Otimização de memória cache em tempo de execução para o processador embarcado LEON3

Lucas Albers Cuminato

Lucas Albers Cuminato

Otimização de memória cache em tempo de execução para o processador embarcado LEON3

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Vanderlei Bonato

USP – São Carlos
Junho de 2017

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

C969o

Cuminato, Lucas Albers
Otimização de memória cache em tempo de execução para o processador embarcado LEON3 / Lucas Albers Cuminato; orientador Vanderlei Bonato. - São Carlos - SP, 2017.
78 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2017.

1. Computação reconfigurável. 2. Memória cache. 3. LEON3. 4. Sistemas embarcados. 5. FPGA. I. Bonato, Vanderlei, orient. II. Título.

Lucas Albers Cuminato

Runtime cache optimization for embedded processor LEON3

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Vanderlei Bonato

USP – São Carlos
June 2017

Agradecimentos

Agradeço à minha família pelo apoio incondicional. Agradeço também ao meu orientador pela enorme paciência e boa vontade que sempre teve comigo. Agradeço ao Prof. Dr. Pedro Diniz, meu orientador durante estágio de 1 ano no exterior na Universidade do Sul da California, EUA no *Information Sciences Institute* pelos conselhos dados e pela paciência durante a elaboração deste trabalho. Por fim, minha gratidão especial à cinco grandes amigos, Cale, Ian, Kemelli, Marie e Melbe, a quem realmente considero como irmãos, que me ajudaram muito durante o estágio nos Estados Unidos e não pouparam esforços para que eu sempre me sentisse em casa. Por fim agradeço também à FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) e ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) pelo financiamento deste trabalho, através dos processos 2011/15094-5, 2012/11042-3, e 130007/2012-9, respectivamente.

Resumo

O consumo de energia é uma das questões mais importantes em sistemas embarcados. Estudos demonstram que neste tipo de sistema a *cache* é responsável por consumir a maior parte da energia fornecida ao processador. Na maioria dos processadores embarcados, os parâmetros de configuração da *cache* são fixos e não permitem mudanças após sua fabricação/síntese. Entretanto, este não é o cenário ideal, pois a configuração da *cache* pode não ser adequada para uma determinada aplicação, tendo como consequência menor desempenho na execução e consumo excessivo de energia. Neste contexto, este trabalho apresenta uma implementação em hardware, utilizando computação reconfigurável, capaz de reconfigurar automática, dinâmica e transparentemente a quantidade de *ways* e por consequência o tamanho da *cache* de dados do processador embarcado LEON3, de forma que a *cache* se adeque à aplicação em tempo de execução. Com esta técnica, espera-se melhorar o desempenho das aplicações e reduzir o consumo de energia do sistema. Os resultados dos experimentos demonstram que é possível reduzir em até 5% o consumo de energia das aplicações com degradação de apenas 0.1% de desempenho.

Abstract

Energy consumption is one of the most important issues in embedded systems. Studies have shown that in this type of system the cache consumes most of the power supplied to the processor. In most embedded processors, the cache configuration parameters are fixed and do not allow changes after manufacture/synthesis. However, this is not the ideal scenario, since the configuration of the cache may not be suitable for a particular application, resulting in lower performance and excessive energy consumption. In this context, this project proposes a hardware implementation, using reconfigurable computing, able to reconfigure the parameters of the LEON3 processor's cache in run-time improving applications performance and reducing the power consumption of the system. The result of the experiment shows it is possible to reduce the processor's power consumption up to 5% with only 0.1% degradation in performance.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo

1

Introdução

Nas últimas quatro décadas, a evolução dos microprocessadores com relação à velocidade de processamento seguiu fielmente as predições da lei de Moore (Gelsinger, 2006; Steehler, 2007), dobrando o poder de processamento a cada dezoito meses. Pode-se dizer o mesmo sobre a capacidade de armazenamento das memórias, que também evoluiu na mesma taxa. Entretanto, a velocidade de acesso à memória não acompanhou este crescimento, criando desta forma um gargalo entre o processador e a memória. Como o desempenho computacional do sistema é altamente dependente da cooperação entre processador e memória, caso exista algum gargalo entre estes componentes, certamente haverá perda de desempenho (Jheng et al., 2010). Para tentar solucionar esse problema, a IBM desenvolveu na década de 60 um componente de hardware capaz de minimizar esta diferença de velocidade entre processador e memória, componente este conhecido como memória *cache*. A empresa foi pioneira ao integrar este recurso em seus processadores (Asaduzzaman, 2009).

Um dos motivos pelo qual a utilização de uma *cache* de dados ou de instrução permite uma melhora expressiva no desempenho do sistema está diretamente relacionado com o comportamento padrão que os programas costumam seguir ao longo de sua execução (Jacob et al., 2007). Ao longo do tempo, os pesquisadores descobriram algo em comum com o comportamento dos algoritmos, de forma que foi possível extrair determinados padrões dessas observações. A primeira constatação foi de que o acesso à memória não é totalmente aleatório, como era esperado. Adicionalmente, descobriu-se também que acessos a determinados locais da memória ou a endereços vizinhos a estes locais tendem

a se repetir ao longo da execução do programa (Jacob et al., 2007). Cabe mencionar que estes padrões foram descobertos a partir de observações práticas e que não é verdadeiro afirmar que qualquer algoritmo ou programa tem que obrigatoriamente apresentá-los. A partir desses padrões, os pesquisadores concluíram que se um programa referencia um dado, é muito provável que em um futuro próximo este dado seja utilizado novamente e que se um programa referencia um dado, é muito provável que em um futuro próximo dados vizinhos também sejam utilizados. O primeiro fenômeno é conhecido como localização temporal e o segundo localização espacial (Hennessy e Patterson, 2007). A *cache* explora justamente esses dois conceitos (Jheng et al., 2010).

A presença da *cache* no sistema, entretanto, pode acabar prejudicando a previsibilidade e causar um aumento excessivo no consumo de energia. Em sistemas embarcados (Marwedel, 2006), o consumo de energia é sem dúvida uma questão fundamental, pois estes sistemas frequentemente possuem fontes de energia de baixa capacidade (Asaduzzaman, 2009). Com base neste contexto, elaborar alguma técnica que aperfeiçoe a *cache* de instruções e de dados de acordo com a necessidade da aplicação em execução, de maneira que este processo seja realizado em tempo de execução, pode resultar em reduções expressivas no consumo de energia do sistema. Está técnica é conhecida como reconfiguração dinâmica da *cache* (Sundararajan et al., 2011). Com a técnica pode-se ainda reduzir o *miss rate* da *cache* e como consequência melhorar o desempenho da aplicação em execução.

1.1 Objetivos

Nesta seção será apresentado o objetivo geral deste projeto assim como seus objetivos específicos.

1.1.1 Objetivo Geral

Este projeto tem como objetivo principal otimizar a *cache* de dados do processador *soft-core* LEON3 de forma que seja possível a reconfiguração de seus parâmetros em tempo de execução. Os parâmetros que se deseja configurar dinamicamente são:

- Tamanho da *cache*;
- Grau de associatividade.

Como resultado final deste trabalho, espera-se melhorar o desempenho do sistema e reduzir o consumo de energia dinâmica de modo automático e transparente ao usuário.

1.1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Desenvolver um monitor em hardware para avaliar o desempenho da aplicação em execução de acordo com a configuração de *cache* atual, fornecendo dados como *cache miss* e *cache hit*;
- Mapear quais configurações de *cache* são as mais importantes considerando o impacto que estas configurações possuem no desempenho da aplicação em execução;
- Alterar o controlador de memória *cache* de dados do LEON3 de forma a permitir a reconfiguração dinâmica de seus parâmetros;
- Desenvolver um algoritmo em software ou hardware que ative a reconfiguração da *cache* de acordo com algum parâmetro, como por exemplo, quantidade de instruções executadas, taxa de *read miss* e *read hit*;
- Avaliar o impacto em relação ao desempenho que a reconfiguração de *cache* promove, assim como verificar se há *overhead* na solução proposta;
- Avaliar o impacto em relação ao consumo de energia que a reconfiguração promove através de ferramentas que calculem o consumo de energia da *cache* ou com a utilização de componentes de hardware que monitorem o consumo instantâneo da plataforma de hardware a ser utilizada neste trabalho.

1.2 Justificativa

O ganho de desempenho de um sistema é expressivo quando há uso de *cache* (Mahapatra, 1999). Entretanto, sua presença pode acabar prejudicando a previsibilidade e causar um aumento do consumo de energia do sistema. (Segars, 2001) demonstrou, através de uma análise do consumo de energia do processador ARM920T, que a *cache* de instruções e de dados deste processador são responsáveis por 44% de toda a energia consumida, como ilustrado na Figura 1.1.

Em sistemas embarcados, esse aumento de consumo pode ser muito prejudicial, de modo que o ganho de desempenho pode não compensar o consumo extra de energia (Asaduzzaman et al., 2007). O consumo de energia é sem dúvida uma das questões mais importantes em sistemas embarcados, pois frequentemente possuem fontes de energia de baixa capacidade (Asaduzzaman, 2009). Desta forma, elaborar alguma técnica que aperfeiçoe a *cache* de instruções e de dados de acordo com a necessidade da aplicação em

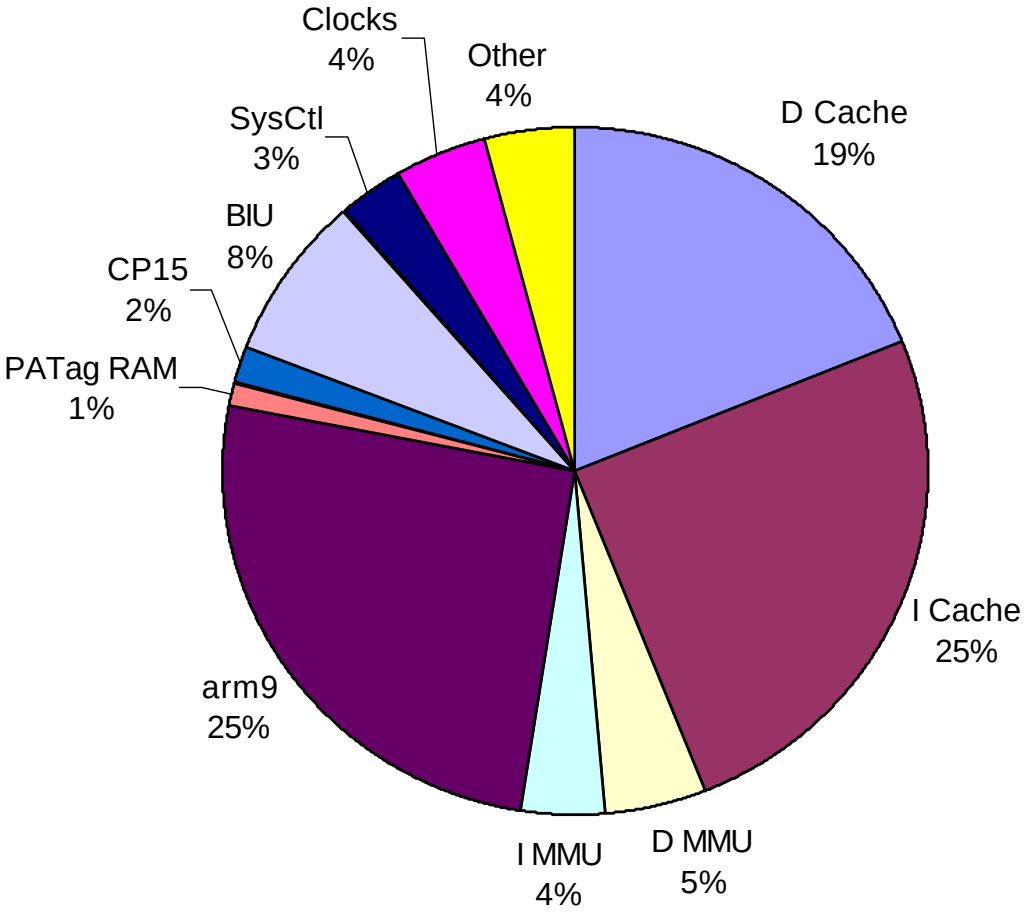


Figura 1.1: Análise do consumo de energia do processador ARM920T (Segars, 2001).

execução, de maneira que este processo seja realizado em tempo de execução, pode resultar em reduções expressivas no consumo de energia do sistema. Além disto, pode-se também obter melhor desempenho, uma vez que a *cache* é otimizada para aquela aplicação em execução.

A proposta deste trabalho foca justamente neste aspecto da otimização da *cache* em tempo de execução, considerando os parâmetros apresentados na seção 1.1.1. Estes recursos de reconfiguração serão implementados diretamente no hardware controlador de memória *cache* de dados do sistema.

1.3 Organização da Dissertação

Esta dissertação de mestrado está organizada da seguinte forma. O Capítulo 2 apresenta a revisão bibliográfica que aborda o tema de *cache* reconfigurável. O Capítulo 3 aborda os conceitos sobre o funcionamento de uma memória *cache*. O Capítulo 4 apresenta os materiais e métodos que foram utilizadas para implementação e validação do projeto. O

Capítulo 5 apresenta a implementação da arquitetura proposta. Por fim, o Capítulo 6 apresenta os resultados obtidos e o Capítulo 7 a conclusão.

Capítulo

2

Revisão Bibliográfica

A reconfiguração dinâmica da *cache* não é um assunto novo para a comunidade científica da área de arquitetura de hardware e tem sido objeto de estudo de vários pesquisadores (Abella e González, 2006; Albonesi, 1999; Gordon-Ross e Vahid, 2007; Zhang et al., 2004). Basicamente, esses trabalhos propõem o monitoramento da taxa *cache miss* e *cache hit* em tempo de execução e caso estes valores passem de um limite pré-estabelecido, há a reconfiguração dinâmica. Entretanto, a reconfiguração fica restrita a um ou dois parâmetros, geralmente alterando apenas o tamanho da *cache* e/ou o grau de associatividade. No artigo de Albonesi (Albonesi, 1999), é proposto uma *cache* que pode variar de grau de associatividade e consequentemente de tamanho apenas habilitando e desabilitando diferentes *ways* da *cache*, reduzindo assim o consumo de energia dinâmica[1]. Para a implementação da técnica proposta, é levado em consideração que a *cache* já é fisicamente particionada, ou seja, uma *cache 4-way set associative* de 16 *KBytes*, por exemplo, é fisicamente particionada em quatro sub-blocos de 4 *KBytes* cada, sendo que cada bloco representa uma *way* da *cache*. Tendo então como premissa que a *cache* já está particionada, Albonesi (Albonesi, 1999) propõe que as *ways* da *cache* sejam ativadas e desativadas através de uma lógica implementada em hardware. Entretanto, a decisão sobre quando ativar ou desativar as *ways* é implementada em software e controlada através de registradores. A lógica adicional implementada por (Albonesi, 1999) é demonstrada na Figura 2.1.

[1]Energia dinâmica corresponde à energia consumida pelo circuito CMOS nas cargas e descargas das capacitâncias internas e quando há comutação dos transistores (Kofuji et al., 2012).

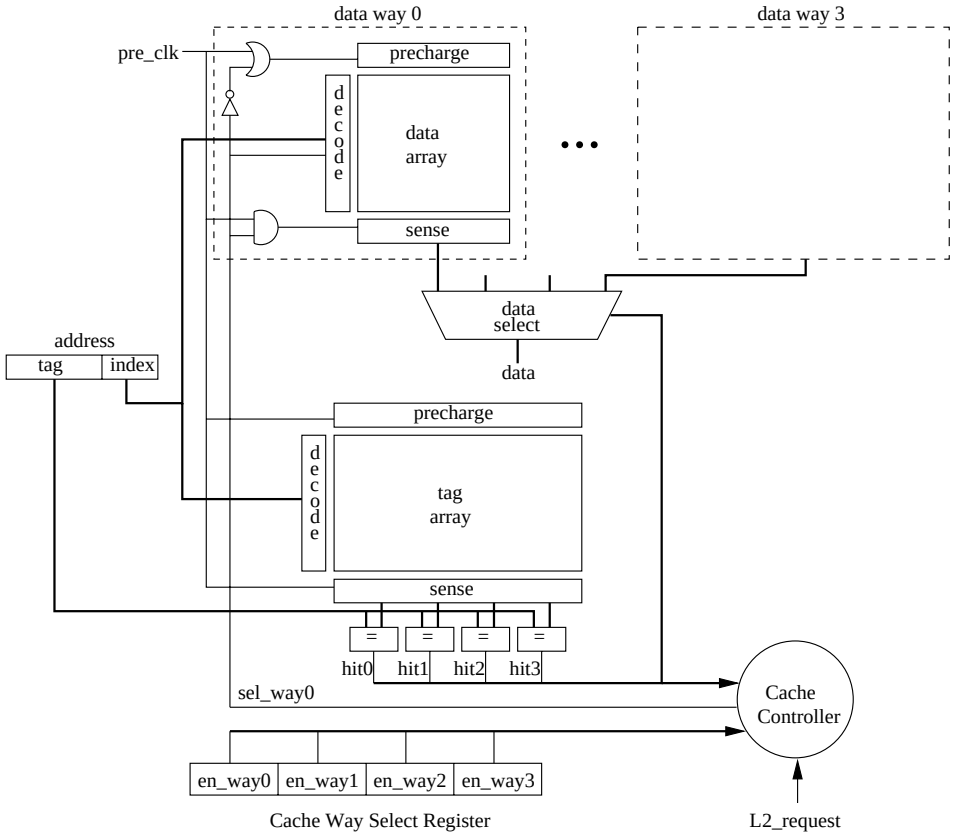
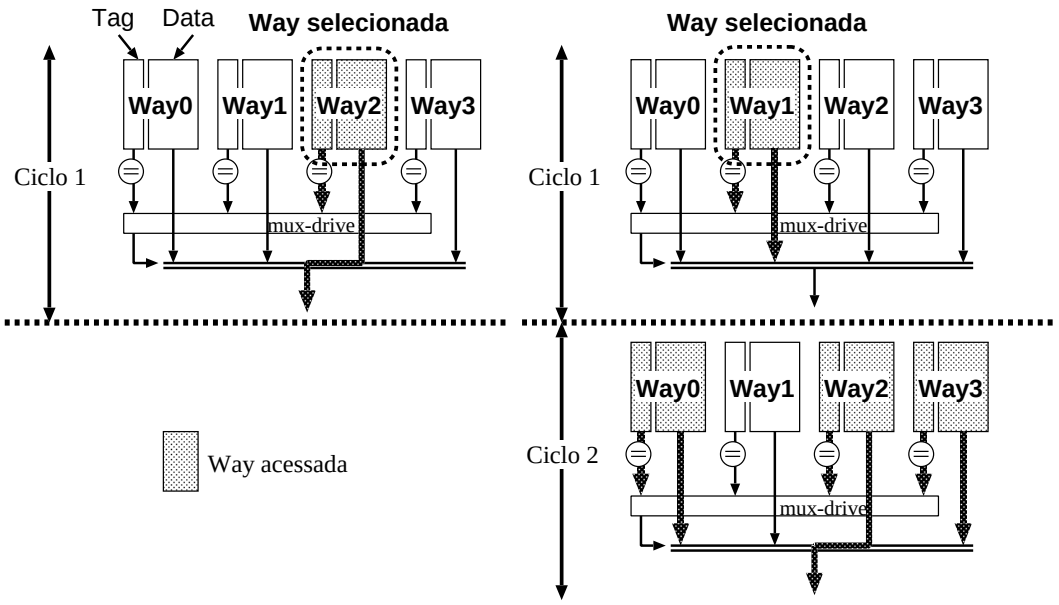


Figura 2.1: Lógica proposta por (Albonesi, 1999) para ativar e desativar *ways* em uma *cache 4-way*.

O controle sobre quais *ways* ativar é realizado através do registrador *Cache Way Select Register*, e possui um *bit* para cada *way* da *cache*. Se o *bit* de uma determinada *way* no registrador de controle for zero a *way* é desabilitada. Desabilitar a *way* previne que ela seja acessada e desta forma há redução no consumo de energia dinâmica. Os resultados obtidos por (Albonesi, 1999) mostram que, com a técnica proposta, é possível reduzir em até 40% o consumo de energia com relação a uma *cache 4-way set associative*, comprometendo em apenas 2% o desempenho da aplicação. Esta técnica é conhecida como *way-shutdown*.

Way-Prediction (Calder et al., 1996; Inoue et al., 1999; Ma et al., 2001; Powell et al., 2001) é outra técnica conhecida para se reduzir o consumo de energia dinâmica que tenta prever qual *way* da *cache* será acessada. Se a previsão estiver correta, apenas uma *way* da *cache* é acessada e não há necessidade em consultar as outras *ways* da *cache*, Figura 2.2(a). Entretanto, se a previsão estiver errada todas as outras *ways* da *cache* precisam ser checadas, Figura 2.2(b). Esta técnica é mais apropriada para as *caches* que possuem acessos previsíveis, como por exemplo a *cache* de instruções. A *cache* de dado possui acesso irregular e não previsível e neste caso teria-se mais previsões erradas do que certas, aumentando assim a latência de acesso.



(a) Previsão correta. (b) Previsão errada.

Figura 2.2: *Cache* com *Way-Prediction* (Inoue et al., 1999).

Outra solução é apresentada por (Yang et al., 2000), sendo proposta uma abordagem conhecida como reconfiguração *set-only*. Neste método o tamanho da *cache* é alterado ligando e desligando as linhas da *cache*. Quando uma linha é desligada, o tamanho da *cache* diminui e há redução no consumo de energia estática[2]. A taxa *miss rate* foi utilizada como métrica para ativar a reconfiguração, realizando uma operação *shift* nos *bits* do campo *index* para alterar o tamanho da *cache*, conforme ilustrado na Figura 2.3.

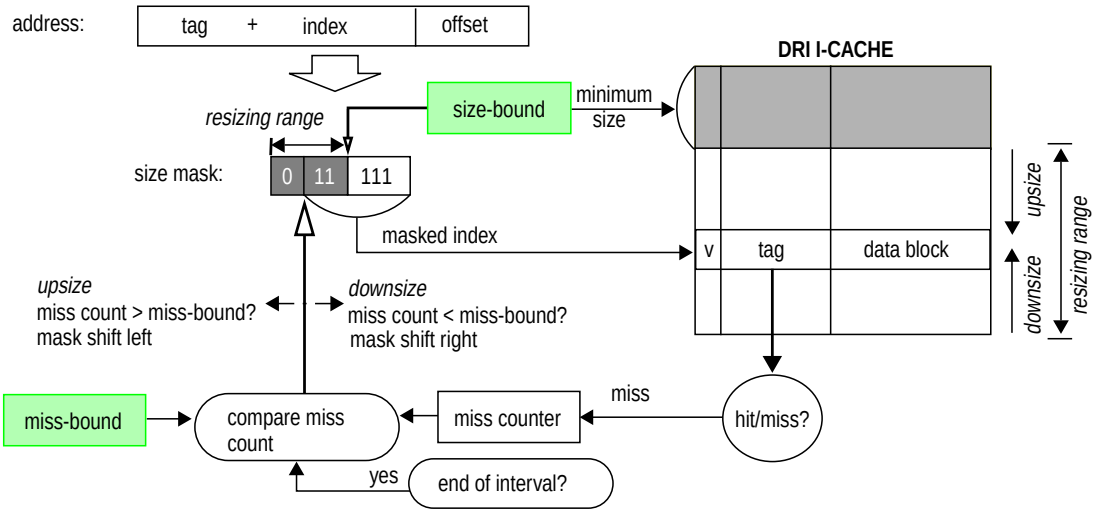


Figura 2.3: Lógica proposta por (Yang et al., 2000) para alterar dinamicamente o tamanho da *cache* de instruções

[2]Energia estática corresponde à energia consumida pelo circuito CMOS devido a corrente de fuga interna presente nos transistores do circuito integrado (Kofuji et al., 2012).

Esta abordagem é mais complexa pois reduzir ou aumentar o tamanho da *cache* implica em alterar dinamicamente a quantidade de *bits* dos campos *tag* e *index*. Em *caches* menores há menos *bits* no campo *index* e mais no campo *tag* e em *caches* maiores o contrário. A solução proposta por (Yang et al., 2000) utiliza no campo *tag* a quantidade máxima de *bits* necessários para a menor *cache* possível. Desta forma, dependendo do tamanho da *cache* escolhida, pode-se ter mais *bits* para um campo do que para o outro.

Há ainda trabalhos que focam em otimizar outros parâmetros da *cache* como por exemplo a política de substituição (Jaleel et al., 2010; Lee et al., 2001).

Capítulo

3

Memória Cache

Este capítulo descreve detalhadamente o que é uma memória *cache*, como a utilização de uma *cache* pode melhorar substancialmente o desempenho de um processador, como é feito o acesso à memória *cache*, quais os tipos de erros (*misses*) que um programa em execução pode causar na *cache* e por fim, apresenta como a *cache* pode ser organizada logicamente (*direct-mapped, set associative, fully associative*).

3.1 Visão geral

A *cache* é um dispositivo de hardware inventado pela IBM na década de 60 com o objetivo de minimizar a latência existente entre a unidade de processamento e a memória principal. Esta latência entre os componentes surgiu devido à rápida evolução na capacidade de processamento dos processadores e da evolução não tão expressiva com relação à redução do tempo de acesso à memória principal. Como a unidade de processamento se tornou cada vez mais rápida, o tempo entre buscar uma instrução na memória e executar esta instrução diminuiu. Para que o processador funcione em sua capacidade máxima de execução, a memória precisa "servir" o processador com instruções e dados na mesma velocidade em que ele as "consome". Como este não é o caso, criou-se uma lacuna (*gap*) entre estes componentes. Ao analisarmos o gráfico apresentado na Figura 3.1, percebemos claramente a formação deste *gap* ao longo dos últimos anos. Note que foi preciso utilizar escala logarítmica no eixo das ordenadas para melhor representar o gráfico.

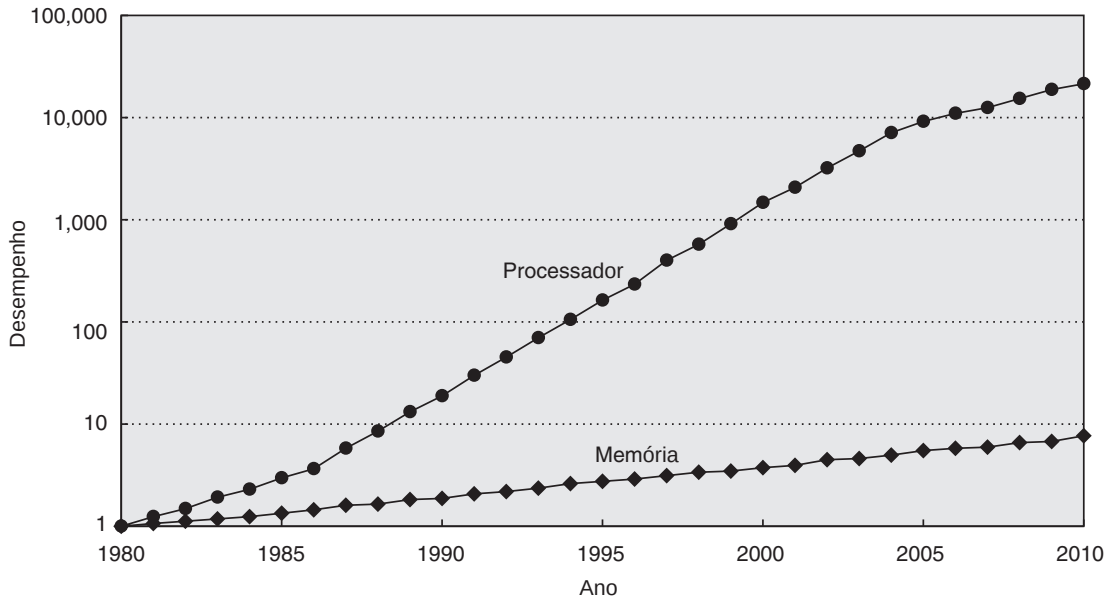


Figura 3.1: Evolução no desempenho dos processadores e memória DRAM ao longo dos últimos anos. Adaptado de (Hennessy e Patterson, 2007).

Neste contexto, a *cache* é um banco reduzido de memória (ordem de KBytes) com tempo de acesso muito baixo (ordem de 0,5 a 5 ns) e que nos processadores recentes encontra-se integrada ao *die*[1] do processador. Por ser uma memória muito rápida, o processador primeiro faz a busca da instrução ou dado na memória *cache*. Caso o dado não seja encontrado, o processador então faz a busca no próximo nível, que pode ser outra *cache*, a memória principal ou no pior dos casos, o disco magnético. A este tipo de organização dá-se o nome de hierarquia de memória, onde primeiro é consultado a memória no primeiro nível e, consequentemente, a mais rápida do sistema até chegar ao último nível e, consequentemente, à memória de maior tempo de resposta. Para melhor entendimento, observe na Figura 3.2 como as memórias estão organizadas e como o processador as acessa, do topo até a base da pirâmide, passando por cada nível e consultando o próximo nível caso o dado não seja encontrado no nível atual. Como descrito anteriormente, a utilização deste tipo de hierarquia reduz o tempo de acesso à memória devido à existência dos princípios de localidade temporal e espacial presentes na grande maioria dos programas, caso contrário, de nada serviria a *cache* (Hennessy e Patterson, 2007; Patterson e Hennessy, 2005).

