UFSM

TRABALHO DE GRADUAÇÃO

EXPLORAÇÃO DE ALTERNATIVAS ARQUITETURAIS DE MEMÓRIA CACHE EM SISTEMAS EMBARCADOS BASEADOS EM JAVA

Aluno:

Mateus Beck Rutzig

Orientador:

Antonio Carlos Schneider Beck Filho

Prof . Coorientador

Antonio Marcos de Oliveira Candia

Santa Maria, RS, Brasil

2004

EXPLORAÇÃO DE ALTERNATIVAS ARQUITETURAIS DE MEMÓRIA CACHE EM SISTEMAS EMBARCADOS BASEADOS EM JAVA

Por

Mateus Beck Rutzig

Trabalho de Graduação apresentado ao Curso de Graduação em Ciência da Computação – Bacharelado, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de

Bacharel em Ciência da Computação

Curso de Ciência da Computação

Trabalho de Graduação n° 196
Santa Maria, RS, Brasil
2004

Universidade Federal de Santa Maria

Centro de Tecnologia

Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada, aprova o

Trabalho de Graduação

EXPLORAÇÃO DE ALTERNATIVAS ARQUITETURAIS DE MEMÓRIA CACHE EM SISTEMAS EMBARCADOS BASEADOS EM JAVA

elaborado por

Mateus Beck Rutzig

como requisito parcial para obtenção do grau de

Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA

____

Antonio Carlos Schneider Beck Filho
(Orientador)

____

Antonio Marcos de Oliveira Candia
(Prof. Coorientador)

____

Marcelo Pasin

____

Andréa S. Charão

Santa Maria, 12 de Dezembro de 2004

“Este trabalho é dedicado a toda a minha família”

Agradecimentos

Agradeço a todas as pessoas que estiveram comigo ao longo da minha vida, principalmente aos meus pais Valton e Sonia, e ao meu irmão Cristiano, que sempre estão no meu lado em qualquer momento. Também agradeço ao meu orientador Antonio Carlos Schneider Beck Filho, vulgo Caco, que esteve à disposição para resolver os problemas deste trabalho (não foram poucos!). Não esquecendo dos colegas e amigos que de alguma forma também contribuíram para a realização deste trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS..vii
LISTA DE FIGURAS..viii
LISTA DE TABELAS...x
RESUMO...xi
1 INTRODUÇÃO 1
2 REVISÃO BIBLIOGRÁFICA 6
2.1 Cache em Sistemas Embarcados baseados em Java............................6
2.2 Arquiteturas com cache 8
2.2.1 picoJava I..9
2.2.2 ARM10E..9
2.2.3 AMD Sempron..10
2.2.4 Femtojava...11
2.3 Características de memória cache..15
2.3.1 Organização da memória cache...16
2.3.2 Exploração da localidade espacial...17
2.3.3 Associatividade..20
2.3.4 Algoritmos de substituição...22
2.3.5 *Write-Back* e *Write-Through* 24
3 SISTEMA DE PARAMETRIZAÇÃO..27
3.1 Funcionamento do simulador...28
3.2 Descrevendo a estrutura do simulador..30
4 RESULTADOS DE SIMULAÇÃO...34
4.1 Programas para *Benchmark*..34
4.2 Metodologia...36
4.3 Resultados de desempenho da memória cache.....................................37
4.4 Resultados de área da memória cache..46
4.5 Resultados de potência da memória cache...51
4.6 Soluções..57
5 CONCLUSÕES E TRABALHOS FUTUROS...59
5.1 Trabalhos Futuros...60
5.1.1 Femtojava..60
5.1.2 Expansões do simulador...61
REFERÊNCIAS...63
ANEXO I- TABELA DE RESULTADOS...65
ANEXO II- SIMULADOR NA LINGUAGEM C...73

LISTA DE ABREVIATURAS E SIGLAS

CACO-PS	Cycle-Accurate COnfigurable Power Simulator
GCC	Gnu Compiler Collection
IMDCT	Inverse Modified Discrete Cosine Transform
JIT	Just In Time
JVM	Java Virtual Machine
LFU	Least Frequently Used
LRU	Least Recently Used
PDA	Personal Digital Assistent
SIMD	Single-Instruction, Multiple-Data
VLIW	Very Long Instruction Word

LISTA DE FIGURAS

Figura 2.1: Potência consumida, em porcentagem, pelas unidades do processador ARM920T. [ARM 2004]....8
Figura 2.2: Die do processador AMD Sempron [Tom 2004]....11
Figura 2.3: Os cinco estágios do *pipeline* do Femtojava *Low-Power*....13
Figura 2.4: Pirâmide de hierarquia de memória....15
Figura 2.5: Exemplo de um endereço de memória realizando um acesso a uma memória cache....16
Figura 2.6: Exemplo de uma organização de memória cache explorando o uso da localidade espacial. [PATTERSON 2003]....19
Figura 2.7: Exemplos dos tipos de associatividade em uma memória cache....21
Figura 2.8: Exemplo do método de *Write-Back* para uma informação modificada....24
Figura 2.9: Exemplo do método de *Write-Through* para uma informação modificada...25
Figura 3.1: Entradas e Saídas do simulador proposto....28
Figura 3.2: Arquivo que descreve um trace de acessos à memória....30
Figura 3.3: Estrutura que descreve uma cache em linguagem C....30
Figura 3.4: Demonstração dos campos da estrutura na memória cache....31
Figura 3.5: Duas memórias cache com tamanho 64 e com diferentes localidade espacial....32
Figura 4.1 Ciclos perdidos por cache miss em cada algoritmo para diferentes localidades espaciais, na cache de dados....38
Figura 4.2: Ciclos perdidos por cache miss em cada algoritmo para diferentes tamanhos de cache, na cache de dados....39
Figura 4.3: Ciclos perdidos por cache miss em cada algoritmo para diferentes associatividades usando o algoritmo LRU, na cache de dados....40
Figura 4.4: Ciclos perdidos por cache miss em cada algoritmo para diferentes associatividades usando o algoritmo LFU, na cache de dados....41
Figura 4.5: Ciclos perdidos por cache miss em cada algoritmo para diferentes localidades espaciais, na cache de instruções....42
Figura 4.6: Ciclos perdidos por cache miss em cada algoritmo para diferentes tamanhos de cache, na cache de instruções....43
Figura 4.7: Ciclos perdidos por cache miss em cada algoritmo para diferentes associatividades usando o algoritmo de LRU, na cache de instruções....45

Figura 4.8: Ciclos perdidos por cache miss em cada algoritmo para diferentes associatividades usando o algoritmo de LFU, na cache de instruções............................46
Figura 4.9: Área ocupada pela cache de dados, em células lógicas, parametrizando a localidade espacial...47
Figura 4.10: Área ocupada pela cache de dados, em células lógicas, parametrizando a associatividade..48
Figura 4.11: Área ocupada pela cache de instruções, em células lógicas, parametrizando a localidade espacial...50
Figura 4.12: Área ocupada pela cache de instruções, em células lógicas, parametrizando a associatividade..51
Figura 4.13: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, para diferentes localidades espaciais na cache de instruções........................52
Figura 4.14: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, parametrizando o tamanho da cache de dados..53
Figura 4.15: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, parametrizando a associatividade da cache de dados...................................54
Figura 4.16: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, para diferentes localidades espaciais na cache de instruções........................55
Figura 4.17: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, parametrizando o tamanho da cache de instruções.......................................56
Figura 4.18: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, parametrizando a associatividade da cache de instruções............................57

LISTA DE TABELAS

Tabela 1.1: Resultados de simulações em diferentes métodos de execução de Java, tomado como base o método Interpretado [O`Connor 1997]..3

Tabela 2.1: Lista da freqüência de instruções em processadores que interpretam bytecodes. [O'Connor 1997]...6

Tabela 2.2: Características de área e potência do processador ARM10E [ARM 2003]..10

Tabela 2.3: Instruções suportadas pelo processador Femtojava [Ito 2001]....................12

Tabela 4.1: Número de ciclos gastos, na arquitetura Femtojava *Low-Power*, com os algoritmos utilizados na simulação..36

Tabela 4.2: Resultado dos melhores desempenhos para cada algoritmo.......................58

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Centro de Tecnologia
Universidade Federal de Santa Maria

EXPLORAÇÃO DE ALTERNATIVAS ARQUITETURAIS DE MEMÓRIA CACHE EM SISTEMAS EMBARCADOS BASEADOS EM JAVA

Aluno:
Mateus Beck Rutzig

Orientador:
Antonio Carlos Schneider Beck Filho

Prof . Coorientador

Antonio Marcos de Oliveira Candia

Os Sistemas embarcados estão ficando cada vez mais complexos, disponibilizando serviços mais sofisticados como: acesso à Internet, tela colorida, entre outros. Assim, estes dispositivos necessitam de uma capacidade computacional maior.

Entretanto, a maioria dos sistemas embarcados é voltada ao mercado de dispositivos móveis, ou seja, são dispositivos dependentes de bateria e possuem como uma de suas principais características seu tamanho reduzido.

Assim, na inserção de um componente em uma arquitetura voltada à sistemas embarcados existe a preocupação com a potência consumida e com a área ocupada pelo mesmo.

Este trabalho apresenta a simulação da inserção de uma memória cache na arquitetura Femtojava [Ito 2001], sendo esta uma arquitetura que executa código nativo Java e foi criada especificamente para sistemas embarcados. Para isto foi implementado um simulador que fornece a possibilidade de parametrização de algumas características de memória cache de modo fácil e rápido. Por fim, são expostos e analisados os resultados destas simulações, sempre com a preocupação no aumento do desempenho, levando em conta também o consumo de potência e a área consumida em todas as simulações de memória cache.

CAPÍTULO 1

INTRODUÇÃO

Atualmente, o crescimento da produção de eletroeletrônicos faz com que a tecnologia se encontre cada vez mais presente em nosso cotidiano, facilitando as tarefas do dia-a-dia. Exemplos destes são celulares, PDA, tocadores de mp3, sistemas de alarmes e mecanismos de injeção eletrônica de combustíveis [Schelett 1998]. Estes dispositivos são denominados de sistemas embarcados, que cada vez mais disponibilizam serviços mais sofisticados como: acesso à Internet; tela colorida com gráficos animados; rádio; televisão; jogos, entre outros. Conseqüentemente, estas características fazem com que estes dispositivos necessitem de uma capacidade computacional ainda maior.

Alguns problemas são encontrados quando este assunto é tratado em sistemas embarcados. Existem duas particularidades nestes dispositivos que necessitam de uma atenção especial: os sistemas embarcados, na sua grande maioria, são produzidos com o objetivo de diminuir de tamanho; e também, em sistemas embarcados portáteis, por estes serem geralmente dependentes de uma bateria, existe a necessidade dos mesmos permanecerem ligados o máximo de tempo possível sem necessitar de recarga. Assim, há a preocupação adicional pelo consumo de energia.

Entretanto, quando a capacidade computacional do sistema aumenta a tendência é que ocorra também um crescimento da área do dispositivo e ao mesmo tempo um consumo maior de energia.

Conclui-se então que o objetivo, em sistemas embarcados, não é focado somente em obter a máxima capacidade computacional, e sim obter o desempenho suficiente para atender aos requisitos da aplicação, economizando-se em potência.

Com o crescimento da quantidade e a diversidade de dispositivos embarcados no mercado, a *Sun Microsystems* implementou uma tecnologia que portasse Java para sistemas embarcados. Assim Java tornou-se bastante atrativa para ser usada nestes sistemas, por obter características que facilitam a implementação de aplicativos para os mesmos. Dentre estas características está a sua portabilidade, assim, os sistemas implementados em Java podem ser simulados e testados em quaisquer plataformas para após ser implantado na arquitetura final. Outras características de Java são a robustez, segurança e a orientação a objetos. Por esta última característica, a modelagem e programação destes sistemas são facilitadas, visto que estes podem ser projetados em um nível mais alto de abstração, diminuindo o tempo de projeto.

Além do mais, Java foi desenvolvida para ser transmitida pela Internet. Conseqüentemente, programas em Java ocupam um pequeno espaço na memória de instruções, característica essencial para aplicativos que são executados em sistemas embarcados, já que os mesmos possuem restrições de memória.

Por todas estas facilidades, Java está cada vez mais sendo utilizada em sistemas embarcados. É previsto que no mínimo 80% dos telefones celulares em 2006 irão suportar Java [Lawton 2002].

Existem várias maneiras de se executar os *bytecodes* (linguagem de máquina) Java. Uma delas é de modo interpretado, com o uso de uma JVM

(*Java Virtual Machine*). O grande problema deste método é que ele torna a execução muito lenta, pelo princípio que Java é uma linguagem interpretada, já que foi criada para ser portável.

Então, foram criadas alternativas para a execução mais veloz de Java. Uma delas é fazer o uso de um compilador JIT (*Just-in-time*) [Krall 1998], o que faz com que o principal objetivo de Java, a portabilidade, seja perdido. Outra forma de executar *bytecodes* Java é diretamente em *hardware*, provendo uma execução com maior desempenho e mantendo a portabilidade do código, já que executa *bytecodes*, como mostra a Tabela 1.1.

Tabela 1.1: Resultados de simulações em diferentes métodos de execução de Java, tomado como base o método Interpretado [O`Connor 1997]

Método	Sistema	Benchmarks	
		Javac	Raytracer
Nativo	picoJava-I	15.2	19.6
JIT	Pentium	2.9	3.9
	486	2.6	2.3
Interpretado	Pentium	1.3	1.5
	486	1.0	1.0

Na Tabela 1.1 [O`Connor 1997] é demonstrada uma comparação de desempenho entre os três métodos de execução de Java, adotando como base para a comparação o método interpretado em uma arquitetura 486. Com o Javac, compilador padrão da *Sun* que gera *bytecodes* Java, houve um aumento de 260% de desempenho do método interpretado para o compilado JIT. Passando para a execução nativa (diretamente em *hardware*), ocorreu um aumento de desempenho de mais de 15 vezes, quando comparado com o método interpretado sendo executado em uma arquitetura 486, indicando que a execução Java em *hardware* é extremamente vantajosa em relação aos demais métodos de execução.

Quando se executa nativamente Java é alcançado o objetivo de desempenho, mantendo a portabilidade, como citado anteriormente. Este

método de execução beneficia os sistemas de tempo real, executados em sistemas embarcados, que devem responder a eventos produzidos por outros dispositivos com os quais interage e, para muitos deles, um atraso nessa resposta pode ser considerado uma falha no sistema.

A família Femtojava [Ito 2001] é um exemplo de arquiteturas que foram criadas especificamente para sistemas embarcados, visando restrições de área e potência. Atualmente, esta família possui diferentes versões, cada uma visando diferentes níveis de desempenho e consumo de potência, fornecendo ao projetista a oportunidade de escolher a que melhor se encaixa na sua aplicação. A versão Femtojava *Low Power* [Beck 2003] é dotada de um pipeline de 5 estágios. Está disponível também, em uma versão VLIW (*Very Long Instruction Word*) [Beck 2003b], além da versão Multiciclo [Ito 2001].

Como já citado anteriormente, os dispositivos embarcados estão cada vez mais sofisticados, necessitando de um desempenho maior. Assim os pesquisadores estão direcionando seus esforços para este gênero de arquitetura, sempre preocupados com o impacto do desempenho no aumento de consumo de potência e área ocupada pela arquitetura.

Atualmente, já existem diversos trabalhos que realizam estudos sobre o desempenho e a potência consumida em memórias cache em sistemas embarcados [Zhang 2003, Nicolau 2003]. Entretanto, não existe nenhum focado no comportamento de caches em processadores de pilha que executam nativamente Java.

Este modelo específico de processador possui características, que serão explicitadas no Capítulo 2, extremamente favoráveis para a inserção de uma memória cache na arquitetura, tais como: dados que fornecem uma grande freqüência de instruções de acesso à memória e economia de potência quando da presença de uma memória cache na arquitetura. Neste mesmo capítulo será feita uma breve revisão bibliográfica sobre

arquiteturas que são dotadas de cache, apresentando algumas características das mesmas.

No Capítulo 2 também serão apresentados alguns conceitos sobre memória cache, tais como: localidade espacial, associatividade e algoritmos de substituição de dados da cache, que são de extrema importância pra o entendimento do restante do trabalho.

Todavia, para inserção de qualquer unidade em uma arquitetura é necessária a realização de simulações na mesma, para que se possam analisar os resultados e chegar na melhor configuração possível, para posteriormente partir à prototipação da mesma. Além do mais, é importante ressaltar que para melhor explorar o espaço de projeto é necessário obter dados em um alto nível de abstração. Por estes motivos, foi implementado um simulador de memória cache, que será explicitado no Capítulo 3.

Este simulador produz resultados possibilitando a análise dos mesmos e, conseqüentemente, a localização das soluções mais viáveis para cada um dos diversos tipos de algoritmos executados na arquitetura. No Capítulo 3 será demonstrada também, a metodologia que foi usada para implementá-lo e a descrição da sua estrutura principal.

No Capítulo 4 é apresentado o conjunto de *benchmarks* usado na execução do simulador, mostrando as características de cada um. Também é explicitada a metodologia utilizada na obtenção dos resultados gerados pelo simulador com os respectivos *benchmarks*. Finalmente, são analisados os resultados obtidos em termos de desempenho, potência e área.

O último capítulo deste trabalho apresenta sugestões de trabalhos futuros, como o estudo do impacto da cache em outras versões do Femtojava e a possibilidade de implementação de uma configuração de cache, obtida como resultado deste trabalho, em VHDL para o Femtojava *Low Power*.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

2.1 Cache em Sistemas Embarcados baseados em Java

Nesta seção será demonstrado, inicialmente, o potencial da implementação de uma cache em arquiteturas que executam nativamente Java.

A Tabela 2.1 apresenta uma lista de freqüências de instruções em processadores que interpretam *bytecodes* Java.

Tabela 2.1: Lista da freqüência de instruções em processadores que interpretam bytecodes. [O'CONNOR 1997]

Classes de Instruções	Frequência dinâmica (%)
Loads (Variáveis locais)	34.5
Loads (Memória)	20.2
Cálculo (Inteiro/ Ponto-flutuante)	9.2
Branches	7.9
Calls/ Returns	7.3
Stores (Variáveis locais)	7.0
Push constante	6.8
Stores (Memória)	4.0
Variadas Operações na Pilha	2.1
Todas as outras instruções	0.6
Novos Objetos	0.4

Os processadores que interpretam bytecodes Java são processadores de pilha, ou seja, todas as operações com dados ocorrem através de

instruções “Empilha” e “Desempilha”. Isto ocorre pelo fato da JVM (*Java virtual Machine*) ser baseada neste modo de computação.

Como pode ser observado na Tabela 2.1, há um grande potencial para o estudo de memória cache em processadores de pilha, já que a segunda classe de instruções mais freqüente em um *trace* (arquivo de log de acessos à memória principal de um certo algoritmo) em um processador, que executa nativamente Java, são instruções de busca de dados na memória principal (*loads*). Esta classe de instruções mostra uma freqüência de 20,2% do total de instruções. Ou seja, provavelmente, estes dados irão se encontrar na memória cache, fornecendo uma resposta mais rápida ao processador.

Entretanto, como já observado anteriormente, em sistemas embarcados o enfoque não é somente em desempenho, mas sim em desempenho levando em conta o consumo de energia, por razões já citadas, e área ocupada. Por este motivo, é necessário balancear alguns parâmetros na implementação de uma organização de memória cache. Parâmetros como tamanho, associatividade, exploração da localidade espacial, que serão conceituados neste capítulo, são relevantes no equilíbrio entre desempenho, área e consumo de energia.

O fator principal para economia de energia quando uma memória cache é utilizada é a diferença de como esta e a memória principal são alimentadas. Enquanto a memória principal, que é uma memória dinâmica, necessita em certos intervalos de tempo ser realimentada para que possa manter os dados armazenados, a memória cache é estática, não necessitando desta realimentação para manter seus dados armazenados. Além disso, geralmente a memória principal utiliza um barramento externo ao processador, ao contrário da memória cache, aumentado ainda mais o seu consumo de potência. Conseqüentemente, a inserção de uma memória cache em uma arquitetura diminuirá o consumo de energia total do sistema.

Como ilustra a Figura 2.1, a cache é uma grande consumidora de potência dentro do processador. Todavia, equilibrando as suas características pode-se economizar, em média, 60% da sua potência consumida, sem contabilizar o consumo da memória principal[Zhang 2003].

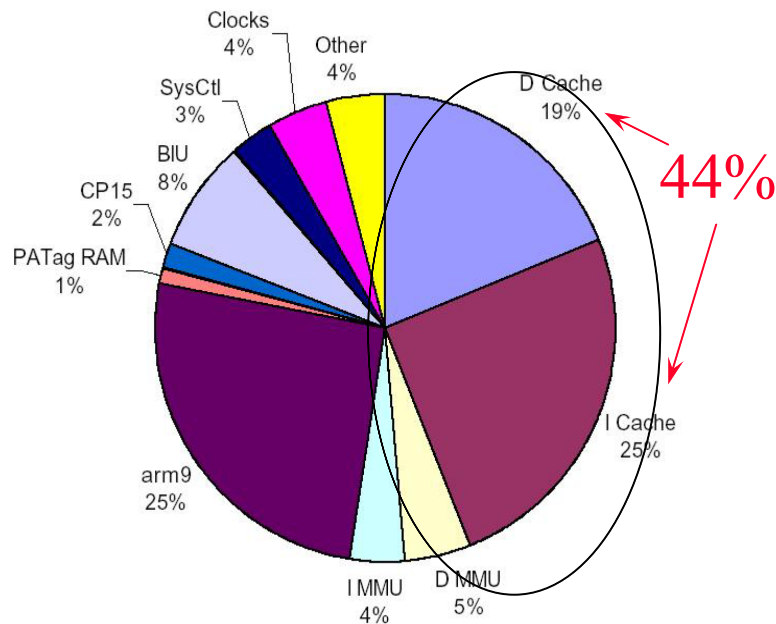

Figura 2.1: Potência consumida, em porcentagem, pelas unidades do processador ARM920T. [ARM 2003]

Portanto, demonstra-se que existe um espaço de projeto a ser explorado em sistemas embarcados quando estes são dotados de uma hierarquia de memória.

2.2 Arquiteturas com cache

Nesta seção serão apresentadas algumas arquiteturas de propósitos diferentes: a primeira criada para execução nativa de Java em sistemas embarcados; já a segunda incorpora Java no decorrer do seu desenvolvimento; adicionalmente é apresentada uma arquitetura desktop para ilustrar a diferença da mesma para com arquiteturas embarcadas; e por fim a arquitetura embarcada que será utilizada neste trabalho.

2.2.1 picoJava I

O processador pico-Java I, produzido pela *Sun Microsystems*, [O`Connor 1997, McGhan 1998, Hangal 1999] foi o primeiro processador Java para sistemas embarcados a ser disponibilizado no mercado.

O picoJava-I suporta todo o conjunto de instruções Java, entretanto, apenas estão implementadas diretamente em *hardware* aquelas que são as mais comuns em programas Java. Instruções mais complexas são executadas através de microcódigo ou através de emulação.

O projeto pico-Java I permite caches de dados e instruções com tamanhos variáveis. Isto significa que existe a flexibilidade de escolha de tamanho de caches para a economia de área no processador, já que esta é uma das características críticas em sistemas embarcados. A cache de instruções da arquitetura pico-Java I pode ser configurada para ter tamanho entre 0 a 16 Kbytes, dependendo da necessidade do sistema [O'Connor 1997].

A cache de dados deste processador possui uma associatividade por conjunto de 2 posições por conjunto. A mesma utiliza a política *write-back* para efetuar a cópia de dados modificados à memória principal, que será explicada detalhadamente neste capítulo, e o seu tamanho pode admitir os seguintes valores: 0, 1, 2, 4, 8 e 16 Kbytes.

