**Jorge Mamoru Kobayashi**

# Entropy: Algoritmo de Substituição de Linhas de Cache inspirado na Entropia da Informação

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia Elétrica.

São Paulo
2010

**Jorge Mamoru Kobayashi**

# Entropy: Algoritmo de Substituição de Linhas de Cache inspirado na Entropia da Informação

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia Elétrica.

Área de concentração:
Sistemas Digitais

Orientador:
Prof. Dr. Mario Donato Marino

São Paulo
2010

Aos meus filhos, Pedro e Marina, por compreenderem minha ausência antes mesmo de saberem o significado desta palavra.

À minha esposa Márcia, por docemente suportar os piores momentos desta jornada.

Aos meus pais, Jorge e Nilza, por me ensinarem que o conhecimento é algo eterno e só ele constrói a liberdade.

Às minhas irmãs, Mayko e Marisa, por cuidarem a todo tempo do irmão menor.

# Agradecimentos

Agradeço ao Prof. Dr. Mário Donato Marino pelo voto de confiança concedido a mim quando aceitou orientar este trabalho. Pela intensa dedicação e entusiasmo com esta pesquisa. Pelas palavras de motivação e pelo rigor quando foi necessário. Seu senso de dever a ser cumprido é um exemplo que irá muito além dos limites deste trabalho.

Agradeço ao Prof. Dr. Jorge Kinoshita e ao Prof. Dr. Siang Wun Song pelos cuidadosos comentários e correções durante o processos de qualificação.

Ao amigo Evaldo Horn de Oliveira, pelo incentivo e apoio para que eu iniciasse e levasse adiante este trabalho quando as regras diziam que não seria possível.

Aos colegas do laboratório LAHPC - *Laboratory of Architecture and High Performance Computing*, em especial Darlon Vasata, Charles Rodamilans, Filipe Scoton e Arthur Baruchi pela ajuda em diversas ocasiões.

Ao Prof. Dr. Henrique Schützer Del Nero (*in memoriam*) que lançou a centelha e, sem perceber, plantou uma ideia em uma mente.

# Resumo

Este trabalho apresenta um estudo sobre o problema de substituição de linhas de *cache* em microprocessadores. Inspirado no conceito de Entropia da Informação proposto em 1948 por *Claude E. Shannon*, este trabalho propõe uma nova heurística de substituição de linhas de *cache*. Seu objetivo é capturar e explorar melhor a localidade de referência dos programas e diminuir a taxa de *miss rate* durante a execução dos programas.

O algoritmo proposto, *Entropy*, utiliza a heurística de entropia da informação para estimar as chances de uma linha ou bloco de *cache* ser referenciado após ter sido carregado na *cache*. Uma nova função de decaimento de entropia foi introduzida no algoritmo, otimizando seu funcionamento. Dentre os resultados obtidos, o *Entropy* conseguiu reduzir em até 50,41% o *miss rate* em relação ao algoritmo *LRU*.

O trabalho propõe, ainda, uma implementação em *hardware* com complexidade e custo computacional comparáveis aos do algoritmo *LRU*. Para uma memória *cache* de segundo nível com 2-Mbytes e *8-way associative*, a área adicional requerida é da ordem de 0,61% de *bits* adicionais.

O algoritmo proposto foi simulado no *SimpleScalar* e comparado com o algoritmo *LRU* utilizando-se os *benchmarks SPEC CPU2000*.

Palavras-chave: Localidade, Processador, Linha de *cache*, *LRU*, Entropia da Informação, SimpleScalar

# Abstract

This work presents a study about cache line replacement problem for microprocessors. Inspired in the Information Entropy concept stated by Claude E. Shannon in 1948, this work proposes a novel heuristic to replace cache lines in microprocessors. The major goal is to capture the referential locality of programs and to reduce the miss rate for cache access during programs execution.

The proposed algorithm, Entropy, employs that new entropy heuristic to estimate the chances of a cache line to be referenced after it has been loaded into cache. A novel decay function has been introduced to optimize its operation. Results show that *Entropy* could reduce miss rate up to 50.41% in comparison to LRU.

This work also proposes a hardware implementation which keeps computation and complexity costs comparable to the most employed algorithm, *LRU*. To a 2-Mbytes and 8-way associative cache memory, the required storage area is 0.61% of the cache size.

The Entropy algorithm was simulated using SimpleScalar ISA simulator and compared to *LRU* using SPEC CPU2000 benchmark programs.

Keywords: Locality, Processor, Cache line, LRU, Information Entropy, SimpleScalar

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Algoritmos

# Lista de Abreviaturas e Siglas

**ANSI** - American National Standards Institute

**CLI** - Command Line Interface

**CMP** - Chip Multi Processor

**CPI** - Cycles per Instruction

**DRAM** - Dynamic Random Access Memory

**DL1** - Data Level 1

**DL2** - Data Level 2

**FIFO** - First In First Out

**HTTP** - Hyper Text Transfer Protocol

**ILP** - Instruction Level Parallelism

**IL1** - Instruction Level 1

**IL2** - Instruction Level 2

**I/O** - Input and Output

**IPC** - Instructions per Cycle

**ISA** - Instruction Set Architecture

**KB** - Kilo Bytes

**LOB** - Large Object

**LFSR** - Linear Feedback Shift Register

**L1** - Level 1 Cache

**L2** - Level 2 Cache

**PISA** - Portable Instruction Set Architecture

**MKPI** - Misses per Kilo Instructions

**MRU** - Most Recently Used

**MT** - Multi-threading

**NUCA** - Non-Uniform Cache Architecture

**OOO** - Out of Order

**POE** - Plan of Execution

**LRU** - Least Recently Used

**SMT** - Simultaneous Multi Threading

**SPEC** - Standard Performance Evaluation Corporation

**SRAM** - Static Random Access Memory

**TLB** - Translation Lookaside Buffer

**TLP** - Thread Level Parallelism

**UL2** - Unified Level 2

**VLIW** - Very Long Instruction Word

**WBHT** - Write Back History Table

# Glossário

access time - Intervalo de tempo entre uma instrução de load/store disparada por um pipeline de processador e a localização da posição na memória cache onde o dado ser gravado ou lido da memória, page 19

address space - Intervalo que contém todos os possíveis endereços disponíveis para armazenar dados em memória, page 9

benchmark - Conjunto de programas utilizados para comparação de desempenho de processadores ou arquiteturas de processadores, page 9

block size - Grandeza que define a quantidade de dados armazenada por segmento da cache. Quanto maior o block size, maior a quantidade de linhas de dados são armazenadas e recuperadas por operação de leitura e gravação na memória principal e na memória principal do sistema, page 14

branch prediction - Predição de condição de desvio que permite ao processador antecipar dados necessários ao processamento das instruções apontadas pelo desvio, page 4

cache - Nível de memória mais rápido dentro da hierarquia de memória. Usualmente implementada dentro circuito de processamento, page 2

cache bank - A memória cache é fisicamente implementada em multiplos bancos (banks). Cada banco pode apresentar latência (tempo) de acesso e serviço maior ou menor dependendo de sua localização e distância em relação às unidades lógicas e aritméticas do processador, page 42

cache block - Agrupamento de múltiplas conjuntos ou linhas de dados e armazenados dentro da memória cache, page 2

cache controller - Circuito do processador responsável por gerenciar o uso da memória cache, page 3

cache hit - Evento no qual um dados solicitado é encontrado armazenado na memória cache, page 2

cache line - Sinônimo de cache block, page 2

cache line pinning - Evento que ocorre quando uma linha ou bloco de dados fica indefinidamente confinada na cache, page 54

cache miss - Evento que ocorre quando um dado solicitado não é encontrador na memória cache, page 3

cache set - Uma memória cache é segmentada em múltiplos conjuntos (sets) onde cada conjunto armazena um número fixo de linhas ou blocos de dados, page 31

cache slices - Arquitetura de processador que organiza a memória cache em múltiplas áreas com tempo de acesso distintos, page 4

capacity miss - Evento que ocorre quando não há endereços livres dentro de um cache ou cache set e uma nova referência a um endereço não carregada no cache provoca um evento de miss no cache, page 68

clock cycle - Unidade de tempo utilizada para marcar ou sincronizar o processamento de um sistema de computador. Pode-se medir esse tempo através da razão entre unidade de segundo por frequência de oscilação dos circuitos do processador, page 2

conflict miss - Evento que ocorre quando um endereço referenciado é mapeado para a mesma posição no cache ou cache set provocando conflito e consequentemente um evento de miss no cache, page 69

copy back - Técnica empregada pelo cache controller para copiar uma linha ejectada de um nível mais alto da memória cache para um nível mais baixo e imediato à que originou a ejeção da linha de dados, page 13

core - Núcleo completo de processamento, contendo unidades lógicas e aritméticas, registradores, memória e demais estruturas necessárias ao processamento completo de instruções e dados, page 5

cpu bound - Característica de uma thread cujo tempo de execução é predominantemente no processador, page 20

cycle time - Intervalo de tempo entre o último load/store de um dado no registrador de um pipeline e a atualização do endereço de memória cache até que esteja disponível para a próxima instrução, page 19

die - Pastilha contendo todos os circuitos que compõe um processador, page 6

fast-forwarding - Técnica de descarte de instruções iniciais de um programa em simulação, page 72

in-order - Técnica de execução de instruções onde, após a decodificação das instruções pelo pipeline, as mesmas são executadas na sequência exata da fase de recuperação (fetch) e decodificação (decode), page 63

io bound - Característica de uma thread cujo tempo de execução é predominantemente em operações de leitura e escrita em dispositivo externo, tais como discos magnéticos, page 20

load - Operação na qual o cache controller carrega no pipeline um dados a ser utilizado por instruções, page 7

lookup table - Estruta de memória com acesso rápido utilizado para armazenamento de dados históricos dentro do processador. Também é usualmente empregada para indexar dados, page 43

memory bound - Característica de um thread cujo tempo de execução é predominantemente em operações de leitura e escrita na memória principal, page 20

memory controller - Circuito responsável por gerenciar o uso da memória principal do sistema, page 6

memory tracing - Técnica que permite gerar o rastro de todos os endereços de memória referenciados por um programa durante sua execução. Permite caracterizar o padrão de acesso à memória, page 10

miss rate - Taxa ou frequência com que ocorrem eventos de cache miss, page 9

multi-task - Sistema computacional que permite a execução simultânea de múltiplos programas de um ou mais usuários, page 2

multi-thread - Processador cujo núcleo de processamento permite que duas ou mais threads de execução seja processadas simultaneamente no mesmo pipeline, page 5

multi-user - Sistema computacional que permite múltiplos usuários simultaneamente utilizando os recursos computacionais, page 2

multicore - Processador com mais de um núcleo completo de processamento, contendo unidades lógicas e aritméticas, registradores, memória e demais estruturas necessárias ao processamento completo de instruções e dados, page 3

n-way - Nível de associatividade de uma memória cache. Uma cache é segmentada em múltiplos conjuntos (sets) e cada conjunto N linhas de cache, page 31

non-blocking cache - Arquitetura de memória cache que permite que novas instruções referenciem endereços na cache enquanto se completa o serviço à instruções anteriores criando o efeito de paralelismo no acesso à memória cache, page 4

out-of-order - Técnica empregada para obter paralelismo na execução de instruções de uma thread. O processador decodifica as instruções e identifica trechos de instruções que podem ser executadas em ordem arbitrária sem comprometer a corretude do programa, page 6

pipeline - Encadeamento de estruturas e circuitos lógicos e de processamento de um processador. Contém múltiplos estágios sendo cada um deles responsável por uma etapa do processamento completo de um conjunto de dados e instruções, page 2

single core - Processador que possui apenas um único núcleo completo de processamento, page 5

single thread - Processador cujo núcleo de processamento permite a execução de uma única thread por pipeline, page 5

slave memories - Conceito primordial de memória auxiliar que deu origem à memória cache atualmente implementada em processadores, page 25

speculative execution - Técnica empregada para obter maior desempenho na execução de uma thread. O processador executa antecipadamente trechos de instruções tais como, por exemplo, desvios condicionais. Confirmando-se o desvio condicional, o processamento já estará concluído e o desempenho do programa será maior, page 6

stall - Evento de interrupção no processamento de uma thread que ocorre quando o dado necessário ao processamento não está disponível em registradores dentro do pipeline, page 3

stateless application - Aplicação ou programa cujas threads de processamento são concluídas em poucos ciclos de clock e os dados utilizados e carregados em memória cache não recebem novas referências por parte de outras threads, page 36

store - Operação na qual o cache controller armazena na memória cache um dado que acabou de ser processador pelo pipeline do processador, page 7

structs - Estrutura de dados usualmente empregada em linguagens de programação estruturada e que agrupa variáveis e constantes através de um endereço de referência, page 27

tag array - Estrutura encadeada que armazena os valores de identificados dos blocos de dados a serem armazenados na memória cache. Cada linha ou block de cache é identificado com um tag, page 42

thread - Sub-divisão de um programa agrupando instruções e dados, page 3

time slice - Fatia de tempo destinado a uma thread para execução de uma ou mais instruções, page 3

victim cache - Arquitetura de cache que armazena em área auxiliar da cache os blocos que foram ejetados da cache. Esta técnica visa contornar os efeitos colaterais de substituições equivocadas de linhas de cache, page 4

working set - Conjunto completo de dados necessários ao processamento de um programa de computador. Todos os dados utilizados ou produzidos pelo programa de computador, page 9

workload - Carga de processamento composta por instruções enviada ao processador, page 10

write back - Sinônimo de copy back, page 21

write buffers - Estruturas de memória cache auxiliares que permitem que uma linha ou bloco de dados alterado seja temporariamente armazenado nesta área enquato é completamente copiado para a memória principal, permitindo que novos acessos ocorram à memória cache, page 21

# 1 Introdução

Nas últimas décadas a indústria de semicondutores vem, consistentemente, demonstrando vigor no avanço dos processadores de propósito geral. Inicialmente restritos a grandes instituições de pesquisa privadas e governamentais, os processadores, em especial os microprocessadores, hoje são vastamente encontrados em equipamentos e utilitários empregados no âmbito doméstico, corporativo e científico. O uso dos processadores partiu de um cenário mono programado, onde um único programa era executado por vez, até chegar no estado atual onde múltiplos programas compartilham os recursos de processamento intensificando o desafio de explorar os recursos dos sistemas com eficiência.

Os processadores evoluíram de implementações elementares, capazes de executar apenas algumas operações específicas e de propósito definido, até dispositivos ultrasofisticados capazes de executar diversas instruções por ciclo, concebidos a partir da confluência multi-disciplinar de processos de engenharia. Hoje, são capazes de endereçar a necessidade computacional de uma gama enorme de aplicações partindo de simples navegadores de *web* até aplicativos de computação gráfica, simuladores de realidade virtual e jogos eletrônicos, estes últimos, caracterizados por sua alta demanda computacional.

A demanda computacional também aumentou nas últimas décadas. A cada nova geração de processadores um número cada mais aplicações científicas, corporativas e domésticas foram surgindo, fazendo com que os microprocessadores se alastrassem cada vez mais. A constante taxa de evolução dos microprocessadores favoreceu sua adesão em larga escala. Sistemas baseados em microprocessadores colocaram à disposição dos usuários um poder de processamento cada vez maior e mantendo o custo destes sistemas em patamares inferiores ao custo de sistemas centralizados como Mainframes e computadores vetoriais. Aliado à disponibilidade de compiladores e ferramentas de desenvolvimento, o uso os microprocessadores foi gradualmente se alastrando pelo mercado consumidor.

Para suprir a crescente demanda computacional, as empresas e a comunidade ci-

entífica se empenharam em produzir, a cada nova geração, processadores mais rápidos. Novas funcionalidades foram adicionadas tornando-os capazes de corresponder à demanda de usuários ávidos por mais recursos computacionais. Estas novas funcionalidades e aplicações aceleraram a proliferação dos microprocessadores. Com a adoção cada vez maior dos microprocessadores viabilizou-se a economia de escala. Este ciclo virtuoso de demanda versus evolução tornou os microprocessadores onipresentes em todos os aspectos da sociedade contemporânea.

Com a proliferação dos computadores no cotiano doméstico e corporativo surgiram novas formas de explorar a crescente capacidade computacional dos processadores. Os ambientes computacionais tornaram-se *multi-users* e *multi-tasking*. A partir do momento que diversos programas passaram a usar e compartilhar os mesmos recursos computacionais, novas técnicas de execução das instruções oriundas destes programas precisaram ser desenvolvidas. Além do aumento de desempenho, os desenvolvedores de processadores passaram a lidar com questões relacionadas a concorrência aos recursos internos destes processadores. Paralelizar a execução de dezenas ou centenas de programas tornou-se, particularmente, desafiador para os projetistas de processadores.

Para lidar com o desafio de executar com rapidez os programas e manter o processador constantemente em operação, os projetistas concentraram seus esforços nas técnicas que exploram o paralelismo intrínseco do código dos programas (*Instruction Level Paralellism - ILP*) e no desenvolvimento e uso de memórias *cache* (PATTERSON; HENNESSEY, 2007). As otimizações que exploram *ILP* buscam identificar o maior número de instruções que podem ser executadas ao mesmo tempo e, assim, aproveitam ao máximo *time slice* de processamento.

Em contrapartida, as memórias *cache* exploram a propriedade da localidade de referência dos programas mantendo próximo do *pipeline* do processador a maior quantidade de instruções e dados úteis durante sua execução. Estas instruções e dados são armazenados na *cache* em blocos contíguos e o processador acessa instruções e dados através destes *cache blocks* ou *cache lines*. Manter o máximo de *cache lines* úteis na memória *cache* é um dos principais fatores para garantir o bom desempenho dos programas.

Quando uma instrução que está em execução no processador referencia um endereço de memória e este endereço é encontrado na memória *cache* diz-se que ocorreu um *hit* na *cache*. Quando um *hit* acontece, o dado solicitado é prontamente carregado nos registradores e a instrução em execução pode seguir seu fluxo no *pipeline* do processador. Isso ocorre porque as memórias *cache* são rápidas (1 a 20 *clock cycles*) e

estão próximas das unidades de execução, por isso, conseguem alimentar o processador rapidamente.

Se a instrução referencia um dado e este não é encontrado na memória *cache* diz-se que houve um *miss* na *cache*. O *cache miss* força o processador (*cache controller*) a solicitar que este dado venha da memória, o que provoca uma parada (*stall*) na execução da *thread*, pois o tempo necessário para trazer o dado da memória principal é da ordem de centenas de ciclos de *clock* de processador e a *thread* não pode seguir a execução até que os dados referenciados sejam trazidos da memória. Na ocorrência de um *stall* devido a um *cache miss*, uma grande parcela do *time slice* da *thread* é desperdiçado, prejudicando o desempenho do programa.

Os processadores modernos possuem técnicas que buscam minimizar o desperdício de tempo de processamento devido a um *cache miss*. Neste caso, ocorre um chaveamento de *threads* e a nova *thread* é executada no *pipeline* enquanto o miss relativo a primeira é atendido. Estas técnicas são conhecidas como *Multithreading - MT*. Porém, é necessário que a outra *thread* esteja pronta para ser executada. Ou seja, que os dados a serem referenciados por ela estejam na memória *cache*. Caso contrário, esta nova *thread* também sofrerá um *stall*. As paradas no *pipeline* tem grande impacto no desempenho dos programas. Os eventos de *cache miss* correspondem de 50% a 75% das causas de *threads* que não estão prontas para execução (PATTERSON; HENNESSEY, 2007) e estão inativas no processador. Por isso é fundamental manter o máximo de dados e instruções úteis na memória *cache*.

A dinâmica de funcionamento dos sistemas computacionais modernos leva a uma concorrência previsível no uso da memória *cache* do processador. Esta concorrência força o processador (*cache controller*) a gerenciar o uso desta memória. Com dezenas de milhares de instruções e dados sendo carregados na *cache* nos endereços disponíveis para armazenamento, em algum momento, estes endereços poderão ser totalmente utilizados.

Quando todos os endereços de armazenamento da *cache* estão utilizados, o *cache controller* deve decidir quais linhas devem ser retiradas da *cache* para que novas linhas contendo instruções ou dados necessários ao processamento sejam trazidas para perto do processador. Esta escolha é particularmente desafiadora para o *cache controller*, pois retirar da *cache*, equivocadamente, as linhas que serão utilizadas nos próximos ciclos de execução comprometerá sensivelmente o desempenho do(s) programa(s).

Outro aspecto importante dos sistemas atuais diz respeito a necessidade de garantir

a coerência e a consistência dos dados que estão armazenados na memória *cache*. Em computadores multiprocessados e com o surgimento de processadores *multi-core*, um endereço de memória pode vir a ser lido ou gravado por mais de um processador ou *core* de processador. Acessos aos endereços de memória por múltiplos processadores ou *cores* faz surgir múltiplas cópias locais do mesmo dado dentro da memória *cache* de cada *core* que referenciou o endereço. Quando existem múltiplas cópias de um dado armazenadas em várias *caches* e um dos processadores executa uma instrução que altera (*write*) sua cópia local, as cópias armazenadas nos demais processadores precisam ser invalidadas.

A invalidação das cópias de dados provocará um *cache miss* da próxima vez que algum dos demais processadores referenciar novamente sua cópia local. Se a *cache* do processador for do tipo *inclusivo*, as cópias existentes no primeiro nível (*L1*) deve ser ejetadas deste nível de *cache* e do próximo nível acima do *L1*. Os eventos relacionados a coerência e consistência das memórias *cache* estão diretamente relacionados com inserção e substituição de linhas nas *caches*.

Diversas técnicas são empregadas visando otimizar o uso das memórias *cache*. O objetivo primordial destas técnicas de otimização de *cache* é melhorar a taxa de *hit* na *cache*, com o objetivo de manter o bom desempenho dos programas. Processadores com *branch prediction* recuperam especulativamente da memória principal as linhas de instruções e os dados que poderão ser utilizados por desvios condicionais na sequência de código em execução. Se a predição do desvio se confirmar, as instruções e dados já estarão disponíveis na *cache* e serão prontamente usados aumentando o desempenho dos programas. Caso esta predição não se confirme, haverá uma penalidade no *cache hit*, pois alguma linha teve que ser ejectada da *cache* para trazer a linha especulada. Essa penalidade pode provocar um *cache miss*.

No caso de processadores que implementam a técnica *non-blocking cache* (PATTERSON; HENNESSEY, 2007), o processador continua disparando instruções de *load* para a *cache* mesmo enquanto ocorrem *stalls* em uma ou mais das instruções (PATTERSON; HENNESSEY, 2007) em execução. Essa técnica sobrepõe a execução de instruções com acessos à memória *cache* e evita que o *pipeline* do processador fique parado devido a uma penalidade de acesso à *cache*.

Em processadores que implementam *victim caches* (ZHANG; ASANOVIC, 2005), as linhas ejetadas são retidas em endereços reservados da *cache* ao invés de serem copiadas de volta para a memória principal. Isso melhora o *cache hit* nas situações em que linhas úteis ao processamento foram ejetadas da *cache*. Programas com padrão

não uniforme de acesso à memória podem ejetar linhas úteis ao processamento e esta técnica pode ajudar a compensar este efeito negativo.

Processadores com grande quantidade de *cores* podem implementar técnicas como *multi-sliced level-2 sharing caches* (MARINO, 2006) onde os *cache slices* são compartilhados por 4 (quatro) ou mais *cores*. Esta técnica melhora o *cache hit* e reduz consumo de banda *intrachip*. O compartilhamento de recursos internos dos processadores é interessante, pois simplificam problemas de coerência e consistência de *cache*. Ao compartilhar os *cache slices* entre múltiplos *cores*, o número de cópias locais dos dados será reduzido. Isso implica em um número menor de *broadcasts* de invalidação no barramento de endereços que interliga os *cores*.

Neste particular estudo, serão discutidos critérios que compõem uma política de substituição de linhas de *cache* e como estas políticas devem explorar, ao máximo, a propriedade da localidade dos programas, pois a localidade de referência representa um ponto fundamental no desempenho dos processadores e sistemas computacionais. O bom desempenho de uma política de substituição de linhas de *cache* contribui positivamente para o desempenho dos programas independente da arquitetura do processador ser *single core*, *multi-core*, *single thread* ou *multi-thread*.

Esta dissertação está organizada da seguinte forma. Na próxima secção 1.1 serão apresentadas justificativas para o estudo sobre o problema de substituição de linhas de *cache* e a motivação deste estudo. Na secção 1.2 serão apresentados os objetivos principais deste estudo enquanto que na secção 1.3 será apresentado o método utilizado para o estudo do problema. Os detalhes de funcionamento do simulador utilizado na metodologia de estudo será apresentado no capítulo 6. Na secção 1.4 serão apresentadas as contribuições principais deste trabalho. No capítulo 2 o problema de substituição de linhas de *cache* será discutido em maiores detalhes enquanto que os aspectos do princípio de localidade referencial constam no capítulo 3. O estado da arte em arquitetura de processadores é apresentado no capítulo 4. O funcionamento do algoritmo proposto neste estudo é apresentado no capítulo 5 e os resultados obtidos e as conclusões deste estudo serão apresentados no capítulo 7.

## 1.1 Justificativa

Durante os últimos 40 (quarenta) anos, o avanço dos microprocessadores foi sustentando, predominantemente, pela evolução da engenharia no domínio de materiais. O constante avanço na engenharia permitiu a criação de materiais dotados de proprieda-

des físicas adequadas para a criação de transistores cada vez mais rápidos. A cada nova geração de semicondutores, os transistores apresentaram geometrias cada vez mais reduzidas.

Esse avanço permitiu que as memórias *cache* fossem construídas dentro do mesmo *die* do processador. Desta forma, aumentou-se a quantidade de posições de memória localizadas próximo às unidades lógicas e aritméticas. Ter mais memória e unidades de processamento dentro dos circuitos garantiu o aumento de desempenho (STOJCEV; TOKI; I., 1000) dos processadores pelo melhor uso da localidade dos programas.

As técnicas empregadas para aumentar o desempenho de processadores são bem recebidas pelo mercado quando não alteram a forma como as aplicações dos usuários funcionam e não requerem recompilação. Em geral, o avanço no desempenho dos processadores observado nas últimas décadas preservou a abstração dos usuários em relação ao *hardware*. O funcionamento e a compatibilidade dos programas foram mantidos. Técnicas como *0ut-of-Order execution* (STOJCEV; TOKI; I., 1000), *branch prediction* (WALL, 1991), *speculative execution* (WALL, 1991) dentre outras exploraram o *ILP - Instruction Level Parallelism* dos programas aumentando o desempenho sem que o usuário precisasse explicitamente programar o código de forma a se beneficiar destas técnicas.

Além das técnicas mencionadas acima que exploram o paralelismo de instruções, a organização da memória em diferentes níveis traz benefícios para o desempenho sem que o usuário ou desenvolvedor do *software* se preocupe em que nível da hierarquia de memória os dados e instruções utilizados pelos programas em execução estão armazenados. Essa é tarefa a ser executada pelo *memory controller* e *cache controller*. Nesta gerência, a política de substituição de linhas de *cache* busca maximizar o desempenho dos programas tratando os acessos à memória de forma abstrata para o usuário final. A política de substituição de linhas de memória *cache* é o ponto crucial na técnica que utiliza estrutura de hierarquizada de memória nos processadores. Ela é quem garante que a memória hierarquizada será bem utilizada e que o princípio da localidade será bem explorado.

O avanço dos microprocessadores não foi acompanhado pelo avanço em igual escala da tecnologia *DRAM - Dynamic Random Access Memory* utilizada na implementação de memória principal de computadores (PATTERSON; HENNESSEY, 2007). Essa lacuna entre o desempenho dos processadores e da memória compromete o desempenho dos programas. As memórias *cache* servem para minimizar o problema da latência de acesso à memória principal. Em geral, a latência de acesso à memória

principal e às *caches* é medida em número de ciclos de *clock*. Na tecnologia atual, o acesso à memória principal leva em torno de 200 (duzentos) ciclos de *clock* para ser completada enquanto o acesso à memória *cache* de primeiro nível leva entre 1 (um) e 2 (dois) ciclos de *clock*. No caso de um acesso ao segundo nível de *cache*, tem-se entre 5 (cinco) e 10 (dez) ciclos de *clock*. Portanto, o algoritmo de substituição de linhas de *cache* deve manter o maior número de endereços úteis dentro da memória *cache*. Se as instruções e dados estão mais mais próximos do processador, a relação processador/memória do sistema será melhorada.

Outro exemplo de abordagem de otimização é encontrado na arquitetura *Very Long Instruction Word - VLIW*. Nesta particular implementação, o compilador é responsável por identificar o *ILP* dos programas (PATTERSON; HENNESSEY, 2007). Da mesma forma, a alocação de registradores e a referência à memória no instante mais apropriado é determinado explicitamente via *software*. O *POE - Plan of Execution* é determinado pelo compilador em tempo de geração de código. Isso permite que o processador seja mais simples do ponto de vista de desenho e verificação. No entanto, mesmo estando a cargo do compilador determinar o plano de execução, a política de gerenciamento das linhas armazenadas na *cache* fica a cargo do processador e precisa ser eficiente. No entanto, em (HALLNOR; REINHARDT, 2000) é proposta uma arquitetura de *cache* cujo gerenciamento da *cache* é feito explicitamente por *software*. Esta abordagem será melhor analisada no capítulo 4. O potencial intrínseco das abordagens via *software* reside no emprego de técnicas mais sofisticadas do que as possíveis de implementação via *hardware*. Tais abordagens gerenciadas por *software* tendem a manter os circuitos físicos mais simples.

Virtualmente, os limites de ganho de desempenho através do aumento da frequência de operação e através da exploração do *ILP* (OLUKOTUN; H., 2005) dos programas foram atingidos. Isto incentivou a comunidade científica e fabricantes de processadores a empregar arquiteturas *multi-cores* (*CMP - Chip Multi Processor*) (NAYFEH; HAMMOND; OLUKOTUN, 1996) e de processadores *Multithreading* (*SMT - Simultaneous Multithreadinng*). Tais técnicas visam explorar o *TLP - Thread Level Paralellism* dos programas ao invés de se limitar ao *ILP*. Permite-se um maior número de *threads* ativas e carregadas no *pipeline* do processador. Esta maior concorrência faz com que o gerenciamento dos endereços disponíveis na *cache* seja ainda mais importante, pois um número maior de *loads* e *stores* ocorrem na *cache* provenientes destas múltiplas *threads*.

O nível de concorrência às estruturas internas dos *chips CMP* e *SMP*, tais como

o barramento de comunicação, barramento de transmissão de dados, barramento de interrupções, posições e bancos de memória *cache* entre outros é acentuado. A maior quantidade de *threads* em execução acaba por agravar problemas como latência de acesso à memória *cache* e aumenta a demanda por banda interna para tráfego de dados *intra-chip* e *inter-chip* e consequente re-projeto das estrutura internas dos processadores *CMP* e *SMT*.

Se os *cores* do *chip CMP* ou *SMT* implementam *caches* privados, haverá múltiplas cópias locais nas diversas memórias *caches* que precisarão ser mantidas em coerência e consistência. A taxa de *cache miss* e *cache hit* devem ser focos de uma política que consiga gerenciar os endereços de memória *cache* e que minimiza a ejeção indevida de blocos devido às invalidações de cópias locais de blocos de *cache*. Por outro lado, se os *cores* implementam *cache* compartilhado, a quantidade de múltiplas cópias será reduzida (CHISHTI; POWELL; VIJAYKUMAR, 2005), embora haja uma concorrência maior pelos endereços de armazenamento deste *cache* compartilhado. Desta forma, o algoritmo de substituição de linhas deverá priorizar os blocos de dados mais úteis as diversas *threads* em execução.

Assim, um ponto crítico no desempenho de um processador é o mecanismo de substituição (HILL, 1987) de linhas de memória *cache*. Neste particular aspecto, qualquer otimização que explore a localidade temporal e espacial dos programas, vai melhorar e aumentar o desempenho dos programas, pois as linhas de dados mais utilizadas serão mantidas na *cache*. Reduzir o número de *cache misses* garante que mais instruções já decodificadas e carregadas no *pipeline* sejam completadas a cada ciclo de *clock*. Reduzir o número de *cache misses* decrementa o número de operações realizadas pelo *memory controller* , reduzindo a quantidade de dados/instruções trafegados entre a memória principal e a memória *cache*. Este tráfego de dados entre memória *DRAM* e *cache* é uma das penalidades mais severas que o processador sofre durante a execução das instruções. Assim, existe a oportunidade de explorar novas técnicas e heurísticas de gerenciamento de memória *cache* que explorem a localidade aumentando o *cache hit* dos programas melhorando seu desempenho.

Busca-se, através deste estudo, encontrar uma nova forma de gerenciar o espaço disponível na memória *cache* do processador e reduzir o número de *cache misses*. A importância deste gerenciamento se deve ao fato de que, durante a execução de múltiplos programas no mesmo computador, os endereços de armazenamento da *cache* estarem completamente utilizados pelos diversos blocos de dados solicitados pelas instruções em execução. O padrão de acesso à memória e a forma com que se distribui no *address space* pode variar consideravelmente. Essa variação determina se os endereços disponíveis na memória *cache* serão consumidos rapidamente ou em um intervalo

mais distante na janela de execução do programa. Porém, os endereços livres dentro da *cache* se esgotarem, o *cache controller* inicia a substituição de linhas de *cache* para minimizar o impacto negativo que a falta de endereços disponíveis na *cache* causa no desempenho dos programas.

A programa em execução e seus dados de entrada (*working set*) tem relação direta com o desempenho e eficiência da memória *cache*. Programas que fazem muitas referências a poucos endereços de dados tendem a armazenar todos os blocos necessários à sua execução na memória *cache*. Já no caso de programas que fazem muitas referências a grande quantidade de endereços de dados, há uma tendência a utilizar o espaço disponível na *cache* rapidamente. Independente da característica de acesso do programa, o espaço disponível na *cache* precisa ser gerenciado em algum momento quando em regime de funcionamento. Este gerenciamento implica na escolha de uma linha de dados que pertence a alguma *thread* e posterior substituição por outra de outra linha de outra *thread*. Uma vez que o padrão de acesso de um programa não é conhecido *a priori*, o algoritmo de substituição de linhas deve buscar um critério para escolher quais linhas são mais relevantes e mantê-las na *cache*.

