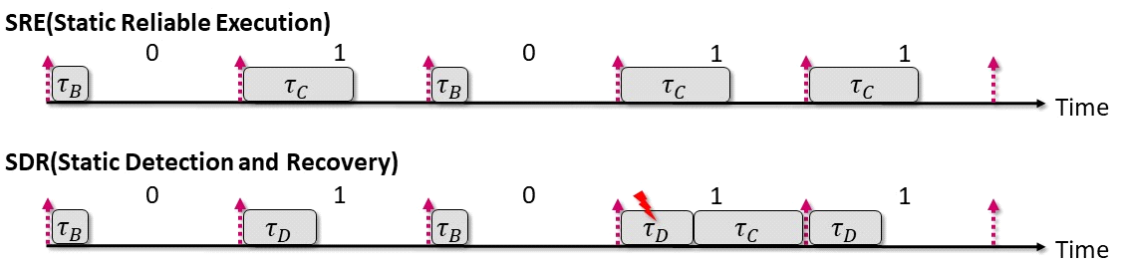


그림 1. (m,k)-패턴이 $(0,1,0,1,1)$ 일 때 정적 기법의 SRE와 SDR에서 태스크 스케줄링 간트 차트.
Fig. 1. Gantt chart of task scheduling in SRE and SDR of static technique when the (m,k)-pattern is $(0,1,0,1,1)$.

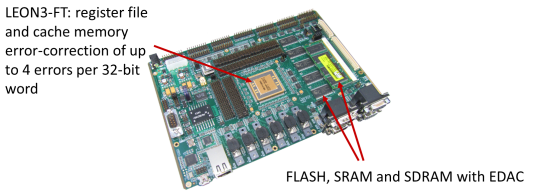


그림 2. Cobham Gaisler의 온보드 컴퓨터 GR712RC 개발 보드: Ramon Chips Ltd의 RadSafeTM 기술로 방사선에 강화되게 설계됨. RadSafeTM의 하드웨어 강화 기술에는 총 이온화 선량(TID), 오류 수정 기능이 있는 flip-flop/SRAM/DRAM, 그리고 데이터 버스의 체크 비트 기반 오류 수정이 포함됨.
Fig. 2. Target on-board computer, GR712RC development board manufactured by Cobham Gaisler: It employs radiation-hard-by-design methods from the RadSafeTM technology from Ramon Chips Ltd. The hardware hardening techniques in RadSafeTM include TID (total ionizing dose), flip-flop/SRAM/DRAM with error-correction, and checkbits-based error-correction in data bus.

를 이용한 결함 감지, 정정 기능을 제공한다. FTMCTRL 메모리 컨트롤러는 SRAM에 선택적으로 BCH EDAC(Error Detection and Correction)의 적용 여부를 설정할 수 있다[10]. 32-비트 word와 7-비트 체크섬을 이용하고, 1-비트 결함의 정정, 2-비트 이상 결함의 감지가 가능하며, MCFG3 레지스터의 TCB에 체크섬이 저장된다[10]. 결함 종류에 따라서 AHB status register의 NE(New Error), CE(Correctable Error) 비트가 설정되며, 정정 불가능한 결함일 경우 AMBA ERROR 인터럽트가 발생하고, *data_access_exception trap(tt=0x09)*이 생성되며 프로세서가 정지된다. 해당 trap에 대한 핸들러는 구현되어 있지 않으며, 소프트웨어적으로 결함을 처리하기 위해선 시스템이 종료하지 않아야 하므로 섹션 3.2에서 이를 해결하기 위한 구현을 다룬다.

결함 감지: 하드웨어 EDAC의 결함 검출 정보는 GR712RC의 AHB status register 2개의 비트에 저장된다. 정정이 가능한 결함(1-비트)인 경우 NE=1, CE=1로 설정되고, 정정이 불가능한 결함(2-비트 이상)인 경우 NE=1, CE=0으로 설정된다[10]. 태스크 수준에서 결함을 검출하기 위해선 태스크를 실행할 때마다 해당 레지스터를 읽어야하며, 이는 추가적인 연산 오버헤드를 가진다. 하드웨어 결함 감지를 이용한 태스크 연산은 $\tau_{D_{HW}}$로 표기하고, τ_B와의 최악실행시간 비교는 $wcet_{D_{HW}} > wcet_B$와 같다.

