論文91-28B-9-2

RISC 병렬 처리를 위한 기억공간의 효율적인 활용 알고리듬

(An Efficient Storage Reclamation Algorithm for RISC Parallel Processing)

李 喆 源* 林 寅 七*

(Chul Won Lee and In Chil Lim)

要 約

본 논문에서는 객체지향 프로그래밍 환경에서의 RISC 병렬처리 성능 향상을 위한 기억공간의 효율적인 활용 알고리듬을 제안한다. RISC 병렬 처리시 동적인 기억 장소 할당 및 빈번한 메모리 액세스는 전체 수행 시간의 상당한 부분을 차지하고 있으며 RISC 병렬 처리의 성능을 저하시키는 주된 요인이 되고 있다.

제안된 메모리 재사용 알고리듬은 빈번한 메모리 액세스를 요구하는 RISC형 컴퓨터의 기억 장소를 효율적으로 할당함으로써 RISC 병렬 처리의 수행 성능을 향상시킬 수 있다.

본 논문에서 제안한 알고리듬을 SUN SPARC(4.3BSD UNIX) 상에서 C 언어로 실현하여 Bubble Sort 예제에 적용함으로써 그 효율성을 입증한다.

Abstract

In this paper, an efficient storage reclamation algorithm for RISC parallel processing in the object oriented programming environments is presented.

The memory management for the dynamic memory allocation and the frequent memory access in object oriented programming is the main factor that decreases RISC parallel processing performance.

The proposed algorithm can be efficiently allocated the memory space of RISCy computer which is required the frequent memory access, so it can be increased RISC parallel processing performance.

The proposed algorihm is verified the efficiency by implementing C language on SUN SPARC (4.3 BSD UNIX).

I. 서 론

VLSI 기술의 발달로 마이크로 프로세서와 메모리의 성능이 우수해지고 가격이 저렴해지면서 이 장점을 최대한으로 이용할 수 있는 병렬처리 시스템에 대한 관심이 높아지고 있다. 그러나 현재의 병렬처리 시스템 분야는 아직 해결해야할 많은 문제점들을 안고 있다. 특히 그 동안 병렬처리 하드웨어 분야가 많은 발전을 한 것에 비하여 병렬처리 소프트웨어 분야는 큰 성과를 거두지 못하고 있는 실정이며 앞으로의 주요 연구분야이다.[1]

특히 기존 컴퓨터 시스템에서의 프로그래밍과 같이 하드웨어의 구조를 고려할 필요없이 병렬성을 갖는 고급언어를 이용하여 병렬처리를 행하는 방법은 아직 많은 연구를 필요로 하고 있는데, 이러한 목적이 달성된다면 병렬처리 시스템은 기존의 컴퓨터 시스템에서 처럼 여러 분야에 걸쳐 광범위하게 사용될

*正會員, 漢陽大學校 電子工學科

(Dept. of Elec. Eng., Hanyang Univ.)

接受日字 : 1991年 7月 6日

수 있을 것이다. 이 방법은 사용자가 작성한 프로그램을 컴파일하는 과정에서 이를 동시에 처리할 수 있는 여러개의 모듈로 나누고 이 모듈들을 커널과 가상 머신의 지원을 받아 여러 프로세스로 구성된 병렬처리 하드웨어 시스템에서 수행시키는 것이다.[1-4]

이러한 방법중의 하나인 객체지향 프로그래밍은 소프트웨어 공학적 측면에서 구조적 프로그래밍 개념을 심화시킨 것으로 프로그래밍 환경의 새로운 형식과 개념을 유도하였다. 객체는 실세계의 객체들과 일대일로 대응되어, 제기된 문제의 분석 및 모델링이 쉽게되고 클래스의 상속성 및 다형성, 캡슐화 등의 특성으로 인해 소프트웨어의 재사용성을 증가시키며 요구 변화에 따른 대처가 빨라 유지, 보수가 용이하고 전체적인 프로그래머의 생산성을 향상시킬 수 있는 장점을 가진다. 그러나 이러한 장점에도 불구하고 객체지향 언어는 빈번한 프로시듀어 호출에 의해 발생되는 메모리의 할당은 처리 속도를 감소시키는 주된 요인이 되고 있으며 이를 위한 효율적인 메모리 운영의 중요성이 요구되고 있다. 공유 메모리를 사용하는 병렬 프로세서에서 메모리의 빈번한 액세스는 병렬처리 효과를 감소시키며 소프트웨어적인 처리를 어렵게 하고 있다.[1-5]

또한 객체 지향 개념을 RISC 병렬처리에 구현할 경우 빈번한 프로시듀어 호출 및 수행시 객체의 동적인 생성과 자동적인 메모리 재구성 등의 특성으로 인하여 RISC의 우수한 특성과 장점을 최대한 활용하는데 어려움이 따른다.[5-7]

본 논문에서는 RISC 병렬 처리의 성능향상을 위한 기억공간의 효율적인 활용 알고리듬을 제안한다. 본 논문에서 제안한 알고리듬은 시스템내의 모든 객체를 생존 기간에 따라 분류하고 이를 별도로 관리함으로써 RISC 병렬 처리의 전체적인 성능 향상을 얻을 수가 있다.