Assim, pela razões discutidas nos parágrafos anteriores, justifica-se a proposição e avaliação de uma nova heurística de substituição de linhas de *cache* com o intuito de melhorar a utilização da memória *cache* e gerar impacto positivo no desempenho dos programas do *benchmark* a ser executado. Utilizando um novo algoritmo de substituição de linhas de *cache* que seja capaz de explorar mais intensamente a localidade espacial e temporal do programas em execução, o desempenho dos programas será melhorado, pois o número de penalidades de acesso (*miss rate*) à memória principal vai ser reduzido.

Em (KOBAYASHI, 2007), como parte dos trabalhos obrigatórios da disciplina *PCS5720 - Sistemas Operacionais*, foi proposto o algoritmo *Entropy* para efetuar a substituição de páginas de memória. Uma lista ligada representava a memória principal e o algoritmo alocava posições desta lista sequencialmente à medida que as referências a endereços eram efetuadas. Assim, o comportamento da memória principal foi simulado. Ao esgotar os endereços livres de memória, o algoritmo substitui as páginas de memória alocadas para dar espaço às novas requisições. Um conjunto de programas de *benchmark* foi escolhido e, durante a execução destes programas, foram

gerados arquivos de *memory tracing*. Estes arquivos continham a sequência de acessos aos endereços de memória principal efetuados pelos programas *CS, Pool, CPP* e outros *workloads* multiprogramados *Multi1, Multi2, Multi3*. As simulações do algoritmo consistiram da aplicação do *Entropy* sobre os arquivos de *trace* de memória e as estatísticas de *page faults* foram coletadas.

Os resultados obtidos em (KOBAYASHI, 2007) mostraram que o novo algoritmo apresentou, em média, desempenho 64,44% melhor do que o *LRU* para os arquivos de *trace* de memória utilizados nas simulações. Esses resultados justificam a investigação deste algoritmo na substituição de linhas de *cache*. A despeito das diferenças conceituais entre substituição de páginas de memória de sistemas operacionais e linhas de *cache* de processador, seu funcionamento é análogo. Uma vez utilizados os endereços disponíveis em cada um destes níveis de memória, o critério que leva à substituição de páginas de memória é o mesmo que leva à substituição de linhas de *cache*. Ou seja, o objetivo é manter os dados mais úteis ao processamento das instruções o mais próximos possível do *pipeline* do processador. Na disciplina de *PCS5702 - Arquitetura de Computadores* foi apresentada uma monografia (KOBAYASHI, 2008) onde o algoritmo *Entropy* foi implementado para realizar substituição de linhas de *cache*. Na ocasião, foram apresentados os resultados preliminares que sustentaram a motivação de estudar o algoritmo proposto na substituição de linhas de *cache*.

## 1.2 Objetivos

Diante dos desafios e limites enfrentados pela comunidade científica e fabricantes de semicondutores, será avaliada uma nova técnica de substituição de linhas de *cache*. Neste estudo não será levado em consideração o fato de o processador ser *single core* ou *multi-core*, pois, as políticas de substituição de linha de *cache* se aplicam a ambos os casos. O propósito primordial é buscar um melhor aproveitamento da localidade de referência dos programas e fazer com que as memórias *cache* guardem mais linhas úteis à execução dos programas. Através deste novo algoritmo de substituição de linhas de *cache* busca-se reduzir o número de *cache misses* durante a execução das instruções dos programas.

O ganho de desempenho através do aumento da frequência de operação do processador e através da exploração do *ILP - Instruction Level Parallelism* dos programas atingiu seu limite lógico (OLUKOTUN; H., 2005). As novas arquiteturas de processadores *CMP - Chip Multi Processor* e *SMT - Simultaneous Multi Thread* são efetivas,

mas, dependem do *TLP - Thread Level Parallelism* dos programas. Busca-se, desta forma, atingir um desempenho maior na execução dos programas através da redução do número de eventos de suspensão (*stall*) decorrentes de *cache miss*. A nova heurística de substituição de linhas de *cache* proposta neste trabalho tem seu funcionamento inspirado na Entropia da Informação de *Claude E. Shannon*. O bom desempenho desta nova política pode ser traduzido em bom desempenho de processadores *single-core* ou *multi-core*. Independente da implementação e uso de alguma destas arquiteturas mencionadas acima, a política de substituição de linhas de *cache* exerce papel fundamental no desempenho do processador reforçando a relevância deste estudo.

Através de simulações utilizando *benchmarks*, a eficiência desta nova heurística será comparada a tradicional heurística *LRU - Least Recently Used*. No caso do *LRU*, o bloco de dados cujo acesso se deu há mais tempo durante a execução acaba sendo o melhor candidato a ser substituído. Outras heurísticas como a *FIFO*, levam em conta o primeiro bloco a entrar na *cache* a ser retirado, caso seja necessário abrir espaço na *cache* para um novo bloco. As diferenças de funcionamento da heurística proposta e o funcionamento das heurísticas comumente utilizadas serão apresentadas no capítulo 5.

Ainda, como objetivo secundário, espera-se constatar que esta nova heurística apresenta um comportamento mais flexível do que as heurísticas *LRU*, *FIFO* e *Random*. Por flexível, entendemos ser a capacidade desta nova heurística em promover a substituição das linhas de *cache* levando em consideração mais do que um único critério fixo de escolha como fazem as heurísticas mencionadas acima.

Busca-se projetar um algoritmo que subsidie esta nova heurística levando em consideração a simplicidade e complexidade de funcionamento. Uma implementação deste algoritmo no *cache controller* do processador requer que o custo de processamento seja mantido em um nível que não comprometa a complexidade do circuito e da memória *cache*. Este trabalho não tem como foco principal a especificação detalhada do funcionamento do circuito físico que implementa o algoritmo proposto. No entanto, são apresentadas as estruturas necessárias a sua implementação em *hardware*, mostrando a viabilidade técnica do algoritmo proposto.

Para avaliar a nova heurística, os programas de *benchmark* que serão utilizados apresentam características variadas no que diz respeito ao acesso a dados. Busca-se representar o maior espectro possível de aplicações encontradas no atual ambiente computacional. Os programas de *benchmark* escolhidos cobrem aplicações cujo acesso à memória é intenso tal como as aplicações de *Data Warehouse*. Incluem programas onde há predomínio de operações aritméticas inteiras tal como os sistemas transaci-

onais *online* e, analogamente, onde há predomínio de operações em ponto flutuante caracterizando os sistemas de computação científica. Para tanto, o *benchmark* escolhido é composto de programas reais que apresentam as características mencionadas acima. Este conjunto de programas será melhor descrito na secção 1.3.

Utilizando-se um simulador de *ISA - Instruction Set Architecture*, as memórias *cache* vão ser modeladas e parametrizadas para utilizar diferentes algoritmos de substituição de linha de *cache*. Deste simulador serão extraídas as estatísticas de acesso à *cache*, a taxa de acerto (*cache hit*) e demais estatísticas que sejam relevantes para caracterizar o desempenho do novo algoritmo proposto.

## 1.3 Metodologia

A metodologia de pesquisa consiste na utilização de simulador de *ISA - Instruction Set Architecture* e de modelos referência de processadores de mercado tais como *IBM Power*, *Fujitsu SPARC64*, *Intel Core 2* e *AMD Opteron*. A partir destes processadores, foram extraídos parâmetros para a modelagem da memória *cache* de segundo nível dentro do simulador de *ISA*. Detalhes como tamanho da *cache* de primeiro e segundo nível, associatividade da *cache*, tamanho de bloco de dados entre outros foram utilizados para que se chegasse aos parâmetros de definição utilizados nas simulações durante este estudo.

Para avaliar o desempenho, funcionalidade e eficiência da nova heurística proposta foi utilizado um simulador de microarquitetura capaz de modelar um processador superescalar com estruturas semelhantes aos processadores citados no parágrafo acima. A partir de simulações utilizando programas de *benchmark*, foi possível comparar o algoritmo *Entropy* com o algoritmo *LRU*, cuja utilização é vastamente encontrada nos processadores disponíveis no mercado atualmente.

O simulador escolhido foi o *SimpleScalar*, desenvolvido na Universidade de *Wisconsin*, *Madison*. Este simulador de microarquitetura permite a configuração de parâmetros de definição da memória *cache* de forma bastante simples. Um conjunto de opções controlam as definições da *cache*, política de substituição de linhas, latência de acesso e demais parâmetros relevantes ao funcionamento de um processador. Pode-se utilizar tais parâmetros através de um arquivo de configuração ou através da *CLI - Command Line Interface* do simulador. Esta facilidade de uso conferida pelo simulador permitiu a criação de *scripts* de simulação que modificam os parâmetros relevantes a este estudo de forma a automatizar a execução dos programas do *benchmark*.

Um particular aspecto que influenciou na escolha do simulador *SimpleScalar* foi

a facilidade com que os módulos que simulam o funcionamento da *cache*, *instruction sets*, *ISA* e demais componentes de um processador são adicionados ou estendidos no *SimpleScalar*. Para acrescentar uma nova funcionalidade ao simulador basta inserir um novo módulo ou alterar o código original do módulo responsável por prover a funcionalidade que se deseja simular. Cada módulo é implementado no *SimpleScalar* de forma isolada. Isso permite que a inclusão ou alteração do simulador seja feita de sem comprometer o funcionamento dos demais módulos. O *SimpleScalar* é desenvolvido em linguagem *C* o que torna a usabilidade do código bastante confortável. Tal modularidade do *SimpleScalar* permitiu a rápida inserção do algoritmo novo no seu código original.

Para implementar a nova heurística do algoritmo *Entropy* e fazer o gerenciamento da *cache*, o algoritmo foi desenvolvido em *C* e inserido no módulo do simulador que gerencia a *cache*. Durante a simulação dos programas, todas as referências à memória, penalidades, latências e demais grandezas foram tratadas pelas sub-rotinas definidas neste módulo responsável pela *cache*. Mantendo-se a compatibilidade de todas as chamadas de sub-rotinas do *SimpleScalar*, o novo algoritmo de substituição de linhas foi inserido sem gerar qualquer incompatibilidade com os demais módulos do simulador.

O simulador *SimpleScalar* sofreu um ajuste no módulo de *cache*. Neste módulo foi inserido o código do novo algoritmo e a inclusão das opções de parametrização que permitiram instruir o simulador a usar a nova política de substituição. Não foram efetuadas alterações nas demais definições da *cache* tais como penalidades, *copy back*, etc. Maiores detalhes sobre a modularidade e funcionamento do simulador serão apresentados no capítulo 6.

Uma vez tendo implantado o algoritmo *Entropy* no módulo de gerenciamento de *cache* do simulador de *ISA*, tem-se disponível para a simulação a nova política proposta. Por se tratar de um módulo isolado que compõe o simulador, todas as demais sub-rotinas do *SimpleScalar* permaneceram inalteradas. Após implantar a nova política foi cumprida uma fase inicial de simulações com amostras aleatórias de programas do *benchmark* apenas com o intuito de assegurar que algoritmo apresentava comportamento correto.

Da mesma forma, as saídas (*outputs*) dos programas de *benchmark* executados foram gravados em arquivos e verificados para garantir que produziram a mesma saída. A comparação dos arquivos de *tracing* contendo os rastros das operações de substituição permitiu, também, constatar que o *Entropy* apresentava o comportamento esperado segundo a heurística que o define.

Utilizando-se o mesmo conjunto de programas de *benchmark* para um mesmo conjunto de dados de entrada (*inputs*), os algoritmos *Entropy* e *LRU* foram simulados no *SimpleScalar*. Além do *Entropy* e *LRU*, o simulador de *ISA* possui implementados os algoritmos *FIFO* e *Random*. Estes dois últimos algoritmos também foram utilizados durante as simulações com o propósito de comparação. Porém, seus resultados não serão contemplados neste estudo, pois o algoritmo adotado como base da comparação foi o *LRU*. Para cada bateria (*batch*) de execução, as estatísticas foram coletadas e agrupadas para subsidiar as comparações. O simulador de *ISA* é responsável por coletar e gerar estas estatísticas durante a execução dos programas de *benchmark*. Através destas estatísticas pode-se comparar indicadores tais como *cache misses*, *cache miss rate*, *IPC - Instructions per Cycle*, *CPI - Cycles per Instruction* entre outras grandezas relacionadas ao desempenho da *cache*.

Para isolar o comportamento e os efeitos sobre a *cache* da nova política de substituição foram assumidos valores equivalentes de latência de acesso para o algoritmo *Entropy* e *LRU*. Da mesma forma, o tamanho da *cache*, do *block size* e das penalidades de acesso à *cache* foram mantidos iguais na simulação com ambos algoritmos. Ou seja, para cada bateria de simulações alterou-se apenas a forma como o *cache controller* escolhe a linha que será ejetada da *cache* quando todos os endereços estiverem sendo usados. Levando-se em conta que o *LRU* foi adotado como *baseline* para os índices de desempenho de um algoritmo de substituição de linha de *cache*, os resultados do algoritmo *Entropy* obtidos nas simulações foram normalizados em relação aos resultados obtidos nas simulações do algoritmo *LRU*. Essa normalização permite uma análise relativa do desempenho do *Entropy* em função do desempenho do *LRU*.

Para geração de dados e estatísticas que permitam a análise foi adotado um processo cíclico envolvendo os seguintes passos:

i Ajuste dos parâmetros que definem o *cache*

ii Selecionar a política de substituição de linhas de *cache*

iii Executar programas *SPEC CPU2000*

iv Compilar resultados para criar arquivo de entrada para gerador de gráficos

v Voltar ao passo 1

Este processo repetitivo foi realizado em quantidade necessária para que se tivesse uma massa de dados suficiente para a análise e conclusão. Os programas do *benchmark SPEC CPU2000* executados nestas etapas serão discutidos mais adiante na secção 7.1.

### 1.3.1 Recursos Empregados

Atualmente existem ferramentas de simulação extremamente sofisticadas que auxiliam no desenho e simulação funcional de processadores nos estágios anteriores a *prototipagem*. Tais ferramentas são denominadas simuladores de *ISA - Instruction Set Architecture* e implementam, em *software*, o conjunto completo de instruções de uma arquitetura de processador.

Em particular, duas categorias de simuladores estão à disposição para uso dos desenvolvedores de processadores. Os simuladores funcionais e os simuladores completos. Nos simuladores funcionais, o código objeto dos programas é executado pelo simulador e as estatísticas são computadas para todas as instruções executadas. As estatísticas são produzidas apenas para as instruções do código do programa executado. Nos simuladores completos, uma imagem de sistema operacional é carregada e executada pelo simulador. Pode-se reproduzir um ambiente computacional completo no simulador completo. A partir deste ambiente operacional completo, o código objeto dos programas é executado e as estatísticas computadas. Estas estatísticas contemplam as instruções dos programas simuladores e, também, as instruções provenientes do sistema operacional. Todo o *overhead* do sistema operacional incide sobre as estatísticas coletadas pelo simulador completo.

Nesta pesquisa será utilizada uma infraestrutura baseada em software de código aberto e que apresentem licença condicionada a uso em projetos acadêmicos. Escolheu-se o simulador *SimpleScalar* , também, devido ao fato de ser um simulador de código aberto. Essa condição permitiu livre acesso ao código fonte e à documentação. Além da disponibilidade do código fonte para alteração, a rapidez nas simulações e, principalmente, a simplicidade em ser estendido com novos módulos e funcionalidades foram critérios fundamentais na escolha do simulador de *ISA SimpleScalar*. Este último aspecto foi decisivo na escolha.

Quanto aos requisitos de *hardware*, foram utilizados equipamentos multiprocessados baseados em arquitetura *X64*, rodando sistema operacional de código aberto *GNU Linux*. Por se tratar de simulações de ambientes isolados, nenhum recurso e/ou infraestrutura de rede para interconexão de computadores se fez necessário.

Para simular o *workload* de processamento foram utilizados programas do *benchmark SPEC CPU2000*. Os programas que compõem o *benchmark SPEC CPU2000* (SPEC, ) apresentam uma combinação variada de operações de aritmética inteira e em ponto flutuante. O intuito do uso deste *benchmark* é fazer com que o processador seja submetido uma carga de processamento intensa e, desta forma, caracterizar os *worklo-*

*ads* tipicamente encontrados na computação doméstica, corporativa e científica. Neste estudo serão utilizados os programas *ammp, applu, apsi, art, bzip, crafty, eon, equake, fma3d, gcc, gzip, lucas, mesa, mgrid, twolf, vpr*. Estes programas e seus respectivos arquivos de entrada (*inputp*) foram disponibilizados em conjunto com o simulador *SimpleScalar*. No capítulo 7 serão apresentados maiores detalhes sobre o *benchmark SPEC CPU2000* e a metodologia detalhada de testes.

## 1.4 Contribuições

Este trabalho introduz uma nova heurística de substituição de linhas de *cache* de microprocessadores. Esta nova heurística baseia-se na Entropia da Informação de *Claude E. Shannon*. O algoritmo que implementa esta nova heurística captura e explora a localidade de referência dos programas de computador utilizando o conceito de entropia da informação e, assim, estima a probabilidade de uma linha de *cache* ser reutilizada durante a execução do(s) programa(s). Durante o processo de revisão de trabalhos relacionados, não se constatou nenhuma proposta ou estudo que aplicasse o conceito da Entropia da Informação para explorar a localidade referencial dos programas no uso da memória *cache*.

Mantendo a complexidade em *hardware* equivalente ao do algoritmo *LRU*, o *Entropy* apresenta um comportamento dinâmico que favorece seu desempenho em relação aos algoritmos convencionais *LRU, FIFO, Random*. Esse dinamismo se traduz na variação de seu funcionamento frente aos programas com diferentes padrões de acesso à memória. Quando executando programas caracterizados por localidade temporal, o comportamento do *Entropy* se assemelha ao *LRU*, retendo no *cache* as linhas mais recentemente utilizadas. Quando no programa predomina a localidade temporal, por exemplo, em longos *file scans*, o comportamento do *Entropy* se assemelha ao algoritmo *MRU - Most Recently Used*. Essa variação de comportamento favorece o objetivo de capturar a localidade de referência dos programas, permitindo que as linhas mais relevantes para o processamento sejam retidas no *cache*. Isso contribui positivamente para o desempenho dos programas. O algoritmo *Entropy* contribui com o incremento de desempenho sem incorrer em aumento de complexidade de desenho e de verificação dos circuitos.

Adotando o algoritmo *Entropy* como algoritmo de substituição de uma memória *cache* de segundo nível de 1024-kbytes e associatividade *8-way* e comparando-o ao *LRU*, foi constatado que o algoritmo proposto reduziu a taxa de *misses* de 8 (oito) dentre os 16 (dezesseis) programas do *benchmark SPEC CPU2000* executados durante as simulações. O melhor caso foi obtido ao se executar o programa *art*, do *suite SPEC*

*2000*, onde o algoritmo *Entropy* chegou a obter a expressiva redução de 50.41% da taxa de *misses* quando comparado ao *LRU*. A redução obtida foi bastante expressiva devido a alta diversidade de comportamento apresentada nos 16 (dezesseis) diferentes *benchmarks*.

Para os demais programas, onde o *Entropy* nao proporcionou uma redução total da taxa de *misses*, como mostrado no capítulo de resultados, pode-se perceber diversas fases onde o *Entropy* apresentou taxas de *misses* menores. Como parte do algoritmo *Entropy*, este trabalho apresenta também como contribuição a implementação inicial de uma função de decaimento adequada aos programas do *SPEC 2000*.

Para ter uma melhor base de comparação entre o *Entropy* e o *LRU*, ao invés das típicas taxas totais que são usualmente apresentadas (QURESHI et al., 2007; KIM; BURGER; KECKLER, 2002; ZHANG; ASANOVIC, 2005; DYBDAHL; STENSTRöM; NATVIG, 2006) este trabalho inova ao apresentar a evolução das taxas de *misses* ao longo da execução dos *benchmarks* do *SPEC CPU2000*.

Este trabalho também propõe uma implementação em *hardware* do *Entropy*, com complexidade comparável ao do *LRU* e que requer apenas 0.61% mais *bits* de capacidade para ser implementada. Maiores detalhes desta implementação em *hardware* serão apresentados na secção 5.2 do capítulo 5. Este circuito está baseado em lógica discreta e elementos de memória.

# 2 O Problema de Substituição de Linhas de *Cache* em Processadores

Os microprocessadores modernos possuem grande capacidade computacional devido à evolução da tecnologia dos semicondutores e às técnicas desenvolvidas para explorar o *ILP* dos programas. No entanto, a tecnologia utilizada para implementar a memória principal dos computadores não sustentou um avanço na mesma taxa (PATTERSON; HENNESSEY, 2007) que a dos processadores. Usualmente, as memórias principais são implementadas utilizando a tecnologia *DRAM - Dynamic Random Access Memory*, cujo *throughput* não evoluiu no mesmo ritmo que o *throughput* interno do processador (BURGER; GOODMAN; KäGI, 1996).

A defasagem entre a velocidade com que os *pipelines* dos processadores executam as instruções versus a velocidade e vazão com que as memórias conseguem alimentar os processadores agravou-se à medida que os processadores evoluíram e tornaram-se cada vez mais rápidos (BURGER; GOODMAN; KäGI, 1996). Em um processador moderno, como o *IBM Power6*, centenas de instruções permanecem ativas em estado de execução dentro do *pipeline* do processador. Para que elas sejam executadas é necessário que os dados que virão a ser referenciados estejam o mais próximo possível do *pipeline* para que sejam rapidamente carregados. Na eventualidade destes dados não estarem disponíveis prontamente na *cache*, eles precisam ser trazidos da memória principal. Até que a *memory controller* carregue a linha na *cache* do processador o *pipeline* sofre uma parada (*stall*). Técnicas como execução *OOO - Out of Order* tendem a minimizar e mascarar os efeitos negativos destes *stalls*, permitindo que outros segmentos de código da *thread* continuem sendo executados. No entanto, o desempenho geral do programa acaba sendo afetado negativamente devido às paradas (*stalls*) que ocorrem devido a um *cache miss*.

Para minimizar os efeitos negativos da penalidade de acesso à memória principal, que diminuem sensivelmente o desempenho dos programas, os computadores implementam a memória de forma hierarquizada. Para fornecer um bom balanço entre

desempenho, densidade por unidade de área e custo de fabricação, a memória hierarquizada é construída utilizando-se de tecnologias diferentes em cada um de seus níveis. Os processadores implementam, ao menos, dois níveis de memória *cache on-chip* além da memória principal. Os primeiros níveis da memória são denominados memórias *caches*. Tais memórias são menores em área, porém, mais rápidas do que a memória principal. A tecnologia utilizada para implementar os primeiros níveis de *cache* é a do tipo *SRAM - Static Random Access Memory*. Esta tecnologia permite um alto *throughput* dados e um baixo consumo de energia necessária para reter os dados na memória *SRAM* (PATTERSON; HENNESSEY, 2007).

As operações de leitura de dados em memória *SRAM* não necessitam de *refresh* da célula de memória. Por isso, o *access time* e o *cycle time* desta memória são muito próximos. Isso permite que este tipo de memória sustente altas taxas de transferência de dados, ouse seja, alto *throughput* (WILTON; JOUPPI, 1993). Dessa forma, as memórias *SRAM* são ideais para serem usadas *on-chip*, ou seja, dentro do processador. O foco no *design* das memórias *SRAM* está no desempenho e baixa latência ao invés de densidade de armazenamento (PATTERSON; HENNESSEY, 2007). Porém, seu custo de produção é de 8(oito) a 16(dezesseis) (PATTERSON; HENNESSEY, 2007) vezes maior que o da tecnologia *DRAM - Dynamic Random Access Memory*, razão pela qual a memória principal dos computadores é implementada utilizando-se a memória do tipo *DRAM*. O *throughput* das memórias *DRAM* é inferior ao das memórias *SRAM*. Isso ocorre, pois, a cada acesso à uma memória *DRAM* é necessário fazer o *refresh* da célula de memória que armazena o dado. Isso aumenta seu *cycle time*, aumenta o consumo de energia e latência, tornando-as mais lentas.

Para compensar essa diferença de latência de acesso entre as memórias *cache* (*SRAM*) e a memória principal (*DRAM*) é necessário que se equilibre o uso da hierarquia de memória. Para tanto, o problema de substituição de linhas de *cache* concentra-se na tarefa de escolher quais linhas deverão ser mantidas nas *caches* rápidas, evitando acessos à memória principal *DRAM* que é mais lenta. Essa escolha deve priorizar as linhas de dados com maior chance de serem reusadas pelas instruções ativas que estão no *pipeline*. Desta forma, a principal função do algoritmo de substituição de linhas de *cache* é explorar a localidade de referência, buscando reter nas memórias *cache* os dados mais relevantes. Ao reter na *cache* as linhas mais úteis ao processamento, ele procura aumentar a taxa de *cache hit* dos programas em execução. Ao se combinar áreas de memória menores e mais rápidas (*SRAM*) com áreas de maior densidade de armazenamento (*DRAM*) procura-se criar um bom balanceamento entre custo e desempenho no sistema de memória.

A eficácia da arquitetura de memória hierarquizada apoia-se na propriedade dos programas de computador conhecida como *Localidade de Referência*. Esta propriedade, que será detalhada no capítulo 3, prevê que, para um programa arbitrário, o acesso aos endereços de memória ocorre de forma não uniforme (SMITH, 1982). Esta falta de uniformidade faz com que o mesmo subconjunto de endereços seja acessado recorrentemente dentro de uma janela tempo (instruções). Da mesma forma, esta propriedade prevê que determinados endereços que serão acessados nas próximas instruções encontram-se contiguamente alinhados aos endereços que acabaram de ser referenciados pelas instruções em execução (SMITH, 1982).

Apesar das memórias *cache* conferirem grande desempenho em comparação aos acessos diretos à memória principal *DRAM*, sua capacidade útil é reduzida. Assim, na memória *cache* existe uma quantidade reduzida de endereços para armazenar os dados a serem processados pelas instruções. Em situações em que o computador processa apenas um programa, eventualmente, pode ocorrer desta pequena área útil do *cache* ser suficiente para armazenar todo o *working set* do programa. No entanto, esta não é a realidade da computação moderna. O ambiente computacional atual é multiprogramado e existem múltiplos programas simultaneamente em execução e cada um deles contendoseu próprio *working set*. Nos sistemas *multi-core* a tendência é existir um número ainda maior de programas em execução simultânea o que reduz as chances de uma memória *cache* reter o *working set* de todos os programas integralmente.

Outro aspecto importante é que os programas em uso no âmbito doméstico, corporativo e científico apresentam uma característica de uso de recursos diversificada entre *cpu bound*, *IO bound* e *memory bound*. Isso reduz a chance do *working set* destes programas caberem integralmente na memória *cache*. Assim, os endereços e a área disponível nas *caches* rapidamente se esgota. Em regime de funcionamento, a *cache* precisa ser gerenciada e passa a ser necessário retirar uma linha que esteja na *cache* para dar lugar a outra linha que tenha sido referenciada pelas instruções em execução.

A operação de substituição de linha de *cache* é uma operação de alto custo. A operação de substituição não é composta somente da escolha de qual bloco de dados será removido da memória *cache*. Toda vez que uma linha é retirada da *cache* é necessário que seja verificada a consistência e a coerência deste bloco em relação aos demais níveis da memória hierarquizada. Se a linha sofreu uma alteração (*store*) no último acesso, ao ser removida da *cache* esses dados deverão ser gravados na memória em uma operação chamada de *write back* ou *copy back*. Se o processador é *multi-core* e existem múltiplas cópias deste bloco em outras *caches* privadas, estas cópias

locais precisam ser invalidadas (ZHANG; ASANOVIC, 2005) através do barramento de controle do sistema.

Durante a execução dos programas, as instruções podem alterar o valor de um dado presente na *cache*. Quando isso ocorre, o *cache controller* utiliza um *bit* especial para indicar que aquela linha foi alterada e que essa alteração ainda não foi refletida na memória principal. O estado desta linha é denominado *dirty*. Pode acontecer desta linha *dirty* ser a escolhida para ser substituída. Ao mesmo tempo, uma linha nova que é trazida da memória (*fetched*) para ser colocada na *cache* não pode ser inserida na mesma posição que a linha *dirty* até que o processo de *copy back* desta linha tenha ocorrido com sucesso. O *cache controller* precisa copiar a linha *dirty* de volta para a memória principal antes de colocar a nova linha no seu lugar. Essa operação prejudica o desempenho da *cache*, pois representa uma alta penalidade. Para contornar essa penalidade, algumas arquiteturas de *cache* empregam áreas chamadas de *write buffers* (PATTERSON; HENNESSEY, 2007). Um exemplo desta implementação são os processadores *ARM10* (ARM, 2000) e *Cortex* (ARM, 2008).

Dado o custo da operação de substituição de linhas e seu impacto no desempenho do processador, torna-se muito interessante explorar a localidade dos programas, de forma a minimizar esta operação e aumentar a taxa de *cache hit*. Ainda, a implementação do algoritmo de substituição é feita, usualmente, em *hardware*. Esse requisito força os projetistas de memória *cache* a buscarem a simplicidade na implementação destas memórias.

O custo de funcionamento de um algoritmo de substituição de linhas de *cache* deve consumir uma parcela ínfima do *cache controller* para não comprometer o desempenho final do processador. Qualquer aumento no tempo necessário para inspecionar se o dado solicitado está disponível na *cache* ou qual bloco deve ser substituído afeta diretamente o desempenho final dos programas em execução. Uma análise de *trade-offs* em termos de quantidade de *bits* e circuitos lógicos necessários usualmente é empregada para manter a complexidade sob controle. Um exemplo disso ocorre com o algoritmo *LRU*. Quando seu custo de implementação em *hardware* é alto em relação conjunto completo do processador, pode-se adotar uma aproximação (*Pseudo-LRU*) cuja implementação é mais simples e barata, reduzindo a necessidade de *bits* extra de armazenamento e de circuitos lógicos (REINEKE et al., 2007).

Além da tarefa desempenhada pelo *cache controller*, outras abordagens são em-

pregadas para auxiliar na redução de substituições de linhas de *cache* (PATTERSON; HENNESSEY, 2007). Os compiladores mais modernos, tais como *GNU gcc 4.x* e *Intel C Compiler*, permitem a geração de códigos binários otimizados através de *profile*. Outras técnicas disponíveis em nível de compilador e que diminuem o número de substituição de linhas de *cache* são:

- *Instruction and Data Rearrangement* As instruções e dados de um programa podem ser reordenados sem afetar a corretude do programa [ (MCFARLING, 1989) citado por (PATTERSON; HENNESSEY, 2007)]. Essa reordenação de instruções e dados reduz o número de *misses*, pois, as instruções e dados arranjados na memória de forma espaça são colocados próximos (localidade espacial) de forma a serem armazenados no mesmo *cache block*. Assim, aumenta-se a localidade espacial das instruções e dados e as chances de um *cache hit* (PATTERSON; HENNESSEY, 2007).

- *Branch Straightening* Durante a geração do código objeto, o compilador pode detectar um desvio *branch* com maior chance de ser executado. Para aumentar o *hit rate*, o compilador pode intercalar o bloco de instruções e o bloco de dados deste *branch* com os blocos da sequencia original, inserindo-os logo após a instrução que causa o *branch*. Isso aumenta a localidade espacial do programa (PATTERSON; HENNESSEY, 2007). A figura 2.1 abaixo ilustra essa otimização.

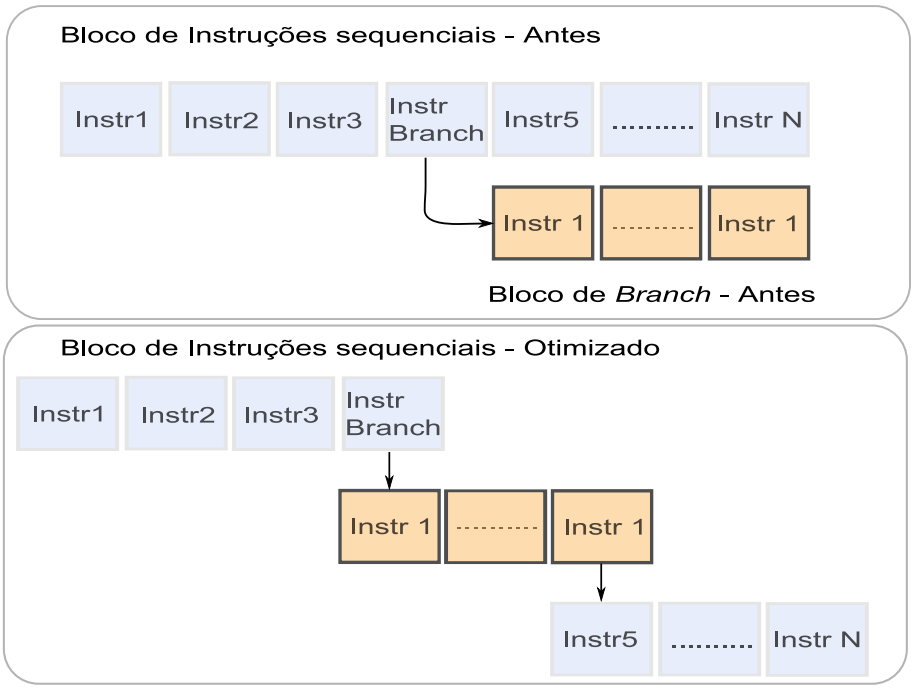


**Figura 2.1:** *Branch Straightening*

- *Loop Interchange* Alguns programas são desenhados com *loops* aninhados (PATTERSON; HENNESSEY, 2007) que acessam os dados de forma não sequencial. Isso prejudica a localidade espacial dos programas e aumenta o *cache miss rate*. O compilador pode identificar os *loops* aninhados e reordená-los de forma a aumentar a localidade espacial dos programas. O exemplo 2.1 ilustra um *loop* aninhado onde a localidade não é explorada de forma otimizada. [1]

**Listagem 2.1:** Loop aninhado original

```
for (j = 0; j < 100; j++)
 for (i = 0; i < 10000; i++)
  x[i][j] = 2 * x[i][j];
```

A reorganização dos *loops* maximiza o uso dos dados na *cache*. Isso evita que as linhas que os armazenam sejam substituídas antes de ser novamente referenciada (PATTERSON; HENNESSEY, 2007). O exemplo 2.2 mostra a otimização que o compilador faz de forma a explorar melhor a localidade temporal e aumentar o *miss hit* na *cache*. [2]

**Listagem 2.2:** Loop aninhado otimizado

```
for (i = 0; i < 10000; i++)
 for (j = 0; j < 100; j++)
  x[i][j] = 2 * x[i][j];
```

- *Blocking* Esta técnica permite aumentar a localidade temporal e reduzir o número de *cache misses* (PATTERSON; HENNESSEY, 2007). Quando o programa manipula vetores e matrizes, o acesso pode ocorrer por linhas em um vetor/matriz e por colunas no outro vetor/matriz. Ao invés de acessar os dados de uma linha ou coluna inteira, o compilador pode organizar em sub-vetores ou sub-matrizes. O exemplo de código 2.3 ilustra a multiplicação de duas matrizes antes da otimização.