결함 정정: 정정이 가능한 결함(1-비트)은 하드웨어 EDAC에 의해 자체적으로 처리되기 때문에 하드웨어를 이용한 결함 정정은 태스크 실행 시간에 오버헤드를 더하지 않는다. 하지만 2-비트 이상의 MBE(Multi-bit per word error)는 하드웨어 EDAC로 처리할 수 없고, 다음 섹션에서 설명하는 소프트웨어를 이용한 재실행으로 처리할 수 있으며 $\tau_{C_{HW/SW}}$로 표기한다.

2.4 소프트웨어 결함 감지 및 정정

결함 감지: 소프트웨어를 이용한 결함 감지는 결함으로 인해 태스크 연산의 출력, 결과가 변경된다는 가정을 기반으로 한다. 소프트웨어를 이용한 결함 감지에서는 태스크를 두 번 중복 실행하고, 두 실행의 결과를 비교하여 결함이 있는지 확인한다. 이러한 추가적인 실행으로 인해 소프트웨어를 이용한 결함 감지의 실행 시간 오버헤드는 하드웨어를 이용한 방식보다 훨씬 크며, 이때 소프트웨어를 이용한 결함 감지

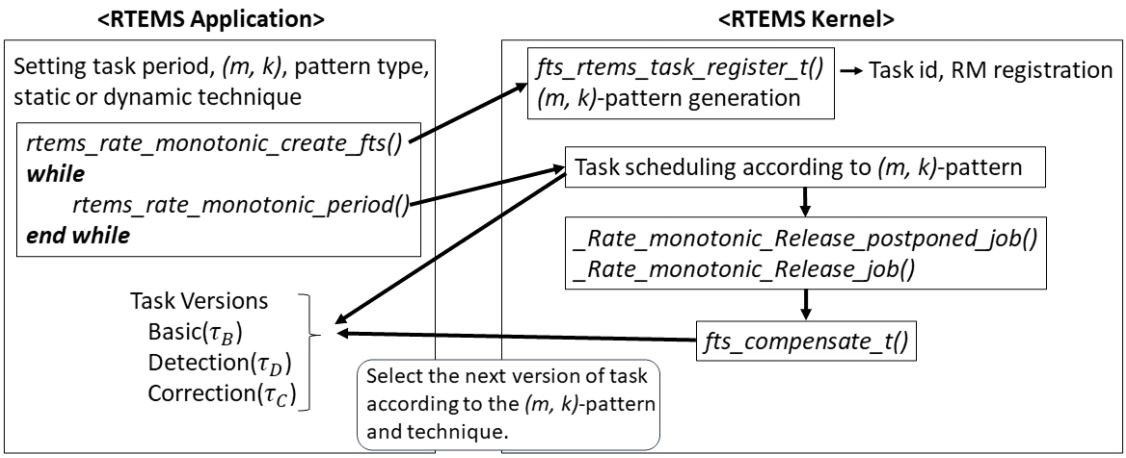


그림 3. RTEMS 애플리케이션과 커널간의 흐름도.
Fig. 3. Flowchart between the RTEMS application and kernel.

태스크는 $\tau_{D_{SW}}$로 표기하고, $\tau_{D_{HW}}$ 그리고 τ_B와의 최악 실행시간 비교는 $wcet_{D_{SW}} > wcet_{D_{HW}} > wcet_B$와 같다.

결함 정정: 소프트웨어를 이용한 결함 정정은 $\tau_{C_{SW/SW}}$로 표기되며 기본적인 태스크(τ_B) 연산의 삼중 실행을 기반으로 한다. 세 번의 중복 실행 후 연산의 출력 값은 세 가지 결과의 majority voting에 의해 결정된다. 따라서 결함 정정의 시간 오버헤드가 결함 감지 시간의 오버헤드 보다 훨씬 크며, 최악실행시간을 비교하면 $wcet_{C_{SW/SW}} > wcet_{D_{SW}} > wcet_B$와 같다.

