UM MODELO TRANSPARENTE DE MEMÓRIA *SCRATCHPAD* PARA ARQUITETURAS DE PROPÓSITO GERAL

FELIPE SILVA LOREDO

UM MODELO TRANSPARENTE DE MEMÓRIA *SCRATCHPAD* PARA ARQUITETURAS DE PROPÓSITO GERAL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Mariza Andrade da Silva Bigonha
Coorientador: Claudionor José Nunes Coelho Junior

Belo Horizonte

Fevereiro de 2015

Loredo, Felipe Silva.

L868u Um modelo transparente de memória Scratchpad para arquiteturas de propósito geral [manuscrito]. / Felipe Silva Loredo. - 2015.
xvi, 175 f. il.

Orientadora: Mariza Andrade da Silva Bigonha.
Coorientador: Claudionor José Nunes Coelho Junior.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 173-175
.
1. Computação – Teses. 2. Engenharia de software – Teses. 3 Compiladores (Computadores)- Teses. I. Bigonha, Mariza Andrade da Silva. II. Nunes Coelho, Claudionor José . III. Universidade Federal de Minas Gerais; Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Um modelo transparente de memória scratchpad para arquiteturas de propósito geral

FELIPE SILVA LOREDO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - UFMG

PROF. CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR - Coorientador
Departamento de Ciência da Computação - UFMG

PROF. ANTÔNIO OTÁVIO FERNANDES
Departamento de Ciência da Computação - UFMG

PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 27 de fevereiro de 2015.

Resumo

As memórias *scratchpad* são amplamente utilizadas em arquiteturas embarcadas para permitir a gerência da memória por *software*, mas tradicionalmente em projetos de propósito geral as aplicações existentes estão limitadas ao uso da *cache*, pois os benefícios dela e o código legado tornam as memórias *scratchpad* inviáveis. Este trabalho apresenta um modelo transparente para a memória *scratchpad* para arquiteturas de propósito geral, permitindo o seu uso quando ele é benéfico, sem gerar qualquer ruído para o código legado quando ele não é. Propõe-se uma abordagem mista para a gerência da memória no nível *on-chip* provendo a *cache* e a memória *scratchpad* ao mesmo tempo. Nos experimentos realizados uma melhoria de até 17% no desempenho foi observada. Como resultado, este trabalho apresenta uma solução para a produção de um ambiente diversificado, que concilia as demandas de eficiência com as de retrocompatibilidade, possibilitando atender de forma eficiente um maior número de aplicações com requisitos distintos.

Abstract

The scratchpad memories are widely used in embedded architectures to allow software memory management, but traditionally in general purpose projects existing applications are limited to the cache usage, because its benefits and the legacy code make the scratchpad memories unviable. This dissertation presents a transparent model for scratchpad memories for general purpose architectures, allowing its use when it is beneficial, without generating any noise to the legacy code when it is not. It is proposed a mixed approach to memory management in on-chip level providing the cache and scratchpad memory at the same time. In the experiments an improvement up to 17 % in performance was observed. As a result, this dissertation presents a solution for production of a diverse environment that accommodates the efficiency demands with backward compatibility, enabling efficiently meet a wider range of applications with different requirements.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Desde os primórdios da Computação a relação entre o processador e a memória forma um dos mais importantes gargalos da tecnologia (Patterson & Hennessy, 2012). Muito estudo tem sido desenvolvido na área de arquitetura de computadores para conciliar o processador rápido e a memória lenta, uma vez que melhorias nessa relação podem beneficiar o desempenho da máquina de maneira expressiva (Muchnick, 1997) (Hennessy & Patterson, 2007).

Em arquiteturas de propósito geral usualmente próximo ao processador há uma memória *on-chip*, pequena, cara e rápida empregada como uma *cache*. As organizações da hierarquia de memória controladas por *software* e as mistas - parte controladas por *software* e parte por *hardware*, comuns em ambientes embarcados, possuem em sua memória *on-chip* uma estrutura conhecida como memória *scratchpad*. *Local Stores* ou *Shared Memory* são outros nomes empregados para designar o mesmo recurso em diferentes arquiteturas. A memória *scratchpad* é um conjunto de endereços de memória com seus respectivos espaços de armazenamento, cuja gerência contrasta com as abordagens tradicionais, por precisar ser feita estaticamente ou controlada explicitamente pela aplicação. Geralmente a memória *scratchpad* é empregada para:

- dar ao processador a capacidade de realizar *stores* locais, no nível da *cache*, cujos dados são acessíveis de um a alguns poucos ciclos da CPU.
- Diminuir o consumo de energia e a área ocupada, uma vez que a lógica complexa da memória *on-chip* gerenciada por *hardware* (*cache*) pode ser dispensada.
- Fornecer garantias de tempo real ou previsibilidade ao desempenho obtido pela aplicação. Se a hierarquia de memória no nível da *cache* é gerenciada por *hardware*, é inviável predizer com precisão qual será o desempenho obtido, pois ele

depende das outras aplicações em execução e das decisões que serão tomadas pelo *hardware* e que estão fora do controle da aplicação.

- Melhorar a eficiência, pois o compilador ou o programador podem decidir quais dados serão alocados à *scratchpad*, e neste caso, decisões mais acertadas podem ser tomadas, elegendo dados críticos para serem armazenados no nível mais rápido da hierarquia. Além disso, a memória *scratchpad* posiciona-se como uma alternativa para os casos citados por Williams et al. (2007) em que conflitos e padrões complexos de acesso a dados não são tratados de forma eficiente pela *cache*.

- Melhorar o desempenho de aplicações paralelas. A *scratchpad* permite que as execuções concorrentes evitem conflitos nos acessos *on-chip*, além de facilitar o reúso de dados e reduzir o tráfego *off-chip*.

A gerência da memória por *software* situa-se no extremo oposto da hierarquia de memória tradicional que busca abstrair esta complexidade das aplicações. Ao prover esta alternativa, como contrapartida, a arquitetura repassa para a aplicação a responsabilidade de administrar os diferentes níveis da hierarquia de memória, solicitando a movimentação dos dados de forma explícita. Neste caso, algumas organizações de memória são possíveis quando consideram-se a *scratchpad* e a *cache* (Panda et al., 1999) (Cook et al., 2009):

1. a abordagem tradicional, onde só a *cache* está disponível e a hierarquia é completamente gerenciada por *hardware*,

2. a organização onde somente a memória *scratchpad* está disponível,

3. e a organização mista onde tanto a *cache* quanto a *scratchpad* estão disponíveis.

A Seção 2.2 discute com mais detalhes a memória *scratchpad*, suas vantagens e desvantagens.

O objetivo desta dissertação de mestrado inclui a discussão, especificação e o desenvolvimento de um modelo de hierarquia de memória para processadores de propósito geral, na tentativa de melhorar o desempenho da máquina com uma abordagem mista e transparente, provendo um *hardware* dotado da *cache* e da memória *scratchpad*. Para ratificar as ideias propostas e investigar sua viabilidade, foram realizados experimentos via simulação de *hardware*, comparando o comportamento e o desempenho obtido de uma amostra de aplicações executando em um ambiente tradicional e no ambiente proposto. Especificamente foram realizados:

1. a validação da arquitetura mista proposta via execução de experimentos no ambiente resultante.

2. Análise e validação do processo de compilação voltado para a arquitetura proposta.

3. Identificação, análise e validação do comportamento de otimizações considerando a arquitetura da hierarquia de memória mista proposta.

1.1 Motivação

Se ao longo do tempo a quantidade de memória disponível cresceu para atender a alta demanda das aplicações, por outro lado, a sua latência não diminuiu na mesma proporção. A Figura 1.1 mostra a diferença entre o ganho de desempenho do processador e da memória ao longo dos anos. Nela fica claro que apesar do aumento da quantidade de memória disponível, a disparidade entre a velocidade do processador e da memória aumentou de forma significativa. Isso é evidenciado pela diferença entre as curvas e pela escala logarítmica expressa no gráfico. Considerando este cenário, uma organização eficiente que consiga conciliar a crescente disparidade entre estes componentes da máquina é ainda mais importante, de modo que este espaço esteja disponível para os desenvolvedores com o menor tempo de acesso possível.

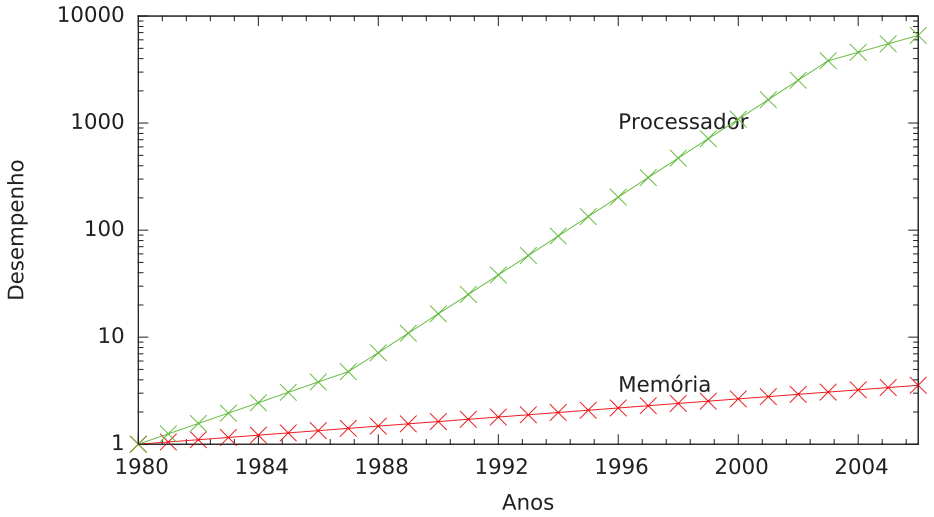

Figura 1.1: Desempenho do processador e da memória ao longo dos anos. Dados obtidos de Patterson & Hennessy (2012).

$$M_{16,16} = \begin{pmatrix} 254 & 1 & 183 & \dots & 122 & 47 & 14 \\ 67 & 56 & 220 & \dots & 225 & 226 & 49 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 10 & 56 & 88 & \dots & 232 & 248 & 184 \\ 210 & 15 & 71 & \dots & 53 & 53 & 147 \\ 87 & 74 & 18 & \dots & 255 & 95 & 214 \end{pmatrix}$$

Figura 1.2: Matriz M utilizada como entrada do Código 1.1 para calcular um histograma. M representa uma imagem com 16 linhas e 16 colunas.

A hierarquia de memória é o recurso empregado para amenizar os efeitos desse gargalo. Memórias de diferentes tamanhos implementadas com diferentes tecnologias são organizadas em uma configuração hierárquica que busca manter nos níveis mais rápidos e próximos ao processador os dados com maior probabilidade de uso futuro. Neste esquema a *cache* pode ser vista como uma interface entre o microprocessador e a memória lenta externa ao *chip*. Comumente nos processadores de propósitao geral a gerência da hierarquia é feita automaticamente pelo *hardware*, mas devido a forma como os dados são acessados pelas aplicações, alguns padrões de acesso podem degenerar a eficiência da abordagem que utiliza a *cache*. Para ilustrar este fato considere o exemplo apresentado a seguir.

Bastante utilizados na Computação Gráfica, os histogramas são uma estimativa gráfica da distribuição de probabilidade de uma variável contínua. Definem-se intervalos com determinadas larguras e a seguir o número de elementos pertencentes a cada um desses intervalos é computado. Por exemplo, utilizando uma imagem preto e branco com um sistema de cores cujos valores de cada pixel variam de 0 a 255, é possível criar um histograma supondo que os intervalos começam em 0, têm uma largura igual a 1 e portanto, possui 256 intervalos.

Código 1.1: Código utilizado para computar o histograma de uma imagem cujos valores de cada pixel variam de 0 a 255.

```
1 int histograma[256];
2 int p;
3 for (int i = 0; i < m; i++)
4         for (int j = 0; j < n; j++){
5                 p = M[i][j];
6                 histograma[p] = histograma[p] + 1;
```

Se a matriz $M_{m=16,n=16}$, conforme ilustrado na Figura 1.2, representa uma imagem 16x16 construída conforme descrito acima, o Código 1.1 é usado para computar o histograma da imagem dada por M. Nele o arranjo *histograma* declarado na linha 1 é utilizado para computar o histograma de M, o for da linha 3 percorre as linhas de M, o for da linha 4 percorre as colunas, na linha 5 a variável p recebe o valor do pixel da posição (i, j) de M e finalmente, para indicar que mais uma ocorrência da cor lida foi encontrada, p é utilizado na linha 6 para acessar a posição da cor no arranjo *histograma*, a incrementando em uma unidade. Neste exemplo dois padrões de acesso são ilustrados: os acessos a M e os acessos ao arranjo *histograma*. Os acessos à matriz M, realizados de cima para baixo e da esquerda para a direita, obtêm um bom desempenho utilizando a *cache*. Ao acessar o primeiro elemento da primeira linha, $(0, 0)$, uma falha compulsória ocorre e ao trazer esse dado dos níveis mais lentos da hierarquia de memória, todos os elementos seguintes na mesma linha também são trazidos, remetendo a um comportamento básico da hierarquia, pois esses elementos são alocados como posições subsequentes da memória e transferidos sob um mesmo bloco de dados. Dessa forma os próximos acessos resultam em acertos, até que a próxima linha seja alcançada e o processo repetido.

Dois pontos merecem destaque: 1) supôs-se que uma linha de M seria trazida de uma vez como um bloco, nesse caso o tamanho do bloco coincide com o número de elementos da matriz; 2) blocos menores ou linhas com número maior de elementos teriam um funcionamento análogo, mas com as falhas acontecendo em momentos distintos.

Finalmente, os acessos ao arranjo *histograma* do Código 1.1 precisam ser considerados. Para indexá-los, os valores das posições (i, j) de M são utilizados, o que potencialmente pode levar a uma não linearidade nesses acessos. Neste exemplo, o elemento $(0, 0) = 254$ é utilizado para indexar *histograma* na primeira iteração, enquanto na segunda, o elemento $(0, 1) = 1$ é usado. Portanto, na primeira iteração, a penúltima posição ou o último bloco que compõe *histograma* é necessário, enquanto na segunda, seria usado o primeiro. Esse exemplo evidencia a não linearidade no acesso aos dados do arranjo *histograma*, fazendo com que a *cache* não consiga prever de forma eficiente esses acessos. Em um contexto onde ela esteja disponível, uma alternativa para evitar todas as falhas nos acessos a esse arranjo, seria armazená-lo completamente na memória *scratchpad*. Outra otimização possível, seria carregar parcialmente os dados de M para uma porção da memória *scratchpad* antes deles serem necessários, evitando também as falhas compulsórias. Um leitor mais atento pode questionar que no máximo 17 falhas ocorrem nos acessos a *histograma* até que ele esteja inteiramente na *cache*. E embora regiões pouco utilizadas do arranjo *histograma* possam ser despejadas da *cache* para imagens muito grandes e que geram muitas falhas em um conjunto reduzido

de endereços, de modo geral a eficiência obtida pela *cache* é boa o suficiente para a maioria dos casos. Contudo vale salientar que outras classes de problemas com histogramas maiores são possíveis e essas classes poderiam apresentar um comportamento bastante diferente quando consideradas suas entradas. Este exemplo foi apresentado para ilustrar que há casos reais em que as técnicas existentes empregadas na hierarquia de memória gerida por *hardware* podem ser insuficientes para prover um número de falhas próximo ao mínimo global possível. Este trabalho propõe uma alternativa para atender esses casos, sem negligenciar o código legado e as aplicações que comportam-se melhor com a *cache*. Mais detalhes são apresentados nas Seções 1.2 e 1.3 e assunto é tratado em profundidade no Capítulo 3.

1.2 Definição do Problema

Considerando os casos em que algum rearranjo ou outras otimizações são possíveis e os casos em que essas otimizações não são viáveis, conforme ilustrado na Seção 1.1, uma reorganização da arquitetura da hierarquia de memória no nível da *cache* poderia resolver grande parte dos problemas relacionados a não linearidade dos acessos, e, ao mesmo tempo, tirar proveito quando os acessos são orientados pelos princípios de localidade, melhorando o desempenho da máquina e diminuindo o desperdício de memória para fazê-lo.

Nas abordagens tradicionais, a hierarquia é considerada uma hierarquia verdadeira, pois os dados só estão presentes em um nível se também o estiverem nos níveis subsequentes, ou em uma abstração análoga a essa ideia. A dissertação apresentada neste documento pretende romper a hierarquia verdadeira de modo que os dados em alguns casos específicos possam estar em um nível mais rápido sem estar nos níveis mais lentos. A *cache*, grande o suficiente para isso em máquinas modernas, seria reorganizada e dividida em duas partes, uma que opera de modo tradicional para explorar o princípio da localidade (a base da hierarquia de memória, vide Seção 2.1) e outra que seria gerenciada pela própria aplicação para tratar os casos em que a heurística do princípio da localidade não é aplicável (vide 2.1). Essa abordagem já é difundida nos microprocessadores embarcados, é conhecida como memória *scratchpad* (Panda et al., 1999). A título de ilustração, considerere novamente o trecho de Código 1.1. Se no lugar da abordagem tradicional da *cache* for considerada a memória *scratchpad*:

1. do modo mais simples, havendo espaço suficiente, M e *histograma* poderiam ser alocados na memória *scratchpad* e dessa forma, independente do padrão de acesso, os dados sempre estariam disponíveis na memória *on-chip*;

2. em um cenário em que não é possível armazenar M completamente na *scratchpad*, se instruções adequadas fossem fornecidas para isso, os dados poderiam ser mantidos, enviados e trazidos da memória principal, conforme a necessidade da aplicação - neste caso seria necessário algum acréscimo ao conjunto de instruções;

3. M poderia permanecer na *cache* e o arranjo *histograma* na memória *scratchpad*;

4. é válido também, assumir que não é possível alocar ambos na *scratchpad* e, neste caso, a *cache* seria utilizada com um desempenho análogo à abordagem tradicional.

A memória *scratchpad* é extremamente comum em arquiteturas embarcadas, pois nelas o conjunto de aplicações a serem executadas é bem definido, e portanto, é possível tirar proveito dessa organização para otimizar o desempenho, o consumo de energia e minimizar o custo e a área ocupada. Quando as arquiteturas embarcadas são comparadas com as arquiteturas de propósito geral, as aplicações executadas no *hardware* de propósito específico são conhecidas e bem definidas, assim, elas são compiladas menos vezes e principalmente por ocasião do projeto do *chip*. Dessa forma, por ocorrer menos vezes e em um ambiente de desenvolvimento mais controlado, o tempo gasto na compilação dessas aplicações pode ser maior, podendo-se investir mais tempo na aplicação de otimizações do código, desde que seja mantida a restrição de que o tempo gasto, embora possa ser mais longo, seja aceitável. Em contrapartida, o uso da memória *scratchpad* é algo raro nas arquiteturas de propósito geral. Se opondo as soluções embarcadas onde as aplicações a serem executadas são fixas e pré-determinadas, os ambientes de propósito geral normalmente possuem uma grande quantidade de código legado e executam uma grande variedade de aplicações com os mais diversos objetivos. Nesta conjuntura, vários desafios se impõem para a adoção da memória *scratchpad*. Há, dentre outras, algumas questões que foram identificadas e respondidas ao longo do desenvolvimento desse trabalho de pesquisa:

1. como resolver o fato de que o código legado precisa continuar funcionando nas novas arquiteturas? Para que isto ocorra, o projeto da memória *scratchpad* precisa ser um recurso opcional para os novos projetos e transparente para os antigos. As aplicações podem o utilizar se quiserem, mas podem ignorá-lo sem que isso impacte no seu código e impacte pouco no seu desempenho.

2. A troca de contexto gerenciada pelo sistema operacional é afetada pela existência da memória *scratchpad*, uma vez que ela representa uma quantidade de dados

extras relativos aos processos. Como tratar o fato de que esta quantidade considerável de dados extras precisam ser salvos na troca de contexto? A abordagem proposta é mista e nela a memória *scratchpad* é tratada como um subconjunto da *cache*. Portanto, utilizando-se da infraestrutura já existente, os dados adicionais podem ser tratados de forma simples por uma abordagem similar àquela utilizada tradicionalmente pela *cache*.

3. A quantidade de memória rápida disponível próxima do processador é limitada e pode ser mais interessante, do ponto de vista da eficiência obtida pela hierarquia de memória, mantê-la toda disponível sob a organização de uma *cache*. De modo geral, é benéfico para as aplicações possuir uma parte do nível mais alto da hierarquia de memória gerenciável? Quais os *trade-offs* envolvidos? A gerência da memória *on-chip* por *software* pode melhorar a eficiência das aplicações, mas de modo geral aumenta a complexidade a ser tratada pelos desenvolvedores. Aplicações com uma alta localidade temporal ou espacial tendem a se beneficiar da *cache*. Nesses casos a utilização da *scratchpad* é difícil de ser justificada. Por outro lado, há aplicações que demandam alta eficiência e têm um padrão de acesso a dados complexo, não apresentando um bom desempenho com a *cache*, nesses casos a memória *scratchpad* representa uma alternativa melhor. Dessa forma, uma abordagem ideal deve oferecer os recursos para esses dois tipos de aplicações, aquelas que se beneficiam da *scratchpad* deveriam tê-la a disposição sem influenciar as que lidam melhor com a *cache*.

4. As linguagens de programação podem ser consideradas uma camada de uma série de abstrações que são utilizadas para tratar a complexidade das máquinas. Dessa forma, dentre outros fatores, os recursos disponibilizados por elas fazem dos compiladores, também, responsáveis pela eficiência obtida pelas aplicações em uma determinada arquitetura. Como o compilador deve ser organizado e quais otimizações devem ser propostas quando se considera a possibilidade da memória *scratchpad*? Como as otimizações tradicionais se comportam neste novo cenário? Como a reorganização do código deverá ser conduzida para obter um bom desempenho da máquina minimizando o desperdício de memória? Para responder a essas questões e permitir o uso simplificado da gestão da memória por *software*, criou-se uma biblioteca para linguagens de programação de alto nível. Embora as otimizações padrão do compilador tenham sido mantidas no desenvolvimento dos experimentos deste trabalho, as otimizações relacionadas a memória *scratchpad* foram realizadas manualmente no nível da linguagem de alto

nível utilizando a biblioteca criada. O resultado dessa abordagem foi verificado e avaliado, conforme discutido no Capítulo 4.

5. Quais alterações são necessárias no conjunto de instruções para prover um ambiente misto em que exista uma *cache* e uma *scratchpad*? Para responder a essa questão introduziram-se quatro instruções de máquina. Elas permitem alocar e liberar a memória na *scratchpad* e realizam a movimentação de dados entre a memória principal e a memória *on-chip*.

6. Como processadores paralelos seriam afetados pela existência da memória *scratchpad*? A memória *scratchpad* é uma solução para aplicações paralelas evitarem conflitos nos acessos *on-chip*, facilitar o reúso de dados e reduzir o tráfego *off-chip*.

1.3 Metodologia

Apesar de ser consenso nas arquiteturas de propósito geral, a *cache* deixa lacunas para diversas classes de algoritmos que poderiam ser beneficiadas pela gestão da memória por *software*, como processamento multimídia, aplicações de tempo real e aplicações científicas. A *cache* é uma solução genérica adotada para todas as aplicações, mas o padrão de acesso aos dados pode ser distinto entre essas aplicações e pode ser ainda, diferente do esperado pelas heurísticas espaciais e temporais (vide Seção 2.1). Um bom conhecimento do domínio desses problemas, pode permitir uma gestão mais eficiente do pouco e concorrido espaço disponível na memória próxima ao processador e por consequência produzir uma solução mais eficiente. Espera-se, com este trabalho de pesquisa avançar nessa direção, conciliando as demandas da maioria das aplicações com a introdução de novos recursos para a gestão benéfica da memória por *software*.

Neste cenário, dois pontos precisam ser considerados: 1) *hardware* – quais alterações são necessárias para a introdução da memória *scratchpad*; 2) *software* – como o recurso pode ser disponibilizado para os desenvolvedores. Para o *hardware*, propôs-se a inclusão de um *bit* de controle S para cada bloco da *cache* de tal forma que, quando assinalado, ele determina um tratamento especial para aquele espaço, impedido que ele seja submetido às políticas de substituição da *cache*. Associado a ele, instruções de máquina adequadas foram definidas para alocar, liberar e transferir dados para essas posições de memória. Para o *software*, a fim de permitir a utilização desses recursos, além da linguagem de máquina apropriada, definiu-se uma biblioteca de alto nível contendo quatro funções, para alocação, liberação e transferência de dados da e para a memória principal.

Com o objetivo de mostrar a viabilidade da solução proposta, na condução dos experimentos foi selecionado e usado um *benchmark* com um conjunto de cinco aplicações, mensurando o desempenho e o fluxo de dados entre a memória *on-chip* e a memória principal, além de analisar outros aspectos não previstos inicialmente pela abordagem.

Especificamente, os experimentos visaram: (1) verificar os resultados obtidos por uma reengenharia transparente da hierarquia de memória para contemplar a memória *scratchpad*; (2) melhorar a taxa de acertos quando comparada com as abordagens tradicionais, medindo essa diferença para identificar quais as implicações dessa abordagem e quais são os resultados obtidos pela aplicação da proposta. Durante a execução dos experimentos foram observados ganhos no desempenho de até 17%, mas em contrapartida, no pior caso, o código gerado pelo compilador aumentou em até 102%, com o compilador configurado para efetuar o maior número de otimizações possível.

1.4 Contribuições

Solucionados os desafios enumeradas na Seção 1.2, a dissertação proposta neste documento produziu como resultado as seguintes contribuições:

1. um simulador de *hardware* que provê a ideia da arquitetura mista com memória *scratchpad* e a *cache*.

2. Uma memória *scratchpad* que permite a gerência da memória por *software* utilizando uma abordagem transparente, permitindo seu uso quando ele é benéfico, sem gerar qualquer ruído quando ele não é.

1.5 Organização do Texto

Esta dissertação está organizada em cinco capítulos, incluindo este capítulo introdutório.

O **Capítulo 2** apresenta a revisão de bibliografia relacionada ao tema proposto.

O **Capítulo 3** apresenta a proposta deste trabalho para prover um ambiente misto em que tanto a *cache* quanto a memória *scratchpad* estejam disponíveis. Ele apresenta as alterações realizadas no *hardware*, as alterações efetuadas no conjunto de instruções, passando pela abordagem utilizada no âmbito do *software*, apresentando uma biblioteca de alto nível que viabiliza o uso da memória *scratchpad*, seguida pela

discussão do processo de compilação empregado e finalizando com o simulador utilizado para verificar a proposta e realizar os experimentos.

O **Capítulo 4** apresenta o conjunto de experimentos realizados para verificar e avaliar a proposta utilizada. Inicialmente discute-se cada um dos experimentos que compõem o *benchmark*, seguido pelo detalhamento da sua execução. O capítulo termina com a análise dos resultados obtidos.

O **Capítulo 5** finaliza este texto, faz um apanhado geral do trabalho, discute os resultados obtidos e enumera alguns trabalhos futuros que poderiam se originar a partir desta pesquisa.

Capítulo 2

Revisão Bibliográfica

2.1 Hierarquia de Memória

Burks, Goldstine e Von Neumann em *"Preliminary discussion of the logical design of an electronic computing instrument"*, dizem: "Idealmente seria desejável uma memória de capacidade indefinidamente grande, tal que qualquer agregado particular de 40 dígitos binários, ou palavras, pudesse estar imediatamente disponível, isto é, em um tempo que é comparável ou consideravelmente menor que o tempo de operação de um multiplicador eletrônico rápido..." "Parece não ser fisicamente possível alcançar tal capacidade. Nós somos, portanto, forçados a reconhecer a possibilidade de construir uma hierarquia de memórias, cada uma com uma capacidade maior que a anterior, mas que é menos rapidamente acessível" (Burks et al., 1946). Este pequeno trecho ilustra duas questões relevantes que se mantêm desde os primórdios da Ciência da Computação: há uma diferença considerável entre a latência da memória e a velocidade de operação dos processadores e os programadores, embora o *hardware* não forneça, desejam quantidades ilimitadas de memória rápida.

Limitações tecnológicas e custos de produção sempre impediram que a memória fosse grande e tão rápida quanto o processador. A tecnologia se desenvolveu de modo que a quantidade de memória disponível cresceu, porém o ganho de desempenho em relação a latência não acompanhou esse ritmo. Assim, foi necessário criar um mecanismo para prover a abstração, do ponto de vista do *software*, de uma memória grande e rápida. Em linhas gerais, a hierarquia de memória.

Embora os pioneiros da computação já tivessem antevisto a necessidade da hierarquia, a primeira proposta de gerência automática desse mecanismo surgiu em 1962 na Universidade de Manchester (Kilburn et al., 1962) para memória virtual. Na época a IBM propôs o mecanismo para sua linha de computadores, porém o recurso foi adiado

e disponibilizado apenas em 1972 para a família 370. E a quase setenta anos depois, as questões levantadas pelos pioneiros da computação (Burks et al., 1946) ainda continuam relevantes:

1. a discrepância entre a velocidade da memória e do processador pode chegar a três ordens de grandeza em um cenário atual. A Figura 1.1 mostra que ao longo dos anos o ganho de desempenho dos processadores aumentou a uma taxa muito superior a da memória.

2. Toda a execução de uma ou mais instruções envolvem um ou mais acessos a memória, uma vez que no modelo de Von Neumann, a base das máquinas até hoje, dados e instruções são tratadas de maneira uniforme.

3. Com a crescente demanda das aplicações por mais memória, o descompasso entre o tempo de acesso aos dados na memória e a velocidade do processador (Figura 1.1) e o fato de que mais espaço implica em menos velocidade produz um dos mais importantes gargalos da tecnologia.

4. A ideia da hierarquia de memória perdura e é um dispositivo básico até hoje para mediar a relação entre a memória grande o suficiente para as necessidades das aplicações, mas lenta do ponto de vista da CPU.

Muito trabalho foi desenvolvido na área e a hierarquia de memória é algo presente em praticamente todas as arquiteturas modernas de propósito geral (Patterson & Hennessy, 2012). Ela baseia-se na transferência dos dados entre seus diferentes níveis sob o formato de blocos. Se o conteúdo desses blocos for composto pelos próximos dados a serem solicitados pelo processador, cria-se a ilusão de uma memória grande e rápida. O bom desempenho da hierarquia está baseado no princípio da localidade das aplicações. A ideia é que na maior parte do tempo elas acessam uma pequena porção do seu espaço de endereçamento. Assim, a gerência automática dos dados é realizada mantendo-se próximo ao processador, em um nível mais rápido e menor, os dados que se supõe que serão utilizados em um futuro próximo. Há dois tipos de localidade:

1. localidade temporal: um dado usado recentemente será utilizado novamente em breve;

2. localidade espacial: os dados criados e armazenados em uma região próxima ao dado usado por último, serão utilizados em breve.

O objetivo é que a cada nível da hierarquia, a memória mais rápida, cara e escassa armazene os dados com maior probabilidade de uso em um futuro próximo. Em

uma organização comum para processadores de propósito geral, no nível mais rápido tem-se a *cache* (geralmente implementada como uma SRAM), seguida pela memória principal (geralmente implementada como uma DRAM) e pela memória secundária (disco magnético). Variantes deste modelo são bem comuns, como por exemplo, uma hierarquia com vários níveis de *cache*. De modo geral, a organização tradicional mais simples é suficiente para o estudo e o desenvolvimento de esquemas mais elaborados, uma vez que o princípio é análogo.

O diagrama da Figura 2.1 esquematiza a hierarquia de memória resumindo essas ideias. No topo da pirâmide está a memória de menor latência e menor abundância, na base, o oposto, maior tempo de acesso e mais espaço de armazenamento.

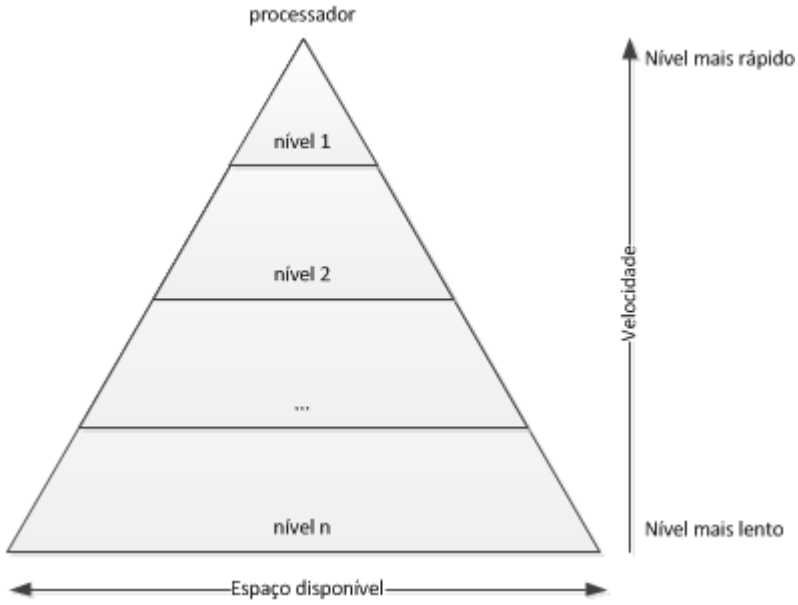

Figura 2.1: Esquema para a hierarquia de memória. Adaptada de (Patterson & Hennessy, 2012).

