MÁRCIO GERMANO PERISSATTO

SIMULADOR DE MULTIPROCESSADORES SUPERESCALARES COM MEMÓRIA COMPARTILHADA: SMS-MC

MARINGÁ

2007

MÁRCIO GERMANO PERISSATTO

SIMULADOR DE MULTIPROCESSADORES SUPERESCALARES COM MEMÓRIA COMPARTILHADA: SMS-MC

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. João Angelo Martini

MARINGÁ

2007

Dados Internacionais de Catalogação-na-Publicação (CIP)
(Biblioteca Central - UEM, Maringá – PR., Brasil)

Perissatto, Márcio Germano
P446s Simulador de multiprocessadores superescalares com memória compartilhada : SMS-MC / Márcio Germano Perissatto. -- Maringá : [s.n.], 2007.
88 f. : il., figs.

Orientador : Prof. Dr. João Angelo Martini.
Dissertação (mestrado) - Universidade Estadual de Maringá. Programa de Pós-Graduação em Ciência da Computação, 2007.

1. Simulador de multiprocessadores superescalares de memória compartilhada. 2. Memória compartilhada. 3. Coerência de cache. 4. SimpleScalar. 5. SMS-MC. 6. Arquitetura de computadores. I. Universidade Estadual de Maringá. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22.ed. 004.22

MÁRCIO GERMANO PERISSATTO

SIMULADOR DE MULTIPROCESSADORES SUPERESCALARES COM MEMÓRIA COMPARTILHADA: SMS-MC

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Aprovado em 04/09/2007.

BANCA EXAMINADORA

Prof. Dr. João Angelo Martini
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Ronaldo Augusto de Lara Gonçalves
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. José Hiroki Saito
Universidade Federal de São Carlos – DC/UFSCar

Dedico esta dissertação à pessoa mais importante da minha vida, Emília, minha esposa, que com sua paciência e amor me ajudou a chegar até aqui, vencendo mais esta etapa de nossas vidas.

Agradecimentos

Dedico meus sinceros agradecimentos para:

– Deus, pelo dom da vida;

– meus pais, por abrirem as portas deste mundo maravilhoso e me ensinarem o caminho dos estudos;

– a minha esposa Emília, pela revisão e incentivo neste trabalho;

– o professor doutor João Angelo Martini, pela orientação e incentivo;

- o professor doutor Ronaldo A. L. Gonçalves, pela ajuda nos momentos de dúvida na codificação e pela confiança;

– a coordenadora do curso de Mestrado em Ciência da Computação, professora doutora Itana Maria Gimenez, pelo apoio sempre manifestado;

– o professor doutor Candido Ferreira Xavier, pela oportunidade de ingressar no mundo da computação paralela;

– ao meu grande amigo Bruno de Pádua Melo pela força no Inglês nas horas difíceis;

– a Rede de Supermercados São Francisco, empresa no qual trabalho, pela apoio para a conclusão deste curso;

– todos os colegas do Mestrado em Ciência da Computação da UEM.

“Ele não sabia que era impossível.

Foi lá e fez.”

Jean Cocteau

Resumo

A busca por maior poder computacional tem motivado diversos pesquisadores da área de Arquitetura de Computadores a dedicar grandes esforços para vencer as limitações físicas impostas pelos equipamentos disponíveis atualmente.

O estudo de máquinas paralelas é foco de esforços de diversos pesquisadores desta área, assim como a simulação destas máquinas paralelas.

Desta forma, apresentamos neste trabalho o projeto e implementação de um Simulador de Multiprocessadores Superescalares de Memória Compartilhada (SMS-MC), baseado na estrutura do simulador SimpleScalar, um consagrado simulador de Arquitetura de Computadores.

O simulador permite realizar diversas análises relacionadas à memória compartilhada e protocolos de coerência de *cache*, gerando estatísticas para os *benchmarks* que nele podem ser executados.

Apresentamos ainda alguns *benchmarks* que foram desenvolvidos para demonstrar a utilização do simulador, mostrando suas funcionalidades e possibilidades de uso.

Finalizamos este trabalho apresentando algumas possíveis expansões para o simulador SMS-MC, que podem ser implementadas para aumentar a abrangência da ferramenta.

Palavras-Chaves: Memória Compartilhada, Coerência de Cache, SimpleScalar.

Abstract

The quest for more computational power have motivated several researchers in the computer architecture area to dedicate a large amount of effort in overcoming the physical barriers imposed by the current available equipments.

The study of parallel machines is the focus of the efforts of many researchers in this area, and so is the simulation of these parallel machines.

We introduce in this work, the project and the implementation of a superscalar multiprocessor simulator with shared memory (SMS-MC), based on the SimpleScalar simulator structure, a well known simulator for computer architecture.

The simulator allows us to perform numerous analises over the shared memory and the cache coherence protocols, generating statistics for the benchmarks that can later be used in the simulator.

We present here some of the benchmarks that were developed to demonstrate the usage of the simulator, showing its functions and its uses.

We conclude this work by presenting some expansion possibilities for the SMS-MC simulator, which can be implemented to improve.

Key words: Shared Memory, Cache Coherence, SimpleScalar.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Neste capítulo apresentamos uma motivação para o estudo de arquiteturas paralelas, assim como uma classificação das mesmas baseada no sub-sistema de memória. Apresentamos também uma sucinta motivação ao uso de simuladores e uma visão geral da seqüência deste trabalho.

O aumento do poder de processamento tem sido cada vez mais almejado por diversas áreas da computação. Enquanto alguns pesquisadores buscam melhorar as técnicas empregadas no desenvolvimento dos processadores, tentando aumentar o desempenho (otimizando os circuitos lógicos internos), outros buscam melhorar técnicas de utilização destes *hardwares*, tentando na maioria das vezes agrupar equipamentos de menores portes visando obter maior capacidade computacional no conjunto do que com os *hardwares* isolados. Outros ainda buscam também dividir as tarefas em sub-tarefas menores, para que possam ser executadas em *hardwares* com diversos processadores, objetivando concluir o processamento num tempo menor.

Na área de Arquiteturas Paralelas busca-se fornecer meios e técnicas de utilizar *hardwares* com diversos processadores para conseguir o desempenho esperado. Máquinas Paralelas são foco de pesquisa, pois são capazes de atender a demanda de uma vasta gama de aplicações que requisitam alto desempenho.

Máquinas Paralelas podem ser classificadas em três categorias, com base no sub-sistema de memória:

- Memória Distribuída ((NITZBERG; LO, 1991), (ZHOU et al., 1992), (MILTON, 1998)): Neste modelo cada processador da arquitetura possui sua própria memória local, de forma que o programa executado em cada processador possui um espaço de endereçamento local, invisível aos outros processadores. Todas as tarefas de comunicação inter-processos devem ser feitas via troca de mensagens e fica a cargo do programador gerenciar este trânsito dos dados;

- Memória Compartilhada ((COX et al., 1994), (DING; BHUYAN, 1992), (LEBLANC;

MARKATOS, 1992)): Neste modelo todos os processadores compartilham o mesmo espaço de endereçamento, tornando possível que dados manipulados por um processador sejam visíveis aos demais processadores. As rotinas de comunicação inter-processos são feitas através do acesso a variáveis compartilhadas, o que permite que o programador não se preocupe com a forma na qual os dados trafegam entre os processadores ;

- Memória Compartilhada Distribuída ((STUMM; ZHOU, 1990), (HENNESSY; HEINRICH; GUPTA, 1999), (DWARKADAS et al., 1999)): Neste modelo, cada processador possui uma memória local, entretanto consegue acessar os espaços de endereçamento dos demais processadores. Todas as tarefas de movimentação e acesso aos dados são implementadas diretamente pelo *hardware*, utilizando rotinas de envio e recepção de mensagens. Este modelo tem sido foco de diversas pesquisas atuais (GOLAB et al., 2007), (ROGERS; PRVULOVIC; SOLIHIN, 2006), (GOLAB; HENDLER; WOELFEL, 2006), (ZHANG et al., 2006), (BARTON et al., 2006), (BASUMALLIK; EIGENMANN, 2006), (MALEN; LOTTIANUX, 2002), (THANALAPATI; DANDAMUDI, 2001), (STOLLER, 1998), (CIERNIAK; LI, 1995), pois busca associar as melhores qualidades dos modelos de memória compartilhada e memória distribuída, adicionando a possibilidade do programador poder distribuir os dados nas memórias locais da maneira que for mais interessante.

Diversos *hardwares* já são equipados com mais de um processador, mas devido ao custo nem sempre são acessíveis à maioria dos estudantes e até mesmo a algumas organizações de ensino, principalmente públicas. Desta forma, torna-se difícil investigar as técnicas e o funcionamento dessas arquiteturas multiprocessadas sem possuir um equipamento para trabalhar e realizar testes.

Tentando resolver este problema da dificuldade de se ter uma máquina paralela para pesquisas é que surge a idéia de se utilizar simuladores de arquiteturas, de memória, de redes de interconexão etc.

Dentro de um grande universo de simuladores de arquitetura de computadores, o SimpleScalar (AUSTIN; LARSON; ERNST, 2002) se destaca por ser um *toolkit* de simulação que permite simular diversos níveis de abstração, incluindo memória, *caches*, previsores de desvios, *pipelines* entre outros dispositivos de *hardware*. A ferramenta SimpleScalar tem prestígio internacional, sendo intensamente utilizada por renomados pesquisadores da área de arquitetura de computadores. Diversos pesquisadores baseiam seus experimentos nesse *toolkit*.

Neste contexto, o objetivo do presente trabalho consistiu em viabilizar uma ferramenta de simulação para Arquitetura Paralela com Memória Compartilhada denominada SMS-MC (Simulador de Multiprocessadores Superescalares com Memória Compartilhada), com base no

núcleo do SimpleScalar.

Nesta dissertação é apresentada uma contextualização sobre os fundamentos de sistemas multiprocessados, modelos de memória (mais especificamente o modelo de memória compartilhada), simulação, incluindo uma revisão sobre os principais trabalhos relacionados dentro do escopo de simulação de arquiteturas paralelas e memória compartilhada. São discutidos também os protocolos de coerência de *cache*, a modelagem e implementação do SMS-MC e os testes e resultados aplicados a este simulador.

A estrutura desta dissertação está organizada da seguinte forma: o capítulo 2 apresenta uma revisão sobre Arquiteturas Paralelas, detalhando sua importância e apresentando a principal taxonomia para Arquiteturas Paralelas; o capítulo 3 apresenta os modelos de memória disponíveis em Arquiteturas Paralelas, detalhando os principais tipos de protocolos utilizados; o capítulo 4 apresenta uma revisão sobre os trabalhos relacionados; o capítulo 5 apresenta o Simulador SMS-MC proposto neste trabalho, descrevendo sua funcionalidade, implementação e recursos disponíveis; o capítulo 6 apresenta os resultados obtidos com o uso do simulador proposto, incluindo alguns exemplos de *benchmarks* desenvolvidos para o mesmo; o capítulo 7 discute as conclusões da pesquisa, as perspectivas para o uso da ferramenta proposta e possíveis trabalhos futuros baseados na ferramenta apresentada.

.

2 *Arquiteturas Paralelas*

Neste capítulo apresentamos uma das taxonomias mais aceitas na área de arquitetura de computadores, a taxonomia de Flynn, incluindo suas estruturas e principais exemplos. Situamos o simulador apresentado neste trabalho dentro desta taxonomia e introduzimos o problema da coerência de cache nas máquinas paralelas com memória compartilhada e caches individuais.

É fácil notar que sistemas computacionais com elevado poder de processamento têm se tornado cada vez mais importantes e necessários nas atividades e pesquisas atuais. No entanto, com o constante crescimento deste poder computacional, cada vez mais a velocidade de processamento de instruções, acesso à memória e armazenamento está aproximando-se do seu limite físico.

Como tentativa de contornar este problema, pesquisadores buscam desenvolver novos métodos para atingir maior poder computacional, sendo um deles a tentativa de agrupar recursos computacionais (processadores, memórias) de maneira que possam trabalhar em sinergia, provendo maior poder computacional do que um único processador poderia fornecer. Quase que naturalmente o paralelismo torna-se uma das alternativas para a superação desta limitação.

Sendo as Arquiteturas Paralelas foco principal deste trabalho, faz-se necessária uma discussão acerca da taxonomia de Arquiteturas Paralelas.

2.1 Taxonomia de Flynn

Provavelmente a classificação de arquiteturas de computadores mais amplamente conhecida seja a sugerida por Flynn (FLYNN, 1972). A taxonomia de Flynn tem como base os conceitos de fluxo de instruções e fluxo de dados. A classificação de Flynn identifica quatro categorias de arquiteturas:

- **SISD (Single Instruction / Single Data)** : Este modelo representa as máquinas com

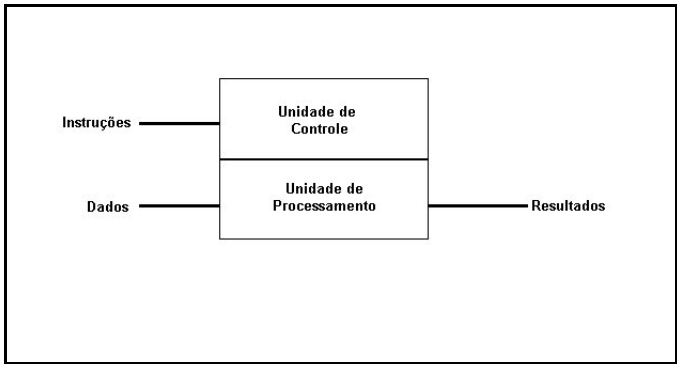

Figura 2.1: *Arquitetura SISD*

um único fluxo de instruções e um único fluxo de dados. Estas máquinas não exploram paralelismo de dados nem de instruções. A execução ocorre de forma seqüencial. Neste modelo inclui-se a máquina originalmente proposta por von Neumann, os PCs e alguns *mainframes*. A Figura 2.1 mostra a estrutura básica da classe SISD.

- **SIMD (Single Instruction / Multiple Data)** : Este modelo inclui máquinas com vários processadores, capazes de executar uma mesma instrução sobre um conjunto de dados independente em cada processador. Alguns processadores, conhecidos como processadores vetoriais, enquadram-se nesta categoria, sendo utilizados principalmente em operações matemáticas que envolvam matrizes e vetores. A Figura 2.2 ilustra a estrutura desta classe de máquinas. Exemplos desta categoria são: ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP e CM-1.

- **MISD (Multiple Instruction / Single Data)** : Este modelo representa as máquinas capazes de executar diversas instruções sobre uma mesma seqüência de dados. Acredita-se que esta classe de máquinas seja praticamente vazia. A Figura 2.3 mostra a estrutura das máquinas da classe MISD.

- **MIMD (Multiple Instruction / Multiple Data)** : Este modelo, o mais avançado na taxonomia, representa as máquinas capazes de executar diferentes instruções sobre diferentes massas de dados. Nesta categoria enquadram-se praticamente todas as máquinas paralelas, que no geral utilizam diversos processadores e memórias, sendo controladas por um único núcleo de sistema operacional. A Figura 2.4 expõe a estrutura básica de uma máquina desta classe. Algumas características que subdividem as máquinas do modelo MIMD são: quantidade de processadores, redes de interconexão, tipos de memória e *caches*.