또한, 본 논문에서 제안한 알고리듬을 SUN SPARC (4.3 BSD UNIX) 상에서 C 언어로 실현하여 Bubble Sort 예제에 적용함으로써 그 효율성을 입증한다.

Ⅱ. 객체 지향 병렬 처리 시스템과 가베이지 콜렉션

1. 객체 지향 병렬 처리 시스템

객체 지향 병렬처리 시스템이란 병렬처리 시스템을 구성하는 기본원리인 계산모델이 객체지향 프로그래밍 개념에 근거한다는 것을 의미하며, 객체 지향 언어는 그 자체에 동시성을 가지고 있다는 것을 전제로 하고 있다. 사용자가 병렬처리 하드웨어 시스템의 구조를 의식함이 없이 객체지향 언어로써 프로그램을 작성하면 그 객체지향 언어 컴파일러 및 객체지향 시스템 커널은 그 프로그램에서 동시에 처리될 수 있는 모듈로 분리하고 그들을 여러 개의 프로세서를 가지고 있는 하드웨어 시스템에서 병렬처리하도록 하는 것이 객체지향 병렬처리 시스템의 목적이다.

프로그램의 동시성은 한 타스크에 관련된 프로그램을 비교적 독립적인 여러개의 모듈로 분리하여 표현하는 것을 기본으로 하고 있는데, 일반적으로 모듈의 수행 가능한(executable) 형태가 프로세스라고 할 수 있다. 따라서 한 소프트웨어를 어떻게 여러개의 프로세스로 표현하는가 하는것이 소프트웨어 동시성에 가장 중요한 점이다.

2. 객체 지향 프로그래밍의 요소

객체는 여러개의 자료 구조(state)와 이들 자료구조에 적용 가능한 연산(method)으로 구성되며 캡슐화와 데이타의 추상화로 설명된다. 객체지향 프로그래밍은 기존의 절차언어와는 달리 데이타와 프로시듀어를 결합한 모듈의 기능을 갖고 있다. 데이타와 프로시듀어 결합에 의한 모듈은 정보 은폐의 기능을 제공하고 있으며 캡슐화와 추상적 데이타의 기본을 이루고 있다.

또한, 객체 지향 프로그래밍에서는 모든 존재와 그 동작이 객체로 표현되므로 프로그래밍이란 결국 객체를 서술한 것이 된다. 여기서 객체를 서술 한다는 것은 그 자체가 어떤 구조를 갖고 있으며, 그 객체가 받을 수 있는 메세지는 어떠한 것이 있는가, 또한 그 객체가 메시지를 받았을 때 어떤 방식으로 처리 될 것인가 등의 내용을 정의하는 것이다. 객체에 대한 이러한 정의를 클래스라 한다.

그리고 상속성은 객체지향 프로그래밍 언어의 가장 큰 특징인 코드의 재사용을 가능하게 해주는 개념이다. 객체 지향 프로그래밍 언어의 한 클래스는 일반적으로 이를 클래스를 좀 더 세분한 하위클래스로 정의 된다. 이때 원래의 클래스를 상위 클래스라고 한다. 즉, 어떤 클래스를 정의할 경우 기존의 한 클래스에서 새로운 하위 클래스를 정의할 수 있다.

상속성이란 하위클래스가 자신의 상위클래스에서 정의된 모든 변수와 메소드를 물려받는 성질을 말한다. 다시 말해서 하위클래스에서는 상위클래스에서 정의된 모든 변수와 메소드를 그대로 물려 받아 여기에 자신의 고유한 특성을 추가하거나 이를 재정의 할 수가 있는데 이를 구체화(specialization)라 한다. 여러 클래스에 걸쳐 상속이 발생될 경우 클래스 계층 구조의 가장 아래 부분에 위치한 클래스가 가장 구체화되는 모습을 보이게 되며 위쪽에 위치한 클래

스일수록 일반화(generalzation) 된다.