Ⅲ. FTS 구현

[16]에서는 유럽 우주국과 [15]의 FTS 기법을 RTEMS 실시간 운영체제의 RM 스케줄러를 수정하여 구현하고자했고, 오픈소스로 공개했다. 하지만 커널 수준과 FTS 알고리즘에 많은 오류가 존재하여 실행할 수 없고, 애플리케이션 태스크 수준에서 결함 주입, 결함 검출, 그리고 결함 정정을 구현하지 못했다. 또한 더 이상 지원되지 않는 RTEMS-4.12 버전을 이용한 문제가 있다.

본 논문에서는 [15, 16]를 기반으로 인공위성에서 사용되는 LEON-3FT 기반의 GR712RC 개발 보드, RTEMS-5.0[21], 그리고 RCC(RTEMS Cross Compiler)-1.3-rc6[22] 환경에서 RTEMS 커널 코드의 일부를 수정하여 오류를 해결한 후 RTEMS 태스크 수준에서 결함 주입, 결함 검출, 그리고 결함 감내 기법을 추가로 구현했다. GR712RC, RTEMS, 그리고 RCC를 이용한 개발환경 설정은 Cobham Gaisler의 RCC User's Manual[22]을 참고하여 진행했다.

3.1 RTEMS 커널 수준

RTEMS 커널에서는 2개의 소스코드 파일(*fts_t.c, fts_t.h*)이 추가되고, 기존의 4개의 파일(*ratemon.h, ratemoncreate.c, ratemonperiod.c, Makefile.am*)이 수정된다. RTEMS에서 RM 스케줄러를 사용하기 위해선 애플리케이션에서 *rtems_rate_monotonic_create()* API(Application Programming Interface)를 이용하여 Period를 생성하고, while-loop에서 *rtems_rate_monotonic_period()*를 호출하여, 미리 정한 주기에 태스크를 스케줄링한다. RTEMS 애플리케이션과 커널의 전체적인 흐름은 그림 3과 같이 도식화 된다.

ratemoncreate.c, ratemon.h: RM FTS의 Period를 생성하기 위한 *rtems_rate_monotonic_create_fts()* 함수는 *ratemoncreate.c*에 구현되며, 이때 *fts_t.c*에 구현된 *fts_rtems_task_register_t()* 함수를 호출하여 커널에 태스크 id, RM 등의 정보를 등록한다. 이때 *rtems_rate_monotonic_create_fts()* API를 *ratemon.h*에 추가하여 커널 빌드과정에서 발생하는 오류를 해결했다.

ratemonperiod.c: RM FTS에서 태스크를 release 할 때 (m,k)-패턴에 따라 다음에 실행할 태스크를 결정하게 된다. 따라서 태스크를 release하는 *_Rate_monotonic_Release_postponed_job()* 함수와 *_Rate_monotonic_Release_job()* 함수에서 *fts_t.c*에 구현된 *fts_compensate_t()* 함수를 호출한다. 그리고 Period를 시작하는 *rtems_rate_monotonic_period()*에서 태스크가 FTS에 등록되어있는지 확인하고, 새로운 Period를 시작할 때 기존에 있던 태스크들을 모두 *rtems_task_delete()*을 이용해 제거한다. 이때 존재하지 않는 태스크를 제거하려고 하거나, 다음 태스크가 실행되지 않는 문제가 있어서, 태스크를 제거하는 코드를 모두 삭제하여 오류를 해결했다.

fts_t.c, fts_t.h: FTS의 (m,k)-패턴을 생성하고, 정적기법과 동적기법에 따른 알고리즘들이 구현되어 있는 소스코드 파일이다. 애플리케이션에서 설정한 (m,k)와 E, R 타입에 따라 (m,k)-패턴을 생성하는 과정에서 패턴이 제대로 생성되지 않는 많은 오류들을 수정했고, *fts_t.h*에서 *rtems.h*를 include하여 커널 빌드과정에서 계층이 무너지는 문제가 있어서 이를 해결했다.