A memória utilizada pela *cache* é de baixa latência, pois é fabricada com uma tecnologia diferente daquela utilizada na fabricação dos *chips* presentes na memória principal. Geralmente, a memória *cache* é fabricada utilizando memória do tipo *Static Random-Access Memory* (SRAM) enquanto os *chips* da memória principal utilizam a tecnologia *Dynamic Random-Access Memory* (DRAM). A vantagem da SRAM é justamente a baixa latência

[1] O *die* é um pequeno bloco de material semicondutor que contêm toda a lógica do circuito integrado.

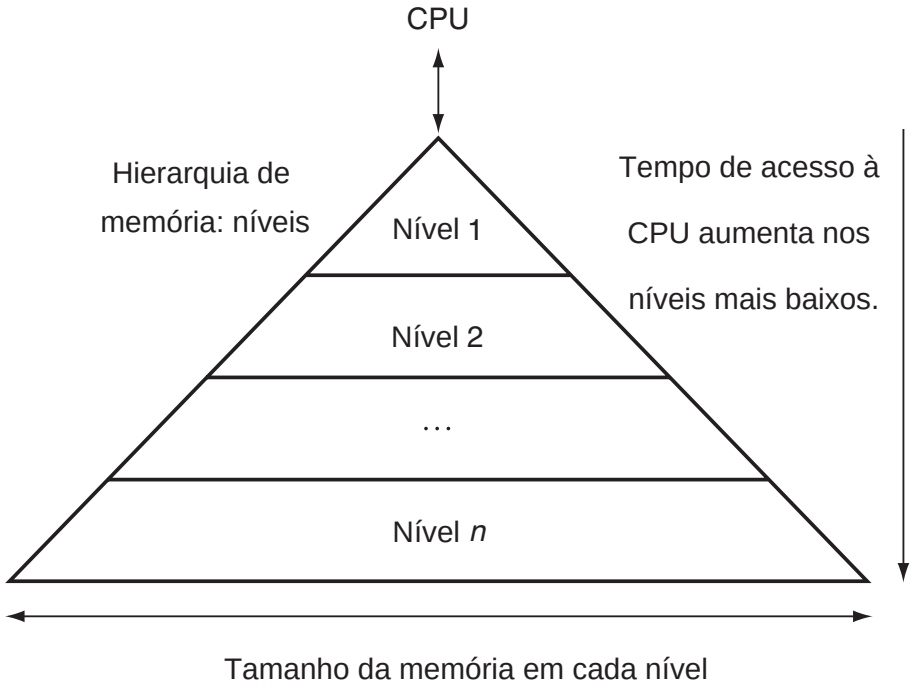


Figura 3.2: Hierarquia de memória (Patterson e Hennessy, 2005).

nas operações de leitura/escrita, porém o custo por *bit* deste tipo de memória é muito maior se comparado ao custo da tecnologia DRAM. Nas memórias do tipo DRAM o custo por *bit* é menor, pois este tipo de memória utiliza menos área de silício por *bit* do que as memórias SRAM, e desta forma pode-se construir *chips* com maiores capacidades de armazenamento com a mesma quantidade de silício (Patterson e Hennessy, 2005).

Em uma memória SRAM convencional, cada *bit* utiliza quatro transistores para armazenamento de seu valor e mais dois transistores para controle das operações de leitura e escrita a este *bit* (Kulkarni et al., 2007). Desta forma, são necessários seis transistores do tipo MOSFET (Kulkarni et al., 2007) para construir uma memória de um único *bit*. A Figura 3.3 representa a estrutura interna de um *bit* em uma memória 6T-SRAM. No entanto, para a construção de um *bit* em uma DRAM convencional é utilizado apenas um transistor do tipo MOSFET e um capacitor, como pode ser observado na Figura 3.4. A estrutura de um *bit* em uma memória DRAM é mais simples e ocupa menos espaço e por este motivo o custo por *bit* é reduzido.

Outra tecnologia que é utilizada para armazenamento de dados é o disco magnético, sendo este o que possui maior latência entre as outras duas tecnologias, mas com a vantagem de possuir o menor custo por *bit*. A Tabela 3.1 apresenta o tempo de acesso e o preço por GB dos três tipos de memórias citados no texto.

Como pode ser observado na Tabela 3.1, a memória SRAM pode ser até 100 vezes mais rápida do que uma memória DRAM. Entretanto, o custo por *bit* com relação às

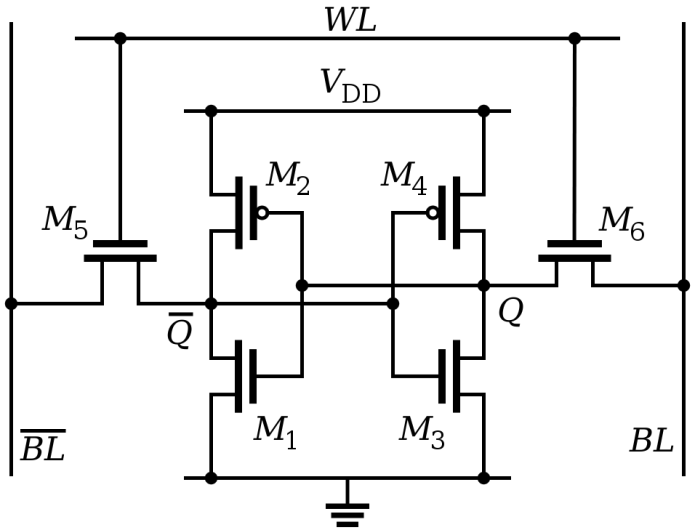


Figura 3.3: Estrutura de 1-*bit* em uma memória CMOS 6T-SRAM (Kulkarni et al., 2007).

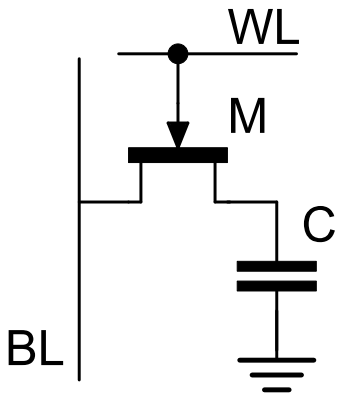


Figura 3.4: Estrutura de 1-*bit* em uma memória 1T1C-DRAM.

Tabela 3.1: Tempo de acesso médio à memória e custo por GB (Dean, 2009; Patterson e Hennessy, 2005).

Tipo da memória	Tempo de acesso médio	Custo por GB (*USD*)
SRAM	0,5 — 5 ns	2.000 — 5.000
DRAM	20 — 50 ns	5 — 10
Disco Magnético	5.000.000 — 20.000.000 ns	0,14 — 0,20

outras tecnologias é extremamente elevado. Este é um dos motivos pelo qual não se tem *caches* de tamanhos grandes, sempre na ordem de *KBytes* e em processadores mais modernos, de alguns *MBytes*. Outros motivos importantes que devem ser considerados e que influenciam na escolha do tamanho da *cache* estão relacionados com o aumento na complexidade do controlador da *cache* à medida que a *cache* cresce de tamanho e também com o consumo de energia do processador.

Hoje em dia quase todos os processadores utilizam uma *cache* integrada ao *die* do processador. A Figura 3.5 mostra o diagrama interno do processador Intel Pentium, onde a *cache* de instrução é chamada de *code cache* e a *cache* de dados é chamada de *data cache*. Nota-se que as *caches* ocupam aproximadamente 30% de toda a área do processador.

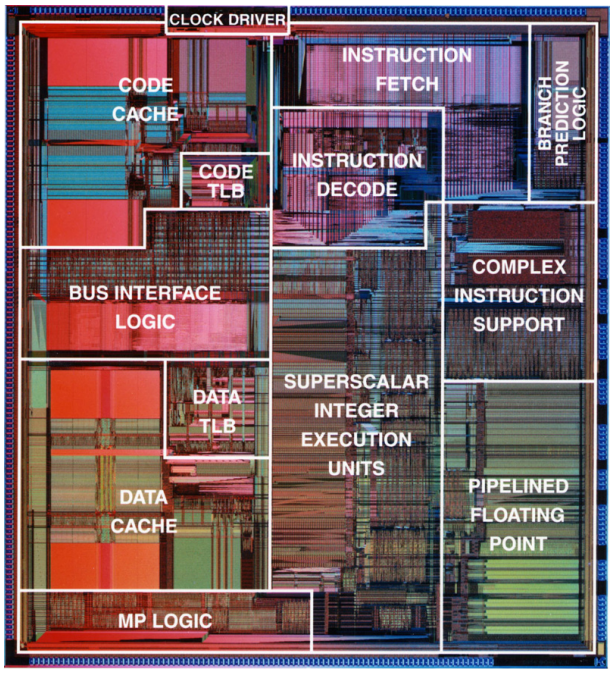
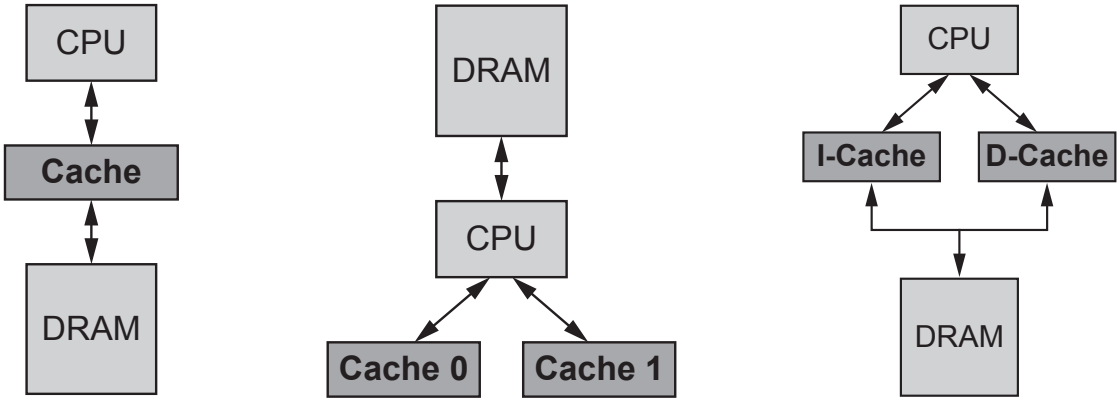


Figura 3.5: Diagrama interno do processador Intel Pentium (Computer History, 2012).

A hierarquia mais comum é manter a *cache* entre o processador e a memória principal (Jheng et al., 2010), Figura 3.6(a), mas existem casos, especialmente em processadores DSP, em que a *cache* pode ser acessada de forma paralela à memória principal, Figura 3.6(b). A *cache* pode ainda ser organizada de forma separada, uma apenas para armazenar referências às instruções e outra para armazenar referências aos dados, como ilustrado na Figura 3.6(c). A grande maioria dos processadores implementa *caches* independentes, uma para as instruções e outra para os dados para evitar a ocorrência de conflitos estruturais entre os diferentes estágios do pipeline de instruções do processador (Zhang et al., 2004).

3.2 Detalhes de operação

Quando o processador precisa ler ou escrever em um determinado endereço na memória principal, ele primeiro verifica a *cache* à procura deste endereço. Se o endereço estiver na *cache*, significa que o dado correspondente à aquele endereço também está armazenado na *cache*, e portanto, o processador lê ou escreve o dado direto na *cache*, operação que é muito mais rápida se comparada ao tempo de acesso à memória principal. Neste caso, houve a ocorrência de um *cache hit*, ou seja, o dado encontra-se armazenado na *cache*. Se



(a) *Cache* de propósito geral.

(b) *Cache* comum em processadores DSP (*scratchpad memory*).

(c) *Caches* independentes (instruções e dados).

Figura 3.6: Diferentes tipos de organização da memória *cache* (Jacob et al., 2007).

o dado não está na *cache* chamamos de *cache miss*, o que implica copiar o dado presente da memória principal para a *cache*, ou vice-versa. Essa operação possui alto custo em relação ao tempo de acesso.

A proporção entre a quantidade de acessos a *cache* que resultaram em *cache hit* e a quantidade total de acessos a *cache* é denominada *hit rate*, e é uma medida muito importante para avaliar o desempenho da *cache* dada uma aplicação ou algoritmo. A proporção entre a quantidade de *cache miss* e a quantidade total de acessos é chamada de *miss rate*, e também é um parâmetro importante para avaliar o desempenho da *cache*.

3.2.1 Cache Read

A operação de *cache read* define-se quando é requisitada uma leitura de um endereço à memória principal. Neste contexto, pode-se ter *read hit*, quando o dado está presente na *cache*, ou *read miss*, quando o dado não está presente na *cache*. Para que seja possível encontrar um determinado dado na *cache*, o endereço ou sub-partes do endereço deste dado também precisam ser armazenados. A este endereço é dado o nome de *tag*. Como pode ser observado na Figura 3.7, a *cache* contém um subconjunto de dados da memória principal e utiliza o valor do *tag* para indexar o dado.

Se a *tag* a ser procurada estiver na *cache*, temos um *cache hit* e o dado correspondente é retornado ao processador. No caso de um *cache miss*, o dado será copiado da memória principal para a *cache*, entretanto esta operação envolve alocar espaço dentro da *cache* para armazenar o novo par *tag-data*. Para isso, a *cache* irá remover um elemento interno existente e alocar o novo par. A heurística ou algoritmo utilizado para escolher qual elemento retirar da *cache* é chamado de *replacement policy* ou política de substituição.

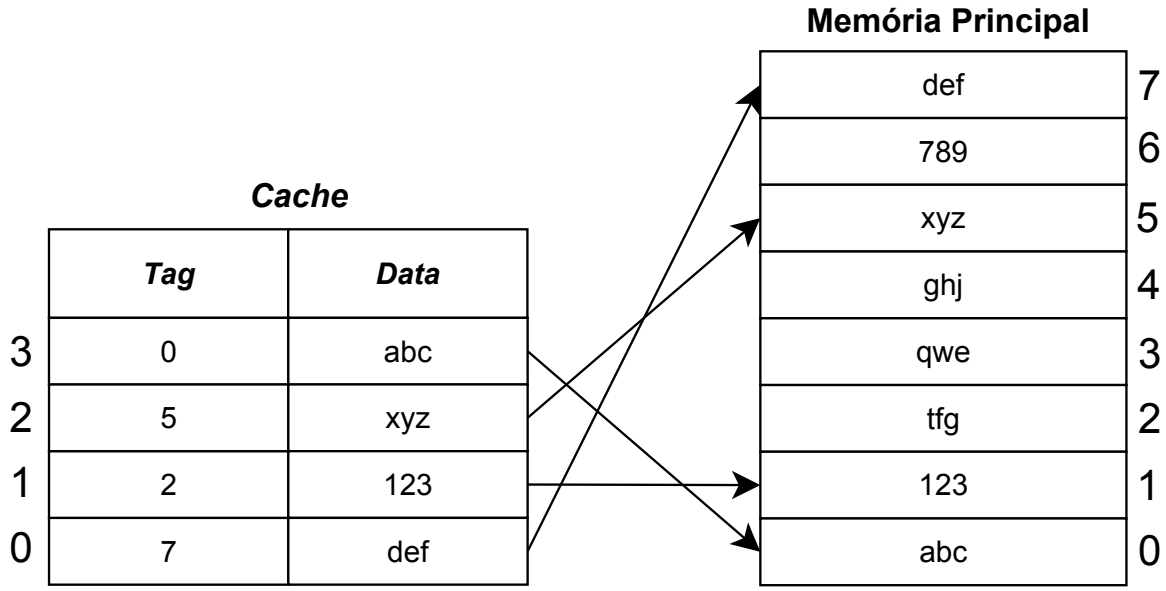


Figura 3.7: Mapeamento entre a *cache* e a memória principal.

O problema fundamental de qualquer algoritmo de substituição é que deve prever qual elemento não será mais utilizado no futuro, sendo este elemento o melhor candidato a ser retirado. Como ainda não é possível prever o futuro, os algoritmos de substituição implementam heurísticas capazes de identificar a probabilidade de um elemento não ser referenciado mais a frente e assim podem tomar a decisão de remover o de maior probabilidade, por exemplo. Um algoritmo muito popular e que também é conhecido em outras áreas da computação é o *Least Recently Used* (LRU) ou o Menos Recentemente Usado. Este algoritmo remove da *cache* o elemento que está há mais tempo sem ser referenciado. Neste caso, a *cache* inclui *bits* extras em cada elemento para manter o histórico de acesso daquele elemento. Existem ainda outros algoritmos como o *Random*, que escolhe aleatoriamente um elemento a ser removido e o *Most Recently Used* (MRU), que, ao contrário do LRU, irá remover o elemento mais recentemente utilizado. O algoritmo de substituição tem grande influência no desempenho da *cache*, pois pode diminuir drasticamente a quantidade de *cache miss* e aumentar o *cache hit*.

3.2.2 Cache Write

A operação de *cache write* define-se quando é requisitada uma escrita de um determinado dado a um endereço da memória principal. Neste cenário, em algum momento o dado deve ser escrito na memória principal, e quem define este momento é a política de escrita, ou *write policy*, adotada pela *cache*. Assim, como ocorre com a operação de *read*, onde se pode ter *read hit* ou *read miss*, na operação de *write* tem-se *write hit* ou *write miss*.

3.2.2.1 Write hit

No *write hit*, o endereço de memória a ser escrito já está presente na *cache* e neste contexto são duas as políticas mais comuns: *write-through* e *write-back*. A primeira é mais simples de ser implementada, pois não requer hardware adicional. Nesta política, o dado é escrito na *cache* e imediatamente repassado à memória principal. Já na política *write-back*, a escrita do dado na memória principal não é imediata, ela ocorre apenas dentro da própria *cache*. Entretanto, para implementar esta política, a *cache* deve marcar internamente quais elementos foram sobrescritos (*dirty*). Na eventual remoção de um elemento marcado como sobrescrito (*dirty*), o dado deste elemento é escrito na memória principal. Após este processo, pode-se dizer que há coerência entre o dado da *cache* e o dado na memória principal.

3.2.2.2 Write miss

No *write miss*, o endereço de memória a ser escrito não está presente na *cache* e neste contexto são duas as políticas mais comuns: *write-allocate* e *no-write allocate*. Na primeira, como o dado não está presente na *cache*, ele é primeiro carregado para a *cache* para então ter-se um *write hit*. Esta política é similar ao procedimento executado quando há ocorrência de um *read miss*. Na segunda política, o dado não é carregado para dentro da *cache* e a operação de escrita deve ocorrer diretamente na memória principal. Nesta política, apenas os dados resultantes de operações de leituras são armazenados dentro da *cache*. Apesar de ser possível utilizar qualquer combinação entre as políticas de *write hit* e *write miss*, faz mais sentido se elas forem combinadas da seguinte forma:

- **Write-back** com **write-allocate**: buscam-se primeiramente os dados da memória principal para que escritas subsequentes sejam realizadas apenas na *cache*, sendo necessário apenas atualizar o *dirty bit*.

- **Write-through** com **no-write allocate**: pode não fazer sentido utilizar *write-allocate* já que toda requisição de escrita irá imediatamente para a memória principal. Entretanto, alocar o dado na *cache* pode contribuir para melhores taxas de *read hit*.

O fluxograma da Figura 3.8 representa uma *cache* com as políticas de *write-through* e *no-write allocate*. Para as operações de escrita, também inclui as operações de leitura em função do *read hit* ou *miss*. Quando há um *write hit*, o dado é escrito primeiro na *cache* e depois escrito imediatamente na memória principal. Caso se tenha um *write miss*, o dado é escrito apenas na memória principal.

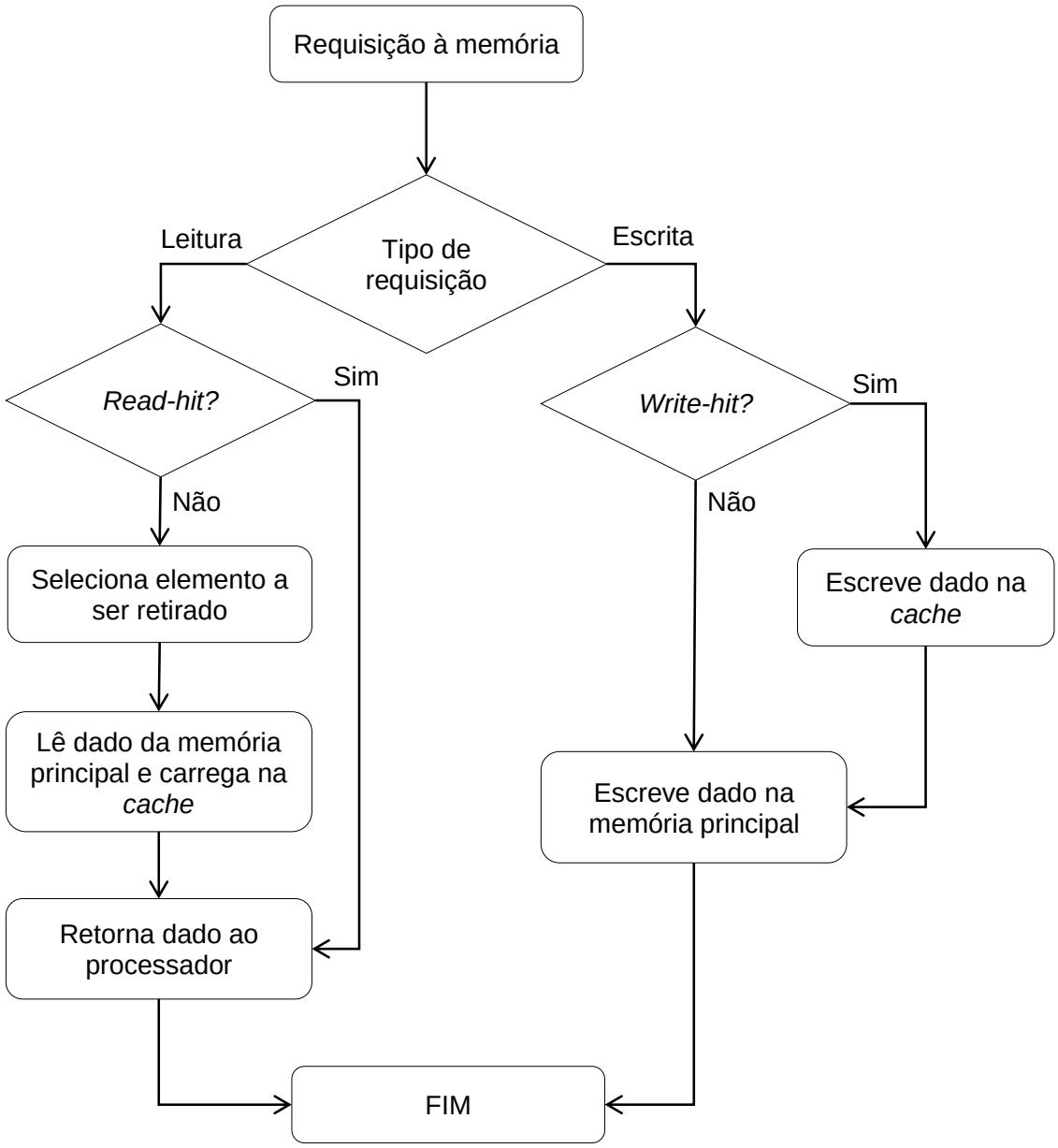


Figura 3.8: Fluxograma de uma *cache write-through* com *no-write allocate*, incluindo as operações de leitura e escrita.

O fluxograma da Figura 3.9 representa uma *cache* com as políticas de *write-back* e *write-allocate*. Quando há um *write hit*, o dado é escrito apenas na *cache* e o *dirty-bit* é ativado, indicando que aquele bloco de dados está incoerente com a memória principal.

Caso se tenha um *write miss*, como agora a política é *write-allocate*, devemos alocar um bloco da *cache* aos novos dados. Deve-se então selecionar um bloco da *cache* para ser removido de acordo com a política de substituição e verificar se o *dirty-bit* deste bloco está ativo. Se estiver, o bloco a ser removido deve ser escrito na memória principal. Em seguida, o bloco de dados referente ao endereço a ser escrito é carregado da memória

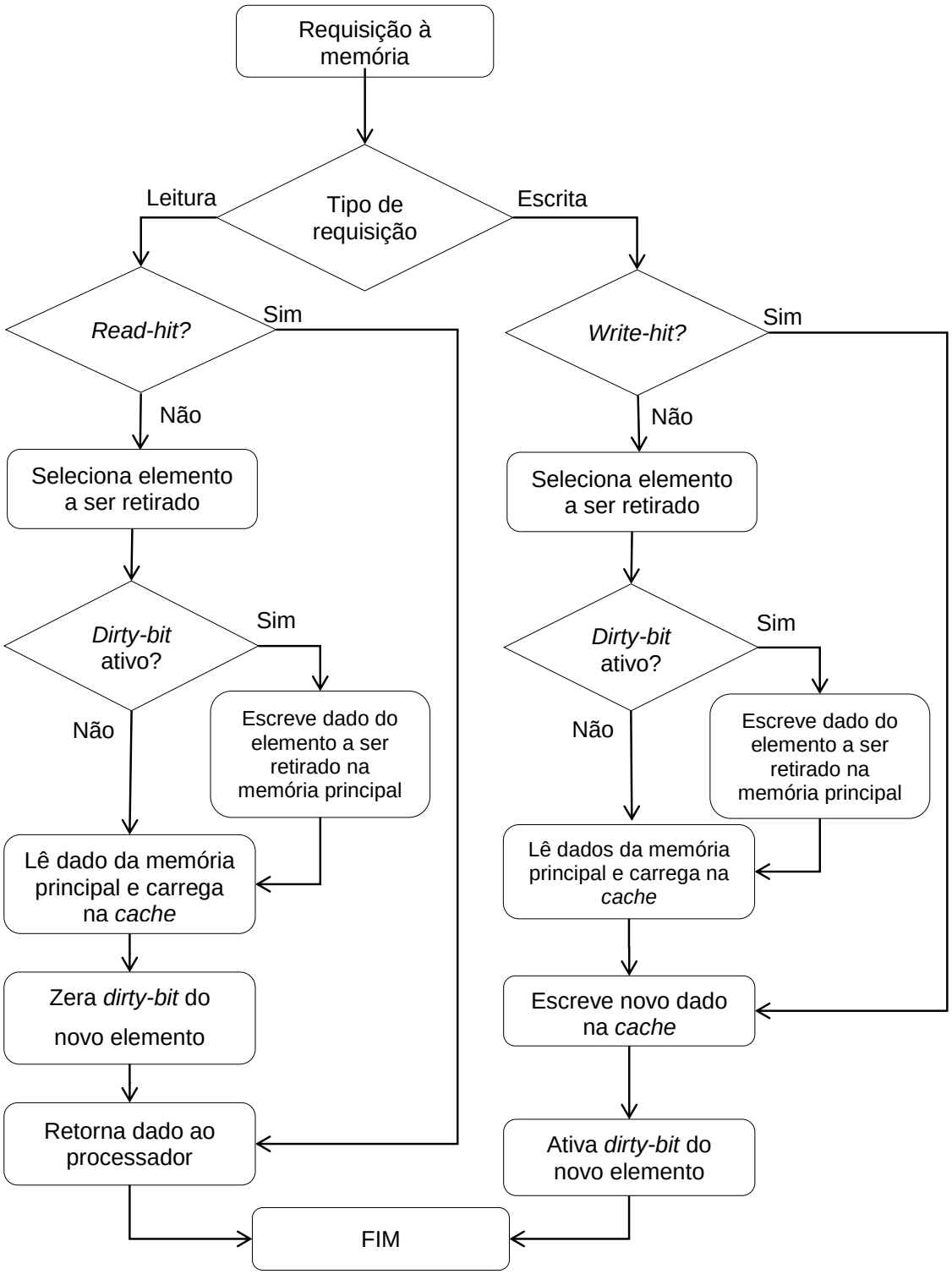


Figura 3.9: Fluxograma de uma *cache write-back* com *write-allocate*, incluindo as operações de leitura e escrita.

principal para dentro da *cache*, gerando um *write hit*. Neste momento, o novo dado é

escrito diretamente na cache e o *dirty-bit* é ativado. Com essa política também se deve checar o *dirty-bit* na ocorrência de um *read miss*.

3.2.3 Categorias de Cache Miss

Várias análises em relação ao comportamento da *cache* foram realizadas ao longo dos últimos anos na tentativa de se encontrar a melhor combinação entre tamanho da *cache*, tamanho da linha e associatividade (estes elementos são apresentados nas seções 3.2.4 e 3.2.5). Uma das contribuições mais significantes na realização destas análises foi feita por Mark Hill, que categorizou o *cache miss* em três tipos diferentes, modelo conhecido como Três Cs (3Cs) (Hill e Smith, 1989):

- A primeira categoria é conhecida como erros compulsórios, ou do inglês *compulsory misses*. Este tipo de *miss* é causado quando se faz referência ao dado pela primeira vez. O tamanho da *cache* e a associatividade não interferem na quantidade de *compulsory misses*. Uma forma de reduzí-los seria através de técnicas de *prefetching*, onde o dado é carregado para a *cache* mesmo sem ser requisitado. O tamanho da linha pode ajudar a reduzir o número de *compulsory misses*.