3. 가베이지 콜렉션

가베이지 콜렉션이란 사용되지 않고 있는 메모리 영역을 수거하여 사용 가능한 영역으로 재구성하는 작업을 말한다. 리스트 처리용 언어가 구현된 60년대 초부터 지금까지 여러가지 다양한 알고리듬들이 개발 구현되어 왔다. 특히 빈번한 메모리 액세스와 프로시듀어 호출이 발생되는 객체지향 언어에서는 효율적인 메모리 관리와 성능향상을 위해 그 중요성이 커지고 있다.

Ⅲ. Smalltalk에서의 기억공간 활용 기법

1. Smalltalk 상에서의 변수 분류

일반 프로그래밍 언어에서 변수는 특정한 타입의 값을 저장하는 역할을 하고 있으나, Smalltalk 시스템 상에서는 모든 객체는 동적으로 생성되며 따라서 변수 자체가 정적인 기억장소를 차지하고 있는 것이 아니라 어딘가에 저장된 객체를 가리키는 포인터일 뿐이다. 또 변수에는 타입이 없으므로 한 변수는 어떤 클래스의 객체라도 가리킬 수 있다.

변수의 이름은 객체가 갖고 있는 포인터를 참조하기 위해 수식에서 사용된다. 하나의 변수는 서로 다른 시점에 서로 다른 객체를 가질 수가 있으며 수식이 실행될 때 변수가 갖고 있는 포인터는 변경된다. 다시 말해서 객체의 배정문은 객체 그 자체를 복사하는 것이 아니라 그 객체의 포인터를 복사하는 것이다.

Smalltalk 내에서 변수의 종류는 다음과 같이 세가지로 구분할 수 있다.

1) 지역 변수(Local Variable)

지역변수는 하나의 객체에 의해서만 액세스할 수 있는 변수를 말하며 이에는 인스턴스 변수(instance variable) 및 임시 변수(temporary variable)로 구분할 수 있다.

인스턴스 변수는 각 객체내의 내부적인 상태를 표시하는 변수로서 다른 언어에서 볼 때 레코드 구조의 필드와 유사하다. 이 변수는 이름에 의해 참조되는 변수(named instance variable)와 정수형 인덱스에 의해 참조되는 변수(indexed instance variable) 로 구분되며 이들은 한 객체의 생존 주기동안 메모리에 존재한다.

또한, 임시 변수는 메소드가 호출될 때 생성되며 메소드에 관련된 인수와 임시 변수들이 포함된다. 이 변수에 대한 생존 주기는 메소드 실행 완료와 함께 끝난다. Smalltalk 내의 지역변수는 모두 동일한 생존 기간을 갖고 있다.

2) 클래스 변수(Class Variable)

클래스 변수는 하나의 클래스내에서 모든 인스턴스들이 액세스할 수 있는 변수이다. 클래스 변수의 생존 기간은 그 클래스의 생존 기간과 연관되어지나, 어떠한 인스턴스의 생존 기간과는 관계가 없다.

3) 공유 변수(Shared Variable)

시스템내에서 언제 어디서든지 액세스할 수 있는 변수이다. 공유 변수는 모든 객체들이 사용할 수 있는 전역 변수(global variable)와 클래스 계층 구조내에서 2개 이상의 클래스들이 사용 가능한 풀변수(pool variable)로 나누어 진다. 공유 변수는 프로그램에 의해 제거될 때까지 시스템내에 존재하게 된다.

Smalltalk 시스템에서는 모든 객체가 동적으로 생성, 소멸되며 또한 대부분의 객체가 짧은 생존 주기를 갖는 임시 변수와 인스턴스 변수로 구성된다. 따라서 이러한 객체들의 관리를 위한 효율적인 메모리 운영은 시스템의 성능 향상을 위해 중요한 요인이 된다.

2. Smalltalk 시스템상에서의 가베이지 콜렉션

Smalltalk 객체들에 대한 가베이지 콜렉션을 위해서 메모리 영역을 그림1과 같이 두개의 논리 영역(logical area)으로 구분한다. NewSpace에서는 활동성 객체(active object)를 위한 메모리 공간으로 사용되며, OldSpace는 비활동성 객체(inactive object)을 위한 영역이다.

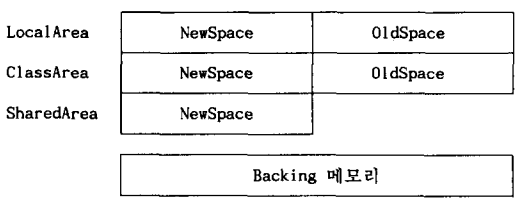
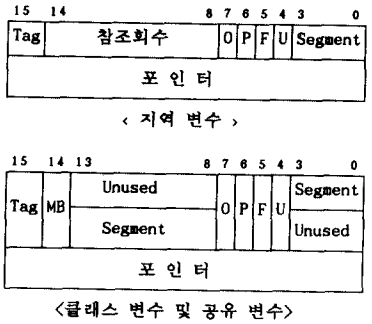


그림 1. 메모리 영역의 구성
Fig. 1. Organization of memory space.

