다중스레드 모델의 성능 향상을 위한 가용 레지스터 기반 캐싱 기법

고 훈 준[†] · 권 영 필[††] · 유 원 희[†††]

요 약

다중스레드 모델(multithreaded model)은 폰 노이만 모델의 실행에 대한 지역성과 데이터플로우 모델의 비동기적 자료 가용성(data availability), 묵시적 병렬성(implicit parallelism)을 결합한 혼합 구조이다. 최근 다중스레드 모델의 성능 향상을 위해 많이 연구되는 분야중의 하나는 이미 폰 노이만 모델에서 효율성이 인정된 캐쉬에 관한 연구이다. 명령어 캐쉬나 오퍼랜드 캐쉬를 이용하기 위해서는 다중스레드 모델에 캐쉬 메모리를 부가시켜야 한다. 그러나 캐쉬 메모리를 부가하면 다중스레드 모델을 구현하기 위해서 많은 비용이 소모되는 단점이 발생한다. 이러한 문제점들을 해결하기 위해 캐쉬 메모리를 부가시키지 않고 다중스레드 모델의 가용 레지스터를 이용하여 캐싱을 수행하는 기법을 다중스레드 모델에 적용하였다. 가용 레지스터 기반 캐싱 기법은 스레드 수행 중 사용하지 않는 레지스터를 이용하여 캐쉬와 동일한 효과를 이루고자 하는 기법이다. 다중스레드 모델은 레지스터 최적화 과정동안 레지스터의 수를 계산할 수 있으므로 이 기법을 쉽게 적용할 수 있다. 그리고 가용 레지스터 기반 캐싱 기법을 다중스레드 모델에 적용함으로서 프레임 메모리에서 발생하는 접근 충돌과 병목 현상을 제거할 수 있다. 본 논문에서 제안한 레지스터 기반 캐싱 기법을 여러 프로그램에 적용한 결과 다중스레드 모델의 전체적인 성능 향상이 나타났다. 또한 캐쉬 메모리 기반 캐싱 기법과 비교한 결과 비슷한 실행 시간이 나타났다.

A Register-Based Caching Technique for the Advanced Performance of Multithreaded Models

Hoon Joon Kouh[†] · Young Feel Kwon[††] · Weon Hee Yoo[†††]

ABSTRACT

A multithreaded model is a hybrid one which combines locality of execution of the von Neumann model with asynchronous data availability and implicit parallelism of the dataflow model. Much researches that have been made toward the advanced performance of multithreaded models are about the cache memory which have been proved to be efficient in the von Neumann model. To use an instruction cache or operand cache, the multithreaded models must have cache memories. If cache memories are added to the multithreaded model, they may have the disadvantage of high implementation cost in the model. To solve these problems, we did not add cache memory but applied the method of executing the caching by using available registers of the multithreaded models. The available register-based caching method is one that use the registers which are not used on the execution of threads. It may accomplish the same effect as the cache memory. The multithreaded models can compute the number of available registers to be used during the process of the register optimization, and therefore this method can be easily applied on the models. By applying this method, we can also remove the access conflict and the bottleneck of frame memories. When we applied the proposed available register-based caching method, we found that there was an improved performance of the multithreaded model. Also, when the available register-based caching method is compared with the cache based caching method, we found that there was the almost same execution overhead.

키워드 : 다중스레드 모델(multithreaded model), 레지스터 최적화(register optimization), 캐싱(caching)

1. 서 론

다중스레드 모델(multithreaded model)은 폰 노이만 모델의 장점인 실행에 대한 지역성과 데이터플로우 모델의 장점인 비동기적 자료 가용성(data availability), 묵시적 병렬성(implicit parallelism)을 결합시킨 혼합형 계산 모델이다 [2, 3, 7]. 이 모델은 스레드라는 수행단위를 기반으로 동작한다. 또한 전역 메모리 참조와 같은 긴 지연시간을 요구하는 연산 분할은 프로세스의 유휴시간을 초래하지 않고서 계산들간에 빠른 문맥 교환을 가능하게 하여 병렬처리 시스템의 수행성능을 향상시킬 수 있다[4, 9].

※ 본 논문은 2000년도 정보통신부 대학기초 연구지원 사업에 의해 수행되었음.
† 정 회 원 : 인하대학교 대학원 전자계산공학과
†† 정 회 원 : (주)다이알로직 코리아
††† 종신회원 : 인하대학교 전자계산공학과 교수
논문접수 : 2000년 12월 21일, 심사완료 : 2001년 6월 8일

스레드 코드는 컴파일러에 의해서 실행순서가 결정된 명령어 그룹이다. 스레드 코드의 품질에 따라 전체적인 다중스레드 모델을 기반으로 한 병렬 시스템의 수행 성능이 결정되기 때문에, 스레드 코드를 생성하는 컴파일러의 역할이 중요한 위치를 점하고 있다[8]. 또한 서로 논리적으로 연관된 스레드가 모여 하나의 코드 블록을 이룬다. 다중스레드 모델에서 코드 블록을 수행하기 위해서는 코드 블록을 스레드 메모리에 적재하고, 코드 블록을 수행하기 위해 필요한 프레임 정보를 프레임 메모리에 적재해야 한다. 다중스레드 모델에서 스레드는 적재된 프레임 정보를 기반으로 실행한다. 각 스레드의 수행 방식은 스레드를 수행하기 위해 필요한 동기화 인자들의 점화 규칙(firing rule)에 의해 수행되고, 수행이 끝난 스레드는 새로운 동기화 인자를 생성하여 다른 스레드의 수행을 유도한다.

다중스레드 모델에서 각 스레드는 지역성을 가지며, 스레드의 수행은 동기화 인자에 의해 결정되는 특징을 이용하여 다중스레드 모델의 수행 성능을 향상시키려는 연구가 시도되었다. 스레드의 특징을 이용하여 다중스레드 모델의 수행 성능을 향상시키려는 노력으로 폰 노이만 모델에서 이미 효율성이 입증된 캐쉬(cache)와 선반입(prefetching)을 이용한 방법이 제안되었다.

이러한 연구 중 하나로 Kavi는 데이터플로우 기반 다중스레드 모델에 대하여 데이터 캐쉬와 명령어 캐쉬, I-구조 캐쉬를 설계하였다[5]. 또 다른 연구로는 스레드를 수행하는데 필요한 동기화 인자를 반입(fetch)하기 위해 프레임을 접근하는 동안 프레임 메모리를 접근하는 통로에 병목현상이 발생하기 때문에 동기화 캐쉬를 설계하여 동기화 인자를 선반입하는 기법이 연구되었다[11]. 그러나 다중스레드 모델의 수행 성능 향상을 위해서 캐쉬 메모리를 다중스레드 모델에 부가시키는 방법은 다중스레드 모델의 실행 모델 구현 비용이 증가하는 결과를 가져온다. 현재 캐쉬 메모리의 가격이 하락했을지라도 실행 모델에 부가시키기에는 부담을 주며, 캐쉬 메모리를 부가하기 위한 실행 모델의 설계 상 변경이 불가피하게 되어 불필요한 오버헤드가 발생하고 구현 비용이 증가한다.