Para resolver a disparidade entre a velocidade de operação do processador e da memória, Wilkes (1965) sugeriu pela primeira vez a inclusão da memória *cache* na hierarquia chamando-a, na ocasião, de *slave*. A *cache* pode ser dividida em uma *cache* de instruções e uma *cache* de dados ou em um único *chip* que trata todos os dados de maneira uniforme. Na maioria das máquinas recentes, a *cache* integra o mesmo *chip* da CPU (memória *on-chip*) e os tempos de acesso variam de um a poucos ciclos da máquina. Com tamanhos relativamente grandes nos processadores modernos, chegando até 8 MB em microprocessadores da família *Haswell* da Intel®(Intel Corporation, 2013) - nesses processadores cada *core* possui 32 KB para dados e 32 KB para instruções na *cache* L1, 256 KB para cada *core* do processador para a *cache* L2 e até 8 MB para a *cache* L3 que é compartilhada entre todos os *cores*.

Uma das grandes vantagens da gestão automática da hierarquia de memória é sua transparência. As aplicações não têm conhecimento e nem obrigação de gerenciar qualquer nível da hierarquia, trabalhando como se a memória fosse uma única entidade de endereços sequenciais e contíguos. Isso inclui uma camada de abstração que permite reduzir a complexidade visível para os programadores. Porém, embora essa solução apresente um bom desempenho para a maioria das aplicações, trata-se de uma solução genérica que espera acessos a memória com um determinado padrão. Quanto maior a localidade dos dados, mais eficiente será a execução da aplicação. Já aplicações que não apresentam um padrão predizível pelas aproximações temporal e espacial podem ser penalizadas. Este trabalho busca permitir que essas aplicações possuam uma alternativa viável para atender as demandas por mais eficiência quando elas existirem.

No contexto da hierarquia de memória definem-se os acertos e as falhas. Os acertos são os acessos a memória em que os dados são encontrados nos níveis superiores da hierarquia. Portanto, o tempo de acerto refere-se ao tempo total gasto para acessar os dados solicitados, isto é, o tempo de acesso ao nível mais alto da hierarquia incluindo-se aí, o tempo necessário para acessar os dados propriamente ditos, determinar se o acesso é um acerto ou uma falha e o tempo necessário para transferir os dados desse nível para o processador. Em contrapartida as falhas tratam os casos em que os dados solicitados não estão presentes no nível mais alto da hierarquia e assim, o tempo de falha ou a penalização por uma falha, inclui o tempo necessário para localizar, acessar e trazer os dados do nível subsequente para o nível mais rápido, mais o tempo de transferir esses dados para o processador. A taxa de acertos pode ser utilizada portanto, como uma medida de desempenho para a hierarquia, isto é, a proporção de acessos à memória que são encontrados nos níveis superiores da hierarquia. Considerando que os níveis mais próximos do processador são por construção menores, contudo mais rápidos, quanto maior a taxa de acertos melhor o desempenho do sistema como um todo.

No limite, quando a taxa de acertos é alta o suficiente, o desempenho do sistema se comporta como se a velocidade da memória fosse a do nível mais próximo ao processador, mas com o tamanho dos níveis mais distantes, isto é, uma memória grande e rápida. Portanto, é fácil entender por que o principal objetivo das organizações da hierarquia de memória é maximizar a taxa de acertos de todos os níveis.

2.2 Memória *Scratchpad*

O projeto de processadores embarcados geralmente possuem requisitos muito rígidos quanto a consumo de energia e área ocupada, podendo também figurar outros itens

como por exemplo, garantias de tempo-real e eficiência de execução. Como a lógica complexa das *caches* chegam a ocupar mais de 50% da área *on-chip* e quando construídas como SRAMs consomem entre 25% e 45% do consumo total de energia do *chip* (Panda et al., 1999), esses altos percentuais atrelados a elas, as tornam um alvo preferencial para busca de alternativas.

Quando a gestão da memória *on-chip* é deixada para o software, geralmente utiliza-se o termo memória *scratchpad* para denotar essa abordagem. No nível da *cache*, cujos dados são acessíveis de um a alguns poucos ciclos da CPU, adiciona-se uma camada controlada explicitamente pela aplicação. Na abordagem sem a *scratchpad*, um *store* ou um *load* efetuados pelo processador são semanticamente análogos a armazenar e a carregar os dados como se eles estivessem na memória principal. Quando há a *scratchpad* este armazenamento ocorre diretamente em um nível análogo ao da *cache*, **não** sendo portanto, equivalente a armazenar tais dados na memória *off-chip*. Como a gestão da memória *on-chip* é realizada por *software*, parte ou toda a lógica complexa da *cache* pode ser eliminada para economizar área, diminuir o consumo de energia, melhorar o tempo de execução e obter garantias de relacionadas a eficiência. Por isso, a memória *scratchpad* é uma boa solução para atender todas ou parte das demandas dos ambientes embarcados. As vantagens da memória *scratchpad* são: a redução no consumo de energia, redução não área *on-chip* ocupada, a possibilidade de se obter garantias de tempo-real, melhorias na eficiência da execução, gestão mais eficiente no acesso a dados para aplicações paralelas. As desvantagens da memória *scratchpad* incluem: o aumento na complexidade das aplicações, dificuldades relacionadas a troca de contexto e ao aumento na quantidade de dados a serem salvos quando ela ocorrer e a retrocompatibilidade entre diferentes versões de uma mesma arquitetura que pode ser comprometida ou perdida de acordo com a abordagem utilizada. As **??** e **??** abordam, respectivamente, cada um desses itens.

As Seções 2.2.1 e 2.2.2 apresentam uma revisão dos principais trabalhos de pesquisa relacionados a gestão da memória efetuada por *software* e a Seção 2.2.3 apresenta os trabalhos sobre a memória *scratchpad* aplicada para ambientes de propósito geral.

2.2.1 Hierarquia Gerenciada por *Software*

Avanços da indústria de semicondutores estão direcionando o desenvolvimento dos computadores para componentes cada vez mais miniaturizados e com diversos recursos integrados ao mesmo *chip*. Isto permite obter máquinas com desempenho crescente, mas por outro lado torna-se necessário administrar um número milionário e até bilionário de transitores gerando calor. Neste contexto, agrava-se ainda mais o crescente custo

relativo, do ponto de vista do desempenho e do consumo de energia, o acesso a componentes externos ao *chip*, como a memória principal, por exemplo. Williams et al. (2007) compararam um *benchmark* funcionando no processador STI Cell com algumas alternativas, como o AMD Opteron e o Cray X1E, discutindo a complexidade de usar o paralelismo e de mapear algumas aplicações científicas na hierarquia de memória controlada por software do STI Cell. O processador contém um núcleo PowerPc para controle e oito núcleos simples chamados *Synergistic Processing Elements* (SPE), cada um com uma memória local e um controlador de fluxo para elas. Os *loads* e *stores* efetuados por cada SPE acessam apenas sua memória local e dependem de uma unidade DMA para mover os dados de e para a memória principal. Para os autores, a hierarquia de memória controlada por software complica bastante o modelo de programação e por isso com o objetivo de simplificar a abordagem proposta, eles consideraram apenas os oito SPEs, ignorando o núcleo PowerPC e utilizaram uma abordagem de paralelismo de dados, onde cada núcleo executa as mesmas operações sobre dados diferentes. Eles argumentam que a gestão de memória controlada por software oferece previsibilidade a eficiência das aplicações usando um modelo analítico simples. Apesar das dificuldades de programação impostas pela arquitetura, a alta largura de banda da memória e gestão da DMA por software são grandes vantagens para aplicações com uso intensivo da memória. Usando abstrações de alto nível é possível ajustar os acessos ao nível mais rápido da hierarquia para um padrão específico de acesso, o que permite ao desenvolvedor alcançar boas melhorias no desempenho. Os autores sugerem que os bons resultados obtidos, nas comparações efetuadas observaram-se acelerações de 2 até 150 vezes no desempenho, são fortes evidências de que a arquitetura de memória controlada por software do processador Cell é melhor para explorar a largura de banda da memória do que as abordagens tradicionais baseadas em *caches*.

Williams et al. (2008) estudam outro exemplo em que uma arquitetura controlada por *software* pode ser útil para obter melhorias significativas no desempenho e na diminuição do consumo de energia. No artigo os autores argumentam que a mudança ocorrida na computação de alto desempenho, de processadores com alta frequência de *clock* para processadores multicore, propiciam a obtenção de um tempo de execução menor, eficiência energética e confiabilidade. A partir dessa premissa, eles apresentam um estudo focado na aplicação LBMHD[1], usando diferentes arquiteturas para executá-la. Utilizaram-se o AMD Opteron X2, Intel Clovertown, STI Cell QS20 Blade, Intel Itanium 2 e o Sun Niagara 2, pesquisando para cada arquitetura quais eram seus li-

[1] um algoritmo em mesoescala para simular a turbulência isotrópica homogênea em magneto-hidrodinâmicos dissapativos – do original *mesoscale algorithm for simulating homogeneous isotropic turbulence in dissipative magnetohydrodynamics*

mitadores de desempenho e quais estratégias de otimizações de código poderiam ser utilizadas para identificá-los. Os autores argumentam que o desempenho é limitado pela: (1) ausência de recursos no mapeamento de páginas virtuais, (2) largura de banda insuficiente para as *caches*, (3) alta latência da memória, e, (4) unidades de alocação ineficientes. Para tentar extrair mais desempenho das máquinas utilizadas, uma das técnicas empregadas foi o *software prefetching* usando desdobramento de *loops*. A estratégia foi criar uma espécie de *double buffer* carregando todos os dados que seriam necessários na próxima iteração. O processador Cell exigiu uma versão dedicada do código, utilizando um esquema similar de *double buffering*, mas sem *prefetch*, usando invés disso o DMA controlado por software e *stores* locais. Mesmo sendo considerado pobre na computação aritmética de precisão dupla (*double-precision arithmetic*), os resultados obtidos pelo Cell foram bastante expressivos, alcançando um consumo energético mais eficiente e uma aceleração de até 7 vezes. Para os autores, apesar de um *hardware* moderno e bastante sofisticado, a gestão manual entre a memória *on-chip* (*local store*) e a memória principal do Cell é a única diferença entre as arquiteturas avaliadas que justifica tamanha diferença. Por outro lado, o *trade-off* relacionado a essa melhoria está atrelado a um um alto preço: um ambiente de programação substancialmente mais complexo, com mais compromissos para gerenciar, o que resulta em um grande acréscimo no esforço de desenvolvimento.

2.2.2 Memória *Scratchpad* para Ambientes Embarcados

Motivada pelos processadores embarcados e por sua necessidade de economizar energia, a memória *scratchpad* surgiu como uma importante inovação e foi amplamente estudada como uma alternativa para a hierarquia de memória tradicional (Panda et al., 1997) (Panda et al., 1999) (Francesco et al., 2004) (Kandemir et al., 2001) (Angiolini et al., 2003).

Panda et al. (1997) propõem a utilização da memória *scratchpad* para diminuir o número de conflitos da *cache*. Os desenvolvedores deveriam cuidadosamente propor uma partição atribuindo os dados para a *scratchpad*. O texto procura se contrapor ao projeto das máquinas que possuem apenas a *cache*, discutindo um ambiente misto com a memória integrada ao *chip* do processador gerenciada pelo *software* e pelo *hardware*. Os autores argumentam que otimizações para alocação apenas de escalares não seriam gerais o suficiente para todos os casos e otimizações sobre vetores não seriam sempre aplicáveis, pois alguns usos mais complexos de vetores tornariam as otimizações estáticas inócuas. Isto posto, o trabalho de Panda et al. (1997) apresenta um algoritmo que se beneficia deste ambiente misto obtendo mais desempenho que as abordagens

tradicionais.

Por não necessitar da lógica complexa presente na *cache* gerenciada por *hardware*, a memória *scratchpad* consegue colaborar com a economia de energia dos ambientes embarcados (Panda et al., 1999) (Francesco et al., 2004). Além disso, a crescente disparidade da velocidade do processador e da memória tornam o subsistema de memória um limitador de desempenho extremamente relevante. Quando consideram-se aplicações que processam grandes volumes de dados, as restrições adicionais no consumo de energia e de custo, tornam o gargalo da memória ainda mais crítico. Baseando-se nestes argumentos, o trabalho de Panda et al. (1999) trata de aspectos relativos a memória *scratchpad* para processadores embarcados. No texto são discutidas questões relativas ao projeto, síntese e otimizações a serem realizadas no código das aplicações para arquiteturas desse tipo. A memória *scratchpad* é apresentada como uma organização alternativa para a memória com o objetivo de melhorar o desempenho da *cache*. Focados nas organizações de propósito específico, em seu trabalho, os autores supõem que apenas um conjunto pré-determinado de aplicações é executado no processador dotado da memória *scratchpad*. Com isso, está disponível para estas aplicações um tempo maior de compilação, o que permitiria a execução de um grande número de otimizações sobre o código fonte. Mais ainda, conhecendo a priori as aplicações que serão executadas pelo processador - é possível dimensionar e propor uma organização da *cache* e da *scratchpad* pensando nelas. Problemas relativos ao código legado do conjunto de instruções e a troca de contexto, por exemplo, podem não existir ou se for o caso, serem tratados partindo-se da prerrogativa de que o conjunto de aplicações é fixo. Assim, não seria obrigatoriamente necessário desocupar a *scratchpad*, por exemplo na troca de contexto. Conforme sugerido na discussão ela poderia ser dividida entre as várias aplicações e os seus dados nunca serem submetidos a quaisquer políticas de substituição. A proposta dos autores é que a organização e o particionamento dos dados entre a *cache* e a *scratchpad* seja estática.

2.2.3 Memória *Scratchpad* para Máquinas de Propósito Geral

Cook et al. (2009) propõem uma espécie de memória *scratchpad* para processadores de propósito geral. No texto, o recurso é nomeado como *local stores*, pois ao efetuar um *store* na *scratchpad*, os dados são armazenados na memória local, *on-chip*, o que não é semanticamente análogo a armazenar esses dados na memória principal. Isso se contrapõe a um *store* na *cache*, porque neste caso a arquitetura da hierarquia de memória abstrai a existência de diferentes níveis, ao armazenar um dado na *cache*, sob o ponto de vista do processador, é análogo a armazená-lo na memória principal. Baseando-se

também, no fato de que a abordagem padrão baseada na *cache* não requer qualquer esforço por parte do programador e concilia diferentes interesses do projeto do processador, como custo, velocidade e capacidade do subsistema de memória, o trabalho de Cook et al. (2009) sugere que esta é uma boa solução para a maioria dos casos e que por isso teria obtido êxito como a solução padrão empregada na maioria das arquiteturas. Por outro lado, aplicações específicas poderiam se beneficiar de maneira substancial quando o processador provê a possibilidade dos *local stores*, pois o acesso alinhado das *caches*, que trabalham transferindo blocos, implicam no desperdício de espaço quando um dado pouco acessado é utilizado ou ainda quando há uma estrutura de dados que ocupa pouco espaço de um bloco. Além disso, embora técnicas de *prefetching* possam beneficiar muitas aplicações antevendo a necessidade de um dado antes que sua requisição ocorra, para algumas aplicações esses mecanismos geram previsões imprecisas que podem reduzir o desempenho e aumentar o consumo de energia. Um mapeamento pobre entre o padrão de acesso da aplicação e as heurísticas que tentam prevê-los também podem resultar em grande aumento no tráfego de dados de e para a memória *off-chip*, implicando em grande perda desempenho. No trabalho são abordados desafios relacionados a essa possibilidade, incluindo o aumento do tamanho dos processos dos usuários que precisariam ser salvos e recuperados pelo sistema operacional todas as vezes que uma troca de contexto acontece e uma solução para uma configuração mista em que a *cache* e *scratchpad* coexistam. Os autores argumentam que de modo geral a *cache* pode apresentar resultados melhores do que uma solução mista, já que apenas parte da aplicação poderia se beneficiar da memória *scratchpad*.

Moreira et al. (2010) realizam um estudo breve sobre memória *scratchpad* para processadores de propósito geral. No trabalho uma análise é realizada comparando a *cache* e a memória *scratchpad* que é tratada como uma parte disjunta da memória *on-chip*.

2.2.4 Vantagens da Memória *Scratchpad*

2.2.4.1 Redução no Consumo de Energia e na Área Ocupada

Se parte da lógica complexa atrelada a *cache* puder ser cortada, é trivial que haja uma redução na área por ela ocupada e que o seu consumo de energia seja menor. Por outro lado, se essa inteligência é removida da *cache*, ela precisa ser absorvida por outra parte do projeto, seja no *hardware* ou no *software*. Uma alteração desse tipo só é justificável se algum aspecto relevante do sistema melhorar. Portanto, ao propor uma memória *scratchpad*, precisam ser considerados, para avaliar o resultado obtido, as instruções adicionais atreladas a gestão por *software*, o consumo de energia e o tempo adicional do

processador. As tarefas antes executadas pelo *hardware* da memória *on-chip*, precisam agora ser realizadas pelas instruções executadas pela máquina. Considerando esses fatos, (Banakar et al., 2002) relatam uma redução da área ocupada de 34% e uma redução média no consumo de energia de 40% utilizando a gestão da memória por *software*.

2.2.4.2 Garantias de Tempo-Real

Quando a gerência da hierarquia memória é realizada por *hardware*, pode ser extremamente difícil prever qual será o desempenho de uma dada aplicação naquele ambiente. Isso ocorre porque pode não ser viável deduzir qual será o comportamento da hierarquia de memória em um dado instante de tempo qualquer. Além de questões inerentes a implementação do *hardware*, que podem estar além do domínio do desenvolvedor, há outras questões relativas a execução multi-tarefa do processador e as preempções relacionadas a elas. Seria necessário ter conhecimento além do projeto e do comportamento do *hardware*, sobre o estado em que todas as outras aplicações executando na máquina se encontram. O que obviamente é inviável na prática. Por outro lado, se o desenvolvedor é capaz de gerenciar a memória, por mais que a aplicação ainda esteja sujeita as políticas de preempção da máquina, o seu desempenho pode ser calculado para o pior caso. Geralmente a esse fato relacionam-se as garantias exigidas pelas aplicações de tempo-real. Essa classe de sistemas precisa que parâmetros de tempo de resposta na execução das aplicações possam ser previstos e controlados. As *Transparent Scratchpad Memories* podem ser utilizadas para ajudar a alcançar esses requisitos.

2.2.4.3 Melhorias na Eficiência da Execução

Algumas aplicações expressam uma localidade difícil de ser prevista pelas abordagens tradicionais que utilizam heurísticas temporais e espaciais. Por conhecer o domínio do problema tratado pelo algoritmo, um desenvolvedor buscando mais eficiência pode utilizar-se desse conhecimento para produzir um código com uma gestão mais adequada da memória. O tráfego dos dados entre os níveis da hierarquia pode ser planejado de acordo com a aplicação, adiantando a suas demandas. Se isso for feito adequadamente, melhorias na eficiência podem ser observadas. Como memória é algumas ordens de grandeza mais lenta que o processador, se o número de falhas da *cache* diminuir, mesmo que um número maior de instruções para a gestão da memória por *software* seja exigido, ganhos expressivos no tempo de execução das aplicações podem ser observados (Panda et al., 1999).

2.2.4.4 Acessos Paralelos

Soluções paralelas precisam compartilhar pelo menos um nível da hierarquia de memória. Isso é inerente a essa abordagem, pois o problema precisa ser dividido em partes menores a serem paralelizadas, que eventualmente precisam ser consolidadas em uma única solução. Se dois ou mais desses fluxos paralelos de execução concorrem por espaços da *cache*, o desempenho da aplicação pode ser bastante penalizado. Por exemplo, o uso de dados diferentes mapeados para a mesma linha da *cache*, podem provocar falhas sucessivas para vários dos acessos realizados. Alguma engenharia pode ser utilizada para evitar que isso ocorra, mas eventualmente, de acordo com o número de execuções paralelas, isso pode ser bastante complexo. A gestão por *software* da memória pode ser útil também nesses casos. Se cada execução paralela faz a gerência do seu espaço e não há sobreposição, conflitos como os relatados não ocorrerão. Para os dois casos, em algum momento uma sincronização será necessária para produzir a solução final. Williams et al. (2007) reportam uma aceleração de 2,7 até 7 vezes em simulações científicas utilizando uma arquitetura paralela com gestão da memória por *software*.

2.2.5 Desvantagens da Memória *Scratchpad*

Se na gestão da memória por *software* os recursos providos pelo *hardware* são transferidos para o *software*, espera-se que parte da complexidade antes presente na *cache* seja inserida no código da aplicação. Este fato configura uma das principais desvantagens das *scratchpad memories*. Além disso, complicações adicionais podem aparecer quando consideram-se diferentes versões da mesma arquitetura que podem ter requisitos de compatibilidade. Este é um recurso bastante comum em processadores comerciais de propósito geral. Geralmente eles possuem várias versões que se distinguem pela quantidade e qualidade do *hardware* disponibilizado. A quantidade de memória SRAM disponível próxima ao processador pode determinar o quanto de memória *scratchpad* e de *cache* são ofertadas para as aplicações. Em um contexto geral isso pode diminuir a compatibilidade das aplicações dentro da mesma arquitetura. Outro fator a ser considerado são as trocas de contexto. A memória *scratchpad* representa dados adicionais a serem salvos como parte do estado dos processos, podendo as encarecer de modo inviável.

2.2.6 Análise Crítica

Os trabalhos de Panda et al. tratam apenas de ambientes embarcados de onde se originam as ideias da memória *scratchpad*. Mesmo assim, vários aspectos relevantes para um projeto de propósito geral são citados nestes trabalhos, como questões relativas ao processo de compilação e questões arquiteturais do *chip*.

A maioria dos estudos sobre a memória *scratchpad* são focados em arquiteturas embarcadas, onde as primeiras discussões foram realizadas, como em (Panda et al., 1997) e (Panda et al., 1999). Muitos aspectos importantes do projeto de arquiteturas de propósito geral são tratadas nestes trabalhos, como o processo de compilação e questões sobre a arquitetura do *chip*. Cook et al. (2009) trata de diferentes organizações para a arquitetura do subsistema de memória, tais como a solução padrão da *cache*, apenas *local stores* disponíveis e uma abordagem mista entre ambos. Seu principal objetivo é prover a gestão da memória *on-chip* por *software* com o mínimo de alterações arquiteturais, mantendo o código legado compatível. Eles sugerem uma espécie de memória *scratchpad* cuja gestão é transparente, onde nenhuma instrução adicional é necessária. O sistema operacional é o responsável por administrar o tamanho da memória *scratchpad* de modo que as operações realizadas nos endereços marcados como tal não são sujeitos as políticas de substituição da *cache*.

Esta dissertação de mestrado diferencia-se das propostas anteriores (Cook et al., 2009) por tentar manter a memória *scratchpad* sobre o controle da aplicação, utilizando a infraestrutura já existente para a *cache* e trabalhando com a mínima ou a menor quantidade de interferência externa possível.

O trabalho proposto nesta dissertação usa as ideias apresentadas por Panda et al. (1999) e Cook et al. (2009) como base para o desenvolvimento de um processador de propósito geral que utiliza-se do processo de compilação e de um conjunto de instruções específico, para tentar gerar código mais eficiente e que potencialmente beneficie toda a aplicação quando comparado às abordagens tradicionais e aquela proposta por Cook et al. (2009) que beneficia apenas alguns trechos do código.

2.3 Compilação para a Memória Scratchpad

Hiser & Davidson (2004) apresentam uma solução para o uso das linguagens de alto nível que permite atender as restrições de tempo relativas ao mercado de processadores embarcados. Eles argumentam que a grande quantidade de diferentes configurações de *hardware* para a hierarquia de memória, projetados para atender objetivos de desempenho e economia de energia, podem exigir o uso de código *assembly* dificultando

o desenvolvimento das aplicações de modo relevante. Este tipo de projeto pode ser utilizado para administrar os dados, permitindo economia de energia e acelerações na execução. Por exemplo, algumas configurações possíveis para um sistema poderiam incluir uma memória *on-chip*: (1) somente com a memória *scratchpad*, (2) apenas a *cache*, ou (3) com a *cache* e a memória *scratchpad* ao mesmo tempo.

A hierarquia de memória pode ser particionada como um conjunto de diferentes partes disponíveis para o compilador ou no nível do código *assembly*. Para os autores estes diferentes tipos de ambiente são um problema para os projetistas de compiladores de linguagens de alto-nível. O compilador precisa atribuir trechos de um programa para estas partes heterogêneas de memória e se isso não for feito de forma eficiente o desempenho obtido pode ser drasticamente afetado. Neste contexto, os autores apresentam o algoritmo EMBARC que tenta obter um padrão de acesso realista, utilizando técnicas de *profiling* para atribuir variáveis em tempo de compilação, procurando otimizar o desempenho e o consumo de energia da aplicação. Uma entrada de teste é utilizada para executar o *profile* e o algoritmo mantém uma lista de intervalos de vida das variáveis em conjunto com uma contagem de referências. Mesmo nos casos em que não é possível utilizar a técnica devido as rotinas de biblioteca e as bibliotecas dinâmicas, os endereços relacionados a cada variável são monitorados para cada execução de um *load* ou de um *store*, com um tempo relevante, segundo os autores, alcançando "menos de 25 minutos por *benchmark* para concluir". O algoritmo recebe um arquivo com a especificação da arquitetura alvo e tenta estimar o custo de alocar diferentes dados às diferentes partições de *hardware* disponíveis.

Nguyen et al. (2005) propôs um esquema de alocação de memória onde o tamanho da memória *scratchpad* pode ser desconhecido em tempo de compilação, o que contrasta com as outras abordagem que geralmente requerem um tamanho pré-determinado. Essa exigência estática relativa ao tamanho da *scratchpad* produz um código gerado que não é portável para outras versões da mesma arquitetura que possuem projetos diferentes para a memória *on-chip*. A proposta do trabalho é criar um instalador de *software* personalizado, responsável por resolver a alocação da *scratchpad* antes do programa executar pela primeira vez, usando um pré-processador para modificar o executável. Para os autores, os métodos de alocação da memória gerida por *software* são categorizados em dois tipos: estáticos e dinâmicos. Nos estáticos os dados armazenados na memória *scratchpad* não têm sua organização alterada durante a execução. Nos dinâmicos os dados armazenados podem mudar dinamicamente durante a execução. No trabalho argumenta-se que não importa que tipo de técnica é aplicada, o tamanho fixo da memória *scratchpad* é uma fraqueza para ambos. Para ambientes embarcados o código executado é tipicamente fixado no momento do pro-

jeto do *hardware*, não mudando depois. Em algumas circunstâncias, por outro lado, o *hardware* pode evoluir e obrigar o *software* a coexistir com diferentes tamanhos da memória *scratchpad*. O método proposto no trabalho seria aplicável para os casos em que o tamanho da *scratchpad* não é conhecido, não sendo possível decidir quais dados deveriam ser armazenados nela. Usando informações obtidas com técnicas de *profiling* a partir de múltiplas execuções da aplicação com entradas diferentes e expressivas, algumas variáveis são atribuídas para endereços desconhecidos e armazenados em listas de endereços pendentes. Cada variável recebe um indicador de prioridade que também é armazenado no executável e utilizado com um algoritmo guloso para decidir quais dados devem ser alocados à memória *scratchpad*. Finalmente, para reduzir o custo adicional da primeira execução da aplicação, o algoritmo proposto executa uma pré-computação de todos os possíveis tamanhos da memória *scratchpad*.

Francesco et al. (2004) apresentam um conjunto de mudanças em linguagens de alto nível para permitir a gerência da memória *scratchpad* em tempo de execução. Eles sugerem uma mudança no *hardware* introduzindo uma DMA para fazer as transferências entre a memória principal e a *scratchpad* mais eficientes. Em outros trabalhos os dados a serem copiados da e para a *scratchpad* utilizam um barramento local entre a *cache* e a memória *scratchpad*. Nestes casos, com a intervenção do processador, os dados precisam ser transferidos da memória principal para a *cache* e então da *cache* para a *scratchpad*. A DMA proposta no trabalho permite transferências entre a memória principal e a *scratchpad* sem ou com pouca interferência do processador. Um administrador do espaço livre disponível na *scratchpad* é utilizado para implementar a abstração de alto nível proposta. Para utilizar este espaço, o desenvolvedor inicialmente precisa escolher quais dados deveriam ser armazenados na *scratchpad*, realizando algum particionamento quando os dados não couberem completamente. Para os autores a *scratchpad* é normalmente mal utilizada pelas aplicações porque a maioria das estratégias de alocação são estáticas. Vários algoritmos têm um comportamento intrinsecamente dinâmico, por isso para eles é muito difícil predizer em tempo de projeto quais e quantos dados deveriam ser armazenados na *scratchpad*. Adicionalmente, a DMA e a *scratchpad* são programadas usando a linguagem *assembly* e na ausência de alternativas de alto nível é muito difícil utilizá-las para alocações dinâmicas, o que justifica o *framework* proposto na pesquisa. No trabalho, a fragmentação é tratada e a eficiência energética é avaliada para diferentes abordagens, reportando uma redução no consumo de até 40% quando utilizando *scratchpad* em comparação com a *cache*. Eles reportam também, uma melhoria no tempo de execução de 40% no uso da *scratchpad* com a DMA e de 20% sem ela.

Para Udayakumaran & Barua (2003) há dois tipos de alocação para a memória

scratchpad: o primeiro tipo tenta emular o comportamento da *cache* no software e o segundo separa as variáveis em dois grupos. Simular a *cache* não é uma boa alternativa porque isso implica em custo adicional em tempo de execução, incremento no tamanho do código, aumento no consumo de energia e ausência de garantias de tempo real. Como outra opção, algumas variáveis são separadas em um grupo a ser alocado para a SRAM (memória *on-chip* rápida e cara) e as outras alocadas para a DRAM (memória *off-chip* lenta e barata – memória principal). Em oposição a alocação estática que para os autores negligencia o comportamento dinâmico das aplicações, na pesquisa eles apresentam um método dinâmico para alocação de variáveis globais e de pilha. Os dados acessados frequentemente são atribuídos a SRAM via transferências com a memória principal realizadas em pontos da aplicação que são executados com pouca frequência. A *cache* tenta manter durante sua execução os dados que são utilizados com mais frequência e seu conteúdo muda durante a execução refletindo o comportamento dinâmico do sistema. Para ser considerada uma alternativa eficiente, a *scratchpad* deveria possuir também este comportamento dinâmico, o que justifica o trabalho, já que esta é uma desvantagem presente em praticamente todos os trabalhos anteriores. São consideradas as mudanças dos requisitos do programa durante a execução, sem o uso de sinais de controle, baixo custo adicional e tempos de acesso predizíveis. Como na abordagem da *cache*, os dados podem ser movimentados para a memória principal, mas no caso da memória *scratchpad* com administração do compilador. Usando dados de *profiling* coletados anteriormente, o compilador insere códigos para copiar os dados que ele supõe que serão utilizados em breve, para transferir os dados entre a DRAM e SRAM.

A solução proposta por Udayakumaran & Barua (2003) é baseada na criação de *intervalos de tempo* em certos pontos do código para construir uma estrutura chamada *Grafo de Relacionamento de Dados do Programa* [2]. O código é otimizado usando um modelo de custo para avaliar os benefícios de cada transferência usando um algoritmo guloso para maximizar o benefício geral. Além disso, otimizações executadas em tempo de compilação são aplicadas tentando reduzir o custo das transferências. A abordagem procura fazer menos ou no pior caso o mesmo número de transferências que são realizadas pela *cache*. Diferente do que a *cache* faz, simplesmente ignorando este fato, dados acessados somente uma vez não são trazidos para a SRAM e dados mortos não são enviados para a memória principal. O algoritmo guloso age inicialmente dividindo o programa em regiões em que as decisões de alocação não mudam, elas são tomadas no início da região e são mantidas até seu fim. Depois disso, um intervalo de tempo é

[2] do original *Data Program Relationship Graph*

atribuído a cada região, de modo que este processo cria uma ordem total entre tais intervalos e a execução de cada região é alcançada na ordem especificada pelos intervalos de tempo. O algoritmo avalia o custo de trazer uma variável da memória principal para a memória *scratchpad* considerando o custo de derramar outras variáveis da *scratchpad* para a memória principal. A cada ponto do programa mais de um intervalo de tempo pode ser atribuído, porque um mesmo ponto pode ser alcançável a partir de diferentes origens. Os autores sugerem como pontos promissores criar as regiões no início de cada função e antes de cada *loop*. Algumas dificuldades são discutidas para a abordagem proposta, incluindo a fragmentação da *scratchpad*, como o algoritmo deveria agir quando o espaço solicitado não está disponível na *scratchpad*, pontos com diferentes intervalos de tempo, caminhos condicionais e funções recursivas. O método proposto ignora o *heap*, esta limitação deve-se ao seu comportamento dinâmico, dependente dos dados e possivelmente desconhecido em tempo de execução. No processo de alocação dois cenários são considerados: o *hardware* dotado de DMA e sem ela. *Loops* e funções são utilizados para alocar a *scratchpad* evitando o crescimento do código e na presença da DMA este processo apresenta melhorias no desempenho. Comparando o resultado do trabalho com alocação estática, são reportados acelerações na execução de 11% a 38% e uma redução média de 61% no acessos a memória principal.

