論文97-34C-2-6

여분소자 라인을 이용한 배열구조의 재구성 방법

(Reconfiguration Method for Array Structures using Spare Element Lines)

金 炯 奭 *, 崔 相 昉 **

(Hyoung Seok Kim and Sang Bang Choi)

요 약

여분소자의 행과 열을 사용하여 배열구조 메모리를 재구성하는 방법은 수율을 증가시키는 유용한 방법으로 사용되어 왔다. 그러나 사용 가능한 여분소자의 행과 열의 수가 각각 제한되어 있는 재구성 문제는 NP-complete으로 알려져 있다. 본 논문에서는 고장소자를 포함하지 않는 행과 열을 제거하는 고장소자의 집단화를 이용하여 배열구조를 재구성하는 알고리즘을 제안한다. 기존의 그리디 알고리즘은 가장 간단한 복구 방법으로서 고장소자의 수가 n일 때 $O(n^2)$의 시간 복잡도를 갖지만 복구율은 매우 낮다. 그리고 과도 검색 알고리즘은 모든 가능한 경우에 대하여 재구성을 시도하므로 복구율은 매우 높지만 $O(2^n)$의 시간 복잡도를 갖는다. 본 논문의 알고리즘은 거의 모든 경우에 대해 과도 검색 알고리즘과 같은 복구율을 제공하지만 시간 복잡도는 최악의 경우 $O(n^3)$으로, 대부분의 경우 그리디 방법과 거의 같은 시간에 배열구조를 재구성 할 수 있다. 시뮬레이션을 통하여 재구성 수행시간과 복구율에 대하여 다른 알고리즘과 비교하였다.

Abstract

Reconfiguration of a memory array using spare rows and columns has been known to be a useful technique to improve the yield. When the numbers of spare rows and columns are limited, respectively, the repair problem is known to be NP-complete. In this paper, we propose the reconfiguration algorithm for an array of memory cells using faulty cell clustering, which removes rows and columns without any fault to condense overlapped faulty cells into a rectangular cluster. The previous greedy algorithm is the simplest reconfiguration method with the time complexity of $O(n^2)$, where n is the number of faulty cells, however the repair rate is very low. Whereas the exhaustive search algorithm has a high repair rate, but the time complexity is $O(2^n)$. The proposed algorithm provides the same repair rate as the exhaustive search algorithm for almost all cases and runs as fast as the greedy method. It has the time complexity of $O(n^3)$ in the worst case. We show that the proposed algorithm provides more efficient solutions than other algorithms using simulations.

I. 서 론

오늘날 16M나 64M DRAM 등과 같은 배열구조를 갖는 고집적도의 VLSI(very large scale integration)/WSI(wafer scale integration)가 생산되고 있다. 그러나 집적도가 커짐에 따라 기술의 한계로 인해 수율은 낮아지며, 이 수율을 높이기 위한 방법이 많이 연구되고 있다. 특히 고장소자들이 발생하여도 여분소자를 써서 재구성(reconfiguration)하여 동작을 가능하게 하는 고장허용(fault tolerant) 재구성 구조에 대한 연구가 활발하다.

재구성 방식을 사용하는 시스템은 프로세서 배열구조와 메모리 배열구조로 나눌 수 있는데, 프로세서 배

* 正會員, (주) 데이콤
(DACOM Corp.)
** 正會員, 仁荷大學校 電子工學科
(Dept. of Electronic Eng., Inha Univ.)
接受日字:1996年2月5日, 수정완료일:1997年2月12日

열구조는 각 프로세서간의 연결 구조를 여분의 트랙과 스위치를 사용하여 고장난 프로세서들을 고장나지 않은 정상 프로세서로 대치한다. 그러나 메모리 배열구조에서는 각 소자들이 크기가 매우 작은 콘덴서 및 트랜지스터로 단순하게 구성되어 있기 때문에, 프로세서 배열구조에서와 같이 각 소자들을 트랙과 스위치 등을 사용하여 복잡하게 연결할 수가 없다. 따라서 이러한 메모리 배열구조에서는 고장소자를 포함하는 한 행이나 열 전체소자를 단순히 여분소자의 행이나 열로 대치하는 것이 효과적인 방법이 된다.

본 논문의 목적은 고장허용 재구성 구조를 갖는 메모리에서 각 소자들 간의 상호 연결 구조를 논리적으로 유지하면서 고장소자들을 행 또는 열 단위의 여분소자들로 대치시켜, 원래대로 전체 시스템 동작을 가능하게 해 줄 수 있는 효율적인 알고리즘을 찾는 것이다. 이러한 방법은 소위 "집합적 스위칭(set switching)" 복구 방법으로 알려져 있다. 이것의 단점은 여분소자의 소비가 크다는 것이지만 복구 방법이 비교적 간단하기 때문에, 이 단순성이 충분히 여분소자의 낭비를 보상할 수가 있어 메모리와 같은 고집적도를 갖는 배열구조에서 많이 사용되고 있다[1] [2]. 그러나 사용 가능한 여분소자의 행과 열의 수가 각각 제한되어 있는 재구성 문제는 NP-complete으로 알려져 있다[3].

지금까지 제안된 배열구조의 재구성 방법으로는 Tarr의 그리디 방법(greedy method)[4], Day의 과도 검색 알고리즘(exhaustive search algorithm)[5], Kuo & Fuchs의 브랜치 앤 바운드 알고리즘(branch and bound algorithm)[3], Hasan의 critical-set을 이용한 방법[6], 그리고 몇 가지의 경험적인(heuristic) 방법들이 있다. Tarr의 그리디 방법은 Mera II (memory error-repair analyzer)라는 시스템을 사용하여, 먼저 여분소자의 행과 열 중에서 가장 많이 남아 있는 것을 선택한 후, 그 행이나 열 방향에서 고장소자들이 가장 많이 존재하는 라인(line)부터 제거한다. 만일 여분의 행과 열의 수가 같으면 행과 열에 관계없이 고장소자가 가장 많은 라인부터 제거한다. 이 방법은 일종의 그리디 방법으로서 고장소자의 수를 n이라고 할 때 시간 복잡도(time complexity)는 $O(n^2)$으로 적지만, 고장소자의 패턴에 따라 시스템이 복구될 수 있는데도 그 방법을 찾지 못하는 경우가 많이 있다.

Day의 과도 검색 알고리즘은 첫번째 단계인 강제적 복구 분석(forced-repair analysis)에서 각 행에 대하여 고장소자들의 갯수가 여분의 열의 수보다 많으면 예비 행을 사용하여 우선 복구한다. 각 열에 대해서도 똑같은 방법을 적용한다. 두번째 단계인 희소-복구 분석(sparse-repair analysis)에서는 첫번째 단계에서 복구되지 않고 남아 있는 고장소자들에 대해 모든 경우를 검색하면서 선택적으로 복구를 해 나간다. 첫번째 단계를 수행하기 앞서 복구에 필요한 소자가 여분의 소자들보다 더 많으면 초기에 중단하는 기술(early abort technique)도 사용하고 있다. 그러나 이 알고리즘에서 강제적 복구 분석은 일종의 그리디 방법이며, 희소 복구 분석에서는 거의 모든 경우를 찾아 재구성을 시도하기 때문에 $O(2^n)$정도의 많은 시간이 걸린다.

