Qplus-T 내장형 인터넷 시스템에서 멀티 태스크 프로그램을 위한 원격 트레이스 디버거

(A Remote Trace Debugger for Multi-Task Programs in Qplus-T Embedded Internet System)

이 광 용 † 김 흥 남 ††

(Kwang-Yong Lee) (Heung-Nam Kim)

요 약 최근 인터넷의 급속한 성장으로 Web TV, PDA 및 Web phone과 같은 장치들이 인터넷에 연결되기 시작하고 있다. 그러나, 이러한 장치들은 복잡한 실시간 응용 시스템을 지원하기 위해 RTOS와 같은 실시간 운영체제가 필요로 하게 되었으며, 특히, 내장형 인터넷 응용 시스템을 개발하기 위한 디버거 등과 같은 적절한 도구들의 부족으로 개발하는데 어려움을 겪고 있다. 이에, 본 논문에서는 Qplus-T 실시간 운영체제 내장형시스템을 위한 새로운 트레이스포인트 디버깅 도구를 제안하다. 이 도구는 타이밍 트레이스포인트들을 이용하여 실시간 응용 소프트웨어의 디버깅을 쉽게 한다. 전통적인 브레이크포인트 디버거에 비해, 이 트레이스포인트 디버거는 온라인 및 오프라인 분석을 위해 응용 프로그램의 데이터를 동적으로 수집하고 기록하는 기능을 제공한다. 그리고, 응용프로그램의 실행을 멈추거나 원래의 실행 속도에 참견이 거의 없이 실행중인 응용프로그램의 변수들에 새로운 값을 할당해 보기 위한 수단으로도 제공된다. 본 논문에서 제시하는 트레이스포인트 디버거는 Qplus-T 인터넷 응용프로그램과 같은 타겟 실시간 응용 프로그램에 수많은 모니터링 트레이스포인트들을 추가하기 위한 효과적인 방법을 제공하며, 실행 중에 응용프로그램의 행위를 모니터하고 분석하기 위한 트레이스포인트를 설정할 수 있다. 또한, RTL(Real-Time Logic)표현을 이용하여 타이밍 문제를 명세화하고 검출할 수 있어 기존 트레이스포인트 디버거와는 다르다.

키워드 : 실시간운영체제, 정보가전, 원격 트레이스 디버거, 실시간 시스템, 내장형 인터넷 시스템, 큐플러스-T 실시간운영체제

Abstract With the rapid growth of Internet, many devices such as Web TVs, PDAs and Web phones, begin to be directly connected to the Internet. These devices need real-time operating systems (RTOS) to support complex real-time applications running on them. Development of such real-time applications called embedded internet applications, is difficult due to the lack of adequate tools, especially debuggers.

In this paper, we present a new tracepoint debugging tool for the Qplus-T RTOS embedded system, which facilitates the instrumentations of the real-time software applications with timing trace-points. Compared with traditional breakpoint debugger, this trace-point debugger provides the ability to dynamically collect and record application data for on-line examination and for further off-line analysis. And, the trace-points can also provide the means for assigning new values to the running application's variables, without neither halting its execution nor interfering with its natural execution flow. Our trace-point debugger provides a highly efficient method for adding numerous monitoring trace-points within a real time target application such as Qplus-T internet applications, utilizing these trace-points to monitor and to analyze the application's behavior while it is running. And also, our trace debugger is different from previous one in that we can specify and detect the timing violations using its RTL (Real-Time Logic) trace experiments.

Key words: Real-time operating system(RTOS), Internet Appliance, Remote trace debugger, Real-time system, Embedded internet system, Qplus-T RTOS

† 정 회 원 : 한국전자통신연구원 인터넷정보가전연구부 연구원
kylee@etri.re.kr
†† 비 회 원 : 한국전자통신연구원 인터넷정보가전연구부 연구원
hnkim@etri.re.kr
논문접수 : 2002년 5월 3일
심사완료 : 2002년 11월 19일

1. 서 론

최근 인터넷의 급속한 성장으로 Web TV, PDA, 및 Web phone과 같은 장치들이 인터넷에 연결되기 시작하고 있다. 그러나, 이러한 장치들은 복잡한 실시간 응용 시스템을 지원하기 위해 RTOS와 같은 실시간 운영체제가 필요로 하게 되었으며, 특히, 내장형 인터넷 응용 시스템을 개발하기 위한 디버거 등과 같은 적절한 도구들의 부족으로 개발하는데 어려움을 겪고 있다[1-7].

인터넷 정보가전 용 응용 시스템 개발은 대체로 연성 실시간 시스템(soft real-time system) 개발 분야 이지만, 오디오, 비디오, 이미지 처리 등과 관련된 태스크들의 시간 및 타이밍의 파악기법이 요청되고 있고, 실제로 산업계에서는 현재 만들고 있는 제품뿐만 아니라 기 개발되어 있는 제품에서 발생된 버그를 손쉽게 파악할 수 있는 기법을 요청하고 있다. 예를 들어, 자동차의 ECU 테스팅 기법과 비슷하게 제품에 맞는 디버깅용 테스트 케이스(test-case) 프로그램들을 여러 가지 만들어 놓고 버그가 의심되는 오류에 맞는 테스트케이스를 선택하여 제품을 실제로 정지가 없이 가동시켜가면서 버그를 추적하고 찾아내는 기법을 만들고 싶어한다.

이에, 본 논문에서는 본 연구소에서 자체 개발한 QPlus-T 정보가전 용 실시간 운영체제[1]에서 수행되는 실시간 소프트웨어의 개발에 있어 모니터링 트레이스포인트(tracepoint)들을 이용하여 실시간 수행 중에 발생되는 시스템 버그들을 멈춤이 없이 쉽게 찾아낼 수 있게 하는 원격 트레이스 디버거를 제시한다.

하드웨어 개발에 있어 시그널들을 모니터링하기 위하여 트레이스포인트(혹은 테스트포인트)를 이용하는 것은 하드웨어 디버깅의 기본 작업으로 볼 수 있다. 작업 환경으로부터 실시간에 연속적으로 시그널들을 수집하고 그것을 시간과 함께 시각적으로 보는 것은 시스템의 행위를 이해하기 위한 기초를 제공한다.

본 논문에서 제시하고자 하는 논스톱 디버깅을 위한 트레이스포인트 설정기법은 실시간 소프트웨어의 개발에 있어 명령어 레벨에서 하드웨어 개발과 마찬가지로 모니터링 트레이스포인트들을 설정할 수 있게 함으로써 실시간 수행 중에 시스템의 행위를 시간과 함께 파악할 수 있게 하며, 간단한 점검 목록에 따라 제대로 동작하고 있는지를 점검할 수 있게 한다.

이처럼, 모니터링 트레이스포인트들을 이용하여 실시간 응용을 구현하는 것은 제품 테스팅 및 디버깅 능력에 커다란 도움을 준다. 모니터링 트레이스포인트들은 응용 시스템의 실행을 가시화하고, 온라인(on-line)에 상태를 파악하기 위해 동적으로 응용 시스템의 자료를 시간정보와 함께 수집/저장하며, 게다가 오프라인(off-line) 분석을 위해 사용할 수 있다. 디버깅 시 이 모니터링 트레이스포인트를 이용하는 것은 수행 중인 프로그램의 변수들에 어떤 참견이나 본래의 실행흐름을 멈추지도 않으면서도 새로운 값을 할당할 수 있게 하며, 응용 프로그램의 논리적인 실행 흐름을 관찰할 수 있는 수단을 제공한다.

현재, 트레이스포인트를 이용한 대표적인 디버깅 방법들은 다음과 같은 것들이 있다[7].

- Logic Analyzer
- Hand-instrumented code("printf debugging")
- Automated code instrumentation systems

Logic Analyzer는 하드웨어 디버깅을 위한 기본적인 장비 이긴 하지만 사용하기에는 그 가격이 만만치 않으며, 크기가 큰 정보가전 용 응용 소프트웨어의 디버깅을 위해 사용하는 것은 부적절하다. 그리고, 위의 두 번째 예에서처럼 손으로 직접 테스팅 할 프로그램에 "printf" 문과 같은 디버깅 코드를 추가하여 디버깅하는 기법을 가장 많이 사용하고 있으나, 가장 원시적인 방법이며 쉬운 방법이기는 하나 멀티태스크 프로그램과 같은 응용에서는 디버깅이 어려운 요소도 있다. 마지막 예에서와 같이 트레이스포인트에 디버깅 코드를 자동으로 삽입해 주는 도구들도 있으나 아직 그 수도 적으며 테스팅에도 한계가 있으며, 실시간 시스템의 개발에 있어 중요한 타이밍과 같은 분석에는 아직 미흡한 면이 많다.

2. 새로운 트레이스 디버깅 기법 제안

2.1 breakpoint vs. tracepoint

인터넷 정보가전 응용과 같은 멀티태스크 프로그램들의 수행과정에서 발생하는 오류를 손쉽게 디버깅(debugging)하기 위해서는 프로그램의 실제 실행에 영향 미치는 탐침 효과(probe-effect)를 최소화하고, 사건 기반형 디버깅 기법을 사용하여 동일한 입력에 대하여 동일한 결과를 항상 얻도록 재실행(replay) 가능하게 하여 병렬처리의 비결정성을 해결해야 하며, 편리한 사용자 인터페이스를 제공함으로써 오류 검출 과정에 대한 어려움을 감소시킬 필요가 있다.

멀티태스크 프로그램을 브레이크포인트(breakpoint) 설정에 의한 방법으로 디버깅[8-10]하게 되면 기본적으로 응용 프로그램의 실행을 멈추게 함으로써 실제 프로그램 행위를 변경하는 결과를 초래하게 되고, 실세계의 시스템 장치들은 실제로 멈추지 않고 그 실행을 계속하게 되는데 디버깅을 위해 강제적으로 멈추게 함으로써 그 근본상태를 변경시키고, 그로 써 디버깅을 어렵게 하

는 경우가 많다.

이에 비해, 트레이스포인트를 이용한 디버깅 기법[7,10]은 본래의 속도로 프로그램을 수행시키면서 원하는 위치에서 정확한 자료를 수집할 수 있게 함으로써 본래의 속도로 수행될 때만 발생하는 버그들이나, 아주 짧은 시간에 발생하는 혹은 그와 반대로 아주 긴 시간에 발생하는 버그들, 그리고 예측하기 어렵거나, 다시 재현 불가능한 버그들, 그리고 실제 필드 조건에서만 발생하는 버그들을 쉽게 찾을 수 있게 한다.