2.4 Conclusão

Os processadores tradicionais de propósito geral possuem um requisito quase pétreo: a retrocompatibilidade. As inovações e mudanças na tecnologia necessariamente precisam respeitar o código legado. Todo o patrimônio de *software* existente precisa continuar funcionando a cada nova versão da máquina. Além da base instalada, as facilidades no desenvolvimento, restrições no tempo e no custo são determinantes para a escolha da plataforma utilizada pelos desenvolvedores. A *cache* é uma solução suficientemente boa para a maioria das aplicações e alterações no seu tamanho, no tempo de acesso e nos algoritmos que a gerenciam são transparentes. A responsabilidade pela complexa gestão da memória é tirada dos ombros dos desenvolvedores e atribuída ao hardware.

Por outro lado, apesar de favorecer a simplicidade, a solução padrão ofertada pela *cache* não é a melhor opção para todos os casos. Para algumas aplicações requisitos como os discutidos na Seção 2.2 são mais importantes do que os fornecidos pela gestão de memória por *hardware*. Além disso, os limites da tecnologia CMOS empregada nas máquinas atuais, relativos a dissipação de calor e a frequência permitida para os

processadores, levaram a indústria a paralelizar máquinas que antes eram *single-core* (Patterson & Hennessy, 2012), como no caso dos processadores da família Intel Nehalem (Intel Corporation, 2013), por exemplo, que podem possuir até quatro cores. Apesar de trabalhar em um nível muito menor de paralelismo do que o disponibilizado pelas GPUs, essas máquinas também possuem um viés paralelo. Seria interessante que essas máquinas disponibilizassem para as aplicações paralelas recursos que as permitissem utilizar melhor essa capacidade. Como já discutido, a gestão da memória por *software* é capaz de ofertar essa possibilidade, mas idealmente o código legado não deveria ser afetado com a sua inclusão. Em linhas gerais a proposta deste trabalho, Um Modelo Transparente de Memória *Scratchpad* para Arquiteturas de Propósito Geral, é introduzir a gerência manual da memória para processadores de propósito geral, respeitando suas restrições e oferecendo novas possibilidades para os desenvolvedores sem afetar os não interessados nela.

Capítulo 3

Um Modelo Transparente de Memória *Scratchpad* para Arquiteturas de Propósito Geral

O objetivo deste trabalho é propor um ambiente misto de propósito geral em que a *cache* e a memória *scratchpad* estão disponíveis. Para fazê-lo é necessário adotar uma abordagem em que consideram-se o *hardware* e o *software*. Este capítulo apresenta a implementação da memória *scratchpad* de forma transparente (vide Seção 3.1.1) considerando essas duas vertentes. A Seção 3.1 trata do projeto do *hardware* em alto nível, das decisões tomadas durante seu desenvolvimento e de suas implicações para a arquitetura. Na Seção 3.2 são descritos os recursos criados para que os programadores possam utilizar a memória *scratchpad* a partir de linguagens de alto nível. A Seção 3.4.2 discute o simulador usado para verificar o comportamento do *hardware* e estudar os resultados obtidos. São enfatizadas as suas partes alteradas e como ele é utilizado para verificar a hipótese do trabalho proposto nesta dissertação. Para finalizar o capítulo, a Seção 3.5 apresenta as considerações finais em relação a implementação proposta para a memória *scratchpad*.

3.1 Hardware

Compartilhando o mesmo *chip* do processador normalmente há uma memória rápida e pequena que pode ser utilizada com diferentes configurações. Nas arquiteturas de propósito geral o mais comum é que essa memória seja usada como uma *cache*, compondo o mais alto nível da hierarquia de memória gerida por *hardware*. Embora mais

complexas, organizações de *cache* com múltiplas entradas associativas por conjunto são bastante utilizadas para obter um desempenho melhor a partir dos princípios de localidade temporal e espacial. Por exemplo, a Figura 3.1 mostra uma organização associativa por conjunto com quatro entradas. Neste modelo, um bloco de dados pode ser encontrado em uma de quatro posições da linha da *cache* para onde ele é mapeado.

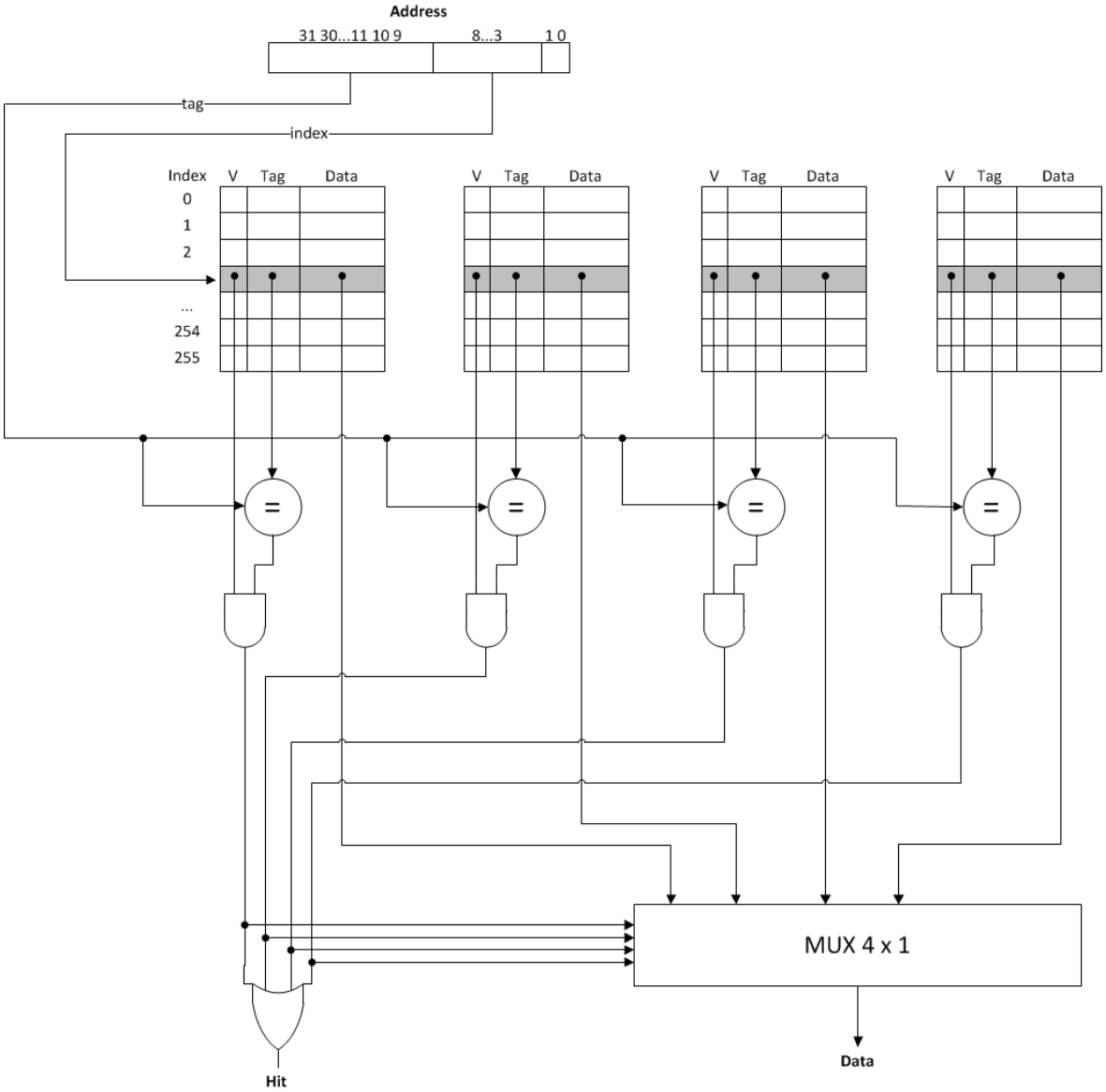

Figura 3.1: *Cache* associativa por conjunto com quatro entradas

Utilizando essa solução difundida nas arquiteturas de propósito geral é possível disponibilizar as memórias *scratchpad* de modo simplificado e transparente, adicionando um *bit* de controle S a cada uma das entradas de uma *cache* associativa por conjunto – conforme ilustrado na Figura 3.2. Para que esta estratégia funcione, o comportamento da *cache* deve ser alterado de modo que as entradas onde S está ativo

nunca sejam submetidas as suas políticas de substituição. A título de elucidação, o trecho exibido no Código 3.1 pode ser utilizado para controlar a substituição de um bloco de dados da *cache*. Inicialmente, nas linhas de 4 a 8, o algoritmo procura identificar se o bloco em avaliação possui algum dado válido, caso não tenha, ele pode ser utilizado e portanto, não há um conflito na *cache*. Se o bloco não estiver livre, nas linhas de 9 a 14, o código prossegue identificando qual o bloco usado a mais tempo e que não está assinalado por S. Neste exemplo empregou-se a política do bloco menos recentemente utilizado para promover a substituição no caso de um conflito, mas outras políticas poderiam ser empregadas de modo análogo, bastando para isso adicionar uma verificação que garanta que o bloco a ser substituído não faz parte da memória *scratchpad*. Se em paralelo a este comportamento for definido um conjunto de operações que permita a administração por *software* dos blocos assinalados por S, obtem-se como resultado uma memória *scratchpad* funcional.

Código 3.1: Código para controlar a substituição de blocos da *cache* com a memória *scratchpad*.

```
1  uint32 replace;
2  uint32 older = INT_MAX;
3  for (int i = 0; i < associativitySize; i++){
4    // If this space is still blank, use it
5    if (! cache.lines.data[i][address.index].v){
6      replace = i;
7      break;
8    }
9    // Chooses the older (can not be spm)
10   else if (data[i][address.index].lru < older
11         && ! data[i][address.index].S){
12     older = data[i][address.index].lru;
13     replace = i;
14   }
15 }
```

3.1.1 Transparência

O conceito de transparência está relacionado a provisão da memória *scratchpad* sem que isso afete o código legado, algo muito importante para as arquiteturas de propósito geral. Por não possuir chamadas para a biblioteca de *software*, as aplicações existen-

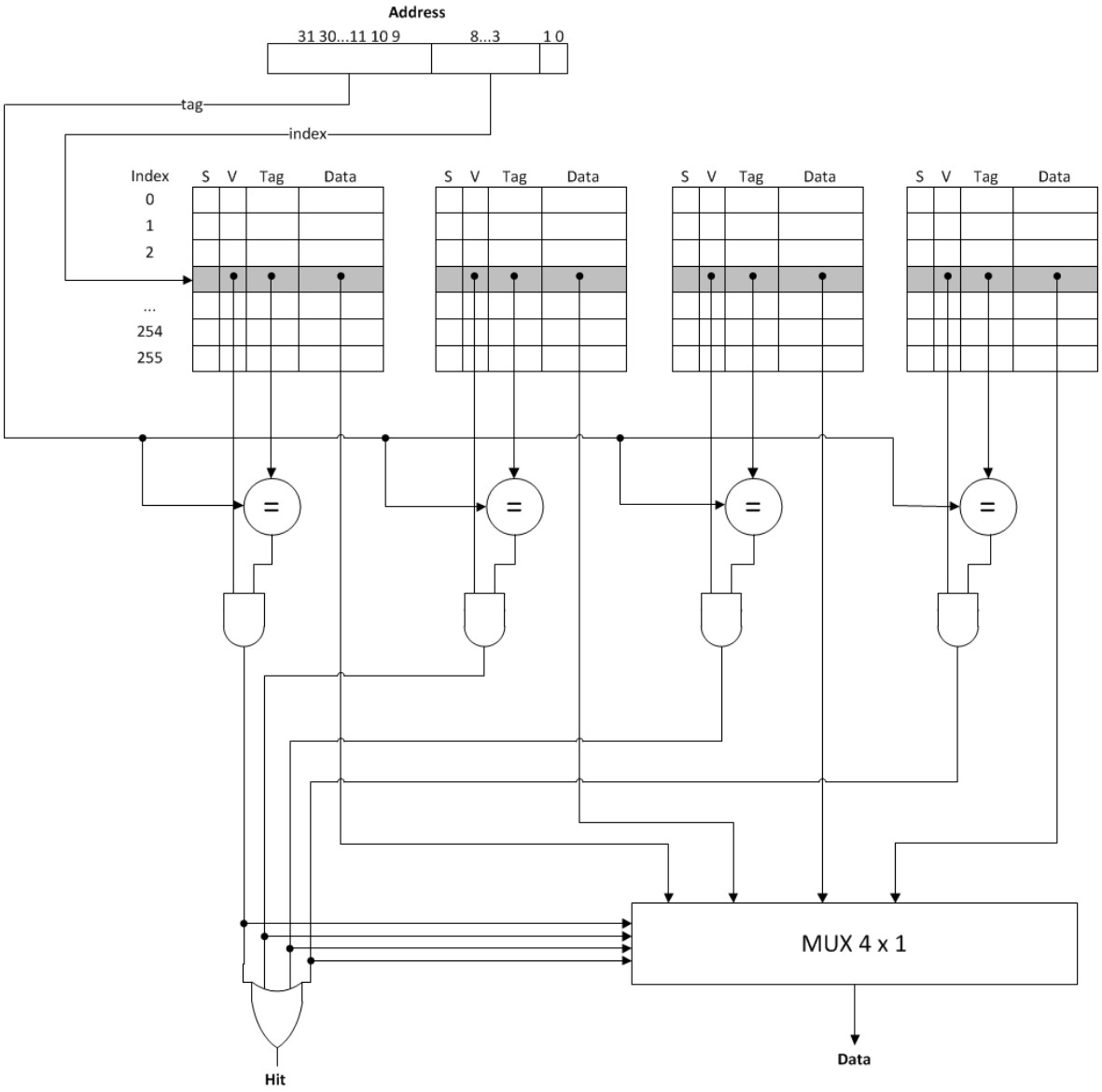

Figura 3.2: *Cache* associativa por conjunto com quatro entradas alterada para fornecer a memória *scratchpad*

tes tratam a memória *on-chip* como se ela fosse um endereço da memória principal, exatamente de acordo com a abstração fornecida pela hierarquia de memória. Mesmo quando um ou mais blocos de uma linha estão assinalados pelo *bit* S, o comportamento da aplicação é o mesmo fornecido pela abordagem tradicional, sem a *scratchpad*. Naturalmente, o número de blocos utilizados como memória *scratchpad* pode influenciar o desempenho da aplicação. Por exemplo, quando uma falha da *cache* ocorrer, se a memória *scratchpad* ocupar muito espaço, um bloco contendo dados que serão usados em breve pode ser substituído pelos dados necessários no momento, degradando o desempenho da aplicação. Ao utilizar a memória *scratchpad* o desenvolvedor deve manter

esses compromissos em mente, mas por outro lado, é imperativo que a arquitetura também imponha algumas restrições relativas a alocação da memória rápida.

Em relação a troca de contexto, dependendo da abordagem usada, nem todos os endereços podem ser alocados como memória *scratchpad*. Por exemplo, supondo um cenário em que todos os blocos de uma linha estejam alocados como memória *scratchpad*, quando uma preempção ocorrer para trocar a aplicação em execução, se os dados marcados por S forem mantidos na memória *on-chip*, a retro-compatibilidade é perdida. Ao tentar utilizar um determinado endereço, se todas as posições da linha relacionada a ele não puderem ser derramadas para o próximo nível da hierarquia, este endereço estará inacessível à aplicação. Em um cenário menos pessimista, um uso inadequado da alocação da *scratchpad* pode reduzir uma *cache* associativa por conjunto em uma *cache* com mapeamento direto. Além disso, se a maior parte dos blocos estão ocupados pela memória *scratchpad*, o desempenho das demais aplicações pode ser prejudicado de forma expressiva. Assim, caso os dados da aplicação sejam mantidos na memória *on-chip* durante a troca de contexto, o *hardware* precisa impor algum tipo de limitação para garantir viabilidade da execução e a eficiência de todas as aplicações existentes em um esquema de multiprocessamento.

Durante a troca de contexto também é possível derramar os dados de outra aplicação alocados na memória *scratchpad* para o próximo nível da hierarquia de memória. Isso impacta no desempenho e na semântica da *scratchpad*, pois a rigor os dados não mais estarão sempre disponíveis na memória mais próxima ao processador. Como consequência, enquanto a aplicação estiver executando, os dados da memória *scratchpad* nunca seriam substituídos por um conflito da *cache*. Por outro lado, o comportamento global do *hardware* pode determinar a substituição de blocos para os quais se espera um funcionamento análogo ao de uma memória *scratchpad*. Neste caso, quando um endereço atribuído a *scratchpad* for solicitado e ele não estiver disponível na memória *on-chip*, ocorre um evento que podemos chamar de falha da memória *scratchpad*. Portanto, teríamos de uma semântica diferente das propostas discutidas no Capítulo 2 para a gestão da memória por *software*, aproximando-se mais de uma abordagem intermediária. Além disso, como o endereço pode não estar imediatamente disponível no nível mais rápido da memória, uma perda de desempenho é esperada.

Na hierarquia de memória tradicional, apenas com a *cache*, embora a eficiência das aplicações em execução possa ser influenciada pela existência do multiprocessamento, esse fato é muito sutil para aplicações com a mesma prioridade. Fatores como utilização da *cache*, por exemplo, podem fazer com que diferentes aplicações influenciem no desempenho umas das outras. Se uma aplicação que faz uso ostensivo da *cache* é submetida a uma preempção e substituída por outra que também o faz, após

algum tempo de execução parte dos dados da *cache* podem ter sido substituídos. Ao retomar à execução, a aplicação inicial sofrerá com uma série de falhas. Por outro lado, se uma aplicação é substituída por outra que faz pouco uso da memória, a troca de contexto é bem menos impactante. Dependendo da abordagem adotada pelo *hardware* para implementar a memória *scratchpad* e as restrições impostas por ele para o seu uso, este problema pode ser agravado de modo relevante. Caso o número de blocos da *cache* assinalados pelo *bit* S seja grande o suficiente, como o espaço disponível para uma aplicação pode depender das demais em execução, a influência entre elas pode se tornar um fator com mais peso para a avaliação do desempenho da arquitetura. Os experimentos realizados nessa dissertação de mestrado consideraram apenas uma aplicação em execução, por esse motivo a multiprogramação e troca de contexto não ocorreram e muito embora tenham sido discutidas, não foram alvo de avaliação.

3.1.2 Memória de Dados e Instruções

Outra questão concernente à memória *scratchpad* está relacionado com a sua alocação, que pode ser estática ou dinâmica. Na estática os dados atribuídos à *scratchpad* são imutáveis até o término da execução da aplicação. Assim, neste caso o mapeamento entre o espaço disponível na *scratchpad* e as estruturas de dados a serem armazenadas nela é feito em tempo de compilação. A a alocação dinâmica permite que essa atribuição ocorra nos dois momentos, o compilador pode definir quais dados são armazenados na *scratchpad*, mas durante a execução esses dados podem mudar. Ao oferecer a memória *scratchpad* com recursos para alocação dinâmica, é necessário responder a seguinte pergunta: a alocação da memória *scratchpad* deve tornar os endereços desse espaço visíveis e administrados pelo programador? Caso os endereços estejam visíveis, as chamadas para a biblioteca de alocação precisa receber como parâmetro o endereço a ser alocado.

Do ponto de vista do compilador, essa é a abordagem mais simples, porém para o desenvolvedor além da gestão da memória por *software*, é necessário controlar os endereços utilizados, resultando em mais complexidade. É desejável que o compilador abstraia essa complexidade, permitindo chamadas para alocação dinâmica nos moldes da biblioteca *malloc* da linguagem C.

Ao discutir o modelo de alocação da memória *scratchpad*, também é importante ponderar qual o tipo de arquitetura utilizada. Na arquitetura de Von Newmann (Patterson & Hennessy, 2012) os dados armazenados na memória não têm tipo, eles são indistinguíveis uns dos outros. Dessa forma dados e instruções são tratados de maneira uniforme, o que permite que todas as informações de uma aplicação sejam armazenadas

em uma única memória, com seu tipo sendo atribuído ao uso que se faz deles. Dentre as vantagens dessa abordagem, há a uniformidade no tratamento de diferentes dados, a simplificação da arquitetura da hierarquia de memória e a possibilidade da aplicação alterar seu próprio código fonte. Outra alternativa, conhecida como arquitetura de Harvard (Francillon & Castelluccia, 2008), bastante utilizada em ambientes embarcados, possui uma memória de dados e outra memória para instruções. Nela ao carregar uma aplicação a partir dos níveis mais baixos da hierarquia, os dados da aplicação são direcionados para a memória de dados e as instruções para a memória instruções, sem uniformidade no seu tratamento. Embora as informações armazenadas na memória não possuam a definição de tipo, é válido considerar que há dois tipos básicos que podem ser atribuídos à memória: o tipo instruções e o tipo dados, sendo ambos diferenciados pelo seu armazenamento na memória de instruções ou de dados.

Para simplificar este projeto, cujo objetivo é prova de conceito e não finalidade comercial, deixou-se a cargo do programador a responsabilidade da gestão dos endereços usados. Além de agregar alguma complexidade ao desenvolvimento das aplicações de *benchmark*, essa opção teve implicações no desenvolvimento do *hardware*. Como ficou a cargo do desenvolvedor controlar quais os endereços alocados e a arquitetura adotada nesse estudo é a de Von Newmann, tornou-se necessário que o desenvolvedor soubesse quais endereços seriam usados pelo compilador durante a escrita do código, isto é, saber os endereços utilizados durante a compilação antes de compilar o programa. Porém, os endereços da memória *scratchpad* foram tratados de modo que os dados armazenados nos níveis superiores não têm correspondência com os níveis inferiores, permitindo que a alocação seja realizada sem sobrescrever as instruções do programa, mas não impedindo a colisão com eventuais alocações efetuadas pelo compilador para as variáveis e estruturas de dados, criando em algum sentido, um comportamento que remete a arquitetura de Harvard.

Suponha um cenário em que uma abordagem mista seja utilizada para tratar dados e instruções e que a arquitetura no geral seja do tipo Von Newmman, mas haja uma *cache* para dados e outra para instruções. Nesse caso um *store* efetuado para o endereço 0 da memória *scratchpad*, manteria os dados armazenados no nível *on-chip* da hierarquia de memória e como há duas *caches* distintas, uma instrução armazenada no endereço 0 não seria afetada pelo *store*. Por outro lado, uma avaliação mais cuidadosa pode ser necessária dependendo da implementação e da abordagem utilizada para a troca de contexto. Neste trabalho incluíram-se aos endereços alocados na *scratchpad* um deslocamento grande o suficiente para garantir que colisões não ocorreriam sobrescrevendo os dados atribuídos pelo compilador a *cache*. Deve-se observar que esta dificuldade é inerente ao tratamento uniforme dado aos endereços pelo processador,

pois do seu ponto de vista, um endereço na *scratchpad* ou na hierarquia tradicional são semanticamente análogos. Na codificação dos testes realizados, para os casos em que não era necessária uma grande quantidade de espaço na memória *scratchpad*, foram usados endereços com valores mais baixos, 0, 4, 8, 12, etc., supondo, conforme o descrito, que as instruções da aplicação não seriam sobrescritas.

3.1.3 Transferência de Dados

Um recurso importante que precisa ser provido pelo *hardware* é a transferência de dados entre a memória principal e a memória *scratchpad*. Quanto menor a interferência do processador melhor, pois quanto mais tempo do processamento puder ser economizado na transferência, mais eficiente será a máquina. Com esse objetivo, definiram-se as instruções discutidas na Seção 3.1.4, que permitem ao processador solicitar a movimentação de dados entre a memória *scratchpad* e a memória principal. Para tornar a solução proposta mais eficiente, embora o processador solicite tais transferências, optou-se por implementá-las sob a forma de uma DMA. Além disso, foi necessário definir qual seria a quantidade de dados a serem transferidos por essa unidade. Em um extremo, transferir apenas uma palavra, implicaria em desperdício da largura de banda do barramento da memória e em um número elevado de chamadas para a instrução de transferência. Por outro lado, um bloco muito grande representaria um tempo muito longo para efetuar a transferência e uma quantidade de dados, na maioria dos casos, acima do desejado. Assim, considerando que a memória *scratchpad* foi implementada como blocos assinalados das linhas da *cache*, foi natural a opção pela transferência de dados dimensionada pelo tamanho do bloco da *cache*. Isso simplificou o projeto do ponto de vista arquitetural, pois como o comportamento é o mesmo, as transferências efetuadas para a memória *scratchpad* puderam usar o *hardware* já existente da *cache*. Além disso, a avaliação de desempenho também é amplamente simplificada, pois o tempo de falha e de acerto da *cache* pode ser utilizado como o tempo necessário para transferir os dados dos níveis subsequentes para a memória *scratchpad*. Isso é extremamente relevante, porque uma solicitação de transferência para a memória *scratchpad* pode se originar de diferentes níveis. A transferência pode partir de níveis mais rápidos da hierarquia de memória e não apenas da memória principal. Se os dados solicitados lá estivessem disponíveis, uma requisição desse tipo poderia ser atendida, por exemplo, pela *cache* de Nível 2. Finalmente, há também alguns casos em que os dados podem precisar ser transferidos da *cache* de Nível 1 direto para a memória *scratchpad*. Embora esse último caso possa agregar eficiência à solução proposta, ele complica um pouco o *hardware*, pois um caso extra não previsto originalmente no *hardware* da *cache* precisa

ser tratado: a cópia de dados originada na memória *on-chip* com destino nela mesma.

3.1.4 Instruções

Após definir as alterações que seriam realizadas no *hardware* da hierarquia de memória, foi necessário definir algumas instruções e adicioná-las ao conjunto da arquitetura para prover a gerência da memória *on-chip* por *software*.

Decidiu-se que a movimentação de dados entre os registradores e a memória *on-chip* continuaria sendo realizada pelas instruções *load* e *store*. É importante ressaltar a presença da transparência nesse contexto: do ponto de vista do processador todos os endereços requisitados são análogos, não importando se os dados estão armazenados na memória principal ou na memória *scratchpad*. Porém em conformidade com o discutido, se o endereço fizer parte da memória *scratchpad*, uma falha nunca ocorrerá ao executar um *load* ou um *store*.

Para gerenciar a memória *scratchpad* foram definidas quatro instruções: *SPALLOC*, *SPFREE*, *SPLOAD* e *SPSTORE*. Cada uma delas é descrita nas subseções a seguir.

3.1.4.1 Instrução *SPALLOC*

A instrução *SPALLOC*, *SPALLOC RS*, é utilizada para alocar um bloco de dados na memória *scratchpad*. Ela ativa o *bit S* do bloco do endereço recebido no registrador *RS*. O Código A.4 mostra o pseudocódigo usado para implementar a instrução *SPALLOC*.

Código 3.2: Pseudocódigo para a instrução *SPALLOC*.

```
1  Address spAlloc(Address address){
2
3    uint32 replace = getReplaceBlockIndex(address);
4    Address baseAddress =
5          int(address / blockSizeBytes) * blockSizeBytes;
6
7    // Alocado como scratchpad
8    data[replace][baseAddress.index].S = true;
9    // Marcar os dados como validos para uso pelos
10   // ldr e str tradicionais
11   data[replace][baseAddress.index].v = true;
12   data[replace][baseAddress.index].tag = baseAddress.tag;
13
```

```
14    return address;
15 }
```

3.1.4.2 Instrução *SPFREE*

A instrução *SPFREE* cuja sintaxe é *SPFREE RS* é usada para liberar um bloco de dados na memória *scratchpad*. Ela desativa o *bit S* do bloco do endereço recebido no registrador *RS*. O Código 3.3 mostra o pseudocódigo utilizado para implementar a instrução *SPFREE*.

Código 3.3: Pseudocódigo para a instrução *SPFREE*.

```
1  void spFree(Address address){
2
3    uint32 blockIndex = getBlockIndex(address);
4    Address baseAddress =
5          int(address / blockSizeBytes) * blockSizeBytes;
6
7    // Liberar a scratchpad
8    data[blockIndex][baseAddress.index].S = false;
9    data[blockIndex][baseAddress.index].v = false;
10
11 }
```

3.1.4.3 Instrução *SPLOAD*

A instrução *SPLOAD*, *SPLOAD RS RD*, mostrada no trecho de Código **??** é utilizada para transferir dados dos níveis subsequentes da hierarquia de memória para a memória *scratchpad*. Ela recebe como parâmetros, o endereço de origem no registrador *RD*, o endereço de destino no registrador *RS* e transfere um bloco de dados do endereço de origem na memória principal para o endereço de destino na memória *scratchpad*. O Código 3.4 mostra o pseudocódigo utilizado para implementar a instrução *SPLOAD*.

Código 3.4: Pseudocódigo para a instrução *SPLOAD*.

```
1  void spLoad(Address memAddress, Address spAddress){
2
3    Address baseSpAddress =
4      (int)(spAddress / blockSizeBytes) * blockSizeBytes;
5
```

```
6     uint32 spmIndex = associativitySize;
7     // Obter o buffer associado a scratchpad
8     for (uint32 i = 0; i < associativitySize; i++){
9       if (data[i][baseSpAddress.index].S &&
10            baseSpAddress.tag == data[i][baseSpAddress.index].tag){
11        spmIndex = i;
12        break;
13      }
14    }
15
16
17    uint32 pos = 0;
18    // Copiar os dados
19    for (uint32 i = 0; i < blockSizeBytes; i+=4){
20
21          uint32 value = nextLevel->readMemoriaWord(memAddress + i);
22
23      *((uint32*)(
24        data[spmIndex][baseSpAddress.index].data +
25        (baseSpAddress.blockOffset + pos)*4)) = value;
26
27      pos++;
28    }
29  }
```

3.1.4.4 Instrução *SPSTORE*

A instrução *SPSTORE*, cuja sintaxe é *SPSTORE RS RD*, é utilizada para transferir dados da memória *scratchpad* para a memória principal. Ela recebe dois parâmetros, o registrador *RS* contendo o endereço de origem, o registrador *RD* contendo o endereço de destino e transfere um bloco de dados do endereço de origem na memória *scratchpad* para o endereço de destino na memória principal. O Código 3.5 mostra o pseudocódigo utilizado para implementar a instrução *SPSTORE*.

Código 3.5: Pseudocódigo para a instrução *SPSTORE*.

```
1  void spStore(Address memAddress, Address spAddress){
2
```

```
3     Address baseSpAddress =
4       (int)(spAddress / blockSizeBytes) * blockSizeBytes;
5
6     uint32 spmIndex = associativitySize;
7     // Obter o buffer associado a scratchpad
8     for (uint32 i = 0; i < associativitySize; i++){
9       if (data[i][adSp.index].S &&
10          baseSpAddress.tag == data[i][baseSpAddress.index].tag){
11        spmIndex = i;
12      }
13    }
14
15    uint32 pos = 0;
16
17    for (uint32 i = 0; i < blockSizeBytes; i+=4){
18      uint32 value = *(
19        (uint32*)(
20          data[spmIndex][baseSpAddress.index].data +
21          (baseSpAddress.blockOffset + pos)*4));
22      nextLevel->writeMemoriaWord(baseMemAddress + i, value);
23
24      pos++;
25    }
26  }
```

3.2 Software

Projetar o *hardware* necessário para disponibilizar a memória *scratchpad* em arquiteturas de propósito geral é parte importante, mas não a única de uma proposta que objetiva prover a gestão da memória *on-chip* por *software*. Além do hardware em si, das instruções e das decisões de projeto correlatas, é necessário prover uma biblioteca de alto nível para a utilização e a avaliação da memória *scratchpad*. Sem ela seria extremamente complexo e provavelmente inviável compilar os testes de verificação e avaliação da arquitetura proposta. O código gerado pelo compilador precisaria ser alterado manualmente ou o código precisaria ser traduzido manualmente para a linguagem *assembly*. Além das grandes dificuldades inerentes a essa abordagem, a ava-

liação dos resultados obtidos provavelmente seria bastante injusta, pois o compilador utilizaria um alto número de otimizações que dificilmente seriam efetuadas pela compilação manual. Levando isso em consideração, criou-se uma biblioteca para a linguagem C sob a forma de quatro funções que são discutidas a seguir.

3.2.1 Função *spalloc*

A função *spalloc*, cujo cabeçalho é mostrado no Código 3.6, é utilizada para alocar um bloco de dados na memória *scratchpad*. A função recebe como parâmetro um endereço e retorna um ponteiro para o espaço alocado. Seria interessante que *spalloc* recebesse como parâmetro a quantidade de dados a serem alocados e não o endereço, ficando a cargo do compilador resolver essas questões. Para efeitos de simplificação optou-se por implementá-la com o endereço e embora isso represente um complicador do ponto de vista do desenvolvedor, não há perda de generalidade. Chamadas sucessivas podem ser utilizadas para alocar espaço contíguo na memória, o que pode resultar em um alto número de chamadas para a função e embora isso possa representar uma desvantagem, essa proposta apresenta um bom desempenho e é geral o suficiente para permitir o seu uso com dados de diferentes tamanhos e tipos.

