■ 2010년도 학생논문 경진대회 수상작

플래시 메모리 저장 장치를 사용하는 프로그램의 성능 향상을 위한 정적 분석 기법의 응용

(Applying Static Analysis to Improve Performance of Programs using Flash Memory Storage)

백 준 영 † 조 은 선 ††
(Joon-Young Paik) (Eun-Sun Cho)

요 약 플래시 메모리는 휴대성, 저전력, 대용량의 특징을 갖고 있어 휴대용 기기에서의 사용이 증가하고 있다. 그러나 하드디스크와는 달리 플래시 메모리는 읽기 연산에 비해 쓰기 연산의 비용이 상대적으로 커서, 쓰기 연산 횟수 감소에 대한 연구가 요구된다. 본 논문에서는 데이터 쓰기 연산 횟수를 감소시키기 위해, 플래시 메모리에 저장된 데이터를 다루는 응용 프로그램을 재작성하여 저장될 데이터를 적절히 재배치하기 위한 정적 분석 기법을 제안하였다. 이 기법은 프로그램을 정적 분석해서 쓰기 연산 부분을 파악하고, 이들을 분리해내어 저장되도록 프로그램을 재작성 함으로써, 수행 시간에 전체 쓰기 영역이 줄어들도록 하는 것이다. 따라서 본 논문에서는 프로그램에서 다루어지는 데이터 중 쓰기 가능한 영역을 얻어내는 분석과 가능한 작은 개수의 페이지에 쓰기 대상 부분이 모여 있도록 재배치하기 위한 분석을 고안하였다. 정적 분석 결과는 자주 수행되는 프로그램 경로에 대한 프로파일링 결과와 조합되어 보다 실제적인 분석 결과를 얻고자 하였으며, 결과적으로, FAST 시뮬레이터 상에서 데이터 처리 성능을 향상시키는 데에 기여함을 보였다.

키워드 : 플래시 메모리, 프로그램 정적 분석, 프로파일링, 프로그램 재작성, FAST

Abstract Flash memory becomes popular storage for small devices due to its efficiency, portability, low power consumption and large capacity. Unlike on hard disks, however, write operation on flash memory is much more expensive than read operation, so that it is critical for performance enhancement to reduce the number of executions of write operation. This paper proposes static analysis to rewrite a program to reduce the total number of write operations by merging writable data in a minimum number of pages. To achieve this, we collect information about writable areas by static analysis, and about frequently executed paths by profiling for practicality, and combine both to rewrite the application program to reallocate data. The performance enhancement gained from the proposed methods is shown using a FAST simulator.

Key words : flash memory, program static analysis, profiling, program rewriting, FAST

· 이 논문은 2010년 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. 2010-0013386)

† 학생회원 : 충남대학교 컴퓨터공학과
lucadi@cnu.ac.kr
†† 종신회원 : 충남대학교 컴퓨터공학과 교수
eschough@cnu.ac.kr
(Corresponding author임)

논문접수 : 2010년 5월 31일
심사완료 : 2010년 10월 20일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제16권 제12호(2010.12)

1. 서 론

플래시 메모리는 비휘발성의 반도체 메모리로서, 외부 충격에 강하고 빠른 접근 속도를 갖는 특징이 있다. 따라서 스마트 폰, PMP(Portable multimedia player) 등과 같은 소형 정보 기기의 휴대성, 강한 내구성 및 저전력 등의 요구 사항에 부합하는 장점을 지니고 있다. 플래시 메모리의 종류로는 NOR와 NAND의 두 가지 종류가 있으나[1], 경제성 등이 우수한 NAND 플래시 메모리가 더 광범위하게 사용된다(그림 1)[2].

그림 2는 NAND 플래시 메모리의 구성을 보여준다. 연속된 데이터는 하나의 페이지(page)로 묶여지는데 저장된 데이터에 대한 읽기와 쓰기 연산은 페이지 단위로 이루어진다. 연속된 여러 페이지가 모이면 하나의 블록(block) 단위가 된다.

플래시 메모리의 특징 중 하나는 읽기 연산(read)과 쓰기 연산(write)의 속도가 서로 다르다는 것이다. 표 1에서 볼 수 있듯이 읽기 연산은 쓰기 연산에 비해서 일반적으로 2.5배 정도 빠르다고 알려져 있다[3].

이 외에도 플래시 메모리는 제자리에 덮어 써서 갱신하는 연산을 지원하지 않는다는 특징이 있다. 따라서 특정 위치의 데이터를 변경하기 위해서는 변경 대상 데이터를 포함하고 있는 부분을 초기화시킨 후, 변경된 내용을 새로 저장하는 과정이 필요하다. 이 때 초기화시키는 연산을 지움 연산(erase)이라고 하는데, 이 지움 연산은 페이지가 아닌 블록 단위라는 점이 읽기/쓰기 연산과 다른 점이다.

따라서 한 페이지의 일부분이 변경되는 경우, 그 페이지 전체에 대해 다시 쓰기 연산이 수반되며, 더욱이 그 포함하는 블록 전체에 대해 지움 연산을 선행해야 하는 단점을 가지게 된다. 이와 같은 플래시 메모리의 특징은

그림 1 플래시 메모리의 사용 예(캠코더)[2]

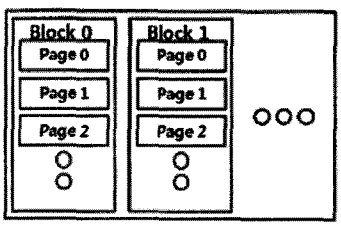


그림 2 NAND 플래시 메모리 구조

표 1 NAND 플래시 메모리의 접근 속도[3]

	Read	Write	Erase
Samsung K9WAG08UIA 16Gbit SLC NAND	80 us (2 KB)	200 us (2 KB)	1.5 ms (128 KB)

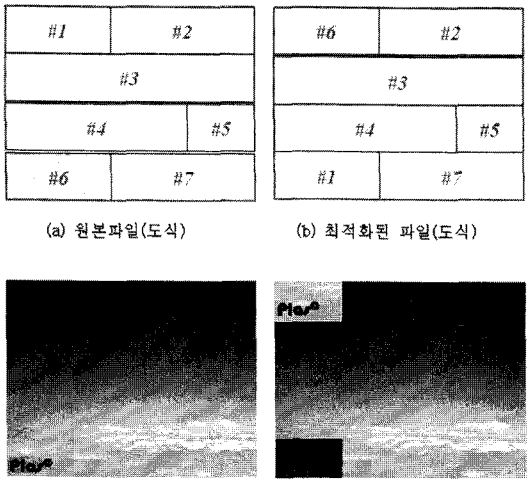
전통적인 하드디스크나 기존 저장 장치에 쓰이던 응용 소프트웨어나 파일 포맷을 그대로 사용하기에는 불필요한 성능 저하를 초래할 가능성을 가지고 있다.

예를 들어 크기가 큰 이미지의 특정 위치에 워터 마크 등을 삽입하는 경우와 같이 큰 파일의 극히 일부만 수정 되는 경우, 같은 페이지 내에 갱신될 부분과 무관한 영역, 즉 워터 마크가 아닌 영역에 대한 쓰기 연산이 불가피하게 발생하게 된다. 더구나 파일이 여러 페이지로 구성되고, 이 중 쓰기 연산이 필요한 부분이 양적으로는 적으나 분포가 산재되어 있다면, 그 비효율성이 심화된다. 즉, 전체 페이지의 수가 p, 한 페이지를 쓰는데 걸리는 시간을 tw라 하면 쓰기 연산되는 부분이 전체 페이지에 고루 분포하고 있는 경우에 대한 수행 시간은 $p*tw$이다. 그러나 실제 쓰기 연산이 필요한 부분만 쓸 수 있다면, pb가 페이지 크기, wb는 쓰기 연산되는 부분의 바이트수라 할 때, $\lceil wb/pb \rceil * tw$ 가 된다. 따라서 쓰기 연산이 필요한 부분이 전체 페이지에 비해 적다면 (즉, $\lceil wb/pb \rceil << p$ 일 때), 쓰기 연산이 필요하지 않은 영역에서 불필요한 연산이 발생한다.