Kuo & Fuchs의 브랜치 앤 바운드 알고리즘은 첫번째 단계에서 먼저 반드시 복구해야 할 것을 분석(must-repair analysis)하는데, 이것은 Day의 강제적 복구 분석과 같은 방법을 사용하고 있다. 그리고 두번째 단계에서는 Day의 알고리즘에서 소개된 초기에 중단하는 방법을 사용한다. 마지막 세번째 단계에서는 브랜치 앤 바운드 알고리즘을 사용하여 복구 방법을 찾게 된다. 이 단계는 Day의 희소-복구 분석과 비슷한 방법을 사용하지만 검색 범위를 제한하면서 복구 방법을 찾게 된다. 그러나 대부분의 경우 검색 범위가 크게 줄어들지 않으며, 또한 제한적인 검색을 위해서 시간함수를 사용하여 검색된 데이타를 정렬하고 검색 범위의 양을 줄이기 위해 데이타를 비교하여 같은 것은 제거하므로, 경우에 따라서 이러한 과정들이 오히려 알고리즘의 수행시간을 크게 증가시킬 수 있다. 따라서 전체 수행시간은 최악의 경우 $O(2^n \cdot n^2)$정도가 된다.

위에서 설명한 방법들은 대부분의 경우 복구율이 낮거나 수행시간이 많이 소비된다는 단점이 있다. 본 논문에서는 이러한 문제를 직관적 이해가 쉬운 배열구조상의 직교좌표에서 고장소자의 집단화를 이용하여 설명을 하고, 이것을 다시 알고리즘으로의 구현이 용이한 이분그래프(bipartite graph)[7]로 변환하였다. 제안된 알고리즘에서는 변환된 이분그래프의 최대 매칭(maximum matching)[8]을 이용한 초기 중단 방법을 사용하여 알고리즘의 수행시간을 감소시켰으며, 거의 모든 경우에 대해 검색하는 대부분의 기존 방식과는 달리 집단 연결 소자를 이용함으로써 매우 효율적으로 배열구조를 재구성할 수 있다.

본 논문은 다음과 같이 구성되어 있다. II장에서는 본 연구에서 제안한 배열구조 메모리의 재구성 알고리

즘을 직교좌표와 이분그래프의 두 가지 모델을 사용하여 제안한다. III장에서는 고장소자를 포함하는 배열구조에서 복구가 가능하기 위한 필요조건과 충분조건에 대해 설명한다. IV장에서는 고장소자들이 발생할 수 있는 패턴에 대해 설명한 후, 본 논문의 방법과 기존의 방법을 여러 가지의 고장소자 패턴에 대해 시뮬레이션을 통하여 성능을 비교 평가한다. 마지막으로 V장에서 결론을 맺는다.

II. 배열구조 메모리의 재구성 알고리즘

1. 직교좌표를 사용한 배열구조의 모델링

본 장에서는 배열구조 메모리의 재구성 문제를 이해가 쉬운 배열구조상의 직교좌표로 모델링한 후, 이분그래프를 사용하여 구체적으로 알고리즘을 구현하였다. 배열구조에서 고장소자가 존재하지 않는 각 좌표축 상의 점을 제거하여 고장소자들을 좌표축 상에 논리적으로 인접하게 놓는 과정을 고장소자의 집단화(faulty-cells clustering)라고 정의한다. 이러한 고장소자의 집단화는 좌표를 압축(condensing)하는 효과를 나타낸다. 각 고장소자들 사이에 위치한 정상적으로 동작하는 행이나 열은 제거한 후, 임의의 두 고장소자들의 행 또는 열의 위치가 서로 중복되면 그것들을 하나의 집단 고장소자로 간주한다. 즉 행 또는 열 방향으로 중복된 고장소자들을 하나의 집합으로 묶는다. 이렇게 압축한 고장소자들의 집단은 매우 효율적인 재구성 방법을 제시하여 준다. 그림 1의 (b)는 고장소자를 포함한 배열구조 (a)를 집단화한 결과를 그린 것이다.

각 집단화된 고장소자들은 직사각형의 형태로 나타낼 수 있으며, 가로와 세로의 길이 중 긴 쪽을 b_i, 짧은 쪽을 a_i라고 할 때 복구 효율은 $P_i=b_i/a_i$로 정의한다. 복구 효율은 긴 쪽의 길이를 짧은 쪽의 길이로 나눈 값으로써 항상 1보다 크거나 같다. 본 절에서 제안한 직교좌표를 사용한 재구성 방법은 다음과 같은 단계로 수행된다.

1 단계 : 배열구조의 직교좌표에서 고장소자를 포함하지 않는 행과 열을 제거한 후 고장소자들을 집단화한다.

2 단계 : 각 집단화된 고장소자들에 대하여 복구 효율을 구한 후, 복구 효율이 가장 큰 고장소자 집단을 선택한다.

3 단계 : 선택된 고장소자 집단의 가로나 세로중 길이가 긴 쪽을 결정한 후, 고장소자들이 가장 많은 한 라인을 복구한다.

4 단계 : 복구된 소자를 제외한 나머지 고장소자만을 사용하여 1 단계부터 3 단계까지 모든 고장소자들이 전부 복구될 때까지 반복한다.

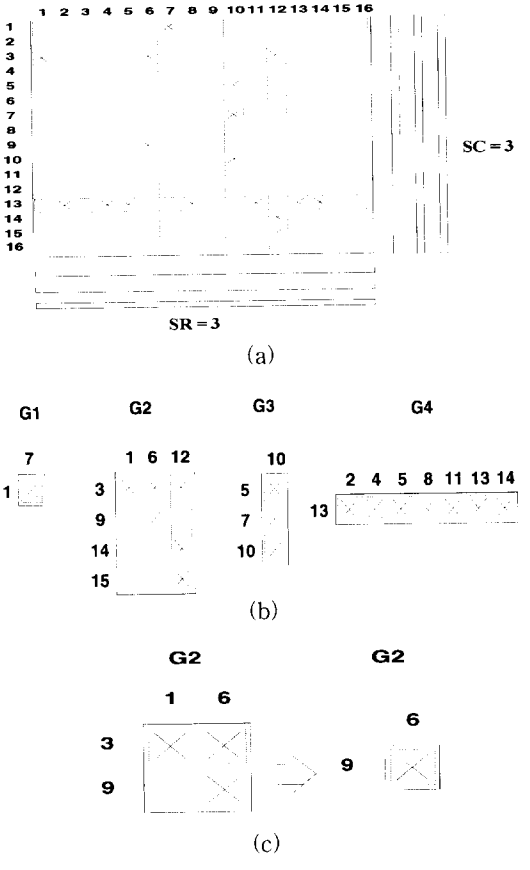


그림 1. 16x16 배열구조에서 고장 소자의 집단화를 이용한 복구 방법

Fig. 1. Reconfiguration of 16x16 array using faulty-cell clustering.