Código 3.6: Cabeçalho da função *spAlloc*.

```
1 void* __cdecl spalloc(int address);
```

3.2.2 Função *spfree*

Para trabalhar em conjunto com a função de alocação, definiu-se a função *spfree* para liberar a memória alocada na *scratchpad* – conforme o cabeçalho mostrado no Código 3.7. A função *spfree* recebe como parâmetro um dos endereços que constituem o bloco a ser liberado, mas como no caso anterior seria interessante que ela recebesse como parâmetro um ponteiro para o dado a ser liberado e não para o endereço. Pelo mesmo motivo, a simplificação do projeto, optou-se por implementá-la com o endereço, o que não resulta em perda de generalidade.

Código 3.7: Cabeçalho da função *spFree*.

```
1 void __cdecl spfree(int address);
```

3.2.3 Função *spload*

Para efetuar a transferência de dados da memória principal para memória *scratchpad* definiu-se a função *spload* cujo cabeçalho é descrito no Código 3.8. A função *spload* recebe o endereço de destino na memória *scratchpad* e um ponteiro para os dados a serem copiados para tal local. A função copia o tamanho de um bloco de dados da *cache* e embora receba o segundo parâmetro como um ponteiro inteiro, qualquer tipo de dados pode ser passado utilizando-se *casting*.

Código 3.8: Cabeçalho da função *spload*.

```
1 void __cdecl spload(int destAddress, int* src);
```

3.2.4 Função *spstore*

A função *spstore* efetua a transferência de dados da memória *scratchpad* para memória principal. A função, cujo cabeçalho é mostrado no Código 3.9 recebe o endereço de origem na memória *scratchpad* e um ponteiro para o local de destino na memória principal. A função copia o tamanho de um bloco de dados da *cache* e embora receba o segundo parâmetro como um ponteiro do tipo inteiro, qualquer tipo de dados pode ser passado utilizando-se *casting*.

Código 3.9: Cabeçalho da função *spstore*.

```
1 void __cdecl spstore(int srcAddress, int* dest);
```

3.3 Compilação

O processo de compilação da biblioteca fornecida para uso da memória *scratchpad* foi baseado no projeto LLVM (LLVM Developer Group, 2015). As alterações realizadas no compilador foram bastante sucintas. Ao encontrar as chamadas para as funções da biblioteca, ele assume que elas serão tratadas posteriormente, deixando a sua resolução para os próximos passos da compilação. Ao invocar a etapa responsável pela tradução do *byte code* do LLVM para o *assembly* da arquitetura alvo (no caso ARM), todas as ocorrências das chamadas relacionadas à memória *scratchpad* são substituídas pelas instruções equivalentes. Por exemplo, a chamada para a função *spload* no LLVM é mapeada para o seguinte código intermediário

```
call void @spload(i32 16, i32* %src)
```

Neste caso o LLC, que é o aplicativo responsável pela tradução do *byte code* LLVM para o *assembly* ARM, foi alterado para gerar como saída a instrução *SPLOAD*, ou por exemplo, algo análogo a

```
9c:     27f000f1        spload
```

Para realização dos experimentos, as otimizações do compilador sempre permaneceram ativas em seu nível mais agressivo, mas por outro lado, elas **não** foram alteradas para considerar a existência da memória *scratchpad*. Dessa forma, conforme já discutido, todas as otimizações realizadas neste trabalho, foram definidas em uma linguagem de alto nível e realizadas manualmente. Mais detalhes e exemplos a respeito das otimizações realizadas podem ser vistos no Capítulo 4.

3.4 Estratégias de Implementação

3.4.1 Hardware

Nas etapas iniciais deste trabalho, a intenção era que o *hardware* proposto fosse implementado utilizando um FPGA (*Field Programmable Gate Array*), a ideia era que uma verificação mais precisa da proposta fosse apresentada. Para fazê-lo optou-se pela solução *XUPV5-LX110T* oferecida pela Xilinx, mas a escassez de documentação e sua superficialidade prejudicaram muito a adoção da FPGA ainda nos estágios iniciais de desenvolvimento. Devido ao volume de trabalho a ser realizado, a grande complexidade que o uso do FPGA representava para o projeto e as restrições de tempo, optou-se pela utilização de um simulador, o que também é razoável, pois o trabalho situa-se principalmente no nível do conjunto de instruções.

3.4.2 Simulador

Para o desenvolvimento dos experimentos buscou-se utilizar uma arquitetura de propósito geral, amplamente utilizada e que não fosse demasiadamente complexa para os propósitos deste trabalho. A opção por uma arquitetura RISC (Patterson & Hennessy, 2012) foi natural quando consideraram-se esses requisitos. Dentre um leque de alternativas, a arquitetura ARM (ARM Limited, 2005), mais especificamente o processador ARM 9 foi selecionado. Tomada essa decisão, buscou-se um simulador funcional que implementasse o conjunto de instruções ARM e que considerasse de forma realista o número de ciclos gastos com cada instrução. Embora ele não atenda esses dois requisitos, este trabalho de pesquisa foi baseado no simulador disponibilizado por Serrano et al. (2007) e implementado na linguagem C++. Segundo seus autores, trata-se de

uma biblioteca para simulação da arquitetura de microprocessadores ARM 9 sob a plataforma x86 com sistema operacional *Windows*. A biblioteca simula um *core* básico enquanto permite que sejam implementados os subsistemas específicos da máquina cujo comportamento deseja-se reproduzir.

Como os processadores baseados na arquitetura ARM podem ser fabricados por diferentes empresas para propósitos distintos, diferentes implementações, com diferentes organizações de memória são possíveis. Por isso, o simulador selecionado como ponto de partida não possuía qualquer codificação relacionada a hierarquia de memória. O primeiro passo do desenvolvimento foi, portanto, codificar a hierarquia de memória com dois níveis: a *cache* e a memória principal. Essa implementação foi baseada nas descrições realizadas por Patterson & Hennessy (2012). Finalizada essa etapa foi incluída ao projeto uma *cache* de Nível 2 e várias aplicações de teste foram executadas para verificar a correção da implementação realizada.

Como o ambiente provido pelo simulador não possui qualquer sistema operacional, algumas das facilidades oferecidas por eles precisaram ser disponibilizadas para a execução dos experimentos. Para isso instruções artificiais para tratar entrada e saída de dados em arquivos de texto foram criadas. Dessa forma, para realizar a entrada e a saída de uma aplicação em execução, o simulador lê e escreve os dados a partir e para um arquivo de texto específico conforme o definido nas instruções criadas para este propósito. Também foi definida uma instrução para habilitar a realização de *log*, permitindo auferir a eficiência das aplicações em execução de acordo com o número de falhas e acertos registrados pelo subsistema de memória e de acordo com o número de ciclos gastos na execução da computação útil da aplicação. Finalmente uma instrução de parada foi definida (*halt*), ao encontrá-la a execução do simulador é encerrada, o que consequentemente encerra, também, a execução da aplicação. Outra facilidade provida pelo sistema operacional que trata do carregamento das instruções da aplicação, precisou ser realizada pelo simulador. Os programas são carregados em formato binário, uma instrução por linha, a partir de um arquivo de texto para a memória principal antes que qualquer computação ou aferição seja efetuada. Após essa etapa, o simulador inicia a execução do programa a partir do endereço 0 da memória até encontrar uma instrução de parada (*halt*).

Para abstrair a *cache* e os diferentes níveis da hierarquia de memória, utilizou-se uma classe de C++. Para atender aos objetivos deste trabalho, adicionou-se àquela estrutura os dados necessários para implementar a memória *scratchpad* e quatro funções para as operações que ela precisa realizar: *spLoadMem*, *spStoreMem*, *spAlloc* e *spFree*. Respectivamente elas são utilizadas para copiar um valor da memória principal para a *scratchpad*, copiar um valor da memória *scratchpad* para a memória principal, alocar

um bloco de uma determinada linha da *cache* como *scratchpad* e liberar um bloco de uma determinada linha da *cache* como *scratchpad*. Além das alterações realizadas na hierarquia de memória, a classe que trata as instruções também foi modificada para incluir as instruções descritas na Seção 3.1.4.

Outra limitação do simulador está relacionada à sua incapacidade de realizar *prefetching*. Isso é extremamente relevante, pois pode influenciar de maneira expressiva o desempenho resultante obtido. A maioria das máquinas modernas fazem uso deste recurso para tentar antecipar a necessidade dos dados evitando que falhas ocorram. Efetivamente, ao constatar um resultado muito expressivo durante a execução dos experimentos realizados nesse trabalho de pesquisa, percebeu-se que o *prefetching* seria necessário para a avaliação e principalmente comparação entre as abordagens atribuídas ao *hardware* nos experimentos. Seguindo a descrição de Panda et al. (1999) a última alteração realizada no simulador foi a inclusão de um mecanismo de *prefetching* que mesmo não sendo extremamente complexo e com técnicas extremamente avançadas, permite uma comparação razoável entre as diferentes configurações possíveis da máquina. Os principais pontos de modificação para adequação do simulador são apresentados no Apêndice A.

3.4.3 Estruturas de Dados do Simulador

A seguir são apresentadas as principais estruturas de dados introduzidas no código fonte do Simulador, codificadas na linguagem C++.

3.4.3.1 Classe ARM9Contexto

É a estrutura que armazena informações relacionadas ao estado de um núcleo do processador – neste trabalho utilizou-se um processador com apenas um núcleo. Agrupa informações como o banco de registradores, incluindo o PC – contador de programa, e o SP – apontador da pilha, além de sinais de controle para controlar o estado da máquina e *bits* de controle para indicar *overflow*, resultado zero, indicador negativo, etc.

3.4.3.2 Classe ARM9Core

Esta classe abstrai o processador, agrupando as entidades que o compõe, como o contexto, a memória e a *cache*.

3.4.3.3 Classe ARM9RutinasARM

É a classe responsável pela simulação das instruções, fazendo as chamadas necessárias para os métodos e funções que alteram o estado da máquina. Ela é responsável também pela verificação do número de ciclos gastos por cada instrução e por garantir que eventuais paradas (*stall*) e *harzards* sejam tratados corretamente. Esta estrutura foi um dos pontos onde alterações foram realizadas para o desenvolvimento deste trabalho. Como não há sistema operacional funcionando na máquina, foi incluído o código para as instruções artificiais responsáveis pela entrada de dados, saída de dados, geração de *log*, parada (*halt*) e a chamada para as instruções específicas para a memória *scratchpad*. Mais especificamente foram incluídos os métodos

- *readInt*, *writeInt* para emulação da leitura e escrita da entrada e saída de dados;
- *openInput* e *closeInput* respectivamente para a abrir e fechar o arquivo de entrada;
- *openOutput* e *closeOutput* respectivamente para abrir e fechar o arquivo de saída;
- *toggleLog* para ativar e desativar a realização de *log*. A primeira chamada para este método ativa a realização de *log*, a chamada seguinte o desativa e assim sucessivamente. Quando ativo ele gera arquivos de texto contendo os ciclos consumidos na execução de cada instrução, o número de falhas e o número de acertos produzidos pelos diferentes níveis existentes de *cache*.

Nenhum método para instrução de parada foi adicionado, pois ao identificá-la a execução do simulador é imediatamente interrompida. Uma abordagem equivalente foi utilizada para as instruções relativas a memória *scratchpad*, pois como elas são implementadas por outras classes, ao identificar tais instruções, os métodos que as tratam são invocados.

3.4.3.4 Classe ARM9Cache

É a classe responsável por implementar a *cache*. A codificação original proposta por Serrano et al. (2007) não possuía a codificação da hierarquia de memória, pois isso varia bastante entre as implementações ARM existentes. Dessa forma, baseado nas ideias expostas por Patterson & Hennessy (2012), toda a hierarquia de memória foi codificada para este projeto de pesquisa. Os dados são armazenados em *bytes* e a classe mantém uma ligação com o próximo nível da hierarquia de memória. A abstração realizada supôs que todos os níveis da hierarquia de memória seriam descendentes desta classe, pois o funcionamento de todos eles é análogo, mudando apenas a latência. Por esse

motivo o tempo gasto pelo processo de simulação para acessar os dados armazenados na estrutura foi definido como um parâmetro configurável. Assim todos os níveis da *cache* e mesmo a memória principal foram codificados como subclasses com uma latência mais alta. Nesta classe foram codificados os métodos para tratar as instruções relacionadas a memória *scratchpad*:

- *uint32 spAlloc(uint32 address, uint32 *delay)* – realiza a alocação de um bloco de dados na memória *on-chip* como memória *scratchpad*. O bloco a ser alocado é definido pelo endereço passado como parâmetro;
- *void spFree(uint32 address)* – realiza a liberação ou remove da memória *scratchpad* o bloco de dados do endereço passado como parâmetro pertence. O método não executa qualquer operação de limpeza, assinalando apenas o *bit* de validade (V) e o *bit* da memória *scratchpad* (spm) do bloco para *false*.
- *void spLoadMem(uint32 memAddress, uint32 spAddress, uint32 *delay)* – realiza a transferência dos dados do bloco passado em *memAddress* nos níveis mais lentos da hierarquia de memória para o bloco passado em *spAddress* da memória *scratchpad*.
- *void spStoreMem(uint32 memAddress, uint32 spAddress, uint32 *delay)* – realiza a transferência dos dados do bloco passado em *spAddress* na *scratchpad* para o bloco passado em *memAddress* dos níveis mais lentos da hierarquia de memória.

3.4.3.5 Classe ARM9Memory

Trata-se de uma subclasse de *ARM9Cache*, representando a memória principal.

3.5 Conclusão

Este capítulo apresentou a implementação da memória *scratchpad* de forma transparente para arquiteturas de propósito geral. A memória *scratchpad* pode ser disponibilizada empregando uma solução extremamente simples: adicionando um *bit* de controle por bloco da *cache*. As implicações e decisões relacionadas a essa abordagem foram apresentadas e discutidas. Embora algumas simplificações tenham sido realizadas e algumas dificuldades inerentes a introdução da gestão da memória *on-chip* por *software* tenham aparecido, a implementação proposta mostrou-se viável e eficiente, cujos resultados são discutidos no Capítulo 4.

Capítulo 4

Experimentos

Este capítulo descreve os experimentos usados para mostrar a viabilidade e avaliar os resultados obtidos pela abordagem proposta. Para isso utilizou-se um *microbenchmark* com quatro aplicações: adição de matrizes, cálculo de histogramas, algoritmo *quicksort* e o algoritmo de Dijkstra. Todos os algoritmos foram rescritos utilizando a biblioteca de alto nível fornecida para empregar os recursos da memória *scratchpad*. De modo geral essa tarefa não é simples, pois para melhorar a eficiência das aplicações é necessário garantir que os dados sejam trazidos para a *scratchpad* com antecedência o suficiente para que eles estejam disponíveis antes que eles sejam requisitados pelo processador, sendo preciso identificar quais são os dados críticos devem ser selecionados e mantidos na memória *on-chip* – em uma espécie de *prefetching* via gestão de memória por *software*. Levando isso em consideração procurou-se selecionar aplicações representativas, mas que não fossem demasiadamente complexas. A intenção era que esses aplicativos representassem casos bons e ruins para a memória *scratchpad* permitindo uma boa avaliação deste projeto. Além disso, devido a ausência da infraestrutura oferecida pelo sistema operacional, várias versões dos algoritmos foram criadas de modo que o código fosse o mais independente possível dessas bibliotecas, por exemplo, para alocação dinâmica de dados.

Para realizar os experimentos, para que ele tivesse um paralelo com uma organização real, o Simulador descrito na Seção 3.4.2 teve sua configuração baseada no processador Samsung Exynos 4210 (Samsung Electronics, 2011): 64 MB de memória principal, 32 KB de *cache* de dados Nível 1 com 256 *bytes* por linha, conjunto com 4 entradas (256 B/line, 4-WAY) e latência de 4 ciclos, 1 MB de *cache* Nível 2 com 256 *bytes* por linha, conjunto com 16 entradas (256 B/line, 16-WAY) e latência de 20 ciclos. Para fornecer os parâmetros de comparação, a máquina foi configurada com os seguintes cenários de execução:

- apenas com a *cache* e com *prefetching* desabilitado;
- apenas com a *cache* e *prefetching* habilitado;
- com a *cache* e a memória *scratchpad* simultaneamente disponíveis e o *prefetching* desabilitado;
- com a *cache* e a memória *scratchpad* simultaneamente disponíveis e o *prefetching* habilitado.

A seguir são apresentadas e discutidas cada uma das aplicações do *benchmark*, os resultados obtidos pelas execuções dos experimentos nos cenários descritos e a interpretação desses resultados.

4.1 *Benchmark*

4.1.1 Adição de Matrizes

O primeiro teste utilizado foi um algoritmo para adição de duas matrizes. Embora trate-se de um exemplo simples, além de didático e permitir uma gradação na complexidade dos problemas utilizados, ele também é útil para mostrar o funcionamento da memória *scratchpad* modelada e compará-la com as outras abordagens disponíveis. O Código 4.1 mostra a solução originalmente utilizada para resolver o problema. Inicialmente das linhas 10 a 26 os dados são carregados para a memória, inicializando os arranjos *mat1* e *mat2* com as informações das matrizes que eles abstraem. Da linha 29 até a 34 a computação é realizada, somando as posições (i, j) de *mat1* e *mat2*, e armazenando seu resultado no arranjo *res*. Finalmente, das linhas de 37 a 45 o resultado do processamento é salvo no arquivo de saída.

Código 4.1: Solução dada originalmente para computar a adição de duas matrizes.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "sp.h"
4
5 int main(void){
6   const int SIZE = 4096;
7
8   int mat1[SIZE][SIZE], mat2[SIZE][SIZE], res[SIZE][SIZE];
9
```

```
10    FILE * file;
11    file = fopen ("In.txt","r");
12    int entr;
13    // Matriz 1
14    for (int i = 0; i < SIZE; i++)
15      for (int j = 0; j < SIZE; j++){
16        fscanf(file, "%d", &entr);
17        mat1[i][j] = entr;
18      }
19    // Matriz 2
20    for (int i = 0; i < SIZE; i++)
21      for (int j = 0; j < SIZE; j++){
22        fscanf(file, "%d", &entr);
23        mat2[i][j] = entr;
24      }
25
26    fclose(file);
27
28    toggleLog();
29    // Compute
30    for (int i = 0; i < SIZE; i++){
31      for (int j = 0; j < SIZE; j++){
32          res[i][j] = mat1[i][j] + mat2[i][j];
33      }
34    }
35    toggleLog();
36
37    // Save
38    FILE * fileOut;
39    fileOut = fopen ("Out.txt","w");
40
41    for (int i = 0; i < SIZE; i++){
42      for (int j = 0; j < SIZE; j++)
43        fprintf(fileOut, "%d ", res[i][j]);
44      // fprintf(fileOut, "\n");
45    }
```

```
46
47    fclose(fileOut);
48
49    return 0;
50 }
```

O Código 4.2 mostra a versão do algoritmo para adição de duas matrizes usando a memória *scratchpad*, também representadas por *mat1* e *mat2*. Da linha 11 até a 28 os dados são carregados para a memória principal. Da linha 32 a 35 é alocado espaço suficiente para conter quatro linhas das matrizes, para que seja possível o carregamento antecipado dos dados a serem utilizados na próxima iteração da adição. No trecho de código de 37 até 40, os dados da primeira linha das matrizes são trazidos para a memória *scratchpad*. Das linhas 42 até 64 a computação da adição é efetivamente realizada, a seguir nas linhas de 46 a 58 o algoritmo carrega alternadamente na memória *scratchpad* os dados necessários na próxima iteração, realizando uma espécie de *tiling* para produzir um *prefetching* por *software*. Finalmente, os valores antecipadamente carregados são utilizados para computar a adição, conforme mostrado nas linhas que vão de 60 até 63.

Código 4.2: Código utilizado para computar a adição de duas matrizes utilizando a memória *scratchpad*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "sp.h"
4
5  #define SIZE 4096
6  #define SPM_BASE 800
7
8  int main(void){
9     int mat1[SIZE][SIZE], mat2[SIZE][SIZE], res[SIZE][SIZE];
10
11    FILE * file;
12    file = fopen("In.txt","r");
13    int entr;
14    // Matriz 1
15    for (int i = 0; i < SIZE; i++)
16       for (int j = 0; j < SIZE; j++){
17          fscanf(file, "%d", &entr);
```

```
18          mat1[i][j] = entr;
19        }
20
21      // Matriz 2
22      for (int i = 0; i < SIZE; i++)
23        for (int j = 0; j < SIZE; j++){
24          fscanf(file, "%d", &entr);
25          mat2[i][j] = entr;
26        }
27
28      fclose(file);
29
30      toggleLog();
31
32      int lim  = 2*SIZE + 2*SIZE;
33      int *spm = (int*)spalloc(SPM_BASE);
34      for (int i = 16; i < lim; i += 16)
35        spalloc(SPM_BASE + i);
36
37      for (int i = 0; i < SIZE; i+=16)
38        spload((int)(spm + i), &mat1[0][i]);
39      for (int i = 0; i < SIZE; i+=16)
40        spload((int)(spm + 2*SIZE + i), &mat2[0][i]);
41
42      // Compute
43      int soma;
44      int desloc = SIZE;
45      for (int i = 0; i < SIZE; i++){
46        // Antecipar o carregamento se nao for estourar a matriz
47        if (i+1 < SIZE){
48          for (int j = 0; j < SIZE; j+=16)
49            spload((int)(spm + j + desloc), &mat1[i+1][j]);
50
51          for (int j = 0; j < SIZE; j+=16)
52            spload((int)(spm + 2*SIZE + j + desloc), &mat2[i+1][j]);
53        }
```

```
54
55      if (desloc == SIZE)
56        desloc = 0;
57      else
58        desloc = SIZE;
59
60      for (int j = 0; j < SIZE; j++){
61        soma = *(spm + j + desloc) + *(spm + 2*SIZE + j + desloc);
62        res[i][j] = soma;
63      }
64    }
65    toggleLog();
66
67    // Save
68    FILE * fileOut;
69    fileOut = fopen ("Out.txt","w");
70
71    for (int i = 0; i < SIZE; i++){
72      for (int j = 0; j < SIZE; j++)
73        fprintf(fileOut, "%d ", res[i][j]);
74    }
75
76    fclose(fileOut);
77
78    return 0;
79  }
```

4.1.2 Histograma

Um histograma, como o da Figura 4.1, é a representação gráfica para as distribuições de frequência. As medidas ou observações são agrupadas em uma escala horizontal e verticalmente marcam-se as alturas iguais ao número de vezes que cada uma dessas classes ocorre (FREUND, 2004).

O Código 4.3 implementa o cálculo de um histograma para uma imagem em preto e branco com um sistema de cores cujos valores de cada pixel variam de 0 a 255. A matriz *brightnessLevel* representa cada um dos *pixels* da imagem e é preenchida da linha 15 até a 24, enquanto o arranjo *hist* é utilizado para conter a frequência de cada

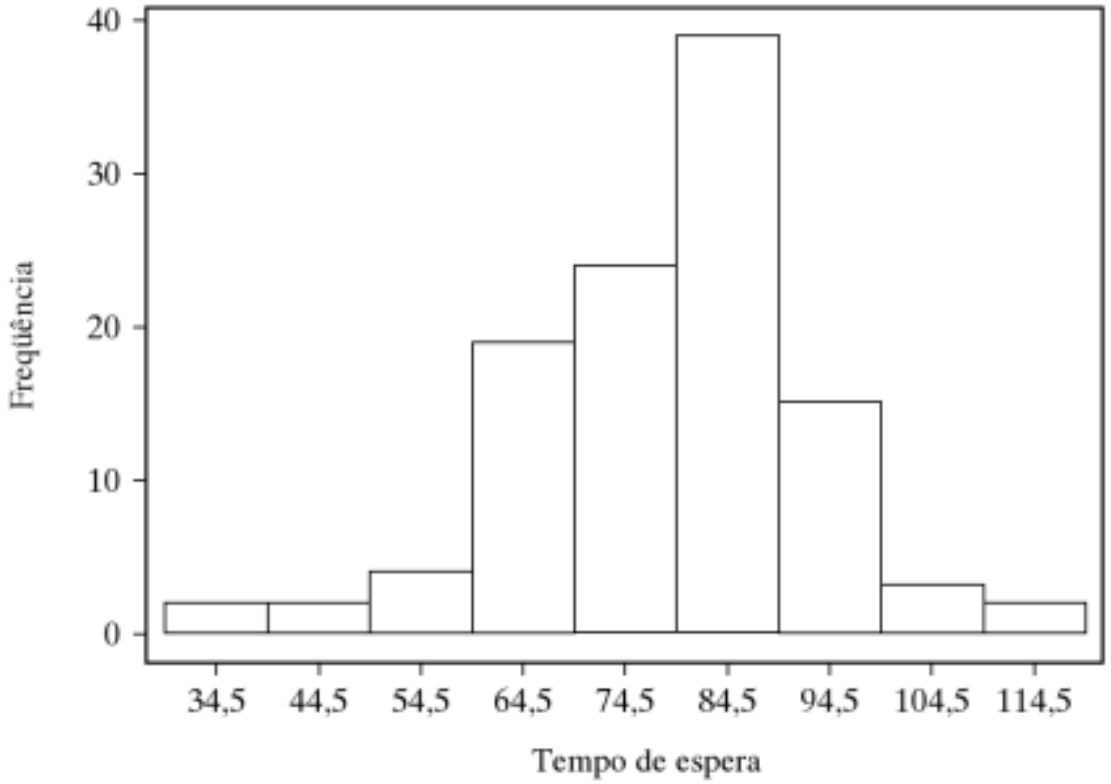

Figura 4.1: Histograma para os tempos de espera entre erupções do gêiser *Old Faithful*. Extraído de (FREUND, 2004).

valor possível para um *pixel*, sendo calculado da linha 27 até a 34. Nelas o valor do *pixel* é lido a partir de *brightnessLevel* e armazenado em *level*, sendo usado para permitir o incremento de *hist* a partir de seu valor. As linhas de 37 a 47 são utilizadas para salvar o valor do histograma calculado.

Código 4.3: Código para computar o histograma de uma imagem preto e branco cujos valores de cada pixel variam de 0 a 255.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "sp.h"
4
5 int main(void){
6   const int SIZE = 512;
7
8   int brightnessLevel[SIZE][SIZE];
9   int hist[256];
```

```
10
11    // Prepare
12    for (int i = 0; i < 256; i++)
13      hist[i] = 0;
14
15    FILE * file;
16    file = fopen ("In.txt","r");
17    int entr;
18    for (int i = 0; i < SIZE; i++)
19      for (int j = 0; j < SIZE; j++){
20        fscanf(file, "%d", &entr);
21        brightnessLevel[i][j] = entr;
22      }
23
24    fclose(file);
25
26    toggleLog();
27    // Compute
28    int level;
29    for (int i = 0; i < SIZE; i++){
30      for (int j = 0; j < SIZE; j++){
31        level = brightnessLevel[i][j];
32        hist[level] = hist[level] + 1;
33      }
34    }
35    toggleLog();
36
37    // Save
38    FILE * fileOut;
39    fileOut = fopen ("Out.txt","w");
40
41    for (int i = 0; i < 256; i++)
42      fprintf(fileOut, "%d ", hist[i]);
43
44    fclose(fileOut);
45
```

```
46    return 0;
47 }
```

O Código 4.4 é a versão do Código 4.3 rescrito considerando a memória *scratchpad*. De modo análogo ao exemplo anterior, as linhas de 10 a 22 carregam os dados para a memória e as linhas de 65 a 72 realizam a saída da computação, salvando os dados em disco. Inicialmente, da linha 25 até a 33, a memória *scratchpad* é alocada, considerando-se o espaço necessário para armazenar duas linhas do arranjo *brightnessLevel* que representa a imagem, mais todo o vetor *hist* que representa o histograma. A alocação é efetuada a cada 16 endereços devido ao tamanho do bloco da *cache*. Neste exemplo com um esquema de *double buffering* um *prefetching* por *software* é realizado carregando antecipadamente as linhas de *brightnessLevel*. Assim, nas linhas de código de 31 a 33 a primeira linha deste arranjo é carregada para a memória *scratchpad* e as linhas de 37 a 40 inicializam o vetor *hist* utilizado para computar o histograma. O trecho de código que começa em 42 e termina em 63, computa o histograma. Neste conjunto, de 46 a 56 a linha de *brightnessLevel* a ser utilizada na próxima iteração é carregada, alternando entre os dois espaços alocados para esse fim. Esse carregamento é realizado de modo alternado para que haja tempo o suficiente entre a solicitação do carregamento e o seu uso subsequente nas iterações seguintes. Finalmente, as linhas de 58 a 61 utilizam os dados carregados na memória *scratchpad* para calcular o histograma. É interessante notar que o vetor *hist*, que também foi armazenado na *scratchpad*, mesmo sendo pequeno, nunca estará sujeito as falhas da *cache*, sejam elas geradas por conflitos ou compulsórias.

Código 4.4: Código usado para computar o histograma utilizando a memória *scratchpad*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "sp.h"
4
5  #define SIZE 512
6  #define SPM_BASE 800
7
8  int main(void){
9
10    int brightnessLevel[SIZE][SIZE];
11    int* hist;
12
```

```
13    FILE * file;
14    file = fopen ("In.txt","r");
15    int entr;
16    for (int i = 0; i < SIZE; i++)
17      for (int j = 0; j < SIZE; j++){
18        fscanf(file, "%d", &entr);
19        brightnessLevel[i][j] = entr;
20      }
21
22    fclose(file);
23
24    toggleLog();
25    // Tamanho e Histograma
26    int lim = 2*SIZE + 256;
27    int *spm = (int*)spalloc(SPM_BASE);
28    for (int i = 16; i < lim; i += 16)
29      spalloc(SPM_BASE + i);
30
31    for (int i = 0; i < SIZE; i+=16){
32      spload((int)(spm + i), &brightnessLevel[0][i]);
33    }
34
35    int desloc = SIZE;
36
37    // Prepare
38    hist = spm + 2*SIZE;
39    for (int i = 0; i < 256; i++)
40      hist[i] = 0;
41
42    // Compute
43    int level;
44    for (int i = 0; i < SIZE; i++){
45
46      if (i + 1 < SIZE){
47        for (int j = 0; j < SIZE; j+=16){
48          spload((int)(spm + j + desloc),
```

```
49              &brightnessLevel[i + 1][j]);
50          }
51        }
52
53        if (desloc == 0)
54          desloc = SIZE;
55        else
56          desloc = 0;
57
58        for (int j = 0; j < SIZE; j++){
59          level = *(spm + j + desloc);
60          hist[level] = hist[level] + 1;
61        }
62      }
63      toggleLog();
64
65      // Save
66      FILE * fileOut;
67      fileOut = fopen ("Out.txt","w");
68
69      for (int i = 0; i < 256; i++)
70        fprintf(fileOut, "%d ", hist[i]);
71
72      fclose(fileOut);
73
74      return 0;
75  }
```

4.1.3 Algoritmo de Dijkstra

O algoritmo de Dijkstra (Dijkstra, 1959) é utilizado para encontrar o caminho mais curto em um grafo com arestas de peso não negativo. O Código 4.5 mostra uma versão do algoritmo utilizando uma matriz *graph* para representar o grafo. Nela um valor positivo p em uma posição (i, j) indica que há uma aresta no grafo com peso p, saindo do vértice identificado pelo índice da linha i para o vértice identificado com o índice da coluna j. Um peso nulo indica que não há uma aresta entre o par (i, j). Além disso, utilizam-se: (1) um vetor *dist* para representar as distâncias dos vértices do grafo para

seus antecessores, (2) *prev* como a lista de antecessores de cada vértice, (3) o vetor q com o escalar *qCount* para representar a fila dos elementos que ainda não foram usados para calcular a distância dos vértices alcançáveis nas próximas iterações do algoritmo e (4) o vértice inicial ou raiz *source*. No exemplo, as linhas de 27 a 33 carregam o grafo para a memória principal e as linhas de 38 a 72 efetivamente realizam a computação. De 38 a 44 as inicializações previstas no algoritmo de Dijkstra são realizadas, o *while* que se inicia na linha 46 percorre todos os vértices, calculando a distância total do caminho que passa por ele até cada um dos seus sucessores, as atualizando no *if* da linha 67, se isso for vantajoso. Detalhadamente, o *for* da linha 50 seleciona o vértice com menor distância u que ainda não tenha sido utilizado, conforme o controle realizado por q, armazenando-o em *dist*. A linha 58 remove o vértice selecionado de q e em 59 utilizando *qCount* decrementa-se a quantidade de vértices restantes em q. O *for* que se inicia na linha 61 calcula a distância da raiz até os sucessores do vértice u se o caminho que leva até eles passar por u. Se este caminho for menor que os já existentes, conforme verificado na linha 67, a distância do sucessor é atualizada e u é feito seu antecessor, finalizando o algoritmo.

