C++ 객체의 CORBA 기반 분산 시스템으로의 정적 할당

Static Allocation of C++ Objects to CORBA-based Distributed Systems

최 승 훈*
Seung-Hoon Choi

요 약

요약 분산 시스템의 전체적인 성능에 가장 큰 영향을 미치는 요인 중의 하나는, 소프트웨어 컴포넌트를 어떻게 효율적으로 분산시키는가 하는 것이다. 현재 태스크 기반의 시스템을 분할하여 분산 환경에 할당하는 문제는 연구가 많이 진행되었으나, 객체 지향 프로그램을 구성하는 각 객체들을 분산 객체 환경에 할당하는 기법에 대한 연구는 상대적으로 미약하다. 본 논문에서는 이미 개발되어 있는 C++ 응용 프로그램을 분할하여 C++ 객체들을 CORBA 기반의 분산 객체 환경에 할당하기 위한 그래프 모델을 정의하고, 이를 바탕으로 한 분산 객체 할당 알고리즘을 제안한다. 분산 시스템의 성능은, 주로 객체 간의 병렬성, 각 프로세서에 드는 부하의 균등성, 네트워크 상의 통신량에 의해 결정된다. 이 세 가지 요인을 동시에 최적화하는 해를 찾기 위하여, 본 논문의 분산 객체 할당 기법은 Niched Pareto 유전자 알고리즘(NPGA)에 바탕을 두고 있다. 전형적인 C++ 응용 프로그램에 대한 CORBA 시스템에서의 실험을 통하여 본 논문의 그래프 모델과 객체 할당 알고리즘의 유효성을 검증한다.

Abstract

One of the most important factors on the performance of the distributed systems is the effective distribution of the software components. There have been a lot of researches on partitioning and allocating the task-based system, while the studies on the allocating the objects of the object-oriented system into the distributed object environments are very little relatively. In this paper, we defines the graph model for partitioning the existing C++ application and allocating the C++ objects into CORBA-base distributed system. In addition, we propose a distributed object allocation algorithm based on this graph model. The performance of distributed systems is determined by the concurrency between objects, the load balance among the processors and the communication cost on the networks. To search for the solutions optimizing the above three factors simultaneously, the object allocation algorithm of this paper is based on the Niched Pareto Genetic Algorithm (NPGA). We performed the experiment on the typical C++ application and CORBA system to prove the effectiveness of our graph model and our object allocation algorithm.

1. 서 론

최근 CORBA 표준[1,2], DCOM과 같은 분산 객체 기술이 발전함에 따라 소프트웨어 공학은 이를 지원 하기 위한 기술 개발을 필요로 한다. 그 중에서도 기존의 객체 지향 프로그램을 분산 객체 플랫폼으로 효율적으로 이전시키기 위한 기술이 절실하다. 이러 한 연구의 필요성을 반영하여 최근 재공학 기술의 초 점이, 재래식 시스템을 이해하여 고 수준의 설계 정보 를 추출하고 그 시스템이 제공하는 기능이 무엇인지를 파악하는 것으로부터 기존의 시스템을 분산 객체 환경에 어떻게 효율적으로 변경시킬 것 인가로 옮겨 가고 있다[3].

분산 객체 시스템의 전체적인 성능에 가장 큰 영향 을 미치는 요인 중의 하나는, 소프트웨어 컴

* 덕성여자대학교 컴퓨터과학부 전임강사

포넌트를 어떻게 효율적으로 분산시키는가 하는 것이다. 이를 위하여, 객체 지향 시스템에서의 객체 클러스터링 이나 할당에 대한 연구가 많이 진행되 었는데, 주로 가상 메모리 성능 향상, 통신 성능 향상[4], 부하 균등 성 향상[5], 런타임 오버헤드의 향상, 전반적인 성능 향상 등을 위한 기법들이 제안되었다. 병렬 객체 지향 (concurrent object oriented) 개발 환경이나 메시지 전송 시스템에서도 다양한 객체 할당을 위한 기법들을 제안되었다[6,7,8]. 그러나 이러한 방법들은 통신 비용과 병렬성의 값을 단순히 더한 값을 기준으로 하여 객체 모델을 분할하거나, 네트워크를 이루는 각 프로세서의 연결 상태를 고려하지 않거나, 객체들 간의 병렬성을 고려하지 않는 등 여러 가지 한계점을 가지고 있다.

분산 환경에서 객체 할당을 위한 최적의 해를 찾는 문제는, n개의 객체를 m개의 프로세서에 할당하는 모든 방법 중에서 어떤 기준을 만족하는 최적의 해를 찾는 문제이다. 이 최적화 문제는 객체와 프로세서의 개수가 커지면 거대한 탐색 공간을 가지게 되므로 최적 해를 찾기가 쉽지 않다. 또한 분산 시스템의 성능에 영향을 미치는 요인으로 객체들 사이의 통신 비용과 병렬성, 각 프로세서에 드는 부하의 균등성 등과 같이 여러 가지가 존재하기 때문에 이들을 동시에 만족시키는 최적 해를 찾기가 쉽지 않다. 이러한 NP-complete 문제의 해를 찾기 위해 모든 탐색 공간을 탐색하는 것은 실행 시간이 과도하며 따라서 일반적으로 휴리스틱 알고리즘을 사용한다.

지금까지 분산 시스템의 객체 할당에서 주로 사용된 기법은, 객체들 사이의 친밀도 또는 관련성을 나타내는 메트릭을 정의하고 이 메트릭의 값에 따라 서로 관련성이 큰 객체들을 하나의 클러스터로 묶은 후 이들을 각 프로세서에 할당하는 방법을 사용하였 다. 그러나 분산 시스템의 성능에 영향을 미치는 여러 가지 요인들이 서로 측정 단위가 다를 뿐 만 아니라 서로 영향을 미치거나 경쟁을 하기 때문에, 단순히 하나의 메트릭을 정의하여 그 값을 기준으로 최적 해를 찾는 것은 불합리하다.

따라서 분산 객체의 할당 문제는 여러 가지 목적을 동시에 만족하는 해를 찾는 다중 목적 함수 문제로 다루어져야 하며, 이러한 다중 목적 함수의 최적화 문제를 해결하는데 있어서 다중 함수 유전자 알고 리즘 (Multiobjective Genetic Algorithm)의 유용성 이 잘 알려져 있다[9,10]. 본 논문에서는 다중 함수 유전자 알고리즘의 하나인 Niched Pareto 유전자 알고리즘 (NPGA)을 이용하여 최적의 할당 해를 찾는다.

본 논문의 구성은 다음과 같다. 제2장에서는 관련 연구로서 기존의 객체 분할 및 할당 기법을 살펴본 다. 제3장에서는, 객체 분할 및 할당을 지원하는 그래프 모델을 정의한다. 제4장에서는 병렬성, 통신 비용, 부하 균등성을 동시에 최적화하기 위하여 NPGA에 바탕을 둔 분산 객체 할당 알고리즘을 제 안한다. 제5장에서는 본 논문에서 제안한 객체 할당 알고리즘을 구현하고, 알고리즘이 생성한 해들의 성능을 평가하여 그 유효성을 검증한다. 마지막으로 제6장에서 결론 및 향후 연구 과제를 기술한다.

2. 관련 연구

객체 할당이란, 분산의 단위인 객체 또는 객체 클러스터를 시스템 성능을 향상시키기 위하여 실제 물리적인 프로세서에 할당하는 과정을 의미한다. 실제 객체가 할당될 물리적인 프로세서를 결정하는 과정은, 객체 클러스터링에 의해 시스템이 어느 정도 추상화 된 상태에서 수행되면 탐색 공간의 범위가 훨씬 줄어든 상태에서 진행될 수 있으므로 더욱 효율적이다.

일반적으로 객체 클러스터링과 할당 기법은 분산 시스템의 성능을 향상 시키기 위해서 서로 결합되어 사용되며, 지금까지 객체 클러스터링 및 할당을 지원하는 여러 가지 시스템들이 제안되었다. 액터 (actor) 기반의 프로그래밍 언어에서의 객체

분할 및 분산 기법[11], 객체 클럼프(Object Clumps)[5], Concert 시스템[12], Emerald 시스템[13], Commandos시스템[14] 등이 그것이다.

특히 본 논문의 객체 할당 기법과 관련하여 여러 가지 기법들이 제안되었다. DIAMOND 시스템[7]은, 작은 크기의 액티브 객체로 이루어진 분산 시스템을 개발하기 위한 분석, 설계, 구현을 지원하는 소프트 웨어 개발 환경이다. 이 시스템에서는 작은 크기의 액티브 객체들을 객체들 사이의 연관성을 기준으로 여러 개의 그룹으로 분할하는 클러스터링 기법을 제안하였다. 이 기법의 목표는 객체들 사이의 통신량 과 런타임 오버헤드를 줄이고, 객체 사이에 존재하는 잠재적인 병렬성을 최대한 이용하여 시스템의 성능 을 향상시키는데 있다. 그러나, 이 기법은 클래스 계 층 구조와 동적 바인딩과 같은 객체 지향 패러다임의 여러 가지 특징을 반영하지 않았다. 또한, 부하 균등 성을 유지하기 위한 기법이 포함되지 않았으며, 이기 종 분산 환경을 고려하지 않았기 때문에 각 메소드의 실행 시간이 할당된 프로세서의 컴퓨팅 능력에 상관 없이 항상 일정하게 계산되는 단점을 가지고 있다.

Chang과 Tseng[6]은, FIFO 링크로 이루어진 메시지 전송 시스템에서 객체들 사이의 통신량이 시스템의 성능에 가장 큰 영향을 미치는 요인으로 보고, 서로 다른 노드에 할당되어 있는 객체들 사이의 통신량을 줄이기 위한 기법을 제시하였다. 본 기법은, FIFO 링크로 이루어진 메시지 전송 시스템에서 메시지가 메시지 수신 객체에 도착할 때까지 발생하는 오버 헤드와 해당 메소드의 코드를 찾을 때 까지 발생하는 오버헤드를 최소화 함으로써 시스템의 성능을 향상 시키고자 하였다. 그러나, 이 기법은 객체 사이의 병렬성을 고려하지 않았으며, 각 노드 사이의 부하 균등성을 유지하기 위해서 단순히 각 노드 당 객체의 개수를 제한하는 방법을 사용하였다. 또한 각 프로 세서의 컴퓨팅 능력을 포함한 모델을 제시하지 못 하였다.