2.2.2 ARM10E

Outro exemplo de processador típico para sistemas embarcados é o ARM1026EJ-S [ARM 2003], da família ARM10E. Este processador possui a tecnologia Jazelle [ARM 2003]. Esta tecnologia incorporou um terceiro conjunto de instruções neste processador, que suporta um subconjunto de instruções para a execução nativa de Java.

Anteriormente, as arquiteturas ARM suportavam dois conjuntos de instruções: o conjunto ARM, onde todas as mesmas têm um comprimento

de 32 *bits*, e o conjunto *Thumb*, que compacta as instruções mais usadas em um formato de 16 bits. Existe um bit no registrador de estado que indica qual o modo que o processador está trabalhando.

Como o pico-Java I, a tecnologia Jazelle divide as instruções Java em classes: executadas diretamente, emuladas e não definidas. A versão ARM1026EJ-S possui, da mesma forma que o Picojava I, cache de instruções e dados separadas e de tamanho configurável, podendo variar entre 4 a 128 KB, com associatividade de 4 posições por conjunto. A Tabela 2.2 mostra a diferença de área e potência consumida entre a arquitetura ARM10E sem cache e com a inserção de uma memória cache.

Tabela 2.2: Características de área e potência do processador ARM10E [ARM 2003]

ARM10E (versão ARM1026EJ-S)	
Área com cache (mm^2)	*4.2*
Área sem cache (mm^2)	*2.7*
mW/MHz com cache	*1.05*
mW/MHz sem cache	*0.6*

Como pode ser observado na Tabela 2.2, na inserção de uma memória cache no ARM10E ocorre um aumento de área de 1.5 mm^2 ,ou seja, 55,5 % de acréscimo na área ocupada, e um aumento de potência de mais de 0,4z mW/MHz, correspondente à quase 75% do consumo total do sistema.

2.2.3 AMD Sempron

A menção desta arquitetura serve para demonstrar as grandes diferenças existentes entre os processadores voltados a desktop e a sistemas embarcados. O processador AMD Sempron é um processador para desktop de alto desempenho. Sua arquitetura é compatível com instruções de 32 bits, incluindo suporte para SSE (*Streaming SIMD Extensions*), SSE 2, MMX, 3D Now e instruções herdadas da família x86.

A arquitetura possui dois níveis de memória cache, no primeiro nível possui cache de dados e instruções separadas. A cache de dados primária, ou também denominada *L1 Data Cache*, possui 64 kbytes de tamanho e associatividade 2 por conjunto. A cache de instruções primária, ou também denominada *L1 Instruction Cache*, também possui 64 kbytes de tamanho e associatividade 2 por conjunto.

Entretanto a cache secundária, ou *L2 cache,* armazena tanto dados como instruções e, dependendo da versão do processador, seu tamanho é de 256 kbytes ou 512 Kbytes exclusivos, ou seja, possuem dados diferentes na cache primária e na secundária, com associatividade 16 por conjunto.

Conseqüentemente com esta configuração a cache do AMD Sempron consome uma grande quantidade de potência e ocupa uma grande área no sistema, como pode ser observado na Figura 2.2.

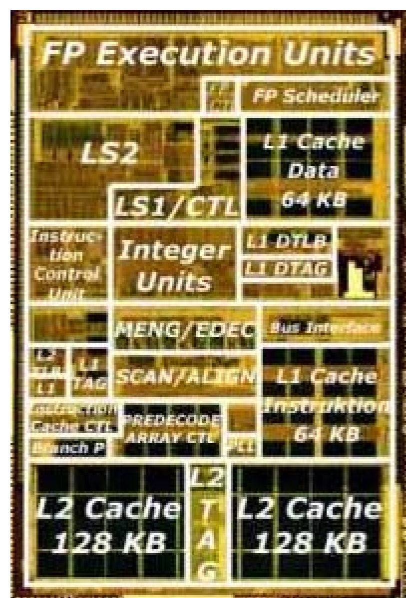

Figura 2.2: Die do processador AMD Sempron [Tom 2004]

2.2.4 Femtojava

O processador Femtojava [Ito 2001] foi criado com restrições de área e potência visando especificamente sistemas embarcados. O Femtojava implementa um subconjunto de instruções Java: são apenas 68 no total.

Neste subconjunto encontram-se instruções necessárias para operações básicas de pilha, manipulação de vetores, desvios condicionais e incondicionais, execução de métodos estáticos e acesso a campos de classes.

A Tabela 2.3 mostra o conjunto de instruções suportadas pelo Femtojava.

Tabela 2.3: Instruções suportadas pelo processador Femtojava [Ito 2001]

Tipo de instrução	
Aritméticas e lógicas	iadd, isub, imul, ineg, ishr, ishl, iushr, iand, ior, and ixor
Controle de fluxo	goto, ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, return, ireturn, invokestatic
Pilha	iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, bipush, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, swap
Load/Store	iload, iload_0, iload_1, iload_2,iload_3, istore, istore_0, istore_1, istore_2, istore_3
Vetor	iaload, baload, caload, daload, iastore, bastore, castore, sastore, arraylength
Estendidas	Load_idx, store_idx, sleep
Outras	Nop, iinc, getstatic, putstatic

Os *bytecodes* estendidos são necessários para executar instruções de E/S, programação de interrupções e também para colocar o processador em modo suspenso. O microcontrolador Femtojava só pode executar código de classes (isto é, não pode alocar objetos dinamicamente) porque seu conjunto de instruções apenas suporta *invokestatic, return e ireturn* como instruções para manipulação de métodos.

A organização de memória é baseada na alocação de quadros (*frames*) como manda a especificação da linguagem Java. Além do mais, o Femtojava implementa o sistema de E/S mapeado em memória. Outras características do Femtojava são o conjunto reduzido de instruções,

arquitetura *Harvard*, pequeno tamanho e facilidade de inserção e remoção de instruções.

Existem, atualmente, 3 versões do processador Femtojava:

- o Femtojava Multiciclo [Ito 2001] ,
- o Femtojava *Low-Power* [Beck 2003b] dotado de um *pipeline* de 5 estágios e
- o Femtojava VLIW [Beck 2003].

Todas estas versões foram implementadas com a preocupação de aumento de desempenho do sistema, com a potência consumida e também pela área ocupada pelos processadores.

Atualmente, nenhuma das versões citadas acima, do processador Femtojava, são dotadas de memória cache. Assim, explica-se a motivação deste trabalho, que é realizar um estudo de memória cache, adotando o Femtojava como arquitetura base, mais especificamente, a versão Femtojava *Low-Power* [Beck 2003b].

A versão Femtojava *Low-Power* possui um *pipeline* de cinco estágios: busca de instruções, decodificação, busca de operandos, execução e escrita de resultados, como pode ser observado na Figura 2.3. Uma das principais características deste processador é a implementação da pilha de operandos (*operand stack*) e do depósito de variáveis locais do método em um banco de registradores, ao invés de usar a memória principal para estes propósitos, como é feito no Femtojava Multiciclo

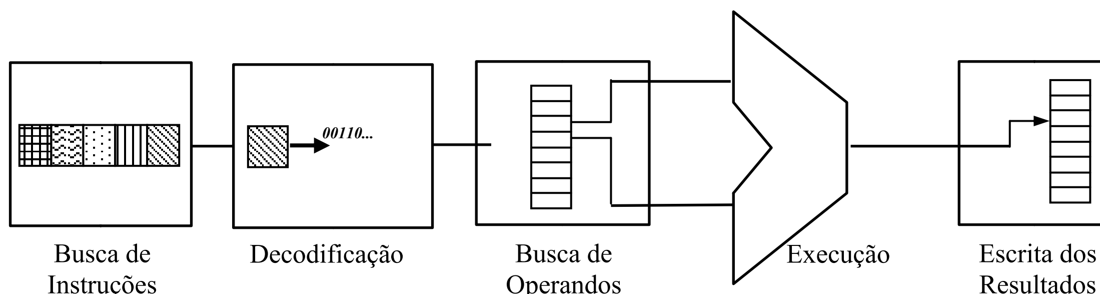

Figura 2.3: Os cinco estágios do *pipeline* do Femtojava *Low-Power*

O primeiro estágio do *pipeline* é o de busca de instruções, representado na Figura 2.3. Este estágio faz a requisição de instruções da memória de programa através de uma palavra de 32 *bits*. Ele é basicamente composto por uma fila de instruções de 9 registradores de 1 byte de comprimento cada, um registrador para indicar a seqüência de instruções a ser buscada na memória e um somador para endereçar o próximo conjunto de instruções a ser buscados na seqüência.

O segundo estágio, o estágio de decodificação, tem basicamente quatro funções: gerar a palavra de controle para a instrução corrente; informar o tamanho desta instrução para a fila de instruções, de modo que a próxima instrução do fluxo fique exatamente no primeiro lugar da fila, já que, o tamanho das instruções é variável; analisar a dependência de dados das instruções; fazer a análise de *forwarding*, passando estas informações para os estágios seguintes.

No terceiro estágio é onde os operandos são escolhidos e buscados para a execução da instrução. Ele é basicamente composto por um banco de registradores de duas portas de leitura. Neste banco podem ser feitas duas leituras independentes ou uma escrita a cada ciclo de relógio. A pilha de operandos e o repositório de variáveis locais do método estão localizados neste banco de registradores.

No estágio de execução, o quarto estágio, é onde as instruções desempenham o seu papel. É composto de uma ULA capaz de executar as tarefas de adição e subtração, além das funções lógicas básicas. Um multiplicador, uma unidade de *load/store*, um deslocador (*shifter*) e uma unidade de desvio também se encontram neste estágio, mas em diferentes blocos, para facilitar a futura parametrização do processador. Finalmente, o quinto estágio, escrita dos resultados, é responsável por salvar, se necessário, o resultado do estágio de execução no banco de registradores.

2.3 Características de Memória Cache

Em muitas arquiteturas utilizadas atualmente, nota-se a presença de uma hierarquia de memória, exemplificada na Figura 2.4.

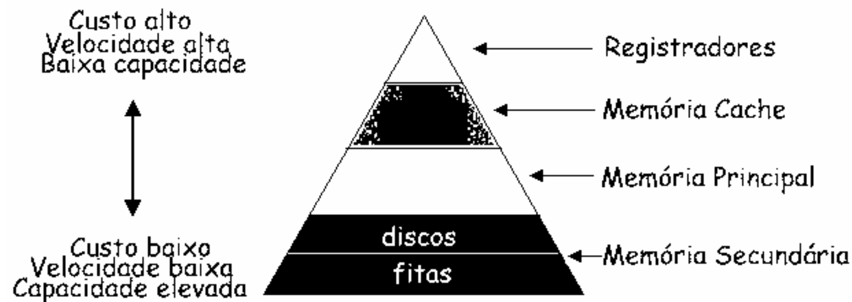

Figura 2.4: Pirâmide de hierarquia de memória

Como ilustra a Figura 2.4, a hierarquia de memória fornece à arquitetura um equilíbrio entre o custo muito alto das memórias estáticas (registradores e memória cache) e a elevada capacidade das memórias dinâmicas (memória principal) e de memórias secundárias, assim tornando a arquitetura veloz e ao mesmo tempo com um montante significativo de memória.

Porém, como pode ser identificado na Figura 2.4, quanto mais inferior for o nível na pirâmide, menor a velocidade da memória. Ou seja, se houvesse capacidade para manter todas as informações das aplicações no nível dos registradores não iria ser necessário o deslocamento a um nível inferior da pirâmide. Mas como este fato é impossível de ocorrer, por motivos de capacidade e custo, criou-se um nível intermediário na arquitetura entre os registradores e a memória principal, como pode ser observado na mesma figura, que fornece uma velocidade maior que a memória principal e ao mesmo tempo uma capacidade maior do que a dos registradores. Enquanto a memória principal fornece um tempo de acesso de 60 à 120 ns, o acesso a uma memória cache é da ordem de 5 à 25 ns. [Patterson 2003]

2.3.1 Organização da Memória Cache

A Figura 2.5 ilustra uma organização simples de memória cache com sete entradas e capacidade de armazenamento de sete informações.

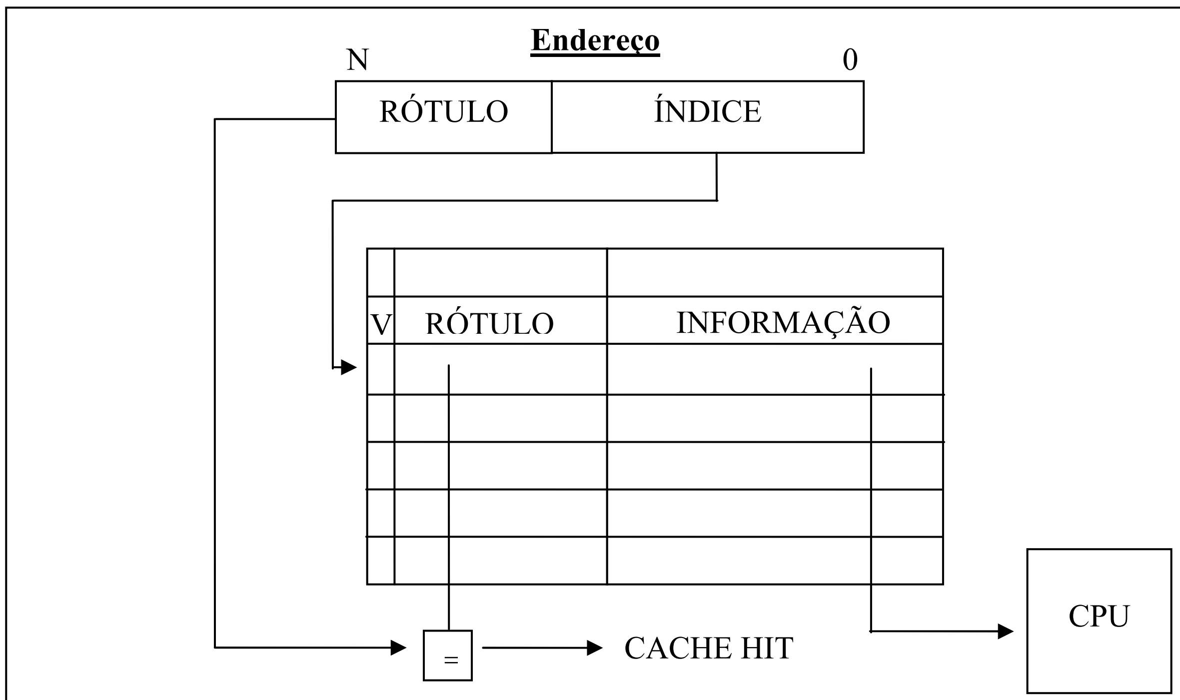

Figura 2.5: Exemplo de um endereço de memória realizando um acesso a uma memória cache.

O modelo representado na Figura 2.5 demonstra um acesso de um endereço em uma entrada da memória cache. Os bits menos significativos do endereço, que são os bits de índice, fornecem a entrada da memória cache que o dado deverá ser encontrado, em um caso de busca . Após esta identificação, é verificado o bit V, que informa se a entrada especificada contém uma informação válida, ou seja, certifica-se que esta entrada possui o último valor da mesma. Conhecida qual a entrada da memória cache e identificado que o bit V confirma que a mesma é válida, tem-se que certificar se a informação que se busca é realmente aquela que se encontra armazenada na memória cache. Para este propósito é realizada a comparação dos bits mais significativos do endereço, chamados de bits de rótulo (*tags*), com os respectivos bits da entrada da cache especificada

anteriormente pelos bits de índice. Caso o resultado da comparação fornecer que os bits são iguais, obtém-se um acerto na memória cache (*cache hit*). Assim a informação requisitada será entregue ao processador para ser utilizada. Caso contrário, a informação que está armazenada na cache não é aquela que está sendo buscada, sendo assim ocorrendo uma falta na cache (*cache miss*). Deste modo, a arquitetura terá que realizar a busca desta informação em um nível inferior da hierarquia de memória.

No caso de uma escrita na cache, ilustrada na Figura 2.5, ocorre primeiramente a verificação dos bits de índice do endereço, com a finalidade de localizar a entrada correspondente do mesmo na memória cache. Encontrada a entrada, são copiados os bits de rótulo do endereço para o seu respectivo campo, e finalmente a informação é copiada para sua posição.

A Figura 2.5 ilustrou apenas um exemplo de organização de uma memória cache. Vários conceitos de memória cache existem, e tornam esta organização variável dependendo da necessidade da arquitetura em questão. Nas próximas seções deste capítulo serão explicitados os conceitos que tornam a organização da memória cache flexível.

2.3.2 Exploração da Localidade Espacial

Quando o conceito de localidade espacial é explorado, no momento que ocorre uma falta na cache, palavras adjacentes ao endereço que gerou a falta na memória cache são buscadas. Pelo fato destas possuirem uma grande probabilidade de serem utilizadas nas próximas requisições do processador.

A utilização do conceito de localidade espacial faz com que ocorra um aumento de desempenho da memória cache pelo fato da tendência dos programas utilizarem endereços adjacentes a aqueles que foram usados

recentemente. Conseqüentemente o número de acertos na memória cache aumenta.

Esta afirmativa pode ser evidenciada, principalmente, analisando um *trace* de acessos a uma memória de instruções, onde estes são, freqüentemente, em endereços adjacentes. A comprovação deste fato ocorre pelo motivo das instruções, na maioria dos algoritmos, serem executadas de forma seqüencial, somente invalidando esta seqüência com instruções de saltos condicionais e incondicionais, chamadas de procedimento e troca do endereço corrente, como poderá ser observado no Capítulo 4.

A organização da memória cache ilustrada na Figura 2.5 não faz o uso do conceito de localidade espacial, ou seja, no momento que ocorre uma falta na cache somente o endereço que gerou esta falta será buscado na memória principal. Já a Figura 2.6 ilustra uma organização de memória cache que explora o uso da localidade espacial com a utilização de blocos de quatro informações para cada entrada da cache. A memória ilustrada na Figura 2.6 é uma cache que possui mapeamento direto, conceito que será explicitado na Seção 2.3.3.

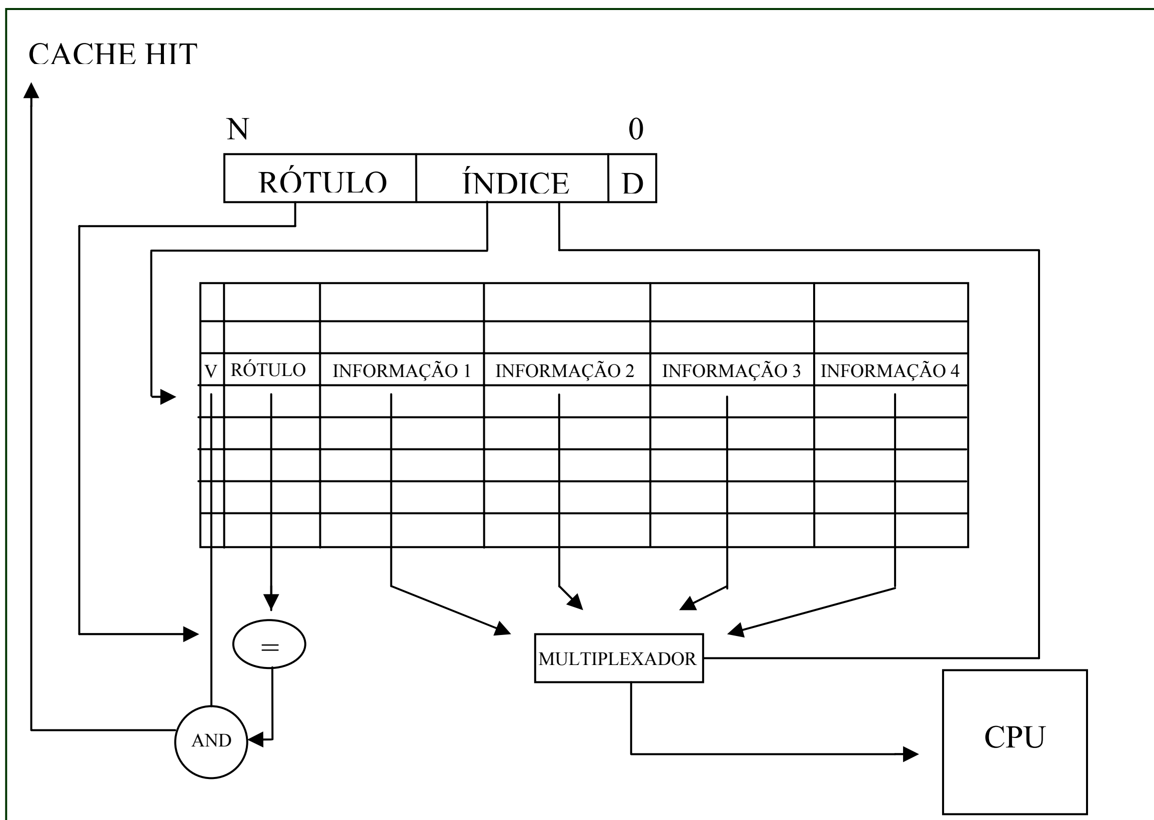

Figura 2.6: Exemplo de uma organização de memória cache explorando o uso da localidade espacial. [PATTERSON 2003]

A busca de uma informação na memória cache quando se explora a localidade espacial é realizada de forma diferente. De forma análoga a cache sem localidade espacial, os bits de índice irão informar qual a entrada que a informação investigada deve se encontrar, o bit V irá confirmar se a entrada especificada contém informações válidas e os bits de rótulo irão garantir que a informação investigada é realmente a que está na cache. Porém, até este momento não é conhecida qual das quatro informações da entrada especificada é a informação requerida, e para isto são designados os bits D, que fornecem o deslocamento da palavra dentro da entrada da cache. Deste modo, a informação requerida é selecionada pelo multiplexador e levada ao processador para ser executada.

O desempenho da cache pode diminuir se o tamanho escolhido para a localidade espacial for considerado grande comparado ao tamanho total

da cache, pois o número total de entradas da cache irá ser pequeno e as mesmas serão constantemente modificadas. Sendo assim, as informações adjacentes trazidas quando da ocorrência de uma falta não ficarão tempo suficiente na cache para serem acessadas, ocorrendo um número maior de faltas na cache. Outro fator problemático no aumento tamanho do bloco da memória cache é o custo da falta. Quando se tem uma organização de cache sem localidade espacial, como ilustra a Figura 2.5, quando ocorre uma falta, o tempo para a busca da informação em um nível inferior da hierarquia de memória é o tempo de busca de uma informação, ou seja, um tempo t. Entretanto, em uma organização de cache ilustrada na Figura 2.6, o custo na ocorrência de uma falta na cache é de $4t$, ou seja, a espera na transferência do bloco da memória principal para a memória cache cresce proporcionalmente ao crescimento do tamanho do bloco. Portanto, o crescimento exagerado na localidade espacial pode causar uma diminuição no desempenho da memória cache [Patterson 2003].

A exploração da localidade espacial, como poderá ser observado no Capítulo 4, causa um grande impacto de desempenho na cache de instruções. Isto se confirma pelo fato de um *trace* de instruções ser executado seqüencialmente na memória, tirando grande proveito do uso da localidade espacial, aumentando o número de acertos na cache.

2.3.3 Associatividade

A utilização do conceito de associatividade em uma organização de memória cache fornece uma maior flexibilidade de posições para a locação de informações na cache.

Existem 3 tipos de associatividade:

- a mapeada diretamente,
- a associativa por conjunto e
- a totalmente associativa.

Uma organização mapeada diretamente segue a regra que um endereço somente poderá ser colocado em uma posição na cache, não proporcionando nenhuma flexibilidade de posições para a cache.