Código 4.5: Algoritmo de Dijkstra.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "sp.h"
4
5  #define SIZE 200
6  #define INFINITY 9999999
7  #define UNDEFINED -23
8
9  int main(void){
10   int graph[SIZE][SIZE];
11   int dist[SIZE];
12   int prev[SIZE];
13   int source;
14   int u;
15   // Queue
16   int q[SIZE];
17   int qCount = SIZE;
18
19   FILE * file;
```

```
20    file = fopen("In.txt", "r");
21    int entr;
22
23    // Source
24    fscanf(file, "%d", &entr);
25    source = entr;
26
27    // Graph
28    for (int i = 0; i < SIZE; i++){
29      for (int j = 0; j < SIZE; j++){
30        fscanf(file, "%d", &entr);
31        graph[i][j] = entr;
32      }
33    }
34    fclose(file);
35
36    toggleLog();
37
38    for (int i = 0; i < SIZE; i++){
39      dist[i] = INFINITY;
40      prev[i] = UNDEFINED;
41      q[i] = 1;
42    }
43
44    dist[source] = 0;
45
46    while (qCount > 0){
47
48      // Get the vertex with the smaller distance
49      int min = INFINITY;
50      for (int i = 0; i < SIZE; i++){
51        if (min > dist[i] && q[i]){
52          min = dist[i];
53          u = i;
54        }
55      }
```

```
56
57      // Remove u from q
58      q[u] = 0;
59      qCount--;
60
61      for (int v = 0; v < SIZE; v++){
62        if (graph[u][v] == 0)
63          continue;
64
65        int alt = graph[u][v] + dist[u];
66
67        if (alt < dist[v]){
68          dist[v] = alt;
69          prev[v] = u;
70        }
71      }
72    }
73
74    toggleLog();
75
76    // Save
77    FILE * fileOut;
78    fileOut = fopen("Out.txt", "w");
79
80    // Previous
81    for (int i = 0; i < SIZE; i++){
82      fprintf(fileOut, "%d ", prev[i]);
83    }
84
85    // Distance
86    for (int i = 0; i < SIZE; i++){
87      fprintf(fileOut, "%d ", dist[i]);
88    }
89
90    fclose(fileOut);
91  }
```

O Código 4.6 apresenta a versão alterada do algoritmo de Dijkstra com a inclusão do uso da memória *scratchpad*. Até a linha 48 são tratados o carregamento da matriz de adjacências para a memória principal e a alocação da memória *scratchpad*. Uma das alterações realizadas para melhorar a eficiência do algoritmo pode ser vista nas linhas de 52 a 55. Nelas os arranjos *dist*, *prev* e *q* são armazenados na memória *scratchpad*, além de um arranjo *line* que foi definido para conter todos os sucessores de um vértice. O *for* da linha 57 inicializa as estruturas de dados do algoritmo, a linha 63 inicializa a raiz definindo a sua distância e a seguir o *while* da linha 65 efetivamente realiza a computação. O vértice com menor distância da lista de vértices disponíveis é obtido no bloco que se inicia na linha 67 e vai até a 74, e finalmente, a seguir é carregada na memória *scratchpad* toda sua lista de adjacência conforme as linhas 76 e 77. As demais etapas do processamento são análogas.

Código 4.6: Algoritmo de Dijkstra utilizando a memória *scratchpad*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "sp.h"
4
5  #define SIZE 200
6  #define INFINITY 9999999
7  #define UNDEFINED -23
8  #define BLOCK_SIZE 16
9  #define SPM_BASE 800
10
11 int main(void){
12    int graph[SIZE][SIZE];
13    int* dist;
14    int* prev;
15    int* line;
16    int source;
17    int u;
18    // Queue
19    int* q;
20    int qCount = SIZE;
21
22    int lineSize = SIZE / BLOCK_SIZE;
23    if (SIZE % 16 != 0)
```

```
24      lineSize++;
25    lineSize *= 16;
26    // Allocate SPM
27    // Matrix line, matrix line, dist, prev, q
28    int lim = 4 * lineSize;
29    int *spm = (int*)spalloc(SPM_BASE);
30    for (int i = 16; i < lim; i+=16)
31      spalloc(SPM_BASE + i);
32
33    FILE * file;
34    file = fopen("In.txt", "r");
35    int entr;
36
37    // Source
38    fscanf(file, "%d", &entr);
39    source = entr;
40
41    // Graph
42    for (int i = 0; i < SIZE; i++){
43      for (int j = 0; j < SIZE; j++){
44        fscanf(file, "%d", &entr);
45        graph[i][j] = entr;
46      }
47    }
48    fclose(file);
49
50    toggleLog();
51
52    dist = spm;
53    prev = (spm + lineSize);
54    q = (spm + 2 * lineSize);
55    line = (spm + 3 * lineSize);
56
57    for (int i = 0; i < SIZE; i++){
58      dist[i] = INFINITY;
59      prev[i] = UNDEFINED;
```

```
60      q[i] = 1;
61    }
62
63    dist[source] = 0;
64
65    while (qCount > 0){
66
67      // Get the vertex with the smaller distance
68      int min = INFINITY;
69      for (int i = 0; i < SIZE; i++){
70        if (min > dist[i] && q[i]){
71          min = dist[i];
72          u = i;
73        }
74      }
75
76      for (int i = 0; i < lineSize; i += 16)
77        spload((int)(line+i), &graph[u][i]);
78
79      // Remove u from q
80      q[u] = 0;
81      qCount--;
82
83      for (int v = 0; v < SIZE; v++){
84        if (line[v] == 0)
85          continue;
86
87        int alt = line[v] + dist[u];
88
89        if (alt < dist[v]){
90          dist[v] = alt;
91          prev[v] = u;
92        }
93      }
94    }
95
```

```
96    toggleLog();
97
98    // Save
99    FILE * fileOut;
100   fileOut = fopen("Out.txt", "w");
101
102   // Previous
103   for (int i = 0; i < SIZE; i++){
104     fprintf(fileOut, "%d ", prev[i]);
105   }
106
107   // Distance
108   for (int i = 0; i < SIZE; i++){
109     fprintf(fileOut, "%d ", dist[i]);
110   }
111
112   fclose(fileOut);
113 }
```

4.1.4 Quicksort

O algoritmo Quicksort (Hoare, 1961), é uma técnica de ordenação considerada eficiente por realizar no caso médio $O(n \log n)$ comparações, apesar de serem $O(n^2)$ no pior caso. O Código 4.7 apresenta uma implementação recursiva do Quicksort. Os dados a serem ordenados são carregados no arranjo *mat* no *for* que se inicia na linha 16, em 23 a chamada inicial para o algoritmo é realizada e o restante da função *main* é utilizado para realizar a saída com o resultado do processamento do algoritmo. A função *quickSort* que se inicia na linha 39 é quem efetivamente implementa o algoritmo. Na linha 43 o pivô x é selecionado utilizando o ponto equidistante dos limites *esquerda* e *direita* passados como parâmetros e que representam os limites a serem percorridos do arranjo *valor*. O *while* que se inicia na linha 45 é responsável por realizar todas as trocas possíveis dos elementos utilizando o pivô selecionado anteriormente como parâmetro de comparação. Os elementos maiores que o pivô devem ser enviados para posições cujo índice seja maior que o do pivô e os elementos menores que o pivô devem ser enviados para as posições cujo índice seja menor que o do pivô, sempre trocando-se dois elementos que estejam em posições que não atendam essas diretrizes. O *while* que começa na linha 46 localiza um elemento maior que o pivô x e com índice menor.

O *while* da linha 49 localiza um elemento menor que o pivô x e com índice maior. Encontrados tais elementos o *if* da linha 52 faz a troca dos dois, de modo que o elemento maior que x seja copiado para a posição do menor, e vice-versa. A seguir se ainda houver elementos não ordenados, as chamadas recursivas das linhas 61 e 64 são efetuadas para tratá-los.

Código 4.7: Implementação recursiva do quicksort.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "sp.h"
4
5  void quickSort(int valor[], int esquerda, int direita);
6
7  int main(void){
8    const int SIZE = 8192;
9
10   int mat[SIZE];
11
12   FILE * file;
13   file = fopen ("In.txt","r");
14   int entr;
15
16   for (int i = 0; i < SIZE; i++){
17     fscanf(file, "%d", &entr);
18     mat[i] = entr;
19   }
20   fclose(file);
21
22   toggleLog();
23   quickSort(mat, 0, SIZE-1);
24   toggleLog();
25
26   // Save
27   FILE * fileOut;
28   fileOut = fopen ("Out.txt","w");
29
30   for (int i = 0; i < SIZE; i++){
```

```
31      fprintf(fileOut, "%d ", mat[i]);
32    }
33
34    fclose(fileOut);
35
36    return 0;
37  }
38
39  void quickSort(int valor[], int esquerda, int direita){
40      int i, j, x, y;
41      i = esquerda;
42      j = direita;
43      x = valor[(esquerda + direita) / 2];
44
45      while(i <= j){
46          while(valor[i] < x && i < direita){
47              i++;
48          }
49          while(valor[j] > x && j > esquerda){
50              j--;
51          }
52          if(i <= j){
53              y = valor[i];
54              valor[i] = valor[j];
55              valor[j] = y;
56              i++;
57              j--;
58          }
59      }
60      if(j > esquerda){
61          quickSort(valor, esquerda, j);
62      }
63      if(i < direita){
64          quickSort(valor, i, direita);
65      }
66  }
```

O Código 4.8 mostra a versão recursiva do algoritmo quicksort alterada para utilizar a memória *scratchpad*. A única diferença relevante desse algoritmo em relação a versão original ilustrada no Código 4.7 é que todos os dados do vetor a serem ordenados são copiados para a memória *scratchpad* como pode ser visto nas linhas de 31 a 33. A seguir usando essa cópia das informações, a ordenação é realizada impedindo que quaisquer falhas ocorram. Deve-se observar que em alguns casos não seria possível carregar todos os dados do vetor na *scratchpad* e outra técnica poderia ser necessária.

Código 4.8: Código utilizado para computar a adição de duas matrizes utilizando a memória *scratchpad*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "sp.h"
4
5  void quickSort(int valor[], int esquerda, int direita);
6
7  #define SIZE 32
8  #define VAR_SIZE 16
9  const int SPM_SIZE = VAR_SIZE + SIZE;
10
11 int main(void){
12
13    int mat[SIZE];
14    int *vbase;
15    int *spm = spalloc(0);
16    for (int i = VAR_SIZE; i < SPM_SIZE; i += VAR_SIZE)
17       spalloc(i);
18
19    FILE * file;
20    file = fopen("In.txt", "r");
21    int entr;
22    // Matriz 1
23    for (int i = 0; i < SIZE; i++){
24       fscanf(file, "%d", &entr);
25       mat[i] = entr;
26    }
27    fclose(file);
```

```
28
29    toggleLog();
30
31    vbase = spm;
32    for (int i = 0; i < SIZE; i += VAR_SIZE)
33      sload((int)(spm + i), &mat[i]);
34
35    quickSort(vbase, 0, SIZE - 1);
36    toggleLog();
37
38    // Save
39    FILE * fileOut;
40    fileOut = fopen("Out.txt", "w");
41
42    for (int i = 0; i < SIZE; i++){
43      fprintf(fileOut, "%d ", vbase[i]);
44    }
45
46    fclose(fileOut);
47
48    return 0;
49  }
50
51  void quickSort(int valor[], int esquerda, int direita){
52    int i, j, x, y;
53    i = esquerda;
54    j = direita;
55    x = valor[(esquerda + direita) / 2];
56
57    while (i <= j)     {
58      while (valor[i] < x && i < direita){
59        i++;
60      }
61      while (valor[j] > x && j > esquerda){
62        j--;
63      }
```

```
64        if (i <= j){
65          y = valor[i];
66          valor[i] = valor[j];
67          valor[j] = y;
68          i++;
69          j--;
70        }
71      }
72      if (j > esquerda){
73        quickSort(valor, esquerda, j);
74      }
75      if (i < direita){
76        quickSort(valor, i, direita);
77      }
78  }
```

4.2 Execução dos Experimentos

Esta seção trata da execução dos experimentos e das saídas produzidas por eles. Em linhas gerais utilizaram-se entradas geradas aleatoriamente com diferentes tamanhos para que múltiplas execuções fessem realizadas para verificar o desempenho e a correção dos algoritmos. Duas verificações das saídas foram realizadas, uma analisando a saída esperada para um caso não muito grande e outra comparando-a com uma versão produzida pelo mesmo algoritmo compilado para a arquitetura Intel x86, empregando, neste caso, o compilador sem qualquer modificação. Utilizaram-se quatro versões da arquitetura proposta, conforme já assinalado: (1) somente a *cache* sem *prefetching*, (2) somente a *cache* com *prefetching* habilitado, (3) a *cache* e a memória *scratchpad* disponíveis sem *prefetching* e (4) a *cache* e a memória *scratchpad* disponíveis com *prefetching* habilitado.

Considerando as quatro configurações da máquina citadas acima, a seguir apresentam-se os resultados da execução dos experimentos de acordo com cada algoritmo selecionado, medindo-se:

- o número de ciclos gastos para executar;
- o número de falhas obtidas pela execução do algoritmo;
- o número de acertos obtidos pela execução do algoritmo.

Para medir os resultados e propiciar uma avaliação razoável do que foi obtido, procurou-se medir a computação útil realizada pelos algoritmos. As inicializações e carregamento de dados para a memória principal não foram considerados úteis, tende side considerado apenas o processamento diretamente relacionado ao problema em si. Outro item que precisa ser levado em consideração está relacionado às transferências realizadas pelo *hardware* com destino na memória *scratchpad*. Embora a *scratchpad* possa gerar *stalls* e ela efetivamente não gere falhas relacionadas à memória *on-chip*, como ela utiliza a infraestrutura disponível para a *cache*, nos experimentos a seguir o número de falhas atribuídos à memória *scratchpad* na verdade expressam o pior caso que pode ocorrer para ela: os dados requisitados estão disponíveis apenas na memória principal, devendo neste cenário, realizar a transferência mais custosa.

4.2.1 Adição de Matrizes

A Tabela 4.1 mostra a execução do algoritmo para adição de matrizes em um ambiente em que apenas a *cache* estava disponível. A coluna Entrada mostra o tamanho da entrada utilizada, a coluna *Cache* o número de ciclos gastos para executar o algoritmo sem *prefetching*, a coluna *Cache Prefetch* o número de ciclos gastos para executar o algoritmo quando o *prefetching* estava habilitado, a coluna Diferença o número de ciclos economizados quando o *prefetching* estava habilitado e a coluna Desempenho ganho o percentual correspondente a diferença da coluna anterior.

Tabela 4.1: Desempenho em ciclos do algoritmo para adição de matrizes apenas com a *cache*.

Entrada	*Cache*	*Cache Prefetch*	Diferença	Desempenho ganho
32	17813	13915	3898	21,9%
64	69654	38931	30723	44,1%
128	274023	151135	122888	44,8%
256	1088310	596733	491577	45,2%
512	4339044	2372777	1966267	45,3%
1024	17329117	9464114	7865003	45,4%

A Tabela 4.2 mostra a execução do algoritmo para adição de matrizes em um ambiente em que a *cache* e a memória *scratchpad* estavam disponíveis. A coluna Entrada mostra o tamanho da entrada utilizada, a coluna *Scratchpad* o número de ciclos gastos para executar o algoritmo sem *prefetching*, a coluna *Scratchpad Prefetch* o número de ciclos gastos para executar o algoritmo quando o *prefetching* estava habilitado, a coluna

Diferença o número de ciclos economizados quando o *prefetching* estava habilitado e a coluna Desempenho ganho o percentual correspondente a diferença da coluna anterior.

Tabela 4.2: Desempenho em ciclos do algoritmo para adição de matrizes com a *cache* e a memória *scratchpad* disponíveis.

Entrada	*Scratchpad*	*Scratchpad Prefetch*	Diferença	Desempenho ganho
32	110758	110758	0	0,0%
64	172690	171681	1009	0,6%
128	1025710	1023174	2536	0,2%
256	4097324	4095411	1913	0,0%
512	30849378	30846884	2494	0,0%
1024	123439346	123436090	3256	0,0%

A Tabela 4.3 mostra a comparação das execuções do algoritmo para adição de matrizes considerando o melhor caso nos ambientes em que somente a *cache* estava disponível e a alternativa em que tanto a *cache* quanto memória *scratchpad* estavam disponíveis. A coluna Entrada mostra o tamanho da entrada utilizada, as colunas *Cache* e *Scratchpad* o número de ciclos gastos para executar o algoritmo nos dois casos, a coluna Diferença o número de ciclos economizados quando utilizou-se a máquina com a memória *scratchpad* disponível e a coluna Desempenho ganho o percentual correspondente a diferença da coluna anterior.

Tabela 4.3: Comparativo entre os resultados obtidos para a *cache* e para a memória *scratchpad*.

Entrada	*Cache*	*Scratchpad*	Diferença	Desempenho ganho
32	103915	110758	-6843	-6,6%
64	38931	171681	-132750	-341,0%
128	151135	1023174	-872039	-577,0%
256	596733	4095411	-3498678	-586,3%
512	2372777	30846884	-28474107	-1200,0%
1024	9464114	123436090	-113971976	-1204,3%

A Tabela 4.4 mostra o número de falhas da execução do algoritmo para adição de matrizes. As colunas representam o número de falhas para as quatro configurações da máquina quando os algoritmos foram executados.

A Tabela 4.5 mostra o número de acertos da execução do algoritmo para adição de matrizes, onde as colunas representam o número de acertos para as quatro configurações da máquina quando os algoritmos foram executados.

Tabela 4.4: Número de *falhas* obtidas na execução do algoritmo para adição de matrizes

Cache	*Cache Prefetch*	*Scratchpad*	*Scratchpad Prefetch*
129	2	129	129
515	3	515	515
2051	3	2052	2052
8195	3	8196	8196
32771	3	32772	32772
131075	3	132098	132098

Tabela 4.5: Número de *acertos* obtidos na execução do algoritmo para adição de matrizes

Cache	*Cache Preftch*	*Scratchpad*	*Scratchpad Prefetch*
1919	2046	3967	3967,00
7869	8381	16249	16249,00
31101	33149	64756	64756,00
123645	131837	258285	258285,00
493053	525821	1021942	1021942,00
1969149	2100221	4074488	4074488,00

4.2.2 Histograma

A Tabela 4.6 mostra a execução do algoritmo para cálculo do histograma em um ambiente em que apenas a *cache* estava disponível. A coluna Entrada mostra o tamanho da entrada utilizada, a coluna *Cache* o número de ciclos gastos para executar o algoritmo sem *prefetching*, a coluna *Cache Prefetch* o número de ciclos gastos para executar o algoritmo quando o *prefetching* estava habilitado, a coluna Diferença o número de ciclos economizados quando o *prefetching* estava habilitado e a coluna Desempenho ganho o percentual correspondente a diferença da coluna anterior.

Tabela 4.6: Desempenho em ciclos do algoritmo para cálculo de histogramas apenas com a *cache*.

Entrada	*Cache*	*Cache Prefetch*	Diferença	Desempenho ganho
32	27321	18461	8860	32,4%
64	102766	70885	31881	31,0%
128	404147	280130	124017	30,7%
256	1609002	1116480	492522	30,6%
512	6426676	4459560	1967116	30,6%
1024	25692803	17832770	7860033	30,6%

A Tabela 4.7 mostra a execução do algoritmo para cálculo de histogramas em um ambiente em que a *cache* e a memória *scratchpad* estavam disponíveis. A coluna Entrada mostra o tamanho da entrada utilizada, a coluna *Scratchpad* o número de ciclos gastos para executar o algoritmo sem *prefetching*, a coluna *Scratchpad Prefetch* o número de ciclos gastos para executar o algoritmo quando o *prefetching* estava habilitado, a coluna Diferença o número de ciclos economizados quando o *prefetching* estava habilitado e a coluna Desempenho ganho o percentual correspondente a diferença da coluna anterior.

Tabela 4.7: Desempenho em ciclos do algoritmo para cálculo de histogramas com a *cache* e a memória *scratchpad* disponíveis.

Entrada	*Scratchpad*	*Scratchpad Prefetch*	Diferença	Desempenho ganho
32	20677	20677	0	0,0%
64	77389	77383	6	0,0%
128	303503	303505	-2	0,0%
256	1205986	1205985	1	0,0%
512	4892337	4892343	-6	0,0%
1024	19549477	19549469	8	0,0%

A Tabela 4.8 mostra a comparação das execuções do algoritmo para cálculo de histogramas para o melhor caso nos ambientes em que somente a *cache* estava disponível e a alternativa em que tanto a *cache* quanto memória *scratchpad* estavam disponíveis. A coluna Entrada mostra o tamanho da entrada utilizada, as colunas *Cache* e *Scratchpad* o número de ciclos gastos para executar o algoritmo nos dois casos, a coluna diferença o número de ciclos economizados quando utilizou-se a máquina com a memória *scratchpad* disponível e a coluna Desempenho ganho o percentual correspondente a diferença da coluna anterior.

Tabela 4.8: Comparativo entre os resultados obtidos para a *cache* e para a memória *scratchpad*.

Entrada	*Cache*	*Scratchpad*	Diferença	Desempenho ganho
32	18461	20677	-2216	-12,0%
64	70885	77383	-6498	-9,2%
128	280130	303505	-23375	-8,3%
256	1116480	1205985	-89505	-8,0%
512	4459560	4892343	-432783	-9,7%
1024	17832770	19549469	-1716699	-9,6%

A Tabela 4.9 mostra o número de falhas da execução do algoritmo para cálculo do

histograma. As colunas representam o número de falhas para as quatro configurações da máquina quando os algoritmos foram executados.

Tabela 4.9: Número de *falhas* obtidas na execução do algoritmo para cálculo de histogramas

Cache	*Cache Prefetch*	*Scratchpad*	*Scratchpad Prefetch*
81	8	66	3007
273	8	258	12031
1041	8	1026	48127
4113	8	4098	192511
16401	8	16385	771070
65553	8	65537	3082238

A Tabela 4.10 mostra o número de acertos da execução do algoritmo para cálculo de histogramas. As colunas representam o número de acertos para as quatro configurações da máquina quando os algoritmos foram executados.

Tabela 4.10: Número de *acertos* obtidos na execução do algoritmo para cálculo do histograma

Cache	*Cache Preftch*	*Scratchpad*	*Scratchpad Prefetch*
1967	2040	66	3007,00
7919	8184	258	12031,00
31727	32760	1026	48127,00
126959	131064	4098	192511,00
507887	524280	16385	771070,00
2031599	2097144	65537	3082238,00

4.2.3 Algoritmo de Dijkstra

A Tabela 4.11 mostra a execução do algoritmo de Dijkstra em um ambiente em que apenas a *cache* estava disponível. A coluna Entrada mostra o tamanho da entrada utilizada, a coluna *Cache* exibe o número de ciclos gastos para executar o algoritmo sem *prefetching*, a coluna *Cache Prefetch* mostra o número de ciclos gastos para executar o algoritmo quando o *prefetching* estava habilitado, a coluna Diferença exibe o número de ciclos economizados quando o *prefetching* estava habilitado e a coluna Desempenho ganho mostra o percentual correspondente a diferença da coluna anterior.

A Tabela 4.12 mostra a execução do algoritmo de Dijkstra em um ambiente em que a *cache* e a memória *scratchpad* estão disponíveis. A coluna Entrada mostra o

Tabela 4.11: Desempenho em ciclos do algoritmo de Dijkstra apenas com a *cache*.

Entrada	*Cache*	*Cache Prefetch*	Diferença	Desempenho ganho
32	55389	45750	9639	17,4%
64	213918	181688	32230	15,1%
128	826286	704233	122053	14,8%
256	3171570	2683327	488243	15,4%
512	12218243	10259179	1959064	16,0%
1024	49742331	41867728	7874603	15,8%

tamanho da entrada utilizada, a coluna *Scratchpad* mostra o número de ciclos gastos para executar o algoritmo sem *prefetching*, a coluna *Scratchpad Prefetch* exibe o número de ciclos gastos para executar o algoritmo quando o *prefetching* está habilitado, a coluna Diferença exibe o número de ciclos economizados quando o *prefetching* está habilitado e a coluna Desempenho ganho exibe o percentual correspondente a diferença da coluna anterior.

Tabela 4.12: Desempenho em ciclos do algoritmo de Dijkstra com a *cache* e a memória *scratchpad* disponíveis.

Entrada	*Scratchpad*	*Scratchpad Prefetch*	Diferença	Desempenho ganho
32	43580	43479	101	0,2%
64	172350	172349	1	0,0%
128	665365	665377	-12	0,0%
256	2525226	2525210	16	0,0%
512	9513725	9513725	0	0,0%
1024	34689794	34689802	-8	0,0%

A Tabela 4.13 mostra a comparação das execuções do algoritmo de Dijkstra para o melhor caso nos ambientes em que somente a *cache* está disponível e a alternativa em que tanto a *cache* quanto a memória *scratchpad* estão disponíveis. A coluna Entrada mostra o tamanho da entrada utilizada, as colunas *Cache* e *Scratchpad* exibem o número de ciclos gastos para executar o algoritmo nos dois casos, a coluna Diferença mostra o número de ciclos economizados quando utilizou-se a máquina com a memória *scratchpad* disponível e a coluna Desempenho ganho mostra o percentual correspondente a diferença da coluna anterior.

A Tabela 4.14 mostra o número de falhas da execução do algoritmo de Dijkstra. As colunas representam o número de falhas para as quatro configurações da máquina quando os algoritmos foram executados.

Tabela 4.13: Comparativo entre os resultados obtidos para a *cache* e para a memória *scratchpad*.

Entrada	*Cache*	*Scratchpad*	Diferença	Desempenho ganho
32	45750	43479	2271	5,0%
64	181688	172349	9339	5,1%
128	704233	665377	38856	5,5%
256	2683327	2525210	158117	5,9%
512	10259179	9513725	745454	7,3%
1024	41867728	34689802	7177926	17,1%

Tabela 4.14: Número de *falhas* obtidas na execução do algoritmo de Dijkstra.

Cache	*Cache Prefetch*	*Scratchpad*	*Scratchpad Prefetch*
71	16	66	63
267	25	258	243
1091	57	1079	1039
4268	146	4135	4097
16721	292	16656	16624
65876	252	44161	43097

A Tabela 4.15 mostra o número de acertos da execução do algoritmo de Dijkstra. As colunas representam o número de acertos para as quatro configurações da máquina quando os algoritmos foram executados.

Tabela 4.15: Número de *acertos* obtidos na execução do algoritmo de Dijkstra.

Cache	*Cache Preftch*	*Scratchpad*	*Sratchpad Prefetch*
2935	2990	3964	3967,00
13255	13497	17360	17375,00
54590	55624	71002	71042,00
214013	218135	279614	279662,00
819957	836386	1083766	1083798,00
3458784	3524408	3340562	3340626,00

4.2.4 Quicksort

A Tabela 4.16 mostra a execução do algoritmo quicksort em um ambiente onde apenas a *cache* está disponível. A coluna Entrada mostra o tamanho da entrada utilizada, a coluna *Cache* exibe o número de ciclos gastos para executar o algoritmo sem *prefetching*, a coluna *Cache Prefetch* exibe o número de ciclos gastos para executar o algoritmo

quando o *prefetching* estava habilitado, a coluna Diferença mostra o número de ciclos economizados quando o *prefetching* está habilitado e a coluna Desempenho ganho exibe o percentual correspondente a diferença da coluna anterior.

Tabela 4.16: Desempenho em ciclos do algoritmo quicksort apenas com a *cache*.

Entrada	*Cache*	*Cache Prefetch*	Diferença	Desempenho ganho
32	5411	5291	120	2,2%
64	10336	10119	217	2,1%
128	22841	22138	703	3,1%
256	48795	47353	1442	3,0%
512	102876	100971	1905	1,9%
1024	220597	217325	3272	1,5%
2048	472223	460109	12114	2,6%
4096	1024943	1011502	13441	1,3%
8192	2209169	2180744	28425	1,3%

A Tabela 4.17 mostra a execução do algoritmo quicksort em um ambiente em que a *cache* e a memória *scratchpad* estão disponíveis. A coluna Entrada mostra o tamanho da entrada utilizada, a coluna *Scratchpad* mostra o número de ciclos gastos para executar o algoritmo sem *prefetching*, a coluna *Scratchpad Prefetch* ilustra o número de ciclos gastos para executar o algoritmo quando o *prefetching* está habilitado, a coluna Diferença exibe o número de ciclos economizados quando o *prefetching* está habilitado e a coluna Desempenho ganho mostra o percentual correspondente a diferença da coluna anterior.

Tabela 4.17: Desempenho em ciclos do algoritmo quicksort com a *cache* e a memória *scratchpad* disponíveis.

Entrada	*Scratchpad*	*Scratchpad Prefetch*	Diferença	Desempenho ganho
32	5298	5298	0	0,0%
64	9980	9980	0	0,0%
128	22040	22040	0	0,0%
256	46971	46971	0	0,0%
512	99452	99452	0	0,0%
1024	213771	213771	0	0,0%
2048	445235	445235	0	0,0%
4096	935189	935189	0	0,0%
8192	1800756	1800756	0	0,0%

A Tabela 4.18 mostra a comparação das execuções do algoritmo quicksort para o melhor caso nos ambientes onde somente a *cache* está disponível e a alternativa em que

tanto a *cache* quanto memória *scratchpad* estão disponíveis. A coluna Entrada mostra o tamanho da entrada utilizada, as colunas *Cache* e *Scratchpad* ilustram o número de ciclos gastos para executar o algoritmo nos dois casos, a coluna Diferença exibe o número de ciclos economizados quando utilizou-se a máquina com a memória *scratchpad* disponível e a coluna Desempenho ganho mostra o percentual correspondente a diferença da coluna anterior.

Tabela 4.18: Comparativo entre os resultados obtidos para a *cache* e para a memória *scratchpad*.

Entrada	*Cache*	*Scratchpad*	Diferença	Desempenho ganho
32	5291	5298	-7	-0,1%
64	10119	9980	139	1,4%
128	22138	22040	98	0,4%
256	47353	46971	382	0,8%
512	100971	99452	1519	1,5%
1024	217325	213771	3554	1,6%
2048	460109	450235	14874	3,2%
4096	1011502	975189	76313	7,5%
8192	2180744	1800756	379988	17,4%

A Tabela 4.19 mostra o número de falhas da execução do algoritmo quicksort. As colunas representam o número de falhas para as quatro configurações da máquina quando os algoritmos são executados.

Tabela 4.19: Número de *falhas* obtidas na execução do algoritmo quicksort.

Cache	*Cache Prefetch*	*Scratchpad*	*Scratchpad Prefetch*
6	5	6	6
9	7	6	6
14	10	14	14
24	12	24	24
39	24	39	39
73	48	73	73
137	99	137	137
267	155	267	267
543	303	581	581

A Tabela 4.20 mostra o número de acertos da execução do algoritmo quicksort. As colunas representam o número de acertos para as quatro configurações da máquina quando os algoritmos são executados.

Tabela 4.20: Número de *acertos* obtidos na execução do algoritmo quicksort.

Cache	*Cache Prefetch*	*Scratchpad*	*Scratchpad Prefetch*
434	435	466	466
876	878	940	940
2032	2036	2160	2160
4522	4534	4778	4778
9675	9690	10187	10187
20857	20882	21881	21881
44861	44899	46909	46909
97440	97552	96455	96455
209885	210125	199313	199313

4.3 Análise de Resultados

Os algoritmos selecionados para compor o *benchmark* de avaliação são simples. Eles foram selecionados de maneira que representassem um conjunto relevante de problemas da computação, mas que não fossem demasiadamente complexos. De modo geral a reescrita de código para a utilização da memória *scratchpad* é bastante complexa e para a demonstração e avaliação da proposta deste projeto de pesquisa, considerou-se o conjunto escolhido como representativo.

Os algoritmos para adição de matrizes e cálculo de histograma apresentam uma grande localidade espacial. Ambos são baseados na leitura de uma ou mais matrizes de modo sequencial e como o armazenamento desses dados é linearizado pelos compiladores das linguagens de alto nível, uma boa técnica de *prefetching* pode melhorar a eficiência nesses casos de modo substancial. Considerando algoritmo para adição de matrizes esse argumento fica muito nítido. Na Tabela 4.4 a primeira coluna mostra o número de falhas geradas quando o *hardware* não dispõe de *prefetching*. A medida que o tamanho da entrada é dobrado o número de falhas é aproximadamente multiplicado por 4, pois como tratam-se de matrizes, ao duplicar seu tamanho, sua largura e sua altura são duplicadas, obtendo-se este fator. Quando o *prefetching* é habilitado a diferença é impressionante: o número de falhas da computação útil é constante e como consequência há um ganho de desempenho de cerca de 44%. Neste caso a introdução da memória *scratchpad* precisaria superar um dos melhores casos da abordagem da *cache* onde a localidade espacial é altíssima. Na Tabela 4.3 é possível ver que a abordagem com a memória *scratchpad* não conseguiu melhorar o desempenho obtido pela *cache*, sendo na verdade bastante pior. Uma abordagem que poderia resultar em uma eficiência análoga a da *cache* com *prefetching* habilitado, consistiria em carregar os dados diretamente para memória *scratchpad* na inicialização do algoritmo. Por ser

considerada injusta para comparação, essa abordagem não foi utilizada. O gráfico da Figura 4.2 compara o desempenho obtido pelas abordagens da *cache* e da *scratchpad*. Nela é possível observar que a medida em que o tamanho da entrada aumenta, o número de ciclos utilizados pela *scratchpad* aumenta com uma derivada muito superior a da *cache*, seja ela com ou sem *prefetching*.