이를 극복하기 위한 하나의 방법으로 파일 데이터를 쓰기 데이터와 그렇지 않은 데이터로 분리하여 재배치 할 수도 있다. 이것은 데이터의 포맷을 바꿈으로써 가능하나, 이러한 포맷들은 대부분 표준화되어 있는데다가 여러 가지 파일 포맷을 일일이 플래시 메모리용으로 바꾸는 것은 비현실적이다. 또한 표준화되어 있지 않은 데이터 포맷인 경우라 해도, 응용 프로그램 개발자가 프로그램 개발 단계에서 데이터의 어느 부분이 읽기 전용인지 등을 미리 계산하여 파악한 후 같은 페이지에 스스로 정리해서 넣도록 하는 것 역시 무리가 있다.

본 논문에서는 이러한 단점을 보완하기 위한 방법으로, 프로그램 수행 중 사용되는 데이터와 플래시 메모리 내의 주소간의 매핑을, 프로그램 수행 전에 조정하여 데이터를 재배치하는 방법을 사용한다. 즉, 동일한 페이지에 쓰기 연산 가능 데이터와 읽기 전용 데이터가 같이 존재한다면, 읽기 전용 데이터는 쓰기 연산 데이터로 인해서 쓰기 연산될 수 있다. 그러므로 쓰기 연산 가능 데이터와 읽기 데이터를 분리하여 각각을 페이지 단위로 재배치시킴으로써 불필요한 쓰기 연산을 줄일 수 있다.

그림 3(a), (b)는 도식화하여 나타낸 예로써 각 라인은 한 페이지를 의미하며 512 byte이다. 만일 데이터 #2, #6은 쓰기 가능 데이터이고(짙은 부분) #1, #3, #4,



(a) 원본파일(도식) (b) 최적화된 파일(도식)

(c) 원본파일(그림) (d) 최적화된 파일(그림)

그림 3 제안된 기법의 예

#5, #7는 읽기 전용 데이터일 때, 그림 3의 (a)는 2개 페이지에 쓰기 가능 데이터가 포함되어 있기 때문에, 총 1024 bytes 크기의 쓰기 연산이 발생할 수 있다. 그러나 그림 3의 (b)는 쓰기 가능 데이터인 #6이 #2와 동일한 페이지로 재배치되었기 때문에, 쓰기 가능 데이터를 포함한 1개 페이지만이 쓰기 연산된다. 즉, 파일 데이터 재배치 기법을 적용하면 512 bytes 크기의 쓰기 연산이 발생하게 되어 쓰기 연산 범위가 감소된다. 만일 이것이 이미지 파일이었다면 원본 파일과 최적화된 파일 예는 그림 3(c), (d)와 같이 나타날 수 있으며, 그림 3(a), (b)에 각각 해당된다. 쓰기 가능 데이터가 워터마크에 해당하는 부분으로 가정하기 때문에, 좌측 하단의 기호가 상단으로 재배치되어 저장됨을 알 수 있다.

이러한 방법은 프로그램의 사전 분석 결과를 이용하여 쓰기 연산 가능 데이터와 읽기 연산만이 이루어지는 데이터를 구분하는 것으로부터 출발한다. 또한 어느 데이터가 어디에 저장되는 것이 효율적인지를 체계적으로 사전에 분석하여 최대한 적절한 재배치를 가능하도록 하는 정적 분석 기법과 프로그램 재작성 기법의 성능이 매우 중요하다. 본 논문에서는 이를 위해 효과적인 데이터 재배치를 가능하게 하는 프로그램 정적 분석과 프로그램 재작성 기법을 제안한다.

먼저 프로그램 분석 기법을 통해 응용 프로그램 내에서 쓰기 데이터를 다루는 명령 부분과 그렇지 않은 부분을 분리한다. 그리고 쓰기 데이터를 가급적 모아서 배치하여 처리하도록 명령을 변경한다. 결과적으로 그림 4처럼 변경된 프로그램(*Program P**)은 기존 포맷의 파일 대신 효율적인 포맷으로 변경된 파일(*File A**)을 다루어 플래시 메모리에서 데이터 처리 속도가 증가하게 된다.

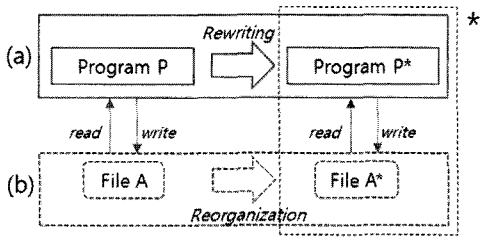


그림 4 응용 프로그램 재작성을 통한 데이터 재배치

본 논문의 구성은 다음과 같다. 2장에서는 플래시 메모리에서 쓰기와 읽기가 혼재해 있는 상황에서의 속도 향상을 위한 관련 연구들을 살펴보고, 3장에서는 데이터 재배치를 위한 주요 기술인 프로그램 정적 분석 기법을 중심으로 소개한다. 4장에서는 성능 분석 결과를 설명하고, 5장에서는 정리와 향후 연구 과제를 다룬다.

2. 관련 연구

2.1 플래시 메모리 변환 계층(FTL)

앞서 언급했듯이 기존 저장 장치와 다른 플래시 메모리의 특성들로 인해서, 플래시 메모리를 기존의 시스템에 적용시키는 것이 쉽지 않다. 이러한 제약을 극복하기 위해서, 파일 시스템과 플래시 메모리 사이에 위치한 플래시 메모리 변환 계층(FTL)[4-8]이 제안되었다. 플래시 메모리 변환 계층은 프로그램 실행에 의해서 발생된 논리 주소를 실제 물리 주소로 변환시키는 역할을 한다. 쓰기 연산 명령의 경우, 플래시 메모리 변환 계층의 주소 변환 과정을 통해서 쓰기 데이터는 플래시 메모리의 새로운 위치에 저장된다. 본 논문에서 제안하는 방법에 의해 재작성된 프로그램도 파일 데이터의 주소를 변환시킨다는 면에서 플래시 메모리 변환 계층과 유사한 효과를 가진다.

플래시 메모리 변환 계층은 지움 연산을 유발하는 쓰기 연산들을 로그 블록에 저장시키는 경우, 이것은 비용이 큰 지움 연산의 발생을 지연시킴으로써 성능을 향상시킨다. 본 논문의 제안되는 기법의 결과도 적절한 단위의 로깅과 함께 적용되었을 때 효율성이 극대화된다.

하지만, 플래시 메모리 변환 계층과 달리, 프로그램 재작성을 통해 이를 수행하게 되면 프로그램을 분석한 결과를 토대로 진행되므로, 각 응용 프로그램의 의미구조가 데이터 재배치에 반영된다는 장점을 가진다. 첫째, 데이터의 사용 방식을 예측하여 이에 맞는 배치를 할 수 있으며, 둘째 플래시 메모리 변환 계층의 페이지 단위 대신 프로그램 내의 쓰기연산 단위인 바이트 단위로 보다 정교하게 데이터 재배치를 할 수 있다. 본 논문에서는 이와 같은 데이터 재배치를 지원하는 프로그램 분석 및 재작성 기법을 제안한다.

2.2 프로그램 분석 기법과 플래시 메모리 성능 개선

프로그램 분석 기법[9-11]을 이용한 기존 연구들은 주로 읽기 연산 최적화를 통한 성능 개선을 주제로 이루어져왔다[12,13]. 이는 쓰기의 오버헤드가 큰 플래시 메모리의 특성 때문에, 일반적인 데이터 보다는 수정이 불가능한 데이터들(예: 프로그램 파일)의 보관용으로 사용될 것으로 가정되어 왔었기 때문이다. 이러한 연구들은 플래시메모리에 저장된 프로그램 중 필요한 부분을 RAM에 적기에 옮겨주기 위한 것이 궁극적인 목표로 하고 있으며, 이를 위해 시간적으로 인접한 사용(temporal locality)이 예상되는 코드를 가급적 하나의 페이지에 넣고자 하는 것을 다루고 있다.