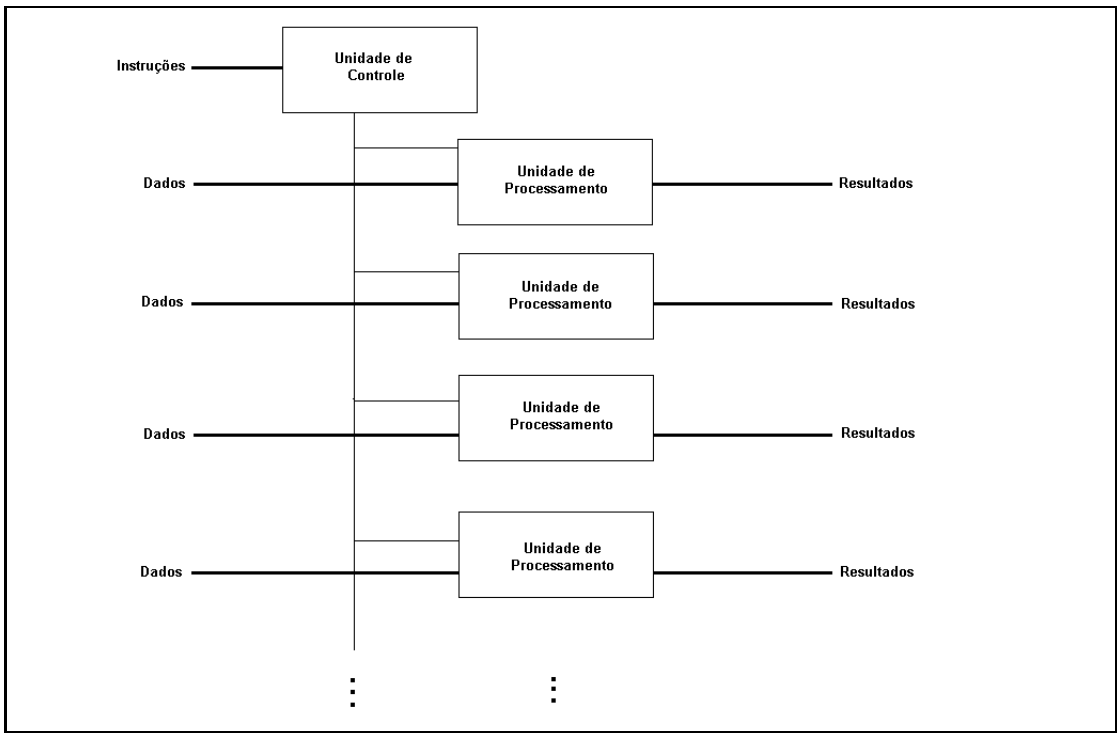

Figura 2.2: *Arquitetura SIMD*

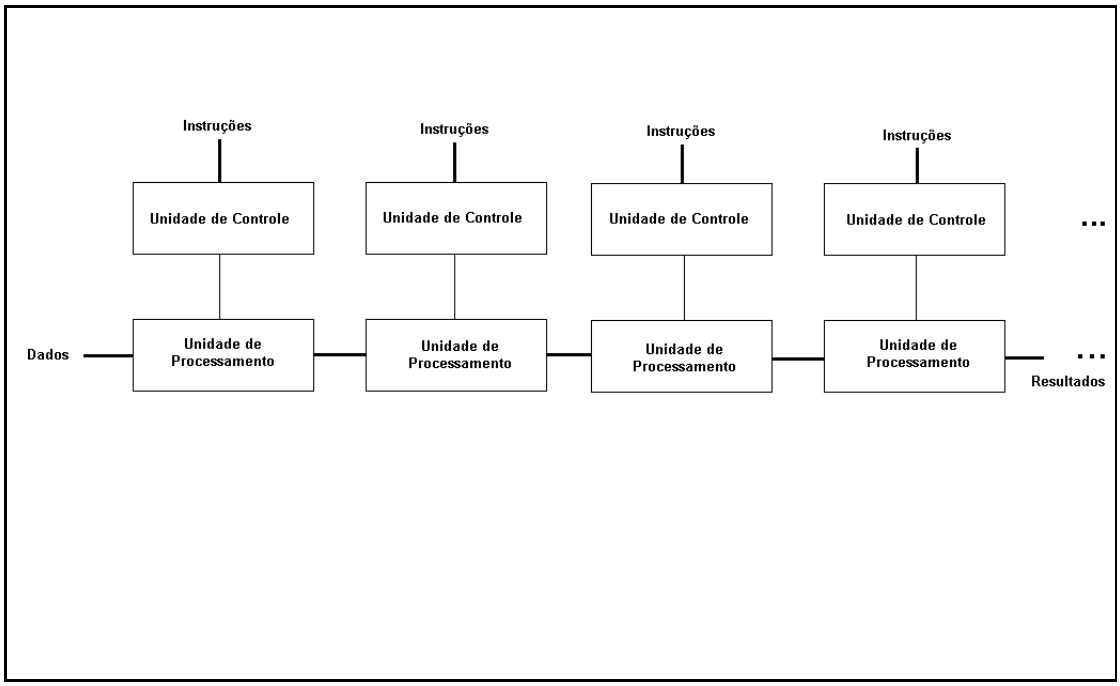

Figura 2.3: *Arquitetura MISD*

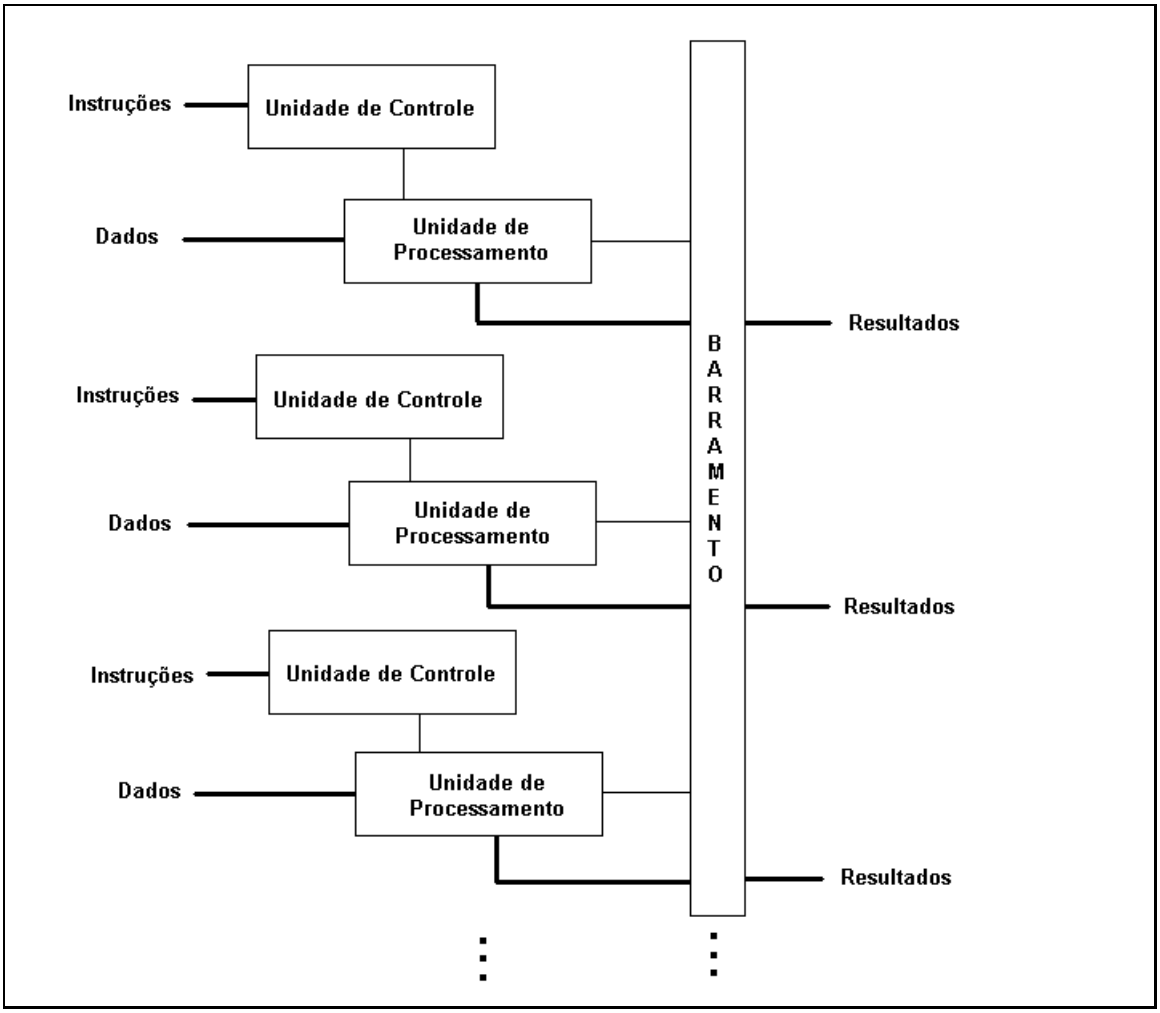

Figura 2.4: *Arquitetura MIMD*

Neste trabalho, apresentamos a implementação de um simulador de arquiteturas que enquadra-se no modelo MIMD. Neste modelo são muito comuns dois tipos de sub-sistemas de memória: os baseados em memória distribuída e os baseados em memória compartilhada.

Os multicomputadores de memória distribuída são caracterizados pela presença de um espaço de endereçamento exclusivo a cada processador, que não pode ser acessado pelos outros processadores. Uma de suas vantagens é o baixo custo quando deseja-se ampliar a capacidade da memória, principalmente quando comparado com o outro modelo. Entretanto, quando utilizados com aplicações que exigem muita comunicação inter-processos, apresentam queda de desempenho.

Os sistemas com memória compartilhada possuem espaços de endereçamento comuns a todos os processadores. Estão sujeitos a diversos problemas relacionados a coerência dos dados, considerando que cada processador pode executar instruções independentes, tentando alterar o mesmo endereço de memória. Uma das principais vantagens deste tipo de máquina é que fornece um modelo de programação simples, facilitando o desenvolvimento de aplicativos paralelos.

Considerando a disputa que ocorre nos multiprocessadores de memória compartilhada pelo uso destes recursos compartilhados, a latência do acesso à memória tende a aumentar, prejudicando seu desempenho.

Uma das soluções diretas para este problema é o uso de *caches*, que podem ser tanto privadas quanto compartilhadas, sendo as caches privadas mais interessantes pois tendem a atender a maioria das referências à memória localmente.

Entretanto, a utilização de *caches* privadas introduz um outro problema no contexto, que é a "Coerência de Cache". Este problema ocorre devido à possibilidade de existirem múltiplas cópias de um mesmo dado em diferentes *caches* simultaneamente, possibilitando que haja uma visão inconsistente da memória.

Para solucionar este problema de "Coerência de Cache" podem ser utilizadas soluções baseadas em *software* ou *hardware*. Comercialmente os métodos baseados em *hardware* são mais utilizados, sendo conhecidos como "Protocolos de Coerência de Cache".

A implementação do simulador descrito neste trabalho inclui dois modelos de protocolos de coerência de *cache*, ou seja, duas soluções para coerência baseadas em *hardware*. A estrutura disponibilizada neste simulador permite que novos protocolos sejam incluídos sem qualquer interferência nos recursos que já estão disponíveis no mesmo.

2.2 Alguns exemplos práticos

Alguns projetos de máquinas paralelas vem sendo desenvolvidos por grupos de pesquisa no mundo todo. Apresentamos, a seguir, alguns exemplos que representam uma parte da pesquisa na área.

2.2.1 Arquitetura FLASH

A arquitetura FLASH (*FLexible Architecture for SHared memory*) (KUSKIN et al., 1994) foi um projeto desenvolvido pela Universidade de Stanford, com objetivo de fornecer um multiprocessador escalável, capaz de suportar diversos modelos de comunicação, incluindo memória compartilhada e protocolos de troca de mensagens.

A chave deste projeto foi o desenvolvimento de um controlador, chamado MAGIC (*Memory And General Interconnect Controller*), que é um chip baseado no *Protocol Processor*, uma *engine* baseada no projeto do processador TORCH.

O MAGIC executa basicamente duas tarefas: movimentação de dados e manipulação de estados. Assim, ele explora a independência que existe entre as tarefas separando o controle do processamento dos dados.

O projeto FLASH foi desenvolvido pela Universidade de Stanford em parceria com o ITO (*Information Technology Office*) e colaboração da *MIPS Technologies*.

2.2.2 Hydra

Hydra (WULF, 1981) é o projeto de uma nova microarquitetura de multiprocessadores com caches compartilhadas, novos mecanismos de sincronização, tecnologia de circuitos integrados avançados e tecnologia de compiladores paralelos. No Hydra, quatro processadores de alto desempenho são integrados numa única pastilha.

O Hydra usa um único chip de cache compartilhada, melhorando assim o desempenho relativo à largura de banda e latência entre os múltiplos processadores. Utilizando um único chip, a comunicação inter-processos e a latência na sincronização são comparadas com implementações de multiprocessadores baseados em barramento.

2.2.3 TRIPS

O projeto TRIPS (SANKARALINGAM et al., 2006) é um projeto que visa desenvolver uma nova classe de tecnologia escalável e de multiprocessadores de alto desempenho chamados EDGE (*Explicit Data Graph Execution*) (BURGER et al., 2004).

O primeiro membro da arquitetura EDGE foi chamado de TRIPS, sendo desenvolvido na Universidade do Texas, em Austin. O TRIPS fornece concorrência de instruções de alto-nível e conseqüentemente alto desempenho, sem necessitar de mudanças no modelo de programação.

2.2.4 Simultaneous Multithreading

O *Simultaneous Multithreading* (SMT) (Susan J. Eggers and Joel S. Emer and Henry M. Levy and Jack L. Lo and Rebecca L. Stamm and Dean M. Tullsen, 1997) é um modelo de processador que combina a tecnologia de *hardware multithreading* com processadores superescalares, permitindo que múltiplas *threads* tenham instruções executadas em cada ciclo.

Diferentemente de muitas outras arquiteturas *multithreads*, em que um único contexto está ativo num dado ciclo, o SMT permite que todos os contextos de *threads* possam simultaneamente competir por recursos.

O projeto do *Simultaneous Multithreading* foi desenvolvido pela Universidade de Washington.

2.2.5 Projeto Wisconsin Wind Tunnel

O projeto Wisconsin Wind Tunnel (REINHARDT et al., 1993) é focado no desenvolvimento de máquinas paralelas com suporte a memória compartilhada.

O projeto é dividido em três fases:

- a primeira fase examina as memórias compartilhadas colaborativas, que simplificam o hardware de memória compartilhada, permitindo ao software gerenciar a movimentação dos dados;
- a segunda fase propõe uma interface que permite aos programadores implementar e utilizar troca de mensagens, memória compartilhada ou uma combinação híbrida entre estes dois modelos de comunicação inter-processos;

- a terceira fase busca utilizar previsão e especulação para aumentar o desempenho, reduzindo a latência de comunicação.

2.2.6 Projeto MIT Alewife

O Alewife (AGARWAL et al., 1995) é um multiprocessador de grande escala que integra coerência de cache, memória compartilhada distribuída e troca de mensagens num único *framework* de hardware integrado.

Cada nó do Alewife usa uma unidade de inteiros *Sparcle* de 33MHz, 64KBytes de cache e 4MBytes de memória principal compartilhada. Os nós trocam mensagens entre si através de uma rede de interconexão *mesh*.

3 *Memória Compartilhada*

Neste capítulo apresentamos uma visão geral sobre memórias compartilhadas e o problema da coerência de cache que surge nesta arquitetura. Apresentamos algumas soluções baseadas em software e hardware. Os Directory Protocols e os Snoopy Protocols são apresentados e exemplificados. Dois exemplos de protocolos baseados nas políticas write-invalidate e write-update são descritos e detalhados.

Este tipo de memória, muito utilizada em máquinas multiprocessadas, permite que todos os processadores utilizem o mesmo espaço de endereçamento, facilitando o trabalho do programador, pois evita que ele tenha que se preocupar com a maneira com a qual os dados trafegam entre os processadores (como é o caso da memória distribuída), tornando mais simples as tarefas de comunicação inter-processos.

Entretanto, o programador precisa atentar-se à maneira como os dados são divididos entre os processadores, evitando que ocorram conflitos nos acessos e reduzindo as tarefas de sincronização.

Os sistemas de memória compartilhada podem ser implementados tanto com memórias centralizadas quanto com memórias distribuídas, sendo que em ambos os casos, geralmente existe uma memória *cache* associada a cada processador visando reduzir a latência.

Isso cria um outro tipo de problema nesta arquitetura: a coerência de *cache*. Como cada processador possui sua *cache* local, diversas cópias de um dado compartilhado podem existir. Para tratar este problema existem os protocolos de coerência de *cache*, que servem para realizar as operações nas *caches* locais de modo a manter a coerência dos dados.

Nestes sistemas de memória compartilhada, as operações de `leitura / escrita` precisam ser controladas para que os dados mantenham-se coerentes. Assim, instruções de `leitura` concorrentes podem ser executadas sem muita preocupação pelos programadores, entretanto, todas as operações de `escrita` em memória precisam ser implementadas de forma atômica, evitando que valores escritos sejam perdidos.

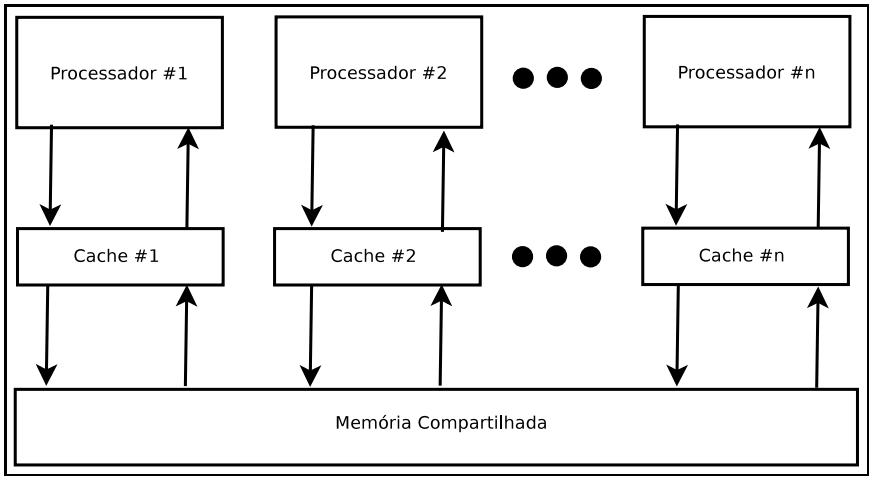

Figura 3.1: *Memória Compartilhada com Cache Individual por Processador*

Estes controles que devem existir numa arquitetura de memória compartilhada retiram do programador a necessidade de preocupar-se com detalhes de acessos e transferências de dados, mas aumentam a complexidade do *hardware* que fará o controle destes acessos simultâneos.

3.1 Coerência de Cache

Como explicado anteriormente, o problema da "Coerência de Cache" ocorre quando utiliza-se memória compartilhada com *caches* individuais nos múltiplos processadores. Desta forma, diversas cópias de um mesmo dado podem existir nas memórias *caches* simultaneamente. Caso os processadores realizem livremente atualizações em suas *caches*, uma visão inconsistente da memória será formada, o que pode induzir a um mal funcionamento do programa em execução.

As soluções para este problema podem ser desenvolvidas tanto em *software* quanto em *hardware*. As soluções baseadas em *software* limitam-se a separar os dados que podem ser alocados nas memórias *caches* e os que não podem. Resumidamente, um novo rótulo é adicionado aos endereços, dizendo se são "*cachebles*" ou "*non-cachebles*". Assim, dados compartilhados não podem ser armazenados nas *caches* dos processadores, garantindo que não existam diversas cópias deste mesmo dado.

As soluções baseadas em *hardware*, também conhecidas como "Protocolos de Coerência de Cache" são comercialmente mais utilizadas (STENSTRöM, 1990). Estas soluções tratam o problema de coerência de *cache* utilizando reconhecimento dinâmico de inconsistência dos dados em tempo de execução. Uma das grandes vantagens deste tipo de implementação é que ela não exige qualquer responsabilidade do programador ou do compilador relativa à manutenção de coerência de *cache*.

Entre os protocolos implementados em *hardware* existe ainda uma divisão, que representa a forma com que os dados são distribuídos entre os processadores: os *Directory Protocols* (baseados num diretório centralizado ou distribuído) e os *Snoopy Protocols* (baseados em barramento compartilhado).

Outra característica que diferencia os protocolos de coerência de cache é a quantidade de estados que cada protocolo implementa. Os protocolos mais simples e que apresentam desempenho um pouco reduzido possuem de três a cinco estados e exigem maior complexidade na tarefa de manter a coerência. Os protocolos mais elaborados possuem quantidades maiores de estados e oferecem um desempenho superior, exigindo menor complexidade para manter a coerência.

As próximas duas sub-seções discutem *Directory Protocols* e *Snoopy Protocols*.

3.1.1 Directory Protocols

Este tipo de protocolo possui uma estrutura (em forma de diretório, tabela) que armazena informações sobre os dados contidos nas diversas *caches* locais, ou seja, possui toda a informação sobre em qual *cache* determinado dado está e qual o seu estado.

Sempre que uma nova requisição de `leitura` ou `escrita` é feita para uma memória *cache* local, o diretório é verificado para tomar-se as devidas ações nas *caches* remotas de modo a manter a coerência dos dados.

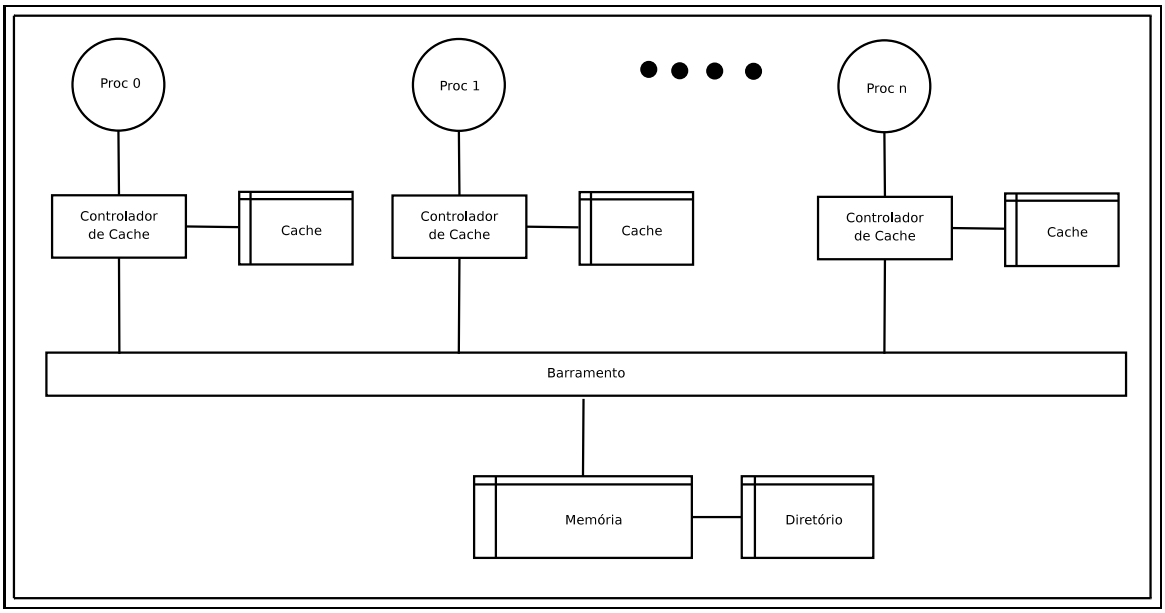

Figura 3.2: *Estrutura básica de uma cache que utiliza protocolo baseado em diretórios.*

De modo geral, os diretórios são armazenados na memória principal e representam cada

bloco de memória como um mapa de *bits*, indicando a presença do bloco nas diversas *caches*. Cada *bit* deste vetor representa o estado do bloco em uma das *caches* locais.

A Figura 3.2 mostra a estrutura básica de um sistema com diversos processadores e suas respectivas *caches* locais, assim como a estrutura do protocolo baseado em diretórios.

Este tipo de protocolo apresenta a desvantagem de utilizar um único ponto central para pesquisa no diretório e ainda o elevado *overhead* de comunicação entre os controladores de *cache* locais e o diretório centralizado.

Por outro lado, diferentemente dos protocolos *Snoopy* (descritos a seguir), as operações realizadas pelo controlador de *cache* não precisam enviar mensagens utilizando *broadcast* para as demais *caches*, direcionando as mensagens somente para as *caches* que sabe-se conterem os dados compartilhados que estejam sendo manipulados no momento.