Makefile.am: [16]의 RTEMS 버전과 본 논문의 5.0 커널의 디렉토리, 계층이 다르기 때문에 커널 빌드과정에서 오류가 발생한다. *rcc-1.3-rc6/cpukit/rtems* 경로에 위치한 *Makefile.am*에 *librtems_a_SOURCES += src/fts_t.c* 코드를 추가하여 FTS 소스코드 파일의 경로를 적어 오류를 해결했다.

최종적으로 *rcc-1.3-rc6/cpukit/rtems/src* 경로에 *fts_t.c, ratemonperiod.c, ratemoncreate.c*가 있고, *rcc-1.3-rc6/cpukit/include/rtems/rtems* 경로에 *fts_t.h, ratemon.h*가 있으며 이후 RCC User's Manual[22]을 따라 커널 부트스트랩, 빌드, 컴파일 과정을 거쳐서 RTEMS 커널 수준 수정을 마치고, 애플리케이션에서 RM FTS 기능들을 사용할 수 있다.

3.2 RTEMS 애플리케이션 태스크 수준

RTEMS 애플리케이션 태스크 수준에서는 태스크의 주기, (m,k), E 또는 R 타입, 정적 또는 동적기법 등을 설정하여 커널 수준에서 구현된 RM FTS를 이

용하여 태스크들을 스케줄링한다. FTS 스케줄러의 사용은 *rtems_rate_monotonic_create_fts()*를 이용하여 설정 값들에 따른 Period를 생성하고, while-loop에서 *rtems_rate_monotonic_period()*로 태스크를 주기적으로 스케줄링한다. 그림 3에 RTEMS 애플리케이션과 커널의 흐름도가 도식화되어 있다.

태스크는 2개를 구현했으며, τ_1은 John Hennessy의 Stanford baby benchmark[23]를 수행하며 이는 10개의 알고리즘(8개의 정수 연산 그리고 2개의 실수 연산)으로 구성된다. τ_2는 Whetstone benchmark[24]의 N1 floating-point operation을 수행한다.

Trap 핸들러: 앞의 섹션 2.3에서 메모리에 정정 불가능한 결함이 발생하면 AMBA ERROR 인터럽트가 발생하고, *data_access_exception trap(tt=0x09)*이 생성되며 프로세서가 정지된다고 설명했다. 소프트웨어적으로 정정 불가능한 결함을 처리하기 위해선 시스템이 종료되지 않아야 하므로, 해당 trap의 핸들러를 구현하였다. Cobham Gaisler의 application note[25]에서 BCC(Bare-C Cross Compiler) version 2일 때 trap 핸들러의 예제 코드를 확인할 수 있다. SPARC 프로세서의 *bcc_set_trap()* API를 이용하여, trap 테이블을 업데이트하는데, 이는 RTEMS에서 사용할 수 없는 문제가 있다. [25]를 참고하여 표 1과 같이 RTEMS의 *set_vector()* API를 이용해 trap에 대한 핸들러 함수(*My_handler()*)를 추가하고, *My_handler()* 함수에 인라인 어셈블리 코드를 작성하여 트랩된 명령어를 다시 실행시켜 SRAM에 정정 불가능한 결함(2-비트 이상)이 발생해도 프로세서가 정지되지 않게 구현했다.

표 1. 정정 불가능한 결함이 발생해도 프로세서가 정지되지 않게 구현한 *My_handler()* trap 핸들러 그리고 *set_vector()*를 이용한 핸들러 추가.
Table 1. A *My_handler()* trap handler implemented so that the processor doest not stop even when an uncorrectable fault occurs and the handler using *set_vector()* added.

```
1:  const int
    TT_DATA_ACCESS_EXCEPTION=0x09;
2:
3:  void My_handler()
4:  {
5:      __asm__ volatile ("jmp %l1");
6:      __asm__ volatile ("rett %l2");
7:  }
8:
9:  rtems_task Init()
10: {
11:     set_vector(My_handler,
        TT_DATA_ACCESS_EXCEPTION,0);
12: }
```

결함 감지: 하드웨어 EDAC를 이용한 결함 감지 태스크($\tau_{D_{HW}}$)는 표 2와 같이 AHB status register의 NE, CE 비트를 읽어서 정정 가능한 결함, 정정 불가능한 결함을 판별할 수 있다. 태스크를 실행할 때 마다 표 2의 과정을 수행하여 태스크 수준에서 결함을 감지할 수 있게 구현된다. 소프트웨어를 이용한 결함 감지 태스크($\tau_{D_{SW}}$)는 각 태스크의 벤치마크 연산을 두 번 중복 실행하고, 두 실행의 결과를 비교하여 결함의 발생 여부를 확인할 수 있다.

