데이터 재구성 기법을 이용한 고성능 FFT

박 능 수[†] · 최 영 호[††]

요 약

대규모 신호처리 변환을 신속하게 처리하기 위해서는 캐시 메모리를 효과적으로 이용하는 것이 중요하다. 대규모 DFT 계산에서는 stride 액세스로 인한 캐시 충돌 적중 실패로 인하여 캐시 성능이 상당히 떨어지게 되고 이로 인해 전체적인 성능이 저하하게 된다. 본 논문에서는 메모리 계층 구조를 고려한 동적 데이터 재배열(Dynamic Data Layout) 방법을 개발하였다. 제시된 방법은 stride를 가지는 계산 단계(computation stage) 사이에 데이터를 동적으로 재구성을 하여 캐시 적중 실패를 줄이는 것이다. 또한 트리 구조 FFT 계산 방법에서 FFT 크기와 데이터 stride 액세스를 기초로 하여 가능한 모든 인수분해 트리 중에서 최소 실행시간을 가지는 최적의 인수 분해트리를 찾아내는 탐색 알고리즘을 개발하였다. 성능 향상을 확인하기 위하여 제시된 방법을 기존의 FFT 알고리즘에 적용하여 Pentium 4, Alpha 21264, Athlon™ 64, UltraSPARC III에서 실험하였다. 실험 결과에 따르면 기존의 FFT 패키지들과 비교하여 제시된 방법을 적용한 FFT가 최대 3.37배의 성능 향상을 얻을 수 있었다.

키워드 : 동적 데이터 배열, 캐시, 캐시 적중 실패, FFT, 메모리 계층구조

High-Performance FFT Using Data Reorganization

Neungsoo Park[†] · Yungho Choi[††]

ABSTRACT

The efficient utilization of cache memories is a key factor in achieving high performance for computing large signal transforms. Nonunit stride access in computation of large DFTs causes cache conflict misses, thereby resulting in poor cache performance. It leads to a severe degradation in overall performance. In this paper, we propose a dynamic data layout approach considering the memory hierarchy system. In our approach, data reorganization is performed between computation stages to reduce the number of cache misses. Also, we develop an efficient search algorithm to determine the optimal tree with the minimum execution time among possible factorization trees considering the size of DFTs and the data access stride. Our approach is applied to compute the fast Fourier Transform (FFT). Experiments were performed on Pentium 4, Athlon™ 64, Alpha 21264, UltraSPARC III. Experiment results show that our FFT achieve performance improvement of up to 3.37 times better than the previous FFT packages.

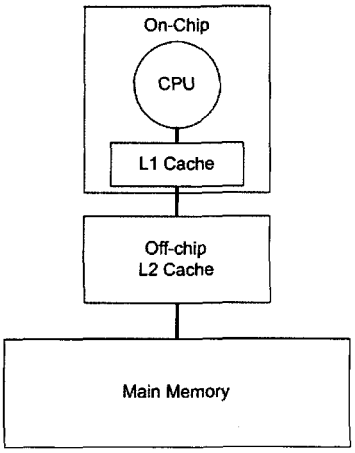
Key Words : Dynamic Data Layout, Cache, Cache Miss, FFT, Memory Hierarchy

1. 서 론

고속 푸리에 변환 (FFT: Fast Fourier Transform)은 많은 신호처리 응용에 중요한 계산 단계 중에 하나이다[8,9]. 이러한 FFT를 효율적으로 계산하기 위하여 여러 플랫폼에서 알고리즘적인 기술 개발이 이루어져 왔다. 대부분 이러한 알고리즘은 플로팅 포인트 연산 횟수를 줄임으로서 FFT의 계산효율을 높여왔다. 또한 DFT 계산 특징과 현대 시스템의 메모리 구조 특성을 고려하여 계산 성능을 향상시키고자 노력을 해오고 있다[4, 5, 11]. 최근에는 카네기멜론대학교(CMU)와 MIT에 의해 FFT를 트리 구조로 표현하여 구현된 소프트웨어 패키지들이 개발 되었다[4, 6]. 이러한 트리 구조 FFT는 트리의 말단 노드 (leaf node) 계산을 조건 분기문(conditional branch)을 사용하지 않은 직선 코드(straight line unrolled code)로 이루어져 상당히 높은 최적화를 얻었다. 전체적으로 작은 크기의 FFT에서는 좋은 성능 향상을 얻어 내었다. 하지만, FFT의 크기가 커져 갈수록 계산 성능이 급격히 떨어졌다. 이러한 성능하락은 현대 컴퓨터 시스템의 메모리 구조에 그 원인이 있다.

현대 컴퓨터 시스템에서는 프로세서와 메모리 사이의 전송 속도 차이가 고성능 계산 능력을 얻는데 장애가 되고 있다. 이러한 장애를 극복하기 위하여 (그림 1)과 같이 프로세서와 메모리 사이에 캐시(cache) 메모리를 두어 응용 프로그램의 참조(reference)에 시간적 또는 공간적 집약성(locality)

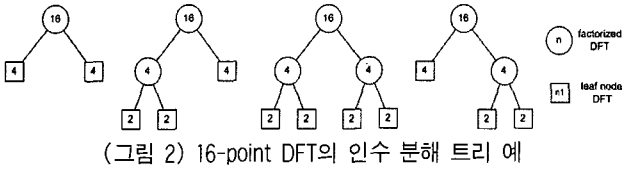
※ 이 논문은 2004년도 건국대학교 학술진흥연구비 지원에 의한 논문임
† 종신회원 : 건국대학교 컴퓨터공학부 교수(주저자)
†† 정 회 원 : 건국대학교 전기공학과 교수(교신저자)
논문접수 : 2005년 3월 8일, 심사완료 : 2005년 4월 27일



(그림 1) 메모리 계층구조 시스템

을 이용하여 고성능을 얻고 있다. 일반적으로 현대 컴퓨터의 캐시 구조는 direct-mapped 또는 작은(2-way 또는 4- way를 의미함) set-associative로 이루어져 있다. 이러한 구조에서는 일정한 거리를 가지는 연속적인 데이터 액세스, 즉 stride 액세스는 응용의 참조에서 공간적 집약성을 잃게 하여 캐시 적중률을 낮추게 된다. 특히 direct-mapped이나 작은 set-associative 캐시에서는 응용 계산 시에 몇 개의 데이터가 캐시 안의 같은 라인에 사상(mapping)이 되어 생기는 충돌 적중 실패(conflict miss) 때문에 성능이 급격하게 하락하게 된다[2]. 이러한 stride 액세스가 트리구조 FFT 계산에 발생하게 되어, 큰 FFT의 계산에서 캐시의 적중 실패를 급격히 증가시키고, 전체적인 성능을 하락시키게 된 것이다.