Quando a cache possui uma associatividade por conjunto tem-se um número fixo de posições, definidas previamente, que são unidas formando conjuntos. Com a formação destes conjuntos a cache não é mais indexada por entradas, mas sim por conjuntos. Sendo assim, um endereço poderá ser colocado em somente um conjunto, mas como o mesmo é formado por mais de uma posição, este endereço poderá estar em qualquer destas. Este tipo de associatividade fornece uma maior flexibilidade na locação de um endereço. Dependendo da arquitetura estes conjuntos variam de no mínimo 2 posições até podendo chegar ao tamanho total da cache.

Uma organização de memória cache que possui apenas um conjunto que abrange o tamanho total da cache é denominada totalmente associativa. Portanto se tem uma flexibilidade total na inserção de um endereço na memória cache, ou seja, o mesmo poderá ser colocado em qualquer posição da cache.

A Figura 2.7 ilustra os três tipos de associatividade em memória cache.

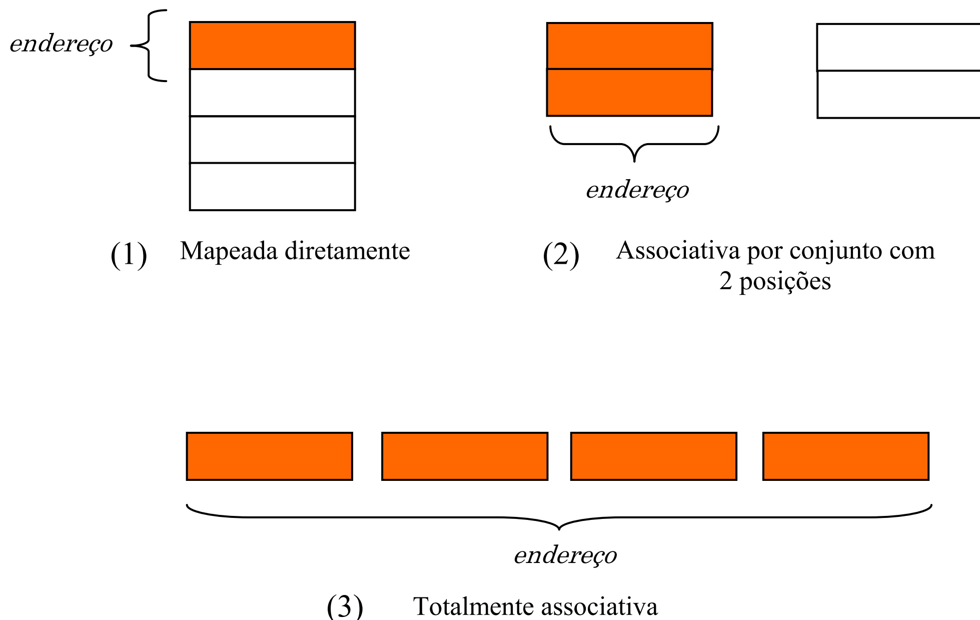

Figura 2.7: Exemplos dos tipos de associatividade em uma memória cache.

O desenho (1) da Figura 2.7 ilustra uma cache mapeada diretamente, sendo assim um endereço somente poderá ser inserido em uma posição da cache. O desenho (2) da Figura 2.7 ilustra uma cache com associatividade de duas posições por conjunto, ou seja, um endereço poderá residir em duas diferentes posições da cache. Finalmente o desenho (3) da Figura 2.7 ilustra uma cache totalmente associativa, onde um endereço poderá ser colocado em qualquer das posições da memória cache.

A utilização da associatividade em uma organização de memória cache traz como vantagem a diminuição da taxa de faltas na mesma, pelo fato de existir uma maior flexibilidade na escolha da posição em que o endereço poderá residir na cache.

A desvantagem na utilização da associatividade é observada na busca pela informação dentro do conjunto. Quando ocorre uma requisição de uma informação em uma cache com associatividade primeiramente é identificado qual conjunto esta informação está localizada, após isto é necessária a busca da informação dentro do conjunto sem qualquer tipo de indexação. Portanto se tem um custo alto de pesquisa seqüencial dentro do conjunto. Na proporção que se aumenta o grau de associatividade, ou seja, um conjunto agrega mais posições, a busca se torna mais cara, pelo fato de haver a necessidade da implementação de um *hardware* mais complexo para realizar a busca em paralelo. No entanto, se não ocorrer esta implementação a busca ficará mais lenta, então o projetista é quem irá decidir se prioriza uma resposta rápida ou uma implementação de *hardware* mais simples.

2.3.4 Algoritmos de Substituição

Quando é utilizado o conceito de associatividade em uma organização de memória cache, também é necessário empregar um algoritmo que eleja qual o bloco será substituído dentro do conjunto

especificado. No caso de uma cache mapeada diretamente não é necessário o uso de nenhum algoritmo de substituição, pois o conjunto é formado por somente um bloco e haverá apenas um rótulo a ser substituído.

Estes algoritmos têm o objetivo de diminuir o número de futuras faltas no acesso a cache. Existem diversos algoritmos que podem ser usados para esta função. Exemplos de alguns destes são:

- o LRU (*Least Recently Used*) ,
- o FIFO (*First in, First Out*) ,
- o LFU (*Least Frequently Used*) e
- o algoritmo aleatório, entre outros.

O algoritmo LRU elege para ser removido da cache sempre o bloco que foi menos recentemente usado do conjunto, ou seja, o bloco que não é utilizado há mais tempo. Este algoritmo idealiza que o bloco do conjunto que não é utilizado há mais tempo não será utilizado em um futuro breve, então ele deve ser o eleito para ser removido.

O algoritmo FIFO elege para ser removido sempre o primeiro que entrou no bloco, ou seja, o bloco mais antigo do conjunto. Parte-se do princípio que os blocos mais antigos apresentarão uma menor probabilidade de serem acessados no futuro do que os blocos que estão a menos tempo no conjunto.

O algoritmo LFU nomeia para ser removido da cache o bloco que possui uma menor freqüência de acessos. Ou seja, espera-se que os blocos com maior freqüência de acessos continuem com a mesma freqüência no futuro, aumentando o número de acertos, e aqueles que são menos freqüentemente acessados sejam removidos.

O algoritmo aleatório não possui nenhuma política de substituição de blocos, ou seja, é gerado um número pseudo-aleatório a partir de uma semente passada como argumento para o algoritmo. O valor gerado é o número do bloco que será removido.

No Capítulo 4 serão mostrados os resultados obtidos com o simulador dos algoritmos LRU e LFU, onde será realizada uma comparação dos mesmos com diferentes configurações de cache.

2.3.5 Write-Back e Write-Through

Outro fator importante que deve ser levado em conta na organização de uma memória cache é em que momento efetuar a cópia os dados à memória principal.

Existem duas políticas para realizar esta cópia:

- *Write-Back* e
- *Write-Through*.

O método *write-back*, ou também chamado de *copy-back*, segue o princípio de manter as informações arquivadas na cache, onde as informações modificadas na cache não serão atualizadas imediatamente na memória principal. As informações modificadas somente serão copiadas para a memória principal quando o bloco que armazena as mesmas na cache for substituído.

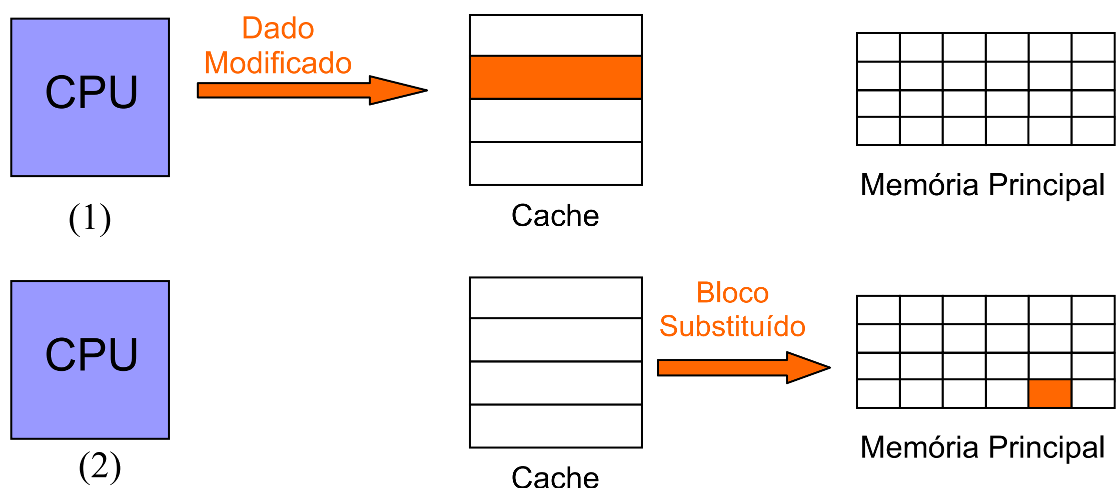

Figura 2.8: Exemplo do método de Write-Back para uma informação modificada.

A Figura 2.8 exemplifica uma informação modificada pelo processador usando o método de *Write-Back* para efetuar a cópia para a memória principal. Na etapa (1), o processador executou uma instrução que modificou uma informação já existente na cache. Assim o bloco da cache que loca esta informação recebe o novo valor da mesma. Conservando o valor antigo da informação na memória principal. Na etapa (2), o bloco que contém esta informação foi substituído na memória cache, sendo assim a posição da memória principal que se refere a esta informação recebe a mesma atualizada.

O método *Write-Back* é muito utilizado, pois diminui o número de cópias de informações modificadas à memória principal, gerando assim um melhor desempenho e economizando potência devido aos acessos à memória principal. Além do mais, as informações podem ser escritas pelo processador na velocidade da cache ao invés de ser na velocidade da memória principal, com o uso deste método.

Por outro lado, a política *Write-Through* regra que quando o processador executa uma escrita de uma informação, imediatamente o novo valor da informação modificada é propagado também para a memória principal.

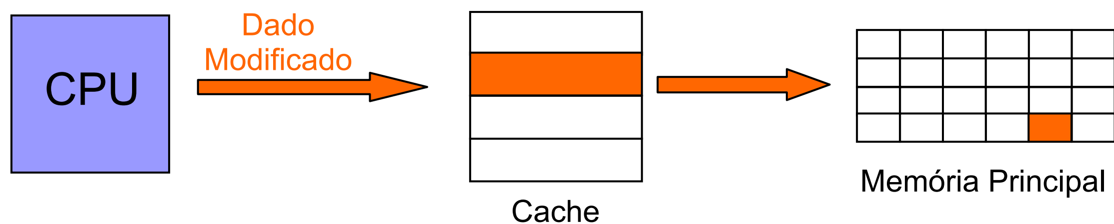

Figura 2.9: Exemplo do método de Write-Through para uma informação modificada.

A Figura 2.9 ilustra um exemplo de uma informação sendo modificada pelo processador com o método de *Write-Through*. Diferentemente da Figura 2.8, a Figura 2.9 somente possui uma etapa, pois

com o método *Write-Through* nota-se que quando o processador envia a informação modificada à cache, também é enviada paralelamente à memória principal o novo valor da mesma, atualizando as posições de ambas as memórias.

Quando se tem uma organização de cache com associatividade grande, ou seja, quando a cache possui blocos que agregam muitas informações, é mais vantajoso fazer o uso do método de *Write-Through*, pois o tratamento da cópia individual é mais rápido do que utilizar *Write-Back* e forçar a cópia do bloco inteiro à memória principal [Patterson 2003].

CAPÍTULO 3

SISTEMA DE PARAMETRIZAÇÃO

Na realização deste trabalho, foi proposta a criação de um sistema de parametrização capaz de realizar simulações, e com isso, obter dados de diferentes configurações de memória cache, com a finalidade de melhor explorar o espaço de projeto.

O objetivo principal na implementação deste simulador foi parametrizar algumas características de cache sem que fosse necessária a implementação em hardware de cada configuração. O simulador foi construído para medir o impacto das diferentes configurações de cache em desempenho, área e potência no processador Femtojava *Low-Power* com o intuito de localizar as soluções mais satisfatórias.

O simulador foi escrito totalmente em Linguagem C, por motivos de desempenho e para ser mais facilmente integrado ao simulador CACO-PS (*Cycle-Accurate COnfigurable Power Simulator*) [Beck 2003]. CACO-PS é um simulador de código compilado que calcula a potência baseada na taxa de chaveamento de diferentes componentes do sistema [Simunié 1999], onde estes também são descritos em linguagem C. A ferramenta foi compilada e testada no sistema operacional Linux Fedora Core 1, com o compilador GCC versão 3.2.2 . O código fonte do simulador está disponível no Anexo II.

3.1 Funcionamento do Simulador

A ferramenta é capaz de simular diferentes configurações de cache parametrizando as seguintes características, como:

- o tamanho da memória cache,
- a localidade espacial da cache,
- a associatividade e
- dois algoritmos de substituição de blocos, o algoritmo LRU e o algoritmo LFU.

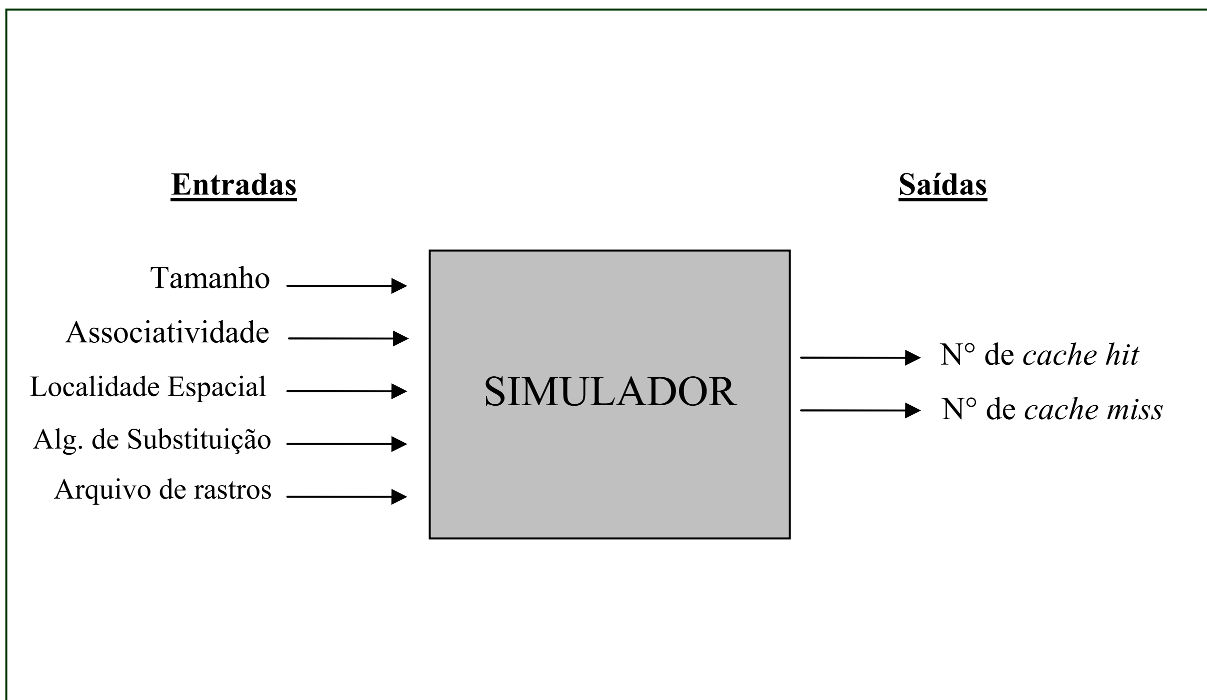

Figura 3.1: Entradas e Saídas do simulador proposto.

A Figura 3.1 ilustra com quais dados de entradas são necessários ao simulador e quais dados de saída ele fornece ao usuário. A primeira entrada do simulador refere-se ao tamanho da memória cache e ao tamanho da memória principal. Estes parâmetros foram inseridos para que se possam realizar simulações com diversos tamanhos de cache e diversos tamanhos

de memória principal, analisando qual é a influência deste no número de faltas na cache.

O parâmetro associatividade foi inserido como entrada do simulador para explorar uma maior flexibilidade na locação das informações na cache, e posteriormente analisar o quanto este influi no desempenho da mesma.

Conjugado ao parâmetro associatividade foram inseridos dois algoritmos de substituição, para que exista uma variação no método de substituição das informações dentro dos conjuntos associativos. O algoritmo LRU retira as informações que não foram utilizadas há mais tempo dentro do conjunto, e o algoritmo LFU substitui as informações que tiveram uma freqüência menor de uso em relação às outras informações do conjunto, como já explicado no capítulo 2. Nota-se que o comportamento dos dois algoritmos são totalmente diferentes, e por este motivo que eles foram implementados.

Existem outros algoritmos de substituição. Um deles é o algoritmo randômico, que substitui as informações sem nenhum método fixo, e sim de forma aleatória. O algoritmo randômico tem um desempenho pior que os outros dois algoritmos citados anteriormente, para associatividades relativamente pequenas, que é o caso deste trabalho [Patterson 2003]. Assim, para restringir o espaço de exploração de projeto e diminuir o número de configurações possíveis, este algoritmo não foi implementado.

Outro parâmetro disponível no simulador, também ilustrado pela Figura 3.1, é a configuração da localidade espacial da memória cache, já explicado no capítulo anterior.

Para que todos os parâmetros anteriores possam ser simulados autenticamente, tem-se que avaliar os mesmos com *traces* de algoritmos em uma memória cache. Portanto, como ilustra a Figura 3.1, o simulador contém como última entrada um arquivo que descreve um *trace* de acessos

de um algoritmo em uma memória cache. O arquivo em questão possui a seguinte estrutura ilustrada na Figura 3.2.

```
1          R  06FA
2.         W  7A89
3.         R  007E
```

Figura 3.2: Arquivo que descreve um trace de acessos à memória.

O arquivo de *trace* ilustrado na Figura 3.2 descreve na linha 1 com o primeiro argumento "R", uma leitura na memória principal, no endereço fornecido pelo segundo argumento "06FA". Já na linha 2, o primeiro argumento "W" indica uma escrita na memória principal no endereço "7A89", fornecido pelo segundo argumento. Os endereços estão no formato hexadecimal.

Com todos os parâmetros de entrada, anteriormente explicitados, informados ao simulador, ao final da execução será fornecido o número e a percentagem de faltas (*cache miss*) e acertos (*cache hit*) ocorridos no algoritmo cujos acessos à memória foram passados como parâmetro no formato mostrado anteriormente.

3.2 Descrevendo a estrutura do simulador

Como já esclarecido anteriormente, o simulador foi escrito em linguagem C e tem como estrutura principal, a que segue na Figura 3.3.

```
1.        typedef struct{
2.                int rotulo;
3.                int dado[4];
4.                int valido;
5.                int palavra;
6.        }cache[MAX];
```

Figura 3.3: Estrutura que descreve uma cache em linguagem C.

A estrutura ilustrada na Figura 3.3 está demonstrada graficamente na Figura 3.4, comparando os campos da estrutura com os campos da memória cache, onde o campo rótulo representa a variável inteira *rotulo* da estrutura. Como na Figura 3.4 há a exploração da localidade espacial, contendo 4 informações por entrada, estas informações são representadas pelo vetor de inteiros *dado* na estrutura do simulador. O bit V do desenho representa a variável inteira *valido* na estrutura. E finalmente, a variável inteira *palavra* da estrutura está representada no desenho pelos bits D, ou seja, a mesma irá fornecer o deslocamento da palavra dentro da entrada específica na cache e somente será utilizada se houver mais de uma palavra por entrada na cache.

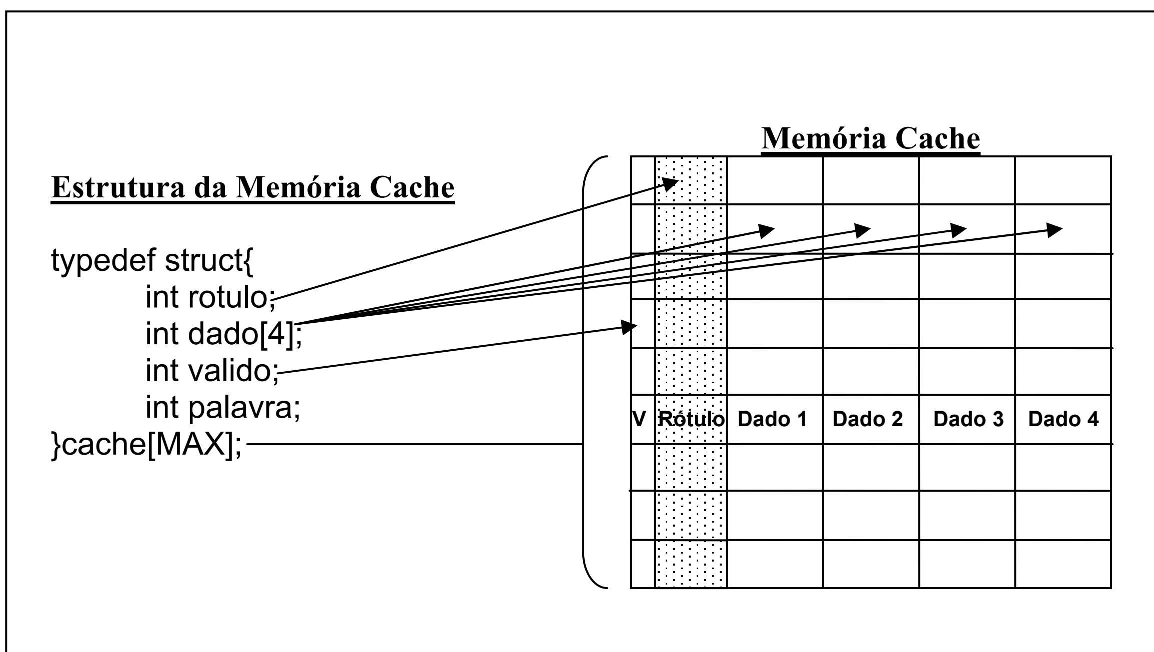

Figura 3.4: Demonstração dos campos da estrutura na memória cache.

O número de entradas da cache é dado pela definição *MAX*, que é variável para um mesmo valor de tamanho de cache passado como argumento no inicio da simulação. Isto se justifica, pelo motivo que o número de entradas depende diretamente da localidade espacial também indicada como argumento da simulação. Por exemplo, se que um tamanho de cache passado para a ferramenta fosse de 64 com uma localidade

espacial de uma informação por entrada na cache, como ilustra o desenho (2) da Figura 3.5, então haveria 64 entradas na cache. Entretanto, se o mesmo valor de tamanho fosse passado, e desta vez uma localidade espacial de quatro palavras por entrada, assim haveria somente 16 entradas na cache, como mostra o desenho (1) da Figura 3.5.