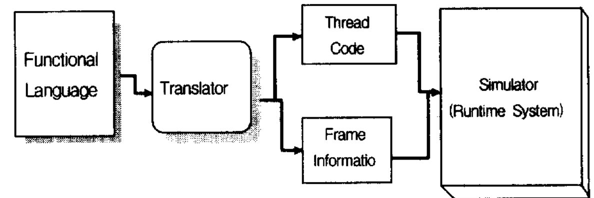
다중스레드 모델의 실행 모델 구현 비용이 증가되지 않으면서 기존 캐쉬 기법을 이용하기 위해서 본 논문에서는 가용 레지스터 기반 캐쉬 기법(Available Register-Based Caching)을 제안한다. 다중스레드 모델은 스레드를 수행시킬 때 스레드 프로세서의 레지스터를 기반으로 연산을 수행한다. 스레드 수행에 필요한 계산 레지스터를 제외한 나머지 레지스터는 사용하지 않는 상태(idle state)에 있게 된다. 따라서 이 기법은 스레드가 수행될 때 사용하지 않는 가용 레지스터를 이용하여 캐쉬와 동일한 효과를 가진다.

본 논문은 다음과 같이 5개의 장으로 구성된다. 2장에서는 다중스레드 모델의 설계 요소와 구성에 따라 기존 다중스레드 모델을 살펴보며, 3장은 기존 다중스레드 모델의 성능 향상을 위한 방법으로 가용 레지스터 기반 캐쉬 기법의 배경과 세부적인 방법을 제안하고, 4장은 제안한 가용 레지스터 기반 캐쉬 기법의 성능 실험 및 평가를 한다. 그리고 5장에서는 본 논문의 결론을 제시하고 향후 연구 과제를 기술한다.

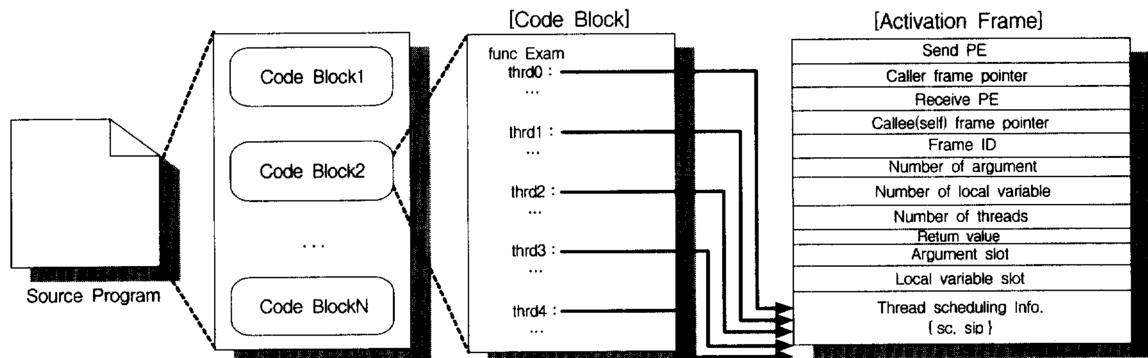
2. 다중스레드 모델

2.1 실행 모델

일반적인 다중스레드 모델은 (그림 2-1)과 같이 함수 언어를 원시언어로 사용한다. 스레드로 구성된 코드 블록을 수행하기 위해 번역기는 스레드 코드(thread code)와 프레임 정보(frame information)라는 부가적인 정보를 생성한다[10].



(그림 2-1) 다중스레드 모델의 수행 과정



(그림 2-2) 코드 블록과 활성 프레임 구조

다중스레드 모델에서 사용하는 원시 프로그램은 (그림 2-2)와 같이 프로그램내의 함수들이 각각의 코드 블록으로 표현되고, 각 코드 블록은 여러 개의 스레드로 구성된다. 하나의 코드 블록 내에 있는 스레드들은 실행을 위해 공통된 저장 공간인 프레임 메모리(frame memory)를 사용한다. 프레임 메모리에는 스레드를 실행하기 위해 코드 블록 생성시 만들어진 프레임 정보로부터 도출된 스레드의 참조 변수나 동기화 계수, 스레드 시작 주소 등의 정보가 저장된다.

프레임 정보는 프레임 메모리에 적재되어 활성 프레임(activation frame)을 생성하며, 활성 프레임은 프레임 메모리에 리스트 형태로 구성된다. 그리고 논리적으로 연관된 스레드로 구성된 코드 블록은 여러 프로세서에서 병렬적으로 수행되고 각 스레드는 데이터플로우의 점화 규칙에 의해 동적으로 스케쥴링 된다.

다중스레드 실행 모델의 동작은 다음과 같다. 함수 호출 관계에서 각 호출자와 피호출자는 논리적으로 연관된 스레드로 구성된 코드 블록으로 표현되며, 호출자 코드 블록의 스레드는 피호출자의 실행을 요구하고, 종료된다. 호출된 피호출자의 코드 블록은 실행을 위해 스레드 스케줄러에 의해 활성 프레임을 프레임 메모리에 적재한다. 비평가인자 함수언어(nonstrict functional language)의 특성에 의해 호출자의 각 인자는 비동기 메시지의 형태로 피호출자의 프로세서로 보내지고 피호출자의 스레드는 필요한 값이 결정될 때마다 점화 규칙에 의해 동기화 처리가 이루어진다. 피호출자 코드 블록의 실행이 완료되면, 호출자는 피호출자로부터 반환되는 값을 비동기 메시지 형태로 전달받게 된다. 전달받은 값에 의해 호출자의 다른 스레드가 활성화되고, 이와 같은 방식으로 스레드가 계속적으로 실행되어 호출자의 실행을 완료한다.

2.2 기존 다중스레드 모델의 성능 향상 기법

다중스레드 모델의 성능에 영향을 미치는 요소에는 스레드의 실행 방식과 동기화 구조, 그리고 메모리 모델이 있으며 이는 효율적인 다중스레드 모델을 구성하는 중요한 요소가 된다.

다중스레드 모델의 성능 향상에 관한 연구는 <표 2-1>과 같이 여러 가지 방향으로 시도되고 있다. 대부분의 다중스레드 모델은 폰 노이만 모델에서 그 실효성이 입증된 캐쉬를 이용하여 성능 향상을 이루려하고 있다. 이는 폰 노이만 모델에서 실효성이 입증된 성능 향상 기법을 손쉽게 다중스레드 모델에 적용할 수 있는 이점이 있다. Kavi는 ETS(Explicit Token Store) 방식을 갖는 데이터플로우 실행 방식을 갖는 다중스레드 모델에 대하여 명령어 캐쉬, 데이터 캐쉬, I-구조 캐쉬를 설계하여 성능 향상을 입증하였다.