Um exemplo clássico deste protocolo é a implementação do DASH, criado pela Universidade de Stanford para seu multicomputador que usa uma rede bi-dimensional para comunicação entre os processadores (LENOSKI et al., 1990).

3.1.2 Snoopy Protocols

Os *Snoopy Protocols* utilizam um barramento compartilhado onde enviam mensagens de *broadcast* para avisar as demais *caches* sobre suas operações nos dados. Todas as *caches* "escutam" as mensagens deste barramento e verificam seus dados para avaliar se necessitam realizar alguma ação para manter a coerência dos dados no sistema como um todo.

Antes de realizar qualquer operação sobre um dado na *cache*, o protocolo verifica se é necessário consultar as demais *caches* sobre seus estados e caso necessário encaminha as devidas mensagens pelo barramento. Dependendo das mensagem recebidas, o controlador da *cache* aguarda as confirmações das outras *caches* para que possa efetivar a operação.

A principal diferença entre uma *cache Snoopy* e uma *cache* em sistemas uni-processadores está no controlador de *cache* e o uso do barramento compartilhado. No caso das *caches Snoopy*, o controlador de *cache* é uma máquina de estados finitos que implementa um protocolo de coerência de *cache*. A Figura 3.3 mostra o diagrama de transições de estados que exemplifica o funcionamento da máquina de estados finitos utilizada na implementação dos protocolos *write-invalidate*.

Este tipo de protocolo difere dos baseados em diretório principalmente pelo fato de não utilizar um controle centralizado, sendo caracterizado pelo controle distribuído das *caches*.

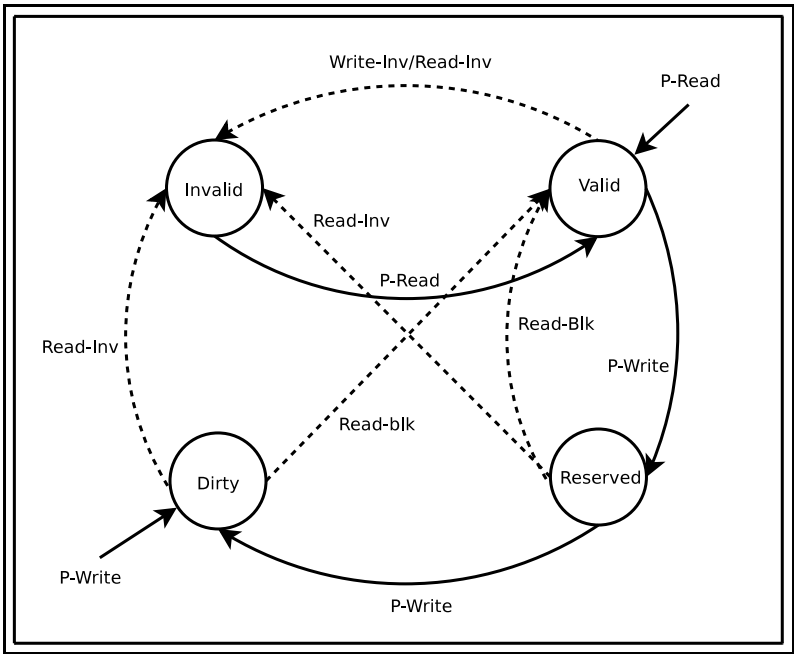

Figura 3.3: *Diagrama de Transições de estados dos protocolos com política write-invalidate.*

3.2 Políticas de Coerência

Os *Directory Protocols* e os *Snoopy Protocols* podem possuir duas políticas de escrita fundamentais:

- *Write-Invalidate*: Este tipo de implementação, também conhecida como "Baseada em Invalidação", envia mensagens de invalidação às demais *caches* caso uma operação de `escrita` seja solicitada na *cache* local. Permite que diversas *caches* possuam o dado no modo "`leitura`", mas somente uma *cache* com o dado no modo "`escrita`". Toda escrita em blocos compartilhados começa com a invalidação do mesmo dado em todas as outras *caches*. Depois que a *cache* obtém o privilégio de escrever no bloco, todas as demais escritas subseqüentes deste mesmo bloco que forem necessárias nesta mesma *cache* podem ser executadas até que outra *cache* solicite permissão de `leitura` ou `escrita` para este bloco. São exemplos de implementação deste tipo de protocolo: *Synapse* (RAINA, 1992), *Berkeley* (KATZ et al., 1985) e *Ilinois MESI* (Mark S. Papamarcos and Janak H. Patel, 1984).

- *Write-Update*: Este tipo de implementação, também conhecida como "Baseada em Atualização", envia mensagens com o dado atualizado para todas as demais *caches*, permitindo que o dado permaneça coerente em todas as *caches* do sistema. São exemplos de implementação deste protocolo: *Firefly* (THACKER; STEWART, 1987) e *Dragon* (ARCHIBALD; BAER, 1986).

O funcionamento dos protocolos baseados na política *write-invalidate*, geralmente implementados como protocolos de quatro estados (*MESI*, descrito nas seções seguintes) é baseado nas seguintes primitivas (STENSTRöM, 1990):

- *Read hit*: sempre podem ser executados localmente nas *caches*, não ocorrendo qualquer mudança de estado na *cache* local ou nas *caches* remotas. Isto ocorre porque o dado válido já está disponível na *cache* local;
- *Read miss*: se nenhuma cópia alterada do dado existe em qualquer outra *cache*, o valor do dado na memória é o atual e válido. Este valor (da memória) é consistente e pode ser passado à *cache* que precisa do dado para fornecer a `leitura` solicitada por seu processador. Se uma cópia alterada do dado existir em qualquer outra *cache*, a *cache* com o dado alterado atualiza a memória e o dado é enviado para a *cache* solicitante. Neste último caso, ambas as *caches* terão dados válidos e a memória estará atualizada também, isto porque a operação era de `leitura`;
- *Write hit*: se a cópia do dado compartilhado está presente na *cache* e está no estado *exclusivo* ou *alterado*, então a operação de `escrita` pode ser realizada localmente na *cache* e o novo estado do bloco de dados da *cache* será *alterado*. Se o bloco de dados existe em alguma das outras *caches* remotas, uma mensagem solicitando a invalidação do bloco deve ser enviada para todos os outros controladores de *cache*, assim, a operação de `escrita` pode ser executada localmente e o novo estado passa a ser *alterado*;
- *Write miss*: o dado existe em alguma das *caches* remotas com o estado *alterado*, exigindo que o valor do dado na memória seja atualizado. Uma solicitação de invalidação de leitura é enviada a todas as *caches*, garantindo que ninguém possua qualquer cópia do dado, mesmo que em modo *leitura*, pois a *cache* local pretende realizar uma operação de `escrita` no bloco. Após as confirmações, o dado é atualizado localmente na *cache* e seu novo estado passa a ser *alterado*;
- *Replacement*: se o dado encontra-se alterado na *cache* este valor precisa ser atualizado na memória principal.

Já os protocolos baseados na política *write-update* funcionam baseados nas seguintes primitivas (STENSTRöM, 1990):

- *Read hit*: sempre podem ser executados localmente nas *caches*, não ocorrendo qualquer mudança de estado na *cache* local ou nas *caches* remotas. Isto ocorre porque o dado válido já está disponível na *cache* local;

- *Read miss*: caso hajam cópias compartilhadas do bloco, a *cache* com o dado válido fornece o dado para a *cache* que precisa do mesmo. Caso o estado do bloco seja *alterado*, a *cache* remota que possui o dado atualiza a memória (realizando um *write-back*) e muda o estado do bloco na sua *cache* para *válido*. Assim, a *cache* solicitante pode ler o bloco da memória e ambas passam a possuir o bloco com estado *compartilhado*;

- *Write hit*: se o bloco já existir na *cache* local, com os estados *válido*, *exclusivo* ou *alterado*, a operação de `escrita` pode ser realizada normalmente na *cache* local, e o novo estado do bloco passa a ser *alterado*. Se o estado do bloco é *compartilhado*, todas as outras cópias do bloco são atualizadas, incluindo a memória. Neste caso todas as *caches* que possuem o bloco passam a utilizar o estado *compartilhado* para o bloco;

- *Write miss*: o dado não está presente na *cache* local. Caso outras *caches* possuam o dado, ele é então atualizado na memória e devolvido à *cache* solicitante que pode então realizar sua atualização no valor do dado e novamente replicar sua alteração para todas as demais *caches* que possuam o bloco. Neste caso, todas as *caches* que possuem o bloco utilizam o estado *compartilhado* para ele. Caso nenhuma outra *cache* possua o dado no momento da solicitação de `escrita`, o controlador de *cache* lê o dado da memória, atualiza-o e utiliza o estado *alterado* na sua cópia do dado;

- *Replacement*: se o dado encontra-se alterado na *cache* este valor precisa ser atualizado na memória principal.

Os *Snoopy Protocols* são muito populares pela sua facilidade de implementação e utilizados em *Symmetric Multiprocessors Systems* como o Alliant FX da *Alliant Computer Systems* que utiliza políticas *write-invalidate* para manter a coerência de suas *caches* (STENSTRöM, 1990).

Os *Snoopy Protocols* baseados em invalidação são utilizados em processadores das fabricantes Intel e AMD, duas das maiores fabricantes de processadores no mundo atualmente. O Ilinois MESI é uma das implementações desta categoria de protocolos que melhor representa a classe, sendo, portanto, detalhado de maneira mais aprofundada na seção seguinte. Este protocolo também é implementado no simulador SMS-MC.

3.3 O Protocolo Ilinois MESI

O protocolo MESI (também conhecido como *Protocolo de Ilinois*) possui este nome devido aos seus quatro possíveis estados: *Modified, Exclusive, Shared e Invalid*. É o principal repre-

sentante da categoria dos protocolos baseados em invalidação, podendo ser implementado tanto como um protocolo "*Snoopy*" quanto um protocolo "*Directory based*".

É um dos protocolos mais utilizados comercialmente, sendo inclusive utilizado pela Intel para suportar a implementação de *caches write-back* da linha *Pentium* em substituição às *caches write-through* da linha *486*.

Os estados utilizados no protocolo MESI possuem um significado específico para cada letra:

- ***M**odified:* Indica que a linha de dados da *cache* encontra-se modificada, divergindo do conteúdo da memória. Neste estado, o dado está disponível somente na *cache* local.
- ***E**xclusive:* Indica que o dado encontra-se presente somente nesta *cache* e que possui o mesmo valor que existe fisicamente na memória.
- ***S**hared:* Indica que a linha da *cache* possui o mesmo valor que o conteúdo da memória e pode estar presente neste mesmo modo em alguma outra *cache* do sistema.
- ***I**nvalid:* Neste estado, a linha de *cache* não possui dados válidos.

O funcionamento do *Ilinois MESI* segue a política *write-invalidate*, conforme descrita nas seções anteriores.

3.4 O Protocolo Firefly

Este protocolo, representante da categoria dos protocolos baseados em atualização, foi desenvolvido por um laboratório de pesquisas da *DEC (Digital Equipament Corporation)*[1] para uso em sua estação de trabalho multiprocessada *DEC Firefly - MicroVAX-2* (ARCHIBALD; BAER, 1986).

O protocolo é implementado com três estados: *Private, Shared e Dirty*.

- O estado "*private*" é utilizado quando o bloco de *cache* está presente somente na *cache* local e possui o mesmo valor que o da memória.
- O estado "*shared*" representa que o bloco de *cache* está presente em mais de uma *cache* e possui o mesmo valor que o da memória.
- O estado "*dirty*" é usado para representar que o bloco de *cache* está presente somente na *cache* local e seu valor difere do valor armazenado na memória.

[1]Hoje este laboratório faz parte dos laboratórios da HP - Hewlett-Packard.

Para manter a consistência, este protocolo propaga a alteração de um dado para as demais *caches* todas as vezes que uma atualização ocorre, garantindo assim que todas as *caches* que possuem o dado estejam com a cópia mais recente dele.

O funcionamento do protocolo *Firefly* segue a política *write-update*, descrita nas seções anteriores.

O diagrama de estados deste protocolo é baseado na Figura 3.4, que mostra as transições de um protocolo baseado na política *write-update*.

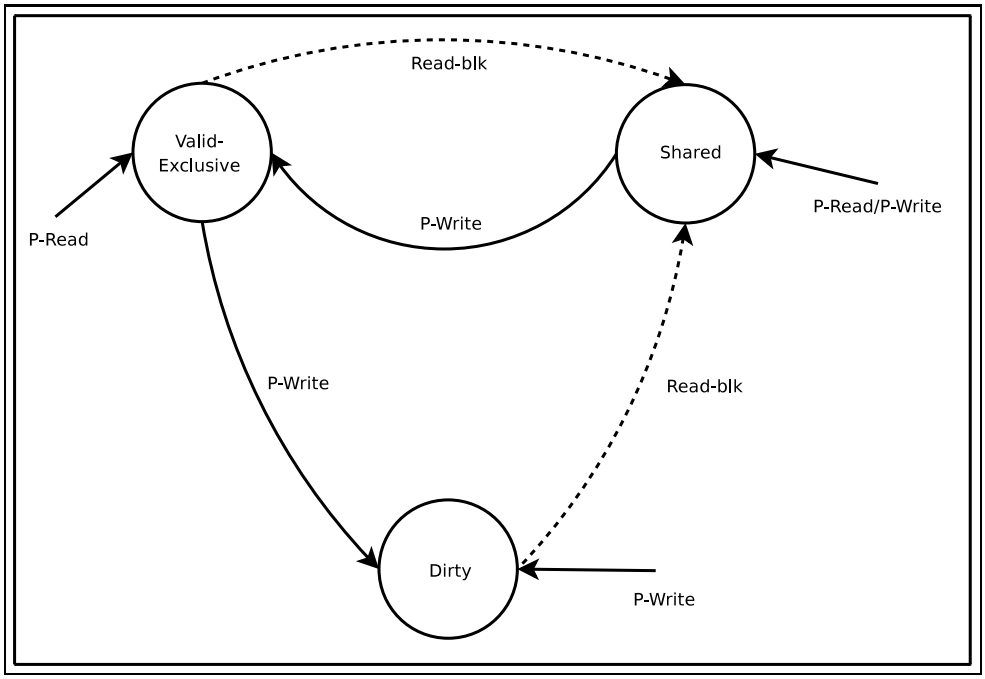

Figura 3.4: *Diagrama de Transições de estados dos protocolos com política write-update.*

.

4 Simuladores

Neste capítulo apresentamos as motivações para o uso de simuladores no estudo de arquitetura de computadores. Descrevemos também alguns simuladores disponíveis nesta área, discutindo suas principais características.

Nem sempre se tem em mãos todos os materiais necessários para pesquisa ou estudo em determinada área. Isso torna-se ainda mais evidente quando estes materiais de estudo possuem custo muito elevado. Ainda existem os casos em que o que está sendo proposto nas pesquisas não existe atualmente no mercado.

Os simuladores são uma boa alternativa para estes e outros problemas. Podem ser empregados para suprir a falta de um equipamento ou até mesmo para testar o funcionamento e desempenho de algo que só existe ainda em projeto. Com isto, os pesquisadores podem executar seus programas e avaliar a corretude do modelo proposto sem que o *hardware* seja fisicamente implementado.

Assim, é comum o uso de simuladores nos projetos de novas arquiteturas, principalmente buscando avaliar as características de desempenho destes projetos, sem que seja construído fisicamente o protótipo do que se projetou. Este tipo de uso facilita o conhecimento e avaliação dos parâmetros arquiteturais, facilitando os ajustes antes que o protótipo real seja construído e avaliado.

Comum também é o uso de simuladores para o estudo de arquiteturas que não possuem fácil acesso, seja pelo custo ou pela escassez de exemplares do *hardware*.

Na área de Arquitetura de Computadores existem diversos simuladores disponíveis, cada qual abrangendo uma determinada sub-área da arquitetura, facilitando os estudos e avaliações de parâmetros de configuração de equipamentos.

4.1 Trabalhos Relacionados

Diversos simuladores têm sido desenvolvidos para prover suporte às pesquisas na área de Arquitetura de Computadores. Variam principalmente em complexidade e eficiência de execução, dependendo do nível de detalhamento que o simulador oferece.

O SimpleScalar (BURGER; AUSTIN, 1997), por exemplo, é um simulador dirigido por execução, cujas principais vantagens são a alta flexibilidade, portabilidade, desempenho e facilidade de expansão. A principal desvantagem deste simulador é que ele modela uma arquitetura monoprocessada.

Um outro simulador baseado no SimpleScalar, que mantém todas as suas características e adiciona a funcionalidade de simular um ambiente multiprocessado com memória distribuída é o MULTIPROC (GONÇALVES, 2000). Este simulador emula o funcionamento de diversos processadores (como numa arquitetura paralela), permitindo que vários aplicativos sejam executados simultaneamente em cada um dos processadores. Uma das dificuldades encontradas neste simulador é que ele não permite que os processos se comuniquem uns com os outros, além de simular somente o modelo de memória distribuída.

Para tratar a comunicação inter-processos no MULTIPROC, foi desenvolvido o simulador SMS (SANDRI; MARTINI; GONÇALVES, 2004), que toma como base o MULTIPROC e implementa rotinas de troca de mensagens entre os processos (programas simulados). Este simulador permite que os processos troquem informações entre si, mas mantém o modelo de memória distribuída.

Diversos outros simuladores foram desenvolvidos e são apresentados sucintamente a seguir.

O Limes (MAGDIC, 1997) implementa um simulador multiprocessado com memória compartilhada, utiliza *threads* para simular o multiprocessamento. Exige que a aplicação seja implementada com certas macros (desenvolvidas pelo *Argonne National Lab*) para definir o nível de paralelismo da mesma. Executa em ambientes Linux. Suporta o modelo de *threads* (processos leves) e implementa alguns protocolos de coerência de *cache* (WTI, Berkeley, Dragon e WIN). O grande diferencial deste simulador é que permite que os protocolos de coerência sejam descritos numa linguagem semelhante à VHDL, facilitando testes de novos protocolos.

O Proteus (BREWER et al., 1992) é um simulador bastante complexo; simula multiprocessadores, fornece bibliotecas de troca de mensagens e gerenciamento de memória e *threads*.

O RSIM (HUGHES et al., 2002) modela em detalhes uma arquitetura superescalar e é bastante utilizado nas simulações envolvendo *cache*. Foi desenvolvido seguindo a descrição do

processador MIPS R10000. Seu objetivo é considerar o modelo de memória e *cache* utilizados pelo processador na avaliação do desempenho da arquitetura.

O MINT (VEENSTRA J.E.; FOWLER, 1994) modela uma arquitetura MIPS (R3000 em específico) e também utiliza *threads* para simular um ambiente multiprocessado.

O ABSS (SUNADA; GLASCO; FLYNN, 1998) é principalmente utilizado para simulações de memória. Seus executáveis têm a restrição de executar em *hardware* Sparc UltraStation ou superiores que suportem o SPARC v9.

O Tango Lite (HERROD, 1993) simula um ambiente multiprocessado de memória compartilhada utilizando *threads*. Ele permite que os códigos sejam escritos em C e Fortran, suportando a plataforma MIPS R3000.

Naraig Manjikian, em alguns trabalhos (MANJIKIAN, 2001b) (MANJIKIAN, 2001a) discute também sobre alterações feitas por ele no código do SimpleScalar para suportar multiprocessamento utilizando *threads*. Ele implementa uma interface gráfica para visualizar os acessos à memória em sistemas monoprocessados e multiprocessados, assim como depurar códigos no simulador multiprocessado.

A utilização destes simuladores aplica-se tanto para a análise de desempenho das arquiteturas que eles representam quanto para a experimentação de parâmetros arquiteturais, como largura de barramento, métodos de acesso à memória e comunicação inter-processos.

4.2 O Simulador SimpleScalar

Como discutido anteriormente, diversos simuladores de arquiteturas encontram-se disponíveis atualmente, entretanto um dos mais difundidos e consolidado mundialmente é o *Toolkit SimpleScalar*. Ele é utilizado amplamente nas maiores universidades e centros de pesquisa do mundo para investigações referentes à arquitetura de computadores.