그림 1은 위의 방법을 예를 들어 설명한 것이다. 그림 (b)는 고장소자를 포함하는 배열구조 (a)를 4개의 고장소자 집단으로 압축한 것이다. 그림 (a)에서 SR과 SC는 여분소자의 행과 열의 수를 각각 나타낸다. 만일 여분소자 라인이 행과 열에 대한 제한이 없이 주어진다면, 각 고장소자 집단은 서로 분리되어 있기 때문에 각 집단에서의 최적의 복구 방법이 전체에서도 최적의 방법이 되나, 여분소자의 행과 열의 수가 각각 제한이 있을 때에는 그렇지 않다. 이러한 제한을 효과적으로 해결하기 위하여 각 고장소자 집단의 복구 효율을 비교

하여 복구 효율이 가장 높은 고장 집단을 먼저 복구한다. G_i의 복구효율을 EG_i로 표시하면 G_1의 복구 효율 $EG_1 = 1/1$, $EG_2 = 4/3$, $EG_3 = 3/1$, $EG_4 = 7/1$이므로, G_4, G_3, G_2, G_1의 순서로 복구를 한다. 복구 효율이 가장 큰 고장 집단을 선택한 후, 그 고장 집단의 가로나 세로 중에 길이가 긴 쪽을 택하여 복구를 시작한다. 만일 여분소자의 행, 열이 모두 사용 가능하고 집단화된 고장소자의 가로보다 세로의 길이가 더 길면, 최적의 방법은 항상 세로에서 찾게 된다. 즉 여분소자의 행 또는 열이 적게 사용되는 방향에서 복구를 시작한다. 예를 들면 G_2에서는 가로보다 세로가 더 길므로 열 방향으로 먼저 복구하게 된다. 다음 단계는 선택된 방향에서 고장소자들이 가장 많이 존재하는 라인을 선택하여 복구한다. 이 라인은 여분소자에 대한 고장소자의 밀도가 가장 높다. 그림 (b)에서는 G_4의 R_{13}을 복구한다. 본 논문에서 R_i는 i번째 행을 C_j는 j번째 열을 나타낸다. 그 다음 다시 고장소자들을 집단화시켜 (그림 (b)의 경우는 G_4가 한번에 복구됨으로 다시 집단화할 필요가 없음) 복구 효율이 가장 큰 고장소자 집단 G_3를 선택한다. G_3는 세로의 길이가 더 길므로 C_{10}을 복구한다. G_2는 세로가 더 길므로 세로에서 고장소자가 가장 많은 C_{12}를 복구하며, G_2의 남은 고장소자들을 더 작은 집단으로 압축한다(그림 (c) 참조). 이때 경우에 따라 한 고장소자 집단이 두 개 이상의 작은 집단으로 나뉠 수도 있다. 다시 G_2의 R_3을 복구하면 남은 G_2와 G_1은 각각 한 개의 고장소자만을 포함하게 되고, 여분소자의 행과 열도 각각 한 개씩 남아 있으므로 임의로 하나씩 복구하면 된다. 그림 1의 (a)는 17개의 고장소자를 포함하는 16×16 배열구조를 각각 3개의 여분소자 행과 열을 사용하여 재구성한 결과를 나타낸 것으로, 점선으로 표시된 행과 열은 여분소자 라인으로 대치된다. 그림 1과 같이 기존의 알고리즘에서는 해결하지 못하는 경우에 위 방법을 적용하여 본 결과, 대부분 매우 효율적인 복구 방법을 제시하였다.

2. 이분그래프를 사용한 재구성 알고리즘

배열구조의 재구성 문제는 이분그래프를 사용하여 표현할 수 있다[5] [8]. 이분그래프에서 노드집합은 두 부분집합 N_1과 N_2로 나뉘어지고, 모든 링크는 N_1의 한 노드와 N_2의 한 노드를 서로 연결한다. 재구성 가능한 배열구조의 복구문제에서 각 행은 N_1상의 한 노드로 표시하고, 각 열은 N_2상의 한 노드로 표시한다. 예를 들면 그림 2의 (b)에서 $R_1 \in N_1$은 그림(a)의 첫 번째 행을 $C_2 \in N_2$는 두번째 열을 나타낸다. 배열구조에서 고장소자 (i, j)는 노드 R_i와 C_j를 연결하는 링크에 해당한다. 본 논문에서는 이분그래프를 $BG = (N_1, N_2, E)$로 나타낸다. 여기서 E는 고장소자를 나타내는 링크의 집합이다. 배열구조의 재구성 문제는 이분그래프에서 모든 링크와 인접하는 최소 노드의 집합(node cover)을 찾는 문제로 변형할 수 있다. 직교좌표를 이용한 모델링에서는 x나 y축이 서로 중첩되어 있는 고장소자들을 묶어 집단화시켰는데, 이분그래프에서는 집단화한 고장소자들이 연결 컴포넌트(connected component)로 나타난다. 그림 2의 (b)는 (a)의 12개 고장소자를 이분그래프의 링크로 나타낸 것이며, 그림 (c)의 G_1과 G_2는 그림 (b)의 이분그래프를 구성하는 두 연결 컴포넌트를 그린 것이다.

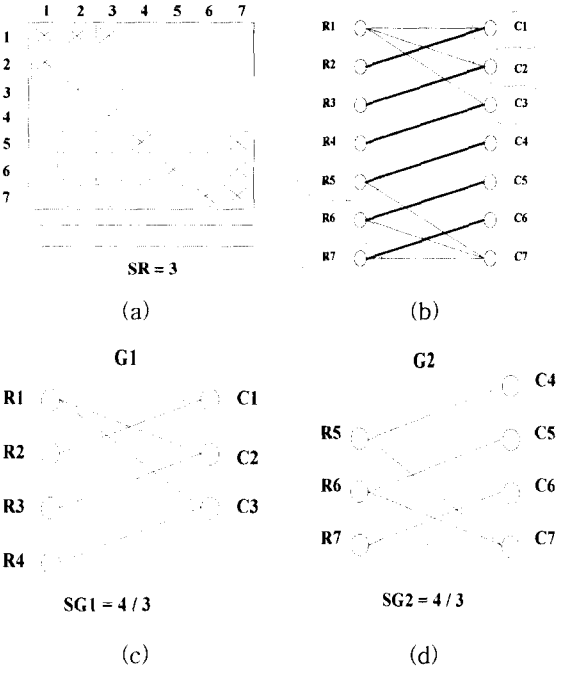


그림 2. 이분 그래프를 이용한 배열구조의 재구성 방법

Fig. 2. Reconfiguration of an array using bipartite graph.

