Gnu Ada'95 런타임 라이브러리 (GNARL): 태스킹의 구현과 성능향상

문 승 진† · 오 동 익††

요 약

GNARL은 GNAT Ada'95 컴파일러의 런타임 시스템으로, Ada'95언어의 멀티태스킹 기능의 지원을 담당한다. 공개 소프트웨어인 GNARL과 GNAT을 사용하면 프로그래밍언어, 컴파일러, 실시간 기능을 지원하는 런타임 시스템에 대한 연구를 자유롭게 수행 할 수 있는데, 본 연구에서는 GNARL과 GNAT의 총체적인 구조와 구현 원리 및 Ada'95의 멀티태스킹의 효율성, 예측성 및 안정성 향상에 대한 연구 결과에 대해 설명한다.

Gnu Ada'95 Runtime Library (GNARL): Tasking Implementation and Performance Improvement

Seung-Jin Moon† · Dong-Ik Oh††

ABSTRACT

GNARL is the runtime system of the Gnu NYU Ada'95 compiler(GNAT). It implements the multi-tasking features of the Ada programming language and together they provide a freely available test-bed for experimentation in language, compiler, and runtime support for real-time programming. In this paper, we give an overview of GNAT and GNARL. We then describe the results of our research toward improving efficiency, predictability, and reliability of Ada'95 multi-tasking.

1. 서 론

Ada는 국제적으로 표준화 된 객체지향 언어로서, 본래는 군사용 실시간 프로그램 개발을 위해 만들어진 언어이나 일반 응용 프로그램 개발에도 적합성이 입증된 범용 프로그래밍 언어다. 1983년에 그 첫 번째 언어표준이 발표된 이후에 미 국방성 (DoD)에서 만들어지는 모든 분야의 프로그램 개발에 의무적으로 사용해야 했던 언어이며 현재에도 군사관련분야에서 선호되는 언어로서 일반적인 객체지향 프로그래밍 언어들이 포함하는 기능이외에 군사분야 프로그래밍에 적합하도록 고급언어 레벨에서의 병행(병렬) 프로그램기능을 포함하고 있다. 이러한 프로그래밍의 주요 핵심은 태스크라는 개체인데 이는 하나의 독립된 실행의 개체로서 스스로의 상태를 가지고 서로 다른 태스크들과 독립적으로 실행되지만 프로세스의 메모리를 공유한다. 병행(병렬)처리의 기능을 가진 실시간 언어는 Ada이외에도 여럿 있으나 [1,2,3], 실제적인 실시간 응용프로그

※ 이 논문은 1998년도 순천향대학교 학술연구조성비 일반과제연구로 지원을 받아 수행하였음.
† 정 회 원 : 수원대학교 전자계산학과 교수
†† 정 회 원 : 순천향대학교 컴퓨터학부 교수
논문접수 : 1998년 7월 24일, 심사완료 : 1998년 9월 2일

램 개발을 위해 가장 광범위하게 사용되어져 온 언어가 Ada이다.

1995년에 표준화 된 Ada'95 [4]는 원래의 Ada(Ada '83)[5]의 개정판으로서 새로운 객체지향적인 기능과 실시간 기능들을 포함하고 있다. Ada'95에 새로 포함된 기능 중 실시간 태스킹에 관계된 중요한 기능들은 다음과 같다:

- 유동적 태스크 스케쥴링 모델 (flexible task scheduling model)
- 동적 태스크 우선순위 (dynamic priorities)
- 새로운 태스크 동기화 방법 (protected types)
- 우선순위 전승 (priority inheritance)
- 다양한 엔트리 큐잉 방법 (multiple entry queuing policies)
- 정교한 클럭 (high-resolution clock)
- 절대값으로 표현되는 딜레이 (absolute time delays)
- 비동기식 컨트롤 이양 (asynchronous transfer of control : ATC)

Gnu Ada'95 런타임 라이브러리 (Gnu Ada'95 Runtime Library: GNARL)는 GNAT Ada'95컴파일러 [7]의 태스킹과 이의 실시간 기능을 구현하는 기능을 담당한다. GNAT은 Gnu계통의 컴파일러로서 New York University에서 최초 개발되어졌고 현재는 Ada Core Technology(ACT)에서 개발/유지되고 있는 범용 Ada '95 컴파일러이다. GNARL은 Florida State University에서 POSIX/Ada Real-Time 프로젝트의 일환으로 개발되었고 현재는 ACT에서 역시 유지되고 있다. 대학의 연구 프로젝트로 시작한 GNARL개발의 목적은 상용화되는 Ada'95 태스킹의 구현의 방향을 선도하는 것으로서, 이를 통해서 Ada'95태스킹에 대한 연구와 개발에 기초와 도움을 제공하고자 하는 것이었다. GNARL의 구현상의 목표는 시맨틱 상 완전하고 올바르며, 쉽게 포팅 할 수 있고, 또 효율적인 태스킹을 구현하는데 있었다.

대부분의 이러한 개발 초기의 목표들은 달성되었다. GNAT과 GNARL의 소스코드는 이미 공개되어 있고 많은 기계와 운영체제 상에 포팅 되어져 사용되고 있다. 그러나 GNARL의 개발과정에 있어서 방대한 언어 표준의 모든 기능들을 호환성 있게 또 제한된 시간 내에 구현해야 하였기에, 실시간 시스템에 필요한 타이밍 예상 기능을 포함한 태스킹의 효율적인 구현을 제공하는데 어느 정도의 미흡한 점도 있었다. 현재 GNARL이 전체적으로 완성되었고 또 실제적으로 사용되어지고 있지만, 이제는 그 개발 과정상 있었던 미흡한 점을 다시 생각해 보고 또 언어 표준 상에는 명시되지는 않았지만 실시간 태스킹을 위해 필요하다고 생각되는 기능들의 추가가 필요한 때라고 생각되어진다.

이 논문에서 우리는 먼저 Ada'95의 실시간 태스킹 기능들이 GNAT/GNARL에서 어떻게 구현되고 있는지에 이해하기 위해 컴파일러와 런타임 시스템의 구조에 대해서 먼저 설명하고자 한다. 그리고 그러한 구조하에서 어떻게 태스킹의 효율이 향상될 수 있는 지에 관해서 두 가지의 방법을 제시하고자 하는데, 그 첫 번째는 운영체제 시스템 호출의 수를 줄이는 방법이고 또 다른 방법은 언어의 기능을 제한함으로써 전체적인 태스킹의 효율을 늘리는 방법이다, 우리는 본 연구에서 실제적인 GNARL 코드의 변경을 통해 태스킹의 성능향상을 비교하였고 본 논문을 통해 이러한 개선 방향의 타당성에 대해 보고하고자 한다.

이후의 논문은 다음과 같이 구성되어져 있다. 2장에서는 GNAT과 GNARL의 전반적인 구조와 태스킹을 구현하기 위한 그들의 연관관계를, 3장에서는 POSIX 운영체제 상에서 구현 된 GNARL에 대해서 설명한다. 4장에서는 시스템 호출을 줄이는 방법으로서의 태스킹의 성능향상과 제한 (Restriction) 프래그마를 사용한 성능향상에 대해 설명하고 5장에서는 GNARL개발의 현 주소 및 앞으로의 연구 방향에 대해서 설명하고자 한다.

2. GNAT과 GNARL의 구성

GNAT은 Ada'95언어의 컴파일러로서 컴파일러의 front-end를 구현하고 있다. 이 컴파일러의 back-end에는 GCC [8] 코드 생성기가 사용되며 이를 통해서 실제적인 목적코드가 생성된다. GNAT의 front-end는 자체적인 구문분석의 개체를 생성하고 이를 GCC back-end 가 이해 할 수 있는 형태의 개체로 변환시키는데. GCC의 back-end는 이식성이 뛰어나 C뿐만 아니라 C++, Modula3, Fortran, Object-C, Ada'95 컴파일

러의 back-end로도 널리 사용되고 있다. GNAT은 GCC의 back-end를 사용함으로써 GCC의 안정적이고 효과적인 코드생성, 그리고 여러 타겟에 이식되는 범용성을 그대로 답습한다.