**Listagem 2.3:** Multiplicação de Matriz

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++){
  r = 0;
  for (k = 0; k < N; k++) {
   r = r + y[i][k] * z[k][j];
   x[i][j] = r;
```

[1] Fonte: Computer Architecture Quantitative Approach, J. L. Hennesy and David A. Patterson
[2] Fonte: Computer Architecture Quantitative Approach, J. L. Hennesy and David A. Patterson

```
    }
  }
```

O exemplo 2.4 mostra a otimização que o compilador faz de forma a explorar melhor a localidade espacial e temporal e a aumentar o *miss hit* na *cache*. O fator de otimização do *loop* é a constante *B*. A melhora no desempenho é da ordem de *B* (PATTERSON; HENNESSEY, 2007).[3]

**Listagem 2.4:** Multiplicação de Matriz otimizada em blocos

```
for (jj = 0; jj < N; jj = jj+B)
 for (kk = 0; kk < N; kk = kk+B)
  for (i = 0; i < N; i++)
   for (j = jj; j < min(jj+B, N); j++) {
    r = 0;
    for (k = kk; k < min(kk+B, N); k++){
     r = r + y[i][k] * z[k][j];
     x[i][j] = x[i][j] + r;
    }
   }
```

- *Compiler-Controlled Prefetching* Os compiladores podem inserir no código objeto instruções de *load* antecipadas. Ou seja, ao identificar uma instrução que carregará um dado, o compilador insere um *load* deste dado muitas instruções antes da posição real desta instrução no código sequencial original. Essa é uma tentativa de fazer o *prefetching* (PATTERSON; HENNESSEY, 2007) do dado e garantir que ele esteja no *cache* quando a instrução real o solicitar (CHEN et al., 1991).

A substituição de linhas ou blocos de *cache* é um ponto crítico no desempenho dos processadores. Se o *cache controller* e também o compilador conseguirem antecipar e prever a reutilização de uma linha armazenada e priorizar sua retenção na memória *cache*, ciclos de *clock* valiosos para as instruções não serão desperdiçados. O número de *cache misses* vai ser diminuído e o desempenho dos programas será maior.

[3]Fonte: Computer Architecture Quantitative Approach, J. L. Hennesy and David A. Patterson

# 3 Localidade e Mecanismos de Substituição de Linhas de *Cache*

A memória organizada em forma de hierarquia causa um impacto positivo no desempenho dos programas. Ao empregar memórias *cache* mais próximas do processador pode-se explorar a propriedade da *localidade de referência*. Esta propriedade dos programas foi originalmente percebida na década de 60 (sessenta) (DENNING, 2005) durante pesquisas realizadas pela comunidade científica para implementar o gerenciamento de memória virtual. Em 1965, *Wilkes* enunciou o princípio de funcionamento das *slave memories* (WILKES, 1965) que explora a localidade dos programas. Mais tarde, as *slave memories* ficaram conhecidas como *cache memories*. A constatação desta propriedade inspirou o *design* das memórias *cache* nos processadores. Sua eficácia foi tão significativa que esta técnica se estendeu por todo âmbito da computação indo muito além do âmbito da memória dos computadores.

## 3.1 Localidade Temporal

Para os programas de computador, observa-se que o padrão de acesso aos endereços de memória frequentemente se dá de forma irregular (JOHNSON; HWU, 1997). Independente da característica da aplicação, observa-se que o acesso aos endereços, dentro de um intervalo regular de tempo de execução, acaba incidindo sobre os mesmos endereços já referenciados previamente. Ou seja, se um determinado endereço foi solicitado em algum momento durante a execução do programa, existe uma chance maior, em relação a um endereço que ainda não tenha sido referenciado, de que este endereço seja utilizado novamente. Esta particular situação configura a *localidade temporal* (DENNING, 2005; PATTERSON; HENNESSEY, 2007) de acesso à memória. Na localidade temporal, os blocos de endereços já referenciados são novamente solicitados após um número de ciclos de *clock* ou instruções executadas. Essa reincidência de acesso pode se dar em intervalos regulares na execução dos programas. Ainda, todo o espaço de endereçamento acaba sendo dividido em *clusters* e estes *clusters* são acessados recorrentemente em diferentes intervalos de tempo.

Esta divisão do espaço de endereçamento em *clusters* foi enunciada por (DENNING, 2005). Ilustrando cada um dos *clusters* de localidade denotado-os por $L_i$, o acesso recorrente aos endereços dentro deste *cluster* ocorre durante o tempo $T_i$. Assim, durante a execução de um programa a localidade temporal, em relação os endereços alocados para sua execução, é dada pela tupla:

$$< (L_0, T_0), (L_1, T_1), ..., (L_n, T_n) >$$

Idealmente, uma boa política de gerenciamento de linhas reteria na *cache* o conjunto de endereço $L_i$ pelo tempo $T_i$.

Esta localidade temporal é usualmente observada em programas que possuem *loops* de execução. Ou seja, dentro da estrutura do código do programa, um mesmo conjunto de dados e instruções é executado repetidamente sob determinadas condições. Em especial, programas cujo código faz uso extenso de variáveis globais tendem a apresentar localidade temporal, uma vez que tais variáveis tendem a ser referenciadas diversas vezes durante a execução e em estágios distintos do programa. Uma vez alocada a variável global, diversas sub-rotinas podem referenciar esta variável global fazendo com que o programa apresente localidade temporal desta variável global.

Para aproveitar esta propriedade, a memória hierarquizada do computador emprega memórias *cache* para armazenar os dados cuja chance de acesso seja maior, e mantém estes dados mais próximos das unidades de processamento. Assim, os dados mais úteis para o processamento são mantidos na *cache*. Por estarem mais próximos do processador, eles podem ser rapidamente carregados durante a execução evitando que o processador fique parado aguardando que os dados venham da memória principal. Para maximizar este efeito positivo das *caches* sobre o desempenho dos programas, os algoritmos de substituição de linhas de *cache* trabalham para reter estas linhas cuja chance de reutilização seja maior.

No entanto, reter as linhas de dados com maior chance de reuso em *cache* é uma tarefa relativamente simples considerando um *workload* composto de um único programa. Uma vez que a escala de tamanho dos semicondutores permite a construção de memórias *cache* na ordem de *megabytes*, aumenta-se a chance de que todo o conjunto de instruções e dados (*working set*) necessários à execução de um único programa possa ser inteiramente armazenado na *cache*. No entanto, a realidade computacional atual é *multi-threaded* e *multi-tasking*. Considerando que o *workload* de processamento atual é composto de centenas de milhares de *threads* compartilhando as mesmas áreas de *cache* (FEDOROVA et al., 2005), decidir quais linhas de dados devem ser retidas em *cache* e quais devem ser substituídas ganha um grau de complexidade

diferenciado. Este problema de compartilhamento de recursos de memória pode ser agravado quando ocorre migração das *threads* entre os diversos *cores* de um processador *CMP* (CONSTANTINOU et al., 2005). Isso porque surgirá um *overhead* adicional no gerenciamento destas múltiplas cópias do mesmo dados devido à migração de *threads* entre os *cores* de processamento. Por isso, é de extrema relevância ter um bom funcionamento do algoritmo de substituição de linha de *cache*.

## 3.2 Localidade Espacial

Uma segunda característica da localidade refere-se ao comportamento de programas cujo conjunto de dados é armazenado contiguamente. Comumente, os programas desenvolvidos em linguagem estruturada, tais como *C* e *C++*, tendem a agrupar os dados em estruturas como vetores, sequências, *structs* e outras estruturas de dados (SMITH, 1982). Mesmo em programas escritos utilizando-se outro paradigma de programação, como orientação a objetos ou programação paralela, há tendência de agrupar dados através de variáveis declaradas dentro do mesmo objeto ou módulo. Isso acaba por forçar que os endereços de armazenamento destas variáveis sejam alocados de forma contígua no espaço de endereçamento de memória. Quando os endereços de memória utilizados por um programa estão alocados contiguamente, dizemos que o programa apresenta *localidade espacial* (DENNING, 2005; PATTERSON; HENNESSEY, 2007). Existe uma chance maior de que os dados armazenados subsequentemente ao último bloco referenciado sejam referenciados nas próximas instruções.

Decorrente desta forma de alocação de dados dos programas, ocorre que, para um determinado acesso a uma posição de memória, existe uma chance maior, em relação a um endereço que ainda não foi acessado, de que um endereço subsequente ao último endereço acessado seja o próximo a ser solicitado. Esse acesso em sequência caracteriza a localidade espacial. Ou seja, os endereços utilizados pelos programas estão próximos uns dos outros.

Assim, para explorar a localidade espacial as memórias *caches* procuram armazenar endereços de dados contíguos em um mesmo bloco. Quando um endereço em particular for solicitado, ele estará armazenado em um bloco. Junto a este endereço estarão armazenados endereços subsequentes. Ao ser transferido da memória principal para a memória *cache*, os endereços subsequentes serão trazidos para a *cache* em uma única operação porque dentro de uma mesma linha ou bloco de *cache* estão armazenados os dados de múltiplos endereços de memória. No próximo acesso a um endereço por parte do processador, se ocorrer a localidade espacial, tal endereço já se encontrará armazenado na *cache*. Portanto, vai ocorrer um *cache hit* e o desempenho

final do programa será maior.

A decisão de quantos endereços serão agrupados no mesmo bloco ou linha de *cache* é particularmente importante. Na terminologia de Arquitetura de Processadores, esta grandeza é conhecida como *cache block size* ou *cache line size*. Ela determina a quantidade de linhas que serão agrupadas em um mesmo bloco ou linha com vistas à explorar a localidade espacial. A escolha deste parâmetro pode determinar o bom desempenho para programas que apresentam intensa localidade espacial, pois múltiplos endereços estarão prontamente disponíveis na *cache*. Consequentemente, um número menor de substituições de linhas será necessário durante a execução deste programa. A figura 3.1 ilustra uma memória *cache* organizada em blocos.

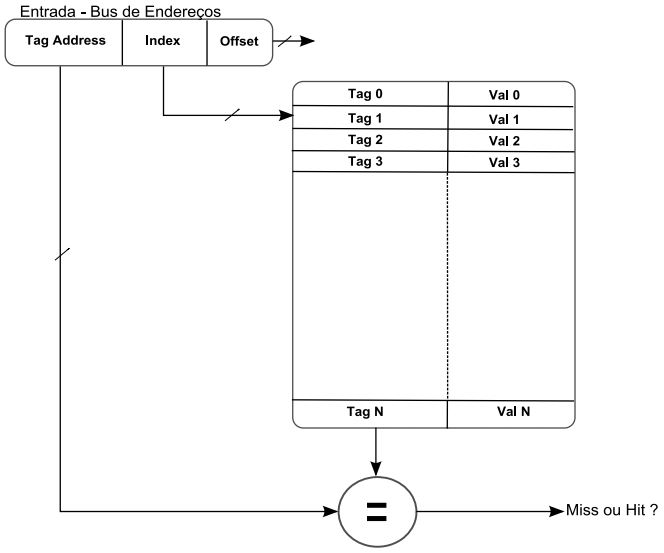


**Figura 3.1:** *Cache* organizado em blocos

Porém, para programas que não apresentam intensa localidade espacial ou em sistemas com *workloads* multiprogramados com padrões de acesso variados, trazer blocos de dados muito grandes pode acabar por agravar a escassez de endereços necessários para atender a todos os *working sets* dos programas em execução. Isso deve-se ao fato de que, trazer blocos maiores implica em um número maior de endereços ser trazido da memória principal para a *cache*. Se o programa não possui forte localidade espacial, muitos destes endereços não serão utilizados, ocupando endereços preciosos na *cache*. Também, como os blocos de dados são maiores, haverá um número menor de blocos na *cache*. Isso poderá aumentar a taxa de *cache miss*, forçando o aumento da frequência com que as operações de substituição de linhas são executadas e, consequentemente, prejudicando o desempenho dos programas.

A escolha do *block size* é uma decisão de projeto importante do ponto de vista de arquitetura. Não basta projetar uma *cache* com um tamanho arbitrário de *cache block size*. O custo de acesso e da transferência de um bloco de dados com muitos

endereços é proporcional ao seu tamanho. Este custo aumenta a penalidade decorrente desta operação (PATTERSON; HENNESSEY, 2007). A escolha do tamanho do bloco ou linha de *cache* representa um importante *trade-off* no *design* do processador, pois *block size* contendo muitos endereços pode provocar até mesmo o aumento de *miss rate* (PATTERSON; HENNESSEY, 2007). Ainda, se a localidade espacial é baixa, um *block size* muito grande pode provocar o desperdício de área na *cache*, pois linhas que não serão referenciadas são mantidas na *cache* (QURESHI; SULEMAN; PATT, 2007).

A seguir descreve-se o funcionamento dos principais algoritmos de substituição de linhas de *cache*.

## 3.3 Algoritmo FIFO

Usualmente, os algoritmos de substituição de linhas de *cache* adotam um critério fixo para escolher qual linha de *cache* será substituída. Dentre as diversas classes de algoritmos, podemos separá-los em duas categorias principais: os algoritmos que se baseiam no uso ou referência e os algoritmos que não se baseiam no uso ou referência de endereços. De um modo geral (SMITH, 1982), pode-se dizer que os algoritmos que levam em consideração o uso ou referência a um determinado endereço apresentam bom desempenho no processamento de programas fortemente caracterizados pela localidade temporal. Já os algoritmos que não se baseiam no uso ou referência adotam outro critério (SMITH, 1982), como por exemplo a ordem de entrada na *cache* para escolher qual linha a ser substituída.

O caso do algoritmo *FIFO - First In First Out* enquadra-se na classe de algoritmos que utilizam um critério não baseado no uso ou referência. Para seu funcionamento, leva-se em consideração a ordem de entrada de uma linha na *cache*. Ou seja, a linha que estiver há mais tempo armazenada na *cache* será a melhor candidata a ser substituída quando houver necessidade de carregar uma nova linha na *cache* e todos os endereços estiverem ocupados. A primeira linha a entrar na *cache* será a primeira a sair da *cache*. Como o critério de substituição leva em conta a ordem de entrada da linha, não há a necessidade de manter, em *hardware*, um inventário de acessos ou referências aos endereços de memória. Isso o torna simples de implementar e sua execução é eficiente (SMITH, 1982).

Uma vez que a linha mais antiga será a próxima a ser substituída, basta manter um contador para cada *cache set* (SMITH, 1982). Esse contador é incrementado a cada substituição de linha. O contador aponta para a próxima linha a ser substituída.

Esse contador é atualizado ciclicamente de forma que as linhas migrem naturalmente da posição de inserção até a posição de substituição. Por migração de linhas de *cache* entenda-se a mudança e atualização do contador que mantém a ordem de entrada e que aponta qual linha será substituída. Este contador de controle pode ser implementado através de um *stream* de registradores para cada *cache set*. A cada substituição, o controlador da *cache* atualiza esta cadeia de registradores fazendo com que o último registrador aponte para qual linha daquele *cache set* será substituída (SMITH, 1982) na próxima vez que esta operação for necessária. A figura 3.2 ilustra este mecanismo de funcionamento da *cache* que implementa política de substituição *FIFO*.

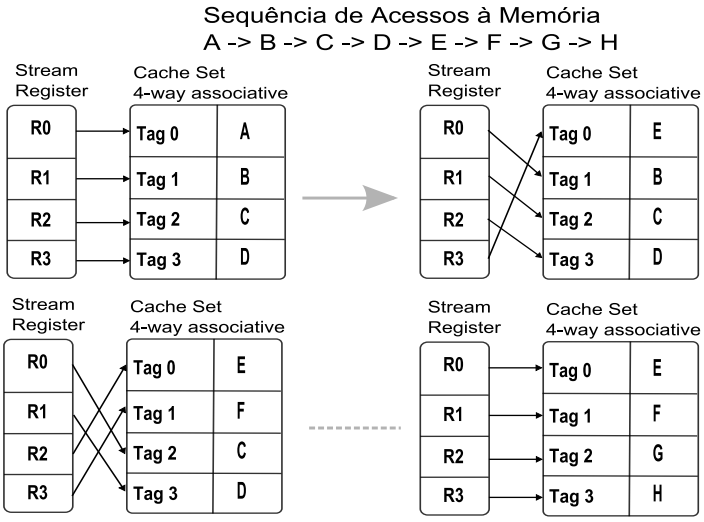


**Figura 3.2:** *Cache Set FIFO Register Ring*

O algoritmo *FIFO* dispensa o uso de um histórico de acesso às linhas armazenadas na *cache*, levando em consideração apenas a ordem de entrada. Em consequência, a quantidade de acesso a uma linha já carregada não influencia positivamente a retenção desta linha na *cache* por um tempo maior. Mesmo que uma linha na *cache* seja intensamente referenciada enquanto está carregada, esta linha seguirá o fluxo normal de substituição segundo o critério de ordem de entrada estipulado pelo algoritmo *FIFO*. O tempo de permanência de uma determinada linha dentro da *cache*, segundo a heurística *FIFO*, será proporcional ao número de linhas disponíveis no *cache set*. Ou seja, quão maior for o *cache set*, mais tempo uma linha permanecerá armazenada até que seja substituída.

A grandeza que determina o número de linhas no *cache set* é denominado *associatividade* da *cache*. Exemplificando, se um *cache set* é definido com associatividade 8 (*8-way*), a expectativa é que uma linha que tenha sido inserida na *cache* permaneça armazenada por, no mínimo, 8 instruções do tipo *load*. Um outro exemplo onde o *FIFO* apresentará um comportamento indesejado é quando o programa faz referências a um

mesmo endereço de dados e os demais programas referenciam endereços dispersos. Mesmo que este endereço esteja sob intenso acesso, após um número de instruções *load* maior que a associatividade da *cache*, a linha intensamente referenciada pelo primeiro programa será ejetada da *cache*, pois o algoritmo *FIFO* terá reciclado o *cache set* inteiro. Isso vai provocar um aumento de *cache misses*, parando o *pipeline* do processador.

Devido a sua característica de funcionamento, programas caracterizados por localidade temporal curta podem apresentar desempenho aceitável frente ao algoritmo *FIFO*. Um exemplo de localidade temporal curta ocorre quando um endereço de dados é carregado na *cache* e, após ter sido utilizado, volta a ser referenciado novamente logo após poucas instruções terem sido executadas. Ou seja, o intervalo entre duas referências ao mesmo endereço é de poucas instruções. Nesta situação, os dados são referenciados novamente antes que sejam substituídos pelo critério de ordem de entrada na *cache*. Exemplificando, suponha um *cache set* com associatividade 4 (*4-way*). Para a sequência de acesso aos endereços: $\{A,B,A,D,A,C\}$ teremos 1(um) *miss* e 2(dois) *hits* para o endereço $\{A\}$. A figura 3.3 ilustra esse exemplo.

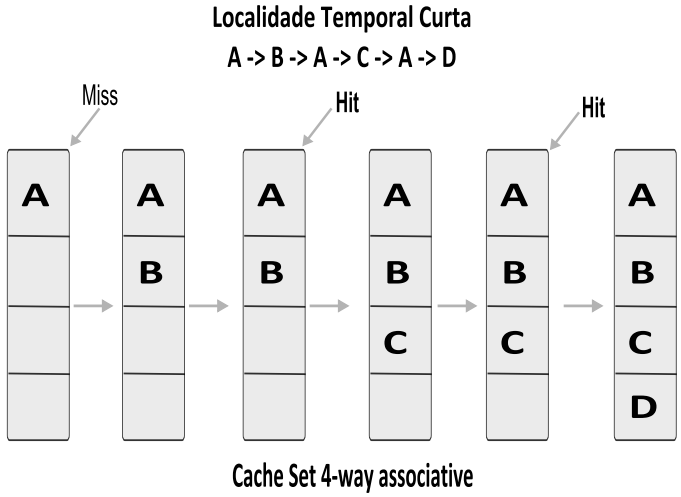


**Figura 3.3:** *Cache FIFO - Localidade Temporal Curta*

Em contrapartida, se a localidade temporal existir, mas, os endereços forem acessados após longos intervalos de instruções, a chance de encontrar as linhas necessárias na *cache* será reduzida. Após muitos ciclos de *clock* ou de instruções do tipo *load*, todas as linhas da *cache* terão sido retiradas para dar espaço a novas linhas referenciadas durante este longo intervalo. Utilizando o mesmo exemplo acima de um *cache set 4-way* e, para uma sequência de referência a endereços $\{A,B,C,D,E,A\}$ teremos 2 (dois) *misses* para o endereço $\{A\}$. Isso ocorre porque o algoritmo *FIFO* não prioriza a retenção na *cache* independente do número de acessos ou referências que o programa faz a uma linha durante o período que ela está carregada na *cache*.

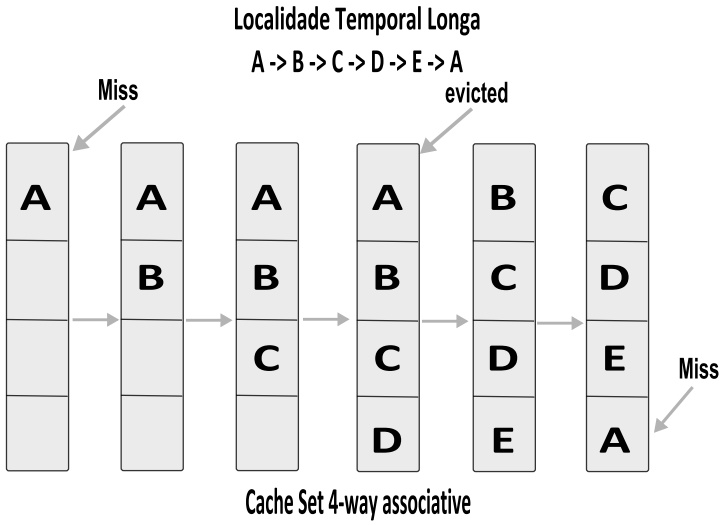


**Figura 3.4:** *Cache FIFO - Localidade Temporal Longa*

O algoritmo *FIFO* pode ser uma opção interessante para processadores cuja disponibilidade de área de circuito é reduzida. Seu desempenho pode ser aproximado ao de algoritmos mais elaborados (SMITH, 1982), porém, com uma implementação simples. Pode-se implementar o *FIFO* na *cache* através de um *register ring*. Devido às suas características, o algoritmo *FIFO* torna-se uma alternativa interessante para processadores embarcados, tais como *Intel XScale*, *ARM9* e *ARM10* (REINEKE et al., 2007).

A figura 3.5 ilustra o funcionamento de uma *cache* que implementa a substituição de linhas usando o algoritmo *FIFO*:

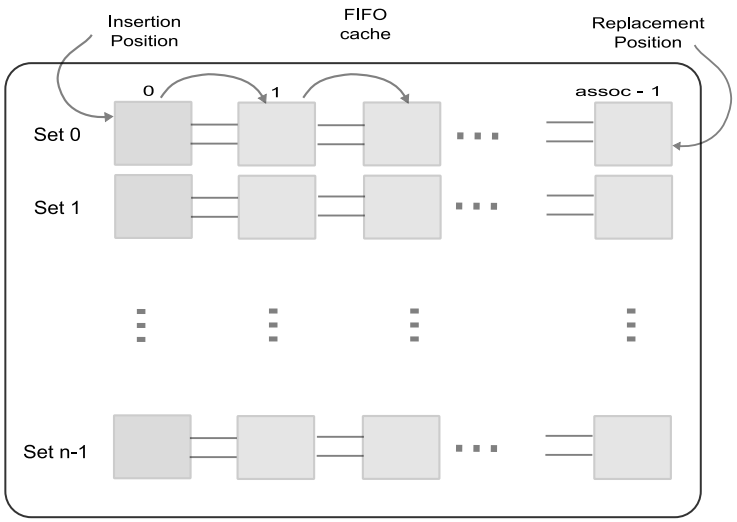


**Figura 3.5:** *Cache FIFO*

Apesar da simplicidade de funcionamento e da ausência de uma priorização de retenção da linha mais utilizada, o algoritmo *FIFO* apresenta desempenho comparável, em determinadas circunstâncias, a algoritmos como o *LRU*. (SMITH, 1982) mostrou

que, para um conjunto particular de programas, em média, a taxa de *miss rate* do algoritmo *FIFO* foi 12% superior ao do *LRU* para os mesmos programas. Se o *cache set* for muito grande o custo de cálculo do *LRU* pode ser proibitivo. Nestes casos ele pode ser aproximado pelo *FIFO* (PATTERSON; HENNESSEY, 2007).

## 3.4 Algoritmo LRU

Uma outra classe de algoritmos de substituição de linhas visa priorizar a retenção na *cache* utilizando um critério baseado no acesso ou referência. Baseando-se no histórico de utilização das linhas armazenadas no *cache sets* estima-se quais destas linhas tem maior previsibilidade de acesso num futuro próximo (BELADY, 1966). Nesta classe de algoritmo encontramos o algoritmo *LRU*, que pode ser entendido como um refinamento do algoritmo *FIFO* (BELADY, 1966). Existem variações deste algoritmo, mas, todos em geral mantêm controle sobre a utilização e referência aos endereços. A partir deste controle, o algoritmo prioriza a retenção de uma linha na *cache* em detrimento de outras.

Ao contrário do algoritmo *FIFO*, que leva em conta apenas a ordem de entrada das linhas na *cache*, o algoritmo *LRU* leva em consideração os acessos que uma linha recebe enquanto permanece armazenada na *cache* (BELADY, 1966). Esse critério permite ao algoritmo decidir qual linha deverá ser substituída quando todos os endereços estiverem em uso. Para tanto, utiliza-se um *bit* adicional para cada linha que indica o seu *status*. Quando há um acesso a uma linha que já está carregada na *cache*, seu *status bit* é ajustado para indicar que ela foi recentemente acessada (BELADY, 1966). Como o algoritmo *LRU* escolhe para a substituição a linha cujo acesso se deu há mais tempo, esta última linha acessada estará longe da posição de substituição, pois, o *status bit* indicará que a linha foi referenciada recentemente. O algoritmo *LRU* apoia-se no passado para prever o futuro (PATTERSON; HENNESSEY, 2007).

À medida que as linhas são carregadas na *cache*, o controlador de *cache* mantém um histórico que permite contabilizar quão recente se deu o último acesso às linhas carregadas e decidir qual será a melhor candidata à substituição. Para isso, para cada bloco de dados existem, além dos *tags* e *bits* de validade, os *LRU bits*. O *cache controller* limpa os *LRU bits* quando ocorre um *cache miss*. Assim que uma nova linha é carregada, o seu *LRU bit* é atualizado para indicar que ela foi recentemente usada. Quando houver a necessidade de substituir uma linha da *cache* o *cache controller* escolherá as linhas que não tem o *LRU bit* ativado.

Em particular, para *workloads* cujo o acesso à memória se dá de forma sequencial,

esta abordagem *LRU* acaba por manter mais tempo na memória *cache* blocos de dados que não serão usados novamente. Isso causa o desperdício de endereços valiosos para armazenar os dados das instruções em execução (JIANG; ZHANG, 2002). Em situações em que o programa apresenta localidade temporal esparsa ou processamento de programas que efetuam leitura de *working sets* maiores do que o tamanho da *cache*, o algoritmo *LRU* vai reciclar completamente os endereços carregados na *cache*. Isso vai ocorrer antes que as instruções voltem a referenciar os dados que haviam sido inicialmente carregados na *cache* (QURESHI et al., 2007).

No processador *AMD Opteron* é empregada a técnica denominada *victim cache* (JOUPPI; FULLYASSOCIATIVE, 1990; PATTERSON; HENNESSEY, 2007) para minimizar os efeitos negativos deste comportamento. Em um contexto de gerenciamento de acesso a endereços de múltiplas *threads* concorrentes e independentes entre si, e que compartilham o mesmo *cache set*, o *LRU* acabará por ejetar da *cache* os blocos de dados que seriam usado nos próximos ciclos e instruções do processamento. As *victim caches* ajudam a minimizar este efeito colateral, pois, retêm em uma área intermediária as linhas ejetadas acreditando que esta ejeção foi feita equivocadamente, isto é, dão uma segunda chance, evitando a ejeção desnecessária da linha.

Um aspecto positivo do *LRU* é que, nas situações em que o programa apresenta localidade temporal coesa, da mesma forma como descrito no algoritmo *FIFO*, o desempenho do algoritmo *LRU* é satisfatório frente aos algoritmos mais sofisticados e complexos. Programas com *loops* de execução curtos tendem a reter as linhas na *cache* por tempo suficiente para que o *loop* reinicie e, ainda, os dados sejam encontrados e referenciados na *cache*.

Apesar de ser necessário manter, em *hardware*, estruturas para contabilizar e controlar quais linhas foram recentemente acessadas e o *status* de cada uma delas, o *LRU* pode ser implementado com um bom balanceamento entre complexidade e desempenho. Por isso, grande parte dos microprocessadores de propósito geral utilizam esta abordagem como, por exemplo, os processadores *AMD Opteron* (PATTERSON; HENNESSEY, 2007) e *Fujitsu Sparc64 V* (INOUE, 2004). Quando o tamanho da *cache* implementada dentro do processador aumenta, tendência esta observada nos processadores modernos, a manutenção do *status* de todas as linhas apresenta complexidade proporcional a este aumento de tamanho de *cache*. Este aumento pode vir a inviabilizar o uso do *LRU* como política de substituição de linhas de *cache*. Nestas situações, pode ser adotada uma variação, por exemplo o *pseudo-LRU*, para manter o algoritmo funcionando e contornar o aumento de complexidade e de custo computacional.

A figura 3.6 ilustra o mecanismo *LRU* de substituição de linhas de *cache*.

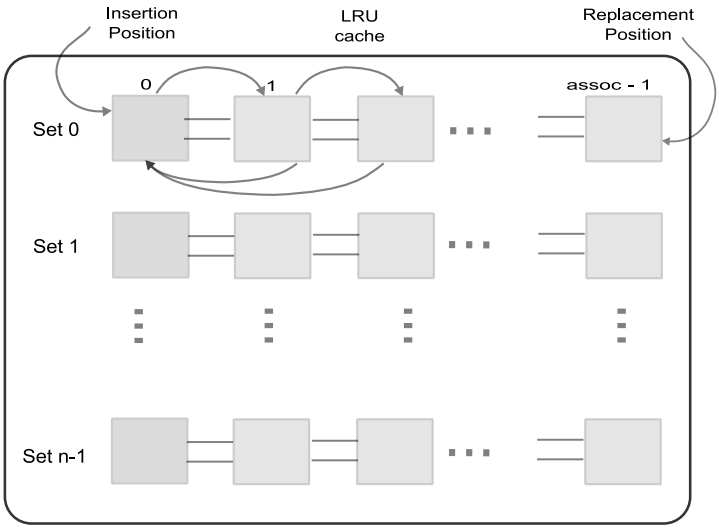


**Figura 3.6:** *Cache LRU*

## 3.5 Algoritmo Random

Uma política de substituição de linhas de *cache* baseada em um critério aleatório pode apresentar bons resultados em situações onde o *working set* do(s) programa(s) utiliza um conjunto reduzido de endereços durante o processamento. Da mesma forma, se o padrão de referência aos endereços é esparso, a localidade espacial e temporal deste(s) programa(s) será baixa. Na eventual necessidade de substituir uma linha de *cache* que esteja carregada, o algoritmo aleatório escolherá uma dentre as posições disponíveis no *cache set*. Todas as linhas de *cache* tem a mesma chance de serem substituídas. Nenhuma sofisticação adicional é embutida neste neste algoritmo *Random*. Não é mantida nenhuma estrutura adicional de controle em *hardware*. Não é necessário manter um inventário de acessos aos endereços carregados. Utiliza-se um gerador aleatório de endereços, por exemplo um *LFSR - Linear Feedback Shift Register*, para apontar para algum dos endereços do *cache set* que será substituído. A figura 3.7 ilustra a integração do gerador de endereços aleatórios com a memória *cache*. [1]

Uma vez que a linha a ser substituída pode ser qualquer uma dentre as que estão na *cache*, não há necessidade de implementar estruturas de controles similares aos que o algoritmo *LRU* utiliza para determinar qual será a próxima linha a ser substituída. Nas *caches* em que a associatividade é muito baixa, a chance do algoritmo *Random* ejetar uma linha útil torna-se maior do que a mesma *cache* com associatividade mais alta. Exemplificando, em uma *cache 4-way associative*, a chance do algoritmo ejetar uma linha útil é de 25% ao passo que em uma *cache 8-way associative* a chance do mesmo ocorrer é de 12.5%.

[1] O diagrama não leva em consideração o barramento de controle

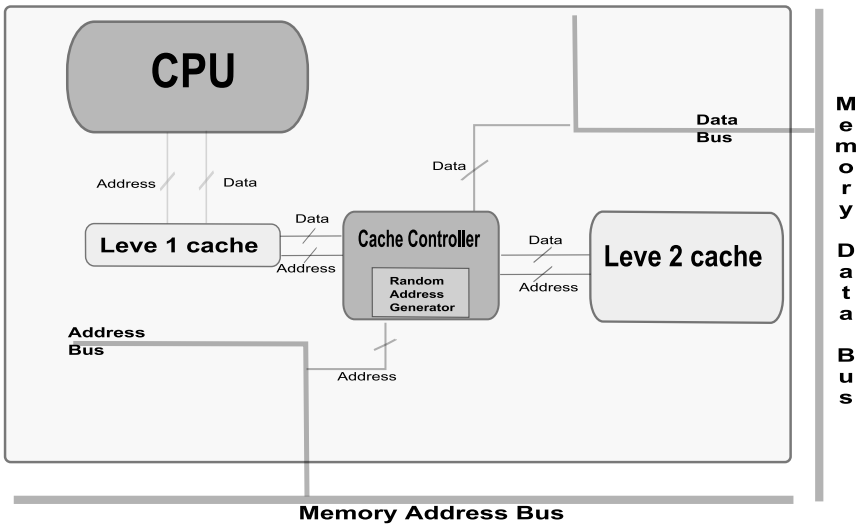


**Figura 3.7:** *Cache Random Block Diagram* - Inspirada em (HILL, 1987)

Nos casos onde o(s) programa(s) requisita(m), recorrentemente, uma grande quantidade de endereços (*working set*), a chance de, no sorteio aleatório, uma linha de memória *cache* útil ser ejetada é maior. O algoritmo *Random* não tenta capturar a localidade temporal ou a localidade espacial do(s) programa(s). O algoritmo não tenta reter na *cache* as linhas que possam ter maior chance de serem referenciadas novamente. Não há uma priorização de algumas linhas em detrimento de outras. (PATTERSON; HENNESSEY, 2007) mostra que para *caches* pequenas (*16-KB*, *64-KB*) o algoritmo *Random* perde em desempenho para os algoritmos *LRU* e *FIFO*. No entanto, para *cache* maiores do que os valores mencionados acima o desempenho do algoritmo *Random* é equivalente ao desempenho dos algoritmos *LRU* e *FIFO*.

Nas situações em que se processam centenas de milhares de pequenas *threads* independentes, os dados utilizados durante o processamento não precisam ser mantidos na *cache*, pois a natureza destas aplicações pressupõe o descarte completo de seu contexto (estado) de processamento. Aplicações que utilizam o protocolo *HTTP - HyperText Transfer Protocol* são exemplos deste tipo de aplicação. Este tipo particular de aplicação é conhecida como *Stateless Applications* (PETER, 2008). O processamento das *threads* é concluído na primeira vez que são carregadas no *pipeline*. Estas *threads* não voltam a ser processadas novamente e seu estado (*state*) não é mantido na *cache*. Neste contexto, os dados carregados na *cache* para execução de suas instruções não serão utilizados novamente no futuro. Assim, pode-se escolher aleatoriamente qualquer linha da *cache* para ser substituída sem prejuízos para o funcionamento das outras *threads*.

Um outro benefício desta abordagem é a possibilidade de se implementar o algoritmo usando poucos circuitos e poucas estruturas de controle em oposição ao que

acontece com algoritmos mais complexos com o *LRU*. Assim, sua simplicidade torna-o interessante para pequenos sistemas embarcados onde o compromisso de projeto é a economia de energia do circuito e não o desempenho discreto dos programas que rodam neste ambiente. Estruturas mais simples precisam de circuitos mais simples e que consomem menor potência. Exemplo deste emprego está no processador ARM 1020T (ARM, 2000) e ARM Cortex-A (ARM, 2008) destinados a computação em dispositivos móveis.

Em (PATTERSON; HENNESSEY, 2007), demonstra-se que, para um processador *Alpha* com *cache block size* de *64-bytes* e rodando programas *SPEC2000*, que o *LRU* se sobressai frente ao *FIFO* e *Random* quando o *cache size* é pequeno. Para *cache size* maiores a diferença entre o desempenho dos três algoritmos é marginal.

# 4 Estado da Arte

A indústria de semicondutores e a comunidade científica que pesquisa sobre Arquitetura de Computadores é proeminente. O avanço científico sustentado em diversas frentes desta área de pesquisa apoia-se, cada dia mais, nos avanços conquistados na área da computação e engenharia de sistemas. Os sistemas computacionais, em detrimentos de todos os desafios técnicos, vêm evoluindo e aumentando sua capacidade computacional. Esta capacidade permite que a ciência, em geral, possa desenvolver modelos e simulações sofisticadas e, assim, avançar a fronteira do conhecimento nas suas diversas frentes.

Diversos trabalhos na área de Arquitetura de Computadores foram propostos. Em especial, a pesquisa a cerca da hierarquia e da organização da memória dos computadores trazem à comunidade resultados muito positivos e relevantes. O ponto em comum entre os diversos trabalhos relacionais à hierarquia de memória é o objetivo de encontrar métodos e otimizações que consigam esconder e minimizar os efeitos negativos da latência de acesso aos dados por parte dos processadores. Da mesma forma, a pesquisa na área de Arquitetura avançou nas proposta de desenho de processadores com múltiplos *cores* (*CMP*)(BARROSO et al., 2000; BURGER; GOODMAN; KAGI, 1995; NAYFEH; HAMMOND; OLUKOTUN, 1996; BURGER; GOODMAN; KäGI, 1996; L. NAYFEH B. A., 1997; KUMAR; JOUPPI; TULLSEN, 2004; KUMAR et al., 2004; CHISHTI; POWELL; VIJAYKUMAR, 2005; FEDOROVA et al., 2005; MARINO, 2006) e com múltiplas *threads* por *core* (*SMT*) (KUMAR; JOUPPI; TULLSEN, 2004; OLUKOTUN; H., 2005; L. NAYFEH B. A., 1997; TULLSEN; EGGERS; LEVY, 1995; SPRACKLEN; ABRAHAM, 2005). As propostas apresentadas exploram o paralelismo que existe em nível de *threads* nos programas.

Existe um aspecto que sempre é levado em consideração no desenho dos processadores, o gerenciamento da hierarquia de memória e como projetar esse mecanismo de forma a manter o *pipeline* do processador alimentados com instruções e dados. Este aspecto é relevante para todas as arquiteturas de processadores modernos, independente de serem *single core*, *multi-cores*, *single threaded* ou *multi-threaded*. Este gerenciamento da hierarquia de memória é tratado através do algoritmo de substituição de

linhas de *cache*. Seu funcionamento deve garantir que os dados e instruções mais úteis para o processamento estejam mais próximos das unidades de processamento. Em linhas gerais, o algoritmo deve otimizar o uso da memória disponível nas áreas de *cache*, explorando a localidade dos programas e melhorando o desempenho dos programas.

Em (QURESHI; SULEMAN; PATT, 2007) são apresentados os problemas decorrentes do desperdício de endereços na memória *cache* quando os programas não apresentam localidade espacial. As *caches* são projetadas para trabalhar com um *cache block size* que tem como objetivo explorar a localidade espacial. No entanto, quando os programas não apresentam essa propriedade ocorre o desperdício da área de *cache*, pois o *cache block* armazena diversas linhas que não são utilizadas. Os autores propõem a utilização de uma técnica chamada *LDIS - Line Distillation* que retém na *cache* apenas as linhas que são referenciadas e ejeta as linhas que não são referenciadas enquanto estão carregadas na *cache*. Ao invés de ejetar um bloco inteiro contendo múltiplas linhas, essa técnica confere maior granularidade no gerenciamento da área de *cache*. Os endereços liberados pela técnica *LDIS* podem, então, ser reutilizados por outras linhas. Esta técnica foi avaliada simulando programas de *benchmark* que fazem intenso acesso à memória e apresentou uma redução da ordem de 30% na taxa média de *miss* e um aumento de 12% (doze por cento) no *IPC - Instructions per Cycle* para uma *cache* de 1-MBytes (*um megabyte*) com associatividade *8-way* no segundo nível da memória *cache*.