- A segunda categoria é conhecida como erros de capacidade, ou do inglês *capacity misses*. Estes *misses* são causados quando a *cache* simplesmente não tem tamanho suficiente para alocar todos os blocos de dados necessários para a execução de um programa. O tamanho da *cache* é o único fator que interfere na quantidade de *misses* deste tipo.

- A terceira e última categoria é conhecida como erros de conflito, ou do inglês *conflict misses*. Estes *misses* são causados quando duas ou mais linhas que são muito utilizadas são mapeadas na mesma posição na *cache*. Por exemplo, no caso de duas linhas que são muito utilizadas e mapeadas na mesma posição, para que a primeira linha possa ser armazenada, a segunda precisa ser removida. Entretanto, para que a segunda linha possa ser armazenada, a primeira precisa ser removida. Ou seja, há disputa entre as linhas pela mesma posição na *cache*. Pode-se reduzir a quantidade deste tipo de *miss* com a utilização de *caches* associativas ou com o aumento do tamanho da *cache* no caso de uma *cache* do tipo *direct-mapped*. Em *caches* associativas, a política de substituição também pode influenciar na quantidade de *conflict misses*.

3.2.4 Estrutura de um elemento da cache

Um elemento da *cache* geralmente segue a estrutura proposta na Figura 3.10. O campo *tag* contém o endereço ou sub-partes do endereço do bloco de dados *data block*. O campo *data block* contêm os dados provenientes da memória principal. O campo *flag bits* pode armazenar *bits* para controle do elemento, como por exemplo, o *dirty-bit* e/ou informações extras necessárias para a execução do algoritmo de substituição.

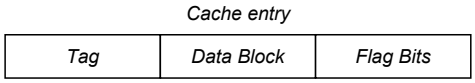


Figura 3.10: Estrutura de uma elemento na *cache*.

O tamanho da *cache*, ou *cache size*, é definido como a quantidade de dados que a *cache* consegue armazenar e pode ser calculado pela multiplicação da quantidade de *bytes* de um *data block* pela quantidade de blocos que a *cache* suporta. Por exemplo, para uma *cache* que consegue armazenar 32 *bytes* em um *data block* e que suporta até 64 blocos, o tamanho da *cache* é de 2 *Kbytes*. O tamanho da linha da *cache*, ou *cache line*, é definido como a quantidade de *bytes* de um *data block*. Neste caso, a *cache* teria tamanho de 2 *KBytes* e tamanho de linha de 32 *bytes*.

Quando o processador faz uma requisição de leitura ou escrita à memória principal, considerando a hierarquia de memória da Figura 3.6(a) ou 3.6(c), a *cache* será acessada à procura do dado requisitado. Para saber onde o dado foi alocado internamente à *cache*, utiliza-se o valor do índice que é extraído diretamente do endereço do dado, conforme ilustrado na Figura 3.11.

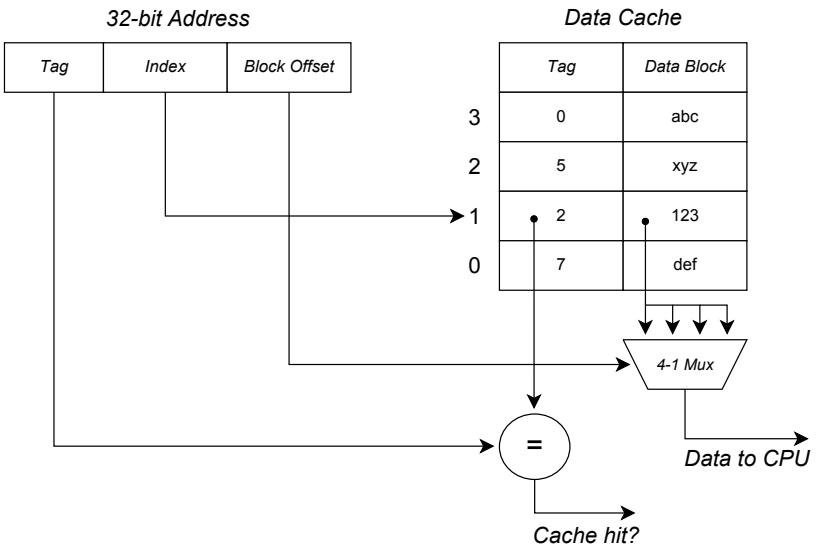


Figura 3.11: Diagrama de uma *cache direct-mapped*.

O mesmo ocorre quando a *cache* precisa alocar o dado internamente. O valor da *tag* também é extraído do endereço. Na operação de leitura, se a *tag* do endereço for igual à *tag* armazenada na *cache* tem-se um *read hit* e o valor do campo *block offset* é utilizado para indicar qual dado de dentro de *data block* devemos retornar ao processador. Caso as *tags* sejam diferentes, temos um *read miss* e a *cache* deve buscar o dado direto da memória principal.

3.2.5 Associatividade

No diagrama da Figura 3.11, cada valor do campo *index* sempre é mapeado para a mesma posição na memória *cache*. Por exemplo, se o valor de *index* for "1", este valor sempre será mapeado para a posição "1" na memória *cache*. A este tipo de *cache* é dado o nome de *direct-mapped*. A política de substituição não faz sentido neste tipo de *cache* já que o mapeamento é um-para-um. Há ainda outro tipo de organização que irá mapear cada valor do campo *index* para várias posições na memória *cache*, ou seja, um mapeamento um-para-muitos. A esse tipo de *cache* é dado o nome de *cache N-way set associative*, onde N é a quantidade de posições diferentes em que se pode alocar o dado. A Figura 3.12 apresenta uma *cache 2-way set associative*, pois cada bloco de dados pode ser alocado em duas posições físicas distintas na memória *cache*.

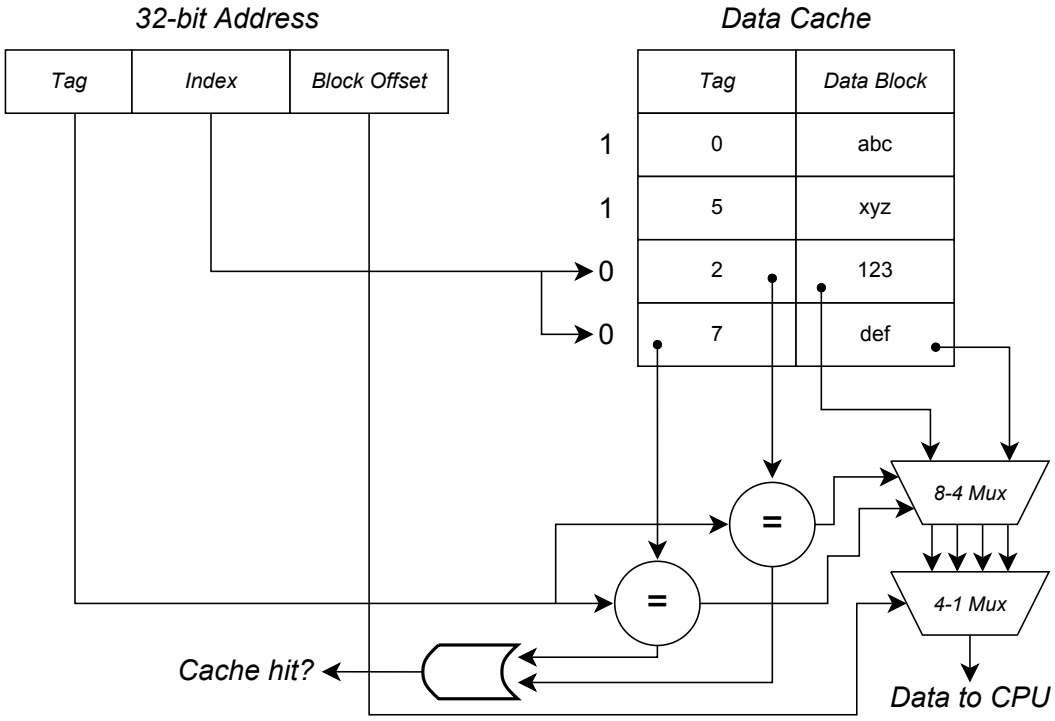


Figura 3.12: Diagrama de uma *cache 2-way set associative*.

Por fim, há a *cache* chamada de *fully associative*, Figura 3.13, que mapeia todos os possíveis valores de *index* à mesma posição na memória *cache*, mapeamento muitos-para-um. Ou seja, o bloco de dados pode estar em qualquer lugar na *cache*. A associatividade da *cache* tem a vantagem de reduzir o número de *cache miss* da categoria *conflict*, pois dois ou mais blocos de dados em que o valor de *index* é o mesmo podem agora ser armazenados em lugares diferentes e ao mesmo tempo. Não é necessário remover um para alocar o outro. Por outro lado, o *overhead* da associatividade pode não compensar a redução no número de *cache misses*. Na leitura, se um bloco de dados pode estar em dois lugares diferentes, é necessário checar ambos os lugares. Na Figura 3.12, as *tags* nas posições "0" e "1" devem ser checadas para encontrar o bloco de dados desejado. Procurar em mais lugares torna o hardware mais complexo e pode aumentar o tempo de acesso e o consumo de energia.

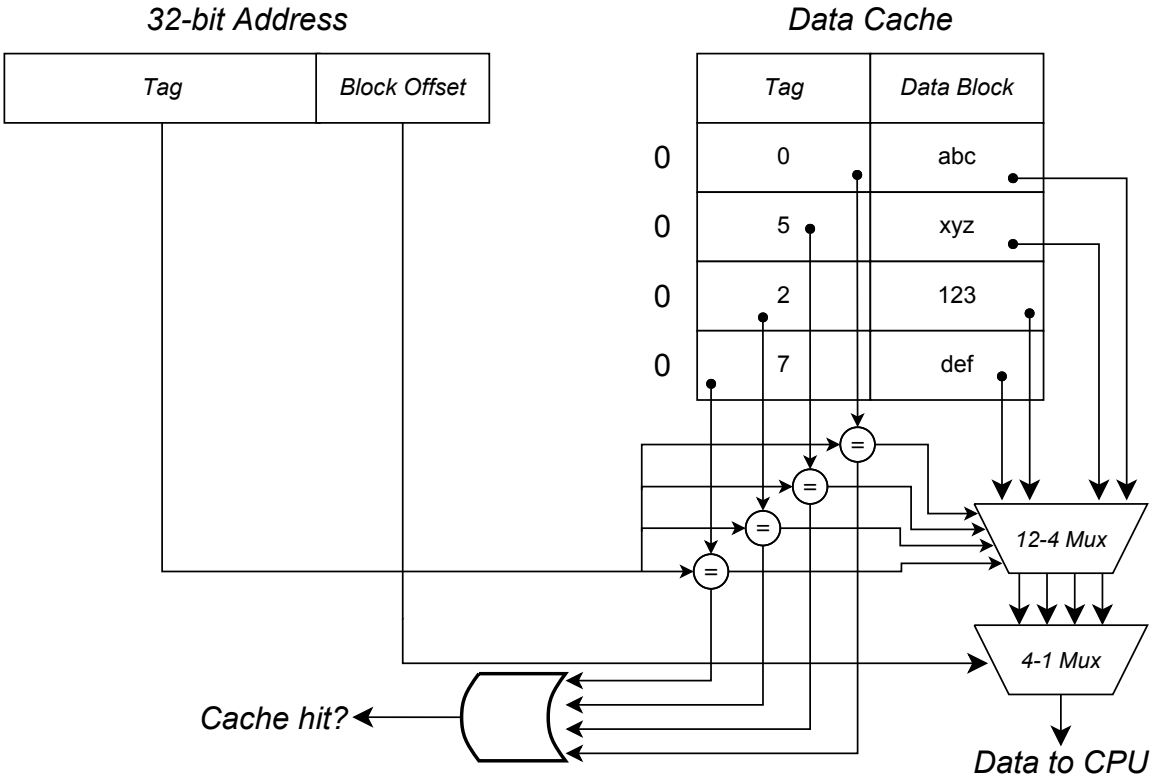


Figura 3.13: Diagrama de uma *cache fully associative*.

Capítulo

4

Material e Método

Este capítulo apresenta uma descrição das plataformas de desenvolvimento e do processador LEON3 e os métodos que foram utilizados durante a elaboração deste trabalho.

4.1 Plataformas de Hardware e FPGA

Foram utilizados dois hardwares diferentes para a implementação da arquitetura proposta. O primeiro é contido no *kit* de desenvolvimento ML605, edição Virtex-6, fornecido pela empresa Xilinx. Esse *kit* é composto por uma placa de desenvolvimento que possui um *Field-Programmable Gate Array* (FPGA) (Bobda, 2007) Virtex-6 XC6VLX240T, com 241.152 células lógicas, 14.976 *KBits* de memória *on-die* e 768 blocos para processamento de sinais digitais, *Digital Signal Processing* (DSP). Cada bloco DSP contém um multiplicador 25x18 *bits*, um somador e um acumulador. A placa possui ainda um *socket* DDR3 SO-DIMM para suporte de memória externa, porta *Gigabit Ethernet* e *PCI Express Gen2*. A Figura 4.1 mostra a placa de desenvolvimento da Xilinx.

O segundo hardware é contido no *kit* de desenvolvimento *Stratix IV GX FPGA Development Kit* fornecido pela empresa Altera. Esse *kit* é composto por uma placa de desenvolvimento que possui o FPGA Stratix IV GX, com 228.000 elementos lógicos, 17.133 *Kbits* de memória *on-die* e 1288 blocos para processamento de sinais digitais. A placa também possui *socket* DDR3 SO-DIMM com suporte a até 512 *Mbyte* de memória DDR3 SDRAM. Ambas as placas são bastante similares pois possuem basicamente os mesmos

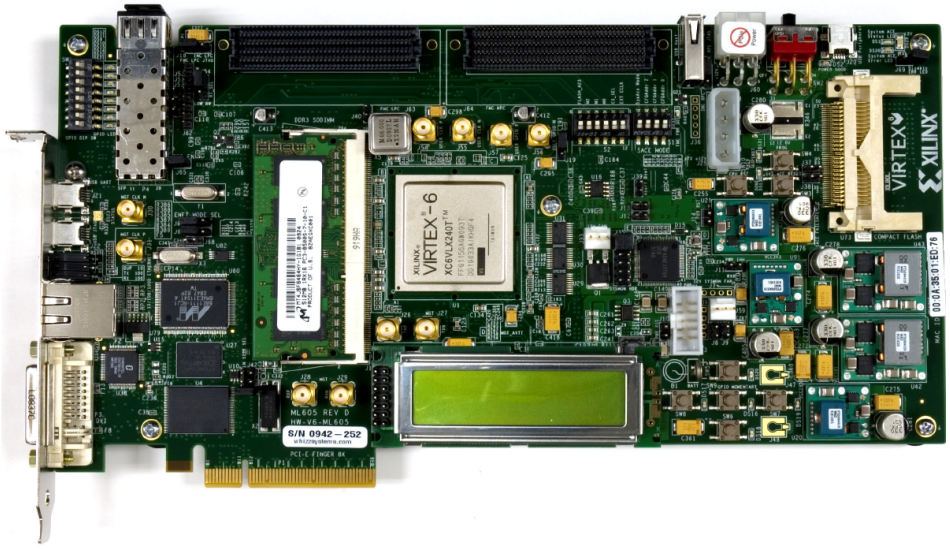


Figura 4.1: Placa de desenvolvimento Xilinx ML605 (Xilinx, 2012b).

periféricos. Apesar de serem FPGAs de fabricantes diferentes e utilizarem nomenclaturas diferentes para descreverem os elementos internos do FPGA, a quantidade de hardware que pode ser alocado em cada FPGA é basicamente a mesma. Inclusive a quantidade de memória *on-die* de cada FPGA é praticamente a mesma. A Figura 4.2 mostra a placa de desenvolvimento da Altera. A placa da Altera contém ainda um CPLD MAX II com 2210 elementos lógicos. Este CPLD pode ser programado, por exemplo, como um monitor do consumo de energia da FPGA.

Figura 4.2: Placa de desenvolvimento Altera Stratix IV GX (Altera, 2012).

4.2 Processador LEON3

Neste projeto foi utilizado o processador LEON3 da Aeroflex Gaisler (Gaisler, 2012) com frequência em 100 MHz. O LEON3 é um tipo de processador soft-core de 32-*bits* escrito totalmente em linguagem *Verilog Hardware Description Language* (VHDL) sintetizável em FPGA e que possui arquitetura compatível com o SPARC V8 (SPARC International, 1992). Este soft-core foi escolhido para ser utilizado neste trabalho, pois é altamente configurável, de código disponível sob a licença GNU GPL, o que permite modificações livres em seu código fonte, e por possuir documentação do sistema LEON3 bem detalhada. Essas características são imprescindíveis para o sucesso do projeto uma vez que é necessário alterar aspectos avançados na programação da memória e do controlador da *cache*. A Figura 4.3 apresenta a arquitetura interna do processador LEON3. O foco deste trabalho será alterar o componente responsável por controlar a *cache* de dados do processador.

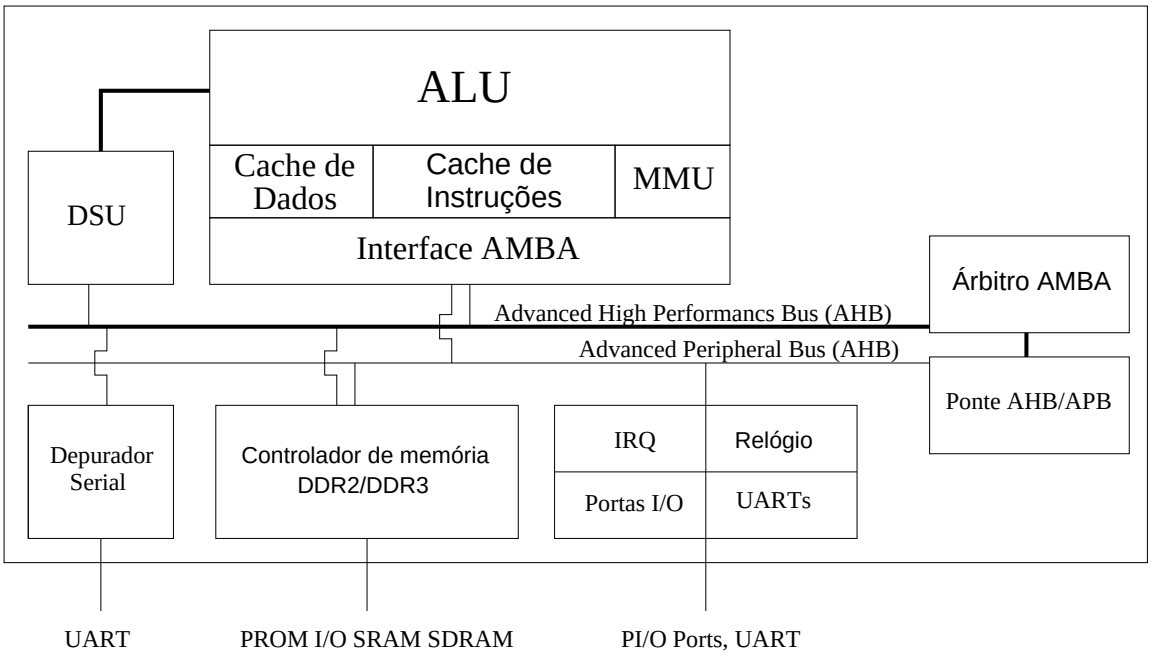


Figura 4.3: Diagrama interno simplificado do processador LEON3 (Eisele, 2002).

Um processador soft-core é um microprocessador totalmente descrito em software, geralmente em alguma linguagem de descrição de hardware como VHDL ou Verilog, e que pode ser sintetizado em dispositivos programáveis, como os FPGAs. Um processador soft-core, quando sintetizado para um FPGA, torna-se muito flexível, pois os parâmetros de configuração do processador e até as suas funcionalidades podem ser alteradas e regravadas conforme a necessidade (Plavec, 2004). O Altera NIOS II (Altera, 2011) e o Xilinx's MicroBlaze (Xilinx, 2008) são outros exemplos de soft-cores muito utilizados na área acadêmica, porém por serem produtos comerciais e de código fechado, a flexibili-

dade fornecida por ambos é muito limitada, não sendo possível a realização do trabalho proposto.

4.3 Ferramentas EDA

As ferramentas de *Electronic Design Automation* (EDA) são softwares utilizados para projetar sistemas eletrônicos como, por exemplo, placas de circuito impresso, circuitos integrados e síntese de código VHDL em *netlists* de portas lógicas. Neste projeto, são utilizadas as ferramentas EDA da empresa Xilinx e Altera. A ferramenta ISE Design Suite v13.4. da Xilinx é utilizada para sintetizar o código do LEON3 para a plataforma de desenvolvimento com o FPGA Virtex-6. Já para sintetizar o LEON3 para o FPGA da Altera é utilizado o Quartus II v12.1 SP1. Para a simulação do código fonte VHDL do LEON3 é utilizado a ferramenta ModelSim SE 10.0c da empresa Mentor Graphics.

4.4 Métodos de validação

O código fonte do LEON3 é bastante extenso e muito complexo, pois há diversos módulos do processador que interagem entre si. Ainda, por ser um processador que é disponibilizado comercialmente e não apenas para atividades acadêmicas, o código em VHDL é altamente otimizado, o que prejudica o entendimento imediato e a legibilidade do mesmo. Assim, qualquer alteração realizada no código fonte, mesmo que pequena, pode ter impacto negativo em outras partes do processador. Portanto, foi necessário validar todas e quaisquer alterações realizadas no código.

Foram duas as formas escolhidas para a etapa de validação:

- Simulação comportamental do processador, através da ferramenta ModelSim, antes e após as modificações.
- Execução exaustiva de determinadas aplicações, dependendo do tipo de modificação.

A simulação comportamental pode não ser suficiente em certas situações pois esta não simula o *timing* do processador. Entretanto, este tipo de simulação é bastante válida e foi utilizada extensivamente para validar pequenas alterações que tiveram que ser realizadas no processador. Se o comportamento do processador é igual antes e depois da modificação, esta pode ser validada. Esta simulação também foi bastante utilizada para se validar novos componentes de hardware que tiveram que ser desenvolvidos para integrar o LEON3 com o controlador de memória DDR3 da Altera.

A validação por execução exaustiva de aplicações foi utilizada quando as alterações do código do LEON3 eram mais complexas com possível impacto no *timing* do caminho

crítico. Foi utilizada esta técnica de validação no desenvolvimento dos contadores de *miss* e *hit* dentro do controlador da *cache* de dados do LEON3 e no desenvolvimento da técnica de *way-shutdown*, responsável pela reconfiguração da *cache* em tempo de execução, modificações realizadas também no controlador da *cache* de dados.

Para a validação do contador de *misses* e *hits* na *cache* de dados foi desenvolvida uma aplicação simples em C que permitisse contar manualmente a quantidade de *hits* e *misses* que a aplicação deveria ter ao final de sua execução. Se os valores dos contadores de *misses* e *hits* forem iguais ao da contagem manual, pode-se validar o contador. Esta aplicação tem o seguinte formato:

```
/* Define o vetor com alinhamento de 16 bytes pois cada linha da cache
*  neste caso consegue armazenar exatos 16 bytes.
*  Portanto o vetor inteiro cabe em uma única linha da cache.
*/

//O compilador GCC garante que o trecho de memória abaixo é continuo.
unsigned char cache_line_vector[16] __attribute__ ((aligned(16)));

unsigned char c;
unsigned char d;
unsigned char e;
unsigned char f;

//reseta os contadores de miss e hit da cache
reset_counters();

//Define o intervalo de endereços que o contador deve considerar
//Portanto apenas os acessos ao vetor serão contabilizados pelo contador.
set_dcache_loaddr_1((unsigned long) &cache_line_vector); //Menor endereço

//set upper address range 1
set_dcache_hiaddr_1((unsigned long) &cache_line_vector[15]); //Maior endereço

//Ativa o intervalo
dcache_en_range_1();

/* O controlador da cache de dados NÃO faz pre-fetching de uma linha inteira,
*  apenas 4 bytes por vez.
*  Esta primeira etapa valida o contador de read-hit e read-miss
*/

//Primeiro miss do tipo compulsory
//Os dados dos elemento 0, 1, 2 e 3 são colocados na cache pois
```

```
//fazem parte da mesma palavra.
c = cache_line_vector[0];    //read-miss

//Segundo miss do tipo compulsory devido à falta da técnica de line pre-fetching
//Os dados dos elemento 12, 13, 14 e 15 são colocados na cache pois
//fazem parte da mesma palavra.
d = cache_line_vector[15]; //read-miss

//Primeiro hit pois o dado já estava na cache
//devido acesso anterior ao elemento 0 do vetor.
e = cache_line_vector[3]; //read-hit

//Segundo hit pois o dado já estava na cache
//devido acesso anterior ao elemento 15 do vetor.
f = cache_line_vector[14]; //read-hit

//Neste ponto os elementos 0, 1, 2, 3, 12, 13, 14 e 15 estão na cache.

//Segunda etapa para validar o contador de write-hit
cache_line_vector[14] = 'a'; //write hit.
cache_line_vector[15] = 'b'; //write hit.
cache_line_vector[12] = 'b'; //write hit.
cache_line_vector[13] = 'b'; //write hit.

cache_line_vector[0] = 'a'; //write hit.
cache_line_vector[1] = 'b'; //write hit.
cache_line_vector[2] = 'b'; //write hit.
cache_line_vector[3] = 'b'; //write hit.

//Terceira etapa para validar o contador de write-miss
//Os elementos 4, 5, 6 e 7 NÃO são colocados na cache devido
//à política de no-allocate
cache_line_vector[7] = 'b'; //write miss.

//Os elementos 8, 9, 10 e 11 NÃO são colocados na cache devido
//à política de no-allocate
cache_line_vector[10] = 'b'; //write miss.

//Quarta etapa para validar a política de no-allocate
//Testando que os elementos 4, 5, 6 e 7 realmente não estão na cache
//Devemos ter um read miss
f = cache_line_vector[5];  //read miss
```

Deve-se ter no final da execução da aplicação acima os seguintes valores:

- Read-miss: 3
- Read-hit: 2
- Write-miss: 2
- Write-hit: 8

4.5 Aplicações

As seguintes aplicações foram utilizadas neste trabalho para avaliar, testar e validar a arquitetura de *cache* proposta.

- Algoritmo *Breadth-First Search*.
- Multiplicação de matrizes esparsas.
- Algoritmo *Push Relabel*.

Estas aplicações foram escolhidas pois são conhecidas por terem seu desempenho muito dependente da configuração da *cache* do processador.

4.5.1 Aplicação Breadth-First Search

Este algoritmo é bastante conhecido na área de estrutura de dados. É um algoritmo notavelmente ineficiente para lidar com grafos esparsos, uma vez que o padrão de acesso à memória é aleatório e não estruturado (Brodal et al., 2004). Desta forma, o algoritmo não explora o conceito de localidade espacial. A aplicação foi executada com 1024 nós.

4.5.2 Multiplicação de Matrizes Esparsas

A aplicação de multiplicação de matrizes esparsas também é outro algoritmo muito conhecido na área da álgebra linear. Devido à forma como é implementado, este algoritmo pode ser dividido em duas etapas. A primeira etapa realiza a decomposição (fase *Factor*) e a segunda etapa realiza a multiplicação da matriz (fase *Solve*). Por serem operações distintas, pode-se analisar cada uma das etapas de forma independente. A matriz utilizada neste algoritmo tem tamanho 700x700 com 200.000 valores. Cada valor pode variar entre -1000 e 1000. Os valores são escolhidos de forma pseudo-aleatória, sempre com a mesma semente. Portanto, a sequência de números pseudo-aleatórios é sempre a mesma em todas as execuções.

4.5.3 Algoritmo Push Relabel

O algoritmo *Push Relabel* é baseado no algoritmo de *Ford-Fulkerson* sendo um dos mais eficientes para se calcular vazão máxima em grafos ponderados. O algoritmo foi implementado com 1000 nós, onde cada aresta do grafo possui um peso gerado aleatoriamente, sempre com a mesma semente. Dado dois pontos no grafo, o objetivo do algoritmo é encontrar o caminho com maior vazão entre a fonte e o destino. Este é um problema bastante complexo e que também faz uso intenso da memória RAM, e como consequência, da memória *cache*.