〈표 2-1〉 기존 다중스레드 모델의 성능 향상 기법

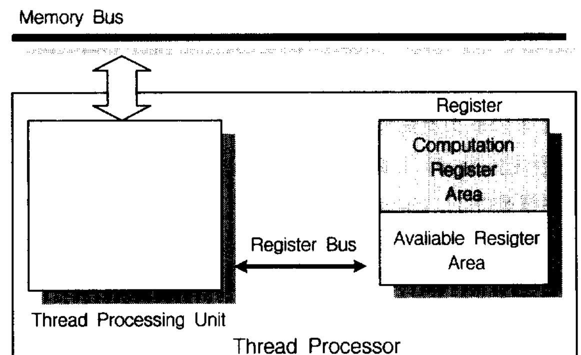
모델	스레드 실행 방식	동기화 방식	스케쥴링 방식	성능 향상 방식
Monsoon	비중단형	묵시적	선입선출	파이프라인
TAM	비중단형	명시적	퀀텀 단위	I-구조 캐쉬
P-RISC	비중단형	명시적	선입선출	없음
Tera MTA	비중단형	묵시적	선입선출	하드웨어적 스레드 스위칭
*T	비중단형	명시적	선입선출	캐쉬
DAVRID	비중단형	묵시적	선입선출	캐쉬

그러나 다중스레드 모델에 폰 노이만 모델에서 실효성이 입증된 캐쉬를 사용하기 위해서는 실행 모델의 설계가 변경되어야 하고, 캐쉬 비용, 캐쉬 실패와 같은 캐쉬와 관련된 여러 가지 문제점이 발생될 수 있다. 이러한 문제점을 개선하기 위해 여러 가지 방법이 제안되었다. 대표적인 캐쉬 보완 기법으로는 캐쉬 데이터를 미리 캐쉬로 선반입하여 캐쉬 실패율을 줄이는 선반입 기법이 제안되었다[12]. 선반입 기법은 선반입하는 방식에 따라 순차적인 선반입과 수행 흔적에 의한 예측 선반입으로 나눌 수 있다. 이러한 선반입 기법을 도입함으로써 캐쉬 실패율은 개선될 수 있지만 다중스레드 실행 모델의 설계 변경과 캐쉬를 부가하기 위한 비용은 여전히 해결되지 않는 문제로 남는다.

3. 가용 레지스터 기반 캐쉉 기법

3.1 가용 레지스터 기반 캐쉉

본 논문에서는 다중스레드 모델의 실행 모델 구현 비용이 증가되지 않으면서 캐쉬와 동일한 효과를 얻기 위한 방법으로 가용 레지스터 기반 캐쉉 기법을 제안한다. 다중스레드 모델은 스레드를 수행시킬 때 스레드 프로세서의 레지스터를 기반으로 연산을 수행한다. (그림 3-1)은 스레드 프로세서의 구조를 나타낸다. 스레드 프로세서는 스레드 수행에 필요한 계산 레지스터(computation register)를 제외한 나머지 레지스터는 사용하지 않는 상태에 있게 된다. 본 논문에서 제안한 가용 레지스터 기반 캐쉉 기법은 스레드가



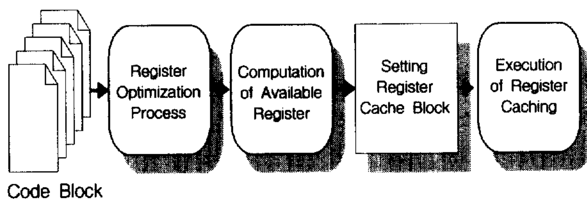
(그림 3-1) 스레드 프로세서의 구조

수행될 때 사용되지 않는 가용 레지스터를 이용하여 레지스터 캐쉬를 설계함으로서 다중스레드 모델의 성능을 향상시키고자 한다.

이 기법은 프레임 메모리에 저장되어 있는 동기화 인자의 잦은 접근으로 발생하는 프레임 메모리의 접근 충돌과 병목 현상을 제거하는 방법으로 설정된 레지스터 캐쉬에 스레드 수행에 영향을 미치는 동기화 인자를 선반입하여 사용하는 동기화 인자 캐싱을 적용한다.

동기화는 실행될 코드 블록의 정보를 가지고 있는 프레임 메모리에 접근하고 동기화 인자를 레지스터 캐쉬에 선반입하고 접근함으로써 이루어진다. 프레임 메모리에는 동기화를 위해 실행될 코드 블록의 각 스레드에 해당하는 동기화 정보를 가지고 있다. 동기화 정보는 각 스레드의 동기화 계수기와 스레드의 시작주소의 쌍으로 이루어져 있다. 값이 결정될 때마다 결정된 값과 관련 있는 스레드의 동기화 계수기를 사용하여 동기화 계수를 감소시키게 된다. 동기화 계수가 0이 되면 해당 스레드를 실행시키기 위해 프레임 포인터 FP와 스레드 포인터 IP를 스레드 처리기로 전송한다.

가용 레지스터 기반 캐싱 기법은 (그림 3-2) 과정으로 수행한다. 수행 단계는 먼저 코드 블록에서 연산의 수행을 위해서 사용되는 레지스터 최적화 과정을 수행한다. 레지스터 최적화를 수행하면 스레드 수행을 위해 필요한 계산 레지스터의 수를 알 수 있으므로 코드 블록이 수행될 때 사용하지 않는 가용 레지스터를 구할 수 있다. 가용 레지스터의 수가 계산되면 계산된 가용 레지스터를 이용하여 레지스터 캐쉬 블록을 설정한다. 그리고 레지스터 캐쉬에 코드 블록 수행 시 스레드의 수행을 결정하는 동기화 인자들을 선반입하여 프레임 메모리 접근 충돌과 병목 현상을 제거한다.



(그림 3-2) 가용 레지스터 기반 캐싱 기법의 수행 단계

레지스터 최적화 과정은 각 스레드에 대해서 명령어 수준 레지스터 최적화를 수행하며, 스레드의 레지스터 최적화 방법은 Chaitin의 그래프 컬러링을 변형시킨 기법을 이용하여 수행한다[1]. Chaitin의 레지스터 할당 알고리즘은 레지스터 할당을 위해 단순화 단계에서 가용 레지스터의 수보다 큰 차수를 갖는 노드를 대피코드로 생성하여 코드 스케줄링을 재 수행한다. Chaitin의 레지스터 할당 알고리즘은 대피코드에 의해 발생되는 코드 스케줄링 재 수행 때문에 비용이 많이 드는 단점이 있다. 그러나 다중스레드 모델에서 코드 스케줄링은 데이터플로우 그래프를 분할할 때 수행되므로 레지스터 할당 시 코드 스케줄링은 고려하지 않아도 된다. 따라서 각 스레드에 대한 명령어 수준 레지스터 최적화 알고리즘은 Chaitin의 그래프 컬러링 알고리즘에서 대피코드를 생성시키는 단순화 단계를 수행하지 않고, 간섭 그래프를 기반으로 컬러링을 수행하여 레지스터를 할당하는 방법을 사용한다.

