동적으로 할당된 구조체를 위한 압축된 필드 재배치
(Compact Field Remapping for Dynamically Allocated Structures)

김 정 은 † 한 환 수 ††
(Jeongeun Kim) (Hwansoo Han)

요 약 내장형 시스템과 범용 시스템의 가장 큰 차이는 유한한 전력인 배터리를 사용한다는 것과 대용량의 디스크를 사용하지 않고 메모리에 의존한다는 것이다. 특히 멀티미디어 데이타를 처리하는 응용프로그램이 늘어감에 따라 메모리 사용량이 기하급수적으로 증가하고 있어서 메모리가 성능과 에너지 소비의 병목지점으로 작용하게 되었다. 따라서 데이타 접근 비용을 줄이고자 하는 시도가 많이 이루어지고 있다. 대부분의 프로그램은 지역성을 갖는다. 지역성은 한번 참조된 데이타가 조만간 다시 참조된다는 시간적 지역성(temporal locality)과 근접한 곳에 할당된 데이타끼리 함께 참조된다는 공간적 지역성(spatial locality)으로 나눌 수 있다. 최근의 많은 임베디드시스템은 이 두 가지 지역성을 이용한 캐시 메모리를 사용함으로써 메모리 접근 시간을 대폭 줄이고 있다. 우리는 이 논문에서 낭비되는 메모리 공간을 줄이고, 캐시 실패율(cache miss rate)과 프로그램 수행시간을 줄일 수 있도록 구조체 형식의 데이타를 항목(field) 별로 재배치시키는 알고리즘을 제안하고자 한다. 이 알고리즘은 동적으로 할당되는 구조체의 각 필드를 압축된 형태로 모아서 재배치함으로써, 실험에서 사용한 Olden 벤치마크의 L1캐시 실패는 평균 13.9%를, L2 캐시 실패는 평균 15.9%를 이전 연구들보다 줄일 수 있었다. 수행시간 또한 이전의 방법보다 평균 10.9% 줄인 결과를 얻을 수 있었다.

키워드 : 데이타 재구성, 구조체 항목 재배치

Abstract The most significant difference of embedded systems from general purpose systems is that embedded systems are allowed to use only limited resources including battery and memory. Especially, the number of applications increases which deal with multimedia data. In those systems with high data computations, the delay of memory access is one of the major bottlenecks hurting the system performance. As a result, many researchers have investigated various techniques to reduce the memory access cost. Most programs generally have locality in memory references. Temporal locality of references means that a resource accessed at one point will be used again in the near future. Spatial locality of references is that likelihood of using a resource gets higher if resources near it were just accessed. The latest embedded processors usually adapt cache memory to exploit these two types of localities. Processors access faster cache memory than off-chip memory, reducing the latency. In this paper, we will propose the enhanced dynamic allocation technique for structure-type data in order to eliminate unused memory space and to reduce both the cache miss rate and the application execution time. The proposed approach aggregates fields from multiple records dynamically allocated and consecutively remaps them on the memory space. Experiments on Olden benchmarks show 13.9% L1 cache miss rate drop and 15.9% L2 cache miss drop on average, compared to the previously proposed techniques. We also find execution time reduced by 10.9% on average, compared to the previous work.

Key words : Data reorganization, field remapping

1. 서 론

범용 시스템과 비교했을 때 내장형 시스템의 가장 큰 특성은 유한한 자원에 의존해야 한다는 점이다. 정해진 전력만을 소비해야 하고, 대용량의 디스크가 아닌 작은 용량의 메모리를 이용하는 것이 대표적이다. 단순한 작업을 반복했던 내장형 시스템에서 많은 양의 데이타 처

† 학생회원 : 한국과학기술원 전산학과
jekim@arcs.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
hshan@cs.kaist.ac.kr
논문접수 : 2004년 10월 7일
심사완료 : 2005년 8월 18일

리 요구가 증가함에 따라 시스템 설계의 복잡도가 점점 증가하고 있다. 이에 메모리는 성능의 가장 큰 병목지점일 뿐만 아니라 에너지 소비의 주요한 요인 중 하나로 작용한다. 따라서 특별한 하드웨어를 추가 하거나 새로운 시스템 구조를 제안 하여 메모리의 성능을 개선시키는 방안, 컴파일러에 의한 소프트웨어 수준에서의 개선 방법들이 연구되어 왔다. Panda와 그의 동료들은 [1,2]에서 메모리와 관련된 다양한 최적화 기법들을 조사하고 있다. 메모리 대역폭(memory bandwidth)을 높이기 위해 현대의 내장형 시스템들은 캐시를 장착하고 있는 경우가 많다. 지역성 개선을 통한 최적화는 데이타의 재사용성과 인접된 데이타의 사용가능성을 향상시켜 메모리 접근 비용을 감소시키는 결과를 얻을 수 있다. 따라서 지역성을 개선시키기 위한 데이타 재배치는 응용프로그램들의 성능을 개선시키는데 도움이 된다[3-5]. 그러나 위 연구들은 정적으로 할당되는 메모리의 성능만을 개선시키는데 그치고 있다. 많은 데이타를 이용하는 응용프로그램일수록 정적 데이타 보다 동적 데이타를 많이 이용하고 있다. 따라서 동적으로 할당되는 데이타들에 대한 최적화가 필요하다.

전통적으로 컴파일러를 이용한 최적화는 프로그램의 정적 메모리 할당에 중요한 역할을 해 왔다. Truong을 비롯한 연구원들은 필드 재구성과 인스턴스간의 겹침을 이용하여 데이타 레이아웃을 바꾸는 기법을 제안하고 있다[6]. 여러 레코드에서 자주 사용되는 특정 필드들을 묶어 메모리상에 위치시키는 방식이다. 그러나 구조체를 같은 크기의 두 부분(chunk)으로 나누어야 하는 단점과 동적으로 할당되는 데이타의 재배치에는 적용할 수 없다는 문제점을 가지고 있다. Chilimbi와 그의 동료들은 구조체 형식 데이타의 참조로 발생하는 레벨 2 캐시 실패율을 줄일 수 있는 최적화 기법을 제안하였다[7]. 구조체의 여러 항목 중, 자주 사용되는 항목(hot field)과 자주 사용되지 않는 항목(cold field)으로 나누어 분리한 뒤, 자주 사용되는 항목을 갖고 있는 구조체(hot structure)에 자주 사용되지 않는 항목을 가지고 있는 구조체(cold structure) 정보를 가리키는 포인터 항목을 추가하는 방법을 제안하고 있다. 이 때 하나의 구조체 자료구조가 둘로 나뉘고 포인터 항목이 추가됨으로써 메모리 사용량이 증가하게 된다. 또한 자주 사용되지 않는 영역을 참조하고자 할 때 포인터 항목을 통한 간접 접근 방식을 따라야 하므로 메모리 접근 횟수가 늘고 이로 인한 캐시 실패 오버헤드가 발생할 수 있다. 반면 Rabbah와 Palem은 구조체 데이타 간의 메모리 배치를 변경하여 할당하는 방법을 제안하였다[8]. 하나의 레코드를 이루는 필드끼리 인접하도록 주소를 부여하지 않고 다른 레코드에 속한 동일한 필드들끼리 인접하도록 주소를 부여하는 방식이다. 그러나 그 방법은 동적으로 할당되는 레코드의 경우 메모리 낭비가 발생할 수 있다는 단점을 가지고 있다. 따라서 우리는 이 논문을 통해 필드 크기에 구애 받지 않고, 필드 사이에 메모리 낭비가 발생하지 않도록 레코드를 동적으로 할당하는 방법론을 제안하고자 한다.