Capítulo

5

Implementação da Arquitetura Proposta

Uma visão geral da arquitetura proposta neste projeto é demonstrada na Figura 5.1. Os módulos destacados em verde foram implementados por completo e os módulos destacados em azul são componentes existentes que sofreram alterações. Os módulos "Controlador de Memória DDR3 (Avalon-MM)" e "Ponte Avalon-MM/AMBA-AHB", como o próprios nomes indicam, são respectivamente o controlador de memória DDR3 da placa da Altera e uma ponte entre o barramento *Avalon-MM*, que a memória DDR3 da Altera utiliza, e o barramento *AMBA-AHB*, que é o barramento que o LEON3 utiliza. Estes módulos não estão presentes na versão do LEON3 implementada na placa da Xilinx. O controlador de memória DDR3 da Xilinx já vem configurado nas versões mais atuais do LEON3. Até a finalização deste trabalho, não havia nenhuma configuração do LEON3 disponível no site da Gaisler, empresa responsável pelo LEON3, capaz de utilizar o controlador de memória DDR3 da Altera. Sendo assim, a integração do LEON3 com o controlador de memória DDR3 da Altera realizada neste trabalho é inédito. A seguir serão detalhados todos os módulos da Figura 5.1 que foram implementados e alterados durante o desenvolvimento deste trabalho.

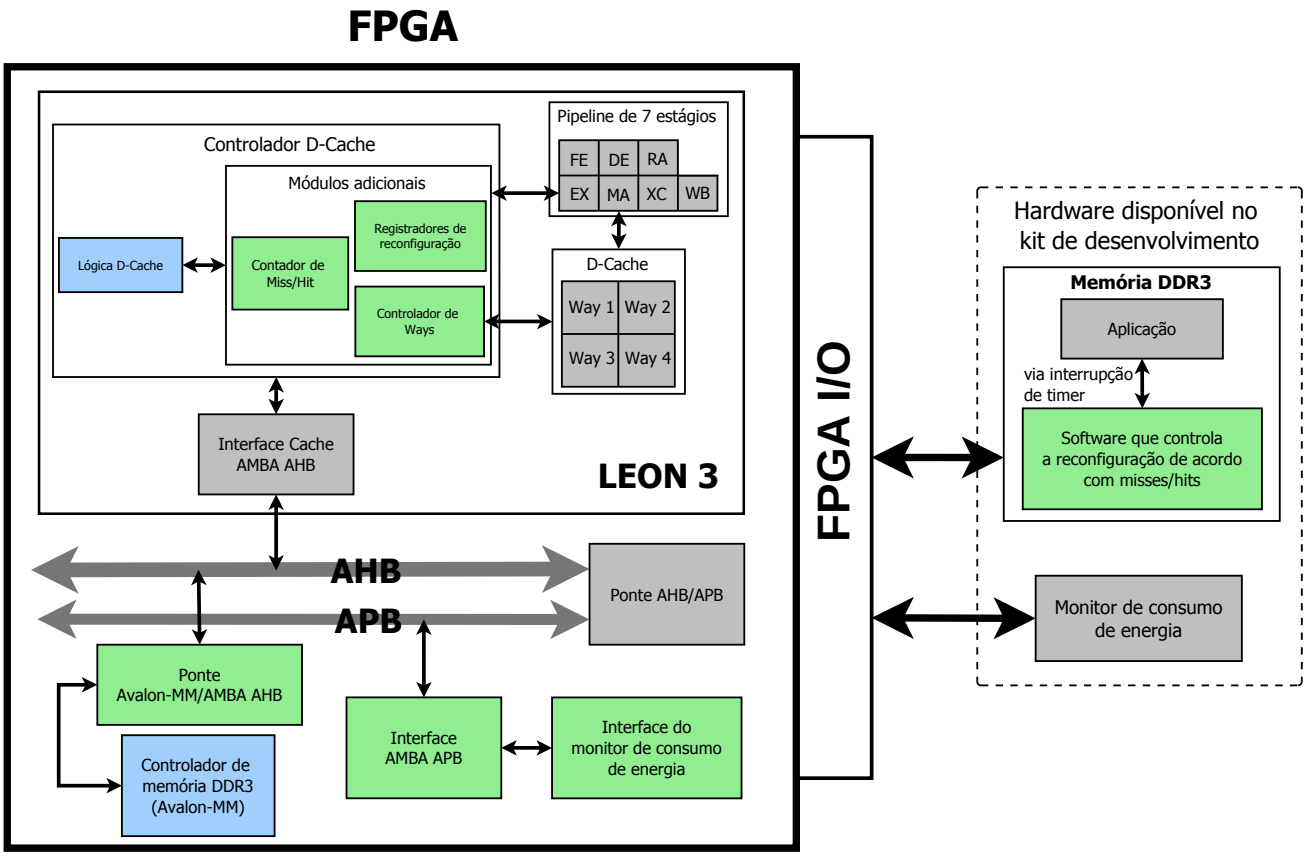


Figura 5.1: Visão geral da arquitetura proposta implementada nos *kits* de desenvolvimento da Xilinx e Altera.

5.1 Contadores de miss e hit

De acordo com os objetivos do projeto especificados no Capítulo 1, a reconfiguração da *cache* deve ser realizada em tempo de execução na tentativa de se reduzir o consumo de energia. O primeiro passo portanto foi definir quais métricas utilizar para ativar a reconfiguração da *cache* de dados. As taxas de *read miss* e *read hit* da *cache* de dados foram as métricas escolhidas. O LEON3 atualmente não implementa nenhuma funcionalidade que indique a quantidade de *hits* e *misses* da aplicação em execução. Logo, foi necessário implementar como parte do controlador da *cache* de dados do LEON3 um módulo capaz de contar a quantidade de *read misses* e *read hits*. Este módulo é indicado pelo componente "Contador de Miss/Hit" na Figura 5.1.

Os contadores foram implementados dentro do controlador da *cache* de dados do LEON3 em forma de registradores de 32-*bits*. São 4 no total: *read miss*, *read hit*, *write miss* e *write hit*. Os últimos 2 contadores não são utilizados atualmente mas estão disponíveis caso seja interessante considerá-los em trabalhos futuros.

```
-- DATA CACHE HIT/MISS COUNTER BLOCK --

read_miss_counter       : std_logic_vector(31 downto 0); -- dcache read miss_counter
read_hit_counter        : std_logic_vector(31 downto 0); -- dcache read hit_counter
write_miss_counter      : std_logic_vector(31 downto 0); -- dcache write miss_counter
write_hit_counter       : std_logic_vector(31 downto 0); -- dcache write hit_counter
high_addr_1             : std_logic_vector(31 downto 0); -- upper limit address 1
low_addr_1              : std_logic_vector(31 downto 0); -- lower limit address 1
en_range_1              : boolean;                       -- enable range 1
proc_ctx                : ctxword;                       -- process context
```

Os registradores *read_miss_counter* e *read_hit_counter* armazenam os valores de *misses* e *hits*. Estes foram inseridos no trecho de código VHDL do controlador da *cache* de dados onde ocorre a checagem de *miss* e *hit*. Se houver um *read miss*,

```
read_miss_counter = read_miss_counter + 1
```

se houver um *read hit*

```
read_hit_counter = read_hit_counter + 1
```

Os registradores *low_addr_1* e *high_addr_1* indicam que apenas os endereços dentro do intervalo *low_addr_1* <= *endereço* <= *high_addr_1* serão considerados pelos contadores. Assim, pode-se por exemplo, monitorar precisamente a quantidade de *misses* e *hits* de um vetor na memória. O registrador *en_range_1* apenas controla se os contadores estão ativos ou não.

A *cache* de dados do LEON3 é do tipo *Virtually Indexed, Virtually Tagged (VIVT)*, ou seja, é utilizado o endereço virtual tanto para indexar a *cache* como para os valores da *tag*. Este tipo de *cache* tem a vantagem de ser mais rápida, pois não é necessário consultar a *MMU* para acessar a *cache*. Entretanto, há duas desvantagens: os problemas de sinônimos e homônimos. Os problemas de sinônimos ocorrem quando diferentes endereços virtuais são na verdade mapeados para os mesmos endereços físicos. Este problema deixa a *cache* em um estado inconsistente pois caso aconteça uma escrita em um destes endereços virtuais, apenas 1 linha da *cache* ficará consistente com a memória principal, invalidando todas as outras linhas que apontam para o mesmo endereço físico. Uma das formas de resolver este problema é executar a operação de *flush* na *cache* a cada troca de contexto. Já o problema de homônimos é mais simples de ser solucionado. Este problema ocorre quando um mesmo endereço virtual é mapeado em diferentes endereços físicos. Para solucionar este problema basta adicionar para cada linha da *cache* e na *MMU* um identificador de contexto. Se a linha da *cache* que está sendo acessada possui o mesmo contexto que a *MMU*, então o dado presente nesta linha da *cache* é referente ao processo

que fez a requisição. Por este motivo, foi necessário incluir no módulo contador de *miss* e *hit* o registrador *proc_ctx*, para indicar se o processo atual em execução pelo processador é o processo que se quer monitorar. Apesar de ambos os problemas aparecerem apenas quando há a presença de um Sistema Operacional (SO), e neste projeto não há SO, o módulo contador já está preparado caso seja preciso adicionar um SO em trabalhos futuros.

A arquitetura SPARC v8, que é a arquitetura implementada pelo LEON3, define duas instruções especiais, *lda* e *sta*, semelhantes as instruções *load* e *store* mas que levam em consideração um parâmetro extra conhecido como identificador de espaço de endereçamento, ou do inglês, *Address Space Identifier (ASI)*. Este parâmetro *ASI* indica que a operação de *load* ou *store* irá acessar um espaço de endereçamento diferente do usual. Por exemplo, uma operação de *load* no endereço 0xA sem *ASI* é diferente da mesma operação de *load* mas com ASI=1. O endereço 0xA sem ASI não é o mesmo que o endereço 0xA com ASI=1. Como pode ser observado pela Figura 5.1, o controlador da *cache* de dados recebe a instrução em execução do *pipeline* no estágio EX, que significa *Execution*. Portanto, é possível mapear na memória os registradores referentes aos contadores de *miss* e *hit* com um valor de ASI a escolher. Desta forma, não há interferência no espaço de endereçamento da aplicação. Esta técnica também foi utilizada para criar alguns comandos dentro do controlador da *cache* de dados, como por exemplo, o comando para resetar apenas os contadores e o comando para resetar todos os parâmetros do módulo contador, inclusive os contadores.

O valor ASI=0x1A foi escolhido para mapear os registradores dos contadores na memória, sendo que cada registrador possui o seguinte endereço:

- *read_miss_counter*: 0x10
- *read_hit_counter*: 0x20
- *write_miss_counter*: 0x100
- *write_hit_counter*: 0x110

Para os comandos, são definidos os seguintes endereços:

- *reset all*: 0x30
- *reset counters only*: 0x34

Finalmente, os valores podem ser acessados pela aplicação executando instruções em *assembly* na seguinte forma:

```
        .text
        .align 64
        .global command_mhblock_lda
command_mhblock_lda:
        retl
        lda     [%o0] 0x1a, %o0

        .global command_mhblock_sta
command_mhblock_sta:
        retl
        sta     %o1, [%o0] 0x1a
```

ou através de um *wrapper* em código C:

```
extern inline int command_mhblock_lda(int cmd);
extern inline void command_mhblock_sta(int cmd, int value);

#define get_dcache_read_miss()      command_mhblock_lda(0x10)
#define get_dcache_read_hit()       command_mhblock_lda(0x20)
#define get_dcache_write_miss()     command_mhblock_lda(0x100)
#define get_dcache_write_hit()      command_mhblock_lda(0x110)
#define reset_cacheconfig()         command_mhblock_lda(0x30)
#define reset_counters()            command_mhblock_lda(0x34)
#define set_dcache_loaddr_1(x)      command_mhblock_sta(0x40, (x))
#define set_dcache_hiaddr_1(x)      command_mhblock_sta(0x50, (x))
#define dcache_en_range_1()         command_mhblock_sta(0x160, 0x1)
#define dcache_dis_range_1()        command_mhblock_sta(0x160, 0x0)
```

Com este *wrapper* em C, basta a aplicação chamar a função *get_dcache_read_miss()* para obter a quantidade de *read misses* até aquele momento ou *get_dcache_read_hit()* para obter a quantidade de *read hits*.

5.2 Monitores de consumo de energia e interfaces

O monitor de consumo de energia é de vital importância para o projeto pois sem esta informação não é possível comparar os resultados com relação ao consumo de energia de uma arquitetura com reconfiguração dinâmica da *cache* e uma arquitetura com *cache* convencional. Portanto, é necessário desenvolver um componente de hardware interno à FPGA ou utilizar alguma ferramenta externa que informe à aplicação em execução qual o consumo de energia instantâneo. De posse desta informação, juntamente com as informações de *hits* e *misses* da *cache* de dados, é possível mapear quais configurações de *cache* são as melhores para uma determinada aplicação. Esta técnica é conhecida como *offline profiling* e foi utilizada para extrair algumas informações que foram posteriormente incluídas no algoritmo de reconfiguração.

Foram implementados três monitores diferentes para medir o consumo de energia, sendo que os dois primeiros foram desenvolvidos utilizando a plataforma de desenvolvimento da Xilinx. Estes dois monitores e boa parte do trabalho proposto foram desenvolvidos durante estágio no exterior em Los Angeles, EUA no *Information Sciences Institute* da Universidade do Sul da California, com supervisão do Prof. Dr. Pedro Diniz.

O monitor de consumo de energia ideal é aquele que é executado em paralelo com o processador, pois assim não é preciso interromper a aplicação para amostrar os dados referentes ao consumo de energia instantâneo. Se fosse necessário realizar uma chamada de função durante a execução do programa para obter o valor do consumo, no fundo o que estaríamos medindo seria o consumo instantâneo da função que acabamos de chamar. Para solucionar este problema, primeiramente foi proposto o desenvolvimento de um monitor em hardware e que fosse executado em paralelo com o processador. A ideia era que esse monitor amostrasse os valores do consumo de energia em registradores mapeados na memória do processador, e ao final da execução poderia ser feita uma chamada de função para obter estes valores.

5.2.1 Primeiro monitor (Virtex-6)

A placa de desenvolvimento da Xilinx dispõe de alguns componentes que auxiliam o desenvolvimento de um monitor em hardware. O primeiro deles é conhecido como *System Monitor*, e é um componente instanciável em código VHDL que disponibiliza várias informações sobre a FPGA, como por exemplo, temperatura e voltagem de entrada. Estas informações são obtidas de sensores internos na FPGA, conforme ilustrado na Figura 5.2. Outra funcionalidade importante do *System Monitor* é que é possível amostrar entradas analógicas externas, 17 entradas no total. Este componente é totalmente configurável, sendo possível selecionar quais entradas externas devem ser amostradas.

Duas destas entradas estão conectadas nos terminais de um resistor externo na placa de desenvolvimento. O resistor possui resistência conhecida, 5 mΩ, e está instalado na linha de alimentação da FPGA. Assim, é possível amostrar o valor referente à queda de voltagem sobre o resistor e calcular, pela Lei de Ohm, a corrente instantânea consumida:

$$I = \frac{V}{R} \tag{5.1}$$

Onde I é a intensidade da corrente elétrica medida em ampère (A), V é a diferença de potencial elétrico (ou tensão) medida em volt (V) e R é a resistência elétrica medida em ohm (Ω). Supondo que a queda de tensão sobre o resistor seja de 1 mV, pela Fórmula 5.1 temos:

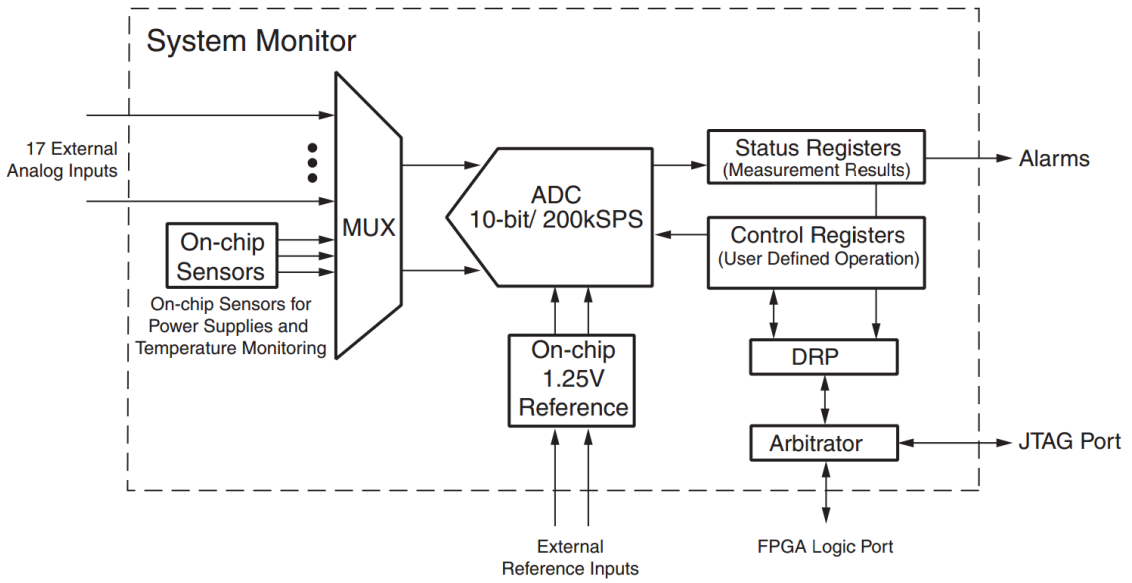


Figura 5.2: *System Monitor* presente na Virtex-6 (Xilinx, 2012b).

$$I = \frac{1mV}{5m\Omega} = 200mA \tag{5.2}$$

Como o System Monitor nos fornece o valor da voltagem de entrada, através do uso de sensores internos à FPGA, e como temos agora o valor da corrente sendo consumida pela linha de alimentação da FPGA, podemos, portanto calcular o valor da potência elétrica através da fórmula:

$$P = V \times I \tag{5.3}$$

Onde P é a potência elétrica instantânea medida em Watt (W), V é a diferença de potencial elétrico (ou tensão) medida em volt (V) e I é a intensidade da corrente elétrica medida em ampère (A). Supondo que o valor da voltagem de entrada da FPGA seja de 1V, pela Fórmula 5.3 temos a potência elétrica instantânea gerada pela FPGA:

$$P = 1V \times 200mA = 200mW \tag{5.4}$$

Com posse da potência instantânea, fica trivial calcular o consumo de energia da FPGA em *Joules*. É preciso apenas amostrar vários valores de potência instantânea durante a execução da aplicação, calcular a média destes valores e multiplicar pelo tempo total gasto em segundos para executar a aplicação. Desta forma, obtemos o valor do consumo de energia médio da FPGA durante a execução da aplicação. Vale lembrar que as aplicações são executadas sem o auxílio de um sistema operacional. A inicialização

da memória, controle de interrupções, *drivers* dos periféricos de comunicação e etc, estão disponíveis em bibliotecas *linkadas* estaticamente na aplicação.

O *System Monitor* possui apenas um único registrador de resultado para cada tipo de medida. Por exemplo, se requisitarmos ao componente para amostrar o valor da voltagem da alimentação de entrada, este valor é guardado no registrador de resultado. Se logo na sequência requisitarmos uma segunda amostragem, o valor anterior é sobrescrito e perdido. Esta limitação anula completamente o propósito do monitor em paralelo, pois teríamos que interromper a aplicação a cada instante para recuperar o valor amostrado. O cenário ideal seria se tivéssemos vários destes registradores de resultado, salvando os valores amostrados a cada instante pré-definido. Portanto, foi necessário desenvolver um componente de hardware, com vários registradores, que fizesse automaticamente as requisições ao *System Monitor* e ao mesmo tempo salvasse os valores de retorno em um dos vários registradores disponíveis. Ao final da execução da aplicação, teríamos os valores referentes ao consumo da FPGA salvo nestes vários registradores.

O processador LEON3 se comunica com periféricos externos e também com a memória externa DDR3 através de um barramento de alto desempenho, conhecido como *AMBA AHB Bus*. Desta forma, foi preciso também desenvolver no componente de hardware uma interface compatível com o padrão *AMBA AHB*, para integrar o componente ao sistema e se ter acesso aos registradores de resultados com o uso de instruções *load/store* pela aplicação. Esta abordagem, entretanto não funcionou. O *System Monitor* é um componente caixa-preta desenvolvido pela Xilinx e erros inesperados aconteciam frequentemente. O maior problema era não poder confiar nos valores retornados pelo *System Monitor*. Até se tentou medir os valores dos resistores manualmente com a ajuda de um osciloscópio, de forma a confirmar se os valores retornados pelo *System Monitor* estavam corretos. Contudo, por algum motivo desconhecido, ao se conectar as pontas de teste do osciloscópio nos terminais dos resistores, a placa reinicia instantaneamente. Além disso, o conversor analógico/digital do *System Monitor* tem precisão de apenas 10-*bits*, e como estamos trabalhando na casa de mV, mA e mW, a precisão fornecida não seria suficiente. Apesar de ser a melhor abordagem para o projeto, foi necessário encontrar alguma outra forma de se medir o consumo de energia da placa.

5.2.2 Segundo monitor (Virtex-6)

Devido às falhas frequentes do primeiro monitor, foi necessário desenvolver um segundo monitor, mais confiável e que retornasse valores com maior precisão.

A placa de desenvolvimento da Xilinx contêm dois gerenciadores de energia que são responsáveis por converter a tensão de entrada de 12V, em várias tensões menores como 1V, 1.5V, 3.3V, etc, e assim distribuí-las aos periféricos da placa. Isto é necessário,

pois os diversos periféricos presentes na placa geralmente possuem tensão de alimentação diferentes. Por exemplo, a memória externa DDR3 é alimentada com uma tensão de 3.3V. Por outro lado, a FPGA é alimentada com uma tensão de apenas 1V.

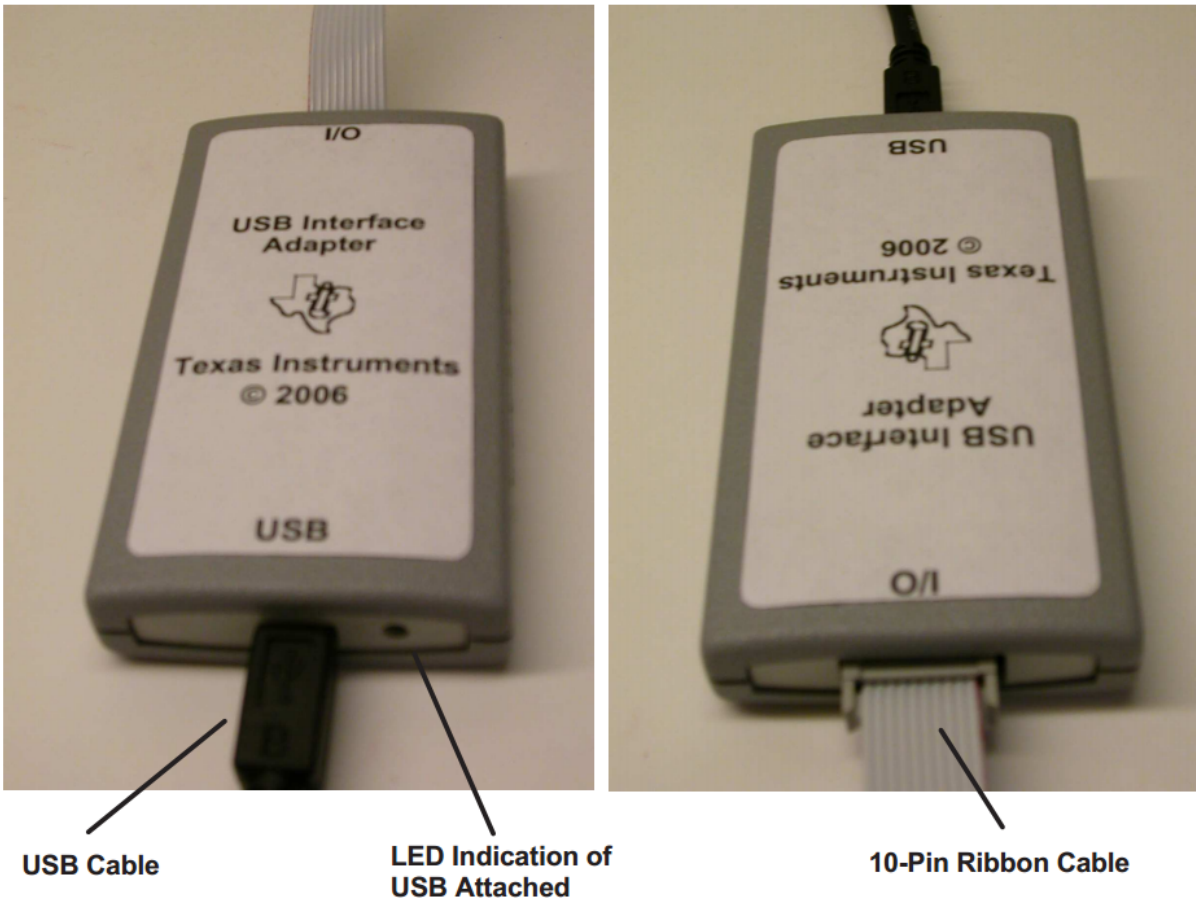


Figura 5.3: Adaptador externo para monitorar o consumo de energia da FPGA.

Este gerenciamento é realizado por um circuito integrado da Texas Instrument, modelo UCD9240. Além de controlar os valores das tensões, ainda fornece mecanismo de monitoramento destas tensões. O monitoramento é realizado através de uma ferramenta externa, Figura 5.3, e de um software fornecido pela Texas Instrument. A ferramenta se conecta ao barramento *PMBus* da placa e o software faz a leitura do consumo da potência instantânea de cada linha de alimentação controlado pelos gerenciadores de energia. A Figura 5.4 mostra todas as linhas de alimentação que são controlados pelo circuito integrado. As duas de maior interesse são a VCCINT, de 1V e a VCC3V3, de 3.3V. A primeira é a linha de alimentação do núcleo da FPGA e a segunda é a alimentação da memória externa DDR3. É possível realizar o *offline profiling* das aplicações com este monitor uma vez que se têm os valores da potência consumida por essas duas linhas de alimentação.

O software para leitura dos valores de consumo de potência da Texas Instrument é

disponível apenas para Windows. Como todo o desenvolvimento do projeto estava sendo feito em Linux, foi preciso entender o protocolo de comunicação entre a ferramenta e o PC para então desenvolver uma aplicação em Linux que fizesse o mesmo que o software original da Texas Instrument. O protocolo é aberto e bem documentado pela empresa, portanto não foi uma barreira para a continuidade do projeto. A aplicação desenvolvida para Linux continha apenas a parte necessária para dar prosseguimento ao projeto, ou seja, monitorar apenas as linhas de alimentação de 1V e de 3.3V. Além disso, foi possível ter maior controle sobre quando iniciar o monitoramento e quando pará-lo. Apesar deste monitor não ser o ideal, pois a aplicação em execução não tem acesso aos valores do consumo, essa abordagem é suficiente para realizar o *offline profiling* das aplicações.

Device	Rail #	Rail Name	Schematic Rail Name	Vout (V)	PG On (V)	PG Off (V)	On Delay (ms)	Rise (ms)	Off Delay (ms)	Fall (ms)	Vout Over Fault (V)	Response	Iout Over Fault (A)	Response	Temp Over Fault (°C)	Response
UCD9240 (Addr 52)	1	Rail #1	VCCINT	1	0.925	0.9	5	10	5	10	1.1	Shut down	14	Shut down	80	Shut down
	2	Rail #2	VCC2V5	2.5	2.313	2.25	10				2.75					
	3	Rail #3	VCCAUX	2.5	2.325	2.25	5				2.8		9			
UCD9240 (Addr 53)	1	Rail #1	MGT _AVCC	1.025	0.948	0.923	5	20	5	10	1.128	Shut down	14.5	Shut down	80	Shut down
	2	Rail #2	MGT _AVTT	1.25	1.156	1.125					1.375					
	3	Rail #3	VCC1V5 _FPGA	1.5	1.388	1.35	10	10			1.65					
	4	Rail #4	VCC3V3	3.3	3.052	2.97	5	5	0	0	3.63					

Figura 5.4: Linhas de alimentação monitoradas pelos gerenciadores de energia (Xilinx, 2012a).

5.2.3 Terceiro monitor (Stratix IV)

Por fim, o terceiro monitor foi desenvolvido utilizando a plataforma da Altera. A placa de desenvolvimento da Altera contêm dois conversores analógico/digital de 24-*bits* que são utilizados para amostrar a queda de tensão de vários resistores espalhados pela placa, semelhante à placa da Xilinx. De acordo com as informações técnicas do ADC, uma amostragem demora em torno de 140ms. Dois destes resistores estão conectados nas linhas de alimentação do núcleo da FPGA e da DDR3. A potência é calculada exatamente da mesma forma como explicado na seção 5.2.1. Com o valor da potência instantânea e com o tempo gasto pela aplicação, calcula-se a energia consumida pela FPGA e DDR3 durante a execução da aplicação.

Os conversores analógico/digital presentes na placa possuem barramento SPI e estão interligados tanto no FPGA como no CPLD, como pode ser observado na Figura 5.5. O CPLD e a FPGA são interligados via um barramento de dados de 32-*bits* e 25-*bits* de endereço. Há ainda 4 sinais *(write enable, chip select, output enable, byte enable)* de controle e 1 sinal de *clock*.