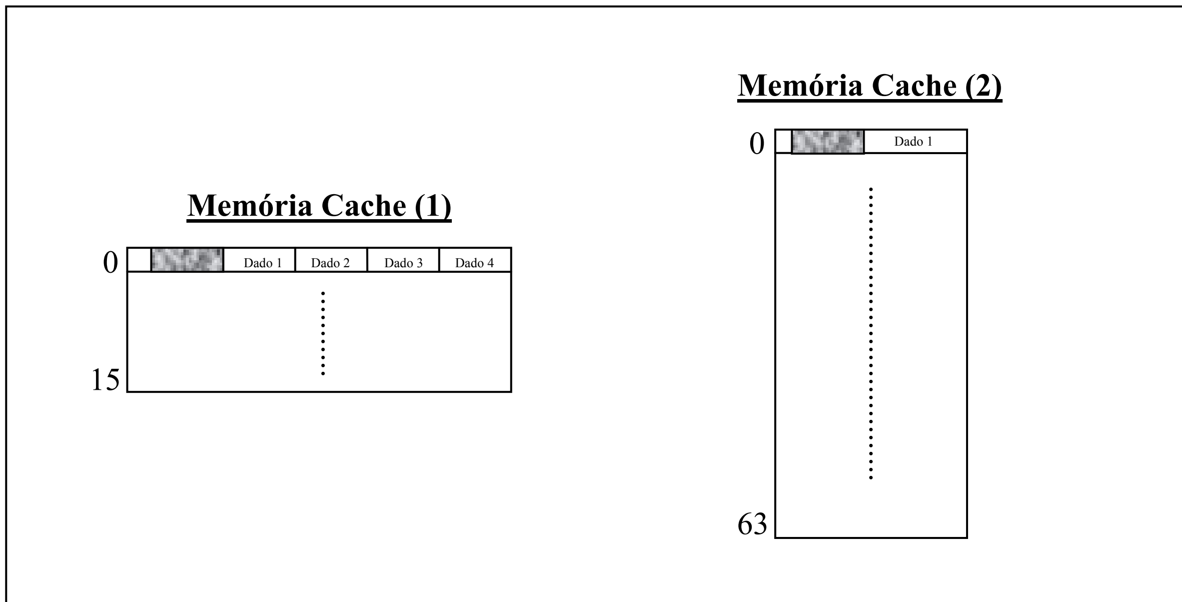

Figura 3.5: Duas memórias cache com tamanho 64 e com diferentes localidades espaciais

Na Figura 3.5 mostra-se que variando a localidade espacial da cache, e deixando o tamanho constante toda a estrutura da cache é modificada. Quando existe uma cache com tamanho 64 e 64 entradas (cache (2)) são necessários 6 bits para indexar as entradas da cache (campo de índice no endereço). Se os endereços forem de 16 bits, restam 10 bits para o campo de rótulo. Entretanto, com uma cache de tamanho 64 e com 16 entradas (cache (1)), necessitam-se somente de 4 bits para o campo de índice e restam 10 bits para o rótulo e 2 bits para indicar o deslocamento da informação dentro da entrada.

A solução para este problema é a utilização de máscaras que são criadas a partir dos argumentos de tamanho da memória cache e de localidade espacial fixados no início da simulação. Estas máscaras possuem a função de separar o número de bits corretos do endereço para os campos de rótulo, índice e deslocamento, dependendo dos valores passados pelos argumentos.

A alocação de memória física para toda a estrutura apresentada anteriormente é realizada estaticamente, por dois motivos:

- desempenho do simulador, evitando a alocação de memória em tempo de execução e
- pelo fato do tamanho da cache ser passado como argumento ao simulador e seu valor nunca ser modificado durante a execução.

O único inconveniente da alocação estática é que o valor passado tem que ser sempre menor que o definido.

CAPÍTULO 4

RESULTADOS DE SIMULAÇÃO

Neste capítulo, primeiramente, serão apresentados os algoritmos utilizados nas simulações. Posteriormente, será descrita qual a metodologia usada na obtenção dos resultados. Em seguida, serão apresentados os resultados obtidos na execução do simulador, parametrizando algumas características de memória cache, ressaltando o impacto dos resultados no desempenho, na área e na potência consumida.

Por motivo de restrição de espaço, a tabela com todos os resultados simulados neste capítulo está no Anexo I.

Como existem várias configurações possíveis, em alguns experimentos a seguir, irão ser escolhidas certas configurações de cache que explicitam diferenças grandes da respectiva característica a qual se quer esclarecer.

4.1 Programas para *Benchmark*

Os algoritmos usados na simulação foram implementados e simulados na arquitetura Femtojava *Low Power*. Foram utilizados:

- um algoritmo para cálculo do seno, bastante utilizado em bibliotecas aritméticas. Este algoritmo utiliza o método CORDIC (*Coordinate Rotation Digital Computer*) [Volder 1959] para calcular o resultado;

- o IMDCT (*Inverse Modified Discrete Cosine Transformation*), uma importante parte de algoritmo de descompressão multimídia;
- o sistema Crane [Moser 1999] que foi proposto como um *benchmark* na área de modelagem e síntese de sistemas, envolvendo modelos heterogêneos de computação;
- três diferentes algoritmos de ordenação foram utilizados: usando as técnicas *bubblesort*, *selectsort* e *quicksort*;
- um algoritmo de MP3 que é dividido em 7 partes, trata-se de um algoritmo que executa 4 quadros de 40 kbit, em 22050 Hz e
- uma biblioteca que emula somas de ponto flutuante, já que o processador Femtojava *Low Power* não possui esta unidade a fim de economizar área. Esta realiza 20 somas de dois números em ponto flutuante e colocando os resultados em um vetor na memória;

A principal razão na escolha destes algoritmos é pelo fato de eles serem bastante utilizados no domínio de sistemas embarcados. Outra razão, se dá pelo fato dos mesmos possuírem características diferentes na execução, de acesso à memória. Então, esta escolha foi feita para que ocorra uma melhor exploração na obtenção dos resultados, pois isto causa impacto nos acertos na cache.

Os algoritmos se diferem na freqüência de comandos de controle, como desvios condicionais, existentes na execução dos mesmos. Assim, algoritmos que possuem muitos destes comandos são denominados de *controlflow*, já aqueles em que a freqüência destes é baixa são chamados de *dataflow*.

Dentre os algoritmos selecionados, o algoritmo Crane é considerado *controlflow*, ou seja, possui muitos comandos de controle em seu código, ao contrário do IMDCT, que é *dataflow*.

4.2 Metodologia

Utilizando o simulador explicitado anteriormente foram realizadas simulações englobando todos os algoritmos citados na Seção 4.1, na versão Femtojava *Low Power*, de diversas configurações de cache. Destas simulações, resultou o número de cache *misses* e cache *hits* de cada algoritmo em questão.

Posteriormente, foi analisado o desempenho, em número de ciclos, na arquitetura Femtojava *Low-Power*, para cada algoritmo simulado anteriormente. Para esta tarefa foi utilizado o simulador CACO-PS. Com este simulador foram obtidos os valores, em números de ciclos, apresentados na Tabela 4.1. É importante ressaltar que a arquitetura Femtojava já se encontra disponível em uma versão VHDL pronta para prototipação. Assim, o simulador apenas é usado para diminuir o tempo de execução dos diversos algoritmos.

Tabela 4.1: Número de ciclos gastos, na arquitetura Femtojava *Low-Power*, com os algoritmos utilizados na simulação.

Algoritmo	*Número de ciclos*
Bubblesort (10 elementos)	2424
Bubblesort (100 elementos)	339797
Crane	36656
Somas ponto flutuante	14531
IMDCT	40306
Quicksort (10 elementos)	1516
Quicksort (100 elementos)	13239
Selectsort (10 elementos)	1930
Selectsort (100 elementos)	134090
CORDIC	755
Mp3 Parte 1	242153
Mp3 Parte 2	109396
Mp3 Parte 3	64488
Mp3 Parte 4	41587
Mp3 Parte 5	35895
Mp3 Parte 6	159017
Mp3 Parte 7	1790671

Para cada algoritmo mostrado na Tabela 4.1, foram gerados dois *traces* de acesso à memória: um para a memória de dados e outro para a de

instruções. Assim, consegue-se simular, separadamente, a cache de dados e a cache de instruções, já que como poderá ser visualizado nos resultados, as duas possuem características totalmente diferentes em relação ao tráfego de informações.

Com os resultados obtidos nas simulações de cache e de número de ciclos gastos com os algoritmos, foi realizado o relacionamento destes, resultando no número total de ciclos perdidos por cache *misses*, em cada configuração de cache, para todos os algoritmos demonstrados na Tabela 4.1. Conseqüentemente, com estes resultados pode-se comparar o desempenho da cache entre as diversas configurações simuladas.

Para realizar a exploração do impacto da área nas configurações da cache, foi estimada a quantidade de células lógicas para cada configuração de cache investigada. Esta estimativa foi feita a partir de diferentes tamanhos de memória prototipadas em FPGA a partir de suas descrições em VHDL. Serão demonstrados as diversas configurações de cache e os resultados de área obtidos com as mesmas.

A obtenção dos dados de potência, tanto para o processador quanto para a memória cache, foi feita a partir da estimativa do número de capacitâncias de *gates* equivalentes (CG) para cada componente no sistema. Assim, é possível obter a potência dinâmica relativa consumida pelo sistema.

4.3 Resultados de desempenho da memória cache

Cache de dados

Parametrização da localidade espacial

Como já explicitado no Capítulo 2, a utilização da localidade espacial explora a freqüência da utilização de endereços adjacentes a àquele que foi utilizado recentemente.

A Figura 4.1 ilustra o número de ciclos perdidos por cache *misses* nos algoritmos citados na Seção 4.1, parametrizando a localidade espacial em cada um destes, simulados em uma cache de dados com tamanho de 256 entradas e mapeada diretamente.

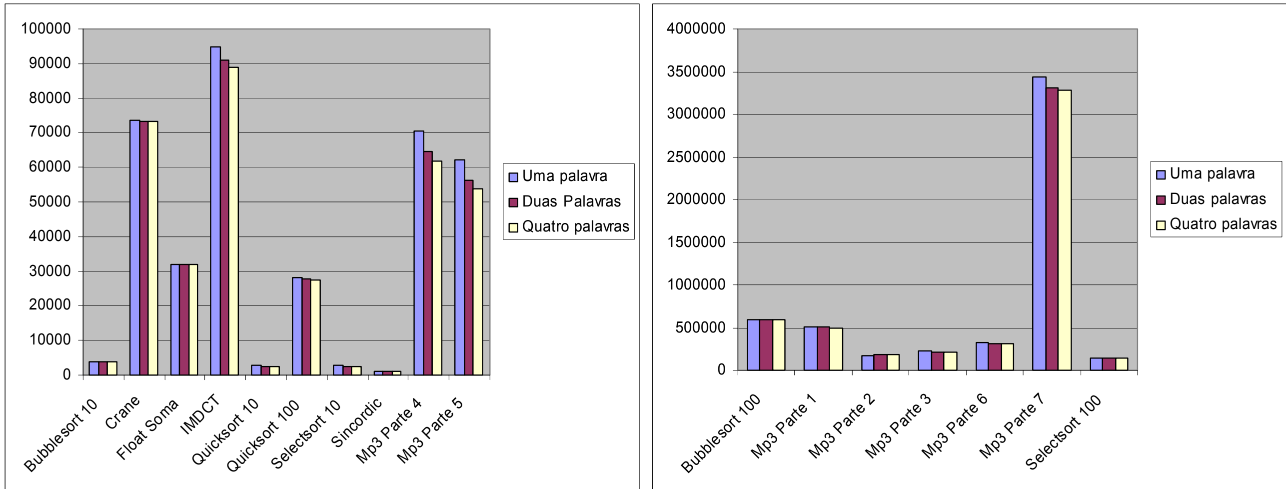

Figura 4.1 Ciclos perdidos por cache miss em cada algoritmo para diferentes localidades espaciais, na cache de dados

Avaliando a Figura 4.1, nota-se que com o aumento da localidade espacial, na maioria dos algoritmos, há uma leve melhora no desempenho destes. Isto é explicado pelo fato de os dados geralmente não possuírem uma execução seqüencial, não explorando, portanto, a utilização da localidade espacial.

Observa-se esta avaliação com o algoritmo Crane que, com uma palavra, tem sua execução dada em 73.588 ciclos. Com duas palavras, o desempenho aumenta para 73.371 ciclos, uma redução de apenas 0,3%. Posteriormente, executando o mesmo mas explorando a localidade espacial com 4 palavras, sua execução é realizada em 73.224 ciclos, ocorrendo uma redução de 0,5 % em relação à execução sem o uso da localidade espacial.

Parametrização do tamanho da memória cache

O parâmetro tamanho foi explorado para observar o impacto que o mesmo causa em diferentes tipos de algoritmos. Para isto, foram escolhidos

algoritmos que ocupam pouca memória como o *Quicksort*, e outros que ocupam muita memória, como o algoritmo de MP3.

A Figura 4.2 ilustra o número de ciclos perdidos por cache miss em uma cache de dados, parametrizando diversos tamanhos de cache para cada algoritmo. Para gerar esta figura foi utilizada uma cache mapeada diretamente com localidade espacial de 4 palavras.

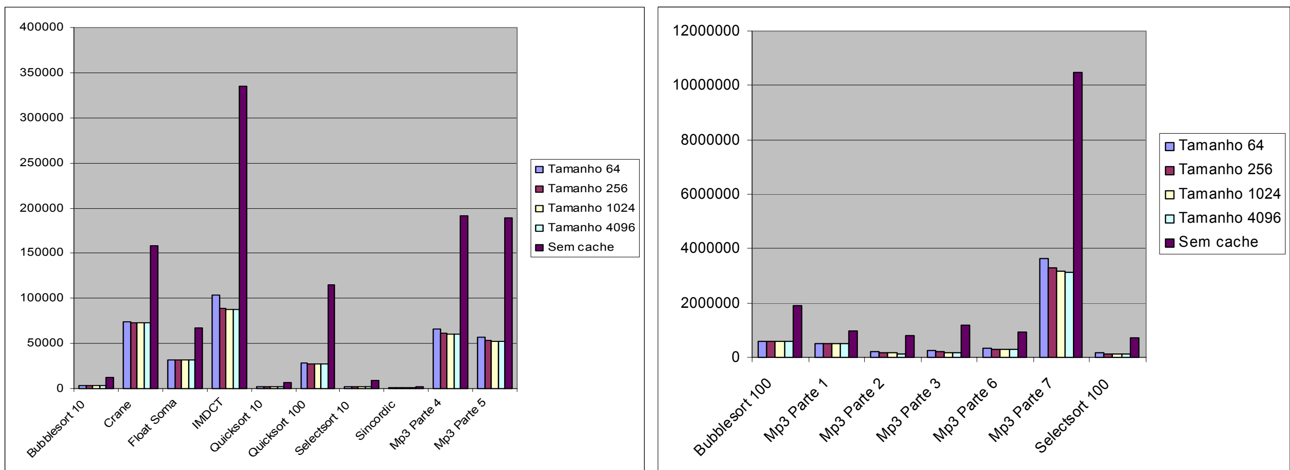

Figura 4.2: Ciclos perdidos por cache miss em cada algoritmo para diferentes tamanhos de cache, na cache de dados.

Na avaliação da Figura 4.2, pode-se observar que há um ganho drástico, no número de ciclos, na inserção de uma cache na arquitetura. Observa-se também que alguns algoritmos estabilizam o desempenho enquanto a cache aumenta de tamanho, pelo motivo de os mesmos não necessitarem de uma grande quantia de memória.

Em outros casos há uma crescente melhoria do desempenho quando do aumento na capacidade da memória cache. Isto pode ser evidenciado no algoritmo de MP3 (Parte 6): quando o mesmo é simulado em uma cache com tamanho 64 ele gasta 328.144 ciclos, já com o aumento da cache para 256 o algoritmo é executado em 306.367 ciclos, um rendimento de 6,6% no desempenho. Acrescentado mais memória ao sistema, fixando o tamanho em 1024 entradas, o mesmo executa em 301.586 ciclos, adicionando mais 1,6% no desempenho anterior. Finalmente, simulando uma cache com

tamanho de 4096 o algoritmo consome 300.186 ciclos que chega a um rendimento no desempenho de 8,5% em relação a cache com tamanho 64.

Portanto, é observado que alguns algoritmos, por ocuparem uma área de dados pequena, não necessitam de tanta memória para obter um desempenho estável. Já outros, por ocuparem áreas maiores, continuam com um crescente desempenho ao passo que é adicionada mais cache ao sistema.

Parametrização da associatividade da cache

O parâmetro associatividade foi inserido nas simulações para obter dados quanto à flexibilidade de se substituir as informações em diferentes posições da memória cache. Assim, foram feitas simulações com dois diferentes algoritmos: LRU e o LFU.

A Figura 4.3 ilustra o número de ciclos perdidos por cache *miss* em cada algoritmo em uma cache de dados de tamanho de 1024 entradas, que não faz exploração da localidade espacial. Variando a associatividade nos mesmos, nesta figura é utilizado o algoritmo LRU para realizar as substituições das informações. Já na Figura 4.4 todas as características da Figura 4.3 permanecem, com exceção do algoritmo de substituição, que nesta simulação é o LFU.

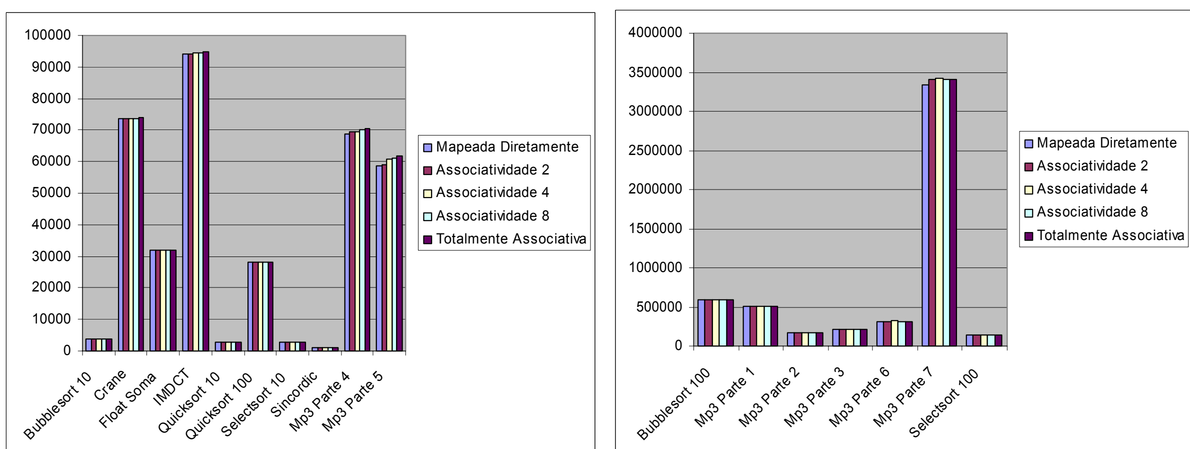

Figura 4.3: Ciclos perdidos por cache miss em cada algoritmo para diferentes associatividades usando o algoritmo LRU, na cache de dados.

Observa-se na Figura 4.3, que com o algoritmo de substituição LRU a maioria dos algoritmos simulados não obtiveram uma melhoria no desempenho com o aumento da associatividade. A tendência dos algoritmos, com o aumento da associatividade, foi de se manter com o mesmo desempenho (apesar de que em alguns casos ocorreu uma pequena diminuição). Um exemplo deste ocorreu com o algoritmo IMDCT, em que com uma associatividade 4 foi executado em 94.605 ciclos. Já em uma cache com associatividade total este algoritmo apresentou 94.633 ciclos em sua execução.

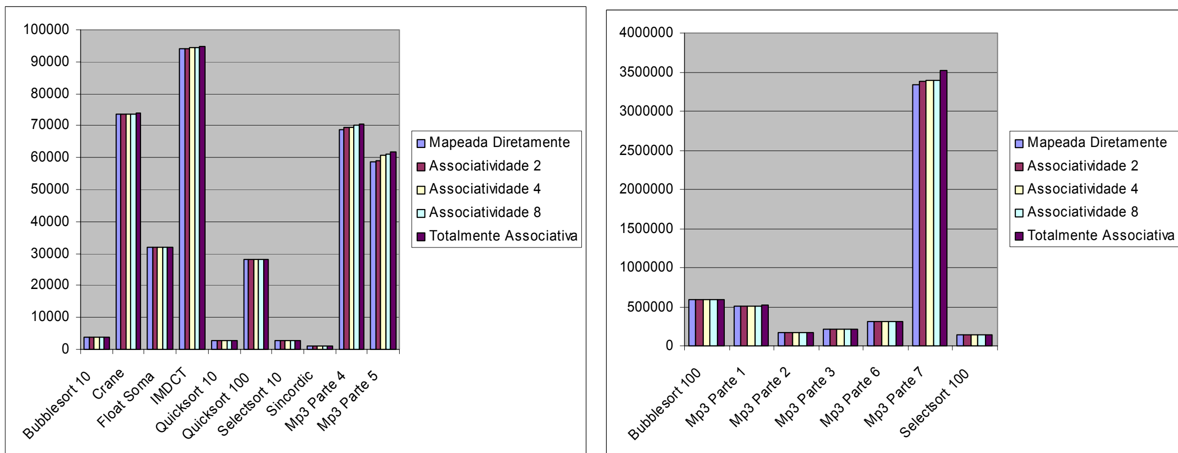

Figura 4.4: Ciclos perdidos por cache miss em cada algoritmo para diferentes associatividades usando o algoritmo LFU, na cache de dados.

Na Figura 4.4 foi utilizado o algoritmo LFU. Observa-se nesta figura que, do mesmo modo que ocorreu com o algoritmo LRU, com o aumento da associatividade não se obteve melhoras em nenhum algoritmo. Ao contrário, muitos destes tiveram um acréscimo em seus ciclos, com o aumento da associatividade. Exemplo disto ocorreu com o algoritmo de MP3 (Parte 7) que com uma cache mapeada diretamente foi executado em 3.333.464 ciclos. Já com uma com associatividade 2 teve um aumento de 50.036 ciclos na execução do mesmo.

Assim, ressalta-se que o aumento da associatividade em conjunto com estes dois algoritmos de substituição não trouxeram ganhos significativos em termos de desempenho, tratando-se da cache de dados.

Cache de Instruções

Parametrização da localidade espacial

A utilização da localidade espacial em uma cache de instruções possui características totalmente distintas das mesmas em uma cache de dados. Isto se dá pelo comportamento desta memória, onde seus acessos acontecem geralmente por endereços seqüenciais. Sendo assim, o uso da localidade espacial em uma cache de instruções causa um maior impacto no desempenho do que em uma cache de dados, como se observa na Figura 4.5.

A Figura 4.5 ilustra o número de ciclos perdidos por cache *misses* nos algoritmos citados na Seção 4.1, parametrizando a localidade espacial em cada um destes, simulados em uma cache de instruções com tamanho 256 e mapeada diretamente.

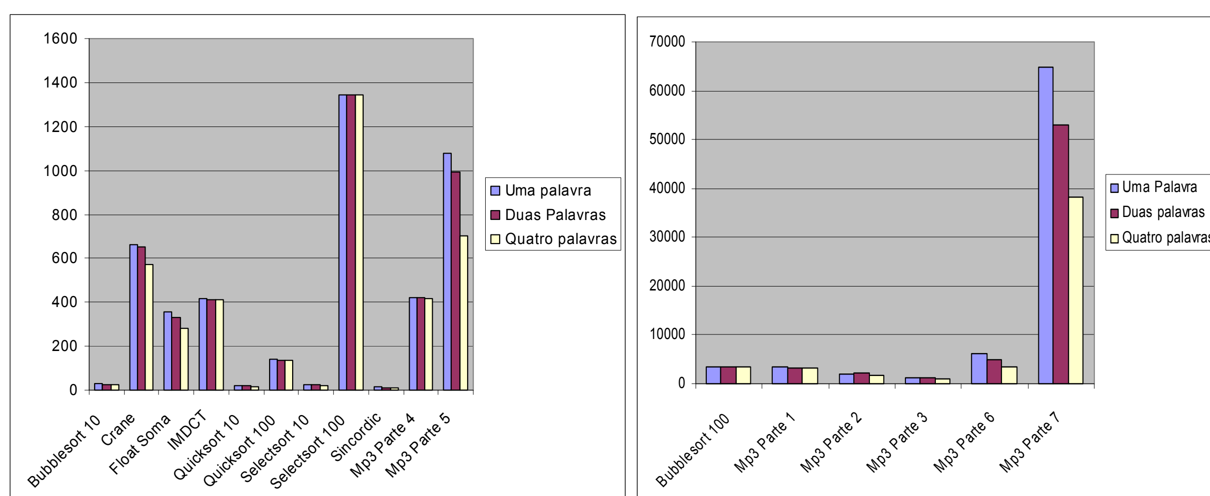

Figura 4.5: Ciclos perdidos por cache miss em cada algoritmo para diferentes localidades espaciais, na cache de instruções.