Desenvolvido pela Universidade de Wisconsin-Madison, o *SimpleScalar* fornece um conjunto de simuladores e ferramentas para simulação que permitem ao usuário fazer diversos níveis de análise de desempenho em processadores superescalares. O simulador é disponibilizado gratuitamente pela Universidade para fins não comerciais.

O *Toolkit SimpleScalar* fornece diversos simuladores, com graus diferentes de especificidade, podendo ser utilizado para simulações envolvendo caches, previsores de desvio, acessos à memória, emissão de instruções fora de ordem, caches não-bloqueantes e execução expeculativa.

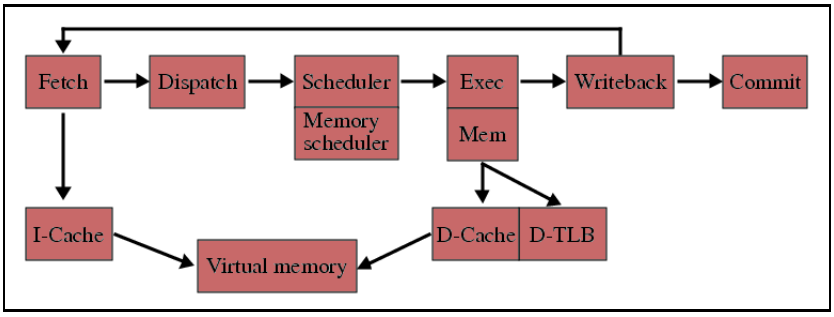

Figura 4.1: *Estrutura modular do sim-outorder - SimpleScalar*

O *SimpleScalar* fornece um compilador *gcc* específico que gera código *assembly SimpleScalar* para aplicativos desenvolvidos em C, o que permite que *benchmarks* escritos em linguagem C possam ser compilados e testados no simulador.

A arquitetura do *SimpleScalar* é derivada do MIPS-IV ISA com algumas pequenas diferenças:

- operações de *load* e *store* suportam dois modos de endereçamento: registrador+registrador e auto-incremento/decremento;
- uma operação de raiz quadrada;
- um formato de instrução estendida de 64 bits.

O *SimpleScalar* usa um espaço de endereçamento de memória virtual com 31 *bits* de endereço. A arquitetura é toda definida num arquivo chamado *ss.def*, em que um conjunto de *macros* define cada uma das instruções.

O simulador mais detalhado disponível no *Toolkit* é o *sim-outorder*, que implementa emissão e execução de instruções fora de ordem baseado na técnica da *"Register Update Unit"* (SOHI, 1990), que faz uso dos *reorder buffers*.

Este simulador, assim como os demais do *Toolkit* é uma ferramenta modular, o que facilita a sua expansão, tornando-o um simulador muito utilizado como base para o desenvolvimento de novas ferramentas de simulação.

A estrutura modular do *SimpleScalar sim-outorder* pode ser vista na Figura 4.1.

4.3 Os Simuladores MULTIPROC e SMS

O simulador MULTIPROC foi desenvolvido tendo como base o *simulador sim-outorder*, do *Toolkit SimpleScalar*. Ele é baseado na replicação das estruturas do simulador *sim-outorder*,

comportando-se como um conjunto de simuladores *sim-outorder* executando paralelamente.

O SMS por sua vez, foi baseado no simulador MULTIPROC, que já implementava o multiprocessamento, entretanto não permitia que os processadores simulados trocassem informações entre si.

Assim, o SMS implementou rotinas que permitem aos processadores trocarem mensagens entre si, fornecendo um conjunto de bibliotecas que permite o envio e recepção destas mensagens.

Em ambos os simuladores, os espaços de endereçamento de memória e cache são individuais por processador, garantindo que cada processador simulado comporte-se como um processador exclusivo dentro da arquitetura simulada.

As estatísticas geradas por estes simuladores também foram modificadas para devolver os resultados sintetizados por unidade de processamento, fazendo com que cada um dos parâmetros estatísticos do *SimpleScalar* pudesse ser avaliado para cada CPU destes simuladores multiprocessados.

.

5 O simulador SMS-MC

Neste capítulo apresentamos os objetivos do desenvolvimento do simulador SMS-MC, assim como suas características e diferenciais. Apresentamos a metodologia utilizada no seu desenvolvimento e explicamos o funcionamento da política aplicada aos protocolos implementados. Descrevemos ainda os principais detalhes da implementação apresentando alguns trechos da estrutura do simulador.

O objetivo principal deste trabalho consistiu em viabilizar uma ferramenta de simulação para Arquiteturas Paralelas com Memória Compartilhada, baseada na estrutura do SimpleScalar e seus derivados (como o MULTIPROC e o SMS). Este simulador representa a classe das máquinas MIMD, segundo a Taxonomia de Flynn, e implementa alguns protocolos de coerência de *cache* para ilustrar a tecnologia envolvida.

O desenvolvimento de uma ferramenta com tais requisitos não poderia ser fundamentada sobre uma estrutura que não estivesse consolidada. Assim, baseado no SimpleScalar, um simulador difundido mundialmente e já consolidado na área de Arquitetura de Computadores, o SMS-MC tende a ter um maior potencial de utilização no ensino e pesquisa.

A ferramenta serve como apoio para avaliação de parâmetros referentes à coerência de *cache* e uso de memória compartilhada, gerando estatísticas úteis na avaliação das arquiteturas simuladas.

Além do propósito final que é permitir avaliação dos modelos de memória compartilhada e seus protocolos, a ferramenta também pode ser utilizada no ensino de desenvolvimento de aplicações paralelas, uma vez que permite a execução (simulação) de equipamentos com múltiplos processadores e diversos protocolos de coerência de *cache*.

Como é baseado numa arquitetura modular, o simulador SMS-MC permite ainda que novos protocolos de coerência de *cache* sejam implementados e inseridos no módulo de *cache*, facilitando a avaliação prática e fácil para possíveis novos protocolos.

5.1 Características

O simulador SMS-MC possibilita a execução de diversos processos simultaneamente, cada qual sendo executado por um processador. Cada processador simulado possui uma memória local que é empregada para carregar o programa a ser executado e para a manipulação de variáveis não compartilhadas.

Um espaço de memória específico dentro do simulador disponibiliza a todos os processadores uma área compartilhada. Assim, cada programa (escrito em linguagem C e gerado com um compilador *cross-compiler*) pode declarar variáveis como compartilhadas, tornando-as visíveis para os demais processadores e programas em execução no simulador. É importante ressaltar que na implementação do SMS-MC disponibilizou-se somente *caches* de dados compartilhados, não permitindo a existência de *caches* de instruções compartilhadas, pois isto tomaria um tempo para a definição de primitivas para compartilhamento de código e alterações no código fonte do compilador utilizado para gerar os códigos executáveis do SMS-MC, o que não foi previsto no cronograma e nos objetivos deste trabalho.

Com a estrutura montada pelo simulador, nada impede que as *caches* de instruções possam suportar instruções compartilhadas; o que ainda não é possível no simulador é como definir estes trechos de código compartilhados dentro dos executáveis.

O simulador suporta parâmetros para ajustar o *delay* do acesso à memória compartilhada e o modelo de coerência de *cache* a ser simulado: *none* (nenhum), *snoopy* ou *directory* (baseado em diretórios). O modelo de coerência *none* é utilizado para as *caches* do simulador que não possuem dados compartilhados, como no caso de *caches* de instruções.

Ambos os protocolos de coerência de *cache* implementados no simulador utilizaram a política *write-invalidate* (*Snoopy* e *Directory*) e seguiram o modelo de quatro estados **MESI**. A principal diferença nas duas implementações deve-se ao fato da implementação do protocolo *Snoopy* realizar comunicação exclusivamente através de mensagens no barramento em *broadcast* enquanto o protocolo *Directory* realiza comunicação através de mensagens aos processadores que realmente contêm os blocos de *caches* (semelhante a uma comunicação *multicast*).

Para permitir análises de desempenho de arquiteturas com diferentes protocolos de coerência de *cache* e memória compartilhada, o simulador SMS-MC gera um conjunto de estatísticas sobre todas as operações realizadas nas memórias *caches*, fornecendo valores para cada uma das operações realizadas nas transições de estados dos blocos das *caches* compartilhadas. Outras estatísticas, antes fornecidas pelo simulador SimpleScalar continuam sendo fornecidas pela ferramenta SMS-MC.

5.2 Diferenciais

Baseado na arquitetura do SimpleScalar, o simulador proposto dispõe de um alto grau de detalhamento da arquitetura modelada. O multiprocessamento fornecido pelo simulador oferece uma alternativa para o uso de *threads*, visto que pode-se executar programas totalmente independentes no simulador e o mesmo irá simular vários processadores trabalhando de forma independente.

Como o simulador é baseado no SimpleScalar, praticamente todos os *benchmarks* desenvolvidos para o SimpleScalar podem facilmente ser adaptados para o simulador proposto, adicionando-se é claro as novas funcionalidades de memória compartilhada presentes no SMS-MC.

Assim, todo o nível de detalhamento que já pode ser obtido com o SimpleScalar, também está disponível neste novo simulador, como por exemplo a possibilidade de alterar as especificações das *caches* de dados e instruções, os parâmetros e modelos dos previsores de desvios, a largura dos barramentos de busca, entre outros.

Este simulador executa códigos compilados para a arquitetura PISA (*Portable Instruction Set Architecture*), um sub-conjunto das instruções da arquitetura MIPS. Logo, com um compilador *gcc* alterado para tal, pode-se compilar códigos C para serem testados diretamente no simulador, não ficando atrelado aos *benchmarks* já existentes.

Mantendo a modularidade do SimpleScalar, o SMS-MC re-implementa os módulos de memória e de *cache*, criando uma camada superior que oculta do processador qual o tipo de *cache* está sendo utilizada, fornecendo interfaces para que os dados compartilhados sejam manipulados. Facilmente novos modelos de *cache* e protocolos de coerência podem ser implementados e incorporados a esta camada de alto nível do módulo de *cache* do simulador, permitindo expandir ainda mais a utilização e abrangência do simulador.

5.3 Metodologia

A construção do simulador foi baseada no simulador SMS, que é um simulador de multiprocessadores de memória distribuída baseado no código do SimpleScalar.

O SimpleScalar é capaz de simular um processador MIPS, incluindo previsores de desvios, memória, *caches* e diversos outros parâmetros. O SMS é a versão multiprocessada do SimpleScalar, permitindo que seja avaliada a execução de diversos processadores ao mesmo tempo, como numa máquina paralela.

Tomando como base o simulador SMS, que já simula o uso de múltiplos processadores, cada qual com sua memória e *cache* individuais, implementamos o simulador SMS-MC (Simulador de Multiprocessadores Superescalares de Memória Compartilhada), que fornece a funcionalidade de usar regiões de memória compartilhada, proporcionando um espaço de endereçamento que é comum a todos os processadores.

Dessa forma, criamos uma estrutura de memória específica para armazenar os dados que são compartilhados. Quando algum processador tenta acessar um dado (ou endereço) que é compartilhado, sua requisição é enviada para esta memória. Um desvio na rotina de inicialização do simulador foi criado para indicar ao simulador que carregue e organize os dados compartilhados nesta memória específica. Isto acontece através de uma interrupção.

As rotinas que efetuam `leitura` e `escrita` de dados na memória foram alteradas de forma a permitir que acessos à porção de memória compartilhada sejam efetuados de forma transparente para o processador e para o programador. Assim, quaisquer operações sobre os dados compartilhados são realizadas normalmente, como se fossem sobre variáveis exclusivas.

Uma biblioteca com funções para declarar variáveis compartilhadas e iniciar o modo "*memória compartilhada*" foi disponibilizada para que o programador possa desenvolver seus aplicativos fazendo uso do espaço de endereçamento compartilhado. Esta biblioteca deve ser incluída em todos os aplicativos (ou *benchmarks*) que são executados no simulador.

O simulador SMS-MC suporta dois protocolos de coerência de *cache*: um baseado na técnica *Snoopy* e outro baseado na técnica Directory. Ambos são implementações de protocolos de invalidação utilizando quatro estados[1].

Baseado nas descrições do protocolo MESI, discutidas anteriormente, implementamos estes dois protocolos seguindo o roteiro de execução que é discutido em detalhes na Seção 5.4.

5.3.1 Implementação da Memória Compartilhada

Durante os testes na elaboração do projeto do SMS-MC notou-se que todas as variáveis declaradas no início do programa principal de qualquer *benchmark* desenvolvido para o simulador possuíam o mesmo endereço físico de memória. Assim, se as variáveis fossem declaradas com os mesmos nomes e na mesma ordem em todos os *benchmarks* simulados paralelamente, elas ocupariam exatamente as mesmas posições de memória.

Assim, desviando os acessos de leitura e escrita destes endereços de memória para uma

[1]Os protocolos de coerência de *cache* que implementam quatro estados geralmente são conhecidos como protocolos MESI.

região de memória específica dentro do simulador, todos os processadores acessariam os mesmos dados, fornecendo a funcionalidade de memória compartilhada para estas variáveis.

Desta forma, definimos um *layout* para ser utilizado nas implementações de *benchmarks* do SMS-MC, que é caracterizado por um arquivo que contém todas as definições de variáveis compartilhadas (chamado *shared.def*), sendo incluído em todos os *benchmarks* exatamente na mesma posição (no início do programa principal), garantindo que todos os *benchmarks* possuam as mesmas variáveis definidas no arquivo *shared.def* exatamente nas mesmas posições.

A partir deste princípio, algumas macros foram definidas para serem utilizadas com objetivo de instruir ao simulador SMS-MC quais as posições físicas destas variáveis definidas no *shared.def* e quais os seus tamanhos. Um padrão para utilização destas macros foi definido, garantindo que elas sejam as primeiras instruções a serem executadas por um dos *benchmarks* em execução no simulador, inicializando as variáveis compartilhadas no SMS-MC.

Uma alteração na máquina de execução do simulador SMS-MC foi realizada para garantir que o simulador carregue todos os endereços de variáveis compartilhadas definidas no shared.def e informados pelas macros. Assim, enquanto todas as variáveis compartilhadas não tiverem sido carregadas nas devidas estruturas, o simulador executa operações NOP para os *benchmarks* aguardando que as operações de inicialização sejam executadas.

Dessa forma, realizando um desvio nas rotinas de *load* e *store*, podemos direcionar todas as operações nas posições de memória compartilhada para a nova região de memória que foi definida internamente no simulador justamente para conter os dados destas variáveis compartilhadas.

5.4 Funcionamento do MESI

No início de uma simulação, todos os blocos presentes na *cache* estão no estado "*Invalid*" .

Quando um processador deseja realizar uma operação sobre um dado compartilhado, primeiramente verifica em sua *cache* se já possui este dado. Caso possua o dado no estado "*Exclusive*" ou "*Modified*", ele pode realizar normalmente a operação, considerando somente o tempo de acesso à *cache*. Neste caso, o estado do bloco deve ser alterado para "*Modified*" caso a operação seja de `escrita`.

Se o bloco na *cache* estiver com o estado "*Shared*" e a operação seja de `leitura`, a operação pode ocorrer normalmente, novamente com a latência do acesso à *cache*. Entretanto, caso o bloco esteja no estado "*Shared*" e a operação seja de `escrita`, o processador deve avisar todos

os demais processadores [2] que irá atualizar o dado, invalidando as demais cópias do dado em outras *caches*. Vale lembrar que caso alguma das outras *caches* possua o mesmo dado com o estado "*Modified*", ela deve realizar um *write-back* para armazenar o valor do dado na memória.

Na implementação de um protocolo MESI existem 4 situações básicas que devem ser tratadas:

- *Read hit;*
- *Read miss;*
- *Write hit;*
- *Write miss.*

Para tratar cada uma destas situações, algumas ações nas *caches* são necessárias e são detalhadas a seguir.

5.4.1 Read hit

Esta situação ocorre numa *cache* local quando seu processador tenta executar uma operação de `leitura` num bloco que já está em *cache*.

O estado do bloco precisa ser "*Exclusive*" , "*Shared*" ou "*Modified*" para que a operação seja executada com sucesso. Neste caso, o controlador de *cache* não precisa enviar nenhuma mensagem para as outras *caches*.

A única latência no processo será a do acesso ao dado na *cache*.

5.4.2 Read miss

Esta situação ocorre numa *cache* local quando seu processador tenta executar uma operação de `leitura` num bloco que ainda não está na *cache*.

Neste caso, o controlador da *cache* precisa enviar uma mensagem às outras *caches* [3] para saber se alguma delas possui o bloco desejado e qual seu possível estado (*Modified*, *Exclusive*, *Shared* ou *Invalid*).

[2] Neste caso, vale observar que se o protocolo estiver implementando um *Snoopy Protocol* ele avisará todas as demais caches remotas, numa operação de *broadcast*. Sendo o protocolo implementado baseado nos *Directory Protocols* somente serão notificadas as caches que possuem uma cópia do dado solicitado.

[3] Esta mensagem só é necessária caso a implementação seja baseada nos *Snoopy Protocols*, pois neste caso não possui informação alguma sobre quais caches podem conter o bloco que está sendo manipulado.

Neste estado, três possíveis caminhos podem ser seguidos:

- Numa primeira situação, nenhuma das outras *caches* possui o bloco desejado. Assim, o controlador de *cache* lê o dado da memória e o armazena na *cache* com estado marcado como "*Exclusive*". A latência neste caso seria a de ler o dado da memória, mais os tempos de enviar uma mensagem para todas as demais *caches* e receber as devidas respostas delas.

- Numa segunda situação, pelo menos uma das *caches* possui o bloco no estado "*Exclusive*" ou "*Shared*". Neste caso, o controlador envia uma nova mensagem informando que deseja "compartilhar" o bloco. Em seguida lê o dado da memória e notifica as demais *caches* que possuem uma cópia do dado para que alterem o estado do bloco para "*Shared*". Nesta situação, a latência seria a de ler o dado da memória, enviar uma mensagem a todas as *caches* solicitando o estado do bloco e, enviar uma nova mensagem solicitando o compartilhamento do bloco.

- Numa terceira situação, uma das *caches* remotas retorna a informação de que possui o bloco com o estado "*Modified*". Neste caso, uma nova mensagem solicitando o compartilhamento do bloco é enviada, obrigando a *cache* que possui o dado alterado a realizar um *write-back* do dado e alterar o estado do mesmo para "*Shared*". A *cache* local pode então ler o bloco da memória e marcar o estado do mesmo como "*Shared*". Neste caso, a latência total seria a soma das latências de leitura do bloco da memória, mais o envio das mensagens solicitando o estado do bloco nas demais *caches*, mais o envio das mensagens de compartilhamento do bloco, mais o tempo de *write-back* do bloco modificado para a memória.

5.4.3 Write hit

Esta situação ocorre numa *cache* local quando o seu processador tenta executar uma operação de `escrita` num bloco que já está na *cache*.

O estado do bloco precisa ser "*Exclusive*" ou "*Modified*" para que a operação possa ser concluída com sucesso. Neste caso, o controlador da *cache* não precisa enviar mensagens a nenhuma das outras *caches*. A latência para esta situação é a de executar uma operação de `escrita` num bloco local na *cache*.

5.4.4 Write miss

Esta situação ocorre numa *cache* local, por dois motivos, quando o processador tenta executar uma operação de `escrita` num bloco de *cache*:

- o bloco não está presente na *cache* local;
- o bloco está presente na *cache* local, mas com o estado "*Shared*".

Em ambas as situações, o controlador da *cache* envia mensagens para todas as outras *caches*, solicitando que invalidem suas cópias do bloco que deseja atualizar. Caso alguma das outras *caches* possua o dado com o estado "*Modified*", deve fazer uma operação *write-back* e marcar seu bloco localmente com o estado "*Invalid*".