Welch와 Hammer[8]는 분산 병렬 프로그램의 분할 및 할당을 자동화 할 수 있는 기법을 제안하고, ADA 프로그램을 통해서 이를 적용하였다. 이 기법의 문제점 중의 하나는, 프로그램의 중간 표현(IR: Inter-mediate Representation)인 CRG 그래프에서 패키지와 서브 프로그램을 서로 구별하지 않고 모두 원으로 표현하였다. 객체 지향 프로그램 관점에서 보면 패키 지는 객체, 서브 프로그램은 메소드에 해당하는데 이들을 구별하지 않았기 때문에 객체 지향 프로그램 에 이 모델을 적용하기에는 문제가 있다. 또한 이 기법은, 메소드 호출 타입, 클래스 계층 구조, 동적 바인딩 등의 객체 지향 모델 특성들을 반영하지 않았다. 또한 병렬성 메트릭을 정의할 때, 단지 어떤 프로그램 구성 단위가 특정 태스크에 의해서 배타적 으로 사용되는 경우에 병렬성이 있다고 추정하였다. 즉, 메소드 호출 타입(동기적 또는 비동기적)에 따라 병렬성을 구하는 것이 아니라, 단지 그 모듈이 배타 적으로 사용되는지 안 되는지에 따라 병렬성의 값이 정해진다. 또한, 병렬성 메트릭이 0과 1 사이의 값을 가질 때에는 통신량을 측정하는 메트릭 값에 의해 분할을 수행함으로써, 통신량 보다는 병렬성에 더 높은 우선 순위를 부여하였다. 본 논문에서는, 객체 사이의 병렬성을 보다 정확히 계산할 수 있는 메트릭 을 정의하고, 분산 시스템의 영향을 미치는 두 요인 인 통신량과 병렬성을 독립적으로 고려한다.

3. C++ 객체의 정적 할당을 위한 그래프 모델

제3장에서는 C++ 프로그래밍 언어로 개발된 기존 의 시스템을 CORBA 기반의 분산 객체 플랫폼 에서 실행되도록 재공학 하는 경우, 객체들을 효율적 으로 할당하기 위한 그래프 모델을 제안한다. 본 논문의 객체 할당 기법은 정적 할당 기법을 가정한다. 정적 객체 할당이란, 프로그램 실행 전에 모든 객체를 각 프로세서에 할당하는

방법으로서 주로 컴파일 시 또는 설계 시에 객체 할당을 수행한다. 정적 할당 기법에서는, 프로그램의 실행이 시작되면 각 객체는 자신이 할당된 프로세서 이외의 프로세서로 재할당 되거나 이주하지 않는다.

본 논문의 분산 객체 할당 문제를 단순화하기 위하여, C++ 프로그램과 CORBA 환경에 대하여 다음과 같은 사항들을 가정하였다.

- add와 multiply와 같은 모든 기본적인(primitive) 연산자의 실행 비용은 고정되어 있으며, 각 함수 또는 메소드의 실행 비용은 이러한 기본 명령어 (instruction)의 개수로 나타낼 수 있다.
- 모든 분기점(branch point)에 대하여 같은 확률로 제어가 이동함을 가정한다. 따라서, 분기문의 총 실행 비용은 그 분기 문을 구성하는 각 문장 의 실행 비용과 그 문장으로 제어가 이동할 확률 의 곱을 모두 더함으로써 구할 수 있다. 분기문 의 총 실행 비용을 구하는 식은 (식 1)과 같다. 여기에서 P_i는 제어가 i 문장으로 분기할 확률 (= 1 / 총 분기 개수)을 나타내며, $C(i)$는 i 문장의 실행 비용을 의미한다.

$$\sum_{i=1}^{\text{분기문을구성하는문장의개수}} P_i \times C(i) \qquad (1)$$

- 상위 클래스(superclass)로부터 계승된 메소드의 코드는 하위 클래스에 복사(duplicate)된다. 즉, 하위 클래스에서는 계승된 메소드를 실행하기 위하여 상위 클래스의 메소드에 대한 포인터를 얻어올 필요가 없다.
- 각 클래스의 모든 인스턴스(instance)는 오직 하나의 노드에만 할당되고 생성되며, 따라서 코드 이주(code migration)를 지원하지 않는다.
- 할당하고자 하는 클래스의 총 개수는 프로세서 또는 노드의 개수보다 크다.

분산 객체 할당을 위한 그래프 모델은, C++ 응용 프로그램을 표현하기 위한 소프트웨어 모델, 이질적 분산 환경을 표현하기 위한 하드웨어 모델, 각 클래 스가 어느 프로세서에 할당되었는지를 표현하기 위 한 분산 객체 할당 모델로 구성된다.

3.1 소프트웨어 모델

3.1.1 메소드 통신 그래프 (MCG: Method Communication Graph)

메소드 통신 그래프는, 객체에 포함된 메소드 사이에 존재하는 메시지 호출 관계를 표현하는 비순환적(acyclic)이고 방향성이 있으며(directed) 웨이 트(weighted)가 있는 그래프이다. C++ 클래스의 각 메 소드는 이 그래프에서 노드로 모델링되며, 서로 다른 객체에 포함된 두 메소드 사이의 호출(invocation)은 방향성 있는 에지(edge)로 모델링 된다. 에지의 방향 은 메시지를 송신한 객체(caller)의 메소드로부터 메 시지를 수신하는 객체(callee)로 향한다.

노드의 웨이트는, 메소드의 부하 또는 실행 시간을 나타내며 기본 명령어(primitive instruction)의 개수로 표현된다. 에지의 웨이트는 두 메소드 간의 통신 비용, 코드 상에서 메시지 송신 메소드가 수신 메소 드를 직접적으로 호출한 횟수, 메소드 호출 타입 (동기적 또는 비동기적), 프로그램 실행 동안 송신 메소드가 수신 메소드를 호출하는 총 횟수, 프로그램 실행 시 메소드 사이에 존재하는 잠재적인 병렬성 등을 나타낸다. MCG는 다음과 같이 여덟 개의 구성 요소를 가진 튜플로 정의된다.

MCG = (M, MR, MW, MIT, MIW, DMIF, TMIF, MPC)

MCG의 각 구성 요소들은 다음과 같이 정의된다.

■ **M (Methods, 메소드들) = {$c_i.m_j$}**

이것은 MCG의 노드를 구성하는 메소드 집합을 의미한다. 여기에서 $c_i.m_j$는 클래스 i (= c_i)에 포함되어 있는 메소드 j (= m_j)를 가리킨다. 만약 i = 0 이고 j ≠ 0 이면, $c_i.m_j$는 전역 함수(global function)를 의미하며, i = 0 이고 j = 0 ($c_o.m_o$)이면 $c_i.m_j$는 메인 함수(main function)를 의미한다.

MCG의 나머지 구성 요소들을 정의하기 위하여, 다음과 같이 메소드 사이의 호출 관계를 나타내는 몇 가지 용어들이 적용된다.

[정의 3-1] 명시적 호출 (explicit call)

$c_i.m_k$의 코드 안에서 $c_j.m_l$을 호출하는 문장이 존재 하면, $c_i.m_k$는 $c_j.m_l$를 *'명시적으로 호출한다'*고 한다.

[정의 3-2] 직접 호출 (direct call)

$c_j.m_l$의 실행이, $c_i.m_k$가 호출한 다른 객체의 메소드에 의해서 시작되는 것이 아니라, $c_i.m_k$에 의한 직접적인 메시지 전송에 의해 시작되는 경우, $c_i.m_k$는 $c_j.m_l$를 *'직접적으로 호출한다'*고 한다. 명시적인 호출은 모두 직접 호출에 해당하며, 그 역은 성립하지 않는다.

[정의 3-3] 정적 호출 (static call)

$c_i.m_k$가 $c_j.m_l$를 직접적으로 호출할 때, 다음 두 경우에 $c_i.m_k$는 $c_j.m_l$를 *'정적으로 호출한다'*고 한다.

(i) $c_i.m_k$의 코드 안에서 c_j에 대한 변수를 선언하고 멤버 참조 연산자(.)를 통해서 메소드 m_l을 호출하는 경우

(ii) $c_i.m_k$의 코드 안에서 c_j에 대한 포인터 변수를 선언하고, 간접 멤버 참조 연산자(->)를 통해서 가상적(virtual)이 아닌 메소드 m_l을 호출하는 경우

[정의 3-4] 동적 호출 (dynamic call)

$c_i.m_k$가 $c_j.m_l$를 직접적으로 호출할 때, $c_i.m_k$의 코드 안에서 c_j에 대한 포인터 변수를 선언하고, 간접 멤버 참조 연산자(->)를 통해서 가상적인 메소드 m_l을 호출하는 경우, $c_i.m_k$는 $c_j.m_l$를 *'동적으로 호출한다'*고 한다.

[정의 3-5] 묵시적 호출 (implicit call)

$c_i.m_k$가 $c_j.m_l$를 동적으로 호출하는 경우, 동적 바인딩으로 인해 c_j의 자손 클래스에 의해 오버라이 드(override)된 메소드가 런타임 시에 호출될 수 있는데, 이 경우 $c_i.m_k$는 자손 클래스의 그 메소드를 *'묵시적으로 호출한다'*고 한다.