본 논문에서는 캐시의 효율을 높이기 위하여 계산 중간에 메모리 안에 데이터를 동적으로 데 구성하는 동적 데이터 재배열(Dynamic Data Layout: DDL) 방법을 제안한다. 일정한 거리를 가지는 비연속적인 데이터 액세스를 연속적인 데이터 액세스로 전환하는 데이터 재구성을 함으로서 응용 프로그램의 공간적 집약성(spatial locality)을 증가시켜 캐시 적중 실패를 줄일 수 있다. 따라서 트리구조의 FFT 계산 단계 중간에 이러한 데이터 재구성을 적용하여 프로세서와 메모리 사이의 캐시를 효율적으로 이용하여 전체적인 성능 향상을 얻을 수 있다. 하지만 데이터 재구성 자체는 메모리 안에 데이터를 처리하는 연산이므로 하나의 연산 오버헤드가 된다. 따라서 재구성에 의한 오버헤드가 얻어지는 이득보다 작아야 전체적인 성능 향상을 얻을 수 있다. 일반적으로 작은 FFT 크기에서는 이득이 작아 성능 향상을 얻기 힘들지만, FFT 크기가 커질수록 그 성능 이득이 커져서 전체적인 성능 향상을 얻을 수 있다. 또한 트리구조 FFT에서는 어떠한 트리구조를 선택하느냐에 따라 그 성능이 달라진다.



(그림 2) 16-point DFT의 인수 분해 트리 예

따라서 최소 실행시간을 가지는 트리를 찾아내는 것이 성능 향상을 얻는데 중요하다. 기존의 방식에서는 이러한 트리를 선택하는데 각 노드의 크기만을 고려하였다. 하지만 본 논문에서 제시된 방식에는 노드의 크기뿐만 아니라 각 노드에서의 stride 액세스도 중요한 요소이다. 따라서 트리구조 FFT에서 어느 시점에 데이터를 재배열하여 데이터 액세스에 변화를 줄지 결정하는 것은 전체적인 성능 향상을 이루기 위하여 중요하다. 본 논문에서는 데이터 재구성을 포함하는 최적의 트리구조를 찾기 위하여 dynamic programming 기법을 이용한 탐색 알고리즘을 개발하였다. 성능 향상을 확인하기 위하여 제시된 DDL 방법과 탐색 알고리즘을 기존의 FFT 알고리즘에 적용하여 FFT 패키지를 개발하여 Alpha 21264, Pentium 4, Athlon™ 64, UltraSPARC III에서 실험하였다. 실험을 통하여 새로 구현된 FFT 패키지로 최대 3.37배의 성능 향상을 얻을 수 있었다.

본 논문은 다음과 같이 구성되어 있다. 제2장은 간단히 DFT 분할 알고리즘에 대하여 설명하고, 제3장은 트리구조 DFT에 stride 액세스가 캐시에 미치는 영향을 설명한다. 제4장은 본 논문에서 제시한 FFT를 위한 동적 데이터 재구성 기법에 관하여 기술한다. 제5장은 그 실험 결과를 제시하고 성능을 평가하고, 제6장은 본 논문의 결론을 내린다.

2. DFT 분할법

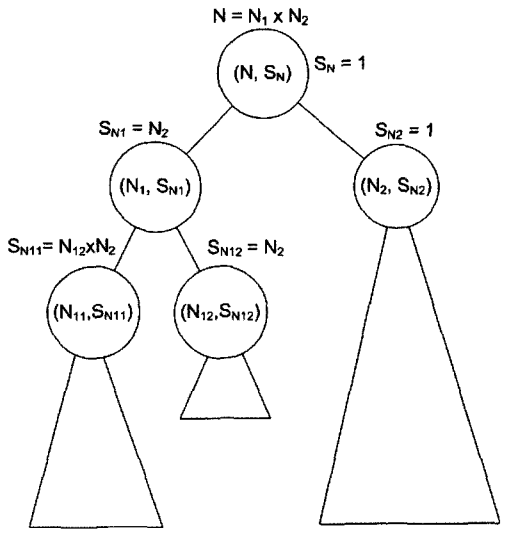
Discrete Fourier Transform(DFT)의 근본적인 divide-and-conquer 특성을 이용하여, 변환 행렬을 희소 행렬(sparse matrix)의 곱으로 인수 분해하여 아주 체계적으로 표현할 수 있다. 즉, $N\times N$의 항등행렬 (identity matrix: $I(N)$)와 작은 크기의 변환행렬의 텐서 곱 (tensor product) 형태의 간단한 수식형태로 표현을 할 수가 있다. 한 예로, 일반적은 Cooley- Tukey DFT[10]로 다음과 같이 표현할 수가 있다.

$$DFT(N) = L(N, N_2) \cdot (I(N_2) \otimes DFT(N_1)) \cdot T(N, N_1) \cdot (DFT(N_2) \otimes I(N_1)) \quad (1)$$

(1)

(2)