결함 정정: 하드웨어 EDAC를 이용했을 때 결함 정정 태스크($\tau_{C_{HW/SW}}$)는 $\tau_{D_{HW}}$를 재실행하여 구현할 수 있다. 소프트웨어를 이용한 결함 정정 태스크($\tau_{C_{SW/SW}}$)는 기본 태스크(τ_B)를 세 번 실행하고, 세 가지 벤치마크 연산 결과의 majority voting으로 구현된다.

결함 주입: 구현한 결함 감지, 결함 정정이 제대로 동작하는지 확인하기 위해 Cobham Gaisler의 application note[25]를 참고하여, 결함 주입을 구현하였다. SRAM에 EDAC를 적용했을 때 생성되는 체크섬이 MCFG3 레지스터의 TCB에 저장되며, 이때 TCB에서 1-비트, 2-비트 이상을 수정하여 인위적으로 결함을 주입할 수 있다. SRAM에서 데이터를 로드할 때 TCB의 체크섬을 확인하여 결함이 있으면 인터럽트, 트랩이 발생하는데, 체크섬이 수정된(결함이 주입된) 변수의 메모리에서 데이터를 로드할 때 캐시 메모리의 영향을 없애기 위해 *bcc_loadnocache()* API를 이용하여 강제로 캐시 미스를 유도한다. 해당 API는

표 2. AHB status register의 NE, CE 비트를 읽어서 결함의 종류 판별.
Table 2. The type of fault is detected by reading NE and CE bits of AHB status register.

```
1:  #define CE      (1 << 9)
2:  #define NE      (1 << 8)
3:
4:  if (AHBSTATUS & NE) != 0
5:      if (AHBSTATUS & CE) != 0
6:          return Correctable Error;
7:      else
8:          return Uncorrectable Error;
9:      end if
10: else
11:     return No Error;
12: end if
```

RTEMS에서 사용할 수 없기 때문에 표 3와 같이 *r32_no_cache()* 함수에 인라인 어셈블리 코드를 작성하여 구현했다.

표 3. 캐시 메모리의 영향을 없애기 위한 강제로 캐시 미스를 위한 코드.
Table 3. Code for forced cache miss to eliminate the effect of cache memory.

```
1:  static inline unsigned int
    r32_no_cache(uintptr_t addr)
2:  {
3:      unsigned int tmp;
4:      __asm__ volatile ("lda [%1] 1, %0\n":
    "=r"(tmp) : "r"(addr));
        return tmp;
5:  }
```

Ⅳ. 실 험

앞에서 설명한 RTEMS의 커널, 애플리케이션 태스크 수준에서의 구현을 마친 후 GR712RC에서 태스크들의 버전에 따른 최악실행시간을 측정했다. 태스크 버전들에 따른 도식은 그림 4에 도시되어 있으며, 태스크들의 버전에 따른 최악실행시간은 표 4에 정리되어있다.

τ_B는 벤치마크 연산만 수행하며, $\tau_{D_{HW}}$는 벤치마크 연산 이후 AHB status register를 읽어서, 결함의 발생

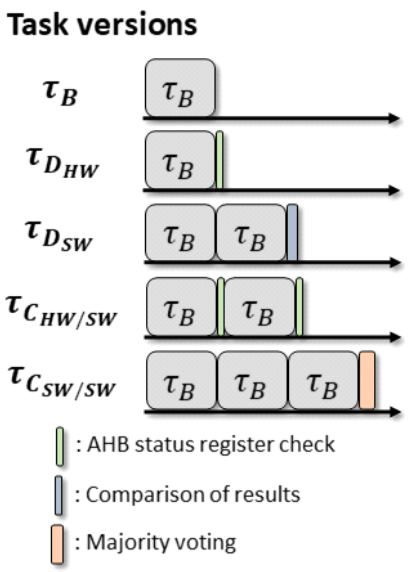


그림 4. 태스크 버전들에 따른 도식.
Fig. 4. Figure according to task versions.