Diferentemente da cache de dados, na cache de instruções a utilização da localidade espacial beneficia, drasticamente, no desempenho alguns algoritmos, como ilustra a Figura 4.5.

Observa-se este fato com o algoritmo de MP3 (Parte 7) que, sem o uso da localidade espacial, é executado em 6.471.536 ciclos. Com o uso da

localidade espacial com duas palavras o mesmo é executado em 5.292.197 ciclos, ocorre uma redução de 18,3% no número de ciclos. Já com o uso de 4 palavras o algoritmo executa em 3.814.000 ciclos, o que representa uma melhora de desempenho de 41% em relação a configuração de cache sem o uso de localidade espacial.

Nota-se, entretanto, que em alguns algoritmos não há uma melhoria significativa de desempenho. Explica-se isto pelo fato de os mesmos possuírem muitas instruções como desvios condicionais ou incondicionais e chamadas de procedimento. Estas instruções irão fazer com que não ocorram acessos seqüenciais às instruções em grande quantidade.

Parametrização do tamanho da memória cache

Nesta subseção é parametrizado o tamanho da memória cache para mesurar o impacto no desempenho para a cache de instruções. A Figura 4.6 ilustra os ciclos perdidos por cache *misses* com os algoritmos utilizados neste trabalho, em diferentes configurações de tamanho de memória cache que arquivam instruções. Para gerar esta figura foi utilizada uma cache mapeada diretamente com localidade espacial de 4 palavras.

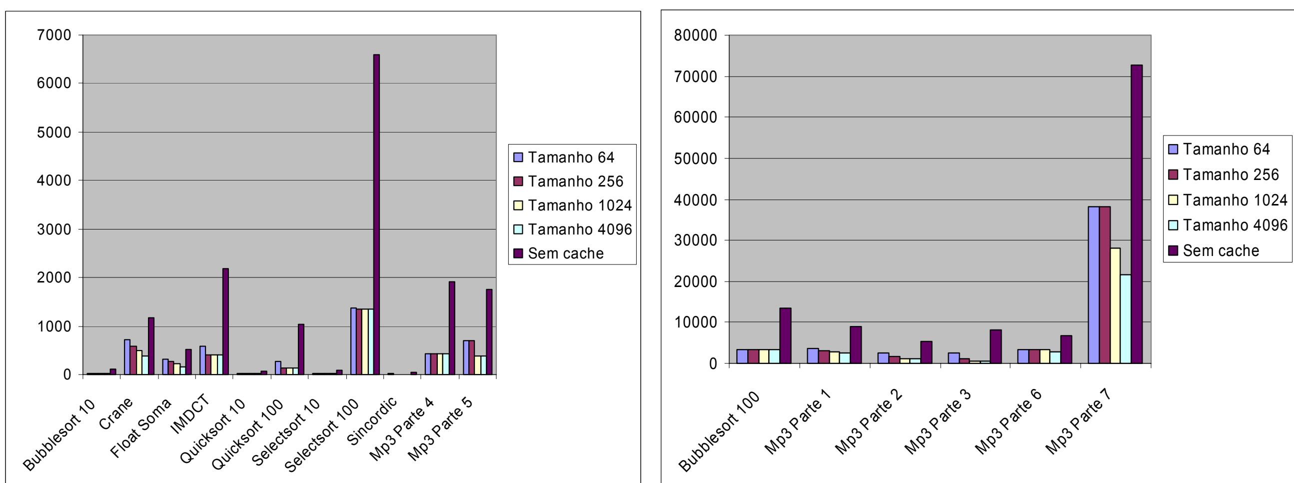

Figura 4.6: Ciclos perdidos por cache miss em cada algoritmo para diferentes tamanhos de cache, na cache de instruções.

Do mesmo modo que a cache de dados, observa-se na Figura 4.6 que alguns algoritmos chegam em uma configuração de tamanho de cache e estabilizam seu desempenho, não obtendo mais nenhuma melhoria. Acontece este fato com algoritmos que possuem poucas instruções ou que ficam concentrados em pequenas partes do código e, provavelmente, as repetem muitas vezes, não necessitando de grande quantia de memória. No outro lado estão os algoritmos que ocupam uma maior quantia de memória, como o algoritmo de MP3. Este algoritmo, como ilustra a Figura 4.6, obtém uma crescente melhora no desempenho ao passo que o tamanho da memória cache aumenta. Em uma cache de tamanho 64 entradas o algoritmo é executado em 3.822.540 ciclos. Já em uma cache de tamanho fixado em 256 entradas o número de ciclos diminui para 3.814.000, com um rendimento no desempenho de 0,3%. Com uma memória maior, de tamanho 1024 entradas, o mesmo é executado em 2.804.299 ciclos, ocorrendo uma melhora no desempenho de 26,5% em relação ao tamanho 256 entradas. No mesmo caminho, com 4096 entradas, o resultado da execução do algoritmo foi de 2.162.364 ciclos, ou seja, um rendimento de desempenho de 44% em relação à cache de tamanho 256 entradas.

Assim, nota-se que algoritmos maiores, e que provavelmente exploram grandes ou diversas partes de código, necessitam de uma cache com maior capacidade para obterem um desempenho melhor.

Parametrização da associatividade da cache

A Figura 4.7 ilustra o número de ciclos perdidos por cache *misses* em cada algoritmo em uma cache de instruções com 1024 entradas e sem explorar a localidade espacial (da mesma forma de quando foi analisada esta parametrização para a cache de dados), parametrizando a associatividade nos mesmos. Nesta figura é utilizado o algoritmo LRU para realizar as substituições das informações. Já na Figura 4.8 todas as

características da Figura 4.7 permanecem, modificando somente o algoritmo de substituição para o LFU.

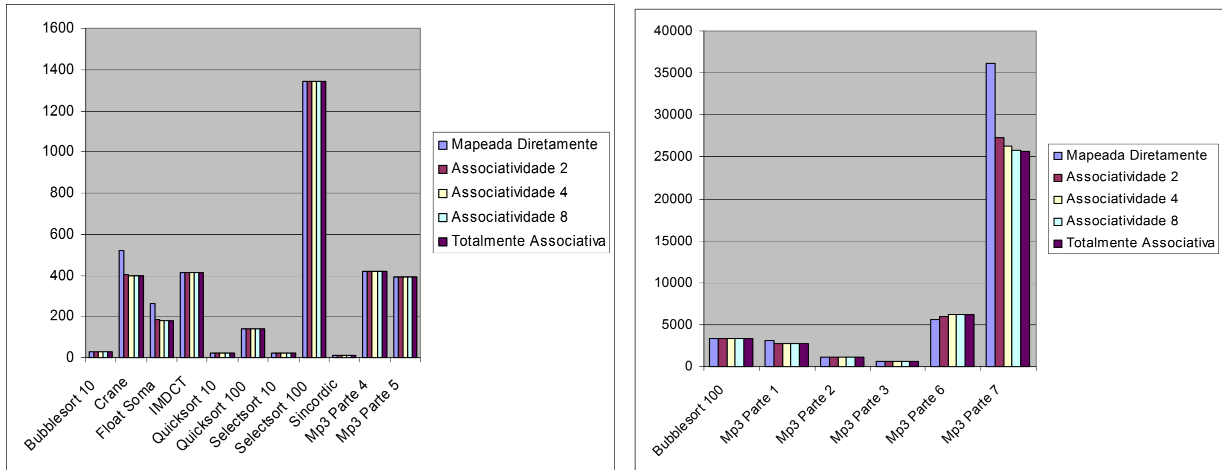

Figura 4.7: Ciclos perdidos por cache miss em cada algoritmo para diferentes associatividades usando o algoritmo de LRU, na cache de instruções.

Diferentemente da cache de dados, a Figura 4.7 ilustra que, em alguns algoritmos, há um rendimento melhor no crescimento do grau de associatividade, na cache de instruções, usando LRU. Um exemplo disto é o algoritmo de MP3 (Parte 7), que com uma cache mapeada diretamente foi executado em 3.607.731 ciclos. Já com uma associatividade 2 houve um decréscimo de 876.596 ciclos na execução e, posteriormente, quando fixada a associatividade 4, o numero de ciclos caiu para 2.624.756.

Em outros casos ilustrados na Figura 4.7, não houve nenhuma modificação no desempenho dos algoritmos com a inserção de associatividade, usando o LRU. Entre eles está o *bubblesort*, ordenando dez elementos, que permaneceu com 2.816 ciclos em todas as configurações de associatividade.

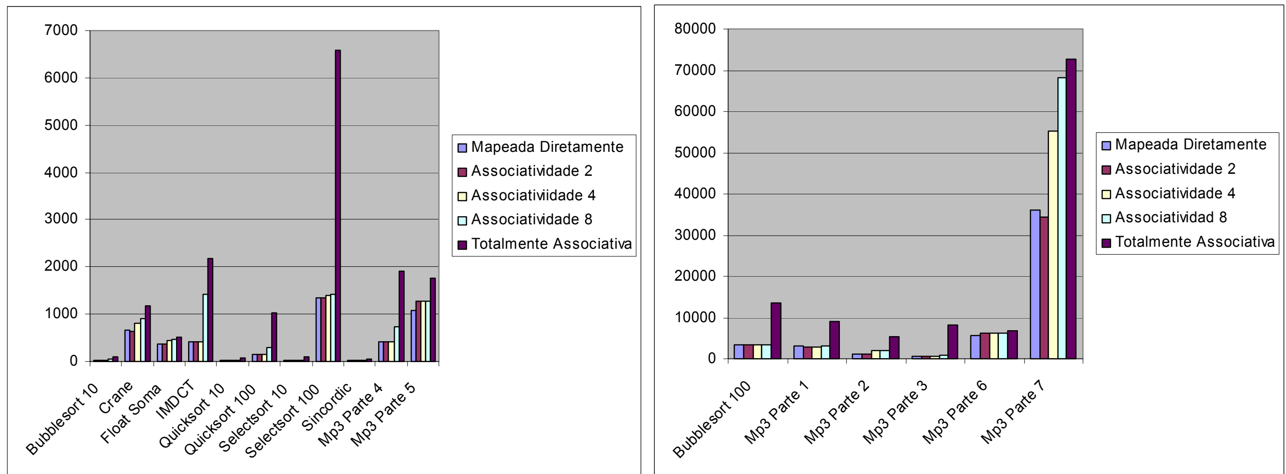

Figura 4.8: Ciclos perdidos por cache miss em cada algoritmo para diferentes associatividades usando o algoritmo de LFU, na cache de instruções.

A Figura 4.8 lustra o número de ciclos perdidos por cache miss em cada algoritmo, com associatividade variável, usando o algoritmo de substituição LFU.

Analisando a Figura 4.8 pode-se notar que, com o algoritmo LFU, nenhuma melhora com o aumento da associatividade em uma cache de instruções é obtida, em nenhum dos algoritmos simulados. Pelo contrário, o aumento da associatividade causa uma redução de desempenho. Por exemplo, o algoritmo Crane, em uma cache mapeada diretamente, é executado em 51.986 ciclos. Já com uma associatividade 8, o mesmo passa a ser executado em 61.898 ciclos, ou seja, uma redução de 19 % no desempenho do algoritmo.

Assim, nota-se que o aumento da associatividade, com o algoritmo LFU, simulado com os algoritmos escolhidos, não é uma boa escolha para aumentar o desempenho na cache de instruções.

4.4 Resultados de área da memória cache

Como já citado anteriormente, uma das preocupações em sistemas embarcados é a área ocupada pelo mesmo. Assim é importante analisar, quando da inserção de uma unidade em uma arquitetura voltada a sistemas

embarcados, a área que a mesma irá ocupar na arquitetura. Portanto, esta seção será voltada à análise dos resultados de área obtidos nas simulações para a arquitetura Femtojava *Low Power*. Nesta arquitetura todas as instruções possuem 8 bits e os dados 16 bits.

Cache de dados

Parametrização da localidade espacial

A Figura 4.9 ilustra a área ocupada, em células lógicas, na cache de dados variando a localidade espacial para diferentes tamanhos de memória cache. Percebe-se que o aumento da área da memória não é proporcional ao aumento da capacidade da memória, ou seja, a cache com tamanho de 64 entradas ocupa 3.918 células lógicas, enquanto a cache de 256 entradas ocupa 14513 células lógicas. Comparando as duas áreas tem-se um aumento de quase 3,7 vezes, sendo que o aumento da capacidade da cache é de 4 vezes. Isto ocorre pela economia de bits nos rótulos da cache, ou seja, quando se aumenta o número de entradas, diminui o número de bits para o rótulo, pelo fato de haver mais bits para indexar as entradas da cache.

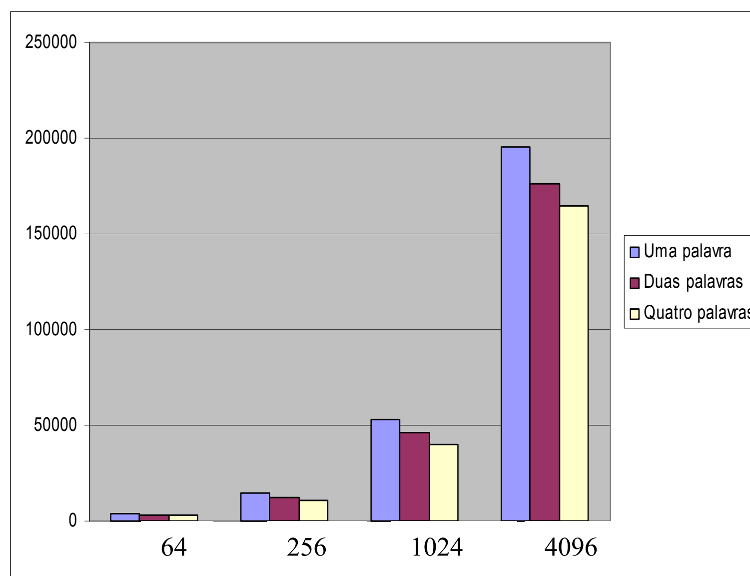

Figura 4.9: Área ocupada pela cache de dados, em células lógicas, parametrizando a localidade espacial.

Também pode ser observado na Figura 4.9 que, para um tamanho constante de memória e um aumento na localidade espacial da mesma, ocorre uma diminuição de área. Isto comprova o fato de se economizar nos rótulos utilizados para a comparação do endereço. Ou seja, para uma cache de uma palavra com tamanho 64, o número de bits para indexar as entradas da cache é maior do que a cache com o mesmo tamanho e quatro palavras por entrada. De fato, naquela existe a necessidade de indexar 64 entradas e nesta, somente 16. Assim, restam menos bits para o rótulo.

Parametrização da associatividade

A Figura 4.10 ilustra os resultados obtidos nas simulações de área para diferentes graus de associatividade, em uma cache de dados. O aumento do grau de associatividade para um mesmo tamanho de memória cache faz com que ocorra, também, um aumento na área ocupada pela memória. Isto se dá pelo fato de, quando o grau de associatividade é aumentado, um maior número de bits para o rótulo por bloco da cache é necessário.

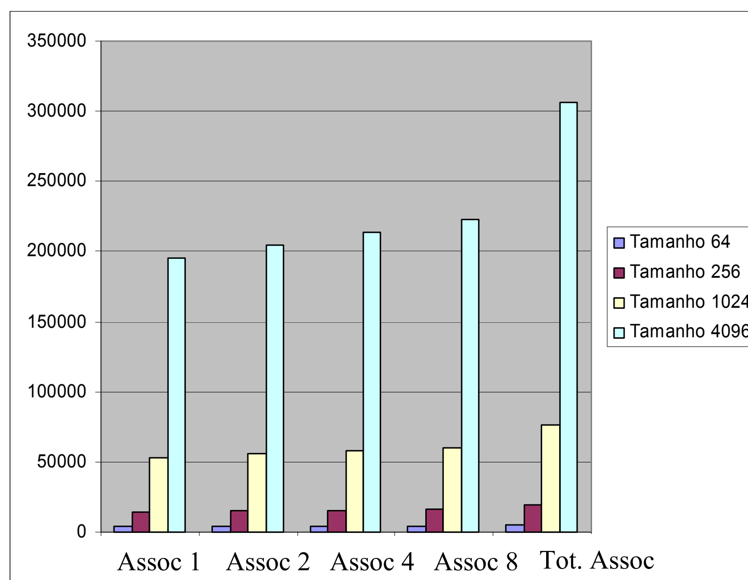

Figura 4.10: Área ocupada pela cache de dados, em células lógicas, parametrizando a associatividade.

Sendo assim, como pode ser observado na Figura 4.10, uma cache de dados de tamanho 64 mapeada diretamente (Associatividade 1) ocupa 3.919 células lógicas, enquanto a cache com o mesmo tamanho mas com associatividade 8 ocupa 4.354 células lógicas. Isto comprova que o aumento da associatividade causa um aumento na área da memória cache.

Cache de instruções

Parametrização da localidade espacial

A Figura 4.11 ilustra a área ocupada, em células lógicas, na cache de instruções variando a localidade espacial para diferentes tamanhos de memória cache.

Explicitada as características da cache de dados com diferentes valores de localidade espacial, a Figura 4.11 ilustra as mesmas peculiaridades que ocorrem com a cache de dados parametrizando a localidade espacial em uma cache de instruções. A única diferença entre as mesmas está na área ocupada para um mesmo tamanho nos dois tipos de cache. Ou seja, com um tamanho 256 entradas e duas palavras por entrada a cache de dados ocupa 12.192 LCs (células lógicas), enquanto a mesma configuração na cache de instruções ocupa 7.547 LCs. Justifica-se isto pelo fato de os dados na arquitetura Femtojava *Low Power* possuírem 16 bits, ao contrário das instruções que possuem somente 8 bits. Portanto, quanto maior é a palavra da cache, proporcionalmente, é menor o impacto do rótulo na área.

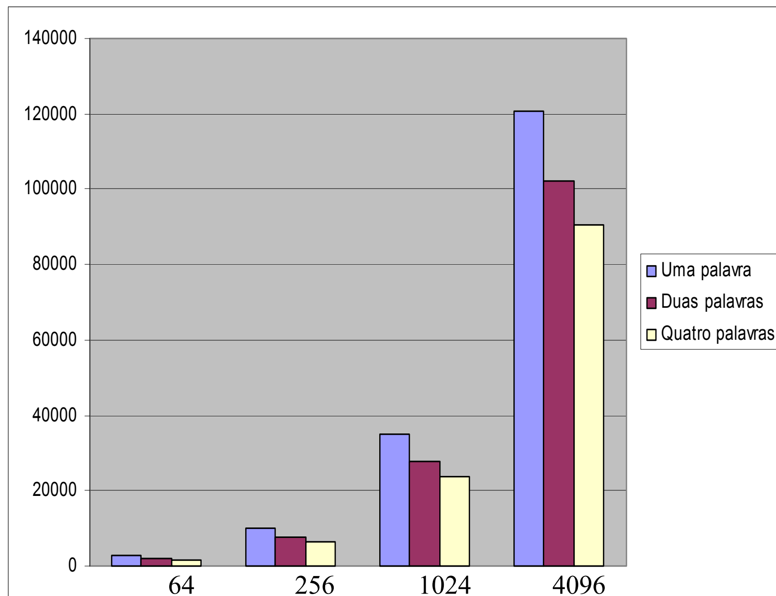

Figura 4.11: Área ocupada pela cache de instruções, em células lógicas, parametrizando a localidade espacial.

Parametrização da associatividade

A Figura 4.12 ilustra os resultados obtidos nas simulações de área para diferentes graus de associatividade, em uma cache de instruções.

Assim como na localidade espacial, as características da cache de dados e de instruções em relação à associatividade também são idênticas, como pode ser observado na Figura 4.12 e 4.11. Somente diferenciando uma da outra na área ocupada com o mesmo tamanho de memória cache, como já foi demonstrado anteriormente, pelo fato dos dados serem compostos de 16 bits, na arquitetura Femtojava *Low Power*, e as instruções de apenas 8 bits.

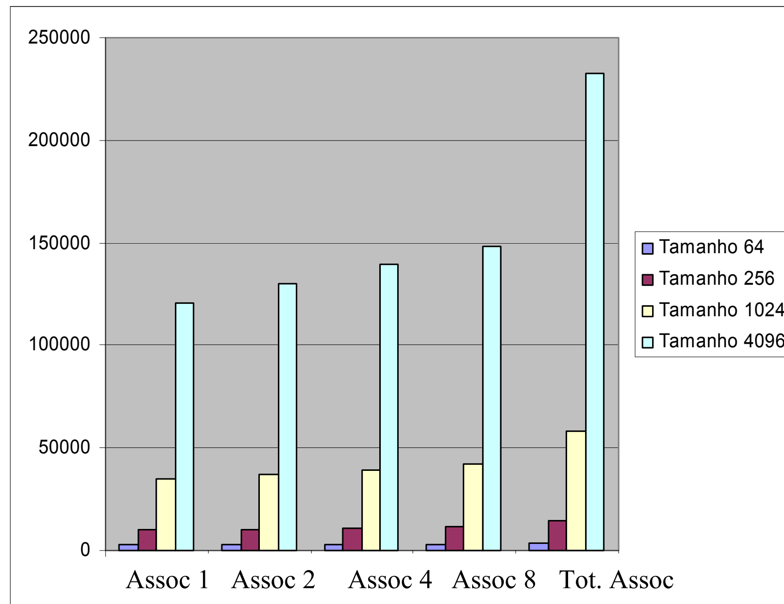

Figura 4.12: Área ocupada pela cache de instruções, em células lógicas, parametrizando a associatividade.

4.5 Resultados de potência da memória cache

Como acontece em todo sistema embarcado, os usuários desejam utilizar o mesmo seguidamente por muitas horas sem que devam recarregar a bateria. Portanto, um fator importante que se deve avaliar, quando da inserção de uma unidade em uma arquitetura voltada para sistemas embarcados, é a potência consumida pela mesma.

Assim esta seção trata de fazer esta avaliação parametrizando algumas características de cache para notar o impacto que as mesmas causam na potência consumida pela cache.

Cache de dados

Parametrização da localidade espacial

Na exploração da localidade espacial, alguns algoritmos vistos anteriormente fornecem um melhor desempenho. Todavia, ainda não é conhecido o consumo de potência no aumento do número de palavras por entrada.

Sendo assim, a Figura 4.13 ilustra a potência consumida, em chaveamento de portas, nos diferentes algoritmos, parametrizando a localidade espacial na cache de dados.