구체적으로, GDB의 경우 브레이크포인트가 실행될 시점에 응용 프로그램의 수행을 멈추고 디버거가 프로그램 제어를 갖게 되며 그 시점에서 디버깅과 관련된 컴맨드(command)들 예를 들어 레지스터(register) 정보 혹은 메모리(memory) 정보를 본다든지 하는 명령들을 통해 디버깅하게 되는 것에 비해, 트레이스포인트의 경우에는 트레이스포인트에서 트레이스 디버거는 응용 프로그램에 대한 어떤 제어나 참견이 없이, 그 트레이스포인트와 관련된 디버깅 행위를 타겟에서 직접 수행해주고, 그 행위들에 의해 수집된 결과는 자동화된 분석도구 혹은 디버거에 의해 나중에 검토할 수 있게 트레이스 버퍼 같은 곳에 저장한다.

이처럼, 트레이스포인트를 이용한 디버깅은 응용프로그램의 실제 실행에 영향이 미치지는 않는 방향에서 디버깅 가능하게 하고, 트레이스 버퍼와 같은 것을 통해 사후에 결정적(deterministic)으로 응용 프로그램을 디버깅 가능하도록 하는 기법을 제공한다.

2.2 RTL을 사용한 새로운 트레이스 디버거

본 논문에서 제안한 트레이스 디버거의 목적은 프로그램이 타겟에서 실행하는 동안 실시간에 사건발생 시간과 함께 트레이스정보를 수집하고 있다가 본래의 속도로 수행될 때만 발생하는 시간관련 버그들이나, 아주 짧은 시간에 발생하는 오류 혹은 그 반대로 아주 긴 시간에 걸쳐서 발생하는 시간관련 버그들을 RTL(Real-Time Logic) 모니터링 기법[11-14]을 사용하여 쉽게 파악해내고 수정할 수 있게 하는 것이다.

현재, 디버거로써 트레이스포인트를 이용하여 디버깅 기법을 사용하는 것으로 가장 대표적인 것은 GNU GDB의 "Introspect" 기능으로 다음과 같은 단계로 실시간에 프로그램을 트레이스 하는 기법이다.

단계1. 트레이서를 시작한다.

```
(gdb) load test.o
(gdb) tstart
```

단계2. 트레이스포인트를 설정한다.

```
(gdb) trace tree.c:find
```

단계3. 각각의 트레이스 포인트들에서 수집할 자료를 정의한다.

```
(gdb) actions
> collect tree, tree->key
> collect tree->vector.p[10]
> collect $regs
> collect $locals
> end
```

단계4. 프로그램을 실행한다.

```
(gdb) run
```

단계5. 수집된 트레이스 자료를 이용하여 실행된 결과를 순차적으로 재실행해보면서 중간중간의 결과를 분석한다.

```
(gdb) tfind line 123
(gdb) print tree->vector.p[10]
(gdb) print key == tree->key
(gdb) tfind next
```

이상과 같이, GNU GDB의 "Introspect" 기능은 단계2처럼, "trace" 컴맨드를 통해 응용 프로그램에 직접 트레이스 포인트를 설정하고, 단계3처럼, 그 트레이스포인트에서 수집할 자료를 "collect" 함수를 통해 정의하고, 단계4에서 실행하면서 저장해두었다가 실행이 종료된 후에, 단계5에서처럼 "tfind (trace find)" 명령어를 이용하여 사후 분석한다.

그러나, 아직 이 기법에는 실시간 시스템 개발에 있어 가장 중요한 요소중의 하나인 시간관련 제약사항[15,16]을 트레이스 해볼 수 없다든지, 조건부 트레이스포인트(conditional tracepoint)를 설정 못한다든지, 트레이스 도중에 즉, 프로그램이 실행하는 도중에 트레이스버퍼의 내용을 볼 수 없다든지 하는 문제들이 있다. 그리고, 반영구적인 테스트스위트(testsuite)의 개발이 어렵게 되어 있어 진정한 의미의 실시간 시스템 테스팅 도구로서의 역할이 미비하며, 현재는 "Programming in the small" 개발 환경에서 적용 가능한 트레이싱 기법이지 "Programming in the large" 개발환경 즉, 수십에서 수백 개로 이루어져 있는 응용 프로그램 개발에서 있어 적용 가능한 테스팅 기법으로 볼 수 없다. 그리고, 무엇보다도 현재는 호스트 쪽의 트레이스 디버깅 컴맨드 기능에 대하여 소스 공개가 된 수준이나, 타겟 쪽에서의 트레이스포인트 디버깅관련 기술은 공개되고 있지 않다.

이에, 본 논문에서는 GNU GDB의 트레이스 디버깅 한계를 보완한 다음과 같은 새로운 트레이스 디버깅 모델(그림 1 참조)을 소개한다.

QpTracer = GDB for Qplus-T + Tracer with RTL

이 모델을 살펴보면, 기존 트레이스 기법들과는 다르게 C혹은 C++로 작성된 소스프로그램에 RTL-태그(tag)들을 통해 트레이스포인트를 설정하게 되고, 소스 프로그램 실행에 따라 그 RTL-태그들에서 발생하는 트레이스 트랩 사건[RTL로 표현하면 @(e,i)사건]에 반응하여 그 RTL사건과 1:1맵핑되어 있는 테스트 케이스의 특정 함수가 실행되고, 이 함수에서는 트레이스 버퍼에 RTL 사건들의 실행이력을 저장하거나 이미 앞서 저장시켜 놓은 실행이력 정보를 이용하여 현재, RTL 사건에서의 타이밍 제약의 만족 여부를 실행 시(Run-Time)에 검증한다. 또한, 필요에 따라서는 사후에 트레이스 버퍼를 접근하여 재실행 등등의 방법을 통해 프로그램의 실행상태를 분석할 수 있도록 한다.

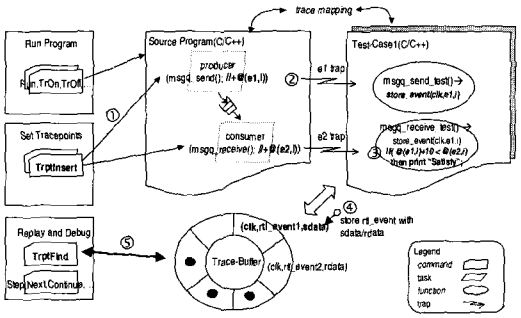


그림 1 새로운 트레이스 디버깅 모델

이처럼, 본 논문에서 제안한 기법은 기존 기법들에 비해 다음과 같은 장점들을 갖게 된다.

- 소스코드와 테스트케이스의 분리를 통해 반영구적인 테스트스위트 즉 테스트케이스들을 이용한 디버깅이 가능토록 함
- C/C++코드에 의한 테스트케이스 작성을 통해 테스팅 표현력을 확장시킴
- RTL-태그를 사용함으로써 실시간 시스템 개발에 필수적인 타이밍 제약사항의 실시간 논리 분석/설계/구현을 쉽게 함
- 트레이스 도중에도 트레이스 버퍼를 접근하여 그 내용을 살펴볼 수 있게 함
- 테스트케이스 작성을 돕기 위한 트레이스 스터브 함수 생성기와 같은 프리프로세싱 도구들을 제공함으로써 "Programming-in-the-small" 개발환경뿐만 아니라 "Programming-in-the-large" 개발환경에서도 적용 가능한 테스팅 기법 제공
- 응용 프로그램에의 인스트럭션 레벨뿐만 아니라 메모리의 절대 어드레스를 직접 접근할 수 있게 함으로써 커널수준의 디버깅도 가능하게 함

3. 응용 시스템의 타이밍 모니터링을 위한 트레이스 함수의 정의

실시간 시스템의 실행에 있어 모니터링될 대상은 사건들의 발생 시점이다[5]. 하나의 사건은 응용 프로그램에서 하나의 상태에서 다른 상태로 변경시키는 관심 있는 부분이다. 예를 들어, 생산자(producer) 태스크에서 메시지를 소비자(consumer) 태스크에게 보내는 경우 혹은 소비자 태스크가 그 메시지를 받는 경우 등등은 응용 프로그램의 상태를 변경시키는 주요 사건들로 볼 수 있다. 본 논문에서는 응용 프로그램에서 발생되는 주요 사건들간의 타이밍 제약사항을 표현하기 위해 RTL (Real-Time Logic)에 기반 한 표기법을 사용한다.

본 절에서는 응용 시스템의 소스 프로그램으로부터 직접 사건들을 식별해 내고 이들로부터 타이밍 제약사항을 모델링 하는 과정과, 텍스트로 모델링 된 타이밍 제약을 그래픽 하게 타이밍 제약성 그래프로 바꾸는 방법과, 이것을 타이밍 violation을 점검하는 규칙으로 바꾸어 트레이스 함수들에 내장 시킴으로써 테스트케이스를 작성하는 과정에 대해 간략히 소개 한다.

3.1 응용 프로그램의 타이밍 제약사항 식별

소스 프로그램으로부터 상태를 변경시키는 사건들을 식별하고, 이들 사건들로부터 다음 예에서처럼 마감시간(end-to-end deadline) 혹은 지연 타이밍 제약(delay timing constraints)를 비정형(informal)적으로 정의한다.

- 마감시간 제약사항 예: 태스크 T의 실행 사건을 E라 하고, 그 태스크가 실행을 완성할 때의 사건을 E′이라 하고, 그 태스크는 주기적으로 실행되고 있다고 할 때, 태스크 T의 실행이 100ms의 마감시간(deadline)을 갖고 있다는 의미는 E와 E′ 사건의 발생이 100ms 내에 이루어져야 함을 의미한다.
- 지연시간 제약사항 예: 무한히 반복되는 태스크 T의 i번째 특정 사건을 E라하고, 이 사건의 다음 사건 즉, i+1번째 사건을 E′이라 할 때, 태스크 T에서의 E사건의 반복에서 1000ms의 지연시간(delay)이 주어져야 한다는 의미는 E와 E′의 사건이 1000ms의 지연을 두고 실행되어야 함을 의미한다.

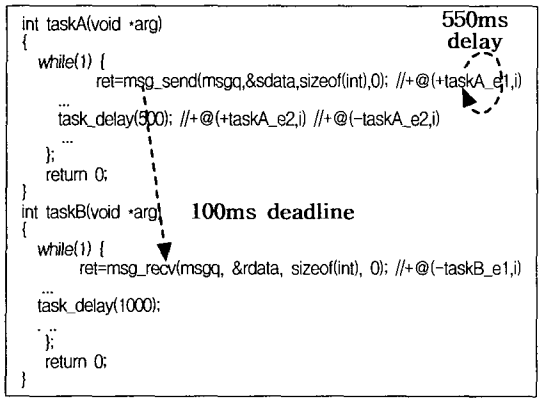
이처럼, 타이밍 제약사항을 분석하기 위해서는 사건들에 대한 발생시점 및 관계를 기록할 필요가 있다.