또한, 시스템내의 변수를 앞에서 기술한 세가지 형태로 구분하고 이를 관리하기 위한 참조표(Reference Table)를 설정한다. 그림2는 참조표의 구성 및 내용을 도시화한 것이다.

1) 지역 변수에 대한 가베이지 콜렉션

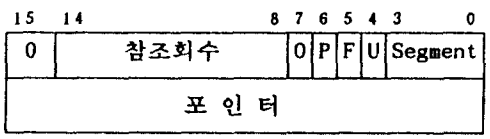
지역변수(임시변수 및 인스턴스 변수)는 NewSpace에서 생성되며 그의 생존기간이 끝나면(즉, store 명령 수행후) 그 객체에 대한 태그 비트를 0으로 두



여기서, Tab bit 0 : 지역변수
1 : 클래스 변수 및 공유변수
MB(마크비트) 0 : backing 메모리 참조
1 : OldSpace 참조
참조회수 객체의 참조회수
0 (odd bit) 0 : odd
1 : even
P(포인터비트) 0 : 상수값
1 : 변수
F(Free bit) 0 : in use
1 : free
Segment 메모리 내의 segment 번호
포인터 segment 내의 offset

그림 2. 참조표의 구성
Fig. 2. Organization of reference table.

고 참조 회수 및 포인터를 참조표에 등록한 후 OldSpace로 이동한다. 여기서 사용되는 참조표는 포인터를 키로 갖는 hash table로써 구성된다.



이 지역변수의 재사용 시에는 해당 객체의 포인터를 찾아서 참조회수를 감소시키고 OldSpace에서 NewSpace로 재복사하여 사용한다.

사용이 끝난 후 참조회수를 조사하여 0이면 메모리 영역에서 해제시키고 0이 아니면 참조표에 재등록한다. 그림3은 지역변수에 대한 가베이지 콜렉션의 과정을 도시화한 것이다.

2) 클래스 변수에 대한 가베이지 콜렉션

클래스 변수는 NewSpace에서 생성되어 그 생존기간 동안 사용되며 사용기간이 끝나면 Tag bit를 1

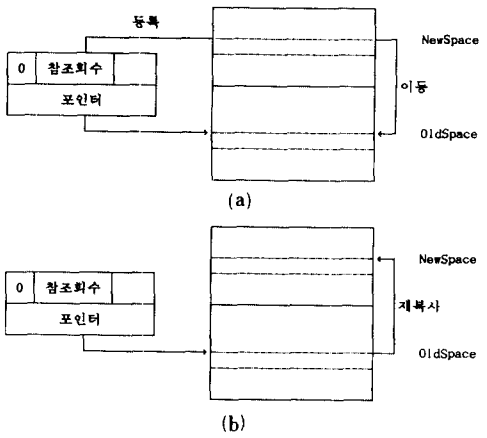
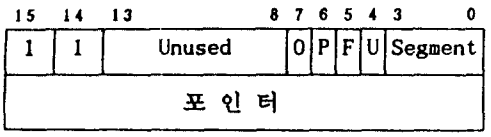


그림 3. 지역변수의 메모리 참조
(a) 저장 (b) 재사용
Fig. 3. Memory reference of local variable.
(a) Store, (b) Reuse.

로 세트하고 클래스 변수에 대한 내용을 참조표에 등록한 후, OldSpace로 이동된다. 이 객체는 NewSpace에서 제거된다. 클래스 변수를 위한 참조표의 구성은 다음과 같으며 객체의 상태를 표시하기 위하여 마크 비트를 추가로 설정한다.



클래스 변수가 재사용될 경우 해당 클래스에 대한 포인터를 참조표로부터 탐색하여 마크비트가 1이면 OldSpace에서 NewSpace로 복사하여 사용하고 참조표에서 그 객체에 대한 마크비트를 0으로 리세트 한다.

그 객체에 대한 사용이 끝나면 backing 메모리로 이동시킨 후 OldSpac로부터 해제시킨다. 참조표를 탐색하여 마크비트가 0인 객체를 재사용할 경우에는 backing 메모리로 부터 복사하여 사용한다.

그림4는 클래스 변수에 대한 메모리 참조 과정을 도시화한 것이다.