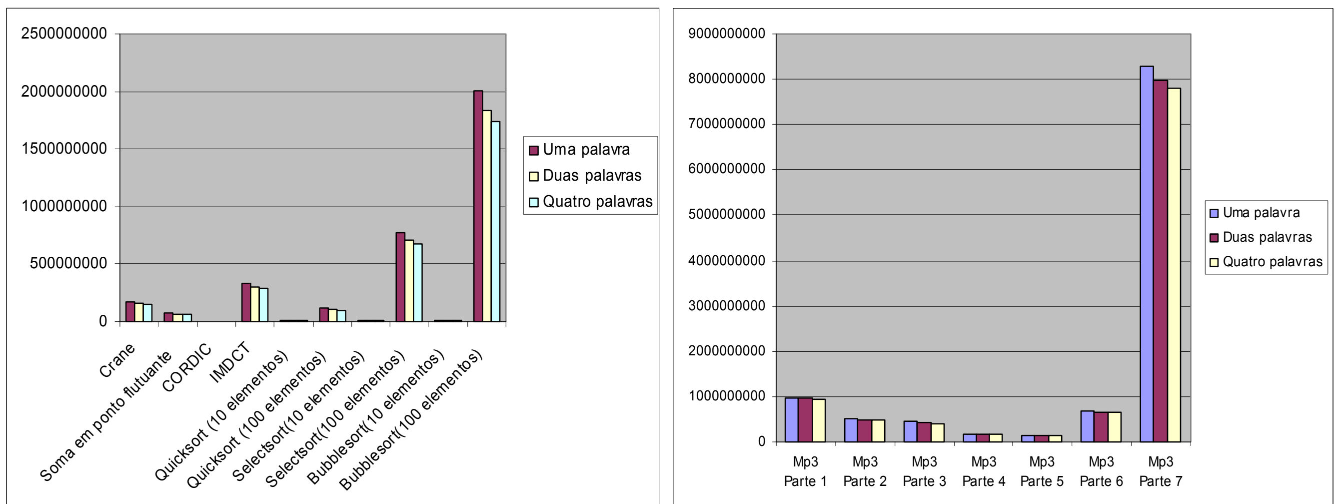

Figura 4.13: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, para diferentes localidades espaciais na cache de dados

Observa-se na Figura 4.13 que ao ocorrer um aumento na exploração da localidade espacial, em todos os algoritmos, há uma diminuição na potência consumida pelos mesmos. Isto se deve ao fato de que a potência consumida é diretamente proporcional à área ocupada pela cache.

Como já explicitado na seção de localidade espacial em área, há uma diminuição na área da cache ao ponto que a localidade espacial aumenta. Portanto, quando a exploração da localidade espacial cresce em uma cache, a área ocupada da mesma diminui, conseqüentemente, ocorre também a diminuição da potência consumida.

<u>Parametrização do tamanho da memória cache</u>

A capacidade de uma memória cache é um fator importante no desempenho da mesma, mas o equilíbrio entre o tamanho e o consumo de potência deve existir em sistemas embarcados.

Assim, como ilustra a Figura 4.14 com a cache de dados configurada em diferentes tamanhos, deve-se avaliar o consumo de potência nas

diversas configurações de tamanho da memória cache, observando o impacto que este tamanho causa na potência consumida.

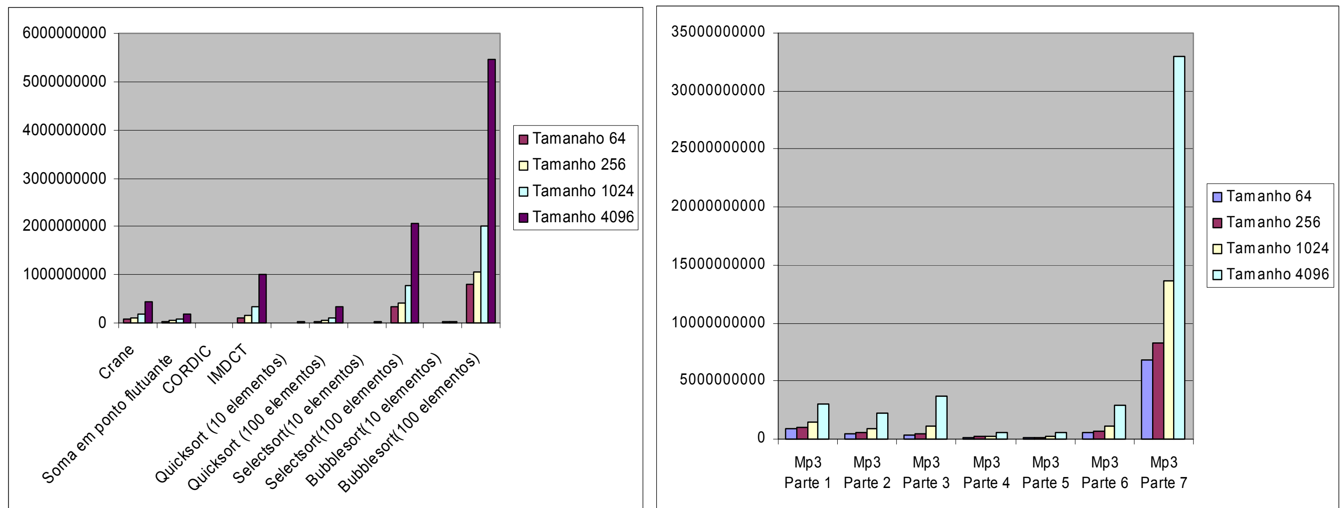

Figura 4.14: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, parametrizando o tamanho da cache de dados

Como pode ser observado nesta mesma figura, para um mesmo algoritmo há um crescimento no consumo de potência quando do aumento da capacidade da memória cache. Como já explicitado anteriormente, o consumo de uma memória cache é diretamente proporcional à área.

Sendo assim, para somente um acesso de qualquer algoritmo em uma cache de tamanho 256, é consumida maior potência que este mesmo acesso em uma cache de tamanho 64. Um exemplo é o algoritmo *bubblesort* ordenando 100 elementos: em uma cache com tamanho 1024 o consumo para executar este algoritmo é de 2.008.643.216 CGs. Já com tamanho 4096 o mesmo algoritmo consome 5.469.851.536 CGs, demonstrando que o crescimento do consumo esta diretamente ligado com o crescimento da área da cache.

Parametrização da associatividade

A associatividade é uma característica importante na memória cache. Deve-se assim verificar o impacto da mesma no consumo de potência, para

que se possa equilibrar o fator desempenho com o grau de associatividade inserido na cache.

A Figura 4.15 ilustra a simulação de consumo de potência para diversos graus de associatividade nos algoritmos utilizados em uma cache de dados, para um mesmo tamanho de cache.

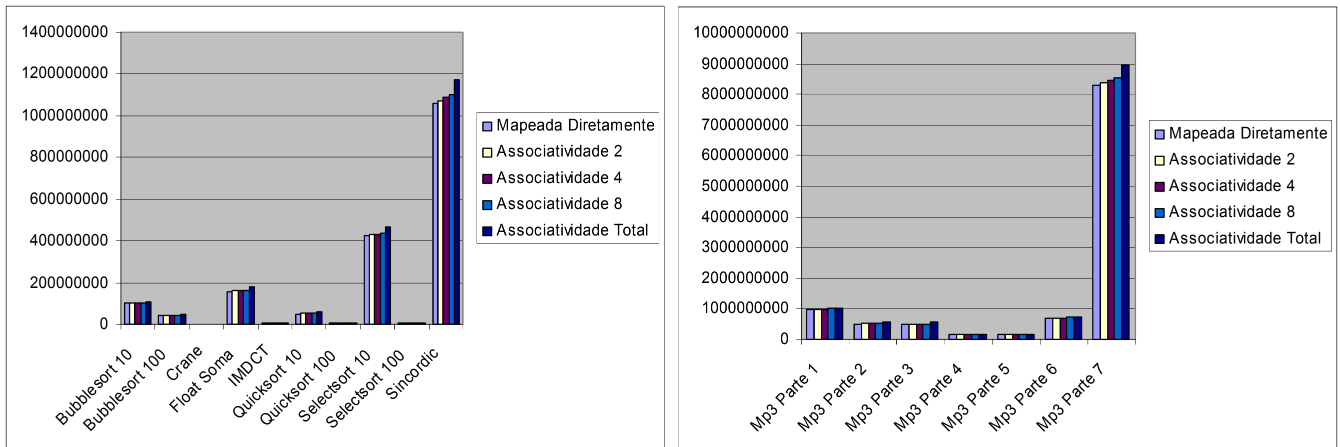

Figura 4.15: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, parametrizando a associatividade da cache de dados

Analisando a Figura 4.15, observa-se que a potência consumida, em todos os algoritmos, aumenta à medida que cresce o grau de associatividade. Similarmente à localidade espacial, ocorre na associatividade que o fato da potência consumida ser diretamente proporcional a área da cache, faz com que exista este gasto maior de potência em caches com associatividade maior. Como já explicitado anteriormente, a área de uma cache aumenta quando há uma elevação no grau de associatividade da mesma. Portanto, quando ocorre uma elevação no grau de associatividade a área ocupada pela cache se torna maior, conseqüentemente, aumenta a potência consumida pela mesma.

Cache de instruções

Parametrização da localidade espacial

Como foi observado anteriormente, o consumo de potência é diretamente proporcional à área ocupada pela cache de dados. Assim, não pode ser diferente com a cache de instruções, como ilustra a Figura 4.16.

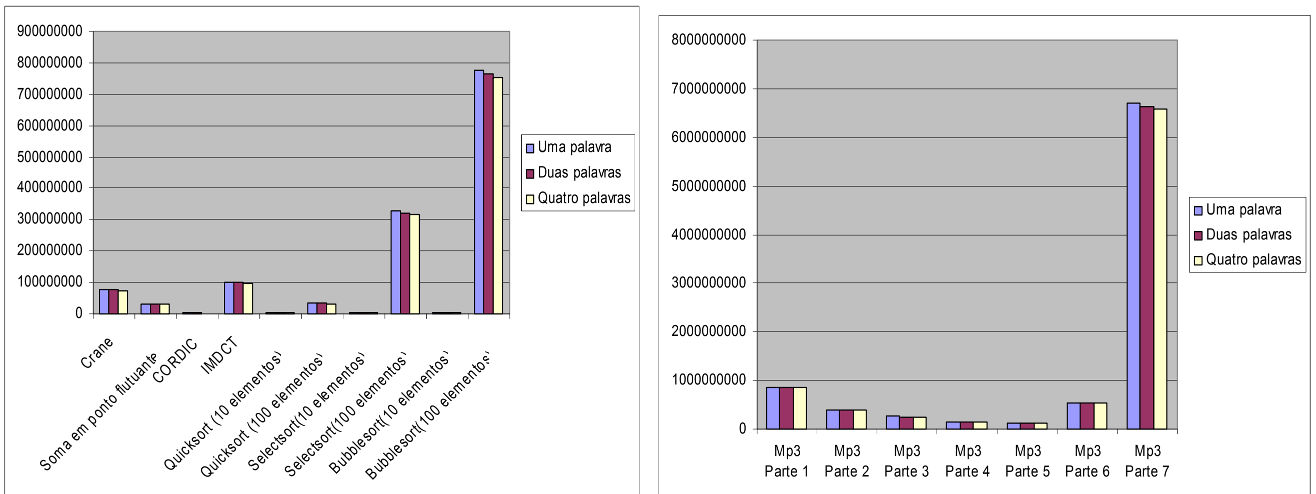

Figura 4.16: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, para diferentes localidades espaciais na cache de instruções

Observa-se na Figura 4.16 que, similarmente a cache de dados, o aumento da exploração da localidade espacial causa a diminuição no consumo de potência da cache. Ao aumentar-se o número de palavras por entrada na cache há uma redução na área ocupada e, conseqüentemente, produz-se uma redução no consumo de potência da mesma. Pelo motivo de os dados, na arquitetura Femtojava *Low Power*, serem de 16 bits e as instruções de 8 bits, para uma mesma configuração de cache, a cache de dados irá consumir mais potência do que a de instruções, pois aquela irá ocupar uma área maior.

Parametrização do tamanho da memória cache

Na mesma linha da cache de dados, a cache de instruções possui as mesmas características quando se observa o consumo de potência dos algoritmos na parametrização do tamanho da mesma. A Figura 4.17 ilustra

o consumo de potência dos algoritmos em diversos tamanhos em uma cache de instruções.

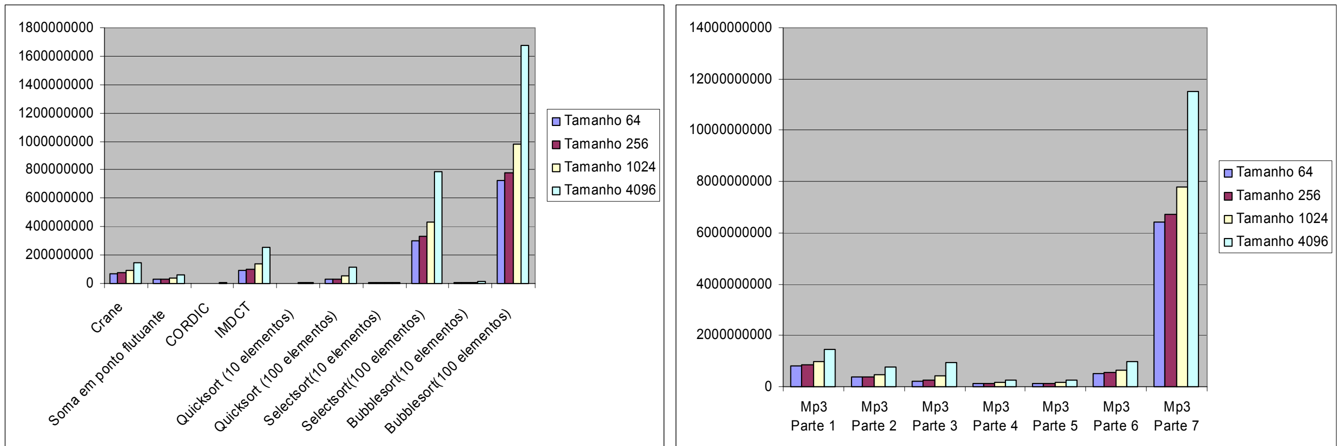

Figura 4.17: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, parametrizando o tamanho da cache de instruções

Como anteriormente observado na cache de dados, o consumo de potência é diretamente proporcional à área, fazendo com que em execuções de um mesmo algoritmo, em caches com diferentes tamanhos, ocorra o consumo desigual de potência. Nota-se na Figura 4.17, que a potência consumida está ligada diretamente ao tamanho da cache, pelo fato da mesma depender da área ocupada pela cache.

Parametrização da associatividade

As características de consumo de potência parametrizando a associatividade na cache de dados são válidas também para a cache de instruções, como pode ser observado na Figura 4.18. Esta ilustra o consumo de potência dos algoritmos utilizados variando a associatividade, na cache de instruções.

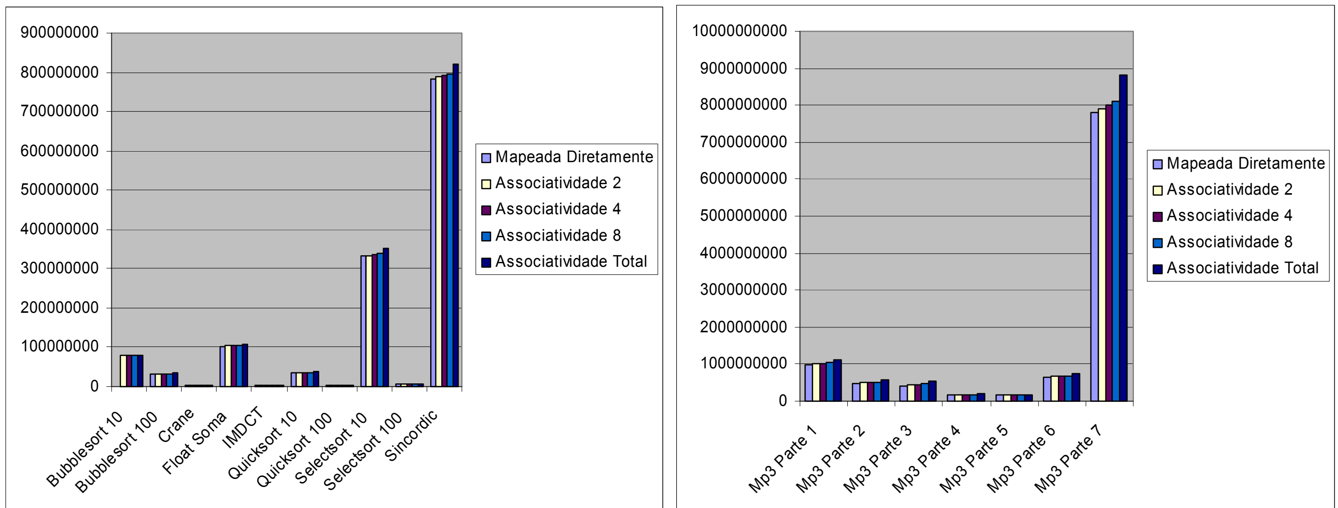

Figura 4.18: Consumo de potência, em chaveamento de portas (CG), nos algoritmos utilizados, parametrizando a associatividade da cache de instruções

Observa-se na Figura 4.18 que, similarmente a cache de dados, no aumento da associatividade ocorre também um aumento na potência consumida, pelo mesmo motivo citado anteriormente, ou seja, a potência é diretamente proporcional à área. Sendo que o aumento do grau de associatividade faz com que, por motivos já citados, a área da cache cresça e, conseqüentemente, o consumo de potência tenha uma elevação.

4.6 Soluções

A análise dos resultados permite determinar quais configurações de cache possuem um melhor desempenho. É evidente que uma configuração não irá satisfazer todos os algoritmos, pelo fato dos mesmos apresentarem características diferentes, então, será realizada a análise para cada algoritmo separadamente.

Tabela 4.2: Resultado dos melhores desempenhos para cada algoritmo

	Cache de dados				Cache de instruções			
	Localidade	Assoc. LRU	Assoc. LFU	Tamanho	Localidade	Assoc. LRU	Assoc. LFU	Tamanho
Bubblesort 10	1	Map. Diret.	Map. Diret.	64	2	Map. Diret.	Map. Diret.	64
Bubblesort 100	1	Map. Diret.	Map. Diret.	64	1	Map. Diret.	Map. Diret.	64
Crane	2	Map. Diret.	Map. Diret.	256	4	4	2	4096
Float Soma	1	Map. Diret.	Map. Diret.	1024	4	4	Map. Diret.	4096
IMDCT	4	Map. Diret.	Map. Diret.	1024	2	Map. Diret.	Map. Diret.	256
Quicksort 10	2	Map. Diret.	Map. Diret.	64	4	Map. Diret.	Map. Diret.	64
Quicksort 100	4	Map. Diret.	Map. Diret.	256	2	Map. Diret.	Map. Diret.	256
Selectsort 10	2	Map. Diret.	Map. Diret.	64	4	Map. Diret.	Map. Diret.	64
Selectsort 100	1	Map. Diret.	Map. Diret.	256	1	Map. Diret.	Map. Diret.	256
Sincordic	2	Map. Diret.	Map. Diret.	64	2	Map. Diret.	Map. Diret.	256
Mp3 Parte 1	4	Map. Diret.	Map. Diret.	64	2	2	2	4096
Mp3 Parte 2	1	Map. Diret.	Map. Diret.	4096	4	Map. Diret.	Map. Diret.	1024
Mp3 Parte 3	4	Map. Diret.	Map. Diret.	1024	4	Map. Diret.	Map. Diret.	1024
Mp3 Parte 4	4	Map. Diret.	Map. Diret.	1024	4	Map. Diret.	Map. Diret.	64
Mp3 Parte 5	4	Map. Diret.	Map. Diret.	1024	4	Map. Diret.	Map. Diret.	1024
Mp3 Parte 6	2	Map. Diret.	Map. Diret.	256	4	Map. Diret.	Map. Diret.	4096
Mp3 Parte 7	4	Map. Diret.	Map. Diret.	4096	4	Totalmente	2	4096

A Tabela 4.2 mostra os melhores resultados, observando somente o desempenho dos algoritmos utilizados.

Todos estes resultados foram obtidos sem levar em conta o consumo de potência e a área ocupada, ou seja, a Tabela 4.2 somente foi construída para observar-se o impacto das características da cache nos diferentes algoritmos.

A evidência da omissão da observação da área ocupada e potência dissipada se dá pelo motivo que cada sistema embarcado possui características diferentes, onde o projetista é que deve realizar o equilíbrio destes para cada sistema.

CAPÍTULO 5

CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho mostrou a exploração de espaço de projeto em sistemas embarcados baseados em Java através da análise de diferentes configurações de memória cache para o sistema. Utilizando-se o simulador construído pôde-se explorar diversas configurações de memória cache com algoritmos que são executados na arquitetura Femtojava *Low Power* e que pertencem ao domínio dos sistemas embarcados. Sendo assim, há a possibilidade de se realizar diversas simulações com inúmeras configurações de cache, analisá-las e separar as melhores alternativas para serem implementadas. Apesar do simulador somente ser utilizado no Femtojava *Low Power*, o mesmo é de propósito geral, desde que siga a nomenclatura de entrada explicada anteriormente.

Assim, com este simulador é possível produzir resultados com quaisquer algoritmos, em qualquer arquitetura e com diversas configurações de memória cache. Facilitando e agilizando a inserção de uma memória cache em uma arquitetura voltada à sistemas embarcados.

O capítulo 2 mostrou a potencialidade de pesquisa de memória cache em processadores que executam nativamente Java em sistemas embarcados, com a preocupação no desempenho, na área ocupada e a potência consumida dos mesmos. Posteriormente, algumas configurações de cache

em arquiteturas conhecidas foram apresentadas. Adicionalmente, o capítulo 2 descreveu alguns conceitos de cache, necessários para a compreensão na posterior utilização no simulador. O capítulo 3 caracterizou o simulador de memória cache implementado. Utilizando este simulador foram obtidos resultados com alguns algoritmos, avaliando-os e separando a melhor solução para cada um deles, como demonstrado no capítulo 4.

5.1 Trabalhos Futuros

5.1.1 Femtojava

Avaliação em outras versões do Femtojava

Neste trabalho, o simulador de memória cache somente foi testado na arquitetura Femtojava *Low Power*. Então futuramente, pretende-se fazer análise semelhante à feita neste trabalho nas versões Multiciclo e *VLIW* do processador Femtojava.

Na versão Multiciclo pretende-se observar o impacto no desempenho, pois esta versão tem toda a pilha alocada na memória principal. Na versão *VLIW* observar-se-á o comportamento da cache de instruções, pois a mesma seria muito mais requisitada devido ao tamanho do pacote de instruções.

Impacto da orientação a objetos na cache

No futuro o simulador será utilizado para medir o impacto que as instruções adicionais para a alocação e controle de objetos, além do coletor de lixo, para Java, custam em termos de desempenho e potência. Além do mais, políticas de caches especiais projetadas especificamente visando a alocação de objetos podem ser construídas.

5.1.2 Expansões no simulador

Algoritmos de Substituição

Neste estudo foram pesquisados alguns algoritmos utilizados para realizar a substituição de dados na cache quando existe o uso da associatividade. Porém, foram implementados no simulador somente os algoritmos LRU e LFU, por apresentarem características totalmente diferentes.

Assim, pode-se incorporar ao simulador mais alguns algoritmos de substituição com o objetivo de explorar a utilização da associatividade na cache, verificando assim se existe um algoritmo que ofereça resultados melhores do que os atualmente implementados.

Write-Back e Write-Through

A política de cópia de dados à memória principal (*write-back* e *write-through*) é mais um parâmetro a ser inserido neste estudo. A implementação destas políticas no simulador é uma boa exploração para trabalhos, principalmente na diminuição da potência consumida na arquitetura.

Implementação no CACO-PS

Com o simulador CACO-PS, como já citado anteriormente, é possível descrever estruturalmente, em diferentes níveis de abstração, qualquer arquitetura.

Então, pode-se escolher qualquer configuração de cache desejada e implementá-la neste simulador, a fim de simular, de forma real, a potência consumida pela cache.

Além do mais, a maneira com que o simulador foi construído facilita a integração do mesmo com o simulador CACO-PS.

Implementação em VHDL

Como as versões do Femtojava já estão descritas em VHDL, outra proposta de trabalho seria descrever a cache nesta linguagem, para obter-se

uma simulação real da mesma na arquitetura Femtojava e permitir uma futura prototipação.

Referências

[ARM 2003] ARM, Limited. High Performance, Jazelle-enhanced Macrocell, <http://www.arm.com/products/CPUs/ARM1026EJS.html>,2003.