연결 컴포넌트 G_i에서 행을 나타내는 노드의 집합은 $GR_i \leq N_1$, 열을 나타내는 노드의 집합은 $GC_j \leq N_2$로 표시한다. 본 논문에서 $|GR_i|$는 집합 GR_i의 노드 수를 나타내고, $|GC_j|$는 GC_j의 노드 수를 나타낸다. 여기서 노드 수는 복구할 때 사용되는 여분소자의 행이

나 열의 갯수를 암시한다. 각 G_i의 노드들은 서로 중첩되지 않기 때문에, 전체 노드 집합 N_1과 N_2는 각각 GR_i과 GC_i의 합과 같다. 즉 $N_1 = \cup_i GR_i$, $N_2 = \cup_i GC_i$. 만일 $|GR_i| > |GC_i|$라고 하면, 복구 효율은 $EG_i = |GR_i|/|GC_i|$가 된다. 앞 절에서 설명한 방법과 같이 고장소자를 포함하는 배열구조의 재구성은 복구 효율이 가장 큰 컴포넌트 G_i부터 시작한다. 각 컴포넌트를 복구하는 최적의 방법은 컴포넌트내의 모든 링크와 인접하는 최소 노드의 집합, 즉 노드 커버를 찾으면 된다. 일반 그래프에서 이러한 집합을 찾는 문제는 NP-Complete으로 알려져 있으나, 이분그래프에서는 그렇지 않으며 또한 König-Egerváry 정리로부터 최대 매칭 문제와 같아진다[8].

행과 열에 관계없이 배열구조를 복구할 수 있는 여분소자 라인의 최소 갯수는 각 연결 컴포넌트 G_i의 노드 커버의 합과 같다. 이분그래프를 사용한 배열구조의 재구성 알고리즘은 다음과 같은 단계로 실행된다.

1 단계 : 고장소자를 포함하는 어레이 구조를 이분그래프로 표현한다.

2 단계 : 각 연결 컴포넌트 G_i에 대하여 최대 매칭을 수행한다. 모든 G_i에 대한 최대매칭의 크기의 합이 여분소자의 행과 열의 수를 더한 것보다 크면 복구 수행을 중단한다(early abort). 주어진 여분소자 라인에 비해 고장소자의 수가 너무 많을 때는 어떠한 방법을 사용해도 복구가 불가능하기 때문에, 나머지 단계를 수행할 필요없이 초기에 중단하여 알고리즘의 수행시간을 줄이기 위한 것이다. 이것은 다음 장에서 복구 방법이 존재하기 위한 필요 조건이 된다.

3 단계 : 각 연결 컴포넌트 중에서 복구 효율이 가장 큰 G_i를 선택한다.

4 단계 : 선택된 G_i의 GR_i와 GC_i 중에서 작은 집합을 선택하여, 그 안에서 가장 많은 링크와 연결된 노드를 찾아 제거한다. 이분그래프에서 링크는 고장소자를 의미함으로, 해당되는 고장소자들이 여분소자 라인에 의하여 복구되는 것을 의미한다.

5 단계 : 3, 4 단계의 과정을 여분소자의 행 또는 열이 모두 사용될 때까지 또는 모든 링크가 제거될 때까지 계속 반복한다.

6 단계 : 만일 여분의 행 또는 열 중에 어느 한 쪽만을 모두 사용했다면, 나머지 여분 소자 라인을 사용하여 복구한다.

이분그래프를 사용한 재구성 알고리즘에서는 최대 매칭을 이용한 초기 중단방법을 적용하기가 용이하다. 그림 2의 (a)는 Kuo & Fuchs가 예로 든 문제로 기존의 그리디 방법으로는 해결되지 않는다. 그리디 방법은 가장 고장소자가 많은 것을 우선적으로 선택하므로, 이 경우에는 먼저 1번째 행과 7번째 열을 복구하게 된다. 그러나 남은 고장소자들을 복구하기 위해서는 6개의 여분소자 라인이 필요하나, 실제로 남은 것은 4개 라인이다. 고장소자를 포함하는 배열구조 (a)를 그림 3의 알고리즘에 따라 수행하면 다음과 같다. 그림 (b)는 (a)를 이분그래프로 변형한 것으로, 먼저 최대 매칭을 수행하여 굵은 선으로 표시하였다. 최대 매칭의 크기와 여분소자 라인의 합이 모두 6으로 같기 때문에, 복구가 가능한 것으로 판단되어 다음 단계를 계속 수행한다. 이 그래프를 연결 컴포넌트로 분리하여 그리면 그림 (c)의 G_1과 G_2가 얻어진다. 각 컴포넌트의 복구 효율은 $EG_1 = EG_2 = 4/3$로 같으므로, 임의로 G_1을 선택한다. 4단계에서 $|GC_1| < |GR_1|$이므로 GC_1에서 가장 많은 링크와 연결된 노드를 찾아 복구한다. 그림 (c)의 RC_1에서 C_1, C_2, C_3와 연결된 링크의 수는 모두 두 개로 같으므로 임의로 C_1을 선택한다. G_1에서 C_1과 이에 연결된 링크를 제거한 것을 G'_1으로 나타내면, G'_1과 G_2의 복구율은 $EG'_1 = 3/2$, $EG_2 = 4/3$이 되어 3단계에서 다시 G'_1이 선택된다. 이러한 방법으로 알고리즘을 계속 수행하면, 우선 G_1에서 C_1, C_2, C_3을 3개의 여분소자 열을 사용하여 각각 복구하게 되며, G_2에서는 R_5, R_6, R_7을 3개의 여분소자 행을 사용하여 각각 복구하게 된다.

이분그래프 $BG = (N_1, N_2, E)$에서 최대 매칭 알고리즘의 시간 복잡도는 일반적으로 $O(min(|N_1|,|N_2|) \cdot E)$이므로, 고장소자의 수가 n일 경우는 $O(n^2)$이 된다. 3단계에서 복구 효율이 가장 큰 G_i를 선택하고, 4단계에서 가장 많은 링크와 연결된 노드를 찾아 제거한 후 G_i를 재구성하는데 $O(n^2)$의 시간이 필요하다. 이런 과정을 매번 수행할 때마다 적어도 한 개 이상의 고장소자가 복구되며, 최악의 경우 n번 알고리즘을 수행하게 된다. 따라서 전체 알고리즘의 수행시간은 최악의 경우 $O(n^3)$이 된다. 이것은 본 논문에서 제안한 알고리즘이 빠른 시간 내에 수행될 수 있음을 나타내며, 4장의 시뮬레이션에서 분명해진다.

```
Algorithm Reconfiguration;
begin
    transform_problem_to_biograph();
    /* test for early abort */
    if ((sr+sc) >= maximum_matching())
    begin
        /* normal process when both sr and sc are
        available */
        while ((fault_cont > 0)and(sr > 0)and(sc > 0))
        begin
            clustering_faults();
            find_the_most_effective_cluster();
                /* of the repair rate */
            decide_direction_for_repairing();
            find_the_max_faults_line_and_remove();
        end
        /* residual process when faults are remained
*/
        if ((fault_count > 0)and((sr > 0)or(sc > 0)))
        begin
            find_the_max_faults_line_and_remove();
        end
        if( fault_count <= 0 )
            final_result = SUCCESS;
        else
            final_result = FAIL;
    end
    else /* early abort */
        final_result = FAIL;
end
```

그림 3. 재구성 알고리즘
Fig. 3. Reconfiguration algorithm.