■ **MR (Method Relationships, 메소드 사이의 관계) = {mr_{ikjl}}**

이것은 메소드 사이의 직접 호출 관계를 나타낸다. 만약 $c_i.m_k$가 $c_j.m_l$를 정적 또는 동적으로 호출한다면 mr_{ikjl} = 1이다. 단, $c_i.m_k$가 $c_j.m_l$를 동적으로 호출한 다면 묵시적 호출을 고려해야 한다. 따라서 $c_i.m_k$가 $c_j.m_l$를 동적 호출하고 $c_j.m_l$이 가상 함수라면 mr_{ikjl} = 1이 될 뿐만 아니라, c_j의 모든 자손 클래스(subclass) c_d에 대하여 mr_{ikdl} = 1이 된다. 단, c_j나 c_j의 자손 클래스 중 추상 클래스(abstract class)에 대한 mr 값은 0이 된다. 그 이유는 추상 클래스는 런타임 시에 인스 턴스가 생기지 않아 호출 관계가 성립할 수 없기 때 문이다. 집합 MR의 정의에 의해, 간접 호출에 대한 용어가 다음과 같이 정의된다.

[정의 3-6] 간접 호출 (indirect call)

mr_{xyik} = 1이고 mr_{ikjl} = 1이면, $c_x.m_y$는 $c_j.m_l$을 *'간접적으로 호출한다'*라고 한다. 일반적으로 $c_u.m_v$가 $c_x.m_y$를 간접적으로 호출하고 $c_x.m_y$가 $c_j.m_l$을 간접적으로 호출하면, $c_u.m_v$는 $c_j.m_l$을 간접적으로 호출한다.

■ **MW (Method Workloads, 메소드의 부하) = {mw_{ik}}**

mw_{ik}는 $c_i.m_k$의 실행 시 드는 비용 또는 부하를 의미하며, 본 모델에서는 일정 개수의 기본 명령

어의 배수로 표현됨을 가정한다. 이것은 $c_i.m_k$를 실행하는 데 드는 시간(time complexity)으로 표현될 수도 있다.

■ **MIT (Method Invocation Type, 메소드 호출 타입) = {mit_{ikjl}}**

이것은 동기적 또는 비동기적 메소드 호출 타입을 나타낸다. 만약 $c_i.m_k$가 $c_j.m_l$를 비동기적으로 직접 호출하면 mit_{ikjl} = 1 이고, 동기적으로 직접 호출하면 mit_{ikjl} = 0 이다. 이 값은 메소드 또는 객체 사이의 병렬성을 측정하기 위한 바탕을 제공한다.

■ **MIW (Method Invocation Workloads, 메소드 호출 비용) = {miw_{ikjl}}**

miw_{ikjl}는, $c_i.m_k$가 $c_j.m_l$를 호출할 때의 통신 비용을 의미한다. 만약 i = j 이면, 같은 객체의 두 메소드를 의미하므로 miw_{ikjl} = 0 이다. 일반적으로 주어진 데이터가 네트워크 회선을 통해 전송되는 시간은, 전송 되는 데이터의 양, 스위칭(switching)의 종류, 각 노드들의 연결 상태(topology), 각 네트워크 회선의 대역폭(bandwidth), 네트워크 전체의 총 부하량 등에 의해 결정된다. 이 중에서 전송되는 데이터 양이 통신 비용에 가장 큰 영향을 미친다. 문제를 단순화 하기 위하여, 본 모델에서는 모든 노드로의 접근은 동일한 일정 시간 내에 이루어짐을 가정하며, 두 메 소드 사이의 통신 비용은 전송되는 데이터의 양으로 표현한다. 데이터의 양은 일정한 데이터 블록의 배수 로 표현되거나 전송 시의 패킷(packet) 수로 표현된다.

■ **DMIF (Direct Method Invocation Frequency, 직접적인 메소드 호출 횟수) = {$dmif_{ikjl}$}**

$dmif_{ikjl}$는 $c_i.m_k$의 직접적인 호출에 의해 런타임 시에 $c_j.m_l$이 실행되는 횟수를 나타낸다. $c_i.m_k$의 직접 호출에 의해 $c_j.m_l$이 실행되는 경우는 다음 세 가지로 나누어진다.

(i) $c_i.m_k$에 의한 정적 호출에 의해 $c_j.m_l$가 실행되는 경우

(ii) $c_i.m_k$에 의한 동적 호출에 의해 $c_j.m_l$가 실행되는 경우

(iii) $c_i.m_k$에 의한 묵시적 호출에 의해 $c_j.m_l$가 실행되는 경우 따라서, $dmif_{ikjl}$은 (식 2)에 의해 구할 수 있다.

$$dmif_{ikjl} = SN_{ikjl} + DN_{ikjl} + IN_{ikjl} \quad (2)$$

= ($c_i.m_k$의 정적 호출에 의해 $c_j.m_l$이 실행되는 횟수) + ($c_i.m_k$의 동적 호출에 의해 $c_j.m_l$이 실행되는 횟수) + ($c_i.m_k$의 묵시적 호출에 의해 $c_j.m_l$가 실행되는 횟수)

(식 2)의 각 인자는 다음과 같이 정의된다.

• SN_{ikjl} (Number of Static Invocations)

집합 S_{ikjl}를, $c_i.m_k$가 $c_j.m_l$을 정적 호출하는 문장들의 집합이라고 할 때, SN_{ikjl}은 (식 3)으로 구할 수 있다.

$$SN_{ikjl} = \sum_{s \in S_{ikjl}} P(if_s) \times N(loop_s) \quad (3)$$

(식 3)에서 $P(if_s)$과 $N(loop_s)$의 의미는 다음과 같다.

① $P(if_s)$

만약 호출문 s가 if나 switch-절과 같은 분기문(conditional statement) 안에 포함되었다면, 제어가 그 문장으로 옮겨오기 위한 확률에 따라 호출 횟수 가 달라진다. $P(if_s)$는 그러한 확률을 나타내며, 호출 문장이 if나 switch-절에 포함되지 않는 경우에는 $P(if_s)$의 값이 1이 된다. 본 모델에서는 모든 분기 점에 대하여 같은 확률로 제어가 이동함을 가정한다.

② $N(loop_s)$:

만약 호출문 s가 while-루프에 포함되었다면,

호출 회수에 루프 반복 횟수를 곱해야 한다. N ($loop_s$) 는 이러한 반복 횟수를 나타낸다.

- DN_{ikjl} (Number of Dynamic Invocations)

동적 호출 시 수신 메소드가 실행되는 횟수를 구하는 방법은, 동적 바인딩으로 인하여 더욱 복잡 해진다. 실행 횟수는, 분기점에서 호출문으로 제어가 이동할 확률, 반복 루프에서 메소드 호출 문장이 반복 수행 될 횟수 뿐 만 아니라, 그 객체가 런타임 시에 실제로 호출될 확률에 영향을 받는다. 집합 D_{ikjl}을, $c_i.m_k$가 $c_j.m_l$을 동적으로 호출하는 문장들의 집합이라고 할 때 DN_{ikjl}의 값은 (식 4)와 같이 표현된다.

$$DN_{ikjl} = \sum_{d \in D_{ikjl}} P(if_d) \times N(loop_d) \times P(dyn_d, c_j) \quad (4)$$

여기에서 $P(if_s)$, $N(loop_s)$, $P(dyn_d, c_j)$의 의미는 다음과 같다.

① $P(if_d)$는 $P(if_s)$ 경우와 동일하다.
② $N(loop_d)$는 $N(loop_s)$ 경우와 동일하다.
③ $P(dyn_d, c_j)$

이 인자는 객체 지향 패러다임의 큰 특징인 클래스 계층 구조와 동적 바인딩을 고려하는 인자이다. 어떤 메소드 $c_j.m_l$이 동적 호출을 통해서 불려지면, 실제로 런타임 시에 실행되는 메소드가 그 클래스의 메소드($c_j.m_l$)인지 자손 클래스들의 메소드인지를 컴파일 시에는 결정할 수 없다. 예를 들어 $c_i.m_k$가 c_j에 대한 포인터를 통해서 가상 함수 m_l을 호출하는 경우, 정적 코드 분석을 통해서는 부모 클래스 c_j와 그 자손 클래스들 중에서 어떤 m_l이 실제로 실행되는지 결정할 수 없다. 따라서 동적 호출이 발생하는 경우에는 런타임 시에 특정 클래스의 메소드가 호출될 확률을 구해야 하며, $P(dyn_d, c_j)$ 인자가 이것을 다룬다. 본 모델에서는 부모 클래스와 자손 클래스들이 똑같은 확률로 불려진다고 가정한다. 일반적으로 집합 D_{ikjl}의 원소 d에 대하여, 런타임 시에 특정 클래스 cx가 호출될 확률 $P(dyn_d, c_x)$의 값은 클래스 c_j의 자손 클래스 개수에 따라 달라지며, 구하는 식은 (식 5, 6)과 같다.

만일 c_x가 추상 클래스가 아니면 (5)

$$P(dyn_d, c_x) = 0$$

만일 c_x가 추상 클래스이면 (6)

$$P(dyn_d, c_x) = \frac{1}{(1 + c_j \text{의 자손클래스의개수} - ACN(c_j))}$$

(식 5)에서, $ACN(c_j)$ (number of abstract class)는 c_j 자신과 c_j의 자손 클래스 중에서 추상 클래스의 개수를 의미한다. 분모에서 $ACN(c_x)$를 빼는 이유는, 추상 클래스는 인스턴스가 생기지 않아 호출 관계가 성립될 수 없기 때문이다. 또한 c_x가 추상 클래스 이면 c_x의 인스턴스가 생기지 않기 때문에 $c_x.m_l$이 호출될 수 없고 따라서 $P(dyn_d, c_x)$의 값은 0이 된다 (식 6).