3) 공유 변수에 대한 가베이지 콜렉션

지역변수와 클래스 변수를 제외한 공유변수는 프로그램 수행시간과 동일한 생존기간을 가진다. 공유변수는 NewSpace에서 생성되고 사용이 끝나면 태그 비트를 0으로 세트하고 공유변수에 대한 내용을 참

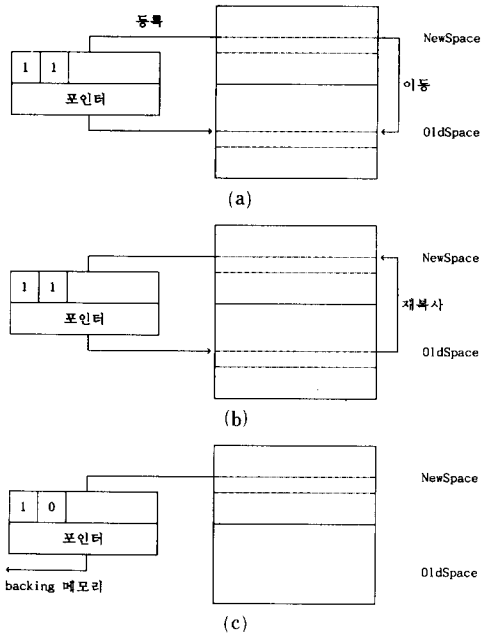
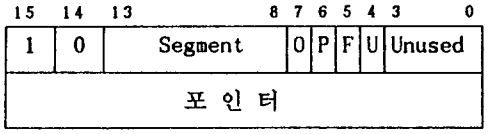


그림 4. 클래스 변수의 메모리 참조
(a) 저 장 (b) 재사용 (c) 사용후
Fig. 4. Memory reference of class variable.
(a) Store, (b) Reuse, (c) After using.

조표에 등록한 후 backing 메모리로 이동되며 그 객체에 대한 NewSpace내의 메모리 공간을 해제 시킨다.

공유변수에 대한 참조표의 구성은 다음과 같다.



공유변수의 재사용시는 참조표의 포인터를 이용하여 backing 메모리로 부터 복사하여 사용한다.

그림5는 공유 메모리에 대한 메모리 참조 과정을 도시화한 것이다.

또한, Backing 메모리는 Off-Line으로 Mark/Sweep 기법에 의하여 가베이지 콜렉션이 된다. 그림 6은 Backing 메모리의 Mark/Sweep 과정을 나타낸다.

3. 병렬 처리를 위한 가베이지 콜렉션

병렬 처리는 프로그램을 좀 더 빨리 수행할 수 있는 가장 확실한 방법중의 하나이다. 반도체 기술의

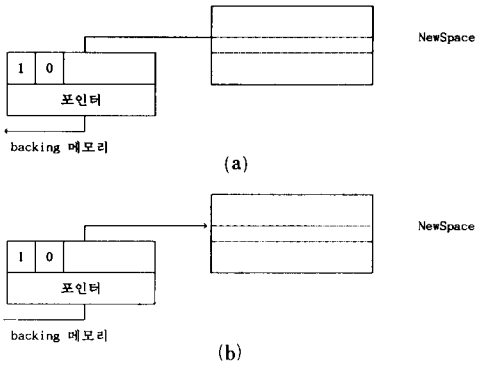


그림 5. 공유 변수의 메모리 참조
(a) 저 장 (b) 재사용
Fig. 5. Memory reference of shared variable
(a) Store, (b) Reuse.

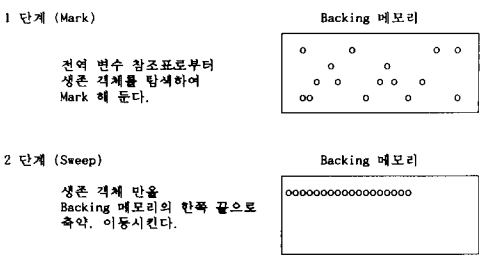


그림 6. Backing 메모리에서의 Mark/Sweep
Fig. 6. Mark/Sweep in backing memory.

발달로 프로세서의 가격이 지속적으로 하락함에 따라 병렬 처리에 대한 관심이 고조되고 있으며 이는 인공지능 등의 복잡한 처리를 필요로 하는 응용 프로그램에 특히 중요시 되고 있다.

병렬 처리의 하드웨어적인 분야는 많은 진전을 보고 있으나 이를 이용하기 위한 소프트웨어 측면의 병렬 처리 언어의 개발이 요구되고 있으며 이러한 언어는 다른 시스템으로의 이전이 쉽고 사용하기 쉬우며 또한, 성능 면에서 효율적이어야 한다.

객체지향 언어에서 객체는 자율성을 가진 처리 모듈이며 하나의 프로그램은 여러개의 객체 및 메시지 전송으로 표현되고 있다. 객체는 메시지가 존재할 때만 동작하므로 메시지 전송은 객체 사이의 제어 흐름을 의미한다.