Ⅲ. 복구 가능 조건

일반적으로 집단화한 고장소자 G_i를 복구할 수 있는 방법은 여러 가지가 존재할 수 있고, 이러한 복구 가능한 각 방법을 원소로 갖는 집합을 P_i로 표시한다. 집합 P_i의 한 원소 $(R_{i,j},\ C_{i,j})$, $j = 1, 2, \ldots, m$은 $R_{i,j}$ 개의 예비 행과 $C_{i,j}$ 개의 예비 열을 사용하여 G_i를 복구하는 방법을 나타낸다. 각 원소들 간에는 다음과 같은 조건을 만족한다고 가정한다.

$$(R_{i,j} + C_{i,j}) < (R_{i,j+1} + C_{i,j+1}) \quad j = 1,\ 2,\ \cdots,\ m-1$$

즉 고장소자 집단 G_i를 복구할 때, 각 복구 방법에서 사용되는 여분소자 라인의 합이 작은 순서대로 각 원소들을 정렬해 놓는다. 다음은 고장소자 집단 G_i에 대한 복구 방법 P_i를 나타낸 것이다.

$$P_i = \{(R_{i,1},\ C_{i,1}),\ (R_{i,2},\ C_{i,2}),\ \cdots\}$$
$$(R_{i,1}+C_{i,1}) < (R_{i,2}+C_{i,2}) < \cdots$$

각 집단화한 고장소자의 복구 방법 P_i에서 첫번째 원소인 $(R_{i,1},\ C_{i,1})$을 사용하여 복구하면, 전체적으로 최소의 여분소자 라인 S_{min}을 사용하여 복구하는 방법이 된다.

$$S_{min} = \sum_i R_{i,1} + \sum_i C_{i,1} = SR_{min} + SC_{min}$$

각 집단화한 고장소자를 최소의 여분소자 라인을 사용하여 복구할 때, 필요한 여분소자의 행과 열의 수와 배열구조에서 주어진 행과 열의 수 간의 관계는 다음과 같이 세 가지로 분류할 수 있다.

첫번째는 절대 복구 가능한 경우로서, 최소의 여분소자 행과 열의 수가 주어진 여분소자 행과 열의 수보다 각각 작기 때문에 복구 방법이 항상 존재한다.

$$SR_{min} \le C_R\ ,\quad SC_{min} \le C_C \tag{1}$$

여기서 C_R과 C_C는 각각 주어진 여분소자의 행과 열의 수를 나타낸다.

두번째는 절대 복구 불가능한 경우로서, 복구에 필요한 최소의 여분소자 행과 열의 수가 주어진 여분소자 행과 열의 수보다 각각 커서 복구 방법이 결코 존재할 수 없다.

$$SR_{min} > C_R\ ,\quad SC_{min} > C_C \tag{2}$$

세번째는 선택적 복구 가능한 경우로서, 필요한 최소의 여분소자 행이나 열 중 어느 하나만이 주어진 행 또는 열의 수보다 커서 경우에 따라 복구가 가능할 수도 있고 그렇지 않을 수도 있다.

$$SR_{min} > C_R,\ SC_{min} < C_C \text{ 또는 } SR_{min} < C_R,\ SC_{min} > C_C \tag{3}$$

고장소자를 포함하는 배열구조에 대하여 복구 가능한 방법이 존재하기 위한 필요조건은 다음과 같다.

$$SR_{min} + SC_{min} < C_R + C_C \tag{4}$$

고장소자를 포함하는 배열구조를 복구할 수 있는 방법이 존재하기 위해서는 절대 복구 가능하던가 또는 선택적 복구 가능한 경우가 되어야 한다. 절대 복구 가능한 경우는 주어진 식 (1)로부터 조건 (4)가 만족됨을 쉽게 알 수 있다. 선택적 복구 가능한 경우에서 최소의 여분소자를 사용하면 복구할 때 사용되는 전체 여분소자 라인의 수는 최소화할 수 있지만, C_R과 Cc가 각각 제한되어 있어 실제로는 모든 고장소자를 복구할 수

없다. 이 경우 최소의 여분소자 라인을 사용하는 것은 아니지만 복구 가능한 방법을 찾기 위해 한 고장소자 집단 G_k의 ($R_{k,1}$, $C_{k,1}$)대신 ($R_{k,2}$, $C_{k,2}$)나 ($R_{k,3}$, $C_{k,3}$) 등을 사용하여 재구성을 시도할 수 있다. 복구하는데 필요한 최소의 여분소자 행의 수는 C_R보다 작고 열의 수는 C_C보다 크다고 가정하자. 즉 $SR_{min} < C_R$, $SC_{min} > C_C$인 경우에 한 고장소자 집단 G_k의 ($R_{k,1}$, $C_{k,1}$)대신 ($R_{k,2}$, $C_{k,2}$)를 사용하여 배열구조를 복구할 수 있다면, $(C_{k,1} - C_{k,2}) > (SC_{min} - C_C)$을 만족해야 한다. 즉 $C_{k,2}$는 $C_{k,2} < C_{k,1} - (SC_{min} - C_C)$가 되도록 충분히 작아야 된다. ($R_{k,1}$, $C_{k,1}$)대신 ($R_{k,2}$, $C_{k,2}$)를 사용하면 여분소자 열에 관해 다음과 같은 관계식을 얻는다.

$$SC'_{min} = \sum_{i \neq k} C_{i,1} + C_{k,2} < \sum_{i} C_{i,1} - (SC_{min} - C_C)$$

한편 $(R_{k,1} + C_{k,1}) < (R_{k,2} + C_{k,2})$이므로 $R_{k,2} > R_{k,1} + (C_{k,1} - C_{k,2}) > R_{k,1} + (SC_{min} - C_C)$이 된다. 즉 $R_{k,2} > R_{k,1} + (SC_{min} - C_C)$이므로 여분소자 행에 관해서는 다음과 같은 관계식을 얻는다.

$$SR'_{min} = \sum_{i \neq k} R_{i,1} + R_{k,2} > \sum_{i} R_{i,1} + (SC_{min} - C_C)$$

SR'_{min}은 SR_{min} 보다 $(SC_{min} - C_C)$이상 커지며, 이것은 최대 $(C_R - SR_{min})$까지 증가할 수 있다. 선택적 복구 가능한 경우에서 각 G_i에 대하여 최소의 여분소자 라인을 사용하면 배열구조를 복구할 수는 없지만, 다른 가능한 방법이 존재하기 위해서는 $(SC_{min} - C_C) < (C_R - SR_{min})$을 만족해야 한다.