본 논문에서는 소스프로그램으로부터 직접 확장된 RTL-태그를 사용하여 관심 있는 사건들을 식별해내며,

이들로부터 마감시간 혹은 지연시간과 관련된 타이밍 제약사항을 식별해 낸다. 소스프로그램의 RTL-태그들은 다음과 같이 RTL의 하나의 사건과 같은 형태로 표현하지만 소스프로그램에 직접 추가되기 때문에 소스라인의 주석 문 형태로 삽입이 되고, 소스 라인의 하나의 문장(statement)이 시작되기 전 혹은 후의 트레이스포인트를 설정하기 위해 사건이름 e앞에 각각 + 혹은 - 접두사(prefix)를 둔다. ± 옵션이 없는 경우에는 디폴트로 +의미 즉, 문장이 시작하기 직전의 트레이스포인트 설정을 의미한다.

//+@((e,i)

아래의 예에서의 타이밍 제약은 "msg_send" 사건 즉, @(+taskA_e1,i) RTL-태그로부터 "msg_recv" 사건 즉, @(+taskB_e1,i) RTL-태그까지의 자료 송신 마감시간은 100ms이어야 하며, "msg_send"에서 다음 번 "msg_send"의 지연시간은 550ms 이상이 되어야 하며, taskA의 태스크 지연 시간이 시작(@(+taskA_e2,i))부터 종결(@(-taskA_e2,i))시까지 500ms가 되어야 한다.



이상의 응용 프로그램의 타이밍 요구사항을 비정형적인 형태로 정리하면 다음과 같다.

t1. taskA의 "msg_send"에서 자료를 보낸 시간과 taskB의 "msg_recv"에 자료가 도착한 시간의 차는 최대 100ms를 초과해서는 안 된다.

t2. taskA의 메시지 송신 간격은 적어도 550ms 이상이어야 한다.

t3. taskA의 태스크 지연시간이 500ms 이상이 되어야 한다.

3.2 RTL 설계 및 적법성(legality) 검증

일단, 응용 프로그램의 타이밍 요구사항이 식별이 되면 RTL 형태로 타이밍 제약사항 들을 정형화한다. RTL의 가장 간단한 형태의 표현은 다음과 같이 정의된다.

$$T_1 \le T_2 \pm D(D>0) \quad (1)$$

여기에서, D는 양의 정수(positive integer)이고, T1과 T2는 @(e,i) 혹은 0값을 갖으나, T1, T2 중 최소한 하나는 사건을 표현해야 한다.

마감시간 제약사항의 경우에는 시간상으로 $\forall_i$ @(e,i) ≤@(f,i)인 두 사건들 사이에 @(e,i) 사건 이후에 마감시간 D(D>0)이내에서 @(f,i) 사건이 발생되어야 함을 의미하므로 다음과 같은 형태로 표현이 가능하다.

$$\forall_i @(f,i) \le @(e,i) + D(D>0)$$

한편, 지연 제약사항의 경우는 시간상으로 $\forall_i @(e,i) \le @(f,i)$인 두 사건들 사이에 @(e,i) 사건 이후 지연시간 D(D>0)이후에 @(f,i) 사건이 발생되어야 함을 의미하므로 다음과 같은 형태로 표현이 가능하다.

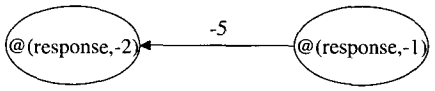
$$\forall_i @(e,i) \le @(f,i) - D(D>0)$$

예를 들어, 다음과 같은 지연시간 제약사항의 예는 다음 그림 2a에서와 같이 -5값을 아크(arc)로 갖고 동치비교 기호(≤)와 같이 @(response,-1) 사건으로부터 @(response,-2)으로 향하는 즉, ←의 방향성 그래프로 표현되며, @(response,-2)이후 5단위시간 지연 후에 @(response,-1) 사건이 발생됨을 의미한다.

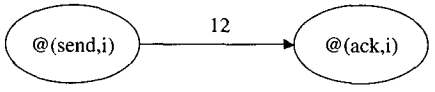
$$@(\text{response},-2) \le (@(\text{response},-1) - 5$$

또한, 다음과 같은 마감시간 제약사항은 @(send,i) 사건 후 12단위시간 내에 @(ack,i) 사건이 발생되어야 함을 의미하고, 시간 제약성 그래프로 표현하면 그림 2b와 같이 +12값을 아크로 갖고 @(send,i) 사건으로부터 @(ack,i) 사건으로 향하는 즉, 동치비교 기호와 같은 방향의 방향성 그래프(directed-graph)로 표현된다.

$$\forall_i @(\text{send},i) \le @(\text{ack},i) \wedge$$
$$\forall i @(\text{ack},i) + @(\text{send},i) + 12$$



(a) delay constraints



(b) deadline constraints

그림 2 타이밍 제약성 그래프

타이밍 제약성 그래프에서의 임의의 두 노드 즉, u, v 간의 경로(path)는 u로부터 v로의 아크(arc)들의 순서로 표현되며, 특정 경로의 길이(length)는 경로상의 모든 아크들의 가중치(weight) 값의 합으로 표현된다.

타이밍 제약 사항들이 명세화 되면 이것을 타이밍 제약성 그래프로 표현한 후 지연시간 제약사항 및 마감시간 제약사항을 지키고 있는지 점검해야 한다. 점검하는 방법은 그래프 상의 모든 경로의 길이에 대해 음수 사이클(negative cycle)이 존재하는지 확인하는 것이다[11-14]. 이것은 요구사항의 적법성(legality)을 점검하는 것이다.

앞 3.1절의 예에서 식별한 타이밍 요구사항을 시간제약사항 표현기법에 따라 표현하면 각각 다음과 같다.

t1. 마감시간 제약사항의 RTL 표현

$$\forall_i(@(+\text{taskA\_e1},i)\le@(-\text{taskB\_e1},i)\wedge @(-\text{taskB\_e1},i)\le@(+\text{taskA}_{e1,i})+100))$$

t2. 지연시간 제약사항의 RTL 표현

$$\forall_i@(+\text{taskA\_e1},i-1)\le@(+\text{taskA\_e1},i)-550$$

t3. 태스크 지연시간 제약사항의 RTL 표현

$$\forall_i@(+\text{taskA\_e2},i)\le@(-\text{taskA\_e2},i)-500$$

3.3 타이밍 제약사항의 내장

아래의 코드에서처럼 프로그램의 적절한 시점에서 소스코드에 타이밍 제약사항을 내장(embed)시킨 것과 같은 방법으로 특정 트레이스포인트에 연결된 트레이스함수 내부에 타이밍 제약사항을 내장 시킨다.

```
temp = read_sensor();temp = read_sensor();
if( @val(temp,-1) - @val(temp,-2) > 200) {
        shutdown_device();
}
```

그러나, 본 논문에서 제안하는 기법에서는 기존 기법들과는 다르게 소스코드에 직접 타이밍 제약사항을 내장 시키는 것이 아니라, 트레이스 시점에서 수행되는 트레이스함수(trace function)에 타이밍 제약사항을 내장 시킨다.

프로그램 실행 중 타이밍 제약사항의 만족여부는 RTL 타이밍 제약사항의 표현을 특정 트레이스포인트에서 만족하게 되는지를 확인함으로써 점검한다. 물론, 지연시간 혹은 마감시간 제약사항의 경우에는 다음과 같은 상태에서 타이밍 제약사항을 어기게 됨(violation)을 알 수 있다.

- 지연시간 제약사항 violation : 0번 노드로부터 e_n노드까지의 음수 길이 $-T(T(0)$를 갖는 경로에 대해, 노드 e_n에 해당하는 사건이 시간 T전에 발생하게 되는 경우에 violation이 발생한다고 볼 수 있음
- 마감시간 제약사항 violation : 0번 노드로부터 최소 경로의 최소 길이를 T라 하고 그 노드를 e_m이라 할 때, e_m이 T 혹은 T시간 전에 발생하지 않으면 violation이 발생한다고 볼 수 있음

이러한, 제약사항의 violation 조건에 따라 3.2절에서 정의한 RTL 타이밍 제약사항을 만족하는지 여부를 if 문으로 표현하며, violation 발생시 적절한 오류 메시지와 함께 필요에 따라서는 적절한 조치를 함께 한다. 본 논문에서 제시하는 트레이스함수에서는 유틸리티 함수로 특정 사건의 발생 시점을 트레이스 버퍼로부터 읽어오는 "get_clk(e,i)" 함수와 트레이스버퍼에 사건정보를 저장하는 "store_event(e,i)" 함수를 사용한다. 이 함수는 사건 e의 i번째 발생한 시점을 트레이스버퍼에서 찾아 리턴하거나 그 정보를 저장한다.

아래 예는 3.2절의 RTL 타이밍 제약사항을 트레이스함수에 내장시킨 예이다.

t1. taskB_e1_post() 함수 ==>

```
void taskB_e1_post(int i, QDB_IU_REGS *regs)
{
    int rdata = *(int *)LOCAL_ADDR(rdata_offset);
    store_event( @(taskB_e1_post,i) ,,rdata);
    if( get_clk( @(taskB_e1_post,i) ,0 ) <= get_clk( @(taskA_e1_pre,i) ,rdata) +100 )
                                print( 100 ms deadline : satisfy !!! );
    else
      print( 100 ms deadline : unsatisfy !!! );
    ...
}
```

t2. taskA_e1_pre() 함수 ==>

```
void taskA_e1_pre(int i, QDB_IU_REGS *regs)
{
    int sdata = *(int *)LOCAL_ADDR(sdata_offset);
    store_event( @(taskA_e1_pre,i) ,sdata);
    if( get_clk( @(taskA_e1_pre,i) ,sdata-1 ) <= get_clk( @(taskA_e1_pre,i) ,sdata )  550 )
                print( 550 ms delay: satisfy !!! );
    else
                print( 550 ms delay: unsatisfy !!! );
    ...
}
```

t3. taskA_e2_post() 함수 ==>

```
void taskA_e2_pre(int i, QDB_IU_REGS *regs)
{
    store_event( @(taskA_e2_pre,i) ,i++);
}

void taskA_e2_post(int i, QDB_IU_REGS *regs)
{
    store_event( @(taskA_e2_post,i) ,i++);
    if( get_clk( @(taskA_e2_pre,i) ,i) <= get_clk( @(taskA_e2_post,i) ,i)  500 )
                print( 500 ms delay: satisfy !!! );
    else
                print( 500 ms delay: unsatisfy !!! );
    ...
}
```

4. 원격 트레이스 디버거의 구현

멀티태스크 프로그램의 수행과정에서 발생하는 오류를 손쉽게 검출하고 수정하기 위해서는 다음과 같은 원리들을 만족해야 한다.