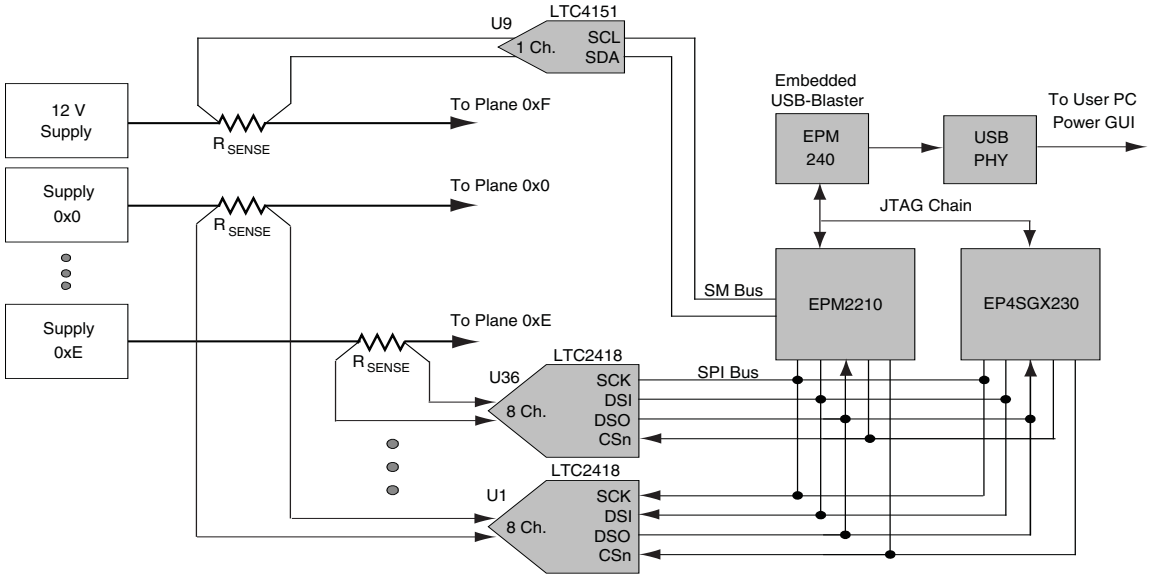


Figura 5.5: Conversores analógico/digital, modelo LTC2418, presentes na placa da Altera.

A Altera disponibiliza um projeto para o CPLD que implementa o protocolo de comunicação definido pelo fabricante do ADC. Desta forma, foi necessário apenas desenvolver uma interface de comunicação entre o CPLD e o barramento APB do LEON3. Os registradores do CPLD foram mapeados em memória através da interface desenvolvida, via barramento APB. Portanto, basta acessar um endereço específico no espaço de endereçamento do LEON3 e definido pela interface desenvolvida para se obter os valores da corrente, em mA, das linhas de alimentação do FPGA e da DDR3. O código C abaixo demonstra como foi feito o acesso aos registrados do CPLD via interface APB:

```
#define BASE_POWER_ADDR    0x80000800
#define FPGA_A             0x08
#define DDR3_A             0x10

int get_register(int reg)
{
    int * x = (int *) (BASE_POWER_ADDR + reg);
        return *x;
}

int get_fpga_current(void)
{
    return get_register(FPGA_A);
```

```
}

int get_ddr3_current(void)
{
    return get_register(DDR3_A);
}
```

5.3 Memória DDR3 e ponte Avalon/Amba (Stratix IV)

A integração do controlador de memória DDR3 da Altera no LEON3 foi bastante trabalhoso, por dois motivos: o controlador disponibilizado pela Altera é um componente caixa preta, dificultando o entendimento do mesmo, e por conta da diferença entre o barramento de comunicação utilizado pelo controlador de memória (Avalon-MM) e o barramento utilizado pelo LEON3 (AMBA). Desta forma, foi necessário desenvolver um componente em hardware para realizar a ponte entre o barramento Avalon-MM e o barramento AMBA AHB.

Os sinais do barramento Avalon-MM de um periférico *slave* são apresentados na Figura 5.6 e os sinais do barramento AMBA/AHB são apresentados na Figura 5.7.

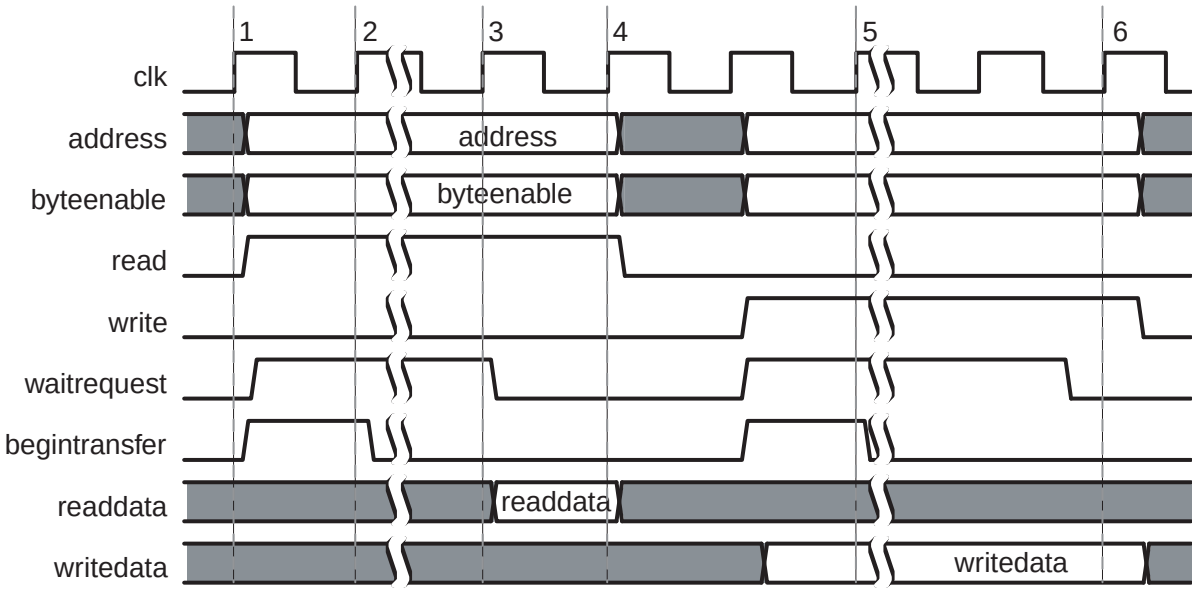


Figura 5.6: Transação no barramento Avalon-MM (Altera, 2010).

Como pode ser observado nas Figuras 5.6 e 5.7, os sinais de ambos os barramentos são parecidos. Entretanto, uma transação no barramento AMBA/AHB é divido em duas fases, *address phase* e *data phase*, enquanto no barramento Avalon-MM a transação acontece em uma única fase. A primeira parte, *address phase*, tem duração de apenas 1 ciclo de *clock* e é nesta parte que o endereço do dado a ser lido/escrito e os *bits* de controle são

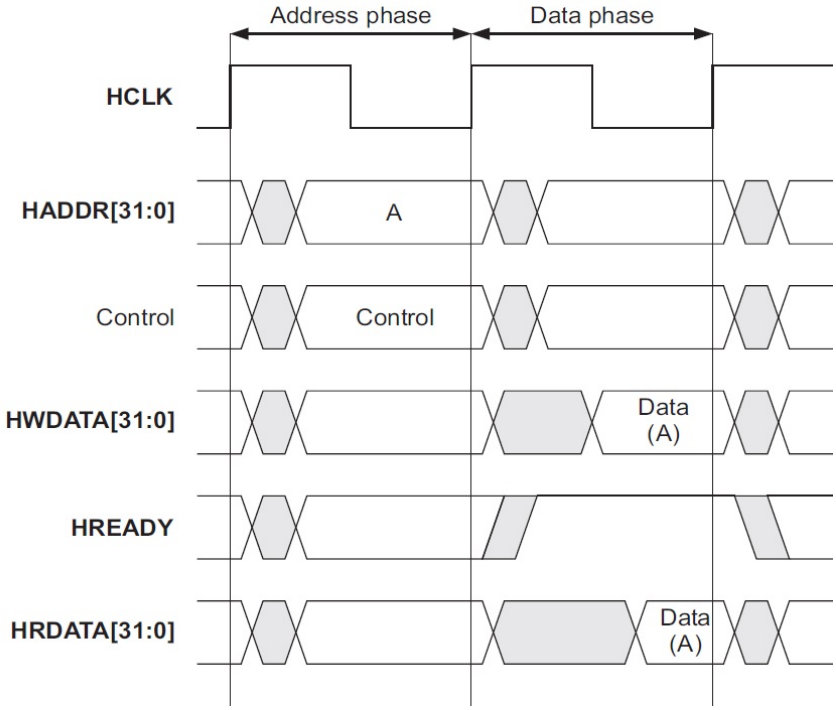


Figura 5.7: Transação no barramento AMBA-AHB (ARM, 1999).

enviados ao barramento. A parte *data phase* pode ter duração de vários ciclos de *clock* e é nesta parte que o dado lido/escrito é enviado/recebido do barramento. Desta forma, o componente desenvolvido precisou basicamente tratar desta diferença de fases para que as transações entre os barramentos fossem corretamente sincronizadas.

5.4 Registradores de reconfiguração e controlador de ways

A arquitetura proposta para a reconfiguração dinâmica da *cache* segue a ideia apresentada por Albonesi (Albonesi, 1999), conhecida como *way-shutdown*. Nesta técnica, as *ways* da *cache* são ativadas e desativadas em *run-time* conforme a necessidade da aplicação, com *overhead* mínimo por conta da reconfiguração e com modificações relativamente simples do hardware atual.

O processador LEON3 implementa uma memória para cada *way* da *cache*. Quando o processador é sintetizado, o compilador detecta automaticamente o uso de memória interna pelas *ways* da *cache* e mapeia estas memórias em *block RAMs* da FPGA. A Figura 5.8 exemplifica como é organizada a *cache* de dados no LEON3.

Na FPGA Virtex-6 e na Stratix IV, cada *block RAM* possui um sinal de entrada *Enable*. Este sinal é configurado para o nível lógico "1" quando são realizadas operações de escrita ou leitura no bloco. O controlador da *cache* de dados pode manter este sinal de *Enable* no nível lógico "0" e, como consequência, desativar uma determinada *way* da *cache*. Apesar do *block RAM* estar fisicamente implementado na FPGA e consumindo energia

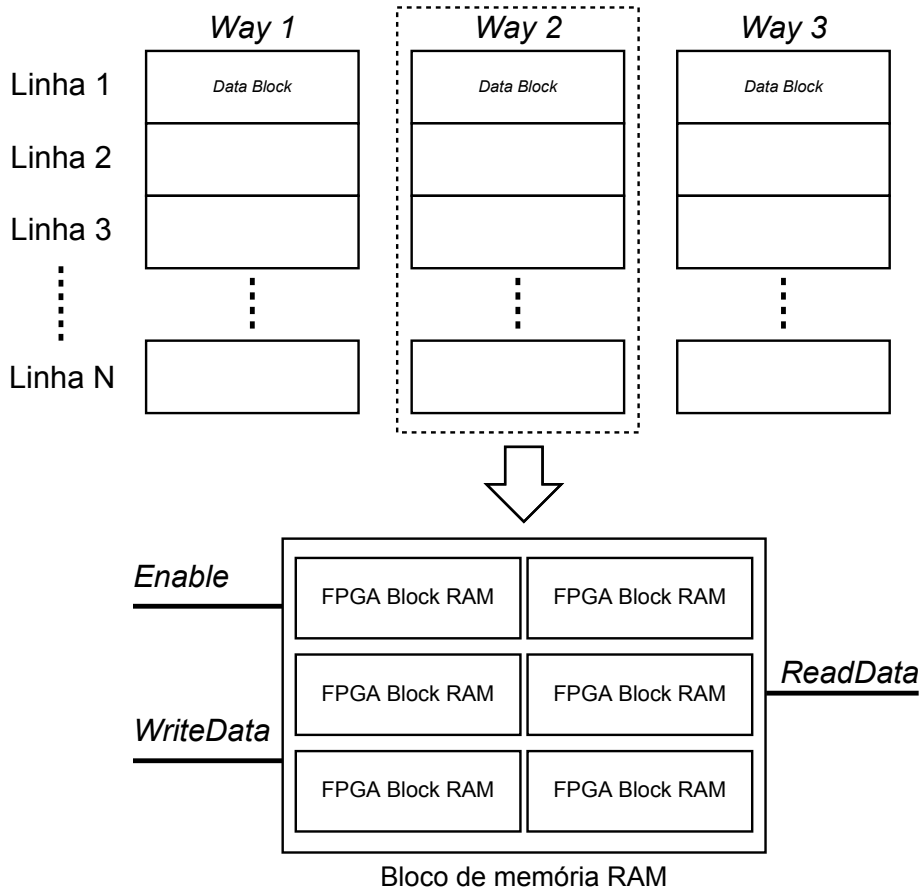


Figura 5.8: Organização física da *cache* de dados no LEON3

estática, o fato de não permitir que determinada *way* seja acessada reduz o consumo de energia dinâmica, que é o principal tipo de energia consumida em circuitos com a tecnologia CMOS.

Para controlar qual *way* da *cache* desativar, foi adicionado no controlador de *cache* de dados um registrador de 4-*bits*, onde o *bit* "0" do registrador representa a *way* de número 4, o *bit* "1" do registrador representa a *way* de número 3, e assim por diante. Quando o *bit* é configurado para o nível lógico "0", a *way* correspondente é desativada. Este registrador foi então mapeado na memória do LEON3 para poder ser acessado através de instruções *assembly lda/sta*, conforme explicado na seção 5.1. Portanto, a decisão sobre qual *way* ativar ou desativar é realizada por software. Outras modificações tiverem que ser feitas no controlador da *cache* para que fosse possível realizar a reconfiguração dinâmica. Por exemplo, o algoritmo de substituição de linha teve que ser modificado. No entanto, a modificação foi simples. Ao invés de considerar todas as 4-*ways* como presentes e ativas, o algoritmo deveria ser informado dinamicamente e em tempo de execução quantas *ways* estão habilitadas. O algoritmo deve ter conhecimento desta informação, caso contrário poderia tentar remover ou incluir dados em uma *way* desabilitada.

A política de escrita da *cache* de dados é *write-through* e portanto, não é preciso se preocupar com a perda dos dados quando uma *way* é desabilitada. Os dados sempre estarão coerentes com o próximo nível de memória, neste caso a memória RAM externa. No entanto, pode-se ter um pouco mais de *read misses* ao diminuir a associatividade da *cache*, uma vez que os dados presentes nas *ways* desabilitadas não podem ser mais acessados, gerando *compulsory misses* caso estes dados precisem ser acessados novamente. Entretanto, como será apresentado no Capítulo 6, a quantidade de *misses* extras é desprezível.

Na arquitetura reconfigurável proposta todas as 4-*ways* precisam estar fisicamente implementadas e disponíveis no hardware. Foi escolhida a configuração de *cache* com 4-*ways* e 256 *KBytes* de tamanho total. Portanto, cada *way* da *cache* possui tamanho de 64 *KBytes*. Ao desabilitar uma *way*, o tamanho total da *cache* é reduzido em 64 *KBytes*. Logo, a arquitetura reconfigurável é limitada em apenas 4 configurações distintas, conforme mostrado na Tabela 5.1.

Tabela 5.1: Configurações disponíveis na arquitetura reconfigurável proposta.

Número de *ways* habilitadas	Tamanho total da *cache*
1	64 KB
2	128 KB
3	192 KB
4	256 KB

Esta abordagem limita consideravelmente o número de configurações diferentes que a *cache* pode assumir. Por exemplo, não é possível configurar a *cache* em 2-*ways* com 64 *KBytes*. Apesar da complexidade elevada, é possível desenvolver em trabalhos futuros uma *cache* reconfigurável que permite assumir valores de associatividade independentemente do tamanho da *cache*.

5.5 Algoritmo de reconfiguração

A reconfiguração dinâmica da *cache* é ativada via software de acordo com os valores de *read miss* e *read hit*. O algoritmo de reconfiguração é implementado em forma de interrupção de *timer*. O *timer* é configurado para gerar uma interrupção a cada 140ms. Este valor foi escolhido pois o ADC do monitor de consumo de energia, conforme explicado na seção 5.2.3, completa uma conversão a cada 140ms. É importante comentar que este software foi executado apenas na plataforma da Altera por indisponibilidade da plataforma da Xilinx e por este motivo foi utilizada a terceira versão do monitor descrito na seção 5.2.3.

O código C que ativa a reconfiguração dinâmica da *cache* é demonstrado a seguir:

```
1
2 #define MISS_THRESHOLD 2
```

```
3
4  static unsigned char ways_config[] = {0b1000, 0b1100, 0b1110, 0b1111};
5  current_fpga_power += get_fpga_current();
6  current_ddr3_power += get_ddr3_current();
7  current_board_power += get_board_current();
8  current_miss = get_lower_dcache_read_miss();
9  current_hit = get_lower_dcache_read_hit();
10
11 if(enable_reconfig)
12 {
13   current_miss_rate = ((float)current_miss/(float)(current_miss + current_hit))*100;
14   last_miss_rate = (float)(last_miss/(float)(last_miss + last_hit))*100;
15
16       if((current_miss_rate > last_miss_rate) &&
17       ((current_miss_rate - last_miss_rate) <= MISS_THRESHOLD))
18       {
19          //keep current cache
20       }
21       else if((current_miss_rate > last_miss_rate) &&
22       ((current_miss_rate - last_miss_rate) > MISS_THRESHOLD))
23       {
24       //increase cache ways
25       if(current_cache_config < 4)
26       {
27            next_cache_config = current_cache_config + 1;
28            set_dcache_ways(ways_config[next_cache_config]&0xf);
29                }
30       }
31       else
32       {
33          //avoid going up and down repeatedly
34          if((current_cache_config > 1) && (last_cache_config != (current_cache_config - 1)))
35          {
36           //reduce cache ways
37           next_cache_config = current_cache_config - 1;
38           set_dcache_ways(ways_config[next_cache_config]&0xf);
39       }
40
41       last_miss = current_miss;
42       last_hit = current_hit;
43       last_cache_config = current_cache_config;
44       current_cache_config = next_cache_config;
45 }
```

O algoritmo calcula o *read miss rate* do intervalo atual e do intervalo anterior, linhas 13 e 14, utilizando as métricas *read miss* e *read hit* do intervalo atual e do intervalo anterior. Após o calculo, o algoritmo checa se o *read miss rate* atual é maior que o *miss rate* do intervalo anterior. Se for verdade, o algoritmo irá checar o quão maior é o *miss rate* atual em comparação com o anterior. Se esta diferença for maior que *MISS_THRESHOLD*, ou seja, 1%, é bem provável que a aplicação irá se beneficiar de uma *cache* maior. A *cache* é então dinamicamente reconfigurada, aumentando a associatividade em 1 e o tamanho total em 64 *KBytes*. Caso a diferença esteja abaixo de *MISS_THRESHOLD* pode-se manter a

cache atual. Estes dois casos são válidos apenas se o *miss rate* atual for maior que o *miss rate* do intervalo anterior. Na situação em que o *miss rate* atual é menor que o anterior, pode-se tentar reduzir o tamanho da *cache* para reduzir o consumo de energia. O valor *MISS_THRESHOLD* de 1% foi retirado de observações práticas realizadas na etapa de *offline profiling* das aplicações, que estão descritos na seção 6.1.

Capítulo

6

Resultados obtidos

Nesta seção serão apresentados os resultados obtidos durante a execução das três aplicações mencionadas no Capítulo 4, na arquitetura com *cache* fixa e também na arquitetura proposta com *cache* reconfigurável.

6.1 Arquiteturas com cache fixa

Como primeira etapa foi realizado o *offline profiling* das aplicações descritas no Capítulo 4, em 24 arquiteturas de hardware distintas. Cada arquitetura possui uma configuração da *cache* de dados diferente. O objetivo desta etapa foi identificar a configuração de *cache*, entre as combinações possíveis, que obtinha o menor consumo de energia, para todas as três aplicações. É importante destacar que todas as etapas descritas nesta seção foram executadas na plataforma de hardware da Xilinx, utilizando o segundo monitor de energia mencionado no Capítulo 5. As 24 arquiteturas de hardware são apresentadas na Tabela 6.1.

Cada *way* da *cache* no LEON3 pode ter entre 1-256 *KBytes* e por este motivo não foi possível sintetizar todas as combinações de tamanho de *cache* e número de *ways*. Por exemplo, a configuração com 1 *KByte* de tamanho total e 2-*ways* requer que cada *way* possua 0.5 *KByte* de tamanho, o que não é permitido pelo processador LEON3. Portanto, são deixadas de fora três combinações, conforme demonstrado na Tabela 6.1.

Tabela 6.1: Arquiteturas geradas para realização do *offline profiling*. São 24 arquiteturas no total, cada uma com uma configuração de *cache* de dados diferente.

Tamanho da *cache*	Número de *ways*		
	1-way	*2-ways*	*4-ways*
1 KByte	**x**	**-**	**-**
2 KBytes	**x**	**x**	**-**
4 KBytes	**x**	**x**	**x**
8 KBytes	**x**	**x**	**x**
16 KBytes	**x**	**x**	**x**
32 KBytes	**x**	**x**	**x**
64 KBytes	**x**	**x**	**x**
128 KBytes	**x**	**x**	**x**
256 KBytes	**x**	**x**	**x**

As aplicações foram executadas no LEON3 e foram coletados, para cada aplicação e para cada arquitetura, os seguintes dados:

- Consumo da FPGA em Watts;
- Consumo da memória externa em Watts;
- Tempo de execução em milissegundos;
- *Read misses*;
- *Write misses*;
- *Read hits*;
- *Write hits*.

E com estes dados foi possível calcular:

- *Miss rate*;
- *Hit rate*;
- Consumo de energia médio em *Joules*.

Com posse destes dados foi possível identificar a configuração de *cache* que produzia o menor consumo de energia, para cada aplicação.

6.1.1 Aplicação Breadth-First Search

O resultado do consumo de energia está demonstrado na Figura 6.1. Nota-se que para esta aplicação, a melhor configuração de *cache* é a 4-*way* com 64 *KBytes* de tamanho total, embora a diferença de consumo para a configuração de 2-*way* seja mínima. Observando as Figuras 6.1 e 6.2 nota-se que, apesar da taxa de *miss* ser sempre menor com o aumento do tamanho da *cache*, o consumo de energia não segue este padrão e começa a não compensar o gasto extra de energia com a *cache* apenas para reduzir em uma parcela que pode ser considerada pequena da taxa de *miss*. Como nosso objetivo é reduzir o consumo de energia, devemos escolher a configuração com menor consumo, mesmo que se tenha uma pequena degradação no desempenho.

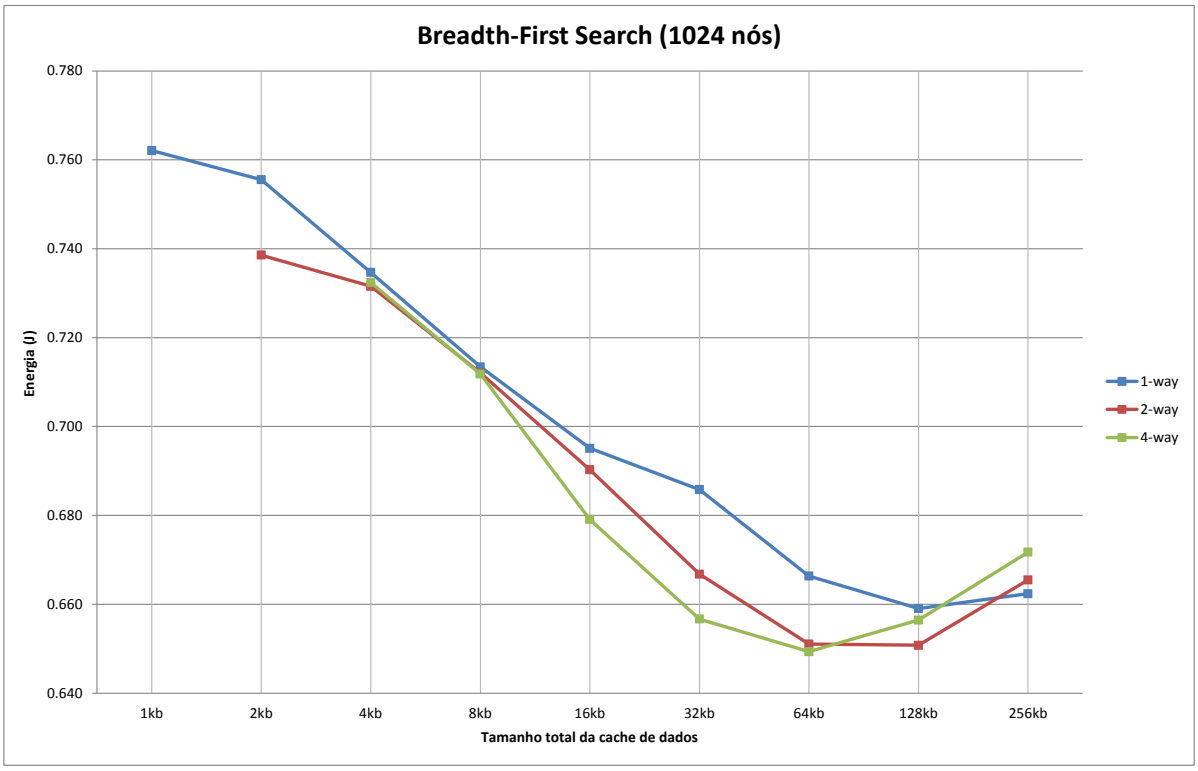


Figura 6.1: Consumo de energia da aplicação BFS para várias configurações de *cache* diferentes.

A Figura 6.3 mostra a diferença entre a taxa de *miss* da *cache* de dados com configurações adjacentes. Nota-se que conforme o tamanho da *cache* aumenta, a diferença na taxa de *miss* entre *caches* com tamanhos adjacentes diminui.

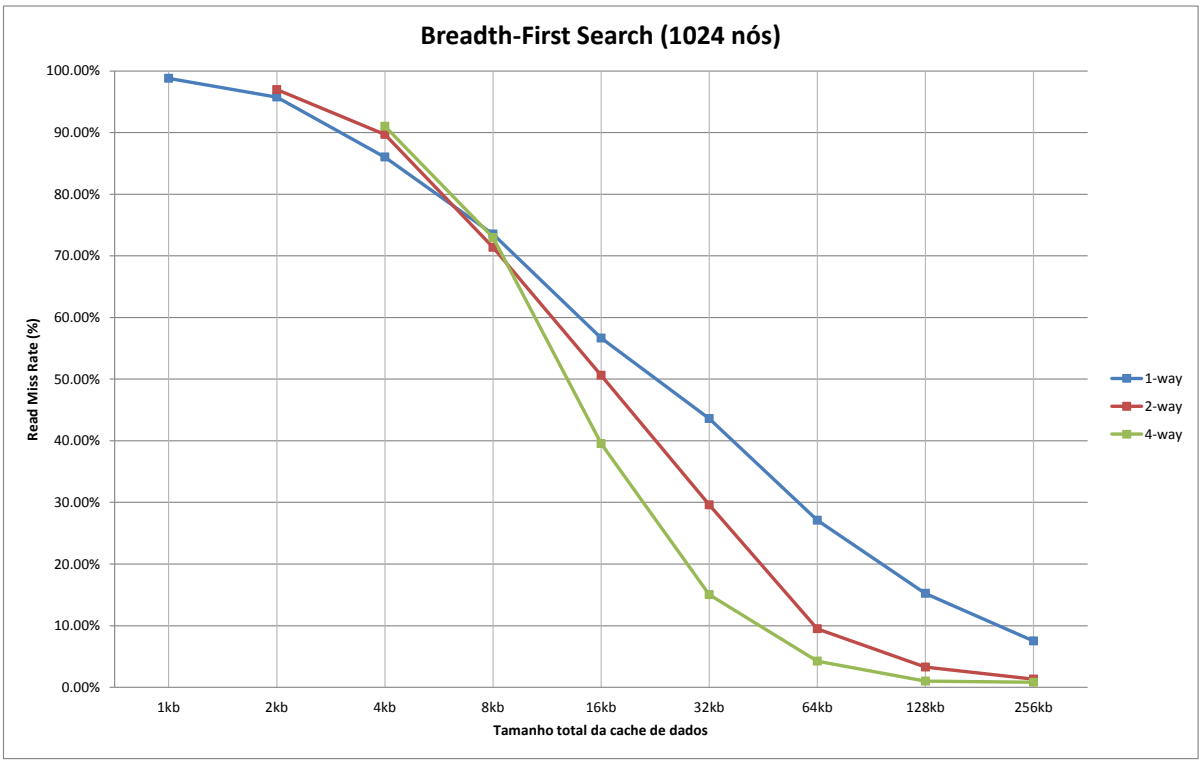


Figura 6.2: *Miss rate* da aplicação BFS para várias configurações de *cache* diferentes.