위와 반대인 경우, 즉 $SR_{min} > C_R$, $SC_{min} < C_C$인 경우에 복구 가능한 방법이 존재하기 위해서는 $(C_C - SC_{min}) > (SR_{min} - C_R)$을 만족해야 함을 알 수 있다. 두 경우 모두 $SR_{min} + SC_{min} < C_R + C_C$가 되며, 따라서 이 관계식은 고장소자를 포함하는 배열구조가 복구 가능하기 위한 필요조건이 된다. 복구 가능한 방법이 존재하기 위한 충분조건은 절대 복구 가능한 경우에 해당됨으로 다음 관계식을 만족해야 한다.

$$SR_{min} \leq C_R \ , \ SC_{min} \leq C_C$$

Ⅳ. 시뮬레이션 결과 및 분석

VLSI 칩에서 발생하는 고장소자들이 다음과 같은 여러 가지 패턴으로 분포하여 발생된다.

(1) 각 고장소자가 독립적으로 발생하는 경우 (random faults)

(2) 여러 고장소자가 행이나 열에 동시에 발생하는 경우 (overlapped faults)

(3) 여러 고장소자들이 덩어리로 뭉쳐서 발생하는 경우 (clustered faults)

첫번째 경우는 고장소자들이 독립적으로 흩어져 발생하는 경우로서 고장소자들 간의 의존성이 거의 없기 때문에 어떠한 알고리즘을 사용하여도 비슷한 복구율을 갖는다. 그러나 두번째나 세번째 경우는 한 행이나 열에 여러 고장소자들이 발생하고 고장소자들 간의 의존성이 높기 때문에 사용하는 알고리즘에 따라 복구율은 큰 차이를 보인다.

대부분의 경우 고장소자들은 집단으로 뭉쳐서 발생하고 있음이 여러 논문을 통하여 발표되었다[9][10]. 고장소자들이 덩어리로 뭉쳐서 발생하면 이것들이 x나 y축상에 서로 중첩되어 발생할 확률도 커진다. 예를 들어 2개의 고장소자 덩어리가 x나 y축상에 서로 중첩되어 발생하는 경우, 본 논문에서 제안한 집단화 과정을 거치면 하나의 고장소자 집단으로 간주되며, 결과적으로 적은 수의 여분소자 라인을 사용하여 복구할 수 있는 방법을 쉽게 찾을 수 있다.

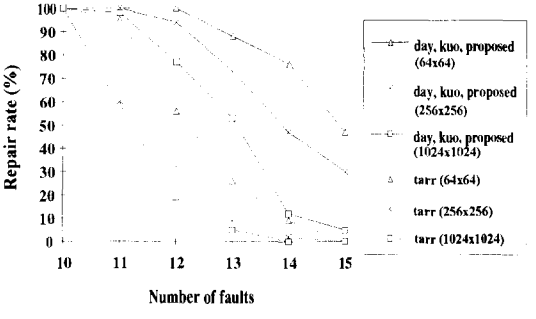
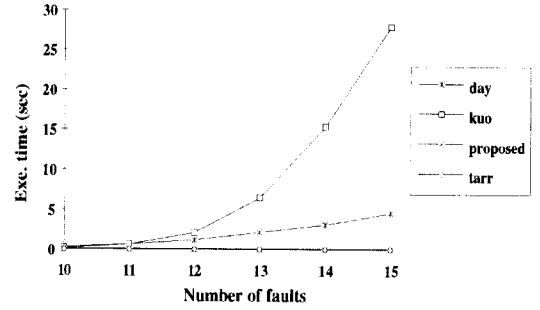


그림 4. 고장 소자가 독립적으로 발생하는 경우의 복구율

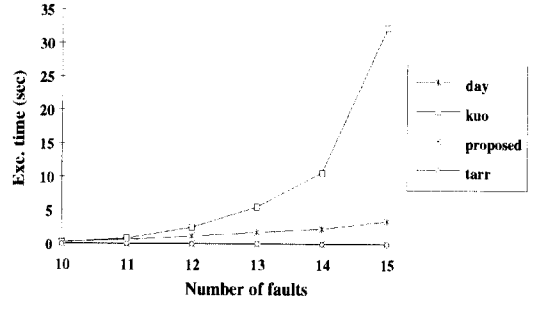
Fig. 4. Repair rates when random faults are generated.

본 논문에서는 제안한 알고리즘, Tarr의 그리디 알고리즘, Day의 과도 검색 알고리즘, 그리고 Kuo의 브랜치 앤 바운드 알고리즘의 성능을 비교하기 위하여 동일한 상황하에서 시뮬레이션을 수행하였다. 위의 알고리즘들은 C언어를 사용하여 구현하였으며 SPARK-station에서 수행하였다. 첫번째 고장소자 패턴인 경우 주어진 갯수의 고장소자를 난수 발생기를 사용하여 생성하였다. 두번째나 세번째 고장소자 패턴인 경우에는

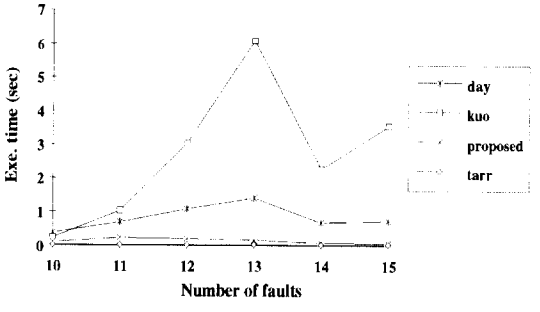
각 행과 열에 대하여 고장소자의 발생 확률값을 난수 발생기를 사용하여 얻었다.



(a)



(b)



(c)

그림 5. (a) 64x64 배열구조에서의 알고리즘 수행시간 (b) 256×256 배열구조에서의 알고리즘 수행시간 (c) 1024×1024 배열구조에서의 알고리즘 수행시간

Fig. 5. (a) Execution times for 64x64 array. (b) Execution times for 256×256 array. (c) Execution times for 1024×1024 array.