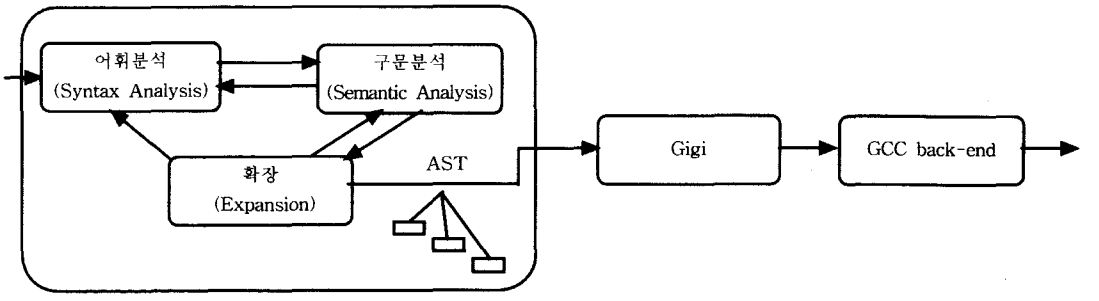
GNAT의 front-end는 4개의 단계로 구성되어 진다. 전체적인 컴파일러의 구성은 (그림 1) 에 나타나 있다. Gigi를 제외한 front-end는 처음에는 아주 제한된 Ada 언어로 쓰여졌으며 컴파일러의 구현이 진전됨에 따라 스스로를 컴파일 하는 방식을 통해 새로운 기능을 반복적으로 추가/사용하여 이제는 완전한 Ada'95 언어로 구현되어져 있다.

GNAT의 여러 front-end의 단계들은 추상 어휘 트리 (Abstract Syntax Tree: AST)를 사용하여 정보를 교환한다. 이 트리는 단순히 어휘에 관한 정보뿐만 아니라 구문에 관한 정보를 교환하는데도 사용된다. 제일먼저 Top-down Recursive 파서가 AST를 생성하고 구문분석단계에서 어휘에 관한 정보가 프로그램에 사용 된 이름과 타입을 지정하는데 사용되어진다. 이 단계에서 AST는 다양한 구문 속성을 부여받게 된다. 확장단계에서는 AST를 GCC back-end가 이해하는 형태로 변환하기가 용이하도록 재구성하는데 이 모든 단계들이 리컬시브하게 반복되어 진다. 소스의 형태에 따라서 단순히 구문의 재구성이 이루어 질 수도 있고 또는 코드의 확장이 일어 날 수도 있다. 이러한 코드의 확장의 예는 런타임 시스템에 필요한 서비스를 요구하는 호출을 생성을 하는 것이 포함된다. AST가 완전히 확장되고 분석되어진 후에는 Gigi가 이를 GCC 트리로 변환하고 이것이 GCC back-end에 주어지어 실제적인 목적코드가 생성되어진다. 더 자세한 GNAT compiler의 구성은 [9]를 참조하기 바란다.

GNARL은 GNAT 컴파일러의 일부분으로서 Ada의 태스킹을 담당하는 역할을 한다. 응용 프로그램의 태스킹에 해당되는 부분은 컴파일러가 직접적으로 그것의 목적코드를 생성하지 않고 태스킹 라이브러리로 독립되어져 있는 모듈에 대한 호출로 변환을 하고 이러한 모듈의 집합체인 GNARL에서 그 해당하는 태스킹에 대한 서비스를 제공한다. 이처럼 컴파일러와 런타임 시스템으로 태스킹을 분리하는 것은 그것의 시맨틱이 컴파일러가 직접 목적코드로 생성하기에는 너무 복잡하기 때문이다. 컴파일러가 생성한 GNARL에 대한 서비스 호출은 응용 프로그램이 링크 될 때 GNARL에서 제공하는 목적모듈과 링크되어 실제적인 프로그램의 실행모듈을 제공하게 된다.

이와 같이 컴파일러가 생성해 낸 서비스 호출들은 GNARL에 의해 태스킹의 기능이 제공되지만, 이러한 호출들이 제대로 작동 될 수 있는 환경을 제공해야 되는 의무는 컴파일러가 가지고 있다. 이러한 환경은 GNARL로 주고 또 되돌려 받는 변수들의 할당 및 초기화, 그리고 그것의 적합한 블록 네스팅등의 설정이 포함된다. 컴파일러와 런타임 시스템의 이러한 역할분담은 컴파일러/런타임의 조합마다 그 정도의 차이는 있을 수 있지만, 태스킹에 대한 역할을 컴파일러와 런타임 시스템의 형태 나누는 것은 일반화 되어있는 구현기법이다.

태스킹을 지원하기 위한 컴파일러와 런타임 시스템의 역할분담에 대한 이해를 돕기 위하여 GNAT이 생성한 중간코드의 예를 살펴보고자 한다. (그림 3)에 나와있는 GNAT의 중간코드는 (그림 2)에 나와있는 태스킹 코드를 컴파일 한 것이다. 대문자로 나타나 있는



(그림 1) GNAT 컴파일러 front-end의 구성
(Fig. 1) Front-End Structure of GNAT Compiler

이름들은 GNARL에 대한 호출인데, 이러한 호출에 대해 GNARL은 적합한 기능을 제공해야 한다. 다른 모든 이름들은 컴파일러가 생성해야 되는 환경정보를 나타낸다. 이 중간코드를 좀더 자세히 살펴보면, 첫째로 프로그램은 ENTER_MASTER라는 GNARL 호출을 함으로써 새로운 마스터레벨을 생성한다. 마스터레벨은 어떠한 블록안에 생성된 태스크의 올바른 종료를 위해서 필요한 정보이다. 다음은 컴파일러가 simple_taskTB 라는 함수로 변형한 태스크의 실행코드 (task body)에 해당하는 내용을 가지고 GNARL의 CREATE_TASK라는 서비스를 호출하게 된다. 그러면 이 호출은 _task_id라는 새로 생성될 태스크의 아이디를 제공하고, 이 생성될 태스크를 _chain이라는 리스트에 삽입하게 된다. 이제는 main 프로그램이 _chain을 가지고 ACTIVATE_TASKS라는 서비스를 호출하여 새로운 태스크의 실제적인 가동을 요구하고 자신은 계속적으로 그 이후의 프로그램 코드를 실행하게 된다. 이제 이 런타임 시스템에 대한 요구가 새로운 실행의 개체를 생성하게 되고 그리하여 main프로그램과 지금 생성된 태스크간에 병행(병렬)적인 실행이 이루어지게 되는 것이다. 자신에게 주어진 코드의 내용을 전부 실행한 후, 두 개의 실행개체 (main과 새로 생성된 태스크)는 "at end" 핸들러에 명시 된 내용을 수행한 후 종료되게 된다.

```
procedure Expansion_Example is
   X : Integer;
   task Simple_Task;
   task body Simple_Task is
      Y : integer;
   begin
      Y := 2;
   end Simple_Task;
begin
   X := 1;
end Expansion_Example;
```

(그림 2) 확장 될 Ada 태스킹 코드
(Fig. 2) Ada Tasking Code to be Expanded

```
procedure expansion_example is
   procedure _clean is
   begin
      COMPLETE_MASTER;
   end _clean;
begin
   ENTER_MASTER;
   _chain : aliased activation_chain := activation_chain'(
     activation_chain'null);
   x : integer;
   task type simple_taskT;
   type simple_taskTV is limited record
      _task_id : task_id;
   end record;
   procedure _init_proc (_init : in out simple_taskTV
   _master :
      master_id; _chain : in out activation_chain) is
   begin
      CREATE_TASK (... _master, simple_taskTB'address,
                   _chain,  _init._task_id ...);
   end _init_proc;
   _master : constant master_id := current_master;
   simple_task : simple_taskT;
   _init_proc (simple_taskTV!(simple_task), _master, _chain);
   procedure simple_taskTB (_task : access simple_taskTV) is
      procedure _clean is
      begin
         COMPLETE_TASK;
      end _clean;
   begin
      y : integer;
      COMPLETE_ACTIVATION;
      y := 2;
   at end
      _clean;
   end simple_taskTB;
   ACTIVATE_TASKS (_chain'unchecked_access);
   x := 1;
at end
   _clean;
end expansion_example;
```

(그림 3) 그림2의 확장된 Ada 중간코드
(Fig. 3) Ada Intermediate Code Expanded Using the Code in Fig.2

GNAT은 GNARL에서 제공하는 서비스에 대한 호출을 확장의 단계에서 생성하고 분석한다. 그 서비스 호출에 대한 형식은 GNARL 모듈 패키지의 스펙 부분에 존재하는데, 컴파일러가 GNARL로 보내야 하고 또 받아들여야 하는 데이터의 타입도 GNARL 묘듈의 스펙을 통해 제공되고 있다. 이러한 타입들에 대한 정보를 컴파일러가 알고 있음으로써 컴파일러는 런타임에 대한 올바른 형태의 서비스 호출을 생성할 수 있다.