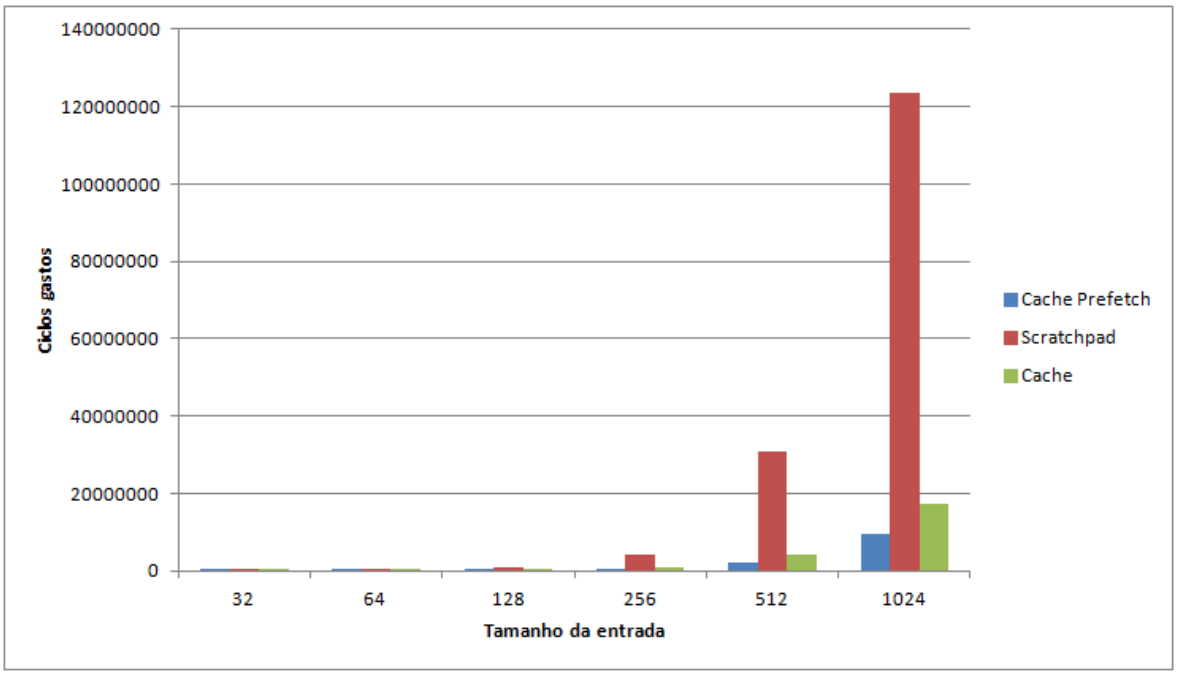

Figura 4.2: Gráfico comparativo do desempenho obtido pelos algoritmos para adição de matrizes com e sem a memória *scratchpad*.

O comportamento do algoritmo para cálculo do histograma é bastante similar ao da adição de matrizes. O acesso à matriz que contém as informações da imagem é bastante linear e embora o acesso ao vetor que armazena o histograma possa apresentar pouca linearidade, ele não chega a ser um limitador de desempenho por seu tamanho reduzido. Quando a entrada aumenta, a imagem torna-se maior e portanto a matriz aumenta, mas o vetor do histograma permanece com tamanho constante. Como o acesso a essa matriz é extremamente linear, o desempenho obtido com o emprego de *prefetching* apresentam bons resultados. Quando comparados a abordagem da *cache* com *prefetching* e da memória *scratchpad*, nota-se que a *cache* é 9% melhor. Isso pode ser atribuído ao fato de que além de competir com o alto nível de localidade do algoritmo, a versão com memória *scratchpad* também precisa amortizar as instruções que são adicionadas para realizar sua gerência. O gráfico da Figura 4.3 compara o desempenho da *cache*, da *cache* com *prefetching* habilitado e da memória *scratchpad*. Nele observa-se que dada a alta localidade espacial do problema, a versão da *cache* com

prefetching apresenta um desempenho superior ao apresentado pela *scratchpad*. Por outro lado, a *cache* sem *prefetching* possui um desempenho inferior e, além disso, a solução empregada pela *scratchpad* utiliza um espaço relativamente pequeno da memória *on-chip*. A execução do algoritmo na abordagem tradicional vai povoando todas as posições da memória rápida, em contrapartida para uma solução paralela um controle bem definido do espaço utilizado por cada *thread* pode ser mais relevante do que a diferença de eficiência reportada, pois o comportamento da *cache*, que não pode ser definido pela aplicação, poderia acarretar em um alto número de conflitos nos acessos paralelos.

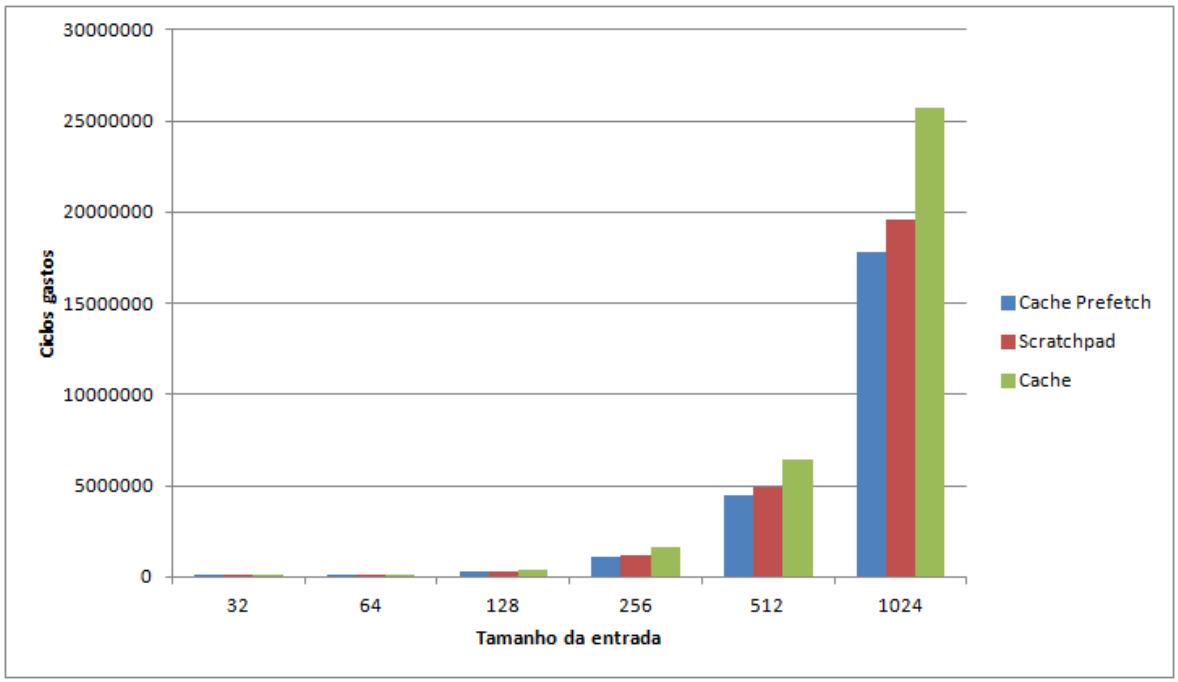

Figura 4.3: Gráfico comparativo do desempenho obtido pelos algoritmos para cálculo de histogramas com e sem a memória *scratchpad*.

Para o algoritmo de Dijkstra o cenário é diferente. Embora o grafo tenha sido representado como uma matriz e as linhas dessa matriz sejam inteiramente percorridas a cada iteração, o caminhamento geral do grafo não possui uma alta localidade espacial. A cada iteração, o algoritmo busca o nó de menor distância da lista de nós pendentes e neste caso, não necessariamente ele é o nó referente a próxima linha da matriz. Assim quanto menos linear for este conjunto de acessos, pior será o desempenho da *cache*. Para esses casos utilizando um *prefetching* por *software* que busca antecipar essa necessidade obteve-se os resultados mostrados na Tabela 4.13 e sumarizados no gráfico da Figura 4.4. Com a solução proposta obteve-se um desempenho até 17% melhor para as entradas avaliadas. Um limitador para obter-se uma solução mais eficiente com a

memória *scratchpad* pode ser visto no trecho de Código 4.9. Entre a chamada para *SPLOAD* que carrega os dados na linha 1c8 e a seu primeiro uso na linha 1f8 há poucas instruções, e portanto, não há código o suficiente para consumir o tempo necessário para que a conclusão do carregamento inicial, obrigando o processador a realizar um *stall*. Por outro lado, para entradas maiores um laço mais extenso é realizado no carregamento, implicando em mais tempo entre o carregamento inicial e seu primeiro uso, o que eventualmente elimina essa espera. Isso pode ser utilizado para justificar o melhor desempenho para as instâncias maiores analisadas.

Código 4.9: Trecho de código assembly com endereços em hexadecimal gerado pela compilação do algoritmo de Dijkstra.

```
1c8:    27f000f1        <spload>
1cc:    e28aa010        add     sl, sl, #16
1d0:    e2866040        add     r6, r6, #64     ; 0x40
1d4:    e2855040        add     r5, r5, #64     ; 0x40
1d8:    e35a0b01        cmp     sl, #1024       ; 0x400
1dc:    bafffff7        blt     1c0 <main+0x1c0>
1e0:    e59d1004        ldr     r1, [sp, #4]
1e4:    e3a00000        mov     r0, #0
1e8:    e3a03a01        mov     r3, #4096       ; 0x1000
1ec:    e7810109        str     r0, [r1, r9, lsl #2]
1f0:    e0841100        add     r1, r4, r0, lsl #2
1f4:    e7911008        ldr     r1, [r1, r8]
1f8:    e3510000        cmp     r1, #0
1fc:    0a000006        beq     21c <main+0x21c>
200:    e7942109        ldr     r2, [r4, r9, lsl #2]
204:    e0821001        add     r1, r2, r1
208:    e7942100        ldr     r2, [r4, r0, lsl #2]
20c:    e1510002        cmp     r1, r2
210:    b1a02004        movlt   r2, r4
214:    b7a21100        strlt   r1, [r2, r0, lsl #2]!
218:    b7829003        strlt   r9, [r2, r3]
21c:    e2800001        add     r0, r0, #1
220:    e3500b01        cmp     r0, #1024       ; 0x400
224:    1affffff        bne     1f0 <main+0x1f0>
228:    e59d0008        ldr     r0, [sp, #8]
...
```

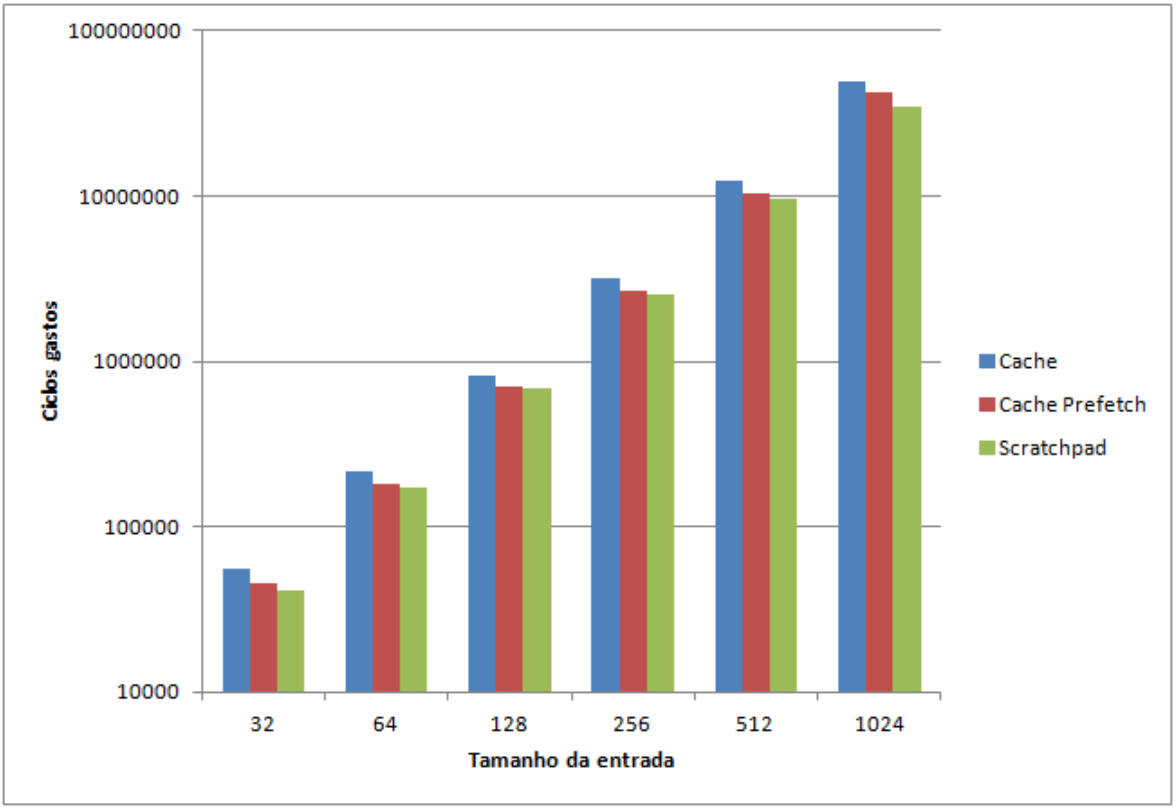

Figura 4.4: Gráfico comparativo do desempenho obtido pelos algoritmos de Dijkstra com e sem a memória *scratchpad*.

Como mostrado na Tabela 4.18 e no gráfico da Figura 4.5, a versão proposta do algoritmo quicksort também apresentou um desempenho até 17% melhor utilizando a memória *scratchpad*. Embora os dados a serem ordenados sejam representados por um vetor, a medida que o tamanho da entrada cresce e o número de chamadas recursivas aumenta, a previsibilidade necessária para o uso da localidade torna-se mais complexo e portanto, difícil de realizar o *prefetching*. Além disso, a solução empregada para este caso foi de colocar todo o vetor a ser ordenado dentro da memória *scratchpad*. Inicialmente os dados são copiados a partir da memória principal e a seguir são utilizados sempre a partir da memória *scratchpad*, resultando em um desempenho melhor por eliminar as falhas. Naturalmente, esta não é uma solução geral, pois para casos muito grandes ela não poderia ser empregada. Para que ela seja funcional todos os dados do vetor a ser ordenado precisam caber na memória *scratchpad*.

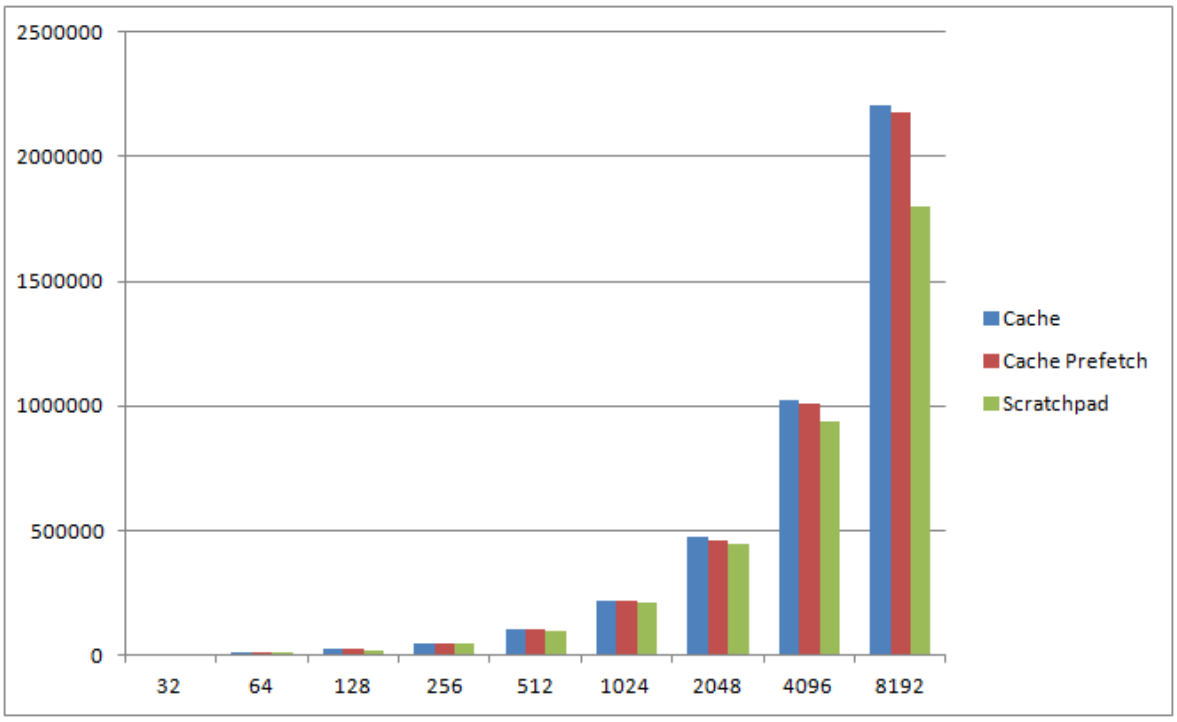

Figura 4.5: Gráfico comparativo do desempenho obtido pelas versões do algoritmo quicksort com e sem a memória *scratchpad*.

4.4 Conclusão

Este capítulo apresentou os experimentos realizados e os resultados obtidos por eles. Um *microbenchmark* com quatro algoritmos foi avaliado, mostrando que aplicações com alta localidade espacial tendem a funcionar melhor na *cache*. Por se tratar de um mecanismo gerenciado por *hardware* a *cache* ofereceu um desempenho melhor do que a alternativa oferecida por *software* que implica em uma sobrecarga maior. A *scratchpad* mostrou uma solução até 17% melhor para problemas onde a localidade não era tão evidente. Nesses casos as técnicas de *prefetching* não conseguiram prever quais blocos deveriam ser disponibilizados com antecedência na memória *on-chip*. A medida que o tamanho das entradas aumentou, a incapacidade da *cache* de antecipar a necessidade das aplicações tornou-se mais evidente. Finalmente, como os problemas analisados eram propositalmente pequenos, espera-se que em casos mais complexos resultados ainda melhores possam ser alcançados, pois neles o volume de dados pode ser maior e ao mesmo tempo o padrão de acesso ser complexo demais para obter um bom resultado com as técnicas baseadas nos princípios de localidade (Williams et al., 2007).

Capítulo 5

Conclusão

A proposta deste trabalho de pesquisa era prover uma arquitetura de propósito geral com a gestão da memória *on-chip* mista, tanto realizada por *hardware* quanto por *software*. O processo de desenvolvimento do projeto de pesquisa como um todo envolveu as áreas de *software* e *hardware*, indo da definição de uma *interface* de alto nível para os desenvolvedores, passando pelo processo de compilação, até níveis mais baixos como o conjunto de instruções e as alterações a serem feitas no *hardware*. O desenvolvimento iniciou-se com a seleção da arquitetura e do simulador para permitir a verificação da proposta e a execução de alguns experimentos para avaliar possíveis vantagens nesta abordagem. O simulador selecionado não possuía a hierarquia de memória, por isso ela precisou ser inteiramente codificada. Como a implementação foi realizada pela simulação idêntica do comportamento da hierarquia de memória com a *cache* e com a memória principal, neste ponto algumas dificuldades foram encontradas, pois o processo de depuração tornou-se bastante complicado. Embora aplicações pequenas tenham sido utilizadas, calcular a linha da *cache* e o bloco relacionado a diferentes endereços não foi um processo simples. Vencida esta etapa, foi realizada a alteração no *hardware* para incluir a memória *scratchpad* e as instruções para gerenciá-la. De modo geral, como a alteração proposta é simples, essa inclusão não apresentou grandes dificuldades. A seguir, definiu-se a biblioteca de alto nível para permitir a utilização viável da memória *scratchpad*. Mesmo tendo consumido mais tempo, principalmente devido a interação com o compilador, esta etapa do trabalho também não apresentou grande dificuldade, destacando-se apenas o tratamento dado aos ponteiros. Devido a precedência da linguagem C e a forma como as funções da biblioteca foram definidas, foi necessário tomar certos cuidados na utilização dos endereços alocados para a memória *scratchpad* de forma que o índice dos arranjos considerassem seu tipo de dados. Por exemplo, uma posição de um vetor de inteiros corresponde a 4 *bytes*, mas o seu uso

inadequado pode gerar um endereço desalinhado.

Para avaliar o resultado obtido, selecionaram-se experimentos representativos com oportunidades de otimização para a *scratchpad*, mas que fossem menores, pois o uso da gestão de memória por *software* agrega bastante complexidade ao código. Algoritmos tradicionais da computação com muita localidade espacial e com um padrão de acesso simples ao dados foram escolhidos, reescritos e suas execuções mostraram que eles até podem se beneficiar da memória *scratchpad*, mas conforme esparado, em uma arquitetura com *prefetching* essa vantagem é inexistente, pois a alternativa gerenciada pelo *hardware* obtém mais eficiência com menos complexidade para o desenvolvedor. Embora para esses casos a solução proposta utilizando a memória *scratchpad* tenha apresentado um resultado bastante inferior, isso não significa que trata-se de um resultado definitivo, pois abordagens mais eficientes podem ser possíveis e de modo geral a *scratchpad* utiliza um espaço muito menor da memória *on-chip*, o que poderia beneficiar soluções paralelas. Por outro lado, quando o padrão de acesso aos dados mostrou-se mais complexo, o *hardware* não foi capaz de identificar um padrão de utilização que permitisse a realização eficiente do *prefetching*. Nesses casos a proposta apresentou um desempenho até 17% superior quando comparado a abordagem tradicional da *cache*. Porém, como já discutido, aplicações simples foram selecionadas para a verificação e avaliação do trabalho, mas resultados ainda mais expressivos podem ser obtidos neste cenário ou mesmo para as aplicações do *benchmark* utilizado se soluções mais efetivas forem propostas.

Embora exigências rígidas de eficiência não componham os requisitos de todas as aplicações, a abordagem apresentada permite que todos os tipos de aplicações sejam atendidos por meio de uma proposta transparente. Aquelas que têm essa demanda e possuem um padrão complexo de acesso aos dados podem se beneficiar de maneira expressiva e aquelas em que isso não é importante ou não é viável, continuam funcionando como se o *hardware* não tivesse qualquer alteração. Dessa forma, este projeto de pesquisa apresentou resultados satisfatórios e a solução proposta mostrou-se viável, mas seu principal resultado é sua capacidade de propiciar desempenho adicional para algumas classes de aplicações usando uma solução simples e transparente.

As principais contribuições oriundas desse projeto compreendem um simulador de *hardware* que provê a ideia da arquitetura mista com memória *scratchpad* e a *cache*, e uma memória *scratchpad* que permite a gerência da memória *on-chip* por *software* utilizando uma abordagem transparente, permitindo seu uso quando ele é benéfico, sem gerar qualquer ruído quando ele não é.

5.1 Trabalhos Futuros

Como trabalhos futuros destacam-se os itens discutidos a seguir. Seria interessante que o compilador fosse capaz de realizar otimizações para memória *scratchpad*, de modo que a complexidade do uso da gestão da memória por *software* fosse deslocada para o compilador, tornando o uso da *scratchpad* mais simples e mediante a obtenção de resultados satisfatórios, mais interessante. Para a simplificação deste projeto, a biblioteca de alto nível provida realiza a alocação baseada em endereços, mas seria interessante que o compilador fosse capaz de realizar a alocação para diferentes estruturas conforme seu tipo de dados. Outro trabalho possível seria a melhoria da gestão da memória por *software* via algum tipo de *prefetching* associado a instrução *SPLOAD*. Isso poderia ser feito trazendo os dados de modo especulativo para a *cache* de Nível 2 da hierarquia, de tal forma que as próximas transferências para a memória *scratchpad* poderiam se originar a partir de níveis mais rápidos da hierarquia de memória. Finalmente, um estudo mais profundo de aplicações mais complexas poderia ser conduzido para avaliar quais resultados seriam obtidos ao empregar a gestão da memória por *software*.

Apêndice A

Código Fonte do Simulador

Este apêndice apresenta os principais arquivos alterados do simulador (Serrano et al., 2007) utilizado neste projeto de pesquisa. Embora outros arquivos também tenham sido alterados, eles não são apresentados neste documento devido ao seu alto número de linhas de código.

A.1 Implementação da *Cache*

Os arquivos discutidos nesta seção apresentam a codificação realizada para a *cache*. Como a memória *scratchpad* foi projetada como uma modificação de parte do *hardware* da *cache*, a maior parte da sua implementação pode ser vista nas estruturas de dados a seguir.

Código A.1: Arquivo ARM9Cache.h

```
1  #ifndef ARM9CacheH
2  #define ARM9CacheH
3
4  #include "Utils.h"
5  #include "ARM9RutinasARM.h"
6  #include <stdio.h>
7
8  #define MEMORY 100
9  #define INST_CACHE 101
10 #define DATA_CACHE 201
11 #define DATA_CACHE_2 202
12
```

```
13 class ARM9Data {
14 public:
15   bool v;
16   bool spm;
17   uint32 tag;
18   uint32 lru;
19   uint8 *data;
20
21   // Define se e originado de prefetching
22   bool fromPrefetching;
23 };
24
25 typedef struct {
26   uint32 address;
27
28   uint32 blockOffset;
29   uint32 index;
30   uint32 tag;
31 } ARM9Address;
32
33 class ARM9Cache {
34 protected:
35   // Cache size in bytes
36   uint32 cacheSizeBytes;
37   uint32 cacheSizeBytesAssociativity;
38
39   ARM9Data **data;
40
41   // Block offset
42   uint32 blockOffsetStart, blockOffsetEnd;
43   uint32 blockSizeBytes;
44
45   // Index
46   uint32 indexStart, indexEnd;
47
48   // Tag
```

```
49    uint32 tagStart, tagEnd;
50
51    // Number of pages for associativity
52    uint32 associativitySize;
53
54    // Cache delay
55    uint32 cacheDelay;
56
57    // Cache next level
58    ARM9Cache *nextLevel;
59
60    // Just for control the log
61    ARM9RutinasARM *rutinas = NULL;
62
63    uint32 lruCounter;
64
65    ARM9Address getAddress(uint32 address);
66  public:
67    int id;
68
69    // Constructor
70    ARM9Cache(uint32 cacheSizeBytes,
71      uint32 blockSizeBytes,
72      uint32 associativitySize,
73      uint32 cacheDelay,
74      ARM9Cache* nextLevel,
75      ARM9RutinasARM* rut);
76
77    ~ARM9Cache();
78
79    virtual uint32 readMemoriaWord(uint32 address, uint32 *delay);
80    virtual uint32 readMemoriaWord(uint32 address, uint32 *delay,
81                                   bool ignoreSPM, bool doPrefetch,
82                                   bool prefetching);
83
84    virtual void writeMemoriaWord(uint32 address,
```

```
85                              uint32 value, uint32 *delay);
86
87    uint32 getReplaceBlockIndex(ARM9Address address);
88    void replaceContent(ARM9Address address, uint32 replace,
89                              uint32 *delay, bool prefetching);
90
91    uint32 spAlloc(uint32 address, uint32 *delay);
92    void spFree(uint32 address);
93    void spLoadMem(uint32 memAddress, uint32 spAddress,
94                              uint32 *delay);
95    void spStoreMem(uint32 memAddress, uint32 spAddress,
96                              uint32 *delay);
97
98    bool equivalentAddress(uint32 address1, uint32 address2);
99
100   void logEvent(ARM9Address ad, char *type);
101
102   virtual void clearLocality(void);
103
104   uint32 getBlockSizeBytes(void){
105     return blockSizeBytes;
106   };
107 };
108
109 #endif
```