프로그램 분석 기법을 적용하여 쓰기 연산의 오버헤드를 극복하고자 한 연구도 최근에 나타나고 있다. 그 중 하나는 일괄적으로 저장되는 데이터의 쓰기 양을 감소시키기 위해 실행 중 갱신된 데이터만을 식별해서 선택적으로 쓰기 연산을 하는 연구이다[14]. 이것은 본 논문이 목표로 하는, 데이터 포맷을 효율적으로 바꾸어 성능을 향상시키는 개념과는 무관하지만, 두 방법 모두 쓰기 연산이 유발하는 오버헤드를 줄인다는 측면에서 상호 보완적으로 사용될 수 있다.

본 논문에서 제안하는 아이디어의 일부는 이전 학술대회에 발표 된 바 있다[15]. 그러나 플래시 메모리 데이터 관리 절차 전체에 대한 논문의 한 부분으로 다루어졌었으므로, 본 논문에서 소개되는 구체적인 프로그램 분석 알고리즘이나 최적화를 위한 기술적인 내용에 대한 심도 있는 논의와는 거리가 있다.

3. 프로그램 분석 기법

3.1 데이터 재배치를 위한 정보 수집

데이터 재배치를 위한 정보 중 가장 중요한 것은 쓰기 가능 범위이다. 쓰기 가능 범위란 응용 프로그램에 의해서 갱신되는 파일 데이터 영역을 의미하므로, 쓰기 연산이 시작 되는 데이터 상의 위치와 크기를 분석, 수집하는 것이 필요하다. 논의를 단순하게 하기 위해, 본 논문에서는 응용 프로그램에서 쓰기 연산은 write() 함수를 통해서만 이루어지며, 쓰기 연산의 시작 위치를 나타내는 lseek() 함수가 존재할 수 있다고 가정한다.

기본 블록은 연속적으로 실행되는 코드들의 단위를 의미하며, 본 논문에서는 일반적인 정의[9]에 따른 분리와 함께 추가적으로 write() 함수를 기준으로 앞/뒤 부분으로 분리하였다. 따라서 write() 함수는 각 기본 블록의 마지막에 위치된다.

쓰기 가능 범위는 write() 함수와 lseek() 함수의 매개 변수의 값을 분석하여 얻을 수 있다. 이를 위해서 각 블록에서 사용 가능한 변수와 값의 쌍을 수집하는 알고리즘을 제안한다. 하향식 분석 방법을 이용하여 모든 기본 블록을 순회하며, 각 기본 블록의 데이터 정보의 변화가 없을 때까지 반복하는 반복 분석 기법을 이용한다. 알고리즘의 실행 과정은 다음과 같다.

1) 기본 블록 X를 방문한다.
2) 기본 블록 X의 선행 노드 중 하나를 선택하고 다음을 실행한다.
   (a) 기본 블록 X에 속한 모든 실행문 (statement)의 정보를 갖고 있는 *Stat*[X]를 분석하여, 각 실행문의 *LHS(left hand side)* 변수와 그 값으로 지정될 *RHS(right hand side)* 정보의 쌍을 *INFO*[X]에 저장한다. 따라서 *INFO*[X]는 기본 블록 X에 의해 실행 중 주기억장치에 저장될 변수-값 쌍을 추상적으로 표현하고 있다.
   evaluate() 함수는 이러한 *INFO*[X]를 토대로 선행 기본 블록 Y에서 전달된 정보인 *OUT*[Y]을 이용하여 *RHS*를 보다 최대한 상수와 가까운 형태로 바꾸어주는 과정을 추상화 하고 있다. *RHS*가 입력 인자 값처럼 값이 수행 중에 결정되는 부분을 포함하고 있는 경우에는 'input' 등의 변수명으로 표현하고 진행한다. 예를 들어, *OUT*[Y]={[a, 100], [b, input*20]}, *INFO*[X]={[c, 200]}이고, *Stat*[X] = {[x, a+c], [y, b+a]}라면, *INFO*[X]={[x, 300], [y, input*20+100]}가 된다.
   (b) 모든 실행문을 대상으로 (a)를 실행한다.
   (c) 아래와 같은 데이터-흐름 공식(data transfer function)을 이용하여 *EDGE*[X][Y]와 *OUT*[X]를 구한다.
   $EDGE[X][Y] \leftarrow INFO[X] \cup (OUT[Y] \nabla^{1)} INFO[X])$
   $OUT[X] = \cup_{Y \in Predecessor[X]} EDGE[X][Y]$
3) 기본 블록 X의 모든 선행 노드들을 대상으로 과정 2)를 반복 실행한다.
4) 위와 같은 과정을 모든 기본 블록들에 적용하며, 각 블록의 데이터 정보가 변하지 않을 때까지 반복한다.

그림 5는 각 블록에서 의미 있는 변수와 값의 쌍을 수집하는 알고리즘을 의사코드 형태로 나타낸 것이다. 프로그램에 있는 실행문의 개수를 N이라 하면, 각 자료구조(*INFO*, *OUT*)가 가질 수 있는 변수의 개수도 지정문의 개수가 N을 넘지 않으므로 evaluate() 함수나 *EDGE*[X][Y]의 시간복잡도는 한 rhs에 포함되는 연산자의 수가 상수 개로 제한된다고 가정할 때 $O(N)$으로 볼 수 있다. 그리고 기본 블록의 개수와 블록에 속한 최대 실행문의 개수도 N을 넘지 못하고, 선행 블록을 의미하는 *PB*에 포함된 블록의 개수도 N보다 작으므로, K가 while 문의 반복횟수라 하면 알고리즘의 시간 복

1) ∇은 *OUT*[Y]와 *INFO*[X]가 동일한 lhs를 가지고 있을 경우에, *OUT*[Y]에서 [lhs, result] 정보를 제외한다. 즉, *OUT*[Y]={[x, 100], [y, 200]}, *INFO*[X]={[x,300]}일 경우, *OUT*[Y] *INFO*[X]는 {[y, 200]}이 된다.

```
BlockList : a set of basic blocks
Stat[X] : a set of pairs of lhs and rhs for operations in X,
        X∈ BlockList, ex) Stat[X]={[lhs, rhs], [,]... }
INFO[X]=OUT[X]=NULL, X∈ BlockList
PB[X] : a set of predecessor blocks of X, X∈ BlockList
EDGE[X][Y] = NULL, X∈ BlockList, Y∈ PB[X]

change ← true
While change == true, do
    change ← false
    For every X∈ BlockList, do
        old_OUT ← OUT[X]
        For every Y∈ PB[X], do
            For every S ∈ Stat[X], do
                result ← evaluate(OUT[Y], INFO[X], S.rhs)
                INFO[X] ∪= [S.lhs, result]
            endFor
            EDGE[X][Y] ← INFO[X] ∪ (OUT[Y] ▽ INFO[X])
            OUT[X] ∪= EDGE[X][Y]
        endFor
        if old_OUT != OUT[X], then
            change ← true
        endIf
    endFor
endWhile
```

그림 5 쓰기 연산의 사용 가능한 변수와 값의 쌍의 수집 알고리즘

잡도는 $O(K \cdot N^4)$이다. 공간적으로는 전체 알고리즘 수행 동안 각 블록마다 최대 N개의 변수를 가지는 OUT[X]를 보관해야 하며 각 변수가 가질 수 있는 값의 가지 수가 M이라고 한다면, OUT[X]의 공간복잡도는 $O(K \cdot N \cdot M)$이다. ($INFO$[X], $EDGE$[X][Y]는 각 블록에 대한 수행마다 사용되는 임시 공간이므로 크게 차지하지 않는다.) 그런데 K는 기본 블록의 방문 순서를 바꾸는 등의 데이터 플로우 분석 속도를 향상시키는 방법을 사용한다면 통상 2~3회로 줄일 수 있고[16], 단조 증감하는 값에 대해서는 반복문의 조건문 내의 최대 또는 최소값을 결과값으로 지정하는 방법을 써서 한 변수가 가지는 값의 개수를 줄이는 방법을 사용한다면 M도 상수나 N에 비례하는 수로 간주할 수 있다[17].