- 프로그램의 실제 실행에 영향 미치는 탐침 효과(probe effect)를 최소화해야 함
- 사건 기반형 디버깅 기법을 사용하여 동일한 입력에 대하여 동일한 결과를 항상 얻도록 재실행 가능하게 하여 병렬처리의 비결정성을 해결해야 함
- 사용하기 쉬운 사용자 인터페이스를 제공함으로써 오류 검출 과정에 대한 어려움을 감소시킬 필요가 있음.
- 다수의 태스크들이 병행적으로 수행되는 실시간 시스템 개발환경에서 필수 점검 요소인 타이밍 및 시간 제약을 검증할 수 있는 기법의 제공도 중요함

본 논문에서 제안하는 트레이스 디버거는 이러한 요구사항을 만족시키기 위한 구조로 되어 있다.

4.1 Qplus-T 교차 디버깅 환경

Q+Esto 원격 디버깅을 위한 교차(cross-development) 환경[1-3]은 다음 그림 3에서와 같다. 이 그림에서와 같이 원격 교차 디버거가 수행되는 호스트 시스템과, 디버깅 모니터를 수행하는 타겟 시스템(여기에서는 SA110을 탑재한 EBSA21285 보드를 지칭함), 그리고 이들을 연결하는 시리얼(serial) 라인 혹은 이더넷(ethernet) 라인으로 구성된다. 대부분의 원격 디버깅 환경에서는 시리얼라인과 이더넷 라인을 동시에 활용한다. 이의 구현 형태를 좀 더 자세히 살펴보면 호스트 시스템은 Windows NT/95/98/2000을 플랫폼으로 하는 범용 컴퓨터로 구성되어 있으며, 이 호스트 컴퓨터에 원격 트레이스 디버거가 구현되어 있으며, 타겟 시스템은 EBSA21285 하드웨어를 기반으로 구성되고 이 하드웨어 위에 Qplus-T커널과 응용프로그램, 그리고 이 응용프로그램을 트레이스 모니터링하기 위한 트레이스 데몬 시스템들로 구성되어 있다.

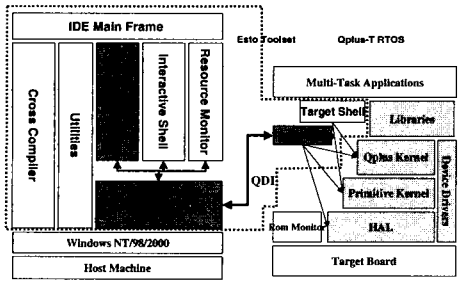


그림 3 원격 트레이스 디버깅 환경

Qplus-T 실시간운영체제[1]는 한국전자통신연구원에서 1998.11-2000.12사이에 개발한 정보가전용 실시간 운영체제로 디지털 TV와 인터넷 TV 등에 활용이 가능한 기능을 갖추고 있으며, 국제 표준 및 업계 표준과의 호환성을 최대한 보장한다. 또한 이 Qplus-T는 다양한 정보가전 기기들에 쉽게 활용될 수 있게 하기 위하여 아래 그림 3과 같이 조립성 및 확장성을 가질 수 있는 다계층(Multi-layer) 구조로 구성되어 있다. 타겟 하드웨어와의 인터페이스 계층인 하드웨어 추상화 계층(HAL:Hardware Abstraction Layer)은 커널의 하드웨어 의존성을 피하기 위한 계층으로서 CPU와 메모리, 그리고 타이머 등과 같은 핵심 하드웨어 요소에 대한 추상화 함수를 제공하여 이식성을 높인 부분이다. 그 다음 계층은 기본 커널(PK: primitive kernel)로 쓰레드를 기반으로 하는 요소들로 구성되어 있으며, 기본 스케줄러 및 동기화 메커니즘을 제공한다. 이 부분은 Qplus-T의 핵심 부분이라 할 수 있으며 신뢰도가 높은 RTOS인 VRTX[17]의 나노커널(Nanokernel)을 기반으로 구현된 부분이다. Qplus-T 커널 계층(QP: Qplus-T kernel)은 태스크들 및 ISR(Interrupt Service Routine)들 사이의 통신 및 동기화 메커니즘을 위해 프로세서간 통신을 제공하며, 태스크와 네트워크, 그리고 메모리 관리에 대한 사용자 API를 제공한다. 그리고 I/O 서브시스템(IOS)은 I/O 디바이스를 통한 데이타 입출력의 일관성 유지를 위한 POSIX 호환 인터페이스 부분으로서, I/O 디바이스 드라이버의 등록과 제거를 용이하게 해준다.

4.2 트레이스 디버거 아키텍쳐 및 디버깅 시나리오

구체적으로, 트레이스포인트 기반 멀티태스크 디버거의 아키텍쳐는 다음 그림 4에서와 같이 호스트에 트레이스 디버거 GUI부분과 트레이스매니저, 트레이스엔진, 리모트 타겟 매니저로 구성되고, 타겟에는 트레이스데몬으로 구성된다. 이 트레이스포인트 기반 멀티태스크 디버거도 브레이크포인트 기반의 멀티태스크 디버거와 마찬가지로 윈도우 NT 위에서 작성하고 컴파일한 오브젝트 파일을 타겟보드(여기에서는 홈서버 혹은 Web-PAD용 셋탑박스)에 적재한 뒤 실행하면서 생성된 멀티태스크들(혹은 쓰레드들)을 추적/관리 한다. 타겟에서의 트레이스데몬은 각각의 응용 태스크들에 설정된 트레이스포인트에서의 트리거링에 따라 하나의 테스트케이스로 미리 만들어져 있는 트레이스함수들을 호출함으로써 태스크들을 적절한 시점에 타이밍 제약사항 같은 것을 모니터링하고 그 결과를 수집하여 트레이스가 종료가 된 후에 재실행 혹은 디버그를 위한 분석자료로 활용된다.

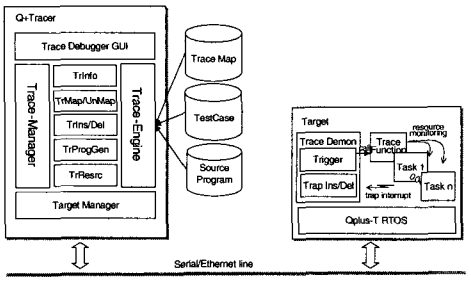


그림 4 원격 트레이스 디버거 구조

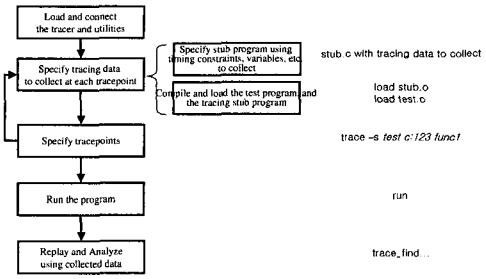


그림 5 Programming-In-the-Small에서의 테스팅

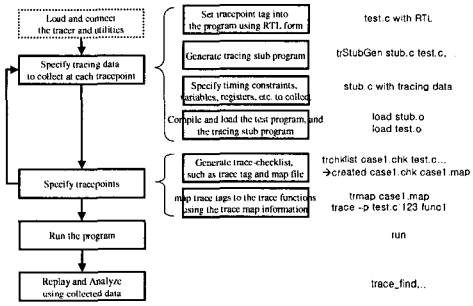


그림 6 Programming-In-the-Large에서의 테스팅

트레이스포인트 기반 원격 멀티태스크 디버거의 목적은 프로그램이 타겟에서 실행하는 동안 실시간에 트레이스정보를 수집하고 있다가 사후에 본래의 속도로 수행될 때만 발생하는 버그들이나, 아주 짧은 시간에 발생하는 오류 혹은 그 반대로 아주 긴 시간에 걸쳐 발생하는 버그들 등등을 재실행 기법 등을 통해 쉽게 분석할 수 있게 하는 것이다.

트레이스 포인트를 사용하여 디버깅을 하기 위한 일반적인 시나리오는 다음과 같다.

단계 1. 테스트할 프로그램을 분석하여 테스트 케이스(test-case)들을 생성한다(트레이스 맵 작성, 트레이스 함수 작성).

단계 2. 테스트할 포인트 즉, 트레이스포인트를 테스트할 프로그램에 설정한다.

단계 3. 테스트할 프로그램을 실행한다.

단계 4. 타겟에서 수집된 자료를 토대로 재실행 등을 통해 시스템의 상태를 조사 분석한다(bug 찾기).

이상의 단계들을 테스트할 프로그램의 크기에 따라 좀 더 세분화해보면 그림 5 및 그림 6과 같이 두 종류로 분류 가능하다. 그림 5의 경우는 소규모 프로그래밍 환경에서의 테스팅 절차로 프로그래머가 직접 트레이싱 자료가 담겨져 있는 트레이스함수 즉, 그림의 "stub.c"를 작성한 후 트레이스 설정 컴맨드를 활용하여 트레이스포인트를 지정한 후 실행하는 방법이고, 그림 6의 경우는 그림 5와는 달리 테스트할 프로그램에 RTL-태그를 삽입하고 이로부터 "stub.c" 트리거함수를 "trStubGen"와 같은 컴맨드를 통해 자동생성하여 테스트케이스들과 "case1.map"과 같은 트레이스 맵핑 파일 정보를 생성한 후, 이를 통해 트레이싱하는 기법을 사용한다. 프로그래밍 규모 혹은 테스트할 요소의 수에 따라 테스트기법을 달리 적용할 수 있다.

4.3 호스트에서의 트레이스포인트 설정/해제/관리

다음 그림 7은 호스트에서의 트레이스 디버거의 내부 구조를 보여주고 있다. 이 그림에서 보듯이 트레이스 디버거는 사용자 인터페이스 모듈, 트레이스 매니저 모듈, 트레이스 디버깅 엔진, 그리고 리모트 디버깅 인터페이스 모듈의 4가지 모듈들로 구성되어 있다. 먼저, 사용자 인터페이스 모듈에서의 트레이스 설정, 해제, 및 관리와 관련된 컴맨드들을 입력으로 하여 트레이스 매니저 모듈에서는 그 컴맨드들을 해석하여 트레이스 디버깅 엔진에 전달하고 트레이스 디버깅 엔진에서 수행된 결과를 받아 다시 사용자 인터페이스 모듈로 전달한다. 그리고, 리모트 트레이스 인터페이스 모듈에서는 호스트의 트레이스 디버깅 엔진으로부터의 타겟 접근 관련 명령에 반응하여 타겟의 트레이스 데몬과의 연결을 담당하는 모듈로 시리얼 혹은 이더넷 통신을 하며, 트레이스포인트의 설정/해제, 트레이스 시작 및 종결, 그리고 트레이스 정보의 접근 등의 작업을 수행한다.