Ainda, (QURESHI et al., 2007) discorre sobre o mau comportamento do algoritmo *LRU* quando os programas são caracterizados por acesso intenso à memória e o *working set* do programa é maior que o *cache size*. Os autores ressaltam que o *LRU* é o algoritmo predominante nas implementações de processadores. As diversas tentativas de melhorar seu desempenho incorram em aumento significativo de área, complexidade no desenho e verificação dos circuitos e baixa resposta quando os *workloads* são favoráveis ao *LRU*. Ao invés de modificar a forma como o *LRU* substitui as linhas da *cache*, os autores propõem uma modificação na forma com que o algoritmo insere as linhas trazidas da memória principal na *cache*. Usualmente, o algoritmo *LRU* insere as linhas recém trazida na porção *MRU - Most Recently Used* da *cache*. Se essas linhas não são referenciadas novamente enquanto estão carregadas, elas vão sendo migradas para a posição *LRU* da *cache* até serem substituídas. A proposta é inserir as linhas diretamente na posição *LRU*, evitando que fiquem armazenadas na *cache* por um tempo desnecessário.

Essa nova política é chamada de *LRU Insertion Policy*. Se houver acesso subsequente às linhas recém inserida na *cache*, esta linha é, então, migradas para a parte *MRU*. Para sustentar essa proposição, os autores mostram que, em média, 60% (ses-

senta por cento) da linhas ejetadas da *cache* não sofreram um segundo acesso após terem sido carregadas. Os autores também propõem criar uma política dinâmica de inserção que escolha, para os diversos *cache sets*, qual a melhor políticas de inserção. Os resultados apresentados mostram um redução de 21% no número *MPKI - Misses per Kilo Instructions*. Esta proposição é interessante porque permite que a *cache* capture melhor a localidade de referência e apresente um comportamento mais adaptativo em relação à característica dos programas, melhorando o desempenho final.

A oportunidade de explorar as áreas de *cache* através de política dinâmica de inserção de linhas é discutida no trabalho (XIE; LOH, 2009). Neste trabalho, os autores reafirmam a necessidade de promover a inserção de linhas na *cache* como uma forma de contornar os problemas de contenção nos endereços de *cache*. Essa contenção é decorrente da não-uniformidade na forma como os programas acessam estes endereços. A proposição deste trabalho está consoante com a proposição dos trabalhos apresentados acima. Ao invés de adotar uma arquitetura de *cache* organizada em *cache sets* com uma associatividade pré-definida, a proposta deste trabalho é realizar a inserção e substituição de linhas na *cache* tratando-a como uma estrutura dividida em múltiplas pseudo-partições. O trabalho destaca os efeitos negativos da posição de inserção adotada pelo algoritmo tradicional *LRU* da mesma forma como foi destacado no trabalho (QURESHI et al., 2007).

Ainda, este estudo apresenta uma contribuição interessante ao discutir a forma como o algoritmo *LRU* promove as linhas de *cache* quando estas recebem referências durante o tempo em que estão carregadas. A contribuição deste trabalho reside na técnica que permite particionar a memória *cache* entre os diversos *cores* do processador de forma dinâmica. Esse particionamento permite a adequação da quantidade de *cache tags* que um processador ou *core* do processador pode utilizar na *cache*. Também, esta nova forma de inserir, promover e ejetar as linhas destas partições da *cache* apresenta resultados da ordem de 21.9% melhores do que os resultados obtidos para uma *cache* convencional usando *LRU* e organizada em *cache sets* com associatividade pré-definida. Finalmente, esse estudo apresenta uma contribuição positiva na forma de gerenciar a *cache* e minimizar os efeitos indesejáveis do comportamento do *LRU* frente aos *workloads* cuja localidade de referência não é favorável ao algoritmo *LRU*.

Outro problema decorrente da longa retenção que o algoritmo *LRU* pode causar é a manutenção de uma linha na *cache* que não será mais utilizada por um tempo maior do que o necessário. Isso decorre da localização da posição de inserção do algoritmo *LRU*. Conforme (QURESHI et al., 2007), o *LRU* insere as novas linhas na porção *MRU* da *cache*. Para contornar os efeitos negativos desta estratégia de inserção de linhas, os autores do trabalho (LIU et al., 2008) propõem um mecanismo capaz

de identificar *dead blocks* armazenados na *cache*. Ao identificar estes *dead blocks*, é possível promovê-los para ocorrer a substituição de linhas. A partir da capacidade de prever que um bloco não será mais utilizado pode-se aumentar a utilização dos endereços da *cache*, pois endereços antes ocupados por dados inúteis serão liberados para uso. Para decidir se um bloco está em *status dead* eles introduzem o conceito de *cache burst*, inspecionando o número de acessos contíguos que um endereço recebe enquanto está na porção *MRU* da *cache*. A partir do momento que se identifica um *dead block* a *cache* pode efetuar a substituição muito antes desta linha se tornar a linha *LRU*. Os autores combinam a nova forma de identificação de *dead blocks* com técnicas como *cache bypassing* e *cache line prefetching* para melhorar os efeitos da técnica proposta. No melhor resultado, os autores mostram que o *cache burst prediction* pode identificar 96% dos *dead blocks* com 96% de precisão. Porém, os resultados finais apresentados ainda indicam um baixo índice de eficiência nos níveis de memória *cache* (17% para L1$ e 27% para L2$).

Com a indústria avançando na adoção da arquitetura de processadores *multi-cores* faz-se necessário rever oportunidades de otimização na forma como a replicação de cópias de dados é feita entre os múltiplos *cores* de um mesmo *chip*. Em (CHISHTI; POWELL; VIJAYKUMAR, 2005), os autores propõem que a replicação das cópias seja controlada entre os diversos *cores* para evitar que múltiplas réplicas consumam espaço desnecessário nas *caches* dos processadores. Este controle na replicação também reduz a pressão por largura de banda entre os *cores*. Para manter as linhas mais úteis próximas dos *pipelines* os autores propõem uma política de inserção das linhas controlada por distância. Esta política é denominada *NuRAPID - Non-Uniform Replacement and Placement Using Distance*. Ao invés de utilizar uma *cache* com associatividade determinada, emprega-se um *data array* e uma estrutura de acessos sequenciais *data-tag* que permite fazer a substituição e inserção de linhas usando ponteiros ao invés de movimentar as linhas com dados. Estes ponteiros indicam onde as linhas devem ser inseridas na *cache*. Estas operações com ponteiros ajudam a reduzir a movimentação de linhas entre os *cores* e reduz o consumo de banda de comunicação.

Também, através destes ponteiros os *tag arrays* de cada processador podem apontar para a mesma posição da *cache* e evitar que seja feita uma cópia adicional controlando, assim, a replicação. Para controlar a inserção e substituição é proposto um mecanismo de agrupamento das linhas da *cache* chamado de *D-groups*. Cada um destes *D-groups* apresentam um tempo uniforme de acesso intra-grupo. Porém, os tempos de acesso inter-grupos são não uniformes. As linhas mais referenciadas são mantidas nos *D-groups* mais próximos dos processadores. Os resultados mostram uma melhora da ordem de 13% para *shared caches* e 8% de melhora para *private caches* usando *wor-*

*kloads* comerciais e do programas do *benchmark SPEC*. Posicionar as linhas de *cache* em regiões mais próximas do *pipeline* representa um boa estratégia para diminuir a latência de acesso aos dados. Pode-se estender esse critério de *D-groups* e acrescentar um histórico que permita refinar a política de posicionamento das linhas nos vários *D-groups*.

Em (CHISHTI; POWELL; VIJAYKUMAR, 2005) o problema de *trade-off* de latência versus capacidade é apresentado e discutido. Da mesma forma, em (ZHANG; ASANOVIC, 2005) é apresentada uma proposta que trata a *cache* em múltiplos *slices* para aproveitar o melhor da arquitetura compartilhada e da arquitetura privada. Do ponto de vista dos processadores, cada *slice* é enxergado como sendo uma *cache* privada com baixa latência. Do ponto de vista do processadores, a área formada por todos os *slices* é tratada como uma *shared cache*. A comunicação *cache-to-cache slices* é feita por uma rede *intra-chip*. Cada *cache slice* se assemelha aos *D-groups* utilizados na abordagem (CHISHTI; POWELL; VIJAYKUMAR, 2005). É proposto um mecanismo de retenção das linhas ejetadas da *cache*, cujo objetivo é capturar melhor a localidade do programas, reduzindo o tráfego de cópias de linhas de dados *intra-chip* através da criação de um *buffer* para armazenar linhas que foram substituídas da *cache* e que, eventualmente, viriam a ser referenciadas novamente. Reduzir este tráfego é importante para maximizar as áreas de *cache* privados de cada um dos *cores* e reduzir os requisitos de área necessários para construir rede de comunicação *intra-chip* para sustentar largas bandas de comunicação.

Em (KIM; BURGER; KECKLER, 2002) os autores discutem os problemas relacionados ao tamanho da área de memória *cache*. Dentre os principais problemas que tangenciam as memórias *cache* o tempo de latência é, em especial, importante na organização de computadores. Com o aumento constante no tamanho das áreas de *cache* dos processadores, surge uma não-uniformidade no tempo de acesso aos dados para as diferentes secções da memória *cache*. O estudo apresenta a oportunidade de explorar esta não-uniformidade de acesso para melhorar o tempo de latência. Para tanto, posiciona as linhas mais úteis em áreas mais próximas dos *pipelines*. Esta proposição assemelha-se à proposta de (CHISHTI; POWELL; VIJAYKUMAR, 2005). A idéia apresentada é permitir que a memória *cache* não-uniforme seja dinâmica e movimente as linhas mais utilizadas para *cache banks* mais próximos dos *pipelines*. Para isso, a *cache* utiliza o algoritmo *LRU* para reordenar os *cache banks* e deixar as linhas mais usadas na região *MRU*, próximas dos processadores. Os resultados apresentados mostram uma melhora de 50% no *IPC* dos programas.

O problema de gerenciamento dos endereços disponíveis na *cache* representa um desafio especial no projeto de processadores. Em (HARDAVELLAS et al., 2009) os

autores avaliam o problema de aumento da latência de acesso devido ao aumento da área de *cache* e identificam que o padrão de acesso dos programas pode ser divido em classes. Propondo uma arquitetura *NUCA - Non Uniform Cache Architecture* capaz de responder às estas classes de acesso, os autores propõem uma forma inteligente de inserir, movimentar (migrar) e replicar as linhas entre os múltiplos *cores* do processador sem incorrer em aumento significativo de *overhead* no protocolo de coerência. Esta proposta identifica as classes de acesso e faz com que o comportamento da *cache* reaja de forma diferente para cada uma das classes. Para viabilizar o funcionamento, a memória *cache* coopera com o sistema operacional para identificar a melhor estratégia para cada uma das classes. Os resultados mostram que a proposta *Reactive NUCA* atinge *speedup* 14% melhor que uma *private cache* e 6% melhor que uma *shared cache*.

(SPEIGHT et al., 2005) propõe utilizar a *cache* de terceiro nível para atuar com uma *victim cache* do segundo nível *L2*. Esta abordagem visa reduzir a demanda por banda de comunicação entre processador e memória, reduzindo a latência de acessos. É proposta uma *WBHT - Write Back History Table* implementada através de uma *lookup table*. Aproveitando esta estrutura, pode-se estender seu funcionamento acrescentando os campos necessários para o funcionamento do *Entropy* descritos na secção 5.2 do capítulo 5.

A finalidade dos algoritmos de substituição de linhas de *cache* é capturar e explorar a localidade dos programas. Esta localidade pode ser temporal e explorada conforme as propostas mencionadas acima, ou pode ser espacial. Em (SOMOGYI et al., 2006), os autores propõem a utilização de um vetor de *bits* que representa o conjunto de blocos acessados durante um intervalo de instruções. Cada intervalo é chamado de *Spatial Generation Region*. A proposição apoia-se no fato de que, para determinados *workloads* tais como banco de dados relacionais, existe uma grande correlação espacial entre blocos de dados. Mesmo quando eles não estão alocados contiguamente, existirá essa correlação. Tal situação foi descrita no capítulo 3. Utilizando-se duas tabelas auxiliares, uma ativa e outra histórica, a proposta registra os padrões de acesso aos blocos de dados usando o vetor de *bits*. Este vetor serve para indicar e prever quais dentre estes blocos serão novamente referenciados. Esta técnica é especialmente interessante, pois, grande parte do *workload* processado em computadores corporativos encaixam-se no perfil de programa descrito neste trabalho mencionado.

O trabalho (SOMOGYI et al., 2006) apresentado acima é bastante interessante e positivo e contribui com o objetivo de capturar a localidade espacial dos programas. Em complemento à esta proposição, (SOMOGYI et al., 2009) propõe uma técnica capaz de capturar a localidade temporal dos programas e explorar ao máximo a opor-

tunidade de redução de penalidades decorrentes de referências à memória *off-chip*. A técnica é denominada *STeMS - Spatio Temporal Memory Streaming* e abrange a localidade espacial e temporal dos programas. Para combinar as duas técnicas de *memory streaming* os autores estendem o conceito de correlação espacial de endereços e criam o conceito de correlação temporal. A proposta baseia-se na observação de que o padrão de acesso dos programas ocorre, em intervalos de tempo, sobre regiões distintas do *address space* e existe correlação espacial entre os blocos dentro destas regiões. Em comparação às técnicas como *stride prefetching* a proposta mostra resultados da ordem de 31% melhores para *workloads* comerciais, resultado este bastante positivo.

Em (BISWAS et al., 2009) os autores apresentam uma proposta que captura a similaridade no acesso aos endereços em ambientes *multi-execution* ou multiprogramados. Há a constatação de que, mesmo em sistemas com este tipo de *workload*, os programas podem fazer referências aos mesmos endereços. Este trabalho apresenta-se na mesma linha do trabalho (SOMOGYI et al., 2009) que captura a correlação espacial para *workloads* comerciais. Esta proposta é relevante, pois reduz o número de referências à memória principal identificando esta similaridade. Tal similaridade é explorada através da fusão (*merging*) de linhas de *cache* idênticas usadas por programas distintos. Os resultados mostram uma melhora da ordem de 2.5 vezes (duas vezes e meia), em média, usando programas de *benchmark*. Esta memória *cache* é denominada *Mergeable cache* e implementa *tags* virtuais e *processor IDs* para controlar a fusão das linhas da *cache*. Esta proposta tem relevância na tentativa de reduzir o número de acessos à memória principal, pois, fundindo as linhas de *cache* idênticas aumenta-se a área útil da *cache*. Ainda, durante a ejeção das linhas da *cache* apenas uma linha e transmitida para a memória principal ao invés de múltiplas cópias desta linha idêntica compartilhada entre diversos programas.

# 5 Entropia da Informação

Em 1948, o matemático *Claude E. Shannon* introduziu, no artigo intitulado *A Mathematical Theory of Communication*, o conceito da Entropia da Informação (SHANNON, 1948). Em sua proposição, *Shannon* estipula que uma sequência de caracteres que formam uma mensagem pode ser interpretada como uma sequência de variáveis aleatórias. Cada ocorrência de um caractere na sequência não é influenciada pela ocorrência dos caracteres anteriores. A cada novo caractere da mensagem que é recebido, aumenta-se a chance de prever o restante da mensagem a ser transmitida. Apesar de serem eventos disjuntos, a observação do último valor (caractere) ocorrido da variável aleatória aumenta o nível de conhecimento (previsibilidade) acerca do próximo caractere que se espera na cadeia de caracteres que forma a mensagem.

A esse *Conhecimento*, *Shannon* atribuiu o nome de Entropia da Informação. Essa Entropia da Informação é a medida da quantidade de informação contida em parte de uma mensagem sendo transmitida. Quanto mais se sabe sobre os caracteres da mensagem que já ocorreram, mais previsível se tornam os próximos caracteres da sequência. A figura 5.1 ilustra, em linhas gerais, esse conceito. Conhecendo o alfabeto da língua portuguesa e as palavras deste idioma, pode-se, a medida que a sequência de letras aparece, deduzir com maior certeza a próxima letra que forma a palavra (sequência). A partir de um determinado trecho da mensagem já transmitido, tem-se um conhecimento suficiente sobre a mensagem, o quê permite que o restante dela seja deduzido.

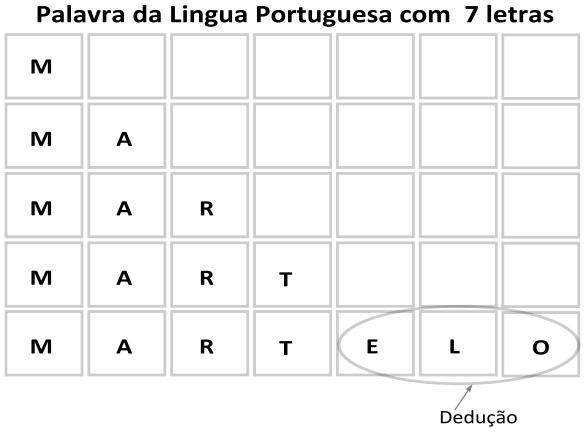


**Figura 5.1:** Entropia da Informação

No contexto de Entropia a estimativa de probabilidade é fundamental, pois, para uma determinada variável aleatória, a Entropia discreta associada àquela variável é dada pela expressão:

$$u(x_i) = log_b(1/p(x_i)) -> u(x_i) = -log_b(p(x_i)), b = \{2, e, 10\} \tag{5.1}$$

Aplicando-se esse conceito a um espaço de endereçamento de memória, se um determinado endereço é referenciado com maior frequência dentro da sequência $X$ observa-se que a Entropia desta sequência, da qual este endereço faz parte, tenderá a um valor inferior. Essa Entropia da sequência diminui, pois, a medida que um particular valor do espaço amostral ocorre com maior frequência, a incerteza contida naquela sequência diminui e, consequentemente, a Entropia dessa sequência. O valor da Entropia da sequência pode até mesmo convergir para zero. Isso significaria a existência de certeza, ou ausência de incerteza, sobre o valor da próxima variável aleatória $x_i$.

Assim, tomando uma sequência de variáveis aleatórias $X = \{x_i : \ i = 1, ...n\}$, cada um dos possíveis valores de $x_i$ possui sua Entropia e incerteza discreta dada pela equação:

$$\begin{aligned} h(x_i) &= u(x_i) * p(x_i), i = \{1, \cdots, n\} \\ H(X = x_i) &= \sum_{i=1}^{n} p(x_i) * u(x_i) \end{aligned} \tag{5.2}$$

Ou seja, para uma sequência de variáveis aleatórias, quão mais conhecidas forem as probabilidades estimadas de ocorrência de cada uma das variáveis aleatórias, menor será o valor da Entropia desta sequência $X$. Neste trabalho, o conceito de Entropia discreta será utilizado para estimar as chances de um endereço de memória ser referenciado novamente após ter sido carregado na memória *cache*.

## 5.1 Algoritmo *Entropy*

Uma vez que não se pode, *a priori*, saber qual será o comportamento e o padrão de acesso à memória *cache* por parte de um programa de computador, tem-se a expectativa de obter impactos positivos através do emprego de uma política de substituição de linha de *cache* que apresente um comportamento capaz de capturar a localidade de referência dos programas. Para tanto, este trabalho introduz pela primeira vez, a utilização do conceito de Entropia discreta sobre a sequência de endereços de memória referenciados durante a execução dos programas de *benchmark* utilizados na simulação. O algoritmo

proposto trata cada endereço de memória dentro do *address space* como sendo um possível valor de ocorrência de uma variável aleatória. A medida que os programas simulados forem referenciando estes endereços, o algoritmo *Entropy* estima a chance de que aquele endereço seja novamente referenciado. Para isso, o algoritmo calcula a Entropia discreta daquele endereço.

O uso deste critério de estimativa de chance visa capturar a localidade de referência dos programas de forma mais efetiva do que as políticas de substituição de linhas de *cache* que se baseiam apenas na temporalidade de uso, como o *LRU*, ou em um critério de ordem (sequência) de referência como o *FIFO*. A figura 5.2 ilustra o funcionamento da memória *cache* que implementa o algoritmo *Entropy*. A ilustração representa a dinâmica de uma *cache* gerenciado pelo algoritmo proposto. Em oposição aos algoritmos *LRU* e *FIFO*, o algoritmo *Entropy* promove e demove uma linha de *cache* às posições mais ou menos suscetíveis a substituição.

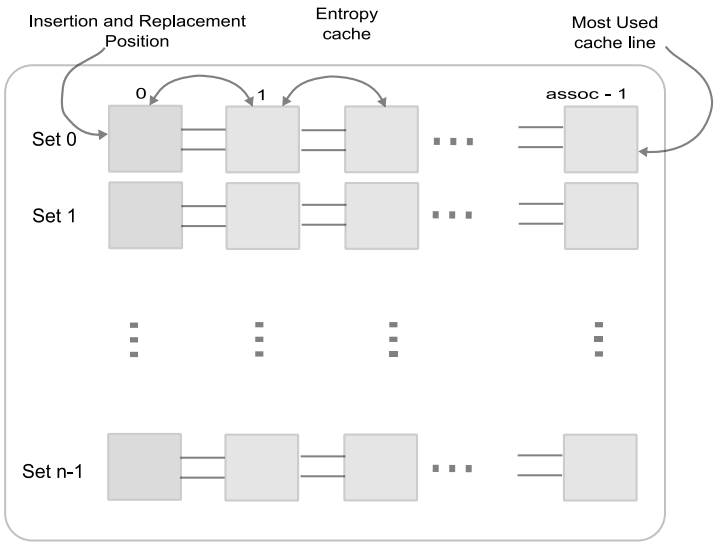


**Figura 5.2:** Cache Entropy

Os algoritmos usualmente empregados apresentam desempenho ruim diante de algumas situações conhecidas (QURESHI et al., 2007). Estas situações são caracterizadas por programas que efetuam longos *file scans*, multiplicação de matrizes esparsas, longos *loops* de execução e no caso de programas cujo *working set* seja maior que o tamanho da *cache* (QURESHI et al., 2007). A proposta deste estudo aborda o problema de substituição de linhas de *cache* usando uma heurística mais seletiva do que a utilizada nos algoritmos *FIFO*, *LRU* e *Random*. Inspirado no conceito de Entropia da informação, inicialmente apresentado e proposto pelo matemático *Claude E. Shannon*, este algoritmo de substituição de linhas de *cache* leva em consideração a frequência com que um determinado endereço de memória é referenciado durante a execução de um programa. Além do instante do último acesso, leva-se em consideração um histó-

rico mais longo de referência para os endereços. Esse aspecto diferencia o algoritmo *Entropy* dos algoritmos tradicionalmente implementados nos processadores.

A utilização de um histórico de referências aos endereços de memória pode causar um efeito colateral. Se um determinado endereço recebe intenso acesso durante parte do programa, seu contador de referências vai possuir um valor alto armazenado no histórico. Se este histórico de acessos for mantido neste indefinidamente e esta informação utilizada para decidir a posição de inserção deste endereço no *cache*, pode-se causar o confinamento definitivo desta linha na *cache*. Para evitar o confinamento de um endereço na *cache*, será utilizada uma função decaimento de Entropia. Ou seja, se um endereço é intensamente referenciado em um segmento do programa ele será priorizado na *cache*. Mas, se este mesmo endereço não sofrer mais acessos subsequentes no decorrer da execução do programa, ele deverá, gradualmente, ter seu histórico reduzido. O efeito da função decaimento é reduzir a importância que uma grande quantidade de referências ao endereço que ocorrem em um trecho antigo da programa comprometam a área disponível em *cache* para os endereços que estão sendo utilizados pelo programa no trecho mais atual da execução.

Se um endereço que teve seu histórico decaído for novamente referenciado, deverá entrar na *cache* com prioridade semelhante e ocupando a mesma posição que um endereço ainda não referenciado ou com poucas referências recentes ocuparia. Isso deve-se ao fato de que os acessos recebidos no passado não terão grande peso frente às referências mais recentes que estejam ocorrendo a outros endereços. Sem a função decaimento, um endereço intensamente referenciado no passado poderia permanecer na *cache* indefinidamente e consumir endereços de memória *cache* que poderiam armazenar dados mais úteis para o trecho atual do programa em execução. Para implementar a função decaimento, utiliza-se o conceito de *ts - time stride* que indica a distância, em ciclos, entre o ponto atual de processamento e a última referência que o endereço recebeu. Quanto maior for o valor de *ts* mais rápido será o decaimento da Entropia para aquele endereço.

Para descrever o algoritmo *Entropy*, suponha que, em linhas gerais, os endereços de memória solicitados por um programa podem ser descritos como sendo uma sequência de variáveis aleatórias $X = \{x_1, x_2, ..., x_n\}$ onde $x_i$ é o endereço de memória solicitado no $i-ésimo$ instante ou ciclo de execução. Para cada programa em execução tem-se uma sequência $X$ formada por um vetor de variáveis aleatórias. O conjunto de todos endereços de memória forma o espaço amostral da variável aleatória $x_i$. Pode-se estimar que, para cada um dos possíveis valores (endereços) que a variável aleatória

pode assumir, existe uma probabilidade associada e definida por:

$$p(x_i) = Pr(x_i) \tag{5.3}$$

Assumindo que a ocorrência de um determinado valor não influencie a probabilidade de ocorrência de nenhum outro possível valor do espaço amostral (*address space*), será estimada a probabilidade de ocorrência de cada valor através da razão entre a frequência com que um determinado endereço é requisitado e a frequência de acessos ao *cache set* onde este endereço reside. Assume-se, desta forma, que existe um caráter de independência entre as possíveis ocorrências da variável aleatória $x_i$. A ocorrência de dois endereços quaisquer é um evento disjunto onde a probabilidade de um não influencia a probabilidade do outro endereço.

Um fato interessante é que, se todos os endereços de memória tivessem uma distribuição equiprovável, seria atingida a Entropia máxima para a sequência $X = \{x_1, x_2, ..., x_n\}$. Nesta particular situação, uma escolha aleatória de qualquer posição do *cache set* seria possível, uma vez que, todas as linhas de *cache* apresentam a mesma probabilidade de ocorrer novamente na próxima instrução do programa. Se essa situação se concretizasse, o comportamento do *Entropy* seria semelhante ao do algoritmo *Random*. Estas variações no comportamento do *Entropy* é que reforçam a expectativa de que ele seja capaz de capturar a localidade dos programas de forma melhor que os algoritmos tradicionais. No entanto, esta situação de distribuição equiprovável configura-se como hipotética, pois, sabe-se que o padrão de acesso aos endereços por parte dos programas é não uniforme ou equiprovável (SMITH, 1982).

A partir do momento em que se tem o critério para estimar a probabilidade de ocorrência de um endereço dada por sua frequência de acesso, pode-se calcular a Entropia de cada um destes endereços solicitados. Para tanto, estes endereços serão tratados como sendo uma variável aleatória discreta. Assim, tem-se a expressão da Entropia discreta como segue:

$$h(X = x_i) = -p(x_i) * log_b(p(x_i)) \tag{5.4}$$

Utilizando a ideia e formulação matemática apresentada acima, o algoritmo proposto fará a substituição das linhas de memória *cache* segundo a medida de sua Entropia discreta. Ou seja, a medida que um determinado endereço é mais solicitado que os demais, sua chance de permanecer na memória *cache* aumenta na razão de sua Entropia discreta.

No entanto, existem situações ou trechos de execução de um programa que podem apresentar efeitos colaterais indesejados se aplicarmos a expressão enunciada em 5.4 arbitrariamente. Uma delas pode ocorrer no trecho inicial de um programa quando o primeiro endereço de memória é referenciado. Neste ponto, sua probabilidade $p(x_1)$ é de 100%. Ao substituir essa probabilidade na expressão acima, o algoritmo *Entropy* retornaria 0 (zero) e esse resultado seria inconsistente. Além disso, hipoteticamente, se um programa efetuasse sucessivos acessos a um mesmo endereço de forma a levar sua probabilidade a 100% a mesma inconsistência ocorreria novamente.

Para contornar este problema, a expressão 5.4 e o algoritmo *Entropy* sofreram ajustes de modo que essa inconsistência fosse evitada. Observa-se que a curva da Entropia sofre uma inversão a partir de um determinado valor de probabilidade. Para evitar que o valor de Entropia tenda a 0 (zero), a expressão 5.4 foi ajustada para criar um ponto de inflexão e acompanhar a evolução da probabilidade dos endereços. A expressão 5.5 mostra como a expressão original foi ajustada. Na expressão 5.5 pode-se notar a presença da função decaimento que ajusta o valor da Entropia discreta. Os efeitos deste ajuste podem ser observados através da figura 5.3 apresentada abaixo.

$$h(x) = \begin{cases} h(x_i) = u(x_i) * p(x_i) * decay(p_i) & \text{if } p(x_i) < 0.36 \\ h(x_i) = 1 - (u(x_i) * p(x_i) * decay(p_i)) & \text{if } p(x_i) >= 0.36 \end{cases} \tag{5.5}$$

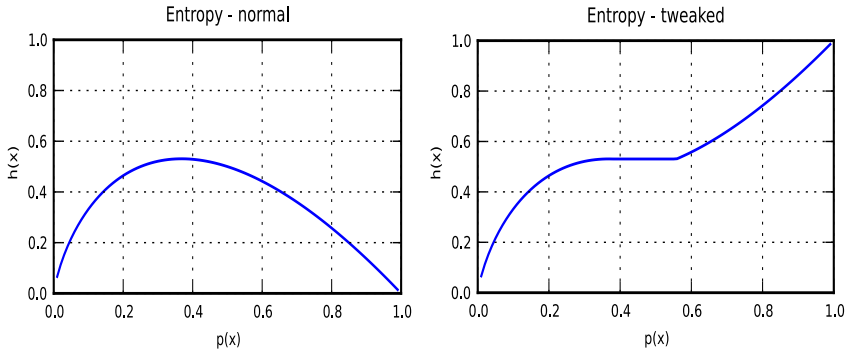


**Figura 5.3:** Entropy graphic before and after adjusts

A tabela abaixo 5.1 mostra um trecho do intervalo de probabilidade e o respectivo ponto neste intervalo a partir do qual o valor da entropia discreta decai ao invés de aumentar junto com a probabilidade $p(x)$. O valor de 0.36 presente na expressão 5.5 foi determinado de forma empírica através da observação desta inflexão na evolução da entropia discreta $h(x)$ em função do aumento de $p(x)$. Por conta desta inflexão em torno de $p(x)$, aplica-se $h(x) = 1 - (u(x_i) * p(x_i) * decay(p_i))$ para que $h(x)$ continue aumentando com $p(x)$ como forma de contorno ao efeito colateral indesejado observado quando $p(x)$ tende a valores muito altos conforme descrito acima.

| $p(x)$ | $h(x)$ |
|---|---|
| 0.363000 | 0.530691 |
| 0.364000 | 0.530708 |
| 0.365000 | 0.530722 |
| 0.366000 | 0.530731 |
| 0.367000 | 0.530736 |
| 0.368000 | *0.530738* |
| 0.369000 | 0.530735 |
| 0.370000 | 0.530729 |
| 0.371000 | 0.530719 |
| 0.372000 | 0.530705 |
| 0.373000 | 0.530687 |

**Tabela 5.1:** *Evolução de $p(x)$ e $h(x)$*

Este algoritmo fará o cálculo, a cada acesso à memória *cache*, da frequência de requisição daquele endereço. Também, será calculada a frequência como que o *cache set* que armazena aquele endereço é acessado. Para todas as operações de acesso ao *cache set* que tenha resultado em um *hit* ou *miss*, será incrementado número de vezes que aquele *cache set* foi acessado. Essas operações podem ser realizadas em paralelo ao acesso à *cache*. A partir do cálculo de frequência e do número de acessos, será estimada, em tempo de execução, a probabilidade dos endereços de memória que já foram acessados pelos programas. Será necessário manter um contador de acessos à memória *cache* para cada um dos *cache sets* e uma estrutura auxiliar que permitirá armazenar a frequência de cada um dos endereços já referenciados naquele *cache set*. Essa estrutura funciona com um histórico e pode ser implementada como um *lookup table* de acesso rápido. Desta forma, a qualquer instante, pode-se obter a Entropia de um endereço de memória que já tenha sido referenciado dentro de um *cache set*. Quando ocorrer uma nova referência a um endereço, este histórico será consultado para verificar se já existe algum valor de Entropia para o endereço referenciado. Caso o endereço já tenha sido referenciado, o histórico é incrementado atualizando as entradas daquele endereço. Caso contrário, uma nova entrada é criada no histórico para este novo endereço.

Durante a execução dos programas, sempre que houver a necessidade de substituir uma linha de dados para dar espaço a um outro endereço que acabou de ser solicitado, o algoritmo *Entropy* escolherá as linhas com o menor valor de Entropia discreta do *cache set* onde o novo endereço será armazenado. Outro ponto importante a ser mencionado sobre o funcionamento do *Entropy* é que, as linhas cujos valores de Entropia forem mais baixos ficarão mais próximas da posição de substituição dentro do *cache set*. Para tanto, dentro do simulador de *ISA SimpleScalar*, cada um dos *cache sets* será tratado como sendo um *heap* mínimo (*min heap*). A propriedade desta estrutura de

dados é tal que, para todo elemento $k$ do *heap*, tem-se:

$$\begin{aligned} &heap[k] <= heap[2*k] and \\ &heap[k] <= heap[2*k+1] k = \{1, \cdots, N\} \end{aligned} \tag{5.6}$$

A decisão de conferir ao *cache set* um aspecto e funcionamento de *min heap* decorre da necessidade de tornar rápido o acesso ao elemento com o menor valor de Entropia dentro do *cache set*. Durante o funcionamento do *Entropy*, a linha com menor Entropia é a que será escolhida para ser removida da *cache*. Portanto, encontrar rapidamente a linha com menor valor discreto é fundamental para o desempenho deste algoritmo. Ao invés de organizar o *cache set* como uma lista linear, organizá-lo como um *min heap* garante que o primeiro elemento da lista será sempre o de menor valor de Entropia.

A estimativa da probabilidade de cada uma das linhas de dados restringe-se ao *cache set* onde este bloco será armazenado. Ou seja, cada um dos *cache sets* possui as estruturas necessárias para armazenar informações sobre a frequência de requisição das linhas e a frequência de acesso aquele particular *cache set*. Isso é necessário, pois, a estimativa de Entropia é feita para cada um dos *cache sets* baseando-se, também, na frequência de acesso do *cache set*. Um fato decorrente disso é que, dois endereços distintos de memória com o mesmo número de referências, porém, armazenados em *cache sets* diferentes podem vir a apresentar valores de Entropia diferentes. Basta que o número de referências de cada um dos *cache sets* sejam diferentes entre si. A figura 5.4 ilustra a organização dos *cache sets* e a estrutura que mantém a frequência de acesso dos endereços armazenados neste *cache set*.

Assim, se um determinado *cache set* for intensamente acessado, sua estatística não influenciará a estimativa de probabilidade e a Entropia de endereços que estejam sendo armazenados em outros *cache sets*. A Entropia discreta de cada um dos endereços pertencente ao subconjunto do *address space* que está mapeado em cada *cache set*. A Entropia, probabilidade de acesso e número de referências retratam a taxa, distribuição e intensidade de referências ao *cache set* cujo endereço está associado. Isso deve-se ao fato de que a *cache* é composta de múltiplos *sets* e cada um destes *sets* tem um índice de associatividade. Essa associatividade faz com que, dentre todos os endereços do *address space*, cada subconjunto de endereços seja sempre mapeado para um determinado *cache set* da memória *cache*.

Desta forma, para uma memória *cache* definida, por exemplo, com os parâmetros $(N,n)$ $,N = sets$ $,n = associativity$, tem-se que $S_i\, i = 0,..,N$ *cache sets* e $X_i = (x_{1,i}, x_{2,i}, ..., x_{n,i})$ o espaço amostral do *cache set* $S_i$.

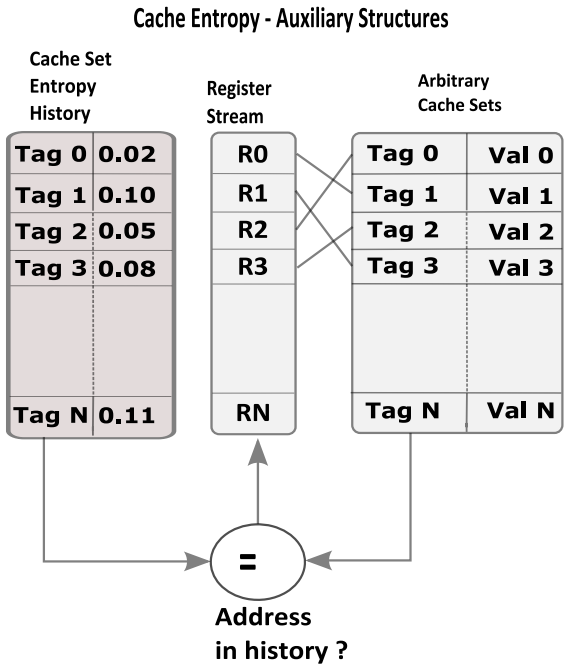


**Figura 5.4:** Cache Entropy sets e estruturas auxiliares

Em linhas gerais, o funcionamento do algoritmo *Entropy* está estruturado conforme ilustra o pseudo-código apresentado abaixo :

**Listagem 5.1:** Pseudo-código Entropy