- IN_{ikjl} (Number of Implicit Invocations)

이 인자는 $c_i.m_k$가 c_j의 조상 클래스를 동적 호출 했을 때, 동적 바인딩으로 인해 $c_j.m_l$이 실행되는 횟수 를 나타낸다. c_j의 조상 클래스의 가상 메소드 m_l이 동적으로 호출되는 경우에, c_j는 명시적으로 호출 되지는 않았지만 이 조상 클래스로부터 계승하여 오버라이드한 $c_j.m_l$이라는 메소드가 런타임 시에 실행될 가능성이 있다. 묵시적 호출에 의해 $c_j.m_l$이 불려질 횟수를 계산하기 위해서는 c_j의 조상 클래 스에 대한 동적 호출로 인해 $c_j.m_l$이 묵시적으로 호출 될 확률을 계산해야 한다. 이 값은, c_j의 모든 조상 클래스 중에서 $c_i.m_k$가 가상 메소드 m_l을 동적 호출 하는 모든

클래스에 대하여 $c_j.m_l$이 호출될 확률을 구해서 다 더하면 된다. 따라서 P_{jl}을, c_j의 조상 클래 스들 중에서 가상 메소드 m_l을 포함하는 클래스들의 집합이라고 할 때, IN_{ikjl}을 구하는 식은 (식 7)과 같이 정의된다. (식 7)의 각 인자는 앞에서 정의된 것과 동일하다.

$$IN_{ikjl} = \sum_{c_p \in P_{jl}} \sum_{d \in D_{ikpl}} P(if_d) \times N(loop_d) \times P(dyn_d, c_j) \quad (7)$$

■ **TMIF (Total Method Invocation Frequency, 메소드 호출 총 횟수) = {$tmif_{ikjl}$}**

$tmif_{ikjl}$는 런타임 중에 $c_i.m_k$의 호출에 의해 $c_j.m_l$가 실행된 총 횟수를 나타내며, $dmif_{ikjl}$의 값을 바탕으로 하여 계산된다. 일반적으로 $tmif_{ikjl}$은 (식 8)과 같이 정의된다.

$$tmif_{ikjl} = dmif_{ikjl} \times \sum_{\forall c_x.m_y : mr_{xyik}=1} tmif_{xyik} \quad (8)$$

메인 함수($c_0.m_0$)가 직접 호출하는 모든 메소드에 대해서는 $tmif_{xyjl}$의 값과 $dmif_{xyjl}$의 값이 같다. 즉, x = 0 이고 y = 0 이면 $tmif_{xyjl} = dmif_{xyjl}$이 성립된다. 또한 $mr_{ikjl} = 0$인 모든 i, k, j, l에 대해서 $tmif_{ikjl}$ 값은 0 이다.

■ **MPC (Method Potential Concurrency, 메소드 사이의 병렬성) = {mpc_{ikjl}}**

mpc_{ikjl}는 두 메소드 $c_i.m_k$와 $c_j.m_l$ 사이의 병렬성을 나타내며, 0과 1 사이의 값을 가진다. 만약 mpc_{ikjl}의 값이 0이면 두 메소드 사이에 병렬성이 없음을 의미하며, 두 메소드는 순차적으로 실행되어야 할 것이다. 즉, 항상 $c_i.m_k$는 $c_j.m_l$를 호출한 후 $c_j.m_l$이 수행을 마치고 결과값을 반환할 때 까지 기다려야 한다. 이 때 두 메소드가 서로 다른 노드에 할당되면, 병렬성으로 인해 얻는 이익이 없으며 메소드 사이의 통신 비용만 증가한다. 따라서 mpc_{ikjl}의 값이 0인 경우에 두 메소드는 동일한 노드에 할당되는 것이 유리하다. 한편, mpc_{ikjl}의 값이 1인 경우에는, 두 메소 드 사이에 병렬성이 존재하며, 이러한 병렬성으로 인한 성능의 향상을 얻기 위해서 두 메소드는 서로 다른 노드에 할당되어야 할 것이다.

프로세스들이 동시에 존재하면 이들은 서로 병렬 수행(concurrent) 된다고 한다. 두 메소드 사이의 병렬성 계산은 MCG에 존재하는 메소드 사이의 호출 관계와 메소드 사이의 호출 타입(동기적 또는 비동기 적) 등의 정보에 의존한다. 먼저, 상호 작용이 없는 메소드 쌍은 서로 독립적으로 실행되므로, $mr_{ikjl} = 0$ 인 모든 i, k, j, l에 대하여 mpc_{ikjl}의 값을 1로 설정한다. 상호 작용이 있는 두 메소드 사이의 병렬성은 한 메소드가 다른 메소드를 동기적으로 호출하는지, 비동기적으로 호출하는지에 따라 달라진다.

● 비동기적 호출 (asynchronous) 인 경우

만약 메소드 $c_i.m_k$가 $c_j.m_l$를 비동기적으로 호출한 다면, $c_i.m_k$ (caller)는 $c_j.m_l$를 호출한 후 블록(blocked) 되지 않고 나머지 코드를 계속해서 실행한다. 이 경우에는 두 메소드를 서로 다른 노드에 할당하는 것이 성능 향상과 실행 시간을 줄이는 효과를 가져 온다. 따라서 (식 9)와 같이 메소드 $c_i.m_k$가 $c_j.m_l$를 비동기적으로 호출하는 경우 MPC의 값은 1이 된다.

$$\forall i,k : mr_{ikjl} = 1 \wedge mit_{ikjl} = 1,\ mpc_{ikjl} = 1 \quad (9)$$

● 동기적 호출 (synchronous) 인 경우

동기적 호출 관계가 있는 두 메소드 사이의 병렬성 계산은 좀 더 복잡하다. 만약 메소드 $c_i.m_k$가 $c_j.m_l$를 동기적으로 호출한다면, $c_i.m_k$는 $c_j.m_l$의 반환 값을 기다려야 하므로 항상 $c_j.m_l$의 수행이 끝날 때 까지 블록 되어 있어야 한다. 일반적으로 $c_i.m_k$와 $c_j.m_l$ 사이에 동기적 호출 관계가 존재한다면, 두 메소드는 동시에 병렬적으로 실행될 수

없으며, 따라서 mpc_{ikjl}의 값은 0이 될 것이다. 그러나, $c_i.m_k$와 $c_j.m_l$ 사이에 동기적 호출 관계가 있더라도 두 메소드가 병렬적 으로 실행될 수 있는 예외적인 상황이 존재한다. 그 이유는 $c_i.m_k$이외에도 다른 메소드들이 $c_j.m_l$을 동기 적으로 호출할 수 있기 때문이다.

예를 들어 MCG의 일부분인 그림 1을 살펴보자. 이 그림에서 세 메소드 $c_i.m_k$, $c_2.m_2$, $c_3.m_3$는 각각 $c_j.m_l$와 동기적 호출 관계를 가지고 있다 ($c_i.m_k$, $c_2.m_2$, $c_3.m_3$는 $c_j.m_l$을 동기적으로 호출하는 모든 메소드라고 가정하자). 이 때 $c_i.m_k$와 $c_j.m_l$ 사이의 동기적 호출 관계 때문에 $c_i.m_k$는 항상 $c_j.m_l$를 기다려야 하는 것처럼 보이지만, $c_i.m_k$와 $c_j.m_l$가 동시에 실행 가능한 경우가 존재한다. 그 이유는 $c_i.m_k$ 뿐 만 아니라 $c_2.m_2$와 $c_3.m_3$가 $c_j.m_l$를 동기적으로 호출할 가능성이 있기 때문이다. 즉, $c_i.m_k$가 $c_j.m_l$을 동기적으로 호출한다면, 이 두 메소드는 같은 노드에 할당되는 것이 유리하고, 따라서 이 둘 사이에는 끄는 힘이 존재한다고 할 수 있다. 그런데 그림 1과 같이 $c_i.m_k$ 뿐 만 아니라 $c_2.m_2$, $c_3.m_3$가 $c_j.m_l$을 동기적으로 호출하는 경우에는 세 쌍에 대해서 어느 메소드 쌍이 더 끄는 힘이 큰가를 측정해야 한다. 끄는 힘이 클수록 병렬성은 작아지며 따라서 같은 노드에 할당하는 것이 유리 하다. 이 값은 세 메소드가 $c_j.m_l$를 동기적 호출하는 총 횟수에 따라 달라진다. 예를 들어, $c_i.m_k$이 $c_j.m_l$을 상대적으로 많이 호출한다면 끄는 힘이 가장 크게 되고, 그 결과 두 메소드 사이의 병렬성은 작아지게 된다.

일반적으로, $c_i.m_k$와 $c_j.m_l$ 사이에 동기적 호출 관계가 있을 때 병렬성 mpc_{ikjl}은 (식 10)과 같이 구할 수 있다. (식 10)에서 $c_i.m_k$가, $c_j.m_l$를 동기적으로 호출하는 유일한 메소드인 경우에는 mpc_{ikjl}의 값은 1 - 1 = 0 이 됨을 알 수 있다.

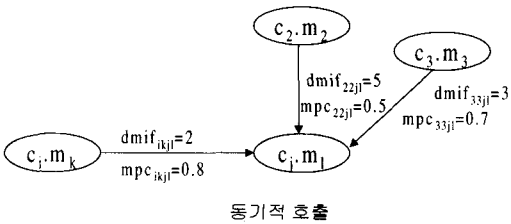
$$mpc_{ikjl} = \text{concurrency between } c_i.m_k \text{ and } c_j.m_l$$
$$= 1 - \frac{c_i.m_k\text{가 } c_j.m_l\text{를 호출했을 때 } c_i.m_k\text{가 블럭될 횟수}}{c_j.m_l\text{을 동기적호출하는 메소드들의 호출 총횟수}}$$
$$= 1 - \frac{c_i.m_k\text{가 } c_j.m_l\text{를동기적으로 호출한 횟수}}{c_j.m_l\text{을 동기적호출하는 메소드들의 호출 총횟수}} \quad (10)$$

그런데, 실행 시의 호출 횟수는 dmif의 값이 아니라 tmif의 값에 의존한다.따라서 더욱 정확한 병렬성을 계산하기 위해서는 dmif인자를 tmif로 확장해야 한다. 또한, 위의 식에서 직접 호출 횟수 뿐 만 아니라 간접 호출 횟수도 고려해야 한다. 예를 들어 그림 2와 같이 $c_j.m_l$의 실행이, $c_i.m_k$의 직접 호출 뿐 만 아니라 간접 호출에 의해서도 가능하기 때문이다.