3. GNARL의 호환성

POSIX는 응용프로그램에서 사용할 수 있는 운영체제 서비스를 표준화 한 인터페이스의 집합이다. 이러한 표준이 등장한 이유는 기존의 UNIX운영체제가 통합화 되어있지 않아 운영체제의 서비스를 사용하여 작성된 응용프로그램들의 호환성에 많은 문제가 있어왔기 때문이다. POSIX의 표준화는 IEEE에서 주관하고 있으며, 여러 소 분야에 대한 표준을 제공한다. 그 중에는 현재 완성된 것도 있고 또 공인의 과정에 있는 것들도 있는데, 그 중에 제일 먼저 POSIX.1[10] 이라는 표준이 존재하며 이는 일반적인 UNIX형태의 운영체제가 제공하는 기능들에 대한 C언어의 인터페이스를 제공한다. 이외에 POSIX.x로 명명되는 여러 가지 표준이 존재하는데 (예로서 POSIX.5는 POSIX.1에 대한 Ada 인터페이스를 정의한 표준이다), 특히 POSIX.1x로 명명되는 POSIX.1에 대한 real-time extension과 thread extension이 준비되었고 이것은 전체적으로 POSIX.1c [11]라는 이름으로 현재 공인의 과정에 있다. 어떤 운영체제가 POSIX의 인터페이스를 제공한다면 그 환경에서 작성된 응용 프로그램은 원칙적으로 운영체제의 종류에 관계없이 POSIX를 제공하는 운영체제 상에서는 호환성을 가지게 된다.

GNARL은 호환성을 높이기 위해서 POSIX에서 지원하는 운영체제 서비스를 사용하여 구현되어 있다. 또한 POSIX를 제공하지 않는 운영체제에나 또는 POSIX를 완벽하게 지원하지 않는 시스템에도 쉽게 이식 될 수 있도록 설계되어 있으며 이를 위해 몇 개의 중간계층을 제공하는 계층적 구조로 구성되어 있다. GNARL의 각 계층은 자신의 상위계층에 필요한 서비스를 프로시져화 된 인터페이스를 통해 제공한다. (그림 4)는 GNARL의 이러한 계층적 구조를 보여주고 있다.

Ada'95 Application Program
GNU Ada Runtime Layer
GNU Low-Level Layer
POSIX Threads (Pthreads) Layer
Operating System
Machine

(그림 4) GNARL의 계층적 구조
(Fig. 4) GNARL Layers

Pthread 계층을 제외한 상위의 계층들은 모두 Ada로 쓰여져 있다. 그러므로 어떤 타겟에 먼저 GNAT이 포팅 되어지고 또 그 타겟에서 POSIX 스레드 서비스가 제공된다면 GNARL은 쉽게 Ada'95의 태스킹 기능을 제공 할 수 있게 된다.

주어진 환경이 POSIX.1c를 제공할 때 GNARL은 쉽게 포팅 되어 질 수 있지만, 만약 그 주어진 시스템이 해당 인터페이스를 제공하지 않거나 또는 그 표준을 완벽하게 지원하지 않을 경우에는 포팅을 위한 부수적인 작업이 필요하다. 일반적으로 현재의 대부분의 운영체제가 POSIX를 100% 완벽하게 지원하지는 못하는 실정이므로 모든 포팅에 있어서 이러한 부수적 작업은 필수적인데 그러한 서로 다른 운영체제의 특성을 감추기 위해서 GNULL 계층과 그 밑의 계층이 존재한다.

이전의 장에서 언급한 바와 같이 GNARL의 최 상위 계층은 Ada 태스킹의 시맨틱을 구현하고 있다. 이 부분은 Ada로 쓰여 있으므로 기종이나 운영체제에 관계없이 100% 이식성이 있다. 이 계층은 GNAT과 GNARL 간의 인터페이스를 제공하며 응용 프로그램의 태스킹을 관장하는데 이러한 GNARL의 역할에는 태스크의 생성, 실행, 종료, 중단, 랑데부, 상호보호 등과 같은 기능이 포함된다.

POSIX의 서비스를 이용하여 구현된 GNARL은 POSIX에서 제공하는 병행처리의 기본단위인 스레드와 Ada에서 제공하는 태스크를 1대1로 대응시켜 Ada의 병행(병렬)처리를 구현 하고 있다. 하지만 GNARL에서의 Ada 태스킹이 POSIX의 서비스를 단순히 1대1로 사용하는 것은 아니다. 왜냐하면 POSIX에서 제공하는 기능들은 스레드 제어를 위한 기본적인 기능들이며 GNARL에서 제공하는 것은 Ada 태스킹의 다양하고 복잡한 시맨틱에 대한 구현이기 때문이다. GNARL의 구현원리는 POSIX에서 제공하는 기본적인 스레드에 관한 서비스를 가지고 Ada에서 요구하는 다각적이고 고차원적인 태스킹 시맨틱을 제공하는 기능을 담당하는 것이다. 예를 들자면 POSIX는 pthread_mutex_lock()과 pthread_mutex_unlock()을 통해 임계구역에서 스레드들 간의 상호배제를 할 수 있는 기능을 제공하는데, GNARL에서는 이러한 기능을 가지고 태스킹에 필요한 상호배제를 성취한다. 개개의 태스크는 자신의 상태를

위해 많은 정보 (Ada Task Control Block : ATCB)를 가지고 있는데 이러한 정보에는 태스크의 현재상태, 종료를 위한 다른 태스크와의 상관관계 등 Ada의 태스킹 시맨틱을 유지하기 위한 각종의 정보들이 포함된다. 이러한 정보들을 그것을 소유한 태스크를 포함하여 다른 모든 태스크들이 참조하거나 수정할 수 있는데, 하나 이상의 태스크가 이를 사용할 때에 일어 날 수 있는 경쟁조건을 배제하기 위해서 위에서 언급한 POSIX 스레드의 기능들이 사용 될 수가 있는 것이다. 또 하나의 예로서 pthread_cond_wait()과 pthread_con_signal()을 예로 들자면 이는 스레드들 간의 동기화를 위해 POSIX가 제공하는 기능으로서 pthread_cond_wait()을 호출하는 스레드는 컨디션 변수(CV)를 명시하고 블록 되는데, 다른 어떤 스레드가 pthread_cond_signal()을 이 CV에 부를 때까지 블록 되어 있는 상태를 유지하게 하는 기능을 제공하는 서비스이다. 이러한 기능들은 Ada의 rendezvous나 protected operation을 구현 할 때의 기초가 되며, 이들을 사용하기 위한 조건 등은 Ada 시맨틱에 의거하여 GNARL이 구현한다. 정확한 Ada 태스킹 시맨틱의 GNARL구현 알고리즘에 대한 내용은 이미 발표된 논문들을 참조하기 바란다[12,13, 14].

4. GNARL 의 태스킹 성능 향상

현재 GNARL과 GNAT은 Ada'95의 모든 시맨틱을 구현하고 있다. GNAT/GNARL은 Ada Compiler Validation Capability (ACVC)라는 공인의 단계를 이미 1997년에 여러 타겟에서 거친 바 있다. GNAT은 현재, 군사관련 분야의 응용프로그램의 개발 등에 사용되고 있고 그러기에 이제는 GANT의 성능향상에 대한 요구가 대두되고 있는 상황이며 특히 태스킹에 관련된 실시간 응용프로그램의 성능향상을 위해 GNARL의 최적화 과정은 필수적으로 거쳐야 하는 단계이다. Ada'83 컴파일러들이 개발 될 당시에도 이러한 런타임 시스템의 성능향상에 대한 연구가 이루어 진 바 있다[16,17]. 하지만 그러한 연구는 이제 새로운 각도로 분석되어야 한다. 왜냐하면, Ada'95에는 새로운 기능들이 추가되었고 이러한 기능들이 전체적인 런타임 시스템의 수행능력에 영향을 미칠 수 있기 때문이다. GNARL과 같이 운영체제의 기반 위에 작성된 런타임 시스템은 이에 더하여 그 구조적 특성에 해당하는 분석이 또한 요구된다. 이러한 관점에서 볼 때 GNARL의 완성은 새로운 연구와 실험의 토대를 제공하고 있다. 이번 장에서 우리는 두 가지 새로운 성능향상의 방법을 제시하여 분석하고자 한다. 첫 번째 방법은 GNARL의 POSIX서비스 호출의 수를 줄이는 방법이고, 두 번째는 GNARL의 기능 축소를 통한 전체적인 태스킹 수행기능의 향상을 가져오는 방법이다.