구체적으로 트레이스포인트 설정/해제/관리와 관련된 기본 명령어들은 트레이스 디버깅 엔진의 내부 컴맨드로 정의되어 있으며, 다음과 명령어들로 구성된다.

• info trace : 설정된 트레이스포인트 정보를 나열한다.

(qptracer) info trace

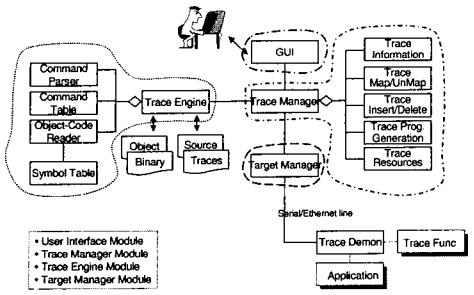


그림 7 트레이스 디버거의 내부구조

- trace : 모니터링할 어플리케이션 프로그램의 특정 라인에 트레이스포인트를 설정한다.

  (qptracer) trace gdb_test_b.c:53 user_call_3 0
- trdel : 어플리케이션에 설정된 트레이스포인트를 삭제한다.

  (qptracer) trdel 3
- trunmap : 현재까지 설정된 트레이스포인트들을 어플리케이션 코드로부터 삭제하는 기능을 수행한다.

  (qptracer) trunmap
- trmap : trace.map 파일로부터 트레이스포인트 정보를 읽어 로딩된 어플리케이션 코드에 트레이스포인트 정보를 세팅한다.

  (qptracer) trmap trace.map
- trstubgen : test 프로그램들로부터 RTL-tag를 읽어 테스트케이스 스터브 함수들을 자동으로 생성하여 하나의 테스트스위트를 구성한다.

  (qptracer) trstubgen test-case1-stub.c test1.c ...
- trchklist : test 프로그램들로부터 RTL-tag를 읽어 테스트 점검목록 파일과 트레이스 맵핑파일을 자동으로 생성한다.

  (qptracer) trchklist test-case1.chk test1.c test2.c ...
- local : 특정함수로부터 지역변수 값들의 오프셋 어드레스를 출력한다. 트레이스 포인트에서의 사용자 함수에서 지역변수 값을 액세스 하기 위해 사용한다.

  (qptracer) local MAX
- global : 전역변수의 내용을 표시한다.

  (qptracer) globals

이상의 기본 트레이스포인트 명령어들을 구성하기 위해서는 트레이스포인트 구조체와 트레이스포인트 체인을 생성하고 이들을 핸들링하기위한 기본 함수들을 구축해야 한다. 다음과 같이 트레이스포인트 정보를 나타내는 구조체는 트레이스포인트 체인을 구성하기 위한 "next" 필드와 트레이스 포인트 번호를 나타내는 "number" 필드, 트레이스포인트 심볼 테이블, 소스라인, 및 프로그램 카운터(pc) 정보를 나타내는 필드, 그리고 사용자 함수 정의 필드의 크게 네 가지 기본 필드들로 구성되어있다. 그리고, 이들 정보를 핸들링하기 위해 트레이스포인트 체인에 트레이스포인트 구조체를 삽입/삭제하는 함수, 트레이스포인트체인으로부터 트레이스포인트 정보를 찾는 기능 그리고, 트레이스포인트 함수 추가 기능 등의 유틸리티 함수들로 구성된다.

```
// tracepoint struct define
struct tracepoint {
        struct tracepoint *next;   // 다음 트레이스포인트
        int number;                // 트레이스포인트 식별 번호
        struct symtab *symtab;     // 심볼테이블
        int line;                  // 트레이스 소스라인
        CORE_ADDR pc;              // 트레이스포인트 어드레스
        char func_name[1024];      // 트레이스 함수 이름
        CORE_ADDR func_addr;       // 트레이스 함수 어드레스
        int func_arg;              // 트레이스 함수의 파라메터
};
```

```
// global variables
static int trPt_count = 0;
struct tracepoint *tracepoint_chain = NULL;

// utilities
static void add_tracepoint(struct tracepoint *tr);
static struct tracepoint *delete_tracepoint(int no);
static struct tracepoint *find_tracepoint_using_number(int no);
static struct tracepoint *find_tracepoint_using_fname_line(char *fname,int line);
static struct tracepoint *find_tracepoint_using_pc(CORE_ADDR pc);
static void TracepointFuncAdd(int trace_num, char *trace_func, int trace_arg);
```

예를 들어, "trace.map" 파일로부터 트레이스포인트 정보를 읽어 로딩된 어플리케이션 코드에 트레이스포인트 정보를 세팅하고 응용프로그램을 실행시키면서 트레이스 자료를 버퍼에 수집하고, 트레이스 자료를 읽어 사용자에게 표시해주는 과정을 메시지 시퀀스 챠트로 살펴보면 다음 그림 8과 같다. 그림에서 살펴보면, "trmap" 컴맨드에 의해 타겟의 트레이스 데몬의 트레이스포인트 배치 핸들러가 반응하여 트레이스포인트를 테스트할 응용 프로그램에 직접 삽입하게 되고, 테스트할 응용프로그램의 실행(run)에 의해 각각의 트레이스포인트들에서 트랩(trap) 인터럽트가 발생하게 되고, 트리거링 처리 객체는 이것에 반응하여 그 트레이스포인트 트랩에 해당하는 트레이스 함수를 찾고 그것을 수행하여 RTL 사건을 저장하고, 그와 동시에 사용자가 추가한 RTL 타이밍 제약(timing-constraints)에 따라 사건 히스토리로부터 안전성(safety)을 실시간에 검증한다. 사용자는 트레이스포인트가 설정된 응용 프로그램의 실행

과 병행해서 트레이스 버퍼에 저장되어 있는 자료를 읽어 그 상태를 관찰할 수 있다.

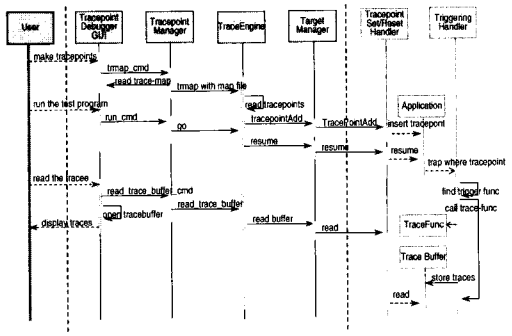


그림 8 트레이스 과정

4.4 타겟에서의 트리거링 처리

타겟의 트레이스 데몬은 다음 그림 9와 같이 구성된다. 시스템 초기화 및 호스트로부터의 컴맨드 패킷 통신을 담당하는 시스템초기화블록, 통신처리블록, 패킷해석블록, 그리고 명령어처리블록 들과, 트레이스포인트를 설정하거나 해제 시켜주기 위한 트레이스 설정/해제 블록 그리고, 응용프로그램의 트레이스포인트로부터의 트레이스 트랩 인터럽트에 반응하여 트레이스함수를 호출해주는 트리거링처리 블록들로 구성되어 있으며, 테스트 대상이 되는 응용 프로그램과 RTL 트레이스 포인트 태그에 1:1대응되는 트레이스 함수들과 이들로부터 입수된 RTL 사건들을 저장하기 위한 트레이스 버퍼로 구성된다.

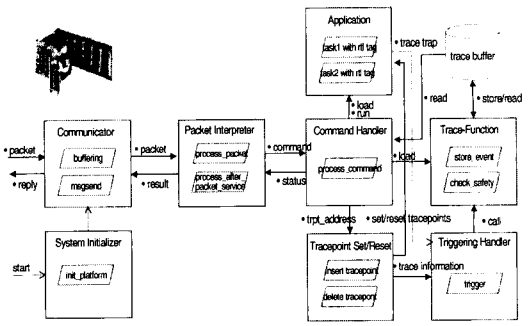


그림 9 타겟에서의 트레이스 데몬 구조

트레이스 설정/해제 블록에서는 어플리케이션 프로그램의 특정 위치에 트레이스포인트 즉 예외코드(exception code)들을 설정 혹은 해제하기 위해서 다음의 2가지 기본 유틸리티 함수들을 사용한다.

- qdbTpAdd(struct breakpoint *bp, int pId) : 브레이크포인트 리스트에 트랩 인터럽트 정보를 추가한다.
- qdbTpDelete(TGT_ADDR_T addr) : 브레이크포인트 리스트로부터 addr에 해당하는 브레이크포인트 정보를 삭제한다.

qdbTpAdd와 qdbtpDelete 함수는 각각 브레이크포인트 리스트에 각각 트레이스포인트 관련 정보를 삽입 혹은 삭제하는 기능으로 다음과 같은 브레이크포인트 구조체정보를 이용하여 리스트를 구성한다.

```
struct breakpoint
{
    dll_t       bp_chain; // 다음 브레이크포인트 혹은 트레이스포인트
    INSTR       *bp_addr; // 브레이크포인트 혹은 트레이스포인트 어드레스
    INSTR       bp_instr; // bp_addr에서의 인스트럭션
    int         bp_task; // 테스크 id
    unsigned    bp_flags;
    unsigned    bp_action; // bp_addr에서의 예외 행위 타입
    void        (*bp_callRtn)(); // bp_add에서의 예외 행위 함수(트레이스 함수)
    int         bp_callArg; // 트레이스 함수 아귀먼트
}
```

여기에서, 트레이스포인트 구조체 정보인 경우 브레이크포인트 정보와의 혼선을 방지하기 위해 "bp->bp_action"을 "QDB_ACTION_TRACE"으로 설정하여 등록한다.