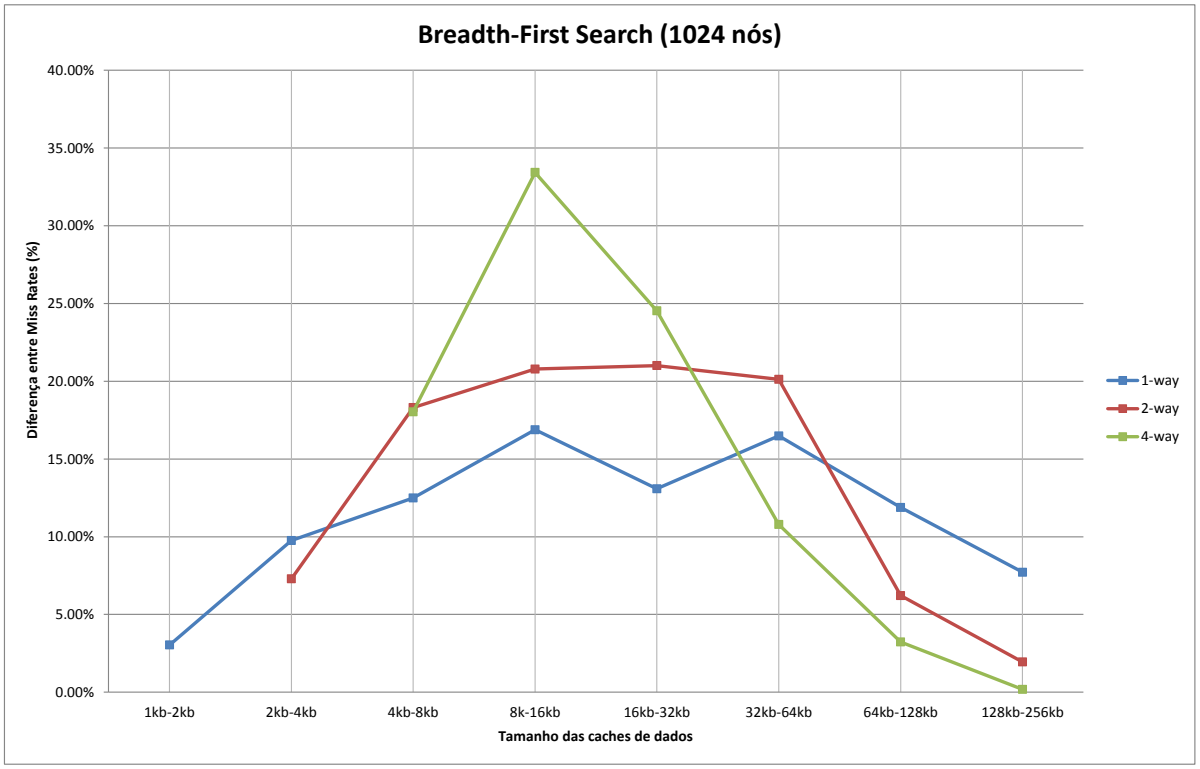


Figura 6.3: Diferença de *miss rate* entre configurações de *cache* adjacentes da aplicação BFS.

6.1.2 Multiplicação de matrizes esparsas

Pelas Figuras 6.4 e 6.5 nota-se que a melhor configuração e que produz o menor consumo de energia para a fase Factor é a *cache* com 1-*way* e 64 *KBytes* de tamanho total, embora a diferença para a *cache* com 2-*ways* e 32 *KBytes* também seja pequena. Já a fase Solve atinge menor consumo de energia quando há presente uma *cache* 2-*ways* de 32 *KBytes* de tamanho. Aqui também nota-se redução da taxa de *miss* à medida que a *cache* de dados aumenta de tamanho. As Figuras 6.6 e 6.7 mostram respectivamente as taxas de *read miss* da fase *Factor* e *Solve*. Nota-se novamente que à medida que o tamanho da *cache* aumenta, a diferença na taxa de *miss* entre *caches* com tamanhos adjacentes diminui. Analisando os gráficos nas Figuras 6.8 e 6.9 fica fácil enxergar este fenômeno. Foi observado que esta diferença, a partir da configuração de 64 *KBytes*, não ultrapassa a margem de 1%. Este valor foi utilizado para definir a estratégia do software de reconfiguração descrito no Capítulo 5.

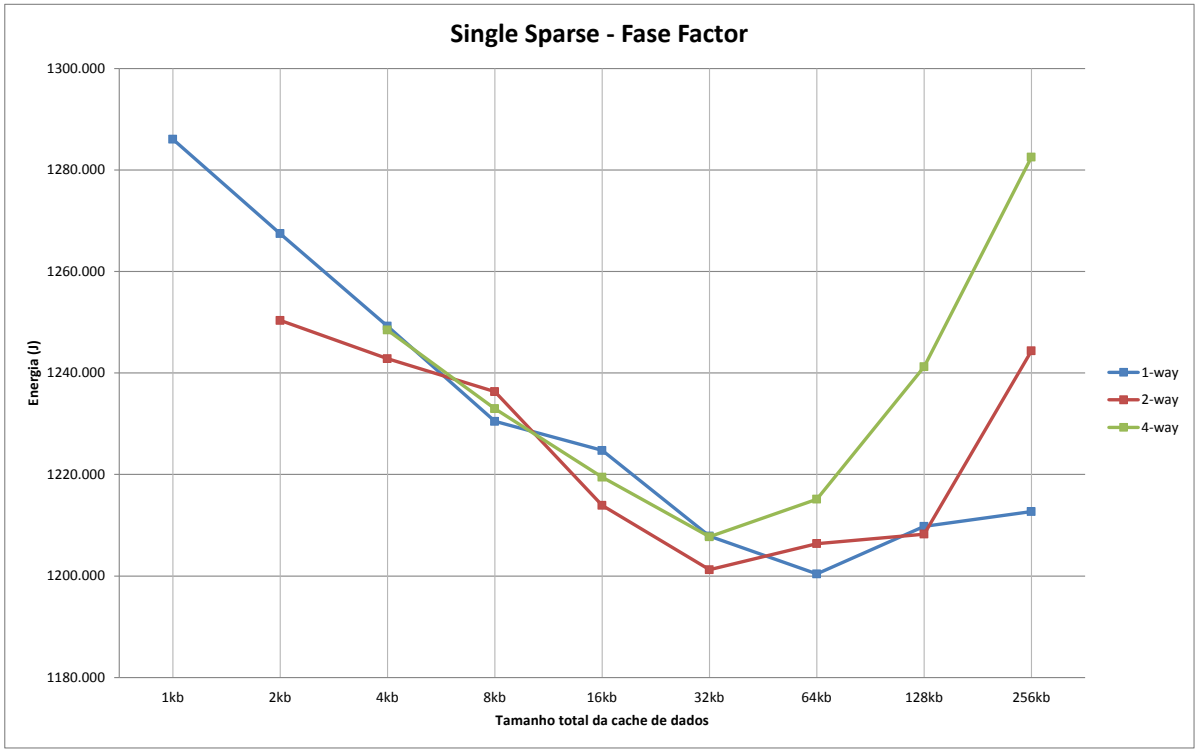


Figura 6.4: Consumo de energia da aplicação de multiplicação de matrizes esparsas, fase *Factor*.

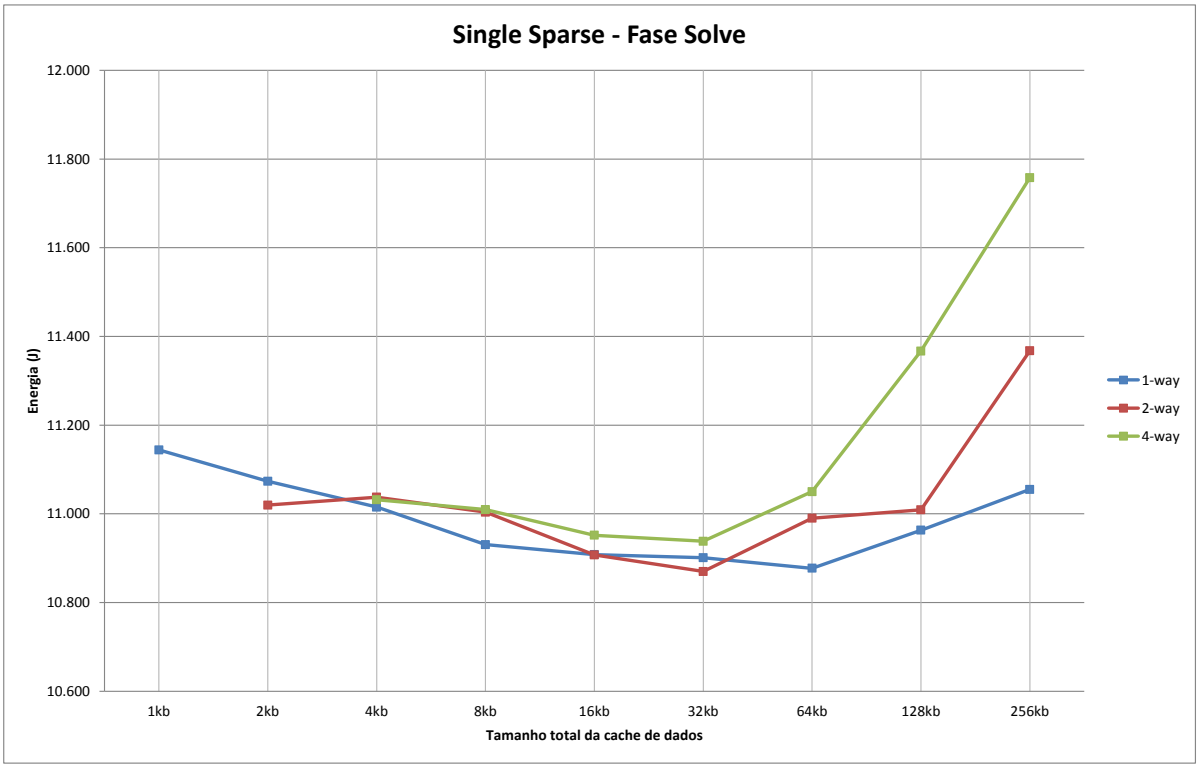


Figura 6.5: Consumo de energia da aplicação de multiplicação de matrizes esparsas, fase *Solve*.

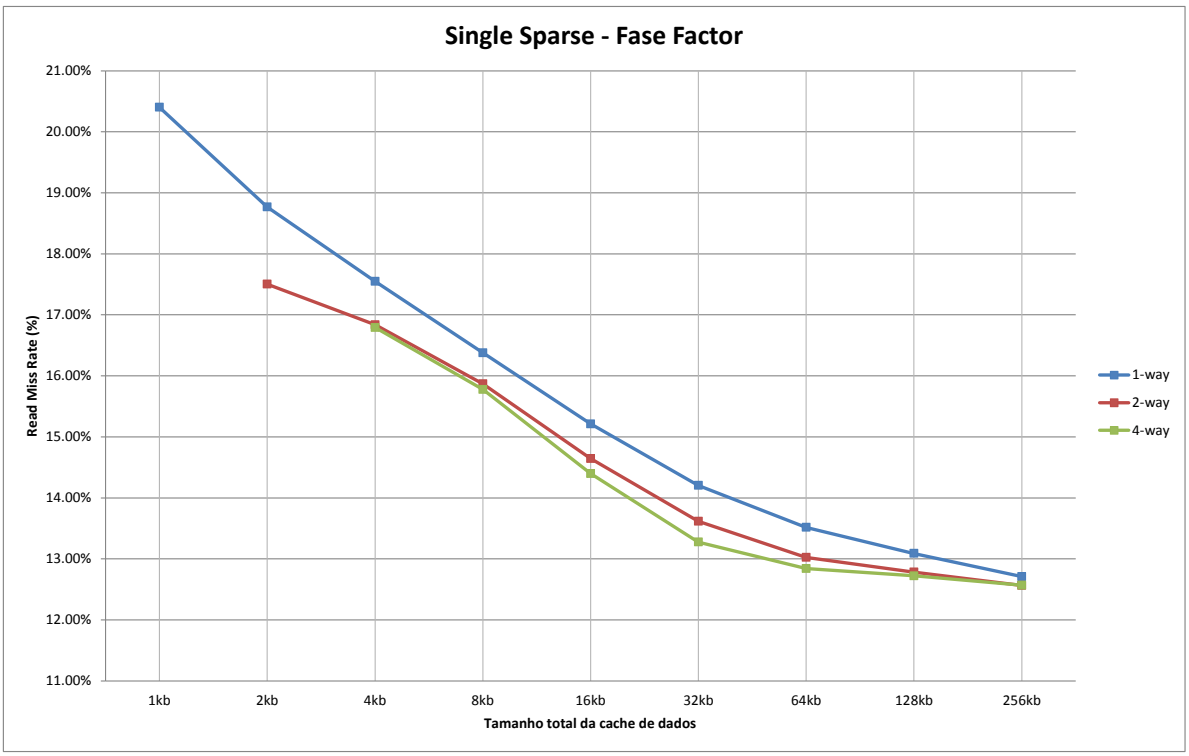


Figura 6.6: *Miss rate* da aplicação de multiplicação de matrizes esparsas, fase *Factor*.

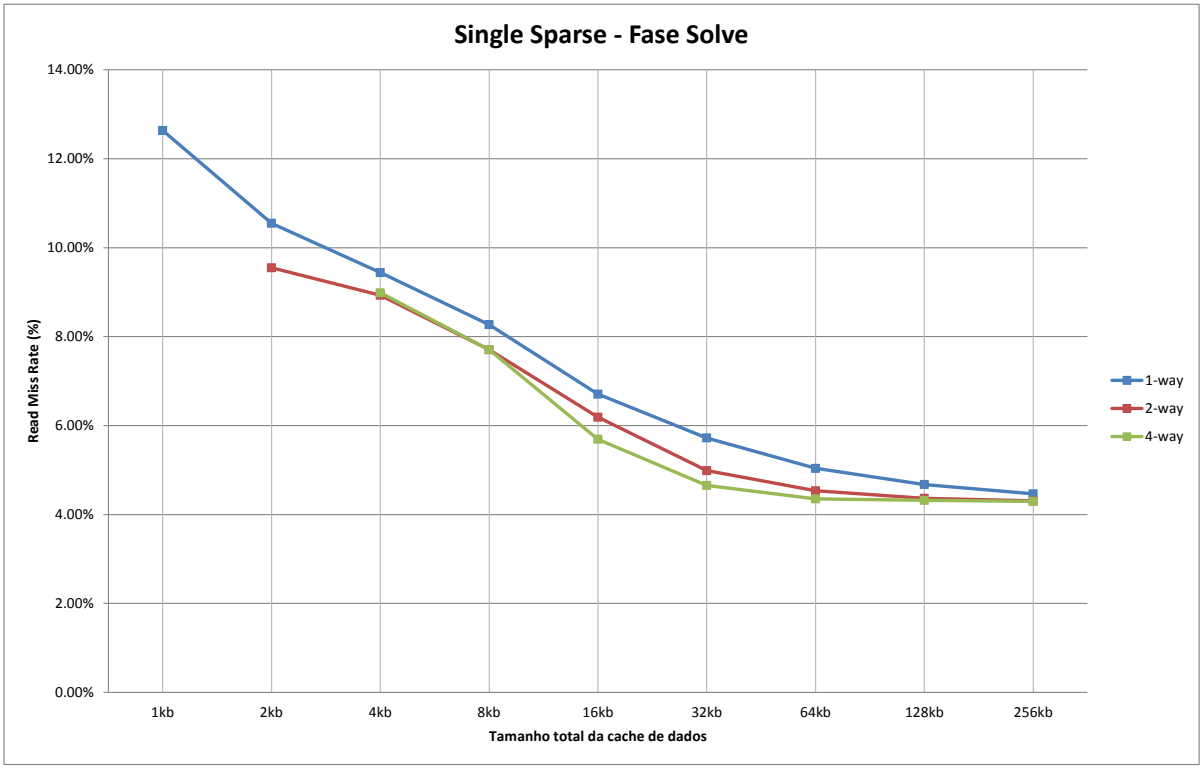


Figura 6.7: *Miss rate* da aplicação de multiplicação de matrizes esparsas, fase *Solve*.

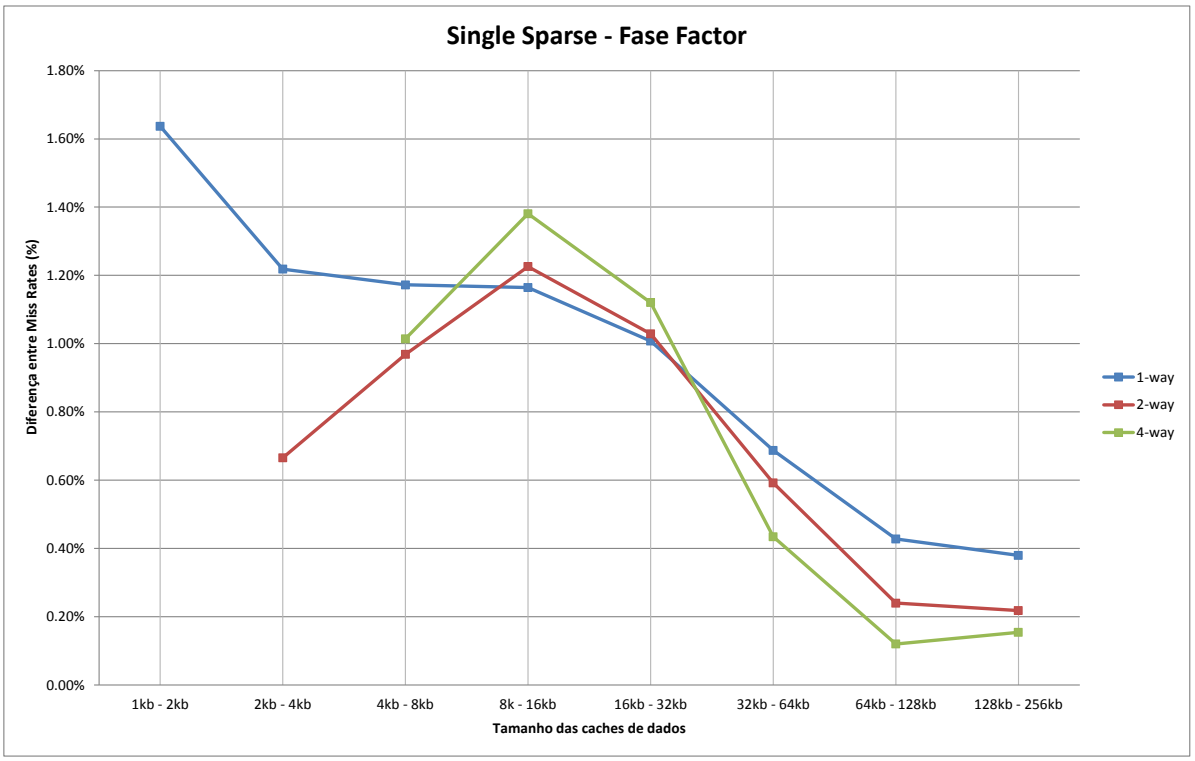


Figura 6.8: Diferença de *miss rate* entre configurações de *cache* adjacentes, fase *Factor*.

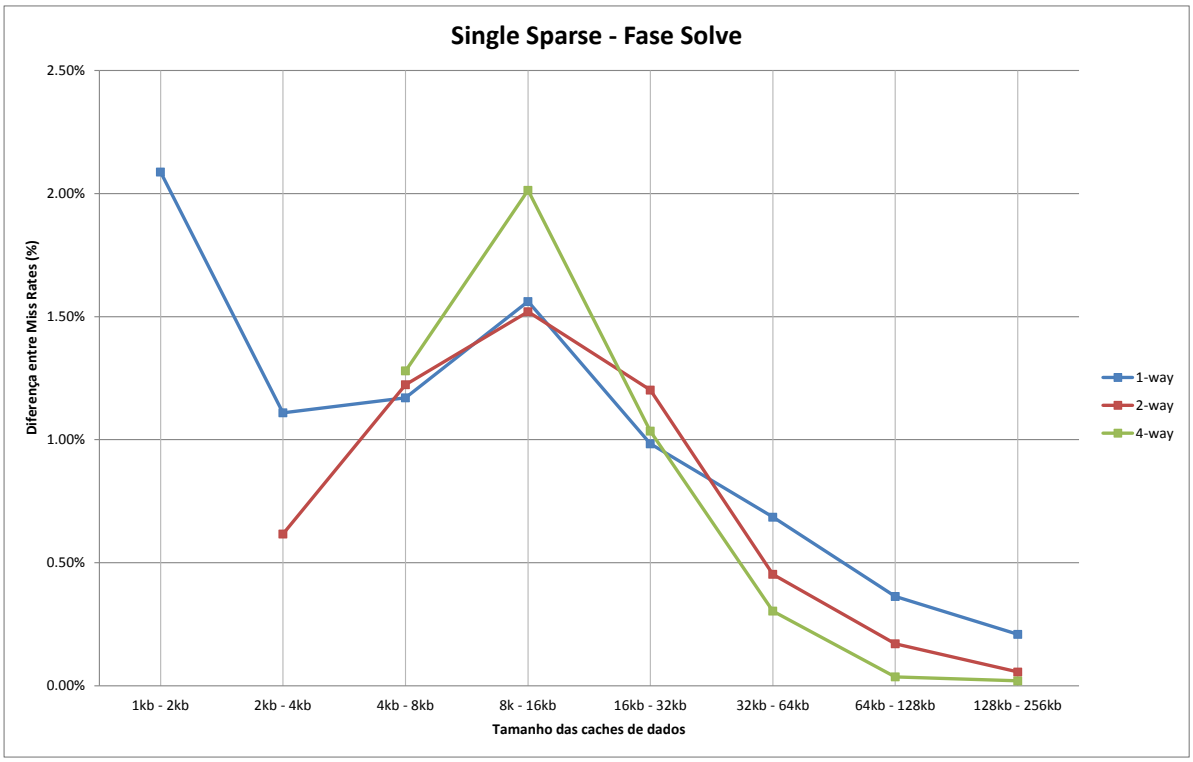


Figura 6.9: Diferença de *miss rate* entre configurações de *cache* adjacentes, fase *Solve*.

6.1.3 Algoritmo Push Relabel

Observando o gráfico de consumo de energia da Figura 6.10, conclui-se que a melhor configuração de *cache* para a aplicação *Push Relabel* é a 1-*way* com 128 *KBytes* de tamanho total. Nota-se novamente, observando os resultados da Figura 6.11, que à medida que o tamanho da *cache* aumenta, a diferença na taxa de *miss* entre *caches* com tamanhos adjacentes diminui. O gráfico da Figura 6.12 é condizente com os resultados observados nos gráficos das Figuras 6.8 e 6.9.

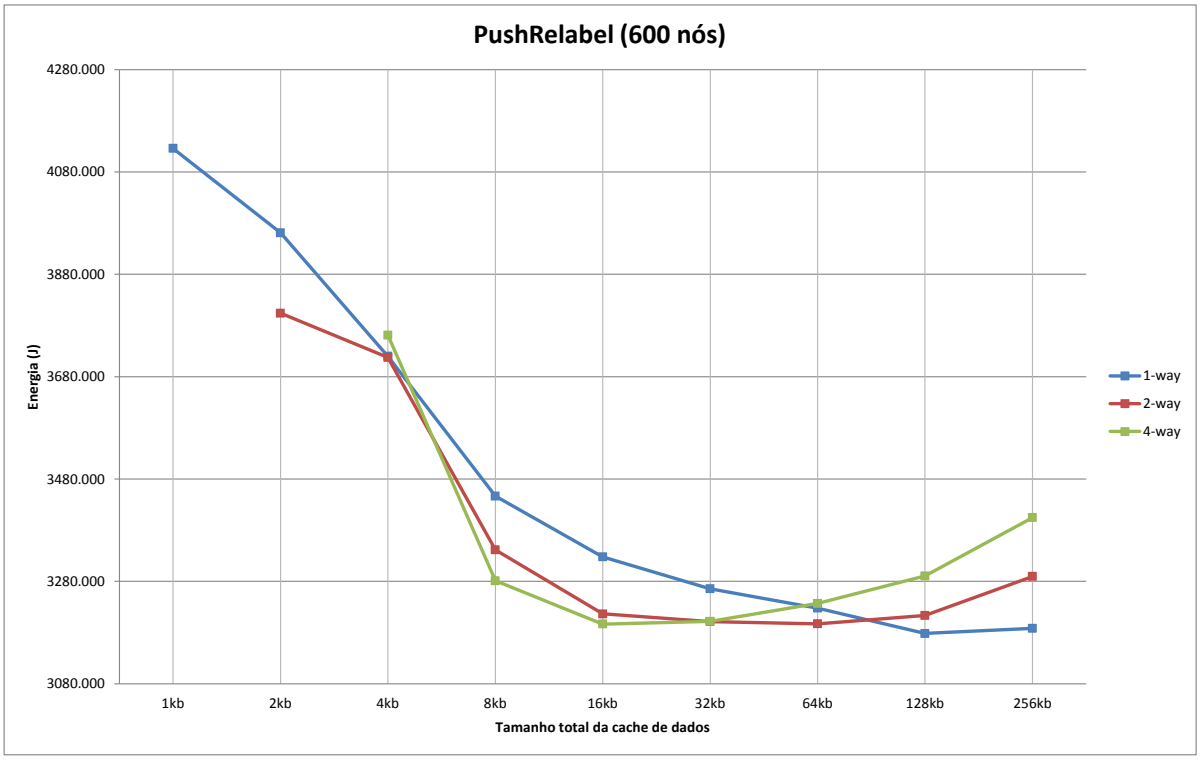


Figura 6.10: Consumo de energia da aplicação *Push Relabel* com 600 nós.

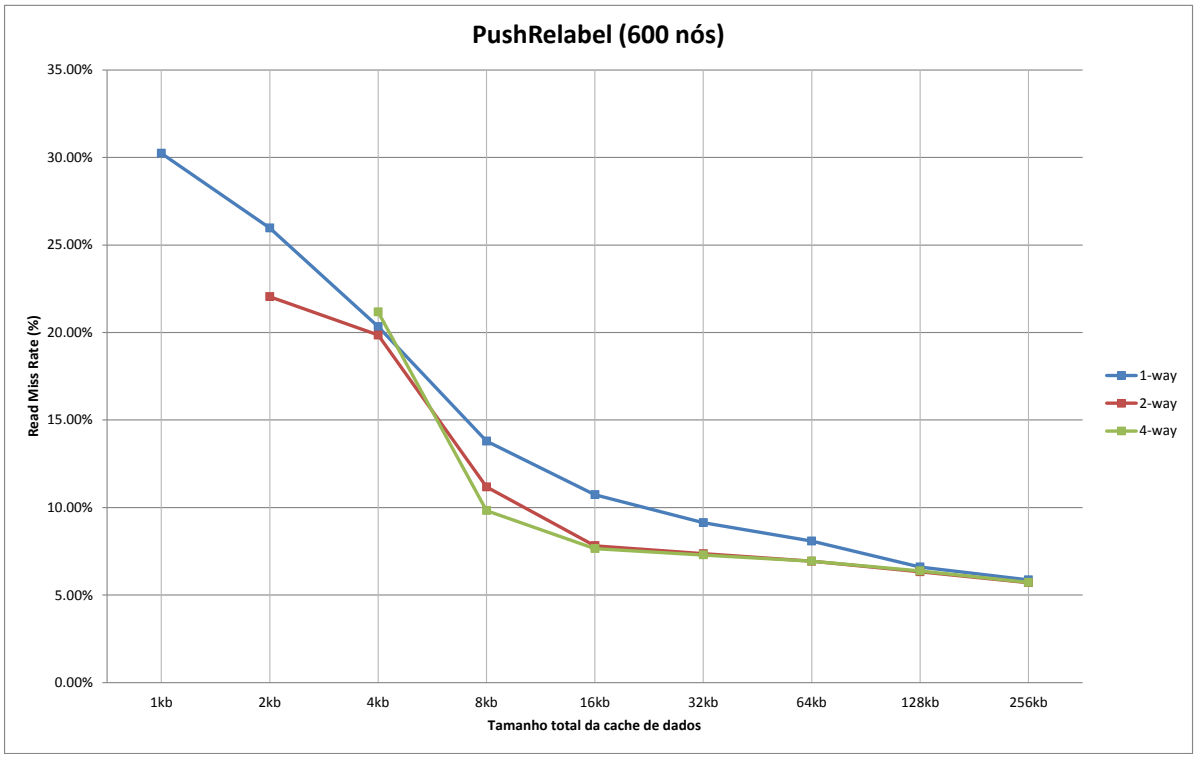


Figura 6.11: *Miss rate* da aplicação *Push Relabel* com 600 nós.