메시지 전송을 비동기적(asynchronous)으로 하게 한다면 활동적인 객체들 사이에는 동시에 여러개의 메시지가 전달되고 각 객체는 그 객체에 전달된 메

시지를 처리함으로써 한 순간에 여러 객체가 동작할 수 있다. 또한, 객체의 생성, 소멸 및 통신은 매우 동적으로 발생하므로 이에 따른 메모리 참조가 빈번하게 발생되며 이를 위한 효율적인 메모리 관리는 병렬 처리의 성능을 향상시킬 수 있다.

특히 RISC 프로세서를 MIMD(Multiple Instruction Streams Multiple Data Streams) 방식의 병렬처리 구조로 설계하고 객체 지향 언어의 메모리 관리기법을 사용한다면 객체 지향의 장점을 충분히 고려함과 동시에 RISC의 하드웨어적인 특성과 조화를 이룰 때 효율적인 객체 지향 RISC 병렬처리를 구현할 수 있을 것이다.

Ⅳ. 가베이지 콜렉션 알고리듬

제안된 알고리듬들을 구축하기 위하여 사용되어지는 구조체를 그림7과 같이 선언, 정의한다.

```
struct space {
       int *firstWord:      /* start of space */
       int size:            /* number of used words in space */
};
struct obj {
       int size:
       int variableType;
       int isForwarded:
       int class:
       union {
         struct obj *fields[]:
         struct obj *forwardingPointer:
       };
};
struct ref_table {
      int tag:
      int mark:
      int ref_cnt:
      int odd:
      int p_bit:
      int free:
      int *offset:
      int *pointer:
};

struct space  NewSpace, OldSpace, BackingMemory:
struct ref_table LocalRef, ClassRef, GlobalRef:
```

그림 7. 사용되는 구조체
Fig. 7. Using structure.

새로운 객체의 생성을 위한 알고리듬은 3가지 변수 모두 동일하며 이를 그림8에 나타낸다.

```
Create(obj)
{
      new_in_NewSpace(obj);
      initialize_ref_tab(obj);
}
```

그림 8. 객체 생성 루틴
Fig. 8. Object creation routine.

지역 변수의 store를 위한 알고리듬을 그림 9에 나타낸다.

```
Local_store(obj)
{
      if(LocalRef.ref_cnt == 0) free_obj(obj);
        else {
             new_in_OldSpace(old_obj);
             old_obj = obj;
             free_obj(obj);
             LocalRef = hash(old_obj);
             LocalRef.tag = 0;
             }
}
```

그림 9. 지역변수의 store 루틴
Fig. 9. Store routine of local variable.

지역 변수의 재사용을 위한 알고리듬을 그림10에 나타낸다.

```
Local_reuse(obj)
{
      LocalRef = hash(obj);
      decrease_ref_cnt(LocalRef.ref_cnt);
      new_in_NewSpace(old_obj);
      old_obj = obj;
      free_obj(obj);
      return(old_obj);
}
```

그림10. 지역 변수 재사용 루틴
Fig. 10. Reuse routine of local variable.

클래스 변수의 store에 대한 알고리듬을 그림11에 나타낸다.

```
Class_store(obj)
{
      new_in_OldSpace(old_obj);
      old_obj = obj;
      free_obj(obj);
      ClassRef = hash(old_obj);
      ClassRef.tag =1;
      ClassRef.mark = 1;
}
```

그림11. 클래스 변수의 store 루틴
Fig. 11. Store routine of class variable.

클래스 변수의 재사용을 위한 알고리듬을 그림12에 나타낸다.

```
Class_reuse(obj)
{
      ClasRef = hash(obj);
      new_in_NewSpace(old_obj);
      if (ClassRef.mark == 1) {
            old_obj = obj;
            ClassRef = hash(old_obj);
            ClassRef.mark = 0;
            free_obj(old_obj);
        }
       else
           old_obj = load_from_backing(obj);
      return(old_obj);
}
```

그림12. 클래스 변수의 재사용 루틴
Fig. 12. Reuse routine of class routine.

공유 변수의 store를 위한 알고리듬을 그림13에 나타낸다.

```
Global_store(obj)
{
      GlobalRef = hash(obj);
      GlobalRef.tag = 1;
      GlobalRef.mark = 0;
      store_to_backing(obj);
      free_obj(obj);
}
```

그림13. 공유 변수의 store 루틴
Fig. 13. Store routine of shared variable.