Caso as outras *caches* possuam o dado, mas em outro estado diferente de "*Modified*", simplesmente marcam este bloco com o estado "*Invalid*". Assim, se o bloco estiver na *cache* local, mesmo com o estado "*Shared*", a operação de `escrita` é realizada, alterando o estado do bloco para "*Modified*". Se o bloco não estiver na *cache*, ele é então carregado da memória para a *cache* e a operação de `escrita` é realizada.

Neste caso, a latência do procedimento seria a de invalidar as demais *caches* e realizar a operação de `escrita` no bloco local da *cache*.

5.5 Implementação

O simulador SimpleScalar, base da implementação do SMS-MC, é um simulador desenvolvido em linguagem C e totalmente modularizado. O início da fase de implementação deste trabalho de mestrado conduziu um estudo aprofundado sobre a estrutura do SimpleScalar.

Sendo modularizado, os arquivos (ou módulos) responsáveis por memória e *cache* foram documentados e depurados para que a tarefa de customização desta parte do simulador pudesse ser executada de maneira mais efetiva.

Assim, alterações foram feitas basicamente em três arquivos: *memory.c/memory.h*, *cache.c/cache.h* e *sms-mc.c/sms-mc.h*.

Algumas estruturas novas foram definidas no arquivo *memory.h* para suportar o modelo de memória compartilhada, incluindo modificações em estruturas já existentes, como as *páginas de memória* compartilhadas, que podem ser notadas no fragmento `Código 1`, onde pode ser

verificada a adição da variável ptab_shared_access representando a quantidade de acessos a páginas de dados compartilhados.

Código 1 Estruturas da memória definidas em memory.h

```
struct mem_t {
    char *name;
    struct mem_pte_t *ptab[MEM_PTAB_SIZE];
    counter_t page_count;
    counter_t ptab_misses;
    counter_t nb_shared;
    counter_t ptab_accesses;
    counter_t ptab_shared_access;
};
```

Outras estruturas foram definidas para implementação da região compartilhada de memória, que foi implementada através de uma lista dinâmica levando em consideração que endereços aleatórios podem ser compartilhados no simulador. Estas alterações podem ser visualizadas no fragmento Código 2, onde nota-se a estrutura dos nós da lista ligada utilizada para implementar a região de memória compartilhada.

Código 2 Estruturas da memória compartilhada definido em memory.h

```
struct nodo_shared{
    md_addr_t address;
    struct nodo_shared *next;
};
struct mem_t *shared;
```

No fragmento Código 3 temos três funções de apoio que foram definidas para auxílio na rotina de leitura e escrita da memória, com objetivo de facilitar a identificação de endereços compartilhados. Caso um endereço solicitado para as rotinas de leitura / escrita seja reconhecido como compartilhado, ocorre um desvio na rotina para a região de memória compartilhada.

Código 3 Rotinas de apoio definidas em memory.h e implementadas em memory.c

```
/* set bit address shared in table of address */
void set_bit_shared( md_addr_t addr);

/* get shared status of address */
int get_bit_shared ( md_addr_t addr);

/* Funcao utilizada para marcar o endereco como compartilhado */
void set_shared( md_addr_t addr,
                 unsigned int size);
```

Uma alteração foi realizada na função de inicialização do sistema de memória, sendo ela responsável por zerar as estatísticas de acesso à memória compartilhada, conforme pode ser visto no fragmento Código 4.

Código 4 Rotinas de inicialização da memória, definidas em memory.h e implementadas em memory.c

```
void mem_init(   struct mem_t *mem){
    int i;
    for (i=0; i < MEM_PTAB_SIZE; i++)
        mem->ptab[i] = NULL;
    mem->page_count = 0;
    mem->ptab_misses = 0;
    mem->ptab_accesses = 0;
    mem->ptab_shared_access = 0;
    mem->nb_shared = 0;
}
```

As principais alterações para disponibilizar o acesso à região compartilhada de memória do simulador SMS-MC foram realizadas na função mem_access(), encontrada no arquivo *memory.c*. Toda vez que o acesso é solicitado para um determinado endereço (seja ele numa operação de leitura ou escrita), a rotina de leitura verifica através de uma rotina auxiliar (get_bit_shared()) se o endereço corresponde a um endereço de memória compartilhada. No caso negativo, o acesso ocorre normalmente, como qualquer outro acesso à memória disponibilizado no simulador. Entretanto, sendo um endereço de memória compartilhada, a rotina faz um desvio para o acesso à estrutura com a memória compartilhada. O fragmento Código 5 mostra a parte principal da implementação da função mem_access().

Ainda no módulo de memória alterações foram executadas também no código da função que gera as estatísticas de acesso, chamada de mem_reg_stats(), incluindo algumas estatísticas referentes aos dados compartilhados, como pode ser visto no fragmento Código 6.

Para implementarmos os protocolos de coerência de cache, fez-se necessário alterações nos módulos cache.c e cache.h. Algumas estruturas foram definidas para representar os protocolos de coerência de *cache* e os estados de um determinado bloco da *cache*, conforme pode ser observado nos fragmentos Código 7 e Código 8.

As rotinas de geração de estatísticas da *cache* também foram alteradas de maneira a prover dados sobre os acessos, estados e o protocolo de coerência simulado. Alguns detalhes da alteração podem ser visualizados no fragmento Código 9.

A rotina utilizada para instanciar as *caches* foi readequada de maneira a suportar a seleção do protocolo a ser utilizado pela cache e zerar os contadores referentes aos acessos de memória

Código 5 Rotina de acesso à memória definida em `memory.h` e implementada em `memory.c`.

```
enum md_fault_type
mem_access( struct mem_t *mem, /* memory space to access */
            enum mem_cmd cmd,  /* Read (from sim mem) or Write */
            md_addr_t addr,    /* target address to access */
            void *vp,          /* host memory address to access */
            int nbytes,        /* number of bytes to access */
            int sn)            /* slot number */
{
    byte_t *p = vp;
    struct mem_pte_t *temp=NULL;
    int _shared=0;
    if(loaded_shared_vars) {
        if( get_bit_shared(addr) ) {
            _shared = 1;
            mem->ptab_shared_access++;
        } else {
            _shared = 0;
        }
    }
    switch (nbytes) {
        case 1: if (cmd == Read) {
                    if( _shared ) *((byte_t *)p) = MEM_READ_BYTE(shared, addr)
                    else *((byte_t *)p) = MEM_READ_BYTE(mem, addr);
                } else {
                    if( _shared ) MEM_WRITE_BYTE(shared, addr, *((byte_t *)p))
                    else MEM_WRITE_BYTE(mem, addr, *((byte_t *)p));
                }
                break;
    }
    return md_fault_none;
}
```

compartilhada, conforme o fragmento `Código 10`.

Para prover acesso à *cache* de dados compartilhados, implementando os dois protocolos de coerência de *cache* disponíveis no SMS-MC, alterações foram realizadas na função `mem_access()`, direcionando o acesso para o módulo correspondente no simulador: *Directory* ou *Snoopy*. Estas alterações podem ser verificadas no fragmento `Código 11`.

5.6 Utilização do Simulador

Para utilização do simulador SMS-MC devemos seguir alguns procedimentos ao implementar os *benchmarks*:

Código 6 Rotina de geração de estatísticas definida em `memory.h` e implementada em `memory.c`.

```
void mem_reg_stats( struct mem_t **mem,
                    struct stat_sdb_t *sdb)
{
    char buf[512], buf1[512];
    int sn;
    for (sn=0;sn<USED_SLOTS;sn++) {
        sprintf(buf, "shared_bytes.cpu_%s", itoa(sn));
        stat_reg_counter(sdb, buf, "total shared bytes",
            &shared->nb_shared, shared->nb_shared, NULL,sn);
    }
    for (sn=0;sn<USED_SLOTS;sn++) {
        sprintf(buf, "ptab_shared_access.cpu_%s", itoa(sn));
        stat_reg_counter(sdb, buf, "total shared page table accesses",
            &mem[sn]->ptab_shared_access, mem[sn]->ptab_shared_access,
            NULL,sn);
    }
    for (sn=0;sn<USED_SLOTS;sn++) {
        sprintf(buf, "%s.ptab_miss_rate_%s", mem[sn]->name, itoa(sn));
        sprintf(buf1, "%s.ptab_misses_%s / %s.ptab_accesses_%s",
            mem[sn]->name, itoa(sn), mem[sn]->name, itoa(sn));
        stat_reg_formula(sdb, buf, "first level page table miss rate",
            buf1, NULL,sn);
    }
}
```

- as variáveis compartilhadas devem ser todas declaradas num arquivo específico, sendo que estarão disponíveis nos aplicativos simulados como variáveis globais. O nome deste arquivo onde define-se as variáveis compartilhadas deve ser *"shared.def"*;

- o arquivo que contém a definição das variáveis compartilhadas (*shared.def*) deve ser incluído (através da diretiva *#include* em linguagem C) em todos os aplicativos que estiverem sendo simulados no SMS-MC na primeira linha após a definição da função *main()* dos aplicativos;

- o arquivo de cabeçalho *shared.h* que contém algumas das definições da implementação de memória compartilhada e funções para controle das variáveis compartilhadas deve estar presente nos aplicativos simulados;

- um dos aplicativos que estará sendo executado no simulador deve ser eleito para iniciar o ambiente de memória compartilhada, devendo executar uma chamada à função *run_shared()*;

- o mesmo aplicativo que executar a função *run_shared()* deve também fazer a inicialização

Código 7 Estruturas do módulo de cache, definidas em `cache.h`.

```
/* cache coherence protocol */
enum cache_coherence_protocol {
    NONE,
    SNOOPY,
    DIRECTORY,
    WUPDATE
};

/* block status values */
#define CACHE_BLK_VALID                0x00000001
#define CACHE_BLK_DIRTY                0x00000002

/* block shared status */
#define SHARED_BLK_MODIFIED     0x00000001
#define SHARED_BLK_EXCLUSIVE    0x00000002
#define SHARED_BLK_SHARED       0x00000004
#define SHARED_BLK_INVALID      0x00000008

/* cache block (or line) definition */
struct cache_blk_t
{
    struct cache_blk_t *way_next;
    struct cache_blk_t *way_prev;
    struct cache_blk_t *hash_next;
    md_addr_t tag;
    unsigned int status;
    unsigned int shared_status;
    tick_t ready;
    byte_t *user_data;
    byte_t data[1];
};
```

das variáveis no ambiente de memória compartilhada, executando uma chamada à função *SHARED(nome_da_variavel)* para todas as variáveis compartilhadas do sistema.

Após seguir todo este procedimento para utilização da memória compartilhada, todas as variáveis definidas no arquivo *shared.def* já estarão disponíveis para utilização em todos os aplicativos simulados, permitindo que seu acesso (dentro da seqüência do programa) ocorra como se fosse uma variável normal, divergindo apenas na forma com que o simulador manipula as questões de coerência e contabiliza as estatísticas internamente.

Código 8 Estrutura da cache, definida em `cache.h`.

```
struct cache_t
{
    char *name;
    int type;
    struct cache_t **parent;
    int nsets;
    int bsize;
    int balloc;
    int usize;
    int assoc;
    enum cache_policy policy;
    enum cache_coherence_protocol protocol;
    unsigned int hit_latency;
    unsigned int
    (*blk_access_fn)(   enum mem_cmd cmd,
                                md_addr_t baddr,
                                int bsize,
                                struct cache_blk_t *blk,
                                tick_t now,
                        int sn);
    int hsize;
    md_addr_t blk_mask;
    int set_shift;
    md_addr_t set_mask;
    int tag_shift;
    md_addr_t tag_mask;
    md_addr_t tagset_mask;
    tick_t bus_free;
    counter_t hits;
    counter_t misses;
    counter_t replacements;
    counter_t writebacks;
    counter_t invalidations;
    counter_t mesi_hits;
    counter_t mesi_misses;
    counter_t mesi_replacements;
    counter_t mesi_writebacks;
    counter_t mesi_invalidations;
    counter_t mesi_share;
    counter_t mesi_read_bus;
    counter_t mesi_write_bus;
    md_addr_t last_tagset;
    struct cache_blk_t *last_blk;
    byte_t *data;
    struct cache_set_t sets[1];
};
```

Código 9 Rotina de geração de estatísticas definida em `cache.h` e implementada em `cache.c`.

```
void cache_reg_stats( struct cache_t **cp, struct stat_sdb_t *sdb) {
    char stat_name[128], stat_formula[128], cache_name[MAX_SLOTS][64];
    int sn;
    for (sn=0;sn<USED_SLOTS;sn++) {
        sprintf(stat_name, "%s.mesi_accesses_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_hits_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_misses_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_replacements_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_writebacks_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_invalidations_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_share_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_bus_read_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_bus_write_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_miss_rate_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_repl_rate_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_wb_rate_%s", cache_name[sn], itoa(sn));
        sprintf(stat_name, "%s.mesi_inv_rate_%s", cache_name[sn], itoa(sn));
    }
}
```

Código 10 Rotina de criação da cache definida em `cache.h` e implementada em `cache.c`.

```
struct cache_t *cache_create(char *name, int type,
             int nsets, int bsize, int balloc, int usize,
             int assoc, enum cache_policy policy,
             enum cache_coherence_protocol protocol,
             unsigned int (*blk_access_fn)(enum mem_cmd cmd,
                      md_addr_t baddr, int bsize,
                      struct cache_blk_t *blk, tick_t now, int sn),
             unsigned int hit_latency, int sn)
{
    struct cache_t *cp;
    cp = (struct cache_t *) calloc(1, sizeof(struct cache_t) );
    if (!cp)
        fatal(sn,"out of virtual memory");
    cp->name = mystrdup(name);
    cp->type = type;   cp->nsets = nsets; cp->bsize = bsize;
    cp->balloc = balloc; cp->usize = usize; cp->assoc = assoc;
    cp->policy = policy; cp->protocol = protocol; cp->hit_latency = hit_latency;
    cp->blk_access_fn = blk_access_fn;
    cp->hits = 0; cp->misses = 0; cp->replacements = 0; cp->writebacks = 0;
    cp->invalidations = 0;
    cp->mesi_hits = 0; cp->mesi_misses = 0; cp->mesi_replacements = 0;
    cp->mesi_writebacks = 0;  cp->mesi_invalidations = 0; cp->mesi_share = 0;
    cp->mesi_read_bus = 0; cp->mesi_write_bus = 0;
    return cp;
}
```

Código 11 Rotina de acesso à cache definida em `cache.h` e implementada em `cache.c`.

```
unsigned int
cache_access(struct cache_t *cp,
             enum mem_cmd cmd,
             md_addr_t addr,
             void *vp,
             int nbytes,
             tick_t now,
             byte_t **udata,
             md_addr_t *repl_addr,
             int sn)
{
    int retorno = 0;
    switch(cp->protocol){
            case NONE: retorno = cache_none_access(
                            cp, cmd, addr, vp, nbytes,
                            now, udata, repl_addr, sn);
            break;
            case SNOOPY:    retorno = cache_snoopy_access(
                            cp, cmd, addr, vp, nbytes,
                            now, udata, repl_addr, sn);
            break;
            case DIRECTORY: retorno = cache_directory_access(
                            cp, cmd, addr, vp, nbytes,
                            now, udata, repl_addr, sn);
            break;
            case WUPDATE:
            default:        retorno = cache_none_access(
                            cp, cmd, addr, vp, nbytes,
                            now, udata, repl_addr, sn);
    }
    return retorno;
}
```

6 *Experimentos e Resultados*

Neste capítulo apresentamos os principais dados fornecidos pelo simulador SMS-MC ao seu usuário, exemplificando seu uso com quatro benchmarks que foram desenvolvidos exclusivamente para este fim. Apresentamos ainda tabelas e gráficos que permitem analisar o desempenho apresentado pelos protocolos de coerência da cache implementados no simulador.

O principal objetivo deste trabalho foi implementar um Simulador de Multiprocessadores Superescalares com Memória Compartilhada, o SMS-MC. A utilização desta ferramenta está direcionada para ensino e pesquisa de sistemas que envolvam memória compartilhada, assim como análise de protocolos de coerência de *cache*.

O uso do simulador como ferramenta de ensino pode ser direcionado à simulação de aplicações paralelas para a arquitetura proposta, podendo também ser considerada a possibilidade de desenvolvimento de novos protocolos de coerência de *cache*.

Como a ferramenta gera um conjunto de estatísticas sobre os acessos à memória e à *cache*, pode-se avaliar o desempenho de certas aplicações em função dos protocolos de coerência de *cache*, permitindo verificar qual protocolo tem melhor desempenho para cada tipo de aplicação.

Na forma como foi implementado, o simulador SMS-MC pode servir ainda como uma plataforma para implementação e testes de outros protocolos de coerência de *cache*, pois já possui toda a interface com o sub-sistema de memória que permite a novos modelos de protocolos serem acoplados, simplesmente respeitando as funções de acesso que já estão definidas nesta interface.

Para ilustrar a utilização dos dois protocolos de coerência de *cache* que foram implementados no simulador SMS-MC, desenvolveu-se também neste trabalho alguns exemplos de aplicações, que podem ser utilizadas de forma didática para o ensino de programação paralela ou para estudos sobre os protocolos de coerência de *cache*.

O simulador SMS-MC fornece ao usuário uma série de estatísticas, como por exemplo: tamanho e estrutura das *caches* individuais por processador, latência dos acessos a cada *cache*, quantidade de níveis de *cache*, se há *caches* de instruções e dados, se são unificadas ou separadas, latência de acessos à memória, número de instruções, número de *loads* e *stores*, modelo e configuração dos previsores de desvios, estatísticas de acessos às *caches* individuais de memória normal e memória compartilhada.

Nas seções subseqüentes apresentamos os *benchmarks* desenvolvidos para demonstrar o uso e funcionamento do simulador, bem como discutimos os resultados fornecidos pelas simulações executadas.

6.1 Aplicativos simulados

A fim de ilustrar a utilização do simulador SMS-MC, quatro aplicativos foram desenvolvidos com fins puramente didáticos[1]: *randomize*, *vetor*, *campo* e *whetstone*.

O aplicativo *randomize* foi desenvolvido com o objetivo de ilustrar como o simulador SMS-MC se comporta e quais estatísticas que ele tem condições de fornecer ao pesquisador, desenvolvedor ou estudante. Este aplicativo realiza leituras e escritas dinâmicas na porção de memória compartilhada, simulando alguns cálculos de forma a gerar operações nos protocolos de coerência de *cache*.

O aplicativo *vetor* foi desenvolvido com objetivo de ilustrar a execução da aplicação que executa o produto de dois vetores de forma paralela. Esta aplicação não foi desenvolvida focando obter o melhor desempenho, mas em ilustrar de forma mais didática possível o uso do simulador SMS-MC e sua estrutura de memória compartilhada. Este aplicativo recebe como entrada dois vetores (possíveis de serem multiplicados entre si), calcula o resultado do produto dos mesmos e devolve como saída o resultado do cálculo destes dois vetores.

O aplicativo *campo* foi desenvolvido com objetivo de ilustrar uma aplicação prática de engenharia, que realiza o cálculo do campo elétrico gerado pela aproximação de duas cargas elétricas puntiformes. Novamente lembramos que o objetivo foi criar um aplicativo didático, sem focar otimizações de desempenho. Este aplicativo recebe como entrada informações sobre as cargas elétricas e posicionamento das cargas puntiformes no espaço 2D (duas dimensões), gerando como saída um arquivo com o campo elétrico gerado por estas cargas. Este arquivo de saída pode ser processado pelo aplicativo *gnuplot* para gerar um gráfico que representa o campo

[1] É importante notar que estes aplicativos foram desenvolvidos sem levar em consideração questões sobre otimização do código paralelo, visando simplesmente demonstrar a utilização do simulador SMS-MC.

elétrico calculado pelo simulador.

O aplicativo *whetstone* foi desenvolvido baseado numa versão escrita em linguagem C do consagrado *benchmark Whetstone* (CURNOW; WICHMANN, 1976). Este *benchmark* é composto por onze módulos independentes que realizam diversos cálculos em variáveis de ponto flutuante. Na implementação para o SMS-MC, cada um destes módulos do *benchmark* foi implementado em um aplicativo separado, a fim de serem executados em CPUs diferentes no simulador.