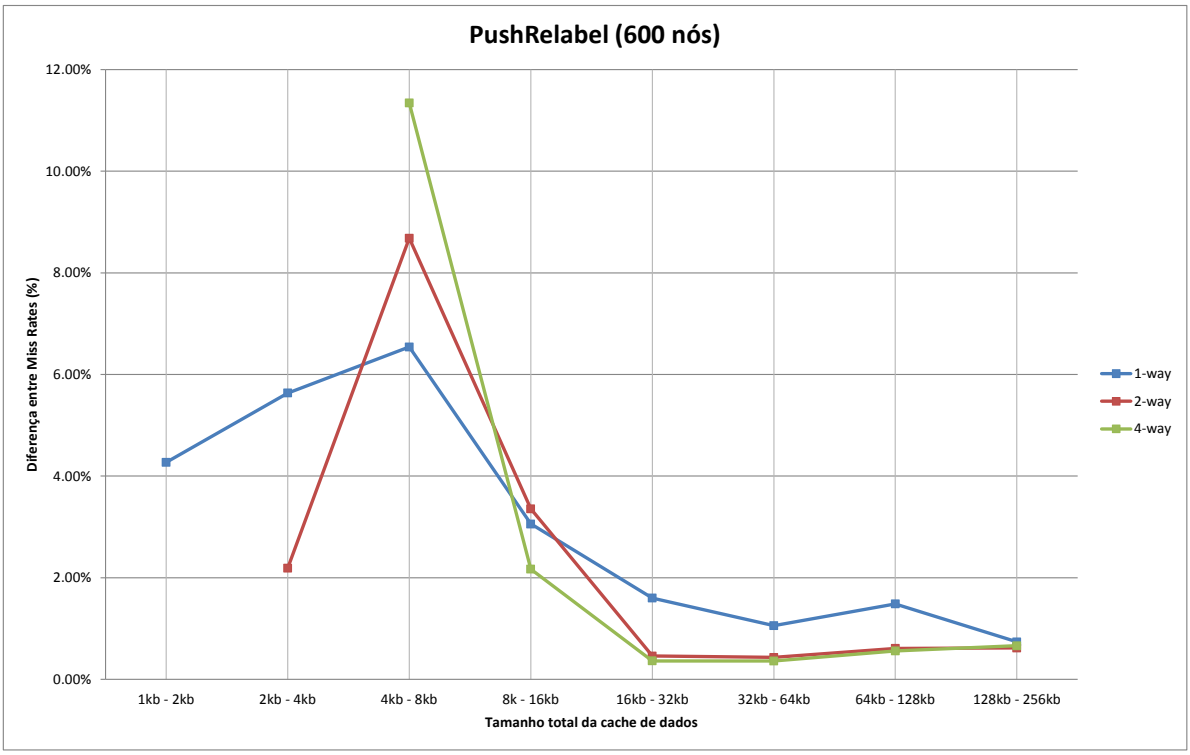


Figura 6.12: Diferença de *miss rate* entre configurações de *cache* adjacentes da aplicação *Push Relabel* com 600 nós.

6.1.4 Offline profiling

Considerando as possíveis configurações de *cache* apresentadas na Tabela 6.1, os resultados são apresentados na Tabela 6.2.

Tabela 6.2: Melhores configurações de *cache* para cada aplicação considerando o menor consumo de energia.

	Aplicação			
	BFS	Matriz esparsa (*Factor*)	Matriz esparsa (*Solve*)	*Push Relabel*
Configuração da *cache*	*4-ways* 64 KB	*1-way* 64 KB	*2-ways* 32 KB	*1-way* 128 KB

O algoritmo de multiplicação de matrizes esparsas pode ser dividido em duas etapas: *Factor*, onde há a decomposição da matriz, e *Solve*, onde acontece efetivamente a multiplicação da matriz. É importante notar que a fase *Factor* possui um perfil de *cache* diferente da fase *Solve*. Neste caso, a reconfiguração dinâmica da *cache* logo após a fase *Factor* pode reduzir o consumo de energia total e melhorar o desempenho da fase *Solve*.

6.2 Arquitetura com cache reconfigurável

As mesmas aplicações foram utilizadas para avaliar a arquitetura reconfigurável. Antes que cada algoritmo entrasse em execução, a rotina de inicialização do software de reconfiguração era chamada. Assim, o timer que ativa a interrupção de software e os contadores de *miss* e *hit* são ativados momentos antes da aplicação começar a executar. A arquitetura reconfigurável foi executada apenas na plataforma da Altera por indisponibilidade da plataforma da Xilinx.

A arquitetura reconfigurável possui fisicamente as 4-*ways* da *cache*, com 256 *KBytes* de tamanho total, e pode assumir as configurações apresentadas na Tabela 5.1 do Capítulo 5. Para efeitos de comparação, será considerada uma arquitetura de *cache* fixa com 4-*ways* e tamanho total de 256 *KBytes*. Escolher esta configuração para realizar a comparação faz todo sentido, pois a arquitetura reconfigurável possui configuração similar em termos de tamanho de *cache* e associatividade. A seguir são mostrados os resultados das aplicações sendo executadas na arquitetura com *cache* reconfigurável.

6.2.1 Aplicação Breadth-First Search

O gráfico na Figura 6.13 demonstra o *miss rate* da aplicação BFS durante sua execução na arquitetura com reconfiguração dinâmica. Este gráfico também inclui a diferença no *miss rate* entre cada intervalo de 140ms. Os intervalos foram agrupados para melhor apresentação gráfica. Percebe-se que o *miss rate* da aplicação permanece praticamente estável durante toda a execução da aplicação. Podemos comparar este resultado com o *miss rate* da *cache* na arquitetura fixa apresentada no gráfico da Figura 6.14. Em ambos os casos, o *miss rate* da aplicação variou em torno de 1%. Portanto, para esta aplicação, não houve degradação significativa no desempenho.

O gráfico na Figura 6.15 apresenta as configurações de *cache* que foram escolhidas durante a execução da aplicação. São mostrados apenas os primeiros 50 intervalos. Observa-se que o algoritmo de reconfiguração tentou reduzir a *cache* para a configuração com 2 *ways* diversas vezes. Entretanto, ao reduzir a *cache*, a diferença de *miss rate* entre o intervalo atual e o intervalo anterior ultrapassou o *MISS_THRESHOLD* definido como 1%. Assim, a *cache* retornou para a configuração ideal com apenas 3 *ways* ativas e tamanho total de 192 *Kbytes*. Este fato ocorreu devido a política do algoritmo de reconfiguração de sempre tentar reduzir o tamanho da *cache*. Neste caso, manter o tamanho da *cache* em 3 *ways* teria sido mais vantajoso do que tentar reduzir a *cache* diversas vezes. Cada redução de *way* implica em *compulsory misses* extras caso a redução não seja bem-vinda.

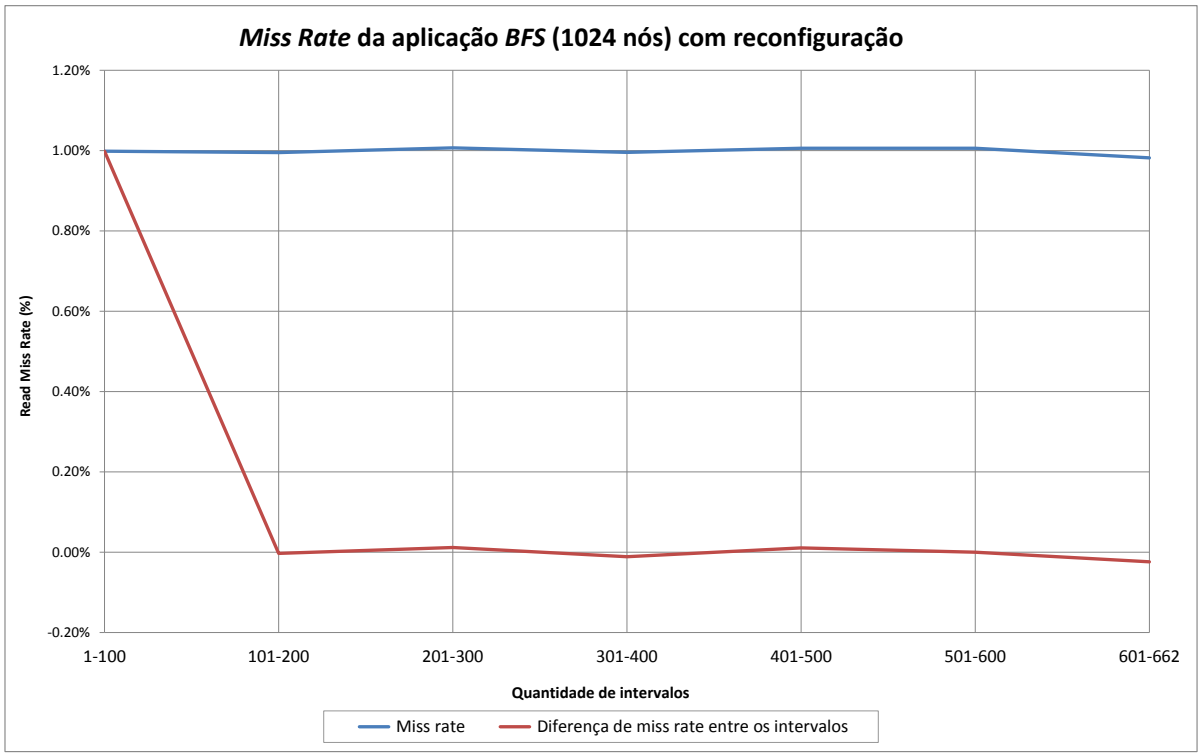


Figura 6.13: *Miss rate* da aplicação BFS e diferença de *miss rate* entre cada intervalo de 140ms na arquitetura reconfigurável.

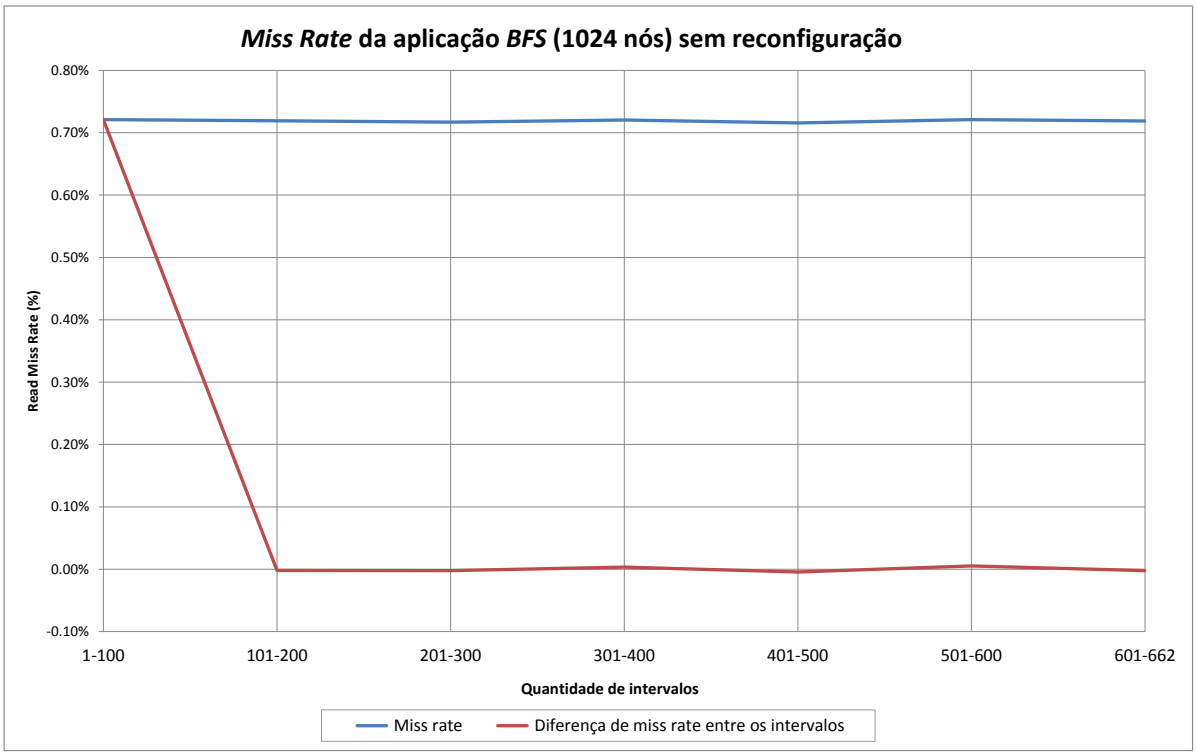


Figura 6.14: *Miss rate* da aplicação BFS e diferença de *miss rate* entre cada intervalo de 140ms na arquitetura de comparação.

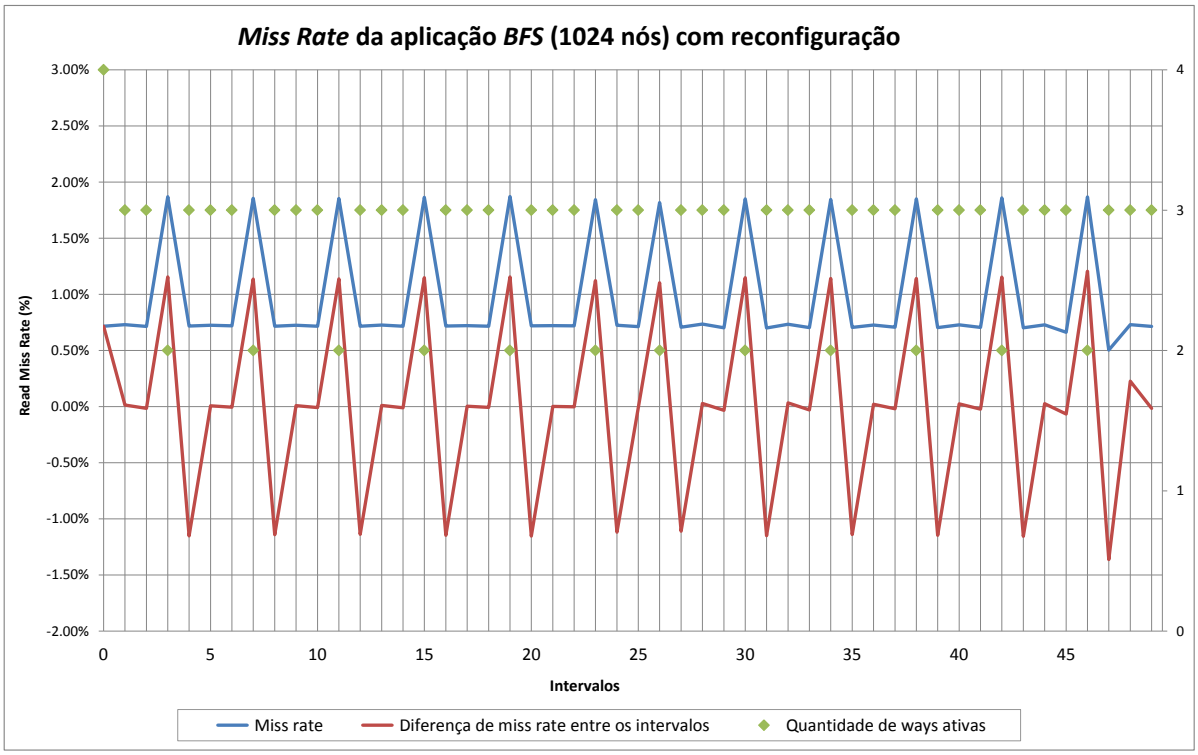


Figura 6.15: Configurações de *cache* escolhidas durante a execução da aplicação BFS, é apresentado apenas os 50 primeiros intervalos.

O gráfico na Figura 6.16 mostra a comparação dos resultados obtidos na arquitetura com *cache* reconfigurável e na arquitetura com *cache* de tamanho fixo.

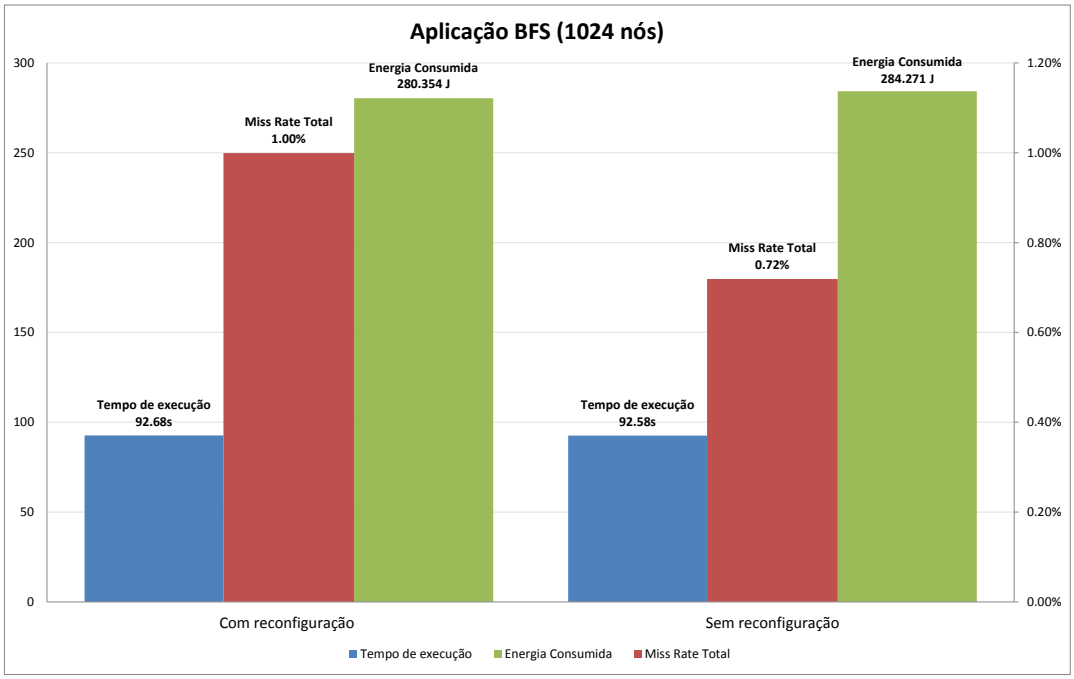


Figura 6.16: Comparação entre a arquitetura com *cache* reconfigurável e a arquitetura com *cache* de tamanho fixo, para a aplicação BFS.

Para esta aplicação, a redução no consumo de energia foi de aproximadamente 1.4% com degradação no desempenho de 0.1%. Se analisarmos a Tabela 6.2, veremos que a melhor configuração de *cache* para a aplicação BFS é a *cache* com 4-*ways* e 64 *Kbytes*. Como não é possível configurar dinamicamente a *cache* com os parâmetros ótimos, a configuração com 3-*ways* e 192 *KBytes* de tamanho total predominou durante a execução desta aplicação. Ainda, provavelmente a redução no consumo de energia não foi tão expressiva pois a arquitetura de comparação possui configuração de *cache* similar à configuração ótima definida para esta aplicação.

6.2.2 Multiplicação de Matrizes Esparsas

O gráfico na Figura 6.17 demonstra o *miss rate* da aplicação *Sparse Single*, com matriz 700x700 e 200.000 valores gerados de forma aleatória e diferentes de 0, durante sua execução na arquitetura com reconfiguração dinâmica. Este gráfico também inclui a diferença no *miss rate* entre cada intervalo de 140ms. Os intervalos foram agrupados para melhor apresentação gráfica.

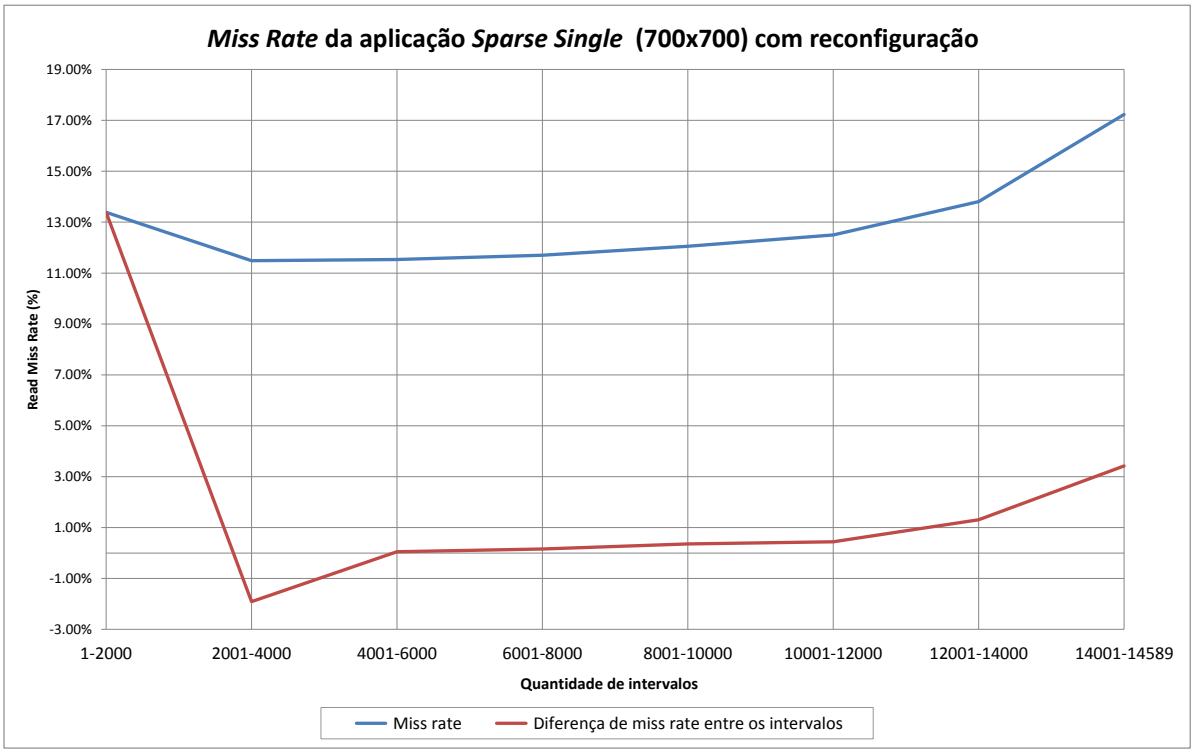


Figura 6.17: *Miss rate* da aplicação Sparse Single e diferença de *miss rate* entre cada intervalo de 140ms na arquitetura reconfigurável.

Percebe-se que o *miss rate* da aplicação tem uma variação bem maior se comparado ao algoritmo anterior, entre 11% e 17%. Podemos comparar este resultado com o *miss rate* da *cache* na arquitetura fixa apresentada no gráfico da Figura 6.18. Em ambos os casos,

o *miss rate* da aplicação variou praticamente dentro do intervalo 11% e 17%. Para esta aplicação também não houve degradação significativa no desempenho.

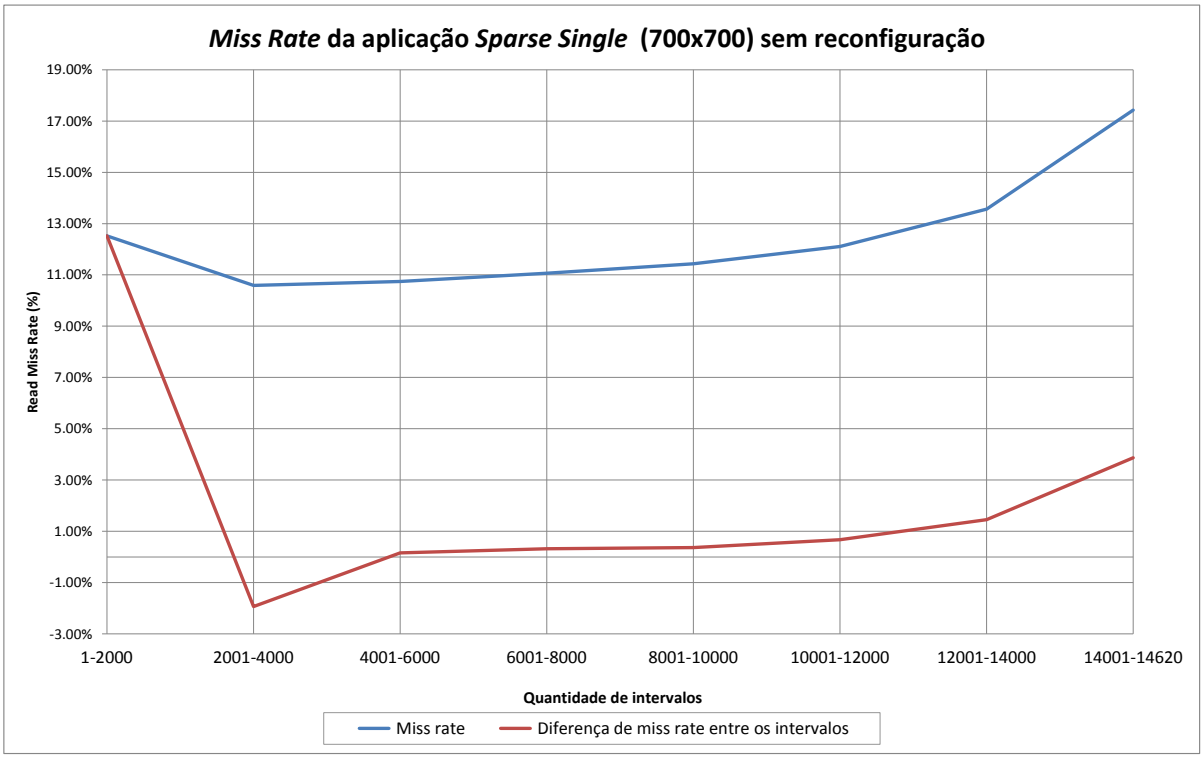


Figura 6.18: *Miss rate* da aplicação Sparse Single e diferença de *miss rate* entre cada intervalo de 140ms na arquitetura de comparação.

O gráfico na Figura 6.19 apresenta as configurações de *cache* que foram escolhidas durante a execução da aplicação. São mostrados apenas os primeiros 50 intervalos. Observa-se que o algoritmo de reconfiguração reduz a *cache* para 3 *ways* logo no segundo intervalo de execução. Entretanto, essa decisão faz o *miss rate* aumentar de 11% para 17% e logo na próximo intervalo, o algoritmo reconhece este aumento na taxa de *miss* e aumenta a *cache* novamente para 4-*ways*. Nos próximos intervalos o algoritmo consegue reduzir a *cache* novamente para 3-*ways* e 2-*ways* sem observar aumento na taxa de *miss rate*. Do intervalo 15 ao 39, a *cache* é configurada com apenas 1 *way* e a taxa de *miss rate* se estabiliza em 15%. Do intervalo 40 ao 50, observa-se como o algoritmo reagiu quando a diferença na taxa de *miss rate* entre o intervalo atual e o anterior ultrapassou o limite de 1%. A *cache* neste caso passou de 1-*way* para 4-*ways*.

O gráfico da Figura 6.20 apresenta a quantidade de vezes que cada configuração de *cache* foi escolhida durante a execução da aplicação. Os resultados estão agrupados em intervalos de 2000. Percebe-se que há predomínio da configuração com 1-*way* e 64 *Kbytes* de tamanho total. Ao final da execução as configurações de *cache* estão mais equilibradas, mas percebe-se que a configuração com 2-*ways* e 128 *Kbytes* começa a prevalecer dentre

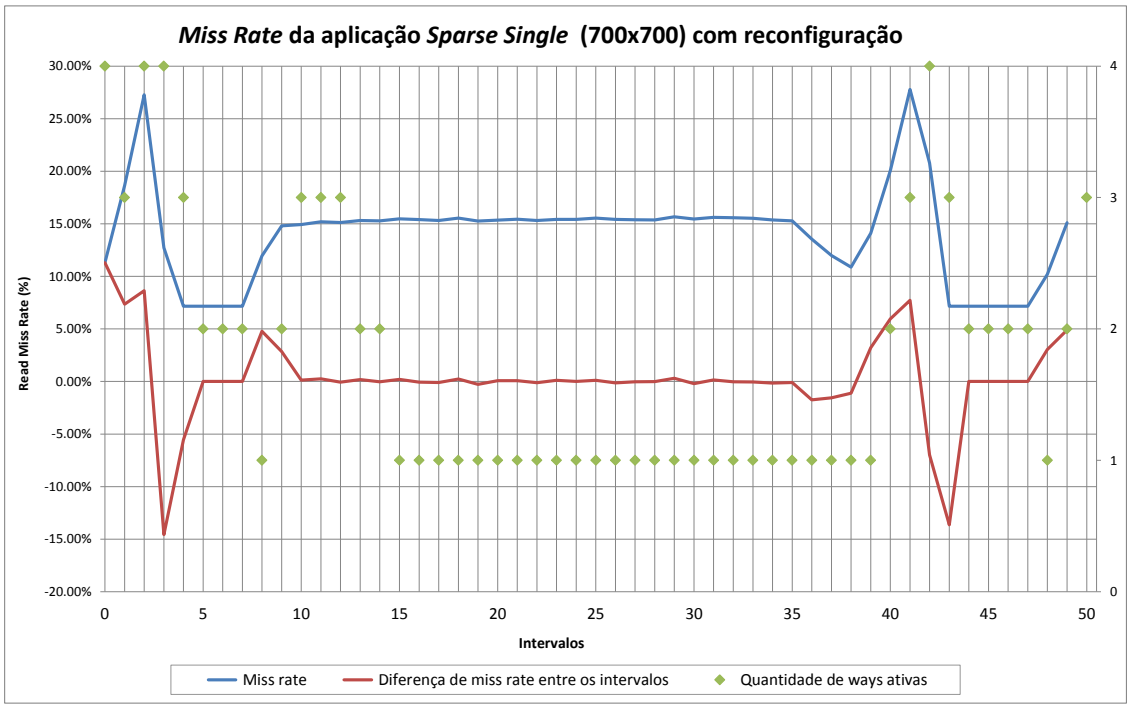


Figura 6.19: Configurações de *cache* escolhidas durante a execução da aplicação *Sparse Single*; é apresentado apenas os 50 primeiros intervalos.

as demais. A etapa inicial e intermediaria correspondem à fase *Factor* da aplicação e a etapa final corresponde à fase *Solve*.