```
if cache_access (cache_name, addr){
 cache->set.references++;
 addr->frequency++;
 if (!hit) {
    miss++;
    replace = pick_addr_to_replace (cache->set);
    copy_block_to_addr (cache->set, addr, replace);
 }
 else {
     hit++;
 }
 recalculate_entropy(cache->set);
 heapify_cache(cache->set);
}
```

A função (circuito) *recalculate_entropy (cache->set)* apontada acima indica a rotina que, a cada acesso ao *cache set*, resultando em um *hit* ou *miss*, recalcula a Entropia de todos os blocos do *cache set* que recebeu o acesso. Esta operação representa o maior custo de processamento no funcionamento do algoritmo *Entropy*. A função (circuito)

*heapify_cache(cache->set)* cuida de reorganizar a memória *cache* de forma a permitir que os novos valores de Entropia dos endereços armazenados naquele *cache set* sejam refletidos na prioridade de retenção da linha na *cache*. O circuito do *Entropy* apresentado na figura 5.10 pode ser implementado através do uso de uma *lookup table*, que mantém as Entropias, um *Register stream* que aponta para as linhas de *cache* e rastreiam qual será a próxima linha a ser substituída e um *bit* adicional de controle usando na *lookup table*. Este *bit* adicional faz a expiração das entradas da *lookup table*.

Esta constante atualização da Entropia dos endereços do *cache set* é necessária para que não ocorra o efeito colateral indesejado conhecido como *cache line pinning*. Se uma determinada linha de *cache* contém um endereço que é intensamente acessado durante um trecho da execução do código, esta linha migrará, rapidamente, para a posição mais alta do *cache set* ficando longe da posição de substituição. A função decaimento de Entropia descrita no início deste capítulo auxilia a evitar este fenômeno indesejado. Na secção 5.2 será apresentada a implementação da função decaimento em *hardware* através do uso de um comparador.

No caso do *Entropy*, a organização e as estruturas do *cache controller* serão bem próxima a utilizada em uma *cache* que implementa algoritmo *LRU*. Porém, serão tratadas de forma diferente. A substituição será feita na posição cujo valor de Entropia da linha sejam a mais baixas. A medida que um bloco de endereços for mais acessado, sua Entropia será recalculada e o bloco migrará para as posições mais protegidas da *cache*. Durante o funcionamento em regime, os endereços mais frequentemente acessados ficarão protegidos e longe da posição de substituição.

O funcionamento do *Entropy* difere-se do *LRU* na hora de tratar um *cache hit*. No *LRU*, quando um endereço já armazenado na *cache* é acessado novamente, este endereço é apontado para o final da lista dos endereços que serão substituídos. No caso do algoritmo *Entropy*, a Entropia deste endereço é recalculada e a linha pode migrar para uma posição mais alta e mais protegida. Se o novo valor de Entropia for maior que a Entropia de um outro bloco adjacente, a linha que acaba de ser acessada terá um prioridade maior de permanecer na *cache*. Caso seu novo valor de Entropia não supere o valor dos outros blocos no mesmo *cache set* esta linha não será promovida em relação às demais. Somente quando ela sofrer novas referências ela poderá superar as demais.

As linhas mais acessadas têm maior valor de Entropia e ficam armazenadas nas posições mais protegidas e distantes da região de substituição. Uma nova linha que seja carregada na *cache* pela primeira vez precisa ser intensamente referenciada para que seu valor de Entropia permita que ela alcance estas posições mais protegidas. Pode-

se dizer que o *Entropy* apresenta uma inércia maior do que o algoritmo *LRU* e *FIFO* na hora de proteger uma linha durante a execução do programa. Isso porque o simples acesso a um endereço não é suficiente para colocá-lo na posição mais protegida da *cache* como acontece com os algoritmos *FIFO* e *LRU*. Em contrapartida, o algoritmo *Entropy* apresenta um inércia maior na operação de substituição das linhas de *cache* quando comparado aos algoritmos *LRU* e *FIFO*. Este comportamento do *Entropy* acaba por conferir alguma similaridade com algoritmos adaptativos utilizados em substituição de páginas de memória, mesmo sem ter sido concebido para apresentar tal adaptabilidade. Pode-se ilustrar essa propriedade com o seguinte exemplo hipotético.

Suponha que um determinado programa apresente um padrão de acesso puramente sequencial de instruções e dados. Porém, esse acesso se dá de forma que não ocorra localidade espacial no acesso aos endereços. Ainda, tal acesso incide sobre uma grande quantidade de endereços. Como no primeiro acesso efetuado o endereço carregado na primeira posição da *cache*, nas referencias subsequentes outros endereços serão solicitados. Não haverá recorrência de acesso ao primeiro endereço que seja suficiente para ele migrar novamente para a posição mais alta e protegida da *cache*. Então, toda referência fará com que um novo endereço seja inserido na primeira posição e logo em seguida removido para dar lugar ao endereço subsequente. Esse acesso sequencial causará sucessivos *misses* compulsórios. Esta situação pode ser verificada em programas que operam estruturas como árvores binárias e *heaps* que armazenam dados do tipo *LOB - Large Objects*. Estas estruturas tem coesão lógica, mas, os dados são armazenados de forma dispersa no espaço de endereçamento, caracterizando a situação descrita acima.

Neste particular exemplo, o comportamento do *Entropy* será muito semelhante ao comportamento do algoritmo *MRU - Most Recently Used*. Em contrapartida, tanto o *FIFO* quanto o *LRU* apresentarão uma maior ineficiência em substituir as linhas de *cache*. Isso porque tais algoritmos inserem as linhas novas na posição mais distante da região de substituição. Como a linha que acabou de ser carregada não será novamente acessada nas próximas instruções, mantê-las na *cache* vai causar desperdício de endereços. Programas contendo *loops* de execução extremamente longos e operando estruturas dinâmicas como listas ligadas e matrizes esparsas podem apresentar este comportamento particular. Da mesma forma, longos *file scans* podem apresentar igual comportamento. Em contraposição, o algoritmo *Entropy* vai descartar as linhas da mesma forma que o *MRU* faz e não desperdiçará endereços da forma com o *LRU* e *FIFO* fazem para este tipo de padrão de acesso.

Por outro lado, o oposto pode ocorrer em uma situação onde o acesso aos endere-

ços se dê de forma aleatória, porém, incidindo em um conjunto pequeno de instruções. O algoritmo *Entropy*, e sua maior inércia em promover as linhas mais recentes às posições mais protegidas, poderá apresentar uma taxa de *miss rate* mais alta que o *LRU* ou o *FIFO*. Isso poderá ocorrer até que o número de referências aos endereços mais úteis deste *cache set* tenham sido em número suficiente de forma que a Entropia os coloque em uma posição mais protegida da *cache*. Nesta situação, o algoritmo *Entropy* pode penalizar o desempenho do programa quando comparado ao *LRU* e *FIFO*.

O diagrama 5.5 abaixo ilustra a dinâmica de funcionamento do *Entropy* para uma sequência aleatória e hipotética de endereço $\{A,B,A,A,B,C,D,E,B,B\}$.

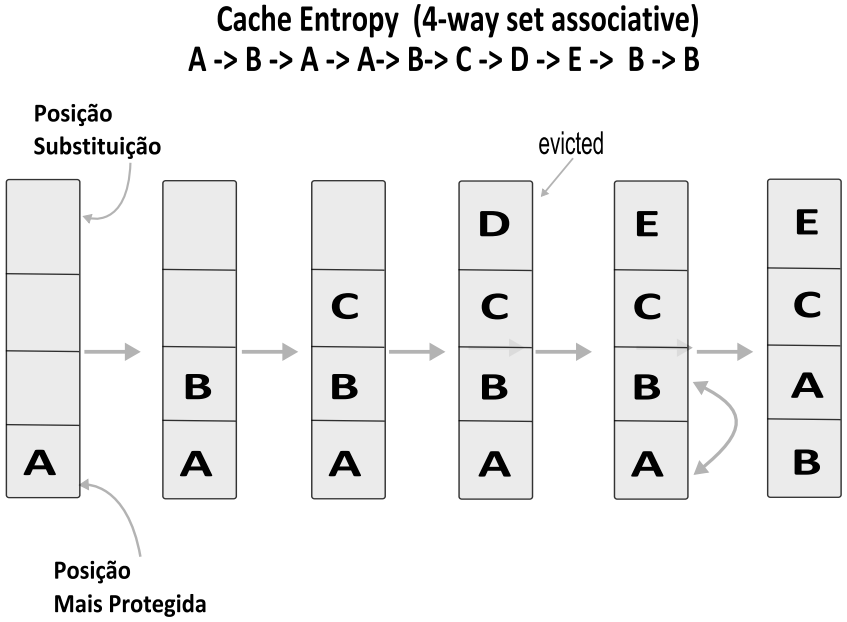


**Figura 5.5:** Funcionamento do Entropy

A mesma sequência de referência aos endereços é tratada de forma diferente pelo algoritmo *LRU*, conforme mostra a figura 5.6 abaixo:

Da mesma forma que no caso do algoritmo *LRU*, existem determinados padrões de acesso à memória que podem configurar casos de uso mais ou menos favoráveis ao funcionamento do *Entropy*. Padrões de acesso caracterizados por repetições de subsequências de endereços tendem a ser mais favoráveis ao *Entropy*. A figura 5.7 abaixo ilustra um exemplo deste padrão de acesso. Neste exemplo, a subsequência $\{AB\}$ se repete continuamente, aumentando a probabilidade destes endereços e retendo-os na *cache*. Após a ocorrência de referenciamento a alguns outros endereços a subsequência ocorre novamente, gerando *hits* na *cache*. Esta padrão de recorrência é favorável ao *Entropy*.

Porém, se o padrão de acesso apresenta uma incidência de acessos a determinados endereços durante um trecho de execução e, após estes acessos iniciais, outras

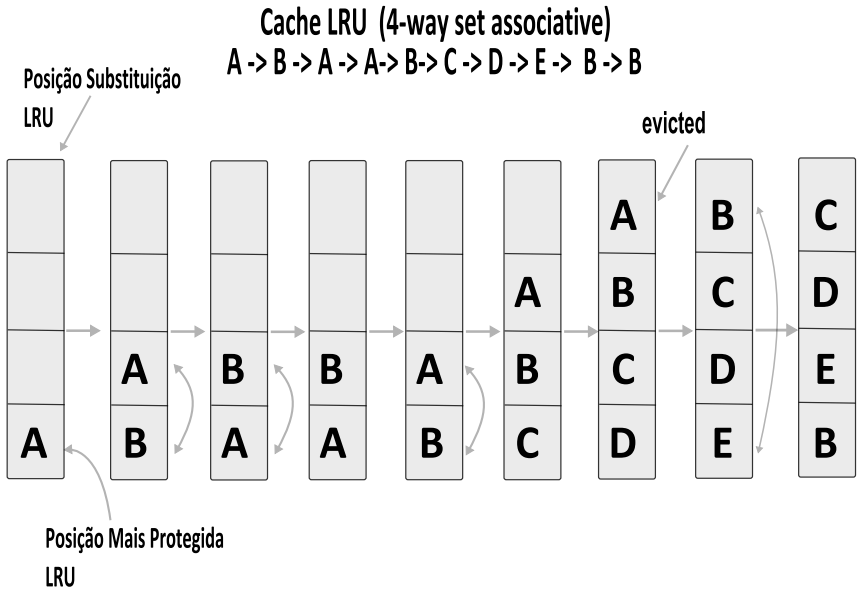


**Figura 5.6:** Funcionamento do LRU

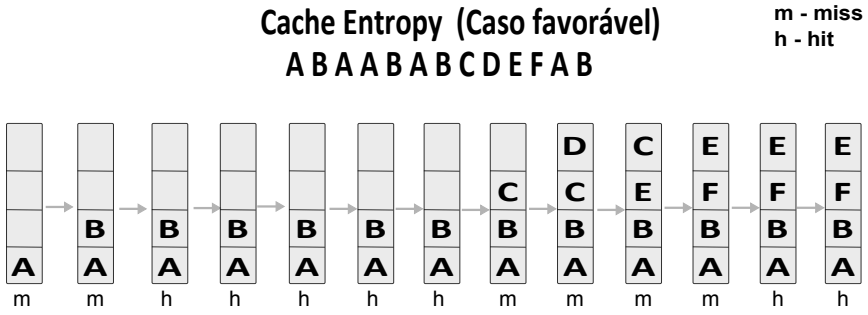


**Figura 5.7:** Caso favorável ao Entropy

subsequências de endereços que não contenham os endereços iniciais passam a serem referenciadas, há uma tendência a ocorrer maior número de *misses* na *cache* quanto comparado com o algoritmo tradicional *LRU*. Isso deve-se à maior inércia do *Entropy* em demover as linhas mais antigas da *cache* e liberar endereços para reter as linhas mais recentemente referenciadas. Este comportamente é ilustrado na figura 5.8 apresentada abaixo.

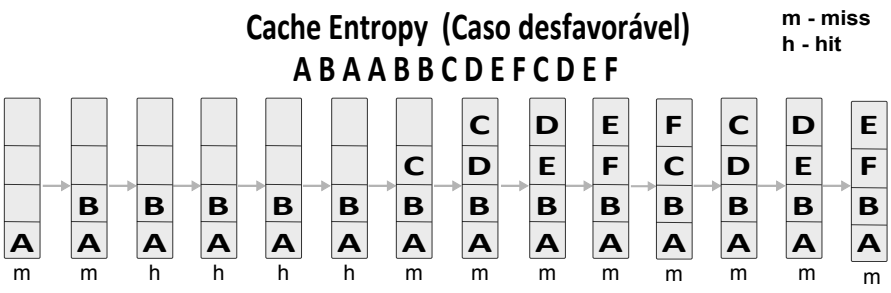


**Figura 5.8:** Caso desfavorável ao Entropy

A seguir, serão apresentados os aspectos da implementação em *hardware* do algoritmo *Entropy*.

## 5.2 Implementação em *Hardware*

No *design* de um processador, leva-se em consideração o custo de implementação do algoritmo de substituição de linhas de *cache*. As considerações giram em torno da quantidade de circuitos lógicos e área (*storage*) utilizados para seu funcionamento. A implementação em *hardware* deve conferir um nível de complexidade computacional adequado para garantir que sua implementação seja viável. Os aspectos de implementação do *Entropy* serão discutidos nesta secção.

A heurística de substituição do algoritmo *Entropy* apoia-se na estimativa de probabilidade de uma linha de *cache* que contém dados. Para isso, a quantidade de referências feitas a um determinado endereço e o total de acessos ao *cache set* em que aquela linha é armazenada são computados no cálculo da Entropia dos endereços. A Entropia de cada endereço tem natureza discreta e dois ou mais endereços podem apresentar o mesmo valor de Entropia para uma mesma memória *cache*. Quando um endereço entra pela primeira vez na *cache* é computado seu valor inicial de Entropia. Se ele for ejetado da *cache* e referenciado novamente, quando entrar novamente na *cache* deverá levar em consideração o valor inicial de Entropia, pois já esteve na *cache*. Isso fará com que o bloco seja inserido em uma posição mais favorável na *cache* desde que os acessos que tenha recebido sejam recentes. Caso contrário, se um bloco possui muitos acessos ocorridos em um trecho antigo da execução do programa, será inserido nas mesmas posições que os blocos quando recebem sua primeira referência.

Na implementação em *hardware* será utilizada uma estrutura adicional de memória para armazenar os valores de Entropia dos blocos que já estiveram carregados na *cache*. Esta estrutura é uma *lookup table* que permite rapidamente identificar se o endereço já foi carregado na *cache* anteriormente e qual foi o último valor de Entropia. Esta nova *lookup table* será chamada de *Entropy lookup table*. Propõe-se uma *Entropy lookup table* por *cache set* e o número de entradas deve ser tal que permita armazenar um histórico de acessos de tamanho relevante e que permita estimar as chances de reuso dos endereços daquele *cache set*. Em *hardware* é proibitivo estimar a probabilidade $p(x_i)$ e Entropia discreta dos endereços utilizando um número de referências que varia como se faz na implementação do *Entropy* usando o simulador *SimpleScalar*. Isso porque o histórico de acessos ao *cache set*, quando implementado em *hardware*, precisa utilizar uma estrutura de tamanho. Esse tamanho deve ser concebido *a priori* e em tempo de projeto. Por isso, os valores de probabilidade dos endereços serão

computados usando uma estratégia diferente da que foi adotada na implementação no simulador de *ISA*.

Para calcular a probabilidade do endereço e sua Entropia discreta, faz-se a razão entre o número de referências ao endereço pelo número de referências ao *cache set* onde o endereço está armazenado. Conforme dito anteriormente, como em *hardware* este cálculo é proibitivo, o denominador desta razão será o número de entradas que a *Entropy lookup table* possui. O número de referências aos endereços será armazenado através de um conjunto adicional de *bits*. Este conjunto de *bits* adicionais será chamado de *Entropy bit stream*. Cada entrada na *Entropy lookup table* será composta de um campo *tag*, que identifica o bloco de dados, e um campo com o *Entropy bit stream* que guarda o número de referências àquele endereço. A cada nova referência, verifica-se se existe uma entrada na *Entropy lookup table* para o endereço acessado e, em caso afirmativo, o *Entropy bit stream* é incrementado funcionando como um contador. Desta forma, em *hardware* a probabilidade será dada pela razão entre o valor do contador *Entropy bit stream* e o número de entradas na *Entropy lookup table*.

Para controlar o espaço necessário para implementar a *Entropy lookup table*, o número de *bits* do *Entropy bit stream* deve ser tal que, a razão entre o número de referências ao endereço contido em uma entrada da *Entropy lookup table* divido pelo número de entradas desta tabela atinja, no máximo, 25% (vinte e cinco por cento). Ou seja, a implementação em *hardware* requer que as probabilidades das linhas sejam valores aproximados. Caso contrário, o custo computacional de se manter estas probabilidades e a Entropia, para todo acesso ao *cache set*, inviabilizaria a sua implementação no circuito. Assumindo essa aproximação, pode-se utilizar um número menor de *bits* para contabilizar os acessos aos endereços por *Entropy lookup table*.

Da mesma forma, o número de entradas que a *Entropy lookup table* armazena será proporcional à associatividade do *cache set*. Para um *cache set* de associatividade $n$, o número de entradas na *Entropy lookup table* será de $n^2$. Exemplificando, se o *cache set* em associatividade *8-way* o número de entradas na *Entropy lookup table* daquele *cache set* será de $8 * 8 = 64$. Para manter a razão máxima de 25% entre a acessos a endereços e acesso ao *cache set* , cada *Entropy bit stream* precisará de 4 *bits*. Assim, para uma *cache* com associatividade *8-way*, *block size* de *64-bytes* e área total de 2-MB (*dois megabytes*), serão necessários 4096 *cache sets*. Além dos *Entropy bit streams* e da *lookup tables*, será necessário utilizar um outro *bit* para indicar que o endereço, enquanto armazenado na *Entropy lookup table*, foi referenciado e está ativo. Este *bit* é denominado *Entropy lookup bit*.

Quando uma referência a um endereço de memória resultar em um *miss*, o *cache*

*controller* atualizará o *Entropy lookup table* para refletir o valor da Entropia da linha referenciada. Se a referência ao endereço resultar em um *hit*, o *cache controller* também deverá atualizar o valor da Entropia da linha que está no *Entropy lookup table*. Por conta da simplificação que foi adotada ao fixar-se o número de *Entropy bit stream* e o número de entradas no *Entropy lookup table*, o circuito que faz o cálculo de Entropia das linhas precisa apenas atualizar os *bits* do *Entropy bit stream* para contabilizar os acessos. Assim, um endereço que sofre o primeiro acesso, para um *Entropy bit stream* de 4 *bits* apresentará o valor 0001. Para um segundo acesso, apresentará o valor 0010, e assim sucessivamente até atingir 1111.

A figura 5.9 ilustra sucintamente esta relação entre a *Entropy lookup table* e os *bits* de controle que permitem contabilizar a quantidade de referências aos endereços e se estão em atividade dentro da *lookup table*.

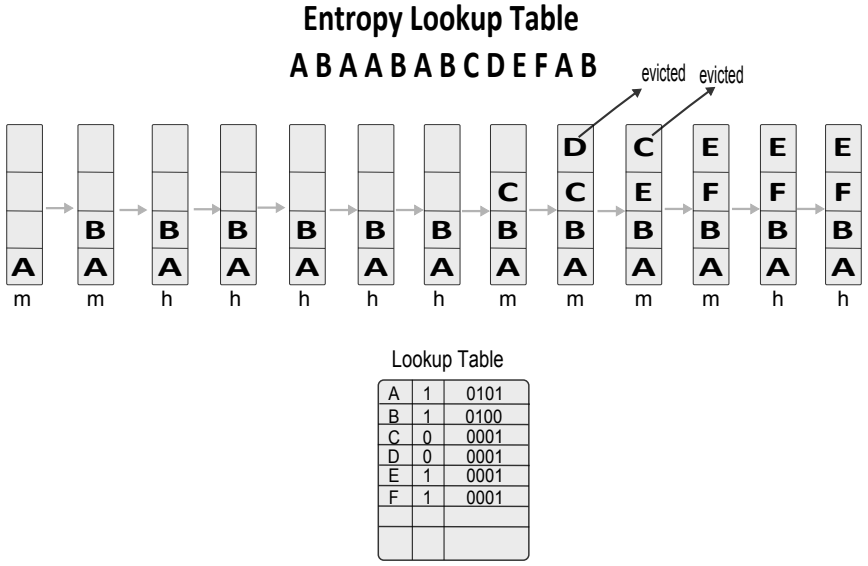


**Figura 5.9:** Exemplo de atualização da Entropy lookup table durante funcionamento

O *overhead* de área será dado pela soma do número de *bits* dos *tags*, dos contadores *Entropy bit streams* e *Entropy lookup bits* multiplicado pelo número de *cache sets* que a *cache* possui. Assumindo que o campo de *tag* do endereço utilize 20 *bits*, como faz o processador *AMD Opteron* (PATTERSON; HENNESSEY, 2007), será necessário uma área adicional de $A = 4096 * (20 + 4 + 1) = 12.5KB$. Ou seja, em uma *cache 8-way associative*, *64-bytes* de *block size* e *2-MB* de área total, a estrutura de memória adicional requerida para o funcionamento do *Entropy* é de $0.61\%$. A tabela 5.2 abaixo sumariza os requisitos de *storage* para o funcionamento do algoritmo *Entropy*.

Conforme mencionado anteriormente, é necessário elaborar um mecanismo que impeça que linhas intensamente acessadas em um trecho antigo do programa fiquem indefinidamente confinadas na *cache*. Esta é a tarefa da função decaimento de Entro-

| Parameter | Value |
|---|---|
| Cache defs | 2-MB; 8-way; 64B; |
| Cache Sets | 4096 |
| Lookup Table | $8^2 = 64$ entries |
| Lookup Table tag | 20-bits |
| Lookup bit stream | 4-bits |
| Lookup bit | 1-bit |
| Lookup Table tag | 20-bits |
| Register Stream | 4096 registers |

**Tabela 5.2:** *Entropy hardware overhead for 2-MB 8-way cache*

pia. Isso evita que estes endereços fiquem confinados na *cache* quando não forem mais acessados. Para implementar esta função decaimento em *hardware*, o *cache controller* descarta completamente o valor de Entropia da linha de *cache* quando esta linha for removida da *Entropy lookup table*. A combinação do circuito comparador *FIFO* e do *Entropy lookup bit* fazem o papel desta função decaimento. Para as linhas que estão inativas e que não são apontadas pelo circuito *FIFO* como as próximas candidatas a deixarem a *Entropy lookup table*, se o *Entropy lookup bit* não estiver ativo, o *cache controller* decrementa um *bit* do *Entropy bit stream* daquela entrada, diminuindo sua probabilidade e Entropia.

O espaço nesta tabela *Entropy lookup table* precisa ser gerenciado. Para decidir qual entrada deverá ser removida da *Entropy lookup table* utiliza-se o *Entropy lookup bit* em combinação com uma política *FIFO*. O *cache controller* deve escolher a entrada mais antiga na *Entropy lookup table* e que não tenha o *Entropy lookup bit* ativado, indicando que aquela entrada não foi referenciada recentemente. Esse *bit* adicional evita que as entradas fiquem armazenadas indefinidamente na *Entropy lookup table*.

Cada vez que uma linha é removida da *cache*, a entrada correspondente a ela é removida da *Entropy lookup table*. Para todas as outras entradas desta *Entropy lookup table* o *Entropy lookup bit* é desativado. A cada nova referência aos endereços que estiverem na *Entropy lookup table*, o *cache controller* ativa novamente o *Entropy lookup bit* daquela entrada para indicar que ela está em atividade e foi referenciada. Além de ativar o *Entropy lookup bit*, o *cache controller* incrementa o *Entropy bit stream* daquela entrada para que esta nova referência seja refletida no valor de probabilidade e Entropia desta entrada. Se a linha que foi removida for novamente referenciada, ela entrará na *cache* e na *Entropy lookup table* com um valor baixo de Entropia e probabilidade. É preciso que este endereço seja frequentemente referenciado para que permaneça na *cache*, pois estes acessos é que provocam o aumento de sua Entropia e probabilidade.

Uma vez que o *cache controller* possui o mecanismo para controlar a probabilidade de reutização de linhas de *cache*, faz-se necessário controlar quais linhas da

memória *cache* são as melhores candidatas a substituição. Para isso, o *cache controller* utiliza um *Register stream* igual ao utilizados pelo algoritmo *FIFO* e ilustrado na figura 3.2 para apontar as entradas de cada *cache set* da memória *cache*. Ao invés de utilizar a ordem de entrada das linhas na *cache* como faz o algoritmo *FIFO*, um circuito lógico compara os valores de Entropia para os *tags* armazenados na *Entropy lookup table* com os *tags* que estão na *cache* e reordena o *Register stream* para que os registrados apontem para as linhas candidatas à substituição em cada um dos *cache sets*. Os valores de Entropia mantidos no histórico são utilizados como *input* para o circuito comparador. Este circuito é responsável por atualizar o *Register stream*. Para garantir a rapidez no uso desta estrutura, o *Register stream* pode ser manipulado da mesma forma que o *min heap* descrito na implementação do simulador de *ISA*. Isso garante um tempo computacional de ordem *O(logN)*, onde *N* é a associatividade do *cache set*.

Abaixo, figura 5.10 ilustra o diagrama de blocos e a organização dos elementos de *hardware* necessários ao funcionamento do algoritmo *Entropy*.

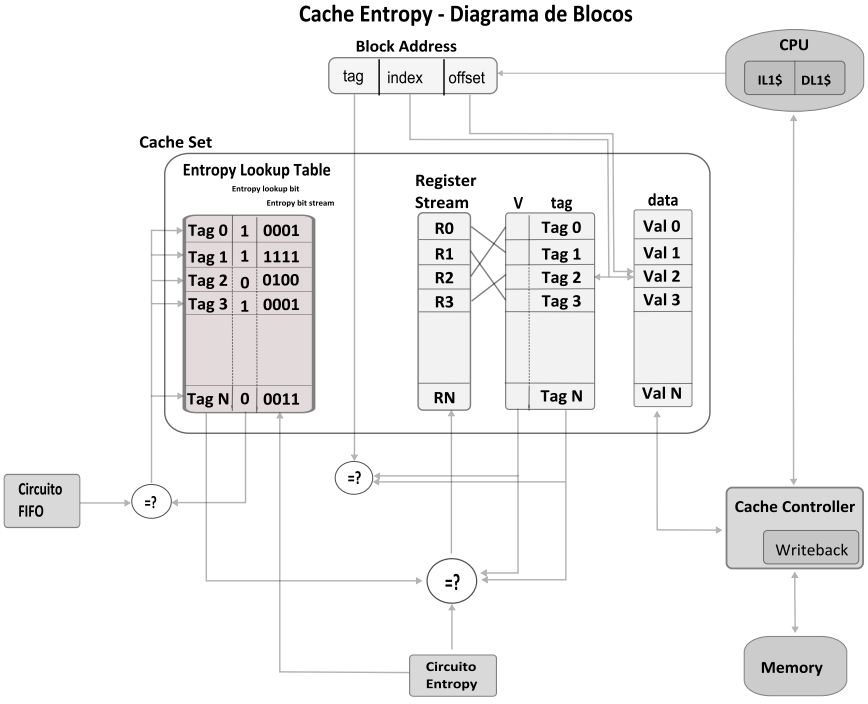


**Figura 5.10:** Diagrama de Blocos de Cache com Entropy

# 6 Simulador *SimpleScalar*

O *SimpleScalar* (AUSTIN, 1999) é um simulador de arquitetura de processador. Sua finalidade é a simulação funcional (arquitetura) e de desempenho (microarquitetura). Por ser um simulador orientado à execução de instruções, programas verdadeiros que tenham sido compilados para o *ISA - Instruction Set Architecture* do *SimpleScalar* podem ser executados completamente sobre o simulador. A execução das instruções pode ser feita *in-order* ou *out-of-order*. Atualmente, o *SimpleScalar* pode executar programas compilados para *ISA Alpha*, *PISA - Portable Instruction Set Architecture*, *ARM* e *x86*. Para gerar os códigos objeto para os diversos *ISA* suportados existem compiladores distribuídos junto com o código fonte do simulador. Como trata-se de um simulador de instruções, qualquer programa que tenha sido compilado para algum destes *ISA* pode ser executado sobre o simulador *SimpleScalar*.

A excelente modularidade do *SimpleScalar* permite grande flexibilidade na modelagem de processadores. A execução dos programas e a simulação das *caches* são feitas com relativa rapidez. Sua acurácia na execução dos programas em processadores modelados torna-o uma excelente escolha para a avaliação deste novo algoritmo proposto. Por ser totalmente modular e escrito em linguagem de programação *ANSI C*, a introdução de uma nova política de gerenciamento e substituição de linhas de *cache* é trivial. A figura abaixo 6.1 ilustra a modularidade do simulador de *ISA SimpleScalar*. Pode-se visualizar o módulo cujo código foi alterado para que o funcionamento do *Entropy* fosse programado no simulador. Apenas o módulo referente à memória *cache* foi alterado de forma a prover a nova política de substituição de linhas de *cache* inspirada na Entropia da Informação.

Nativamente, o *SimpleScalar* permite alternar entre múltiplas heurísticas de substituição de linhas de *cache* e escolher dentre as opções disponíveis para cada um dos níveis de *cache*. Pode-se utilizar as definições no arquivo de configuração do simulador ou mesmo informar os parâmetros de cada um dos níveis de *cache* ao se iniciar a execução com o objetivo modelar os *caches*. Neste estudo será utilizada a versão *3.0d* do simulador *SimpleScalar*, disponível gratuitamente através do *web site* da equipe que o desenvolve e mantém o código do *SimpleScalar*.

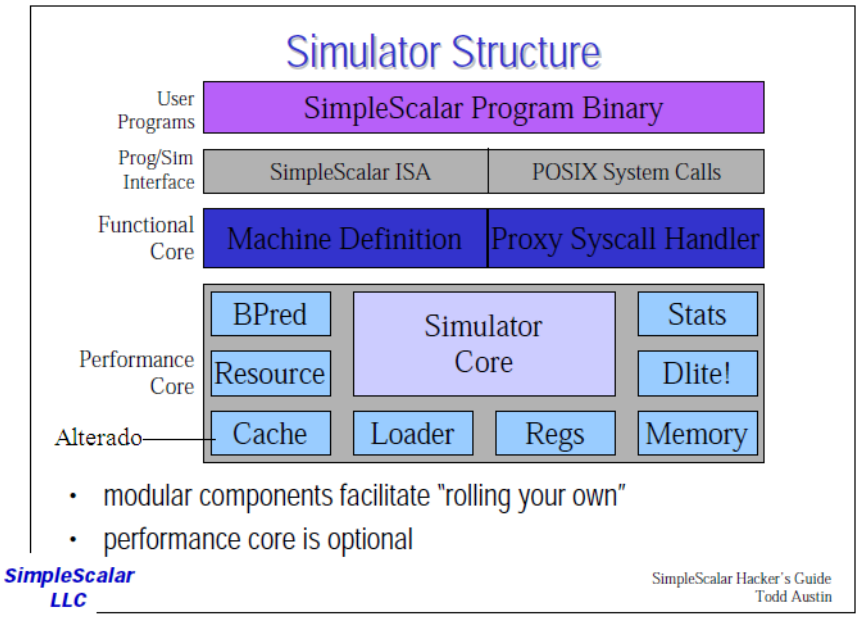


**Figura 6.1:** Estrutura do Simulador SimpleScalar

Apesar de possuir uma versão comercial, neste trabalho, será utilizada a versão acadêmica do *SimpleScalar* destinada à atividade de pesquisa sem fins comerciais. Na versão utilizada, inicialmente, tem-se à disposição três heurísticas de substituição de linhas de *cache* sendo elas a *Random*, *FIFO* e *LRU*. Para efeito de comparação com a heurística *Entropy* será utilizada a heurística *LRU* nas simulações. O simulador ainda permite que sejam alterados grande parte dos parâmetros que definem a memória *cache* como tamanho do bloco, associatividade e penalidades por *cache miss* entre outras grandezas. Para isolar os impactos que a modificação do algoritmo de substituição de linhas de *cache* tem sobre o desempenho dos programas, pode-se modificar os parâmetros de apenas um dos níveis de *cache*. Desta forma, pode-se isolar o impacto em um único nível de *cache* para facilitar a interpretação de resultados.

Um aspecto importante do funcionamento do *SimpleScalar* é que suas estruturas de dados internas representam a memória *cache* e seus de múltiplos *cache sets* com grande similaridade ao encontrado nos processadores modernos. A geometria da *cache*, associatividade dos *cache sets*, penalidades, *TLBs* entre outros parâmetros estão detalhadamente implementados neste simulador de *ISA*. Assim, pode-se considerar que o *SimpleScalar* implementa um modelo real de memória *cache*, o que torna as simulações bastante representativas. Um outro aspecto que influenciou a escolha do simulador *SimpleScalar* foi a vasta adesão e aceitação deste simulador na comunidade científica. Existe um vasto repertório de trabalhos científicos que basearam suas pesquisas neste simulador (AUSTIN, 1999).

Abaixo, a figura 6.2 ilustra a estrutura de *cache* implementada pelo *SimpleScalar*.

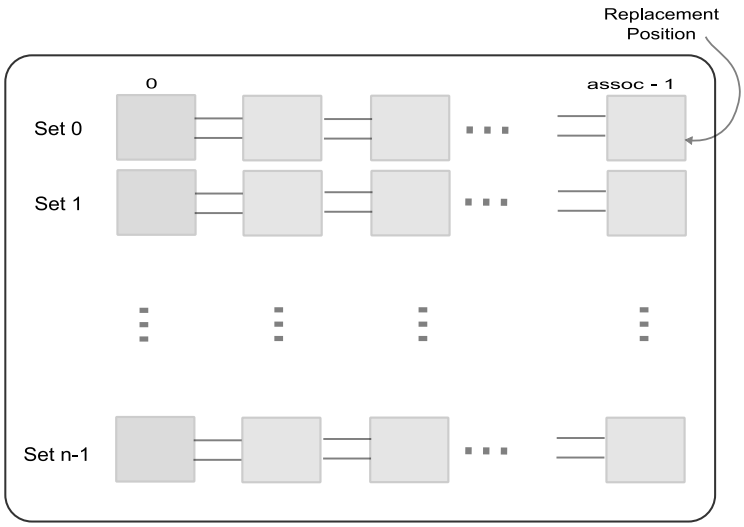


**Figura 6.2:** Cache SimpleScalar

Para definir uma memória *cache* de primeiro ou segundo nível, deve-se levar em consideração algumas grandezas de sua geometria, como segue:

$$Cache = \{N * bs * n\}\ N = \{sets\}\ bs = \{blocksize\}\ n = \{associativity\}$$

O simulador *SimpleScalar* oferece uma vasta gama de parâmetros (item 6.2) que podem ser utilizados para controlar o comportamento dos diversos módulos do simulador. Em especial, neste estudo serão modificados parâmetros que modelam a *cache* de segundo nível. Da mesma forma, o parâmetro que define o algoritmo de substituição de linha será ajustado na memória *cache* de segundo nível. A definição dos valores de latências de memória e penalidade de acesso à memória serão extraídos de arquiteturas de referência como *IBM Power*, *Fujitsu Sparc64* entre outros processadores de mercado de forma a refletir arquiteturas modernas no *SimpleScalar*.

No exemplo 6.1, a *cache* de instruções de primeiro nível tem tamanho $il1 = \{128 * 64 * 1\} = 8KB$ e a política de substituição de linhas configurada para este nível, neste exemplo, é o *LRU*. Os parâmetros que definem o tamanho da *cache*, conforme indicado em 6.2, são o número de *cache sets* (*128*), *block size* (*64*) e *associativity* (*1*). Da mesma forma, a *cache* de dados de primeiro nível tem $dl1 = \{256 * 32 * 1\} = 8KB$ e política de substituição de linha para esta *cache* é o *LRU*.

No caso das definições da *cache* de segundo nível apresentadas em 6.1, mostra-se uma *cache* unificada para instruções e dados. O tamanho é de $ul2 = \{1024 * 64 * 8\} = 512KB$. Ainda neste exemplo, a *cache* é *8-way associative* com política de substituição *Entropy*. Nota-se que na definição das *caches* de primeiro nível a política de substituição foi configurada para utilizar o algoritmo *LRU* enquanto que na *cache* de segundo nível foi definido o algoritmo *Entropy* para realizar a substituição de linhas. Esta fle-

xibilidade na parametrização permite realizar modificações no nível de *cache* que se deseja inspecionar com maior detalhe. Portanto, para acionar o algoritmo *Entropy* no módulo de *cache* do *SimpleScalar*, basta especificar o parâmetro *-e* na definição da *cache* e a substituição de linhas será efetuada pelo *Entropy*.

**Listagem 6.1:** Definição um *cache* L2 com *Entropy*