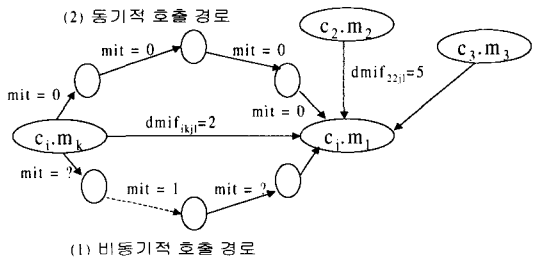
이를 위해서 간접 호출에 대하여 두 가지의 경로 타입을 정의하였다.

[정의 3-7] 비동기적 호출 경로 (asynchronous call path)

$c_i.m_k$이 $c_j.m_l$를 간접 호출할 때, mit_{xyuv} = 1 (비동기 호출)인 두 메소드 $c_x.m_y$와 $c_u.m_v$가 $c_i.m_k$에서 $c_j.m_l$ 까지의 호출 경로 중간에 존재하면, 이 호출 경로를 '*비동기적 호출 경로*'라고 한다.



(그림 1) 동기적 메소드 호출 관계에서의 병렬성



(그림 2) 직접 및 간접 메소드 호출 관계에서의 병렬성

$$mpc_{ikjl} = 1 - \frac{c_i.m_k\text{의 간접 및 직접 호출에 의해 } c_j.m_l\text{이 호출되고 } c_i.m_k\text{이 블럭된 횟수}}{c_j.m_l\text{을 동기적 호출하는 메소드의 호출 총 횟수}}$$

$$= 1 - \frac{\left(\sum_{\forall c_x.m_y : mr_{xyik}=1} tmif_{xyik} \right) \times \sum_{\forall p \in NoAsyncPath_{ikjl}} \left(\Pi_{path}(p) \right)}{\left(\sum_{\forall c_x.m_y : mr_{xyjl}=1 \wedge mit_{xyjl}=0} c_u.m_v : mr_{uvxy}=1 \; tmif_{uvxy} \right) \times \sum_{\forall p \in NoAsyncPath_{xyjl}} \left(\Pi_{path}(p) \right)} \quad (11)$$

[정의 3-8] 동기적 호출 경로 (synchronous call path)

$c_i.m_k$이 $c_j.m_l$를 간접 호출할 때, 호출 경로 중간에 존재하는 모든 메소드$c_x.m_y$,$c_u.m_v$ (mr_{xyuv} = 1)에 대하여 mit_{xyuv} = 0 (동기적 호출)인 호출 경로를 '*동기적 호출 경로*'라고 한다.

$c_i.m_k$가 $c_j.m_l$을 호출한 후 블록 될 횟수를 구하는 식에서 비동기적 호출 경로는 제외시킨다. 그 이유는, 호출 경로 중간에 비동기적 호출이 존재하는 경우에는 $c_i.m_k$가 $c_j.m_l$의 실행이 종료되기를 기다리지 않기 때문에다. 따라서 동기적 메소드 호출 시 두 메소드 사이의 병렬성 메트릭은 (식 11)과 같이 수정된다.

mit_{ikjl} = 0 (동기적 호출)인 모든 메소드 쌍, $c_i.m_k$와 $c_j.m_l$에 대하여,

여기에서, $NoAsyncPath_{ikjl}$는, $c_i.m_k$에서 $c_j.m_l$로의 모든 호출 경로 중 동기적 호출 경로의 집합을 의미하며, $\Pi_{path}(p)$ 는 호출 경로 p에 존재하는 모든 에지의 dmif을 곱한 값을 의미한다.

3.1.2 클래스 통신 그래프(CCG: Class Communication Graph)

CCG는 클래스 간의 통신 패턴과 각 클래스의 여러 가지 특징들을 표현하는 그래프이다. 이 그래프는 방향성이 없고 웨이트가 있는 그래프로서, 각 클래스 또는 객체는 노드로 모델링 되고 클래스 사이의 통신 관계는 에지로 모델링 된다. 노드의 웨이트는 클래스에 포함되어 있는 메소드들의 실행 시 총 부하량을 나타내며, 에지의 웨이트는 클래스 사이의 통신 비용과 클래스 사이의 병렬성을 나타낸다. 이 그래프는 MCG의 각 인자로부터 구해지며, 객체 할당 알고리즘은 CCG에 포함되어 있는 정보를 이용 하여 객체 할당을 수행한다. CCG는 다음과 같은 구성 요소를 가진 튜플로 정의된다.

CCG = (C, CR, CM, CW, CCW, CPC)

CCG의 각 구성 요소들은 다음과 같이 정의된다.

■ C (Classes, 클래스들) = {c_i}

이것은 객체 지향 프로그램을 구성하는 클래스들을 나타내며, c_i는 클래스 i를 의미한다.

■ CR (Class Relationships, 클래스 사이의 관계) = {cr_{ij}}

이것은 클래스 간의 통신 관계를 나타낸다. 클래스 i에 존재하는 어떤 메소드가 클래스 j의 어떤 메소드를 직접적으로 호출하면 cr_{ij} = 1 이고, 그렇지 않은 경우에는 cr_{ij} = 0이다. 만약 i = j이면,

c_{ij} = 0 이다.

■ **CM(Class-Method, 클래스와 메소드의 포함 관계) = {cm_{ik}}**

이것은, 특정 메소드가 어떤 클래스에 속하는지 속하지 않는지를 나타낸다. 만약 클래스 i가 메소드 k를 포함하면, cm_{ik} = 1 이다. 그렇지 않으면 cm_{ik}의 값은 0 이다.

■ **CW (Class Workloads, 클래스의 부하) = {cw_i}**

cw_i는 클래스 i의 실행 시 부하를 의미하며, 이는 클래스 i에 포함되는 각 메소드의 실행 시 부하와 그 메소드가 호출되는 총 횟수를 곱한 값의 합으로 계산 된다. 각 메소드의 부하 단위가 기본 명령어의 개수이므로 cw_i의 값도 기본 명령어의 개수로 표현 된다. cw_i를 구하는 식은 (식 12)와 같이 정의된다.

$$cw_i = \sum_{\forall m_k \in c_i} \sum_{\forall j,l : mr_{jlik}=1} (mw_{ik} \times tmif_{jlik}) \quad (12)$$

■ **CCW (Class Communication Workloads, 클래스 사이의 통신량) = {ccw_{ij}}**

ccw_{ij}는 클래스 i와 클래스 j 사이의 통신 비용을 나타낸다. 이것은, 클래스 i에 포함된 메소드 중 클래스 j와 통신하는 메소드 사이의 통신 비용과 클래스 j에 포함된 메소드 중 클래스 i와 통신하는 메소드 사이의 통신 비용의 총합으로 계산된다. 이 때 각 메소드 간의 통신 비용에는 실행 중 발생하는 총 호출 횟수가 곱해진다. 만약 i와 j가 같다면, ccw_{ij} = 0이다. ccw_{ij}를 구하는 식은 (식 13)과 같이 정의된다.

$$ccw_{ij} = \sum_{\forall m_l \in c_j} \sum_{\forall m_k \in c_i : mr_{ikjl}=1} (miw_{ikjl} \times tmif_{ikjl}) + \sum_{\forall m_k \in c_i} \sum_{\forall m_l \in c_j : mr_{jlik}=1} (miw_{jlik} \times tmif_{jlik}) \quad (13)$$

■ **CPC (Class Potential Concurrency, 클래스 사이의 병렬성) = {cpc_{ij}}**

cpc_{ij}는, 클래스 i와 클래스 j 사이의 병렬성을 나타 낸다. 이것은 클래스 i에 포함된 메소드 중 클래스 j와 통신하는 메소드 사이의 병렬성과 클래스 j에 포함된 메소드 중 클래스 i와 통신하는 메소드 사이의 병렬성의 총합으로 계산된다. 두 메소드 사이에 호출 관계가 없는 경우의 두 메소드 사이의 mpc 값은 1이며, 이 값도 두 클래스 사이의 병렬성 값에 더해 진다.

각 메소드 사이의 통신은 비동기적 호출과 동기적 호출이 구분되며, 비동기적 호출인 경우에는 메소드 사이의 병렬성에 실행 중 총 호출 횟수가 곱해진다. 동기적 호출이나 호출 관계가 없는 경우에는 실행 중 총 호출 횟수가 곱해지지 않으며, 그 이유는 클래스의 병렬성을 결정짓는 영향력에 있어서 비동기적 호출이 미치는 영향이 훨씬 크기 때문이다. cpc_{ij}를 구하는 식은 (식 14)와 같이 정의된다.

$$\begin{aligned} cpc_{ij} = & \sum_{\forall m_l \in c_j} \sum_{\forall m_k \in c_i : mr_{ikjl}=1 \wedge mit_{ikjl}=1} (mpc_{ikjl} \times tmif_{ikjl}) + \sum_{\forall m_k \in c_i} \sum_{\forall m_l \in c_j : mr_{jlik}=1 \wedge mit_{jlik}=1} (mpc_{jlik} \times tmif_{jlik}) \\ & + \sum_{\forall m_l \in c_j} \sum_{\forall m_k \in c_i : mr_{ikjl}=1 \wedge mit_{ikjl}=0} (mpc_{ikjl}) + \sum_{\forall m_k \in c_i} \sum_{\forall m_l \in c_j : mr_{jlik}=1 \wedge mit_{jlik}=0} (mpc_{jlik}) \\ & + \sum_{\forall m_l \in c_j} \sum_{\forall m_k \in c_i : mr_{ikjl}=0} (mpc_{ikjl}) + \sum_{\forall m_k \in c_i} \sum_{\forall m_l \in c_j : mr_{jlik}=0} (mpc_{jlik}) \end{aligned} \quad (14)$$

3.2 하드웨어 모델

분산 시스템을 구성하는 하드웨어의 여러 가지 특징들은 시스템 그래프(SG)로 표현된다. 이 그래프 는 방향성이 없고 웨이트가 있는 그래프로서, 분산 환경을 구성하고 있는 각 프로세서는 노드로 모델링 되고, 각 프로세서 사이의 네트워크 연결 관계는 에지로 모델링 된다. 각 노드의 웨이트는 각 프로세 서의 컴퓨팅 능력을 나타내며, 에지의 웨이트는 네트워크의 데이터 전송 능력을 나타낸다. SG는 다음과 같은 구성 요소를 가진 그래프로 정의된다.