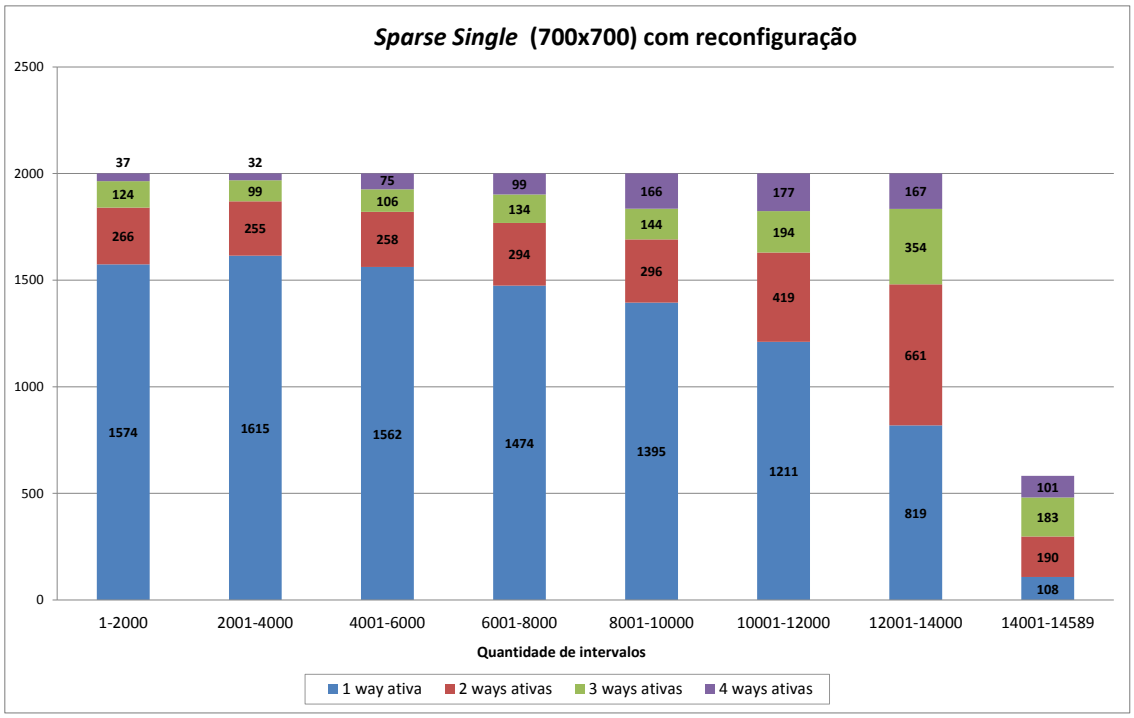


Figura 6.20: Configurações de *cache* escolhidas durante a execução da aplicação *Sparse Single*.

O gráfico na Figura 6.21 mostra a comparação dos resultados obtidos na arquitetura com *cache* reconfigurável e na arquitetura com *cache* de tamanho fixo. Para esta aplicação, a redução no consumo de energia foi de aproximadamente 3.66% com degradação no desempenho de 0.06%. Se analisarmos a Tabela 6.2, veremos que a melhor configuração de *cache* para a aplicação *Sparse Single* é a *cache* com 1-*way* e 64 *Kbytes* de tamanho total para a fase *Factor* e a *cache* com 2-*ways* e 32 *Kbytes* de tamanho total para a fase *Solve*. É interessante notar, analisando o gráfico na Figura 6.20, que o algoritmo de reconfiguração escolheu exatamente as configurações ótimas para a *cache*.

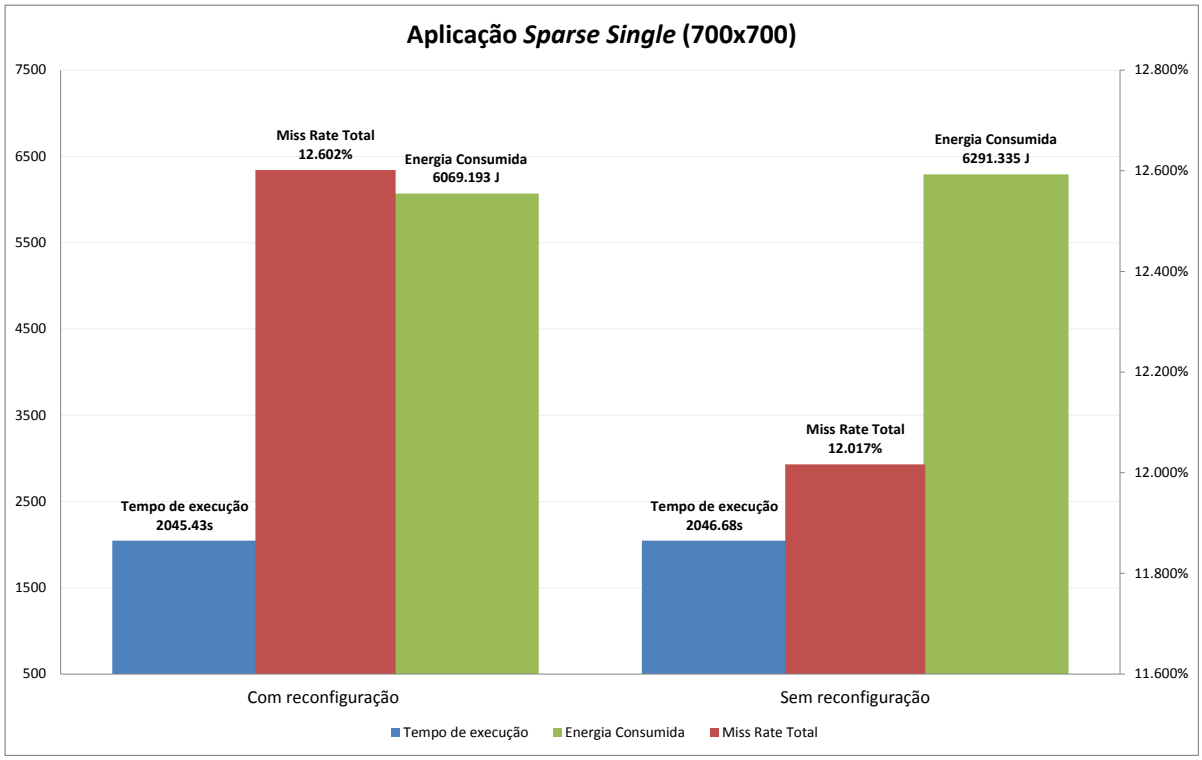


Figura 6.21: Comparação entre a arquitetura com *cache* reconfigurável e a arquitetura com *cache* de tamanho fixo, para a aplicação *Sparse Single*.

6.2.3 Algoritmo Push Relabel

O gráfico na Figura 6.22 demonstra o *miss rate* da aplicação *Push Relabel*, com 1000 nós, durante sua execução na arquitetura com reconfiguração dinâmica. Este gráfico também inclui a diferença no *miss rate* entre cada intervalo de 140ms e a quantidade de *ways* ativas nos intervalos. Os intervalos intermediários foram agrupados para melhor apresentação gráfica. Os intervalos finais e os dois últimos intervalos não foram agrupados pois o algoritmo apresenta comportamento diferente nesta etapa. Percebe-se que o *miss rate* da aplicação tem uma variação entre 2% e 4% durante praticamente todo o período. Há

uma variação extrema no final e o *miss rate* salta de 2% para mais de 18%. Entretanto, comparando este resultado com o *miss rate* da *cache* na arquitetura fixa apresentada no gráfico da Figura 6.23 percebe-se que o comportamento é similar. Em ambos os casos, o *miss rate* da aplicação variou praticamente dentro do intervalo 2% e 4%, atingindo 18% na etapa final da execução. Para esta aplicação também não houve degradação significativa no desempenho.

Ainda analisando a Figura 6.22 percebe-se que a *cache* com 1-*way* e 64 *Kbytes* de tamanho total prevaleceu durante praticamente toda execução da aplicação. No intervalo final da aplicação o algoritmo de reconfiguração aumentou a *cache* para 2-*ways* e 128 *Kbytes* de tamanho total, pois nesta etapa o *miss rate* da aplicação disparou para 18%.

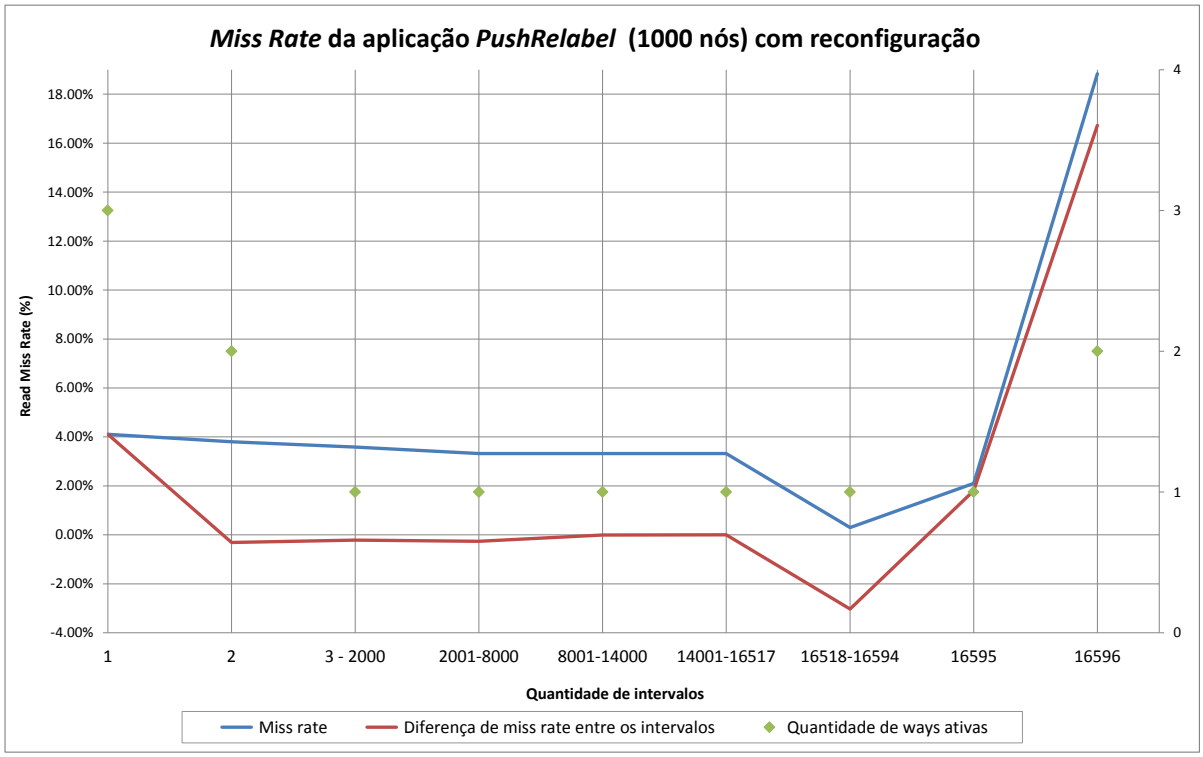


Figura 6.22: *Miss rate* da aplicação *Push Relabel*, diferença de *miss rate* entre cada intervalo de 140ms na arquitetura reconfigurável e quantidade de *ways* ativas durante o intervalo.

O gráfico na Figura 6.24 mostra a comparação dos resultados obtidos na arquitetura com *cache* reconfigurável e na arquitetura com *cache* de tamanho fixo. Para esta aplicação, a redução no consumo de energia foi de aproximadamente 4.13% com degradação no desempenho de 0.34%. Se analisarmos a Tabela 6.2, veremos que a melhor configuração de *cache* para a aplicação *Push Relabel* é a *cache* com 1-*way* e 128 *Kbytes*. Como não é possível configurar dinamicamente a *cache* com os parâmetros ótimos, a configuração com 1-*way* e 64 *KBytes* de tamanho total predominou durante a execução desta aplicação.

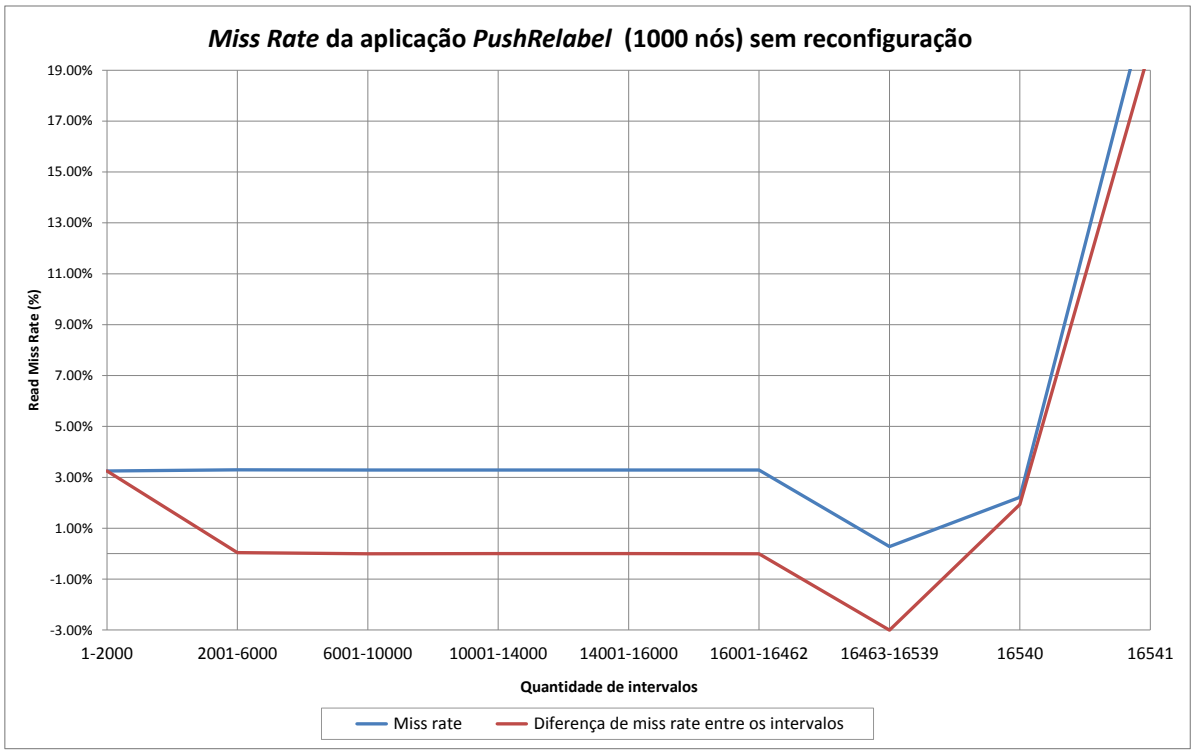


Figura 6.23: *Miss rate* da aplicação *Push Relabel* e diferença de *miss rate* entre cada intervalo de 140ms na arquitetura de comparação.

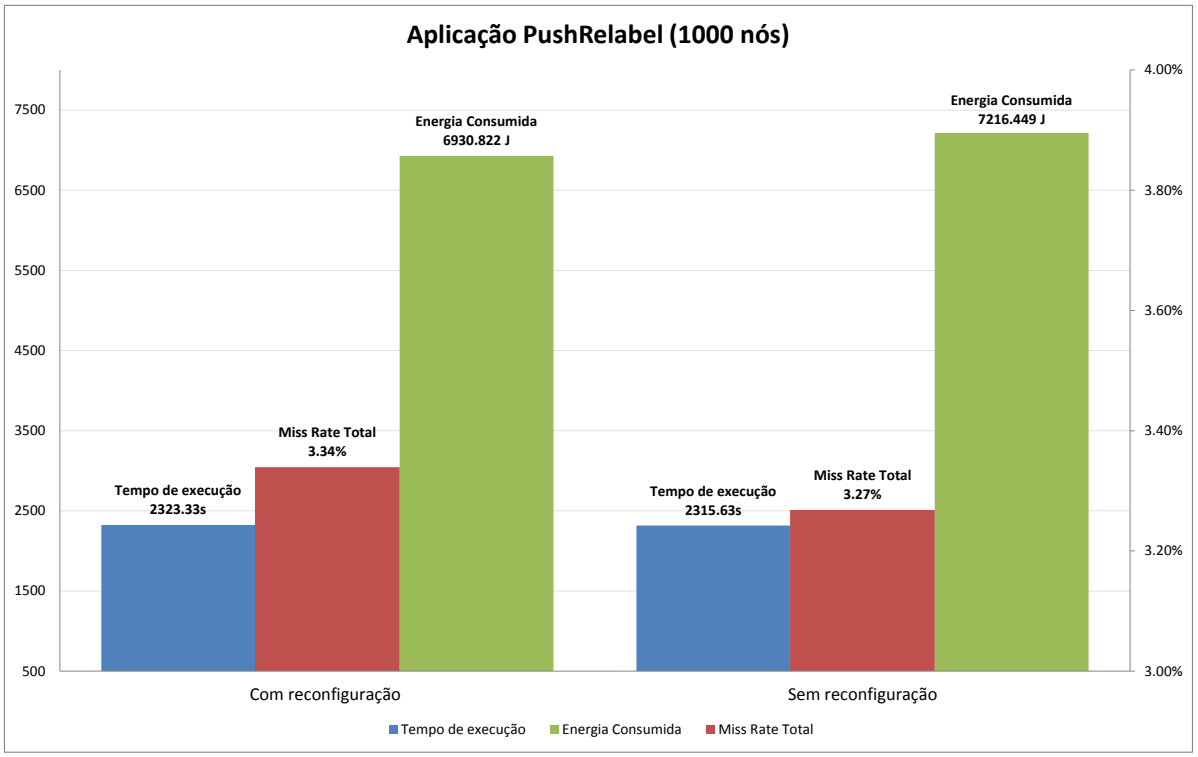


Figura 6.24: Comparação entre a arquitetura com *cache* reconfigurável e a arquitetura com *cache* de tamanho fixo, para a aplicação *Push Relabel*.

Capítulo

7

Conclusão

A reconfiguração dinâmica da *cache* se mostrou eficiente se considerarmos que as modificações no hardware foram simples e com *overhead* mínimo por conta do hardware adicional da solução proposta e do algoritmo de reconfiguração. Os contadores de *miss* e *hit* e o esquema de *way-shutdown* para habilitar e desabilitar as *ways* da *cache* foram as principais alterações realizadas no controlador da *cache* de dados do LEON3. Os contadores são implementados na forma de registradores de 32-*bits* e podem ser acessados através de duas instruções *assembly* especiais, conhecidas como *load alternative*, para ler o registrador, e *store alternative*, para escrever no registrador. Estas instruções recebem como parâmetro um identificador de endereçamento alternativo conhecido na arquitetura SPARC como ASI.

O esquema de *way-shutdown* controla o sinal de *enable* de cada *way* da *cache* de dados. Para desativar uma determina *way* da *cache* basta manter o sinal *enable* desta *way* no nível lógico "0". A *way* desativada não é acessada durante o estágio de *Memory Access* do *pipeline* do LEON3 e como consequência há redução no consumo de energia dinâmica. O controlador da *cache* mantém um registrador de 4-*bits* que indica quais *ways* da *cache* estão ativas. Para reconfigurar a *cache* dinamicamente basta alterar o valor deste registrador. Este registrador pode ser acessado e alterado da mesma forma que os registradores dos contadores de *miss* e *hit*.

O algoritmo de reconfiguração é implementado em uma interrupção de timer do LEON3. Este timer é configurado para gerar a interrupção a cada 140ms. Desta forma,

a solução proposta é transparente ao programador, pois para utilizar a solução proposta basta *linkar* o código do algoritmo de reconfiguração com a aplicação. Há apenas uma função que precisa ser chamada na aplicação para ativar a reconfiguração. O algoritmo analisa as taxas de *read miss* e *read hit* para tomada de decisão. O algoritmo se mostrou eficaz pois conseguiu aproximar-se das configurações de *cache* mais eficientes encontradas durante a fase de *offline profiling*, explicada na seção 6.1.

A integração do controlador de memória DDR3 da Altera com o LEON3 foi bastante complexo e trabalhoso. O controlador de memória DDR3 disponível pela Altera é um componente caixa preta e possui barramento *Avalon-MM*. Este barramento não é compatível com o barramento AMBA AHB/APB do LEON3. Para a integração, foi necessário desenvolver uma ponte entre o barramento *Avalon-MM* e o barramento AMBA AHB. Apesar de os sinais de ambos os barramentos serem parecidos, uma transação no barramento AMBA AHB é divido em duas fases, *address phase* e *data phase*, enquanto no barramento Avalon-MM a transação acontece em uma única fase. Desta forma, o componente desenvolvido precisou basicamente tratar desta diferença de fases para que as transações entre os barramentos fossem corretamente sincronizadas.

A solução proposta permite reduzir o consumo de energia de modo transparente ao usuário. Foi possível reduzir o consumo de energia da aplicações BFS em 1.40%, da aplicação *Sparse Single* em 3.66% e da aplicação *Push Relabel* em 4.13%, com degradação no desempenho que não ultrapassou 0.1% para todas as aplicações. Este tipo de otimização é importante principalmente para sistemas embarcados, pois estes sistemas frequentemente possuem fontes de energia de baixa capacidade. O recurso de reconfiguração dinâmica é um recurso novo e que não havia no LEON3. O projeto está disponível para outros usuários no seguinte link: `https://www.dropbox.com/sh/2a73k3j5cxzavxw/IBz513-Gk6`

7.1 Trabalhos futuros

Ainda é possível explorar a reconfiguração em trabalhos futuros onde seja possível alterar o tamanho da *cache* independentemente do grau de associatividade. Este trabalho, porém, é mais complexo, pois envolve modificar praticamente todo o código fonte do controlador da *cache* de dados do LEON3. Para altera o tamanho da *cache* é preciso alterar o tamanho do campos *tag* e *index* do endereço, conforme explicado no Capítulo 2. O principal fator que dificulta esta abordagem esta relacionado com a forma de implementação do controlador da *cache* de dados do LEON3. Vários barramentos de comunicação dentro do processador são calculados através de *"defines"* dentro do código fonte, ou seja, ao fazer a síntese o compilador calcula exatamente quais os tamanhos necessários dos barramentos e dos campos do endereço. Alterar a forma como o controlador da *cache* de dados interpreta

os *bits* do endereço afetaria praticamente todo o processador, inclusive o *pipeline* teria que ser alterado para suportar este tipo de reconfiguração dinâmica.

Referências

Abella, J.; González, A. Heterogeneous way-size cache. In: *Proceedings of the 20th annual international conference on Supercomputing*, 2006, p. 239–248.

Albonesi, D. H. Selective cache ways: On-demand cache resource allocation. In: *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, 1999, p. 248–259.

Altera Avalon interface specifications. 2010.
Disponível em `http://www.altera.com/literature/manual/mnl_avalon_spec_1_3.pdf`

Altera Nios ii processor reference handbook. 2011.

Altera Stratix iv gx fpga development board. 2012.
Disponível em `http://www.altera.com/literature/manual/rm_sivgx_fpga_dev_board.pdf`

ARM Ambaâ„¢ specification (rev 2.0). 1999.
Disponível em `http://ens.ewi.tudelft.nl/Education/courses/et4351/amba.pdf`

Asaduzzaman, A. Cache optimization for real-time embedded systems. Florida Atlantic University, 2009.

Asaduzzaman, A.; Limbachiya, N.; Mahgoub, I.; Sibai, F. Evaluation of i-cache locking technique for real-time embedded systems. *Innovations in Information Technology*, p. 342–346, 2007.

Bobda, C. *Introduction to reconfigurable computing: Architectures, algorithms, and applications*. Dordrecht: Springer, 483-488 p., 2007.

Brodal, G. S.; Fagerberg, R.; Meyer, U.; Zeh, N. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. *BRICS*, p. 1–19, 2004.

Calder, B.; Grunwald, D.; Emer, J. Predictive sequential associative cache. In: *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*, 1996, p. 244–253.

Computer History, T. o. Computer history museum. 2012.
Disponível em `http://www.computerhistory.org/timeline/?category=cmpnt`

Dean, J. Designs, lessons and advice from building large distributed systems. Big Sky, MT, 2009.

Eisele, K. *Design of a memory management unit for system-on-a-chip platform leon.* Dissertação de Mestrado, University of Stuttgart, 2002.

Gaisler, A. Aeroflex gaisler. 2012.
Disponível em `http://www.gaisler.com/cms/`

Gelsinger, P. Moore's law - the genius lives on. *IEEE SSCS NEWSLETTER*, p. 18–20, 2006.

Gordon-Ross, A.; Vahid, F. A self-tuning configurable cache. *Design Automation Conference*, p. 234–237, 2007.

Hennessy, J. L.; Patterson, D. A. *Computer architecture: A quantitative approach.* Amsterdam: Morgan Kaufmann, 2007.

Hill, M. D.; Smith, A. J. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 1989.

Inoue, K.; Ishihara, T.; Murakami, K. Way-predicting set-associative cache for high performance and low energy consumption. In: *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, 1999, p. 273–275.

Jacob, B.; W. Ng, S.; Wang, D. T. *Memory systems: Cache, dram, disk.* Maryland: Morgan Kaufmann, 2007.

Jaleel, A.; Theobald, K. B.; Steely, Jr., S. C.; Emer, J. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, v. 38, n. 3, p. 60–71, 2010.
Disponível em `http://doi.acm.org/10.1145/1816038.1815971`

Jheng, G.-C.; Duh, D.-R.; Lai, C.-N. Real-time reconfigurable cache for low-power embedded systems. *International Journal of Embedded Systems*, p. 235–247, 2010.

Kofuji, S. T.; Zuffo, J. A.; Soares, J. N. Circuitos integrados cmos. 2012. Disponível em `http://www.lsi.usp.br/~roseli/www/psi2307_2004-Teoria-8-CMOS.pdf`

Kulkarni, J.; Kim, K.; Roy, K. A 160 mv robust schmitt trigger based subthreshold sram. *IEEE Solid-State Circuits Society*, v. 42, p. 2303 – 2313, 2007.

Lee, D.; Choi, J.; Kim, J.-H.; Noh, S.; Min, S.-L.; Cho, Y.; Kim, C.-S. Lrfu: a spectrum of policies that subsumes the least recently used and least frequently used policies. *Computers, IEEE Transactions on*, v. 50, n. 12, p. 1352–1361, 2001.

Ma, A.; Zhang, M.; Asanovic, K. Way memoization to reduce fetch energy in instruction caches. In: *ISCA Workshop on Complexity Effective Design*, MIT, 2001.

Mahapatra, Nihar R., e. a. The processor-memory bottleneck: problems and solutions. *Association for Computing Machinery, Inc.*, 1999.

Marwedel, P. *Embedded system design*. Berlin: Springer US, 2006.

Patterson, D. A.; Hennessy, J. L. *Computer organization and design: The hardware/-software interface*. San Francisco, CA: Morgan Kaufmann, 2005.

Plavec, F. Soft-core processor design. University of Toronto, 2004.

Powell, M.; Agarwal, A.; Vijaykumar, T. N.; Falsafi, B.; Roy, K. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In: *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, 2001, p. 54–65.

Segars, S. Low power design techniques for microprocessors. In: *ISSCC*, San Francisco, 2001.

SPARC International, I. The sparc architecture manual: Version 8. Menlo Park, CA, 1992.

Steehler, J. K. Understanding moore's law—four decades of innovation (david c. brock, ed.). *Journal of Chemical Education*, p. 1278, 2007.

Sundararajan, K.; Jones, T.; Topham, N. Smart cache: A self adaptive cache architecture for energy efficiency. *ICSAMOS*, p. 41–50, 2011.

Xilinx Microblaze processor reference guide. 2008.

Xilinx Ml605 hardware user guide. 2012a.
Disponível em http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf

Xilinx Virtex-6 fpga ml605 evaluation kit. 2012b.
Disponível em http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm

Yang, S.-h.; Powell, M.; Falsafi, B.; Roy, K.; Vijaykumar, T. N. Dynamically resizable instruction cache: An energy-efficient and high performance deep-submicron instruction cache. *Purdue University*, 2000.

Zhang, C.; Vahid, F.; Lysecky, R. A self-tuning cache architecture for embedded systems. *ACM Trans. Embed. Comput. Syst.*, v. 3, n. 2, p. 407–425, 2004.
Disponível em http://doi.acm.org/10.1145/993396.993405