다음 장에서 우리의 연구 동기를 설명할 것이다. 3장에서는 기존의 연구에서 언급되었던 데이타 재배치 알고리즘과 우리가 제안하는 개선된 알고리즘을 설명한다. 실험환경과 다양한 실험 결과는 4장에서 논의 될 것이며, 5장은 이번 연구에 대한 결론과 다음 연구 과제에 대한 언급을 포함하고 있다.

2. 연구 동기

2.1 지역성(locality)

참조의 지역성은 특정 자원을 여러 번 참조하는 현상을 다루는 데서 비롯된다. 참조의 지역성은 크게 시간적 지역성(temporal locality)과 공간적 지역성(spatial locality) 두 가지로 나눌 수 있다. 참조의 시간적 지역성은 한번 참조되면 가까운 장래에 다시 참조될 가능성이 높은 성질을 말한다. 공간적 지역성은 참조가 일어난 곳의 주변의 자원들이 계속해서 참조되는 경향을 의미한다. 이러한 개념은 메모리에도 적용이 가능하다. 예를 들어 순차적으로 코드를 실행하거나 배열이나 스택을 이용할 때, 혹은 프로그래머들이 관련이 있는 변수를 가까운 곳에 선언하는 습관은 한번 접근된 메모리 영역과 근접한 곳의 영역을 계속적으로 접근하게 하는데 이를 공간적 지역성이라 부른다. 뿐만 아니라 순환 문이나 함수, 스택, 합계 계산 등에 사용되는 변수들이 계속해서 사용될 때 시간 지역성이 있다고 말한다.

참조의 지역성을 개선시키는 것은 메모리 최적화의 대표적인 기술 중 하나이다. 이것은 메모리 계층 구조의 여러 수준에서 이루어지고 있다. 특히 캐시는 작지만 빠른 메모리로 메모리와 프로세서 사이의 성능 차이를 중재하기 위해 특별히 설계된 장치이다. 일단 한번 참조된 데이타는 캐시에 저장이 되어 다음 번 접근 시에 느린 메모리가 아닌, 좀 더 빠른 캐시 메모리의 접근만으로 이용할 수 있게 되어 잠재적인 성능 향상을 가져오게 된다.

위에서 살펴본 바와 같이 시간적 지역성과 공간적 지역성 모두 성능개선에 중요한 요소로 작용한다. 그러나 두 가지 지역성이 비례관계에 있는 것은 아니다. 즉, 높은 시간적 지역성이 높은 공간적 지역성을 의미하는 것은 아니다. 하지만 공간적 지역성과 시간적 지역성은 따로 떼어놓고 생각할 수 없는 개념이다. 만약 두 지역성이 모두 높다면, 캐시 실패는 감소되고 성능은 개선될

수 있다. 따라서 이 논문에서 시간적 지역성이 높은 데이타들 사이에 공간적 지역성을 높일 수 있는 메모리 배치 알고리즘을 제안하고자 한다.

2.2 관련 연구

Rabbah와 Palem은 총 데이타 접근 비용을 줄이기 위한 구조체 데이타 재배치 알고리즘을 제안하였다[8]. 구조체는 한 번의 선언으로 다양한 데이타 항목의 집합을 구성할 수 있는 자료구조이다. 많은 응용프로그램들이 구조체를 사용하고 있어 이와 관련된 여러 최적화 기법이 제안되었다. 예를 들어 구조체를 이용한 연결 목록을 탐색하여 키 값과 일치하는 데이타의 특정 항목 값을 변경하는 함수가 있다고 가정하자(그림 1). 각각의 레코드는 키(key) 항목, 데이타(datum) 항목, 그리고 연결 목록에서 다음 레코드를 가리키는 next 항목으로 구성되어있다. 이 함수는 특정 키 값과 일치하는 레코드를 찾기 위해 키 항목과 next 항목을 자주 이용하게 되지만 상대적으로 데이타 항목의 접근은 드물게 일어난다. 구조체 형식 데이타의 전통적인 메모리 배치 방법은 레코드 단위로 할당하는 것이고(그림 2(a)), 레코드의 특정 항목을 참조하고자 할 때 메모리 계층구조에 따라서 캐시 라인 단위로 메모리에서 읽어 들일 때, 당장에 필요 없는 항목까지 함께 읽어 들여야만 한다. 이로 인해

```
Node {
    key_t key;                    // 4 bytes
    data_t datum;                 // 8 bytes
    Node   *next;                 // 4 bytes
} *T;
...
data_t search(s_type *target) {
    Node *cur = T;
    while(cur != NULL) {
        if(cur->key ==target->key);
            return cur->data;
        cur = cur->next;
    }
    return NOT_FOUND;
}
```

그림 1 동기 예제 : Node 구조체 정의와 Node 오브젝트를 이용한 탐색 함수

캐시 메모리상에 유용한 데이타 영역이 줄어들게 되고 캐시 실패를 유발하게 된다. 캐시 실패는 메모리 지연을 발생시켜 성능 저하 및 높은 에너지 소비를 유발하게 되는 주된 요인이므로 캐시 메모리의 공간 지역성(spatial locality)을 높일 수 있도록 지속적으로 접근되는 키 항목과 next 항목들이 같은 캐시 라인 상에 할당되도록 재배치한다면 캐시 실패율을 줄일 수 있게 된다. 이점에 착안하여 Rabbah와 Palem은 공간 지역성을 높이기 위해 전통적인 레코드 단위 배치방식이 아닌, 항목 단위의 배치방식을 제안하였다(그림 2(b)).