표 4. 태스크들의 버전에 따른 최악실행시간(seconds).
Table 4. WCET(seconds) by version of tasks.

Version	τ_B	$\tau_{D_{HW}}$	$\tau_{D_{SW}}$	$\tau_{C_{HW/SW}}$	$\tau_{C_{SW/SW}}$
τ_1	1.421	1.644	2.901	2.844	4.158
τ_2	0.227	0.316	0.392	0.431	0.533

여부를 확인하고, $\tau_{D_{SW}}$는 벤치마크 연산을 두 번한 후 결과를 비교하여 결함의 발생 여부를 판단한다. $\tau_{C_{HW/SW}}$는 $\tau_{D_{HW}}$를 중복 실행하여 결함을 정정할 수 있고, $\tau_{C_{SW/SW}}$는 벤치마크 연산을 세 번한 후 majority voting을 통해 결함을 정정할 수 있다. 결함을 감지, 정정하기 위한 하드웨어의 도움이 있는 경우($\tau_{D_{HW}}$, $\tau_{C_{HW/SW}}$) 소프트웨어만을 이용한 경우($\tau_{D_{SW}}$, $\tau_{C_{SW/SW}}$)에 비해 연산 오버헤드가 매우 작은 것을 표 4를 통해 확인할 수 있다.

τ_1, τ_2의 주기가 각각 8s, 4s이고, $k=16$, SDR, E 패턴, 그리고 결함 주입이 없을 때 RM FTS에서 m의 변화에 따라 하드웨어를 이용한 결함 감지와 소프트웨어를 이용한 결함 감지의 CPU 이용률을 비교한 결과를 표 5에서 확인할 수 있다.

SDR은 $\phi=0$에서 τ_B, $\phi=1$에서 τ_D가 실행되며, 결함 주입이 없으므로 τ_C가 실행되지 않는다. m이 커질수록 τ_D가 많이 수행되기 때문에 CPU 이용률이

표 5. 하드웨어/소프트웨어를 이용한 결함 감지에서 m에 따른 CPU 이용률 결과.
Table 5. CPU utilization result from fault detection using hardware/software.

m	CPU utilization(%)	
	HW fault detection	SW fault detection
0	23.482	22.704
1	23.917	24.253
2	23.676	26.288
3	23.375	26.694
4	23.888	27.379
5	23.998	28.271
6	23.813	29.535
7	23.955	30.557
8	23.763	31.761
9	23.781	32.502
10	23.690	33.742
11	23.598	34.793
12	24.430	35.784
13	24.219	37.429
14	24.187	38.284
15	24.213	39.453
16	24.377	40.639

커지는 것을 확인할 수 있다. $\tau_{D_{SW}}$의 오버헤드가 $\tau_{D_{HW}}$에 비해 매우 크기 때문에 소프트웨어를 이용한 결함 감지가 하드웨어를 이용한 결함 감지에 보다 CPU 이용률이 크게 증가하는 것을 확인할 수 있다.

V. 결 론

본 논문에서는 인공위성에서 사용되는 RTEMS 실시간 운영체제와 SPARC 계열의 위성용 프로세서 개발환경에서 RTEMS의 RM 스케줄러를 수정하여 (m,k)-패턴에 따른 FTS 기법을 구현하고, 애플리케이션 태스크 수준에서 결함 주입, 결함 검출, 그리고 결함 정정 기법을 구현하여 SRAM 메모리에 주입된 결함을 감지, 정정하고 이를 보고하였다.

운영체제의 태스크 수준에서 소프트에러를 감지, 감내하기 위한 대부분의 연구는 이론적인 모델링 위주이며 직접 구현하고, 검증하는 실용적인 사례가 거의 없다. 또한 watchdog timer를 이용한 시스템 재부팅은 시간 오버헤드가 굉장히 크기 때문에 실시간 제약 조건이 존재하는 시스템에서 예기치 않은 결과로 이어질 수도 있다.