6.2 Apresentação dos Resultados

Apresentamos aqui os resultados e estatísticas gerados através da aplicação dos *benchmarks* no simulador, demonstrando os valores gerados pelo simulador no quesito memória compartilhada, mostrando como o mesmo pode ser utilizado para ensino e pesquisa de Arquitetura de Computadores.

6.2.1 Benchmark 1: "Randomize"

Neste *benchmark*, conforme já descrito anteriormente, implementamos um código paralelo com objetivo único de ilustrar o uso de variáveis compartilhadas e o funcionamento do simulador.

O *benchmark* é composto por três aplicativos (*randomize1*, *randomize2* e *randomize3*), cada qual implementado no seu respectivo arquivo (*randomize1.c*, *randomize2.c* e *randomize3.c*) .

As variáveis compartilhadas deste *benchmark* foram definidas no arquivo *shared.def*, conforme o padrão descrito anteriormente. O aplicativo *randomize.c* foi escolhido para inicialização do ambiente compartilhado (execução da função *run_shared()*) e das variáveis compartilhadas (execução da função *SHARED()*).

Este *benchmark* utiliza duas variáveis compartilhadas do tipo inteiro: uma chamada `soma` e outra chamada `mult`, que é uma matriz, conforme pode ser visto no fragmento `Código 12`.

Código 12 Definição de variáveis do Benchmark 1

```
unsigned int soma;
unsigned int mult[50];
```

Os aplicativos *randomize1* e *randomize2* realizam alguns cálculos matemáticos simples sobre estas duas variáveis compartilhadas, enquanto o aplicativo *randomize3* somente executa

algumas operações que nada interferem nas variáveis compartilhadas.

A execução deste *benchmark* foi iniciada com a seguinte chamada:

```
./sms-mc  3  -redir:sim      sim.out
             -redir:prog_00  randomize1.out
             -redir:prog_01  randomize2.out
             -redir:prog_02  randomize3.out
             executa
```

O arquivo `executa` armazena os nomes dos aplicativos que serão carregados para execução em cada processador. No caso deste *benchmark*, o arquivo contém os seguintes valores: *randomize1*, *randomize2* e *randomize3*.

As estatísticas referentes à *cache* de dados compartilhados nível 1 situada no simulador com o nome DL1 estão representadas na Tabela 6.1, considerando que este *benchmark* foi executado uma vez utilizando o protocolo de coerência de *cache Snoopy* e outra vez utilizando o protocolo de coerência de *cache Directory* disponíveis no simulador.

A Tabela 6.1 apresenta em sua primeira coluna os parâmetros que são fornecidos nas estatísticas do simulador SMS-MC referentes à coerência de *cache* e dados compartilhados. A segunda, terceira e quarta colunas da tabela representam os valores das estatísticas do simulador referentes às três CPUs utilizadas por este *benchmark* utilizando o protocolo *Snoopy*. As colunas cinco, seis e sete representam os valores das estatísticas do simulador referentes às três CPUs utilizadas por este *benchmark* utilizando o protocolo *Directory*.

O parâmetro `mesi_access` representa o total de acessos a dados compartilhados, incluindo operações de `leitura` e `escrita` nos dados da *cache*. Este valor é a soma dos dois parâmetros sub-seqüentes: `mesi_hits` (blocos que já estavam na *cache* local quando foram solicitados) e `mesi_missess` (blocos que não estavam na *cache* local quando foram solicitados, gerando falha da *cache*, exigindo que o dado fosse carregado da memória).

Os parâmetros `mesi_replacements`, `mesi_writebacks`, `mesi_invalidations` e `mesi_share` representam respectivamente a quantidade de substituição de blocos compartilhados que estavam armazenados na *cache*, quantidade de blocos compartilhados que estavam na *cache* e foram escritos na memória (*write-back*), quantidade de invalidações que foram enviadas pelo barramento e quantidade de solicitações de compartilhamento que foram enviadas pelo barramento.

As estatísticas `mesi_bus_read` e `mesi_bus_write` representam a quantidade de leituras que ocorreram no barramento, assim como a quantidade de escritas ao barramento realizadas pelos

controladores de *caches*.

Os valores `mesi_miss_rate`, `mesi_repl_rate`, `mesi_wb_rate` e `mesi_inv_rate` representam respectivamente a taxa de faltas na *cache*, a taxa de substituição na *cache*, a taxa de *write-backs* na *cache* e a taxa de invalidações na *cache*, todos referentes às variáveis compartilhadas. Por fim, o parâmetro `shared_bytes` representa a quantidade de *bytes* que estão sendo compartilhados pelo *benchmark* no simulador.

	Snoopy			**Directory**		
	CPU-1	**CPU-2**	**CPU-3**	**CPU-1**	**CPU-2**	**CPU-3**
`mesi_access`	4068	6543	0	4068	6543	0
`mesi_hits`	55	2538	0	55	2538	0
`mesi_misses`	4013	4005	0	4013	4005	0
`mesi_replacements`	0	0	0	0	0	0
`mesi_writebacks`	4005	4011	0	4005	4011	0
`mesi_invalidations`	4012	4005	0	4012	4005	0
`mesi_share`	1	12	0	1	12	0
`mesi_bus_read`	18858	21254	10694	14140	12728	10580
`mesi_bus_write`	22442	25814	10580	18388	17405	10580
`mesi_miss_rate`	0,9865	0,6121	0,0000	0,9865	0,6121	0,0000
`mesi_repl_rate`	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
`mesi_wb_rate`	0,9845	0,6130	0,0000	0,9845	0,6130	0,0000
`mesi_inv_rate`	0,9862	0,6121	0,0000	0,9862	0,6121	0,0000
`shared_bytes`	204	204	204	204	204	204

Tabela 6.1: *Resultados do Benchmark 1*

Como pode ser visto na Tabela 6.1 ou graficamente na figura 6.1, considerando o parâmetro `mesi_bus_read`, o protocolo *Directory* foi $25,02\%$ melhor que o protocolo *Snoopy* para o aplicativo 1 do *benchmark* 1 que foi executado pela CPU1 do SMS-MC. Para o aplicativo 2 do *benchmark* 1, que foi executado na CPU2, o protocolo *Directory* foi $40,11\%$ melhor que o protocolo *Snoopy*. No caso do aplicativo 3 do *benchmark* 1, que foi executado na CPU3 do SMS-MC, o ganho do protocolo *Directory* foi de $1,07\%$, isto porque este aplicativo não realiza qualquer operação sobre as variáveis compartilhadas, tendo-as somente declaradas.

Analisando o parâmetro `mesi_bus_write`, segundo a Tabela 6.1, podemos notar que o protocolo *Directory* foi $18,06\%$ melhor que o protocolo *Snoopy* para o aplicativo 1 do *benchmark* 1. O aplicativo 2 do *benchmark* 1 teve um desempenho $32,58\%$ melhor utilizando o protocolo *Directory* do que o protocolo *Snoopy*. Para o aplicativo 3 do *benchmark* 1 os valores de ambos protocolos foram iguais.

A figura 6.1 mostra de forma gráfica a quantidade de `leituras` e `escritas` no barramento para o *benchmark* 1 utilizando os protocolos de coerência *Snoopy* e *Directory*.

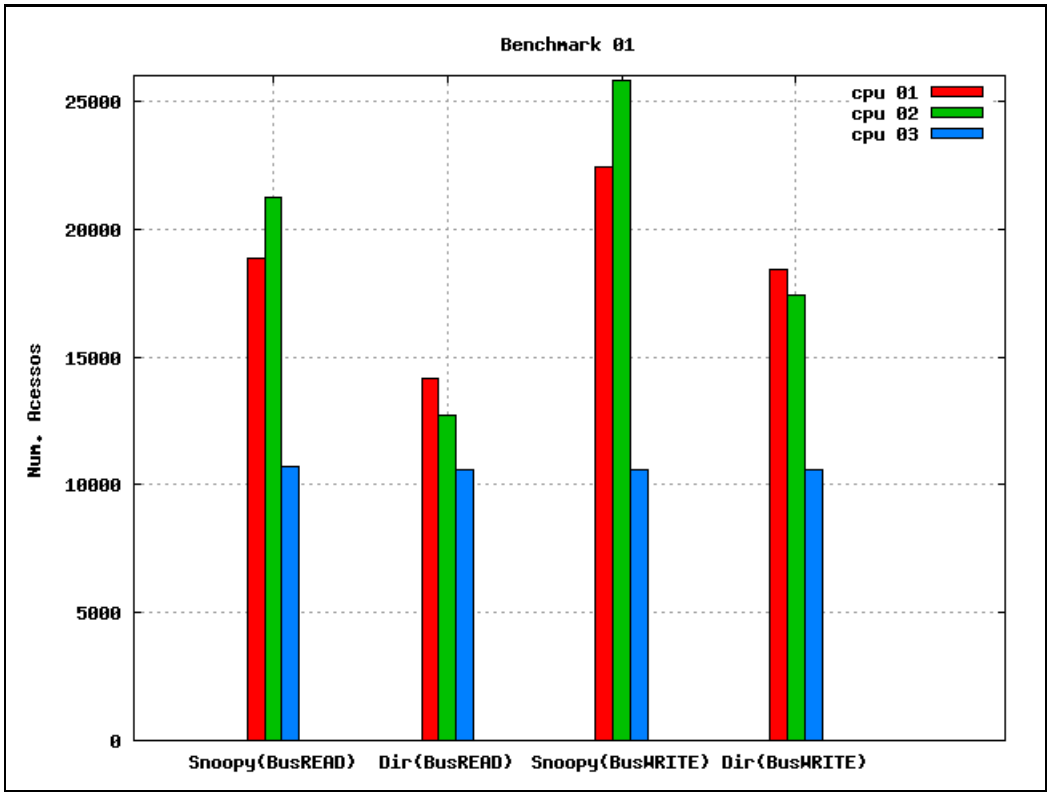

Figura 6.1: *Acessos ao barramento realizados pelo Benchmark 1*

Na figura 6.1, o eixo das ordenadas representa o total de acessos ao barramento e o eixo das abcissas representa os parâmetros de `leitura` e `escrita` para ambos os protocolos: *Snoopy* e *Directory*. `Snoopy(BusREAD)` representa o valor do parâmetro `mesi_bus_read` para o protocolo *Snoopy*. `Dir(BusREAD)` representa o valor do parâmetro `mesi_bus_read` para o protocolo *Directory*. `Snoopy(BusWRITE)` representa o valor do parâmetro `mesi_bus_write` para o protocolo *Snoopy*. `Dir(BusWRITE)` representa o valor do parâmetro `mesi_bus_write` para o protocolo *Directory*.

Cada um dos três aplicativos do *benchmark* 1 é executado numa CPU separada, portanto, para um mesmo parâmetro (`Snoopy/BusRead`, `Snoopy/BusWrite`, `Dir/BusRead` ou `Dir/BusWrite`), o gráfico apresenta três valores, um para cada aplicativo, representados pelas colunas com cores diferentes.

6.2.2 Benchmark 2: Produto de Vetores”

Este *benchmark* implementa um código paralelo capaz de realizar o cálculo do produto de dois vetores, armazenando o resultado num terceiro vetor. Não observamos características de otimização na implementação do *benchmark*, focando principalmente a demonstração do uso do simulador SMS-MC.

O *benchmark* é composto por dois aplicativos (*vetor1* e *vetor2*), cada um implementado no seu respectivo arquivo (*vetor1.c* e *vetor2.c*).

As variáveis compartilhadas deste *benchmark* foram definidas no arquivo *shared.def* conforme o padrão descrito anteriormente. O aplicativo *vetor1.c* foi escolhido para inicialização do ambiente compartilhado (execução da função *run_shared()*) e das variáveis compartilhadas (execução da função *SHARED()*).

Este *benchmark* utiliza três variáveis compartilhadas do tipo vetor de inteiros: uma chamada `matA[250]`, uma chamada `matB[250]` e outra chamada `matC[250]`, conforme pode ser visto no fragmento `Código 13`.

Código 13 Definição de variáveis do Benchmark 2

```
int sem_ld_vars;
unsigned long int matA[250];
unsigned long int matB[250];
unsigned long int matC[250];
```

O aplicativo *vetor1* inicializa as variáveis `matA` e `matB`, associando valores aleatórios para cada uma das posições destes vetores. Após esta inicialização das variáveis, os dois aplicativos *vetor1* e *vetor2* realizam os cálculos para determinar o valor da multiplicação destes dois vetores.

A execução deste *benchmark* foi iniciada com a seguinte chamada:

```
./sms-mc  2  -redir:sim      sim.out
             -redir:prog_00  vetor1.out
             -redir:prog_01  vetor2.out
             executa
```

O conteúdo do arquivo `executa` armazena os nomes dos aplicativos que serão carregados para execução em cada processador. No caso deste *benchmark*, o arquivo contém os seguintes valores: *vetor1* e *vetor2*.

As estatísticas referentes à *cache* de dados compartilhados nível 1 situada no simulador com o nome DL1 estão representadas na Tabela 6.2, considerando que este *benchmark* foi executado uma vez utilizando o protocolo de coerência de *cache Snoopy* e outra vez utilizando o protocolo de coerência de *cache Directory* disponíveis no simulador.

A Tabela 6.2 apresenta em sua primeira coluna os parâmetros que são fornecidos nas estatísticas do simulador SMS-MC referentes à coerência de *cache* e dados compartilhados. A segunda e terceira coluna da tabela representam os valores das estatísticas do simulador refe-

rentes às duas CPUs utilizadas por este *benchmark* utilizando o protocolo *Snoopy*. As colunas quatro e cinco representam os valores das estatísticas do simulador referentes às duas CPUs utilizadas por este *benchmark* utilizando o protocolo *Directory*. O significado de cada parâmetro disponível na tabela são os mesmos que os dos parâmetros do *benchmark* 1 conforme descritos na seção anterior.

	Snoopy		**Directory**	
	CPU1	**CPU2**	**CPU1**	**CPU2**
`mesi_access`	1935	429	1935	429
`mesi_hits`	1799	381	1799	381
`mesi_misses`	136	48	136	48
`mesi_replacements`	30	0	30	0
`mesi_writebacks`	50	11	50	11
`mesi_invalidations`	17	9	17	9
`mesi_share`	251	1039	251	1039
`mesi_bus_read`	3458	4238	3201	2621
`mesi_bus_write`	5853	3159	5839	3132
`mesi_miss_rate`	0,0703	0,1119	0,0703	0,1119
`mesi_repl_rate`	0,0155	0,0000	0,0155	0,0000
`mesi_wb_rate`	0,0258	0,0256	0,0258	0,0256
`mesi_inv_rate`	0,0088	0,0210	0,0088	0,0210
`shared_bytes`	3004	3004	3004	3004

Tabela 6.2: *Resultados do Benchmark 2*

Como pode ser visto na Tabela 6.2 ou graficamente na figura 6.2, considerando o parâmetro `mesi_bus_read`, o protocolo *Directory* foi $7,43\%$ melhor que o protocolo *Snoopy* para o aplicativo 1 do *benchmark* 2 que foi executado na CPU1 do SMS-MC. Para o aplicativo 2 do *benchmark* 2, que foi executado na CPU2, o protocolo *Directory* foi $38,15\%$ melhor que o protocolo *Snoopy*.

Analisando o parâmetro `mesi_bus_write` segundo a Tabela 6.2 podemos notar que o protocolo *Directory* foi $0,24\%$ melhor que o protocolo *Snoopy* para o aplicativo 1 do *benchmark* 2. O aplicativo 2 do *benchmark* 2 melhorou $0,85\%$ utilizando o protocolo *Directory* ao invés do protocolo *Snoopy*.

Na figura 6.2, o eixo das ordenadas representa o total de acessos ao barramento e o eixo das abcissas representa os parâmetros de `leitura` e `escrita` para ambos os protocolos: *Snoopy* e *Directory*. `Snoopy(BusREAD)` representa o valor do parâmetro `mesi_bus_read` para o protocolo *Snoopy*. `Dir(BusREAD)` representa o valor do parâmetro `mesi_bus_read` para o protocolo *Directory*. `Snoopy(BusWRITE)` representa o valor do parâmetro `mesi_bus_write` para o protocolo *Snoopy*. `Dir(BusWRITE)` representa o valor do parâmetro `mesi_bus_write` para o protocolo *Directory*.

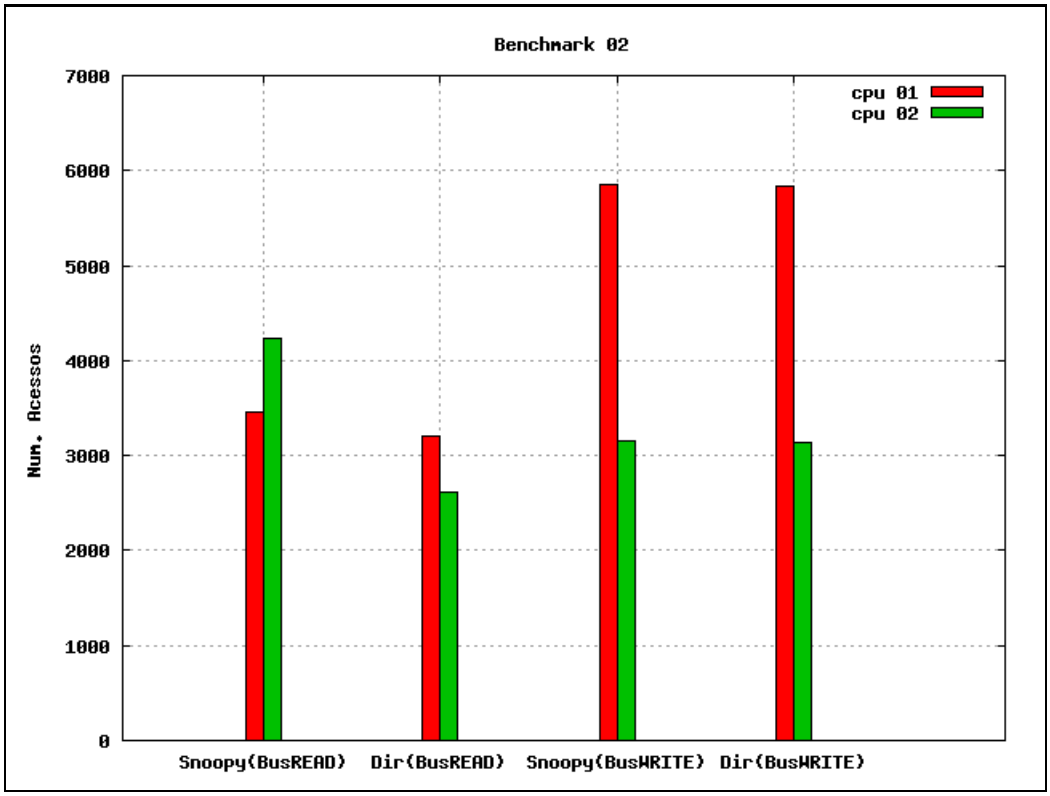

Figura 6.2: *Acessos ao barramento realizados pelo Benchmark 2*

Cada um dos dois aplicativos do *benchmark* 2 é executado numa CPU separada, portanto, para um mesmo parâmetro (`Snoopy/BusRead`, `Snoopy/BusWrite`, `Dir/BusRead` ou `Dir/BusWrite`), o gráfico apresenta dois valores, um para cada aplicativo, representados pelas colunas com cores diferentes.

A figura 6.2 mostra de forma gráfica a quantidade de `leituras` e `escritas` no barramento para o *Benchmark* 2 utilizando os protocolos de coerência *Snoopy* e *Directory*.

6.2.3 Benchmark 3: "Cálculo do Campo Elétrico"

Este *benchmark* implementa um código paralelo capaz de realizar o cálculo do campo elétrico gerado pela aproximação de quatro cargas elétricas puntiformes. O resultado deste *benchmark* gera um arquivo que pode ser processado pelo aplicativo *gnuplot* gerando gráficos 2D representando visualmente o campo elétrico calculado, como o exemplo da Figura 6.3.