본 논문에서는 일반적인 데이터흐름 분석에서와 달리, X 에만 의존하는 기본 블록 입력을 이용하는 대신, 선행 블록 정보를 보존함으로써 경로 정보를 보존하는 입력인 $EDGE$[X][Y]를 도입하였다. 따라서 보다 일반적인 집합 기반 분석에 비해 정교한 쓰기 연산 정보를 구한다[9]. 예를 들어 다음과 같은 프로그램에서 $EDGE$는 선행 블록이 Y인지 Z인지에 따라 동일한 X에 대해 별도의 정보를 유지하게 된다.

즉, 제안하는 분석에서의 $EDGE$[X][Y]는 {x={100}, y={100}}의 값을 갖고, $EDGE$[X][Z]는 {x={200}, y={200}}의 정보를 가짐을 알 수 있다. 반면, 선행 블록의 정보가 없는 경우 일반적인 기본 블록으로의 입력을 IN[X]라 한다면[9], 이것은 {x={100,200}, y={100,200}}의 값을 가지게 된다.

```
if (..) {                    // 기본 블록 Y 시작
        x = 100;
        y = 100;
}                            // 기본 블록 Y 끝
else {                       // 기본 블록 Z 시작
        x = 200;
        y = 200;
}                            // 기본 블록 Z 끝
// 기본 블록 X 시작
lseek(fd, x, SEEK_SET);
wtite(fd, buf, y);
// 기본 블록 X 끝
```

3.2 쓰기 연산 가능 범위 찾기

쓰기 연산 가능 범위는 write() 함수와 lseek() 함수의 매개 변수 정보를 이용하여 얻는다. write() 함수와 lseek() 함수의 함수 원형은 아래와 같다.

ssize_t write(int fildes, const void *buf, size_t nbyte);

off_t lseek(int fildes, off_t offset, int whence);

write()의 매개 변수 *nbyte*의 값은 쓰기 연산의 크기를 의미한다. 그리고 lseek() 매개 변수 *whence*와 *offset*은 각각 위치 탐색 기준과 위치 탐색 기준과 떨어진 거리를 나타낸다. 따라서 *whence*와 *offset*의 조합이 쓰기 연산의 시작 위치를 나타낸다. 본 논문에서는 위치 탐색의 기준을 파일의 시작 위치로 설정하는 *SEEK_SET* 플래그만 있다고 가정한다. lseek()이 없는 write()의 시작 위치는 직전의 쓰기 연산의 시작위치와 크기를 분석하여 얻어낼 수 있다. 이 경우는 해당 값을 위치인자로 가지는 lseek()을 write() 호출 직전에 삽입한 것과 동일하게 분석될 수 있다.

*WR*은 쓰기 연산 가능 범위 정보를 나타낸다. *WR*은 write() 함수와 lseek() 함수의 매개 변수 정보를 나타낸다. *WR.offset*은 쓰기 연산 시작위치를 나타내고, *WR.range*는 쓰기 연산의 크기를 의미한다. 즉, *WR*의 *offset*과 *WR*의 *range*는 각각 *offset*과 *nbyte* 변수의 값을 의미하며, 각 변수의 값은 데이터 재배치 정보를 이용하여 얻는다.

쓰기 연산 가능한 범위는 쓰기 연산의 시작 위치와 크기의 가능한 모든 조합으로 구한다. 따라서 쓰기 연산 가능한 범위는 *offset*과 *range*의 카티션 곱으로 구할 수 있다. 앞서 언급한 대로 변수 값의 분석 결과에 경로 정보가 보존되어 있으므로, 카티션 곱에 대해서도 정교한 결과를 얻을 수 있다. 앞 절의 예에서 본 분석을 적용하면 앞서 계산한 $EDGE$[X][Y]와 $EDGE$[X][Z]를 통해서 <100,100>과 <200,200>의 2개의 쓰기 가능 영역을

찾을 수 있다. 반면, 일반적인 집합 기반 분석을 이용한 결과로는 <100,100>, <100,200>, <200,100>, <200,200>의 총 4개의 쓰기 가능 범위를 구할 수 있고, 실제로는 발생할 수 없는 영역 <100,200>, <200,100>을 포함하게 된다[9].

이러한 분석은 특정 메모리 영역을 포함하는 범위를 안전하게 예측하고자 하는 정적 분석이다. 안전한 예측을 위한 정적 분석은 입력 데이터에 종속적으로 변할 수 있는 부분을 포함하는 경우 결과 범위가 너무 넓어져 유용성이 떨어질 수 있다는 단점이 존재한다. 예를 들어 특히 입력 데이터 값이 쓰기 대상 데이터의 위치나 크기와 관련이 있고, 이 값이 사용자에 의해 동적으로 임의로 부여되는 경우, 극단적으로 전체 데이터를 모두 쓰기 가능한 영역으로 예측할 수도 있다.

그러나, 제한적이나마 입력 데이터가 일정한 특성을 지닌다면, 그 성질을 이용하여 쓰기 가능 영역에 대한 분석을 비교적 정교하게 개선할 수 있다. 예를 들어 그림 3의 파일과 같은 재배치의 경우 쓰기 가능한 영역이 특정 포맷에 대해 늘 좌측 하단이고 크기가 일정하다면, 프로그램 분석을 통해서 변경 데이터 정보와 입력 파일의 관계를 알 수 있고, 각 입력 파일의 정보를 변경 데이터 분석 정보에 적용하여 정확한 재배치 정보를 얻을 수 있기 때문이다.

그림 6은 쓰기 연산 가능한 모든 범위를 구하는 알고리즘이다. *WR*은 쓰기 연산마다 존재하므로 최대 *N*개로 간주할 수 있으므로, 각 *WR.offset*와 *WR.range*가 가질 수 있는 최대 원소의 개수가 각각 N_o와 N_R이라고 한다면, 그림 6 알고리즘의 시간 및 공간 복잡도는 $O(N \cdot N_o \cdot N_R)$가 된다. 프로그램 분석기는 C++ 코드를 위한 정적 분석 툴인 Dehydra[18]를 이용하여 구현되었다.

```
Region ← NULL
For every WR ∈ WR[N], N ∈ [1, ..]
    For every offset ∈ WR.offset do
        For every range ∈ WR.range do
            Region ← (offset, range)
        endFor
    endFor
endFor
```

그림 6 실제 쓰기 연산 가능 범위를 구하는 알고리즘

3.3 쓰기 연산 가능 범위 최적화 조합

앞 절에서 구한 쓰기 가능한 범위는, 분산된 쓰기 연산 가능 영역들을 조합하여 가급적 작은 수의 페이지에 모아서 배치하는 것에 사용된다. 쓰기 가능한 영역의 분포는 여러 형태가 가능하나, 본 논문에서는 논의를 간단하게 하기 위해 모든 쓰기 연산 가능 영역은 페이지 크기보다 작으며, 여러 페이지에 걸쳐있지 않고, 오직 한 페이지 내에 존재한다고 가정한다. 실제로 적용될 때 쓰기 연산 가능 영역이 여러 페이지에 걸쳐있는 경우에는 사전에 페이지 경계에서 쓰기 연산을 분리시키는 전처리를 할 수 있다.