한편, 타겟에서의 트리거링 처리블록에 의한 트리거링 처리는 브레이크포인트 처리에서처럼 어플리케이션의 트레이스포인트에 예외코드를 삽입하고 실행 중 이 예외코드가 실행되게 되면 트랩 인터럽트가 발생되어 다음의 qdbTrap 예외처리기(exception handler) 함수가 실행된다. 그러나, 현재 예외코드는 브레이크포인트와 트레이스포인트에서 동시에 사용하게 되어 있기 때문에 인터럽트가 발생하는 경우 트레이스포인트에 의해 발생한 경우인지 아니면 브레이크포인트에 의해 발생한 경우인지 구분이 명확하지 않은데 이것을 위해 "qdbBpFind" 함수를 이용하여 브레이크포인트 리스트로부터 현재 어드레스 (pc)에 해당하는 "bpInfo" 정보를 찾아내고 "bpInfo.bp_action" 정보가 "QDB_ACTION_TRACE"인가를 판단하여, 이 정보가 트레이스포인트에 의한 인터럽트일 경우에는 트레이싱 정보를 수집하기 위해 사용자가 정의한 트레이스 함수를 호출한다.

```
void qdbTrap(void *param, int sig, cpu_context_t *cx)
// Input :- parameter param, signal sig, context cx
// Output :- breakpoint or tracepoint trap handling
{
    initialize;
    if (Multi-Tasking)
    {
        qdbBpFind (cx->sp->pc,QP_TASK_ID(qp_self)),&bpInfo);
        if( bpInfo.bp_action == QDB_ACTION_TRACE )
        {
            call bpInfo.bp_callRtn(bpInfo.bp_callArg, pRegisters);
            for all tracepoint trap,
                delete the trap and then
                    recovery the original instruction;
            return;
        }
        else
        {
            breakpoint trap handling;
        }
        /* NOTREACHED */
    }
}
```

5. 적용실험

앞서 살펴본 바와 같이 트레이스 포인트를 사용하여 디버깅을 하기 위해서는 아래와 같은 단계들을 거쳐야 한다.

단계 1. 트레이스 함수 작성

단계 2. 테스트 케이스 컴파일 및 오브젝트파일 로딩, 그리고, 트레이스함수 맵핑

단계 3. 프로그램 실행

단계 4. 수집된 자료를 조사 분석- bug 찾기

5.1 적용사례

본 논문에서는 내장형 시스템 개발환경에서 비정지 실시간 디버깅을 위한 원격 트레이스 디버거의 유용성을 검증하기 위해 다음 그림 10과 같은 간단하지만 내장형 시스템에서의 가장 대표적인 예제를 실험예제로 선택하였다. 이 그림에서 보듯이 "producer" 태스크는 "msgq_send()"라는 함수를 이용하여 하나의 메시지큐 즉, "msgq"에 "sdata" 자료를 보내게 되고, 이것을 "consumer" 태스크는 "msgq_receive()" 함수를 통해 적절한 시점에 "rdata" 자료를 가져온다. 이 예제에서 볼 때, "producer" 태스크가 메시지를 보내는 간격은

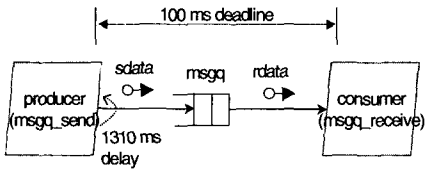


그림 10 producer-consumer 프로그램

1310ms 이후 마다 한 번씩 보내어야 하고, "producer" 태스크의 메시지 송신 후 동일 데이타가 "consumer" 태스크에 도달하는 최대 허용 시간(마감시간)은 100ms 이내로 시간제약이 따르고 있다.

5.2 테스트케이스의 작성

디버깅 시나리오에 따라 첫 번째로 소스프로그램을 분석하여 RTL-태그들과 그 태그에 해당하는 트레이스 함수들을 정의한다. 실험에서는 "msgq_send"와 "msgq_receive" 사이의 마감시간 타이밍 제약과 "msgq_send"와 이전 "msgq_send" 사이의 지연시간 타이밍 제약을 점검하기 위해 각각 "msgq_send" 문장과 "msgq_receive" 문장에 아래와 같이 RTL-태그 즉, 트레이스포인트를 설정하였으며, 4장의 실시간 논리 설계 기법에 의해 다음과 같은 RTL 타이밍 제약을 구한다.

- msgq_send to msgq_receive 100ms 마감시간 제약:

$$\forall i,\ (@(+\text{taskB\_msgq\_send\_ret},i) \le (@(-\text{taskA\_msgq\_recv\_ret},i) \wedge @(-\text{taskA\_msgq\_recv\_ret},i) \le @(+\text{taskB\_msgq\_send\_ret},i) + 100)$$

- msgq_send to msgq_send 1310ms 지연시간 제약:

$$\forall i, @(+\text{taskB\_msgq\_send\_ret}, i-1) \le @(+\text{taskB\_msgq\_send\_ret}, i) - 1310$$

```
tid_con = task_create("Consumer", consumer, 0, 0);
tid_pro = task_create("Producer", producer, 0, 0);

int producer(void *arg)
{
        while(1) {                                          1310ms
            ret = msgq_send(msgq, &sdata,
                sizeof(int), MSGQ_PRIO_NORMAL); //+@(+pro_msgq_send,i)
            ...
            task_delay(1000); //+@(+pro_task_delay,i) //+@(-pro_task_delay,i)
            ...
    };
    return 0;
}                                   100ms deadline
int consumer(void *arg)
{
        while(1) {
            ret = msgq_recv(msgq, &rdata, sizeof(int), 0); //+@(-con_msgq_recv_ret,i)
                                ...
            task_delay(1200);
                                . . .
    };
    return 0;
}
```

RTL-태그 정의와 동시에 각각의 트레이스포인트에서 수집해야 할 자료들을 정의하기 위해 다음과 같이 "trStubGen" 프리프로세서(preprocessor)를 이용하여 "producer-consumer" 프로그램 즉, pro_con.c 파일의 RTL-태그로부터 트레이스 스터브 함수 프로그램 즉, "testcase1.c" 프로그램을 생성한다.

```
(qptracer) trstubgen pro_con_testcase1.c pro_con.c
pro_con_testcase1.c - Trace-stub program generated.
```

생성된 "testcase1.c" 프로그램은 다음의 코드에서와 같이 RTL-태그의 사건이름으로부터 함수이름을 만들고, 함수 내부에는 기본적인 트레이스 디버깅 정보와 타이밍 제약성 점검을 위한 사건정보의 기록등과 같은 기본적인 모니터링 코드들이 추가되어 있으며 마지막 부분에는 사용자가 디버깅 혹은 시스템 모니터링을 위한 코딩 영역이 있다.

```
//+@(+pro_msgq_send,i) at pro_con.c:67
void pro_msgq_send_pre(int I,QDB_IU_REGS *regs)
{
        char *tr_event = "@(+pro_msgq_send,i)";
#ifdef BASIC_TRACE
        char timeStr[256];
        int tid;
        // get clock value
        sprintf(timeStr,"%lu",clock_get_time());
        // get taskId
        tid = task_getid();
        // print basic trace data
        printf("@c/%s:@t/0x%x:@pc/0x%x:@i/%d:@tag/%s:\n",
                timeStr,tid,(unsigned int)regs->pc,I,tr_event);
#endif

#ifdef TIMING_TRACE
        store_event(clock_get_time(),tr_event,0);
#endif

        // TODO : Add your codes for monitoring

}
```

이처럼, 본 논문에서 제시하는 트레이스포인트 설정기법의 큰 장점 중의 하나는 각각의 트레이스포인트에서의 자료정의를 C언어를 이용하여 작성할 수 있다는 것이다. 이것은 C언어를 직접 사용함으로써 테스트 시점에서의 테스트 자료 값들을 자유롭게 선택하고 정의해서 볼 수 있게 하고 테스트를 위해 특별히 다른 언어를 배울 필요 없게 한다. 그리고, 소프트웨어 트레이스포인트 설정기법 중 가장 쉽게 접하는 "printf" 문 삽입에 의한 디버깅처럼 개발자에게 특별한 테크닉을 필요로 하지 않고 손쉽게 사용할 수 있으면서 디버깅할 코드와 별개의 테스트케이스 함수 집합을 구성하게 함으로써 원본 코드의 훼손 없이 디버깅이 가능하게 한다. 개발자가 몇 가지 테스트 케이스를 마련하고서 그 테스트들을 통과하면 어플리케이션 프로그램의 품질을 인정할 때 유용하게 사용될 수 있다.

다음의 코드들은 트레이스 스터브 함수들에 실시간로직 (Real-Time Logic)이 구현된 예로 마감시간 타이밍 제약의 경우에는 @(-taskA_msgq_recv_ret,i) 트레이스포인트에서 즉, 메시지 송신에 의해 자료가 도착되는 시점에서 모니터링하는 것으로 코딩 하고, "msgq_send" 지연시간 타이밍 제약의 경우는 @(+taskB_msgq_send_ret,i) 트레이스포인트에서 지연시간 제약을 점검하는 것으로 코딩하나, 트레이스버퍼로부터 이전 사건이 발생된 시점을 파악하기 위해 "tr_occurrence" 변수를 이용하여 사건정보를 저장하고, 이것을 이용하여 해당 사건의 이전 사건의 발생 시각을 구하여 이것을 통해 타이밍 제약을 점검하는 것으로 코딩한다.

[deadline 타이밍 제약 구현 예] ==>

```
//+@(-con_msgq_recv_ret,i) at pro_con.c:92
void con_msgq_recv_ret_post(int I,QDB_IU_REGS *regs)
{
char *tr_event = "@(-con_msgq_recv_ret,i)";

#ifdef TIMING_TRACE
 store_event(clock_get_time(),tr_event,0);
#endif

 // TODO : Add your codes for monitoring
 {
        int rdata = *LOCAL_ADDR2( sizeof(void *) + sizeof(int) );
        int deadline = 100;
        unsigned int tr_clk1, tr_clk2;

        tr_clk1 = get_clk_from_elh2("@(+pro_msgq_send,i)",rdata);
        tr_clk2 = get_clk_from_elh(tr_event);
        if ( (unsigned int)tr_clk2 <=  (unsigned int)tr_clk1 + deadline )
        {
 printf("@sv/satisfy:@d/%d:@v/tr_clk1=%d:@v/tr_clk2=%d:@v/rdata=%d:\n",\
(unsigned int)(tr_clk1-tr_clk2),tr_clk1,tr_clk2,rdata);
        }
        else
        {
printf("@sv/unsatisfy:@d/%d:@v/tr_clk1=%d:@v/tr_clk2=%d:@v/rdata=%d:\n",\
(unsigned int)(tr_clk1-tr_clk2),tr_clk1,tr_clk2,rdata);
                //task_suspend(tid_pro);
                //task_suspend(tid_con);
        }
 }
}
```

[delay 타이밍 제약 구현 예] ==>

```
//+@(+pro_msgq_send,i) at pro_con.c:67
void pro_msgq_send_pre(int I,QDB_IU_REGS *regs)
{
 char *tr_event = "@(+pro_msgq_send,i)";
#ifdef TIMING_TRACE
 tr_occurrence++;
 store_event(clock_get_time(),tr_event,tr_occurrence);
#endif

 // TODO : Add your codes for monitoring
 {
        int rdata = *LOCAL_ADDR2( sizeof(void *) + sizeof(int) );

        int delay = 1310;
        unsigned int tr_clk1, tr_clk2;

        tr_clk1 = get_clk_from_elh2(tr_event,tr_occurrence-1);

        tr_clk2 = get_clk_from_elh2(tr_event,tr_occurrence);
        if ( (unsigned int)tr_clk1 <=  (unsigned int)tr_clk2 - delay )
        {
 printf("@sv/satisfy:@d/%d:@v/tr_clk1=%d:@v/tr_clk2=%d:@v/rdata=%d:\n",\
(unsigned int)(tr_clk2-tr_clk1),tr_clk1,tr_clk2,rdata);
        }
        else
        {
printf("@sv/unsatisfy:@d/%d:@v/tr_clk1=%d:@v/tr_clk2=%d:@v/rdata=%d:\n",\
(unsigned int)(tr_clk2-tr_clk1),tr_clk1,tr_clk2,rdata);
                //task_suspend(tid_pro);
                //task_suspend(tid_con);
        }
 }
}
```

5.3 트레이스 맵핑

일단, 테스트케이스의 개발이 완료되면 두 번째 단계로 이들을 컴파일하고 타겟으로 로딩하는 작업을 수행하고 트레이스 맵핑 파일을 작성하여 소스프로그램의 트레이스포인트들과 트레이스함수들을 1:1로 맵핑한다.