가용 레지스터 기반 캐싱 기법을 위해 필요한 명령어 수준 레지스터 최적화 알고리즘의 전체적인 개요를 (알고리즘 3-1)로 정형화하였다. T_i는 코드 블록에 있는 i번째 스레드이고, $TSize_i$는 스레드 T_i의 크기(명령어의 개수)이고, I_{lc}는 명령어의 생존 범위, AL_c는 컬러링을 위한 인접 리스트를 나타낸다.

Input : Code Block consisting of threads $T_1, T_2, T_3, \cdots, T_m$
Output : Code Block with Optimized Registers
1 for(i = 1 ; i <= m ; i++) do
2
3 Step 1 : /* live instruction analysis within a thread T_i */
4 I_{lc} Analysis using [Algorithm 3-2] ;
5
6 Step 2 : /* Interference graph creation of T_i*/
7 AL_c Creation based on I_{lc} using [Algorithm 3-3] ;
8
9 Step 3 : /* Coloring execution */
10 for(j = 1 ; j <= $TSize_i$; j++) do
11 Select a node j of the interference graph ;
12 Select a color that is different from colors already assigned to its neighbors ;
13 Set Color Tag of AL_c(j) to the color ;
14 end-for
15
16 Step 4 : /* Register allocation execution*/
17 Register Allocation based on AL_c using [Algorithm 3-4] ;
18
19 end-for

(알고리즘 3-1) Register Optimization

(알고리즘 3-2)는 (알고리즘 3-1)의 첫 번째 단계인 명령어의 생존 범위를 분석하는 알고리즘이다. $TSize_i$는 스레드 i의 크기, IR은 스레드 명령어의 결과를 저장하는 피연산자, I_R[i]는 IR 값을 저장하는 배열, OPS는 피연산의 집합, ILS[j1, j2]는 명령어 생존 범위를 저장하는 배열을 나타낸다.

Input : Thread T_i of Code Block
Output : Array represented the scope of live thread instructions
1 for (j1 ← 1 ; j1 ≤ $TSize_i$; j1++) do
2 for (j2 ← 1 ; j2 ≤$TSize_i$; j2++) do
3 ILS[j1, j2] ← FALSE ;
4 end-for
5 end-for
6
7 for (j1 ← 1 ; j1 ≤$TSize_i$; j1++) do

```
8        IR ← Result Operand From instruction at location j1
              of Thread Ti
9        IR[j1] ← IR ;
10       Initialize OPS by { } ;
11       for ( j2 ← j1 + 1 ; j2 ≤TSizei ; j2++ ) do
12            OPS ← Operands From instruction at j2 of
                    Thread Ti ;
13            If ( IR[j1] ∈ OPS ) then
14              lv ← j2 ;
15            end-if
16       end-for
17       for ( j2 ← j1 + 1 ; j2 ≤ lv ; j2++ ) do
18            ILS[j1, j2] ← TRUE ;
19       end-for
20    end-for
```

(알고리즘 3-2) Liveness analysis of instructions

(알고리즘 3-3)는 (알고리즘 3-1)의 두 번째 단계인 간섭 그래프를 생성하는 알고리즘이다. 간섭 그래프는 인접 리스트로 표현되며, 그래프의 노드는 그래프의 다른 노드를 나타내는 sreg와 link로 구성된 구조체로 표현된다. (알고리즘 3-3)에서 $TSize_i$는 스레드 i의 크기, ILS[j1, j2]는 명령어 생존 범위를 나타내는 배열, link는 노드의 링크, sreg는 노드의 번호를 나타내며 또한 그 노드에는 기호 레지스터가 할당된다. Allocation()은 노드를 생성하는 함수, AL_c는 간섭 그래프를 표현하는 인접 리스트를 나타낸다.

```
Input : Array represented the life scope of thread instructions
Output : The adjacency list representation of the interference
         graph
1      for ( j1 ← 1 ; j1 ≤TSizei ; j1++ ) do
2           Hnode ← Allocation() ;
3           ALc(j1) ← Hnode ;
4           Hnode.sreg ← j1 ;
5           for ( j2 ← j1 + 1 ; j2 ≤TSizei ;  j2++ ) do
6                If ( ILS[j1, j2] == TRUE ) then
7                        Cnode ← Allocation() ;
8                        Cnode.sreg ← j2 ;
9                        Hnode.link ← Cnode ;
10                       Hnode ← Cnode ;
11               end-if
12          Hnode ← Cnode ;
13          end-for
14     Hnode.link ← NULL ;
15     end-for
```

(알고리즘 3-3) Interference graph creation

(알고리즘 3-4)는 (알고리즘 3-1)의 네 번째 단계인 레지스터 할당을 수행하는 알고리즘이다. T_i는 코드 블록에 있는 i번째 스레드를 나타내고, k는 스레드 T_i의 임의의 명령어의 k번째 피연산자를 나타내는 변수, R_{color}는 레지스터의 컬러, AL_c는 컬러링을 위한 인접 리스트를 나타낸다. 또한 함수 operand(i, j, k)는 스레드 i의 j번째 명령어의 k 번째 피연산자를 반환하는 함수이며, k번째 피연산자가 없는 경우에는 0를 반환하는 함수이다.

```
Input : Thread Ti of a Code Block with TSizei
        Adjancy list ALc of interference graph of Ti
Output : Thread Ti with register assignment
1        for (j = 1 ; j <= TSizei ; j++) do
2            k ← 0 ;
3            oprd ← operand(i, j, k)
4            while(oprd ≠ 0) do
5               pos ← 1
6               while(oprd ≠ ALc(pos) -> sreg) pos ← pos + 1
                end-while
7               Rcolor ← Color of Color Tag in ALc(Pos) ;
8             Reg ← Register corresponding to Rcolor ;
9               Change operand(i, j, k) in instruction j of Ti to Reg
10              k ← k + 1
11              oprd ← operand(i, j, k)
12           end-while
13       end-for
```

(알고리즘 3-4) Register Allocation

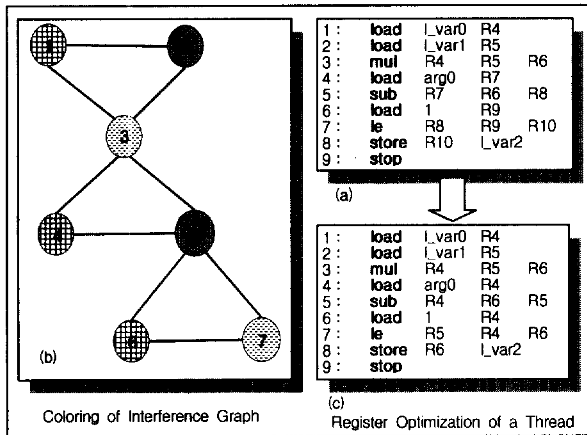
가용 레지스터 기반 캐싱 기법의 첫 번째 단계에서 수행되는 명령어 수준 레지스터 최적화 기법은 선택된 스레드를 분석하여 스레드에 있는 각 명령어의 생존 범위를 분석하고, 분석된 생존 범위를 기반으로 간섭 그래프(interference graph)를 생성한다. 생성된 간섭 그래프를 기반으로 그래프 컬러링을 적용하여 레지스터 최적화를 수행한다.