Código A.2: Arquivo ARM9Cache.cpp

```
1
2 #include "ARM9Cache.h"
3 #include "Decls.h"
4 #include <limits.h>
5
6 // Constructor
7 ARM9Cache::ARM9Cache(uint32 cacheSizeBytes,
8             uint32 blockSizeBytes,
9             uint32 associativitySize,
10            uint32 cacheDelay,
```

```
11              ARM9Cache* nextLevel,
12              ARM9RutinasARM* rut){
13    // Raise a exception if nwords is not a power of two
14    if (!((cacheSizeBytes != 0)
15       && !(cacheSizeBytes & (cacheSizeBytes - 1))))
16      throw "The cache size should be a power of two";
17    if (!((blockSizeBytes != 0)
18       && !(blockSizeBytes & (blockSizeBytes - 1))))
19      throw "The block size should be a power of two";
20    if ((associativitySize < 1)
21       && !(associativitySize & (associativitySize - 1)))
22      throw "The associativity size should be
23 ____________greater than zero and a power of two";
24
25    this->cacheSizeBytes = cacheSizeBytes;
26    this->associativitySize = associativitySize;
27    this->cacheSizeBytesAssociativity =
28              cacheSizeBytes / associativitySize;
29    this->blockSizeBytes = blockSizeBytes;
30
31    //
32    // Calcular o offset do bloco
33    //
34    uint32 blockSizeWords = blockSizeBytes / 4;
35    this->blockOffsetStart = 2;
36    // Calcular o tamanho do offset
37    uint32 blockOffsetSize = 1;
38    while ((blockSizeWords >> blockOffsetSize) > 1)
39      blockOffsetSize++;
40    this->blockOffsetEnd =
41       this->blockOffsetStart + blockOffsetSize - 1;
42
43    //
44    // Calcular o index
45    //
46    // Numero de linhas por associatividade
```

```
47    uint32 lines = cacheSizeBytesAssociativity / blockSizeBytes;
48    // O index comeca no bit que segue o ultimo do bloco
49    this->indexStart = this->blockOffsetEnd + 1;
50    uint32 tmpLines = lines;
51    uint32 indexSize = 1;
52    while ((tmpLines >> indexSize) > 1)
53      indexSize++;
54    this->indexEnd = this->indexStart + indexSize - 1;
55
56    //
57    // Tag
58    //
59    this->tagStart = this->indexEnd + 1;
60    this->tagEnd = 31;
61
62    // Alocar a associatividade
63    data = new ARM9Data*[associativitySize];
64    // Inicializar a associatividade
65    for (uint32 i = 0; i < associativitySize; i++){
66      data[i] = new ARM9Data[lines];
67      for (uint32 j = 0; j < lines; j++){
68        data[i][j].v = false;
69        data[i][j].spm = false;
70        data[i][j].lru = 0;
71        data[i][j].data = new uint8[blockSizeBytes];
72      }
73    }
74
75    this->cacheDelay = cacheDelay;
76
77    // Next cache level
78    this->nextLevel = nextLevel;
79
80    this->lruCounter = 0;
81
82    this->rutinas = rut;
```

```
83 }
84
85 ARM9Cache::~ARM9Cache(){
86   for (uint32 i = 0; i < associativitySize; i++){
87     delete[] data[i];
88   }
89   delete[] data;
90 }
91
92 ARM9Address ARM9Cache::getAddress(uint32 address){
93   ARM9Address ret;
94
95   ret.address = address;
96
97   ret.blockOffset =
98     mid(address, this->blockOffsetStart, this->blockOffsetEnd);
99   ret.index = mid(address, this->indexStart, this->indexEnd);
100  ret.tag = mid(address, this->tagStart, this->tagEnd);
101
102  return ret;
103 }
104
105 uint32 ARM9Cache::getReplaceBlockIndex(ARM9Address address){
106   const uint32 limite = 0xffffff;
107   uint32 replace = limite;
108   uint32 older = INT_MAX;
109   for (uint32 i = 0; i < associativitySize; i++){
110     // If this space is still blank, use it
111     if (! data[i][address.index].v){
112       replace = i;
113       break;
114     }
115     // Choose the older one (can not be a spm)
116     else if (data[i][address.index].lru < older
117              && ! data[i][address.index].spm){
118       older = data[i][address.index].lru;
```

```
119         replace = i;
120       }
121     }
122
123     if (replace >= limite)
124       throw "Could not get a address to be replaced";
125
126     return replace;
127   }
128
129   void ARM9Cache::replaceContent(ARM9Address address,
130               uint32 replace, uint32 *delay, bool prefetching){
131     // This should be executed only if it is a fault
132     uint32 pos = 0;
133     // Used to consider delay of consecutive calls of
134     // readMemoriaWord for emulation purposes correctly
135     uint32 ignore = 0;
136     bool first = true;
137
138     // First address to be replaced.
139     // It should replace from the address base,
140     // passing by the address parameter and ending
141     // with the last address of block
142     int baseAddressId =
143       (int)(address.address / blockSizeBytes) * blockSizeBytes;
144     ARM9Address baseAddress = getAddress(baseAddressId);
145
146     // Bring the block
147     for (uint32 i = 0; i < blockSizeBytes; i+=4){
148
149       if (data[replace][baseAddress.index].v){
150         uint32 oldAddress =
151           data[replace][baseAddress.index].tag << (this->indexEnd + 1) |
152           baseAddress.index << (this->blockOffsetEnd + 1) |
153           (baseAddress.blockOffset + pos) << 2;
154
```

```
155         nextLevel->writeMemoriaWord(oldAddress,
156           *((uint32*)(data[replace][baseAddress.index].data
157             + (baseAddress.blockOffset + pos)*4)),
158           &ignore);
159         ignore = 0;
160       }
161
162       uint32 value =
163         nextLevel->readMemoriaWord(baseAddress.address + i, &ignore);
164       *((uint32*)(
165         data[replace][baseAddress.index].data
166         + (baseAddress.blockOffset
167         + pos)*4)) = value;
168
169       pos++;
170
171       if (first){
172         first = false;
173         *delay += ignore;
174       }
175     }
176
177     // The entry has a valid content
178     data[replace][baseAddress.index].v = true;
179     data[replace][baseAddress.index].tag = baseAddress.tag;
180     data[replace][baseAddress.index].fromPrefetching = prefetching;
181 }
182
183 void ARM9Cache::logEvent(ARM9Address ad, char *type){
184     // Registrar a falha
185     if (rutinas != NULL && rutinas->logging()){
186       int baseAddressId =
187         (int)(ad.address / blockSizeBytes) * blockSizeBytes;
188       FILE* misses = fopen("Misses.txt","a");
189       if (misses != NULL){
190         fprintf(misses,
```

```
191           "Id: %d, Address: %d, Block: %d, Index: %d, Tag: %d, %s\n",
192           this->id,
193           ad.address,
194           ad.blockOffset,
195           ad.index,
196           ad.tag,
197           type);
198         fclose(misses);
199       }
200     }
201 }
202
203 uint32 ARM9Cache::readMemoriaWord(uint32 address, uint32 *delay,
204   bool ignoreSPM, bool doPrefetch, bool prefetching){
205
206   ARM9Address ad = getAddress(address);
207
208   // Patterson p361
209   // hit?
210   for (uint32 i = 0; i < associativitySize; i++){
211     if (data[i][ad.index].v && data[i][ad.index].tag == ad.tag){
212
213       if (ignoreSPM && data[i][ad.index].spm)
214         continue;
215
216       *delay += cacheDelay;
217       data[i][ad.index].lru = lruCounter++;
218
219       // Se tiver origem de prefetching, carregar o proximo
220       // bloco especulando
221       if (data[i][ad.index].fromPrefetching && !prefetching
222           && id == DATA_CACHE){
223         uint32 prefetchDelay = 0;
224         uint32 prefAddress = address + blockSizeBytes + 4;
225         readMemoriaWord(prefAddress, &prefetchDelay, false,
226           false, true);
```

```
227           // Desconsidera que e de prefetching porque ja carregou
228           // o proximo bloco
229           data[i][ad.index].fromPrefetching = false;
230
231           // Adicionar o delay na lista de delays
232           if (rutinas != NULL)
233             rutinas->addDelay(-1, prefAddress, prefetchDelay);
234         }
235
236         if (! prefetching)
237           logEvent(ad, "hit");
238         return *((uint32*)(data[i][ad.index].data + ad.blockOffset*4));
239       }
240     }
241
242     *delay += this->cacheDelay;
243
244     // Who should be replaced? Choose the index using LRU
245     uint32 replace = getReplaceBlockIndex(ad);
246     replaceContent(ad, replace, delay, prefetching);
247
248     // Se estiver fazendo prefetching o bloco trazido deve ser
249     // despejado primeiro
250     uint32 lru;
251     if (prefetching){
252       lru = lruCounter - associativitySize;
253       if (lru < 0)
254         lru = 0;
255     }
256     else
257       lru = lruCounter++;
258
259     data[replace][ad.index].lru = lru;
260
261     // Registrar a falha
262     if (!prefetching)
```

```
263       logEvent(ad, "miss");
264
265     // Realizar o prefetch?
266     if (doPrefetch){
267       uint32 prefetchDelay = 0;
268       uint32 prefAddress = address + blockSizeBytes + 4;
269       readMemoriaWord(prefAddress, &prefetchDelay, false, false, true);
270
271       // Adicionar o delay na lista de delays
272       if (rutinas != NULL && id == DATA_CACHE)
273         rutinas->addDelay(-1, prefAddress, prefetchDelay);
274     }
275
276     return *((uint32*)(data[replace][ad.index].data + ad.blockOffset*4));
277   }
278
279   uint32 ARM9Cache::readMemoriaWord(uint32 address, uint32 *delay){
280     return readMemoriaWord(address, delay, false, prefetch, false);
281   }
282
283   void ARM9Cache::spFree(uint32 address){
284     ARM9Address adSp = getAddress(address);
285
286     // Obter o buffer associado a scratchpad e desaloca-lo
287     for (uint32 i = 0; i < associativitySize; i++){
288       if (data[i][adSp.index].spm){
289         data[i][adSp.index].spm = false;
290         data[i][adSp.index].v = false;
291         break;
292       }
293     }
294   }
295
296   uint32 ARM9Cache::spAlloc(uint32 address, uint32 *delay){
297
298     address = address << 2;
```

```
299
300   ARM9Address ad = getAddress(address);
301   uint32 replace = getReplaceBlockIndex(ad);
302
303   int baseAddressId = (int)(address / blockSizeBytes)
304                             * blockSizeBytes;
305   ARM9Address baseAddress = getAddress(baseAddressId);
306
307   uint32 ignore = 0;
308   bool first = true;
309   uint32 pos = 0;
310
311   // Se o conteudo da cache for valido ele precisa ser salvo
312   // antes de ser alocado para a SPM
313   for (uint32 i = 0; i < blockSizeBytes; i+=4){
314     if (data[replace][baseAddress.index].v){
315       uint32 oldAddress =
316         data[replace][baseAddress.index].tag << (this->indexEnd + 1) |
317         baseAddress.index << (this->blockOffsetEnd + 1) |
318         (baseAddress.blockOffset + pos) << 2;
319
320       nextLevel->writeMemoriaWord(oldAddress,
321         *((uint32*)(data[replace][baseAddress.index].data
322           + (baseAddress.blockOffset + pos)*4)),
323         &ignore);
324       ignore = 0;
325     }
326
327     pos++;
328
329     if (first){
330       first = false;
331       *delay += ignore;
332     }
333   }
334
```

```
335   // Alocado como SPM
336   data[replace][baseAddress.index].spm = true;
337   // Marcar os dados como validos para uso pelo ldr
338   // e str tradicionais
339   data[replace][baseAddress.index].v = true;
340   data[replace][baseAddress.index].tag = baseAddress.tag;
341
342   return address;
343 }
344
345 void ARM9Cache::spStoreMem(uint32 memAddress,
346                            uint32 spAddress, uint32 *delay){
347   spAddress = spAddress << 2;
348
349   ARM9Address adSp = getAddress(spAddress);
350
351   int baseSpAddress = (int)(spAddress / blockSizeBytes)
352                             * blockSizeBytes;
353   ARM9Address adSpBase = getAddress(baseSpAddress);
354
355   int baseMemAddress = (int)(memAddress / blockSizeBytes)
356                              * blockSizeBytes;
357
358   uint32 spmIndex = associativitySize;
359   // Obter o buffer associado a scratchpad
360   for (uint32 i = 0; i < associativitySize; i++){
361     if (data[i][adSp.index].spm){
362       spmIndex = i;
363     }
364   }
365
366   uint32 ignore = 0;
367   bool first = true;
368   uint32 pos = 0;
369
370   // Could not find the place to put the value in this level
```

```
371     *delay += cacheDelay;
372     // Bring the block
373     for (uint32 i = 0; i < blockSizeBytes; i+=4){
374
375       uint32 value = *((uint32*)(data[spmIndex][adSpBase.index].data
376        + (adSpBase.blockOffset + pos)*4));
377       nextLevel->writeMemoriaWord(baseMemAddress + i, value, delay);
378
379       pos++;
380
381       if (first){
382         first = false;
383         *delay += ignore;
384       }
385     }
386 }
387
388 void ARM9Cache::spLoadMem(uint32 memAddress,
389                           uint32 spAddress, uint32 *delay){
390
391     ARM9Address adMem = getAddress(memAddress);
392     ARM9Address adSp = getAddress(spAddress);
393
394     int baseSpAddress = (int)(spAddress / blockSizeBytes)
395                               * blockSizeBytes;
396     ARM9Address adSpBase = getAddress(baseSpAddress);
397
398     uint32 spmIndex = associativitySize;
399     // Obter o buffer associado a scratchpad
400     for (uint32 i = 0; i < associativitySize; i++){
401       if (data[i][adSp.index].spm){
402         spmIndex = i;
403         break;
404       }
405     }
406
```

```
407     // Nao ha bloco alocado para a SPM
408     if (spmIndex == associativitySize)
409       throw "There is not an allocated SPM buffer ";
410
411     // Substituir o bloco da SPM pelo bloco da memoria
412     uint32 ignore = 0;
413     bool first = true;
414     uint32 pos = 0;
415
416     // Bring the block
417     for (uint32 i = 0; i < blockSizeBytes; i+=4){
418
419       uint32 value = readMemoriaWord(adMem.address + i, &ignore,
420                                      true, false, false);
421
422       *((uint32*)(data[spmIndex][adSpBase.index].data
423                   + (adSpBase.blockOffset + pos)*4)) = value;
424
425       pos++;
426
427       if (first){
428         first = false;
429         *delay += ignore;
430       }
431     }
432 }
433
434 bool ARM9Cache::equivalentAddress(uint32 address1, uint32 address2){
435   ARM9Address ad1 = getAddress(address1);
436   ARM9Address ad2 = getAddress(address2);
437
438   // O endereco e dito equivalente se seria armazenado no
439   // mesmo bloco da cache dado outro endereco, ie
440   // pertence a mesma linha e tem a mesma tag
441   return (ad1.index == ad2.index && ad1.tag == ad2.tag);
442 }
```

```
443
444 void ARM9Cache::clearLocality(void){
445   uint32 lines = cacheSizeBytesAssociativity / blockSizeBytes;
446
447   for (uint32 i = 0; i < associativitySize; i++){
448     for (uint32 j = 0; j < lines; j++){
449       if (!data[i][j].spm)
450         data[i][j].v = false;
451       data[i][j].lru = 0;
452     }
453   }
454
455   nextLevel->clearLocality();
456 }
457
458 void ARM9Cache::writeMemoriaWord(uint32 address, uint32 value,
459                                  uint32 *delay){
460   ARM9Address ad = getAddress(address);
461
462   uint32 ignore = 0;
463   // Patterson p361
464   // hit?
465   for (uint32 i = 0; i < associativitySize; i++){
466     if (data[i][ad.index].v && data[i][ad.index].tag == ad.tag){
467       *delay += cacheDelay;
468       data[i][ad.index].lru = lruCounter++;
469       *((uint32*)(data[i][ad.index].data + ad.blockOffset*4)) = value;
470       nextLevel->writeMemoriaWord(address, value, &ignore);
471       return;
472     }
473   }
474
475   // Could not find the place to put the value in this level
476   *delay += cacheDelay;
477   nextLevel->writeMemoriaWord(address,value,delay);
478 }
```

A.2 Implementação das Instruções

As estruturas de dados a seguir estão relacionadas a simulação das instruções. Nestes arquivos encontra-se o código das instruções criadas para permitir entrada e saída de dados, produzir a parada, a realização de *log* e as instruções relacionadas a *scratchpad*.