[BECK 2003] A. C. Beck F., F. R. Wagner, L. Carro, "CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator". *SBCCI'03 – 16th Symposium on Integrated Circuits and Systems Design*. São Paulo, Brazil, September 2003

[BECK 2003b] Beck, A.C.S., Carro, L "A VLIW Low Power Java Processor for Embedded Applications". December (2003)

[HANGAL 1999] S. Hangal, J. M. O'Connor, "Performance Analysis and Validation of the picoJava Processor". *IEEE Micro*, vol. 9, no. 3: pp. 66-72, 1999

[ITO 2001] S. Ito, L. Carro, R. Jacobi, "Making Java Work for Microcontroller Applications". *IEEE Design & Test*, vol. 18, no. 5, Set-Oct. 2001, pp. 100-110

[KRALL 1998] A. Krall, "Efficient JavaVM Just-in-Time Compilation ", *Proceedings of PACT'98*, 1998

[LAWTON 2002] G. Lawton, "Moving Java into Mobile Phones", *Computer*, vol. 35, n. 6, 2002, pp. 17-20

[MCGHAN 1998] H. McGhan, M. O'Connor, "PicoJava: A Direct Execution Engine For Java Bytecode". *IEEE Computer vol. 31*, no. 10, pp. 22-30, 1998

[MOSER 1999] Moser, E., Nebel, W. "Case Study: System Model of Crane and Embedded Control". In: *Proceedings of Design, Automation and Test in Europe* (DATE), Munich, Alemanha, Março, 1999.

[NICOLAU 2003] Dan Nicolaescu, Alexander V. Veidenbaum, Alexandru Nicolau: "Reducing Power Consumption for High-Associativity Data Caches in Embedded Processors". *DATE 2003*

[O'CONNOR 97] O'Connor, J. M., Tremblat, M. Picojava-I: the Java Virtual Machine in Hardware. In *IEEE Micro*, vol. 17, n. 2, Mar-Apr. 1997, 45-53

[PATTERSON 2003] Hennessy, J. L., Patterson, D. A. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 3th edition, 2003

[SCHLETT 1998] Schlett, M. Trends in Embedded-Microprocessor Design. In *Computer*, vol. 31, n. 8, 1998, 44–49

[SIMUNIÉ 1999] T. Simunié, L. Benini, G. De Micheli. "Cycle-Accurate Simulation of Energy Comsumption in Embedded Systems". *Design Automation Conference (DAC),* ACM, 1999

[TOM 2004] Tom's Hardware Guide Processors. AMD Sempron- AMD Sempron is the new Duron, <http://www.tomshardware.com/cpu/20040728/sempron-01.html>, 2004.

[VOLDER 1959] J. Volder, "The CORDIC trigonometric computing technique", *IRE Trans. Electron. Comput.*, v. EC-8, n. 3, Sept. 1959, pp. 330-334

[ZHANG 2003] C. Zhang, F. Vahid, and W. Najjar, "A Highly-Configurable Cache Architecture for Embedded Systems," Int. *Symp. on Computer Architecture*, 2003

Anexo I- Tabela de Resultados

Número de ciclos perdidos por cache miss na cache de instruções(x 100)

	Tamanho 64																													
	1 Palavra										2 Palavras										4 Palavras									
	Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total		Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total		Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total	
	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
Bubblesort 10	28,16	28,16	28,16	41,46	28,16	59,03	28,16	94,17	28,16	102,9	28,02	28,02	27,95	28,51	28,37	28,58	26,76	29,14	26,76	43,7	27,25	27,25	27,18	27,25	27,6	27,39	29,07	27,53	31,8	27,25
Bubblesort 100	3402	3402	3402	4793	3402	7948	3402	12591	3402	13608	3414	3414	3419	3416	3429	3416	3400	3423	3400	5132	3414	3414	3418	3414	3428	3414	3453	3414	3496	3414
Crane	1005	1005	1009	924,8	994,6	989,4	996,4	1089	1021	1165	911,2	911,2	898,8	840,8	881,3	838,2	890,8	872,9	891,1	904	721,3	721,3	702,1	702,2	700,3	687,6	697,7	698,6	700,3	700,5
Float Soma	481,2	481,2	489,9	470,7	492,4	489,5	492,4	510,1	492,4	516,9	398,6	398,6	399,7	390,7	399,7	392,1	399,7	382,1	399,7	393,1	309,5	309,5	307,4	309,3	309,1	312,2	309,1	309,1	309,1	310,9
IMDCT	520,3	520,3	560,3	1461	444,3	2027	433,8	2178	414,8	2187	524,4	524,4	573,7	1152	651,7	1309	799,2	1317	483,9	1372	594,9	594,9	681,2	678,4	835	745,2	1139	793	1139	737,4
Quicksort 10	36,16	36,16	36,16	37,07	35,74	45,75	36,09	58,42	40,22	63,53	34,34	34,34	35,88	33,71	36,79	31,96	39,24	32,8	38,19	36,93	27,27	27,27	29,16	27,48	29,79	26,78	31,12	27,83	32,59	27,69
Quicksort 100	327,8	327,8	348,8	425	337,8	572,1	346,9	887,9	438,8	1033	336,4	336,4	362,6	367,1	369,4	356,5	380,1	359,2	379,6	486	279	279	307,5	271,3	310,7	267,1	343,6	271,1	399,2	269,2
Selectsort 10	28,47	28,47	27,28	29,73	23,92	49,89	23,92	79,43	23,92	89,58	27,14	27,14	29,38	27,7	33,86	41,84	28,26	44,29	32,32	50,87	24,83	24,83	26,51	24,83	29,87	24,83	29,59	25,53	31,34	25,46
Selectsort 100	1386	1386	1366	1428	1372	3609	1345	5457	1345	6590	1385	1385	1385	1371	1371	1385	1368	2893	1344	2980	1370	1370	1384	1370	1405	1370	1425	1370	1456	1370
Sincordic	23,72	23,72	16,23	22,95	13,85	34,01	13,71	38,56	13,71	38,56	21,69	21,69	16,51	21,76	15,53	26,52	13,78	27,85	14,13	27,85	15,95	15,95	13,22	13,5	13,15	14,27	11,4	13,43	11,4	14,62
Mp3 Parte 1	4619	4619	4742	5185	4787	6352	4922	8074	4937	8994	4201	4201	4381	4463	4391	5201	4419	6035	4379	6539	3726	3726	3761	3810	3790	3940	3805	4411	3810	4448
Mp3 Parte 2	3016	3016	3041	4362	3189	5075	3353	5404	4134	5404	3236	3236	3407	3478	3517	3390	3517	3371	3517	3299	2437	2437	2517	2442	2521	2473	2530	2481	2511	2555
Mp3 Parte 3	3180	3180	3275	4719	2983	6154	3424	8020	3862	8183	3375	3375	3702	4157	3784	4258	3860	4598	3777	4763	2422	2422	2598	2794	2688	2823	2827	2836	2827	2943
Mp3 Parte 4	420,1	420,1	420,1	742,4	420,1	1386	420,1	1789	420,1	1910	419,4	419,4	419,4	580,7	419,4	1225	419,4	1386	419,4	1386	418,3	418,3	418,3	418,5	418,3	699,9	418,3	700,2	418,3	659,9
Mp3 Parte 5	1276	1276	1276	1276	1276	1517	1276	1719	1276	1759	1006	1006	1006	1006	1006	1167	1006	1288	1006	1288	701,6	701,6	701,6	701,6	701,6	701,6	701,6	782,2	701,6	822,5
Mp3 Parte 6	6194	6194	6194	6355	6194	6596	6194	6717	6194	6798	4928	4928	4928	5169	4928	5250	4928	5330	4928	5371	3444	3444	3444	3525	3444	3566	3444	3606	3444	3646
Mp3 Parte 7	69087	69087	69114	69131	69123	70480	69131	71614	69131	72642	53389	53389	53399	53391	53399	54013	53399	55146	53399	55562	38225	38225	38225	38225	38225	38301	38225	38462	38225	38930

Tamanho 256																													
1 Palavra										2 Palavras										4 Palavras									
Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total		Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total		Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total	
LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
28,16	28,16	28,16	28,16	28,16	28,16	28,16	41,46	28,16	102,9	26,76	26,76	26,76	26,76	26,76	28,02	26,76	28,51	26,76	43,7	25,85	25,85	25,85	25,92	25,85	26,13	25,85	26,13	25,85	27,25
3402	3402	3402	3402	3402	3402	3402	4793	3402	13608	3400	3400	3400	3400	3400	3414	3400	3416	3400	5132	3400	3400	3400	3400	3400	3400	3400	3400	3400	3414
660,3	660,3	546,1	644,6	541,1	799,1	463,5	909,6	404,6	1165	651,5	651,5	626,4	625,1	647,7	636,1	689,5	691,4	737,4	804,5	574,3	574,3	572,2	540,4	595,3	536,4	609,3	532,3	613,2	550,2
355,2	355,2	318,5	359,2	271	445,6	253,5	470,7	269,1	516,9	332,1	332,1	315,6	305,5	301,9	310,9	301,2	313,5	278,2	317,7	279,5	279,5	271,7	257,7	273,6	247	290,5	241,9	305,4	222,8
414,8	414,8	414,8	414,8	414,8	419,8	414,8	1409	414,8	2187	412,4	412,4	412,4	412,4	412,4	499,2	412,4	1106	412,4	1372	409	409	409	409	409	416,1	409	488,9	409	621,5
21,11	21,11	21,11	21,11	21,11	21,81	21,11	30,63	21,11	63,53	19,36	19,36	19,36	19,36	19,36	19,85	19,36	24,12	19,36	36,93	17,54	17,54	17,54	17,61	17,54	17,61	17,54	17,68	17,54	17,89
138,3	138,3	138,3	138,3	138,3	144,9	138,3	284,6	138,3	1033	136,6	136,6	136,6	136,6	136,6	137,2	136,6	184	136,6	486	134,8	134,8	134,8	134,8	134,8	134,8	134,8	134,9	134,8	140,7
23,92	23,92	23,92	23,92	23,92	28,47	23,92	29,73	23,92	89,58	22,59	22,59	22,59	22,59	22,59	23,71	22,59	24,27	22,59	50,87	21,26	21,26	21,26	21,33	21,26	21,4	21,26	21,4	21,26	21,4
1345	1345	1345	1345	1345	1386	1345	1428	1345	6590	1344	1344	1344	1344	1344	1344	1344	1358	1344	2980	1343	1343	1343	1343	1343	1343	1343	1343	1343	1357
13,71	13,71	13,71	13,71	13,71	19,03	13,71	22,95	13,71	38,56	12,45	12,45	12,45	12,45	12,45	15,88	12,45	20,5	12,45	27,85	10,35	10,35	10,35	10,56	10,35	10,63	10,35	10,84	10,35	10,91
3464	3464	2877	3323	2852	3909	2840	4736	2825	8994	3327	3327	2792	3150	2793	3249	2776	3621	2762	6062	3169	3169	2685	2997	2694	3022	2695	3014	2706	3705
1960	1960	1130	2013	1107	2401	1107	4353	1107	5404	2168	2168	1171	1794	1119	2160	1104	2978	1104	3274	1808	1808	1155	1162	1113	1118	1100	1196	1100	1375
1149	1149	798,3	1192	680,3	2544	680,3	4337	680,3	8183	1302	1302	926,7	1407	672	1887	672	2523	671,9	4342	1064	1064	916	1084	661,4	903,5	661,4	1084	661,5	1456
420,1	420,1	420,1	420,1	420,1	420,1	420,1	742,4	420,1	1910	419,4	419,4	419,4	419,4	419,4	419,4	419,4	580,7	419,4	1386	418,2	418,2	418,2	418,3	418,2	418,3	418,2	418,4	418,2	659,9
1078	1078	1185	1269	1236	1276	1276	1276	1276	1759	993,4	993,4	1006	943	1006	913,6	1006	932,5	1006	1028	701,6	701,6	701,6	699,5	701,6	701,6	701,6	701,6	701,6	822,5
6194	6194	6194	6194	6194	6194	6194	6355	6194	6798	4928	4928	4928	4928	4928	4928	4928	5169	4928	5371	3444	3444	3444	3444	3444	3444	3444	3525	3444	3646
64715	64715	67375	68301	68718	69087	68912	69131	68912	72642	52922	52922	53253	51465	53253	52066	53253	53313	53253	55487	38140	38140	38140	38156	38140	38166	38140	38166	38140	38872

Tamanho 1024																													
1 Palavra										2 Palavras										4 Palavras									
Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total		Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total		Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total	
LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
28,16	28,16	28,16	28,16	28,16	28,16	28,16	28,16	28,16	102,9	26,76	26,76	26,76	26,76	26,76	26,76	26,76	26,76	26,76	43,7	25,85	25,85	25,85	25,85	25,85	25,85	25,85	25,92	25,85	27,25
3402	3402	3402	3402	3402	3402	3402	3402	3402	13608	3400	3400	3400	3400	3400	3400	3400	3400	3400	5132	3400	3400	3400	3400	3400	3400	3400	3400	3400	3414
519,9	519,9	400	428,2	399,7	487,9	399,7	619	399,7	1165	539,1	539,1	401,4	409,1	392,5	443,8	392,5	513,8	392,5	804,5	491,2	491,2	397,2	394,5	382,9	394,5	382,9	408,2	382,8	458,8
263,6	263,6	185,5	230,9	180,7	290,7	180,2	350,6	180,2	516,9	244,7	244,7	179,1	196,8	172,9	227,5	172,2	262,3	172,2	317,7	221	221	172,6	171,5	163,1	172,1	162,3	178,5	162,2	197,3
414,8	414,8	414,8	414,8	414,8	414,8	414,8	414,8	414,8	2187	412,4	412,4	412,4	412,4	412,4	412,4	412,4	412,4	412,4	1372	408,9	408,9	408,9	408,9	408,9	409	408,9	409	408,9	621,5
21,11	21,11	21,11	21,11	21,11	21,11	21,11	21,11	21,11	63,53	19,36	19,36	19,36	19,36	19,36	19,36	19,36	19,36	19,36	36,93	17,54	17,54	17,54	17,54	17,54	17,54	17,54	17,61	17,54	17,89
138,3	138,3	138,3	138,3	138,3	138,3	138,3	138,3	138,3	1033	136,6	136,6	136,6	136,6	136,6	136,6	136,6	136,6	136,6	486	134,8	134,8	134,8	134,8	134,8	134,8	134,8	134,8	134,8	140,7
23,92	23,92	23,92	23,92	23,92	23,92	23,92	23,92	23,92	89,58	22,59	22,59	22,59	22,59	22,59	22,59	22,59	22,59	22,59	50,87	21,26	21,26	21,26	21,26	21,26	21,26	21,26	21,33	21,26	21,4
1345	1345	1345	1345	1345	1345	1345	1345	1345	6590	1344	1344	1344	1344	1344	1344	1344	1344	1344	2980	1343	1343	1343	1343	1343	1343	1343	1343	1343	1357
13,71	13,71	13,71	13,71	13,71	13,71	13,71	13,71	13,71	38,56	12,45	12,45	12,45	12,45	12,45	12,45	12,45	12,45	12,45	27,85	10,35	10,35	10,35	10,35	10,35	10,35	10,35	10,56	10,35	10,91
3161	3161	2752	2755	2749	2939	2749	2975	2727	8994	3006	3006	2692	2687	2692	2774	2697	2779	2690	5958	2889	2889	2602	2605	2603	2630	2605	2614	2611	3424
1107	1107	1107	1107	1107	1925	1107	2008	1107	5404	1104	1104	1104	1104	1104	1676	1104	1772	1104	3274	1100	1100	1100	1100	1100	1100	1100	1100	1100	1375
680,3	680,3	680,3	680,3	680,3	686,3	680,3	778,7	680,3	8183	671,9	671,9	671,9	671,9	671,9	683,9	671,9	703,7	671,9	4342	661,1	661,1	661,1	661,3	661,2	661,3	661,1	661,5	661,1	1295
420,1	420,1	420,1	420,1	420,1	420,1	420,1	420,1	420,1	1910	419,4	419,4	419,4	419,4	419,4	419,4	419,4	419,4	419,4	1386	418,2	418,2	418,2	418,2	418,2	418,2	418,2	418,3	418,2	659,9
391,4	391,4	391,4	597,3	391,4	1078	391,4	1269	391,4	1759	382,1	382,1	382,1	467,2	382,1	623,8	382,1	733,4	382,1	1028	371,6	371,6	371,6	371,8	371,6	371,8	371,6	371,8	371,6	499
5625	5625	5992	6165	6178	6194	6194	6194	6194	6798	4900	4900	4923	4722	4928	4674	4928	4644	4928	5371	3442	3442	3442	3444	3444	3444	3444	3444	3444	3646
36077	36077	27311	34356	26248	55320	25738	68270	25729	72642	34022	34022	26903	30928	25208	35864	24149	39115	24054	41363	28043	28043	23856	24577	22867	24857	21666	25068	21366	25390

Tamanho 4096																													
1 Palavra										2 Palavras										4 Palavras									
Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total		Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total		Assoc 1		Assoc 2		Assoc 4		Assoc 8		Total	
LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
28,16	28,16	28,16	28,16	28,16	28,16	28,16	28,16	28,16	102,9	26,76	26,76	26,76	26,76	26,76	26,76	26,76	26,76	26,76	43,7	25,85	25,85	25,85	25,85	25,85	25,85	25,85	25,85	25,85	27,25
3402	3402	3402	3402	3402	3402	3402	3402	3402	13608	3400	3400	3400	3400	3400	3400	3400	3400	3400	5132	3400	3400	3400	3400	3400	3400	3400	3400	3400	3414
412,7	412,7	399,7	399,7	399,7	416,7	399,7	427,6	399,7	1165	409,5	409,5	392,5	392,5	392,5	397,4	392,5	408,6	392,5	804,5	393,7	393,7	382,8	382,8	382,8	382,9	382,8	388,7	382,8	458,8
180,2	180,2	180,2	180,2	180,2	224	180,2	229,8	180,2	516,9	172,2	172,2	172,2	172,2	172,2	186,8	172,2	193,8	172,2	317,7	162,2	162,2	162,2	162,2	162,2	162,7	162,2	163,1	162,2	197,3
414,8	414,8	414,8	414,8	414,8	414,8	414,8	414,8	414,8	2187	412,4	412,4	412,4	412,4	412,4	412,4	412,4	412,4	412,4	1372	408,9	408,9	408,9	408,9	408,9	408,9	408,9	408,9	408,9	621,5
21,11	21,11	21,11	21,11	21,11	21,11	21,11	21,11	21,11	63,53	19,36	19,36	19,36	19,36	19,36	19,36	19,36	19,36	19,36	36,93	17,54	17,54	17,54	17,54	17,54	17,54	17,54	17,54	17,54	17,89
138,3	138,3	138,3	138,3	138,3	138,3	138,3	138,3	138,3	1033	136,6	136,6	136,6	136,6	136,6	136,6	136,6	136,6	136,6	486	134,8	134,8	134,8	134,8	134,8	134,8	134,8	134,8	134,8	140,7
23,92	23,92	23,92	23,92	23,92	23,92	23,92	23,92	23,92	89,58	22,59	22,59	22,59	22,59	22,59	22,59	22,59	22,59	22,59	50,87	21,26	21,26	21,26	21,26	21,26	21,26	21,26	21,26	21,26	21,4
1345	1345	1345	1345	1345	1345	1345	1345	1345	6590	1344	1344	1344	1344	1344	1344	1344	1344	1344	2980	1343	1343	1343	1343	1343	1343	1343	1343	1343	1357
13,71	13,71	13,71	13,71	13,71	13,71	13,71	13,71	13,71	38,56	12,45	12,45	12,45	12,45	12,45	12,45	12,45	12,45	12,45	27,85	10,35	10,35	10,35	10,35	10,35	10,35	10,35	10,35	10,35	10,91
2655	2655	2624	2661	2606	2690	2596	2722	2594	8994	2615	2615	2591	2608	2574	2606	2561	2614	2551	5955	2555	2555	2540	2545	2532	2533	2526	2524	2509	3351
1107	1107	1107	1107	1107	1107	1107	1107	1107	5404	1104	1104	1104	1104	1104	1104	1104	1104	1104	3274	1100	1100	1100	1100	1100	1100	1100	1100	1100	1375
680,3	680,3	680,3	680,3	680,3	680,3	680,3	680,3	680,3	8183	671,9	671,9	671,9	671,9	671,9	671,9	671,9	671,9	671,9	4342	661,1	661,1	661,1	661,1	661,1	661,1	661,1	661,3	661,1	1295
420,1	420,1	420,1	420,1	420,1	420,1	420,1	420,1	420,1	1910	419,4	419,4	419,4	419,4	419,4	419,4	419,4	419,4	419,4	1386	418,2	418,2	418,2	418,2	418,2	418,2	418,2	418,2	418,2	659,9
391,4	391,4	391,4	391,4	391,4	391,4	391,4	597,3	391,4	1759	382,1	382,1	382,1	382,1	382,1	382,1	382,1	467,2	382,1	1028	371,6	371,6	371,6	371,6	371,6	371,6	371,6	371,8	371,6	499
3672	3672	1736	3876	1736	5625	1736	6165	1736	6798	3719	3719	1697	2520	1697	3339	1697	3691	1697	4145	2869	2869	1649	1651	1649	1670	1649	1683	1649	1973
23345	23345	21005	23407	20231	25446	19705	28033	18844	72642	23462	23462	21961	23413	22235	23655	22894	24276	24054	40295	21624	21624	20607	21059	20820	21012	21243	21011	21364	22883

Número de ciclos perdidos por cache miss na cache de dados

	Tamanho 64									
	1 Palavra									
	Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.	
	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
Bubblesort 10	3887	3887	3901	3901	3901	3901	3922	3915	3929	3915
Bubblesort 100	603326	603326	618817	615219	626097	611425	623717	610459	623878	610613
Crane	74246	74246	76556	76563	76913	73903	76577	73742	77424	75219
Float Soma	32255	32255	32325	32269	33137	32038	34236	32199	34824	32241
IMDCT	95410	95410	95445	94878	96054	94885	94885	94885	94885	94892
Quicksort 10	2615	2615	2692	2664	2692	2629	2692	2657	2678	2685
Quicksort 100	29276	29276	29857	29507	30879	29731	30998	29773	31096	29850
Selectsort 10	2644	2644	2658	2658	2658	2658	2665	2658	2658	2658
Selectsort 100	152178	152178	155202	152360	159332	154558	162279	153641	164785	156231
Sincordic	1119	1119	1119	1161	1119	1161	1119	1168	1119	1168
Mp3 Parte 1	525576	525576	521782	518499	529251	516350	523630	515573	530406	518898
Mp3 Parte 2	181776	181776	175098	170933	176029	166068	168378	165305	164878	164080
Mp3 Parte 3	234028	234028	228141	223717	231690	220077	220889	218551	217340	217193
Mp3 Parte 4	71190	71190	70707	70595	71078	70343	70336	70336	70336	70336
Mp3 Parte 5	63216	63216	63580	62005	65169	61739	64238	61718	65078	61718
Mp3 Parte 6	336670	336670	337888	332428	341822	332246	340338	332267	341906	332211
Mp3 Parte 7	3622165	3622165	3521386	3575174	3541693	3580305	3510564	3581173	3618105	3582790