(그림 3-3)은 어느 원시 프로그램으로부터 변환된 스레드로 구성된 코드 블록이다. 이 코드 블록의 4번째 스레드인 thrd3에 레지스터 최적화과정을 적용하여 얻은 결과는 (그림 3-4)에 주어진다.

```
thrd0 :   receive   arg0
          stop
thrd1 :   load      2         R0
          store     R0        l_var0
          stop
thrd : 2  load      arg0      R1
          load      l_var0    R2
          div       R1        R2        R3
          store     R3        l_var1
          stop
thrd3 :   load      l_var0    R4
          load      l_var1    R5
          mul       R4        R5        R6
          load      arg0      R7
          sub       R7        R6        R8
          load      1         R9
          le        R8        R9        R10
          store     R10       l_var2
          stop
thrd4 :   load      l_var2    R11
          jmpt      R11       label0
          load      l_var1    R12
          jmp       label1
label0 :  load      arg0      R12
label1 :  store     R12       l_var3
          send      l_var3    pfp
          endf
```

(그림 3-3) 데이터플로우 그래프로부터 생성된 스레드 코드

(그림 3-4)의 (a)는 레지스터 최적화가 수행되기 전의 스레드 코드를 나타내고, (b)는 컬러링이 수행된 간섭 그래프를 나타낸다. 컬러링된 간섭 그래프를 나타내는 (b)에서 노드들의 번호는 (a)의 스레드 명령어의 번호이다. 그리고 (c)는 (a)의 스레드 코드에 레지스터 최적화 과정을 수행한 후의 결과이다.



(그림 3-4) 간섭 그래프의 컬러링과 명령어 수준 레지스터 최적화

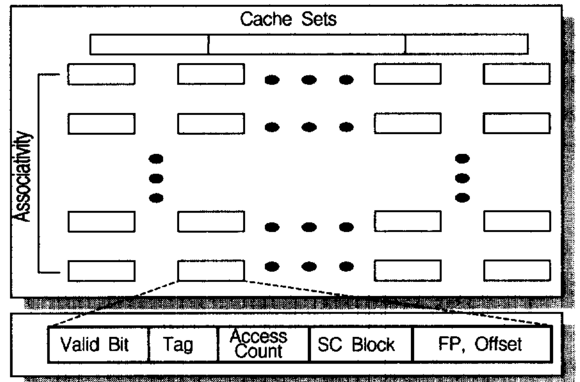
앞에 기술된 기법을 이용하여 코드 블록에 포함되어 있는 모든 스레드에 대해서 레지스터 최적화 과정을 수행하면, 각 스레드에서 사용하는 최적화된 레지스터의 개수를 알 수 있다. 따라서 레지스터 사용이 최적화된 각 스레드의 레지스터의 수를 이용하여 가용 레지스터 기반 캐싱 기법의 두 번째 단계인 가용 레지스터의 개수를 계산할 수 있다.

다중스레드 모델에서 스레드는 스레드 프로세서에서 처리되기 위해 실행 대기 큐에 삽입되어, 순차적으로 스레드 프로세스에서 수행된다. 그러므로 코드 블록이 수행되기 위해 필요한 레지스터의 수는 코드 블록의 각 스레드에서 사용되는 전체 레지스터 수가 아니라 가장 많은 레지스터를 사용하는 스레드의 레지스터 개수가 된다. 이러한 이유로 코드 블록의 가용 레지스터 계산은 코드 블록에 포함된 스레드 중에서 가장 많은 레지스터를 사용하는 스레드의 레지스터 수를 코드 블록 수행에 필요한 최소 레지스터로 설정하여 계산 레지스터를 구한다. 코드 블록 수행에 필요한 계산 레지스터가 구해지면 가용 레지스터의 개수는 전체 레지스터 수에서 계산 레지스터의 개수를 감산함으로써 계산된다. 가용 레지스터 개수를 수학적으로 표현하면 아래와 같이 표현된다.

$$R_{computation} = MAX(\ R_{thread1},\ R_{thread2},\ \cdots,\ R_{threadN}\ )$$
$$R_{avaliable} = R_{total} - R_{computation}$$

위의 식에서 $R_{computation}$는 계산 레지스터의 수이고, R_{thread}는 스레드의 레지스터 수, $R_{avaliable}$는 가용 레지스터, R_{total}은 전체 레지스터의 개수를 각각 의미한다.

앞의 과정을 이용하여 코드 블록이 실행되는데 필요한 계산 레지스터의 수를 계산하여 코드 블록이 수행될 때 사용하지 않는 가용 레지스터의 수를 구할 수 있다. 이렇게 계산된 가용 레지스터를 이용하여 가용 레지스터 기반 캐싱 기법의 세 번째 단계인 레지스터 캐쉬 설정을 수행한다. (그림 3-5)는 레지스터 캐쉬의 구조를 나타낸 레지스터 캐쉬 조직도이다.



(그림 3-5) 레지스터 캐쉬의 조직도

레지스터 캐쉬는 상호 연결된 연상 기억 장치로 구성되며, 레지스터 캐쉬의 각 블록은 Valid Bit, Tag, Access Count, SC Block, (FP, Offset)으로 구성된다. 폰 노이만 모델의 캐쉬와 동일한 의미로 사용되는 Valid Bit와 Tag는 작업 집합(working sets)의 역할을 하는 슈퍼 블록(super block)에서 캐쉬 블록이 사용되고 있는 가와 캐쉬 블록의 ID를 나타낸다. Access Count는 캐쉬의 교환을 위해 캐쉬 블록이 사용된 횟수를 나타내는데 쓰이며, SC Block(synchronization cache block)은 스레드의 점화를 위해 필요한 동기화 인자를 저장하는 공간으로 사용된다. 마지막의 (FP, Offset)은 레지스터 캐쉬에 저장된 동기화 인자의 프레임 메모리 주소를 나타낸다.

코드 블록이 수행될 때 사용되지 않는 가용 레지스터를 이용하여 (그림 3-5)와 같이 레지스터 캐쉬를 설정 한 후에 가용 레지스터 기반 캐싱 기법의 마지막 단계인 스레드의 동기화를 결정하는 동기화 인자를 레지스터 캐쉬에 선반입하는 기법을 수행한다.

다중스레드 모델의 스레드는 적재된 프레임 정보를 기반으로 실행한다. 각 스레드의 수행은 스레드를 수행하기 위해 필요한 동기화 인자들의 점화 규칙에 의해 수행되고, 수행이 끝난 스레드는 새로운 동기화 인자를 생성하여 다른 스레드의 수행을 유도한다. 이러한 다중스레드 모델의 수행 특징을 이용하여 스레드가 수행된 후에 생성된 동기화 인자를 레지스터 캐쉬에 선반입(prefetching)하여 다른 스레드의 수행을 결정하는데 영향을 미치는 동기화 인자를 프레임 메모리로부터 반입하지 않고 레지

스터 캐쉬에서 반입하도록 한다. 동기화 인자 선반입을 수행함으로서 스레드의 동기화에 의해서 발생되는 프레임 메모리의 병목 현상과 메모리 충돌이 방지되고, 동기화 인자가 레지스터 캐쉬에서 빠르게 반입되어 다중스레드 모델의 수행 성능이 향상된다.

3.2 가용 레지스터 기반 캐쉬의 교환 전략

본 논문에서 제안한 가용 레지스터 기반 캐쉬의 캐쉬 교환 전략을 결정하기 위해서 폰 노이만 모델의 캐쉬에서 사용되고 있는 여러 캐쉬 교환 기법을 살펴보았다. 보편적으로 다중스레드의 명령어 캐쉬나 오퍼랜드 캐쉬는 교환 전략으로 가장 오랫동안 사용하지 않은 캐쉬 블록을 교환 블록으로 선정하는 LRU(Least Recently Used) 또는 LRU의 변형 방법을 많이 사용한다.

다중스레드 모델에서 스레드의 수행을 결정하는 동기화 인자는 하나 이상의 스레드 수행을 위해 필요한 동기화 인자로 사용되지만, 다중스레드 모델의 분할 과정에서 스레드를 가급적으로 많은 연산을 수행하는 스레드로 형성시켜 동기화 인자의 종속관계를 하나의 스레드로 모으려고 한다. 이러한 스레드 형성 특징으로 인하여 하나의 동기화 인자는 한 스레드의 동기화에 영향을 미친다. 그러므로 이미 많이 사용된 동기화 인자는 앞으로 스레드의 동기화에 영향을 미칠 확률이 적어지므로 캐쉬 블록의 Access Count의 수가 높은 캐쉬 블록을 레지스터 캐쉬의 교환 블록으로 선정하는 것이 적당하다.

위와 같은 다중스레드 모델의 특징으로 인해 기존에 캐쉬 교환 전략으로 많이 사용하는 LRU 교환 전략은 본 논문에서 제안한 레지스터 캐쉬를 위한 교환 전략으로 적당하지 않다. 따라서 여러 캐쉬 교환 기법 중에서 이전에 가장 많이 사용된 캐쉬 블록을 교환 블록으로 선택하는 Used Words Policy를 레지스터 캐쉬를 위한 교환 전략으로 사용한다[5]. Used Words Policy는 이전에 가장 많이 사용된 캐쉬 블록을 교환 블록으로 선택하기 위해 레지스터 캐쉬의 Access Count를 참조하여 접근 수가 높은 캐쉬 블록을 교환 캐쉬 블록으로 선정하여 캐쉬 블록의 내용을 교환한다. 이와 같은 교환 전략은 프레임 메모리에서 자주 발생할 수 있는 동기화 인자들의 접근 충돌과 병목 현상을 막아주고 동기화 인자를 기반으로 동작하는 다중스레드 모델에서는 스레드의 독립적인 수행을 가능하게 한다. 가용 레지스터의 수가 적은 경우에도 하나의 코드블록에 존재하는 동기화 인자의 수가 적기 때문에 모두 레지스터로 캐쉬이 가능하다. 따라서 동기화 인자의 접근을 위해 레지스터와 프레임 메모리에 접근하는 일은 거의 발생되지 않는다. 그러나 캐쉬을 하는 선반입의 횟수는 증가할 수 있으며 이는 기존의 캐쉬를 사용하는 경우와 동일하다. 또한 Used Words Policy 기반으로 현재 활성 프레임의 동기화 인자만을 캐쉬하므로 부적절한 캐쉬 교환은 거의 발생하지 않는다. 따라서 본 연구에서는 레지스터와 프레임 메모리에 접근하는 2중 동기화는 발생되지 않는다고 가정한다.

4. 실 험

4.1 실험 시 고려사항

본 논문에서는 함수 언어로부터 다중스레드 코드로 변환하는 번역기와 번역된 다중스레드 코드가 실행되는 시뮬레이터를 이용하여 시뮬레이션을 수행하였다. 사용된 시뮬레이터와 번역기는 유닉스 환경에서 구현되었으며, 이 실험을 위하여 기존에 구현된 시뮬레이터에 캐쉬 시뮬레이터를 추가하였다. 시뮬레이터의 수행 환경은 하나의 프로세스에서 다중 프레임을 수행시키는 다중 프레임 단일 프로세서(multi-frame single-processor)를 기반으로 수행되고, 코드 블록 수행을 위해 필요한 레지스터는 16바이트의 크기를 갖는 1024개의 레지스터로 구성된다고 가정하였다. 하나의 레지스터는 실험을 위해서 Valid Bit 1비트, Tag 15비트, Access Count 2바이트, SC Block 8바이트, 그리고 FP와 Offset은 각각 2바이트로 구성하였다. 또한 프로세서 간 통신비용과 구조 메모리의 접근 비용은 코드 블록이 단일 프로세서에서 수행되므로 고려하지 않는다.

<표 4-1>은 실험을 위해 사용된 예제 프로그램의 특성을 기술한 것이다. 각 예제 프로그램은 코드 블록의 수, 코드 블록 크기, 스레드의 수, 명령어 참조 횟수, 동기화 인자 참조 횟수, 사용 레지스터 수로 표현된다. 프로그램 mmp1000은 1000개의 원소를 갖는 일차원 배열 2개를 곱하는 프로그램이고, 프로그램 fact1000은 1000까지의 계승값(factorial)을 구하는 프로그램이다. 프로그램 fibo1000은 1부터 1000까지의 피보나치 수를 구하는 일을 수행하고, l15는 Lawrence Livermore Loop 중 Loop5이다.