먼저, 트레이스 디버거(qptracer)를 통해 테스트할 프로그램이 수행된 상태에서 테스트케이스를 다음과 같이 타겟으로 로딩한다. 현재, 본 실험실에서 개발한 트레이스 디버깅 엔진에는 호스트에서 크로스 컴파일한 오브젝트파일을 타겟으로 로딩/언로딩하는 기능 및 기타 쉘에서 주로 사용하는 명령어 대부분이 구현되어 있다.

```
(qptracer) ld testcase1.o
Module c:/srcqpgdb/gdb/testcase1.o is loaded
```

다음으로, 소스 프로그램에 트레이스포인트와 테스트케이스 파일의 트레이스 함수들과의 맵핑을 시도한다. 트레이스포인트 맵핑은 "trmap" 컴맨드를 통해 이루어지는데, 맵핑 파일은 다음의 맵(map) 파일 예에서와 같이 프로그램 문장의 전 (-s) 혹은 후 (-e)를 나타내는 필드, 파일:라인 필드, 절대 어드레스 필드, 트레이스함수 이름 필드, 그리고 트레이스함수의 파라메터 필드로 구성되어 있다.

[map 파일 예] ==>

```
(rtl-tag, addr, trace-func, trace-args)
@(+pro_msgq_send,i) 0x0 pro_msgq_send_pre *
@(+pro_task_delay,i) 0x0 pro_task_delay_pre *
@(-pro_task_delay,i) 0x0 pro_task_delay_post *
@(-con_msgq_recv_ret,i) 0x0 con_msgq_recv_ret_post *

(pre-post option, filename:line, addr, trace-func, trace-args)
-s pro_con.c:67 0x0 pro_msgq_send_pre *
-s pro_con.c:74 0x0 pro_task_delay_pre *
-e pro_con.c:74 0x0 pro_task_delay_post *
-e pro_con.c:92 0x0 con_msgq_recv_ret_post *
```

이 트레이스 맵 파일은 사용자에 의해 직접 작성이 될 수 있으나 본 논문에서 제시하는 트레이스 디버거에서는 소스프로그램의 RTL-태그를 통해 맵핑파일을 자동으로 생성하는 다음과 같은 "trchklist" 프리프로세서(preprocessor)를 제공한다.

```
(qptracer) trchklist pro_con_testcase.chk pro_con.c
Generate initial trace tag and map file : pro_con_testcase.chk,pro_con_testcase.
map :-
=======================================
  trace_tag  |  file:line  |  trace_func
=======================================
@(+pro_msgq_send,i) pro_con.c:67 pro_msgq_send_pre
@(+pro_task_delay,i) pro_con.c:74 pro_task_delay_pre
@(-pro_task_delay,i) pro_con.c:74 pro_task_delay_post
@(-con_msgq_recv_ret,i) pro_con.c:92 con_msgq_recv_ret_post
```

이 트레이스 점검목록 생성기는 소스프로그램의 RTL-태그로부터 파일이름 및 트레이스포인트 라인정보, 그리고 트레이스함수 이름을 추출하여 앞서 설명한 맵핑파일을 구성한다. 위의 예에서 "testcase1.chk" 파일은 맵핑 파일을 구성하기 위한 임시파일 이름이다.

한편, 테스트 필요에 따라 "trdel" 혹은 "trace" 컴맨드를 통해 특정 테스트 포인트를 삭제 변경 시킬 수 있다. 현재, 설정된 트레이스포인트 정보는 "info tr" 컴맨드를 통해 볼 수 있다. 다음은 이들의 사용 예이다.

[트레이스 맵핑 및 해제] ==>

```
(qptracer) trmap pro_con_testcase.map
Tracepoint 1 with pro_msgq_send_pre at 0x2fb40c: file pro_con.c, line 67
Tracepoint 2 with pro_task_delay_pre at 0x2fb468: file pro_con.c, line 74
Tracepoint 3 with pro_task_delay_post at 0x2fb470: file pro_con.c, line 74
Tracepoint 4 with con_msgq_recv_ret_post at 0x2fb4ec: file pro_con.c, line 92
```

```
Num     Type          Function          Address    What
1       tracepoint pro_msgq_send_pre        0x2fb40c   at pro_con.c:67
2       tracepoint pro_task_delay_pre       0x2fb468   at pro_con.c:74
3       tracepoint pro_task_delay_post      0x2fb470   at pro_con.c:74
4       tracepoint con_msgq_recv_ret_post 0x2fb4ec   at pro_con.c:92
(qptracer) trdel 2
deleted tracepoint 2
(qptracer) info tr
Num     Type          Function          Address    What
1       tracepoint pro_msgq_send_pre        0x2fb40c   at pro_con.c:67
3       tracepoint pro_task_delay_post      0x2fb470   at pro_con.c:74
4       tracepoint con_msgq_recv_ret_post 0x2fb4ec   at pro_con.c:92
(qptracer) trdel 3
deleted tracepoint 3
(qptracer) info tr
Num     Type          Function          Address    What
1       tracepoint pro_msgq_send_pre        0x2fb40c   at pro_con.c:67
4       tracepoint con_msgq_recv_ret_post 0x2fb4ec   at pro_con.c:92
(qptracer) trunmap
deleted tracepoint 1
deleted tracepoint 4
(qptracer) info tr
No tracepoints
```

5.4 실행 및 분석

트레이스포인트들이 설정이 되면 태스트할 프로그램을 실제로 수행시키고 그 결과를 콘솔로부터 수집한다. 테스트할 프로그램은 "qptracer"의 "continue" 컴맨드를 이용하여 실행시킨다.

```
(qptracer) continue
Continuing.
```

"pro_con.c" 프로그램의 경우는 멀티 태스크들이 병행적으로 수행되는 프로그램이기 때문에 적절한 시간동안 발생되는 사건들에 대하여 자료를 관찰하거나 저장해 둔다. 트레이스포인트에서의 타이밍 제약의 만족 되는지 여부에 따라, 오류가 발견되면 개발자는 호스트에서 직접 수행중인 트레이스 기능을 정지시킨 후 트레이스버퍼의 정보를 호스트로 갖고 와서 현재까지 진행된 상황정보에 따라 버그의 원인을 찾는다.

아래의 실행 예는 트레이스 실행 전후의 결과를 보여준다. 다음과 같이 트레이스 수행 전에는 사용자가 응용프로그램에 정의한 메시지가 콘솔을 통해 보여지게 되고, 트레이스 수행 후에는 사용자가 정의한 트레이스 메시지와 함께 트레이스함수의 타이밍 제약의 점검결과를 보여준다. 본 실험 예에서는 두 가지 대표적인 테스트 케이스에 대하여 실험하였는데, 100ms의 마감시간 점검의 경우를 보면 수행 결과가 만족한 결과를 갖고 있으나, 지연시간 테스트 케이스의 적용 예를 보면은 불만족(unsatisfy) 상태를 보여주고 있고 지연시간도 1301~1302ms 정도로 1310ms의 타이밍 제약을 만족시키지 못하고 있음을 알 수 있다.

[트레이스 수행 전] ==>

```
Send_msg: 0, (nmsgs=1)
Rec_msg: 0
Send_msg: 1, (nmsgs=1)
Rec_msg: 1
Send_msg: 2, (nmsgs=1)
Rec_msg: 2
Send_msg: 3, (nmsgs=1)
Rec_msg: 3
Send_msg: 4, (nmsgs=1)
Rec_msg: 4
Send_msg: 5, (nmsgs=1)
Rec_msg: 5
...
```

[트레이스 수행 후] ==>

```
deadline 테스트케이스 적용 시 ==>
Send_msg: 40, (nmsgs=1)
Rec_msg: 40
Send_msg: 41, (nmsgs=1)
Rec_msg: 41
Send_msg: 42, (nmsgs=1)
Rec_msg: 42
Send_msg: 43, (nmsgs=1)
Rec_msg: 43
Sen@sv/satisfy:@d/-1:@v/tr_clk1=314039:@v/tr_clk2=314040:@v/rdata=44:
d_msg: 44, (nmsgs=1)
Rec_msg: 44
Sen@sv/satisfy:@d/-1:@v/tr_clk1=315339:@v/tr_clk2=315340:@v/rdata=45:
d_msg: 45, (nmsgs=1)
Rec_msg: 45
Sen@sv/satisfy:@d/-1:@v/tr_clk1=316639:@v/tr_clk2=316640:@v/rdata=46:
d_msg: 46, (nmsgs=1)
Rec_msg: 46
Sen@sv/satisfy:@d/-1:@v/tr_clk1=317939:@v/tr_clk2=317940:@v/rdata=47:
d_msg: 47, (nmsgs=1)
Rec_msg: 47
Sen@sv/satisfy:@d/-1:@v/tr_clk1=319239:@v/tr_clk2=319240:@v/rdata=48:
d_msg: 48, (nmsgs=1)
Rec_msg: 48
...
```

```
delay 테스트케이스 적용 시 ==>
Send_msg: 227, (nmsgs=1)
Rec_msg: 227
Send_msg: 228, (nmsgs=1)
Rec_msg: 228
@sv/satisfy:@d/764616:@v/tr_clk1=0:@v/tr_clk2=764616:@v/rdata=229:
Send_msg: 229, (nmsgs=1)
Rec_msg: 229
```

```
@sv/unsatisfy:@d/1301:@v/tr_clk1=764616:@v/tr_clk2=765917:@v/rdata=230:
Send_msg: 230, (nmsgs=1)
Rec_msg: 230
@sv/unsatisfy:@d/1302:@v/tr_clk1=765917:@v/tr_clk2=767219:@v/rdata=231:
Send_msg: 231, (nmsgs=1)
Rec_msg: 231
@sv/unsatisfy:@d/1302:@v/tr_clk1=767219:@v/tr_clk2=768521:@v/rdata=232:
...
Send_msg: 247, (nmsgs=1)
Rec_msg: 247
Send_msg: 248, (nmsgs=1)
Rec_msg: 248
Send_msg: 249, (nmsgs=1)
Rec_msg: 249
Send_msg: 250, (nmsgs=1)
Rec_msg: 250
...
```

6. 관련연구의 장/단점 비교

트레이스포인트를 이용한 디버깅 도구들은 여러 종류이다. 소프트웨어 프로그램을 위해서 간단히 개발자 "printf" 문을 프로그램 코드에 직접 삽입하여 디버깅하는 것에서부터 트레이스포인트가 삽입된 코드를 자동으로 생성시켜주는 도구들이 있으며, 하드웨어를 위한 도구로는 논리분석기(Logic Analyzer)와 같이 하드웨어 회로를 직접 찍어가면서 입출력 상태를 가시적으로 보는 것까지 그 종류가 다양하다.