실험에서 하드웨어와 소프트웨어를 이용한 결함 검출, 결함 정정 기법에 따른 태스크들의 최악실행시간을 측정했고, CPU 이용률을 분석했다. 실험결과 (m,k)-패턴의 m을 키울수록 오버헤드가 큰 강화된 태스크가 많이 실행되어 CPU 이용률이 증가하는 것을 확인할 수 있었다. 이때 하드웨어를 이용한 결함 검출 기법이 소프트웨어를 이용한 기법에 비해 오버헤드가 매우 작은 것을 확인할 수 있었다.

소프트에러 신뢰성을 향상하기 위해 (m,k)-패턴의 m을 크게 설정하면 CPU 이용률이 증가하고, 소비 전력의 증가로 발열량이 증가하여 수명신뢰성이 악화될 수 있을 것으로 생각할 수 있다. 향후 주변온도가 크게 변하는 인공위성 시스템에서 (m,k)-패턴의 소프트에러 신뢰성을 기능 안전 표준에 따라 분석하고, CPU의 소비 전력 측정 및 발열 모델링을 하여 온도에 의존적인 고장 메커니즘 분석으로 수명신뢰성(MTTF, Mean Time To Failure)을 얻고, 최종적으로 소프트 에러 신뢰성과 하드에러 신뢰성(수명)을 함께 분석 및 최적화할 계획이다. 본 논문에서는 이를 위한 실험환경을 구축하였다.

References

[1] JEDEC Standard JESD89A, "*Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices*," JEDEC solid state technology association, vol. 1, no. 6, p. 8, 2006.

[2] ISO 26262, "*Road vehicles-functional safety*," International Standard ISO/FDIS, vol. 26262, 2011.

[3] A. Manuzzato, S. Gerardin, A. Paccagnella, L. Sterpone, and M. Violante, "Effectiveness of TMR-based techniques to mitigate alpha induced SEU accumulation in commercial SRAM-based FPGAs," in *Proc. Conf. Radiation Effects Compon. Syst.*, pp. 98-104, 2007.

[4] M. J. Wirthlin, A. M. Keller, C. McCloskey, P. Ridd, D. Lee, and J. Draper, "SEU mitigation and validation of the LEON3 soft processor using triple modular redundancy for space processing," in *Proc. 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, pp. 205-214, 2016.

[5] A. M. Keller and M. J. Wirthlin, "Benefits of complementary SEU mitigation for the LEON3 soft processor on sram-based FPGAs," *IEEE Trans. Nuclear Sci.*, vol. 64, no. 1, pp. 519- 528, 2016.

[6] L. Sterpone and M. Violante, "A new partial reconfiguration-based fault-injection system to evaluate SEU effects in SRAM-Based FPGAs," *IEEE Trans. Nuclear Sci.*, vol. 54, no. 4, pp. 965-970, 2007.

[7] P. S. Ostler, M. P. Caffrey, D. S. Gibelyou, P. S. Graham, K. S. Morgan, B. H. Pratt, H. M. Quinn, and M. J. Wirthlin, "SRAM FPGA reliability analysis for harsh radiation environments," *IEEE Trans. Nuclear Sci.*, vol. 56, no. 6, pp. 3519-3526, 2009.

[8] A. Kohn, R. Schneider, A. Vilela, A. Roger, and U. Dannebaum, "Architectural concepts for fail-operational automotive systems," in *Proc. SAE Tech. Paper Ser.*, p. 8, Apr. 2016,

Accessed: Dec. 11, 2020. [Online]. Available: https://www.sae.org/publications/technical-papers/content/2016-01-0131/,doi:10.4271/2016-01-0131.

[9] A. Kohn, M. Kabmeyer, R. Schneider, A. Roger, C. Stellwag, and A. Herkersdorf, "Fail-operational in safety-related automotive multi-core systems," in *Proc. 10th IEEE Int. SIES*, p. 14, Jun. 2015.