각각의 동적으로 할당되는 레코드의 특정 항목 상대 주소는 다음과 같이 계산할 수 있다.

$$Original(P \rightarrow f_n) = \sum_{i=1}^{n-1} FieldSize(*P.f_i) \quad (1)$$

$$DDRemap(P \rightarrow f_n) = \sum_{i=1}^{n-1} StaggerConstraint \times MaxFieldSize(*P) \quad (2)$$

P는 형식이 R인 특정 레코드를 가리키는 포인터이고 *MaxFieldSize(*P)*는 P가 가리키는 레코드 R의 최대 항목크기를 의미한다. *StaggerConstraint*는 컴파일러에 의해 정의된 값으로 하나의 메모리 풀[1]에 저장될 수 있는 레코드의 개수를 의미한다. 전통적인 배치 방법이나, 알고리즘 상에서 재배치를 하지 않는 경우의 메모리 주소 계산은 식 (1)과 같다. 그리고 재배치 된 레코드의 주소 계산은 식 (2)에 의해 계산될 수 있다.

2.3 한계점

동적으로 할당되는 구조체 형식의 데이타를 DDRemap 식을 이용하여 재배치하는 알고리즘의 경우, 컴파일러는 구조체에 속한 모든 항목들의 크기를 최대 항목 크기로

1) Rabbah의 방법은 일단 충분히 큰 메모리를 선점하고, 선점된 메모리 안에서 구조체 항목을 재배치 하게 된다. 이 때 선점된 영역을 메모리 풀이라 하고, 하나의 메모리 풀 안에 저장 가능한 레코드 개수가 *StaggerConstraint* 값이다. 이에 대한 자세한 언급은 3장 2절에서 하기로 한다.

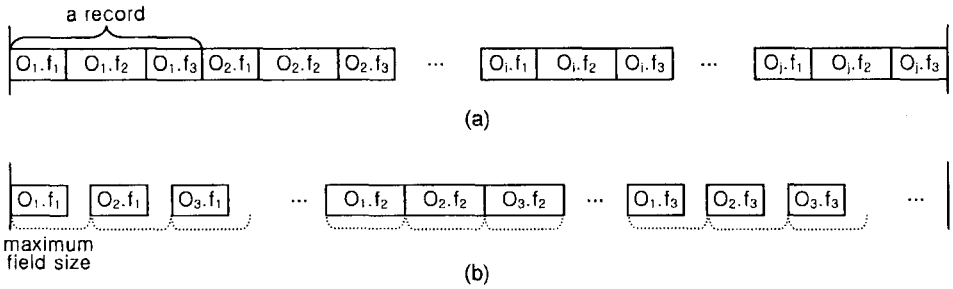


그림 2 (a) 전통적인 방식에 따른 레코드 배치 후의 레이아웃, (b) DDRemap에 의한 동적 레코드 재배치 후의 레이아웃

고정하여 새로운 주소를 할당하고 접근하게 된다. 이 때, 각 레코드의 항목을 접근할 때 주소계산의 복잡도가 전통적인 방식의 경우와 같게 되어 주소 계산에 따른 오버헤드를 고려하지 않아도 된다. 그러나 항목간의 크기가 다를 경우, 모든 항목의 크기를 최대 항목 크기로 고정함으로써 사용되지 않는 메모리가 발생할 수 있다. 낭비되는 메모리의 양은 항목간의 크기 변화량에 비례하게 된다. 예를 들어 구조체 상에서 데이타(datum) 항목이 8 바이트로 선언되고, 키 항목과 next항목이 각각 4바이트의 자료구조로 선언이 될 때, 동적으로 할당되는 레코드의 각 필드는 8바이트씩 차지하게 된다. 따라서 하나의 레코드에 할당되는 총 메모리는 16 바이트가 아닌 24바이트가 되고 약 33%의 메모리 낭비가 발생하게 된다. 그림 3(a)는 이런 상황을 보여주고 있다. 빗금 친 공간이 사용되지 않는 메모리 영역을 의미한다.

유효한 데이타 사이에 사용되지 않는 메모리 공간이 늘어날수록 캐시 실패율은 증가하게 된다. 캐시 실패와 페이지 실패는 메모리 지연을 발생시켜 에너지 소비 증가와 성능저하를 초래하게 된다. 따라서 우리는 이 논문을 통해 메모리에 동적으로 할당되는 구조체 형식 데이타를 그림 3(b)와 같이 메모리 낭비 없이 할당할 수 있는 방법론을 제안하고자 한다.

3. 재배치 알고리즘

우리는 포인터를 이용하여 동적으로 할당되는 데이타를 많이 이용하는 응용프로그램에 초점을 두고, Rabbah와 Palem의 동적 구조체 할당 기법보다 개선된 메모리 최적화 기법을 제안하고자 한다. 데이타를 메모리상에 재배치 한다는 것은, 해당 데이타의 메모리 주소를 바꾼다는 것을 의미한다. 동적으로 할당된 구조체 데이타의 필드를 접근하고자 할 때, "→" 연산자를 이용하여 "*P→f*"와 같이 기술한다. 변수 *P*는 항상 구조체 데이타의 첫 번째 필드의 주소를 갖게 되고, 필드 f의 상대주소를 계산하여 P값에 합산함으로써 필드 f에 접근하게 된다. 따라서 메모리 재배치에 따른 필드의 상대주소 계산방법을 바꿈으로써 구조체 데이타의 필드를 재배치할 수 있다.

3.1 압축된 필드 재배치 알고리즘

동적으로 할당되는 레코드의 재배치 알고리즘의 요점은 그림 4에서 볼 수 있다. 알고리즘의 입력은 재배치 정보를 가지고 있는 프로그램 P이다. 재배치 정보로 NAP를 사용할 수 있다. NAP(*neighbor affinity probability*)란 Rabbah와 Palem이 제안한 개념으로 0과 1 사이 값을 갖는다. NAP 값이 1에 가까울수록 분석단계에서 사용된 메모리배치가 프로파일 된 메모리 접근 순서와 잘 맞는 다는 것을 의미한다. 반대로 0에 가까울수록 메모리 배치와 데이타 접근 패턴이 잘 맞지 않음을 의미하여 재배치의 필요성이 커지게 된다.

```
Input: Program P with annotation.
Output: Field remapped P.

01. for each statement S in P
02.     if S if of the form 'x ← Allocate(StructureSize(R))' then
03.         if R is marked for remapping
04.             replace S with 'x ← MyAlloc(R)'
05.             generate Wrapper(R) if necessary
06.         end if
07.     else if S if of the form 'P→f'
08.         if type of P is marked for remapping then
09.             replace S with 'get_addr(P,f)'
10.         else if
11.     end if
12. end for
```