공유 변수의 재사용에 대한 알고리듬을 그림14에 나타낸다.

```
Global_reuse(obj)
{
      new_in_NewSpace(obj);
      old_obj = load_from_backing(obj);
      return(old_obj);
}
```

그림14. 공유 변수의 재사용 루틴
Fig. 14. Reuse routine of shared variable.

한편, Backing 메모리의 Mark/Sweep은 그림15와 같이 공유변수 참조표로 부터 깊이 우선 탐색으로 이들이 참조하고 있는 생존 객체를 Mark 해 둔 후 메모리를 다시 처음부터 탐색하여 Mark된 객체들만 메모리의 한쪽 끝으로 축약, 이동시킨다. (이때 Mark 여부를 알리기 위하여 isForwarded를 사용한다.)

```
markSweep(obj)
{
    for (i = 0; i <= GlobalVariableTableSize; i++)
         markOf(GlobalVariableTable[i]);

    for (si = di = BackingMemory.firstWord;
         si <= BackingMemory.firstWord + BackingMemory.Size;
         si += (4 + si->size)){
         if  (si->isForwarded == 1){
              di->size = si->size;
              di->variableType = si->variableType;
              di->isForwarded = 0;
              di->class = si->class;
              for (i = 0; i < si->size; i++)
                   di->fields[i] = si >fields[i];
              di += (4 + di->size)}
         }
    }
}

markOf(referrer)
struct object *referrer;
{
   int i;
   struct object *referent;
   for (i = 0; i < referrer->size; i++){
        referent = referrer->fields[i];
        if (referent->isForwarded != 1){
            referent->isForwarded = 1;
            markOf(referent);
        }
   }
```

그림15. Backing 메모리에서의 Mark/sweep 루틴
Fig. 15. Mark/sweep routine in backing memory.

V. 알고리듬 적용 예

본 논문에서 제안한 가베이지 콜랙션 알고리듬을 bubble sort 프로그램에 적용한다.

그림16은 본 알고리듬의 효율성을 입증하기 위해서 입력 파일로 사용된 bubble sort의 Smalltalk 프로그램을 나타낸 것이다.

이 프로그램을 제안된 알고리듬을 이용하여 수행시킨 결과와 기존의 Smalltalk 시스템 내에서 수행한 결과를 비교하면 표1과 같다.

표1에 의하면 메모리 회수 영역 및 생존 객체의 메모리 크기가 향상되어 메모리의 재활용을 효율적으로 할 수 있다는 것을 알 수가 있다. 또한, 기존의 Smalltalk로 수행한 것보다 총 CPU 시간이 향상되었음을 볼 수 있다.

VI. 결 론

본 논문에서는 객체지향 개념을 이용하여 RISC 병렬 처리의 성능을 향상시키기 위한 Smalltalk의 효율적인 기억공간 활용 알고리듬을 제안하였다.

객체 지향 언어를 RISC 병렬처리에 구현할 때 짧은 생존 주기를 가지는 객체들의 동적인 생성과 빈번한 프로시듀어 호출은 보다 복잡한 메모리 관리를

```
Object subclass: #Bubble
  instanceVariableNames: ''
  classVariableNames: 'SortedList RandomList N '
  poolDictionaries: '' !

!Bubble class methods !
bubble
    " Sort RandomList to Sorted List and then Display "
    | i j temp |
    Bubble input.
    SortedList := RandomList.
    SortedList inspect.
    i := 1.
    [i < N ]
        whileTrue: [
            j := i.
            [j <= N]
                whileTrue: [
                    ((SortedList at: (j + 1)) > (SortedList at:j))
                        ifTrue: [
                            temp := SortedList at: (j + 1).
                            SortedList at: (j + 1)
                                    put: (SortedList at: j).
                            SortedList at: j put: temp.].
                    j := j + 1].
            i := i + 1].
    SortedList inspect.!
input
    " Receive Input Data and then save class variables. "
    | cnt cntStream |
    N := Prompter
            prompt: 'Enter No. of Data' defaultExpression: '10'.
    (N isKindOf: Integer)
        ifFalse: ['super error: 'Number should be Integer . . '].
    RandomList := Dictionary new.
    cnt := 1.
    [ cnt <= N ]
        whileTrue: [
            cntStream := WriteStream on:String new.
            cntStream nextPutAll: 'Enter Data # '.
            cnt printOn: cntStream.
            RandomList
                at: cnt
                put:((Prompter prompt: cntStream contents
                                defaultExpression: '10')).
            cnt := cnt + 1].
    RandomList inspect! !
```

그림16. Bubble sort 예제 프로그램
Fig. 16. Bubble sort example program.

표 1. 수행결과 비교표
Table 1. Comparision table of result.