[10] A. Cobham Gaisler, "GR712RC dual-core LEON3FT sparc v8 processor," *Data Sheet*, Jan. 2016.

[11] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *IEEE Trans. VLSI Syst.*, vol. 17, no. 3, pp. 389-402, 2009.

[12] S.-H. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele, "Reliability-aware mapping optimization of multi-core systems with mixed-criticality," in *Proc. Design, Autom. Test Eur. Conf. Exhib.* (DATE), pp. 1-4, Mar. 2014.

[13] J. Xu and B. Randell, "Roll-forward error recovery in embedded realtime systems," in *Proc. Int. Conf. Parallel Distrib. Syst.*, pp. 414-421, 1996.

[14] "*Handling of External Memory EDAC Errors in LEON/GRLIB Systems*," Retrieved Oct. 6, 2020, from https://www.gaisler.com/doc/antn/GRLIB-AN-0004.pdf

[15] K.-H. Chen, B. Bönninghoff, J.-J. Chen, and P. Marwedel, "Compensate or ignore? meeting control robustness requirements through adaptive soft-error handling," *ACM SIGPLAN Notices*, vol. 51, no. 5, pp. 82-91, 2016.

[16] K.-H. Chen, J. Sherill, M. Yayla, and G. Bloom, "Software-based Fault Tolerance: Adding Fault Tolerance Feature in the Rate Monotonic Scheduler," *ESA Summer of Code in Space 2017*, accessed Dec. 9, 2020, [Online]. Available: https://devel.rtems.org/wiki/SOCIS/2017.

[17] B.-S. Yoo, J.-W. Choi, J.-Y. Jeong, S.-W. Kim, "Performance analysis of processors for next generation satellites," *IEMEK J. Embedded Syst. and Appl.*, vol. 14, no. 1, pp. 51-61, 2019.

[18] A. Cobham Gaisler, "*Grmon3 user's manual*," 2017.

[19] G. Quan and X. Hu, "Enhanced fixed-priority scheduling with (m,k)-firm guarantee," in *Proc. 21st IEEE Real-Time Syst. Symp.*, pp. 79-88, 2000.

[20] L. Niu and G. Quan, "Energy minimization for real-time systems with (m,k)-guarantee," *IEEE Trans. VLSI Syst.*, vol. 14, no. 7, pp. 717-729, 2006.

[21] "*RTEMS real time operating system (RTOS)*," http://www.rtems.org.

[22] A. Cobham Gaisler, "*Rcc user's manual*," 2018.

[23] W. J. Price, "A benchmark tutorial," *IEEE Micro*, vol. 9, no. 5, pp. 8-43, 1989.

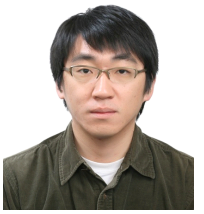
[24] R. P. Weicker, "An overview of common benchmarks," *Computer*, vol. 23, no. 12, pp. 65-75, 1990.

[25] A. Cobham Gaisler, "Handling of external memory edac errors in leon/grlib systems," *Application note*, Doc. No GRLIB-AN-0004, no. 1.1, 2017.

김 범 식 (Beomsik Kim)

2019년 2월 : 아주대학교 전자공학과 학사 졸업
2021년 2월 : 아주대학교 전자공학과 석사 졸업
<관심분야> 실시간 내장형 시스템의 신뢰성 인식 및 내결함성 설계
[ORCID:0000-0002-8992-7691]

양 회 석 (Hoeseok Yang)



2003년 2월 : 서울대학교 컴퓨터공학부 학사 졸업
2010년 2월 : 서울대학교 컴퓨터공학부 박사 졸업
2010년~2014년 : 스위스 취리히 연방공과대학(ETH Zurich) 연구원
2014년~현재 : 아주대학교 전자공학과 부교수
<관심분야> 하드웨어-소프트웨어 통합 설계, 멀티 프로세서의 신뢰성과 발열의 최적화 및 분석. 내장형 시스템을 위한 딥러닝, 비휘발성 메모리와 내장형 시스템 디자인
[ORCID:0000-0002-7929-7470]