SG = (P, PR, PC, NC)

SG의 각 구성 요소들은 다음과 같이 정의된다.

■ **P (Processors, 프로세서들) = {p_i}**

p_i는 분산 환경에 존재하는 각 프로세서를 나타 낸다.

■ **PR (Processor Relationships, 프로세서 사이의 연결 관계) = {pr_{ij}}**

이것은 프로세서 사이의 연결 상태를 나타낸다. 만약 프로세서 i와 프로세서 j가 네트워크 회선에 의해 서로 연결되어 있으면 pr_{ij} = 1 이고, 그렇지 않으면 pr_{ij} = 0 이다.

■ **PC (Processor Capability, 프로세서의 능력) = {pc_i}**

이것은 각 프로세서의 컴퓨팅 능력을 나타낸다. pc_i는 프로세서 i의 컴퓨팅 능력을 나타내며, 일정한 개수의 기본 명령어를 실행하는데 걸리는 평균 시간으로 표현된다.

■ **NC (Network Capability, 통신 회선의 전송 능력) = {nc_{ij}}**

이것은 두 프로세서를 연결하는 통신 회선의 전송 능력을 나타낸다. nc_{ij}는 프로세서 i와 프로세서 j를 연결하는 통신 회선에서 단위 정보를 전송하는데 걸리는 평균 시간으로 표현된다.

3.3 분산 객체 할당 모델

각 클래스가 어느 프로세서에 할당되었는지를 나타내기 위한 분산 객체 할당 모델은 분산 객체 할당 그래프(DOAG: Distributed Object Allocation Graph)로 표현되며, 다음과 같은 구성 요소를 가진다.

DOAG = (NP, CA, CCP, NCP)

DOAG의 각 구성 요소들은 다음과 같이 정의된다.

■ **NP (Number of Processor, 프로세서 개수) = np**

이것은, 분산 환경을 구성하는 프로세서 또는 노드의 총 개수를 나타낸다.

■ **CA (Class Allocation, 클래스 할당) = {ca_{ip}}**

이것은, 각 클래스가 어느 프로세서 또는 노드에 할당되었는지를 나타낸다. 만약 클래스 i가 프로세서 p에 할당되었으면 ca_{ip} = 1 이고, 그렇지 않은 경우에는 ca_{ip} = 0 이다. 모든 클래스 i에 대하여 다음 두 조건을 만족해야 한다. (식 15)는, 객체 할당에서 한 클래스가 여러 개의 프로세서에 중복 할당되지 않아야 함을 의미하며, (식 16)는, 모든 클래스는 적어도 하나의 프로세서에 할당되어야 함을 의미 한다.

$$\forall i, p \neq q, \qquad ca_{ip} \times ca_{iq} = 0 \tag{15}$$

$$\forall i, \qquad \sum_{p} ca_{ip} = 1 \tag{16}$$

■ **CCP(Class-Class-Processor, 두 클래스가 같은 프로세서에 할당) = {ccp_{ij}}**

이것은 두 클래스가 서로 같은 프로세서에 할당 되었는지 또는 다른 프로세서에 할당되었는지를 나타낸다. 만약 두 클래스 i와 j가 같은 프로세서에 할당되었으면, ccp_{ij} = 0 이고, 그렇지 않은 경우에는 ccp_{ij} = 1 이다. ccp_{ij}의 값은 (식 17)에 의해 구할 수 있다.

$$ccp_{ij} = 1 - \sum_{p=1}^{\text{프로세서갯수}} ca_{ip} ca_{jp} \qquad (17)$$

■ **NCP(The Number of Class in Processor, 프로세서에 할당된 클래스 개수) = {ncp_i}**

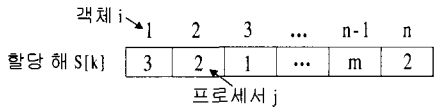
ncp_i는, 프로세서 i에 할당된 클래스들의 개수를 나타낸다.

4. 유전자 알고리즘을 이용한 정적 객체 할당

분산 객체 할당 문제는 세 가지 목적 - 통신 비용의 최소화, 병렬성의 최대화, 부하 균등성의 최대화 - 을 동시에 만족시키는 해를 찾는 다중 목적 함수 최적화 문제이다. 본 장에서는 다중 목적 함수를 위한 유전자 알고리즘 중에서, 다양한 해들을 찾고, 각 해들이 일정한 간격을 유지하도록 하여 균등한 분포의 해를 찾는데 효율적인, Niched 파레토 유전자 알고리즘(NPGA: Niched-Pareto Genetic Algorithm)[9]에 기반한 객체 할당 기법을 제안한다. 본 논문의 객체 할당 알고리즘에서 사용한, NPGA 알고리즘의 각 요소들은 다음과 같다.

4.1 염색체 구조

할당 해를 나타내는 염색체의 구조는, 프로세서의 개수가 m이고 클래스의 개수가 n인 경우 그림 3과 같다.



(그림 3) 객체 할당 해의 염색체 구조

4.2 적합도 함수 (Fitness Function)

유전자 알고리즘에서 품질이 좋은 부모 해를 선택하여 자손을 생성하기 위해서는 모집단 또는 해 집합 에 존재하는 각 해들의 품질을 측정해야 하는데, 이를 위한 함수가 적합도 함수이다. 분산 객체 할당 을 위한 유전자 알고리즘의 적합도 함수 F는 다음과 같이 정의된다.

$$F = (f_1, f_2, f_3)$$

$$f_k : S_i \mapsto \Re^+, \quad k = 1,2,3, \quad i = 1,\ldots,\text{해의개수}$$

여기에서, 각 f_k의 의미는 다음과 같다.

- f_1 (객체 사이의 병렬성 손실량): 이 목적 함수는 각 할당 해에 대한 병렬성의 총 손실량을 계산하는 함수로서, 병렬성 손실량은 (식 18)에 의해 계산된다. 이 식은, 병렬성이 존재하는 두 객체가 서로 같은 프로세서에 할당됨으로 인해 손실되는 병렬성의 합을 의미한다.

$$f_1 = \sum_{p} \sum_{i} \sum_{j \neq i} cpc_{ij} \times ca_{ip} \times ca_{jp} \qquad (18)$$

- f_2 (객체 사이의 통신 비용): 이 목적 함수는 각 할당 해에 대한 전체 네트워크 통신 비용을 계산하는 함수로서, 통신 비용은 (식 19)에 의해 계산된다. 이 식은, 서로 다른 프로세서에 할당된 객체의 쌍에 대한 통신 비용과 네트워크 대역폭을 곱한 값의 합을 의미한다.

$$f_2 = \sum_p \sum_{q \neq p} \sum_i \sum_{j \neq i} ccw_{ij} \times nc_{pq} \times ca_{ip} \times ca_{jq} \quad (19)$$

- f_3 (부하 비균등성): 이 목적 함수는 각 할당해에 대하여, 프로세서에 대한 부하가 어느 정도 불균등하게 분포되었는지를 계산하는 함수로서, (식 20, 21)에 의해 부하 비균등성을 계산한다. 이 식은, 각 프로세서에 할당된 부하와 프로세서 능력의 곱에 대한 분산을 의미한다. 이 분산의 값이 작을수록, 부하의 비균등성이 작아지며, 따라서 부하는 균등하게 분포되었음을 의미한다.

$$f_3 = \left\{ \sum_p \left(\sum_i cw_i \times pc_p \times ca_{ip} - TL/N \right)^2 \right\} / N \quad (20)$$

$$TL = \sum_p \sum_i cw_i \times pc_p \times ca_{ip} \quad (21)$$

4.3 σ_{share}의 크기

Niche의 크기를 나타내는 σ_{share}의 값을, 본 논문에서는 [10]에서 제안된 (식 22)을 이용하여 주어진 해집합의 크기 N으로부터 구한다.

$$N = \frac{\prod_{i=1}^{q} (M_i - m_i + \sigma_{share}) - \prod_{i=1}^{q} (M_i - m_i)}{\sigma_{share}^q} \quad (22)$$

각 목적 함수에 대한 최대값(Mi)과 최소값(mi)을 구하는 방법은 다음과 같다.

- M_1(f_1의 최대값): 객체 사이의 모든 통신이 가장 느린 네트워크 회선을 통하여 일어난다고 가정했을 때의, 네트워크 총 통신 비용을 구한다.
- m_1(f_1의 최소값): 모든 객체가 한 노드에 할당되었을 경우의 네트워크 총 통신 비용을 구한다. 따라서 이 경우의 m_1 값은 0이 된다.
- M_2(f_2의 최대값): 모든 객체가 같은 프로세서 에 할당되었을 경우의 병렬성 총 손실량을 구한다.
- m_2(f_2의 최소값): 모든 객체가 서로 다른 프로세서에 할당되었을 경우의 병렬성 총 손실량을 구한다. 따라서, 이 경우의 m_2 값은 0이 된다.
- M_3(f_3의 최대값): 모든 객체가 가장 느린 하나의 프로세서에 할당되었을 경우에, 각 프로세서 에 걸리는 부하 사이의 분산을 구한다.
- m_3(f_3의 최소값): 각 프로세서에 걸리는 부하가 가장 고르게 분포 되었을 때의 분산을 구한다. 즉, 모든 프로세서에 같은 부하가 걸렸을 때를 의미한다. 따라서, 이 경우의 m_3 값은 0이 된다.

4.4 t_{dom} (토너먼트의 크기)

NPGA에서 토너먼트의 크기(t_{dom})는 알고리즘의 성능에 큰 영향을 미친다. 임의의 실험 데이터에 대한 실험 결과로부터 t_{dom}을 9로 정하였다.

4.5 유전 연산자

■ 선택 연산자 (Selection Operator)

자손을 생성하기 위한 부모를 선택하는 선택 연산자로는 NPGA에서 정의하는 파레토 도미네이션 토너먼트 기법을 사용하였다.