O *benchmark* é composto por quatro aplicativos (*campo1*, *campo2*, *campo3* e *campo4*), cada um implementado no seu respectivo arquivo (*campo1.c*, *campo2.c*, *campo3.c*, e *campo4.c*).

As variáveis compartilhadas deste *benchmark* foram definidas no arquivo *shared.def* conforme o padrão descrito anteriormente. O aplicativo *campo1.c* foi escolhido para inicialização

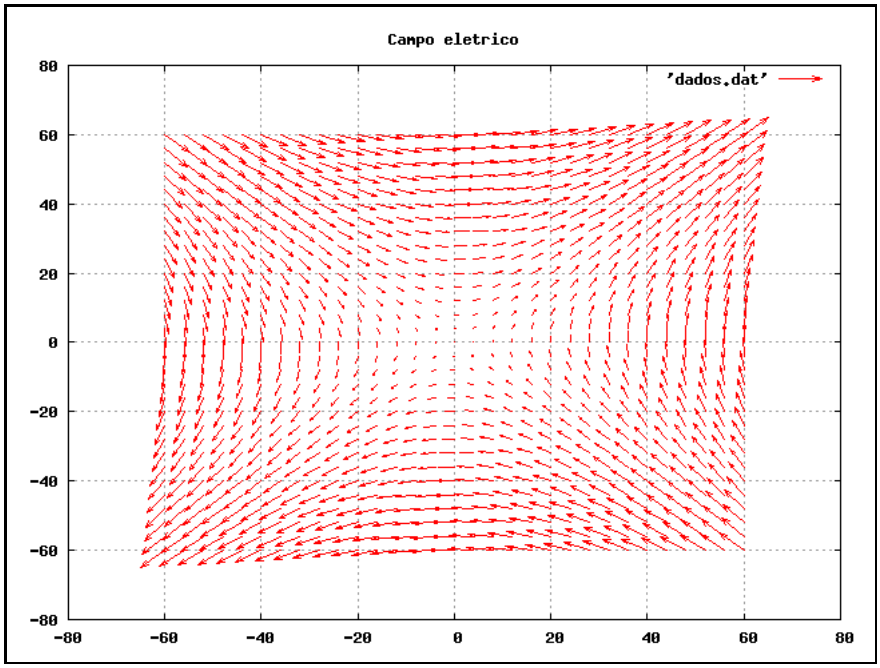

Figura 6.3: *Gráfico do Campo Elétrico gerado pela aproximação de cargas elétricas*

do ambiente compartilhado (execução da função *run_shared()*) e das variáveis compartilhadas (execução da função *SHARED()*).

Este *benchmark* utiliza algumas variáveis compartilhadas dos tipos inteiro e ponto flutuante, conforme pode ser visto no fragmento de `Código 14`. Estas variáveis são utilizadas para armazenar a carga elétrica de cada carga puntiforme e sua posição no espaço 2D. Estas informações são utilizadas pelos quatro aplicativos que compõem o *benchmark* de forma que cada um dos aplicativos consiga calcular o impacto de uma das quatro cargas no espaço e nas outras três cargas.

Código 14 Definição de variáveis do Benchmark 3

```
int sem_ld_vars1;
int sem_ld_vars2;
int sem_ld_vars3;
int sem_ld_vars4;
float C,q1,q2,q3,q4;
float E1[4];
float R1[4];
float E2[4];
float R2[4];
float E3[4];
float R3[4];
float E4[4];
float R4[4];
```

O aplicativo *campo1* inicializa as variáveis compartilhadas, associando cargas elétricas e posições para cada uma das cargas puntiformes. Após esta inicialização das variáveis, os quatro aplicativos realizam os cálculos para determinar o campo elétrico gerado pela ação das cargas. Cada aplicativo fica responsável pelo cálculo relativo a uma das cargas.

A execução deste *benchmark* foi iniciada com a seguinte chamada:

```
./sms-mc  4  -redir:sim      sim.out
             -redir:prog_00  campo1.out
             -redir:prog_01  campo2.out
             -redir:prog_02  campo3.out
             -redir:prog_02  campo4.out
             executa
```

O conteúdo do arquivo `executa` armazena os nomes dos aplicativos que serão carregados para execução em cada processador. No caso deste *benchmark*, o arquivo contém os seguintes valores: *campo1*, *campo2*, *campo3* e *campo4*.

As estatísticas referentes à *cache* de dados compartilhados nível 1 situada no simulador com o nome DL1 estão representadas na Tabela 6.3, considerando que este *benchmark* foi executado uma vez utilizando o protocolo de coerência de *cache Snoopy* e outra vez utilizando o protocolo de coerência de *cache Directory* disponíveis no simulador.

A Tabela 6.3 apresenta em sua primeira coluna os parâmetros que são fornecidos nas estatísticas do simulador SMS-MC referentes à coerência de *cache* e dados compartilhados. A segunda, terceira, quarta e quinta coluna da tabela representam os valores das estatísticas do simulador referentes às quatro CPUs utilizadas por este *benchmark* utilizando o protocolo *Snoopy*. As colunas seis, sete, oito e nove representam os valores das estatísticas do simulador referentes às quatro CPUs utilizadas por este *benchmark* utilizando o protocolo *Directory*.

Como pode ser visto na Tabela 6.3 ou graficamente na figura 6.4, considerando o parâmetro `mesi_bus_read`, o protocolo *Directory* foi $20,56\%$ melhor que o protocolo *Snoopy* para o aplicativo 1 do *benchmark* 3 que foi executado pela CPU1 do SMS-MC. Para o aplicativo 2 do *benchmark* 3, que foi executado na CPU2, o protocolo *Directory* foi $21,56\%$ melhor que o protocolo *Snoopy*. No caso do aplicativo 3 do *benchmark* 3, que foi executado na CPU3 do SMS-MC, o ganho do protocolo *Directory* foi de $23,45\%$. O aplicativo 4 do *benchmark* 3 obteve um ganho de desempenho de $56,22\%$ usando o protocolo *Directory* ao invés do protocolo *Snoopy*.

Analisando o parâmetro `mesi_bus_write` segundo a Tabela 6.3 podemos notar que o pro-

tocolo *Directory* foi $0,001\%$ melhor que o protocolo *Snoopy* para o aplicativo 1 do *benchmark* 3. O aplicativo 2 do *benchmark* 3 teve um desempenho $0,0004\%$ melhor utilizando o protocolo *Directory* do que o protocolo *Snoopy*. Para o aplicativo 3 do *benchmark* 3 o ganho do protocolo *Directory* sobre o protocolo *Snoopy* foi de $0,0002\%$. Já para o aplicativo 4 do *benchmark* 3 o ganho com o protocolo *Directory* em relação ao protocolo *Snoopy* foi de $25,71\%$.

A figura 6.4 mostra de forma gráfica a quantidade de `leituras` e `escritas` no barramento para o *Benchmark* 3 utilizando os protocolos de coerência *Snoopy* e *Directory*.

Na figura 6.4, o eixo das ordenadas representa o total de acessos ao barramento e o eixo das abcissas representa os parâmetros de leitura e escrita para ambos os protocolos: *Snoopy* e *Directory*. `Snoopy(BusREAD)` representa o valor do parâmetro `mesi_bus_read` para o protocolo *Snoopy*. `Dir(BusREAD)` representa o valor do parâmetro `mesi_bus_read` para o protocolo *Directory*. `Snoopy(BusWRITE)` representa o valor do parâmetro `mesi_bus_write` para o protocolo *Snoopy*. `Dir(BusWRITE)` representa o valor do parâmetro `mesi_bus_write` para o protocolo *Directory*.

Cada um dos quatro aplicativos do *benchmark* 3 é executado numa CPU separada, portanto, para um mesmo parâmetro (`Snoopy/BusRead`, `Snoopy/BusWrite`, `Dir/BusRead` ou `Dir/BusWrite`), o gráfico apresenta quatro valores, um para cada aplicativo, representados pelas colunas com cores diferentes.

6.2.4 Benchmark 4: "Versão customizada do Whetstone"

Este *benchmark* é uma customização da versão do *Whetstone Benchmark* escrita em linguagem C. A versão original do *benchmark* possui onze módulos que realizam cálculos de ponto flutuante diversos. Nesta versão customizada para o SMS-MC, alterações foram realizadas no código original para permitir que os módulos executem em CPUs diferentes do simulador. Assim como a versão original em linguagem C, esta versão customizada também não possui o módulo cinco do *benchmark*.

O *benchmark* é composto por dez aplicativos (*whet1*, *whet2*, *whet3*, *whet4*, *whet6*, *whet7*, *whet8*, *whet9*, *whet10* e *whet11*), cada um implementado no seu respectivo arquivo (*whet1.c*, *whet2.c*, *whet3.c*, *whet4.c*, *whet6.c*, *whet7.c*, *whet8.c*, *whet9.c*, *whet10.c* e *whet11.c*).

As variáveis compartilhadas deste *benchmark* foram definidas no arquivo *shared.def* conforme o padrão descrito anteriormente. O aplicativo *whet1.c* foi escolhido para inicialização do ambiente compartilhado (execução da função *run_shared()*) e das variáveis compartilhadas (execução da função *SHARED()*). Este aplicativo do *benchmark* também é responsável por

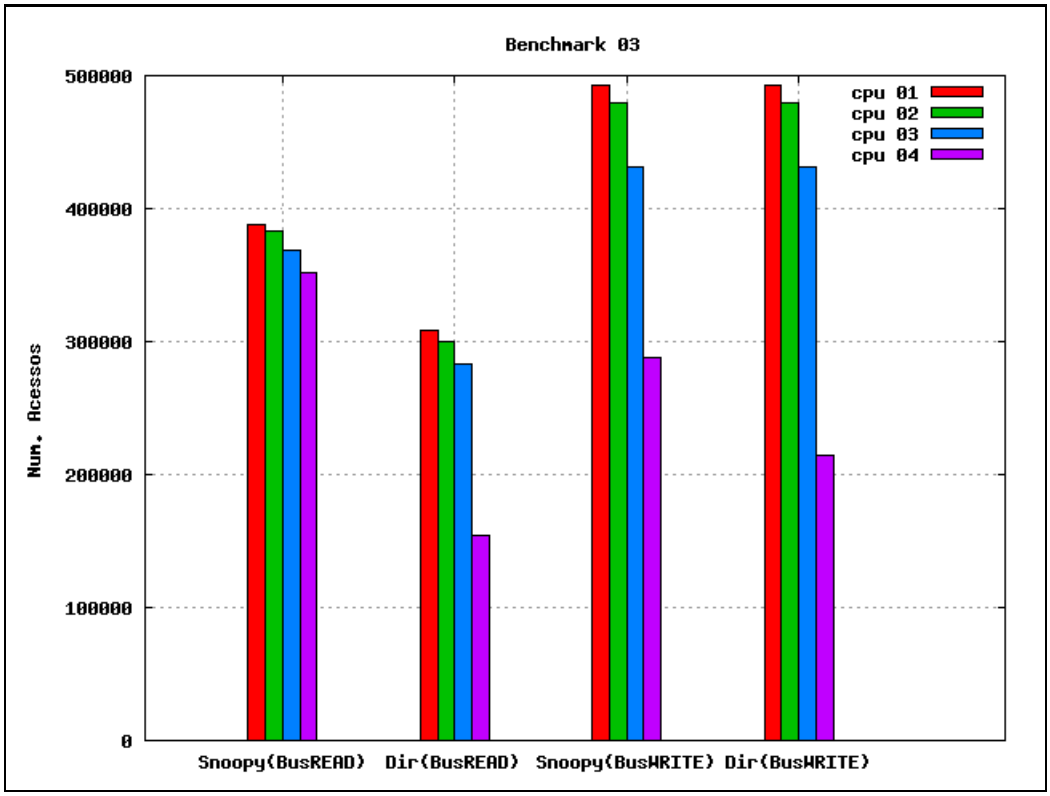

Figura 6.4: *Acessos ao barramento realizados pelo Benchmark 3*

aguardar o término de todos os outros aplicativos e gerar o resultado final do *benchmark*.

Este *benchmark* utiliza diversas variáveis compartilhadas, como pode ser visto no fragmento `Código 15`.

O aplicativo *whet1* inicializa as variáveis principais do *benchmark*, que são utilizadas nos outros aplicativos do mesmo. Após esta inicialização das variáveis, todos os aplicativos executam paralelamente de maneira que cada um dos módulos do *Whetstone* seja executado em uma CPU do simulador.

A execução deste *benchmark* foi iniciada com a seguinte chamada:

Código 15 Definição de variáveis do Benchmark 4

```
int sm1, sm2, sm3, sm4, sm6, sm7, sm8, sm9, sm10, sm11;
int n1, n2, n3, n4, n6, n7, n8, n9, n10, n11;
double t, t1, t2;
double M1_x1, M1_x2, M1_x3, M1_x4;
double M2_e1[4];
double M3_e1[4];
int M4_j;
int M6_j, M6_k, M6_l;
double M6_e1[4];
double M7_x, M7_y;
double M8_x, M8_y, M8_z;
int M9_j, M9_k, M9_l;
double M9_e1[4] ;
int M10_j, M10_k;
double M11_x;
```

```
./sms-mc  10  -redir:sim      sim.out
              -redir:prog_00  whet1.out
              -redir:prog_01  whet2.out
              -redir:prog_02  whet3.out
              -redir:prog_03  whet4.out
              -redir:prog_04  whet6.out
              -redir:prog_05  whet7.out
              -redir:prog_06  whet8.out
              -redir:prog_07  whet9.out
              -redir:prog_08  whet10.out
              -redir:prog_09  whet11.out
              executa
```

O conteúdo do arquivo `executa` armazena os nomes dos aplicativos que serão carregados para execução em cada processador. No caso deste *benchmark*, o arquivo contém os seguintes valores: *whet1*, *whet2*, *whet3*, *whet4*, *whet6*, *whet7*, *whet8*, *whet9*, *whet10* e *whet11*.

As estatísticas referentes à *cache* de dados compartilhados nível 1 situada no simulador com o nome DL1 estão representadas na Tabela 6.4 e na Tabela 6.5, considerando que este *benchmark* foi executado uma vez utilizando o protocolo de coerência de *cache Snoopy* e outra vez utilizando o protocolo de coerência de *cache Directory* disponíveis no simulador.

A Tabela 6.4 apresenta em sua primeira coluna os parâmetros que são fornecidos nas estatísticas do simulador SMS-MC referentes à coerência de *cache* e dados compartilhados. As

demais colunas da tabela representam os valores das estatísticas do simulador referentes às dez CPUs utilizadas por este *benchmark* utilizando o protocolo *Snoopy*. Os significados dos parâmetros disponíveis na tabela são os mesmos que os dos parâmetros do *benchmark* 1 conforme descritos na seção anterior.

A Tabela 6.5 apresenta em sua primeira coluna os parâmetros que são fornecidos nas estatísticas do simulador SMS-MC referentes à coerência de *cache* e dados compartilhados. As demais colunas da tabela representam os valores das estatísticas do simulador referentes às dez CPUs utilizadas por este *benchmark* utilizando o protocolo *Directory*. Os significados dos parâmetros disponíveis na tabela são os mesmos que os dos parâmetros do *benchmark* 1 conforme descritos na seção anterior.

Como pode ser visto nas Tabelas 6.4 e 6.5 ou graficamente na figura 6.5, considerando o parâmetro `mesi_bus_read`, o protocolo *Directory* foi $71,22\%$ melhor que o protocolo *Snoopy* para o aplicativo 1 do *benchmark* 4 que foi executado na CPU1 do SMS-MC. Para o aplicativo 2 do *benchmark* 4, que foi executado na CPU2, o protocolo *Directory* foi $50,56\%$ melhor que o protocolo *Snoopy*. Já no aplicativo 3 do *benchmark* 4, executado na CPU3, o protocolo *Directory* foi $26,15\%$ melhor que o protocolo *Snoopy*. O protocolo *Directory* foi $29,24\%$ melhor que o protocolo *Snoopy* para o aplicativo 4 do *benchmark* 4 que foi executado na CPU4 do SMS-MC. Para o aplicativo 6 do *benchmark* 4, que foi executado na CPU5, o protocolo *Directory* foi $21,33\%$ melhor que o protocolo *Snoopy*. Já no aplicativo 7 do *benchmark* 4, executado na CPU6, o protocolo *Directory* foi $41,02\%$ melhor que o protocolo *Snoopy*. O protocolo *Directory* foi $9,68\%$ melhor que o protocolo *Snoopy* para o aplicativo 8 do *benchmark* 4 que foi executado na CPU7 do SMS-MC. Para o aplicativo 9 do *benchmark* 4, que foi executado na CPU8, o protocolo *Directory* foi $34,61\%$ melhor que o protocolo *Snoopy*. Já no aplicativo 10 do *benchmark* 4, executado na CPU9, o protocolo *Directory* foi $76,60\%$ melhor que o protocolo *Snoopy*. O protocolo *Directory* foi $62,86\%$ melhor que o protocolo *Snoopy* para o aplicativo 11 do *benchmark* 4 que foi executado na CPU10 do SMS-MC.

Como pode ser visto nas Tabelas 6.4 e 6.5 ou graficamente na figura 6.5, considerando o parâmetro `mesi_bus_write`, o protocolo *Directory* foi $0,07\%$ melhor que o protocolo *Snoopy* para o aplicativo 1 do *benchmark* 4 que foi executado na CPU1 do SMS-MC. Para o aplicativo 2 do *benchmark* 4, que foi executado na CPU2, o protocolo *Directory* foi $0,05\%$ melhor que o protocolo *Snoopy*. Já no aplicativo 3 do *benchmark* 4, executado na CPU3, o protocolo *Directory* foi $0,01\%$ melhor que o protocolo *Snoopy*. O protocolo *Directory* foi $0,01\%$ melhor que o protocolo *Snoopy* para o aplicativo 4 do *benchmark* 4 que foi executado na CPU4 do SMS-MC. Para o aplicativo 6 do *benchmark* 4, que foi executado na CPU5, o protocolo *Directory* foi $0,004\%$ melhor que o protocolo *Snoopy*. Já no aplicativo 7 do *benchmark* 4, executado na

CPU6, o protocolo *Directory* foi $0,98\%$ melhor que o protocolo *Snoopy*. O protocolo *Directory* foi $0,0005\%$ melhor que o protocolo *Snoopy* para o aplicativo 8 do *benchmark* 4 que foi executado na CPU7 do SMS-MC. Para o aplicativo 9 do *benchmark* 4, que foi executado na CPU8, o protocolo *Directory* foi $14,93\%$ melhor que o protocolo *Snoopy*. Já no aplicativo 10 do *benchmark* 4, executado na CPU9, o protocolo *Directory* foi $1,48\%$ melhor que o protocolo *Snoopy*. O protocolo *Directory* foi $1,99\%$ melhor que o protocolo *Snoopy* para o aplicativo 11 do *benchmark* 4 que foi executado na CPU10 do SMS-MC.

Na figura 6.5, o eixo das ordenadas representa o total de acessos ao barramento e o eixo das abcissas representa os parâmetros de `leitura` e `escrita` para ambos os protocolos: *Snoopy* e *Directory*. `Snoopy(BusREAD)` representa o valor do parâmetro `mesi_bus_read` para o protocolo *Snoopy*. `Dir(BusREAD)` representa o valor do parâmetro `mesi_bus_read` para o protocolo *Directory*. `Snoopy(BusWRITE)` representa o valor do parâmetro `mesi_bus_write` para o protocolo *Snoopy*. `Dir(BusWRITE)` representa o valor do parâmetro `mesi_bus_write` para o protocolo *Directory*.

Cada um dos dez aplicativos do *benchmark* 4 é executado numa CPU separada, portanto, para um mesmo parâmetro (`Snoopy/BusRead`, `Snoopy/BusWrite`, `Dir/BusRead` ou `Dir/BusWrite`), o gráfico apresenta dois valores, um para cada aplicativo, representados pelas colunas com cores diferentes.