여기서, $N = N_1 \times N_2$, $DFT(N)$은 크기가 N인 DFT, $\otimes$은 텐서 곱의 연산자, $L(N, N_1)$은 stride permutation 행렬이고, $T(N, N_1)$은 twiddle 행렬을 나타낸다[14]. 앞의 인수분해법을 작은 신호 변환 행렬에 반복적으로 적용할 수가 있다. 이렇게 하여 인수 분해된 연결 사슬을 하나의 트리 형태로 표현하고[6], 이를 인수 분해 트리 (factorization tree)라 한다. 주어진 변환에 대하여, (그림 2)에 표현된 것과 같이 적용된 인수분해에 따라서 다른 트리를 얻을 수 가 있다. 본 논문에서 트리에 중간 노드는 인수 분해가 된 FFT 행렬을 의미하여 인수분해 노드라 명하고, 리프 노드 (leaf node), 즉 자식 노드가 없는 노드는 더 이상 인수 분해를 하지 않는 노드를 말한다. 따라서 트리에 루트(root) 노



(그림 3) Cooley-Tukey 알고리즘이 반복 적용된 인수 분해 트리

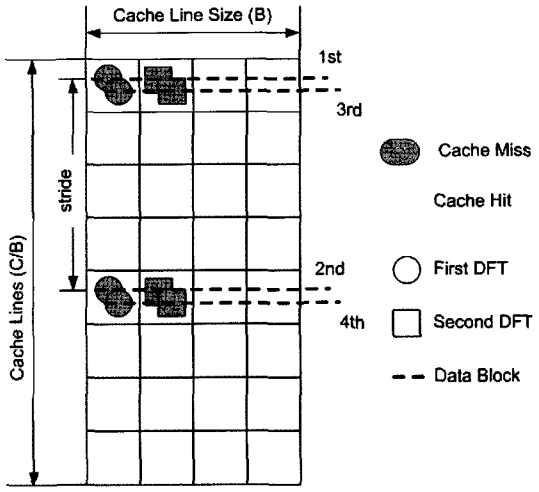
드로 표현되는 하나의 FFT는 리프 노드와 자식 노드를 가지는 중간 노드의 결합으로 묘사하는 divide-and-conquer방법으로 구현이 될 수 있다.

이러한 divide-and-conquer방법은 인수분해 트리의 리프 노드 크기를 작게 하여 계산의 working set 크기를 작게 한다. 그러므로 리프 노드 계산에 필요한 데이터가 캐시 안에 다 올라올 수 있어, 그 계산 성능 향상에 도움을 준다. MIT의 FFTW[4]와 CMU의 FFT[6] 패키지가 divide-and-conquer 특성을 이용하였다. 이러한 알고리즘은 캐시의 크기만을 고려하고 캐시 구조의 특징을 고려하지 않았다. 따라서 알고리즘 구현에 있어서 문제의 크기만을 고려하고 데이터 액세스 패턴은 고려하지 않았다.

3. 트리구조 DFT에 stride 액세스의 영향

인수 분해된 DFT의 스트라이드 데이터 액세스가 캐시 성능에 미치는 영향을 이해하기 위한 간략한 분석을 하였다. 먼저, 식 (1)와 같이 N-point DFT를 Cooley-Tukey 알고리즘을 이용하여 $N_1 \times N_2$로 분해하자. 이때 계산은 먼저 N_2 개의 N_1-point DFT를 계산한다. 이때 N_1-point DFT는 N_2 stride 데이터 액세스를 한다. 다음에 twiddle 곱셈을 한 후에, N_1개의 stride가 1인 N_2-point DFT를 계산한다. 이러한 Cooley-Tukey 알고리즘을 자식 노드에 반복적으로 적용함으로써 divide-and-conquer 방법의 트리구조 계산을 하게 된다. 이러한 트리구조 계산에서는 리프 노드 크기가 점점 작아져서 캐시 크기보다 작게 되고 (그림 3)에서와 같이 리프 노드 계산에 stride 액세스가 포함된다.

인수 분해된 DFT의 stride 액세스가 캐시 성능에 미치는 영향을 분석을 하기 위해 먼저 2-level 메모리 계층 구조를 생각하자. C를 캐시 크기 그리고 B를 캐시 라인(cache line) 크기라 가정하자. 최신 컴퓨터가 direct-mapped 또는 작은



(그림 4) 두 개의 연속적인 리프 노드 DFT의 캐시 동작

set-associative 캐시를 가지지만, 동작 분석을 간략하게 하기 위하여 direct-mapped 캐시라 가정하자. 데이터 액세스는 캐시 적중(cache hit) 또는 적중 실패(cache miss)를 발생시킨다. 캐시 적중 실패는 필수 적중 실패(compulsory miss) 또는 충돌 적중 실패(conflict miss)로 분류된다. 필수 적중 실패는 처음 데이터가 결코 액세스된 적이 없어 처음 인출(fetch)될 때 발생한다. 충돌 적중 실패는 이전에 캐시에 전달된 데이터 블록과 새로 인출되는 데이터 블록이 캐시 내에 같은 라인으로 사상이 되어 새로운 블록으로 대체될 때 발생하게 된다. 일반적으로 캐시 적중 실패는 메모리에 데이터를 캐시로 전달하기 위하여 실행시간이 긴 메모리 연산을 필요로 한다. 그러므로 여러 개의 캐시 적중 실패는 심각한 성능 저하를 야기할 수 있다.