그림 4는 64×64, 256×256, 1024×1024의 배열구조에서 고장소자들이 독립적으로 발생하는 경우의 평균 복구율을 나타낸 것이다. 각 배열에서 여분소자의 행과 열은 5개로 제한하였다. 여기서 본 논문의 방법, Day의 방법, Kuo의 방법은 복구율이 같기 때문에 하나의 그래프로 나타내었다. 즉 본 논문에서 제안한 알고리즘의 복구율은 일종의 과도검색 알고리즘인 Day나 Kuo의 방법과는 실질적으로 같고 그리디 알고리즘인 Tarr의 방법보다는 월등히 좋은 것을 알 수 있다. 각 메모리의 크기가 커짐에 따라 전체적으로 모든 방법들의 복구율이 감소하는 것을 알 수 있다. 이것은 배열의 크기가 증가할수록 고장소자들은 점점 랜덤하게 발생되어, x나 y축상에 서로 중복되어 발생하는 고장소자의 수가 상대적으로 줄어들기 때문이다. 그림 5는 알고리즘의 평균 수행시간을 나타낸 것이다. Day와 Kuo의 방법은 거의 모든 경우를 검색하기 때문에 많은 시간이 걸리는 것을 알 수 있다. 특히 Kuo의 방법은 제한적인 검색을 위해 정렬 및 동일 데이타 제거 등의 작업을 수행하므로, 일반적으로 고장소자들의 수가 많을수록 수행시간은 크게 증가하게 된다. 반면 Tarr의 알고리즘은 단순한 그리디 방법을 사용하기 때문에 수행시간이 가장 적은 것을 알 수 있다. 본 논문의 방법도 Tarr의 방법에 못지 않게 평균 수행시간이 짧은 것을 알 수 있다. 고장소자의 수가 어느 정도 이상 증가하면 수행시간은 오히려 감소하는 경우가 있는데, 이것은 주어진 여분소자 라인의 수에 비해 고장소자의 수가 너무 많이 발생하여 절대 복구 불가능한 경우가 자주 발생하기 때문이며, 이 경우 알고리즘의 수행은 초기에 중단된다.

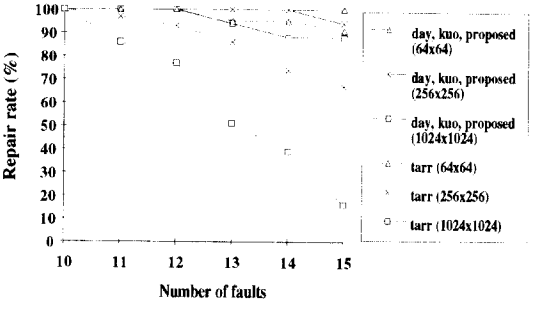
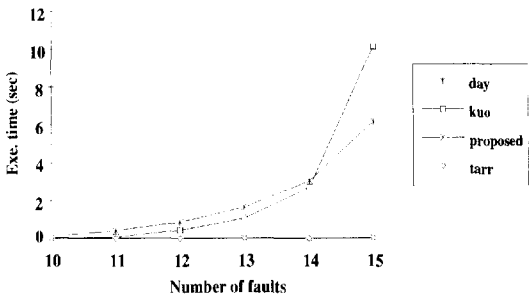


그림 6. 여러 고장소자가 행이나 열에 동시에 발생하는 경우의 복구율

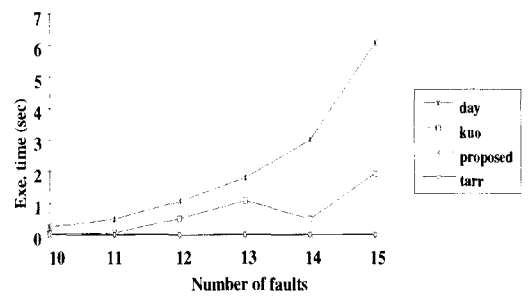
Fig. 6. Repair rates when overlapped faults are generated.

그림 6은 64×64, 256×256, 1024×1024의 배열구조에서 고장소자들이 서로 겹쳐서 발생하는 경우의 복구율을 나타낸 것이다. 그림 4와 비교하면 복구율이 증가한 것을 알 수 있다. 이것은 고장소자들이 x나 y축상에 많이 겹칠수록 여분소자 라인이 적게 사용되어

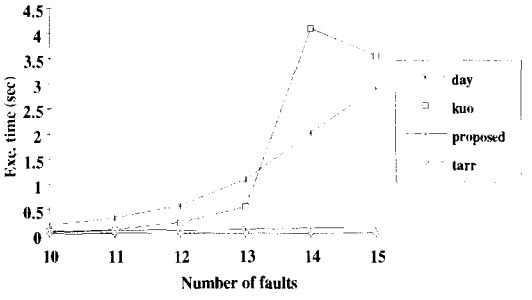
복구율이 증가하기 때문이다. 그림 7은 동일한 경우의 알고리즘 평균 수행시간을 나타낸 것이다.



(a)



(b)

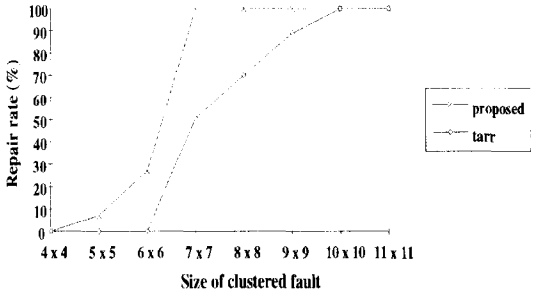


(c)

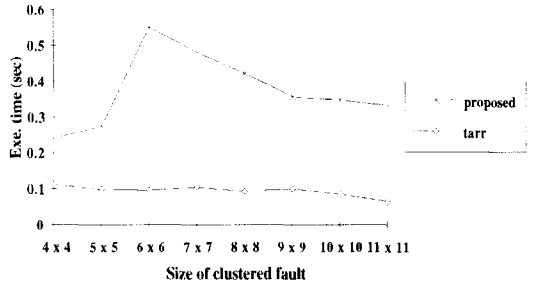
그림 7. (a) 64x64 배열구조에서의 알고리즘 수행시간 (b) 256x256 배열구조에서의 알고리즘 수행시간 (c) 1024x1024 배열구조에서의 알고리즘 수행시간

Fig. 7. (a) Execution times for 64x64 array. (b) Execution times for 256x256 array. (c) Execution times for 1024x1024 array.

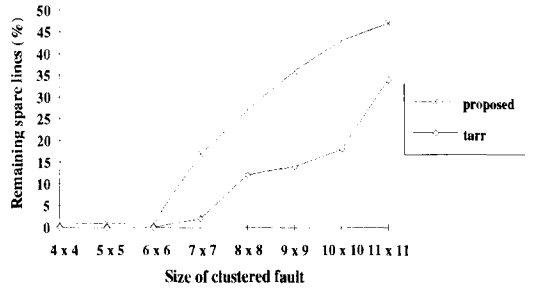
그림 (c)에서 고장소자가 적을 때는 Kuo의 방법이 Day의 방법보다 알고리즘의 평균 수행시간이 빠르나, 고장소자의 수가 많아지면 수행시간이 느려지는 것을 알 수 있다. 이것은 배열이 커지면 고장소자들이 x나 y 축 상에 서로 중복되어 발생하는 정도가 고장소자 수의 증가에 비해 상대적으로 줄어들며, 이 경우 Kuo의 방법은 데이터 정렬 및 비교 동작에 의한 오버헤드 때문에 수행시간이 길어진다. 그림 5와 비교하면 고장소자들이 x나 y축 상에 서로 중복되어 발생하면 알고리즘의 평균 수행시간이 짧아지는 것을 알 수 있다. 이것은 한 여분소자 라인에 의하여 복구되는 고장소자들이 많아짐으로 남아 있는 고장소자들이 급속히 적어지며, 따라서 알고리즘의 수행시간도 줄어들기 때문이다.



(a)



(b)



(c)

그림 8. (a) 1024×1024 배열구조에서 고장 소자가 뭉쳐서 발생하는 경우의 복구율 (b) 알고리즘의 평균 수행 시간 (c) 나머지 여분소자 라인의 비율

Fig. 8. (a) Repair rates when clustered faults are generated in 1024×1024 array (b) Execution times (c) Remaining spare lines.

그림 8는 1024×1024 배열구조에서 고장소자들이

뭉쳐서 발생하는 경우의 시뮬레이션 결과를 나타낸 것이다. 여기서 여분소자의 행과 열의 수는 각각 17개로 제한하였고, 고장소자는 170개 정도 발생시켰다. 각 집단 고장소자의 크기가 증가함에 따라 복구율이 증가하는 것을 알 수 있다. 본 논문의 방법이 Tarr의 방법보다 복구율은 좋지만, 수행시간은 약간 느린 것을 알 수 있다. 또한 복구할 때 사용되는 여분소자는 Tarr의 방법보다 많이 절약될 수 있는 것을 알 수 있는데, 이것은 본 논문의 방법이 Tarr의 방법보다 더 효율적으로 주어진 여분소자를 사용하여 복구함을 나타내는 것이다. 그림에서 Day와 Kuo의 방법의 결과를 표시하지 않은 이유는 많은 고장소자들이 뭉쳐서 발생하는 경우 알고리즘의 수행시간이 너무 길어져 일반 워크스테이션에서는 수행이 불가능하기 때문이다. 이것은 NP-Complete 문제에서 Day나 Kuo의 방법과 같은 과도 검색 알고리즘이 갖는 일반적인 특성이다.

V. 결 론

본 논문의 목적은 배열구조를 갖는 메모리에서 고장소자들을 여분소자의 행 또는 열 단위로 대치시켜 복구하는 알고리즘을 고안하는 것이다. Tarr의 그리디 알고리즘은 가장 간단한 복구방법으로 고장소자가 n일 때 시간 복잡도는 $O(n^2)$이다. 따라서 알고리즘의 수행시간은 짧지만 복구율은 매우 낮다. Day나 Kuo의 방법은 일종의 과도 검색 알고리즘으로 복구율은 높지만 시간 복잡도는 $O(2^n)$ 이상으로, 알고리즘의 수행시간이 길어진다.

본 논문에서는 배열구조의 직교좌표상에서 고장소자의 집단화를 이용하여 효율적으로 재구성할 수 있는 방법을 제안하였다. 또한 직교좌표에서 제안된 방법을 이분그래프로 변환하여 알고리즘으로 구현하였다. 제안된 알고리즘은 최대 매칭을 이용하여 절대 복구 불가능한 경우에는 초기에 알고리즘의 수행을 중단하며, 거의 모든 경우에 대하여 검색하는 Day나 Kuo의 방법과는 달리 집단화된 고장소자를 이용하므로써 매우 효율적으로 배열구조를 재구성 할 수 있다. 본 논문의 알고리즘은 모든 경우에 대해 과도 검색 알고리즘과 거의 같은 높은 복구율을 제공하고, 시간 복잡도는 최악의 경우 $O(n^3)$으로 그리디 방법과 비교하여 알고리즘의 수행시간은 조금 길어지나 대부분의 경우 거의 같은 시간에 배열구조를 재구성할 수 있다. 여러 가지 고장소자의 분포와 패턴에 대하여 시뮬레이션한 결과, 본 논문에서 제안한 알고리즘이 기존의 방법보다 효과적인 재구성 방법임을 확인하였다. 특히 많은 고장소자들이 뭉쳐서 발생하는 경우 기존의 과도 검색 알고리즘은 수행이 불가능하나, 본 논문에서 제안한 방법은 고장소자들을 집단화시킴으로 매우 효율적으로 배열구조를 재구성 할 수 있다.

참 고 문 헌

[1] M. Chean and J. A. B. Fortes, "A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays," IEEE Computer Journal, pp. 55-69, Jan. 1990.

[2] W. R. Moore, "A review of fault-tolerant techniques for the enhancement of integrated circuit yield," Proc. IEEE, pp. 684-698, May 1986.

[3] S. Y. Kuo and W. K. Fuchs, "Efficient spare allocation in configurable arrays," IEEE Design and Test of Computers, vol. 4, no. 1, pp. 24-31, Feb. 1987.

[4] M. Tarr, D. Bloudreau, and Murphy, "Defect Analysis System Speeds Test and Repair of Redundant Memories," Electronics, pp. 175-179, Jan. 12, 1984.

[5] J. Day, "A Fault-Driven Comprehensive Redundancy Algorithm," IEEE Design & Test, vol. 2, no. 3, pp. 35-44, June. 1985.

[6] N. Hasan and C. L. Liu, "Minimum fault coverage in reconfigurable arrays," in Dig. 18th Int. Symp. Fault-Tolerant Comput., pp. 348-353, 1988.

[7] B. W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison Wesley Publishing Company, Inc. 1989.

[8] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, N.J, 1982.

[9] C. H. Stapper, "On yield, fault distributions and clustering of particles," IBM J. Res. Develop., vol. 30, pp. 326-338, May 1986.

[10] F. J. Meyerand and D. K. Pradhan, "Mo-

deling defect spatial distribution," IEEE Trans. Computers, vol. 38, pp. 538-546, Apr. 1989.

[11] D. M. Blough and A. Pelc, "A Clustered Failure Model for the Memory Array Reconfiguration Problem," IEEE Trans. Computers, vol. 42, no. 5, pp. 518-528, May. 1993.

저 자 소 개

金 炯 奭(正會員)

1993년 인하대학교 전자공학과 졸업. 1995년 인하대학교 전자공학과 석사. 1995년 ~ 현재 (주) 데이콤 근무. 주 관심분야는 컴퓨터구조, 컴퓨터 네트워크.

崔 相 昉(正會員)

1981년 한양대학교 전자공학과 졸업. 1988년 University of Washington 석사. 1990년 University of Washington 박사. 1981년 ~ 1986년 LG 정보통신(주) 근무. 1991년 ~ 현재 인하대학교 전자공학과 부교수. 주 관심분야는 컴퓨터구조, 병렬 및 분산처리시스템, Fault-tolerant computing, 컴퓨터 네트워크