4.1 시스템 호출을 줄임으로서 얻는 성능향상

GNARL이 운영체제 위에 계층적으로 구현되어 있는 관계로 GNARL의 수행속도는 운영체제 서비스를 얼마나 자주 요구하는가에 많이 좌우된다. 왜냐하면, 운영체제 서비스는 보통 커널 서비스를 요구하므로 커널로 진입할 때 필요한 문맥전환의 오버헤드가 유저공간에서 수행되는 코드에 비해 매우 크기 때문이다. 이러한 관찰을 토대로 우리는 GNARL이 사용하는 운영체제 서비스가 실제로 얼마만큼의 오버헤드를 부과하는지에 대한 타이밍을 측정하였다. 운영체제 호출의 타이밍은 그것을 측정하는 시스템에 따라 다를 수 있기에, 다양한 실험의 결과를 얻기 위하여 그 타이밍의 측정을 3가지 다른 환경에서 수행하였다. 이러한 측정은 Florida State University에서 개발한 Pthread (FSU Pthreads)[18]를 SunOS (version 4.1.4)운영체제에서, 또 하나는 Solaris 운영체제 (version 2.4)에서 제공하는 스레드 라이브러리를, 그리고 세 번째는 Solaris2.4에 FSU Pthread를 사용하여 실행하였다. 모든 실험은 스레드 라이브러리를 포함한 운영체제의 종류에는 차이가 있지만 동일한 기종에서 수행되었다.

이 결과를 토대로 우리는 어떤 부분의 POSIX 호출을 줄이는 것이 가장 효과적인지를 알 수 있게 되었는데 모든 환경에서 Clock (gettimeofday()), Signal_Masking (pthread_sigmask()) 그리고 Wakeup (pthread_cond_signal())의 오버헤드가 높은 것으로 나타났고, Self (pthread_getspecific()), Set_Priority (pthread_setprio()), Lock/Unlock (pthread_mutex_lock()/unlock()) 등은 어떤 환경에서는 오버헤드가 높은 것으로 확인되었다. 이러한 오버헤드가 높은 시스템 호출의 수를 줄이는 노력은 GNARL에서 이미 수행되어 발표된 바 있다[19]. 그러나 어떤 특정한 시스템 호출의 삭제가 특수한 상황에서만 유용한 경우가 있음을 우리는 이 연구의 과정에서 발견하게 되었다. GNARL에서 사용하

는 Lock/Unlock 호출이 바로 이러한 범주에 속하는 시스템 호출이다.

GNARL에서는 병행(병렬)적인 읽기/쓰기에 있어서 데이터의 일관성을 보장하기 위해서 상호배제를 위한 Lock/Unlock 기능을 사용한다. GNARL에서는 일반적으로 이전의 Ada'83 컴파일러에서 사용한 회수보다 더 많은 Lock/Unlock 서비스를 사용하는데, 이는 이전의 런타임 시스템들은 단일 CPU 상에서 수행되는 것을 전제로 어떤 태스크가 런타임 시스템의 서비스를 받게 될 때 다른 모든 태스크들의 런타임 시스템 접근을 막도록 설계되어 있기 때문이다. GNARL은 다중 CPU의 기계를 생각하여 구현되었는바, 런타임 시스템 서비스 기간동안에 꼭 필요한 태스크들 간의 상호배제만 실현하도록 구현되어 있다. 즉, 이전의 대부분의 런타임 시스템들은 하나의 Lock을 가지고 런타임 시스템 코드를 수행하는 태스크가 어떤 주어진 시간에 하나만 존재하도록, 한 태스크가 런타임 시스템을 들어가면 Lock을 잠그고 런타임 시스템에서의 수행을 마칠 때 Lock을 푸는 형태로 구현되어 있다(이를 단일잠금 방식이라고 하자). 반면에 GNARL은 각각의 태스크의 ATCB에 Lock 변수를 하나씩 할당하고 이를 사용하여 필요한 태스크 사이에서만 상호배제를 이루는 형태를 취하고 있다(이를 다중잠금 방식라고 하자).

다중잠금 방식을 사용하면 우리는 더 높은 병행(병렬)성을 제공하게 된다. 기존의 관념에 의하면 더 높은 병행(병렬)성은 유익하다. 왜냐하면, 다중 CPU상에서 응용프로그램이 실행되는 경우라면 병렬처리가 가능하고, 비록 단일 CPU에서 병행처리를 한다고 해도 우선순위가 높은 실시간 태스크에 빠른 응답시간을 제공할 것이기 때문이다.

하지만 유의할 것은 병행(병렬)성과 시스템 호출의 오버헤드는 사용 중인 환경의 특성과 응용 프로그램의 특성에 따라서 그 효율의 플러스/마이너스가 결정 될 수 있다는 사실이다. 만약 런타임 시스템 내에서의 병행(병렬)성이 필요가 없다면 우리는 GNARL의 많은 Lock/Unlock 호출을 제거하여 태스킹의 성능향상을 도모 할 수가 있을 것이다. 이러한 상관관계가 실제적으로 어떻게 작용하는지를 좀더 정확히 알기 위해서 우리는 단일잠금 방식을 사용하는 GNARL을 구현하여 보았다. 그리고 두 가지의 GNARL 버전을 사용하여 CPU의 수와 태스크간의 상호작용의 정도를 변화해 가면서 각각의 런타임 시스템의 성능을 측정하였다. 성능의 측정을 위해서는 시스템을 단일 사용자 환경에서 가동하여 다른 시스템 작업의 간섭을 최소한 상태에서 dual-loop 방식을 사용하여 반복적으로 안정적인 결과들을 얻고 이의 평균값을 사용하였다.

단일 CPU 상에서 단일잠금 방식을 사용하여 줄일 수 있는 Lock/Unlock의 수는 태스크가 요구하는 런타임 시스템 서비스의 종류에 따라 다르다. 대개의 런타임 시스템 서비스들은 다중잠금 방식을 사용할 경우에 1개에서 3개의 Lock/Unlock 서비스의 쌍을 요구한다. 태스크의 종료에 관계된 서비스는 보통 10개의 Lock/Unlock 서비스 쌍을 요구하고, 태스크의 시동을 위해서는 시동되어지는 태스크의 숫자만큼의 서비스 쌍이 요구된다. 그러므로 우리는 종료에 해당하는 서비스를 태스크가 요구 할 때에 단일잠금 방식을 사용한다면 단일 CPU 환경에서 가장 높은 성능향상을 얻을 수 있을 것이라고 기대하였다.

다중 CPU 상에서도 다중잠금 방식이 효과적이지 않는 상황이 있을 수 있다. 응용프로그램의 특성상 각기 다른 CPU에서 실행되고 있는 태스크들이 한꺼번에 런타임 시스템의 서비스를 요구 할 확률이 적다면 다중잠금 방식을 사용하여도 응용프로그램의 수행속도는 향상되는 것이 없을 것이다. 이론적으로 여러 태스크가 동시에 런타임 시스템 서비스를 요구하고 그 태스크들 사이의 상호관계가 (서로 다른 태스크의 데이타에 접근하는 것) 별로 존재하지 않을 경우에만 다중잠금 방식을 사용하여 얻을 수 있는 장점을 예상 해 볼 수 있을 것이다. 예로서, A라는 태스크가 B라는 태스크와 자주 rendezvous를 하고 또 태스크 C가 태스크 D와 자주 rendezvous를 하지만 A,B 쌍과 C,D 쌍 사이에는 별다른 관계가 존재하지 않는 경우를 생각해 볼 수 있을 것이다. 이러한 경우에 다중잠금 방식은 A,B와 C,D 사이에 완전한 병렬성을 제공하게 될 것이다. 반면에 단일잠금 방식을 사용할 경우에는 모든 태스크들은 런타임 시스템의 서비스를 받는 동안에 순차적인 수행을 하게 될 것이다. 하지만 이 경우에도, 다중잠금 방식을 사용하여 얻는 이득은 하나의 런타임 시스템 서비스를 받는 시간 이상이 될 수는 없다. 만

약 다중잠금 방식을 구현하기 위한 오버헤드가 런타임 시스템 서비스의 수행시간을 두 배로 만든다면 다중잠금 방식을 사용하는 것에는 어떠한 장점도 존재하지 않을 것이다.

이러한 추측을 확인하기 위해서 우리는 단일잠금 방식과 다중잠금 방식을 사용한 GNARL의 성능을 다중 CPU 상에서 비교해 보았다. 우리가 실험을 수행한 기계는 4개의 동일한 성능을 가진 CPU를 장착한 SPARCstation20로서 Solaris 2.5.1운영체제를 사용하는 기계이며 (그림 5)에 이러한 비교를 위해 사용한 응용 프로그램의 소스를 기술하였다. 이 프로그램은 쌍으로 이루어진 태스크들을 가지고 있고, 이 쌍에 한해서만 rendezvous가 일어나고 다른 쌍들간에는 서로 상호관계가 없는 프로그램이다. 우리는 이 프로그램의 성능 측정을 통해 런타임 시스템에 대한 서비스 요청이 빈번할 때, 다중잠금 방식을 사용하는 경우가 단일잠금 방식을 사용하는 경우보다 더 좋은 성능을 나타내리라 기대하였다.

```
declare
   task type Server is                    -- paired tasks;
      entry Go;                           -- Server and Caller
   end Server;
   task type Caller is
      entry ID (Num : integer);
   end Caller;
   task body Server is
   begin
      for I in 1 .. Iteration loop
         accept Go;                       -- accept for
      end loop;                           -- Iteration times
   end Server;
   task body Caller is
      Server_ID : integer;
   begin
      accept ID (Num : integer) do        -- get proper Server ID
         Server_ID := Num;
      end;
      for I in 1 .. Iteration loop
         Servers (Server_ID).Go;          -- call proper task for
      end loop;                           -- Iteration times
   end Caller;
   -- Num_of_Pairs of Server and Caller
   Servers : array (1 .. Num_of_Pairs) of Server;
   Callers : array (1 .. Num_of_Pairs) of Caller;
begin
   for I in 1 .. Num_of_Pairs loop
      Callers (I).ID (I);                 -- assign ID for pairs
   end loop;
end;
```

(그림 5) GNARL 성능 측정을 위한 독립적 rendezvous 코드
(Fig. 5) Rendezvous Code Used for Performance Evaluation of GNARL

<표 1>에서 보고된 타이밍 측정의 결과는 런타임 시스템에 대한 서비스 요구가 늘어남에 따라 다중잠금 방식을 사용하는 것이 더 효과적임을 보여주고 있다. 이 결과를 도출하기 위해서 우리는 기계에 주어진 4개의 CPU 전부를 사용하여 태스크 쌍의 수를 점점 늘려나갔다. 각 쌍의 rendezvous의 회수는 1000번으로 고정하였고, 테이블에 나와있는 시간은 전체 프로그램의 시작에서 종료까지의 시간이다.

〈표 1〉 4개의 CPU상에서 1000 rendezvous를 측정한 결과 (시간은 초)
〈Table 1〉 Times for 1000 Rendezvous on 4-processors, in seconds

Rendezvous 쌍의 수 =>	2	4	8	16	32
단일잠금 방식 GNARL	0.57	3.29	9.60	22.85	50.56
다중잠금 방식 GNARL	0.67	0.81	1.68	3.51	7.32

그런데, Solaris 시스템에서 최적의 성능을 얻기 위해서는 우리는 아주 중요한 한가지 사실을 염두에 두어야 한다. 이것은 병행의 정도 (degree of concurrency)에 관한 것인데, "연산수행 태스크" ("연산수행 태스크"란 수행하는 작업이 런타임 시스템의 서비스, 특히 Sleep/Wakeup을 요구하는 것에 비해 아주 많은 시간을 차지하는 태스크)를 위해서는 충분히 많은 LWP(커널 레벨에서 스케쥴링되는 Solaris 운영체제의 병행(병렬)처리의 단위)를 가동시켜 놓는 것이 중요하다는 것이다. 이는 Solaris 운영체제가 LWP의 숫자 만큼만의 병렬성을 제공하기 때문인데, 비록 4개의 CPU가 있다고 하여도 LWP를 1개만 가동시킨다고 하면 태스크들은 전혀 병렬적인 수행을 이룰 수가 없게 Solaris 운영체제는 설계되어 있다. 4개의 CPU를 가진 기계에서 4개 CPU간의 병렬성을 원한다면, 우리는 최소한 4개의 LWP를 가동시켜 놓아야 하는데 이러할 때 4개의 스레드가 각기 다른 LWP상에서 수행됨으로써 병렬성을 갖게 되기 때문이다. <표 2>에는 4개의 "연산수행 태스크"를 가진 프로그램의 수행속도가 가용될 수 있는 LWP의 숫자에 따라 영향을 받고 있음을 보여주고 있다. 한 개의 LWP에서 2개의 LWP로 그 수를 늘릴 때 전체 프로그램의 실행시간은 반으로 줄었다. 2개에서 3개의 CPU로 늘릴 때는 아무런 변화가 없는데 이는

아직 2개의 태스크가 하나의 CPU를 공유해야 하기 때문이다. 마지막으로, 4개의 LWP를 사용할 때에 수행시간은 다시 반으로 줄어들게 되는데 이는 모든 태스크들이 독립적인 CPU에서 실행되기 때문이다.

〈표 2〉 4개의 "연산 수행 태스크"의 수행속도 (loop을 1000번 반복. 시간은 초).
〈Table 2〉 4 Compute-Bound Tasks (1000 iterations), in seconds.

LWP의 수	1	2	3	4
	0.96	0.48	0.47	0.23

태스크들이 "연산수행 태스크"가 아닐 때 우리는 더욱 더 그 결과에 주의를 기울여야 한다. 왜냐하면 동기화를 위한 운영체제 서비스의 호출 오버헤드가 서로 다른 LWP에 있는 스레드의 경우에 같은 LWP에 있는 스레드의 경우 보다 훨씬 높기 때문이다. 이는 LWP간의 교신(동기화를 위한 시그널링)은 실제적으로 매우 비싼 커널 서비스를 동반하는 반면, 같은 LWP상에서의 스레드간의 교신은 운영체제 커널의 중재 없이 이루어지기 때문이다. 그러므로 어떤 응용 프로그램에 존재하는 태스크들이 상호간에 통신을 자주 한다면 LWP의 수를 증가시켜 병렬의 정도를 늘리는 것이 실제적으로 전체적인 수행효과를 감소시키는 결과를 가져 올 수도 있는 것이다.

<표 3>에서 우리는 Solaris 운영체제 상에서 태스크들이 "연산 수행 태스크"가 아닐 경우에 다중 CPU를 사용하면, 오히려 좋지 않은 수행결과가 나타남을 볼 수 있다. 이 표의 결과는 <표 1>의 결과를 얻기 위해서 사용한 것과 같은 프로그램 (그림 5)을 수행한 결과이다. 하지만 이번에는 태스크 쌍의 숫자를 4개로 고정시키고 대신 LWP의 숫자를 변화시킨 결과를 실험하였다. 한 개 이상의 LWP를 사용할 경우 수행성능이 감소하는 결과를 가져왔는데 이는 LWP가 한 개 이상 일 때 rendezvous가 LWP간의 교신을 요구하게 되기 때문이다.

이러한 결과는 응용프로그램을 사용하는 사람들이 다중 CPU 상에서 한 개 이상의 LWP를 사용하여야 하는지에 대해 신중한 고려를 하여야 한다는 것을 의미한다. 사용자들은 thr_setconcurrency()라는 시스템

〈표 3〉 4개의 rendezvous를 1000번 수행하는 태스크의 쌍 (시간은 초).
〈Table 3〉 4 Pairs of Communicating Tasks (1000 iterations), in seconds.

LWP의 수	1	2	3	4
	0.26	0.69	0.68	0.82

호출을 사용하여 LWP의 수를 동적으로 조절 할 수 있고, 이 논문에서 수행한 실험과 같은 과정을 거쳐서 실제로 병렬 프로그램의 자신의 프로그램의 성능향상에 도움이 되는지에 대해서 살펴보아야 한다. Solaris 스레드와 LWP에 대한 더 자세한 내용은 [20]을 참조하기 바란다.