(그림 3)에서 리프 노드의 stride가 커서 $stride \times N_1 > C$가 되는 경우를 생각해보자. 먼저 캐시 크기 $C = 32$포인트 이고 캐시 라인 $B = 4$ 포인트인 direct-mapped 캐시를 가정하자. 만약 $N_1 = 4$이고 $N_2 = 16$라 가정하면, N_1-point DFT의 데이터가 (그림 4)에서처럼 캐시에 사상하게 된다. 이때 첫 번째와 세 번째 그리고 두 번째와 네 번째 데이터를 포함하는 데이터 블록이 각각 같은 캐시 라인에 사상이 된다. 그러므로 첫 DFT를 계산하는데 캐시 충돌 적중 실패가 발생한다. 더욱이 두 번째 DFT 계산을 시작할 때 첫 번째와 두 번째 데이터 블록을 다시 액세스하게 된다. 그러나 이 블록들은 이미 적중실패에 의해서 세 번째와 네 번째 블록으로 대체되어 다시 충돌 적중 실패를 일으킨다. 이와 같이 데이터가 충분히 사용되기 전에 캐시 라인이 대체되는 캐시 오염(pollution)이 발생하게 되는 것이다.[1] 이러한 캐시 오염과 충돌 적중 실패는 다음의 연속적인 리프 노드 DFT 계산에 영향을 주어서 심각한 성능 저하를 야기한다.

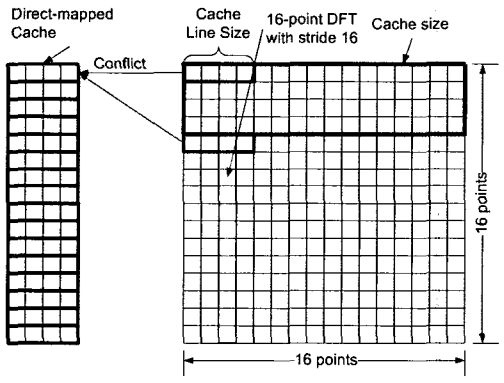
4. FFT를 위한 동적 데이터 재구성

4.1절에서는 동적 데이터 재구성에 관한 설명하고, 4.2절

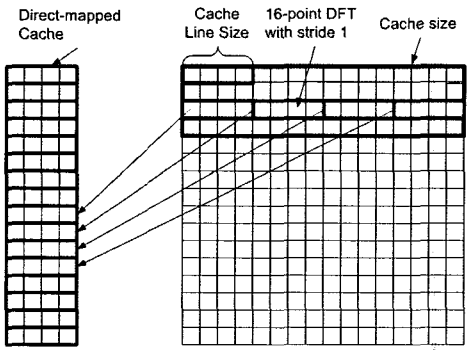
에서는 동적 데이터 재구성 기법을 적용한 FFT 트리에 대하여 논하였다. 그리고 4.3절에서 동적 데이터 재구성 기법을 사용한 최적화 FFT 트리를 찾는 탐색 알고리즘에 관하여 설명하였다.

4.1 동적 데이터 재구성

2장에서 논한 것처럼, 큰 DFT는 인수 분해를 반복적으로 적용하여 인수 분해 트리로 표현을 할 수가 있다. (그림 3)에서와 같이, 이 트리에서 각 노드는 일정한 같은 거리의 데이터를 액세스하는 작은 크기의 DFT로 표현이 된다. 리프 노드 계산 시에 이러한 stride 액세스는 캐시 충돌 적중 실패 또는 오염을 야기하여, 성능 저하를 일으킨다. 이러한 성능 저하는 데이터 액세스 패턴과 메모리 안에 데이터의 배열 사이에 불합치에 의하여 야기될 수 있다. 특히 이러한 불합치는 리프 노드에 계산에 있어서 stride 액세스와 메모리 내의 데이터 배열에서 발생하여 전체적인 성능 저하를 일으키는 것이다. 따라서 인수분해법에 의한 DFT계산의 성능을 향상시키기 위하여, 계산의 데이터 액세스 패턴에 맞추어서 데이터 배열은 동적으로 재구성할 필요가 있다. 이러한 방법을 동적 데이터 배열(***Dynamic Data Layout (DDL)***) 방법이라 한다. 하지만, 이러한 동적 재배열 방법에 의해 계산을 한 후에, 데이터를 필요에 따라 본래의 순서로 복구를 해주는 역재배열을 실행하여야 한다.



(a) DDL 적용하기 전



(b) DDL 적용한 후

(그림 5) 16×16 인수분해에 대한 데이터 액세스 패턴

DDL방법의 효과를 설명하기 위하여, 256-point DFT의 간단한 예를 보자. 이를 Cooley-Tukey 알고리즘을 사용하여 16×16으로 분해하였다. (그림 5)를 보면, 16×16 데이터 포인트들은 행 중심 순서(row-major order)에 따라서 배열이 된다. 효과적인 설명을 위하여, 캐시의 크기는 64 포인트이고 캐시 라인의 크기는 4인 direct-mapped 캐시라 가정을 하자. (그림 5) (a)에서처럼 stride가 16인 16-point DFT를 생각해 보자. 16-point DFT의 매 네 번째 데이터 포인트는 같은 캐시 라인에 사상이 되어 캐시 충돌 적중 실패를 발생시킨다. 그러나 (그림 5) (b)에서처럼 데이터 배열을 재구성한 후에는 모든 데이터가 캐시 안에 사상이 되어 어떠한 캐시 충돌 실패를 야기하지 않는다. 그러므로 계산중에 캐시 적중 실패를 줄이고 또한 각 캐시 라인을 충분히 사용하여 전반적인 성능향상을 얻을 수가 있다.

4.2 동적 데이터 재구성에 따른 DFT 인수 분해 트리

인수분해 트리에서는 이러한 DDL 방법을 어느 노드에나 적용일 시킬 수 있다. 그러나 데이터 재구성은 메모리 액세스를 필요로 하기 때문에 성능 향상 측면에는 또 다른 오버헤드(overhead)가 된다. 그러므로 모든 노드에 데이터 재구성 기법을 적용하는 것은 오버헤드의 증가로 인해 비효율적이다. 따라서 전체적인 성능 향상을 위해서는 재구성 회수를 가능하면 최소화해야 한다. 또한 트리구조 FFT에서는 어떠한 트리구조를 선택하느냐에 따라 그 성능이 달라진다. 따라서 최소 실행시간을 가지는 트리를 찾아내는 것이 성능 향상을 얻는데 중요하다. 기존의 방식에서는 이러한 트리를 선택하는데 각 노드의 크기만을 고려하였다. 3장에서 설명한 것처럼 리프 노드의 크기뿐만 아니라 각 노드에서의 stride 액세스도 성능에 중요한 영향을 준다. 이러한 stride는 데이터 배열에 따라 변하게 되므로 트리구조 FFT에서 어느 시점에 데이터를 재배열하여 데이터 액세스에 변화를 줄지 결정하는 것은 전체적인 성능 향상을 이루기 위하여 중요하다. 따라서 성능을 최대화하기 위해서 어떤 인수분해 트리에서 어떤 노드에서 재구성을 해야 할지 결정할 필요가 있다.