실제 수행 시에 효과를 가질 수 있는 재배치를 위해서는 가급적 동일 경로를 따라 쓰기 가능한 영역을 조합하는 것이 바람직하다. 본 논문에서 제안하는 분석은 경로 정보를 바탕으로 하여 이러한 조합을 가능하게 돕는다. 특히 배타적인 경로에 위치한 쓰기 가능 영역들의 조합을 배제할 수 있도록 해주는 것이 중요한데, 배타적인 경로에 있는 영역들의 조합은 불필요할 뿐 아니라, 쓰기 연산을 더욱 분산시켜 오히려 성능 저하를 유발할 수 있기 때문이다.

또한 본 논문에서 제안하는 분석은 정적 분석 외에도 동적인 프로파일 정보를 구한 후 정적 분석 결과와 결합함으로써, 실제 실행 패턴을 반영한 최적화된 쓰기 영역의 조합을 이룰 수 있도록 도모한다. 따라서 영역 최적화 조합을 얻을 수 있다.

본 논문에서는 최적화된 조합을 찾기 위한 목적 함수를 쓰기 가능 영역들 중에서 실행 빈도가 높은 영역들을 한 페이지에 위치시킬수록 값이 커지도록 설정하였다. 최적화 조합은 쓰기 연산 횟수의 감소를 목적으로 하기 때문에, 다음과 같은 함수의 최대값을 구하는 문제와 동일하다.

$$\sum_{i \in N} P_i \; subject \; to \; \sum_{i \in N} S_i \leq PS$$

P_i와 S_i는 각각 i번째 쓰기 가능 영역의 프로파일 정보와 크기 정보를 의미한다. PS는 한 페이지의 크기를 나타낸다. 즉, 쓰기 가능 영역들의 크기가 페이지 크기보다 작은 조합들 중에서 프로파일 정보가 가장 큰 조합을 찾는다.

본 논문에서는 이러한 목적 함수를 계산하기 위해서 동적 프로그래밍 기법을 적용하였다. 동적 프로그래밍의 사용은 시간 복잡도를 감소시킨다. 브루트 포스(brute force) 방식을 이용한 계산의 시간 복잡도는 $O(2^N)$가 되지만, 동적 프로그래밍 기법을 적용하면, $\max(S_i) * O(N)$가 된다.

동적 프로그래밍은 아래의 재귀 공식을 이용한다.

$$D[i,s] = \begin{cases} D[i-1, PS] & if \;\; S_i > PS \\ max\{D[i-1, PS], D[k-1, PS-S_i] + P_i\} & else \end{cases}$$

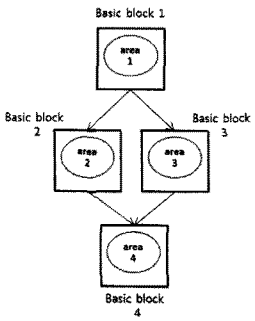
어떤 단계에서의 목적 함수의 최대값은 그 직전 단계까지의 정보를 이용한다. 즉, 재배치 대상 영역의 크기가 페이지보다 크다면, 재귀 공식 적용 대상에서 제외되지만, 영역의 크기가 페이지보다 작다면, 이 전 단계에서의 중간 결과 값들과 현재 고려 대상 영역과의 관계를 고려하여 계산된다.

쓰기 연산 가능 영역 최적화 조합의 알고리즘은 그림 7과 같다. 배타적인 경로에 위치한 영역들이 조합되는 것을 막기 위해, 경로 별로 목적 함수를 적용한다. 최적화된 조합일수록 목적 함수의 값이 크기 때문에, 목적 함수 값이 가장 큰 경로에 위치한 영역들이 조합된다. 새롭게 조합된 영역들은 이후 최적화 조합 과정에서 제외되며, 이러한 과정은 조합 가능한 모든 영역들이 최적화 조합 될 때까지 계속된다.

```
While( any area for all areas is true)
   for each path in all paths
      Dynamic Programming (path)
   endFor
   Select the areas for combination
         on the path with the highest value
   Merge the selected areas
   Set the merged areas false
endWhile
```

그림 7 데이터 재배치를 위한 쓰기 영역 최적화 조합 알고리즘

그림 8은 영역 최적화 조합의 간단한 예이다. 그림 8(a)는 응용 프로그램의 제어 흐름도이다. 제어 흐름도의 노드는 기본 블록을 의미하고, 각 노드는 한 개의 쓰기 가능 영역을 포함한다. 그림 8(b)는 각 쓰기 연산 가능 영역에 대한 정보와 실행 가능한 2가지 경로를 나타낸다. 이 정보들을 이용하여 최적화 조합을 계산한다.



Path 1 : 1 -> 2 -> 4
Path 2 : 1 -> 3 -> 4

area 1	10	1	100
area 2	3	2	150
area 3	7	3	200
area 4	10	4	300

(b)

그림 8 간단한 영역 최적화 조합

최적화 조합을 찾기 위해, 가장 먼저 실행 경로 별로 목적 함수를 적용하여 목적 함수의 값이 가장 큰 경로를 찾는다. 그림 8에서 경로 2의 쓰기 가능 영역 3과 4가 한 페이지에 위치 될 때 목적 함수의 값이 가장 크다. 따라서 쓰기 가능 영역 3과 4를 페이지 단위로 재배치 시킨다. 재배치된 영역들을 제외시킨 후, 최적화 조합 과정을 진행시켜 영역 1과 영역 2를 추가적으로 한 페이지에 재배치시킨다.

4. 성능 분석

4.1 실험 환경

본 논문이 제안하는 분석의 성질 및 성능을 알아보기 위해, 본 분석을 기반으로 재작성된 프로그램의 성능 개선을 알아보는 실험을 실시하였다. 따라서 실험은 프로그램 재작성에 의한 데이터 재구성에 대한 전체 프로세스의 성능 향상을 파악하기 위한 실험 결과를 관찰하였다. 또한 제안하고 있는 정교한 프로그램 분석에 대한 영향을 알아보기 위해 다각도의 실험을 병행하였으며, 실제적인 플래시 메모리 환경과 가깝게 하기 위해 FAST 시뮬레이터[7]를 사용하였다. FAST는 플래시 메모리 주소 변환 장치이며, 파일 시스템에서 요청한 논리 주소를 물리 주소로 변환한다. FAST는 성능 향상을 위해 로그 블록들을 갖는다. 로그 블록은 제자리 쓰기 연산을 지원하지 않는 플래시 메모리 특성으로 인한 성능 저하 현상을 완화시킨다. 즉, 제자리 쓰기 연산을 유발하는 쓰기 요청이 발생하면, 해당 요청이 로그 블록에 실행되도록 하여, 불필요한 지움 연산 발생을 막는다. FAST에서 두 종류의 로그 블록(SW 로그 블록, RW 로그 블록)[2]을 갖지만, 본 논문에서는 RW 로그 블록만(64개)을 사용하였다. FAST 시뮬레이터에서 페이지의 크기는 512 bytes이고, 블록은 32개 페이지로 구성되어, 그 크기는 16 KB이다. 쓰기 연산 속도와 지움 연산 속도는 각각 200 us와 1.5 ms로 설정하였다.

실험 파일 데이터는 512 KB인 파일을 사용하였다. 이 원본 파일은 파일 데이터 재배치 과정을 통해서 여러 쓰기 가능 영역들이 한 페이지에 위치할 수 있는 재배치 파일이 된다. 따라서 재배치 파일은 원본 파일에 비해서 쓰기 가능 영역을 포함한 페이지의 개수가 적다. 제안 기법을 통한 성능 향상은 원본 파일과 재배치 파일에 대한 서로 다른 쓰기 연산 패턴들을 FAST에 적용하여, 그 실행 시간을 비교하여 알 수 있다.

각기 다른 3가지 수행 환경에서 성능 측정을 하였다. 각 수행 환경은 (1) 원본 파일을 사용하는 응용 프로그램(이하, 원본 프로그램), (2) 원본 파일을 사용하는 로깅 기법이 적용된 프로그램(이하, 로깅 프로그램), (3) 재배치 파일을 사용하는 로깅 기법이 적용된 프로그램(이하, 최적화 프로그램)이다.

4.2 분석

4.2.1 응용 프로그램 단계

본 논문에서는 프로그램 재 작성 기반 데이터 재배치 기법의 성능 향상을 극대화시키기 위해 로깅 기법을 적용하였다. 즉, 동일한 페이지에 대한 쓰기 연산들은 로

2) SW 로그블록은 연속적인 쓰기 연산 요청들을 처리한다. RW 로그블록은 비연속적인 쓰기 연산을 위한 로그 블록이다[7].

깅 기법으로 인해서 쓰기 연산 한번으로 대체된다고 가정하였으며. 따라서 실제적인 쓰기 연산 횟수는 쓰기 연산을 포함한 페이지 개수와 동일하다. 로깅 기법에서 쓰기 연산 요청이 발생하면, 해당 쓰기 연산은 실행되지 않는 대신에 쓰기 연산 정보가 로그 정보로써 로깅된다. 결과적으로 쓰기 연산되는 최종 데이터는 로그 정보들을 원본 데이터에 적용하여 생성된다.

로깅 기법을 적용하기 위해서 응용 프로그램 단계에서 추가 연산이 필요하다. 첫째로, 쓰기 연산 정보를 로깅하는 과정이 필요하다. 둘째로, 로그 정보를 원본 데이터에 적용하여 최종 데이터를 생성하는 과정이 추가된다.

표 2는 성능 분석에 사용되는 수치들을 나타낸다. *NW(Number of Write operations)*는 원본 프로그램에서의 쓰기 연산 실행 횟수를 의미하며, *NPF(Number of Pages consisting of a File)*는 파일을 구성하고 있는 페이지 개수를 의미한다. *RWP(Ratio of Written Pages)*는 원본 파일에서 쓰기 연산된 페이지의 비율을 나타내며, *RWPAR(Ratio of Written Pages After Reorganization)*는 재배치 파일에서 쓰기 연산된 페이지의 비율을 나타낸다. *TLI(Time to Log Information)*와 *TALRP(Time to Apply Log Records per Page)*는 로깅 기법으로 인한 추가 비용을 의미한다. *TLI*는 쓰기 정보를 로깅하는데 필요한 시간을 의미하며, *TALRP*는 한 페이지에 대한 로그 정보들을 원본 파일에 적용하여 최종 데이터를 생성하는데 필요한 시간을 의미한다.

원본 프로그램은 로깅 기법을 적용하지 않기 때문에, 이로 인한 추가 시간이 필요하지 않다. 반면, 로깅 프로그램과 최적화 프로그램은 로깅 기법으로 인한 비용이 추가된다. 로깅 프로그램과 최적화 프로그램에서의 로깅 횟수는 *NW*이다. 왜냐하면, 원본 프로그램에서의 쓰기 연산이 로깅 기법에서는 로그 정보로써 저장되기 때문이다. 로그 정보를 이용한 최종 데이터 생성 비용은 쓰기 연산 페이지 개수에 비례하여 증가한다. 이는 최종 데이터 생성이 페이지 단위로 발생하기 때문이다. 즉, 로깅 프로그램과 최적화 프로그램에서의 응용 프로그램 단계의 추가 시간은 아래와 같이 표현된다.

표 2 각 수치에 대한 기호

기호	의미
NW	*Number of Write operations*
NPF	*Number of Pages consisting of a File*
NPB	*Number of Pages in a Block*
NLB	*Number of Log Blocks*
RWP	*Ratio of Written Pages*
RWPAR	*Ratio of Written Pages After Reorganization*
TLI	*Time to Log Information*
TALRP	*Time to Apply Log Records per Page*

로깅 프로그램 : $NW*TLI + NPF*RWP*TALRP$

최적화 프로그램 : $NW*TLI + NPF*RWPAR*TALRP$

4.2.2 FTL 단계

원본 프로그램, 로깅 프로그램과 최적화 프로그램은 동일한 기능을 수행하지만, 쓰기 연산 패턴은 서로 다르다. FTL의 성능은 쓰기 연산 패턴에 영향을 받기 때문에, 각 프로그램들은 FTL에서 서로 다른 성능을 갖는다. 원본, 로깅, 최적화 프로그램들에서 쓰기 연산은 갱신 연산을 의미한다. 따라서 FAST는 쓰기 연산을 먼저 로그 블록에 저장하고, 추후 데이터 블록과 병합 연산[3]시킨다.

표 3은 FAST에서 각 프로그램에 의해서 요구되는 블록의 수와 병합 연산으로 인한 지움 연산 횟수를 보여준다. 본 실험에서의 쓰기 연산은 갱신 연산을 의미하기 때문에, 각 프로그램들의 쓰기 연산들은 로그 블록의 빈 페이지에 순차적으로 저장된다. 따라서 쓰기 연산 횟수가 증가할수록 많은 페이지가 필요하다. 원본 프로그램에서 쓰기 연산 횟수는 *NW*이고, 쓰기 연산 횟수와 변경 페이지의 횟수가 동일하다. 따라서 원본 프로그램에 의해서 요구되는 블록의 수는 $\lceil NW/NPB \rceil$로 표현된다. *NPB(Number of Pages in a Block)*는 블록이 포함하고 있는 페이지 개수이다. 로깅 프로그램과 최적화 프로그램은 각각 $NPF*RWP$개와 $NPF*RWPAR$의 페이지를 변경한다. 따라서 두 프로그램으로 인해서 변경되는 블록의 수는 각각 $\lceil (NPF*RWP)/NPB \rceil$과 $\lceil (NPF*RWPAR)/NPB \rceil$이다. 두 프로그램은 로깅 기법을 적용하여 원본 프로그램에 비해서 페이지를 적게 변경한다. 로깅 프로그램은 원본 파일을 사용하는 반면, 최적화 프로그램은 재배치 파일을 사용하기 때문에, 최적화 프로그램이 로깅 프로그램보다 적은 수의 페이지를 변경한다. 즉, "$NPF*RWPAR \leq NPF*RWP \leq NW$" 관계가 성립된다. 따라서 표 3을 통해서 재배치 파일을 사용하는 최적화 프로그램이 가장 적을 수의 블록을 요구하는 것을 알 수 있다.

FAST에서 병합 연산은 사용 가능한 로그 블록이 없을 때 발생한다. 실행 중에 요구되는 블록의 수가 로그 블록의 수보다 많으면 병합 연산이 발생한다. 병합 연산 발생 횟수는 "실행 중 요구되는 블록의 수 - *NLB*"로 표현된다. *NLB(Number of Log Blocks)*는 FAST의 로그 블록의 개수이다. 병합 연산은 새로운 로그 블록을

3) 해당 로그 블록과 데이터 블록들의 유효한 최신 데이터들을 새롭게 할당된 데이터 블록으로 이동시킨다[7].

표 3 쓰기 연산(갱신 연산)에 의해서 요구되는 블록의 개수 및 지움 연산 횟수 계산을 위한 모델

	프로그램에 의해서 요구되는 블록의 개수	병합 연산으로 인한 지움 연산 횟수
원본 프로그램	$\lceil NW/NPB \rceil$	$(\lceil NW/NPB \rceil - NLB) \times (1+\alpha)$ $if)\ NPB \le NPF,\quad 1 \le \alpha \le NPB$ $else)\quad 1 \le \alpha \le NPF$
로깅 프로그램	$\lceil (NPF \times RWP)/NPB \rceil$	$(\lceil (NPF \times RWP)/NPB \rceil - NLB) \times (1+\beta)$ $if)\ NPB \le NPF,\quad 1 \le \beta \le NPB$ $else)\quad 1 \le \beta \le NPF \times RWP$
최적화 프로그램	$\lceil (NPF \times RWPAR)/NPB \rceil$	$(\lceil (NPF \times RWPAR)/NPB \rceil - NLB) \times (1+\gamma)$ $if)\ NPB \le NPF,\quad 1 \le \gamma \le NPB$ $else)\quad 1 \le \gamma \le NPF \times RWPAR$

생성하기 위해 기존 로그 블록에서 유효한 페이지를 데이터 블록으로 복사한 후 해당 로그 블록을 지우는 것이다. 이 때, 유효한 페이지의 복사는 추가적인 지움 연산을 야기한다. 표 3의 α, β, γ는 병합 연산으로 발생하는 추가 지움 연산 개수를 나타낸다. 추가 지움 연산은 해당 로그 블록에 포함된 유효한 페이지들이 병합되는 데이터 블록들에서 발생한다. 만약 실행 중인 파일의 크기가 로그 블록 크기보다 크다면, 추가 지움 연산 횟수는 로그 블록 내 페이지 개수와 같거나 작다. 그러나 해당 파일이 로그 블록 크기보다 작다면, 추가 지움 연산 횟수는 쓰기 연산 페이지 개수와 같거나 작다. 따라서, 쓰기 연산 페이지가 가장 적게 포함하고 있는 재배치 파일을 사용하는 최적화 프로그램이 가장 적은 수의 추가 지움 연산을 발생시키는 것을 알 수 있다.