Cygnus에서 제공하는 GNU GDB 5.0[9]의 경우에는 트레이스포인트 디버깅을 위해 바이너리 이미지에 직접 트레이스포인트를 설정하고 원하는 자료를 수집하기 위한 컴맨드들을 제공하고 있으며 프로그램 수행 후 GDB의 명령어들을 이용하여 프로그램을 재실행해 볼 수 있는 특징을 갖고 있다. 본 논문에서 제시하는 트레이스포인트 설정기법에서는 C함수를 직접 트레이스포인트와 연결시켜주지만, GDB 5.0의 경우에는 collect라는 컴맨드를 이용하여 트레이스포인트에서 변수 혹은 레지스터들의 값을 수집할 수 있게 되어 있다. 그러나, 문제는 앞장에서 서술한대로 시간관련 정보를 트레이스 해볼 수 없는 등등 몇가지 존재한다.

Rtview Ltd.의 SurroundView 도구[7]는 실시간에 내장형 어플리케이션을 모니터링하고 디버깅하기 위한 도구로써 실시간 소프트웨어를 디버깅하고 내장형 어플리케이션의 행위를 동적으로 모니터링하기 위해 시각화된 도구를 제공하고 있다. 이 SurrondView도 트레이스포인트 설정기법을 이용하여 시스템을 어떤 브레이크포인트 혹은 멈춤이 없이 실행 시에 변수들에 값을 할당해 가면서 모니터링 해볼 수 있도록 되어있다. 이 SurroundView에서 제공하는 트레이스포인트 설정기법은 간단한 트레이스포인트API를 이용해서 개발자가 직접 소스코드에 추가하여 실행시키도록 되어 있다. 이 기법의 문제점은 소스코드에 테스트용 코드를 삽입하여 수행해야 하기 때문에 본의 아니게 코드의 순수성을 깨뜨릴 위험이 있다.

그리고, ToyLotos/Ada[18]는 LOTOS 언어로 작성된 명세로부터 동일 의미의 Ada 코드를 자동으로 생성하여 수행시켜주는 도구로써 생성된 Ada 코드에 elh_exe()라는 테스트포인트 용 코드를 자동으로 삽입하여 시스템의 행위를 모니터링할 수 있게 하였다. 그러나, 이 도구의 한계는 테스트포인트에서의 정보수집에 제한을 갖고 있는 문제점을 갖고 있다.

한편, 기타 디버거를 위한 것이 아닌 단순 모니터링 기능으로써만 존재하는 것으로써 VxWorks의 WindView[19], pSos+의 Esp 같은 도구들[20]이 존재하는데 이 도구들은 앞서 제시한 도구들과 다르게 트레이스포인트들에서의 값을 가시적으로 보여주고자 하는 것이지 개발자와 대화적으로 값을 변경해 가면서 수행할 수 있는 목적으로 개발된 도구들은 아니다.

7. 결론 및 향후 연구방향

본 논문에서는 멀티 태스킹 환경에서 수행되는 내장형 프로그램의 논스톱 실시간 디버깅을 위한 트레이스포인트 디버깅 기법에 대한 내용을 서술하였다. 이 트레이스포인트 설정기법은 Qplus-T 실시간 운영체제 환경에서GNU GDB를 기반으로 구현되며 프로그램의 실행 중단이 없이 디버깅할 때 사용되는 기술이다. 이 기술은 하드웨어 디버깅 시 시그널들을 모니터링 하기 위하여 사용하는 기법과 마찬가지로 소프트웨어 프로그램 디버깅 시 인스트럭션 레벨에서 시스템의 행위를 모니터링하기 위해 트레이스포인트를 설정하고 필요한 디버깅 코드를 수행시킴으로써 시스템 행위를 분석하기 위해 제공되는 것이다.

이로써, 시스템의 행위는 가시화되고, 수행 중인 프로그램의 어떤 변수들에 어떤 참견이나 본래의 실행 흐름을 멈추지 않으면서도 새로운 값을 할당할 수 있게 하며, 시스템의 논리적인 실행흐름을 관찰할 수 있게 한다. 그리고, 궁극적으로 트레이스포인트들에서 수집된 자료를 기반으로 재실행을 해볼 수 있는 여건을 마련할 수 있는 것이 장점이다.

현재, 본 기술에서는 수집된 자료를 바탕으로 재실행 혹은 도표 등을 통해 그래픽하게 가시화하는 기능은 제공하고 있지 않으나 향후에는 이에 대한 연구를 계속할 예정이다. 그리고, 트레이스포인트에서의 테스트할 내용들을 유틸리티 라이브러리 함수 형태로 제공하여 개발자가 손쉽게 테스트케이스를 작성할 수 있게 할 예정이다.

참 고 문 헌

[1] 김홍남, "사용자개발도구연구", 정보가전용 실시간 OS 컨퍼런스(RTOS'99) 자료집, pp.178-196, Nov. 17, 1999.

[2] Kwangyong Lee, Chaedeok Lim, Kisok Kong, and Heung-Nam Kim "A Design and Implementation of a Remote Debugging Environment for Embedded Internet Software," Proceedings of the ACM SIGPLAN 2000 Workshop on Languages, Compilers, and Tools for Embedded Systems, Jun. 18, 2000, pp.105.

[3] Kwangyong Lee, Chaedeok Lim, Kisok Kong, Heung-Nam Kim, "A Design and Implementation of a Remote Debugging Environment for Embedded Internet Software," Lecture Notes in Computer Science vol. 1985, Springer-Verlag, 2001, pp.199-203.

[4] Hideyuki Tokuda and Makoto Kotera, "A Real-Time Tool Set for the ARTS Kernel," *Proceedings of Real-Time Systems Symposium*, 1988.

[5] T. Yasuda, K. Ueki, "A Debugging Technique Using Event History", *Proc. of the Conference on Real-Time Computing Systems and Applications*, pp. 137-141, 1994.

[6] Jack G. Ganssle, "Debuggers for Modern Embedded Systems," *Embedded Systems Programming*, Nov. 1998.

[7] Eldad Maniv, "New Trends in Real-Time Software Debugging," Real-Time Magazine 99-2(http:// www.realtime-info.com), pp.23-25, 1999.

[8] Jonathan B. Rosenberg, *How Debuggers Work*, John Wiley & Sons, 1996.

[9] Mike Loukides and Andy Oram, Programming with GNU Software, O'REILLY, 1997.

[10] Michael Snyder and Jim Blandy, "The Heisenberg Debugging Technology," http:// sources.redhat.com/gdb /talks/esc-west-1999/INTROSPECT.html, 1999.

[11] Shem-Tov Levi and Ashok K. Agrawala, Real-Time System Design, McGraw-Hill Publishing Company, 1990.

[12] Farnam Jahanian and Aloysius K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," IEEE Trans. on Software Engineering, Vol. 12, No. 9, Sep. 1986, pp.890-904.

[13] Sarah E. Chodrow, Farnam Jahanian, and Marc Donner, "Run-Time Monitoring of Real-Time Systems," Monitoring and Debugging of Distributed Real-time Systems, 1995, pp.103-112.

[14] Sitaram C.V. Raju and Farnam Jahanian, "Timing Constraints Monitoring in Distributed Real-time Systems," Monitoring and Debugging of Distributed Real-time Systems, 1995, pp.356-367.

[15] Jeffrey J.P. Tsai, Yao-Dong Bi, and Steve Jennhwa Yang, "Debugging for Timing-Constraint Violations," IEEE Software, pp.89-99, 1996.

[16] Aloysius K. Mok and Guangtian Liu, "Early Detection of Timing Constraint Violation at Runtime," IEEE, 1997.

[17] Microtec, *Spectra Boot and VRTX Real-Time OS*, 1996.

[18] 이광용, 오영배, "ToyLotos/Ada:실시간 Ada 소프트웨어 개발을 위한 정형적 객체행위 시뮬레이션 시스템", 한국정보처리학회 논문지 제6권 제7호, 한국정보처리학회, 1999.7, pp.1789-1804.

[19] WindRiver, WindView 2.0, http://www.windriver.com/ products/html/windview2.html

[20] WindRiver, pRISM+, http://www.windriver.com/ products/html/prism_ds3.html

이 광 용

1991년 숭실대학교 전자계산학과 (학사) 1993년 숭실대학교 대학원 전자계산학과 (공학석사). 1997년 숭실대학교 대학원 전자계산학과(공학박사). 1996~1998년 숭실대학교 생산기술연구소 연구원. 1997~1998년 ETRI 컴퓨터·소프트웨어기술연구소 소프트웨어공학연구부 박사후연수연구원(Post-Doc.) 1999년~현재 ETRI 컴퓨터·소프트웨어기술연구소 인터넷정보가전연구부 선임연구원. 관심분야는 인터넷정보가전, 내장형S/W, 소프트웨어공학, 실시간운영체제 객체지향 실시간 모델링/시뮬레이션/디버깅 도구, 정형기법

김 홍 남

1980년 서울대학교 전자공학과 학사 1989년 미국 Ball State University 전산학 석사. 1996년 미국 Pennsylvania State University 전산학 박사. 1983년~현재 ETRI 컴퓨터·소프트웨어기술연구소 인터넷정보가전연구부 책임연구원(내장형 S/W연구팀 팀장). 관심분야는 실시간운영체제 비디오압축알고리즘, 분산멀티미디어시스템