만약 다중 CPU 상에서 실제적인 병렬처리의 효과가 없어서 단지 한 개의 LWP만 사용한다면, 이미 지적된 바와 같이 런타임 시스템 구현에 있어서 단일잠금 방식을 사용하는 것이 더 유용할 것이다. "연산 수행 태스크"를 가진 응용 프로그램의 경우에도 단일잠금 방식은 사용 될 수 있을 것인데 이는 이러한 태스크들의 경우에, 런타임 시스템 서비스를 요청하는 부분이 연산을 수행하는 부분에 비해 아주 작은 부분만을 차지 할 것이기 때문이다. 즉, 이러한 경우에 런타임 시스템 서비스의 병렬성에 의해 생기는 효과는 거의 감지하지 못할 정도의 수준에 머물 것이기 때문이다.

이러한 실험의 결과를 통해 볼 때에 Solaris의 환경에서는 런타임 시스템의 구현을 위해 단일잠금 방식을 사용하는 것이 프로그램의 일반적인 성능향상을 위해 더 효과적인 방법일 것으로 생각된다. 이러한 결론은 아주 흥미로운 것인데 왜냐하면 일반적으로 병렬의 정도가 높을수록 multi-thread된 런타임 시스템의 효율이 좋을 것이라고 일반적으로 생각 되어져 왔기 때문이다.

4.2 제한 프래그마를 이용한 런타임 시스템 성능 향상

시스템 호출의 숫자를 줄이는 방법은 런타임 시스템의 수행 속도를 향상하는데 유용하게 사용될 수 있다. 하지만 이와 같은 방법을 통한 성능향상에는 그 한계가 있을 수밖에 없다. 이 단원에서 우리는 런타임 시스템기능의 제한을 통한 또 하나의 성능 향상의 기법에 대해 논 하고자 한다.

Ada'95는 "제한"(Restriction: ARM 13.12)[4,6] 이라는 새로운 프래그마를 허용하고 있다. 이 프래그마는 Ada의 복잡한 기능들을 축소하는데 사용되며, 간략한 기능을 원하는 사용자들을 위하여 축소 된 기능을 제공하는데 사용되어 질 수 있다. 사용자가 좀더 효율적이고 안정된 컴파일러를 원할 때, 사용자와 컴파일러 제작자간에 Ada 언어의 부분집합만을 제공하는 것에 합의를 할 수 있게 하고, 이러한 부분적인 기능을 사용하는 응용프로그램은 이 프래그마를 사용하여 언어 기능을 제한할 것을 명시 할 수 있다. Ada'95에 이러한 새로운 프래그마가 등장한 것은 Ada가 오직 하나의 언어 표준만을 고집하였던 것에서부터 그 근본적 사고를 변경한 것을 의미한다. Ada'83은 오직 하나의 표준으로 존재하여 모든 Ada 컴파일러는 100% 같은 신택스와 시맨틱을 제공함으로써 언어표준의 다양성으로 인한 응용프로그램의 비 호환성을 배제하는 점을 강조하여 왔다. 하지만 Ada'95는 언어의 표준화 단계에서 언어의 부분집합을 허용해야 하는가 아닌가에 대한 논란이 많이 있었다. 부분적인 언어를 허용해서는 안 된다는 주장은, Ada가 너무 방대하여 단순화하여야 한다는 여론과 또 새로운 기능들을 첨가해 더욱더 언어를 불려야 한다는 여론을 동시에 수용하기 위해서 양보되어 질 수밖에 없게 되었다. 유일한 언어 표준을 잃게 된다는 데에 대한 염려는, 제한적인 언어를 제공하는 컴파일러는 항상 언어 전부를 제공하는 컴파일러와 함께 제공되어서 이 "제한" 프래그마를 사용하지 않는 응용 프로그램은 언제든지 호환성을 유지할 수 있고, 또 응용프로그램이 "제한" 프래그마를 사용한다 하더라도, 주어진 컴파일러가 그 프래그마에 명시된 모든 제안 사항에 대한 구현을 제공하지 못하는 경우에는 항시 언어 전부를 제공하는 컴파일러 버전을 사용하게 함으로써, 응용프로그램의 비호환성 문제에 대한 우려를 잠재울 수가 있었다.

제한된 언어를 정의하는 방법은 두 가지의 관점에서 생각해 볼 수 있다. 먼저, 런타임 시스템 구현자가 효과적이고 안정적인 태스킹을 위해 필요하다고 생각되는 기능들을 제한 사항에 포함케 하는 방법이다. 두 번째 방법은 응용프로그래머들이 원하는 사항을 바탕으로 하여 제한된 런타임 시스템을 만드는 방법이다. 그러나 후자의 방법에는 문제점이 있다. 왜냐하면 이 방법을 택할 경우에 사용자들이 제한하고자 하는 각각의 기능들의 모든 조합을 위해 수많은 런타임 시스템의 버전을 제공해야 하기 때문이고, 그 숫자는 지수적으로 늘어나기 때문에 실용적인 방법이 될 수 없다.

Ada'95 언어 중 태스킹에서 제한 될 수 있는 몇 개의 기능들은 언어 표준에 제시되어 있다. 하지만 이것은 권장사항이지 어떤 구속력을 가지는 것은 아니다. 런타임 시스템 구현자는 원하고자 하는 성능(그 성능의 기준이 시간적인 효율, 또는 안정성 또는 다른 기준이 될 수도 있겠지만)을 향상시키기에 적합한 몇 개의 기능을 제한하는 런타임 시스템을 제공하면 될 것이다. 물론 이러한 제한 된 기능들이 제외되고도 사용자가 응용 프로그램을 구현 할 수 있는 어느 정도의 언어기능의 수준은 제공해야 한다.

GNARL의 제한을 위해 우리는 "비동기식 컨트롤 이양" (asynchronous transfer of control: ATC)과 "태스크 삭제"(task abortion)기능을 "제한" 집합에 넣을 기능들로 우선 선택하였다. 이유는, 이 두 가지의 기능을 제공하기 위해서 런타임 시스템 구현 전반에 걸쳐 많은 오버헤드가 생기기 때문이고, 그에 더하여 이 기능들은 자체로도 성능 상 오버헤드가 매우 크기 때문인데, 이 기능들이 POSIX의 시그널 서비스와 Ada의 exception을 통해 구현되었는바, 이를 위해 문맥의 저장, 복구, 재가동이 수반되기 때문이다. 한 단계 더 나아가 ATC의 호출/취소의 기능은 네스팅 될 수 있어서 이를 유지하기 위한 정보를 다루는 기능이 런타임 시스템전체에 걸쳐서 유지되어야 한다. 또한 이러한 기능들을 축소하는 것은 많은 응용 프로그램의 입장에서도 적합한데, 왜냐하면 특별히 이러한 기능을 사용하는 응용프로그램의 비중은 전체 응용프로그램에 대비해 생각해 볼 때 그다지 크지 않기 때문이다. "태스크 삭제"는 Ada'83부터 존재하던 기능으로 많은 사람들이 이 기능의 시맨틱이 너무 복잡하고 비 정형적이어서 실시간 시스템이나 안정성에 기반을 둔 응용 프로그램에 사용하기에 부적합하다는 의견을 제시 하여왔다. 마찬가지로 ATC 기능을 Ada'95에 첨가하느냐를 놓고 언어 표준화 단계에 많은 논란이 있었다.

ATC의 호출/취소의 기능은 "태스크 삭제"를 일반화 한 기능이다. 여러 단계로 호출이 네스팅 될 수 있는 ATC는 "태스크 삭제"의 시맨틱을 확장 한 기능이

다. 새로운 ATC 호출이 실행 될 때마다 GNARL에서는 그 호출을 요구하는 태스크(호출자)의 ATCB에 새로운 호출에 대한 계층을 설정하고 (ATC_Nesting_Level) 그에 대한 정보를 유지한다. 이러한 호출 계층 대한 정보는 그 호출이 취소 될 때 호출자의 호출 정보를 삭제하기 위한 자료로 사용되어 진다. 호출자는 취소의 요청을 받게 될 때, 만약 그 호출이 아직도 유효하면 필요한 클린업의 과정을 거치고 아니면 그러한 요구를 단순히 무시하게 된다. 호출 계층 0은 특별한 의미를 가지고 있으며 태스크 자체를 삭제하는 것을 의미한다.

ATC의 호출/취소의 기능은 GNARL에서는 POSIX의 시그널 서비스를 사용하여 구현되어져 있다. POSIX가 설정한 여러 가지의 시그널 중에 이러한 기능을 위해서 하나의 시그널이 사용된다. 즉 호출을 취소하거나 태스크를 삭제하여야 할 때 그 대상 태스크는 Abort_Task_Interrupt 시그널을 받게되고, 그 태스크에 정의된 시그널 핸들러가 수행되어서 호출 취소나 태스크 삭제의 기능이 시작되게 된다.

태스크의 수행 도중 언제나 시그널을 받아 처리할 수는 없다. 왜냐하면 취소나 삭제가 항상 안전하지는 않기 때문이다. 예를 들자면 어떤 태스크가 어떤 Lock 변수를 잠근 상태로 실행되는 도중에 시그널을 받고, 자신의 실행이 중단된다면 태스크의 삭제를 담당한 태스크가 중단된 태스크의 자료를 전혀 접근 할 수가 없게 되어 교착상태가 일어나게 될 것이기 때문이다. 이와 같이 GNARL의 수행도중 Abort_Task_Interrupt를 처리해서는 안 되는 부분이 있는데, 이를 우리는 삭제연기(abort-deferred)구역이라고 이야기한다. GNARL에서는 각각의 태스크의 ATCB에 Deferral_Level 이라는 항목이 있어서, 어떤 태스크가 Abort_Task_interrupt를 현재 처리하는 것이 안정적인지 아닌지에 대한 정보를 유지하고 있다. 만약 현재 시그널을 처리를 할 수 없다면 삭제연기 구역을 벗어날 때 Deferral_Level이 0으로 변하며 그때에 비로서 시그널에 필요한 일들이 수행되어지게 된다. 이 수행되는 작업은 Ada exception을 일으키어 현재 수행되는 코드의 구역을 벗어나는 작용을 하고 그 구역을 벗어나 수행해야 하는 클린업과 같은 일들은 GNARL코드에 제공된 exception 핸들러에서 담당하게 된다.

ATC와 "태스크 삭제"는 태스킹에 있어서 아주 유용한 기능임에는 틀림이 없다. 특히 ATC는 실시간 시스템의 불완전 연산(Imprecise Computation)등을 구현하는데 아주 적절히 사용 될 수 있다. 하지만 POSIX에서 제공하는 시그널 기능은 문맥전환을 수반하는 오버헤드가 아주 높은 서비스이다. 또한 Ada의 exception은 goto와 마찬가지로 longjmp() 시스템 호출을 수반한다. 때문에 ATC의 전체적인 오버헤드는 높을 수 밖에 없다. 게다가 태스크 삭제연기 구역을 보호하기 위해 GNARL 전체에 defer_abortion/undefer_abortion 이라는 함수에 대한 요청이 산재해 있어 이에 따르는 오버헤드가 런타임 시스템의 전반적인 수행에 영향을 미치고 있다.

우리는 ATC와 "태스크 삭제"의 기능을 제한한 또 다른 GNARL의 버젼을 만들어 이러한 기능이 GNARL 전체의 성능에 미치는 영향을 제거 할 수 있었다. 특히 처리되지는 않았지만 이미 유효한 취소/삭제명령을 매번 검사해야 되는 번거로움에서 벗어나게 되었다.

〈표 4〉 PIWG 프로그램 개요
〈Table 4〉 PIWG Programs

프로그램 이름	PIWG Internal Name	프로그램 개관
C1	C000001	Create & Terminate Task -- Task Type used in Procedure
C2	C000002	Create & Terminate Task -- Task Declared in Procedure
C3	C000003	Create & Terminate Task -- Task in Declare Block of Main
T1	T000001	Minimum Rendezvous -- Entry call and return time
T2	T000002	Rendezvous -- Single library-level task, one entry
T3	T000003	Rendezvous -- Two library-level tasks, one entry/task
T5	T000005	Rendezvous -- Ten library-level tasks, one entry/task
T6	T000006	Rendezvous -- One library-level task, ten entries
T7	T000007	Minimum Rendezvous -- No accept body

이러한 제한된 GNARL의 버전을 가지고, 과연 ATC와 "태스크 삭제"를 지원하기 위한 GNARL의 전체의 비용은 얼마만큼 되는가를 알아 볼 수 있었는데, 다음의 비교 결과는 PIWG과 ACES[15] 라는 공인된 Ada 벤치마크 테스트를 통해서 얻은 자료이다.

<표 5>는 여러가지의 태스킹 기능들을 PIWG 테스트를 통해 비교한 자료이다. 개별적 테스트에 대한 간략한 설명은 <표 4>에 나열하였다. 이 자료에 의하면 rendezvous의 성능은 약 17%까지 향상되었고, 태스크 생성과 종료에서는 많은 차이점이 나타나지 않았는데 이는 태스크 생성의 시간이 너무 커서 향상의 결과가 돋보이지 않았기 때문이다.

<표 7>은 Ada'95 컴파일러의 성능을 비교하기 위해 작성된 ACES 벤치마크 테스트를 사용하여 태스킹의 성능을 비교한 결과이다. 각각의 테스트에 대한 설명은 <표 6>에 제공되어 있다.

결과에서 볼 수 있듯이 protected type에 관한 테스트에서 rendezvous에 관한 테스트의 경우보다 제한된 GNARL 버전을 사용하는 경우에 더 좋은 효과를 얻을 수 있었다. 이는 protected type 기능을 수행하는 GNARL의 코드는 원래 오버헤드가 적어서 단순화된 런타임 시스템의 코드로 인해 얻는 효과가 상대적으로 돋보였기 때문이다. 성능향상은 일반적으로 커서 약 30%까지의 향상을 얻을 수 있었다.

5. 결 론

이 논문의 전반부에서는 GNAT/GNARL의 전체적인 구성과 태스킹을 위한 상호 작용, 또 GNARL의 POSIX상에서의 구현에 대해 설명하였다. GNARL은 GNAT 컴파일러의 태스킹을 담당하는 라이브러리로서 컴파일러가 태스킹을 위해 생성하는 서비스에 대한 인터페이스와 그에 부수적으로 필요한 환경자료 및 데이터의 타입을 제공한다. 이러한 컴파일러/런타임 시스템의 분리 형태는 태스킹의 시맨틱이 컴파일러가 인라인 코드로 생성하기에는 너무도 복잡하기에 일반적으로 사용되고 있다. GNARL은 Ada'95 태스킹에 필요한 시

〈표 5〉 PIWG 벤치마크 결과 (시간은 초)
〈Table 5〉 PIWG Benchmarks, in seconds

프로그램 이름	SunOS (FSU Threads)			Solaris (Native Threads)			Solaris (FSU Threads)		
	with ATC	no ATC	Gain	with ATC	no ATC	Gain	with ATC	no ATC	Gain
C1	6986	6970	0.23%	1199	1145	4.50%	8564	8471	1.09%
C2	7001	6998	0.04%	1203	1144	4.90%	8280	8041	2.89%
C3	7020	6988	0.46%	1218	1154	5.25%	8290	8094	2.36%
T1	345	321	6.96%	420	390	7.14%	220	192	12.73%
T2	397	370	6.80%	403	394	2.23%	223	190	14.80%
T3	353	332	5.95%	419	382	8.83%	245	203	17.14%
T5	357	330	7.56%	445	433	2.70%	238	202	15.13%
T6	519	490	5.59%	514	510	0.78%	404	376	6.93%
T7	320	303	5.31%	386	378	2.07%	209	187	10.53%

〈표 6〉 ACES 프로그램 개요
〈Table 6〉 ACES Programs

프로그램 이름	ACES Internal Name	프로그램 개관
A1	tk_rz_task_01_num_01	Rendezvous with one task
A2	Pt_Po_Prot_Access_01	PO access and change object
A3	Pt_Po_Prot_Access_02	PO access and change object:5 tasks
A4	PT_PO_Prot_Access_03	PO access and change object:10 tasks
A5	TK_RZ_Entry_Access_01	Access & change obj protected by a monitor
A6	TK_RZ_Entry_Access_02	Access & change obj protected by a monitor: 5 tasks
A7	TK_RZ_Entry_Access_03	Access & change obj protected by a monitor: 10 tasks

〈표 7〉 ACES 벤치마크 결과 (시간은 초)
〈Table 7〉 ACES Benchmarks, in microseconds

프로그램 이름	SunOS (FSU Threads)			Solaris (Native Threads)			Solaris (FSU Threads)		
	with ATC	no ATC	Gain	with ATC	no ATC	Gain	with ATC	no ATC	Gain
A1	597	576	3.52%	936	831	11.22%	386	358	7.25%
A2	59453	46191	22.30%	39603	26398	33.34%	58496	44359	24.16%
A3	85934	60640	29.43%	43394	28761	33.72%	76344	67535	11.53%
A4	94387	76996	18.42%	59506	51196	13.96%	93950	79962	14.88%
A5	988467	905733	8.36%	1040278	989167	4.91%	803838	690773	14.06%
A6	646631	568736	12.04%	682584	639552	6.30%	511811	430569	15.87%
A7	608961	562233	7.67%	658736	614315	6.74%	502004	431435	14.05%

맨틱을 제공하는 라이브러리로서 호환성을 제공하기 위해서 POSIX 운영체제에 바탕을 두어 개발되었으며 계층화 된 GNARL 구조로 운영체제의 비호환성을 숨기고 포팅이 용이하게끔 설계되어 있다.