그림 4 동적 레코드의 재배치 알고리즘

알고리즘의 첫 번째 단계는 각 문장이 재배치를 필요로 하는 구조체 타입의 메모리를 할당하는 형식인지를 체크하는 것이다(라인 2-3). 조건을 만족할 경우, 해당 문장을 포장 함수로 감싸준다. 위 조건을 만족하지 않는 경우에는 재배치를 필요로 하는 구조체 형식의 동적 데이타를 접근하는 형식의 문장인지 확인한다. 이 조건을 만족할 경우, 포인터를 이용한 메모리 접근을 *get_addr*이라는 매크로로 대체한다. get_addr은 레코드의 특정 항목의 주소를 계산하는 매크로이다. 낭비되는 공간을 없애기 위하여 우리는 *CFRemap(Compact Field Remapping)*이라는 필드의 상대주소 계산식을 제안하고 이는 다음과 같다

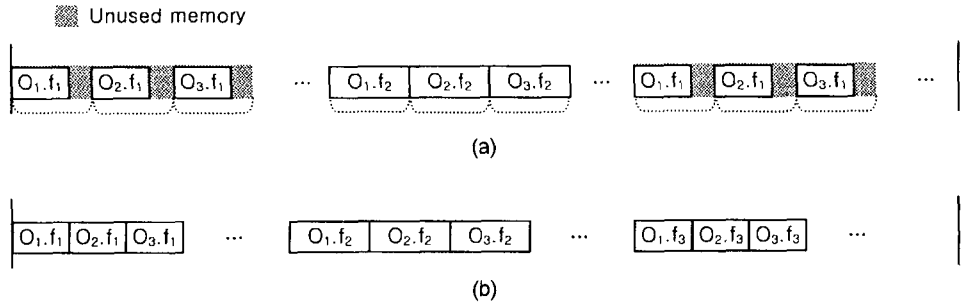


그림 3 (a)메모리 낭비가 발생한 레이아웃 (b)메모리 낭비를 없앤 개선된 방법의 레이아웃

$$CFRemap(P \rightarrow f_i) = \sum_{l=1}^{MaxObjCount-NthObj} FieldSize(f_l) + \sum_{l=2}^{i-1} FieldSize(f_l) \times MaxObjCount + \sum_{l=1}^{NthObj} FieldSize(f_l) \quad (3)$$

여기에서 P는 형식 R인 특정 레코드를 가리키는 포인터이고 *FieldSize*(f_i)는 레코드의 i번째 필드 f_i의 크기를 나타낸다. *MaxObjCount*는 컴파일러에 정의된 값으로 하나의 메모리 풀에 저장할 수 있는 레코드 수를 의미한다. 이 값은 *DDRemap*에서의 *StaggerConstraint*와 유사하지만 *DDRemap*에서는 모든 필드의 크기가 최대필드크기로 동일하다는 가정하에 하나의 메모리 풀 안에 저장될 수 있는 레코드 크기를 의미한다는 점에서 다르다. 따라서 모든 필드의 크기가 같을 경우 *MaxObjCount*와 *StaggerConstraint* 값이 일치하게 되지만, 필드 크기가 다를 경우 메모리 풀 크기가 동일할 때 *MaxObjCount*값은 *StaggerConstraints* 값보다 큰 값을 갖게 된다. 이는 동일한 메모리 영역에 더 많은 레코드를 저장할 수 있음을 의미한다. *NthObj*은 P에 의해 참조되는 레코드가 해당 메모리 풀에서 몇 번째 객체인가를 나타낸다.

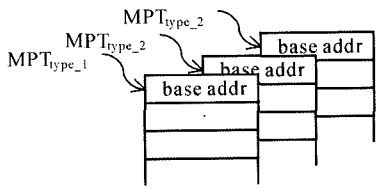
수식 (3)을 이용할 경우 우리는 두 가지 문제점에 봉착하게 된다. 첫 번째는 수식 (1)과 (2)에 비해 계산복잡도가 월등히 높다는 것이다. 그리고 수식 (1)과 (2)와 달리 *NthObj*을 알아내야 한다. 다음 장에서 일단 *NthObj* 값을 알아내는 방법을 살펴보고, 이를 이용하여 계산복잡도를 낮추는 방안에 대하여 살펴보겠다.

3.2 *NthObj* 계산

DDRemap을 통한 필드의 주소 계산방식에 있어서는 모든 필드의 크기가 최대필드크기로 동일하였기 때문에 *NthObj* 없이도 주소 계산이 가능하였다. 그러나 우리가 제안하는 방식에서는 각각의 필드를 위해 필드 크기만큼의 메모리 공간이 할당되기 때문에 *NthObj*을 알 필요가 있다. 여기서 *NthObj*을 알아낼 수 있는 방법 몇 가지를 소개한다.

- **메모리 풀 테이블(MPT) 유지** : 할당되는 모든 메모리 풀을 메모리 풀 테이블(MPT : Memory Pool Table)을 이용하여 유지하고, 동적 레코드를 가리키는 포인터 P의 베이스 주소를 MPT의 값과 비교함으로써 *NthObj*을 얻을 수 있다. 예를 들어 MPT를 그림 5(a)와 같이 구성했을 때, MPT의 각각의 값은 메모리 풀의 시작주소를 저장하고 있다. 멀티미디어 프로그램의 경우 많은 데이타를 다루기 때문에 모든 데이타를 저장하기에 하나의 메모리 풀은 충분하지 않다. 따라서 여러 개의 메모리 풀이 사용되고 각 메모리 풀의 시작주소를 MPT에 저장한다. 한 레코드의 특정 필드를 접근하기 위해 포인터 P가 주어지면, 우선 MPT에 저장된 메모리 풀의 주소와 P의 베이스 주소를 비교한다. 메모리 풀의 크기가 일정하기 때문에 MPT를 검색함으로써 메모리 풀의 시작주소는 쉽게 찾을 수 있고, 메모리 풀의 시작주소와 P의 값의 차이를 P가 가리키고 있는 레코드의 첫 번째 필드 크기로 나누어 주면 손쉽게 *NthObj*값을 얻을 수 있다.

이 방법을 이용할 때 우리는 레코드 형식마다 별도의 MPT를 유지하기 위한 자료구조와 추가적인 메모리를 사용해야 하고, 데이타 양이 증가함에 따라 MPT의 크기도 증가하게 된다. 또한 재배열 된 레코드에 대한 접근 시에 항상 MPT를 검색해야 하기 때문에 검색복잡도와 추가적인 메모리 접근에 대한 부담이 증가하게 된다.



