Universidade Federal do Ceará
Departamento de Engenharia de Teleinformática
Curso de Graduação em Engenharia de Teleinformática

Mila Maracaba Moreira

Sistema para Contagem Automática de Eritrócitos por Análise de Imagens de Amostras de Sangue

Fortaleza – Ceará
Outubro 2012

Autor:
Mila Maracaba Moreira

Orientador:
Prof. Dr. Paulo César Cortez

Sistema para Contagem Automática de Eritrócitos por Análise de Imagens de Amostras de Sangue

Monografia de Conclusão de Curso apresentada à Coordenação do Curso de Graduação em Engenharia de Teleinformática da Universidade Federal do Ceará como parte dos requisitos para obtenção do grau de **Engenheiro de Teleinformática**.

Fortaleza – Ceará
Outubro 2012

Resumo

Os sistemas de diagnóstico de falhas vêm adquirindo importância na computação devido à complexidade dos dispositivos digitais. Seu uso na identificação de problemas de *hardware* tem se tornado uma demanda crescente tanto para empresas especializadas em computadores, como montadoras e fabricantes e mesmo para usuários domésticos, que desejam verificar a integridade do seu equipamento.

Neste trabalho, foi desenvolvida uma ferramenta de diagnóstico de falhas em memórias, chamada MDiag. O *software* foi construído como uma aplicação para sistemas operacionais Linux. Todo o projeto e implementação foi embasado por um amplo estudo sobre falhas em memórias, algoritmos de detecção de falhas em memórias e o controle deste componente através do Linux.

Também foi desenvolvido um sistema automático de geração e inserção de falhas, utilizando funcionalidades de *debug* presentes na maioria dos processadores atuais. Este sistema foi usado para testar e validar o MDiag quanto a cobertura de falhas.

Foram realizados testes reais com placas de memórias defeituosas, onde os resultados do MDiag foram comparados com os de ferramentas utilizadas no mercado.

Palavras-chaves: Modelagem, Teste, Algoritmo, Cobertura, Complexidade, LinuxMM.

Abstract

Fault diagnosis systems has growing in importance on the computing due to the complexity of the digital devices. Its use to identiying hardware's malfunction has become an ascenceding demand for both, companies specialized in computers, such as assemblers and manufacturers, as well as for home users who want to check the integrity of your equipment.

In this study, we developed a tool for fault diagnosis in memories, called MDiag. It was built as an application for Linux operating systems. All the design and implementation was based on an extensive study of memory faults, fault detection algorithms and Linux memory menagement.

It was also developed an automatic system for fault generation and insertion using debug features available in most of current processors. This system was used to test and validate the MDiag as its fault coverage.

Tests were carried out with real faulty memories, where the results where the results achieved by MDiag were compared with those achieved by tools used in the market.

Keywords: Modeling, Testing, Algorithm, Coverage, Complexity, LinuxMM.

Dedico este trabalho primeiramente a Deus, sem o qual nada na minha vida tem sentido. Dedico também à minha mãe Heronildes, meu pai Plínio, minha irmã Jamile, a toda minha família, à minha namorada Suelen e aos professores e amigos que caminharam junto a mim em algum momento da minha formação.

"Everything makes sense a bit at a time. But when you try to think of it all at once, it comes out wrong."

Terry Pratchett

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

DFT *Design for Testability*

CI Circuito Integrado

RAM Memória de Acesso Aleatório (*Random Access Memory*)

VLSI *Very-large-scale integration*

SO Sistema Operacional

HDL Linguagem de Descrição de *Hardware* (*Hardware Description Language*)

SAF *Stuck-At Fault*

TF *Transition Fault*

CF *Coupling Fault*

NPSF *Neighborhood Pattern Sensitive Fault*

PSF *Pattern Sensitive Fault*

MOVI *Moving Invertion*

LSB *Bit* Menos Significativo (*Least Significant Bit*)

MSB *Bit* Mais Significativo (*Most Significant Bit*)

BIST *Built-in Self-test*

LinuxMM Linux *Memory Management*

OOM Escassez de Memória (*Out Of Memory*)

LESC Laboratório de Engenharia de Sistemas de Computação

UFC Universidade Federal do Ceará

GDB GNU Project Debugger

SO-DIMM *Small Outline Dual In-Line Memory Module*

DIMM *Dual In-Line Memory Module*

LTT Lenovo *ThinkVantage Toolbox*

POST *Power On Self Test*

BIOS *Basic Input/Output System*

API Interface de Programação de Aplicativos (*Application Programming Interface*)

Capítulo 1

Introdução

Os sistemas computacionais estão cada vez mais presentes no cotidiano da sociedade. Uma gama crescente de tarefas são confiadas a computadores de vários tipos, desde sistemas embarcados, como equipamentos de controle e segurança, passando pelos computadores pessoais, até grandes servidores, como máquinas utilizadas em *data centers*. Muitos desses sistemas trabalham em operação contínua, como, por exemplo, os chamados Sistemas de Alta Disponibilidade, que podem chegar a taxas de 99,999% de disponibilidade anual. Este caso ilustra a importância, nos dias atuais, de ferramentas que automatizam o processo de manuntenção e ajudam a garantir o funcionamento do sistema. Um tipo de ferramenta deste tipo são os *softwares* de diagnóstico. Eles são utilizados para verificar se os componentes de *hardware* de um sistema estão funcionando corretamente.

Por exemplo, após a montagem de um computador em uma linha de produção, um diagnóstico de todo o *hardware* deve ser realizado para garantir que nenhum componente danificado chegará ao cliente. Outro uso é na identificação de partes defeituosas, que pode ser feita por um usuário que está insatisfeito com o desempenho geral do sistema, ou por um serviço de assistência técnica que precisa substituir uma peça defeituosa. Em Sistemas de Alta Disponibilidade, os diagnósticos podem ser agendados para execuções periódicas, gerando relatórios do estado de conservação do *hardware*.

Para se projetar um diagnóstico, é necessário total domínio e conhecimento a respeito do elemento a ser testado, além de um extenso estudo sobre os tipo de falhas que o componente possa vir a apresentar. Mais ainda, é preciso que o *software*

explore o máximo possível das funcionalidades do dispositivo para que se possa assegurar o seu funcionamento.

Este trabalho se concentra no projeto e desenvolvimento de um sistema de diagnóstico de falhas em memórias de computadores, que é um componente de maior importância para o funcionamento de um sistema computacional e que apresenta alta taxa de falhas.

1.1 Motivação

As matrizes de células de memória estão entre os circuitos *Very-large-scale integration* (VLSI) de maior densidade de transistores por área (ADAMS, 2003). Por possuir uma estrutura essencialmente regular, sua produção está, geralmente, próxima do estado da arte da tecnologia de fabricação de Circuito Integrado (CI). Esses *chips* apresentam uma média muito alta de defeitos físicos, comparados com outros tipos de circuitos. Isto ocorre principalmente devido as diminutas distâncias entre os transistores e a proximidade entre as trilhas condutoras. Esse fato motivou pesquisadores a desenvolverem eficientes algoritmos de testes de memória, que atualmente se encontram em um elevado grau de maturidade para a maioria das aplicações.

Apesar da grande quantidade e da qualidade dos algoritmos de detecção de falhas em memória disponíveis atualmente, poucas aplicações os utilizam para diagnosticar esses componentes em sistemas finais. O uso mais comum é na detecção de falhas de manufatura, logo após a fabricação do *chip*. Esses testes poderiam ser uma ferramenta importante no caso de um usuário (doméstico ou empresa) que deseja diagnosticar um componente de memória já em uso, seja por notar certa instabilidade no funcionamento do sistema ou seja para testar periodicamente um dispositivo que opera em condições de estresse. As opções existentes atualmente para estes fins são escassas e a maioria utiliza algoritmos simples e de baixa cobertura de falhas.

A ferramenta mais utilizada para diagnóstico de memórias em sistemas finais, isto é, sistemas prontos para uso, é o Memtest86+ (MEMTEST86, 2011), *software open source* que executa diretamente após o *boot* do computador, sem utilizar um Sistema Operacional (SO). Sua abordagem evita todo o controle que qualquer SO exerce sobre o *hardware*, permitindo, assim, testar a maior quantidade possível da memória física. Uma desvantagem desta característica é a necessidade de interromper toda a

atividade do sistema para reiniciá-lo, algo muitas vezes incômodo para máquinas de uso pessoal ou mesmo inviável em servidores. Outra desvantagem é que o Memtest86+ utiliza trechos de código em *assembly*, o que só o permite executar em plataforma de arquitertura x86.

Já um diagnóstico rodando sobre Linux possui as vantagens de ser executado nos mais diversos sistemas computacionais, incluindo desde pequenas plataformas embarcadas até servidores de grande porte. Conta também com a possibilidade de diagnosticar um componente de memória com o sistema em pleno funcionamento, sem necessitar da paralisação do sistema. Esta flexibilidade em relação à abordagem do Memtest86+, se dá em detrimento da quantidade de memória testada, pois uma aplicação rodando sobre um sistema operacional jamais poderá ter acesso a toda a memória física disponível.

A motivação para este trabalho é incorporar a uma ferramenta de diagnóstico de *hardware* para sistemas de uso cotidiano, os avançados algoritmos para detecção de falhas em memória disponíveis atualmente.

1.2 Objetivos

Os objetivos gerais e específicos estabelecidos para este trabalho, são apresentados a seguir.

1.2.1 Objetivo Geral

O objetivo principal é desenvolver uma ferramenta de *software* com um conjunto de funcionalidades e algoritmos capazes de detectar com eficácia variados tipos de falhas em módulos de Memória de Acesso Aleatório (*Random Access Memory*) (RAM) de sistemas computacionais utilizando Linux.

1.2.2 Objetivos Específicos

Para guiar o desenvolvimento do trabalho, foram elaborados alguns objetivos específicos:

i. Selecionar os melhores algoritmos para detecção de falhas em memória disponíveis na literatura;

ii. Implementar os algoritmos escolhidos de maneira eficiente e flexível;

iii. Desenvolver um *software* portável, capaz de testar sistemas variados, que utilizem Linux;

iv. Avaliar a eficiência do diagnóstico implementado quanto a sua capacidade de detecção falhas.

1.3 Organização deste Trabalho

Este trabalho está organizado em 5 capítulos. O Capítulo 2 apresenta uma revisão sobre os assuntos pertinentes ao entendimento do projeto. Neste capítulo são abordados os modelos de falhas de memória, um estudo sobre os testes existentes e características do sistema operacional Linux que influenciam na comunicação com o *hardware* em questão. O Capítulo 3 descreve todas as ferramentas usadas no desenvolvimento do trabalho, bem como suas características fundamentais, e o processo seguido para avaliar e testar a ferramenta. No Capítulo 4 são descritos e discutidos os resultados obtidos a partir dos testes realizados e por fim, o último capítulo apresenta as considerações finais e as perspectivas futuras.

Capítulo 2

Fundamentação Teórica

Neste capítulo são descritos alguns fundamentos sobre modelos de falhas em memória e os tipos de algoritmos mais discutidos na literatura. Também são apresentadas algumas características do gerenciador de memória do Linux que são importantes para o projeto do diagnóstico.

2.1 Modelos de Falhas em Memórias

Em sistemas computacionais, as palavras falha, erro e defeito possuem significados distintos. Suas diferenças serão enunciadas como apresentadas por (WEBER, 2001), para uma melhor contextualização.

A falha (*fault*) é um comportamento que ocorre no nível mais baixo do sistema. Geralmente está associada ao *hardware* ou à escrita do código. Por exemplo, uma flutuação na fonte de alimentação ou a troca de um "$>=$" por um "$>$" pode causar uma falha. Assim, estas estão associadas ao universo físico, conforme ilustrado na Figura 2.1.

O erro (*error*) é a representação da falha no universo da informação. Ele se apresenta quando, por conseqüência de uma falha, a informação for corrompida. Quando um estado pode levar a ocorrência de um defeito, pode-se dizer que o sistema está em estado de erro.

O defeito (*failure*) é um desvio da especificação, e ocorre em conseqüência de um erro. O defeito é algo percebido pelo usuário, por isso os defeitos estão associados ao universo do usuário.

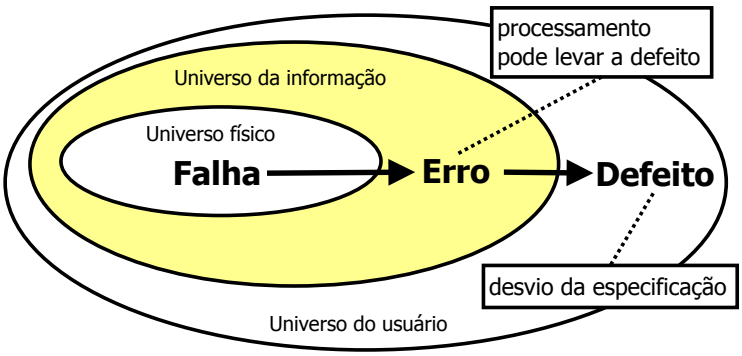

Figura 2.1: Modelo de três universos (WEBER, 2001).

Por exemplo, um chip de memória que apresenta uma falha em um de seus bits (falha, universo físico) pode provocar uma interpretação errada da informação armazenada em uma estrutura de dados (erro, universo da informação) e como resultado o sistema pode negar autorização de embarque para todos os passageiros de um vôo (defeito, universo do usuário). É interessante observar que uma falha não leva, necessariamente, a um erro (aquela porção da memória pode nunca ser usada) e um erro não leva, necessariamente, a um defeito (no exemplo, a informação de vôo lotado poderia eventualmente ser obtida a partir de outros dados redundantes da estrutura).

Neste trabalho será utilizado apenas o conceito de falha, pois estamos interessados em diagnosticar um *hardware*, parte do universo físico.

A descrição de como alguma coisa pode falhar é o modelo de falha (ADAMS, 2003). Esta descrição pode ser feita em vários níveis de abstração. No caso de circuitos integrados, existem os níveis comportamental, funcional, lógico, elétrico e geométrico, mostrados na Figura 2.2.

O nível mais alto de abstração é o comportamental, que visa fazer uma descrição em alto nível do sistema, geralmente auxiliada por uma Linguagem de Descrição de *Hardware* (*Hardware Description Language*) (HDL).

Na modelagem em nível funcional, os circuitos são vistos como caixas pretas, isto é, apenas as entradas e saídas são consideradas, não importando o trabalho interno realizado. Historicamente, esta é a modelagem mais utilizada para testes de memória (ADAMS, 2003).

No nível lógico mais detalhes são acrescentados, passando a serem consideradas

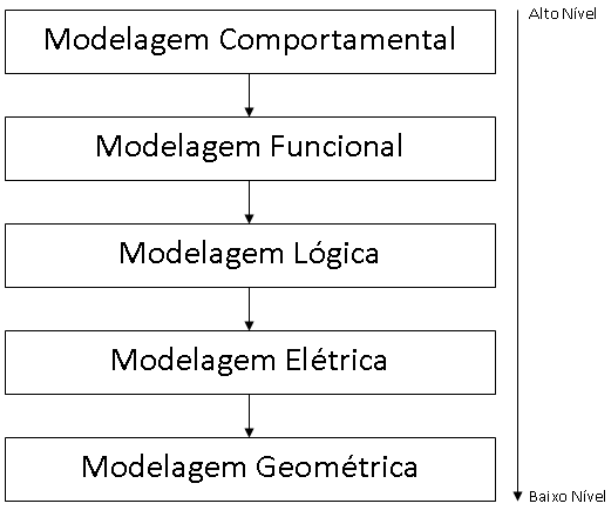

Figura 2.2: Níveis de abstração para modelagem de falhas.

as portas lógicas que compõem os circuitos. Abaixo, está o nível elétrico, onde as falhas são vistas nas operações dos transistores. Por fim, o modelo geométrico se refere a todos os detalhes do processo de fabricação do circuito integrado, como tamanho e localização das portas, distância entre células adjacentes, correntes de fuga entre poços de dopagem, dentre outros (ADAMS, 2003).

Os modelos de falhas funcionais são os mais empregados em testes e diagnósticos de memória (ADAMS, 2003), porque não há interesse na natureza das falhas, mas no seu efeito na funcionalidade dos circuitos. A partir deste ponto trataremos as falhas sempre a nível funcional.

Os modelos de falhas clássicos para circuitos digitais são *Stuck-At Fault*, *Bridging Fault*, *Open Fault* e *Delay Fault*. No entanto, eles não são suficientes para representar todas as falhas importantes em memórias, fazendo-se necessário definir erros mais específicos para este tipo de circuito (NAIR; THATTE; ABRAHAM, 1978; PAPACHRISTOU; SAHGAL, 1985).

As falhas em memórias podem ser classificadas em três categorias: falhas nas células de memória, falhas no decodificador de endereço e falhas dinâmicas. Cada umas destas categorias serão descritas nas próximas seções.

2.1.1 Falhas na Matriz de Células de Memória

As falhas nas células acontecem principalmente devido a curtos de metalização e acoplamento capacitivo (THATTE, 1977). Os principais tipos são *Stuck-At Fault*

(SAF), *Transition Fault* (TF), *Coupling Fault* (CF) e *Neighborhood Pattern Sensitive Fault* (NPSF), que são detalhados adiante.

Para descrever os estados das células de memória com ou sem falhas, uma boa maneira é utilizar diagramas de Markov (DAVID; FUENTES; COURTOIS, 1989). Esses diagramas descrevem o comportamento de sistemas em um espaço estado-tempo. Uma célula livre de qualquer defeito pode receber uma operação de escrita para qualquer estado e, quando lida, retêm a informação na célula. A Figura 2.3 mostra o diagrama de Markov para uma célula de memória livre de falhas. Adota-se neste trabalho a notação utilizada em (ADAMS, 2003), onde as transições de estado representam as operações e S_0 e S_1 indicam que a célula está no estado 0 ou 1, respectivamente. A transição R indica uma operação de leitura e W0 e W1 indicam operações de escrita para o estado 0 e 1, respectivamente.

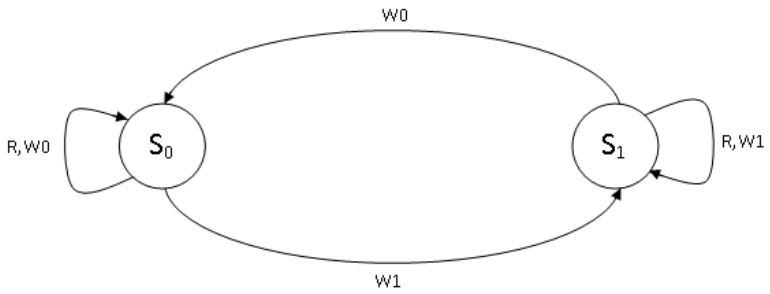

Figura 2.3: Diagrama de Markov para uma célula de memória sem falhas. (ADAMS, 2003)

Stuck-At Fault

Entre os modelos clássicos de falhas em memórias, o mais conhecido e também o mais simples é o modelo SAF, que é utilizado para indicar que uma célula está presa em um determinado estado. A Figura 2.4 mostra o diagrama de Markov de uma falha do tipo SAF. Independente do evento que ocorra, a célula se mantém no seu estado atual.

(a) (b)

Figura 2.4: Diagramas de uma célula de memória com *Sutck-At* 0 (a) e *Stuck-At* 1 (b). (ADAMS, 2003)

Transition Fault

Outro modelo simples de falha é o modelo TF. De certa forma ele se parece com o SAF, mas no caso de uma célula de memória, ela irá reter ambos os estados, mas uma vez escrita para um estado, ela não poderá fazer a transição de volta. Assim, quando a memória é energizada a célula pode estar no estado "0" ou "1" e só pode ser escrita em uma direção. A Figura 2.5 mostra o diagrama de Markov para uma falha do tipo TF.

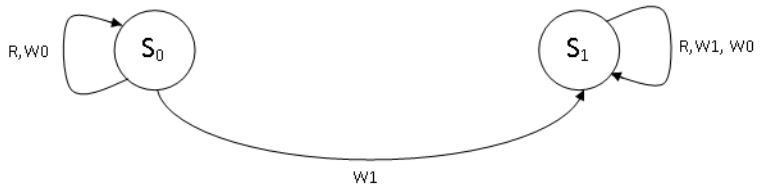

Figura 2.5: Diagrama de uma célula com *Transition Fault*. (ADAMS, 2003)

Coupling Fault

Para ilustrar uma CF é necessário introduzir o diagrama de Markov para duas células. A Figura 2.6 mostra um par de células livres de falha, representadas pelos índices i e j. Cada célula pode ser individualmente lida ou escrita para cada estado independentemente da outra, havendo um total de quatro estados possíveis para o conjunto.

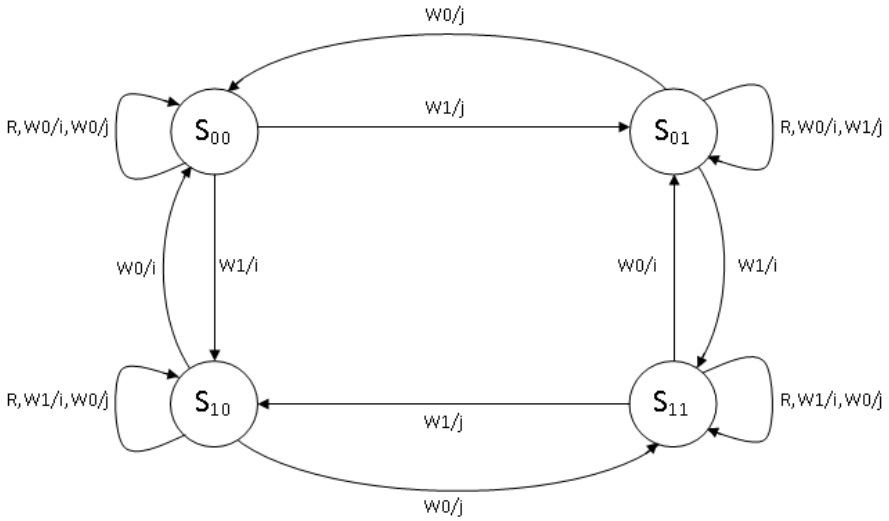

Figura 2.6: Diagrama para duas células sem falhas. (ADAMS, 2003)

Utiliza-se o modelo de falhas CF, ou simplesmente acoplamento, quando há um ou mais pares de células eletricamente acoplados, isto é, quando há interferência eletromagnética entre elas. De forma simplificada, uma célula pode estar acoplada as células da sua vizinhança causando a transição para um estado errôneo ou uma falsa transição.

Os principais fatores que causam a falha CF são a capacitância mútua e a corrente de fuga de uma célula para outra (SUK; REDDY, 1981). Papachristou (PAPACHRISTOU; SAHGAL, 1985) a definiu formalmente como:

> Quando uma operação de escrita que afeta a transição de 0 para 1 ou de 1 para 0 na célula j muda o estado de outra célula i ($i \neq j$), independente do conteúdo da outra célula. Isto não implica que uma transição em i mude o estado de j.

Quando há uma falha CF entre as células, o diagrama de Markov do par passa a ser representado pela Figura 2.7. Como descrito na definição, é possível haver CF em apenas um sentido, onde a célula provoca a falha na outra mas o oposto não acontece. No caso ilustrado, a falha só ocorre quando a célula i está no estado 0 e a célula j faz uma transição de 0 para 1. Caso a célula i, ao invés da j, faça uma transição de 0 para 1, nenhuma falha será provocada. A célula que causa a CF é chamada de célula agressora (ADAMS, 2003) ou acopladora (PAPACHRISTOU; SAHGAL, 1985) e a célula que sofre a falha é chamada de vítima ou acoplada.

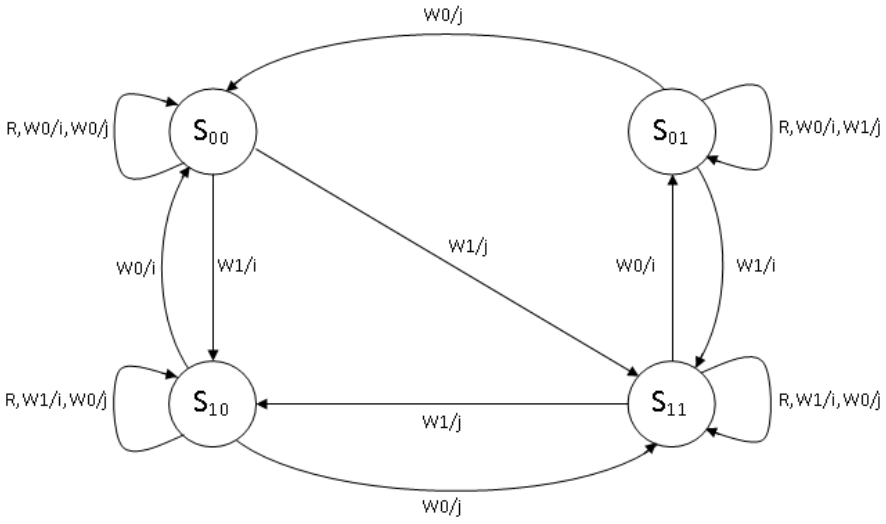

Figura 2.7: Diagrama para duas células com CF. (ADAMS, 2003)

A falha CF pode se apresentar nos seguinte tipos (RAGHURAMAN, 2005):

i. **Inversion Coupling Fault**: Uma transição na célula agressora inverte o conteúdo da célula vítima;

ii. **Idempotent Coupling Fault**: Uma transição na célula agressora força um valor lógico fixo na célula vítima;

iii. **State Coupling Fault**: A célula vítima é forçada para certo estado apenas se a célula agressora estiver em determinado estado. Também conhecida como *Pattern Sensitive Fault* (PSF).

Um conjunto de k células é dito estar k-acoplado (*k-coupled*) se uma transição em uma célula causa uma transição em outra célula do conjunto quando as outras $k-2$ células estão em certo estado. Um caso especial chamado de *restricted k-coupling* é convenientemente definido como um *k-coupling* onde as $k-1$ células não possuem acoplamento entre si. Devido a enorme complexidade dos modelos de falhas para k maior que 3, apenas falhas *2-coupling* e *restricted 3-coupling* têm sido investigadas na literatura (PAPACHRISTOU; SAHGAL, 1985) (ADAMS, 2003).

Neighborhood Pattern Sensitive Fault

O modelo NPSF, se apresenta sobre uma célula chamada de célula-base e sua vizinhança física. Na realidade, o NPSF é um caso especial de *k-coupling* em que as $k-1$ células agressoras estão restritas a uma vizinhança fixa em proximidade física da célula base. Este modelo é bastante estudado por ser mais comum que acoplamentos entre células distantes (PETRU, 2002).

2.1.2 Falhas no Decodificador de Endereço

O decodificador é um circuito de lógica combinacional simples que seleciona uma única célula de memória para um dado endereço. Qualquer falha ocorrida no decodificador fará com que ele se comporte de umas das maneiras:

i. O decodificador não irá acessar a célula endereçada. Além disto, ele pode acessar células não endereçadas;

ii. O decodificador irá acessar múltiplas células, incluindo a célula endereçada.

No caso de múltiplos acessos (ii), podemos ver a falha como uma falha na matriz de células, isto é, como uma CF entre as células afetadas. No caso (i), a célula que deveria ter sido selecionada pode ser vista como *stuck at* 0 ou *stuck at* 1, dependendo da lógica utilizada.

Em todos os casos, podemos visualizar as falhas no decodificador como falhas na matriz de células da memória (NAIR; THATTE; ABRAHAM, 1978) (PETRU, 2002).

2.1.3 Falhas Dinâmicas

As falhas dinâmicas em memórias são também chamadas de falhas na lógica de escrita e leitura (*Read/Write Logic Fault*). Algumas linhas dos circuitos de escrita e leitura podem estar em *stuck at* 0 ou *stuck at* 1. Neste caso, podemos considerar a falha como uma SAF nas células afetadas por essas linhas. As linhas de entrada ou saída de dados das células podem conter curtos ou acoplamentos capacitivos entre elas. Estas falhas podem ser visualizadas como CF entre as células correspondentes.

Portanto, para testes de RAM, todas os tipos de falhas podem ser representados apenas pelos modelos de falhas na matriz de células de memória, isto é SAF, TF, CF e NPSF (NAIR; THATTE; ABRAHAM, 1978; PETRU, 2002).

2.2 Tipos de testes de memória

Uma memória guarda zeros e uns. Se zeros são escritos em todos os endereços da memória e lidos de todos os endereços, então metade das falhas foram cobertas, certo? Na realidade, não.

O teste mencionado é um dos testes mais simples existentes. É chamado de padrão zero-um (*zero-one pattern*) ou padrão trivial. Apesar da simplicidade e deste teste ter cobertura de 100% das falhas SAF, ele é incapaz de detectar a maior parte dos outros tipos de falhas (ADAMS, 2003). A situação descrita na Figura 2.8 é um exemplo onde este teste não é suficiente. Neste caso há uma falha nos decodificadores de forma que, independente do endereço selecionado, sempre a mesma célula é acessada, tanto na escrita quanto na leitura.

Para que seja possível testar memórias de forma mais completa, é necessário usar uma combinação de padrões de teste (*patterns*), onde cada padrão tem a capacidade de detectar certos tipos de falhas. Nenhum padrão sozinho é suficiente para testar uma memória por completo (DEAN; ZORIAN, 1994). Padrões são a essência dos testes

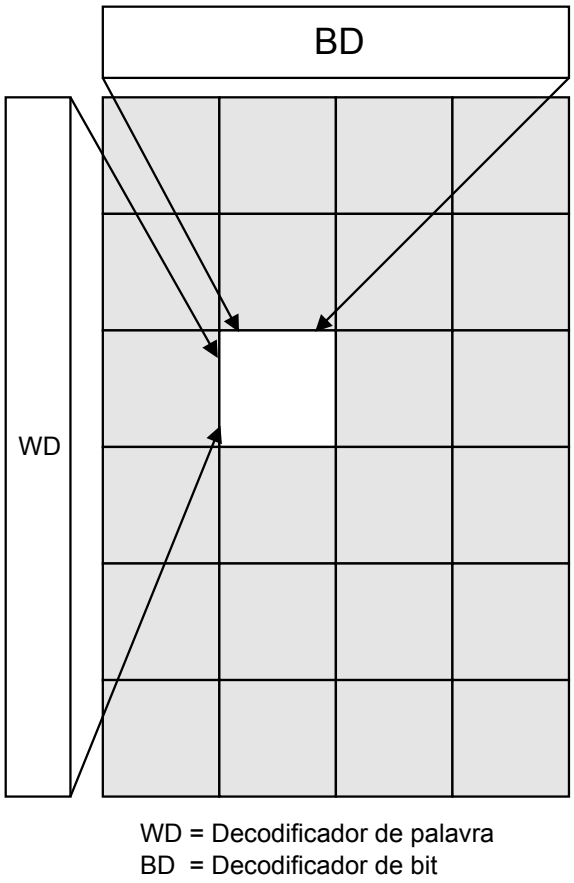

Figura 2.8: Erro não detectável por padrão zero-um.

de memória (ADAMS, 2003).

Para a descrição de testes, adota-se uma notação baseada em (ADAMS, 2003). A Tabela 2.1 mostra a representação do teste zero-um citado anteriormente. Cada linha indica uma sequência de operações que deve ser aplicada a cada célula antes de prosseguir para a próxima sequência. A estas sequências, também dá-se o nome de elementos de teste (PAPACHRISTOU; SAHGAL, 1985). No exemplo temos quatro elementos, do 1 ao 4, cada um com apenas uma operação. As operações podem ser:

- W0 escrever 0 na célula
- R0 ler estado da célula, esperado 0
- W1 escrever 1 na célula
- R1 ler estado da célula, esperado 1

Ao final das operações em um endereço, a passagem de uma célula para outra pode ser de forma ascendente, representada pelo símbolo $\Uparrow$, descendente, representada por $\Downarrow$ ou em qualquer direção, representada por $\Updownarrow$.

Tabela 2.1: *Zero-Ones Pattern.*

1	W0	$\Updownarrow$
2	R0	$\Updownarrow$
3	W1	$\Updownarrow$
4	R1	$\Updownarrow$

Através desta drescrição também é possivel medir a complexidade do teste. Isto é, quantos ciclos são necessários para executar todo o teste. Supondo que uma operação de escrita ou leitura possa ser realizada em um ciclo, o teste apresentado possui uma complexidade $4N$, onde N é a quantidade de células ou o tamanho da memória. Este teste é dito ser de ordem N, representado por $O(N)$. Um teste de complexidade $14N \log_2(N)$ possui ordem $O(N \log_2(N))$. O conceito de complexidade é importante pois os chips de memória crescem constantemente seguindo a Lei de Moore. Isto faz com que testes mais complexos levem tempos impraticáveis para os tamanhos e velocidades das memórias atuais.

Tomando como exemplo uma memória com tempo de acesso de 1,25 ns[1], a Tabela 2.2 mostra os tempos de execução de supostos testes de diferentes complexidades em diferentes tamanhos de memória.

É importante notar que em testes reais os tempos são múltiplos desses valores. Um teste de complixadade $8N$, por exemplo, iria levar $8 \cdot 1,34s = 10,72s$ para testar 1 GB de memória, pois, pela tabela, leva-se $1,34s$ para executar cada N operações. Assim, apenas testes de complexidade até $O(N \log_2(N))$ são aceitáveis (RAGHURAMAN, 2005).

Os padrões de testes de memória são comumente categorizados como caminhantes (*walking*), marchantes (*marching*) e galopantes (*galloping*) (GOOR, 1998).

[1]Tempo de acesso retirado do *datasheet* de uma memória DDR3 SDRAM SODIMM. (MICRON TECHNOLOGY, INC., 2008).

Tabela 2.2: Tempos de execução em memória com tempo de acesso de 1,25 ns.

Tamanho	Complexidade			
N	N	$N \log_2(N)$	$N^{3/2}$	N^2
1 KB	1,28 μs	12,8 μs	40,96 μs	1,31 ms
4 KB	5,12 μs	61,44 μs	327,68 μs	20,97 ms
16 KB	20,48 μs	286,72 μs	2,62 ms	335,54 ms
64 KB	81,92 μs	1,31 ms	20,97 ms	5,37 s
256 KB	327,68 μs	5,90 ms	167,77 ms	1,43 min
1 MB	1,31 ms	26,21 ms	1,34 s	22,91 min
4 MB	5,24 ms	115,34 ms	10,74 s	6,11 h
16 MB	20,97 ms	503,32 ms	1,43 min	4,07 dias
64 MB	83,89 ms	2,18 s	11,45 min	65,16 dias
256 MB	335,54 ms	9,40 s	1,53 h	2,86 anos
1 GB	1,34 s	40,27 s	12,22 h	45,70 anos
2 GB	2,68 s	1,39 min	1,44 dia	182,79 anos
4 GB	5,37 s	2,86 min	4,07 dias	731,18 anos

Testes Caminhantes (*Walking*)

Um teste é dito ser caminhante (*walking*) quando, em cada momento, há apenas uma célula em um estado diferente de todas as outras.

Inicialmente a memória é totalmente preenchida com um padrão, então as operações de escrita e leitura são realizadas em uma célula e ao final da sequência, a célula deve voltar ao estado inicial. A tabela 2.3 mostra um exemplo de elemento para um teste caminhante.

Tabela 2.3: Exemplo de *walking*.

1	R0, W1, R1, W0 $\Uparrow$

O que caracteriza este elemento como caminhante é que a última operação de escrita (W0) retorna a célula para o estado em que ela se encontrava antes do início da sequência (que pode ser notado pela leitura R0).

Testes Marchantes (*Marching*)

Um padrão em marcha (*marching pattern*) é quando o teste muda o estado da célula testada e não a retorna para o estado anterior. Assim, antes de começar o teste, a memória está preenchida com um certo padrão, após uma sequência percorrer metade da memória, metade estará com o padrão inicial e a outra metade estará com o padrão escrito pelo teste. Um exemplo de um elemento *march* é mostrado na tabela 2.4.

Tabela 2.4: Exemplo de *marching*.

1	R0, W1, R1 ⇓

O que diferencia este elemento de um caminhante é que sua última escrita não é necessariamente para o valor inicial da célula.

Testes Galopantes (*Galloping*)

Enquanto os padrões caminhantes e marchantes são orientados a dado, o galopante (*galloping pattern*) é orientado a endereço. A diferença é que esta categoria faz uma checagem do tipo *ping-pong* entre a célula base (célula atualmente testada) e todas as outras da matriz. Como em cada operação é preciso percorrer toda a memória, este tipo de teste leva muito tempo para ser realizado. Sua complexidade é da ordem de $O(N^2)$, o que é bastante, comparada a complexidade de ordem N dos outros dois tipos apresentados.

A cobertura de falhas dos testes galopantes é consideravelmente maior, no entanto o excesso de tempo faz com que sua utilização seja inviável. Como mostrado na Tabela 2.2, um teste que levaria alguns minutos com padrões caminhantes ou marchantes, poderia precisar de anos para ser concluído com um padrão galopante.

2.2.1 Testes Conhecidos

Desde metade do século XX muitos padrões para testes de memória têm sido propostos. Ao longo do tempo eles foram aperfeiçoados tanto no sentido de aprimorar a cobertura de falhas quanto no sentido de reduzir a quantidade de operações realizadas. Neste capítulo trataremos apenas dos algoritmos mais utilizados pela indústria e referenciados na literatura.

Série March

Existe um conjunto tradicional de testes do tipo marchante identificados por letras. Destes, o March C- ganhou destaque em muitos trabalhos por ser muito simples e ainda assim poderoso na detecção de falhas. Seu nome é dado por ser uma otimização de outro padrão, chamado March C, onde algumas ineficiências foram eliminadas. Como pode ser visto na Tabela 2.5, o March C- é um teste $10N$. Ele é capaz de detectar todas as falhas do tipo SAF, *idempotent* CF, TF e parte das falhas de alguns outro tipos (ADAMS, 2003). De acordo com (PETRU, 2002), 81,25% das falhas CF simples entre 2 células são detectadas por este teste.

Tabela 2.5: *March C- pattern.*

1	W0	⇕
2	R0, W1	⇑
3	R1, W0	⇑
4	R0, W1	⇓
5	R1, W0	⇓
6	R0	⇕

Uma melhoria do March C- foi proposta por (ADAMS, 2003). Chamado de Enhanced March C-, o teste, descrito na tabela 2.6, executa $18N$ operações e detecta algumas falhas além das já cobertas pelo March C-.

Tabela 2.6: *Enhanced March C- pattern.*

1	W0	⇕
2	R0, W1, R1, W1	⇑
3	R1, W0, R0, W0	⇑
4	R0, W1, R1, W1	⇓
5	R1, W0, R0, W0	⇓
6	R0	⇕

Moving Invertion

Outro padrão muito conhecido é o *Moving Invertion* (MOVI). Seu funcionamento é um pouco mais complexo que os testes do tipo marchante.

Inialmente toda a memória é preenchida com zeros. Então repete-se o seguinte procedimeno: uma palavra é lida; um bit é escrito para 1; a palavra é lida novamente.

Isto se repete até que todos os bits da palavra estejam com o valor 1 e é feito para todas as palavras da memória. Após completar toda a memória, a operação inversa é aplicada: uma palavra é lida; um bit é escrito para 0; a palavra é lida novamente. Até que todas as palavras estejam com o valor 0 novamente.

Todo este procedimento é repetido n vezes, onde n é o tamanho do barramento de endereço. No entanto, o MOVI utiliza uma forma de endereçamento diferente dos padrões vistos até aqui, chamada de endereçamento não linear. Para cada repetição o endereço é deslocado de forma circular para a esquerda e considera-se o bit n como o *Bit* Menos Significativo (*Least Significant Bit*) (LSB). Por exemplo, na segunda repetição o LSB será o bit 1 e o endereçamento segue a ordem mostrada na tabela 2.7.

Tabela 2.7: *Endereçamento do MOVI na segunda iteração (LSB = bit 1).*

$000\ldots0\underline{0}0$
$000\ldots0\underline{1}0$
$000\ldots1\underline{0}0$
$000\ldots1\underline{1}0$
$\vdots$
$111\ldots1\underline{1}0$
$000\ldots0\underline{0}1$
$000\ldots0\underline{1}1$
$\vdots$

Este teste realiza $12nN\log_2(N)$ operações e detecta falhas de endereçamento, SAF, TF e problemas com o tempo de acesso (PHAN, 2002).

Nair

Um dos grandes trabalhos em testes de memória é (NAIR; THATTE; ABRAHAM, 1978), onde foram propostos dois algoritmos com boa eficiência e cobertura de falhas. O primeiro, chamado de algoritmo A, é de ordem $O(N)$ e detecta todas a falhas simples e acoplamentos até 2 células. O segundo, algortimo B, é de ordem $O(N\log_2(N))$, mas detecta as mesmas falhas do algortimo A com o acréscimo de acoplamentos entre 3 células do tipo restrito (*restricted 3-coupling faults*, ver final da sessão 2.1.1).

Papachristou

O Papachristou é um dos testes mais conhecidos e utilizados e também um dos que apresenta melhor cobertura de falhas com complexidade praticável. Proposto por (PAPACHRISTOU; SAHGAL, 1985), inspirado nos algoritmos A e B de (NAIR; THATTE; ABRAHAM, 1978), o autor desenvolveu um algortimo capaz de detectar todas as falhas detectáveis por A e B, além algumas do tipo NPSF. O teste pode ser dividido em duas etapas, uma de ordem $O(N)$ com a mesma cobertura que o algoritmo A e a segunda de ordem $N\log_2(N)$. No total, são necessárias $38N + 24N\log(N)$ operações.

O teste de $38N$ operações é do tipo marchante e será chamado de Papachristou 1 ou parcial neste trabalho. Ele é equivalente ao Nair A quanto às falhas detectadas e seus elementos são descritos na tabela 2.8.

Tabela 2.8: Algoritmo Papachristou 1.

1	W0	$\Updownarrow$
2	R0, W1, R1	$\Uparrow$
3	R1, W0, R0	$\Uparrow$
4	R0, W1, W0	$\Uparrow$
5	R0, W1	$\Uparrow$
6	R1, W0, W1	$\Uparrow$
7	R1, W0	$\Uparrow$
8	R0, W1, W0	$\Uparrow$
9	R0, W1	$\Downarrow$
10	R1, W0	$\Downarrow$
11	R0, W1, W0	$\Downarrow$
12	R0, W1	$\Downarrow$
13	R1, W0, W1	$\Downarrow$
14	R1, W0	$\Downarrow$
15	R0, W1, W0	$\Downarrow$
16	R0	$\Updownarrow$

O Papachristou 2 ou completo, como será chamado o procedimento inteiro, consiste na aplicação do teste anterior seguido de um segundo teste com as operações descritas na tabela 2.9 utilizando o mesmo endereçamento não linear do MOVI. Um ponto importante é que este teste divide a memória em duas metades e as

operações são realizadas na metade superior ou na inferior, definidas pelo *Bit* Mais Significativo (*Most Significant Bit*) (MSB) do endereço após o deslocamento cíclico do endereçamento não linear.

Tabela 2.9: Segunda etapa do algoritmo Papachristou 2.

1	R0, W1	$\Uparrow t$
2	R0	$\Uparrow b$
3	R0, W1	$\Uparrow b$
4	R1	$\Uparrow t$
5	R1, W0	$\Uparrow t$
6	R1	$\Uparrow b$
7	R1, W0	$\Uparrow b$
8	R0	$\Uparrow t$
9	R0, W1	$\Downarrow b$
10	R0	$\Downarrow t$
12	R0, W1	$\Downarrow t$
12	R1	$\Downarrow b$
13	R1, W0	$\Downarrow b$
14	R1	$\Downarrow t$
15	R1, W0	$\Downarrow t$
16	R0	$\Downarrow b$

MT

Nas últimas décadas pouco se tem avançado em relação à cobertura de falhas dos algoritmos de teste de memória. As novas propostas concentram-se na criação de padrões muito específicos para tipos particulares de falhas, além do desenvolvimento de *hardware* e *cores* acoplados ao próprio circuito de memória para facilitar a aplicação de testes automáticos. Estes *hardwares* fazem parte de um conceito chamado de *Design for Testability* (DFT), que é um conjunto de técnicas que adicionam certas características de testabilidade a circuitos microeletrônicos. Em memórias, é utilizado particularmente circuitos de *Built-in Self-test* (BIST), que são mecanismos que permitem ao hardware fazer testes de forma automática.

Entre os padrões mais recentes ganha destaque um algoritmo chamado de MT (PETRU, 2002). É um teste do tipo marchante com apenas $36N$ operações, mas

que detecta todas as falhas detectadas por Papachristou 2, de ordem $O(N \log_2(N))$, além de cobrir todas as falhas do tipo *3-coupling fault* entre células adjacentes, não apenas as do tipo restrito. Em outras palavras, este teste é capaz de detectar todas as falhas NPSF entre 3 células.

Para isto, a descrição do teste segue uma abordagem um pouco diferente. Ao invés de escrever sempre o mesmo valor em toda a memória, as células são preenchidas com os padrões de fundos mostrados nas tabelas 2.10, nomeados de I_1 a I_6.

(a) I_1

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

(b) I_2

0	1	0	1
0	1	0	1
0	1	0	1
0	1	0	1

(c) I_3

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

(d) I_4

1	0	1	0
1	0	1	0
1	0	1	0
1	0	1	0

(e) I_5

0	0	0	0
1	1	1	1
0	0	0	0
1	1	1	1

(f) I_6

1	1	1	1
0	0	0	0
1	1	1	1
0	0	0	0

Tabela 2.10: Padrões de fundo no algoritmo MT.

O teste consistem em preencher a memória com estes padrões e executar uma sequência de operações de leitura e escrita. Diferente dos outros testes apresentados, os elementos utilizados para sua descrição usam as operações I_x, R e WC (tabela 2.11), que significam respectivamente escrever o padrão de fundo I_x em toda a memória, ler 0 ou 1 dependendo do padrão e do endedreço atual e escrever o complemento do valor presente na célula.

GALPAT

O teste tomado como referência, em termos de cobertura de falhas, é um algoritmo do tipo galopante chamando GALPAT (NAIR; THATTE; ABRAHAM, 1978) (PAPACHRISTOU; SAHGAL, 1985) (RIEDEL; RAJSKI, 1995). Ele cobre praticamente

Tabela 2.11: *MT pattern.*

1	I_1	⇳
2	R, WC, R, WC	⇑
3	R	⇳
4	I_2	⇳
5	R, WC, R, WC	⇑
6	R	⇳
7	I_3	⇳
8	R, WC, R, WC	⇑
9	R	⇳
10	I_4	⇳
11	R, WC, R, WC	⇑
12	R	⇳
13	I_5	⇳
14	R, WC, R, WC	⇑
15	R	⇳
16	I_6	⇳
17	R, WC, R, WC	⇑
18	R	⇳

todos os tipos de falhas conhecidos, no entanto é considerado um teste meramente teórico, pois sua complexidade é de ordem $O(N^2)$.

Para a implementação de testes de memória, não basta conhecer os algortimos. Estes descrevem apenas o conjunto de operações capazes de detectar as falhas, porém uma série características do SO precisa ser levada em considereção. As características pertinentes ao Linux serão apresentadas na próxima seção.

2.3 Gerenciamento de memória do Linux

Para projetar um bom teste que execute sobre toda a abstração imposta por um SO, é necessário conhecer a fundo como o *hardware* em questão é tratado pelo sistema. No caso da memória, o responsável por esta manipulação é o subsistema chamado de gerenciador de memória. Ele é o mecanismo que prover meios para, dinamicamente, alocar porções de memória para os programas que solicitarem e liberá-la para reuso quando não forem mais necessárias (KNUTH, 1997). No

Linux, o subsistema responsável pelo gerenciamento de memória é o Linux *Memory Management* (LinuxMM) (LINUXMM, 2010).

2.3.1 Memória Virtual

Os gerenciadores de memória utilizam o conceito de memória virtual para melhorar a eficiencia em sistemas multitarefas. A memória virtual permite que o gerenciador organize a memória independentemente da disposição física dos circuitos. As aplicações acessam a memória apenas através de endereços virtuais. Cada vez que é feita uma tentativa de acesso, o gerenciador traduz o endereço virtual em um endereço físico, que corresponde a localização do dado como armazenado no hardware (GLASER JOHN F. COULEUR, 1965).

Os *chips* de memória são fabricados como uma matriz de células e são montados, tipicamente, divididos em bancos, que nada mais são que placas contendo alguns CIs de memória e controladores, como na Figura 2.9(a). Graças a memória virtual, a aplicação acessa sua porção de memória como um simples vetor de bytes unidimensional, como mostrado na Figura 2.9(b).

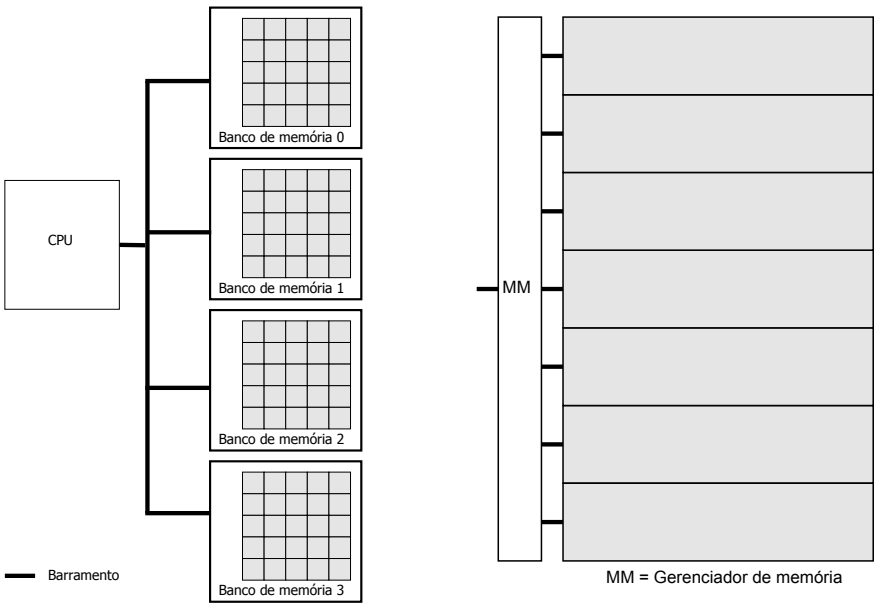

Figura 2.9: Topologia da memória como fisicamente disposta (a) e como vista por uma aplicação (b).

Além disto, porções fragmentadas da memória física podem ser alocadas para

um processo, mas o gerenciador virtualiza estes fragmentos em um espaço contínuo de endereçamento, como ilustrado na Figura 2.10.

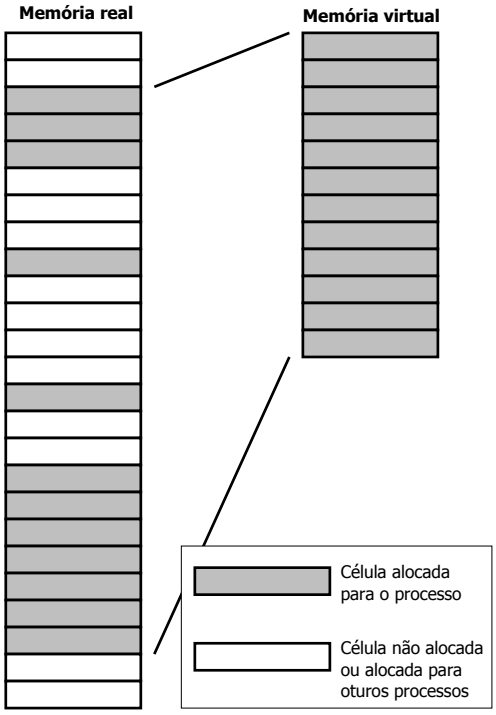

Figura 2.10: Memória virtual como alocada para um processo (adaptado de (TANENBAUM, 2001)).

A paginação é outro mecanismo presente nos gerenciadores e está geralmente associado à memória virtual. Isto faz com que o sistema guarde porções de dados (páginas) da memória principal em uma memória secundária, recuperando-as posteriormente quando seu uso for solicitado. Desta forma, o sistema expande a memória virtual disponível para as aplicações. Geralmente se utiliza como memória secundária um espaço reservado no disco (HD, SSD, etc). Estes dispositivos são mais lentos que as RAMs mas possuem maior disponibilidade de armazenamento e são mais baratos, se levado em conta o valor por byte. (TANENBAUM, 2001)

No Linux, o espaço reservado para paginação é chamado de *swap* e pode estar presente ou não em um sistema, assim como pode ser habilitado, desabilitado ou redimensionado em tempo de execução.

Como o processo de salvar e recuperar as páginas do disco é lento, o Linux evita utilizar a área de *swap* até que a disponibilidade de memória livre esteja baixa.

2.3.2 Escassez de Memória

A Escassez de Memória (*Out Of Memory*) (OOM) é um estado de um sistema computacional, geralmente indesejado, onde nenhuma memória adicional pode ser alocada para uso pelos programas ou pelo próprio sistema operacional.

No caso do Linux, para lidar com este tipo de situação é utilizada uma ferramenta, parte do subsitema LinuxMM, chamada de OOM *Killer*, que consiste em uma tarefa que sacrifica um ou mais processos para liberar memória para o sistema (LINUXMM, 2009).

Programas que utilizam muita memória podem esgotar a memória do sistema, fazendo-o deixar de funcionar. Isto pode, por exemplo, levar a uma situação em que há tão pouca memória que o kernel não pode alocar memória para executar uma operção de liberação de memória. Então, neste caso, o OOM Killer é acionado e identifica os processos a serem sacrificados para beneficiar o resto do sistema.

O OOM Killer usa um esquema de pontuação para decidir qual ou quais processos sacrificar. Os pontos são dados pela função *badness*, que utiliza uma fórmula relativamente simples, documentada no próprio código. As regras para gerar a pontuação são:

- perder o mínimo de trabalho realizado;
- recuperar a maior quantidade de memória;
- não matar (*kill*) nada que tenha utilizado pouca memória;
- matar o mínimo de processos (de preferência apenas um);
- tentar matar o processo que o usuário espera que vá ser morto (menor surpresa).

2.3.3 Cache de Memória

Uma característica do LinuxMM do mecanismo de *cache* de memória. Ele é manipulado pelo Linux Page Cache e funciona da seguinte forma: no Linux, quando um arquivo é acessado, seu conteúdo é copiado para a memória para ser trabalhado (lido e/ou escrito); então, quando o processo termina, o *kernel* pode liberar aquela

memória ou mantê-la em *cache* para caso algum outro processo o acesse. O mesmo ocorre com algumas outras formas de alocação da memória.

Devido ao *cache* em sistemas Linux, após algum tempo de execução de processos, apenas uma pequena porção da memória pode permanecer marcada realmente como livre. A maior parte está sempre preenchida com conteúdo útil, mas nem sempre isso significa que esteja sendo utilizada. Por outro lado há um ganho de desempenho considerável ao acessar recursos recentemente utilizados (em *cache*).

Um conteúdo que esteja em *cache* pode ser liberado quando há uma solicitação por alocação de memória e não há memória livre para atender. Neste caso, o conteúdo do *cache* é descartado e a memória é cedida ao processo que solitou a alocação.

2.4 Resumo do Capítulo

Este capítulo apresentou, de maneira resumida, alguns fundamentos da área de testes de memória que são úteis para melhor compreensão deste trabalho, mostrou os vários tipos de falhas e testes estudados, dando ênfase nas falhas mais comuns e nos testes de complexidade realizável na tecnologia atual. Também foram apresentadas algumas características de como o Linux gerencia a memória física do sistema.

No capítulo seguinte, serão apresentadas as ferramentas utilizadas para o desenvolvimento, teste e validação, além de detalhes da implementação da ferramenta concebida neste trabalho.

Capítulo 3

Metodologia

Neste capítulo são descritos os ambientes utilizados no desenvolvimento, avaliação e teste da ferramenta proposta, chamada de MDiag. Algumas características importantes da implementação, que garantem a eficácia do diagnóstico, são apresentadas na Seção 3.2. Um sistema automático de inserção de falhas desenvolvido para validação do MDiag através de simulação, é detalhado na Seção 3.3. Por fim, é descrito o procedimento de testes em ambientes reais.

3.1 Ambiente de Desenvolvimento e Teste

Para o desenvolvimento deste trabalho, foram utilizados diversos tipos de computadores a fim de garantir compatibilidade com a maior variedade possível de plataformas. Sistemas variando desde kit de desenvolvimento de sistema embarcado até servidores de grande porte com distribuições Linux variadas. Todas as máquinas utilizadas, com exceção de um computador pessoal do autor, fazem parte do acervo do Laboratório de Engenharia de Sistemas de Computação (LESC) da Universidade Federal do Ceará (UFC).

Nenhum tipo de bibliotecas, além das padrões do Linux, foram necessárias durante o desenvolvimento, pois, para favorecer a portabilidade, a ferramenta utiliza apenas funcionalidades presentes no próprio *kernel* do Linux, a partir da versão 2.6.9.

Para simular as falhas em memória, foi utilizada uma abordagem baseada em (PETRU, 2002) (descrita com detalhes na seção 3.3). Um *software* de *debug* com capacidade de estabelecer *breakpoints* em nível de *hardware* foi utilizado para interromper a execução do teste no momento desejado e escrever um valor de erro na

memória, simulando qualquer tipo de falha. O *software* utilizado foi o GNU Project Debugger (GDB), uma das mais consagradas ferramentas de *debug* para Linux.

Foram realizados testes com memórias defeituosas reais e os resultados foram comparados aqueles obtidos por outras ferramentas de diagnóstico do mercado. Os componentes e o procedimento desta avaliação são detalhados na Seção 3.4

3.2 Implementação do MDiag

Para o desenvolvimento de um diagnóstico que atua sobre um sistema operacional, alguns cuidados precisam ser tomados para garantir a eficácia dos testes. Como visto na Seção 2.3, cada acesso à memória passa por uma série de abstrações até chegar ao *hardware*. Isto pode acarretar falsos resultados ou ineficiências no diagnóstico.

Por exemplo, durante o teste o *kernel* pode guardar parte da memória alocada no *swap*, enquanto o restante é testado. Depois, essas páginas podem ser recuperadas e a parte testada pode ser armazenada. A porção resgatada pode estar em qualquer lugar do espaço físico destinado ao processo, até mesmo no lugar da porção que já foi testada, causando uma dupla checagem nestas células e deixando de testar outras.

Por isso o MDiag executa uma série de procedimentos, mostrados na Figura 3.1, antes de executar os algoritmos de teste. O conjunto de operações desde a limpeza do *cache* até a alocação, de fato, da memória é um mecanismo projetado neste trabalho chamado de política de alocação de memória do MDiag, que visa maximizar quantidade de memória coberta pelo diagnóstico sem comprometer estabilidade do sistema.

3.2.1 Política de alocação de memória

É impossível que um diagnóstico implementado como um aplicativo, que executa sobre um Linux sem alterações nos mecanismos de proteção do *kernel*, possa testar toda a memória instalada, isto porque certa quantidade de memória, chamada de área do sistema, é reservada para o próprio SO guardar suas estruturas de dados, executar e gerenciar as aplicações. Muitas outras aplicações executando em paralelo consomem outras porções da área restante. No entanto, quanto mais memória for testada, mais efetivo o diagnóstico será, possibilitando detectar mais falhas. Por isso a política de alocação foi tratada com bastante critério durante este projeto.

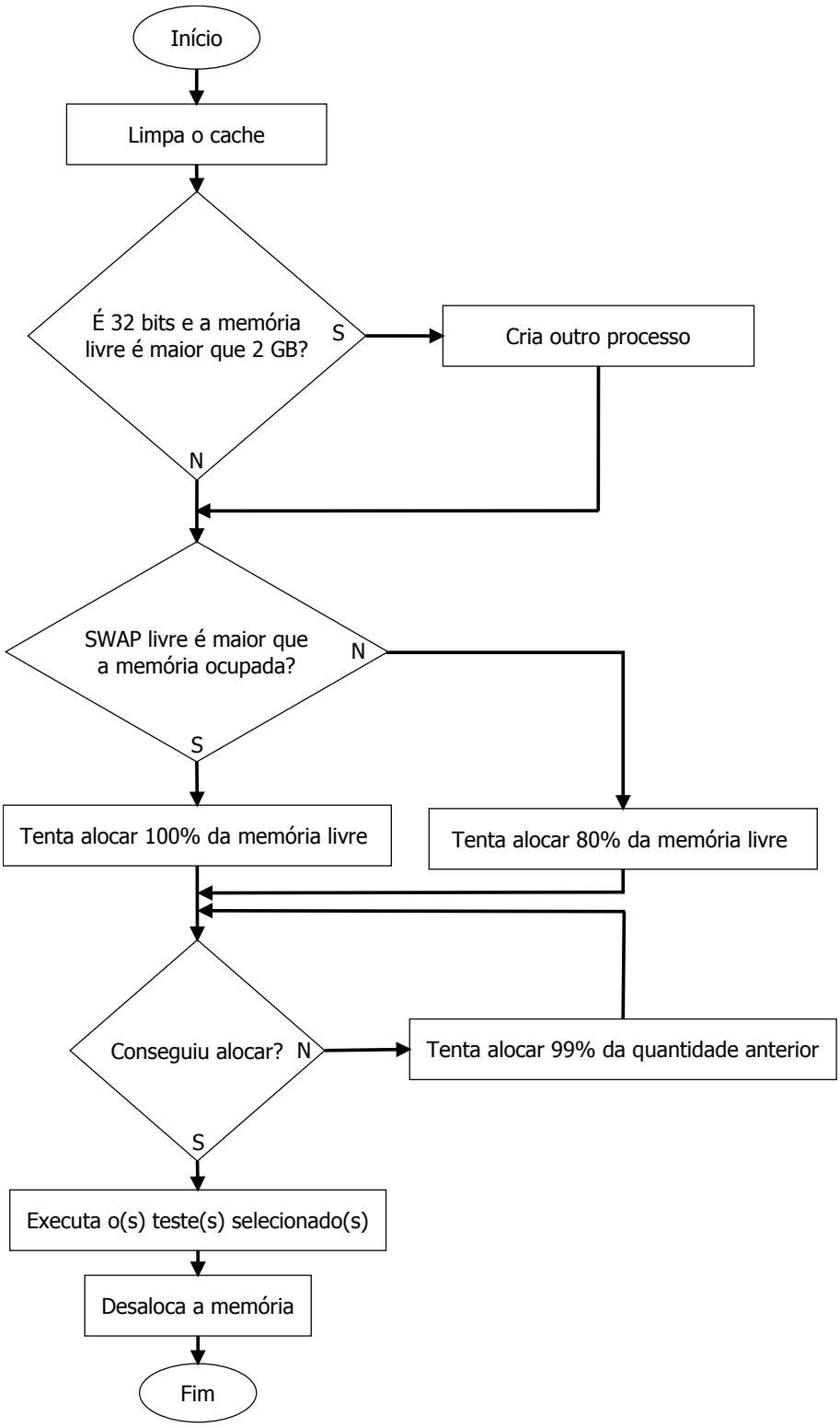

Figura 3.1: Fluxo de execução do MDiag.

O Linux possui, simplificadamente, três estados de memória: alocada, em *cache* e livre. O mecanismo de *cache* foi explicado na Seção 2.3. Como foi dito, após algum tempo em operação a tendência é que apenas uma pequena parte da memória permaneça realmente livre, a maior parte estará sendo utilizada como *cache* ou alocada para algum processo. O MDiag aloca apenas a porção livre da memória, para evitar que o sistema sofra de OOM. Por isso o primeiro passo é a limpeza do *cache*, liberando qualquer parte dispensável da memória e, consequentemente, aumentando a área testada.

Em seguida há o tratamento de uma limitação de sistemas 32 bits. Nestes sistemas o endereçamento máximo acessível por um processo é de 4 GB. O Linux possui um mecanismo chamado *HighMemory* que permite que um *kernel* 32 bits acesse mais de 4 GB de memória física em um *hardware* 64 bits. No entanto, isto permite apenas que mais processos sejam alocados por vez, mas cada um deles ainda terá acesso a, no máximo, 4 GB de memória. Além disto, dentro deste espaço há uma área reservada para que o *kernel* controle aquele processo, além de áreas utilizadas como memória de código e pilha. Assim, o máximo que uma aplicação consegue alocar para uso próprio varia tipicamente em torno de 3 GB.

Para contornar esta limitação, o MDiag verifica se o sistema é 32 bits e se a memória livre é maior que 2 GB. Neste caso, o programa se duplica em dois processos idênticos, mas totalmente independentes (*fork*). Isto faz com que dois testes com os mesmos parâmetros sejam executados simultaneamente, cada um fazendo sua própria tentativa de alocação e ampliando a memória total testada. É claro que ainda assim pode acontecer de nem toda a memória livre ser alocada, mas o limite é dobrado para aproximadamente $6GB$.

A alocação de memória no MDiag é um processo de duas etapas. Primeiramente há a alocação em si (*malloc*), isto é, solicitar ao *kernel* uma porção de memória de tamanho fixo para ser utilizada pela aplicação. Uma vez concedida, esta região deve ser travada (*mlock*). Isto significa que o processo indica ao *kernel* que aquela região de memória não pode ser armazenada em *swap*, garantindo que tudo o que foi alocado esteja realmente na memória física da máquina. Se uma das etapas receber resposta negativa do *kernel*, o processo de alocação falhou. Neste caso, é feita uma nova tentativa de alocação com 99% da quantidade pretendida anteriormente. Este processo se repete até que se obtenha sucesso ou até que a quantidade pretendida se torne abaixo de 10 MB.

Outra medida tomada, na política de alocação, para evitar OOM é de não alocar toda a memória virtual do sistema. Isto é feito assegurando-se de que há espaço suficiente no *swap* para armazenar toda a memória atualmente em uso, se necessário. Caso contrário, apenas 80% da memória livre é alocada. Este número foi alcançado de forma empírica com testes em diversas máquinas reais com diferentes distribuições Linux, diferentes tamanhos de memória e diferentes perfis de uso (muitos processos ou poucos processos). É o maior percentual em que se notou um baixíssimo risco de OOM.

Após passar por toda a política de alocação, finalmente os testes podem ser aplicados sequencialmente à região de memória alocada.

3.2.2 Algorítmos Implementados

Neste trabalho foram utilizados seis algoritmos, permitindo realizar testes mais rápidos ou testes com maior cobertura de falhas. Foram escolhidos os algoritmos de maior reconhecimento na literatura, citados em praticamente todos os artigos e livros da área, com uso consagrado na indústria e com resultados comprovados em análises comparativas de testes de memória (RIEDEL; RAJSKI, 1995; RAGHURAMAN, 2005).

March C-

O March C- é um teste ainda muito utilizado por possuir duas características bastante fortes: entre os testes do tipo marchante, é um dos que possui maior cobertura de falhas; é um teste rápido, com complexidade de apenas $10N$ operações.

No MDiag foi implementada também a variação proposta por (ADAMS, 2003), chamada de Enhanced March C- e descrita na tabela 2.6. Esta forma melhorada do March C- é mais lenta, com $8N$ operações a mais, mas possui uma cobertura de falhas um pouco maior.

As descrições destes testes já foram apresentadas na seção 2.2, especialmente nas Tabelas 2.5 e 2.6.

March G

Da série de testes March, o que obteve melhores resultados até hoje foi proposto por (GOOR, 1998). O March G (tabela 3.1) introduz um novo tipo de elemento além

das escritas e leitura convencionais. É uma pausa entre as sequências, que possibilita a detecção de falha de retenção (*data retention fault*).

Tabela 3.1: *March G pattern.*

1	W0	⇳
2	R0, W1, R1, W0, R0, W1	⇑
3	R1, W0, W1	⇑
4	R1, W0, W1, W0	⇓
5	R0, W1, W0	⇓
6	pausa	
7	R0, W1, R1	⇳
8	pausa	
9	R1, W0, R0	⇳

Papachristou

Para um teste mais completo, o algoritmo adotado foi o proposto por (PAPACHRISTOU; SAHGAL, 1985). É um teste longo, que demanda $38N + 24N\log_2(N)$ operações, o equivalente a $16,9$ minutos para testar 1 GB ou mais de uma hora para testar 4 GB de memória nas condições da tabela 2.2. Apesar de ser um padrão antigo, sua abrangência na detecção de falhas vem sendo confirmada por trabalhos mais recentes (RIEDEL; RAJSKI, 1995) (PETRU, 2002).

No MDiag, foram implementados os algoritmos parcial e completo de Papachristou. Ambas as variações foram apresentados na Seção 2.2.

Todos os algoritmos até aqui foram implementados tomando como células os bytes da memória, portanto cada célula possui 8 bits de tamanho. Os estados 0 e 1 em que a célula pode estar representam um padrão qualquer de 00_h a FF_h e seu inverso (complemento bit-a-bit), permitindo que os testes detectem erros mais variados. Por exemplo, o March C- é capaz de detectar *idempotent* CF se utilizado o estado 0 como o valor 00_h e, por consequência, estado 1 como o valor FF_h, mas não é capaz de detectar *inversion* CF. Já com a utilização do padrão 55_h como o estado 0 e seu inverso AA_h como estado 1, ocorre exatamente o oposto.

MT

O mais recente dentre os algoritmos implementados é o MT (Tabela 2.11). É um teste que possui uma cobertura tão boa quanto Papachristou, mas de complexidade $O(N)$.

Como visto em sua descrição na Seção 2.2, este teste trabalha com base na disposição matricial das células da memória. Desta forma, sua implementação foi um pouco diferente dos demais. Primeiramente, para manter a estrutura matricial pressuposta pelo algoritmo, as células foram tomadas como sendo os *bits* e não mais o *bytes* da memória. Assim, tem-se uma matriz $8 \times N$ de células, podendo-se aplicar os padrões de fundo da tabela I_1 a I_6 da 2.10. Da mesma forma, devido a utilização de padrões já bem definidos no algoritmo, este teste não permite escolher o que os estados 0 e 1 significam, mesmo porque as células agora são apenas bits, que só assumem os valores 0 e 1.

3.2.3 Desalocação da Memória

O próximo passo, consiste na desalocação de toda a memória. Da mesma forma que a alocação, este também é um processo de duas etapas. Primeiro a memória é destravada, para então ser desassociada do processo.

É importante garantir que a memória seja devidamente desalocada, mesmo no caso do programa ter sua execução interrompida, seja pelo usuário ou pelo *kernel*. Isto porque a quantidade de memória reservada para o diagnóstico representa uma porção significativa do total disponível, além da região estar travada, não podendo nem mesmo ser despejada para *swap*.

O MDiag utiliza um manipulador de sinal (*signal handler*) que nada mais é que uma função a ser executada sempre que o *kernel* enviar certo sinal. Este manipulador monitora um sinal do *kernel* que é disparado sempre que há uma tentativa de interrupção ao processo. O manipulador deve ser rápido, pois o *kernel* envia este sinal como aviso de que o programa será fechado, e caso o processamento não cesse, o *kernel* poderá matar o processo abruptamente.

3.3 Inserção de Falhas

A fim de comprovar a eficácia dos testes, foi desenvolvido um ambiente de validação automático com inserção de falhas via *debugging*.

O método utiliza elementos de *debugging* presente na maioria dos processadores atuais. O GDB foi usado como interface de *software* para explorar esses recursos do *hardware*. Foram utilizadas principalmente duas estruturas: *breakpoint* e *watchpoint*. O primeiro é apenas um ponto de parada na execução do programa, congelando-o até que seja dado o comando para continuar. O segundo é similar ao primeiro, mas o critério de parada não está associado a um ponto específico do código, mas a uma variável, função ou posição da memória. A execução é interrompida cada vez que houver uma tentativa de leitura ou escrita no endereço monitorado.

3.3.1 Falhas simuladas

As falhas inseridas foram limitadas aos erros mais desafiadores para os algoritmos de teste de memória, eliminando redundâncias no sistema e otimizando o tempo de validação. Os modelos mais simples, como SAF e TF, não foram levados em consideração, pois todos os algoritmos apresentados possuem cobertura de 100% na sua detecção (RIEDEL; RAJSKI, 1995) (PETRU, 2002) (RAGHURAMAN, 2005). Foram escolhidas apenas falhas representativas, em que é esperado que os testes não possam detectar todas as variações possíveis.

Os modelos de falha que apresentam maior dificuldade na detecção são os de acoplamento. Tanto os CF simples, como NPSF ou, de forma mais geral, falhas de *k-coupling*. Para a validação foram inseridas falhas do tipo *idempotent* CF, *inversion* CF e *3-coupling fault*. Estas foram aplicadas apenas entre células vizinhas, pois, além de ser a situação mais comum encontrada em memórias reais (PETRU, 2002), cobre as três ordenações possíveis de acoplamento: a vítima (célula que sofre transição errônea) acima da agressora (célula que dita o estado da vítima), a vítima abaixo da agressora e a vítima e a agressora no mesmo byte.

Da mesma forma, para eliminar redundância no sistema e otimizar o tempo de validação, os *bits* utilizados foram limitados aos mais representativos. As falhas se combinam apenas entre os bits 0, 1, 6 e 7 de cada *byte*. Assim os casos cobertos são: *bits* nas bordas da célula, *bits* fora das bordas, *bits* vizinhos, *bits* distantes, vítima a direita da agressora e vítima a esquerda da agressora.

De forma mais simples, cada falha foi inserida com todas as combinações entre acoplado e acoplador nos bits destacados na figura 3.2.

Para cada modelo simulado, as combinações possíveis são dadas por: $(b \cdot 3 - 1) \cdot b$ para acoplamento entre duas células e $(b \cdot 3 - 2) \cdot (b \cdot 3 - 1) \cdot b$ para três, com b sendo

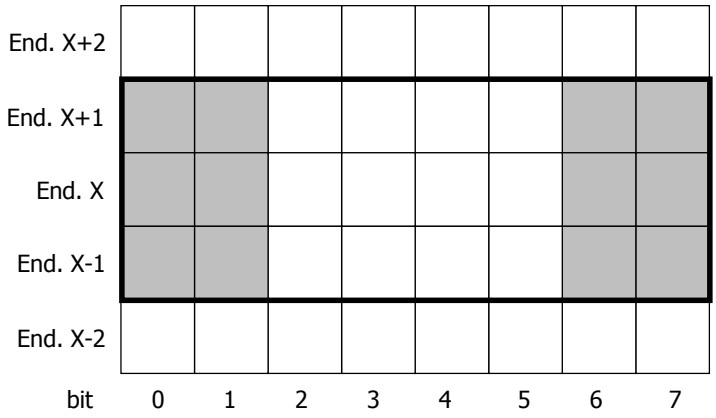

Figura 3.2: Bits atingidos pela inserção de falhas.

a quantidade de *bits* em que as falhas podem ocorrer em cada *byte*. Para a falha *3-coupling fault* há duas possibilidades: que a célula vítima sofra transição quando uma das outras for escrita e a terceira esteja no estado 0 ou quando esta esteja no estado 1.

Portanto, o total de combinações de falhas geradas para b = 4 é:

$2 \cdot (4 \cdot 3 - 1) \cdot 4 = 88$

$+$

$2 \cdot (4 \cdot 3 - 2) \cdot (4 \cdot 3 - 1) \cdot 4 = 880$

$= 968$ falhas.

3.3.2 Sistema de inserção de falhas

A Figura 3.3 mostra os passos do sistema de inserção de falhas elaborado para validar o MDiag.

As falhas são geradas através de um *script* que escreve arquivos com os modelos das falhas utilizando comandos próprios do GDB.

Para otimizar o tempo de validação, as falhas são inseridas em conjuntos. A quantidade é limitada pelo máximo de *watchpoints* que o processador utilizado suporta. Neste trabalho foi possível inserir três falhas a cada execução do ciclo maior da figura 3.3. O endereço de cada uma é gerado a uma distância de 3 células da anterior, possibilitando que os arranjos da figura 3.2 não se sobreponham. Assim,

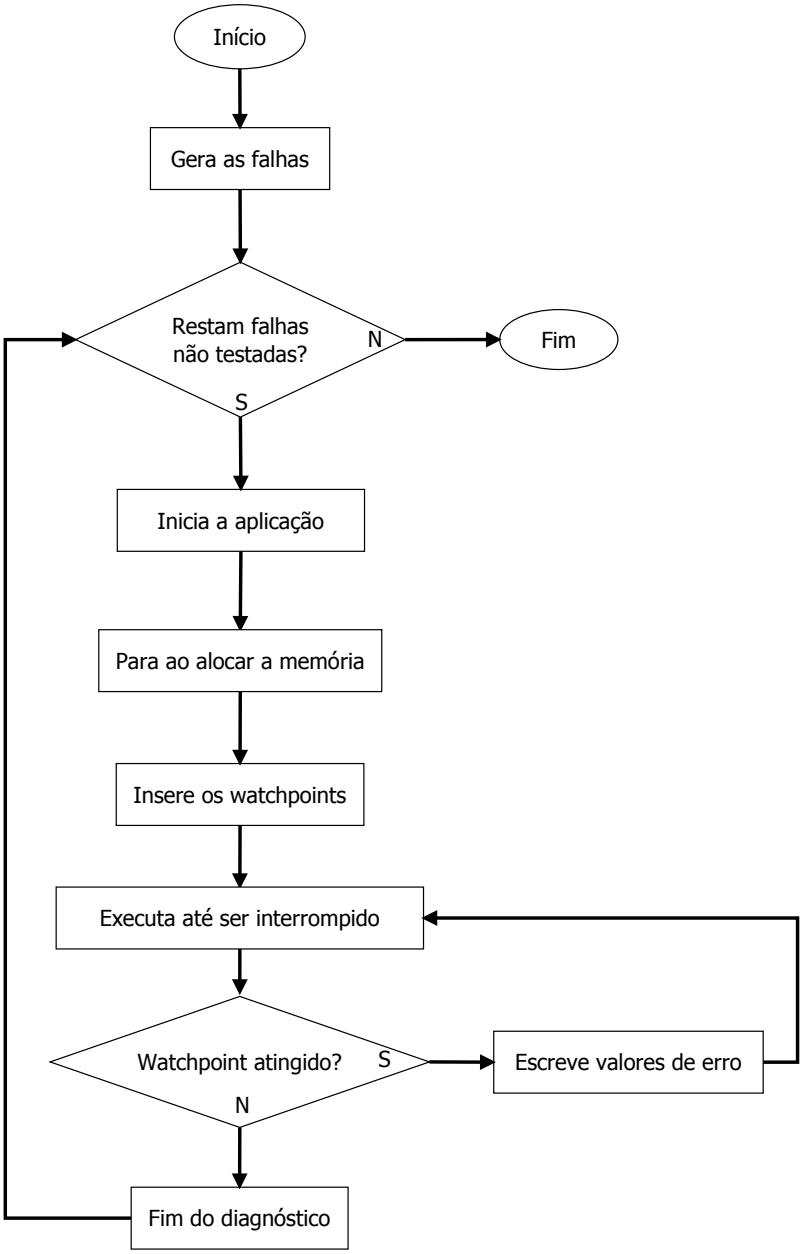

Figura 3.3: Fluxograma do método de inserção de falhas.

foram necessárias 323 iterações para cobrir todas as falhas testadas. A quantidade de memória necessária para comportar n falhas é de $4 \cdot n - 1$. Portanto, apenas 11 bytes são suficientes para comportar as falhas inseridas em uma iteração. Por este motivo, a quantidade de memória alocada pelo MDiag, durante este processo

de validação, foi forçada para o máximo de 1 MB, diminuindo bastante o tempo de teste de cada algoritmo.

O MDiag foi desenvolvido de modo a guardar em um arquivo de relatório todos os acontecimentos relevantes durante a sua execução, como a quantidade de memória alocada, tempo de execução, falhas encontradas, etc. Estes relatórios passaram por um tratamento automático posterior para se obter os resultados relevantes. Estes resultados são apresentados e discutimos na próxima seção.

3.4 Teste em Ambientes Reais

Além do ambiente de simulação de falhas descrito na seção anterior, o MDiag também foi submetido a situações de uso reais a fim de garantir sua utilidade prática.

O acervo utilizado para teste foi composto por dez placas de memória, algumas em perfeito funcionamento, outras com falhas. Metade delas possuíam encapsulamento *Small Outline Dual In-Line Memory Module* (SO-DIMM), próprias para computadores de dimensões reduzidas, como *notebooks* e *netbooks*, e as outras, encapsulamento *Dual In-Line Memory Module* (DIMM), geralmente usadas nos computadores pessoais comuns, estilo *desktop*, ou em servidores.

Nesses testes, o MDiag confrontou dois *softwares* de diagnóstico consagrados no mercado. O primeiro foi o Lenovo *ThinkVantage Toolbox* (LTT) (LENOVO, 2011), desenvolvido pela PC-Doctor (PC-DOCTOR, 2011) para os computadores da fabricante Lenovo. Na realidade este produto reúne um conjunto de diagnósticos que cobre quase todos os componentes da máquina. O segundo foi o Memtest86+ (MEMTEST86, 2011), umas das ferramentas de diagnóstico de memória mais abrangentes em termos de cobertura de falhas.

É importante ressaltar que o LTT foi utilizado com o sistema operacional Windows, enquanto o Memtest86+ é uma ferramenta *stand-alone* que executa diretamente de uma mídia externa, como um CD ou *pendrive* sem carregar nenhum SO. Estas características influenciaram bastante nos resultados, pois afetam diretamente a quantidade de memória testada.

O teste com cada ferramenta foi executado cinco vezes para cada módulo de memória. Estas, por sua vez, foram etiquetadas cegamente, não havendo nenhum conhecimento prévio sobre a presença ou ausência de falhas em cada uma delas.

Uma estimativa do tempo de execução de cada algoritmo implementado pelo

MDiag foi elaborada com base na sua complexidade e levando em consideração o *overhead* das operações realizadas entre as escritas e leituras. Um algoritmo pode exigir apenas $10N$ operações de acesso a memória, mas para ser realizado ainda é necessário alocar memória, executar checagens após as leituras, incrementar o contador de endereço, desalocar memória, etc. Assim, o total de operações de uma implementação deve ser maior que o simples valor da complexidade.

A estimativa utilizou a fórmula 3.1, na qual C é a complexidade do algortimo e O é um parâmetro de tempo médio por operação, levando-se em conta alguns processamentos extras necessários. O valor de O foi medido para cada algoritmo, individualmente, dividindo-se o tempo gasto para executar um elemento de teste pela quantidade de operações naquele elemento.

$$T_{est} = C \cdot O \tag{3.1}$$

3.5 Resumo do Capítulo

Neste capítulo foram descritos os principais passos e medidas tomadas no desenvolvimento da aplicação. Também foi abordada a metodologia de validação elaborada para medir a eficiência dos algoritmos escolhidos.

No capítulo seguinte, são apresentados os resultados dos testes de inserção de falhas utilizados para validar a ferramenta, além de medir o desempenho desta em relação a ferramentas do mercado.

Capítulo 4

Resultados

Neste capítulo serão abordados os resultados adquiridos com os testes realizados a partir da metodologia descrita na seção 3.3 e testes comparativos do MDiag com ferramentas existentes em memórias reais.

4.1 Testes com inserção de falhas

O mecanismo de inserção de falhas apresentado simulou um total de 968 combinações diferentes de falhas dos tipos *idempotent* CF, *inversion* CF e *3-coupling fault*. O tempo total de simulação foi de 88 minutos. A tabela 4.1 mostra a quantidade de falhas detectadas para cada algoritmo.

Tabela 4.1: Falhas detectadas por algoritmo.

Algoritmo	Falhas detectadas	Falhas detectadas (%)
March C-	944	97,52%
Enhanced March C-	956	98,76%
March G	956	98,76%
Papachritou Parcial	964	99,58%
Papachristou Completo	964	99,58%
MT	964	99,58%

O resultado condiz com o esperado, porque a quantidade de falhas detectadas cresce de acordo com a evolução dos algoritmos. Algumas diferenças de cobertura entre os testes não foram percebidas, como entre o Papachristou parcial e completo. Isto é explicado pela pequena variedade de modelos simulados. É de se esperar que,

com a ampliação dessa diversidade, os resultados revelem maior contraste entre os algoritmos.

4.2 Testes com Memórias Reais

Os testes comparativos entre o MDiag e ferramentas existentes foram realizados em um conjunto de 10 placas de memória, algumas possuindo falhas, outras não. As de número 01 a 05 são de encapsulamento DIMM e as de 06 a 10, encapsulamento SO-DIMM. Todas as ferramentas foram executadas nas configurações padrão, isto significa que todas executaram o teste mais completo, com todos os algortimos implementados por cada uma.

Os resultados estão sintetizados na tabela 4.2. Cada teste tem três possibilidades de resultado: nenhuma falha encontrada (✓), uma ou mais falhas encontradas (F) ou não foi possível executar o teste (∅). Esta última significa que a memória não passou no *Power On Self Test* (POST), uma sequência de testes realizada pelo *Basic Input/Output System* (BIOS) que verifica preliminarmente se o sistema se encontra em estado operacional. Outra possibilidade é que o SO não tenha conseguido executar por tempo suficiente para realizar o teste, provavelmente devido ao uso de parte danificada.

Tabela 4.2: Resultados dos testes em memórias reais.

Memória	MDiag	LTT	Memtest86+
Memória 01	✓	✓	✓
Memória 02	∅	∅	∅
Memória 03	✓	✓	✓
Memória 04	✓	✓	✓
Memória 05	∅	∅	∅
Memória 06	F	∅	F
Memória 07	✓	✓	✓
Memória 08	∅	∅	F
Memória 09	✓	✓	✓
Memória 10	F	F+∅	F

A comparação mostra coerência de resultados entre as ferramentas. As memórias 02 e 05 não passaram no POST, então não puderam ser testadas em nenhuma ferramenta. O Memtest86+, com a vantagem de não utilizar SO, conseguiu testar

todas as outras, detectando três placas com falha. Destas, o MDiag não pôde ser executado na de número 08, pois o Linux não conseguiu concluir sua inicialização, mas as outras também foram diagnosticadas com falha. Nos testes com o LTT, o Windows entrou em falha crítica antes de executar a ferramenta. Apenas em uma das iterações com a memória 10 foi possível finalizar o teste, obtendo o mesmo resultado que o Memtest86+ e o MDiag.

O MDiag mostrou-se eficaz na detecção de falhas reais, com resultados compatíveis com o Memtest86+. Por executar sobre Linux, apresentou, ainda, vantagem em relação ao LTT, conseguindo testar memórias com falhas com maior estabilidade.

4.3 Tempo de Execução

Além da cobertura de falhas, é importante considerar o tempo de execução de cada ferramenta. O LTT executa 11 algoritmos em sequência, não possuindo opção para selecionar apenas alguns deles. O Memtest86+ possui 12 tipos de testes e permite selecionar quais serão executados. O MDiag possui 6 algoritmos, também permitindo a seleção de quais serão executados. Assim, o LTT demanda um tempo fixo de execução para uma dada máquina, enquanto o Memtest86+ e o MDiag podem ser configurados para testes rápidos, médios ou longos de acordo com os algortimos selecionados.

Foi feita uma estimativa do tempo de execução de cada algoritmo implementado pelo MDiag, tomando como base na sua complexidade. O valor foi confrontado com o valor real medido. A estimativa levou em consideração parte do *overhead* introduzido pelo processamento extra necessário entre cada operação de escrita/leitura.

A tabela 4.3 traz as medidas e as estimativas do tempo de execução, além da expressão da sua complexidade. Os dados da tabela são mostrados no gráfico da figura 4.1, com exceção do Papachristou Completo que difere dos demais por duas ordens de grandeza.

As estimativas mostram de maneira mais realista o esforço computacional das implementações dos algoritmos, pois entre escritas e leituras na memória há uma série de instruções executadas e estas interferem significativamente no tempo de execução total. Por exemplo, a se basear apenas no acesso à memória, o algoritmo

Tabela 4.3: Tempo de execução.

Algoritmo	Complexidade	Tempo estimado	Tempo medido
March C-	$10N$	56 s	61 s
Enhanced March C-	$18N$	76 s	84 s
March G	$23N$	95 s	121 s
Papachritou Parcial	$38N$	159 s	181 s
Papachristou Completo	$36N + 24N \log_2(N)$	18934 s	18032 s
MT	$36N$	280 s	221 s

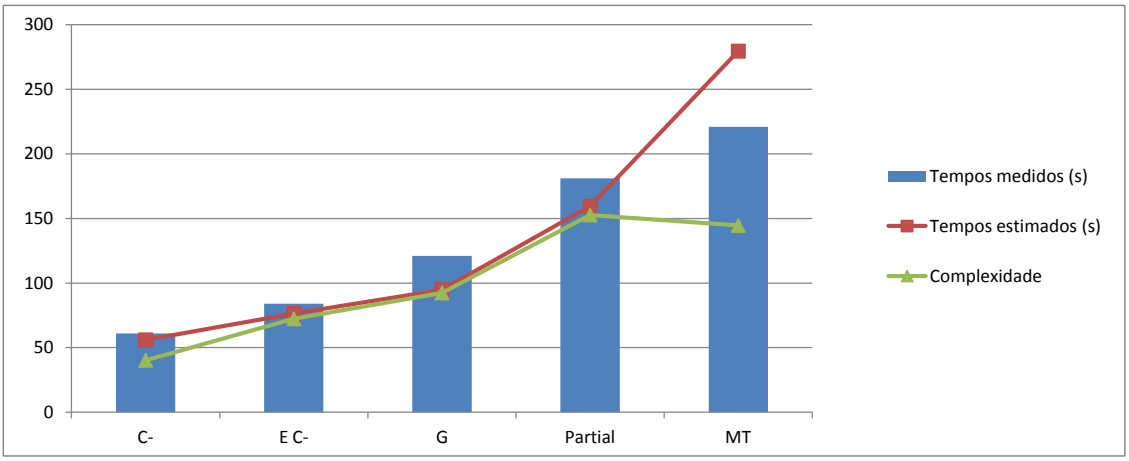

Figura 4.1: Tempo médio de execução *versus* complexidade.

MT deveria ser mais rápido que o Papachristou Parcial. No entanto, suas operações envolvem a manipulação de padrões de fundo complexos, que variam de acordo com o endereço, enquanto o outro apenas escreve e lê o mesmo valor e o seu inverso, repetidamente, tornando o MT mais oneroso em termos de processamento.

4.4 Portabilidade

O MDiag foi compilado e testado em diversas plataformas. Entre elas estão um sistema embarcado com processador ARM, três servidores, dois *notebooks* e dois *desktops*. Estes testes não utilizaram memórias com erros nem inserção de falhas, foram realizados apenas como prova de conceito de que a ferramenta é capaz de atuar em ambientes computacionais variados.

Além de diferentes distribuições Linux (Red Hat, Ubuntu, Suse e CentOS) e sistemas operacionais de 32 e 64 *bits*, o MDiag também foi testado em uma plataforma de desenvolvimento de sistemas embarcados com processador ARM e

em servidores de médio e grande porte com até 16 GB de memória.

Foi desenvolvida uma versão inicializável do Linux, possibilitando diagnosticar memórias de computadores sem SO ou sem Linux instalado. Foi utilizado um *kernel* minimalista, otimizado para executar na maioria dos computadores de arquitetura PC x86 e com o uso mínimo de memória de sistema, permitindo que a quantidade alocada para diagnóstico seja maximizada.

A ferramenta se comportou de maneira estável e nenhum problema de compatibilidade foi detectado durante a execução em todos os ambientes.

4.5 Resumo do Capítulo

Este capítulo apresentou resultados de diversos testes realizados com o MDiag. Seu desempenho e eficácia foram comprovados e comparados com outras ferramentas de diagnóstico.

A seguir serão tiradas as conclusões acerca deste trabalho com base nos resultados obtidos, além de propostas de continuação do que foi desenvolvido.

Capítulo 5

Conclusões

Neste trabalho foi desenvolvida uma ferramenta de diagnóstico de falhas em memórias, nomeada de MDiag. Esta opera sobre o SO Linux, ambiente até então carente de aplicações semelhantes com a qualidade proposta.

Os algoritmos implementados foram selecionados após um extenso levantamento dos testes apresentados em diversas publicações e livros da área. Foram levadas em consideração a cobertura de falhas e a complexidade de cada um, resultando em cinco algoritmos de ordem $O(N)$ e um de ordem $O(N \log(N))$.

Para garantir bons resultados, foi realizado um estudo aprofundado sobre as características de gerenciamento de memória do Linux. Com a familiaridade adquirida, foi possível expandir a quantidade de memória coberta pelo MDiag sem comprometer a estabilidade do sistema.

Para medir a quantidade de falhas detectadas pelos algoritmos, foi projetado um sistema de inserção de falhas que utiliza instruções de *debug* do processador. O sistema simulou uma grande quantidade de falhas de três modelos diferentes. Os resultados coletados confirmaram a excelência dos algoritmos implementados, todos obtendo cobertura acima de 97% das falhas inseridas. O próprio sistema de inserção de falhas é uma contribuição de grande utilidade para análises quantitativas de cobertura de falhas de testes de memória.

O MDiag também foi testado com memórias defeituosas reais, juntamente com outras ferramentas de diagnóstico utilizadas no mercado. Os resultados comparativos mostraram que o MDiag detectou todas as falhas acusadas pelos outros *softwares*.

Os testes de portabilidade realizados mostraram que o MDiag se adapta bem a várias plataformas de *hardware* com Linux. Assim, pode ser utilizado para diagnosticar desde memórias de pequenos sistemas embarcados, com *port* personalizado do *kernel*, até servidores com numerosos módulos e diferentes distribuições Linux.

A realização deste trabalho gerou como contribuição não apenas o diagnóstico em si, que mostrou-se eficiente e portável, mas também um sistema de inserção de falhas totalmente automatizado que pode ser utilizado para avaliação quantitativa de outras implementações de testes de memória. A ferramenta é uma contribuição especialmente interessante para ser utilizada como parte de um sistema maior de diagnóstico de computadores, que pode reunir ferramentas semelhantes aplicadas aos outros componentes de *hardware* da máquina, como processador, disco rígido, placa mãe, etc. O MDiag é particularmente fácil de ser integrado com outras ferramentas por ser facilmente convertido em uma biblioteca que pode ser utilizada como uma Interface de Programação de Aplicativos (*Application Programming Interface*) (API) para testes de memória.

5.1 Perspectivas Futuras

O produto deste trabalho pode ser continuado de diversas maneiras. Como sugestão, estão o aprimoramento do *software* em si, que pode ser feito principalmente através da implementação de novos algoritmos a medida que forem encontrados testes de eficiência superior. Também é desejável o desenvolvimento de uma interface com usuário mais elaborada, que torne a experiência de utilização mais agradável.

Outros tipos de falhas podem ser modelados no sistema de inserção concebido, melhorando as estatísticas de cobertura de falhas dos algortimos.

A ferramenta foi planejada para que seja utilizada como parte de um sistema de diagnóstico para todo o conjunto de *hardware* do computador. Diagnósticos para outros tipos de dispositivos podem ser reunidos em uma única aplicação, capaz de testar não apenas a memória, mas todo o sistema.

Referências Bibliográficas

ADAMS, R. D. *High Performance Memory Testing*. London: Kluwer Academic, 2003. ISBN 1-4020-7255-4.

DAVID, R. *et al.* Random pattern testing versus deterministic testing of rams. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 38, p. 637–650, Maio 1989. ISSN 0018-9340.

DEAN, C. A.; ZORIAN, Y. Do you practice safe tests? what we found out about your habits. In: *Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years*. Washington, DC, USA: IEEE Computer Society, 1994. p. 887–892. ISBN 0-7803-2103-0.

GLASER JOHN F. COULEUR, G. A. O. E. L. *System Design of a Computer for Time Sharing Applications*. 1965. Disponível em: <http://www.multicians.org/fjcc2.html>.

GOOR, A. J. V. D. *Testing Semiconductor Memories: Theory and Practice*. New York, NY, USA: John Wiley & Sons, Inc., 1998. ISBN 0-471-92586-1.

KNUTH, D. *The art of computer programming*. Reading, Mass: Addison-Wesley, 1997. ISBN 0201896834.

LENOVO. *Lenovo Support*. [S.l.], 2011. Disponível em: <http://web.lenovothinkvantagetoolbox.com/index.html>.

LINUXMM. *OOM Killer - linux-mm.org Wiki*. [S.l.], Junho 2009. Disponível em: <http://linux-mm.org/OOM_Killer>.

LINUXMM. *LinuxMM - linux-mm.org Wiki*. [S.l.], Junho 2010. Disponível em: <http://linux-mm.org/>.

MEMTEST86. *Memtest86+ - Advanced Memory Diagnostic Tool*. [S.l.], Janeiro 2011. Disponível em: <http://www.memtest.org/>.

MICRON TECHNOLOGY, INC. *MT16JSF25664H Datasheet*. Boise, USA, 2008. Disponível em: <http://download.micron.com/pdf/datasheets/modules/ddr3/jsf16c256x64h.pdf>.

NAIR, R. *et al.* Efficient algorithms for testing semiconductor random-access memories. *Computers, IEEE Transactions on*, C-27, n. 6, p. 572 –576, june 1978. ISSN 0018-9340.

PAPACHRISTOU, C.; SAHGAL, N. An improved method for detecting functional faults in semiconductor random access memories. *Computers, IEEE Transactions on*, C-34, n. 2, p. 110 –116, feb. 1985. ISSN 0018-9340.

PC-DOCTOR. *PC-Doctor*. [S.l.], 2011. Disponível em: <http://www.pc-doctor.com/>.

PETRU, O. A. C. March test algorithm for 3-coupling faults in random access memories. In: *Preceedings of 2002 WSEAS International Conference Information, Simulation and Manufacturing Systems*. Cancun, México: World Scientific and Engineering Academy and Society, 2002. p. 188–193. ISBN 960-8052-59-9.

PHAN, S. C. C. T. *Programmable moving inversion sequencer for memory bist address generation*. julho 2002. Patent. US 6425103.

RAGHURAMAN, A. Walking, marching and galloping patterns for memory tests. 2005.

RIEDEL, M.; RAJSKI, J. Fault coverage analysis of ram test algorithms. In: *VLSI Test Symposium, 1995. Proceedings., 13th IEEE*. [S.l.: s.n.], 1995. p. 227 –234.

SUK, D. S.; REDDY, S. M. A march test for functional faults in semiconductor random access memories. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 30, p. 982–985, Dezembro 1981. ISSN 0018-9340.

TANENBAUM, A. *Modern operating systems*. Upper Saddle River, N.J: Prentice Hall, 2001. ISBN 0130313580.

THATTE, S. M. Fault diagnosis of semiconductors random accsess memories. Coord. Sci. Lab., Illinois, Maio 1977.

WEBER, T. S. *Tolerância a falhas: conceitos e exemplos*. Universidade Federal do Rio Grande do Sul, 2001.