논문의 후반부에서는 GNARL의 태스킹 성능을 향상시키기 위한 방법을 제시하고 GNARL 코드를 이에 맞게 변경한 후, 이에 대한 벤치마크 테스트 결과를 제공하였다. 첫 번째의 방법에서 우리는 Lock/Unlock 서비스 호출의 수를 줄일 수 있는 방법에 대해서 설명하면서 사용자가 응용프로그램의 특성에 따라 Lock의 종류를 선택하는 기준을 제시하였다. 단일잠금 방식이 단일 CPU 상에서는 효과적인 수행능력을 보일 것은 예상하였던 일이지만 다중 CPU 상에서도 응용프로그램의 형태에 따라 더 효과적일 수 있음이 Solaris 상에서의 실험 결과에서 나타났다. 대부분의 다른 운영체제 환경에서도 이러한 논리가 적용 될 것이라는 예상 아래에 우리는 이제 GNARL의 잠금정책으로 단일잠금 방식을 제시하고자 한다.

태스킹 효율향상 연구의 두 번째 부분에서는 GNARL의 기능 축소를 통해 ATC와 "태스크 삭제"기능을 제한한 런타임 시스템을 구성하였다. 이 두 가지의 기능을 축소함으로써 우리는 GNARL 태스킹 전체의 성능의 향상을 도출할 수 있었는데, 이는 이 기능들을 제공하기 위한 런타임 시스템전체에 미치는 오버헤드가 컸기 때문이다. 두 개의 GNARL의 성능을 비교함으로써 우리는 ATC와 "태스크 삭제"가 제한 프래그마의 대상에 합당한 기능들이라는 결론을 얻게 되었다.

ATC와 "태스크 삭제" 이외에도 기능 축소를 통해 GNARL전체의 성능향상에 도움이 될 수 있는 것들 몇 가지를 더 생각해 볼 수 있다. 이 연구의 기간 중 우리는 Ada'95의 requeue 기능이 어떠한 호출이 현재 어떤 서버에 큐 되어있는지를 검색해야 되는 번거로움으로 인하여 생기는 오버헤드가 큰 것을 알 수 있었다. 현재의 연구에 requeue 기능을 제한하는 것은 포함되지 않았는데 그 이유는 첫째 이 기능이 언어표준에서 제시하는 제한 기능에 포함되지 않았고, 두 번째로는 먼저 ATC와 "태스크 삭제"만의 기능이 GNARL 구현에 미치는 영향을 알고 싶었기 때문이다. 연구의 과정에서 우리는 시그널을 동반한 문맥전환의 오버헤드가 큼을 또한 알 수 있었다. 그러므로 시그널 핸들러 기능의 제한이 GNARL의 성능향상에 또한 도움이 되리라고 생각한다. protected entry barrier의 계산 또한 오버헤드가 높은 기능중의 하나로서 barrier expression을 제한하는 것이 역시 런타임 시스템의 성능향상을 위한 좋은 대상이 될 것이라고 생각한다.

참 고 문 헌

[1] E. Klingerman and A. Stoyenko. Real-Time Euclid: "A Language for Reliable Real-Time Systems," *IEEE Tans. Software Eng.*, pp.941-949, 1986.

[2] Y. Ishikawa, H. Tokuda and C. Mercer. "Object-oriented real-time language design: Construct for timing constraints," In *Proceedings of OOPSLA/ECOOP*, pp.289-298, 1990.

[3] A. Stoyenko and W. Halang. "Extending PEARL for industrial real-time applications," *IEEE Software*, Vol.10, No.4, pp.65-74, 1993.

[4] International Organization for Standadization, International Electrotechnical Commission. *Ada Reference Manual*, ISO/IEC 8625:1995(E), 1995.

[5] United States Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A -1983, 1983.

[6] Ada9X Mapping/Revision Team, Intermetrics, Inc. *Rationale for the Programming Language Ada : Language and Standard Libraries,* 1994.

[7] Ada Core Technologies. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.

[8] R. Stallman. *Using and Porting GNU CC.* Free Software Foundation, 1994.

[9] E. Schonberg and B. Banner. "The GNAT Project: A GNU-Ada9X Compiler," In *TRI-Ada'94 Proceedings,* Baltimore, Maryland, 1994.

[10] Portable Application Standard Committee of the IEEE. *Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language],* IEEE Std 1003.1 - 1990.

[11] Portable Application Standard Committee of the IEEE. *Draft Standard for Information Technolofy - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 2: Threads Extension [C Language],* IEEE Std 1003.1c - 1994.

[12] E. Giering, F. Mueller, and T. Baker. "Implementing Ada9X features using POSIX threads: Design issues," In *Tri-Ada'93 Proceedings,* pp.214 -228, Seattle, Washington, 1993.

[13] E. Giering and T. Baker. "Implementing Ada Protected Objects - Interface Issues and Optimization," In *Tri-Ada'95 Proceedings,* pp.134-143, Anaheim, California, 1995.

[14] D. Oh, T. Baker, and S. Moon. "The GNARL Implementation of POSIX/Ada Signal Services," In *Ada-Europe'96 Proceedings,* pp.276-286, Springer, Verlag, 1996.

[15] CTA Incorporated. Ada Compilation Evaluation System (ACES). Available by anonymous FTP from sw-eng.falls-church.va.us.

[16] A. Habermann and I. Nassi. "Efficient Implementation of Ada Tasks," Technical report, Carnegie Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania, 1980.

[17] Implementation Strategies for Ada Tasking Idioms. In *AdaTEC Conference,* pp.26-30, Arlington, Virginia, 1982.

[18] F. Mueller. "A Library Implementation of POSIX threads under UNIX," In *Proceedings of the USENIX Conference,* pp.29-41, 1993.

[19] D. Oh and T. Baker. "Gnu Ada'95 Tasking Implementation: Real-Time Features and Optimization," In proceedings of the 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS), Las Vegas, Nevada, 1997.

[20] SunPro. *Multithreading Your Ada Applications,* December 1993.

문 승 진

sjmoon@mail.suwon.ac.kr

1986년 텍사스 주립대학교(오스틴) 전산학과 졸업(학사)

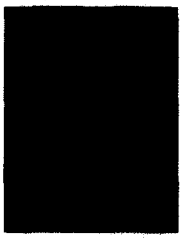
1991년 플로리다 주립대학교 전산학과 졸업(석사)

1997년 플로리다 주립대학교 전산학과 졸업(박사)

1997년 5월~1997년 8월 (주)맥시스템 멀티미디아 및 통신 연구소 소프트웨어 실장

1997년 9월~현재 수원대학교 전자계산학과 전임강사

관심분야 : Ada 프로그래밍언어, 실시간 데이터베이스 시스템, 실시간 시스템 등



오 동 익

doh@ai-cse.sch.ac.kr

1985년 뉴욕시립대학교 전산학과 졸업(학사)

1989년 플로리다 주립대학교 전산학과 졸업(석사)

1997년 플로리다 주립대학교 전산학과 졸업(이학박사)

1997년~현재 순천향대학교 공과대학 컴퓨터학부 전임강사

관심분야 : Ada 프로그래밍언어, 실시간 시스템, 운영체제 등