A figura 6.5 mostra de forma gráfica a quantidade de `leituras` e `escritas` no barramento para o *Benchmark* 4 utilizando os protocolos de coerência *Snoopy* e *Directory*.

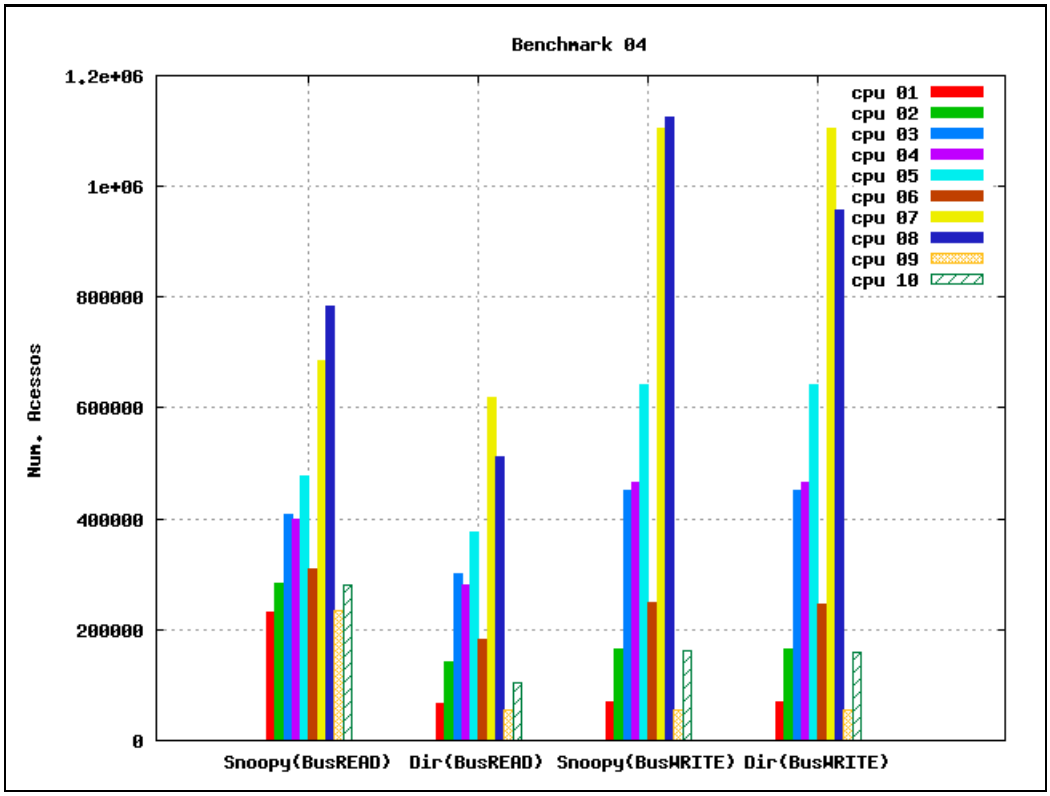

Figura 6.5: *Acessos ao barramento realizados pelo Benchmark 4*

	Snoopy				Directory			
	CPU1	**CPU2**	**CPU3**	**CPU4**	**CPU1**	**CPU2**	**CPU3**	**CPU4**
mesi_access	56215	49636	39735	34825	56215	49636	39735	34825
mesi_hits	56210	44532	39732	29722	56210	44532	39732	29722
mesi_misses	5	5104	3	5103	5	5104	3	5103
mesi_replacements	1	1	1	0	1	1	1	0
mesi_writebacks	5104	0	0	0	5104	0	0	0
mesi_invalidations	2	5102	5102	5103	2	5102	5102	5103
mesi_share	49623	43432	49323	41849	49623	43432	49323	41849
mesi_bus_read	387790	383115	369250	351919	308077	300524	282674	154066
mesi_bus_write	492190	479261	431777	288382	492187	479259	431776	214225
mesi_miss_rate	0,0001	0,1028	0,0001	0,1465	0,0001	0,1028	0,0001	0,1465
mesi_repl_rate	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
mesi_wb_rate	0,0908	0,0000	0,0000	0,0000	0,0908	0,0000	0,0000	0,0000
mesi_inv_rate	0,0000	0,1028	0,1284	0,1465	0,0000	0,1028	0,1284	0,1465
shared_bytes	164	164	164	164	164	164	164	164

Tabela 6.3: *Resultados do Benchmark 3*

	Snoopy									
	CPU1	**CPU2**	**CPU3**	**CPU4**	**CPU5**	**CPU6**	**CPU7**	**CPU8**	**CPU9**	**CPU10**
mesi_access	122	1281	7221	16667	15136	5395	53990	49436	73	5097
mesi_hits	102	1246	7184	13439	9844	5017	53968	49248	48	5086
mesi_misses	20	35	37	3228	5292	378	22	188	25	11
mesi_replacements	1	0	0	0	0	0	0	0	0	0
mesi_writebacks	6	26	21	2395	1498	264	2	200	25	16
mesi_invalidations	11	37	38	3230	5294	378	22	261	27	222
mesi_share	22723	57559	70427	27412	30674	47996	46532	33704	26698	22226
mesi_bus_read	230081	284127	409028	398413	477755	310689	684637	783216	233576	279475
mesi_bus_write	68515	164355	451970	464688	641904	247380	1105210	1124089	56383	161974
mesi_miss_rate	0,1639	0,0273	0,0051	0,1937	0,3496	0,0701	0,0004	0,0038	0,3425	0,0022
mesi_repl_rate	0,0082	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
mesi_wb_rate	0,0492	0,0203	0,0029	0,1437	0,0990	0,0489	0,0000	0,0040	0,3425	0,0031
mesi_inv_rate	0,0902	0,0289	0,0053	0,1938	0,3498	0,0701	0,0004	0,0053	0,3699	0,0436
shared_bytes	348	348	348	348	348	348	348	348	348	348

Tabela 6.4: *Resultados do Benchmark 4 para o Protocolo Snoopy*

	Directory									
	CPU1	**CPU2**	**CPU3**	**CPU4**	**CPU5**	**CPU6**	**CPU7**	**CPU8**	**CPU9**	**CPU10**
mesi_access	122	1281	7221	16667	15136	5395	53990	49436	73	5097
mesi_hits	102	1246	7184	13439	9844	5017	53968	49248	48	5086
mesi_misses	20	35	37	3228	5292	378	22	188	25	11
mesi_replacements	1	0	0	0	0	0	0	0	0	0
mesi_writebacks	6	26	21	2395	1498	264	2	200	25	16
mesi_invalidations	11	37	38	3230	5294	378	22	261	27	222
mesi_share	22723	57559	70427	27412	30674	47996	46532	33704	26698	22226
mesi_bus_read	66220	140478	302057	281916	375864	183246	618337	512152	54646	103789
mesi_bus_write	68470	164279	451924	464653	641880	244966	1105205	956245	55548	158748
mesi_miss_rate	0,1639	0,0273	0,0051	0,1937	0,3496	0,0701	0,0004	0,0038	0,3425	0,0022
mesi_repl_rate	0,0082	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
mesi_wb_rate	0,0492	0,0203	0,0029	0,1437	0,0990	0,0489	0,0000	0,0040	0,3425	0,0031
mesi_inv_rate	0,0902	0,0289	0,0053	0,1938	0,3498	0,0701	0,0004	0,0053	0,3699	0,0436
shared_bytes	348	348	348	348	348	348	348	348	348	348

Tabela 6.5: *Resultados do Benchmark 4 para o Protocolo Directory*

7 Conclusões e Trabalhos Futuros

Este capítulo sintetiza as conclusões embasado nas discussões e resultados expostos nos capítulos precedentes. Discute também as oportunidades de continuidade de trabalhos futuros.

Trabalhos de investigação sobre simuladores de arquiteturas paralelas têm sido extensivamente realizados. Entretanto, uma parcela significativa dos modelos de investigação são comumente baseados em arquiteturas com memória distribuída. O presente trabalho é importante porque contribui para estender o espectro de investigação em arquiteturas paralelas, através de um simulador de arquiteturas paralelas com memória compartilhada.

O uso de simulador facilita o processo de teste, validação e avaliação de características sem a necessidade de construção de um protótipo físico. Pode ainda servir como um grande facilitador para o estudo de arquiteturas para as quais não se tem o *hardware* disponível.

Dentre muitos simuladores de arquiteturas, o SimpleScalar tem sido muito empregado por diversos pesquisadores nas suas investigações, portanto baseamos nosso simulador nesta arquitetura já consagrada, acrescentando as características de gerenciamento de memória que propomos neste trabalho, visando fornecer a capacidade de avaliar parâmetros num sistema de memória compartilhada e fornecer um simulador para os que desejarem pesquisar e realizar experiências num *hardware* multiprocessado de memória compartilhada.

Ao concluir o projeto e implementação do Simulador de Multiprocessadores Superescalares de Memória Compartilhada obteve-se um simulador com diversas funcionalidades, dentre elas, a capacidade de:

- executar aplicativos paralelos permitindo a troca de informação através de memória compartilhada;
- fornecer uma implementação dos modelos de coerência de *cache Snoopy* e *Directory*;
- servir como plataforma para implementação de outros protocolos de coerência de *cache*, permitindo avaliação entre mais tipos de protocolos ou até mesmo a criação de novos modelos de protocolos para coerência de *cache*;

- gerar estatísticas que permitam avaliar projetos arquiteturais de máquinas paralelas superescalares de memória compartilhada;
- ajudar na avaliação de qual protocolo de coerência de *cache* apresenta melhor desempenho em cada tipo de aplicação;
- ajudar a otimizar aplicações baseado no resultado das operações de acesso à memória ou *caches*.

Como ferramenta de ensino, o simulador SMS-MC pode ser empregado tanto no ensino de disciplinas voltadas a Arquitetura de Computadores, analisando diversas configurações de *hardware* que são possíveis de se configurar, quanto à área de programação, servindo como ferramenta para simulação de aplicativos paralelos.

Com finalidade de ferramenta de pesquisa, o SMS-MC pode ser utilizado em análises quantitativas de diversos parâmetros arquiteturais referentes a máquinas paralelas, permitindo dimensionar *caches*, memórias, previsores de desvios e protocolos de coerência de *cache*. Pode ainda permitir análise de novos protocolos de coerência de *cache* e compará-los com os que já existem no simulador ou que possam ser incluídos no mesmo com o uso de módulos.

Para dar continuidade a esta pesquisa, surgem agora novos campos do simulador para serem explorados, como por exemplo a implementação de mais modelos de protocolos de coerência de *cache*.

Pode-se ainda implementar uma opção no simulador SMS-MC que permita que ao ser solicitada a simulação de um determinado *benchmark* pelo simulador, o SMS-MC disponibilize as estatísticas referentes a todos os protocolos de coerência de *cache* de uma única vez, facilitando os estudos sobre desempenho destes protocolos. Assim, ao solicitar a execução do *benchmark*, o simulador iria na verdade executar o *benchmark* para cada um dos protocolos de *coerência* e devolver os dados de uma só vez ao usuário.

Ainda como forma de continuidade do projeto, poderia ser incluída na ferramenta a capacidade de simular um ambiente com o modelo de Memória Compartilhada Distribuída, disponibilizando assim os *DSM Computers*[1], mais uma arquitetura a ser suportada pela família SimpleScalar.

[1] *Distributed Shared Computers - Computadores com Memória Compartilhada Distribuída.*

Referências Bibliográficas

AGARWAL, A. et al. The MIT Alewife machine: architecture and performance. In: *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 1995. p. 2–13. ISBN 0-89791-698-0.

ARCHIBALD, J.; BAER, J.-L. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, ACM Press, New York, NY, USA, v. 4, n. 4, p. 273–298, 1986. ISSN 0734-2071.

AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer*, IEEE Computer Society Press, v. 35, n. 2, p. 59–67, February 2002.

BARTON, C. et al. Shared memory programming for large scale machines. In: *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2006. p. 108–117. ISBN 1-59593-320-4.

BASUMALLIK, A.; EIGENMANN, R. Optimizing irregular shared-memory applications for distributed-memory systems. In: *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 2006. p. 119–128. ISBN 1-59593-189-9.

BREWER, E. A. et al. PROTEUS: A High-Performance Parallel-Architecture Simulator. *SIGMETRICS Perform. Eval. Rev.*, ACM Press, New York, NY, USA, v. 20, n. 1, p. 247–248, 1992. ISSN 0163-5999.

BURGER, D.; AUSTIN, T. M. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, ACM Press, New York, NY, USA, v. 25, n. 3, p. 13–25, 1997. ISSN 0163-5964.

BURGER, D. et al. Scaling to the End of Silicon with EDGE Architectures. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 37, n. 7, p. 44–55, 2004. ISSN 0018-9162.

CIERNIAK, M.; LI, W. Unifying data and control transformations for distributed shared-memory machines. In: *PLDI 95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 1995. p. 205–217.

COX, A. L. et al. Software versus hardware shared-memory implementation: a case study. *SIGARCH Comput. Archit. News*, ACM Press, New York, NY, USA, v. 22, n. 2, p. 106–117, 1994. ISSN 0163-5964.

CURNOW, H. J.; WICHMANN, B. A. A Synthetic benchmark. *The Computer Journal*, v. 19, n. 1, p. 43–49, 1976.

DING, J.; BHUYAN, L. N. Cache coherent shared memory hypercube multiprocessors. In: *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*. [S.l.]: IEEE, 1992. p. 515–520.

DWARKADAS, S. et al. Combining Compile-Time and Run-Time Support for Efficient Software Distributed Shared Memory. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, v. 87, n. 3, p. 476–486, 1999. Disponível em: <citeseer.ist.psu.edu/dwarkadas99combining.html>.

FLYNN, M. Some Computer Organizations And Their Effectiveness. *IEEE Transactions*, IEEE, 1972.

GOLAB, W. et al. Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. In: *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2007. p. 3–12. ISBN 978-1-59593-616-5.

GOLAB, W.; HENDLER, D.; WOELFEL, P. An O(1) RMRs leader election algorithm. In: *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2006. p. 238–247. ISBN 1-59593-384-0.

GONÇALVES, R. A. L. Arquiteturas Multi-Tarefas Simultâneas: SEMPRE - Arquitetura SMT com capacidade de execução e escalonamento de processos. In: *Tese de Doutorado*. [S.l.]: Universidade Federal do Rio Grande do Sul, 2000.

HENNESSY, J.; HEINRICH, M.; GUPTA, A. Cache-coherent distributed shared memory: Perspectives on its development and future challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, v. 87, n. 3, p. 418–429, 1999. Disponível em: <citeseer.ist.psu.edu/328804.html>.

HERROD, S. *Tango lite: A multiprocessor simulation environment*. 1993. Disponível em: <citeseer.ist.psu.edu/herrod93tango.html>.

HUGHES, C. J. et al. RSIM: simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, IEEE Computer Society Press, v. 35, n. 2, p. 40–49, February 2002.

KATZ, R. H. et al. Implementing a cache consistency protocol. *SIGARCH Comput. Archit. News*, ACM Press, New York, NY, USA, v. 13, n. 3, p. 276–283, 1985. ISSN 0163-5964.

KUSKIN, J. et al. The Stanford FLASH multiprocessor. In: *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. p. 302–313. ISBN 0-8186-5510-0.

LEBLANC, T. J.; MARKATOS, E. P. Shared memory vs. message passing in shared-memory multiprocessors. In: *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*. [S.l.]: IEEE, 1992. p. 254–263.

LENOSKI, D. et al. The directory-based cache coherence protocol for the DASH multiprocessor. In: *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*. New York, NY, USA: ACM Press, 1990. p. 148–159. ISBN 0-89791-366-3.

MAGDIC, D. Limes: A Multiprocessor Simulation Environments for PC Plataforms. *Proceedings International Conference on Microelectronics*, University of Belgrade, p. 841–844, 1997.

MALEN, G. V. C. M. J.-Y. B. I. D.; LOTTIANUX, R. Process migration based on gobelins distributed shared memory. In: *Cluster Computing and the Grid 2nd IEEE/ACM International Symposium CCGRID2002*. [S.l.]: IEEE Conference Proceeding, 2002. p. 301–306.

MANJIKIAN, N. More enhancements of the simplescalar tool set. *SIGARCH Comput. Archit. News*, ACM Press, New York, NY, USA, v. 29, n. 4, p. 5–12, 2001. ISSN 0163-5964.

MANJIKIAN, N. Multiprocessor enhancements of the SimpleScalar tool set. *SIGARCH Comput. Archit. News*, ACM Press, New York, NY, USA, v. 29, n. 1, p. 8–15, 2001. ISSN 0163-5964.

Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In: *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 1984. p. 348–354. ISBN 0-8186-0538-3.

MILTON, S. *Thread Migration in Distributed Memory Multicomputers*. Canberra 0200 ACT, Australia, fev. 1998. Disponível em: <citeseer.ist.psu.edu/milton98thread.html>.

NITZBERG, B.; LO, V. Distributed shared memory: A survey of issues and algorithms. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 24, n. 8, p. 52–60, 1991.

RAINA, S. *Virtual Shared Memory: A Survey of Techniques and Systems*. [S.l.], 1, 1992. Disponível em: <citeseer.ist.psu.edu/raina92virtual.html>.

REINHARDT, S. K. et al. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. In: *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1993. p. 48–60. ISBN 0-89791-580-1.

ROGERS, B.; PRVULOVIC, M.; SOLIHIN, Y. Efficient data protection for distributed shared memory multiprocessors. In: *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM Press, 2006. p. 84–94. ISBN 1-59593-264-X.

SANDRI, A. L.; MARTINI, J. A.; GONÇALVES, R. A. L. SMS- Tool for Development and Performance Analysis of Parallel Applications. *Proceedings of the 37th Annual Simulation Symposium (ANSS'04)*, IEEE Computer Society, 2004.

SANKARALINGAM, K. et al. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In: *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006. p. 480–491. ISBN 0-7695-2732-9.

SOHI, G. S. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 39, n. 3, p. 349–359, 1990. ISSN 0018-9340.

STENSTRöM, P. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 6, p. 12–24, 1990. ISSN 0018-9162.

STOLLER, L. C. J. S. M. Making distributed shared memory simple, yet efficient. In: *High-Level Parallel Programming Models and Supportive Environments, 1998. Proceedings. Third International Workshop on*. [S.l.]: IEEE Conference Proceeding, 1998. p. 2–13.

STUMM, M.; ZHOU, S. Algorithms implementing distributed shared memory. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 5, p. 54–64, 1990.

SUNADA, D.; GLASCO, D.; FLYNN, M. ABSS v2.0: a SPARC Simulator. *Proceedings of the 8th Workshop on Synthesis and System Integration of Mixed Technologies*, Stanford University and International Business Machines, Inc., p. 143–149, 1998.

Susan J. Eggers and Joel S. Emer and Henry M. Levy and Jack L. Lo and Rebecca L. Stamm and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, v. 17, n. 5, p. 12–21, march 1997. Disponível em: <citeseer.ist.psu.edu/eggers97simultaneous.html>.

THACKER, C. P.; STEWART, L. C. Firefly: a multiprocessor workstation. In: *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987. p. 164–172. ISBN 0-8186-0805-6.

THANALAPATI, T.; DANDAMUDI, S. An efficient adaptive scheduling scheme for distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, v. 12, n. 7, p. 758–768, July 2001.

VEENSTRA J.E.; FOWLER, R. MINT: a front end for efficient simulation of shared-memory multiprocessors. In: *Proceedings of the Second International Workshop on MASCOTS'94*. [S.l.]: IEEE, 1994. p. 201–207.

WULF, W. A. *HYDRA/C.mmp, an Experimental Computer System.* [S.l.]: McGraw-Hill, 1981.

ZHANG, W. et al. Optimizing compiler for shared-memory multiple SIMD architecture. In: *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*. New York, NY, USA: ACM Press, 2006. p. 199–208. ISBN 1-59593-362-X.

ZHOU, S. et al. Heterogeneous distributed shared memory. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 3, n. 5, p. 540–554, 1992.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas