Instituto de Computação
Universidade Estadual de Campinas

Implementação de cache no projeto ArchC

Henrique Dante de Almeida

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Henrique Dante de Almeida e aprovada pela Banca Examinadora.

Campinas, Abril de 2012.

Paulo Cesar Centoducatte (Orientador)

Rodolfo Jardim de Azevedo (Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR
ANA REGINA MACHADO - CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Almeida, Henrique Dante de, 1982-
AL64i Implementação de cache no projeto ArchC / Henrique Dante de Almeida. – Campinas, SP : [s.n.], 2012.

Orientador: Paulo Cesar Centoducatte.
Coorientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas de computação. 2. Arquitetura de computador. 3. Memória cache. I. Centoducatte, Paulo Cesar, 1957-. II. Azevedo, Rodolfo Jardim de, 1974-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

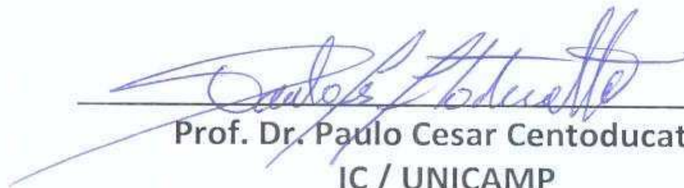
Título em inglês: Cache implementation in the ArchC project
Palavras-chave em inglês:
Computer systems
Computer architecture
Cache memory
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Paulo Cesar Centoducatte [Orientador]
Abel Guilhermino da Silva Filho
Sandro Rigo
Data de defesa: 23-04-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 23 de Abril de 2012, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Abel Guilhermino da Silva Filho
ICMC / USP

Prof. Dr. Sandro Rigo
IC / UNICAMP



Prof. Dr. Paulo Cesar Centoducatte
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Implementação de cache no projeto ArchC

Henrique Dante de Almeida[1]

Abril de 2012

Banca Examinadora:

- Paulo Cesar Centoducatte (Orientador)
- Abel Guilhermino da Silva Filho
  Universidade Federal de Pernambuco
- Sandro Rigo
  Universidade Estadual de Campinas
- Alexandro Baldassin
  Universidade Estadual Paulista "Júlio de Mesquita Filho" (Suplente)
- Edson Borin
  Universidade Estadual de Campinas (Suplente)

[1]Suporte financeiro de: Bolsa do CNPq (processo 134366/2008-5) 2008–2009

Resumo

O projeto ArchC visa criar uma linguagem de descrição de arquiteturas, com o objetivo de se construir simuladores e *toolchains* de arquiteturas computacionais completas. O objetivo deste trabalho é dotar ArchC com capacidade para gerar simuladores de caches. Para tanto foi realizado um estudo detalhado das caches (tipos, organizações, configurações etc) e do funcionamento e do código do ArchC. O resultado foi a descrição de uma coleção de caches parametrizáveis que podem ser adicionadas às arquiteturas descritas em ArchC. A implementação das caches é modular, possuindo código isolado para a memória de armazenamento da cache e políticas de operação. A corretude da cache foi verificada utilizando uma sequência de simulações de diversas configurações de *cache* e com comparações com o simulador dinero. A cache resultante apresentou um *overhead*, no tempo de simulaçao, que varia entre 10% e 60%, quando comparada a um simulador sem cache.

Abstract

The ArchC project aims to create an architecture description language, with the goal of building complete computer architecture simulators and toolchains. The goal of this project is to add support in ArchC for simulating caches. To achieve this, a detailed study about caches (types, organization, configuration etc) and about the ArchC code was done. The result was a collection of parameterized caches that may be included on the architectures described with ArchC. The cache implementation is modular, having isolated code for the storage and operation policies. Implementation correctness was verified using a set of many cache configurations and with comparisons with the results from dinero simulator. The resulting cache showed an overhead varying between 10% and 60%, when compared to a simulator without caches.

Sumário

Resumo **v**

Abstract **vi**

1 Introdução **1**

2 Conceitos Básicos **3**
2.1 Caches 3
2.2 Trabalhos relacionados 10
2.3 ArchC 13

3 Modelos **19**
3.1 *Cache* 19
3.2 ArchC 25

4 Experimentos **31**
4.1 Benchmarks 31
4.2 Cluster 32
4.3 Casos de teste 33
4.4 Desempenho 34
4.5 Validação com dinero 42
4.6 Validação com arquiteturas MIPS e PowerPC 42

5 Conclusões **48**
5.1 Trabalhos futuros 49

Bibliografia **50**

Lista de Tabelas

4.1 Lista de testes do MediaBench utilizados durante a verificação 32
4.2 Lista de testes do MiBench utilizados durante a verificação 33
4.3 Lista de configurações de cache testadas 34
4.4 Configurações de cache selecionadas para os benchmarks. Os valores representam, respectivamente, associatividade, tamanho de índice, tamanho de bloco em bytes, política de escrita (write through ou write back) e política de substituição (fifo ou random) – Programas do MediaBench 35
4.5 Configurações de cache selecionadas para os benchmarks. Os valores representam, respectivamente, associatividade, tamanho de índice, tamanho de bloco em bytes, política de escrita (write through ou write back) e política de substituição (fifo ou random) – Programas do MiBench 36
4.6 Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MediaBench (1) 37
4.7 Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MiBench (1) 38
4.8 Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MiBench (2) 39

4.9 Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MiBench (3) 40
4.10 Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MiBench (4) 41
4.11 Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random ou plrum) 47

Lista de Figuras

2.1 Diagrama esquemático de uma cache com associatividade 2, blocos de 16 bytes e 1024 blocos por via. 6
2.2 Hierarquia de memória *caches* com seus tamanhos relativos. 9
2.3 Diagrama com operação do simulador do ArchC. Na figura, os arquivos de entrada são: sparcv8.ac, sparcv8_isa.ac, sparcv8_isa.cpp. O programa acsim processa os arquivos de entrada, gerando os arquivos intermediários. Os arquivos resultantes podem ser compilados, gerando o simulador sparcv8.x, que pode receber programas de entrada. A biblioteca aclib é a parte do ArchC onde os componentes de armazenamento estão presentes. 14

3.1 Diagrama com a representação dos módulos do ArchC. Na figura, representando um exemplo de arquitetura SPARC com duas caches L1, uma cache L2 e uma memória, sparcv8 é o processador, que possui dois ac_memports para se conectar na hierarquia de memória, cada um deles conectado a um ac_cache por meio de um adaptador chamado ac_cache_if. Ambas as caches se conectam diretamente a um ac_cache, que por sua vez se conecta a um ac_mem. 26

4.1 Desaceleração devido a inclusão da cache – MiBench (1) 43
4.2 Desaceleração devido a inclusão da cache – MiBench (2) 44
4.3 Desaceleração devido a inclusão da cache – MediaBench 44
4.4 Desaceleração devido a inclusão da cache – MiBench (1) – sem decoder cache 45
4.5 Desaceleração devido a inclusão da cache – Mibench (2) – sem decoder cache 46
4.6 Desaceleração devido a inclusão da cache – MediaBench – sem decoder cache 46

Capítulo 1

Introdução

Este trabalho tem como objetivo o desenvolvimento do suporte a *caches* no projeto ArchC, que é uma linguagem de descrição de arquiteturas capaz de gerar ferramentas para facilitar a exploração do espaço de projeto de sistemas processados (SoCs, MPSoCs). As movitações para o desenvolvimento de linguagens de arquiteturas são a possibilidade de realizar validações de sistemas em estágios iniciais de desenvolvimento (em contraste com testes em *Register Transfer Level*), além de permitir uma exploração do espaço de projeto, no início do desenvolvimento de sistemas.

Em suas versões iniciais, ArchC contou com uma implementação de *cache* que podia ser instanciada junto com o processador. No entanto, esta *cache* não foi completamente validada e não possuia o desempenho desejado para o sistema final, sendo descontinuada com o lançamento da versão 2.0 do ArchC.

A nova versão de *cache* proposta neste trabalho foca nos seguinte pilares:

1. Compatibilidade com a sintaxe do simulador das versões anteriores;

2. Corretude verificada através de inúmeras configurações e execuções com outro simulador (dinero);

3. *Overhead* de desempenho pequeno em relação à execução do simulador sem a *cache*.

A compatibilidade com a sintaxe do simulador foi garantida durante a reimplementação do *parser* do ArchC e do novo gerador de código.

A implementação da *cache* deste trabalho foi feita na forma de *templates* de C++, permitindo a instanciação estática do código, evitando alocações dinâmicas de dados durante a execução e permitindo mais otimizações por parte dos compiladores.

Uma das funcionalidades da *cache* implementada é a capacidade de gerar *traces* de execução no formato da ferramenta dinero, altamente utilizada para avaliar *caches*. Assim, uma das fases de verificação foi focada em comparar os resultados gerados por ArchC com

os resultados gerados pelo dinero sobre a mesma execução. Todos os testes funcionaram corretamente.

O *overhead* do simulador variou entre 10% e 60%, em função das configurações de *caches*, valores satisfatórios, dada a inclusão da nova funcionalidade.

Esta dissertação está organizada da seguinte forma: O capítulo 2 descreve os conceitos presentes neste trabalho: a *cache*, o ArchC e os trabalhos relacionados. O capítulo 3 descreve o processo de implementação e da integração da *cache* ao ArchC. O capítulo 4 descreve os resultados experimentais e a validação. O capítulo 5 descreve as conclusões e os trabalhos futuros.

Capítulo 2

Conceitos Básicos

Este capítulo está dividido em duas partes. Na primeira parte são descritos conceitos básicos relacionados à *cache* e na segunda parte são descritas funcionalidades da linguagem de descrição de arquitetura ArchC [3, 20], necessários para o entendimento da implementação e avaliação dos resultados.

2.1 Caches

Uma demanda cada vez maior por memória nos sistemas computacionais, a grande diferença nos tempos de operação dos processadores e tempo de acesso das memórias mais baratas (dinâmicas) e o alto custo das memórias mais rápidas foram os principais motivos para o surgimento do sistema de memória hierárquico como uma solução de compromisso da relação custo-benefício [18]. Uma hierarquia de memória tem a *cache* como um dos seus principais elementos e o seu dimensionamento de forma adequada é o principal fator para o sucesso da hierarquia de memória, onde três fatores competem entre si: o tamanho da memória disponível, a velocidade de acesso e o preço. A *cache* surgiu da ideia de que, embora a memória necessária à execução de um programa/sistema possa ser muito grande, em um intervalo de tempo específico da execução, os programas utilizam um certo subconjunto da informação e seus elementos estão relacionados entre si segundo um padrão de acesso [29]. O princípio da localidade [5] explica como o padrão de acesso à memória durante a execução de um programa pode ser estimado e portanto utilizado para gerenciar as informações na hierarquia de memória. O princípio da localidade se aplica tanto para acessos a instruções quanto a dados e ele pode ser classificado em dois tipos:

Localidade temporal: se um elemento (endereço) é referenciado, ele tende a ser referenciado novamente num curto período de tempo

Localidade espacial: se um elemento (endereço) é referenciado, elementos localizados

em endereços próximos também tendem a ser referenciados num curto período de tempo

O princípio da localidade, nas suas duas vertentes, é respeitado durante a execução de um programa devido a forma como os programas são construídos [5, 18]. Por exemplo, *loops* criam um bloco de instruções e de dados com localidade, assim como dados organizados em vetores ou outras estruturas comumente utilizadas pelos programadores [21]. Pode-se tirar proveito do princípio da localidade organizando a memória de forma hierárquica de maneira que se aproveite o rápido tempo de acesso das memórias mais rápidas, a um custo total próximo ao que seria, se fossem usadas somente memórias baratas (e lentas). Em uma hierarquia de memória, cada elemento é referenciado pelo nível que ele ocupa na hierarquia, e quanto menor o valor atribuído ao nível, mais próximo ele está do processador. O nível de um elemento na hierarquia também indica a capacidade, a velocidade e o custo da memória. Quanto menor o nível, a memória utilizada é mais rápida, mais cara e, portanto, com menor capacidade. Quanto maior o nível, a memória utilizada é mais lenta, mais barata e com maior capacidade.

Neste trabalho serão considerados os níveis de hierarquia constituídos por *caches*. As *caches* são estruturas de memória que, em um determinado instante de tempo, possuem um subconjunto dos dados da memória primária e permitem um acesso rápido a eles (mais rápido que níveis posteriores na hierarquia de memória). Os dados são trazidos e mantidos na *cache* de forma aproveitar o princípio da localidade (espacial e temporal): valores com endereços próximos ao dado referenciado são trazidos para a *cache* junto com o dado e os acessados com frequência são mantidos na *cache*.

Uma *cache* pode ser definida logicamente como uma memória que possui capacidade para armazenar qualquer subconjunto, de tamanho pré-definido, de todos os dados disponíveis na memória principal, embora não simultaneamente. O armazenamento na *cache* é realizado da seguinte forma: dado um conjunto de pares (endereço, dado), representando endereços da memória primária e os dados associados a ele, a *cache* possui capacidade para armazenar um subconjunto destes dados. Para isto, o endereço é dividido em três partes: a etiqueta (*tag*), o índice e o deslocamento (*offset*) ou posição no bloco. A etiqueta corresponde aos bits mais significativos do endereço e representa os valores de endereço superiores à capacidade total de armazenamento de dados da *cache*. Isto é, a etiqueta contém todos os bits de ordem de grandeza maior do que a *cache* pode armazenar. O deslocamento corresponde aos bits menos significativos e existe porque a unidade de trabalho da cache é um bloco com várias palavras. O deslocamento serve para distinguir uma única palavra dentro do bloco armazenado. O índice corresponde aos bits intermediários e representa o endereço de um determinado bloco na *cache*. O primeiro bloco tem índice 0, o segundo tem índice 1, até o tamanho máximo, em blocos, da *cache*. Associada a um bloco de dados na *cache*, é armazenada a etiqueta, formando

o que se convencionou chamar de linha de *cache*. A etiqueta é a parte do endereço da memória primária que a *cache* não consegue tratar diretamente. Sempre que um dado é acessado, uma comparação entre etiquetas (a armazenada na linha de *cache* e a do endereço que está sendo acessado) é realizada para distinguir diferentes endereços que podem ser armazenados na mesma linha.

Para melhor explorar o principio da localidade temporal a *cache* pode ser organizada de forma que endereços com um mesmo índice possam ser armazenados em linhas (ou blocos) de *cache* diferentes e são denominadas de associativas. Diz-se que a *cache* é n-associativa ou que possui n-vias, onde n é o número de linhas diferentes onde endereços com o mesmo índice podem ser alocados.

Cada bloco contém, além da etiqueta e os dados, bits de controle em número e finalidades que dependem das características que se quer associar à *cache*. A todos os tipos de *cache* é necessário, no mínimo, um bit denominado bit de validade que sinaliza se um determinado bloco possui ou não informação válida. No caso de caches que atrasam a escrita, pode-se utilizar uma indicação de pendência, no caso de caches com coerência, é incluído o estado de coerência etc.

Em linhas gerais, dado um par (endereço, palavra de dado) ele será mapeado na *cache* da seguinte forma: o endereço é mapeado em um índice e deslocamento, podendo estar associado a uma via; o dado é mapeado a uma linha, que contém a etiqueta, os *flags* e o bloco de palavras, sendo o dado uma destas palavras, identificado pelo valor do deslocamento.

A cache fornece duas operações, similares aos acessos à memória principal: leitura e escrita. Quando uma palavra é lida ou escrita, uma linha (ou n linhas, no caso de n-associativa) da *cache* é selecionada, utilizando-se para isso o índice. A etiqueta extraída do endereço que está sendo lido ou escrito é comparada com a(s) etiqueta(s) armazenada(s) na(s) linha(s) selecionada(s). Caso o resultado da comparação seja de igualdade, então o deslocamento é usado para localizar a palavra e a operação de leitura/escrita é realizada. Caso contrário, o dado não está presente na cache e um acesso a um nível posterior da hierarquia de memória é realizado, possivelmente trazendo um bloco inteiro para a cache e armazenado na linha selecionada (no caso de n-associativa, bits de controle são consultados e determinam em que linha o bloco de dados será alocado).

Como exemplo, a Figura 2.1 ilustra uma cache em um computador com palavras de 32 bits, espaço de endereçamento de 4GB (32 bits de endereço) e acesso à memória alinhado por palavra. A *cache* tem 32 KB de capacidade, com linhas de 4 palavras (16 bytes), associatividade 2 (2 vias). Assim, tem-se 1.024 blocos por via, implicando que é necessário um índice de 10 bits. Como cada bloco possui 4 palavras é necessário um deslocamento de 2 bits. Os 2 bits menos significativos do endereço de memória são descartados no processo de acesso a um dado devido estes serem alinhados à palavra e

a memória organizada por byte. Desta forma a etiqueta é formada pelos 18 bits (32 - 10 - 2 - 2) mais significativos dos 32 bits do espaço de endereçamento da memória. Na figura, também estão representados os *flags*. Neste caso, o espaço total que a *cache* utiliza, supondo os flags compostos somente por um bit de validade, é de 36,75 KB (301.056 bits), assim divididos: 2 vias com 1024 blocos cada, totalizando 2048 blocos com 147 bits cada assim distribuídos: 4 palavras de 4 bytes (128 bits); 18 bits de etiqueta e 1 bit de validade. Isto representa um *overhead* de 15% em relação a uma memória simples. Para esta *cache* os pares de endereço e dados (0x40000000, 0x1), (0x40000004, 0x2) e (0x20001040, 0x3) possuem etiquetas 0x10000, 0x10000 e 0x8000; índices 0, 0 e 0x104 e posição no bloco 0, 1 e 0, respectivamente. Numa *cache* vazia, os três dados poderiam ser inseridos sem colisão, sendo os dois primeiros inseridos no mesmo conjunto (definido pelo índice), o primeiro na via 0 e o segundo na via 1. Um quarto elemento, com endereço diferente (0x12345040, 0x4), mas com mesmo índice que o terceiro, 0x104 também caberia na *cache*, pois o primeiro é armazenado na posição de associatividade (via) 0 e o segundo com 1.

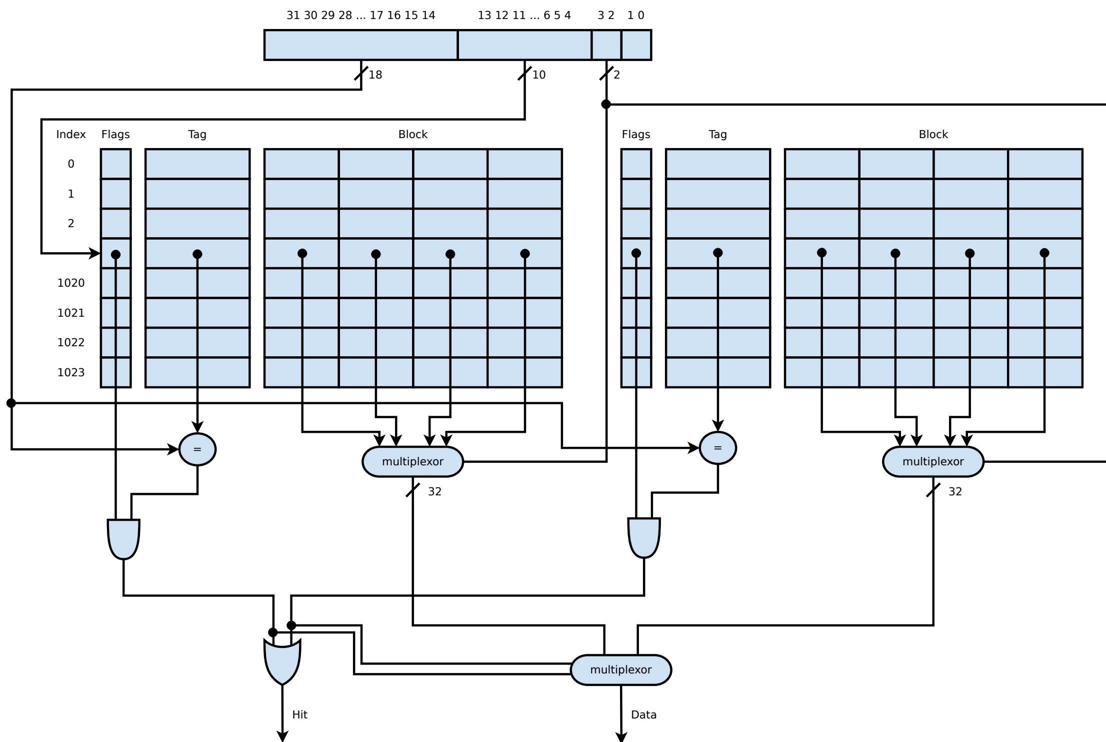


Figura 2.1: Diagrama esquemático de uma cache com associatividade 2, blocos de 16 bytes e 1024 blocos por via.

Quando se estuda as *caches* utiliza-se uma terminologia própria, descrita abaixo. Quando um acesso é feito, a informação solicitada pode estar ou não armazenada na

cache. Definem-se acertos (*hits*), quando a informação requerida está presente e erros (*misses*) quando não está presente [18]. Os acertos e erros podem ser de leitura ou escrita. A taxa de acerto representa a frequência com que os dados são encontrados na *cache*. Quando ocorre um erro de acesso, o tempo extra gasto para recuperar os dados em outros estágios da hierarquia de memória é denominado de penalidade de erro e é muito superior ao tempo de um acerto, sendo assim necessário minimizar a taxa de erros.

Quando uma *cache* possui associatividade 1, ela é chamada de diretamente mapeada (*direct mapped*). Neste caso, o endereço de um dado sempre é mapeado em apenas um local da *cache*. Quando uma *cache* possui tamanho de índice 1, ela é chamada de totalmente associativa (*full associative*). Neste caso, um determinado endereço pode ser mapeado em qualquer local da *cache*, pois a associatividade corresponde ao tamanho, em linhas, da *cache*, ou seja existem tantas vias quanto o número de linhas.

Em *caches* com associatividade maior que 1, sempre que for necessário armazenar dados num índice que está ocupado, deve-se decidir qual dos blocos presentes será descartado. A forma como o bloco é escolhido é definida pela política de substituição (*replacement policy*). O papel da política de substituição é retornar o número da via de associatividade do bloco que será descartado. A política de substituição pode ser desde a mais trivial, como a substituição aleatória, como complexa, por exemplo, guardando o histórico de uso dos blocos nos *flags* e substituindo o bloco que foi usado menos recentemente (*least recently used*, ou LRU).

Uma outra política é a forma como é realizada as operações de escrita. Toda vez que uma operação de escrita é feita na hierarquia de memória, esta escrita pode se propagar imediatamente pela hierarquia ou ser feita apenas na *cache* e neste caso ela fica marcada como pendente. O primeiro caso é chamado de *write-through* e o segundo, *write-back*. Para implementação do segundo caso utiliza-se um *flag* extra, o *dirty* bit, para indicar que um dado foi escrito na *cache*, porém não na memória. O dado é escrito na memória apenas se precisar ser descartado. Assim, é possível reduzir operações de escrita na memória mais lenta quando há diversas escritas em um mesmo bloco. Outra política, também associada à escrita, é a que define o que deve ser feito quando há uma operação de escrita em um dado e ele não se encontra na *cache*, ou seja há um erro de escrita. Se uma escrita for solicitada e o bloco não estiver na *cache*, há duas opções: a primeira, chamada de alocar na escrita (*write allocate*), é ler o bloco da memória, atualizar a *cache* e então escrever na *cache*. A segunda, chamada de não alocar na escrita (*no write allocate*), a operação de escrita pula a *cache* e escreve diretamente no nível posterior da hierarquia, sem trazer o bloco para a *cache* [9].

Com a grande diferença de velocidade entre o processador e a memória primária e uso de uma quantidade grande de dados pelos programas muitas vezes se faz necessário uma hierarquia de memória com várias *caches* consecutivas [19]. A utilização de dois níveis

de *cache*, por exemplo, visa aproveitar o rápido tempo de acerto da *cache* menor junto com o grande espaço da *cache* maior. A vantagem pode ser entendida considerando-se inicialmente apenas uma *cache* e avaliando o tempo médio de acesso aos dados com ela:

$$TempoAcessoMemoria = TempoAcerto + TaxaErro \times PenalidadeErro \quad (2.1)$$

onde $TempoAcessoMemoria$ é o tempo de acesso à memória, $TempoAcerto$ é o tempo gasto com os acertos, $TaxaErro$ é a taxa com que os erros ocorreram e $PenalidadeErro$ é o custo de se fazer o acesso à hierarquia inferior para ler o dado que não está presente na *cache*.

Normalmente o tempo de acerto é da ordem de 1 ou 2 ciclos de execução. Já a penalidade de erro pode ser da ordem de 150 a 200 ciclos [9]. Como exemplo de memória primária, atual, uma memória comercial DDR3-1866 possui um tempo de acesso variando entre 9 ciclos de memória, ou 13,5ns a 667 MHz, no melhor caso (linha previamente selecionada) a até 40,8ns (acesso completo, incluindo a troca de linha), representando uma latência de 80 ciclos em um processador de 2GHz [12]. A discrepância entre os tempos de acerto e erro dão margem à inclusão de uma *cache* adicional, que permite que a penalidade de erro, na equação acima, se torne menor, conforme Figura 2.2. Assim como na *cache* original, o termo predominante de atraso na segunda *cache* é o segundo, mas diferente da primeira *cache*, que precisa ter um tempo de acerto rigorosamente pequeno, a segunda *cache* pode focar na diminuição da taxa de erros, simplesmente tendo um tamanho enorme, sem muita preocupação com seu tempo de acerto [19].

Toda vez que há dados repetidos na hierarquia de memória, existe a possibilidade de haver um problema de coerência entre dois componentes da hierarquia. Quando pelo menos um nível da hierarquia envolvido no problema é de *cache* diz-se que se tem um problema de coerência de *cache* (*cache coherence*). Por exemplo, os dados de uma *cache* e os da memória principal podem se tornar incoerentes (isto é, referindo-se a valores escritos em momentos diferentes), caso um outro dispositivo ou processador escreva diretamente na memória. Uma hierarquia de memória é dita coerente se [9]:

1. Uma leitura feita por um processador P a um endereço X, após uma escrita feita por P no endereço X, sem nenhuma escrita feita por outros processadores no intervalo, sempre retorna o valor escrito por P.

2. Uma leitura feita por um processador P a um endereço X, após uma escrita feita por outro processador Q no endereço X, retorna o valor escrito por Q, desde que exista um intervalo de tempo entre a escrita e a leitura (definido pela arquitetura) e nenhum outro processador escreva durante este intervalo.

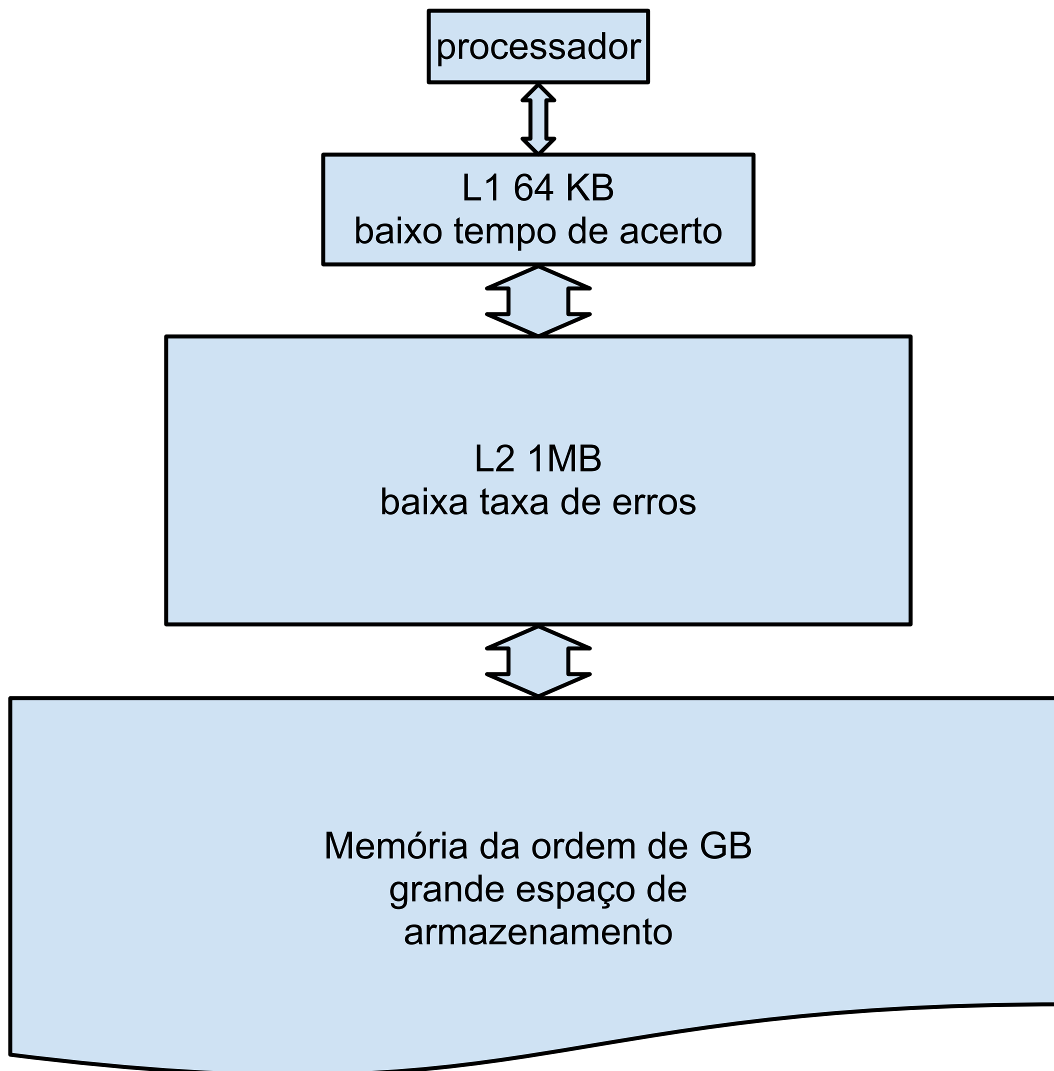


Figura 2.2: Hierarquia de memória *caches* com seus tamanhos relativos.

3. Escritas ao mesmo local são serializadas, isto é, duas escritas ao mesmo local por quaisquer dois processadores são vistas na mesma ordem por todos os processadores.

O problema de coerência de *caches* torna-se mais importante em sistemas multiprocessadores (*multi-cores*) onde cada processador (núcleo) possui sua *cache* individual. Os protocolos de coerência entre *caches* têm a função de gerenciar dados replicados entre *caches*. Há duas classes de protocolos, as baseadas em diretório, que usam um local separado, chamado diretório, para salvar o estado dos dados compartilhados, e os protocolos de *snooping*, em que as *caches* salvam o estado de compartilhamento junto com os blocos na *cache* e anunciam e observam no barramento para descobrir se outras *caches* estão compartilhando informações. A coerência entre *caches* é mais importante em sistemas com múltiplos processadores [9], já que o compartilhamento de dados é inevitável. Como o ArchC não possui um suporte formal a sistemas *multi-core* diretamente no simulador, optou-se, neste trabalho, apenas em deixar margem para implementações de protocolos de coerência, que futuramente poderão ser plugados nas *caches*.

2.2 Trabalhos relacionados

Este trabalho teve como objetivo a construção de um simulador de *cache* para ArchC em nível comportamental. Outros simuladores de *cache* existem, porém construídos para operar em outros níveis de modelagem. O simulador CACTI [23] é um modelador analítico de memória *cache*, que leva em conta as características eletrônicas dos componentes e a organização física típica da *cache* para construir modelos levando em consideração requisitos como: tempo de acesso, área, consumo de energia etc. O CACTI não é um simulador, é um gerador analítico de modelos de *cache*. O CACTI pode ser usado, de forma complementar, com este trabalho, por exemplo, usando o CACTI para escolher qual *cache* usar e usando o ArchC para simular o comportamento da *cache* escolhida. O simulador dinero [6] é um simulador de *cache* gerador de estatísticas. O simulador dinero é chamado de baseado em traço (*trace-driven*), pois ele recebe como entrada um *trace* de acesso à *cache* e retorna as estatísticas. Ele não constrói a memória de armazenamento da *cache*, ele contém o mínimo código necessário para gerar estatísticas de acertos e erros. O simulador dinero foi utilizado neste trabalho para comparar as estatísticas das *caches* implementadas com as estatísticas das *caches* obtidas com o dinero, garantindo a corretude da implementação. O dinero funciona de forma trivial, é executado com os parâmetros da *cache* e recebe um *trace* de acessos à *cache* (com leituras e escritas). A saída é o número de acessos totais à *cache* e o número de erros de acesso. Segue um exemplo de uso do dinero, com a entrada gerada por um *trace* de uma *cache* de dados de um simulador SPARC do ArchC:

```
$$ dineroIV -l1-dsize 2k -l1-dbsize 16 -l1-dassoc 2 -l1-drepl f
---Dinero IV \textit{cache} simulator, version 7
---Written by Jan Edler and Mark D. Hill
---Copyright (C) 1997 NEC Research Institute, Inc. and Mark D. Hill.
---All rights reserved.
---Copyright (C) 1985, 1989 Mark D. Hill.  All rights reserved.
---See -copyright option for details

---Summary of options (-help option gives usage information).

-l1-dsize 2048
-l1-dbsize 16
-l1-dsbsize 16
-l1-dassoc 2
-l1-drepl f
-l1-dfetch d
-l1-dwalloc a
-l1-dwback a
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat D
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
 r 4ffe00 4
w 4ffe00 4
 r 4ffe00 4
w 4ffe00 4
 r 4ffe00 4
w 4ffe00 4
 r 4ffe00 4
w 4ffe00 4
 r 4ffe04 4
w 4ffe04 4
 r 4ffe04 4
w 4ffe04 4
 r 4ffe04 4
 (...)
---Simulation complete.
l1-dcache
 Metrics                    Total   Instrn     Data    Read   Write    Misc
```

```
45  -----------              -------  -------  -------  -------  -------  -------
46  Demand Fetches      1109082        0  1109082   886766   222316        0
47   Fraction of total   1.0000   0.0000   1.0000   0.7995   0.2005   0.0000
48
49  Demand Misses         27498        0    27498    27457       41        0
50   Demand miss rate    0.0248   0.0000   0.0248   0.0310   0.0002   0.0000
51
52  Multi-block refs          0
53  Bytes From Memory    439968
54  ( / Demand Fetches) 0.3967
55  Bytes To Memory      276704
56  ( / Demand Writes)  1.2446
57  Total Bytes r/w Mem 716672
58  ( / Demand Fetches) 0.6462
59
60 ---Execution complete.
```

No exemplo, pode-se ver um pedaço recortado do arquivo de entrada, consistindo de linhas com três valores: uma indicação de operação, o endereço e o tamanho dos dados, em bytes. A linha de comando define os parâmetros da *cache*: 2KB, blocos de 16 bytes, associatividade 2 e política de substituição FIFO. Com a leitura do arquivo de entrada completa, o dinero retorna o resultado da simulação, neste caso totalizando mais de 1 milhão de acessos à *cache* de dados, sendo 79,95% destas, operações de leitura. Foram no total 27.457 erros de leitura indicando uma taxa de erros de leitura de 3,10% para esta simulação.

Diversos outros simuladores baseados em traço existem. Por exemplo, um apanhado sobre simuladores de *cache* [24] enumerou 50 simuladores e os classificou em função de diversas métricas e formas de operação, como a precisão do resultado, velocidade, adequação a medição de múltiplos processos, sistema operacional, etc. Na lista, se incluem simuladores de arquiteturas específicas, com suporte a coleta de *trace*, como o SPIM [15] para MIPS-1, MINT [25] para R3000 e Shade [4] para SPARC-V8, SPARC-V9 e MIPS. Em outros casos, os simuladores adicionam instruções ou anotações aos programas a serem simulados (TRAPEDS [22], MPtrace [7], AE [14] e outros). Implementações mais recentes visam simular as arquiteturas mais recentes, como *multi-cores*, como é o caso do simulador CASPER [10] e formas de acelerar a simulação, por exemplo utilizando GPUs [11] ou FPGAs [28].

Na parte de simuladores funcionais, há o simulador desenvolvido no próprio laboratório, utilizado em um simulador SPARC multi-core com *cache* coerente e transacional [13]. O código foi feito de forma bastante modular e estava prontamente disponível. O código original consiste de um núcleo de armazenamento, com a memória da *cache*, sem nenhum tipo de conexão externa. Encapsulando este núcleo, há uma implementação

da *cache* transacional. O código também utiliza a interface padronizada TLM (*Transaction Level Modelling*) do SystemC e, portanto, possui uma rigorosa coleção de encaixes e embrulhos, encapsulando a interface da *cache* em chamadas TLM. A principal função do protocolo TLM é a função transport() com parâmetros e estruturas bem definidas, como campo de operação (leitura/escrita), endereço, dados, resultados, etc. O protocolo TLM é um protocolo padronizado para cooperação de componentes, em SystemC, de diferentes organizações, mas ele não é necessariamente rápido. Outro simulador de *cache* relevante foi um construído para o próprio ArchC [26, 27]. Este foi um outro simulador funcional de protocolo TLM. Infelizmente, imprevistos foram descobertos após o término deste projeto, como problemas de desempenho e resultados incoerentes com as simulações do dinero, o que resultou na descontinuação desta *cache* do ArchC. Este trabalho tem como objetivo a reimplementação de uma *cache* funcional, no simulador funcional do ArchC, sem os problemas anteriores, focando em corretude, flexibilidade e desempenho.

2.3 ArchC

O ArchC é uma linguagem de descrição de arquiteturas, que permite descrever um processador e sua hierarquia de memória, com o objetivo de prover informação no nível de abstração chamado de projeto em nível de sistema (*system-level design*) [1, 3, 20]. O ArchC permite que, dada uma descrição de arquitetura, sejam geradas automaticamente ferramentas, como montadores, simuladores, editores de ligação e depuradores. As ferramentas permitem toda uma gama de estudos de arquitetura em ambiente controlado, sejam elas já existentes, sejam novas propostas.

No ArchC, um processador é descrito em duas partes, uma, chamada de descrição de recursos onde é descrita a estrutura do processador, contendo, por exemplo, a lista de registradores, o tamanho de palavra e as conexões para a hierarquia de memória. A outra parte está relacionada ao comportamento do processador e descreve o conjunto de instruções (*instrucion set architecture*). Esta descrição inclui tanto o formato das instruções, como as operações que cada instrução realiza no estado do processador e da memória.

Definida a linguagem, o projeto ArchC também provê diversas ferramentas que operam com as descrições. As relevantes para este trabalho são o pré-processador, que lê arquivos de descrição e constrói árvores sintáticas representando a descrição, a biblioteca de simuladores de memória, usadas para construir a hierarquia de memória e conectá-la ao processador e o simulador do processador, chamado acsim. O acsim é um simulador de arquiteturas descritas em nível funcional, isto é, sem informações detalhadas de temporização internas do processador ou da memória.

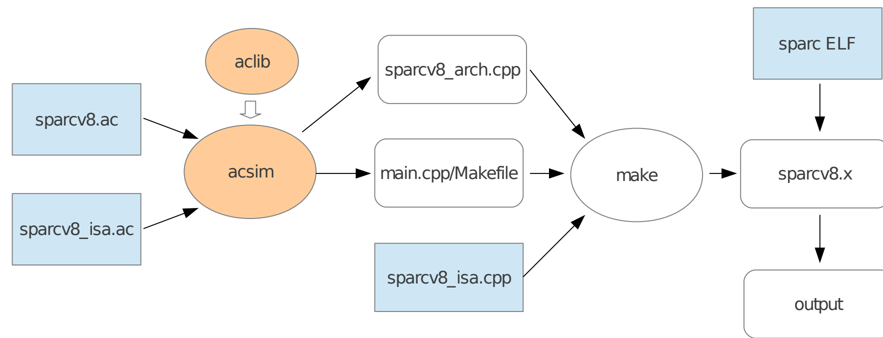


Figura 2.3: Diagrama com operação do simulador do ArchC. Na figura, os arquivos de entrada são: sparcv8.ac, sparcv8_isa.ac, sparcv8_isa.cpp. O programa acsim processa os arquivos de entrada, gerando os arquivos intermediários. Os arquivos resultantes podem ser compilados, gerando o simulador sparcv8.x, que pode receber programas de entrada. A biblioteca aclib é a parte do ArchC onde os componentes de armazenamento estão presentes.

Já existem diversos modelos de processadores modelados em ArchC, em variados estágios de funcionamento. Como exemplo, será discutido o simulador SPARCv8, o mesmo utilizado durante a elaboração deste trabalho. A descrição de recursos do SPARCv8 é mostrada a seguir:

Listing 2.1: Definição de arquitetura em ArchC

```
1  AC_ARCH(sparcv8){
2
3    ac_mem      DM:5M;
4    ac_regbank  RB:256;
5    ac_regbank  REGS:32;
6
7    ac_reg  npc;
8
9    ac_reg<1>  PSR_icc_n;
10   ac_reg<1>  PSR_icc_z;
11   ac_reg<1>  PSR_icc_v;
12   ac_reg<1>  PSR_icc_c;
13
14   ac_reg  PSR;
15   ac_reg  Y;
16
17   ac_reg<8>  WIM;
18   ac_reg<8>  CWP;
```

```
19
20    ac_wordsize 32;
21
22    ARCH_CTOR(sparcv8){
23
24      ac_isa("sparcv8_isa.ac");
25      set_endian("big");
26    };
27  };
```

Descrevendo individualmente cada linha, a primeira define o nome da arquitetura (sparcv8), depois há uma memória de 5MB, um banco de registradores RB com 256 elementos, outro, REGS, com 32 elementos, um registrador chamado npc, 4 registradores de 1 bit usados como *flags*, mais dois registradores de uma palavra, PSR e Y e dois de 8 bits, WIM e CWP. O tamanho da palavra é definido como 32 bits e na seção de construção, é definido o arquivo que contém a descrição do conjunto de instruções e o *endianess* como *big endian*.

A descrição do conjunto de instruções contém os formatos de instruções e a nomenclatura utilizada em códigos de montagem. Parte do código é da seguinte forma:

Listing 2.2: Trecho de código ArchC. A operação set asm define textualmente uma instrução e a operação set decoder define seu código de máquina

```
1     ldd_reg.set_asm("ldd [%reg + %reg], %reg", rs1, rs2, rd);
2     ldd_reg.set_asm("ldd [%reg], %reg", rs1, rd, rs2="%g0");
3     ldd_reg.set_decoder(op=0x03, op3=0x03, is=0x00);
4
5     stb_reg.set_asm("stb %reg, [%reg + %reg]", rd, rs1, rs2);
6     stb_reg.set_asm("stb %reg, [%reg]", rd, rs1, rs2="%g0");
7     stb_reg.set_asm("clrb [%reg + %reg]", rd="%g0", rs1, rs2);    //
         synthetic
8     stb_reg.set_asm("clrb [%reg]", rd="%g0", rs1, rs2="%g0");     //
         synthetic
9     stb_reg.set_decoder(op=0x03, op3=0x05, is=0x00);
```

A descrição define como cada instrução pode ser chamada (tipos de parâmetros e nome do comando) e seus códigos de máquina (*opcode*, argumentos, etc). A descrição do conjunto de instruções é complementada com o código do comportamento das instruções, escrito em SystemC. Por exemplo, a instrução xor com parâmetros nos registradores é descrita da seguinte forma:

Listing 2.3: Instrução XOR, com as chamadas dbg printf (de debug), writeReg (atualiza um registrador) e update pc (atualiza o registrador PC)

```
1 //!Instruction xor_reg behavior method.
2 void ac_behavior( xor_reg )
3 {
4   dbg_printf("xor_reg r%d,r%d,r%d\n", rs1, rs2, rd);
5   writeReg(rd, readReg(rs1) ^ readReg(rs2));
6   dbg_printf("Result = 0x%x\n", readReg(rd));
7   update_pc(0,0,0,0,0, ac_pc, npc);
8 };
```

No código de exemplo há o texto de depuração, a escrita do resultado no registrador de destino e a atualização do contador de programa com o novo endereço.

Como já citado, o ArchC permite a descrição de uma hierarquia de memória e houve algumas tentativas anteriores de implementação do suporte completo a *cache*, porém o suporte foi descontinuado. Este trabalho tem como objetivo substituir a implementação parcial e desatualizada de *cache* por uma implementação completa, funcional e eficiente. A definição da *cache* é feita no arquivo da descrição de recursos, com declarações de *cache* e conexões da hierarquia. Segue um exemplo de uma arquitetura SPARCv8 similar àquela já mostrada:

Listing 2.4: Trecho de código ArchC declarando uma memória, duas caches, definindo o arquivo com o conjunto de instruções, endianess e definindo as conexões entre as caches

```
1  ac_mem      DM:5M;
2  ac_icache   IC("dm", 256, 16, "wt");
3  ac_dcache   DC("2w", 128, 16, "wb", "random");
4  (...)
5  ARCH_CTOR(sparcv8){
6     ac_isa("sparcv8_isa.ac");
7     set_endian("big");
8     IC.bindsTo(DM);
9     DC.bindsTo(DM);
10 };
```

No exemplo, há uma *cache* de instruções, IC, e uma de dados, DC, conectadas à memória principal utilizando o comando “bindsTo”. A *cache* de instruções é mapeada diretamente, tem 4 KB (256 linhas vezes 16 bytes por linha) e política de substituição de escrita do tipo *write through*. A *cache* de dados tem duas vias de associatividade e o tamanho é de 2KB. O tamanho de índice da *cache* de dados é de 64 entradas e não 128, pois a sintaxe original da descrição define o segundo parâmetro como o tamanho do índice multiplicado pela associatividade e não foi alterada nesta implementação. A *cache*

de dados tem política de escrita *write back* e política de substituição aleatória (sempre que um local requisitado estiver lotado, escolhe aleatoriamente a via 0 ou 1 para substituir). Com a hierarquia declarada, é possível alterar o código de comportamento para ler e escrever das *caches*, ao invés da memória, obtendo o resultado esperado.

O modelo sparcv8 também tem suporte a emulação de chamadas de sistema, o que permite a simulação de programas genéricos com ele. Utiliza-se o acsim para gerar o código fonte específico do simulador e este é compilado, resultando num executável, `sparcv8.x`, que pode simular programas feitos para a arquitetura SPARC (com algumas modificações nos scripts do *toolchain*). Segue um exemplo de simulação:

```
$$ sparcv8.x --load=bin/rawdaudio < input/clinton.adpcm > /tmp/archc-test-B3EykY.out

             SystemC 2.2.0 --- Oct 26 2011 21:19:28
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED
ArchC: Reading ELF application file: bin/rawdaudio
ArchC: -------------------- Starting Simulation --------------------

Final valprev=8, index=4
ArchC: -------------------- Simulation Finished --------------------
SystemC: simulation stopped by user.
ArchC: Simulation statistics
    Times: 1.14 user, 0.00 system, 1.15 real
    Number of instructions executed: 6408683
    Simulation speed: 5621.65 K instr/s
cache: IC
Cache statistics:
Read:   miss: 253 (31.864%) hit: 541 (68.136%)
Write:  miss: 0 (0%) hit: 0 (0%)
Number of block evictions: 56
cache: DC
Cache statistics:
Read:   miss: 27457 (3.09631%) hit: 859309 (96.9037%)
Write:  miss: 41 (0.0184422%) hit: 222275 (99.9816%)
Number of block evictions: 27370
```

No exemplo, o simulador sparcv8.x carregou o programa bin/rawdaudio, com seus parâmetros e exibindo uma série de informações na tela. As informações finais são estatísticas das *caches*, pois este simulador foi construído com as novas *caches* L1. No exemplo, foram 6 milhões de instruções executadas e 1 milhão de acessos à *cache* de dados, ou seja, a cada 6 instruções, uma foi de acesso à memória.

O simulador também foi compilado com uma *decoder cache* adicional, por isto, os valores da *cache* de instrução estão reduzidos. A *decoder cache* não faz parte deste trabalho, porém, como está presente no ArchC teve que ser levada em consideração. A *decoder cache* serve para armazenar instruções pré-decodificadas, para acelerar a simulação. A *decoder cache* implementada no ArchC não precisou de modificações para este trabalho. A *decoder cache* é bastante simples, possuindo um *buffer* de tamanho fixo que representa um vetor sequencial de endereços. Sempre que um endereço de instrução nunca antes acessado é lido e está nos limites de espaço do vetor, esta instrução é decodificada e salva no vetor. Caso este endereço seja acessado posteriormente, a função de decodificação não é chamada e a instrução decodificada é simplesmente recuperada do vetor. Esta *cache* de decodificação é rápida, porém adultera consideravelmente os resultados da *cache* de instruções. Ela pode ser ativada ou desativada durante a construção do simulador, bastando para isto, executar o comando `acsim --no-dec-cache`. A decoder cache não representa um componente da arquitetura, é apenas um artifício de simulação, portanto não há nenhum uso para ela numa arquitetura real.

Capítulo 3

Modelos

Neste capítulo é descrita a construção do simulador de *cache* e sua inclusão no ArchC.

3.1 *Cache*

A *cache* utilizada foi construída baseando-se em uma implementação de *cache* modular já existente no laboratório. Originalmente ela foi desenvolvida para ser utilizada em um estudo sobre memória transacional que demandou a implementação de uma *cache* transacional para arquitetura SPARC [13]. O núcleo do código foi extraído do trabalho e sobre ele realizado uma bateria de testes que incluía casos de uso ainda não testados, como resultado foram encontrados alguns erros que foram resolvidos, além de passar por uma refatoração tornando a implementação menos dinâmica e mais estática. Uma nova interface foi implementada, resultando numa nova *cache stand-alone*. Após este processo, e com a garantia de funcionamento correto, a *cache* foi novamente modificada para ser integrada no ArchC, utilizando adaptadores e reimplementando partes antigas e desatualizadas para suportar novas características da *cache* revisada.

O núcleo do código consiste em uma *cache stand-alone*, isto é, uma *cache* contendo apenas sua memória de armazenamento, sem conexões ao processador ou ao restante da hierarquia. A interface da *cache* é a interface de uma implementação em nível comportamental (sem o detalhamento da temporização dos acessos). Os principais elementos da *cache* são:

A memória de armazenamento dos dados: A capacidade de armazenamento de da-

dos é dado pelo tamanho do índice, multiplicado pelo tamanho do bloco, multiplicado pela associatividade.

A memória de armazenamento das etiquetas: Precisa-se de uma etiqueta por bloco, então a quantidade delas é apenas o tamanho do índice multiplicado pela a associatividade.

A memória de armazenamento dos *flags*: A quantidade também é o tamanho do índice vezes a associatividade. A *cache* não define quais *flags* estão presentes. A estrutura de *flags* é definida externamente, junto com a implementação específica da *cache*.

A *cache* também utiliza um ponteiro corrente: A *cache* trabalha guardando em seu estado a posição de trabalho. Esta é a posição que se está querendo acessar ou preencher. O ponteiro indica qual etiqueta está sendo procurada, qual bloco de qual índice de qual via está sendo acessado, bem como o deslocamento no bloco.

Desta forma, a estrutura e a interface principal da *cache* são descritos conceitualmente da seguinte forma:

Listing 3.1: Pseudo código definindo a interface cache core. No código, o estado da cache é definido por cache data, cache tag, cache flags e current. O núcleo da cache deve implementar a interface definida pelo conjunto de funções.

```
1  cache_core {
2          state:
3          cache_data[associativity][index_size][block_size];
4          cache_tag[associativity][index_size];
5          cache_flags[associativity][index_size];
6
7          current {
8                  tag;
9                  associativity;
10                 block;
11                 index;
12                 offset;
13         };
14
15         interface:
16         bool get_block_for_read(address);
17         bool get_block_for_write(address);
18         get_available_block();
19         write_block_single(data);
```

```
20          data read_block_single();
21          write_block(block);
22          block read_block();
23  };
```

As operações de posicionamento do apontador são as três primeiras (linhas 16, 17 e 18 do código). As funções `get_block_for_read()` e `get_block_for_write()` extraem o índice, a etiqueta e o deslocamento sendo procurados e tentam encontrar a etiqueta armazenada em alguma das vias. Caso encontre, o dado está presente, então atualiza a associatividade corretamente e retorna verdadeiro. Caso não encontre, retorna falso. Ambas as funções fazem isto, a diferença é que uma atualiza as estatísticas de leitura e a outra, as estatísticas de escrita. A função `get_available_block()` é usada quando não se encontra os dados na *cache*. Ela faz uma requisição à política de substituição que retorna e atualiza o ponteiro com um valor de associatividade, com o local escolhido para ser substituído. As funções `read_block_single()` e `write_block_single()` lêem ou escrevem um único elemento dentro do bloco e são úteis para a interface com os níveis anteriores da hierarquia de memória, próximos ao processador. As operações `read_block()` e `write_block()` operam em um bloco inteiro e são úteis para a interface com a parte posterior da hierarquia, onde a *cache* opera solicitando e escrevendo blocos.

Além das funções principais da interface, há funções para operar com os *flags*, bem como chamadas ao objeto que controla as políticas de substituição e funções para se retornar as estatísticas da *cache*. Combinando todas as funções, é possível manter a *cache* modular ao mesmo tempo em que se permite construir diversos tipos de *cache* utilizando o mesmo núcleo. Por exemplo, é fácil construir uma *cache* do tipo *write through* com esta interface. Segue como exemplo, o código final da implementação da operação de leitura da *cache write through* no ArchC, escrita em C++:

Listing 3.2: Leitura em cache write through em C++

```
1   const cpu_word *read(address a, unsigned length) {
2           address b = byte_to_word(a);
3           if (!cache.get_block_for_read(b)) {
4                   cache.get_available_block();
5                   a = a/block_size*block_size;
6                   const cpu_word *d = memory.read(a, block_size);
7                   cache.write_block(d);
8                   cache.block_status().set_valid();
9           }
10          if (trace_active) cache_trace->add(trace_read, word_to_byte(b
                ), length);
11          return cache.read_block_single();
12  }
```

Excluindo-se a parte de arredondamento e gravação de *trace*, o código tem apenas 7 linhas. Descrevendo aqui, a primeira tarefa é usar `get_block_for_read()` para ver se o dado está presente na *cache*. Se estiver, simplesmente retorna-o. Se não estiver, encontra um local para colocá-lo com `get_available_block()`. Neste instante, o apontador da *cache* está posicionado no local certo de substituição. O novo bloco é lido da memória com `memory.read()`, é escrito na *cache* com `write_block()` e então o *flag* válido é ativado. Após o dado guardado na *cache*, ele é retornado. Esta implementação, em particular, não restringe que apenas o processador esteja acessando a *cache* e nem que a *cache* acesse apenas uma memória. É possível também que a *cache* esteja conectada a outras *caches*. Como se pode notar, a implementação retorna o endereço da primeira palavra, de forma que uma *cache* possa copiar um bloco grande. A memória é definida como um tipo genérico e, portanto pode também ser uma *cache* idêntica, bastando prover a operação `read()`, conforme declarada acima (isto também requereu que a memória fosse reimplementada para prover interface idêntica, mas a implementação é trivial).

Uma modificação nítida na implementação da nova *cache*, quando comparada com a *cache* original, foi a utilização de tipos genéricos. O núcleo da *cache* original foi projetado de forma que a construção de uma *cache* completa seria utilizando uma construção dinâmica (isto é, alocação e construção com parâmetros em tempo de execução) e com uma casca implementada por herança. No idioma de C++, todos os atributos eram do tipo *private*, o construtor recebia os parâmetros e alocava a memória da *cache*, salvando o endereço em apontadores. Por recomendação (levando em consideração a tentativa anterior, com problemas de desempenho), a implementação do núcleo foi modificada para se tornar mais estática. Ao invés de receber os parâmetros pelo construtor, foram utilizados tipos genéricos para a passagem de parâmetros. Além disto, a visibilidade da interface foi modificada para pública, de forma que, em vez de herança, a *cache* completa fosse construída por composição, sendo o núcleo um atributo da casca. Portanto, foram removidos completamente a herança e a construção em tempo de execução da *cache* original. Ao invés de alocar a memória no construtor, vetores com a memória e variáveis auxiliares foram declarados como atributos privativos da *cache*. Isto foi possível porque todos os parâmetros do vetor se tornaram conhecidos em tempo de compilação por meio de *templates*. Especificamente, o núcleo da *cache* foi declarado como o seguinte tipo genérico:

Listing 3.3: Protótipo da classe cache bhv em C++

```
1 template <
2         unsigned index_size,
3         unsigned block_size,
4         unsigned associativity,
```

```
5          typename cpu_word,
6          typename ADDRESS,
7          typename cache_status_t,
8          typename replacement_policy
9  >
10 class cache_bhv;
```

A quantidade de parâmetros do *template* ficou grande, pois estes parâmetros seriam normalmente declarados no construtor. Os parâmetros são as dimensões da *cache*, o tipo `cpu_word`, que representa o tamanho da palavra do processador, `ADDRESS`, que representa o tamanho do espaço de endereçamento, a classe `cache_status_t`, que representa os flags e a classe `replacement_policy`. Os *flags* aparecem porque o núcleo reserva espaço para eles e também necessita do método `is_invalid()`, usado na hora de encontrar blocos. Da política de substituição são usados três métodos: `block_to_replace()`, a principal função da política, substituir o bloco correto e `block_written()` e `block_read()`, ambos usados para informar à política para atualizar seu estado em função dos acessos à *cache*.

Os benefícios da implementação com tipos genéricos vão desde a maior simplicidade na alocação dos dados até benefícios funcionais. Em laços que são função das dimensões da *cache*, por exemplo, os tamanhos são conhecidos em tempo de compilação e podem ser otimizados.

Esta refatoração foi a maior realizado na *cache* original. As cascas das *caches* também precisaram de atualizações nos *templates*, para poderem repassar os parâmetros para os núcleos. A reescrita, no entanto, não modificou o comportamento da *cache*. Por outro lado, a modificação menor, porém com alteração comportamental importante foi nas funções `read_block_single()` e `write_block_single()`. No código original, estas funções escreviam exatamente uma palavra do tamanho da palavra do processador. Uma das características necessárias para esta implementação era que as *caches* pudessem ser ligadas em outras *caches*. Como cada *cache* tem um tamanho de bloco diferente, a função `read_block_single()` foi modificada para retornar um vetor de palavras e a `write_block_single()` para realizar um laço de escrita em múltiplas palavras. O número de palavras representa o tamanho do bloco do componente anterior na hierarquia de memória. No caso de uma *cache* L1 ligada ao processador, a *cache* funciona como antes, lendo e escrevendo uma única palavra. No caso de uma *cache* L2, a *cache* L1 envia, como parâmetro, a quantidade de palavras necessárias. Isto tornou estas funções mais parecidas com as funções `write_block()` e `read_block()`, utilizadas para acessar a parte posterior da hierarquia, mas ainda assim elas possuem uma diferença funcional: atualizações no estado da política de substituição ocorrem apenas do lado do processador.

As cascas da *cache* completam a implementação, definindo o comportamento e adicionando as conexões externas. A casca sempre inclui o núcleo de armazenamento como um

de seus atributos. A interface externa é a conhecida, possui portas de leitura e escrita e uma referência à memória (ou outra *cache* na hierarquia) e é descrita aqui, conceitualmente:

Listing 3.4: Pseudo código definindo uma cache. O estado da cache contém seu núcleo e uma referência para o nível seguinte na hierarquia de memória, aqui chamado de memory. A interface possui as funções read() e write().

```
1 cache {
2     state:
3     cache_core;
4     memory reference;
5
6     interface:
7     data read(address);
8     write(address, data);
9 };
```

Dois tipos de *cache* foram implementados: *write through* e *write back* (ambas com *write allocate*). A diferença está no momento em que a escrita à memória é feita, a primeira escreve imediatamente e a segunda, no momento que o bloco em questão é substituído. O estado da primeira é composto apenas pelo bit válido, enquanto o da segunda inclui o bit sujo, que indica escrita pendente.

Quatro tipos de política de substituição foram implementados: random, FIFO, PLRUM e LRU. A substituição em cada uma é feita do seguinte modo:

1. A política *random* retorna um valor de associatividade aleatório. O parâmetro na declaração da *cache* é "random".

2. A política FIFO associa um número para cada índice, todos inicialmente valendo zero. Sempre que necessitar de um novo bloco com um índice específico, a política faz um incremento modular, módulo associatividade, e retorna. Assim, o bloco a ser substituído é sempre o bloco que entrou antes na *cache* (mas não o que foi usado menos recentemente). O parâmetro na declaração da *cache* é "fifo".

3. A política LRU (least recently used) ordena o uso dos blocos e sempre remove o que foi usado menos recentemente. O parâmetro na declaração da *cache* é "lru".

4. A política PLRUM (*pseudo least recently used based on mru bits*) simula a política LRU e é descrita em [2]. Esta política marca quais associatividades foram usadas recentemente, mas se todas forem usadas, ele descarta todas, exceto a última. Ao receber a solicitação, retorna uma das que não foi marcada como usada. O parâmetro na declaração da *cache* é "plrum".

Para validar a implementação da *cache stand alone*, uma bateria de testes foi desenvolvida. O princípio que guiou a validação foi tentar construir um conjunto de testes que exercitasse a *cache* completamente, mas sendo do menor tamanho possível. Para isto, foi decidido testar algumas funções internas diretamente, aquelas que foram julgadas mais importantes (funções relacionadas ao desmembramento de endereços e funções de inspeção direta da estrutura interna da *cache*), bem como os testes mais usuais, tratando a *cache* como um componente unitário, gerando entradas e esperando saídas. Para uma boa cobertura de casos de teste, foram gerados pequenos casos de teste manualmente que tentaram cobrir acessos aleatórios na *cache* e acessos em regiões extremas, como o início e o fim da *cache* e nas bordas dos blocos para detectar *buffer overflows*. Casos de teste um pouco maiores foram gerados com padrões automáticos simples que cobriam toda a extensão da *cache*.

Testes unitários foram gerados para o núcleo da *cache*, para uma *cache* inteira e para uma hierarquia de *cache*. Os testes unitários foram feitos com uma inserção de acessos que simulava o processador com uma memória de testes, no lado posterior da hierarquia, que podia ser diretamente inspecionada. Sequências de leituras e escritas foram utilizadas. Após a sequência completar, uma nova sequência de leituras à *cache* foi realizada, bem como uma solicitação de estatísticas e uma inspeção do estado da memória externa. Os resultados, finalmente, foram comparados aos valores esperados para validação.

Por último, foi adicionado ao código suporte a geração de arquivo de *trace*, no formato do dinero, para validação das simulações maiores que seriam realizadas diretamente no ArchC. A interface contém apenas duas funções, o construtor, que recebe o arquivo de destino e a chamada `add()`, que recebe os 3 parâmetros da sintaxe do dinero: operação, endereço e tamanho. As cascas da *cache* também ganharam o `set_trace()`, de forma que o *trace* somente é gerado se ativado.

O desenvolvimento de testes foi uma pausa útil no desenvolvimento da *cache*, que permitiu a descoberta de 2 ou 3 erros menores (relacionados a impressão de mensagens de diagnóstico), permitiu a finalização da *cache* como um componente isolado, que pode ser inserido em qualquer outro projeto e permitiu, muito posteriormente no processo de integração com o ArchC, que a *cache* funcionasse sem erros assim que ela foi integrada ao simulador. A versão *stand-alone* da *cache* pode ser extraída do código fonte do ArchC e usada separadamente.

3.2 ArchC

Com a *cache* completa e funcionando corretamente, o trabalho se direcionou à integração da *cache* ao ArchC. A integração foi realizada em vários passos e foi necessário modi-

ficações em vários pontos do código ArchC. Os passos necessários foram os seguintes:

1. Alterar a notação para combinar com o estilo de código do ArchC
2. Eliminar o código da *cache* antiga
3. Copiar a implementação da *cache* nova para a estrutura de diretórios do ArchC
4. Atualizar os arquivos de projeto do ArchC para incluir os novos arquivos
5. Criar um embrulho da *cache* para o ArchC (para compatibilizar a passagem de parâmetros)
6. Corrigir a implementação das portas do processador para poder operar com memórias e *caches*
7. Modificar o *parser* para suportar as características da nova *cache*
8. Substituir o código que emite (recursivamente) as declarações de *cache* e memória
9. Substituição do código de execução da *cache* pelo novo código
10. Adicionar suporte a *traces* do dinero ativados pela linha de comando

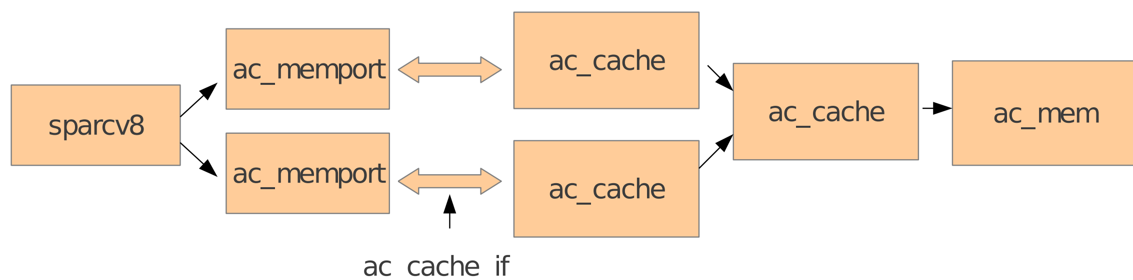


Figura 3.1: Diagrama com a representação dos módulos do ArchC. Na figura, representando um exemplo de arquitetura SPARC com duas caches L1, uma cache L2 e uma memória, sparcv8 é o processador, que possui dois ac_memports para se conectar na hierarquia de memória, cada um deles conectado a um ac_cache por meio de um adaptador chamado ac_cache_if. Ambas as caches se conectam diretamente a um ac_cache, que por sua vez se conecta a um ac_mem.

O passo 1 consistiu em renomear arquivos e classes, incluindo "ac_" no início, e corrigir colisões de nomes.

Os passos 2, 3 e 4 referem-se ao diretório `ac_storage`. Este diretório contém uma biblioteca de componentes de armazenamento. O principal componente de armazenamento do ArchC, quando este não usa uma hierarquia de memória é a classe `ac_storage`, que representa uma implementação funcional de memória. Complementando o `ac_storage`, há o `ac_memport`, que é usado como interface para o `ac_storage` e possui um determinado conjunto de funcionalidades adicionais esperadas tanto pelo simulador (como carga de programas e geração de logs), como pela arquitetura simulada (leitura e escrita de frações de palavras, conversão de *endianness*). Idealmente, o `ac_memport` deveria ser a implementação de uma porta entre o processador e todo o restante da hierarquia da memória, mas acabou acumulando algumas funções da memória principal também. As classes usadas com hierarquia, no entanto, são outras. O ArchC possui código condicional para inclusão de objetos quando a hierarquia é ativada no arquivo de descrição de arquitetura. Neste caso, há as classes `ac_cache` e `ac_mem`. Na implementação original, a primeira era uma *cache* completa com interface TLM e a segunda uma memória completa também com interface TLM. A implementação original não possuía portas entre o processador e a hierarquia, pois estava muito desatualizada. A princípio, foram feitas algumas tentativas de se aproveitar o isolamento do código com e sem hierarquia de memória, mas isto não foi possível, pois o código do ArchC depende consideravelmente do `ac_memport` e seria necessário espalhar modificações no ArchC para suportar uma interface alternativa ao `ac_memport`. Isto leva aos passos 5 e 6. A solução implementada foi criar um adaptador entre o `ac_cache` e o `ac_memport`, chamado `ac_cache_if`, cuja função é fazer a *cache* ter o mesmo formato de chamada do `ac_storage`. Ambos `ac_cache_if` e `ac_memport` fazem apenas algumas reordenações de parâmetros e não movem blocos grandes de memória e, portanto, estes dois adaptadores não interferem no desempenho da *cache*. Como garantia, apenas as *caches* L1 são conectadas às portas do processador, nenhuma outra *cache* tem associada uma porta.

Parte do código do `ac_memport` era incompatível com a *cache* e era relacionado à carga de programas na memória. Este código foi extraído e colocado em uma nova classe, `ac_program_loader`, que pode se conectar tanto num `ac_storage` (usado sem hierarquia de memória), como num `ac_mem` (usado com hierarquia de memória), este último também atualizado e compatível com a nova *cache*. Tanto a nova *cache*, como a nova memória possuem uma interface de leitura e escrita comum, baseada em *templates*, já descrita, sem implementar a interface TLM. Caso necessário, é possível implementar uma casca de *cache* TLM, aproveitando-se o núcleo, a interface com o processador e as políticas de substituição. Assim como o `ac_storage` é utilizado sem hierarquia de memória, o ArchC também possui o `ac_tlm` que permite a conexão com componentes externos. Esta parte da implementação não foi modificada.

Copiada a nova *cache* para o diretório `ac_storage`, atualizado o projeto e consertadas as interfaces, os próximos passos foram o 7 e o 8, atualizar o *parser* e consertar o código de emissão das declarações de *cache*. Estes códigos foram completamente reescritos para evitar qualquer incompatibilidade com as *caches* novas. As modificações tiveram 2 objetivos. O primeiro foi suportar a declaração de todas as *caches* implementadas no arquivo de descrição de arquitetura. São duas sintaxes, uma para *caches* mapeadas diretamente e outra para *caches* com mais de uma via. A diferença entre elas é apenas o último parâmetro, da política de substituição, inexistente na *cache* mapeada diretamente. Segue aqui um exemplo com ambas as sintaxes:

Listing 3.5: Sintaxes de declarações de cache do ArchC

```
1 ac_dcache L1("dm", 512, 32, "wt");
2 ac_dcache L2("8w", 4096, 32, "wb", "random");
```

A primeira declaração é de uma *cache* mapeada diretamente, com tamanho de índice 512, blocos de 32 bytes e escrita do tipo *write through*. A segunda é uma *cache* de 8 vias, com tamanho de índice 512 (4096/8), blocos de 32 bytes, escrita do tipo *write back* e substituição do tipo aleatória. A sintaxe original, com o tamanho do índice multiplicado pela associatividade, foi mantida.

A segunda parte foi fazer com que as declarações com *templates* fossem geradas. Para que os *templates* possam funcionar, a informação de toda a hierarquia posterior, onde as *caches* estão ligadas têm que ser repassadas recursivamente e estar presente em todas as declarações de *caches*. Utilizando o exemplo acima, se a *cache* L1 está conectada à *cache* L2, os parâmetros de *template* da *cache* L2 necessitam estar presentes na declaração da *cache* L1. A *cache* L2, por sua vez precisa dos parâmetros da memória. Supondo uma memória conectada nas caches descritas, por exemplo, são necessárias as seguintes definições recursivas (código simplificado apenas com os parâmetros recursivos):

Listing 3.6: Pseudo código representando as definições recursivas das caches

```
1 ac_mem MEM;
2 ac_cache<ac_mem> L2;
3 ac_cache<ac_cache<ac_mem> > L1;
```

Nos exemplos, as classes em que as *caches* estão ligadas são anotadas nos *templates* e desta forma, é possível que elas sejam definidas em tempo de compilação. O código para isto foi gerado associado ao código de *parse*, após a construção da árvore sintática, com a lista completa da hierarquia. O código também gera construtores que ligam as instâncias L1, L2 e MEM entre si. Cada *cache* contém uma referência ao próximo elemento da hierarquia, que é salva no momento da construção e permanece fixa durante toda execução.

O passo número 9 consistiu na substituição de código de chamadas de acesso à *cache* propriamente ditos. Todos os locais que haviam chamadas de acesso à *cache* foram modificados para utilizar a nova interface (na prática, chamadas de acesso ao `ac_memport`, já existente) e todas as chamadas de função para a *cache* antiga que pararam de fazer sentido na nova implementação foram removidos.

Por último, foi adicionado suporte a ativação do trace do dinero durante a execução da simulação. O formato do parâmetro é "–*trace-cache*=nome,arquivo", substituindo o nome da *cache* e o arquivo de destino do *trace*.

Estas modificações completaram a integração da *cache* ao ArchC. Sob o ponto de vista de usuário, a utilização da *cache* começa pela declaração da hierarquia no arquivo de descrição de arquitetura. No arquivo de descrição do comportamento do conjunto de instruções, acessos que iriam diretamente à memória são substituídos por leituras e escritas na *cache*. Com isto a *cache* entra em funcionamento e no final da execução, as estatísticas são retornadas. Opcionalmente, cada uma das *caches* pode ter um *trace* do dinero.

Como referência, segue de exemplo a API da cache *write through* no ArchC, já escrita em C++:

Listing 3.7: API da cache write through implementada no ArchC. O código mostra apenas os atributos e métodos do template e a implementação foi omitida. Todos os parâmetros que normalmente seriam passados no construtor foram convertidos em parâmetros do template. Além dos parâmetros da cache, há a definição do tamanho da palavra do processador (cpu word), o tipo de objeto representado pela memória da cache (backing store) e o tamanho do endereço de memória (address). É implicitamente necessário que a classe backing store defina as funções read e write com exatamente a mesma linha de definição das funções abaixo, permitindo que as caches se conectem entre si.

```
1  template <
2          unsigned index_size,
3          unsigned block_size,
4          unsigned associativity,
5          typename cpu_word,
6          typename backing_store,
7          typename replacement_policy,
8          typename address = unsigned
9  >
10 class ac_write_back_cache {
11         cache_bhv<index_size, block_size, associativity, cpu_word,
12                 address, write_back_state, replacement_policy> cache;
13         backing_store &memory;
14         ac_cache_trace *cache_trace;
15         bool trace_active;
16
```

```
17          address byte_to_word(address a) {}
18          address word_to_byte(address a) {}
19          ac_write_back_cache(const ac_write_back_cache &) {}
20
21          public:
22          ac_write_back_cache(backing_store &memory_) {}
23          ~ac_write_back_cache() {}
24          void set_trace(std::ostream &o) {}
25          const cpu_word *read(address a, unsigned length) {}
26          void write(address a, const cpu_word *d, unsigned length) {}
27          uint32_t get_size() {}
28          void get_statistics(cache_statistics *statistics) {}
29          void print(std::ostream &fsout) {}
30          void print_statistics(ostream &out) {}
31  };
```

Capítulo 4

Experimentos

Este capítulo descreve a validação da implementação da *cache* utilizando uma coleção de *benchmarks*, avaliando o desempenho e comparando o funcionamento da *cache* com o resultado esperado pelo simulador dinero. As hierarquias de *cache* testadas consistiram em duas *caches* L1, ligadas diretamente à memória, ou ligadas a uma *cache* L2, ligada à memória. As configurações de *cache* foram várias e estão descritas junto com os experimentos.

4.1 Benchmarks

As coleções de *benchmarks* utilizados nos testes são o MediaBench [16] e MiBench [8]. Estes benchmarks são constituídos por diversos programas de variados segmentos da indústria, como telecomunicações, escritório, segurança etc. Versões pré-compiladas destes *benchmarks* para as arquiteturas já descritas em ArchC estão disponíveis para *download* no site *www.archc.org*. A versão para SPARC foi utilizada para verificação do funcionamento das *caches*. O *benchmark* usado na verificação é composto por um total de 20 aplicativos, com variadas versões de executáveis e conjuntos de dados, totalizando 57 execuções. Um *script* com a lista das 57 execuções foi criado e cada uma das execuções foi nomeada com alguma informação significativa do teste, como o nome do executável e o tamanho dos dados de entrada. A lista completa de testes é mostrada nas tabelas 4.1 e 4.2.

Tabela 4.1: Lista de testes do MediaBench utilizados durante a verificação

benchmark	nome do teste
jpeg	cjpeg
jpeg	djpeg
gsm	toast
gsm	untoast
pegwit	decrypt
pegwit	create
pegwit	encrypt
mpeg2	mpeg2decode
mpeg2	mpeg2encode
adpcm	timing
adpcm	rawcaudio
adpcm	rawdaudio

4.2 Cluster

Para a execução dos milhares de casos de teste, foi utilizado o *cluster* do laboratório, junto com o programa Condor [17], que é um gerenciador de execução de programas em lote. O Condor tem como entrada um arquivo de configuração com uma lista de programas a serem executados e uma série de requisitos e configurações associados aos programas. Ao ser disparado, ele encontra as unidades de processamento no *cluster* capazes de atender aos requisitos e executa o máximo de programas possíveis simultaneamente, até completar toda a execução. No caso do *cluster* do laboratório, as unidades de processamento são os *cores* de processadores *multi-core* disponíveis no *cluster*, totalizando 96 unidades de processamento. No caso deste trabalho, o arquivo do Condor possui um pedido de execução para cada configuração de simulador e *benchmark* escolhida para teste.

A lista de máquinas do *cluster* é composta por 4 Intel Core 2 Quad @2,40GHz, 1 Intel Xeon X5650 @2,66GHz (com 24 unidades de processamento no total), 2 Intel Xeon E5645 @2,40GHz (com 24 unidades de processamento no total) e 1 Intel Xeon E5405 @2GHz (com 8 unidade de processamento no total). Todas as máquinas utilizam sistema operacional Ubuntu Server 10.04.3 LTS amd64.

Tabela 4.2: Lista de testes do MiBench utilizados durante a verificação

benchmark	nome do teste	benchmark	nome do teste
network/dijkstra	small	network/dijkstra	large
automotive/susan	corners-large	automotive/susan	edges-small
automotive/susan	corners-small	automotive/susan	smoothing-large
automotive/susan	smoothing-small	automotive/susan	edges-large
telecomm/gsm	toast-large	telecomm/gsm	untoast-small
telecomm/gsm	toast-small	telecomm/gsm	untoast-large
telecomm/adpcm	timing	telecomm/adpcm	rawcaudio-large
telecomm/adpcm	rawcaudio-small	telecomm/adpcm	rawdaudio-small
telecomm/adpcm	rawdaudio-large	consumer/lame	small
consumer/lame	large	consumer/jpeg	cjpeg-large
consumer/jpeg	djpeg-large	consumer/jpeg	cjpeg-small
consumer/jpeg	djpeg-small	security/sha	small
security/sha	large	telecomm/CRC32	small
telecomm/CRC32	large	telecomm/FFT	32768
telecomm/FFT	32768-inverse	telecomm/FFT	8192-inverse
telecomm/FFT	4096	automotive/bitcount	small
automotive/bitcount	large	network/patricia	small
network/patricia	large	automotive/qsort	small
automotive/qsort	large	automotive/basicmath	small
automotive/basicmath	large	office/stringsearch	small
office/stringsearch	large	security/rijndael	decrypt-small
security/rijndael	encrypt-small	security/rijndael	decrypt-large
security/rijndael	encrypt-large		

4.3 Casos de teste

Os casos de teste foram elaborados de forma que uma grande quantidade de configurações de *cache* fosse testada rapidamente. Para isto, foram realizados diferentes tipos de teste. O primeiro foi o mais abrangente sob o ponto de vista de configurações, utilizou apenas o programa rawdaudio do *benchmark* mediabench/adpcm. Foi construída uma hierarquia de memória com uma *cache* L1 de instruções, uma *cache* L1 de dados e uma *cache* L2 unificada. Para as *caches* L1, foram utilizadas todas as combinações de parâmetros presentes na tabela 4.3 (no caso de *caches* mapeadas diretamente, o parâmetro da política de substituição não existe e não foi utilizado). As *caches* de instrução e de dados foram geradas com o mesmo tamanho.

associatividade	1, 2, 4, 8
tamanho de índice	128, 256, 512, 1024, 2048, 4096
tamanho de bloco	4, 8, 16, 32, 64
política de escrita	*write through*, *write back*
política de substituição	FIFO, *random*

Tabela 4.3: Lista de configurações de cache testadas

Para as *caches* L2, foram utilizadas as combinações da tabela 4.3 apenas para associatividade, tamanho de índice e tamanho de bloco, com as seguintes restrições adicionais: o tamanho total em *bytes* sempre maior do que as *caches* L1 e o tamanho do bloco sempre maior ou igual ao da *cache* L1.

Aplicando as regras acima e enumerando todas as *caches* resultam em 17.344 configurações possíveis. Simuladores para todas estas configurações foram gerados no *cluster* e então o teste com o programa rawdaudio foi executado para cada um deles. A saída do programa foi comparada com a saída de referência e a execução correta, confirmada.

A segunda sequência de testes teve como objetivo executar todos os programas do *benchmark*. Para isto, foi escolhida aleatoriamente uma configuração da hierarquia de memória para cada programa. Alguns testes do *benchmark* não estavam funcionando com o simulador SPARC original, sem modificações e, portanto, foram excluídos do teste. São eles: mibench/telecomm/CRC32 (small, large), mibench/netowrk/patricia (small, large), mibench/automotive/bitcount (small, large) e mediabench/pegwit (encrypt). As execuções testadas, portanto, somaram 50. Os testes e as configurações escolhidas estão nas tabelas 4.4 e 4.5.

Da mesma forma, as saídas dos programas foram comparadas com as saídas de referência e as execuções corretas, confirmadas.

4.4 Desempenho

Os testes de desempenho foram realizados da seguinte forma: primeiro, os *benchmarks* foram ordenados pelo seu tempo de execução estimado. Os *benchmarks* excessivamente rápidos foram retirados, bem como os excessivamente lentos. Para cada *benchmark*, foram escolhidas 6 configurações de *cache* distintas, sendo uma representando uma *cache* L1 pequena e rápida, uma representado *caches* L1 e L2 grandes e lentas e mais 4 representando *caches* intermediárias. Um simulador sem *cache* também foi construído como referência. Cada uma das configurações foi executada 10 vezes, o tempo de execução foi

Tabela 4.4: Configurações de cache selecionadas para os benchmarks. Os valores representam, respectivamente, associatividade, tamanho de índice, tamanho de bloco em bytes, política de escrita (write through ou write back) e política de substituição (fifo ou random) – Programas do MediaBench

cache L1	cache L2	benchmark
(4, 1024, 32), (wt, fifo)	(8, 2048, 32), (wb, fifo)	jpeg cjpeg
(2, 2048, 16), (wt, fifo)	(1, 4096, 32), (wb, fifo)	jpeg djpeg
(2, 256, 4), (wt, fifo)	(4, 512, 4), (wb, fifo)	gsm toast
(1, 4096, 16), (wt, fifo)	(8, 1024, 32), (wb, fifo)	gsm untoast
(1, 4096, 32), (wb, fifo)	(2, 2048, 64), (wb, fifo)	pegwit decrypt
(2, 128, 16), (wt, random)	(8, 128, 16), (wb, fifo)	pegwit create
(2, 128, 4), (wb, fifo)	(2, 128, 8), (wb, fifo)	mpeg2 mpeg2decode
(1, 128, 4), (wb, fifo)	(2, 128, 64), (wb, fifo)	mpeg2 mpeg2encode
(8, 256, 64), (wb, fifo)	(8, 4096, 64), (wb, fifo)	adpcm timing
(2, 1024, 8), (wt, fifo)	(4, 4096, 8), (wb, fifo)	adpcm rawcaudio
(4, 256, 4), (wb, fifo)	(8, 4096, 16), (wb, fifo)	adpcm rawdaudio

extraído das estatísticas do ArchC e a média foi tomada. Foram 29 *benchmarks*, com 6 configurações cada, totalizando 174 configurações distintas. Como o *cluster* do laboratório é heterogêneo, foram disparadas apenas 29 tarefas independentes, garantindo que as 60 execuções de cada *benchmark* fossem executadas na mesma máquina.

As *caches* pequenas escolhidas foram sorteadas utilizando os seguintes parâmetros: 1 via, tamanho de índice entre 16 e 2048 blocos, tamanho de bloco entre 4 a 64 *bytes*, escrita *write through*, substituição FIFO e tamanho máximo 32KB.

As *caches* médias escolhidas foram sorteadas utilizando os seguintes parâmetros: *cache* L1 de 2 vias, tamanho de índice 256 ou 512, tamanho de bloco, 16, 32 e 64, escrita *write through* ou *write back* e substituição random; *cache* L2 de 8 vias, tamanho de índice 1024 a 4096, tamanho de bloco 16, 32 e 64, escrita *write through* ou *write back* e substituição random ou LRU, com o tamanho de bloco sempre maior ou igual ao da *cache* L1 e com o tamanho total sempre maior que o da *cache* L1.

As *caches* grandes escolhidas foram sorteadas utilizando os seguintes parâmetros: *caches* L1 e L2 de associatividade 4 ou 8, tamanho de bloco 32 ou 64, escrita *write through* e substituição pseudo-LRU. As *caches* L1 possuem tamanho de índice 512 ou 1024 e as caches L2, 2048 ou 4096. As *caches* L2 possuem tamanho de bloco igual ou maior que a *cache* L1 e tamanho total maior que a *cache* L1.

A lista de configurações utilizadas está nas tabelas 4.7, 4.8, 4.9, 4.10, 4.6.

Os tempos de execução de cada *benchmark* em cada configuração foram divididos pelo tempo de execução sem *cache* e, resultando num fator de desaceleração normalizado (os *benchmarks* com e sem *caches* foram executados sempre na mesma máquina no *cluster*).

Tabela 4.5: Configurações de cache selecionadas para os benchmarks. Os valores representam, respectivamente, associatividade, tamanho de índice, tamanho de bloco em bytes, política de escrita (write through ou write back) e política de substituição (fifo ou random) – Programas do MiBench

cache L1	cache L2	benchmark
(4, 512, 8), (wb, random)	(4, 1024, 8), (wb, fifo)	network/dijkstra small
(8, 2048, 4), (wb, random)	(8, 512, 64), (wb, fifo)	network/dijkstra large
(8, 128, 32), (wt, fifo)	(8, 4096, 64), (wb, fifo)	automotive/susan corners-large
(4, 512, 8), (wb, fifo)	(2, 4096, 32), (wb, fifo)	automotive/susan edges-small
(8, 256, 4), (wb, random)	(4, 4096, 32), (wb, fifo)	automotive/susan corners-small
(1, 256, 16), (wt, fifo)	(2, 4096, 32), (wb, fifo)	automotive/susan smoothing-large
(8, 128, 4), (wb, random)	(2, 4096, 16), (wb, fifo)	automotive/susan smoothing-small
(2, 1024, 16), (wt, fifo)	(4, 2048, 16), (wb, fifo)	automotive/susan edges-large
(8, 512, 32), (wb, fifo)	(4, 4096, 64), (wb, fifo)	telecomm/gsm toast-large
(1, 128, 8), (wb, fifo)	(4, 2048, 32), (wb, fifo)	telecomm/gsm untoast-small
(2, 256, 16), (wt, fifo)	(2, 512, 16), (wb, fifo)	telecomm/gsm toast-small
(4, 256, 8), (wb, random)	(4, 4096, 8), (wb, fifo)	telecomm/gsm untoast-large
(4, 256, 4), (wb, fifo)	(2, 4096, 4), (wb, fifo)	telecomm/adpcm timing
(4, 512, 32), (wt, fifo)	(8, 4096, 64), (wb, fifo)	telecomm/adpcm rawcaudio-large
(4, 4096, 4), (wb, fifo)	(4, 4096, 16), (wb, fifo)	telecomm/adpcm rawcaudio-small
(1, 512, 64), (wt, fifo)	(4, 512, 64), (wb, fifo)	telecomm/adpcm rawdaudio-small
(1, 1024, 16), (wt, fifo)	(8, 512, 32), (wb, fifo)	telecomm/adpcm rawdaudio-large
(2, 1024, 16), (wb, random)	(4, 256, 64), (wb, fifo)	consumer/lame small
(2, 256, 16), (wt, random)	(2, 512, 32), (wb, fifo)	consumer/lame large
(1, 2048, 8), (wt, fifo)	(2, 4096, 32), (wb, fifo)	consumer/jpeg cjpeg-large
(8, 1024, 16), (wt, fifo)	(8, 4096, 16), (wb, fifo)	consumer/jpeg djpeg-large
(1, 512, 4), (wt, fifo)	(1, 512, 8), (wb, fifo)	consumer/jpeg cjpeg-small
(4, 128, 4), (wt, random)	(1, 4096, 4), (wb, fifo)	consumer/jpeg djpeg-small
(1, 512, 4), (wt, fifo)	(8, 1024, 8), (wb, fifo)	security/sha small
(4, 1024, 16), (wt, fifo)	(8, 2048, 64), (wb, fifo)	security/sha large
(1, 512, 8), (wb, fifo)	(2, 1024, 8), (wb, fifo)	telecomm/FFT 32768
(1, 4096, 8), (wt, fifo)	(2, 512, 64), (wb, fifo)	telecomm/FFT 32768-inverse
(4, 128, 16), (wb, random)	(4, 256, 32), (wb, fifo)	telecomm/FFT 8192-inverse
(1, 4096, 4), (wt, fifo)	(2, 2048, 32), (wb, fifo)	telecomm/FFT 4096
(8, 512, 4), (wb, fifo)	(2, 4096, 64), (wb, fifo)	automotive/qsort small
(2, 128, 32), (wt, random)	(4, 1024, 32), (wb, fifo)	automotive/qsort large
(2, 128, 32), (wt, random)	(4, 2048, 32), (wb, fifo)	automotive/basicmath small
(8, 512, 4), (wb, fifo)	(8, 4096, 4), (wb, fifo)	automotive/basicmath large
(2, 512, 4), (wt, random)	(1, 1024, 8), (wb, fifo)	office/stringsearch small
(4, 256, 4), (wb, fifo)	(8, 2048, 8), (wb, fifo)	office/stringsearch large
(8, 128, 8), (wt, random)	(2, 512, 64), (wb, fifo)	security/rijndael decrypt-small
(8, 128, 8), (wb, random)	(4, 256, 16), (wb, fifo)	security/rijndael encrypt-small
(4, 512, 32), (wt, fifo)	(4, 4096, 64), (wb, fifo)	security/rijndael decrypt-large
(4, 128, 4), (wb, random)	(2, 2048, 16), (wb, fifo)	security/rijndael encrypt-large

Tabela 4.6: Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MediaBench (1)

Programa	L1	L2
gsm untoast	16KB 1 16B wt fifo	–
	16KB 2 32B wb random	512KB 8 64B wt random
	32KB 2 64B wt random	1024KB 8 64B wt random
	16KB 2 32B wt random	512KB 8 32B wt random
	32KB 2 32B wb lru	512KB 8 32B wt lru
	128KB 8 32B wt plrum	256KB 4 32B wt plrum
gsm toast	1KB 1 32B wt fifo	–
	8KB 2 16B wb random	128KB 8 16B wt random
	8KB 2 16B wb random	512KB 8 64B wt random
	32KB 2 32B wb random	2048KB 8 64B wt random
	32KB 2 32B wt lru	256KB 8 32B wt lru
	512KB 8 64B wt plrum	1024KB 4 64B wt plrum
adpcm timing	16KB 1 64B wt fifo	–
	32KB 2 64B wt random	512KB 8 64B wb random
	16KB 2 16B wt random	128KB 8 16B wb random
	8KB 2 16B wb random	256KB 8 16B wb random
	8KB 2 16B wt lru	512KB 8 32B wb lru
	256KB 8 32B wt plrum	512KB 8 32B wt plrum
mpeg2 mpeg2decode	256B 1 4B wt fifo	–
	16KB 2 16B wb random	128KB 8 16B wb random
	32KB 2 64B wt random	2048KB 8 64B wt random
	16KB 2 16B wb random	512KB 8 32B wb random
	32KB 2 64B wb lru	2048KB 8 64B wb lru
	64KB 4 32B wt plrum	512KB 4 64B wt plrum
mpeg2 mpeg2encode	4KB 1 16B wt fifo	–
	8KB 2 16B wb random	512KB 8 32B wb random
	32KB 2 64B wt random	512KB 8 64B wt random
	32KB 2 32B wb random	1024KB 8 32B wb random
	16KB 2 32B wb lru	2048KB 8 64B wt lru
	128KB 4 32B wt plrum	512KB 8 32B wt plrum

Tabela 4.7: Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MiBench (1)

Programa	L1	L2
security/rijndael decrypt-small	1KB 1 64B wt fifo	–
	8KB 2 16B wt random	128KB 8 16B wt random
	64KB 2 64B wb random	512KB 8 64B wb random
	32KB 2 32B wt random	1024KB 8 64B wb random
	8KB 2 16B wt lru	256KB 8 32B wt lru
	64KB 4 32B wt plrum	2048KB 8 64B wt plrum
security/rijndael encrypt-small	32KB 1 64B wt fifo	–
	8KB 2 16B wb random	1024KB 8 64B wt random
	32KB 2 32B wb random	512KB 8 64B wt random
	8KB 2 16B wt random	1024KB 8 64B wb random
	8KB 2 16B wb lru	512KB 8 16B wt lru
	128KB 4 32B wt plrum	512KB 4 64B wt plrum
automotive/susan corners-large	256B 1 16B wt fifo	–
	16KB 2 16B wt random	512KB 8 16B wt random
	8KB 2 16B wb random	128KB 8 16B wb random
	32KB 2 32B wb random	512KB 8 64B wb random
	8KB 2 16B wb lru	1024KB 8 32B wt lru
	256KB 8 32B wt plrum	2048KB 8 64B wt plrum
network/dijkstra small	1KB 1 16B wt fifo	–
	16KB 2 32B wt random	512KB 8 64B wb random
	8KB 2 16B wt random	2048KB 8 64B wt random
	32KB 2 64B wb random	2048KB 8 64B wt random
	8KB 2 16B wt lru	128KB 8 16B wb lru
	64KB 4 32B wt plrum	256KB 4 32B wt plrum
consumer/jpeg cjpeg-large	128B 1 8B wt fifo	–
	64KB 2 64B wb random	512KB 8 64B wt random
	16KB 2 16B wb random	256KB 8 32B wb random
	8KB 2 16B wt random	256KB 8 32B wb random
	16KB 2 32B wb lru	1024KB 8 32B wb lru
	512KB 8 64B wt plrum	1024KB 8 64B wt plrum
security/sha large	64B 1 4B wt fifo	–
	32KB 2 32B wb random	512KB 8 32B wt random
	16KB 2 16B wt random	512KB 8 32B wb random
	8KB 2 16B wb random	2048KB 8 64B wt random
	64KB 2 64B wb lru	2048KB 8 64B wb lru
	64KB 4 32B wt plrum	512KB 8 32B wt plrum

Tabela 4.8: Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MiBench (2)

Programa	L1	L2
automotive/susan edges-large	4KB 1 4B wt fifo 8KB 2 16B wb random 8KB 2 16B wb random 8KB 2 16B wb random 8KB 2 16B wb lru 256KB 8 64B wt plrum	– 512KB 8 64B wb random 256KB 8 32B wt random 256KB 8 32B wb random 256KB 8 16B wb lru 2048KB 8 64B wt plrum
network/dijkstra large	512B 1 8B wt fifo 16KB 2 32B wt random 8KB 2 16B wt random 8KB 2 16B wb random 8KB 2 16B wt lru 128KB 4 32B wt plrum	– 1024KB 8 64B wt random 1024KB 8 64B wt random 2048KB 8 64B wb random 512KB 8 64B wb lru 1024KB 8 64B wt plrum
automotive/susan smoothing-large	512B 1 16B wt fifo 16KB 2 16B wt random 32KB 2 32B wt random 16KB 2 32B wt random 8KB 2 16B wt lru 128KB 4 32B wt plrum	– 256KB 8 32B wt random 512KB 8 32B wt random 256KB 8 32B wt random 1024KB 8 64B wt lru 512KB 4 32B wt plrum
security/rijndael decrypt-large	32KB 1 32B wt fifo 16KB 2 16B wt random 16KB 2 16B wb random 64KB 2 64B wt random 16KB 2 32B wb lru 128KB 8 32B wt plrum	– 2048KB 8 64B wb random 512KB 8 64B wt random 512KB 8 64B wt random 1024KB 8 32B wt lru 512KB 4 32B wt plrum
security/rijndael encrypt-large	8KB 1 4B wt fifo 16KB 2 16B wb random 32KB 2 32B wb random 8KB 2 16B wt random 16KB 2 16B wt lru 256KB 8 32B wt plrum	– 512KB 8 16B wb random 512KB 8 32B wb random 256KB 8 32B wt random 512KB 8 32B wb lru 1024KB 8 32B wt plrum
telecomm/adpcm rawdaudio-large	4KB 1 32B wt fifo 16KB 2 16B wb random 64KB 2 64B wt random 16KB 2 16B wb random 16KB 2 16B wt lru 256KB 4 64B wt plrum	– 256KB 8 16B wt random 512KB 8 64B wb random 2048KB 8 64B wb random 512KB 8 16B wb lru 1024KB 8 64B wt plrum

Tabela 4.9: Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MiBench (3)

Programa	L1	L2
telecomm/gsm untoast-large	16KB 1 8B wt fifo	–
	8KB 2 16B wt random	256KB 8 16B wt random
	32KB 2 64B wb random	2048KB 8 64B wb random
	8KB 2 16B wb random	1024KB 8 32B wt random
	8KB 2 16B wb lru	256KB 8 16B wt lru
	128KB 4 64B wt plrum	1024KB 4 64B wt plrum
telecomm/adpcm rawcaudio-large	8KB 1 64B wt fifo	–
	32KB 2 64B wb random	1024KB 8 64B wb random
	64KB 2 64B wb random	2048KB 8 64B wb random
	64KB 2 64B wb random	1024KB 8 64B wb random
	32KB 2 32B wt lru	1024KB 8 32B wt lru
	64KB 4 32B wt plrum	1024KB 4 64B wt plrum
telecomm/FFT 4096	4KB 1 8B wt fifo	–
	8KB 2 16B wt random	512KB 8 32B wt random
	16KB 2 16B wt random	512KB 8 32B wt random
	16KB 2 16B wt random	1024KB 8 32B wb random
	32KB 2 32B wb lru	512KB 8 64B wb lru
	512KB 8 64B wt plrum	2048KB 8 64B wt plrum
automotive/qsort large	1KB 1 8B wt fifo	–
	32KB 2 32B wt random	1024KB 8 32B wt random
	16KB 2 32B wb random	2048KB 8 64B wt random
	16KB 2 16B wt random	512KB 8 64B wb random
	16KB 2 32B wb lru	1024KB 8 64B wt lru
	128KB 8 32B wt plrum	1024KB 8 32B wt plrum
telecomm/adpcm timing	2KB 1 4B wt fifo	–
	32KB 2 32B wt random	256KB 8 32B wb random
	8KB 2 16B wt random	512KB 8 16B wb random
	16KB 2 16B wt random	128KB 8 16B wt random
	8KB 2 16B wt lru	1024KB 8 64B wb lru
	256KB 8 32B wt plrum	1024KB 8 64B wt plrum
telecomm/gsm toast-large	2KB 1 32B wt fifo	–
	16KB 2 32B wb random	2048KB 8 64B wb random
	32KB 2 32B wb random	1024KB 8 64B wb random
	16KB 2 16B wt random	512KB 8 16B wb random
	16KB 2 16B wb lru	256KB 8 32B wb lru
	128KB 4 64B wt plrum	2048KB 8 64B wt plrum

Tabela 4.10: Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random, lru ou plrum) – MiBench (4)

Programa	L1	L2
automotive/basicmath small	8KB 1 8B wt fifo	–
	8KB 2 16B wb random	1024KB 8 64B wb random
	16KB 2 32B wt random	2048KB 8 64B wt random
	16KB 2 16B wb random	2048KB 8 64B wt random
	32KB 2 32B wt lru	256KB 8 32B wb lru
	256KB 4 64B wt plrum	512KB 4 64B wt plrum
telecomm/FFT 8192-inverse	512B 1 4B wt fifo	–
	32KB 2 32B wt random	256KB 8 32B wt random
	16KB 2 32B wb random	512KB 8 32B wt random
	16KB 2 32B wb random	512KB 8 64B wb random
	32KB 2 64B wt lru	1024KB 8 64B wt lru
	128KB 8 32B wt plrum	1024KB 4 64B wt plrum
consumer/lame small	4KB 1 64B wt fifo	–
	64KB 2 64B wt random	2048KB 8 64B wt random
	16KB 2 16B wb random	1024KB 8 64B wb random
	16KB 2 32B wt random	1024KB 8 32B wt random
	8KB 2 16B wb lru	1024KB 8 64B wb lru
	256KB 4 64B wt plrum	2048KB 8 64B wt plrum
telecomm/FFT 32768-inverse	32KB 1 16B wt fifo	–
	16KB 2 32B wb random	1024KB 8 64B wt random
	16KB 2 16B wt random	512KB 8 64B wt random
	8KB 2 16B wb random	256KB 8 16B wt random
	32KB 2 32B wt lru	512KB 8 32B wt lru
	64KB 4 32B wt plrum	512KB 4 32B wt plrum
telecomm/FFT 32768	8KB 1 32B wt fifo	–
	8KB 2 16B wb random	512KB 8 16B wb random
	16KB 2 16B wt random	1024KB 8 64B wt random
	8KB 2 16B wt random	128KB 8 16B wb random
	16KB 2 16B wb lru	1024KB 8 64B wt lru
	128KB 8 32B wt plrum	2048KB 8 64B wt plrum
automotive/basicmath large	256B 1 8B wt fifo	–
	16KB 2 32B wb random	256KB 8 32B wb random
	32KB 2 64B wb random	512KB 8 64B wt random
	16KB 2 32B wb random	512KB 8 32B wb random
	64KB 2 64B wb lru	1024KB 8 64B wb lru
	256KB 8 32B wt plrum	512KB 4 64B wt plrum

Os resultados estão representados nas figuras 4.1, 4.2 e 4.3. Nas figuras, o *benchmark* timing é de tempo fixo, portanto, seu resultado é sempre em torno de 1.

Os resultados indicam uma desaceleração variando entre 10% e 60% para os *benchmarks* e configurações escolhidos. Configurações com escrita *write through* foram mais lentas, quando comparadas a *write back*. Da mesma forma, configurações maiores, que não cabem na *cache* real da máquina de simulação, também foram mais lentas que *caches* menores. A média geométrica de todas as desacelerações, excluindo os benchmarks timing, que são de tempo fixo, foi de 19,9%.

Um teste idêntico foi feito com a *decoder cache* (recurso usado no ArchC para acelerar a simulação da execução de um programa em um processador) desativada. Todas as 145 configurações foram recompiladas sem a *decoder cache*. O mesmo foi feito com a configuração sem *cache*. Após a simulação, os resultados com *cache* foram divididos novamente pelos resultados sem *cache*. Os resultados estão nas figuras 4.4, 4.5 e 4.6.

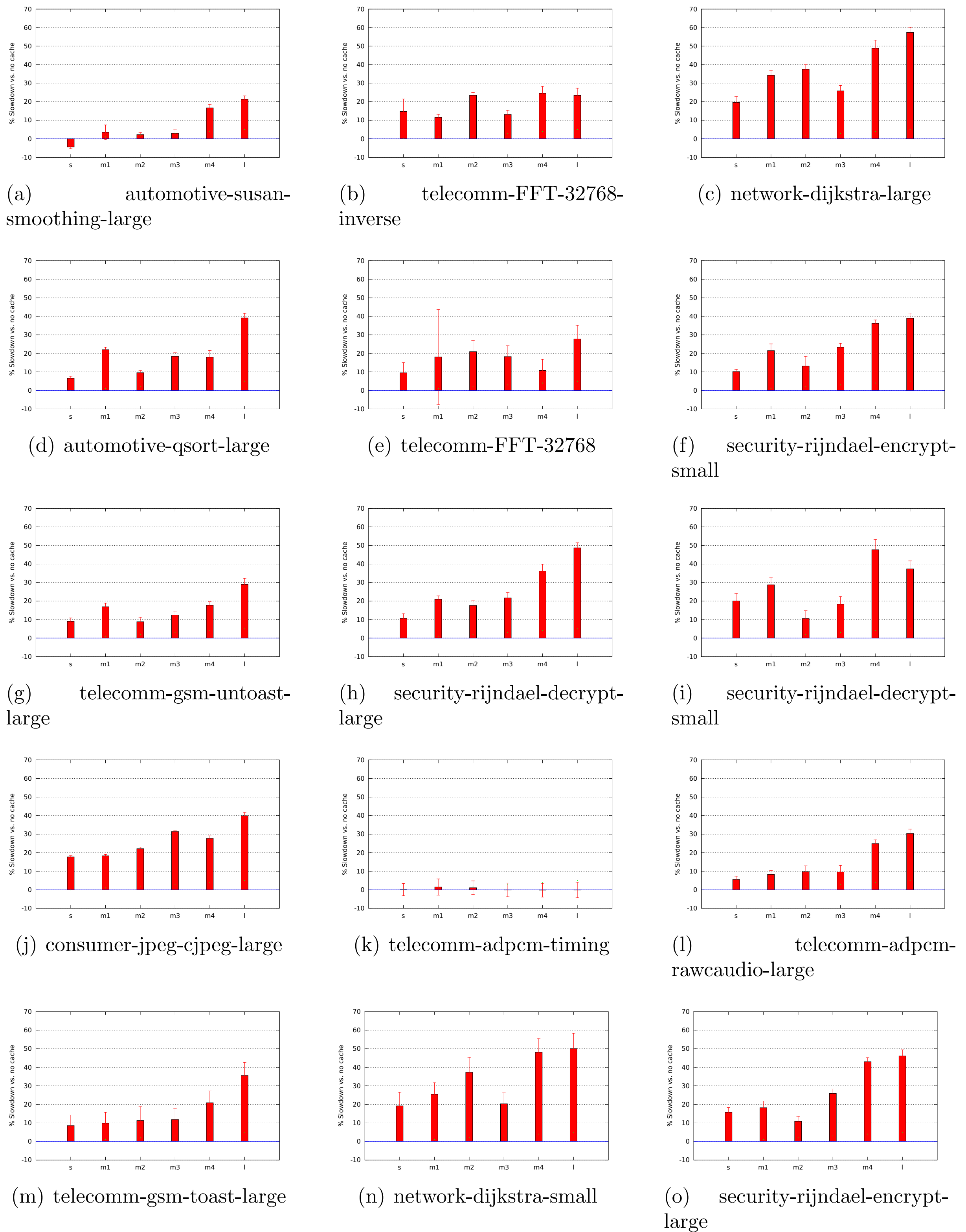
Os resultados indicam uma desaceleração de 8%, embora o motivo desta diminuição é que a desativação da *decoder cache* desacelera a simulação em uma ordem de grandeza e, portanto, o impacto no desempenho das *caches* se torna menor.

4.5 Validação com dinero

A validação com dinero foi realizada com os mesmos *benchmarks* escolhidos para o teste de desempenho, com a exceção da política de substituição, que foi fixa, sempre em FIFO. Neste teste, todas as *caches* L1 de dados e L2 tiveram seus *traces* ativados. Os arquivos de *trace* possuem tamanho da ordem de *gigabytes* e, portanto, ao invés de salvá-los em disco, foram utilizados arquivos do tipo FIFOs e as instâncias do dinero foram disparadas em *background*, evitando, assim, a desaceleração da simulação devido ao tempo de acesso ao disco. Todas as 145 execuções foram validadas pelo dinero.

4.6 Validação com arquiteturas MIPS e PowerPC

Como todas as simulações anteriores foram feitas exclusivamente utilizando o modelo SPARCv8 presente na distribuição do ArchC, o teste final foi um pequeno conjunto de execuções para os modelos MIPS1 e PowerPC, também parte do pacote de modelos do ArchC. Para estes, foram escolhidos 12 *benchmarks* ao acaso e 12 configurações distintas para a *cache*, uma para cada *benchmark*. As execuções escolhidas estão na tabela 4.11:



(a) automotive-susan-smoothing-large

(b) telecomm-FFT-32768-inverse

(c) network-dijkstra-large

(d) automotive-qsort-large

(e) telecomm-FFT-32768

(f) security-rijndael-encrypt-small

(g) telecomm-gsm-untoast-large

(h) security-rijndael-decrypt-large

(i) security-rijndael-decrypt-small

(j) consumer-jpeg-cjpeg-large

(k) telecomm-adpcm-timing

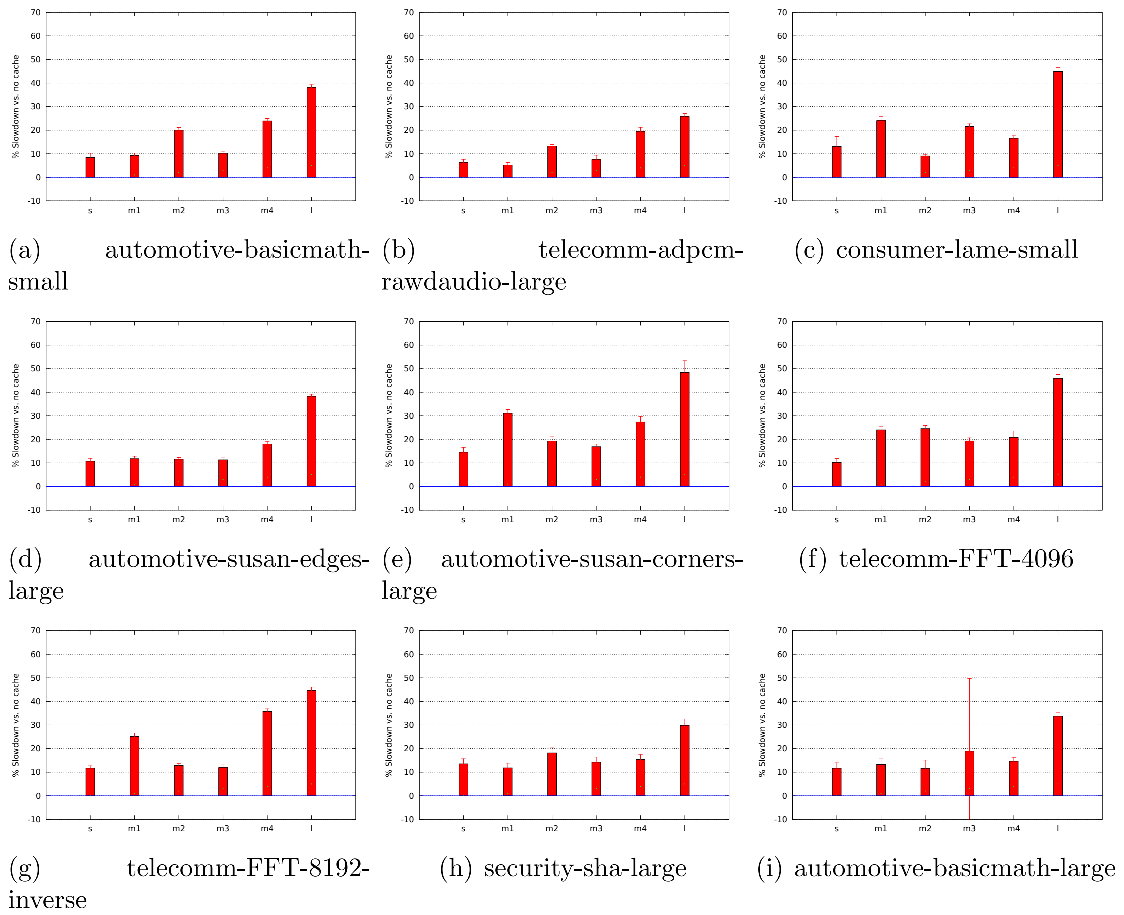
(l) telecomm-adpcm-rawcaudio-large

(m) telecomm-gsm-toast-large

(n) network-dijkstra-small

(o) security-rijndael-encrypt-large

Figura 4.1: Desaceleração devido a inclusão da cache – MiBench (1)

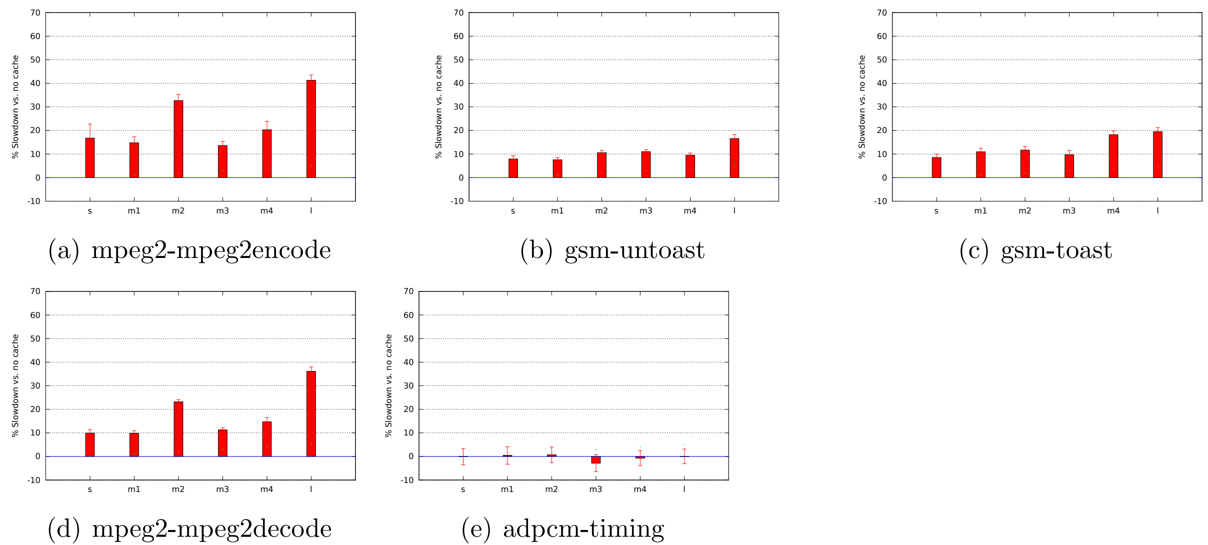


(a) automotive-basicmath-small (b) telecomm-adpcm-rawdaudio-large (c) consumer-lame-small

(d) automotive-susan-edges-large (e) automotive-susan-corners-large (f) telecomm-FFT-4096

(g) telecomm-FFT-8192-inverse (h) security-sha-large (i) automotive-basicmath-large

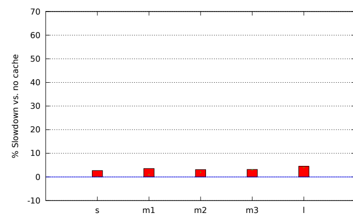
Figura 4.2: Desaceleração devido a inclusão da cache – MiBench (2)



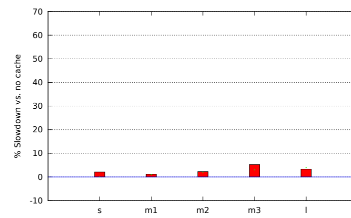
(a) mpeg2-mpeg2encode (b) gsm-untoast (c) gsm-toast

(d) mpeg2-mpeg2decode (e) adpcm-timing

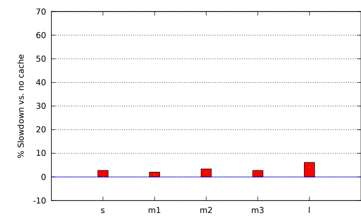
Figura 4.3: Desaceleração devido a inclusão da cache – MediaBench



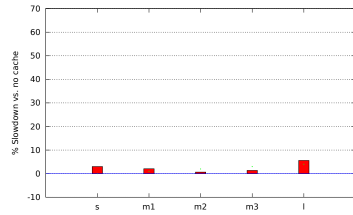
(a) automotive-susan-smoothing-large



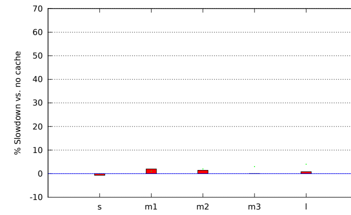
(b) telecomm-FFT-32768-inverse



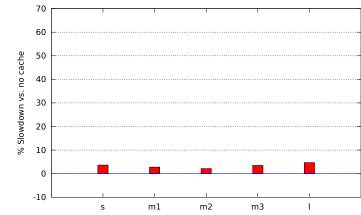
(c) network-dijkstra-large



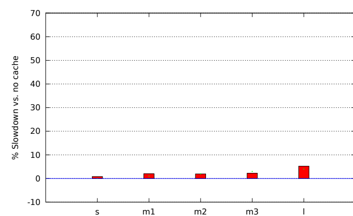
(d) automotive-qsort-large



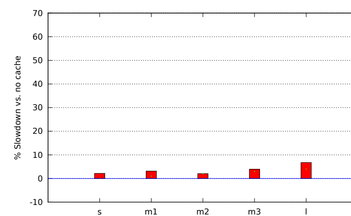
(e) telecomm-FFT-32768



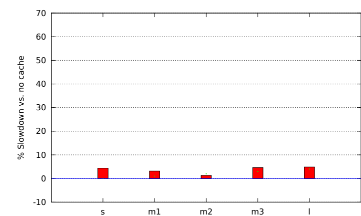
(f) security-rijndael-encrypt-small



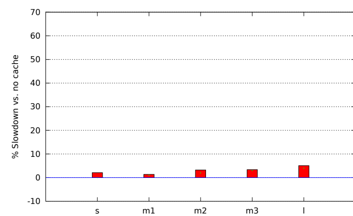
(g) telecomm-gsm-untoast-large



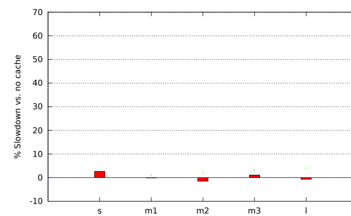
(h) security-rijndael-decrypt-large



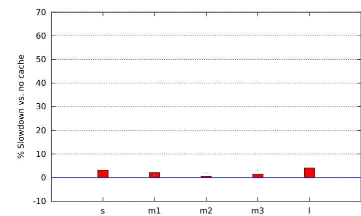
(i) security-rijndael-decrypt-small



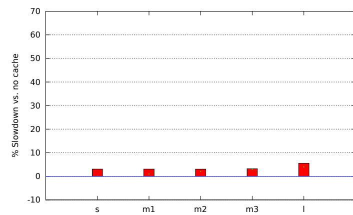
(j) consumer-jpeg-cjpeg-large



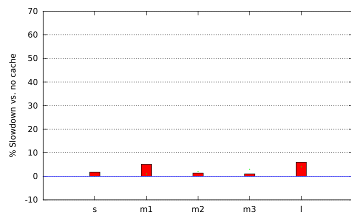
(k) telecomm-adpcm-timing



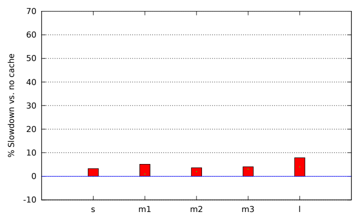
(l) telecomm-adpcm-rawcaudio-large



(m) telecomm-gsm-toast-large

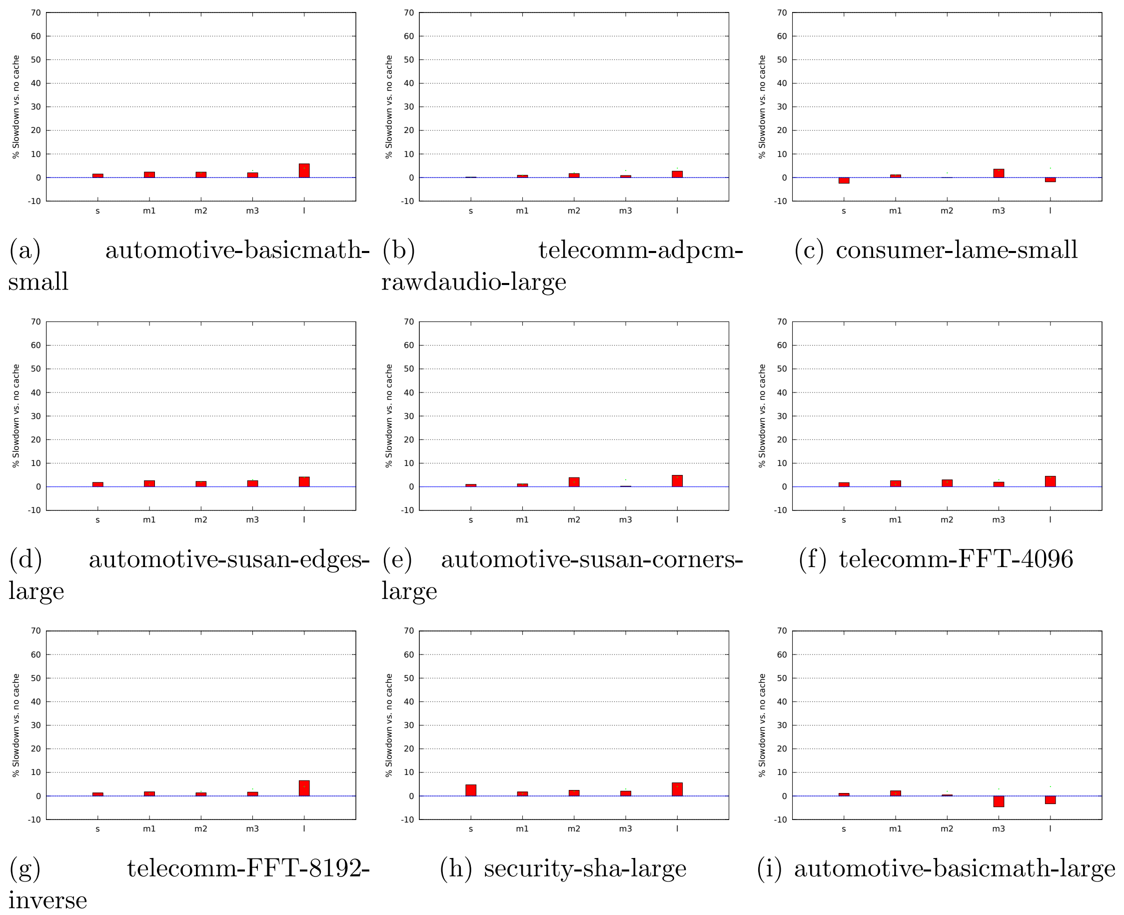


(n) network-dijkstra-small



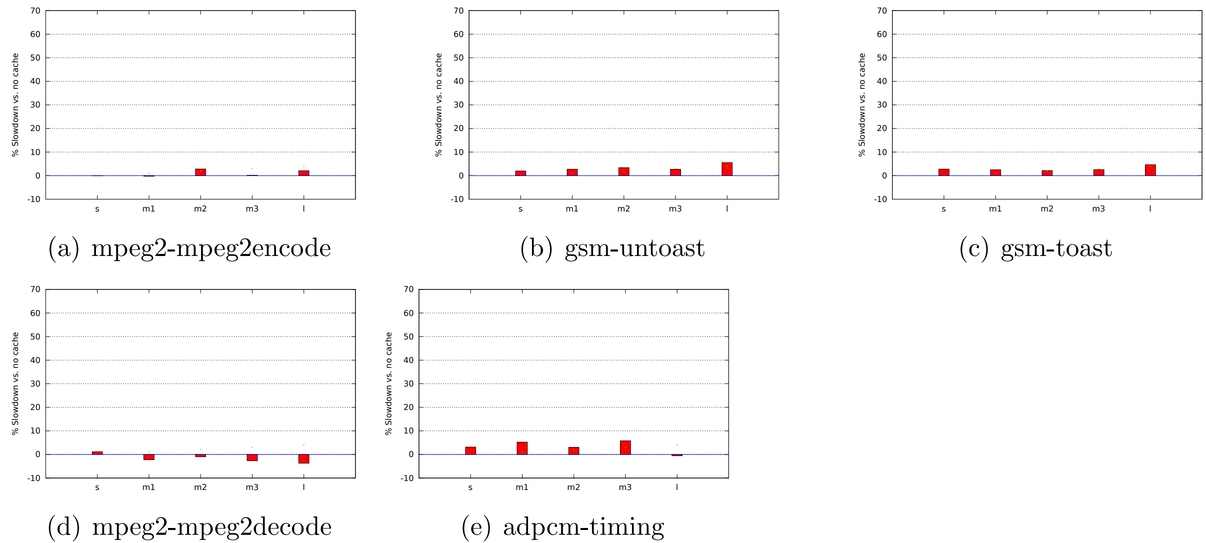
(o) security-rijndael-encrypt-large

Figura 4.4: Desaceleração devido a inclusão da cache – MiBench (1) – sem decoder cache



(a) automotive-basicmath-small
(b) telecomm-adpcm-rawdaudio-large
(c) consumer-lame-small
(d) automotive-susan-edges-large
(e) automotive-susan-corners-large
(f) telecomm-FFT-4096
(g) telecomm-FFT-8192-inverse
(h) security-sha-large
(i) automotive-basicmath-large

Figura 4.5: Desaceleração devido a inclusão da cache – Mibench (2) – sem decoder cache



(a) mpeg2-mpeg2encode
(b) gsm-untoast
(c) gsm-toast
(d) mpeg2-mpeg2decode
(e) adpcm-timing

Figura 4.6: Desaceleração devido a inclusão da cache – MediaBench – sem decoder cache

Tabela 4.11: Configurações de cache selecionadas para o teste de desempenho. Os valores representam, respectivamente, tamanho da cache em bytes, associatividade, tamanho do bloco em bytes, política de escrita (*write through* ou *write back*) e política de substituição (fifo, random ou plrum)

mibench/office/stringsearch small	L1 2KB 2 32B wb fifo
mibench/automotive/susan corners-small	L1 4KB 2 64B wb fifo
mediabench/jpeg djpeg	L1 4KB 2 32B wb fifo
mediabench/adpcm rawdaudio	L1 8KB 2 64B wb fifo
mediabench/adpcm rawcaudio	L1 8KB 2 32B wb fifo
mibench/automotive/susan edges-small	L1 16KB 2 64B wb fifo
mibench/office/stringsearch large	L1 4KB 4 32B wb fifo
mibench/consumer/jpeg djpeg-small	L1 8KB 4 64B wb fifo
mibench/telecomm/gsm untoast-small	L1 8KB 4 32B wb fifo
mibench/security/sha small	L1 16KB 4 64B wb fifo
mediabench/jpeg cjpeg	L1 16KB 4 32B wb fifo
mediabench/pegwit create	L1 32KB 4 64B wb fifo

Todas as execuções geraram a saída corretamente.

Capítulo 5

Conclusões

Neste trabalho, foi desenvolvida uma nova *cache* para o projeto ArchC. A *cache* é modular, configurável, rápida e foi validada sob diversas situações. A verificação de desempenho foi feita com uma coleção de *benchmarks* e configurações representando uma gama variada de *caches*, desde as menores e rápidas, até 2 níveis de *caches* grandes e lentas. A desaceleração da simulação variou entr 10% e 60%, em relação ao tempo de execução do simulador sem o uso da *cache* e um aumento no tempo de execução de 8%, quando comparados simuladores sem a *decoder cache*. Para a avaliação detalhada da *cache*, um *benchmark* rápido foi escolhido e um total de 17.344 configurações de hierarquias com 2 níveis de *cache* foram testadas, com *caches* variando de tamanho desde 512 bytes até 2MB. A *cache* foi testada detalhadamente com o modelo SPARC, presente na distribuição padrão do ArchC e um teste complementar foi realizado, também validando seu uso com os modelos MIPS e PowerPC.

Novas variantes de *cache* podem ser implementadas aproveitando os componentes já existentes. A execução foi verificada em uma gama variada de programas utilizando os *benchmarks* MediaBench e MiBench, com uma lista de configurações de *cache* escolhidas ao acaso. A *cache* tem suporte a geração de *traces* do dinero e com eles, seu comportamento foi validado, comparando as estatísticas retornadas pelo simulador com as retornadas pelo dinero.

5.1 Trabalhos futuros

O funcionamento da *cache* no ArchC dá margem a diversos novos trabalhos. Na própria *cache*, novas políticas de substituição podem ser implementadas, bem como *caches* do tipo *no write allocate*. Variantes de *cache* também podem ser implementadas, como combinações de *caches* L1 e L2 com propriedade de exclusão, isto é, *caches* onde só se encontra um bloco em uma das *caches*, mas nunca nas duas. Uma *cache* com saída TLM permitiria que a parte da hierarquia próxima do processador fosse estática e rápida, enquanto do lado da memória poderia haver uma conexão a um componente externo. Outras implementações seriam *caches* com coerência e transacionais, embora estas implementações seriam mais úteis com um suporte mais completo a multiprocessadores. Neste caminho, também seria interessante a avaliação de uma *cache* com temporização, embora este seria um trabalho de atualização completa do suporte a temporização no ArchC.

Componentes complementares às *caches* podem ser adicionados à coleção do ac_storage, como *write buffers* e *victim caches*. Uma possível evolução seria o suporte no ArchC à proteção de memória, com endereçamento virtual e TLBs.

O uso da *cache* permite a coleta de estatísticas para outras utilidades. Por exemplo, com o auxílio do CACTI, é possível estimar o consumo de energia para operações individuais de acertos e erros de leitura e escrita, o que permitiria uma estimativa do consumo de energia da *cache* durante uma simulação.

Referências Bibliográficas

[1] The archc architecture description language v2.0 reference manual, 2007.

[2] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[3] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The archc architecture description language and tools. *International Journal of Parallel Programming*, 33:453–484, 2005. 10.1007/s10766-005-7301-0.

[4] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '94, pages 128–137, New York, NY, USA, 1994. ACM.

[5] Peter J. Denning. On modeling program behavior. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, AFIPS '72 (Spring), pages 937–944, New York, NY, USA, 1972. ACM.

[6] J. Edler and M.D. Hill. Dinero iv trace-driven uniprocessor cache simulator, 1998.

[7] S. J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 37–47, New York, NY, USA, 1990. ACM.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite.

In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[10] Ravi Iyer. On modeling and analyzing cache hierarchies using casper. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:182, 2003.

[11] Xiaopeng Gao Junjie Ma, Han Wan and Xiang Long. Gpu-based time parallel cache simulator. In *Information Computing and Telecommunications (YC-ICT), 2010 IEEE*, pages 407–410. IEEE, 2010.

[12] kingston.com. Khx1866c9d3t1k2/8gx memory module specifications, 2011.

[13] Fernando Kronbauer, Alexandro Baldassin, Bruno Albertini, Paulo Centoducatte, Sandro Rigo, Guido Araujo, and Rodolfo Azevedo. A flexible platform framework for rapid transactional memory systems prototyping and evaluation. In *RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, pages 123–129, Washington, DC, USA, 2007. IEEE Computer Society.

[14] J. R. Larus. Abstract execution: a technique for efficiently tracing programs. *Softw. Pract. Exper.*, 20(12):1241–1258, November 1990.

[15] J.R. Larus. Spim s20: A mips r2000 simulator. madison 1993.

[16] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[17] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[18] David A. Patterson and John L. Hennessy. *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[19] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of performance-optimal multi-level cache hierarchies. *SIGARCH Comput. Archit. News*, 17(3):114–121, April 1989.

[20] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. Archc: a systemc-based architecture description language. In *16th Symposium on Computer Architecture and High Performance Computing, 2004 - SBAC-PAD 2004*, pages 66 – 73, oct 2004. Best Paper Award.

[21] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.

[22] C. B. Stunkel and W. K. Fuchs. Trapeds: producing traces for multicomputers via execution driven simulation. In *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '89, pages 70–78, New York, NY, USA, 1989. ACM.

[23] Shyamkumar Thoziyoor and Naveen Muralimanohar. Cacti 5.0, 2007.

[24] Richard Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. In *Performance Evaluation: Origins and Directions*, pages 97–139, London, UK, UK, 2000. Springer-Verlag.

[25] J.E. Veenstra and R.J. Fowler. Mint: a front end for efficient simulation of shared-memory multiprocessors. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994., MASCOTS'94., Proceedings of the Second International Workshop on*, pages 201–207. IEEE, 1994.

[26] P. Viana, E. Barros, S. Rigo, R. Azevedo, and G. Araujo. Exploring memory hierarchy with archc. In *15th Symposium on Computer Architecture and High Performance Computing, 2003*, pages 2 – 9, 2003.

[27] Pablo Viana, Edna Barros, Sandro Rigo, Rodolfo Azevedo, and Guido Araújo. Modeling and simulating memory hierarchies in a platform-based design methodology. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, DATE '04, pages 10734–, Washington, DC, USA, 2004. IEEE Computer Society.

[28] M. Watson and J. Flanagan. Simulating l3 caches in real time using hardware accelerated cache simulation (hacs): A case study with specint 2000. In *SBAC-PAD'02*, pages 108–116, 2002.

[29] M. V. Wilkes. Slave memories and dynamic storage allocation. *Electronic Computers, IEEE Transactions on Electronic Computers*, EC-14:270–271, 1965. 10.1109/PGEC.1965.264263.