NthObj= (P-search_base_addr(P)) / MaximumFieldSize

(a)

```
Node {
    no_t NthObj;
  key_t key;
  data_t datum;
  Node *next;
}
```

NthObj= (no_t)*P

(b)

그림 5 (a) MPT 테이블 예제 (b) *NthObj* 필드를 추가하여 수정한 구조체 형식 데이타의 정의

- **구조체에 필드 추가** : 구조체의 첫 번째 필드는 *P라고 기술함으로써 포인터 P로 직접 접근이 가능하다. 따라서 그림 5(b)와 같이 *NthObj* 필드를 재배치할 구조체 형식의 첫 번째 필드로 추가하면 간단히 값을 참조하여 해당 레코드가 메모리 풀에서 몇 번째로 할당되었는지 쉽게 알아낼 수 있다.

이 방법은 아주 간단하게 구현이 가능하고 MPT와 같은 추가적인 자료구조를 사용하지 않지만 기존의 자료구조에 필드를 추가함으로써 재배치를 하지 않았을 때 보다는 많은 메모리를 소비하게 된다. 여기서 *NthObj*을 저장하기 위한 필드의 크기는 하나의 메모리 풀에 저장될 수 있는 레코드 수, 즉 *MaxObjCount*값에 따라 달라지게 된다. 가령 *MaxObjCount*값이 128 미만

인 경우 한 바이트 크기의 데이타 타입(char)만으로도 충분히 표현할 수 있다. 그러나 추가적인 메모리의 소비 문제뿐만 아니라 포인터를 통해 재배치된 레코드의 필드를 접근하고자 할 때 *NthObj*값을 알아내고 필드의 실제주소를 계산하게 되므로 항상 메모리 접근이 두 번 일어나게 된다. 메모리 접근은 프로그램 수행에 있어 비용이 많이 소비되는 작업 중 하나이므로 재배치된 모든 레코드의 이용 시에 메모리 접근이 2배가 된다는 것은 커다란 성능저하를 가져오게 된다.

실험적으로 위의 두 알고리즘을 수행해 본 결과 MPT 혹은 구조체에 추가된 필드를 접근하는 비용이 상당히 커서 오히려 전체적인 수행 시간이 저하되는 것을 알게 되었다. 따라서, 가능한 적은 오버헤드를 가지도록 아래의 기법을 고안하게 되었다.

- **메모리 풀의 시작 주소 고정** : 동적 레코드를 가리키는 포인터 P가 주어졌을 때, P가 가리키는 레코드가 속한 메모리 풀의 시작주소를 알면 메모리 풀의 시작주소와 P의 값의 차이를 첫 번째 필드의 크기로 나눔으로써 *NthObj*값을 알아낼 수 있다. 그런데 기존의 방식대로라면 메모리 풀 역시 동적으로 할당이 되어 MPT 등을 통하여 메모리 풀의 시작 주소를 저장하여야 했다. 그러나 메모리 풀의 주소가 일정한 규칙을 따른다면 메모리 풀의 시작주소를 기록할 필요 없이 레코드를 가리키는 포인터만을 통해서 메모리 풀의 시작주소를 알아낼 수 있다. 예를 들어 메모리 풀 시작주소의 하위 12 비트가 0이고 메모리 풀의 크기가 4096 바이트라고 가정하자. 레코드를 가리키는 포인터 P의 값이 0×4b32900c일 때, P가 속한 메모리 풀의 시작 주소는 0×4b329000이 되고 메모리 풀의 시작주소와 P 값의 차이는 c(=12)가 된다. 포인터 P가 가리키는 레코드의 첫 번째 필드의 크기가 4 바이트라고 가정하면, 결국 참조하고자 하는 레코드는 메모리 풀에서 4번째로 할당된 객체임을 알아낼 수 있다.

메모리 풀의 시작주소를 고정시키기 위하여 그림 6과 같은 이중 포장 함수를 사용할 수 있다. 이 함수가 처음 호출되었을 때 메모리를 관리하는 변수들을 초기화 하고(라인 1) 메모리 풀에 더 이상 사용할 공간이 없으면(라인 2) 일단 하나의 메모리 풀을 할당 받아야 한다. 메모리 풀을 위한 커다란 메모리(뱅크영역)를 미리 예약하게 되는데, 뱅크영역의 크기는 하나의 메모리 풀 크기의 k배로 정한다. k는 컴파일러에 의해 정의되는 값으로 우리는 1000이라는 값을 사용하였다. 뱅크영역을 할당 받으면(라인 4) BankLimit으로 하여금 뱅크영역의 끝부분을 가리키도록 하여(라인 5) 후에 뱅크영역 부족을 체크하도록 한다. 뱅크영역은 malloc() 함수를 통해 할당되므로, 뱅크영역의 시작은 랜덤하게 결정된다. 따라서 마스킹을 통해 BankAlloc 포인터를 특정 위치로 이동시켜준다(라인 6). BankAlloc은 사용 가능한 뱅크영역의 시작위치를 가리키는 포인터이다. 메모리 풀은 뱅크영역의 맨 윗부분부터 미리 정해진 메모리 풀 크기만큼 할당을 받고(라인 8), 사용가능한 뱅크영역을 가리키는 BankAlloc값은 메모리 풀 크기만큼 증가시켜야 한다(라인 9). 하나의 메모리 풀 안에 저장할 수 있는 구조체 데이타의 수(MaxObjCount)는 정해져 있기 때문에 메모리 풀 안에서 마지막 구조체의 첫 번째 필드를 저장한 후의 주소값을 알아낼 수 있다. 이를 PoolLimit에 저장시켜 놓음으로써 후에 메모리 풀의 부족을 쉽게 체크할 수 있다. PoolAlloc은 항상 사용 가능한 메모리 풀의 첫 번째 위치를 가리키므로 AllocatedAddress에 PoolAlloc값을 할당하고(라인 12), PoolAlloc값은 첫 번째 필드 크기만큼 증가시켜줌으로써(라인 13) 다음번 함수 호출에 대비한다. 이중 포장 함수를 사용하면 *NthObj*을 다음과 같은 매크로(macro)를 통해 간단히 계산할 수 있다.

$$NthObj = (P \ \& \ BitMask) \ / \ FieldSize(*P.f_1) \quad (4)$$

*BitMask*는 P가 가리키는 레코드가 속한 메모리 풀

```
Input: Record type R and stagger constant SC.
Output: Valid heap (base) address where R is allocated.
// use 4KB memory pool, k=1000 (in our experiment)

01.  Static Initialize                          // do it once for the first time
02. if PoolAlloc = PoolLimit then               // extend Bank
03.      if BankAlloc + 4KB > BankLimit
04.           BankAlloc ← malloc(4096 * k)
05.           BankLimit ← BankAlloc + 4096 * k - 1
06.           BankAlloc ← (BankAlloc + 4095) & _0xfffff000
07.      end if
08.      PoolAlloc ← BankAlloc
09.      BankAlloc ← BankAlloc + 4096
10.      PoolLimit ← PoolAlloc + MaxObjCount*FieldSize(R.f1)
11. end if
12. AllocatedAddress ← PoolAlloc
13. PoolBase ← PoolBase + FieldSize(R.f1)
14. Return AllocatedAddress
```

그림 6 이중 포장 함수 예제

의 시작 주소를 알아내기 위한 비트 마스크로 0xffffff와 (메모리 풀 크기-1)를 'exclusive or'의 연산을 통해 정할 수 있다.