표 3은 최적화 프로그램이 가장 적은 블록을 필요로 하며, 병합 연산으로 인한 지움 연산 횟수도 가장 작은 것을 보여준다. 즉, 재배치 파일의 사용은 비용이 큰 지움 연산 감소를 통해서 성능을 향상시킨다.

4.3 실험 결과

제안된 기법이 적용된 데이터 재배치가 성능에 미치는 영향을 알기 위해 두 가지 실험을 하였다. 먼저 데이터 재배치 자체로 인한 성능 향상을 알기 위해서 여러 형태의 원본 파일을 재배치 정도를 변화시키면서 성능을 측정하였다. 그리고 재배치 조합 방법에 따른 성능 변화를 알기 위해서 동일한 원본 파일을 조합 영역들이 위치한 경로를 변경하며 성능을 측정하였다.

4.3.1 데이터 재배치로 인한 성능 향상

원본 파일은 512 KB이기 때문에, 총 1024 페이지로 구성된다. 원본 파일에 1024번의 쓰기 연산들을 발생시켰으며, 쓰기 연산들이 원본 파일의 모든 페이지에서 발생하는 경우(1024개 페이지)에서부터, 10개 페이지에 모든 쓰기 연산이 발생하는 경우(10개 페이지)로 변경하면서 성능을 측정하였다. 또한, 데이터 재배치로 인한 성능 개선을 알아보기 위해 *RWPAR*를 *RWP*의 10/100, 40/100, 70/100, 100/100 로 변경하면서 측정하였다.

그림 9는 쓰기 연산 분산도와 데이터 재배치가 성능에 미치는 영향을 보여준다. 쓰기 연산 분산도(*NPF/NW*)란 원본 프로그램에서 쓰기 연산들이 여러 페이지들에 분산된 정도를 수치화한 것이다. 예를 들어 1024개 쓰기 연산이 512개 페이지에 쓰기 연산되었다면, 쓰기 연산 분산도는 0.5가 된다. 원본 프로그램은 쓰기 연산 분산도에 영향을 받지 않는다. 원본 프로그램에서는 실행 중 쓰기 요청이 발생하며, 플래시 메모리에 쓰기 연산 실행 명령을 하기 때문이다. 반면 로깅 프로그램과 최적화 프로그램은 쓰기 연산 분산도가 감소할수록 성능이 개선된다. 로깅 프로그램과 최적화 프로그램은 동일 페이지에 대한 쓰기 연산들을 쓰기 연산 하나로 대체하기 때문이다.

다양한 비율로 변경된 재배치 데이터를 통해서 데이터 재배치로 인한 성능 향상을 알 수 있다. 그림 9에서 쓰기 연산 분산도가 동일한 경우, 데이터 재배치로 인한 성능 변화를 알 수 있다. 재배치된 파일에서 변경된 페이지가 원본 파일에서 쓰기 연산된 페이지에 비해서 작을수록 실행 시간이 감소된다. 이것은 데이터 재배치의 목적이 쓰기 연산 영역들을 가능하면 동일한 페이지 영역에 위치시켜 쓰기 연산되는 페이지의 수를 줄이는 것

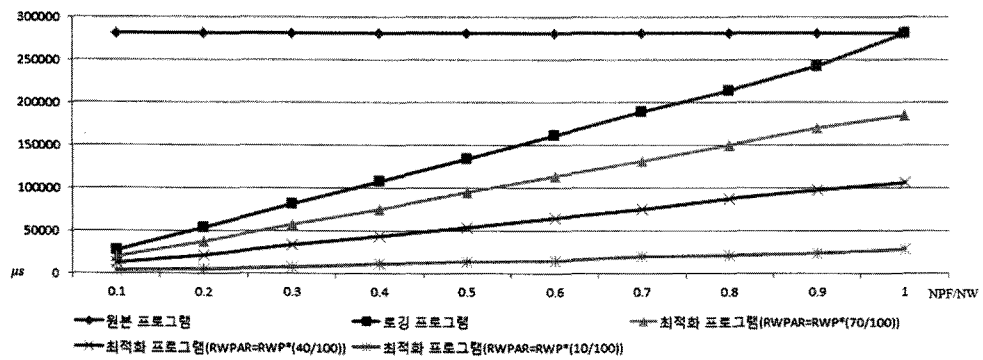


그림 9 쓰기 연산 분산도와 변경된 페이지에 의한 성능 측정 비교

이기 때문이다. 쓰기 연산 분산도가 클수록, 데이터 재배치의 성능 개선 효과가 커진다. 이것은 쓰기 연산 분산도가 클수록 동일 재배치 비율에 대한 재배치된 페이지의 개수가 증가하기 때문이다.

4.3.2 데이터 재배치 조합 결과에 의한 성능 변화

본 논문에서의 데이터 조합은 실행 전에 미리 파일 데이터를 변경시킨다. 따라서 실행 경로가 하나라면 문제가 되지 않지만, 실행 경로가 다수라면 데이터 재배치 조합 결과가 성능에 영향을 미친다. 예를 들어, 실행 가능 경로가 2개(경로 1, 경로 2) 있으며, 만약, 경로 1에 위치한 쓰기 영역들이 재배치 조합되고, 경로 1로 실행이 된다면, 최대 성능 향상을 얻을 수 있지만, 경로 2로 실행이 된다면, 예상된 성능 향상 효과를 얻지 못할 것이다.

이러한 데이터 재배치 조합 결과가 성능에 미치는 영향을 측정하기 위해, 실행 경로가 2개인 프로그램을 실험대상으로 하였다. 측정을 단순화하기 위해 원본 파일(512 KB)의 모든 페이지는 하나의 쓰기 가능 영역을 가진다고 가정하였다. 원본 파일(1024개 페이지)은 3종류 영역으로 분류된다. 영역 1(362개 페이지)은 경로 1을 통해서만 접근하고, 영역 2(362개 페이지)는 경로 2를 통해서만 접근 가능하다. 영역 3(300개 페이지)은 경로 1과 2에서 공통으로 접근 가능한 영역이다. 파일 데이터는 경로 별로 재배치되기 때문에, 경로 1과 경로 2에서 공통으로 접근하는 영역 3의 재배치 형태가 성능에 영향을 미친다. 원본 파일을 사용하였을 경우, 경로 1로의 실행은 662개(영역 1 + 영역 3) 페이지를 변경시키고, 경로 2로의 실행 역시 662개 페이지가 쓰기 연산된다. 반면, 영역 3의 쓰기 영역 중 N개가 영역 1에, 300-N개가 영역 2에 재배치된 파일을 사용하였을 경우, 경로 1은 362+(300-N)개 페이지를, 경로 2는 362+N개에 해당되는 페이지를 포함한다.

그림 10은 영역 3의 재배치 형태에 따른 성능 변화를 보여준다. 프로그램을 경로 1로만 실행 시켜 실행 시간을 측정하였다. 세로축은 원본 파일을 사용하였을 때의 실행 시간을 기준으로 재배치 파일을 사용하였을 때의 실행시간을 상대적인 비율로 표현하였다. 가로축은 영역 3에 있는 쓰기 데이터 중에서 영역 1로 재배치된 비율을 나타낸다. 예를 들어, 영역 3에 위치한 300개 페이지에 중에서 75개 페이지에 위치한 쓰기 데이터가 영역 1로 재배치된다면, 25%(75/300)이 되며, 반대로 75개 페이지에 위치한 쓰기 데이터가 영역 2로 재배치된다면, 75%(225/300)가 된다. 따라서 가로축의 값 100%는 영역 3에 위치한 모든 쓰기 영역들이 영역 1로의 재배치된 것을 의미하므로 362개 페이지만을 변경하고, 반면, 가로축의 값 0%는 영역 3에 위치한 모든 쓰기 영역들이

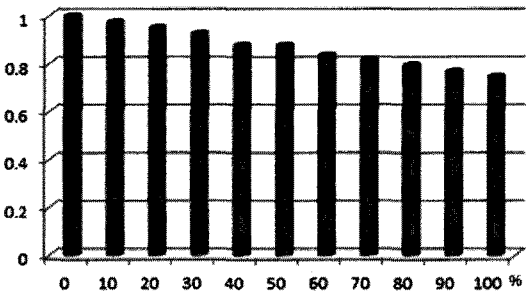


그림 10 공통 영역 재배치 결과가 성능에 미치는 영향

영역 2로 재배치된 것을 의미하므로, 662개 페이지에 대한 쓰기를 유발한다.

따라서 실제 수행 경로가 경로 1일 때는, 영역 3의 쓰기 영역들이 영역 1에 많이 배치될수록 성능이 개선되는 것을 볼 수 있다. 또한, 실행 경로를 중심으로 재배치된 부분이 전체 데이터의 60%가량이라면 20% 정도의 성능향상을 보이고 있다. 그리고 0%처럼 재배치 파일이 실제 실행 패턴과 일치하지 않더라도, 원본 프로그램보다 성능 저하가 없는 것을 확인할 수 있다.

5. 결 론

플래시 메모리는 이동성, 에너지효율성, 신뢰성 등의 특성 때문에 소형 정보기기의 저장 장치로써 유용하게 활용되고 있다. 그러나 이러한 플래시 메모리는 읽기 연산에 비해 쓰기 연산의 비용이 상대적으로 크므로 쓰기 연산 횟수 감소에 대한 연구가 필요하다.

본 논문에서는 플래시 메모리 환경에서 프로그램 분석 및 재 작성 기법을 이용한 효율적인 쓰기 연산을 위한 데이터 재배치 기법에 적용되는 프로그램 분석 기법을 제안하였다. 먼저, 쓰기 연산의 대상이 되는 데이터 범위를 감소시키기 위해 프로그램을 정적으로 분석하여 쓰기 연산 가능한 데이터를 가급적 동일 페이지 내에 배치시키는 프로그램으로 재 작성할 수 있도록 한다. 이때 프로파일링 결과를 정적 분석에 결합시켜 분석 결과에 따른 배치가 최대한으로 효율적인 데이터 배치가 될 수 있도록 연구하였다. 이후, 이렇게 수정된 프로그램을 통해 데이터를 관리하면, 데이터의 포맷을 플래시 메모리의 성질에 맞게 수정하여 관리하는 효과를 가지게 되어, 성능 향상을 얻을 수 있었다.

제안하는 방법은 실험 결과를 통해서 쓰기 연산 분산도가 작을수록, 데이터 재배치 최적화가 이루어질수록 성능 향상을 보일 뿐만 아니라, 쓰기 연산으로 인해서 소비되는 추가적인 영역도 감소시키는 것을 보았다. 현재는 영역 최적화 조합 알고리즘을 보다 정교하고 효과적으로 개선할 수 있도록 경로 조합의 목적 함수 및 알

고리즘 개선에 대해 연구 중에 있으며 임의의 입력 값에 대한 정교한 분석 방법을 연구하여 보다 높은 성능 향상이 가능하도록 하는 것을 목표로 하고 있다.

참 고 문 헌

[1] M-Systems. "Two technologies compared: NOR vs. NAND" White Paper, 91-SR-012-04-8L, Rev 1.1, July 2003.

[2] Canon FS200 Flash Memory Camcorder, http://www.alibaba.com.

[3] Samsung Electronics Co., "NAND flash memory & smartMedia data book," 2004.

[4] Tae-Sun Chung et al, "System Software for Flash Memory: A Survey," *EUC2006*, pp.394-404, Seoul, Korea, 2006.

[5] Y.-G. Lee, D. Jung, D. Kang and J.-S. Kim, "u-FTL: a memory-efficient flash translation layer supporting multiple mapping granularities," in *Proceedings of 8th ACM international conference on embedded software*, pp.21-30, Atlanta, GA, USA, Aug. 2008.

[6] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. "A Space-Efficient Flash Translation Layer for CompactFlash Systems," *IEEE Transactions on Consumer Eletronics*, vol.48, no.2, pp.366-375, May 2002.

[7] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park and H.-J. Song, "A log buffer-based flash translation layer using fully associative sector translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol.6, Issue 3, July, 2007.

[8] Tae-Sun Chung et al., "A survey of Flash Translation Layer," *Journal of Systems Architecture 55(2009)*, pp.332-343, 2009.

[9] Steven S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, 1997.

[10] CORMAC FLANAGAN and MATTHIAS FELLEISEN, "Componential set-based analysis," in *Proceedings of PLDI'97*, Las Vegas, Nevada, USA, 1997.

[11] FLEMMING NIELSON, HANNE RIIS NIELSION and CHRIS HANKIN, Principles of Program Analysis, Springer, 2005.

[12] Chanik Park et al., "Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory," in *Proceedings of the 4th ACM international conference on Embedded Software*, Pisa, Italy, 4, 2004.

[13] Chun-Chieh Lin et al., "Source Code Arrangement of Embedded Java Virtual Machine for NAND Flash Memory," in *Proceeding of the ISCIT'07*, Sydney, Australia, 2007.

[14] Joon-Young Paik, Eun-Sun Cho and Tae-Sun Chung, "Performance Improvement for Flash Memories using Loop Optimization," in *Proceedings of 12th IEEE International Conference on Computational Science and Engineering(CSE' 09)*, vol.2, pp.508-513, Vancouver, Canada, 2009.

[15] Joon-Young Paik, Eun-Sun Cho and Tae-Sun Chung, "Reorganizing Data Blocks in Flash Memory by Program Translation," in *Proceedings of 10th IEEE International Conference on Computer and Information Technology*, Bradford, UK, 2010.

[16] Andrew W. Appel, Modern Compiler Implementation: In ML, Cambridge University Press, New York, NY, 1998.

[17] Hanne Riis Nielson, Flemming Nielson, Semantics with applications: a formal introduction, John Wiley & Sons, Inc., New York, NY, 1992.

[18] Dehydra Home, https://developer.mozilla.org/en/Dehydra.

백 준 영

2008년 충남대학교 전기정보통신공학부 졸업(학사). 2010년 충남대학교 컴퓨터공학과 졸업(석사). 2010년~현재 충남대학교 컴퓨터공학과 박사 과정. 관심분야는 플래시 메모리, 프로그램 분석

조 은 선

1991년 서울대학교 계산통계학과 졸업(계산학전공). 1993년 서울대학교 전산과학과 석사. 1998년 서울대학교 전산과학과 박사. 1999년~2000년 한국과학기술원 연구원. 2000년~2001년 아주대학교 정보통신전문대학원 조교수 대우. 2002년~2006년 충북대학교 조교수. 2006년~현재 충남대학교 컴퓨터공학과 조교수. 관심분야는 프로그램 분석, 상황데이터 모델링 및 언어