동시 고장 시뮬레이터의 메모리효율 및 성능 향상에 대한 연구

김 도 윤, 김 규 철
단국대학교 컴퓨터공학과

Fast and Memory Efficient Method for Optimal Concurrent Fault Simulator

Doyun Kim, Kyuchull Kim
Department of Computer Engineering
Dankook University

Abstract

Fault simulation for large and complex sequential circuits is highly cpu-intensive task in the integrated circuit design process.

In this paper, we propose CM-SIM, a concurrent fault simulator which employs an optimal memory management strategy and simple list operations. CM-SIM removes inefficiencies and uses new dynamic memory management strategies, using contiguous array memory. Consequently, we got improved performance and reduced memory usage in concurrent fault simulation.

I. 서 론

고장 시뮬레이터는 검사패턴 생성 시스템(Test Pattern Generation System)을 구성하는 데 가장 기본이 되는 부분이다. 고장 시뮬레이터는 대상 회로의 고장 특성을 파악하거나, 주어진 테스트 패턴에 대한 성능을 평가하는 데 쓰인다. 또한 테스트 시스템의 일부분으로 테스트를 생성하는 시간을 단축시키는 동시에 생성된 검사패턴을 최적화 하는데 사용된다.

회로를 시뮬레이션 하기 위한 시간은 회로내의 게이트 수가 N일 때 N^2에 비례하는 시간이 요구된다[1]. 따라서, 점점 복잡해지는 집적 회로를 효율적으로 검사하기 위해서 효율적인 시뮬레이터가 요구되어지고 있다.

현재 알려진 고장 시뮬레이션 방법에는 기본적으로 두 가지 형태가 있다. 첫 번째 방법은 하나의 고장을 각각의 패턴에 대해서 시뮬레이션 하는 방법(Serial Fault Simulation)[2]을 기본으로 여러 개의 패턴이나(Parallel Pattern Single Fault Propagation Simulation)[3] 여러 개의 고장에 대해서 확장하는 방법이다. 주로 컴퓨터의 워드크기에 대해서 병렬적으로 시뮬레이션 하는 방법을 사용하며[4] 병렬 연결된 컴퓨터를 사용하여 시뮬레이션 속력을 향상시키는 연구도 진행되어 왔다[5].

두 번째 방법은 하나의 패턴에 대해서 회로내의 모든 고장에 대한 시뮬레이션을 한번에 처리하는 방식으로 그 방법에 따라 연역적 고장 시뮬레이션(Deductive Fault Simulation)[6]과 동시 고장 시뮬레이션(Concurrent Fault Simulation)방식[7]으로 나뉜다.

연역적 고장 시뮬레이션 방식은 동시 고장 시뮬레이션 방식과 많은 유사점이 있지만, 순차회로에서와 같이 이진값 이상을 다루어야 할 경우 처리가 너무 복잡해지므로 거의 사용되지 않는다.

첫 번째 방법을 사용하는 시뮬레이터들은 대체로 메모리 효율이 높으며 두 번째 방법은 그에 비해 빠르지만 메모리 소모가 크다는 단점이 있다.

본 논문에서는 빠르지만 메모리 효율이 낮다고 알려진 동시 고장 시뮬레이션 방식에 대한 연구를 수행하였다. 동시 고장 시뮬레이션 방식의 복잡한 처리과정을 단순화시키며 메모리 관리에 연속적인 배열을 사용함으로 메모리 효율과 처리속력을 증대시키는 방법을 보이고 있다. 제안된 방식으로 구현된 CM-SIM은 이전의 고장 시뮬레이션 방식에 비해 높은 성능을 보였다.

본 논문의 구성은 다음과 같다. 2절에서는 일반적으로 알려진 동시 고장 시뮬레이션 방식의 비효율성을 제거한 CM-SIM의 방식을 설명하고 있다. 3절에서는 본 논문에서 제안한 메모리 관리방식과 구현방식에 대해서 자세한 설명을 하고 있으며, 4, 5절에서는 구현된 CM-SIM에 대한 실험을 통해 제안된 방식의 성능을 평가하며 그에 따른 결론을 맺고 있다.

II. 시뮬레이션 방식

본 논문에서 제안한 시뮬레이션 방식은, 동시 고장 시뮬레이션과 연역적 시뮬레이션 방식의 혼합 형태로

발표된 HYSIM[8]을 그 기본으로 하고 있다. 이 방식은 일반적인 동시 고장 시뮬레이션 방식의 비효율성을 제거하는 것을 통해 메모리 효율과 성능을 향상시키고 있다.

일반적으로 동시적 고장 시뮬레이션 방식은 사건 구동 방식(Event-Driven)을 사용한다. 사건 구동 방식은 일반적인 방법에 비해 뛰어난 성능을 보이며, 이를 사용할 경우 지연 시간의 처리도 가능해진다. 하지만, 회로내의 사건 발생이 많아지면 사건 구동 처리 자체를 위한 부하가 더 커지게 된다. 이전의 연구에 의하면 사건 구동방식은 1%이상의 회로가 활성화될 경우 상대적으로 낮은 성능을 보인다고 한다[9]. 또한 동시 시뮬레이션 방식에서의 대상이 되는 회로와 패턴의 특성은 많은 수의 사건을 발생시키므로 적합하지 않다[10]. 표1은 대상이 되는 회로에 대한 게이트 활성화율을 보이고 있다.

회로	처리된 고장	패턴수	게이트수	활성화율
s298	21376	162	143	92%
s349	17140	91	197	95%
s526	160585	754	224	95%
s713	33658	107	471	67%
s832	111415	377	330	89%
s1238	124303	349	555	64%
s5378	945116	408	3043	76%
s35932	1484134	86	18149	95%

표1. 대상 회로의 게이트 활성화율

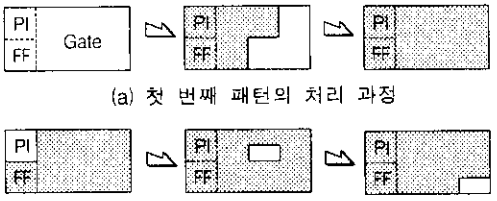
또한, 사건 처리방식은 전파되어온 고장의 삭제과정에서 대상이 되는 고장이 다중경로로 전파된 경우 문제를 야기할 수 있다. 이 경우를 처리하기 위해서는 더 많은 메모리와 복잡한 처리과정이 필요하게 된다.

본 논문에서는 이러한 문제를 해결하기 위해 사건 구동방식 대신 게이트의 레벨순서에 따라 모든 게이트를 처리한다[11]. 입력단에 부가된 패턴은 정해진 레벨 순서에 따라 출력단으로 전파되어 진다. 각 게이트는 그 입력단에 전파되어온 고장목록을 바탕으로 새로운 고장목록을 생성하게 된다. 이 방식은 처리를 단순화 시키고 처리과정에서 고장목록의 동적인 추가 삭제를 최소화 할 수 있다. 이때, 각 게이트의 고장목록은 이전의 고장목록을 제거한 뒤 새로 생성되므로 이전의 고장 목록은 처리가 완료되는 즉시 삭제할 수 있다.

그림1과 그림2는 일반적인 경우와 본 논문에서 사용한 방법의 메모리 맵을 보이고 있다. 이 방식은 동적 메모리 사용 효율을 높여준다[12].

III.메모리 관리(Memory Management)

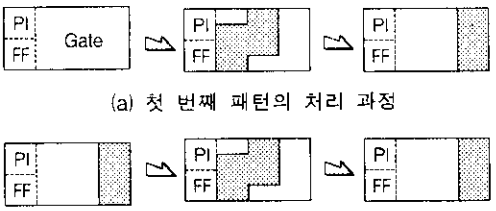
고장 시뮬레이션 방식에서 가장 많이 쓰이는 방식은 연결리스트를 사용하는 방법이다. 연결리스트의 단점은 추가적으로 메모리를 연결하기 위한 포인터 공간이 필요하므로 메모리 사용이 크다는 점과 리스트 중간의 정보를 검색하기 위해서는 처음부터 리스트를 찾아가야 할 필요가 있으므로 그 처리가 느리다는 점에 있다.



(a) 첫 번째 패턴의 처리 과정

(b) 두 번째 패턴의 처리 과정

그림 1. 일반적인 처리과정의 메모리맵



(a) 첫 번째 패턴의 처리 과정

(b) 두 번째 패턴의 처리 과정

그림 2. 새로운 방식에서의 메모리맵

이러한 단점에도 동시적 고장 시뮬레이션 방식을 포함한 대부분의 시뮬레이션 방식에서 요구되는 복잡한 메모리 추가 삭제의 과정을 처리하기 위해서 연결리스트를 사용하는 것이 일반적이다. 연결리스트는 리스트의 내용의 가변적인 추가와 삭제가 가능하며, 각 메모리를 셀 단위로 변경할 수 있다.

본 논문에서 제안한 메모리 관리 방식은 연결리스트를 사용하지 않고 연속된 배열을 사용한다. 3절에서 설명한 시뮬레이션 과정은 다른 이점 외에도 고장목록의 동적인 삽입, 삭제를 최소화시킬 수 있게 한다. 이를 통해 새로운 메모리 관리 방법을 적용할 수 있다.

1.메모리 관리

연속된 메모리를 동적으로 사용하기 위해서 고안된 메모리의 형태는 그림3과 같다.

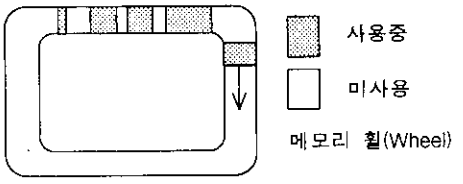


그림 3. 메모리 구성

고리 모양으로 구성된 메모리 공간은 연속적으로 각 게이트의 처리에 필요한 메모리 공간을 제공하며, 사용이 끝난 메모리 공간은 자동으로 새로운 처리과정에 의해 잠식되어 간다. 따라서, 사용될 메모리를 리스

트에 연결하거나 사용된 메모리를 리스트에서 제거해서 메모리 공간에 반환하는 과정이 필요하지 않다.

그 외에 메모리 참조를 위한 연산이 단순해지며, 메모리 공간이 연속적으로 유지되므로 사용 중에 사용이 완료된 메모리를 완전히 제거하는 것이 가능하며, 컴퓨터 시스템의 메모리 효율을 최대로 사용하는 것이 가능하게 된다. 그림 4에서 볼 수 있는 것과 같이 포인터 공간이 필요 없기 때문에 메모리의 낭비를 줄일 수 있다.

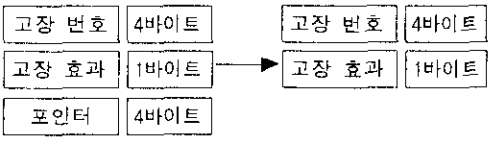


그림 4. 고장목록의 구성

2.메모리 정돈(Garbage Collection)

메모리로 배열을 사용하기 때문에 생기는 문제는 메모리 공간에 생기는 빈 공간들이다. 연결 리스트와는 달리 연속 배열을 사용하는 경우 사용 해제된 메모리 공간도 그 최하단의 메모리가 사용해제 되기까지는 사용이 불가능하게 된다.

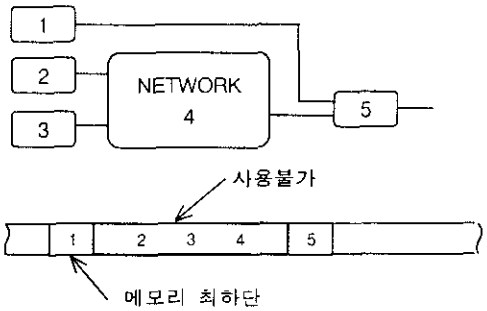


그림 5. 메모리 재활용 문제

그림5는 게이트의 처리 과정에서 발생하는 메모리 재활용 문제를 보이고 있다. 회로내의 5번 게이트가 완전히 처리되기까지 1번 게이트는 메모리의 최하단에 자리잡고 그 중간의 2, 3, 4에서 사용이 완료된 메모리를 재사용하지 못하게 한다. 4장의 실험 결과에서 볼 수 있는 것과 같이 실제 회로에서 같은 이유로 인한 메모리 효율 저하는 무시할 수 없다.

이 문제의 해결에는 두 가지 해결방안이 있을 수 있다. 첫째 방안은 회로의 처리 순서를 최대한 메모리 조각이 생기지 않도록 구성하는 방법이다. 이 방법은 추가적인 처리 없이 약간의 전처리과정을 통해 구성할 수 있다. 하지만, 복잡한 회로에 대해서 안정적인 성능을 가지는 알고리즘의 개발이 어려우며 높은 효율을 기대하기 어렵다는 단점이 있다.

본 논문에서는 두 번째 방안으로 메모리 정돈 과정을 수행한다. 이 과정은 추가적인 처리시간이 요구된다는 단점이 있으나, 재 정돈 시점을 정하는 방법에 따라 원하는 형태의 효율상승을 기대할 수 있다.

3.메모리 확장

메모리를 구성하고 메모리의 유지하는 것 외에 다른 중요한 문제는 할당된 메모리가 고갈되었을 때의 처리이다. 연결 리스트와 달리 배열처리는 연속된 공간상에 메모리가 추가되어야 한다. 메모리 공간의 확장이 요구될 때 연속선상의 메모리 공간을 사용할 수 없을 경우, 원하는 메모리의 확장은 불가능하다.

이와 같은 상황을 막기 위해서 미리 충분한 메모리 공간을 예약해두는 방법을 사용한다. 즉, 실제의 메모리는 할당하지 않고 단순히 메모리 공간만을 예약하여 이 부분이 다른 이유로 사용되지 않게 한다. 현재의 대부분의 운영체제는 각 프로그램에 대해서 독립적이고 충분한 메모리 공간을 제공하므로 메모리 공간의 예약은 효율에 있어서도 전혀 문제될 것이 없다. 또한, 컴퓨터의 하드웨어에 따라 메모리는 고유한 블록 크기로 할당되고 해제되므로, 이 블록 크기를 경계로 메모리 관리 전략을 수립하는 것이 가능하다.

본 논문에서 구현된 CM-SIM은 Win32환경에서 64Kbyte 블록 단위로 메모리의 할당과 해제가 가능하며, 이 블록 크기를 효율적으로 사용함으로 메모리 사용과 실행속도에서 뛰어난 결과를 보이고 있다.

IV. 실험결과 및 분석

본 논문에서는 메모리 사용량과 실행속도의 두 가지 부분에 대해서 실험결과를 비교하였다. 표1, 2, 3은 모두 같은 회로와 같은 패턴에 대해서 실험한 결과이다.

표2에서는 객관적인 비교가 가능한 메모리 사용량에 있어서는 다른 문헌에서 발췌한 결과[13]와 여러 가지 방식으로 구현된 시뮬레이터들을 같이 비교하였다. HYSIM-II는 C++로 구현하였으며, HYSIM[8]을 바탕으로 연결리스트를 사용하는 방식으로 구현되었으며, CM-SIM과 메모리 관리 방식 외에는 거의 공통된 형태로 구현되었다. CM-SIM의 결과는 메모리 재 정돈을 수행하지 않은 결과와 최대로 메모리 정돈을 수행했을 경우를 보이고 있다. 메모리 정돈의 빈도는 처리 속력에 영향을 주는 요인이다. 따라서 그 전략에 따라서 메모리 사용량이 달라진다.

표3의 시뮬레이션 수행 속도에 대해서는 PROOFS와 HYSIM-II방식과 CM-SIM을 비교하고 있다. 각 시뮬레이터들은 Pentium 100PC, NT환경에서 실험되었으며 별도의 최적화 과정 없이 구성되었다. 결과에서 볼 수 있는 것과 같이 구현된 시뮬레이터는 처리하는 회로의 크기가 커지고 발생하는 고장목록의 크기가 커질 경우 뛰어난 성능을 보임을 알 수 있다. 메모리 재

정돈을 수행하는 경우에 있어서는 전체의 20%에 대해 재 정돈을 수행하는 결과를 보이고 있다. 이때, 메모리 효율이 향상되는데 반해 큰 성능저하는 보이지 않음을 알 수 있다.

회로	Coverage	Normal Concurrent (*)	PROOFS (*)	HYSIM-II (†)	CM-SIM (▽)	CM-SIM Garbage Collection (Factor20)
s298	264/308	552	96	8.0	7.5(4.4)	5.2
s349	335/350	656	112	7.7	14.2(4.3)	5.9
s526	418/555	896	120	25.5	31.3(14.2)	21.8
s832	708/870	816	176	11.5	23.9(6.4)	8.3
s1238	1283/1355	728	216	5.5	7.2(3.1)	5.2
s5378	3407/4603	1544	752	73.3	225.2(40.1)	64.3
s35932	34398/39094	5576	5872	857.9	3119.7(476.6)	368

*: [13]의 결과에서 발췌, 본 논문과 같은 회로와 패턴 사용
†: HYSIM을 바탕으로 C++로 구현한 시뮬레이터
▽: 괄호 안의 결과는 완전 메모리 정돈(Full Garbage Collection)을 통한 최소값이다.

표 2. 메모리 사용량 비교 [Kbyte]

회로	PROOFS	HYSIM-II	CM-SIM	CM-SIM (Factor20)
s298	0.38	0.541	0.50	0.51
s349	0.24	0.29	0.30	0.31
s526	6.07	8.11	7.03	7.10
s832	3.72	2.23	2.00	2.12
s1238	2.00	1.53	1.32	1.41
s5378	24.00	15.58	14.00	14.70
s35932	87.50	51.73	24.85	27.40

표 3. 시뮬레이션 수행 속도 비교[초]

CM-SIM은 전체가 C++로 객체 지향적으로 구현되어 있다. 객체 지향언어의 사용은 그 구현과 유지에 매우 유리하지만 실제 성능 면에서는 같은 C로 구현된 프로그램에 비해 크게는 약 10-20%의 성능저하를 보인다[14]. 본 논문에서 비교한 PROOFS는 C를 사용하였으며, C로 구현된 HYSIM은 C++로 구현된 것에 비해 약 10%가량의 성능향상이 있었다.

V. 결론

동시 고장 시뮬레이션 방식에 대해서 개선된 처리 방식을 바탕으로 새로운 메모리 관리 방식을 제안하고 실험하였다. 제안된 새로운 동적 메모리 관리 방식을 사용한 CM-SIM (Contiguous Memory Type Concurrent Fault Simulator)은 실험을 통해 큰 고장목록을 다루는 경우에 있어 뛰어난 성능 향상을 보임을 알 수 있었다.

참고문헌

[1] P.Goel, "Test Generation Cost Analysis and Projections," Proc. of 17th Design Automation Conf. pp. 77-84, 1980.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman, Digital Systems : Testing and Testable Design. New York: Computer Science Press, 1990.

[3] Hyung Ki Lee, "An Efficient, Forward Fault Simulation Algorithm Based on Parallel Pattern Single Fault Propagation", ITC 1991, pp946-955.

[4] Y. Takamatsu and K. Kinoshita, "On the Effect of Size of Fault Word in Parallel Fault Simulation," Journal of Information Processing, vol. 9, pp. 166-169, 1986.

[5] Elizabeth M. Rudnick, "Overcoming the Serial Logic Simulation Bottleneck in Parallel Fault Simulation," Int. Conf. on VLSI Design 1997.

[6] D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," IEEE Trans. on Comput., vol. C-21, pp. 464-471, May 1972.

[7] E. G. Ulrich and T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," Design Auto. Workshop, vol. 6, pp. 145-150, April 1973.

[8] K. Kim and K. K. Suluja, "HYSIM: Hybrid Fault Simulation for Sequential Circuits," VLSI Design, International Journal of Custom Chip Design, Vol. 4. No. 3, pp 181-197, July 1996.

[9] Z. Wang and P. M. Maurer, "LECSIM : A Levelized Event Driven Compiled Logic Simulator," IEEE Design Auto. Conf., pp. 491-496, June 1990.

[10] Brian Grayson, "Statistics on Concurrent Fault and Design Error Simulation," 1995 Intl. Conf. on Computer Design (ICCD-95), pp. 622-627, October 1995.

[11] K. Kim and K. K. Saluja, "On Fault Deletion Problem in Concurrent Fault Simulation for Synchronous Sequential Circuits," IEEE VLSI Test Symp., April, 1992.

[12] K. Kim and K. K. Saluja, "Reduction of Dynamic Memory Usage in Concurrent Fault Simulation for Synchronous Sequential Circuits," Asian Test Symposium

[13] Thomas M. Niermann, "PROOFS: A Fast, Memory Efficient Sequential Circuit Fault Simulator", 27th ACM/IEEE DAC, pp535-540, 1990.

[14] Bald Calder, "Quantifying Behavioral Differences Between C and C++ Programs", the Journal of Programming Language. Vol 2. Num 4, 1994.