이중 포장함수를 사용했을 때는 타입에 따른 포장함수가 필요하게 된다. 하지만 재배치를 하지 않았을 때와 동일한 양의 메모리가 소비되고, *NthObj*을 구하기 위한 추가적인 메모리 접근이 일어나지 않는다. 또한 어플리케이션에서 주로 사용되는 구조체 형식의 자료구조가 대부분 하나 혹은 두 개 정도이므로 이 논문에서는 이중 포장함수를 이용하여 *NthObj*을 구하고 있다.

3.3 주소 계산의 간소화

CFRemapping을 이용함에 있어 남은 과제는 필드 주소의 계산복잡도를 낮추는 것이다. 우선 수식 (3)의 *NthObj*을 수식 (4)로 대치하고 수학적 치환과 교환법칙 등을 이용하여 정리 하면 수식 (5)를 얻을 수 있다.

$$CFRemap(P \rightarrow f_i) = \sum_{f=1}^{i-1} FieldSize(f_i) \times MaxObjCount + (FieldSize(f_i) - FieldSize(f_1)) \times (P \,\&\, BitMask) / FieldSize(f_1) \quad (5)$$

수식 (5)는 재배치 하지 않은 경우의 주소 계산식인 수식 (1)에 비해 곱하기 연산 1번, 나누기 연산 1번, 비트 연산 1번, 뺄셈 1번이 추가된다. 곱하기와 나누기 연산의 경우 덧셈 뺄셈보다도 더 많은 사이클이 소비되는 연산이다. 따라서 피제수나 제수 등이 2의 n제곱 형태를 이룰 때, 곱하기나 나누기를 쉬프트(shift) 연산으로 변경할 수 있다. 필드 크기의 경우, 대부분의 응용프로그램에서 필드의 크기는 2의 제곱 형태이기 때문에, 많은 경우에 '/FieldSize(f_1)'을 '>>Bits(*P.f_1)'로 바꿀 수 있다. 따라서 간소화 된 주소 계산식 (6)을 얻을 수 있게 된다.

$$CFRemap(P \rightarrow f_i) = \sum_{f=1}^{i-1} FieldSize(f_i) \times MaxObjCount + (FieldSize(f_i) - FieldSize(f_1)) \times (P \,\&\, BitMask) >> Bits(f_1) \quad (6)$$

수식 (6)의 경우에도 개선의 여지는 남아있다. 우선 참고하고자 하는 필드와 첫 번째 필드의 크기가 같을 경우, '(FieldSize(f_i)-FieldSize(f_1))'의 값이 0이 되므로 수식 (1)과 계산복잡도가 같게 된다. 뿐만 아니라 필드 크기의 차가 2의 n제곱(단, n>=1)이 될 경우에, 곱하기 연산 역시 쉬프트(shift) 연산으로 바꿀 수 있다.

4. 성능 평가

4.1 실험 환경

OLDEN 벤치마크 버전 1.1의 일부분을 사용하여 실험하였다. OLDEN 벤치마크는 병렬 컴퓨터의 일종인 CM5를 위한 OLDEN C* 컴파일러를 테스트하기 위해 작성된 벤치마크이다. 모든 프로그램은 C로 작성되었고 병렬처리를 위한 라이브러리를 호출하는 매크로를 포함하고 있다. 따라서 OLDEN 벤치마크를 병렬 컴퓨터가 아닌 환경에서 수행시키기 위해서는 매크로를 삭제하는 클리닝 작업이 필요하다. 각 응용 프로그램의 특성은 다음과 같다.

TSP는 유명한 출장 판매원 문제(traveling salesman problem)를 분할 알고리즘과 최 근접지점 휴리스틱(heuristic)을 통하여 풀어내는 프로그램이다. 이 벤치마크는 균형 잡힌 이진 트리(balanced binary tree) 자료구조를 사용하고 있다. Health는 콜롬비아 병원의 건강관리 시뮬레이션 프로그램으로 양방향 리스트(double-linked list)를 이용하고 있다. Perimeter는 퀴드 트리(quad tree) 자료구조를 이용하여 이미지 영역에서의 경계선을 구하는 프로그램이다. 그리고 MST는 잘 알려진 그래프의 최소 신장 트리(minimum spanning tree)를 구현한 벤치마크이다.

이 프로그램들은 대체로 작고 사용자와의 최소한의 상호작용만으로 많은 데이타를 이용하여 자동으로 특정 작업을 반복 수행하는 경향이 있다. 여기서 사용되는 자료구조는 대부분 구조체로, 리스트나 트리를 구성하기 위해 사용되며, 배열은 거의 사용되지 않는다. 이러한 이유로 OLDEN 벤치마크는 데이타 구조체 배치관련 최적화 기법이나 타입 관련 연구 등에 자주 사용되고 있다.

모든 실험은 펜티엄4 - 2.6GHz 프로세서와 1 기가바이트의 주 메모리(main memory)를 탑재한 시스템에서 수행되었다. 이 시스템은 8KB 4-way 데이타 L1 캐시와 512KB 8-way의 L2캐시를 가지고 있고, 캐시의 line size는 64 byte 이다. 또한 모든 벤치마크 프로그램은 고수준 최적화 옵션인 O3와 함께 GNU-C 컴파일러를 이용하여 컴파일 되었다.