DDL 기법에서는 (그림 3)에 묘사되듯이 인수 분해 트리는 노드의 크기와 stride성분으로 표현이 된다. 데이터 배열이 변하게 되면 노드에 데이터 액세스 stride는 변하게 된다. 특정 인수 분해에 대해서도 다양한 데이터 배열이 적용될 수 있고 그에 따라 다양한 데이터 액세스 stride도 변하게 된다. 따라서 FFT 계산 성능을 최대화하기 위해서는 모든 가능한 트리 중에서 최소의 실행시간을 가지는 것을 찾는 탐색 알고리즘을 필요로 한다.

탐색 알고리즘을 위해서 실행 시간을 기초로 하는 N-point DFT에 비용 모델(cost model)을 정의할 필요가 있다. 먼저 stride S를 가지는 N-point DFT를 고려해 보고 이것의 계산 비용을 $DFT(N, S)$이라 표시하자. 주어진 N의 인수 분해가 $N_1 \times N_2$이라면 데이터 재구성을 고려한 Cooley-Tukey 알고리즘을 계산하기 위한 최소 비용은 다음과 같이

```
/* To find an optimal factorization tree for a 2^i-point DFT */
FindOptTree (int i, tree *OptTree)
{
        minTime = Infinite
        for j = 0 to i-1 /* i different factorizations */
                for l = 0 to k-1 /* k different strides */
                        S_l /* l^th different stride of the left subtree*/
                        Tree = GenerateTree(OptTree[j],OptTree[i-j], S_l)
                        Time = FFT_Measure(Tree)
                        if (Time < minTime)
                                OptTree[i] = Tree
                                minTime = Time
                        end if
                end for
        end for
}
```

(그림 6) 동적 데이터 배열을 고려한 탐색 알고리즘

주어진다.

$$\min_{S_i, S_j} [Dr(N, L_{S_i}, L_{S_j}) + N_2 \times DFT(N_1, S_j) + Dr(N, L_{S_j}, L_{S_i}) + N_1 \times DFT(N_2, S_i)] \quad (2)$$

여기서, S_i와 $S_j = 1,2,\cdots$ 이고, $L_{S_i}(L_{S_j})$는 $N_1(N_2)$-point DFT의 계산할 때에 stride가 $S_i(S_j)$ 되도록 하는 데이터 배열을 의미한다. $Dr(N, L_{S_i}, L_{S_j})$는 크기가 N인 데이터 배열을 L_{S_i}에서 L_{S_j}로 재구성하는데 드는 비용이다. 만약 $L_{S_i} = L_{S_j}$이라면, 데이터 재구성은 필요 없게 된다. 데이터를 재구성하는데 드는 비용은 크기가 N인 데이터 배열에 대해 $O(N/B)$의 메모리 액세스 비용을 필요로 한다. 만약 각 노드에서 k 개의 다른 stride를 고려한다면 k 개의 다른 데이터 배열을 필요하게 되므로, 수식 (2)을 평가하는 비용은 $O(k^2)$이 된다.

4.3 최적화 트리 탐색 알고리즘

주어진 N-point DFT는 많은 인수 분해 트리를 가지게 된다. 따라서 DFT 계산 성능을 최적화하는 것은 그 수많은 트리 중에서 최소 실행시간을 가지는 최적화 트리를 찾는 탐색 문제가 된다. 기존의 탐색 알고리즘은 DFT의 크기만을 고려하였고 그 탐색 공간은 $\Theta(5^n/n^{3/2})$이다. 따라서 제시한 알고리즘은 DFT의 크기뿐만 아니라 노드 계산의 데이터 액세스 stride도 고려하기 때문에 그 탐색공간은 기존의 것에 비하여 더 크게 되므로 전수 탐색(exhaustive search)으로 찾는 것은 비현실적이다. 본 논문에서는 이를 효과적으로 찾기 위하여 dynamic programming기법을 이용하고자 한다. dynamic programming 기법에서 트리는 bottom-up 방법으로 구현되고, 그 복잡도는 $O(n^2)$가 된다. 모든 탐색 가능한 stride를 탐색하는 것 대신에 k 개로 제한하여 각 노드에서의 탐색 복잡도를 $O(k^2)$으로 줄였다. 그러므로 제시한 탐색 알고리즘의 복잡도는 $O(k^2n^2)$이 된다. 일반적으로 작은 규모의 DFT 계산에서는 데이터 재구성 비용이 오버헤드가 되어서 데이터 재구성이 없는 정적인 방법으로 하는 것이 유리하다. 하지만, 큰 규모의 DFT 계산에서는 데이터 재구성을 포함한 트리에서 신속한 처리를 할 수가 있다. 따라서 본 탐색 알고리즘은 이러한 데이터 재구성을 고려한 최적 실행시간을 가지는 트리를 제시하게 된다. (그림 6)은 이러한 동적 데이터 재구성과 dynamic programming을 이용한 탐색 알고리즘을 간략하게 나타낸 것이다.