	Tamanho 64									
	2 Palavras									
	Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.	
	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
Bubblesort 10	3838	3838	3845	3845	3845	3845	3866	3852	3922	3852
Bubblesort 100	602549	602549	607176	602290	613728	600316	606651	600218	606714	600533
Crane	74617	74617	76346	76360	76822	73973	77809	73525	79265	74939
Float Soma	32269	32269	32409	32255	32934	31975	34362	32087	34803	32101
IMDCT	92323	92323	92197	91196	93597	91224	91056	90972	90783	90790
Quicksort 10	2580	2580	2629	2608	2629	2594	2678	2608	2650	2615
Quicksort 100	28723	28723	28912	28583	29633	28772	29871	28723	29724	28688
Selectsort 10	2602	2602	2609	2609	2609	2609	2623	2609	2616	2609
Selectsort 100	149581	149581	148160	146711	150379	148797	151940	148748	153774	151065
Sincordic	1070	1070	1070	1091	1070	1091	1077	1098	1077	1098
Mp3 Parte 1	518107	518107	509840	509420	518268	506025	526038	507019	528159	505766
Mp3 Parte 2	207396	207396	199794	195433	192780	180208	191268	165340	185430	157059
Mp3 Parte 3	239355	239355	229688	225775	231494	214449	222065	211901	196942	203942
Mp3 Parte 4	66374	66374	65191	65170	65989	64372	64253	64337	64253	64253
Mp3 Parte 5	58267	58267	57518	56881	58981	56041	57763	55887	57763	55880
Mp3 Parte 6	323167	323167	322726	318771	328606	317798	322614	317315	323510	316580
Mp3 Parte 7	3519251	3519251	3425654	3423092	3483705	3406733	3392887	3397150	3412641	3377277

	Tamanho 64									
	4 Palavras									
	Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.	
	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
Bubblesort 10	3817	3817	3824	3824	3824	3824	3845	3824	3873	3824
Bubblesort 100	607827	607827	604075	596004	613077	594968	598076	595052	598090	594975
Crane	74323	74323	76234	76052	77088	74330	77886	73910	78586	74806
Float Soma	32227	32227	32682	32374	33312	32157	33977	31989	34845	31954
IMDCT	103509	103509	96194	94822	98707	89950	96236	89530	89026	88795
Quicksort 10	2545	2545	2608	2573	2608	2559	2657	2559	2608	2566
Quicksort 100	28730	28730	28709	28149	29451	28324	29360	28268	29059	28219
Selectsort 10	2581	2581	2588	2588	2588	2588	2602	2588	2588	2588
Selectsort 100	151877	151877	145003	143918	146004	144296	146914	144793	147852	145297
Sincordic	1042	1042	1042	1056	1042	1056	1049	1056	1049	1056
Mp3 Parte 1	515909	515909	510869	506949	519136	501412	521005	499865	523350	499753
Mp3 Parte 2	219534	219534	213038	204477	214802	197638	215656	195818	213374	192066
Mp3 Parte 3	270533	270533	251689	237836	257548	224466	265094	224319	266466	222569
Mp3 Parte 4	65667	65667	63392	63329	64834	61453	61082	61460	61082	61418
Mp3 Parte 5	57245	57245	55152	54305	57182	53451	53871	53227	53871	53444
Mp3 Parte 6	328144	328144	320647	318456	329817	313815	315201	313696	312849	312338
Mp3 Parte 7	3641009	3641009	3519356	3451890	3575433	3381302	3425010	3354583	3325274	3337356

Tamanho 256																													
1 Palavra										2 Palavras										4 Palavras									
Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.		Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.		Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.	
LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
3887	3887	3887	3887	3887	3887	3894	3901	3887	3915	3838	3838	3838	3838	3838	3838	3845	3845	3859	3852	3817	3817	3817	3817	3817	3817	3824	3824	3824	3824
589858	589858	589858	589858	598769	589886	616108	589886	620203	589886	589494	589494	589494	589494	595437	589508	602976	589508	606371	589508	589312	589312	589312	589312	595066	589319	596340	589319	597999	589319
73588	73588	73574	73574	73924	73623	75401	73658	74274	75205	73371	73371	73357	73357	73665	73378	75534	73399	74610	74904	73224	73224	73196	73196	73490	73203	75163	73217	75296	74624
31863	31863	31856	31856	31891	31856	31996	31989	32073	32241	31828	31828	31821	31828	31870	31821	32136	31877	32206	32101	31793	31793	31793	31793	31849	31779	32087	31814	33060	31940
94773	94773	94766	94633	94913	94633	94654	94878	94885	94892	90958	90958	90923	90713	91350	90713	90776	90888	90776	90783	88984	88984	88893	88396	94647	88529	89376	88648	88473	88291
2615	2615	2615	2615	2615	2615	2664	2629	2615	2685	2580	2580	2580	2580	2580	2580	2622	2594	2594	2615	2545	2545	2545	2545	2545	2545	2601	2559	2559	2566
27974	27974	28422	28219	29031	28023	29185	28058	29227	28177	27624	27624	27904	27806	28310	27659	28639	27673	28618	27736	27421	27421	27610	27533	27981	27435	28268	27442	28184	27491
2644	2644	2644	2644	2644	2644	2651	2658	2644	2658	2602	2602	2602	2602	2602	2602	2609	2609	2602	2609	2581	2581	2581	2581	2581	2581	2588	2588	2588	2588
140495	140495	140495	140495	140936	140523	140649	140523	140523	140523	140131	140131	140131	140131	140586	140145	140306	140145	140152	140145	139949	139949	139949	139949	140432	139956	140110	139956	139956	139956
1119	1119	1119	1119	1119	1119	1119	1161	1119	1168	1070	1070	1070	1070	1070	1070	1070	1091	1070	1098	1042	1042	1042	1042	1042	1042	1042	1056	1042	1056
513809	513809	513452	512094	516602	511436	513445	511611	514334	516105	501153	501153	499788	498640	504338	498822	504513	498255	500964	502287	495434	495434	493411	492487	501202	491458	501419	491045	501706	491325
170128	170128	165536	164381	166873	163877	164143	163576	164108	163527	181468	181468	160867	158305	159208	153986	154651	153566	153734	153342	178269	178269	167538	164976	156765	150626	153356	149009	148351	148225
222345	222345	219027	217753	220413	217032	217774	216773	216997	216717	208422	208422	201968	202136	202892	201205	199224	198860	196921	196578	207750	207750	197208	198041	198097	193974	190831	190061	185770	186197
70476	70476	70420	70392	70518	70336	70336	70336	70336	70336	64631	64631	64421	64421	64617	64253	64253	64253	64253	64253	61936	61936	61418	61474	61810	61117	61082	61103	61082	61131
62061	62061	62180	61774	62642	61718	61718	61718	61718	61718	56398	56398	56265	56041	56685	55873	56531	55873	55873	55873	53983	53983	53507	53423	54018	53038	53304	53038	53031	53122
319744	319744	320605	315411	329355	314410	329040	314067	331392	314186	310112	310112	310518	307039	316328	306955	317728	305877	319478	305779	306367	306367	306535	303546	311862	304050	309489	303707	310833	303119
3436686	3436686	3426655	3408959	3440487	3509479	3420999	3514554	3416008	3529100	3307025	3307025	3289077	3279816	3307956	3329467	3305884	3333583	3309909	3336985	3279711	3279711	3232496	3236458	3259075	3256765	3244466	3254518	3277898	3245775

Tamanho 1024																													
1 Palavra										2 Palavras										4 Palavras									
Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.		Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.		Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.	
LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU
3887	3887	3887	3887	3887	3887	3887	3887	3887	3915	3838	3838	3838	3838	3838	3838	3838	3838	3838	3852	3817	3817	3817	3817	3817	3817	3817	3817	3817	3824
589858	589858	589858	589858	589858	589858	589858	589858	589886	589886	589494	589494	589494	589494	589494	589494	589494	589494	598153	589508	589312	589312	589312	589312	589312	589312	589312	589312	596886	589319
73574	73574	73574	73574	73574	73574	73574	73574	73805	75205	73357	73357	73357	73357	73357	73357	73357	73357	73553	74904	73196	73196	73196	73196	73196	73196	73196	73196	73378	74624
31856	31856	31856	31856	31856	31856	31856	31856	31982	32241	31821	31821	31821	31821	31821	31821	31821	31821	31898	32101	31779	31779	31779	31779	31779	31779	31779	31779	31807	31940
93968	93968	94045	94465	94605	94591	94493	94633	94633	94892	90020	90020	90097	90496	90678	90615	90524	90657	90650	90783	87969	87969	88011	88151	88452	88207	88228	88214	88249	88284
2615	2615	2615	2615	2615	2615	2615	2615	2615	2685	2580	2580	2580	2580	2580	2580	2580	2580	2580	2615	2545	2545	2545	2545	2545	2545	2545	2545	2545	2566
27974	27974	27974	27974	27974	27974	28261	27981	28114	28177	27624	27624	27624	27624	27624	27624	27890	27631	27932	27736	27421	27421	27421	27421	27421	27421	27603	27428	27722	27491
2644	2644	2644	2644	2644	2644	2644	2644	2644	2658	2602	2602	2602	2602	2602	2602	2602	2602	2602	2609	2581	2581	2581	2581	2581	2581	2581	2581	2581	2588
140495	140495	140495	140495	140495	140495	140495	140495	140495	140523	140131	140131	140131	140131	140131	140131	140131	140131	140152	140145	139949	139949	139949	139949	139949	139949	139949	139949	139956	139956
1119	1119	1119	1119	1119	1119	1119	1119	1119	1168	1070	1070	1070	1070	1070	1070	1070	1070	1070	1098	1042	1042	1042	1042	1042	1042	1042	1042	1042	1056
503568	503568	504058	506557	509245	508321	509903	508965	510379	514096	493712	493712	494216	494874	497184	495721	498773	495903	497828	500642	489442	489442	488903	488721	491150	489162	492382	489015	491381	489792
163177	163177	164318	163681	164738	163597	164038	163513	163989	163513	154056	154056	154210	153559	154882	153314	153706	153258	153559	153258	149289	149289	148827	148057	149814	147826	148057	147679	147868	147616
217235	217235	217284	214897	217795	214351	216920	214288	216731	214218	197915	197915	197453	196151	198321	195535	196970	195444	196466	195087	188570	188570	187443	185721	188948	185357	186428	185252	185462	184783
68831	68831	69328	70336	69475	70329	70287	70322	70336	70322	63616	63616	63840	64253	63917	64253	64246	64246	64253	64246	60858	60858	60942	61082	61054	61082	61082	61075	61082	61096
58519	58519	59114	58799	60647	58239	61284	58239	61718	58239	54431	54431	54634	54424	55439	54130	55866	54130	55873	54130	52415	52415	52415	52226	52982	52079	53031	52079	53031	52107
311393	311393	314676	312261	317546	311911	316048	311792	313570	312723	304659	304659	306458	305044	308166	304778	309090	304659	311421	305114	301586	301586	302783	301572	304134	301145	304113	301131	306346	301355
3333464	3333464	3410128	3383500	3419746	3387973	3412102	3389464	3410443	3525341	3235163	3235163	3271528	3254630	3278556	3256303	3275763	3256177	3267832	3331742	3186499	3186499	3197384	3184777	3207114	3187563	3201150	3186849	3191658	3225076

Tamanho 4096																														
1 Palavra										2 Palavras										4 Palavras										
Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.		Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.		Associatividade 1		Associatividade 2		Associatividade 4		Associatividade 8		Totalmente Ass.		
LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	LRU	LFU	S/ Cache
3887	3887	3887	3887	3887	3887	3887	3887	3887	3915	3838	3838	3838	3838	3838	3838	3838	3838	3838	3852	3817	3817	3817	3817	3817	3817	3817	3817	3817	3824	12609
589858	589858	589858	589858	589858	589858	589858	589858	589886	589886	589494	589494	589494	589494	589494	589494	589494	589494	589515	589508	589312	589312	589312	589312	589312	589312	589312	589312	589319	589319	1891312
73574	73574	73574	73574	73574	73574	73574	73574	73609	75205	73357	73357	73357	73357	73357	73357	73357	73357	73385	74904	73196	73196	73196	73196	73196	73196	73196	73196	73245	74624	158974
31856	31856	31856	31856	31856	31856	31856	31856	31856	32241	31821	31821	31821	31821	31821	31821	31821	31821	31835	32101	31779	31779	31779	31779	31779	31779	31779	31779	31793	31940	67269
93590	93590	93590	93590	93590	93940	93611	94465	94010	94892	89621	89621	89621	89621	89628	89957	89747	90482	90461	90776	87633	87633	87633	87633	87661	87815	87738	88102	88193	88256	335307
2615	2615	2615	2615	2615	2615	2615	2615	2615	2685	2580	2580	2580	2580	2580	2580	2580	2580	2580	2615	2545	2545	2545	2545	2545	2545	2545	2545	2545	2566	7207
27974	27974	27974	27974	27974	27974	27974	27974	27974	28177	27624	27624	27624	27624	27624	27624	27624	27624	27631	27736	27421	27421	27421	27421	27421	27421	27421	27421	27435	27491	115152
2644	2644	2644	2644	2644	2644	2644	2644	2644	2658	2602	2602	2602	2602	2602	2602	2602	2602	2602	2609	2581	2581	2581	2581	2581	2581	2581	2581	2581	2588	9203
140495	140495	140495	140495	140495	140495	140495	140495	140495	140523	140131	140131	140131	140131	140131	140131	140131	140131	140131	140145	139949	139949	139949	139949	139949	139949	139949	139949	139949	139956	710995
1119	1119	1119	1119	1119	1119	1119	1119	1119	1168	1070	1070	1070	1070	1070	1070	1070	1070	1070	1098	1042	1042	1042	1042	1042	1042	1042	1042	1042	1056	1966
498962	498962	499949	498696	500705	498626	500271	503729	500593	509826	490982	490982	491451	490590	492074	490422	493551	492977	493614	498248	487692	487692	486831	486194	487321	485998	488406	487272	489134	488287	969068
155736	155736	155645	156212	158018	162379	157451	163492	159047	163499	149429	149429	149310	149436	150787	152775	151592	153244	152572	153244	145929	145929	145817	145677	146797	147350	146902	147574	147476	147574	798427
210690	210690	209584	210116	211439	212643	212594	212839	213574	212895	193582	193582	192812	193183	194184	194338	194562	194352	195416	194401	184111	184111	183628	183712	184699	184202	184412	184216	185007	184258	1187904
66241	66241	66241	66269	66248	68789	66241	70315	66241	70315	62195	62195	62195	62209	62209	63539	62447	64232	62797	64232	60039	60039	60039	60046	60067	60711	60172	61061	60725	61068	191597
58183	58183	58183	58183	58302	58211	58183	58225	58183	58225	54095	54095	54095	54095	54193	54109	54291	54116	54333	54116	52016	52016	52016	52016	52128	52023	52142	52030	52611	52037	189678
309475	309475	310021	309475	310868	309503	311393	309860	313304	310140	303287	303287	303623	303287	304267	303301	305975	303595	305345	303742	300186	300186	300529	300207	301138	300193	302118	300459	301222	300529	947462
3154824	3154824	3203516	3208171	3281965	3217894	3288412	3198917	3401119	3213309	3133810	3133810	3161061	3158513	3204867	3163238	3260664	3155006	3262729	3166836	3121700	3121700	3136302	3132193	3160529	3136666	3189810	3130422	3186814	3139158	1E+07

Anexo II – Simulador em Linguagem C

```
#include "cache.h"

#define HIT 1
#define MISS -1

void cache_inicializa(cache c, int tam){
      int i;
      miss=0;
      hit=0;
      for(i=0;i<tam;i++){
            c[i].valido=0;
            c[i].tag=0;
            c[i].alg=0;
            c[i].dado[0]=0;
            c[i].dado[1]=0;
            c[i].dado[2]=0;
            c[i].dado[3]=0;
      }

}

unsigned int gera_bits(int tamanho){

      int i=0;
      if(tamanho==1){
            return 0;
      }else{
            while(tamanho>1){
                  tamanho = tamanho /2;
                  i++;
            }
      }

      return i;

}

int calcula_mask_cache(int bits_cache,int bits_assoc){
            int i,mask=0;
            for(i=0;i<(bits_cache-bits_assoc);i++)
                  mask=((mask<<1) |1);
            return mask;


}
int calcula_mask_rotulo(int bits_memoria,int bits_cache,int
bits_assoc){
            int i,mask=0;
            for(i=0;i<(bits_memoria-(bits_cache-bits_assoc));i++){
                  mask=((mask<<1) |1);

            }
            for(i=0;i<(bits_cache-bits_assoc);i++){
                  mask=(mask <<1);

            }
            return mask;
```

```
}

int calcula_mask_palavra(int palavra){
	if(palavra==1){
		return 0;
	}else if(palavra==2){
		return 0x1;
	}else{
		return 0x3;
	}


}

unsigned int cache_calcula_rotulo(unsigned int *end,int
mask_rotulo,int bits_cache,int bits_assoc){
	unsigned int rotulo=0;
	rotulo=((*end & mask_rotulo) >> (bits_cache-bits_assoc));
	return rotulo;

}
unsigned int cache_calcula_palavra(unsigned int *end, int
mask_palavra){
	unsigned int posicao;
	posicao=(*end & mask_palavra);
	return posicao;

}

unsigned int cache_calcula_num_conjuntos(int associatividade,int
tam_cache,int tam_palavra){
	int num_conjs;
	num_conjs= ((tam_cache/tam_palavra) / associatividade);
	return num_conjs;

}


unsigned int cache_calcula_conjunto(unsigned int *end,int
mask_cache,int palavra,int associatividade,int tam_cache){
	unsigned int conjunto, num_conjs,indice;
	num_conjs=
cache_calcula_num_conjuntos(associatividade,tam_cache,palavra);
	conjunto= (int)(((*end & mask_cache) / palavra) % num_conjs);
	indice=cache_calcula_indice_conjunto(conjunto,associatividade);
	return indice;

}

unsigned int cache_calcula_indice_conjunto(int conjunto,int
associatividade){
	unsigned int indice;
	indice= conjunto * associatividade;
	return indice;

}

int LFU(cache c,int prim_indice, int associatividade){
	int i,menor;
```

```
	menor=prim_indice;
	for(i=prim_indice;i<(prim_indice+associatividade);i++){
		if(c[i].alg < c[menor].alg){
			menor=i;
		}
	}
	return menor;


}
int LRU(cache c,int prim_indice, int associatividade){
	int i,menor;
	menor=prim_indice;
	for(i=prim_indice;i<(prim_indice+associatividade);i++){
		if(c[i].alg < c[menor].alg){
			menor=i;
		}
	}
	return menor;


}

int cache_busca(cache c,unsigned int *end, char *tipo,int
mask_cache,int palavra,int mask_rotulo,int bits_cache,int
associatividade,int tam_cache,int bits_assoc,char alg_sub){
	unsigned int prim_indice,rotulo;
	int i;
	i=0;
	prim_indice=cache_calcula_conjunto(end,mask_cache,palavra,associ
atividade,tam_cache);
	rotulo=cache_calcula_rotulo(end,mask_rotulo,bits_cache,bits_asso
c);

	if(alg_sub=='r' || alg_sub=='R'){
		for(i=prim_indice;i<(prim_indice+associatividade);i++){

			c[i].alg=c[i].alg-1;
		}
	}
	if(*tipo=='w'|| c[prim_indice].valido == 0){

		cache_miss();


		return MISS;

	}else{
		for(i=prim_indice;i<(prim_indice+associatividade);i++){

			if(c[i].tag == rotulo && c[i].valido == 1){

				cache_hit();

				if(alg_sub=='f' || alg_sub=='F'){
					c[i].alg =c[i].alg+1;
				}
				if(alg_sub=='r' || alg_sub=='R'){
					c[i].alg =0;

				}
```

```
                        return HIT;

                  }
            }


            cache_miss();

            return MISS;


      }

}

void cache_escreve_dado(cache c,unsigned int *end,int *dados,int
mask_cache,int mask_palavra,int mask_rotulo,int palavra,int
bits_cache,int associatividade, int tam_cache,int bits_assoc, char
alg_sub){
      unsigned int prim_indice,rotulo,posicao,indice_rem;
      prim_indice=
cache_calcula_conjunto(end,mask_cache,palavra,associatividade,tam_cach
e);
      rotulo=cache_calcula_rotulo(end,mask_rotulo,bits_cache,bits_asso
c);
      posicao=cache_calcula_palavra(end, mask_palavra);

      if(alg_sub=='r' || alg_sub=='R'){
            indice_rem=LRU(c,prim_indice,associatividade);
      }else if(alg_sub=='f' || alg_sub=='F'){
            indice_rem=LFU(c,prim_indice,associatividade);
      }
      c[indice_rem].tag=rotulo;
      c[indice_rem].valido=1;
      if(alg_sub=='r' || alg_sub=='R'){
            c[indice_rem].alg =0;

      }
}

void cache_miss(){
      miss = miss +1;
}

void cache_hit(){
      hit = hit +1;
}

void cache_resultados(FILE *arq2, char *nome_arq,int ciclos,int
tamanho, int palavras, int associatividade, char alg_sub){
      float total;
      int ciclos_miss;
      total= (float) (miss + hit);
      ciclos_miss=ciclos*miss;
      printf("Cache Miss= %d, [%.2f %%]\n",miss,(miss/total)*100);
      printf("Cache Hit= %d, [%.2f %%]\n",hit,(hit/total)*100);

}

int main(int argc,char *argv[]){
```

```
	FILE *arq,*arq2;
	char tipo,alg_sub;
	unsigned int end;
	cache c;
	int
resultado,dado,tam_memoria,tam_cache,tam_palavra,bits_memoria,bits_cac
he,bits_assoc,mask_cache,mask_rotulo,mask_palavra,ciclos,assoc;
	tam_memoria=atoi(argv[2]);
	tam_cache=atoi(argv[4]);
	tam_palavra=atoi(argv[6]);
	ciclos=atoi(argv[8]);
	assoc=atoi(argv[10]);
	alg_sub=*(argv[11]+1);
	bits_memoria=gera_bits(tam_memoria);
	bits_cache=gera_bits(tam_cache);
	bits_assoc=gera_bits(assoc);
	cache_inicializa(c,tam_cache);
	mask_cache=calcula_mask_cache(bits_cache,bits_assoc);
	mask_rotulo=calcula_mask_rotulo(bits_memoria,
bits_cache,bits_assoc);
	mask_palavra=calcula_mask_palavra(tam_palavra);

	if(tam_memoria < tam_cache){
		printf("O tamanho da memoria cache nao pode ser maior que o
tamanho da memoria principal\n");
		exit(1);
	}
	if(tam_palavra !=1 && tam_palavra !=2 && tam_palavra !=4){
		printf("Opção de palavras por linha inválida! Digite 1,2 ou
4 para esta opção\n");
		exit(1);
	}
	if(assoc>tam_cache){
		printf("A associatividade não pode ser maior que o tamanho
da memória cache\n");
		exit(1);
	}
	if((bits_cache-bits_assoc) < gera_bits(tam_palavra)){
		printf("Configuração inválida de associatividade e
localidade espacial para o tamanho de cache especificado!!\n");
		exit(1);
	}

	if(argc!=13){
		printf("Digite 'cache -m <tamanho da memoria> -c <tamanho
da cache> -p <numero de palavras por linha> -pm <numero de ciclos
perdidos por cache miss> -a <associatividade> -alg <-f = LFU -r = LRU>
<nome do arquivo>'\n");
		exit(1);
	}
	arq=fopen(argv[12],"r");
	arq2=fopen("total_mp3_inst.txt","a");

	if(arq==NULL){
		printf("O arquivo não existe ou não pode ser aberto!\n");
		exit(1);
	}

	while(!feof(arq)){
		fscanf(arq,"%c %x %d\n",&tipo,&end,&dado);
```

```
	resultado=cache_busca(c,&end,&tipo,mask_cache,tam_palavra,mask_r
otulo,bits_cache,assoc,tam_cache,bits_assoc,alg_sub);
		if(resultado==MISS){
			cache_escreve_dado(c,&end,&dado,mask_cache,
mask_palavra, mask_rotulo,
tam_palavra,bits_cache,assoc,tam_cache,bits_assoc,alg_sub);
		}

	}


	cache_resultados(arq2,argv[12],ciclos,tam_cache,tam_palavra,asso
c,alg_sub);
	fclose(arq);
return 1;


}
```