```
-cache:il1 il1:128:64:1:l
-cache:dl1 dl1:256:32:1:l
-cache:dl2 ul2:1024:64:8:e
```

Fixando os valores para os principais parâmetros de modelagem da *cache* e variando apenas os algoritmos de substituição de página será possível isolar o impacto do algoritmo no desempenho dos programas. Uma vez que o algoritmo *Entropy* não altera a geometria das *caches*, a decisão de manter os mesmos valores de latência e penalidades é aceitável do ponto de vista de arquitetura, pois, independente do algoritmo, a geometria da *cache* será a mesma. Esse critério tem como objetivo estabelecer um patamar de referência (*baseline*) na *cache* e concentrar o estudo nas diferenças de desempenho que surgem da variação do algoritmo de substituição de linhas de *cache*. No capítulo 5, foi proposto e detalhado um circuito para o *Entropy* com o intuito de manter a mesma ordem de complexidade e latência do algoritmo *LRU* justificando sua implementação em *hardware*. Sob essa consideração, é aceitável considerar a geometria das *caches* idênticas mesmo quando se alterna entre algoritmo *LRU* e *Entropy*.

**Listagem 6.2:** Parâmetros de definição de *cache* do *SimpleScalar*

```
<name>:<nsets>:<bsize>:<assoc>:<repl>
<name> - nome do cache sendo definido
<nsets> - numero de sets no cache
<bsize> - block size do cache
<assoc> - associatividade do cache
<repl> - Algoritmo ('e'-Entropy, 'l'-LRU, 'f'-FIFO, 'r'-random)
Exemplo:
-cache:dl1 dl1:4096:32:1:l -dtlb dtlb:128:4096:32:r

Define cache L1$ segregado L2$ unificado:
-cache:il1 il1:128:64:1:l
-cache:dl1 dl1:256:32:1:l
-cache:il2 dl2 -cache:dl2 ul2:1024:64:2:l

Ou uma hierarquia unificada com il1 apontando para dl1:
```

```
-cache:il1 dl1 -cache:dl1 ul1:256:32:1:l
-cache:dl2 ul2:1024:64:2:l
```

Na secção 1.3 foram apresentados os programas de *benchmark* escolhidos para a simulação no *SimpleScalar* do novo algoritmo proposto. Os resultados obtidos durante a simulação do *Entropy* serão comparados com os resultados do algoritmo *LRU* e apresentados no capítulo 7.

# 7 Metodologia Detalhada e Resultados

Pode-se dividir este trabalho em duas fases. Na fase inicial foi elaborada a pesquisa que subsidiou o desenvolvimento do algoritmo *Entropy* e a escolha do simulador de arquitetura *SimpleScalar*. No simulador *SimpleScalar* foi modelado o processador e a configuração da memória *cache* e os programas do *benchmark SPEC CPU2000* foram escolhidos para as simulações. Esta fase inicial teve, também, como objetivo a verificação funcional do algoritmo proposto. Para tanto, o código do simulador foi instrumentado para produzir as evidências (*tracing*) que subsidiassem esta verificação. Na fase final deste trabalho, procurou-se melhorar o desempenho do algoritmo *Entropy* através da implementação de uma função de decaimento da entropia que permite reproduzir o efeito de envelhecimento (*aging*) de linhas que estão armazenadas na *cache* e não são mais referenciadas. Esta função tem papel fundamental no funcionamento do algoritmo, pois evita que as linhas fiquem confinadas na *cache*. Neste capítulo serão apresentados os resultados obtidos nas simulações com o algoritmo *Entropy* utilizando o simulador de *ISA SimpleScalar*.

Para efeito de modelagem, procurou-se seguir os processadores *multi-cores* reais que, geralmente, empregam baixa associatividade no primeiro nível e alta associatividade no segundo nível da hierarquia de memória. Esta decisão justifica-se, pois manter as *caches* de primeiro nível com associatividade baixa garante uma latência de acesso menor, aumentando o desempenho do processador. Por outro lado, a quantidade de *capacity misses* pode aumentar devido à baixa associatividade. Isso decorre do fato de que um *cache set* com baixa associatividade possui um número menor de *cache blocks*. Isso faz com que um número maior de endereços de memória principal sejam mapeados para os mesmos *cache blocks* do mesmo *cache set*, aumentando a concorrência por estes endereços no *cache set*.

Para contornar o potencial aumento na taxa de *capacity misses* decorrente desta baixa associatividade, foi modelado uma *cache* de segundo nível com alta associatividade. A alta associatividade vai conferir a este nível da memória uma capacidade maior de armazenamento de dados durante a execução dos programas. Desta forma, a

*cache* modelada nos testes leva em consideração a baixa latência no primeiro nível e maior capacidade e menor taxa de *conflict misses* no segundo nível da memória.

No entanto, o aumento da associatividade requer uma política de substituição de linhas mais eficaz. Uma vez que se tem um número maior de *cache blocks* em cada *cache set*, a tarefa de gerenciar estes *cache blocks* torna-se mais relevante. Deste fato decorre a decisão de simular e avaliar o algoritmo *Entropy* somente no segundo nível de memória *cache*, pois este nível apresenta alta associatividade. Os impactos positivos decorrentes da política de substituição de linhas de *cache* são mais significativos em *caches* com alta associatividade. Este critério subsidiou a decisão de manter o algoritmo *LRU* nas *caches* de instrução e de dados de primeiro nível e variar entre *LRU* e *Entropy* no segundo nível da hierarquia de memória.

Os modelos desenvolvidos no simulador *SimpleScalar* geraram os arquivos de saída contendo uma compilação extensa de estatísticas coletadas durante a execução dos programas do *benchmark*. A partir destes arquivos foram elaborados os sumários apresentados neste capítulo e que contém os principais indicadores de desempenho do processador e da *cache* para cada um dos programas simulados. Conforme mencionado anteriormente, estas estatísticas compreendem somente a execução das instruções contidas no código dos programas do *benchmark*. Isso decorre do fato de que o *SimpleScalar* é um simulador funcional de *ISA* e dispensa a execução de uma imagem completa de sistema operacional sobre a arquitetura de processador modelada. Ou seja, os indicadores de desempenho contemplam apenas as estatísticas de uso de recursos por parte dos programas do *benchmark* e não contêm nenhum outro tipo de sobrecarga gerada por um sistema operacional.

Dentre os indicadores de desempenho que o *SimpleScalar* produz destacam-se *IPC - Instruction per Cycles*, *CPI - Cycles per Instructions*, *DL1 replacements*, *DL2 replacements*, *DL1 misses* e *DL2 misses*. Para efeito de comparação entre *Entropy* e *LRU* e para a análise dos resultados será utilizada a taxa de *misses* observada no segundo nível da *cache*. Esta taxa corresponde à razão entre o número absoluto de eventos de *miss* pelo número total de acessos à *cache*. Esta taxa é uma boa medida da eficácia e do desempenho do algoritmo de substituição de linhas, pois indica claramente a proporção de vezes que um endereço não foi encontrado na *cache* provocando uma parada no *pipeline*.

## 7.1 Benchmarks SPEC CPU2000

Os programas que compõem o *SPEC CPU2000* exercitam o processador e a arquitetura de memória. Tais programas são modificados pelo *SPEC - Standard Performance Evaluation Corporation* para que não utilizem operações em disco e rede de comunicação. Desta forma, a sobrecarga que tais programas geram fica isolada no sistema processador-memória. Os programas deste *benchmark* podem ser recompilados para a arquitetura em que serão executados. Desta forma, pode-se empregar otimizações em tempo de compilação. Por esta razão, pode-se utilizar os programas do *SPEC* para fazer avaliações de compiladores e técnicas de otimização empregados por estes *software*. Isso permite uma avaliação mais extensa do conjunto compilador-processador-memória. Porém, neste estudo, os programas não sofreram nenhum tipo de ajuste ou alteração e foram utilizados conforme disponibilizados pelo *SPEC*.

O *benchmark SPEC CPU2000* foi, inicialmente, concebido para avaliação de ambientes mono processados. Porém, em sistemas que apresentem múltiplos processadores ou *cores*, pode-se iniciar múltiplas instâncias dos programas do *benchmark* e transformar os resultados discretos de cada uma das instâncias em um taxa (*SPEC CPU2000_rate*). Assim, pode-se testar os processadores e memória de um sistema multiprocessado. Neste estudo, serão utilizados os programas rodando em *single-instance*, ou seja, utilizaremos a simulação discreta dos programas do *benchmark*.

Um aspecto importante a ser observado e que sustenta a escolha do *benchmark SPEC CPU2000* é que o intuito deste estudo é avaliar o desempenho e impacto de uma nova política de substituição de linhas de *cache*. Ou seja, o foco está concentrado na memória *cache* do processador. Assim, o fato de o *SPEC CPU2000* não realizar operações de *I/O* em disco e em rede não compromete sua acurácia das medições e o propósito deste estudo.

A versão dos programas do *benchmark SPEC CPU2000* utilizada nas simulações é a que acompanha o simulador *SimpleScalar*. O código fonte dos programas não está disponível para alteração. As simulações utilizaram os binários que acompanham o *SimpleScalar* sem qualquer modificação. Desta forma, não foi possível qualquer inspeção ou instrumentação no código dos programas do *benchmark* que permitisse aprofundar a análise sobre o comportamento e padrão de acesso à memória por parte destes programas. A caracterização do acesso aos endereços e o comportamento do *Entropy* para os diversos programas serão apresentados com maior detalhe no capítulo 7.

Abaixo, a tabela 7.1 mostra uma breve descrição dos programas contidos no *ben-*

*chmark SPEC CPU2000* (SPEC, ).

| Nome | Descrição (Inteiros) |
|---|---|
| 164.gzip | Utilitário de compressão de dados |
| 175.vpr | Ferramenta de design de rota e disposição de FPGA |
| 176.gcc | Compilador C |
| 181.mcf | Calculador de custo mínimo de fluxo de rede |
| 186.crafty | Programa de Xadrez |
| 197.parser | Processador de linguagem natural |
| 252.eon | Renderização de imagens - Ray tracing |
| 253.perlbmk | Interpretador de linguagem Perl |
| 254.gap | Estudo de teoria dos grupos por computador |
| 255.vortex | Banco de Dados orientado a objetos |
| 256.bzip2 | Utilitário de compressão de dados |
| 300.twolf | Utilitário de simulação de localização e rota de circuitos |
| Nome | Descrição (Ponto flutuante) |
| 168.wupwise | Simulador de Cromo-dinâmica quântica |
| 171.swim | Modelagem de movimento de águas rasas |
| 172.mgrid | Calculador Multi-grid 3D campos de potencial |
| 173.applu | Equações diferenciais parciais Parabólicas e Elípticas |
| 177.mesa | Biblioteca gráfica 3D |
| 178.galgel | Análise de Instabilidade Oscilatório: Dinâmica de Fluídos |
| 179.art | Simular de Rede Neural Artificial |
| 183.equake | Simulador de Elementos Finitos: Modelagem de Terremotos |
| 187.facerec | Reconhecimento de faces: Visão Computacional |
| 188.ammp | Química Computacional |
| 189.lucas | Teste de Primalidade: Teoria dos Números |
| 191.fma3d | Simulação de Colisão: Elementos Finitos |
| 200.sixtrack | Modelo de Aceleração de Partículas |
| 301.apsi | Calculador de problemas de distribuição de temperatura, vento e poluentes |

**Tabela 7.1:** Programas do *SPEC CPU2000 benchmark* (Fonte:www.spec.org)

## 7.2 Resultados

Os resultados apresentados neste capítulo têm como *baseline* a comparação entre *Entropy* e *LRU*. O modelo é formado de um processador *superscalar* (OOO) e memória *cache*, ambos modelados conforme os parâmetros apresentados na tabela 7.2 abaixo. O tamanho de 1024-KB (Kilobytes) para a *cache* de segundo nível foi escolhido de forma que o *working set* dos programas do *benchmark* não ficassem inteiramente armazenados na *cache*, o que reduziria a taxa de *misses* dos programas e prejudicaria a comparação entre os dois algoritmos.

A comparação entre o desempenho do *Entropy* e do *LRU* será feita de forma rela-

| Parameter | Value |
|---|---|
| Pipeline | SuperScalar out-of-order |
| Load/Store Queue | 8 |
| L1 I-cache | 32-kB; 64B line size; 2-way LRU; |
| L1 I-fetch queue size | 4 |
| L1 I-decode queue size | 4 |
| L1 I-issue queue size | 4 |
| L1 I-commit queue size | 4 |
| L1 I-hit latency | 1 |
| L1 D-cache | 32-kB; 64B line size; 2-way LRU; |
| L1 D-hit latency | 1 |
| L2 Unified | 1024-kB; 64B line size; 8-way (LRU or Entropy); |
| L2 D-hit latency | 6 |

**Tabela 7.2:** Parâmetros de arquitetura simulada

tiva através da relação 7.1 apresentada abaixo.

$$Diff = ((entropy_{rate} \ / \ lru_{rate}) - 1) * 100 \tag{7.1}$$

Desta forma, quando o *Entropy* apresentar desempenho melhor que o *LRU*, a razão *Diff* vai apresentar um valor negativo de porcentagem, indicando que houve uma redução na taxa de *miss* em relação ao *LRU*. Caso contrário, a razão *Diff* vai apresentar um valor positivo, indicando que o algoritmo *Entropy* teve um desempenho pior do que o *LRU* e houve aumento na taxa de *misses* durante a execução do programa.

A seguir são apresentadas as estatísticas e os resultados obtidos através da simulações de 20B (vinte bilhões) de instruções para 16 (dezesseis) dos programas do *benchmark SPEC CPU2000*. Nesta janela de execução não foram efetuados descartes (*fast-forwarding*) das instruções iniciais. Além da comparação entre *Entropy* e *LRU* utilizando a taxa de *miss rate*, busca-se caracterizar o comportamento do algoritmo *Entropy* em intervalos de 100M (cem milhões) de instruções. Para tanto, faz-se necessário executar as instruções contidas na parte inicial dos programas de forma a permitir essa avaliação neste trecho.

A figura 7.1 sumariza as taxas de *misses* para os 16 (dezesseis) programas do *benchmark SPEC CPU2000* que foram executados. A diferença relativa entre a taxa de *misses* do *Entropy* e do *LRU* é apresentada sobre a barra do gráfico que corresponde ao resultado do *Entropy*. Os diferentes programas executados possuem diferenças na forma como os endereços de memória são referenciados. Esta variação no comportamento de acesso será caracterizada nas curvas apresentadas abaixo onde a evolução da taxa de *misses* para o *Entropy* e *LRU* serão apresentados. No gráfico 7.1 constam os

valores de *miss rate* ao término da execução dos programas enquanto que nas curvas de *miss rate* será possível acompanhar a evolução deste indicador em intervalos de 100M (cem milhões) de instruções.

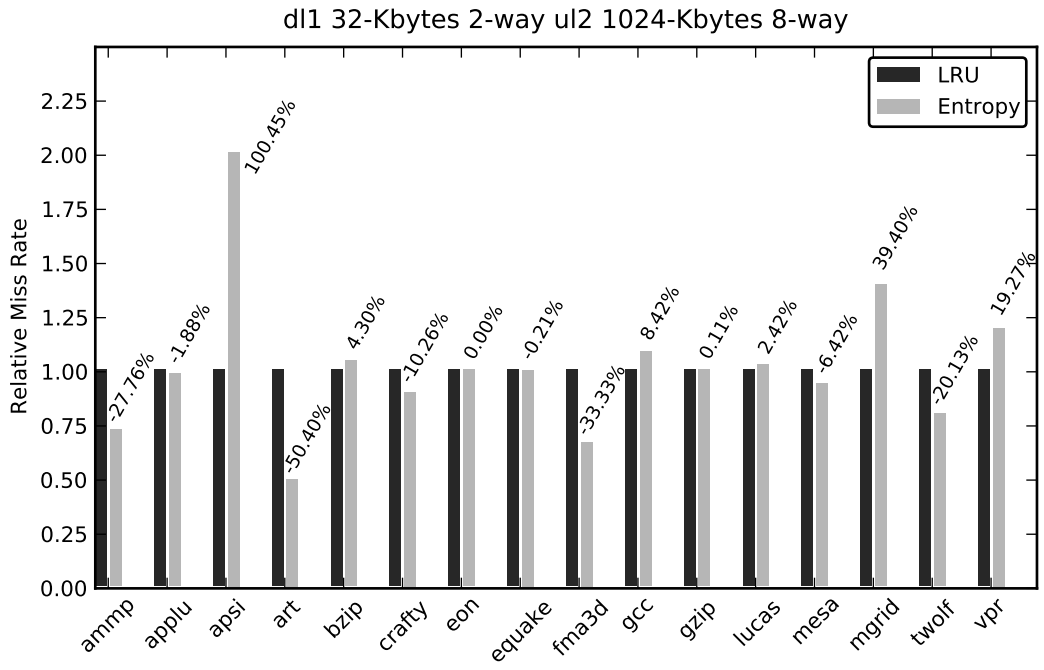


**Figura 7.1:** Relative miss rate LRU versus Entropy

A utilização destas curvas com a taxa de *misses* permite a visualização de trechos dos programas onde o desempenho do *Entropy* esteve melhor ou pior do que o *LRU*. Exemplificando, se o resultado final do *Entropy* foi melhor do que o resultado do *LRU*, não significa dizer que seu desempenho foi melhor durante todos os trechos da execução do programa. A compreensão da evolução do desempenho dos algoritmo será possível através destas curvas.

A partir da figura 7.1 nota-se que 8 (oito) programas apresentam *miss rate* menor quando utilizando o Entropy. Dentre eles, o programa *equake* apresenta um ganho marginal de -0.21%, enquanto que o programa *art* apresenta uma redução significativa de -50.40% no *miss rate*. Os demais programas que apresentaram comportamento favorável ao *Entropy* foram *ammp* com redução de -27.76%, *applu* com -1.88%, *crafty* reduzindo -10.26%, *fma3d* com ganho de -33.33%, *mesa* ganhando -6.42% e, por último, o programa *twolf* com diminuição de -20.13% na sua taxa de *misses*.

O programa *eon* apresenta um resultado final idêntico entre o *Entropy* e *LRU*. Se a variação na política de substituição de linhas de *cache* não apresenta impacto na taxa de *misses* do programa executado, existe forte evidência de que o *working set* do programa esteja sendo quase que completamente armazenado nos endereços disponíveis na memória *cache*. Desta forma, se sempre há endereços disponíveis para

os blocos referenciados pelo programa, a relevância da política de substituição será menor. Em primeira análise, pode-se concluir que esta seja a razão para os resultado apresentado para o programa *eon*.

Existem *workloads* cuja característica de acesso à memória e o *working set* são bem conhecidos. Usualmente os sistemas especialistas, tais como processadores de sinais, apresentam esta previsibilidade. Nestes casos em que o *working set* dos programas pode ser completamente atendido pela memória *cache* pode ser mais vantajoso simplificar o circuito e adotar uma política de substituição de linhas de *cache* mais simples como *FIFO* ou *Random*. Do ponto de vista de arquitetura do processador, a utilização de um algoritmo de substituição de linhas que seja simples não onera a complexidade dos circuitos e pode-se projetar um processador que consome menos energia e dissipa menos calor.

Dentre os *benchmarks*, alguns apresentaram uma taxa de *misses* superior quanto utilizando o *Entropy*, ou seja, um desempenho pior do que o obtido com o *LRU*. Nota-se que o programa *gzip* apresentou aumento marginal de 0.11% na taxa de *misses* enquanto que o programa *apsi* teve sua taxa de *misses* aumentada em 100.45%. Este aumento excessivo no *miss rate* sugere que o algoritmo *Entropy* está retendo na *cache* linhas úteis por tempo inferior ao necessário para que novas referencias ocorram àquele mesmo endereço. Ainda, o algoritmo *Entropy* pode estar retendo linha inúteis ao processamento por tempo superior ao necessário. Neste caso, ocorre aumento de 100% nos *misses*, pode-se concluir imediatamente que a quantidade de endereços livres na *cache* caiu pela metade em comparação ao que se observa com o uso do *LRU*. Adiante serão analisadas as curvas em 7.11 que mostram a evolução da taxa de *misses* para ambos algoritmos durante a execução do *benchmark apsi*.

Para os demais programas simulados, observa-se um aumento de 4.30% no *miss rate* do *bzip*, 8.42% no *gcc*, 2.42% no *lucas*, 39.40% no *mgrid* e 19.17% para o programa *vpr*. Dentre os programas em que o *Entropy* apresentou taxas de *misses* pior do que as do *LRU*, destaca-se três destes programas: *aps*, *mgrid* e *vpr*. A razão desta distinção é a suspeita de que existe um nível de similaridade na forma ou padrão de acesso aos endereços por parte destes programas. Essa afirmação fundamenta-se no fato de que estes programas apresentaram as taxas de *misses* mais altas para o *Entropy* e seus respectivos valores estão bem acima dos valores observados para os demais programas cuja taxa de *miss rate* foi mais alta do que do *LRU*. Ou seja, a causa raiz do baixo desempenho do *Entropy* nestes três programas, possivelmente, é a mesma.

Uma primeira análise dos números finais e absolutos de *miss rate* para os programas do *benchmark* permite identificar quais programas obtiveram ganhos ou perdas de

desempenho. No entanto, a estatística de *miss rate* pode sofrer oscilações ao longo da execução dos programas. Para compreender melhor o comportamento dos programas apresentam-se as curvas de evolução desta taxa. Estas curvas caracterizam a evolução da taxa de *misses* a cada 100M (cem milhões) de instruções e permitem identificar se houve intervalos onde o desempenho do *Entropy* esteve melhor que o desempenho do *LRU* e em intervalos subsequentes passou a apresentar desempenho inferior e vice-versa.

Dois dos programas, *equake* e *art*, apresentaram, respectivamente, menor e maior índice de ganho quando utilizando o algoritmo *Entropy*. Nota-se através da figura 7.2 que durante toda a execução da amostra de instruções a curva de *miss rate* permaneceu sobreposta para ambos algoritmos. Inicialmente, entre o intervalo de 100M até 350M de instruções, houve diminuição gradual da taxa de *misses* seguida de intenso aumento de *misses* até a faixa de 450M de instruções. Após o patamar de 500M de instruções houve aumento gradual de *miss rate* chegando próximo da estabilidade ao final da execução. Este comportamento pode sugerir que a taxa de *misses* deste programa é fortemente determinada por *misses* compulsórios, onde a política de substituição de linhas de *cache* não tem influência direta no desempenho. Isso justifica o aumento intenso na taxa de *misses* a partir do patamar de 350M de instruções. Para uma hipótese mais assertiva sobre as causas deste comportamento faz-se necessário a inspeção do código-fonte dos programas executados nesta simulação. No entanto, os mesmos não se encontram disponíveis para que esta análise mais aprofundada possa ser realizada.

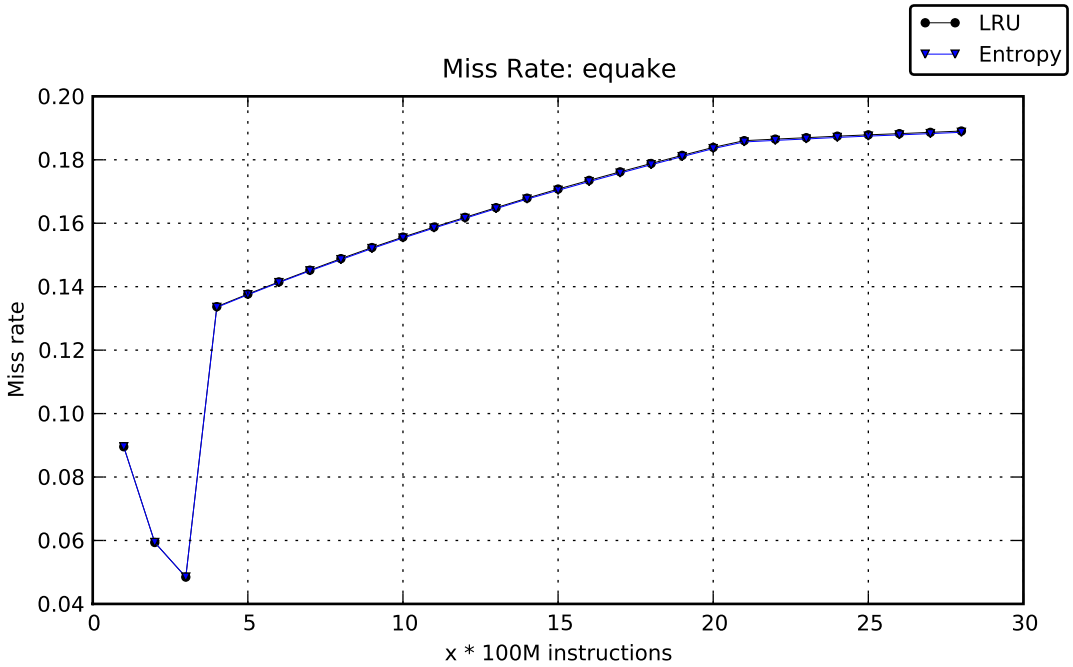


**Figura 7.2:** Miss rate for Equake

No caso do programa *art* que apresentou ganho final de -50.40% e está representado através da figura 7.3, nota-se que a diferença na taxa de *misses* se sustenta durante todo o intervalo de execução de instruções. Do ponto inicial até o ponto final da execução o *Entropy* conseguiu manter a taxa de *misses*, aproximadamente, 50% menor do que o *LRU*. Acredita-se que este comportamento seja o efeito causado pela maior inércia do *Entropy* em substituir uma linha de *cache* tem sobre um programa com localidade referencial longa. Nestes casos, após a linha ter sido carregada na *cache* e sofrer algumas referência subsequentes, o endereço passa um longo intervalo de instruções sem novas referências. O critério de estimativa de chances de reuso da linha faz com o que *Entropy* retenha esta linha por um tempo maior do que o *LRU*. Esse tempo de retenção mais longo aumentaria a taxa de *hits* caso estes endereços inicialmente referenciados voltassem a sofrer acessos após teste tempo de inatividade. Apesar da função decaimento do *Entropy* provocar a redução da entropia da linha quando não há novos acessos, evitando que ela fique confinada na *cache*, ainda assim o *Entropy* pode vir a reter a linha por um tempo maior do que o *LRU*.

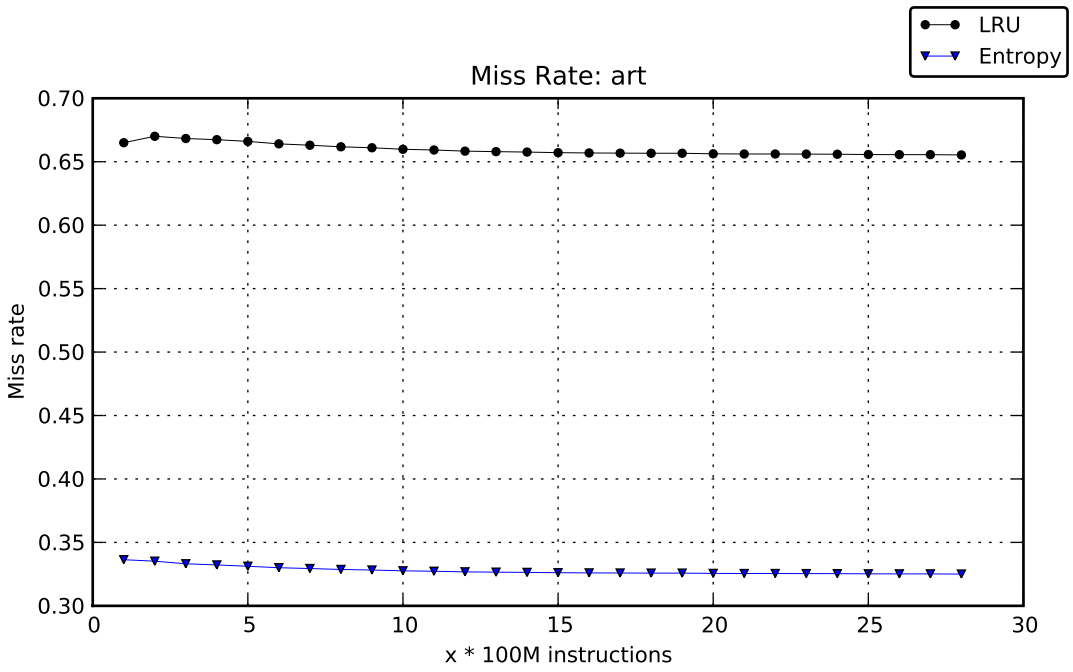


**Figura 7.3:** Miss rate for Art

Na figura 7.4 nota-se que o *Entropy* mantém a vantagem na taxa de *misses* ao longo de todo o intervalo de execução. As variações de aumento e diminuição na taxa de *misses* que se observa, por exemplo, entre 100M, 200M, 400M e 500M de instruções são simétricas entre o *Entropy* e *LRU*. Após este intervalo a taxa de *misses* apresenta um padrão de variação mais suave do que o observado nos intervalos iniciais, porém, mantendo-se a vantagem do *Entropy* em relação ao *LRU*. A vantagem

que o *Entropy* adquire no trecho inicial de execução é, praticamente, mantida ao longo da execução. Os *misses* que ocorrem nos trechos mais avançados da execução provocam novos *misses* quando utilizando *Entropy* e, igualmente, quando utilizando o *LRU*. Porém, a vantagem inicial do *Entropy* é mantida e esta vantagem sustenta o melhor desempenho do *Entropy* até o final da amostra de execução.

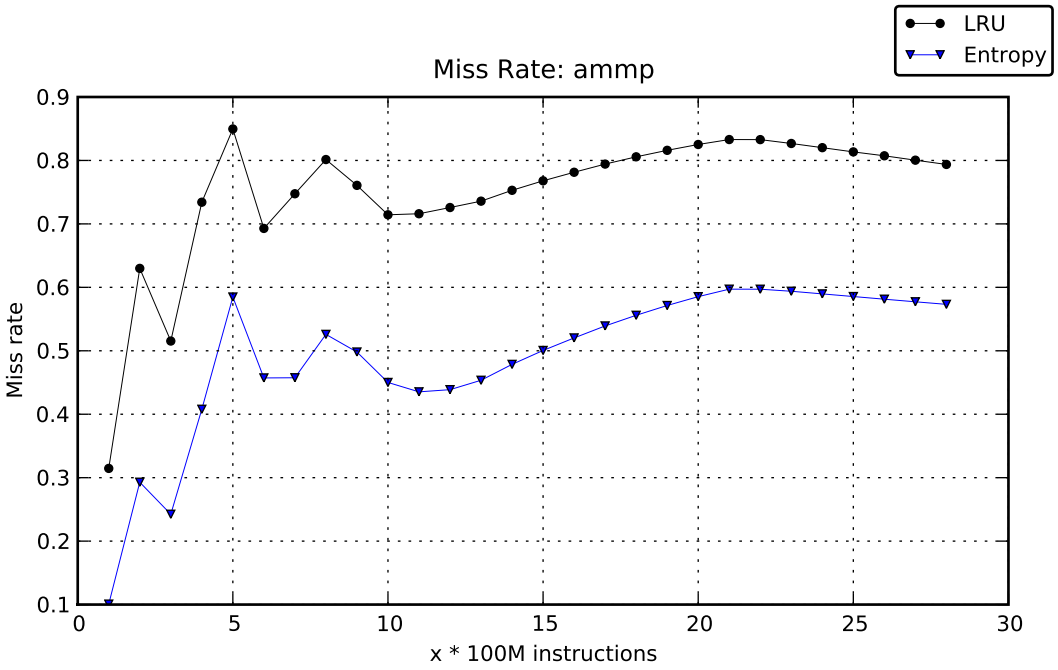


**Figura 7.4:** Miss rate for Ammp

No caso do programa *applu*, o desempenho do *Entropy* é, praticamente, igual ao do algoritmo *LRU* até os 700M (setecentos milhões) de instruções. Acima deste montante de instruções o *Entropy* passa a apresentar vantagem em relação ao *LRU* e esta vantagem vai gradualmente aumentando até por volta dos 900M de instruções de modo que passa a ser visualmente perceptível tal diferença através da figura 7.5. A partir dos 900M de instruções os dois algoritmos passam a apresentar o mesmo perfil de variação no que diz respeito à taxa de misses. Observa-se uma simetria nas curvas da mesma forma que se observou para o programa *ammp*. Assim como ocorreu com este último programa, no caso do *applu*, a vantagem que o *Entropy* abre em relação ao *LRU* se mantém até o final da amostra de instruções simuladas. Uma hipótese para este comportamento de aumento e diminuição de *miss rate* pode ser a intercalação de trechos de *misses* compulsórios com trechos de acesso a estes últimos endereços ou endereços já armazenados na *cache*, provocando o aumento seguido da diminuição do *miss rate*. Sob essa hipótese, o trecho inicial do programa seria dominado por *misses* compulsórios, sobre os quais a política de substituição de linhas tem pouca incidência e efeito.

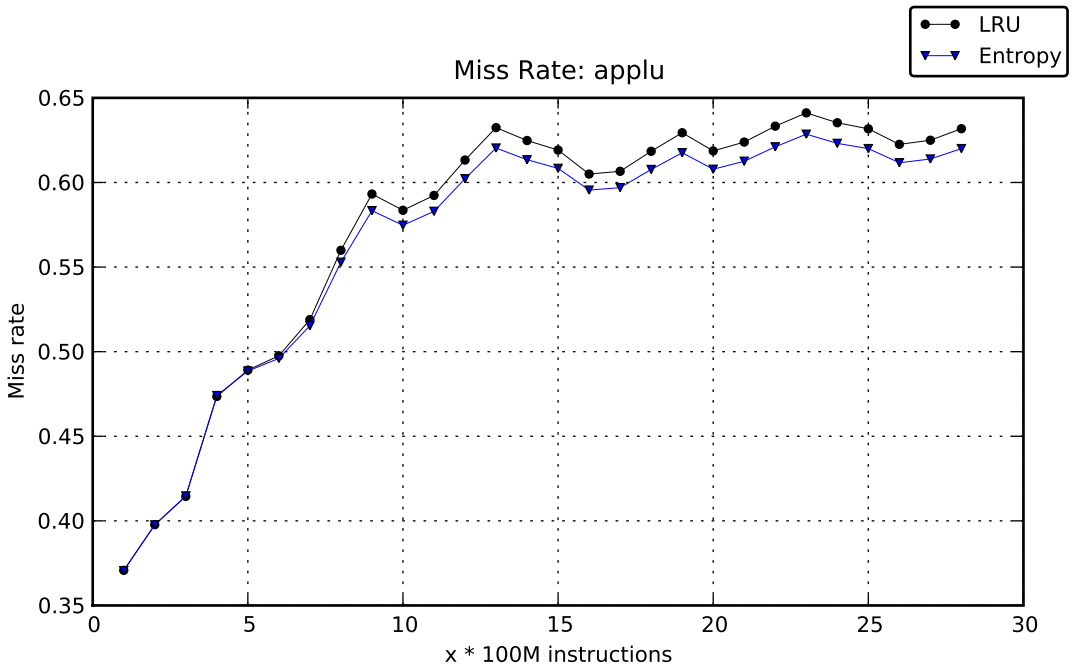


**Figura 7.5:** Miss rate for Applu

O *Entropy* apresenta um ganho final de -10.26% no caso do programa *crafty*. Mas, através da figura 7.6 é possível observar que a taxa absoluta de *misses* é muito baixa para ambos algoritmos utilizados. Isso significa que o programa *crafty* possui um *working set* cujo tamanho pode ser quase que completamente endereçado pela *cache* de segundo nível. No trecho inicial da execução desta amostra, cerca de 2.5% das referências à *cache* resultaram em *misses* e esta taxa caiu para menos de 0.5% dos acessos ao final da amostra. O comportamento deste programa é assintótico e para um amostra maior de instruções o valor final do *miss rate* tanto para o *Entropy* quanto para o *LRU* podem convergir para o mesmo valor. No entanto, o *Entropy* foi capaz de abrir uma vantagem no gerenciamento dos endereços utilizados pelo *crafty* reduzindo a taxa de *misses* mesmo para um programa cujo *working set* é pequeno como no caso do *crafty*.

O mesmo comportamento assintótico pode ser observado para o programa *fma3d* no que diz respeito à curva de *miss rate*. No entanto, no intervalo de 300M até 500M observa-se através da figura 7.7 uma vantagem marginal do *Entropy* em relação ao *LRU*. Ao final da execução, observa-se uma sobreposição de ambas as curvas, porém, o valor final absoluto de *miss rate* do *Entropy* é 0.02% enquanto que o *miss rate* final do *LRU* é da ordem de 0.03%. Esses valores finais justificam o ganho de -33.33% que o *Entropy* apresentou em relação ao *LRU* apesar de a representação gráfica não permitir esta constatação de forma visual.

Da mesma forma que o programa *fma3d* decai assintoticamente, o programa *mesa*

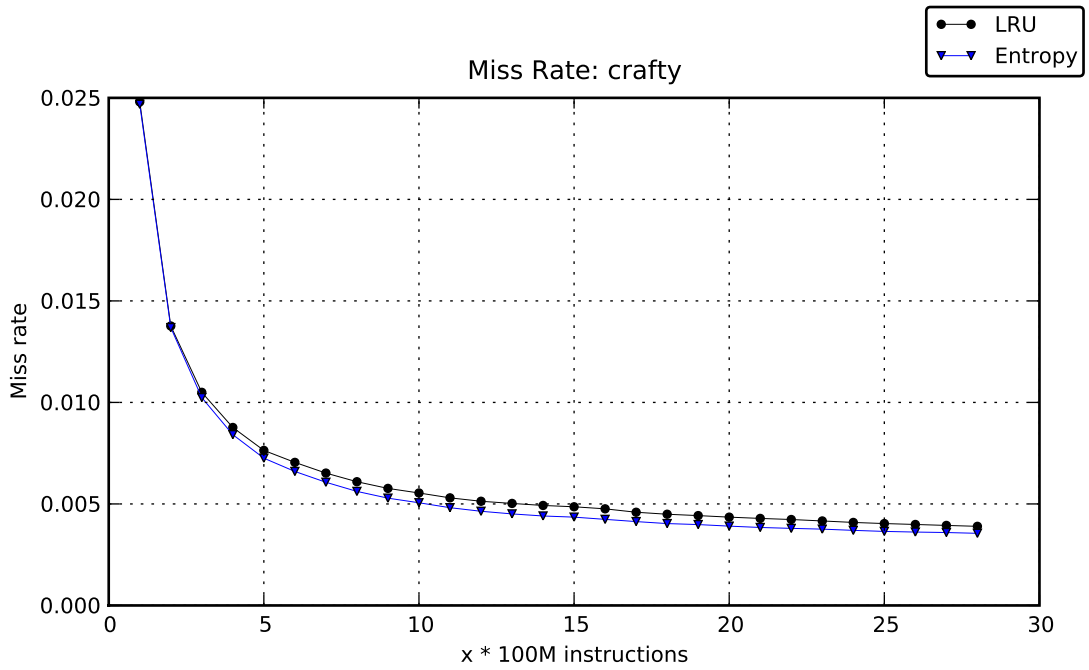


**Figura 7.6:** Miss rate for Crafty

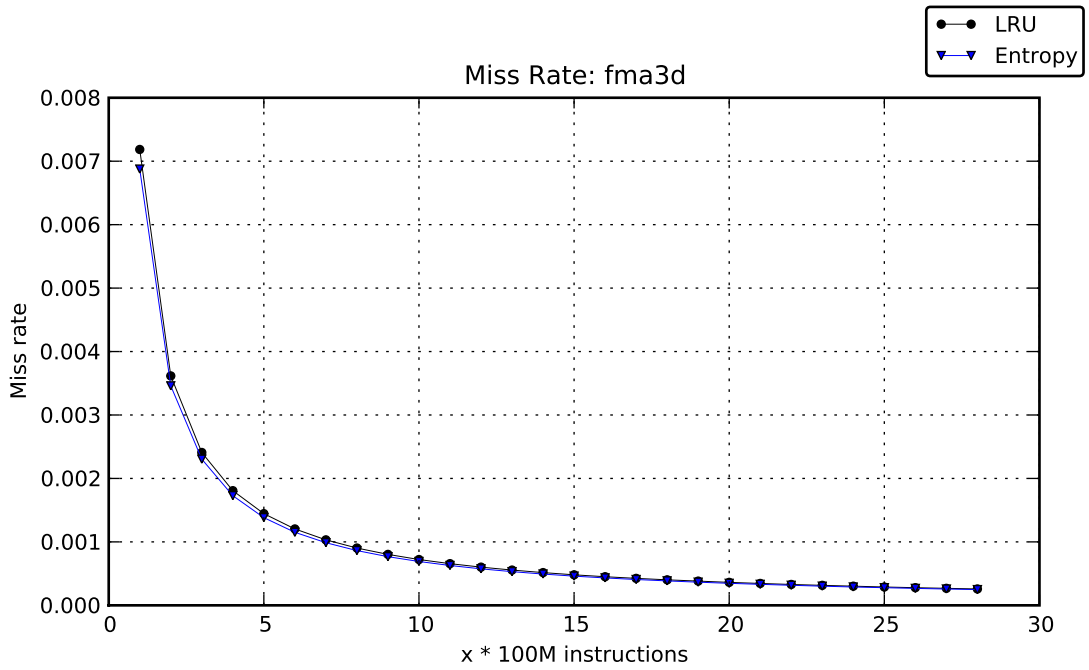


**Figura 7.7:** Miss rate for Fma3d

apresenta intenso decaimento na taxa de *misses* entre o intervalo de 100M a 700M de instruções, conforme apresentado na figura 7.8. Porém, a partir dos 700M de instruções o programa passa a apresentar o mesmo padrão de variação na taxa de *misses* que o observado no programa *applu*. A diferença entre estes dois programas é que no *mesa*, a taxa de *miss* diminui ao longo da execução ao passo que no programa *applu* esta taxa aumenta. Porém, o mesmo comportamento de aumento de *miss rate* seguido

de diminuição do *miss rate* ocorre neste caso. Uma sequência de *misses* compulsórios seguido de novas referências bem sucedidas (*hits*) caracteriza a localidade referencial. Observa-se que a partir de 900M de instruções o algoritmo *Entropy* passa a apresentar vantagem em relação ao *LRU*. Mantendo-se este perfil de acesso aos endereços pode-se supor que a taxa de *misses* para o programa *mesa* entraria em uma faixa de estabilidade com vantagem para o algoritmo *Entropy*.