■ 교배 연산자 (Crossover Operator)

본 알고리즘에서는 1-포인트 교배 연산(one-point crossover)을 사용하였으며, 교배 포인트(cut point)는 임의로 발생된다.

■ **대체 연산자 (Replacement Operator)**

대체 연산자는 다음과 같은 방법을 사용하였다. 먼저, t_{dom} 크기 만큼의 임의의 교체 후보 해들을 선택한다. 그리고 나서, 차례대로 이 해들과 새로 생성된 자손 사이의 도미네이션 관계를 파악한 후, 처음 도미네이트 되는 해와 자손 해를 서로 교체한다. 만약, 교체 후보 해 중에서 자손 해에 의해 도미네이트 되는 해가 하나도 없다면, 교체 후보 해들끼리 도미네이션 관계를 파악한다. 이 때, 다른 해에 의해서 가장 많이 도미네이트 되는 해를 자손과 교체한다.

■ **돌연 변이 연산자(Mutation Operator)와 변이 율(Mutation Rate)**

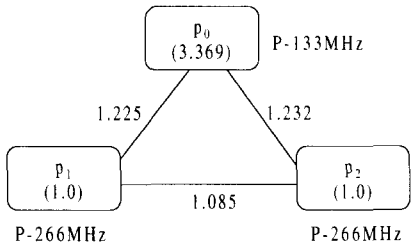
본 논문에서는 가변 돌연 변이율(variable mutation rate) 기법을 적용하였다[15]. 초기 세대에는 탐색 과정에 있어서 돌연 변이율을 높게 하고, 세대가 증가하면서 교배 연산자의 돌연 변이율을 감소시켰다. 이를 위해서 $y^2=x$ 형태의 감소 함수를 사용 하였다. 세대 수 x에 대한 현재 돌연 변이율 y는 (식 23)과 같은 함수에 의해 구해진다. 여기에서 초기 돌연 변이율, 최종 돌연 변이율, 최대 세대 수는 입력으로 주어진다. 최종 돌연 변이율은 0.022로 설정하였으며, 초기 돌연 변이율 값을 0.055로 정하 였다.

$$y = \sqrt{-\frac{(\text{초기돌연변이율}-\text{최종돌연변이율})^2 (x-\text{최대세대수})}{\text{최대 세대수}}} + \text{최종돌연 변이율} \qquad (23)$$

5. 실 험

5.1 실험에 사용된 가상 시스템

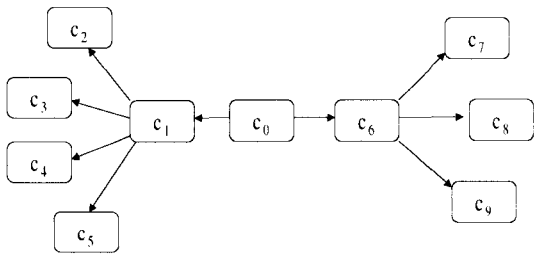
제안된 그래프 모델과 객체 할당 알고리즘의 유효성을 검증하기 위해서 전형적인 C++ 응용



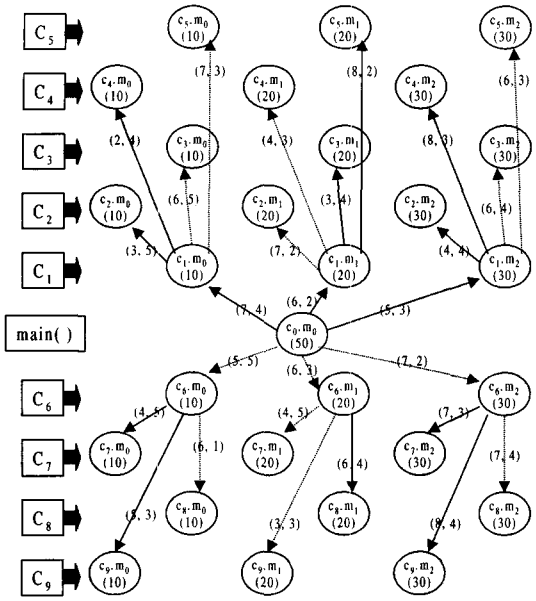
(그림 4) 실험에 사용된 시스템 그래프(SG)

프로그램에 대한 실험을 실시하였다. 본 실험은, Inprise 사의 CORBA 구현 제품인 Visibroker for C++ 3.2와 C++ Builder 3.0 개발 환경 하에서 실시하였다. 객체 할당에 사용된 분산 환경은 프로세서 세 개로 이루 어진 LAN(Local Area Network) 환경이다. 이에 대한 시스템 그래프 SG는 그림 4와 같다.

일반적인 통신 유형을 보여주는 가상의 관리자 객체 기반 시스템을 실험을 위한 소프트웨어 모델로 사용하였다. 이 시스템은 10개의 클래스로 이루어져 있으며 각 클래스는 3개의 메소드를 가지는 것으로 가정하였다. 이 시스템에 대한 전체적인 구조는 그림 5와 같다. 여기에서 c_1과 c_6는 각각의 객체로부터 여러 가지 정보를 추출하여 현재 시스템의 상태를 모니터링 함으로써, 시스템의 성능을 현재 상태에 기 반해서 제어할 수 있는 장점을 가진다. 이 시스 템의 MCG는 그림 6과 같다. 여기에서 메인 함수인 c_0은, 두 개의 관리자 객체 c_1과 c_6를 각각 동기적 정적 호출 과 비동기적 정적 호출을 통해서 호출한다. 각 관리자 객체는 다른 객체의 메소드들과 동기적 혹은 비동 기적 호출을 통해서 통신한다.



(그림 5) 관리자 객체 기반 시스템의 전체적인 구조



(그림 6) 관리자 객체 기반 시스템의 MCG

5.2 실험 결과 및 분석

5.1절에서 정의한 관리자 객체 기반 시스템에 대한 객체 할당 알고리즘 적용 결과의 3차원 그래프가 부록의 그림 7, 8과 같다. 그림 7은 초기 해를 나 타내며 그림 8은 세대 수가 50000일 때의 해집합을 나타낸다. 초기의 해집합보다 세대수가 50000일 때 의 해집합에서 적합도 함수 f_1, f_2, f_3의 값을 최소화 하는 해들이 많음을 알 수 있다.

본 논문의 객체 할당 알고리즘이 생성한 할당 해의 품질을 평가하기 위하여, 먼저 다음과 같이 비교 대상이 되는 할당 방법을 정의하였다.

- RA: 임의 할당 방법(모든 객체를 여러 개의 프로세서에 임의로 할당)
- DIA: DIAMOND[7] 시스템의 객체 클러스터링 기법에 의한 할당 방법
- GADOA: 본 논문에서 제안한 객체 할당 알고리즘에 의한 할당 방법각각의 할당 해에 따라 CORBA 환경에 구현된 분 산 시스템들의 성능을 평가하기 위해서는, 이를 위한 메트릭이 존재해야 한다. 본 절에서는 다음과 같은 두 가지 성능 평가 메트릭을 정의하여, CORBA 환경에서의 분산 시스템 성능을 비교 · 분석한다.

- EXETIME: 응답 시간(response time)
  이 값은, 프로그램의 실행 시간을 의미하며, 본 실험에서는 메인 함수($c_0.m_0$)의 종료 시간을 측정 한다. 이 값이 작을수록 우수한 성능을 나타낸다.
- BLKTIME: 객체 당 평균 블로킹 시간
  이 값은, 프로그램 수행 중 각 객체들이 평균적 으로 어느 시간 만큼 블록(blocked) 되었는지를 나타 내며, (식 24)에 의해 구해진다. 객체가 블록 되는 경우는, 동기적 호출로 인하여 다른 메소드의 실행 결과를 기다려야 할 때 발생한다. 객체 당, 이러한 블로킹 시간이 작을수록 그 객체가 효율적으로 동작하였음을 의미하므로, 이 값이 작을수록 우수한 성능을 나타낸다.

$$BLKTIME = \frac{\sum_{i}^{\text{객체갯수}} c_i\text{가블럭되었던시간}}{\text{객체의갯수}} \quad (24)$$

본 논문의 객체 할당 알고리즘(GADOA)이 생성한 해집합 중에서 DIA 할당 해를 도미네이트(어떤 할당해 S_1의 적합도 함수 f_1, f_2, f_3 값이 다른 할당해 S_2의 f_1, f_2, f_3 값보다 모두 작을 때, S_1은 S_2를 '도미네이트한다'고 함)하는 모든 할당 해와 그에 대한 EXET과 BLKT 측정 결과는 표 1과 같다.

여기에서, RA 보다는 DIA의 해가, DIA의 해보다는 GADOA의 해 중 5개의 해가 EXETIME과 BLKTIME의 값이 작음을 알 수 있으며, 이는 더욱 높은 성능의 분산 시스템을 탐색해 냈음을 의미한다. 결론적으로, 본 논문에서 제안한 객체 할

(표 1) 할당해 및 실험 결과

할당 방법	객체 할당 해	EXE TIME	BLK TIME
	C_i가 할당된 프로세서 P_i	(sec)	(sec)
RA	0 0 1 1 2 0 1 2 2 0	79.711	15.807
DIA	0 0 2 1 0 0 0 0 0 0	40.398	7.964
GADOA(1)	0 2 2 2 2 0 1 1 1 1	30.497	4.488
GADOA(2)	0 0 0 2 0 0 1 1 1 1	32.763	5.95
GADOA(3)	0 2 2 2 0 2 1 1 1 1	28.342	4.154
GADOA(4)	2 0 0 2 0 0 1 1 1 1	22.465	3.679
GADOA(5)	2 2 2 2 0 2 1 1 1 1	19.004	3.481

당 알고 리즘이 임의 할당 방법이나 DIAMOND 시스템에서 제안한 할당 기법 보다 더 좋은 품질의 해를 탐색해 냄을 알 수 있다. 또한, 우리는 본 논문에서 제안한 그래프 모델이 실제 C++ 프로그램의 중요한 성질 들을 효율적으로 반영함을 알 수 있다.