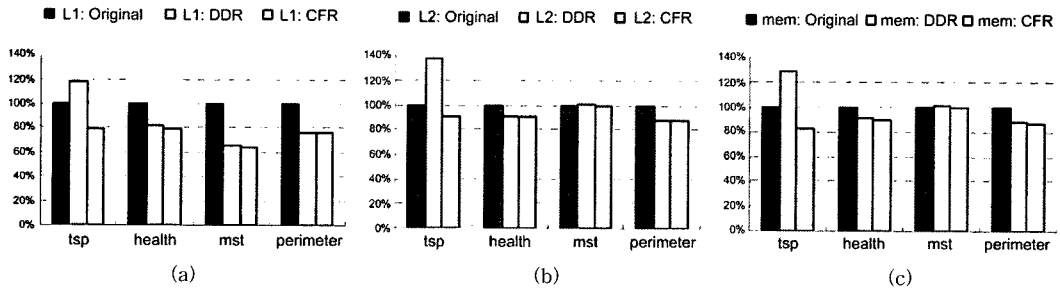
4.2 실험 결과

성능 저하와 에너지 소비를 증가시키는 가장 큰 요인 중 하나가 메모리 지연의 발생이다. 메모리 지연의 정도는 캐시 실패를 측정함으로써 예측할 수 있다. 표 1은 여러 벤치마크에 대한 레벨 1 캐시와 레벨 2 캐시 실패, 그리고 메모리 사용량을 보여준다. 멀티 프로그램 환경에서 실제 캐시 실패율을 측정하는 것은 수행 환경에 따라 오차가 크게 발생할 수 있으므로, 모든 수치는 유명한 시뮬레이션 프로그램인 심플 스칼라(Simple Scalar)에 의해 3번 이상 측정되었고, 가장 좋은 결과가 선택되었다. 첫 번째 칼럼은 사용된 벤치마크를 나타내고, 두 번째 칼럼은 입력 값을 나타낸다. 입력 값이 커질수록 사용되는 메모리의 양이 많아지고, 캐시 실패도 많아지므로 다양한 결과를 얻고자 여러 입력결과에 대한 실험을 수행하였다.

그림 7은 표 1의 결과를 보기 좋게 재구성 한 것이다.

표 1 L1과 L2 데이타 캐시 실패 수와 할당된 총 페이지 크기 비교

Bench marks	Input	Original			DDRemap			CFRemap		
		L1 misses	L2 misses	Mem(kb)	L1 misses	L2 misses	Mem(kb)	L1 misses	L2 misses	Mem(kb)
Tsp	4096	5678	6171	756	7761	8271	888	5203	5709	728
	16384	21033	21463	1716	29587	29789	2236	19115	19558	1016
	500000	2398123	328742	20920	1884191	461633	29228	1357646	296596	19508
health	6,500	36260780	200891	12988	28147850	174175	11320	27121098	173246	11264
	10,20	30618246	2457801	154048	26158740	2326307	145828	25701759	2314565	145096
mst	2048	19995826	853294	13744	6691827	862162	13880	6904696	853682	13748
	3000	42330914	1830223	115512	41022807	1849247	116700	39533743	1831051	115564
perimeter	11	4639289	1548484	97144	3540019	1362930	85548	3503828	1357631	85216
	12	10395273	3460685	216656	7932706	3045931	190732	7850267	3034082	189996
avg. improve (%)		.	.	.	+10.89%	-8.08%	-5.27%	+24.80%	+7.81%	+10.54%



(a) (b) (c)

그림 7 (a) 평균 L1 캐시 실패 수 비교 (b) 평균 L2 캐시 실패 수 비교 (c) 할당된 총 페이지의 평균적 크기 비교

재배열 하지 않았을 경우의 실험 결과를 100%로 보았을 때, DDRemap과 CFRemap의 계산식을 이용하여 재배열 한 경우의 상대적 비교를 보여주고 있다. Tsp의 경우를 제외하고는 DDRemap 식을 통하여 재배열된 경우에 L1 캐시 실패는 평균 25.66%, L2 캐시 실패는 평균 6.76% 감소하였다. CFRemap을 이용하여 재배치한 경우에는 L1 캐시 실패의 경우 재배치하지 않은 경우보다 평균 27.05% 줄어들었고, L2 캐시 실패의 경우에는 평균 7.36% 줄어들었다. 이는 항목 재배치가 캐시 실패율을 줄이는데 도움이 된다는 것을 나타내고, DDRemap보다 CFRemap을 이용하는 것이 더 효과적임을 나타내고 있다. DDRemap을 이용한 경우와 CFRemap을 이용한 경우의 차이는 미미해 보일 수 있으나 이는 DDRemap에서 발생할 수 있는 성능 저하의 경우를 제외하였기 때문이다. 구조체 항목들의 크기가 다를 때, DDRemap을 이용한 재배치는 캐시 메모리 낭비로 인해 캐시 실패율이 증가된다. 표 1과 그림 7에서 응용프로그램 tsp의 결과가 이를 뒷받침한다. tsp는 트리 구성을 위해 7개의 항목을 갖는 구조체를 정의하고 사용한다. 각 레코드의 크기는 36바이트이나, DDRemap을 통해 재배열할 경우, 각 항목은 최대 항목 크기인 8바이트에 맞추어 재배열되므로, 하나의 레코드가 차지하는 메모리 공간은 56 바이트가 된다. 프로그램 수행 시의 입력 값이 4096일 때, 할당되는 메모리 페이지의 총 크기는 888 킬로바이트로 재배치하지 않은 경우에 비해 약 17% 증가하게 된다. 이로 인해 L1 캐시 실패는 재배치하지 않은 경우 보다 오히려 37% 증가하고, L2 캐시 실패의 경우에도 약 34% 증가하게 된다.

그러나 CFRemap을 이용하여 재배치한 경우에는 재배치하지 않은 경우보다 L1캐시 실패가 약 9%, L2 캐시 실패가 약 8% 감소하게 된다. CFRemap을 이용하여 재배치 한 경우에는 메모리 낭비가 발생하지 않아 캐시의 효율을 떨어트리지 않는다. 실제로 할당되는 총 페이지 메모리 크기는 728 킬로바이트로서 재배치하지 않은 경우보다 약 4.7% 가량 줄어든 결과를 얻을 수 있었다.

각 어플리케이션에 할당되는 총 페이지 크기는 CFRemap을 이용한 경우 메모리 풀을 이용하였기 때문에 재배치하지 않은 경우보다 실제로 할당되는 페이지 크기가 평균 10.54% 줄어들게 된다. 역시 메모리 풀을 이용하는 DDRemap의 경우에는 필드 크기가 다를 때 오히려 사용하는 메모리 양이 증가하는 경우가 발생하여 실험에서는 평균 5.3% 늘어났으나, tsp의 경우를 제외할 경우 평균 6.68% 감소하게 된다.