5. 실험 결과

본 논문에서 제시된 DDL 방법을 사용하여 성능 향상됨을 보이기 위해, 카네기멜론 대학교(CMU)에서 개발된 FFT 패키지를 사용하였고, 이를 FFT_SDL[1]으로 명하였다. 이것은 [3]에 제시된 FFT 패키지를 기초로 하였다. 우리는 FFT_SDL 패키지에 DDL 방법을 적용하였고 이를 FFT_DDL이라 명하였다. 본 실험에서 모든 DFT의 데이터 포인트는 double-precision의 복소수로 나타낸다. DDL 방법을 이용한 성능 향상을 나타내기 위하여 FFTW와 FFT_SDL의 성능과 비교하였다. 제시된 DDL 방법이 플랫폼에 상관없이 유용함을 보이기 위하여 최신의 다양한 플랫폼에서 실험을 진행하였다. <표 1>과 <표 2>는 실험에 사용된 플랫폼에 대한 아키텍처 구성 요소, 컴파일러, 그리고 최적화 옵션 등을 요약한 것이다. 시간 측정은 wall clock방법으로 clock() 함수를 사용하였다. 실험에서 정확한 실행 시간을 얻기 위하여, 전체 실행시간이 1초 이상 될 때까지 계산을 반복 수행하였다. 전체 실행시간에서 loop overhead 부분은 빼고 남은 시간의 평균을 각 계산의 실행시간으로 하였다.

1) 기존의 방식은 데이터 배열을 유지하므로 Static Data Layout(SDL)이라고 명하였다.

〈표 1〉 플랫폼 구성 요소

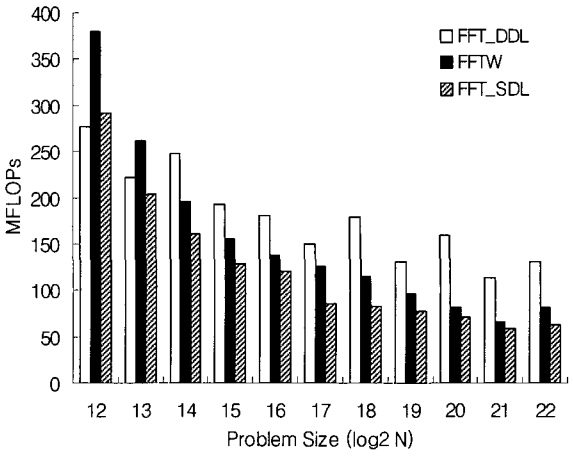
Processor		Pentium 4	AthlonTM 64	Alpha 21264	UltraSPARC III
Clock (MHz)		2600	2000	500	750
L1 Cache	Size (KB)	16	64	64	64
	Line (Bytes)	64	64	64	32
L2 Cache	Size (KB)	512	1024	4096	4096
	Line (Bytes)	64	64	64	64

〈표 2〉 실험에 사용된 컴파일러와 최적화 옵션

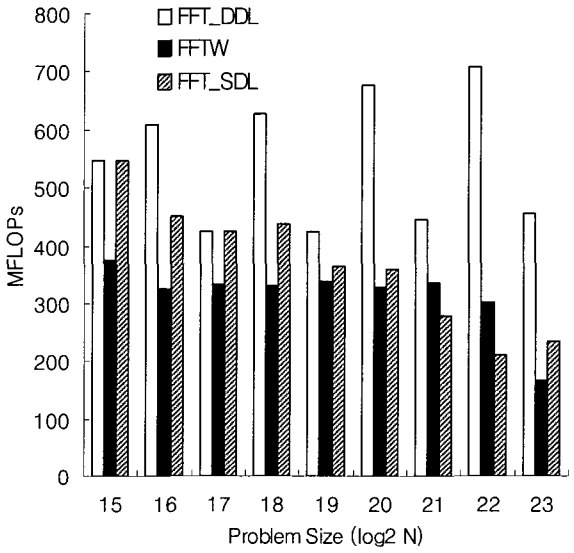
Processor	Compiler	Optimization Flags
Pentium 4	gcc ver. 3.3.2	-06 -fomit-frame-pointer -pedantic -malign-double
Athlon™ 64	gcc ver. 3.2.2	-03 -fomit-frame-pointer -fno-schedule-insns -fschedule-insns2 -fstrict-aliasing -mcpu-pentiumpro -malign-double
Alpha 21264	gcc ver. 2.96	-06 -fomit-frame-pointer -pedantic
UltraSPARC III	gcc ver. 2.8.1	-06 -fomit-frame-pointer -pedantic

(그림 7~10)은 4개의 다른 플랫폼에서 FFT_DDL의 성능과 FFT_SDL과 FFTW의 성능을 비교하여 나타내고 있다. (그림 7)은 Alpha 21264 플랫폼에서 FFT_SDL과 FFTW와 비교하여 FFT_DDL의 성능을 나타낸 비교 그래프이다. N이 2^{13}보다 작은 문제에서는 FFT 계산을 위해 필요한 모든 데이터 포인트가 캐시에 있을 수 있다. 이럴 경우 데이터를 재구성하는 것이 오버헤드가 되어 재구성이 필요 없게 된다. 이럴 경우에는 FFT_SDL이 성능이 좋아 지고 본 연구에서 개발한 DDL 탐색 알고리즘은 FFT_SDL의 트리를 취하게 되어 두 결과는 거의 같은 결과를 낸다. FFT 크기가 2^{13}보다 크고 2^{18}보다 작은 경우, 필요한 모든 데이터 포인터들은 L2 캐시에는 있을 수 있으니 L1 캐시에는 그 일부만이 있게 된다. 이런 경우에 제안되 DDL 방법은 L1 캐시 적중 실패의 수를 줄여 약간의 성능 향상을 얻을 수 있었다. 그러나 FFT의 크기가 L2 캐시의 크기를 넘는 경우, FFT_DDL은 FFT_SDL과 비교하여 최대 2.23배 이상의 성능 향상을 얻었다. FFTW와의 비교에서도 유사한 결과를 얻을 수 있었다. FFTW와 비교하여 FFT_DDL이 최대 2배 정도의 성능 향상을 얻을 수 있었다.

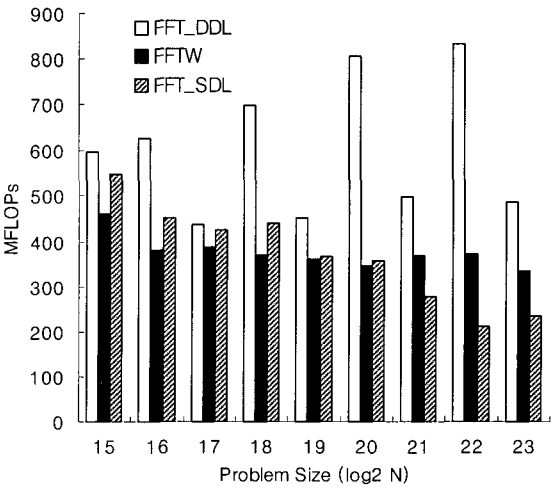
(그림 8)과 (그림 9)는 Pentium 4와 Athlon™ 64 플랫폼에서의 실험에서도 제시된 DDL 방법이 L2 캐시보다 큰 문제에서 기존의 방법보다 뛰어난 성능을 나타냄을 보이고 있다. 이들 최신 플랫폼에서는 L2 캐시도 on-die 캐시로 기존의 머신에 비하여 L2 캐시 적중 실패 비용이 적다. 따라서 FFT 크기가 L1 캐시 크기와 L2 캐시 크기 사이에서는 상대적으로 성능 향상이 이전에 비하여 크지는 않았다. 그러나 L2 캐시 보다 큰 경우에는 FFT_DDL이 FFT_SDL 보다 최대 3.37배 그리고 FFTW와 비교하여 최대 2.23배 까지 성



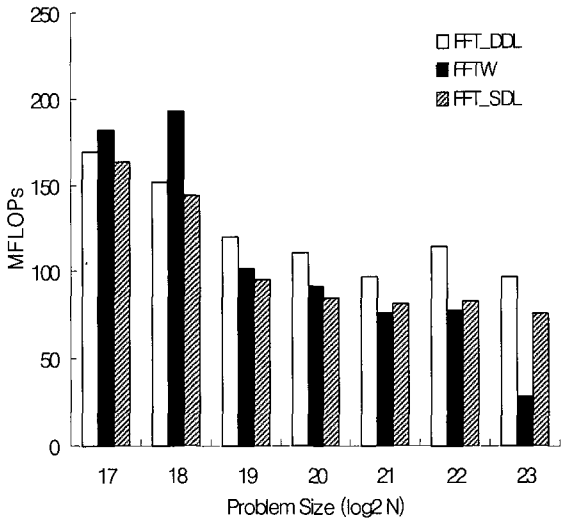
(그림 7) Alpha 21264에서 FFT_DDL의 성능 비교



(그림 8) Pentium 4에서 FFT_DDL의 성능 비교



(그림 9) Athlon™ 64에서 FFT_DDL 성능 비교



(그림 10) UltraSPARC III에서 FFT_DDL 성능 비교

능향상을 얻을 수 있었다. (그림 10)은 UltraSPARC III에서의 DDL 방법에 이용하여 L2 보다 큰 경우에 성능 향상을 얻을 수 있음을 보여주고 있다.

<표 3>은 Athlon™ 64 플랫폼에서 최적화 FFT 트리를 나타내고 있다. 표에서 "s[n]"은 리프 노드로 2^n-point FFT를 계산하는 직선 코드(straight line unrolled code)이다. "ct[s[n1], s[2]]"는 $2^{n1}\times 2^{n2}$로 인수분해 된 FFT로 기존의 데이터 배열의 변화가 없는 FFT_SDL 방법의 Cooley-Tukey 알고리즘이다. "ctddl[s[n1], s[2]]"는 $2^{n1}\times 2^{n2}$로 인수분해 된 FFT로 DDL 방법의 Cooley-Tukey 알고리즘이다. Athlon™ 64 플랫폼에서는 FFT 크기가 L2 캐시보다 작은 경우에는 데이터 재구성 오버헤드가 상대적으로 커서 기존의 SDL 방식 트리보다 계산 성능이 낮아서 제시된 탐색 알고리즘은 자동적으로 FFT_SDL 트리를 최적화 트리로 선택하도록 고안하였다. 하지만, FFT 크기가 커져서 L2 캐시 크기보다 커지면, 제시한 탐색 알고리즘은 자동적으로 데이터 재구성을 하는 FFT_DDL 트리를 최적화 트리로 결정하게 된다.

〈표 3〉 Athlon™ 64 플랫폼에서 최적화 FFT 트리 비교

Size (log2 N)	FFT_SDL	FFT_DDL
10	ct[s[4],s[6]]	ct[s[4],s[6]]
11	ct[s[5],s[6]]	ct[s[5],s[6]]
12	ct[s[6],s[6]]	ct[s[6],s[6]]
13	ct[s[1],ct[s[6],s[6]]]	ct[s[1],ct[s[6],s[6]]]
15	ct[s[4],ct[s[4],s[6]]]	ctddl[ct[s[1],s[6]],ct[s[1],s[6]]]
15	ct[s[5],ct[s[4],s[6]]]	ctddl[ct[s[4],s[5]],s[6]]
16	ct[s[4],ct[s[6],s[6]]]	ctddl[ct[s[3],s[5]],ct[s[3],s[5]]]
17	ct[s[5],ct[s[6],s[6]]]	ctddl[ct[s[4],s[5]],ct[s[3],s[5]]]
18	ct[s[4],ct[s[4],ct[s[4],s[6]]]]	ctddl[ct[s[4],s[5]],ct[s[4],s[5]]]
19	ct[s[5],ct[s[4],ct[s[4],s[6]]]]	ctddl[ct[s[4],s[6]],ct[s[4],s[5]]]
20	ct[s[5],ct[s[5],ct[s[4],s[6]]]]	ctddl[ct[s[4],s[6]],ct[s[4],s[6]]]
21	ct[s[6],ct[s[5],ct[s[4],s[6]]]]	ctddl[ct[s[5],s[6]],ct[s[4],s[6]]]
22	ct[s[6],ct[s[4],ct[s[6],s[6]]]]	ctddl[ct[s[5],s[6]],ct[s[5],s[6]]]
23	ct[s[6],ct[s[4],ct[s[1],ct[s[6],s[6]]]]]	ctddl[ct[s[5],s[6]],ct[s[6],s[6]]]

6. 결 론

본 논문에서 캐시를 기반으로 하는 메모리 계층구조를 가진 플랫폼 상에서 인수분해된 트리구조 DFT의 성능을 최적화 시킬 수 있는 효과적인 방법을 개발하였다. 제안된 방법은 캐시 성능을 향상시키기 위해 stride를 가지는 계산 단계 사이에 데이터를 동적으로 데이터 배열을 재구성하는 방법으로 이를 DDL 방법이라 한다. 제시된 DDL 방법을 사용하여 캐시 적중 실패를 줄임으로서 전체적인 성능향상을 얻을 수가 있다. 또한 트리 구조 FFT 계산 방법에서 FFT 크기와 데이터 stride 액세스를 기초로 하여 가능한 모든 인수분해 트리 중에서 최소 실행시간을 가지는 최적의 인수 분해 트리를 찾아내는 탐색 알고리즘을 개발하였다. 제시된 DDL 방법은 최적화 방법으로 신호 변환 패키지의 상위 알고리즘 레벨에 적용될 수 있다. 본 논문에서는, 제시된 방법으로 계산 성능이 향상됨을 보이기 위하여 FFT 패키지에 적용시켜다. 본 논문에서 제시한 방법의 범용성을 확인하기 위하여 Pentium 4, Alpha 21264, Athlon™ 64, UltraSPARC II 등에서 실험하였다. 실험 결과에 나타난 것처럼, 제시된 FFT_DDL 방법은 FFTW와 FFT_SDL과 비교하여 최대 3.37배의 상당한 성능 향상을 얻을 수 있었다. 또한 이러한 동적 데이터 배열 방법을 이용하여 선형대수나 다른 신호변환 알고리즘에도 적용을 시킬 수 있음을 다른 논문[12, 13]을 통하여 보였다.

참 고 문 헌

[1] A. Ailamaki, D. DeWitt, M. D. Hill, M. Skounakis, "Weaving Relations for Cache Performance," in Proc. 27th International Conference Very Large Data Base, 2001.

[2] D. H. Bailey, "Unfavorable Strides in Cache Memory Systems," Scientific Programming, 1995.

[3] S. Egner, "Zur Algorithmischen Zerlegungstheorie Linearer Transformationen mit Symmetire," Ph.D. Thesis, Universitat Karlsruhe, 1997.

[4] M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," International Conference on Acoustics, Speech, and Signal Processing 1998 (ICASSP 1998), 3, 1998

[5] K. S. Gatlin and L. Carter, "Faster FFTs via Architecture Cognizance," International Conference on Parallel Architectures and Compilation Techniques (PACT2000), Oct., 2000.

[6] G. Haentjens, "An Investigation of Recursive FFT

Implementations," Master's Thesis, Dept. of Electrical and Computer Engineering, Canegie Mellon University, 2000.

[7] J. Johnson and M. Püschel, "In Search of the Optimal Walsh-Hadamard Transform," International Conference on Acoustics, Speech, and Signal Processing 2000 (ICASSP 2000), June, 2000.

[8] M. Linderman and R. Linderman, "Real-Time STAP Demonstration on an Embedded High-Performance Computer," National Radar Conference, 1997.

[9] W. Liu and V. K. Prasanna, "Utilizing the Power of High-Performance Computing," IEEE Signal Processing, September, 1998.

[10] V. Van Loan, "Computational Frameworks for the Fast Fourier Transform," Frontieres in Applied Mathmetics, Vol. 10, SIAM, 1992.

[11] D. Mirkovic, R. Mahassom, and L. Johnsson, "An Adaptive Software Library for Fast Fourier Transforms," Proceedings of the 2000 International Conference on Supercomputing, May, 2000.

[12] N. Prak, B. Hong, and V. K. Prasanna, "Analysis of Memory Hierarchy Performance of Block Data Layout," Proceedings of the 2002 International Conference on Parallel Processing (ICPP 2002), Aug., 2002.

[13] N. Prak and V. K. Prasanna, "Cache Conscious Walsh-Hadamard Transform," International Conference on Acoustics, Speech, and Signal Processing 2001 (ICASSP 2001), May, 2001.

[14] R. Tolimieri, M. An, and C. Lu, "Algorithms for Discrete Fourier Transforms and Convolution," Springer, 1997.

박 능 수

e-mail : neungsoo@konkuk.ac.kr

1991년 연세대학교 전기공학과(학사)

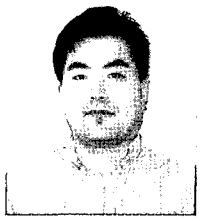
1993년 연세대학교 대학원 전기공학과(석사)

2002년 미국 University of Southern California, 전기공학과(공학박사)

2002년~2003년 삼성전자 책임연구원

2003년~현재 건국대학교 정보통신대학 컴퓨터공학부 조교수

관심분야 : 컴퓨터구조, 임베디드 시스템, 병렬시스템, 멀티미디어 컴퓨팅



최 영 호

e-mail : yunghoch@konkuk.ac.kr

1991년 연세대학교 전자공학과(학사)

1995년 미국 University of Southern California, 전기공학과(공학석사)

2001년 미국 University of Southern California, 전기공학과(공학박사)

2001년~2004년 미국 Intel Corp. 선임 연구원

2004년~현재 건국대학교 공과대학 전기공학과 조교수

관심분야 : 마이크로프로세서, 병렬처리 구조, 네트웍크, 영상압축처리 구조