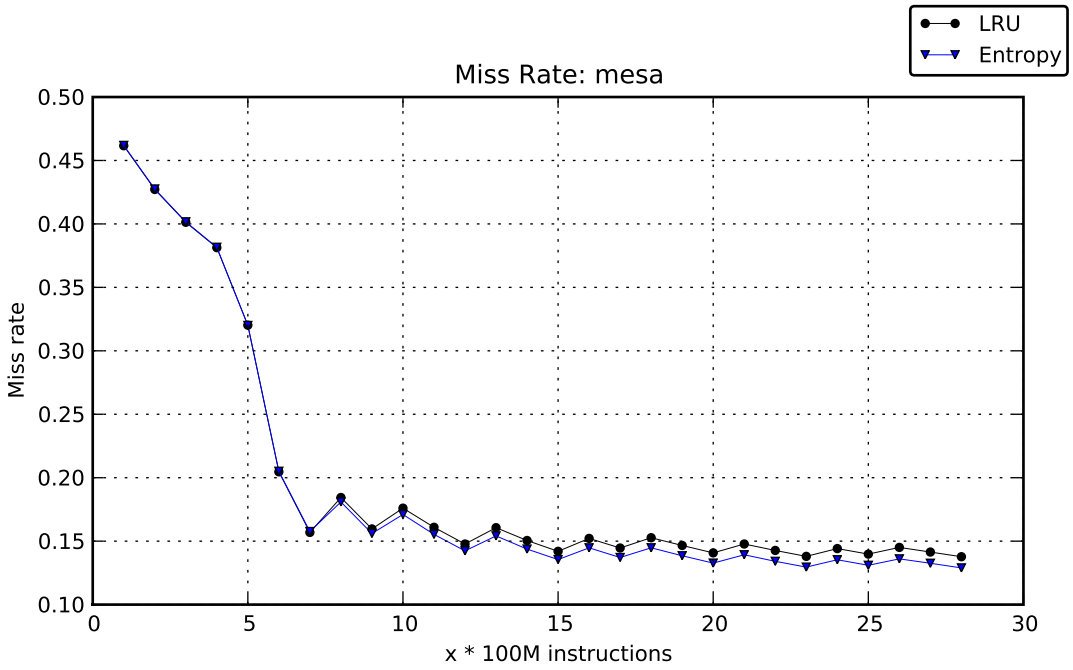


**Figura 7.8:** Miss rate for Mesa

A figura 7.9 o mostra perfil de referência aos endereços de memória por parte do programa *twolf* é caracterizado por intensa variação da taxa de *misses* entre 100M e 600M. Até os 400M de instruções as curvas do *Entropy* e do *LRU* apresentam-se completamente sobrepostas e no intervalo de 400M até 600M pode-se observar leve vantagem para o *Entropy*. No entanto, a partir de 600M de instruções o *Entropy* passa a gerenciar melhor as linhas da *cache* e atinge vantagem sobre o *LRU* até o final da execução. A taxa de *misses* estabiliza e a vantagem conquistada pelo *Entropy* no início da execução garante o melhor desempenho do *Entropy* para o programa *twolf*.

No caso do programa *eon* pode-se observar na figura 7.10 que as taxas de *miss rate* mantiveram-se idênticas ao longo de toda a execução da amostra. Nem mesmo os valores finais absolutos de *miss rate* para ambos os algoritmos apresentaram divergência. Ainda, observando as proporções de *misses* para o programa *eon* pode-se concluir que trata-se de um programa cujo *working set* pode ser endereçado pela *cache*, pois, ao final da amostra a taxa de *misses* não apresentou variação e manteve-se na faixa de 2%. Esta proporção deve-se a *misses* compulsórios.

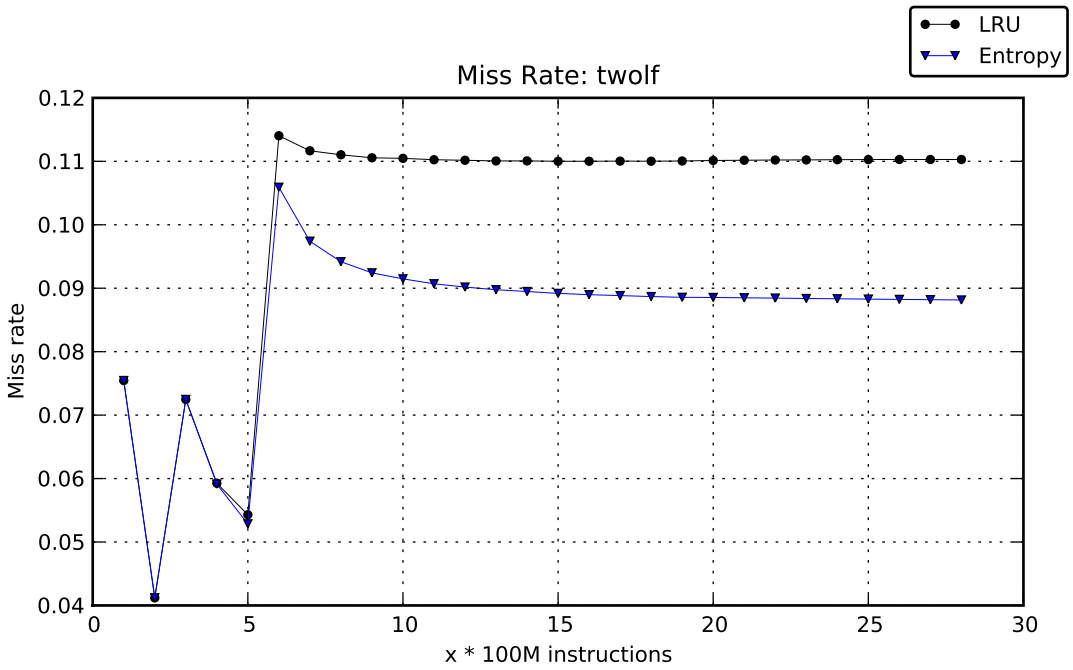


**Figura 7.9:** Miss rate for Twolf

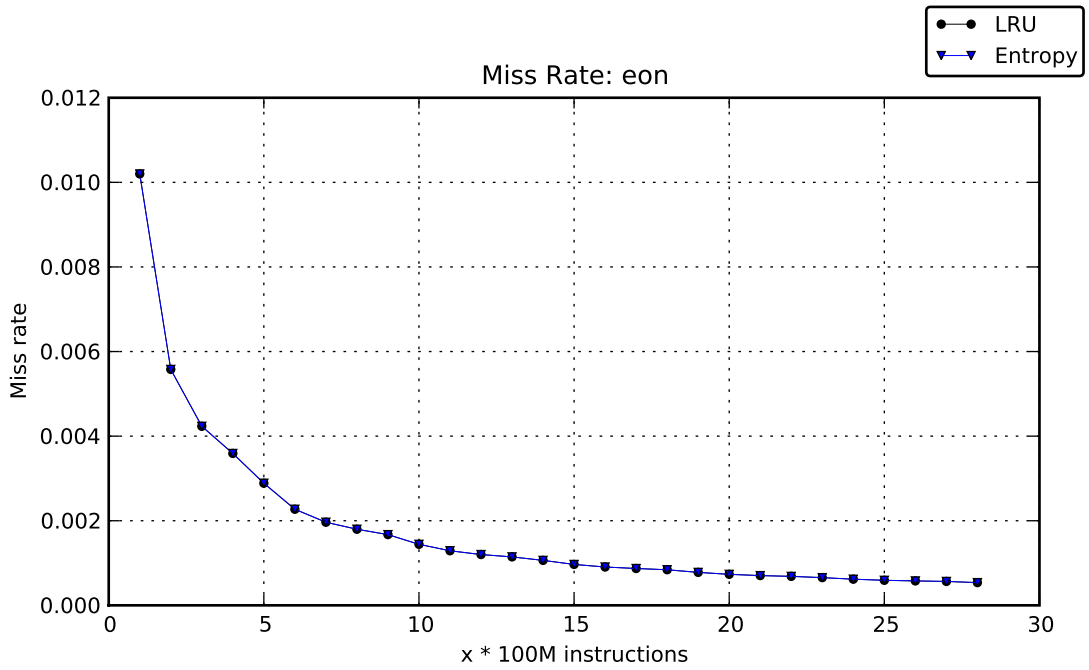


**Figura 7.10:** Miss rate for Eon

Para os programas que apresentaram desempenho inferior, quando utilizado o algoritmo *Entropy*, destaca-se o *apsi*. O comportamento deste programa é caracterizado através da figura 7.11. Até 300M de instruções, o algoritmo *LRU* promove uma redução intensa na taxa de *misses* enquanto que observa-se um aumento nesta mesma taxa para o algoritmo *Entropy*. Após este trecho inicial, o algoritmo *LRU* passa a apresentar uma estabilidade na taxa de *misses*, enquanto que o algoritmo *Entropy* consegue redu-

zir gradualmente a taxa de *misses*. No trecho entre 1.1B e 1.2B (um bilhão e duzentos milhões) de instruções observa-se que houve aumento brusco na taxa de *misses* para o *LRU*, ao passo que este mesmo aumento foi mais suave no caso do *Entropy*. Porém, a grande redução na taxa de *misses* que o *LRU* adquire nos trechos iniciais da simulação se mantém como vantagem até o final da execução. Acredita-se que ocorra uma sequência inicial de *misses* compulsórios seguida de uma sequência longa de referências aos endereços recém carregados na *cache*. Após esta sequência de intenso acesso, o algoritmo *Entropy* terá priorizado na *cache* um conjunto de endereços que receberam o maior número de referências, classificando estes como os melhores candidatos à novas referências. Ao término desta sequência de referências, o algoritmo *Entropy* vai precisar de um trecho maior de acessos à *cache* sem que ocorram *hits* nestes endereços retidos para que eles sejam substituídos da *cache*. Em contrapartida, o algoritmo *LRU* vai deixar os endereços mais acessados sair da *cache* de forma mais rápida. Ao manter endereços por mais tempo na *cache* pode-se diminuir a quantidade de endereços livres nesta *cache*. Acredita-se ser esta a razão para o aumento do *miss rate* no trecho inicial da execução causado pelo *Entropy* enquanto que o *LRU* provocou redução no *miss rate* do programa.

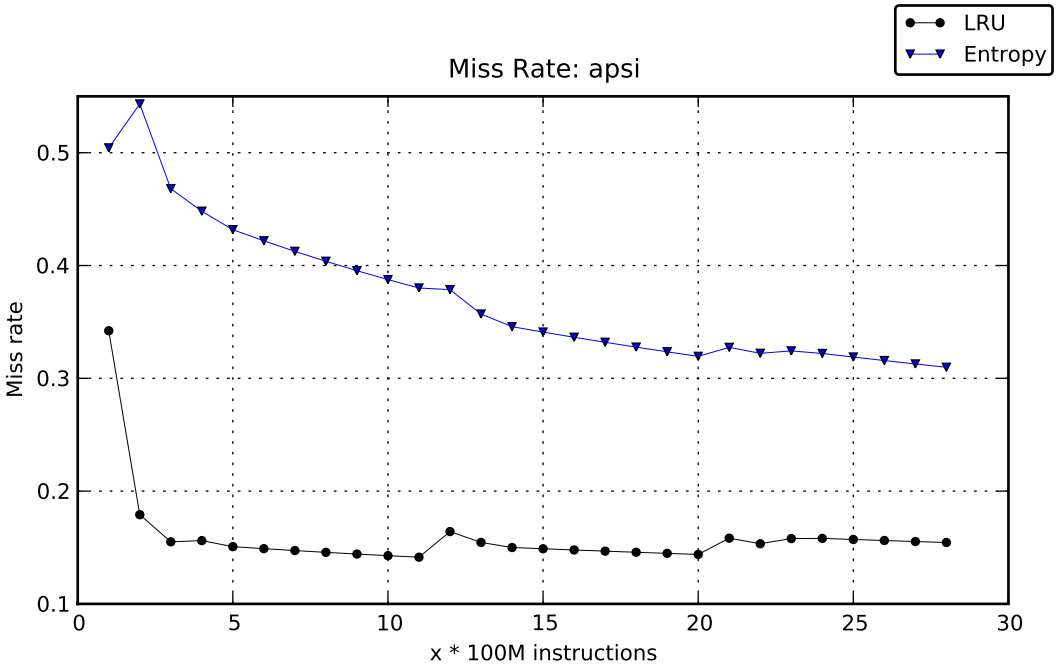


**Figura 7.11:** Miss rate for Apsi

No caso do programa *gzip* a vantagem do *LRU* em relação ao *Entropy* foi de 0.11%. Observando as curvas na figura 7.12, observa-se que esta pequena vantagem do *LRU* sobre o *Entropy* se mantém durante todo os trechos da simulação. Com até 100M de instruções a taxa de *misses* é bastante elevada, estando acima de 50.0%, pode-

se dizer que o perfil de acesso do programa *gzip* é predominantemente compulsório nos trechos iniciais e após os 100M de instruções decai assintoticamente ao longo da execução. Para uma amostra maior de instruções pode-se supor que haverá estabilidade na taxa de *misses* e que o *working set* do programa *gzip* pode ser atendido quase que completamente pela *cache* de 1024-Kbytes de tamanho.

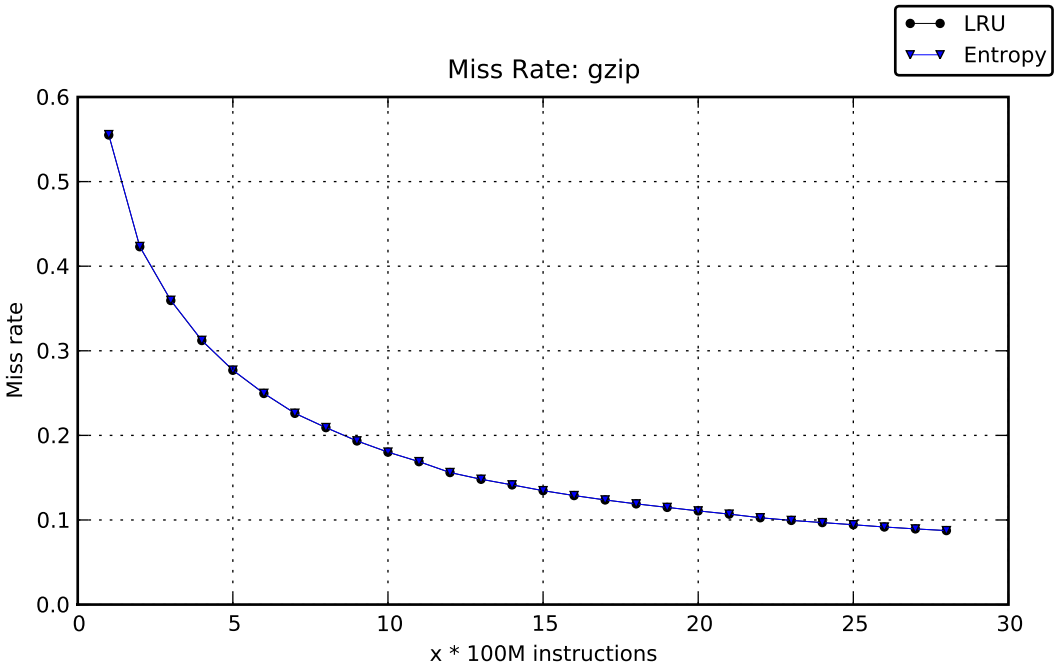


**Figura 7.12:** Miss rate for Gzip

Assim como programa *gzip*, o programa *bzip* apresenta o mesmo perfil de acesso à memória. Ambos os programas tem a mesma funcionalidade de compressão de dados e apresentam alta taxas de *misses* nos trechos iniciais da execução com intensa redução nos trechos subsequentes. Na figura 7.13 pode-se observar este comportamento. Para este programa, o algoritmo *Entropy* e o *LRU* mantêm-se empatados até por volta dos 300M de instruções. Após este trecho o *LRU* passa a apresentar uma taxa de *misses* menor do que a apresentada pelo *Entropy*. O programa apresenta, ainda, uma alternância entre trechos onde a taxa de *misses* diminui, seguida de um trecho onde ela se mantém estável. Esta alternância ocorre ao longo da execução e de forma geral faz com que a taxa de *misses* seja gradualmente reduzida ao longo da execução da amostra. No entanto, a diferença a favor do *LRU* que surge entre 300M e 500M de instruções confere a vantagem final do algoritmo *LRU*.

Para o programa *gcc* o algoritmo *Entropy* apresenta desempenho bastante próximo ao do algoritmo *LRU*. Isso pode ser observado através da figura 7.14. Até 200M de instruções, ambas as curvas permanecem bastante próximas. Após este trecho, o algoritmo *LRU* passa a substituir as linhas da *cache* com maior eficiência e abre

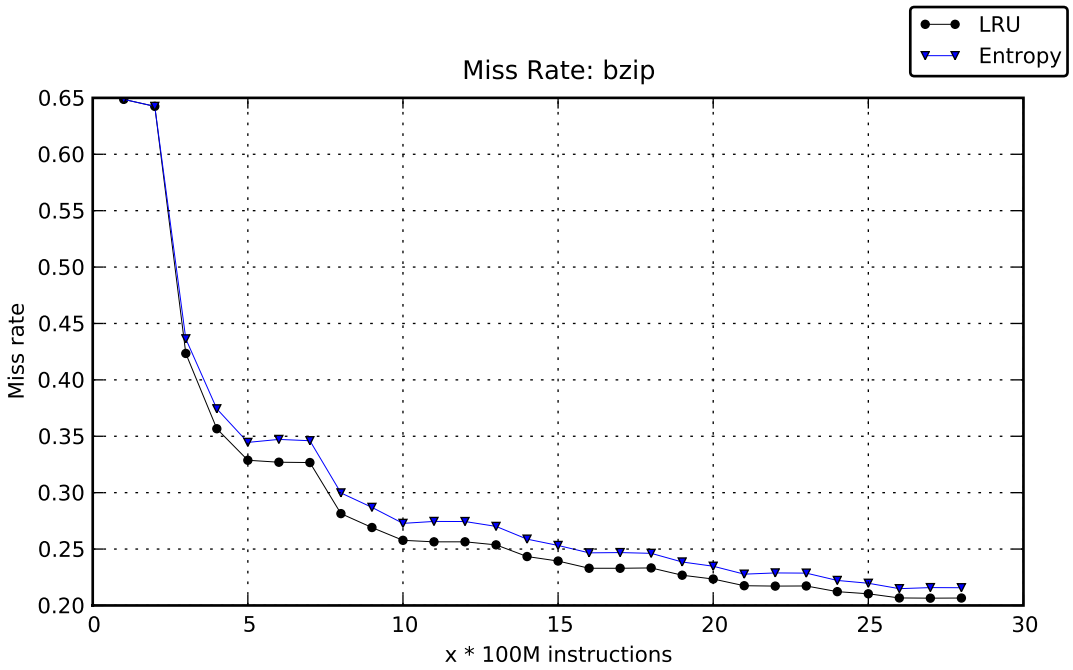


**Figura 7.13:** Miss rate for Bzip

vantagem sobre o *LRU*. O perfil de acesso à memória do *gcc* apresenta dois trechos distintos fortemente caracterizados por uma aumento intenso na taxa de *misses* típica de *misses* compulsórios. A similaridade entre as curvas do *Entropy* e do *LRU* reforça esta hipótese. No entanto, após 2B de instruções o aumento na taxa de *misses* aumenta linearmente no *LRU* e de forma mais intensa no caso do *Entropy*. A figura sugere que, para uma amostra maior de instruções, as taxas de *misses* do *Entropy* e do *LRU* convergiriam para valores finais muito próximos.

No caso do programa *lucas*, apresentado na figura 7.15, a taxa de *misses* do *Entropy* e *LRU* se mantêm equivalentes até o patamar de 1.6B de instruções. A partir deste ponto, ocorre um aumento mais intenso na taxa de *misses* quando utilizando o algoritmo *Entropy*. Este aumento ocorre no trecho entre 1.6B e 2.1B de instruções. Apesar de haver, neste trecho, aumento na taxa de *misses* para o algoritmo *LRU*, este aumento parece ser menos acentuado do que o observado para o *Entropy*. Após este trecho, as curvas mostram-se bastante semelhantes e a diferença entre o desempenho dos dois algoritmos se mantém até o final da simulação.

A figura 7.16 mostra a evolução da taxa de *misses* para o programa *mgrid*. No trecho inicial até 300M de instruções há um aumento intenso na taxa de *misses* sendo que o *LRU* apresenta valor absoluto menor do que o *Entropy*. Após este trecho inicial, ocorre uma brusca redução de *misses* no caso do *LRU* chegando próximo da estabilidade após 1.5B de instruções. Para este mesmo intervalo de instruções, o algoritmo *Entropy* apresenta oscilação na taxa de *misses*. Acredita-se que o motivo deste com-

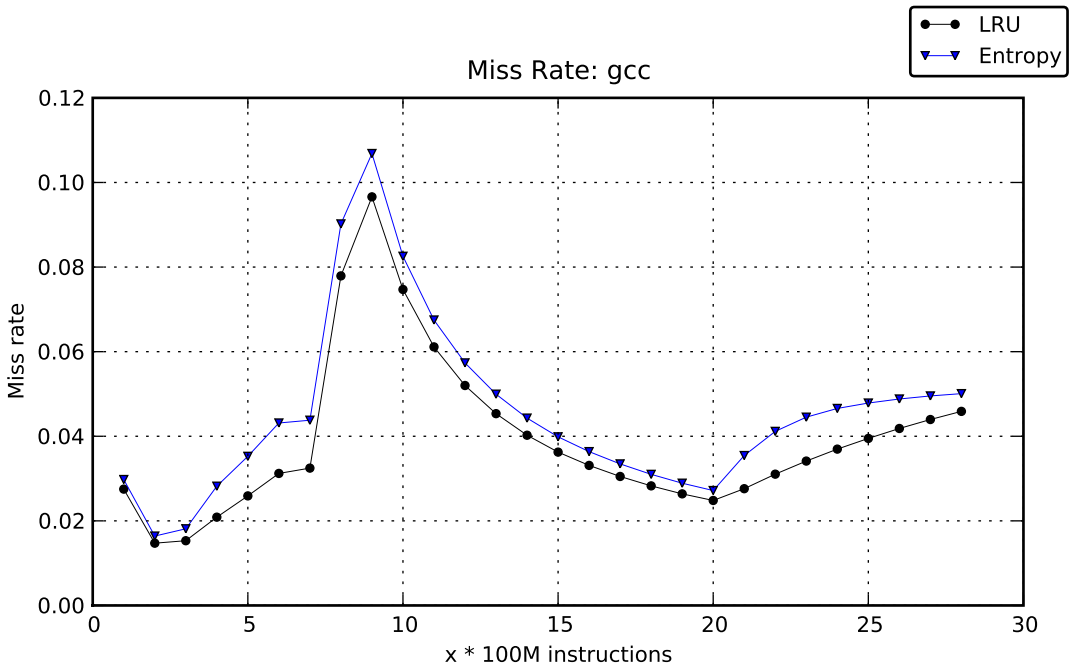


**Figura 7.14:** Miss rate for gcc

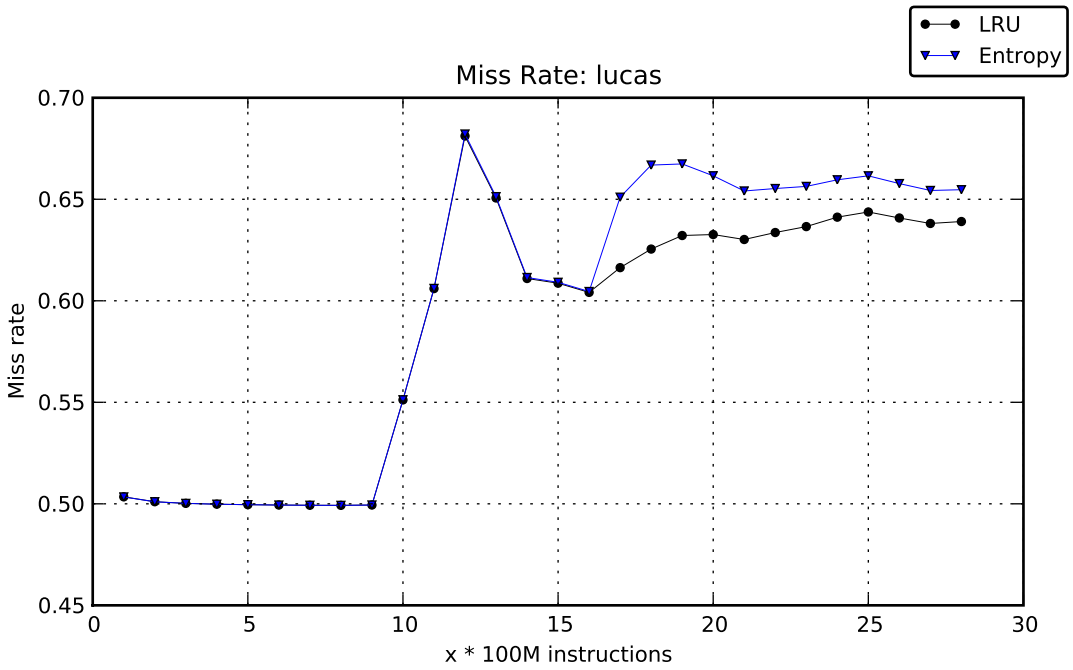


**Figura 7.15:** Miss rate for Lucas

portamento do *Entropy* seja a intercalação de trechos de acesso aos endereços com localidade referencial curta. Nesta situação, as primeiras referências resultariam em *misses* e os acessos subsequentes podem resultar em um número maior de *misses* no caso do *Entropy*. Isso provoca um aumento na taxa de *misses*. Após um número suficiente de referências a estes endereços com localidade referencial, o algoritmo *Entropy* os reteria na *cache* tempo suficiente para ocorrer *hits*, provocando a redução na taxa de

*hits*. Esta pode ser a razão do aumento e decréscimo da taxa de *misses* do *Entropy* conforme apresentado na figura 7.16. Já o algoritmo *LRU* apresenta um comportamento menos oscilatório neste caso, pois, a primeira referência ao endereço com localidade referencial já o colocaria na posição mais protegida da *cache* e os acessos subsequentes resultariam em *hits* imediatamente. Os *misses* compulsórios que ocorrem não afetam a taxa de *misses* conforme observado no trecho 1.5B até o final da execução.

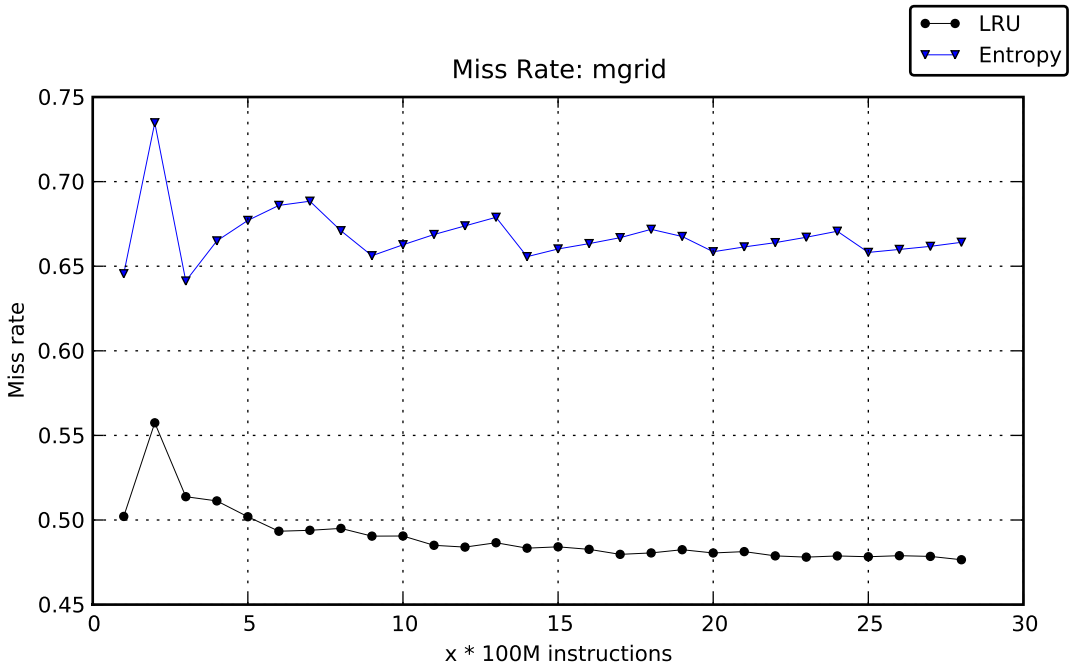


**Figura 7.16:** Miss rate for Mgrid

No caso do programa *vpr*, o algoritmo *Entropy* apresenta resultado melhor até o patamar de 200M de instruções. A partir deste ponto, o algoritmo *LRU* supera o *Entropy* e passa a apresentar resultados melhor. Nota-se que há um padrão semelhante de evolução das curvas para ambos os algoritmos, porém, a partir do patamar de 1.0B de instruções, o aumento na taxa de *misses* do *Entropy* mostra-se mais intenso do que o observado para o *LRU* no mesmo trecho até o final da execução. A partir de 2.5B de instruções, a taxa do *LRU* aparente encontrar-se em estabilidade enquanto que a taxa do *Entropy* aparente estar em declínio.

Resumindo a análise, o algoritmo *Entropy* foi capaz de realizar a substituição de linhas de *cache* de forma mais eficiente em 8 (oito) dos 16 (dezesseis) programas de *benchmark SPEC CPU2000*. Para outros 7 (sete) programas ele apresentou um desempenho inferior e em um dos programas apresentou resultado idêntico ao do *LRU* devido ao *working set* do programa em questão. Pela análise da taxa *misses* ao longo da execução dos programas do *benchmark* constatou-se oscilações bruscas e variações nesta taxa. Tais variações caracterizam o perfil de acesso à memória por parte dos pro-

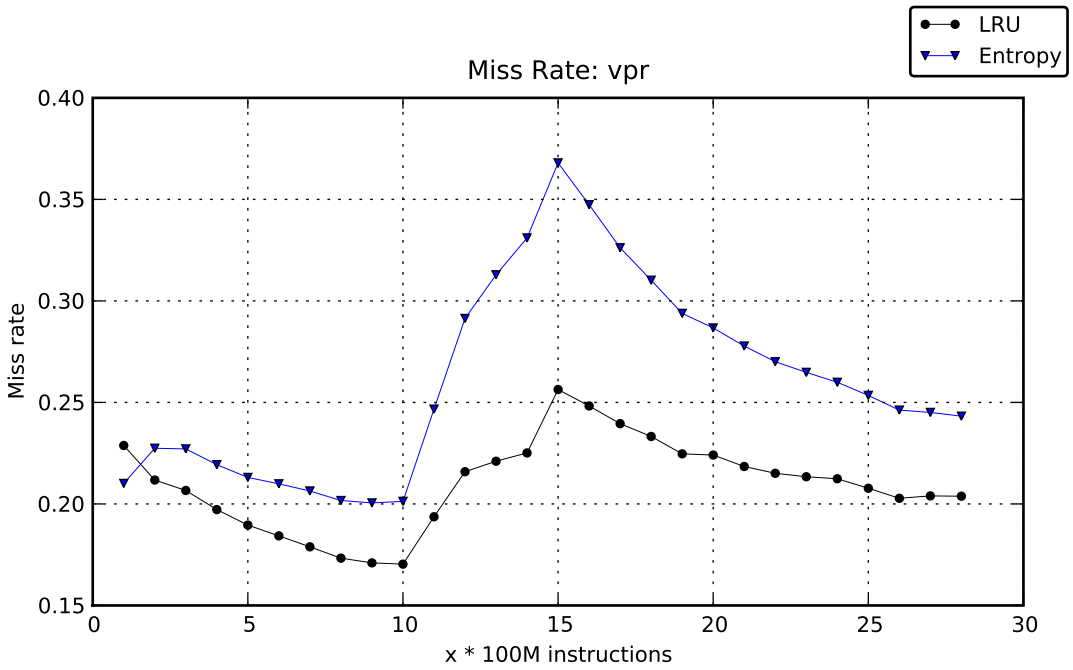


**Figura 7.17:** Miss rate for Vpr

gramas. Ainda, em diferentes trechos da execução de um programa, uma política de substituição de linhas pode apresentar um bom desempenho e no trecho subsequente passar a apresentar um desempenho pior, aumentando a taxa de *misses*. Um experimento que empregue a técnica de *fast-forwarding* de instruções pode simplesmente descartar trechos importantes da execução do programa e levar à conclusões que não sejam precisas. Por esta razão, optou-se nestes estudo por deixar os programas do *benchmark* executarem a amostra de instruções de 20B (vinte bilhões) integralmente sem descarte das instruções iniciais.

# 8 Conclusões e Trabalhos Futuros

Os algoritmos de substituição de linhas tais como o *LRU* levam em consideração somente um critério para escolher qual linha deve ser retirada da memória *cache* quando os endereços estão todos utilizados. O critério utilizado pelo *LRU* faz com este algoritmo tente capturar a localidade temporal de referência mantendo na *cache* as linhas mais recentemente referenciadas e descartando as linhas menos recentemente acessadas. Isso pode fazer com que o *LRU* não capture a localidade temporal, por exemplo, de programas com *working sets* maiores do que a memória *cache*.

Em contrapartida, o *Entropy* leva em consideração, além dos acessos mais recentes, a frequência com que o endereço é referenciado. Por isso, o algoritmo *Entropy* traz contribuições positivas no âmbito da arquitetura de processadores ao empregar uma nova heurística de substituição de linhas de *cache*. Para explorar melhor a localidade temporal dos programas, o algoritmo *Entropy* inspira-se no conceito da Entropia da Informação para estimar as chances de uma linha receber novos acessos após ter sido carregada na *cache* e priorizá-la na *cache* em detrimento de outras linhas com menor chances de reuso.

Um boa política de substituição de linhas tem que retirar da *cache* a linha com menor chance de ser referenciada. Porém, os programas de computador usualmente apresentam grande variação no padrão de acesso à memória e esta escolha torna-se pouco assertiva. Em um determinado trecho da execução de um programa, a política de substituição de linhas pode apresentar um bom desempenho e no trecho subsequente passar a apresentar um desempenho pior, aumentando a taxa de *misses*. Tais variações que ocorrem na taxa de *misses* ajudam a caracterizar o perfil de acessos à memória por parte dos programas.

Por este motivo, os resultados apresentados no capítulo 7 se concentraram em mostrar os ganhos finais obtidos através do novo algoritmo *Entropy* e, também, analisar a evolução da taxa de *misses* ao longo da simulação. Com os dados apresentados, foi possível observar a intensidade e amplitude das variações na taxa de *misses*. Através das curvas de *miss rate* pode-se constatar que o algoritmo *Entropy* e *LRU* intercalam trechos de comportamento divergente com trechos de grande similaridade entre as

curvas de ambos algoritmos.

Observou-se que a vantagem obtida por um algoritmo em um ponto da simulação, invariavelmente, é mantida até o final da simulação. No entanto, esta diferença surge em um trecho específico da simulação. Pôde-se notar, também, os casos em que os algoritmos simulados apresentaram comportamento uniforme em todo o intervalo de execução. O ponto fundamental da análise é que, restringir-se à taxa final de *misses* pode levar a uma conclusão imprecisa sobre o desempenho da política de substituição de linhas. Neste estudo, estendeu-se e critério de análise do desempenho da política de substituição de linhas incluindo as curvas de *miss rate* traçada em pontos pré-definidos durante a simulação.

Analisando os números finais das taxas de *misses*, pôde-se notar uma redução em mais do que 50% dos programas simulados em favor do *Entropy*. Embora o código-fonte não estivesse disponível, acredita-se que tenha sido demonstrada a grande relevância deste estudo e de seus resultados muito promissores. O trabalho culminou no desenvolvimento de uma nova heurística inspirada na Entropia da Informação, no desenvolvimento de um novo algoritmo (*Entropy*) bem como na proposição de sua implementação em *hardware* utilizando circuitos comparadores e lógica discreta. Este circuito foi pensado de forma a manter o *overhead* de *storage* e a complexidade do circuito dentro de patamares comparáveis aos do algoritmo *LRU* para que sua implementação seja viável.

## 8.1 Trabalhos Futuros

Uma hipótese apresentada para justificar a maior taxa de *misses* em alguns dos programas simulados foi a maior inércia do algoritmo *Entropy* em proteger uma linha de dados que acabou de ser referenciada pela primeira vez. Isso deve-se ao peso que a frequência de acesso tem na definição da Entropia discreta da linha de dados. Somente após alguns acessos àquela nova linha é que o algoritmo *Entropy* vai protegê-la contra a substituição.

Como parte de trabalhos futuros, pode-se propor algumas abordagens para contornar a situação de maior inércia em reter as linhas. Uma primeira proposição é a adoção de uma função que acelere esta priorização e retenção de linhas na *cache*. Porém, esta função não pode ser arbitrariamente empregada, pois certamente comprometeria a boa resposta que o *Entropy* apresenta nos casos em que ocorre longos *scans* de endereços que serão referenciados somente uma vez. Acelerar a retenção em caso faria com que o algoritmo mantivesse na *cache* dados desnecessários neste exemplo mencionado.

Faz-se necessário aprofundar a caracterização do padrão de acesso dos programas do *benchmark SPEC CPU2000* para que seja possível projetar o ajuste necessário na retenção das linhas bem como eventuais ajustes na função decaimento de Entropia, que evita o confinamento de linhas na *cache*. Este detalhamento requer a elaboração de um novo critério de *tracing* dos endereços de memória que de forma a levantar dados mais granulares sobre o uso da *cache*.

Uma outra estratégia que pode ser adotada e avaliada é a extensão do algoritmo *Entropy* de forma que se comporte como o algoritmo *LRU* mediante a inspeção dos contadores e estruturas de dados que ele mantém durante a execução. Há a necessidade de elaboração de um critério *ad-hoc* que permita concluir que modificar a heurística de substituição para *LRU* pode reduzir o número de *misses*. A definição deste critério *ad-hoc* pode permitir que o algoritmo se comporte da forma mais aderente ao padrão de acesso à memória que está em execução no processador. A despeito da possibilidade de empregar uma política adaptativa, deve-se levar em consideração a complexidade do circuito que implementa esta política versus o benefício ou ganho que a adaptabilidade pode conferir.

Com a disponibilidade dos códigos-fonte dos *benchmarks* do *SPEC CPU2000*, uma análise mais profunda e detalhada poderia ser feita de modo a ajustar melhor a função de decaimento de Entropia, além de ter uma noção bem mais exata de quais elementos estariam nas memórias *cache*, consequentemente permitindo um melhor acoplamento e possível melhora do desempenho. Finalmente, um compilador poderia tirar proveito gerando um código otimizado com olhos na Entropia.

# Referências Bibliográficas


ARM. *ARM1020T Technical Reference Manual*. [S.l.], 2000.

ARM. *ARM Cortex A Technical Reference Manual*. [S.l.], 2008.

AUSTIN, T. *SimpleScalar LLC*. 1999. `http://www.simplescalar.com/`.

BARROSO, L. A. et al. Piranha: a scalable architecture based on single-chip multiprocessing. In: *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2000. p. 282–293. ISBN 1-58113-232-8.

BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems*, New York, NY, USA, n. 5, p. 78–101, 1966.

BISWAS, S. et al. Multi-execution: multicore caching for data-similar executions. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 37, n. 3, p. 164–173, 2009. ISSN 0163-5964.

BURGER, D.; GOODMAN, J. R.; KäGI, A. Memory bandwidth limitations of future microprocessors. In: *In Proceedings of the 23rd Annual International Symposium on Computer Architecture*. [S.l.: s.n.], 1996. p. 78–89.

BURGER, D. C.; GOODMAN, J. R.; KAGI, A. *The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors*. [S.l.], January 1995. Disponível em: <citeseer.ist.psu.edu/burger95declining.html>.

CHEN, W. et al. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In: *In Proceedings of the 24th International Symposium on Microarchitecture*. [S.l.: s.n.], 1991. p. 69–73.

CHISHTI, Z.; POWELL, M. D.; VIJAYKUMAR, T. N. Optimizing replication, communication, and capacity allocation in cmps. In: *INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*. [S.l.]: IEEE Computer Society, 2005. p. 357–368.

CONSTANTINOU, T. et al. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 33, n. 4, p. 80–91, 2005. ISSN 0163-5964.

DENNING, P. J. The locality principle. *Commun. ACM*, ACM, New York, NY, USA, v. 48, n. 7, p. 19–24, 2005. ISSN 0001-0782.

DYBDAHL, H.; STENSTRöM, P.; NATVIG, L. An lru-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. In: *MEDEA '06: Proceedings of the 2006 workshop on MEmory performance*. New York, NY, USA: ACM, 2006. p. 45–52. ISBN 1-59593-568-1.

FEDOROVA, A. et al. Performance of multithreaded chip multiprocessors and implications for operating system design. In: *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005. p. 26–26.

HALLNOR, E. G.; REINHARDT, S. K. A fully associative software-managed cache design. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 28, n. 2, p. 107–116, 2000. ISSN 0163-5964.

HARDAVELLAS, N. et al. Reactive NUCA: near-optimal block placement and replication in distributed caches. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 37, n. 3, p. 184–195, 2009. ISSN 0163-5964.

HILL, M. D. *Aspects of Cache Memory and Instruction Buffer Performance*. Tese (Doutorado) — EECS Department, University of California, Berkeley, Nov 1987. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5701.html>.

INOUE, A. *SPARC64 V Processor for Unix Servers - 2004*. [S.l.], 2004.

JIANG, S.; ZHANG, X. LIRS: An Efficient Low Inter-reference Recency Set Replacement to Improve Buffer Cache Performance. In: *Marina Del Rey*. [S.l.]: ACM Press, 2002. p. 31–42.

JOHNSON, T. L.; HWU, W.-m. W. Run-time adaptive cache hierarchy management via reference analysis. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 25, n. 2, p. 315–326, 1997. ISSN 0163-5964.

JOUPPI, N. P.; FULLYASSOCIATIVE, O. A. S. *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*. 1990.

KIM, C.; BURGER, D.; KECKLER, S. W. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2002. p. 211–222. ISBN 1-58113-574-2.

KOBAYASHI, J. M. Algoritmo de Substituição de Páginas Entropy. Monografia apresentada para conclusão da disciplina PCS-5720 - Sistemas Operacionais - 2007. 2007.

KOBAYASHI, J. M. Substituição de Linhas de Cache baseado em Entropia. Monografia apresentada para conclusão da disciplina PCS-5702 - Arquitetura de Computadores - 2008. 2008.

KUMAR, R.; JOUPPI, N. P.; TULLSEN, D. M. Conjoined-Core Chip Multiprocessing. In: *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2004. p. 195–206. ISBN 0-7695-2126-6.

KUMAR, R. et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In: *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2004. p. 64. ISBN 0-7695-2143-6.

L. NAYFEH B. A., O. K. H. A single-chip Multiprocessing. *IEEE*, 1997.

LIU, H. et al. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. IEEE Computer Society, Washington, DC, USA, p. 222–233, 2008.

MARINO, M. D. 32-core CMP with multi-sliced L2: 2 and 4 cores sharing a L2 slice. *Computer Architecture and High Performance Computing, Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 141–150, 2006. ISSN 1550-6533.

MCFARLING, S. Program optimization for instruction caches. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 17, n. 2, p. 183–191, 1989. ISSN 0163-5964.

NAYFEH, B. A.; HAMMOND, L.; OLUKOTUN, K. Evaluation of design alternatives for a multiprocessor microprocessor. In: *Proceedings of the 23th International Symposium on Computer Architecture*. [S.l.: s.n.], 1996. p. 67–77.

OLUKOTUN, K.; H., L. The Future of Microprocessors. *Queue*, ACM, New York, NY, USA, v. 3, n. 7, p. 26–29, 2005. ISSN 1542-7730.

PATTERSON, D. A.; HENNESSEY, J. L. *Computer Architecture - A Quantitative Approach*. 4th. ed. [S.l.]: Morgan Kaufman, 2007.

PETER, S. *The stateless state - Remembering what you were doing five minutes ago on the Web*. [S.l.], 2008.

QURESHI, M. K. et al. Adaptive insertion policies for high performance caching. In: *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007. p. 381–391. ISBN 978-1-59593-706-3.

QURESHI, M. K.; SULEMAN, M. A.; PATT. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In: *In Proceedings of the 2007 IEEE 13th international Symposium on High Performance Computer Architecture (February 10 - 14, 2007). HPCA. IEEE Computer Society*. [S.l.: s.n.], 2007.

REINEKE, J. et al. Timing predictability of cache replacement policies. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 37, n. 2, p. 99–122, 2007. ISSN 0922-6443.

SHANNON, E. C. A mathematical theory of communication. *Bell System Technical*, v. 27, p. 379–423 and 623–656, 1948.

SMITH, A. J. Cache memories. *ACM Computing Surveys*, v. 14, p. 473–530, 1982.

SOMOGYI, S. et al. Spatial Memory Streaming. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 34, n. 2, p. 252–263, 2006. ISSN 0163-5964.

SOMOGYI, S. et al. Spatio-temporal memory streaming. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 37, n. 3, p. 69–80, 2009. ISSN 0163-5964.

SPEC. *Standard Performance Evaluation Corporationl*. `http://www.spec.org/cpu2000/`.

SPEIGHT, E. et al. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. In: *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005. p. 346–356. ISBN 0-7695-2270-X.

SPRACKLEN, L.; ABRAHAM, S. G. Chip multithreading: opportunities and challenges. In: *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. [s.n.], 2005. p. 248–252. Disponível em: <http://dx.doi.org/10.1109/HPCA.2005.10>.

STOJCEV, M.; TOKI, T.; I., M. The limits of semiconductor technology and oncoming challenges in computer microarchitectures and architectures. 1000.

TULLSEN, D.; EGGERS, S. J.; LEVY, H. M. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: *In 22nd Annual International Symposium on Computer Architecture*. [S.l.: s.n.], 1995. p. 392–403.

WALL, D. W. Limits of instruction-level parallelism. In: . [S.l.: s.n.], 1991. p. 176–188.

WILKES, M. V. Slave memories and dynamic storage allocation. *EEE Transactions Computers EC-14*, ., p. 270–271, 1965.

WILTON, S. J. E.; JOUPPI, N. P. *An Enhanced Access and Cycle Time Model for On-Chip Caches*. [S.l.], 1993.

XIE, Y.; LOH, G. H. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In: *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009. p. 174–183. ISBN 978-1-60558-526-0.

ZHANG, M.; ASANOVIC, K. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In: *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005. p. 336–345. ISBN 0-7695-2270-X.