6. 결론 및 향후 연구

본 논문에서는, 객체 할당을 지원하기 위하여 기존의 C++ 응용 프로그램의 여러 가지 객체 지향적 특징들을 정형적으로 모델링한 그래프 모델을 정의 하였다. 또한, 이 그래프 모델을 바탕으로 객체들을 분산 환경에 효율적으로 할당하기 위한 객체 할당 알고리즘을 제안하였다. 그리고, 본 논문의 그래프 모델과 객체 할당 알고리즘의 유효성을 검증하기 위하여, 분산 시스템을 대표하는 기본적인 아키텍처 중에 하나인 관리자 객체 기반 시스템을 정의하고 이에 대한 모델을 생성한 후, 그 모델에 대한 객체 할당 해를 CORBA 상에서 구현하여 그 성능을 평가 하였다. 실험 결과, 본 논문의 할당 기법이 임의 할당 기법이나 DIAMOND 시스템의 객체 클러스터링 기법 보다 더 좋은 해를 탐색해 냄을 입증할 수 있었다.

본 논문의 객체 할당 기법은 여러 가지 면에서 기존의 객체 할당 기법과 다르다. 첫째, 본 논문에서 정의한 그래프 모델은 기존의 기법과 달리 클래스 계층 구조와 동적 바인딩과 같은, 객체 지향 패러 다임이 가지는 고유의 여러 가지 특성을 고려했다는 점이다. 두 번째, 본 논문의 객체 할당 알고리즘은 분산 시스템의 성능에 영향을 미치는 세 가지 요인 - 통신 비용의 최소화, 병렬성의 최대화, 부하 균등성 의 최대화 - 들을 하나로 통합하지 않고 독립적으로 고려함으로써, 세 가지 기준을 동시에 만족시키는 해를 보다 다양하게 찾아낸다는 점이다. 세 번째, NPGA 고유의 특성으로 인해 얻어지는 이점으로서, 본 논문의 객체 할당 알고리즘은 최적해를 단지 하나만 생성하는 것이 아니라, 여러 개의 해로 이루 어진 해집합을 생성한다는 점이다. 본 논문의 객체 할당 기법이 추구하는 세 가지 기준은 서로 경쟁하는 관계이므로, 세 가지 기준에서 모두 최적인 해는 존 재하지 않는다. 사용자는 이러한 해집합 중에서 자신의 분산 컴퓨팅 환경의 물리적 성질에 적합한 해를 선택할 수 있다. 예를 들어, 네트워크 회선이 아주 느린 분산 환경에 객체를 할당하려고 한다면, 생성된 할당 해 중에서 통신 비용을 가장 최소화하는 할당 해를 선택해야 할 것이다.

향후 연구 과제로는, 기존의 객체 지향 시스템에서 객체 사이의 메시지 전송 비용 뿐 만 아니라 데이터 접근 비용을 고려한 모델에 대한 연구가 있어야 하겠다. 그리고, 신뢰성이 강조되는 분산 시스템이나 코드 이주(code migration)가 지원되는 분산 환경에서 의 객체 할당 기법에 대한 연구도 진행되어야 할 것이다.

참 고 문 헌

[1] Object Management Group, "The Common Object Request Broker: Architecture and Specification" (http://www.omg.org).

[2] K. Saleh, R. Probert and H Khanafer, "The distributed object computing paradigm: concepts and applications", *Journal of Systems and Soft-*

ware, Vol. 47, Issues 2-3, pp. 125~1311, 1999.

[3] N. Weiderman, L. Northrop, D. Smith, S. Tilley and K. Wallnau, "Implications of Distributed Object Technology for Reengineering", *Technical Report CMU/SEI-97-TR-005*, 1997.

[4] Y. Gourhant, S. Louboutin, V. Cahill, A. Condon, G. Starovic and B. Tangney, "Dynamic Clustering in an Object-Orineted Distributed System", *OOPSLA Workshop on Objects in Large Distributed Systems (OLDS-2)*, 1992.

[5] P. Dickman, "Effective Load Balancing in a Distributed Object-Support Operating System", *International Workshop on Object Orientation in Operating Systems*, 1991.

[6] W. Chang, C. Tseng, "Clustering approach to grouping objects in message-passing systems", *Journal of Object-Oriented Programming*, vol 8, No. 6, 1995.

[7] U. Bellur, "A Methodology for Statically Clustering Active Objects in Distributed System," Doctoral Dissertation, Syracuse University, 1994

[8] L. R. Welch, B. Ravindran, J. Henriques and D. K. Hammer, "Metrics and Techniques for Automatic Partitioning and Assignment of Object-based Concurrent Programs", *In Proceedings of Seventh IEEE Symposium on Parallel and Distributed Processing*, 1995.

[9] J. Horn and N. Nafpliotis, "Multiobjective Optimization Using The Niched Pareto Genetic Algorithm", *Technical Report No.93005, University of Illinois at Urbana-Champaign*, USA, Jul. 1993.

[10] C.M. Fonseca and P.J. Fleming, "Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization", In Proceedings of 5th International Conference On Genetic Algorithms, 1993.

[11] G. Agha, C. Houck and R. Panwar, "Distributed Execution of Actor Programs", *Fourth International Workshop on Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua (Eds.), LNCS 589, pp. 1~17, Springer-Verlag, 1992.

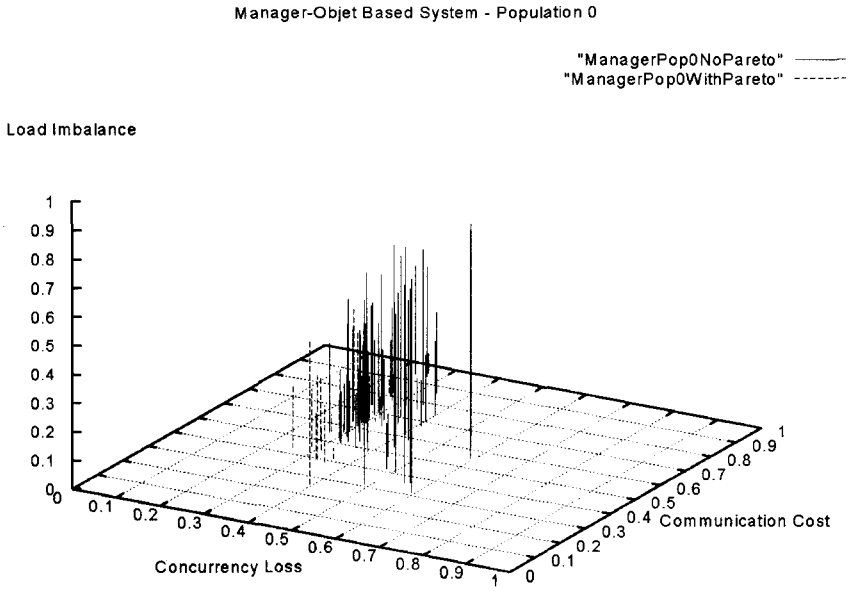
[12] V.K.A. Chien and J. Plevyak, "The Concert System - Compiler and Runtime Support for Efficient Fine-Grained Concurrent Object-Oriented Programs", *Technical Report*, Dept. of Computer Science, UIUC, 1993.

[13] A.P. Black and M.P. Immer, "Encapsulating Plurality", *Technical Report CRL-92/12, Digital Equipment Corp.*, Cambridge Research Laboratory, December 1992.

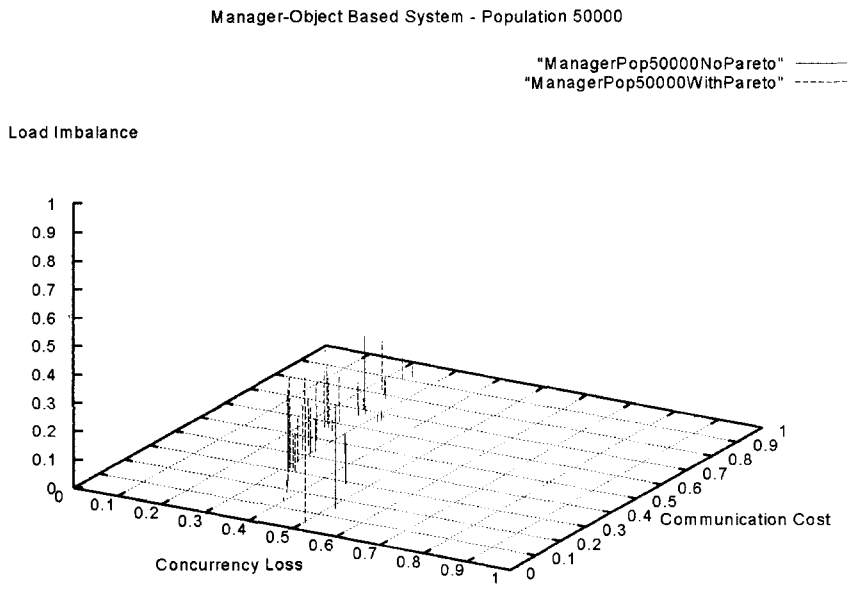
[14] C. Horn and S. Krakowiak, "Object Oriented Architecture for Distributed Office Systems," *In Proceedings of Esprit Conference*, 1987.

[15] P.C.H. Chu, "A Genetic Algorithm Approach for Combinatorial Optimisation Problems," Ph. D Thesis, 1997.

부 록

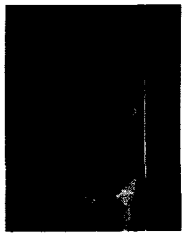


(그림 7) 관리자 객체 기반 시스템의 초기해



(그림 8) 관리자 객체 기반 시스템의 해집합 (세대 수=50000)

◐ 저 자 소 개 ◑



최 승 훈

1990년 서울대학교 계산통계학과 학사
1994년 서울대학교 계산통계학과 석사
1999년 서울대학교 계산통계학과 박사
1999~2000년 삼성전자 중앙연구소
2000년~현재 덕성여자대학교 컴퓨터과학부 전임강사
관심분야 : 컴포넌트 기반 소프트웨어 공학 환경, 분산 객체 시스템, 객체 지향 개발 방법론