표 2는 다양한 입력 값에 따른 각각의 벤치마크를 수행하는데 걸린 시간을 측정한 것이다. 실제 수행시간은 데이타 크기에 따라 차이가 많이 발생한다. 너무 작은 양의 데이타만으로 실행할 경우에는 주소계산을 위한 명령 개수의 증가로 성능차이가 거의 나타나지 않는다. MST 프로그램에 3000이라는 입력 값을 주고 수행시켰

표 2 실행시간 비교와 재배치하지 않은 경우에 대한 개선율

Benchmarks	Input	Origin (sec)	DDRemap (sec/improve %)		CFRemap (sec/improve %)	
tsp	5*105	2.96	2.76	+6.76%	2.77	+6.42%
	5*106	50.57	46.85	+7.36%	46.11	+8.82%
	2*107	881.93	1421.95	-61.23%	649.77	+26.32%
health	10,20	3.97	3.73	+6.05%	3.64	+8.31%
	11,20	15.36	14.49	+5.66%	13.93	+9.31%
mst	3000	6.70	7.00	-4.48%	6.89	-2.83%
	10000	731.25	716.49	+2.02%	712.57	+2.55%
perimeter	11	2.57	1.99	+22.57%	1.964	+23.61%
	12	765.20	618.82	+19.13%	616.19	+19.47%
avg. improve (%)		-	-	+0.43%	-	+11.33%

을 때 이와 같은 현상이 나타난다. 반대로 너무 큰 데이타를 이용하여 메모리 크기를 벗어날 경우, 디스크와의 스왑 현상으로 심 한 성능저하가 발생한다. 따라서 실험적 결과를 바탕으로 메모리 크기 이내의 충분히 큰 입력 값을 이용하였고 3번 이상 측정 후 최소값을 기록하였다. 첫 번째 칸과 두 번째 칸은 사용된 프로그램과 사용된 데이타의 양을 결정하는 입력 값이다. 세 번째 칸(original)의값은 재배치를 하지 않았을 경우의 수행시간이고 네 번째 칸의 값은 DDRemap 을 이용하여 재배치한 경우의 수행시간이다. 여섯 번째 칸의 값은 CFRemap 을 이용하여 재배치한 경우의 수행시간을 의미하고, 다섯 번째 칸과 일곱 번째 칸은 재배치하지 않았을 때 소요된 수행시간을 100%라고 간주했을 때의 상대적 수행시간의 차이를 나타낸 값으로 0보다 큰 값은 재배치를 하지 않은 경우보다 더 빨라졌음을 의미한다. 전반적인 결과는 DDRemap을 이용하여 재배치한 경우와 CFRemap을 이용하여 재배치한 경우의 성능개선차이는 크지 않다. 다만 tsp의 경우에서처럼, 항목간의 크기가 달라 항목 사이에 사용되지 않는 영역이 발생으로 캐시의 효율성이 떨어지는 DDRemap의 경우와 달리 CFRemap을 사용했을 때는 성능저하가 발생하지 않음을 볼 수 있다. 즉, 사용되는 데이타의 양이 작지 않을 때 DDRemap을 사용한 경우에 준하는 성능개선을 이루고, 격심한 성능개선차이는 보이지 않게 된다. 따라서 CFRemap을 이용한 경우에 수행속도가 평균 11.33% 빨라지는 결과를 얻을 수 있었다.

5. 결 론

이 논문에서 우리는 낭비되는 메모리를 줄이고 캐시 실패를 줄이기 위해 동적 레코드 데이타를 재배치하는 개선된 알고리즘을 제안하였다. 알고리즘은 재배치하고자 하는 데이타 형식을 찾아내고 동적으로 할당된 데이타를 재배치하는 과정을 거치게 된다. 여러 벤치마크를 통하여 우리가 제안한 재배치 알고리즘이 이전의 캐시 실패를 줄이기 위한 방법들에 비해 L1캐시 실패의 경우 평균 13.9%, L2 캐시 실패는 평균 15.9% 줄이는 결과를 얻었다. 뿐만 아니라 응용프로그램의 실제 수행 시간도 재배치하지 않은 경우보다 평균 11.33% 빠른 결과를 얻었다.

본 연구의 한계성으로는 제안한 알고리즘이 재배치된 데이타에 대한 주소계산복잡도가 증가하여 전체 수행 명령의 수가 증가할 수 있다는 것이다. 데이타 캐시 실패의 감소로 얻을 수 있는 성능상의 이득과 주소 계산으로 인해 추가된 명령어 사이의 trade-off로 수행시간은 다양성을 보여준다. 따라서 주소 계산복잡도를 좀 더 낮추거나 늘어나는 명령의 수에 비해 데이타 캐시 실패율을 더욱 줄일 수 있는 연구가 앞으로 고려되어야 할 것이다.

참 고 문 헌

[1] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkrani, A. Vandercappelle, and P. Kjeldsberg. Data and memory optimization techniques for embedded systems. ACM Transactions on Design Automation of Electronic Systems, Vol.7, No.2, pp.149-206, April 2001.

[2] P. Panda, N. Dutt, and A. Nicolau. Memory Issues In Embedded Systems-On-Chip : Optimizations and Exploration. Kluwer Academic Publishers, 1999.

[3] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. 1986. In Proceedings of the ACM SICPLAN Conference on Programming Language Design and Implementation, pp.1-12, May 1999.

[4] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. ACM transactions on Programming Languages and Systems, Vol.22, No.3, pp.490-505, May 2000.

[5] P. Panda, N. Dutt, and A. Nicolau. Memory data reorganization for improved cache performance in

embedded processor applications. ACM Transactions on Design Automation of Electronic Systems, Vol.2, No.4, pp.384-409, 1997.
[6] D. N. Truong, F. Bodin, and A. Seznec, Improving cache behavior of dynamically allocated data structures. Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, p.322, 1998.
[7] T. Chilimbi, B. Davison, and J. Larus. Cache-Conscious Structure Definition. Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation, pp.13-24, 1999.
[8] R. M. Rabbah and K. V. Palem. Data Remapping for Design Space Optimization of Embedded Memory Systems. ACM Trans. On Embedded Computing Systems., Vol.2, No.2, pp.186-218, May 2003.

김 정 은

2003년 숭실대학교 컴퓨터학부 학사. 2005년 한국과학기술원 전산학 석사. 2005년~현재 삼성전자 기술총괄 시스템 연구소 연구원. 관심분야는 메모리 최적화, 플래시 메모리

한 환 수

1993년 서울대학교 컴퓨터공학 학사. 1995년 서울대학교 컴퓨터공학 석사. 2001년 University of Maryland, Computer Science PhD. 2001년~2002년 Intel, Architecture Group, Senior Engineer 2003년~현재 한국과학기술원 전산학과 조교수. 관심분야는 모바일 소프트웨어, 임베디드 시스템, 컴파일러, 컴퓨터구조