〈표 4-1〉 실험에서 사용하는 예제 프로그램의 특성

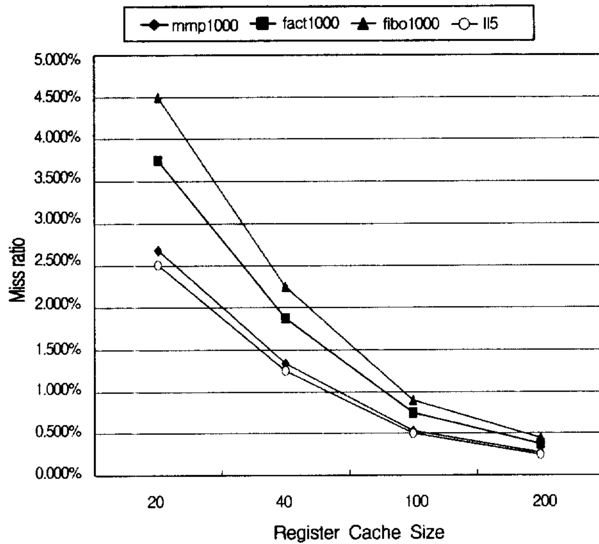
	mmp1000	fact1000	fibo1000	l15
코드 블록의 수	3	1	2	4
코드 블록 크기	120	29	50	166
스레드의 수	23	5	11	32
명령어 참조 횟수	85029	32000	34012	507830
동기화 인자 참조 횟수	28009	8000	10004	182609
사용 레지스터 수	38	11	11	53

4.2 실험 결과

본 논문의 실험을 위해 캐쉬 시뮬레이터와 캐쉬를 사용하지 않는 시뮬레이터를 이용하여 실험을 수행하였다. 실험은 캐쉬 크기에 따른 실패율, Superblock Set 크기에 따른 실패율, 기존의 머신과 레지스터 캐쉬 기법을 사용한 머신의 수행 시간 비교, 캐쉬 메모리를 사용한 머신과 레지스터

캐싱 기법을 사용한 머신의 수행 시간 비교를 통해서 이루어졌다. 각 실험 환경에 따라 시뮬레이터로부터 얻은 수행 결과는 (그림 4-1), (그림 4-2), (그림 4-3), (그림 4-4)와 같이 나타난다.

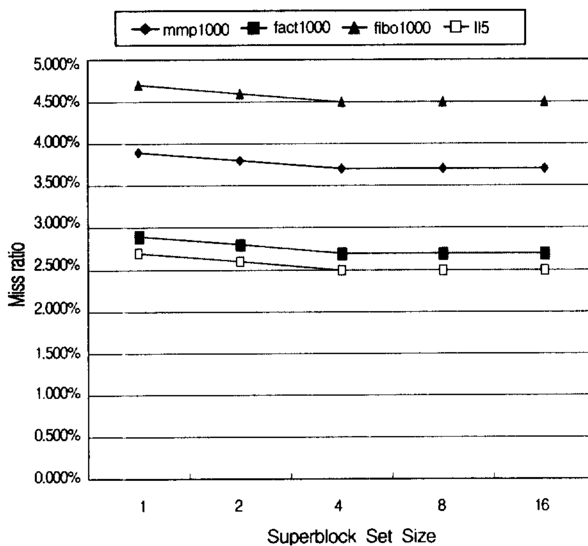
(그림 4-1)은 Superblock Set 크기가 4일 때, 캐쉬 크기에 따른 캐쉬 실패율을 나타낸 그림이다. 실험의 결과, 레지스터 캐쉬의 크기가 증가함에 따라 캐쉬 실패율도 감소함을 알 수 있다.



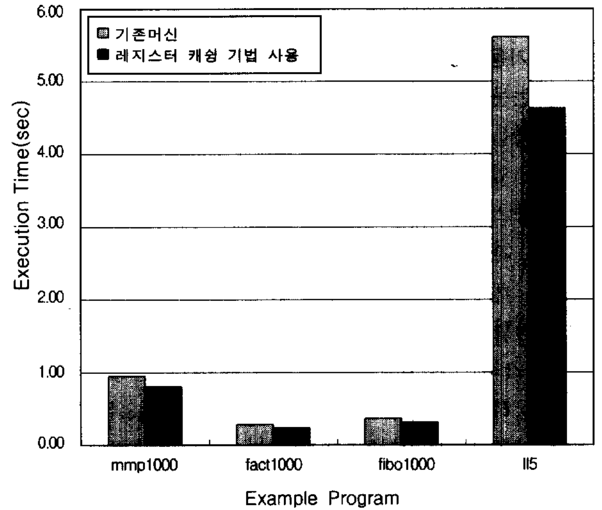
(그림 4-1) 캐쉬 크기에 따른 실패율

(그림 4-2)는 레지스터 캐쉬 크기가 20일때, Superblock Set 크기에 따른 캐쉬 실패율을 나타낸 그림이다. (그림 4-2)의 결과, Superblock Set 크기는 캐쉬 실패율에 커다란 영향을 미치지 않음을 알 수 있다.

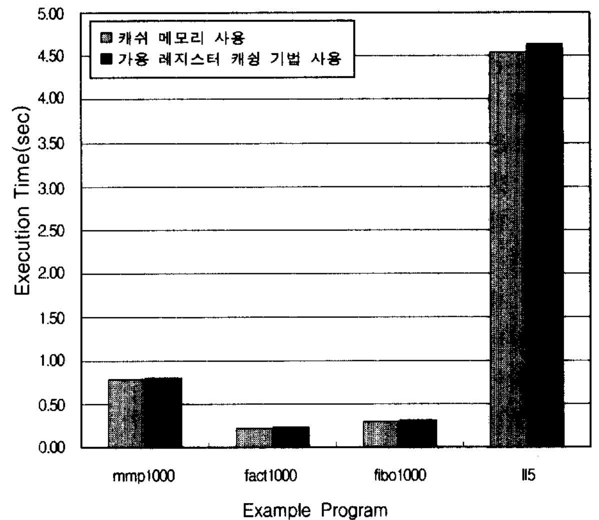
(그림 4-3)은 레지스터 캐쉬를 사용한 머신과 캐쉬를 사용하지 않은 머신의 수행 시간을 비교한 결과를 나타낸 그림이다. (그림 4-3)에서 레지스터 캐쉬를 사용한 다중스레드 모델이 캐쉬를 사용하지 않은 다중스레드 모델보다 적은 수행 시간을 요구한다는 것을 알 수 있다.



(그림 4-2) Superblock Set 크기에 따른 실패율



(그림 4-3) 레지스터 캐쉬 사용 머신의 수행 시간 평가



(그림 4-4) 캐쉬 메모리를 사용한 머신과 수행 시간 비교

(그림 4-4)는 레지스터 캐쉬를 사용한 머신과 캐쉬 메모리를 사용한 머신의 수행 시간을 나타낸 그림이다. (그림 4-4)에서 레지스터 캐쉬를 사용한 머신이 캐쉬 메모리를 사용한 머신보다는 프로그램 수행 시간이 많이 걸리나 그 차이는 매우 작다는 것을 알 수 있다.

전체 실험의 결과 레지스터 캐쉬를 사용한 다중스레드 모델이 레지스터 캐쉬를 사용하지 않은 모델보다 평균 수행 시간이 15% 정도 향상되었다. 또한 본 논문에서 제안한 가용 레지스터 기반 캐싱 기법을 사용한 다중스레드 모델은 캐쉬 메모리를 사용한 다중스레드 모델과 대등한 수행 성능을 보였다.

5. 결 론

다중스레드 모델의 성능을 향상시키려는 연구는 폰 노이만 모델에서 이미 효율성이 입증된 캐쉬와 선반입을 이용하여 성능 향상을 이루려고 하였다. 이러한 연구로 데이터플로우 기반 다중스레드 모델에 대하여 데이터 캐쉬와 명령어 캐쉬, I-구조 캐쉬가 설계되었고, 동기화를 수행할 때 프레임 메모리에서 발생하는 병목 현상을 제거하기 위해 동기화 캐쉬를 설계하여 동기화 인자를 선반입하는 기법이 연구되었다. 그러나 다중스레드 모델의 수행 성능 향상을 위해서 캐쉬 메모리를 다중스레드 모델에 부가시키는 방법은 캐쉬를 이용하기 위해서는 부득이 설계 변경이 필요하다. 또한 캐쉬의 가격이 하락했지만 일반 메모리에 비해 가격이 매우 높으므로 폰 노이만 모델에서도 아직까지 많은 양의 캐쉬를 사용하지 못하고 있다.

본 논문에서는 레지스터를 기반으로 동작하는 다중스레드 모델의 실행 모델 구현 비용이 증가되지 않으면서 캐쉬 메모리와 같은 효과를 이루기 위해서 가용 레지스터 기반 캐싱 기법을 제안하였다. 가용 레지스터 기반 캐싱 기법은 스레드가 수행될 때 사용하지 않는 가용 레지스터를 이용하여 레지스터 캐쉬 블록을 설정한 후 동기화 인자를 레지스터 캐쉬에 선반입하는 방식을 취한다. 그리고 가용 레지스터에 대한 정확한 예상은 스레드 내의 명령어 수준 레지스터 최적화 과정에 의해 수행된다. 이 방법은 설계 구조를 변경할 필요가 전혀 없다. 그리고 부가적으로 하드웨어로 추가되는 것이 전혀 없기 때문에 추가 비용이 거의 들지 않는다. 또한 캐싱 대상이 프레임 전체가 아니고 다중스레드 모델의 성능에 큰 영향을 미치는 동기화 인자만을 레지스터로 선반입하기 때문에 기존 방법보다 캐싱의 양도 적다.

제안한 가용 레지스터 기반 캐싱 기법을 레지스터 캐쉬를 부가시킨 시뮬레이터를 이용하여 실험한 결과 성능 향상을 위한 기법을 수행하지 않는 모델에 비해 월등한 처리율이 결과로서 나타났고, 캐쉬 메모리를 부가시킨 모델과 비교하였을 때도 처리율이 대등하게 나왔다. 본 논문은 캐쉬 메모리를 부가시키지 않고도 다중스레드 모델이 수행될 때 사용되지 않는 레지스터를 이용하여 캐쉬와 동일한 효과를 낼 수 있다는 것을 보였다. 향후 연구로서 캐쉬 교환 정책, 캐쉬 메모리를 위한 효율적인 선반입 기법이 연구되지 않고 있기 때문에, 캐쉬 메모리를 위한 효율적인 선반입 기법의 연구가 필요하다.

참 고 문 헌

[1] J. Chaitin, Gregory, "Register Allocation and Spilling via Graph Coloring," Proc. of the SIGPLAN '82 symp. on compiler construction, Boston, MA, SIGPLAN Notice, Vol.17, No.6, June 1982, pp.98-105.

[2] S. H. Ha, and et al., "Design and Implementation of a Massively Parallel Multithreaded Architecture : DAVRID," Journal of Electrical Engineering and Information Science. Vol.1, No.2, pp.15-25, 1996.

[3] H. H. Hum and G. R. Gao, "Supporting a dynamic SPMD model in a multithread architecture," In Proc. of Compcon Spring'93, 1993.

[4] R. A. Iannucci, "Parallel Machines : Parallel machine Languages The Emergence of Hybrid Dataflow Computer Architectures," Kluwer Academic Publishers, 1990.

[5] K. M. Kavi, "Design of Cache Memories for Multithreaded Dataflow Architecture," Int'l Symp. Computer Architecture, ACM Press, pp.253-264, 1995.

[6] R. S. Nikhil, "Can Dataflow Subsume von Neumann Computing?," In Proc. 16th Annual Int'l Symp. on Computer Architecture, pp.262-272, 1989.

[7] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-Machine Multicomputer : An architectural evaluation," In Proc. 20th Ann. Int. Symp. on Computer Architecture, May, 1993.

[8] K. E. Schauser, D. E. Culler, and T. von Eicken, "Compiler-Controlled Multithreading for Lenient Parallel Languages," 5th ACM Conference on Functional Programming Languages and Computer Architecture, LNCS Vol.523, pp.50-72, 1991.

[9] 권영필, 유원희, "향상된 스레드 생성을 위한 레지스터 최적화 기법", 99춘계 학술발표논문집 제26권 제1호, 한국정보과학회, pp.92-94, 1999.

[10] 고훈준, 권영필, 양창모, 유원희, "향상된 스레드를 위한 중간 언어와 그래프 생성," 한국산업기술학회지, 제1권 창간호, pp.67-74, 1997.

[11] 조승호, 황대준, 한상영, "다중스레드 컴퓨터 구조를 위한 캐쉬 기반 동기화 기법", 한국정보과학회논문지 제21권 제1호, 한국정보과학회, pp.44-52. 1994.

[12] 최종필, 하순회, 전주식, "다중스레딩 프레임 스케쥴링 정보를 이용한 선반입 방식 캐쉬", 한국정보과학회 제24권 제5호, pp.499-508, 1997.

고 훈 준

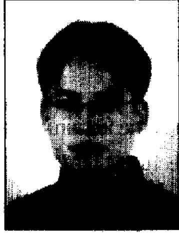
e-mail : hjkouh@hanmail.net

1998년 인하대학교 생물공학과 졸업(학사)

2000년 인하대학교 전자계산공학과 졸업(석사)

2000년~현재 인하대학교 전자계산공학과 박사과정

관심분야 : 컴파일러, 병렬처리, 디버거, 프로그래밍 언어



권 영 필

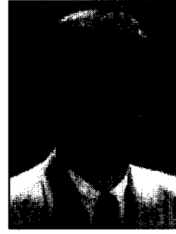
e-mail : yfkwon@hanmail.net

1998년 동양대학교 컴퓨터공학과 졸업 (학사)

2000년 인하대학교 전자계산공학과 졸업 (석사)

2000년~현재 (주)다이알로직코리아

관심분야 : 병렬처리, 분산 컴퓨팅, 프로그래밍 언어



유 원 희

e-mail : whyoo@inha.ac.kr

1975년 서울대학교 공과대학 응용수학과 졸업

1978년 서울대학교 대학원 계산학 전공 (이학석사).

1985년 서울대학교 대학원 계산학 전공 (이학박사)

1992년~1993년 University of California, Irvine 객원연구원

1979년~현재 인하대학교 전자계산공학과 교수

관심분야 : 프로그래밍 언어, 컴파일러, 실시간 시스템, 병렬 시스템