	기존의 Smalltalk	제안된 Smalltalk
생존 객체의 메모리 크기	4.5 Kb	3.0 Kb
총 CPU 시간	280 ms	250 ms
일시 정지 시간	90 ms	10 ms
메모리 회수 영역	390 Kb	400 Kb

요구하게 되고 이에 따른 수행시간 오버헤드는 전체적인 성능 향상을 저해하는 요인으로 작용하게 된다. 이러한 문제를 해결하기 위하여 본 논문에서는 Smalltalk상에서 효율적인 기억공간 관리를 위하여 시스템내의 객체를 생존 주기에 따라 지역 변수와 공유 변수로 나누고 메모리 공간을 두개의 부분 논리 기억 공간으로 구분하였다.

또한, 객체의 복사와 이동을 관리하기 위한 참조표를 두고 태그 비트에 의해 객체의 타입을 구분하여 객체의 동적인 생성의 대부분을 차지하는 지역 변수들의 자동적인 메모리 재사용을 중점적으로 처리하도록 하였으며 비교적 오랜 생존 기간을 갖는 공유 변수는 외부 기억 장소에 두고 지역변수와 함께 이동되는데 따른 불필요한 시간 낭비를 줄일 수 있도록 하였다.

이 기법을 이용함으로써 모든 객체에 대한 참조 회수를 관리하는 기존의 Smalltalk 보다 처리 속도를 향상시킬 수 있었으며 기억 장소의 재사용에서 좋은 효율을 얻을 수 있었다.

參 考 文 獻

[1] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[2] B.J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[3] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Welsey, 1983.

[4] G. Krasner, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983

[5] D.M. Ungar, "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm," ACM Software Eng. Notes/SIGPLNAN Notices Software Engineering Symposium on Practical Software Development Environments, pp. 157-167, Appl. 1984.

[6] W. Sush, A. Samples, D.M. Unger P.Hilfinger, "Compiling Sallltalk-80 to a RISC," OOPSLA '87 Conference Proceedings, pp. 112-116, Oct. 1987.

[7] S. Krueger, "VLSI-Appropriate Gargage Collection Support," VLSI for Artificial Intelligence, Kluwer Academic Pub, pp. 75 84, 1989.

[8] R. Courts, "Improving Locality of Reference in Garbage-Collecting Memory Management System," CACM, vol. 31:9, pp. 1128-1138, Sep. 1988.

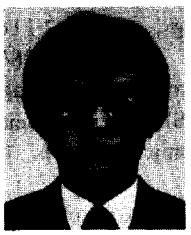
[9[P. Wilson, T. Moher, "Design of the Opportunistic Garbage Collector," OOPSLA '89 Proceedings, pp. 23-35, Oct. 1989.

[10] B. Goldberg, "Generational Reference Counting: A Reduced -Communciation Distributed Storage Reclamation Scheme," OOPSLA '89 Conference Proceedings, pp. 313-321, Oct. 1989.

[11] B. Stroustrup, "What is Object-Oriented Programming?," IEEE Software, pp. 10-20,

May 1988.
[12] J.M. Pendleton, S.I. Kong, E.W. Brown, F. Dunlap, C. Marino, D.M. Unger, D.A. Patterson, and D.A. Hodges, "A 32-bits Microprocessor for Smalltalk," IEEE Journal of Solid-State Circuits, vol. SC-21, no. 5, pp. 741-748, Oct. 1986.
[13] D.A. Patterson, "Reduced Instruction Set Computers," Commum. of the ACM, vol. 28, no. 1, pp. 8-21, Jan. 1985.
[14] 신기정, 장철규, 현승렬, 최환종, 신창섭, 이철원, 임인칠, "객체 지향 RISC 병렬처리를 위한 효율적인 메모리 관리",대한전자공학회 추계종합학술대회 논문집, vol. 13, no. 2, pp. 157－161, 1990 11.
[15] 정은주, 최환종, 신창섭, 신기정, 장홍종, 이철원, 임인칠, "RISC 병렬 처리를 위한 가동 가베이지 콜렉션 알고리듬",대한전자공학회 하계종합학술대회 논문집, vol. 14, no. 1, pp. 225－229, 1991 6.

著 者 紹 介



李 喆 源(正會員)
1962年 10月 7日生. 1985年 한양대학교 공과대학 전자공학과 졸업. 1987年 한양대학교 대학원 전자공학과 졸업. 공학석사학위 취득. 1989年 3月～현재 한양대학교 대학원 전자공학과 박사과정 재학중. 주관심분야는 컴퓨터 아키텍춰, 병렬처리, 객체지향 아키텍춰, DSP processor 설계, HDTV 등임.

林 寅 七 (正會員) 第28卷 第4號 參照
현재 한양대학교 전자공학과 교수