Código A.3: Arquivo ARM9Rutinas.h

```
1  #include "ARM9RutinasARM.h"
2  #include "ARM9Core.h"
3  #include <stdio.h>
4
5  ARM9RutinasARM::ARM9RutinasARM () {
6
7    doLog = false;
8    tablaGenerada = 0;
9
10   int aux[] = {0x0F0F, 0xF0F0, 0x3333, 0xCCCC, 0x00FF, 0xFF00,
11         0x5555, 0xAAAA, 0x3030, 0xCFCF, 0xAA55, 0x55AA,
12         0xA050, 0x5FAF, 0xFFFF, 0xFFFF};
13
14
15   //Genera la tabla precalculada de condiciones
16   for (int i = 0; i < 16; i++) {
17
18     tablaCond[i][0] = (aux[i] & 0x8000) != 0;
19     tablaCond[i][1] = (aux[i] & 0x4000) != 0;
20     tablaCond[i][2] = (aux[i] & 0x2000) != 0;
21     tablaCond[i][3] = (aux[i] & 0x1000) != 0;
22
23     tablaCond[i][4] = (aux[i] & 0x0800) != 0;
24     tablaCond[i][5] = (aux[i] & 0x0400) != 0;
25     tablaCond[i][6] = (aux[i] & 0x0200) != 0;
26     tablaCond[i][7] = (aux[i] & 0x0100) != 0;
27
28     tablaCond[i][8] = (aux[i] & 0x0080) != 0;
29     tablaCond[i][9] = (aux[i] & 0x0040) != 0;
30     tablaCond[i][10] = (aux[i] & 0x0020) != 0;
```

```
31        tablaCond[i][11] = (aux[i] & 0x0010) != 0;
32
33        tablaCond[i][12] = (aux[i] & 0x0008) != 0;
34        tablaCond[i][13] = (aux[i] & 0x0004) != 0;
35        tablaCond[i][14] = (aux[i] & 0x0002) != 0;
36        tablaCond[i][15] = (aux[i] & 0x0001) != 0;
37
38    }
39 }
40
41 void ARM9RutinasARM::openOutput(void){
42    fileOut = fopen("Out.txt","w");
43 }
44
45 void ARM9RutinasARM::openInput(void){
46    fileIn = fopen("In.txt","r");
47 }
48
49 void ARM9RutinasARM::closeOutput(void){
50    fclose(fileOut);
51 }
52
53 void ARM9RutinasARM::closeInput(void){
54    fclose(fileIn);
55 }
56
57 void ARM9RutinasARM::writeInt(void){
58    fprintf(fileOut, "%d ", this->contexto->bancoRegistros[2]);
59 }
60
61 void ARM9RutinasARM::readInt(void){
62    int valor;
63    fscanf(fileIn, "%d", &valor);
64
65    // Save the value in address of r2
66    uint32 memoryDelay = 0;
```

```
67    uint32 dirEfectiva = this->contexto->bancoRegistros[2].u_word
68                         & 0xFFFFFFFC;
69    this->core->writeMemoriaWord(valor, dirEfectiva, &memoryDelay);
70  }
71
72  void ARM9RutinasARM::writeFloat(void){
73    fprintf(fileOut, "%f ", this->contexto->bancoRegistros[0]);
74  }
75
76  void ARM9RutinasARM::readFloat(void){
77    fscanf(fileIn, "%f", &this->contexto->bancoRegistros[0]);
78  }
79
80  void ARM9RutinasARM::writeChar(void){
81    fprintf(fileOut, "%hhu ", this->contexto->bancoRegistros[0]);
82  }
83
84  void ARM9RutinasARM::readChar(void){
85    fscanf(fileIn, "%hhu", &this->contexto->bancoRegistros[0]);
86  }
87
88  void ARM9RutinasARM::addDelay(std::list<ARM9Delay> *delays,
89          uint32 reg, uint32 address, uint32 delay, uint32 pc){
90    ARM9Delay elem;
91    elem.address = address;
92    elem.delay = delay;
93    elem.reg = reg;
94    elem.pc = pc - 4;
95    delays->insert(delays->end(), elem);
96  }
97
98
99  bool removeDelay(const ARM9Delay& value){
100   return (value.delay <= 0);
101 }
102
```

```
103 void ARM9RutinasARM::updateDelays(std::list<ARM9Delay> *delays,
104                   uint32 value){
105   for (std::list<ARM9Delay>::iterator it = delays->begin();
106         it != delays->end(); it++){
107     (*it).delay -= value;
108   }
109 }
110
111 int32 ARM9RutinasARM::verifyHazards(std::list<ARM9Delay> *delays,
112                     ARM9Instruccion32 instr,
113                       int rn,
114                     int rs,
115                     int rm,
116                     uint32 dirEfectiva,
117                     uint32 dirEfectiva2)
118 {
119
120   bool hasAddress = false;
121   // Load/store immediate offset
122   if ((instr.u_word & 0x0E000000) == 0x04000000
123   || (instr.u_word & 0x0E000010) == 0x06000000
124   || (instr.u_word & 0x0FF000F0) == 0x07F000F0){
125     hasAddress = true;
126   }
127
128   ARM9Delay d;
129
130   for (std::list<ARM9Delay>::iterator it = delays->begin();
131           it != delays->end(); it++){
132     d = (*it);
133
134     int dBaseAddress = (int)(d.address /
135           core->getCache()->getBlockSizeBytes())
136           * core->getCache()->getBlockSizeBytes();
137     int dirEfecBaseAddress =
138           (int)(dirEfectiva / core->getCache()->getBlockSizeBytes()) *
```

```
139                 core->getCache()->getBlockSizeBytes();
140     int dirEfec2BaseAddress =
141           (int)(dirEfectiva2 / core->getCache()->getBlockSizeBytes()) *
142                 core->getCache()->getBlockSizeBytes();
143
144     // Verificar se e um dos enderecos passados
145     if (d.address >= 0 && hasAddress){
146       if (dBaseAddress == dirEfecBaseAddress ||
147         dBaseAddress == dirEfec2BaseAddress){
148           return d.delay;
149       }
150     }
151     else if (d.reg >= 0){
152       if (d.reg == rn ||
153         d.reg == rs ||
154         d.reg == rm){
155           return d.delay;
156       }
157     }
158   }
159
160   return 0;
161 }
162
163 void ARM9RutinasARM::toggleLog(void) {
164   doLog = !doLog;
165
166   if (doLog){
167     logFile = fopen("Log.txt", "a");
168     // Ao iniciar o log limpar a localidade
169     core->getCache()->clearLocality();
170   }
171   else
172     fclose(logFile);
173 }
174
```

```
175 bool ARM9RutinasARM::logging(void){
176   return doLog;
177 }
178
179 int ARM9RutinasARM::run (ARM9Core *core, ARM9Contexto *contexto,
180                          int nCiclos) {
181
182   ARM9Instruccion32 instruccion;
183   void *dir;
184
185   this->core = core;
186   this->contexto = contexto;
187
188   int rn, rd, rs, rm, rot, operando1, operando2, acumulador,
189       acumulador2, resultado;
190   int signo1, signo2;
191   int shift, tipoShift;
192   int l, s, c, shifterCarryOut, a, v, esDPI, esLSRO, dirOffset;
193   uint32 dirEfectiva, dirEfectiva2;
194   int u, b, w, i, j, esSignedHalfword, listaRegistros,
195       lsmUserMode, lsmCPSR;
196   int resultado2, esMISR, r, mascara;
197   int64 var1_64, var2_64; uint64 uvar1_64, uvar2_64;
198
199   uint32 memoryDelay, instrDelay;
200   std::list<ARM9Delay> prevDelays;
201   int32 stall;
202   bool stallProc;
203
204   int contador = 0;
205
206   int32 past;
207   int32 ciclosAntigo = nCiclos;
208
209   // Initialization
210   rn = -1;
```

```
211     rs = -1;
212     rm = -1;
213     dirEfectiva = -1;
214     dirEfectiva2 = -1;
215     instruccion.u_word = 0;
216
217 /*-------------------------*/
218 /*  Tabla de saltos
219 /*-------------------------*/
220 #include "ARM9TablaARM.h"
221
222 /*-------------------------*/
223 /*  Lectura de la siguiente instrucción
224 /*-------------------------*/
225
226 LecturaInstruccion:
227
228     rn = -1;
229     rs = -1;
230     rm = -1;
231
232     // Custom
233     if (instruccion.types.b4 == 1 &&
234       instruccion.types.shift_5_6 == 3 &&
235       instruccion.types.b7 == 1 &&
236       instruccion.types.s_20 == 1 &&
237       instruccion.types.opcode_21_24 == 0xf &&
238       instruccion.types.b25 == 1 &&
239       instruccion.types.type_26_27 == 1){
240     }
241     // BX
242     else if (instruccion.types.b4 == 1 &&
243       instruccion.types.shift_5_6 == 0 &&
244       instruccion.types.b7 == 0 &&
245       instruccion.types.rs_8_11 == 0xf &&
246       instruccion.types.rd_12_15 == 0xf &&
```

```
247       instruccion.types.rn_16_19 == 0xf &&
248       instruccion.types.s_20 == 0 &&
249       instruccion.types.opcode_21_24 == 9 &&
250       instruccion.types.b25 == 0 &&
251       instruccion.types.type_26_27 == 0){
252
253       rm = instruccion.types.rm_0_3;
254     }
255     // Data processing
256     else if (instruccion.types.type_26_27 == 0){
257
258       // mov e mvn so tem um operando
259       if (instruccion.types.opcode_21_24 != 0xD
260           && instruccion.types.opcode_21_24 != 0xF)
261         rn = instruccion.types.rn_16_19;
262       rd = instruccion.types.rd_12_15;
263
264       if (!instruccion.types.b25){
265         rm = instruccion.types.rm_0_3;
266
267         if (instruccion.types.b4 && ! instruccion.types.b7)
268           rs = instruccion.types.rs_8_11;
269       }
270     }
271     // PSR Transfer
272     else if (instruccion.types.opcode_21_24 & 0xC == 0x8 &&
273       instruccion.types.b25 == 0 &&
274       instruccion.types.type_26_27 == 0 &&
275       instruccion.types.rn_16_19 == 0xf &&
276       instruccion.types.s_20 == 0 &&
277       instruccion.types.opcode_21_24 & 1 == 0 &&
278       instruccion.types.rm_0_3 == 0 &&
279       instruccion.types.b4 == 0 &&
280       instruccion.types.shift_5_6 == 0 &&
281       instruccion.types.b7 == 0 &&
282       instruccion.types.rs_8_11 == 0){
```

```
283
284       rd = instruccion.types.rd_12_15;
285     }
286     else if (instruccion.types.opcode_21_24 & 0xC == 0x8 &&
287       instruccion.types.b25 == 0 &&
288       instruccion.types.type_26_27 == 0 &&
289       instruccion.types.rd_12_15 == 0xf &&
290       instruccion.types.rn_16_19 == 0x9 &&
291       instruccion.types.s_20 == 0 &&
292       instruccion.types.opcode_21_24 & 1 == 1 &&
293       instruccion.types.b4 == 0 &&
294       instruccion.types.shift_5_6 == 0 &&
295       instruccion.types.b7 == 0 &&
296       instruccion.types.rs_8_11 == 0){
297
298       rm = instruccion.types.rm_0_3;
299     }
300     else if (instruccion.types.opcode_21_24 & 0xC == 0x8 &&
301       instruccion.types.type_26_27 == 0 &&
302       instruccion.types.rd_12_15 == 0xf &&
303       instruccion.types.rn_16_19 == 0x8 &&
304       instruccion.types.s_20 == 0 &&
305       instruccion.types.opcode_21_24 & 1 == 1){
306
307       if (! instruccion.types.b25)
308         rm = instruccion.types.rm_0_3;
309     }
310     // Multiply
311     else if (instruccion.types.b4 == 1 &&
312       instruccion.types.shift_5_6 == 0 &&
313       instruccion.types.b7 == 1 &&
314       instruccion.types.opcode_21_24 & 0xE == 0 &&
315       instruccion.types.b25 == 0 &&
316       instruccion.types.type_26_27 == 0){
317
318       // E trocado
```

```
319       rd = instruccion.types.rn_16_19;
320       rn = instruccion.types.rd_12_15;
321       rs = instruccion.types.rs_8_11;
322       rm = instruccion.types.rm_0_3;
323     }
324     // Multiply long
325     else if (instruccion.types.b4 == 1 &&
326       instruccion.types.shift_5_6 == 0 &&
327       instruccion.types.b7 == 1 &&
328       instruccion.types.opcode_21_24 & 0xC == 4 &&
329       instruccion.types.b25 == 0 &&
330       instruccion.types.type_26_27 == 0){
331
332       // E trocado
333       rd = instruccion.types.rn_16_19;
334       // rn = instruccion.types.rd_12_15; rdLo
335       rs = instruccion.types.rs_8_11;
336       rm = instruccion.types.rm_0_3;
337     }
338     // Single data transfer
339     else if (instruccion.types.type_26_27 == 1){
340
341       // Load?
342       if (instruccion.types.s_20)
343         rd = instruccion.types.rd_12_15;
344       else
345         rs = instruccion.types.rd_12_15;
346       rn = instruccion.types.rn_16_19;
347
348       if (instruccion.types.b25)
349         rm = instruccion.types.rm_0_3;
350     }
351     // Halfword and signed data transfer
352     else if (instruccion.types.b4 == 1 &&
353       instruccion.types.b7 == 1 &&
354       instruccion.types.rs_8_11 == 0 &&
```

```
355       instruccion.types.opcode_21_24 & 2 == 0 &&
356       instruccion.types.b25 == 0 &&
357       instruccion.types.type_26_27 == 0){
358
359       // Load?
360       if (instruccion.types.s_20)
361         rd = instruccion.types.rd_12_15;
362       else
363         rs = instruccion.types.rd_12_15;
364       rn = instruccion.types.rn_16_19;
365       rm = instruccion.types.rm_0_3;
366     }
367     else if (instruccion.types.b4 == 1 &&
368       instruccion.types.b7 == 1 &&
369       instruccion.types.opcode_21_24 & 2 == 2 &&
370       instruccion.types.b25 == 0 &&
371       instruccion.types.type_26_27 == 0){
372
373       // Load?
374       if (instruccion.types.s_20)
375         rd = instruccion.types.rd_12_15;
376       else
377         rs = instruccion.types.rd_12_15;
378       rn = instruccion.types.rn_16_19;
379     }
380     // Block transfer LDM
381     else if (instruccion.types.b25 == 0 &&
382       instruccion.types.type_26_27 == 2){
383
384     }
385     else if (instruccion.types.b4 == 1 &&
386       instruccion.types.shift_5_6 == 0 &&
387       instruccion.types.b7 == 1 &&
388       instruccion.types.rs_8_11 == 0 &&
389       instruccion.types.s_20 == 0 &&
390       instruccion.types.opcode_21_24 & 0xD == 8 &&
```

```
391       instruccion.types.b25 == 0 &&
392       instruccion.types.type_26_27 == 0){
393
394       rn = instruccion.types.rn_16_19;
395       rd = instruccion.types.rd_12_15;
396       rm = instruccion.types.rm_0_3;
397     }
398
399     stallProc = false;
400     while ((stall = verifyHazards(&prevDelays,
401               instruccion,
402               rn,
403               rs,
404               rm,
405               dirEfectiva,
406               dirEfectiva2)) > 0)
407     {
408       nCiclos -= stall;
409       updateDelays(&prevDelays,stall);
410       prevDelays.remove_if(removeDelay);
411       stallProc = true;
412
413     }
414
415     past = ciclosAntigo - nCiclos;
416     if (past <= 0)
417       past = 1;
418     ciclosAntigo = nCiclos;
419     core->memoryTick(past);
420
421     if (!stallProc){
422       updateDelays(&prevDelays, past);
423       prevDelays.remove_if(removeDelay);
424     }
425
426     prevDelays.splice(prevDelays.end(), delays);
```

```
427
428     dirEfectiva = -1;
429     dirEfectiva2 = -1;
430
431     // Registrar o inicio da computacao
432     if (doLog){
433       fprintf(logFile, "Ciclos: %d\n", past);
434     }
435
436     //Antes de leer, comprobamos el numero de ciclos restantes
437     if ((nCiclos <= 0) || (core->getExcepcionActiva() != EXCEP_NONE))
438       return nCiclos;
439
440     //Lee la siguiente instruccion
441     instruccion.u_word =
442       core->readInstruccion (contexto->PC->u_word, &instrDelay);
443     contexto->PC->u_word += 0x4; //+4 bytes
444     // nCiclos -= memoryDelay;
445     memoryDelay = 0;
446
447     if (core->getExcepcionActiva() != EXCEP_NONE)
448       return nCiclos;
449
450     //Obtenemos la direccion de la subrutina a partir del codigo de la
451     //tabla de saltos (bits 27-20)
452     dir =
453       tablaSaltos
454         [instruccion.comunes.codigoTabla]
455         [instruccion.comunes.codigoTabla2];
456
457     //Salta a la subrutina
458     __asm jmp dir
459
460
461 /*-------------------------*/
462 /*   Customizations
```

```
463 /*  Instructions for Allocate SPM,
464 /*  File Input and Output
465 /*-------------------------*/
466 RUT_custom_instructions:
467
468   // spalloc - allocate a SPM address
469   if (instruccion.SPM.cond == 0x0){
470     dirEfectiva2 = contexto->bancoRegistros[0].u_word;
471     resultado = core->spAlloc(dirEfectiva2, &memoryDelay);
472     contexto->bancoRegistros[0].u_word = resultado;
473
474     // Somente o endereco a ser alocado deve ser incluido nos delays
475     addDelay(&delays,-1,dirEfectiva2,memoryDelay,contexto->PC->u_word);
476     nCiclos -= 1;
477     goto LecturaInstruccion;
478   }
479   // spload
480   else if (instruccion.SPM.cond == 0x2){
481     rd = instruccion.SPM.address1;
482     dirEfectiva = contexto->bancoRegistros[0].u_word;
483     dirEfectiva2 = dirEfectiva & 0xFFFFFFFC;
484     core->spLoad(contexto->bancoRegistros[rd].u_word, dirEfectiva2,
485                  &memoryDelay);
486
487     addDelay(&delays,-1,dirEfectiva2,memoryDelay,
488              contexto->PC->u_word);
489     nCiclos -= 1;
490     goto LecturaInstruccion;
491   }
492   // spstore
493   else if (instruccion.SPM.cond == 0x3){
494     rd = instruccion.SPM.address1;
495     dirEfectiva = instruccion.SPM.address2;
496     dirEfectiva2 = dirEfectiva & 0xFFFFFFFC;
497     core->spStore(contexto->bancoRegistros[rd].u_word,
498                   dirEfectiva2, &memoryDelay);
```

```
499       operando1 = contexto->bancoRegistros[rd].u_word;
500
501       addDelay(&delays, -1, operando1, memoryDelay,
502                contexto->PC->u_word);
503       nCiclos -= 1;
504       goto LecturaInstruccion;
505     }
506     else if (instruccion.SPM.cond == 0xf){
507       switch (instruccion.SPM.address1){
508         // Open the input file
509         case 0x0:
510           openInput();
511           break;
512         // Close the input file
513         case 0x1:
514           closeInput();
515           break;
516         // Open the output file
517         case 0x2:
518           openOutput();
519           break;
520         // Close the output file
521         case 0x3:
522           closeOutput();
523           break;
524         // Read int value
525         case 0x4:
526           readInt();
527           break;
528         // Write int value
529         case 0x5:
530           writeInt();
531           break;
532         // Write char value
533         case 0x6:
534           writeChar();
```

```
535           break;
536         // Read a char value
537         case 0x7:
538           readChar();
539           break;
540         // Write a float value
541         case 0x8:
542           writeFloat();
543           break;
544         // Read a float value
545         case 0x9:
546           readFloat();
547           break;
548         // Toggle splog
549         case 0xa:
550           toggleLog();
551           break;
552         // Halt
553         case 0xf:
554           exit(0);
555           break;
556       }
557     }
558
559     nCiclos--;
560     goto LecturaInstruccion;
561
562
563 /*-------------------------*/
564 /*   Rutina ADD (DP)
565 /*-------------------------*/
566 RUT_add:
567
568     //Comprueba el codigo de condicion
569     COMPROBAR_COND
570
```

```
571    //Obtiene los operandos
572    GETOPS_DP
573
574    if (esDPI) {
575      //Rota (expande) el operando inmediato
576      ROTATE_DPI
577    }else{
578      //Desplaza el operando 2
579      SHIFT
580    }
581
582    //ADD
583    __asm {
584      add     eax, ebx     ;suma
585      pushf           ;guarda los flags en la pila
586      mov     resultado, eax   ;guarda el resultado
587    }
588
589    //Guarda los flags
590    GETFLAGS
591
592    contexto->bancoRegistros[rd].u_word = resultado;
593
594    nCiclos--;
595    goto LecturaInstruccion;
596
597
598  /*-------------------------*/
599  /*  Rutina ADC (DP)
600  /*-------------------------*/
601  RUT_adc:
602
603    //Comprueba el codigo de condicion
604    COMPROBAR_COND
605
606    c = contexto->flags.c;
```

```
607
608     //Obtiene los operandos
609     GETOPS_DP
610
611     if (esDPI) {
612       //Rota (expande) el operando inmediato
613       ROTATE_DPI
614     }else{
615       //Desplaza el operando 2
616       SHIFT
617     }
618
619     //ADC
620     __asm {
621       add     eax, ebx     ;suma
622       add     eax, c
623       pushf           ;guarda los flags en la pila
624       mov     resultado, eax  ;guarda el resultado
625     }
626
627     //Guarda los flags
628     GETFLAGS
629
630     contexto->bancoRegistros[rd].u_word = resultado;
631
632     nCiclos--;
633     goto LecturaInstruccion;
634
635
636 /*-------------------------*/
637 /*  Rutina SUB (DP)
638 /*-------------------------*/
639 RUT_sub:
640
641     //Comprueba el codigo de condicion
642     COMPROBAR_COND
```

```
643
644     //Obtiene los operandos
645     GETOPS_DP
646
647     if (esDPI) {
648       //Rota (expande) el operando inmediato
649       ROTATE_DPI
650     }else{
651       //Desplaza el operando 2
652       SHIFT
653     }
654
655     //SUB
656     __asm {
657       mov     operando2, ebx  ;preserva para el flag de carry
658       neg     ebx
659       add     eax, ebx     ;resta
660       pushf           ;guarda los flags en la pila
661       mov     resultado, eax  ;guarda el resultado
662     }
663
664     //Guarda los flags
665     signo2 = signo2 ^ 0x80000000;
666     GETFLAGS_SUB
667
668     contexto->bancoRegistros[rd].u_word = resultado;
669
670     nCiclos--;
671     goto LecturaInstruccion;
672
673
674 /*-------------------------*/
675 /*  Rutina SBC (DP)
676 /*-------------------------*/
677 RUT_sbc:
678
```

```
679   //Comprueba el codigo de condicion
680   COMPROBAR_COND
681
682   c = contexto->flags.c;
683
684   //Obtiene los operandos
685   GETOPS_DP
686
687   if (esDPI) {
688     //Rota (expande) el operando inmediato
689     ROTATE_DPI
690   }else{
691     //Desplaza el operando 2
692     SHIFT
693   }
694
695   //SBC
696   __asm {
697     add     eax, c
698     sub     eax, 1
699     mov     operando1, eax  ;preserva para el flag de carry
700     mov     operando2, ebx
701     neg     ebx
702     add     eax, ebx     ;resta
703     pushf           ;guarda los flags en la pila
704     mov     resultado, eax  ;guarda el resultado
705   }
706
707   //Guarda los flags
708   signo2 = signo2 ^ 0x80000000;
709   GETFLAGS_SUB
710
711   contexto->bancoRegistros[rd].u_word = resultado;
712
713   nCiclos--;
714   goto LecturaInstruccion;
```

```
715
716
717 /*-------------------------*/
718 /*  Rutina RSB (DP)
719 /*-------------------------*/
720 RUT_rsb:
721
722   //Comprueba el codigo de condicion
723   COMPROBAR_COND
724
725   //Obtiene los operandos
726   GETOPS_DP
727
728   if (esDPI) {
729     //Rota (expande) el operando inmediato
730     ROTATE_DPI
731   }else{
732     //Desplaza el operando 2
733     SHIFT
734   }
735
736   //RSB
737   __asm {
738     mov    operando1, ebx  ;preserva para el flag de carry
739     mov    operando2, eax
740     neg    eax
741     add    ebx, eax    ;resta
742     pushf          ;guarda los flags en la pila
743     mov    resultado, ebx  ;guarda el resultado
744   }
745
746   //Guarda los flags
747   signo1 = signo1 ^ 0x80000000;
748   GETFLAGS_SUB
749
750   contexto->bancoRegistros[rd].u_word = resultado;
```

```
751
752     nCiclos--;
753     goto LecturaInstruccion;
754
755
756
757
758 /*-------------------------*/
759 /*   Rutina RSC (DP)       */
760 /*-------------------------*/
761 RUT_rsc:
762
763     //Comprueba el codigo de condicion
764     COMPROBAR_COND
765
766     c = contexto->flags.c;
767
768     //Obtiene los operandos
769     GETOPS_DP
770
771     if (esDPI) {
772       //Rota (expande) el operando inmediato
773       ROTATE_DPI
774     }else{
775       //Desplaza el operando 2
776       SHIFT
777     }
778
779     //RSC
780     __asm {
781       add     ebx, c
782       sub     ebx, 1
783       mov     operando2, eax  ;preserva para el flag de carry
784       mov     operando1, ebx
785       neg     eax
786       add     ebx, eax     ;resta
```

```
787       pushf           ;guarda los flags en la pila
788       mov     resultado, ebx   ;guarda el resultado
789     }
790
791     //Guarda los flags
792     signo1 = signo1 ^ 0x80000000;
793     GETFLAGS_SUB
794
795     contexto->bancoRegistros[rd].u_word = resultado;
796
797     nCiclos--;
798     goto LecturaInstruccion;
799
800
801 /*-------------------------*/
802 /*   Rutina MOV (DP)
803 /*-------------------------*/
804 RUT_mov:
805
806     //Comprueba el codigo de condicion
807     COMPROBAR_COND
808
809     //Obtiene los operandos
810     GETOPS_DP
811
812     if (esDPI) {
813       //Rota (expande) el operando inmediato
814       ROTATE_DPI
815     }else{
816       //Desplaza el operando 2
817       SHIFT_CARRY
818     }
819
820     //MOV
821     __asm {
822       or      ebx, 0        ;OR 0 para obtener los
```

```
823                         ;flags (el x86 no lo hace con move)
824       pushf             ;guarda los flags en la pila
825       mov     resultado, ebx   ;guarda el resultado
826     }
827
828     //Guarda los flags
829     GETFLAGS_SHIFTER
830
831     contexto->bancoRegistros[rd].u_word = resultado;
832
833     if (esDPI)
834       rn = -1;
835     nCiclos--;
836     goto LecturaInstruccion;
837
838
839 /*-------------------------*/
840 /*   Rutina MVN (DP)
841 /*-------------------------*/
842 RUT_mvn:
843
844     //Comprueba el codigo de condicion
845     COMPROBAR_COND
846
847     //Obtiene los operandos
848     GETOPS_DP
849
850     if (esDPI) {
851       //Rota (expande) el operando inmediato
852       ROTATE_DPI
853     }else{
854       //Desplaza el operando 2
855       SHIFT_CARRY
856     }
857
858     //MVN
```

```
859     __asm {
860       mov     eax, 0FFFFFFFFh
861       xor     eax, ebx      ;XOR con eax para obtener
862                             ;los flags (el x86 no lo hace con move)
863       pushf           ;guarda los flags en la pila
864       mov     resultado, eax  ;guarda el resultado
865     }
866
867     //Guarda los flags
868     GETFLAGS_SHIFTER
869
870     contexto->bancoRegistros[rd].u_word = resultado;
871
872     nCiclos--;
873     goto LecturaInstruccion;
874
875
876   /*-------------------------*/
877   /*   Rutina AND (DP)
878   /*-------------------------*/
879   RUT_and:
880
881     //Comprueba el codigo de condicion
882     COMPROBAR_COND
883
884     //Obtiene los operandos
885     GETOPS_DP
886
887     if (esDPI) {
888       //Rota (expande) el operando inmediato
889       ROTATE_DPI
890     }else{
891       //Desplaza el operando 2
892       SHIFT_CARRY
893     }
894
```

```
895     //AND
896     __asm {
897       and     eax, ebx     ;and
898       pushf                ;guarda los flags en la pila
899       mov     resultado, eax   ;guarda el resultado
900     }
901
902     //Guarda los flags
903     GETFLAGS_SHIFTER
904
905     contexto->bancoRegistros[rd].u_word = resultado;
906
907     nCiclos--;
908     goto LecturaInstruccion;
909
910
911 /*-------------------------*/
912 /*   Rutina EOR (DP)       */
913 /*-------------------------*/
914 RUT_eor:
915
916     //Comprueba el codigo de condicion
917     COMPROBAR_COND
918
919     //Obtiene los operandos
920     GETOPS_DP
921
922     if (esDPI) {
923       //Rota (expande) el operando inmediato
924       ROTATE_DPI
925     }else{
926       //Desplaza el operando 2
927       SHIFT_CARRY
928     }
929
930     //EOR
```

```
931     __asm {
932       xor     eax, ebx     ;or exclusiva
933       pushf              ;guarda los flags en la pila
934       mov     resultado, eax  ;guarda el resultado
935     }
936
937     //Guarda los flags
938     GETFLAGS_SHIFTER
939
940     contexto->bancoRegistros[rd].u_word = resultado;
941
942     nCiclos--;
943     goto LecturaInstruccion;
944
945
946 /*-------------------------*/
947 /*  Rutina ORR (DP)
948 /*-------------------------*/
949 RUT_orr:
950
951     //Comprueba el codigo de condicion
952     COMPROBAR_COND
953
954     //Obtiene los operandos
955     GETOPS_DP
956
957     if (esDPI) {
958       //Rota (expande) el operando inmediato
959       ROTATE_DPI
960     }else{
961       //Desplaza el operando 2
962       SHIFT_CARRY
963     }
964
965     //ORR
966     __asm {
```

```
967       or     eax, ebx     ;or
968       pushf           ;guarda los flags en la pila
969       mov    resultado, eax  ;guarda el resultado
970     }
971
972     //Guarda los flags
973     GETFLAGS_SHIFTER
974
975     contexto->bancoRegistros[rd].u_word = resultado;
976
977     nCiclos--;
978     goto LecturaInstruccion;
979
980
981 /*-------------------------*/
982 /*  Rutina BIC (DP)
983 /*-------------------------*/
984 RUT_bic:
985
986     //Comprueba el codigo de condicion
987     COMPROBAR_COND
988
989     //Obtiene los operandos
990     GETOPS_DP
991
992     if (esDPI) {
993       //Rota (expande) el operando inmediato
994       ROTATE_DPI
995     }else{
996       //Desplaza el operando 2
997       SHIFT_CARRY
998     }
999
1000    //BIC
1001    __asm {
1002      not     ebx          ;NOT Op2
```

```
1003      and     eax, ebx     ;Rn and NOT Op2
1004      pushf           ;guarda los flags en la pila
1005      mov     resultado, eax   ;guarda el resultado
1006    }
1007
1008    //Guarda los flags
1009    GETFLAGS_SHIFTER
1010
1011    contexto->bancoRegistros[rd].u_word = resultado;
1012
1013    nCiclos--;
1014    goto LecturaInstruccion;
1015
1016
1017  /*-------------------------*/
1018  /*  Rutina TST (DP)
1019  /*-------------------------*/
1020  RUT_tst:
1021
1022    //Comprueba el codigo de condicion
1023    COMPROBAR_COND
1024
1025    //Obtiene los operandos
1026    GETOPS_DP
1027
1028    if (esDPI) {
1029      //Rota (expande) el operando inmediato
1030      ROTATE_DPI
1031    }else{
1032      //Desplaza el operando 2
1033      SHIFT_CARRY
1034    }
1035
1036    //TST
1037    __asm {
1038      and     eax, ebx     ;and
```

```
1039        pushf           ;guarda los flags en la pila
1040    }
1041
1042    //Guarda los flags
1043    GETFLAGS_SHIFTER
1044
1045    nCiclos--;
1046    goto LecturaInstruccion;
1047
1048
1049 /*-------------------------*/
1050 /*  Rutina TEQ (DP)
1051 /*-------------------------*/
1052 RUT_teq:
1053
1054    //Comprueba el codigo de condicion
1055    COMPROBAR_COND
1056
1057    //Obtiene los operandos
1058    GETOPS_DP
1059
1060    if (esDPI) {
1061        //Rota (expande) el operando inmediato
1062        ROTATE_DPI
1063    }else{
1064        //Desplaza el operando 2
1065        SHIFT_CARRY
1066    }
1067
1068    //TEQ
1069    __asm {
1070        xor     eax, ebx     ;or exlusiva
1071        pushf           ;guarda los flags en la pila
1072    }
1073
1074    //Guarda los flags
```

```
1075    GETFLAGS_SHIFTER
1076
1077    nCiclos--;
1078    goto LecturaInstruccion;
1079
1080
1081  /*-------------------------*/
1082  /*   Rutina CMP (DP)       */
1083  /*-------------------------*/
1084  RUT_cmp:
1085
1086    //Comprueba el codigo de condicion
1087    COMPROBAR_COND
1088
1089    //Obtiene los operandos
1090    GETOPS_DP
1091
1092    if (esDPI) {
1093      //Rota (expande) el operando inmediato
1094      ROTATE_DPI
1095    }else{
1096      //Desplaza el operando 2
1097      SHIFT
1098    }
1099
1100    //CMP
1101    __asm {
1102      mov     operando2, ebx  ;preserva para el flag de carry
1103      neg     ebx
1104      add     eax, ebx     ;resta
1105      pushf           ;guarda los flags en la pila
1106    }
1107
1108    //Guarda los flags
1109    signo2 = signo2 ^ 0x80000000;
1110    GETFLAGS_SUB
```

```
1111
1112    nCiclos--;
1113    goto LecturaInstruccion;
1114
1115
1116 /*-------------------------*/
1117 /*   Rutina CMN (DP)       */
1118 /*-------------------------*/
1119 RUT_cmn:
1120
1121    //Comprueba el codigo de condicion
1122    COMPROBAR_COND
1123
1124    //Obtiene los operandos
1125    GETOPS_DP
1126
1127    if (esDPI) {
1128      //Rota (expande) el operando inmediato
1129      ROTATE_DPI
1130    }else{
1131      //Desplaza el operando 2
1132      SHIFT
1133    }
1134
1135    //CMN
1136    __asm {
1137      add     eax, ebx     ;suma
1138      pushf                ;guarda los flags en la pila
1139    }
1140
1141    //Guarda los flags
1142    GETFLAGS
1143
1144    nCiclos--;
1145    goto LecturaInstruccion;
1146
```

```
1147
1148 /*-------------------------*/
1149 /*  Rutinas Branch y Branch with link (BBL)
1150 /*-------------------------*/
1151 RUT_bbl:
1152
1153   //Comprueba el codigo de condicion
1154   COMPROBAR_COND
1155
1156   //Obtiene los operandos
1157   //Link
1158   l = instruccion.BBL.L;
1159
1160   //Link: Guarda en el registro link la direccion de
1161   // la instruccion siguiente
1162   if (l)
1163     contexto->bancoRegistros[14].u_word = contexto->PC->u_word;
1164
1165   contexto->PC->u_word = contexto->PC->u_word + 4 + (instruccion.BBL.offset
1166
1167   nCiclos-=3;
1168   goto LecturaInstruccion;
1169
1170
1171 /*-------------------------*/
1172 /*  Rutina Branch and Exchange (BX)
1173 /*-------------------------*/
1174 RUT_bx:
1175
1176   //Comprueba el codigo de condicion
1177   COMPROBAR_COND
1178
1179   //Obtiene los operandos
1180   rm = instruccion.BX.Rm;
1181
1182   contexto->PC->u_word = contexto->bancoRegistros[rm].u_word;
```

```
1183
1184    nCiclos-=3;
1185    goto LecturaInstruccion;
1186
1187
1188
1189
1190  /*-------------------------*/
1191  /*  Rutina LDR pre-index (Load single word and unsigned byte)
1192  /*-------------------------*/
1193  RUT_ldr_pre:
1194
1195    //Comprueba el codigo de condicion
1196    COMPROBAR_COND
1197
1198    //Obtiene los operandos
1199    GETOPS_LS_SHIFT
1200
1201    //Calcular la dir. efectiva (y guardarla, si eso)
1202    PREINDEX
1203
1204    if (b) {
1205      resultado = unsigned int (core->readMemoriaByte(dirEfectiva,
1206                                &memoryDelay));
1207    } else {
1208      dirEfectiva2 = dirEfectiva & 0xFFFFFFFC;
1209      resultado = core->readMemoriaWord(dirEfectiva2, &memoryDelay);
1210
1211      //Ajusta la palabra cargada
1212      rot = ((dirEfectiva2>>2) - (dirEfectiva>>2));
1213      __asm {
1214        mov  eax, resultado
1215        mov  ecx, rot
1216        ror  eax, cl
1217        mov  resultado, eax
1218      }
```

```
1219    }
1220
1221    CHK_ABORT
1222    contexto->bancoRegistros[rd].u_word = resultado;
1223
1224
1225    addDelay(&delays, rd, -1, memoryDelay, contexto->PC->u_word);
1226    nCiclos -= 1;
1227    goto LecturaInstruccion;
1228
1229
1230  /*-------------------------*/
1231  /*  Rutina STR pre-index (Store single word and unsigned byte)
1232  /*-------------------------*/
1233  RUT_str_pre:
1234
1235    //Comprueba el codigo de condicion
1236    COMPROBAR_COND
1237
1238    //Obtiene los operandos
1239    GETOPS_LS_SHIFT
1240
1241    //Calcular la dir. efectiva (y guardarla, si eso)
1242    PREINDEX
1243
1244    if (b) {
1245      core->writeMemoriaByte(contexto->bancoRegistros[rd].u_byte,
1246                             dirEfectiva, &memoryDelay);
1247      addDelay(&delays, -1, dirEfectiva, memoryDelay,
1248               contexto->PC->u_word);
1249    } else {
1250      dirEfectiva2 = dirEfectiva & 0xFFFFFFFC;
1251      core->writeMemoriaWord(contexto->bancoRegistros[rd].u_word,
1252                             dirEfectiva2, &memoryDelay);
1253      addDelay(&delays, -1, dirEfectiva2, memoryDelay,
1254                             contexto->PC->u_word);
```

```
1255    }
1256
1257    nCiclos -= 1;
1258    goto LecturaInstruccion;
1259
1260
1261  /*-------------------------*/
1262  /*  Rutina LDR post-index (Load single word and unsigned byte)
1263  /*-------------------------*/
1264  RUT_ldr_post:
1265
1266    //Comprueba el codigo de condicion
1267    COMPROBAR_COND
1268
1269    //Obtiene los operandos
1270    GETOPS_LS_SHIFT
1271
1272
1273    if (b) {
1274      resultado = unsigned int (
1275          core->readMemoriaByte(operando1,&memoryDelay));
1276    } else {
1277      dirEfectiva = operando1 & 0xFFFFFFFC;
1278      resultado = core->readMemoriaWord(
1279          dirEfectiva,&memoryDelay);
1280
1281      //Ajusta la palabra cargada
1282      rot = ((dirEfectiva>>2) - (operando1>>2));
1283      __asm {
1284        mov  eax, resultado
1285        mov  ecx, rot
1286        ror  eax, cl
1287        mov  resultado, eax
1288      }
1289    }
1290
```

```
1291
1292    //Calcular la dir. efectiva (y guardarla)
1293    POSTINDEX
1294
1295    CHK_ABORT
1296    contexto->bancoRegistros[rd].u_word = resultado;
1297
1298    addDelay(&delays, rd, -1, memoryDelay, contexto->PC->u_word);
1299    nCiclos -= 1;
1300    goto LecturaInstruccion;
1301
1302
1303 /*-------------------------*/
1304 /*  Rutina STR post-index (Store single word and unsigned byte)
1305 /*-------------------------*/
1306 RUT_str_post:
1307
1308    //Comprueba el codigo de condicion
1309    COMPROBAR_COND
1310
1311    //Obtiene los operandos
1312    GETOPS_LS_SHIFT
1313
1314
1315    if (b) {
1316      dirEfectiva = operando1;
1317      core->writeMemoriaByte(contexto->bancoRegistros[rd].u_byte,
1318        dirEfectiva, &memoryDelay);
1319      dirEfectiva = operando1;
1320    } else {
1321      dirEfectiva = operando1 & 0xFFFFFFFC;
1322      core->writeMemoriaWord(contexto->bancoRegistros[rd].u_word,
1323        dirEfectiva, &memoryDelay);
1324    }
1325
1326    //Calcular la dir. efectiva (y guardarla)
```

```
1327    POSTINDEX
1328
1329      addDelay(&delays, -1, dirEfectiva, memoryDelay,
1330          contexto->PC->u_word);
1331    nCiclos--;
1332    goto LecturaInstruccion;
1333
1334
1335  /*-------------------------*/
1336  /*  Rutina LDR HW/B pre-index (Load half-word / signed byte)
1337  /*-------------------------*/
1338  RUT_ldr_hw_pre:
1339
1340    //Comprueba el codigo de condicion
1341    COMPROBAR_COND
1342
1343    //Obtiene los operandos
1344    GETOPS_LS_EXTRAS
1345
1346    //Calcular la dir. efectiva (y guardarla, si eso)
1347    PREINDEX
1348
1349    switch (esSignedHalfword) {
1350    case 2:
1351      resultado = (int8)(core->readMemoriaByte(dirEfectiva,
1352        &memoryDelay));
1353      break;
1354    case 3:
1355      resultado = (int16)(core->readMemoriaHalfword(dirEfectiva,
1356        &memoryDelay));
1357      break;
1358    default:
1359      resultado = unsigned int (core->readMemoriaHalfword(dirEfectiva,
1360        &memoryDelay));
1361    }
1362
```

```
1363    CHK_ABORT
1364    contexto->bancoRegistros[rd].u_word = resultado;
1365
1366    addDelay(&delays, rd, -1, memoryDelay, contexto->PC->u_word);
1367    nCiclos -= 1;
1368    goto LecturaInstruccion;
1369
1370
1371 /*-------------------------*/
1372 /*  Rutina STR HW pre-index (Store half-word / signed byte)
1373 /*-------------------------*/
1374 RUT_str_hw_pre:
1375
1376    //Comprueba el codigo de condicion
1377    COMPROBAR_COND
1378
1379    //Obtiene los operandos
1380    GETOPS_LS_EXTRAS
1381
1382    //Calcular la dir. efectiva (y guardarla, si eso)
1383    PREINDEX
1384
1385    core->writeMemoriaHalfword(contexto->bancoRegistros[rd].u_halfword,
1386       dirEfectiva, &memoryDelay);
1387
1388    addDelay(&delays, -1, dirEfectiva, memoryDelay,
1389       contexto->PC->u_word);
1390    nCiclos -= 1;
1391    goto LecturaInstruccion;
1392
1393
1394
1395
1396 /*-------------------------*/
1397 /*  Rutina LDR HW/B post-index (Load half-word / signed byte)
1398 /*-------------------------*/
```

```
1399 RUT_ldr_hw_post:
1400
1401   //Comprueba el codigo de condicion
1402   COMPROBAR_COND
1403
1404   //Obtiene los operandos
1405   GETOPS_LS_EXTRAS
1406
1407   dirEfectiva = operando1;
1408   switch (esSignedHalfword) {
1409   case 2:
1410     // resultado = (int8)(core->readMemoriaByte(operando1,
1411       &memoryDelay));
1412     resultado = (int8)(core->readMemoriaByte(dirEfectiva,
1413       &memoryDelay));
1414     break;
1415   case 3:
1416     resultado = (int16)(core->readMemoriaHalfword(dirEfectiva,
1417       &memoryDelay));
1418     break;
1419   default:
1420     resultado = unsigned int (core->readMemoriaHalfword(dirEfectiva,
1421       &memoryDelay));
1422   }
1423
1424   //Calcular la dir. efectiva (y guardarla)
1425   POSTINDEX
1426
1427   CHK_ABORT
1428   contexto->bancoRegistros[rd].u_word = resultado;
1429
1430   addDelay(&delays, rd, -1, memoryDelay, contexto->PC->u_word);
1431   nCiclos -= 1;
1432   goto LecturaInstruccion;
1433
1434
```

```
1435 /*-------------------------*/
1436 /*  Rutina STR HW post-index (Store half-word / signed byte)
1437 /*-------------------------*/
1438 RUT_str_hw_post:
1439
1440   //Comprueba el codigo de condicion
1441   COMPROBAR_COND
1442
1443   //Obtiene los operandos
1444   GETOPS_LS_EXTRAS
1445
1446   operando1 = dirEfectiva;
1447   core->writeMemoriaHalfword(contexto->bancoRegistros[rd].u_halfword,
1448       dirEfectiva, &memoryDelay);
1449
1450   //Calcular la dir. efectiva (y guardarla)
1451   POSTINDEX
1452
1453   dirEfectiva = operando1;
1454   addDelay(&delays, -1, dirEfectiva, memoryDelay,
1455       contexto->PC->u_word);
1456   nCiclos -= 1;
1457   goto LecturaInstruccion;
1458
1459
1460 /*-------------------------*/
1461 /*  Rutina LDM pre-index (Load multiple)
1462 /*-------------------------*/
1463 RUT_ldm_pre:
1464
1465   //Comprueba el codigo de condicion
1466   COMPROBAR_COND
1467
1468   //Obtiene los operandos
1469   GETOPS_LSM
1470
```

```
1471    //Para cada registro, lo transfiere si su bit esta a 1
1472    for ((u==1)? i = 0 : i=15;
1473        (u==1)? (i<16): (i>=0);
1474        (u==1)?  i++  : i--) {
1475
1476      if (listaRegistros & (0x1<<i)) {
1477        // nCiclos--; See below
1478
1479        //Pre-indexada
1480        dirEfectiva += dirOffset;
1481        //Lectura de memoria
1482        resultado = core->readMemoriaWord(dirEfectiva,&memoryDelay);
1483        nCiclos -= memoryDelay;
1484        memoryDelay = 0;
1485
1486        //Escribe el resultado en el registro correspondiente
1487        LSM_WRITEREGS
1488      }
1489
1490    }
1491
1492    //W=1 (Write-back)
1493    if (w) {
1494      contexto->bancoRegistros[rn].u_word = dirEfectiva;
1495    }
1496
1497    CHK_ABORT
1498
1499    //lsmCPSR (Se restaura el CPSR con el SPSR)
1500    if (lsmCPSR) {
1501      contexto->escribirCPSR(contexto->SPSR[contexto->modo-1].u_word);
1502    }
1503
1504    nCiclos--;
1505    goto LecturaInstruccion;
1506
```

```
1507
1508 /*-------------------------*/
1509 /*  Rutina LDM post-index (Load multiple)
1510 /*-------------------------*/
1511 RUT_ldm_post:
1512
1513   //Comprueba el codigo de condicion
1514   COMPROBAR_COND
1515
1516   //Obtiene los operandos
1517   GETOPS_LSM
1518
1519   //Para cada registro, lo transfiere si su bit esta a 1
1520   for ((u==1)? i = 0 : i=15;
1521       (u==1)? (i<16): (i>=0);
1522       (u==1)?  i++  : i--) {
1523
1524     if (listaRegistros & (0x1<<i)) {
1525       // nCiclos--; See below
1526
1527       //Lectura de memoria
1528       resultado = core->readMemoriaWord(dirEfectiva,&memoryDelay);
1529       nCiclos--;
1530       memoryDelay = 0;
1531
1532       //Escribe el resultado en el registro correspondiente
1533       LSM_WRITEREGS
1534
1535       // Adiciona o delay (baseado no codigo de LSM_WRITEREGS)
1536       if (!((i == rn)
1537           && (core->getExcepcionActiva() == EXCEP_DATA_ABORT))) {
1538         if ((lsmUserMode) && (i >= 8) && (i < 15)){
1539           // Estamos en modo FIQ, los regs 8-12 estan banqueados
1540           if ((i < 13) && (contexto->modo != FIQ_MODE)) {
1541             // Estamos en otro modo, los regs 8-12
1542             // coinciden con los de usuario
```

```
1543                addDelay(&delays, i, -1, memoryDelay, contexto->PC->u_word);
1544              }
1545            }
1546            else{
1547              addDelay(&delays, i, -1, memoryDelay, contexto->PC->u_word);
1548            }
1549          }
1550
1551          //Post-indexada
1552          dirEfectiva += dirOffset;
1553        }
1554
1555      }
1556
1557      //W=1 (Write-back)
1558      if (w) {
1559        contexto->bancoRegistros[rn].u_word = dirEfectiva;
1560      }
1561
1562     CHK_ABORT
1563
1564      //lsmCPSR (Se restaura el CPSR con el SPSR)
1565      if (lsmCPSR) {
1566        contexto->escribirCPSR(contexto->SPSR[contexto->modo-1].u_word);
1567      }
1568
1569      nCiclos--;
1570      goto LecturaInstruccion;
1571
1572
1573  /*-------------------------*/
1574  /*  Rutina STM pre-index (Store multiple)
1575  /*-------------------------*/
1576  RUT_stm_pre:
1577
1578      //Comprueba el codigo de condicion
```

```
1579    COMPROBAR_COND
1580
1581    //Obtiene los operandos
1582    GETOPS_LSM
1583
1584    //Para cada registro, lo transfiere si su bit esta a 1
1585    for ((u==1)? i = 0 : i=15;
1586        (u==1)? (i<16): (i>=0);
1587        (u==1)?  i++  : i--) {
1588
1589      if (listaRegistros & (0x1<<i)) {
1590        nCiclos--;
1591        //Pre-indexada
1592        dirEfectiva += dirOffset;
1593
1594        //Carga el valor del registro correspondiente
1595        LSM_READREGS
1596
1597        //Escritura en memoria
1598        core->writeMemoriaWord(resultado, dirEfectiva, &memoryDelay);
1599
1600        addDelay(&delays, -1, dirEfectiva, memoryDelay,
1601          contexto->PC->u_word);
1602        memoryDelay = 0;
1603      }
1604
1605    }
1606
1607    //W=1 (Write-back)
1608    if (w) {
1609      contexto->bancoRegistros[rn].u_word = dirEfectiva;
1610    }
1611
1612    nCiclos--;
1613    goto LecturaInstruccion;
1614
```

```
1615
1616 /*-------------------------*/
1617 /*  Rutina STM post-index (Store multiple)
1618 /*-------------------------*/
1619 RUT_stm_post:
1620
1621   //Comprueba el codigo de condicion
1622   COMPROBAR_COND
1623
1624   //Obtiene los operandos
1625   GETOPS_LSM
1626
1627   //Para cada registro, lo transfiere si su bit esta a 1
1628   for ((u==1)? i = 0 : i=15;
1629       (u==1)? (i<16): (i>=0);
1630       (u==1)?  i++  : i--) {
1631
1632     if (listaRegistros & (0x1<<i)) {
1633       nCiclos--;
1634
1635       //Carga el valor del registro correspondiente
1636       LSM_READREGS
1637
1638       //Escritura en memoria
1639       core->writeMemoriaWord(resultado, dirEfectiva, &memoryDelay);
1640       addDelay(&delays, -1, dirEfectiva, memoryDelay,
1641         contexto->PC->u_word);
1642       memoryDelay = 0;
1643
1644       //Post-indexada
1645       dirEfectiva += dirOffset;
1646     }
1647
1648   }
1649
1650   //W=1 (Write-back)
```

```
1651    if (w) {
1652      contexto->bancoRegistros[rn].u_word = dirEfectiva;
1653    }
1654
1655    nCiclos--;
1656    goto LecturaInstruccion;
1657
1658
1659  /*-------------------------*/
1660  /*  Rutina SWP (Swap Memory and Register)
1661  /*-------------------------*/
1662  RUT_swap:
1663
1664    //Comprueba el codigo de condicion
1665    COMPROBAR_COND
1666
1667    //Obtiene los operandos
1668    GETOPS_LS
1669
1670
1671    if (b) {
1672      resultado = unsigned int (
1673        core->readMemoriaByte(operando1,&memoryDelay));
1674      addDelay(&delays, rd, -1, memoryDelay, contexto->PC->u_word);
1675      memoryDelay = 0;
1676      CHK_ABORT
1677      core->writeMemoriaByte((uint8)operando2,
1678                             operando1, &memoryDelay);
1679      dirEfectiva = operando1;
1680    } else {
1681      dirEfectiva = operando1 & 0xFFFFFFFC;
1682      resultado = core->readMemoriaWord(dirEfectiva,&memoryDelay);
1683      addDelay(&delays, rd, -1, memoryDelay, contexto->PC->u_word);
1684      memoryDelay = 0;
1685      CHK_ABORT
1686      core->writeMemoriaWord(operando2, dirEfectiva, &memoryDelay);
```

```
1687    }
1688
1689    contexto->bancoRegistros[rd].u_word = resultado;
1690
1691    addDelay(&delays, -1, dirEfectiva,
1692             memoryDelay, contexto->PC->u_word);
1693    nCiclos--;
1694    goto LecturaInstruccion;
1695
1696
1697 /*-------------------------*/
1698 /*  Rutina MUL (Multiply and accumulate)
1699 /*-------------------------*/
1700 RUT_mul:
1701
1702    //Comprueba el codigo de condicion
1703    COMPROBAR_COND
1704
1705    //Obtiene los operandos
1706    GETOPS_MUL
1707
1708    v = contexto->flags.v;
1709
1710    //MUL
1711    if (a) {
1712      //Acumula
1713      __asm {
1714        mov eax, operando1
1715        mov ebx, operando2
1716        mov edx, 0
1717        mul ebx          ;edx:eax = eax * ebx
1718        add eax, acumulador  ;eax = eax + acc
1719        pushf          ;guarda los flags en la fila
1720        mov resultado, eax
1721      }
1722
```

```
1723    }else{
1724      //No acumula
1725      __asm {
1726        mov eax, operando1
1727        mov ebx, operando2
1728        mov edx, 0
1729        mul ebx          ;edx:eax = eax * ebx
1730        pushf          ;guarda los flags en la fila
1731        mov resultado, eax
1732      }
1733    }
1734
1735    //Guarda los flags
1736    GETFLAGS
1737    contexto->flags.v = v;
1738
1739    contexto->bancoRegistros[rd].u_word = resultado;
1740
1741    nCiclos-=5;
1742    goto LecturaInstruccion;
1743
1744
1745
1746
1747 /*-------------------------*/
1748 /*  Rutina MULL (Multiply and accumulate long)
1749 /*-------------------------*/
1750 RUT_mull:
1751
1752    //Comprueba el codigo de condicion
1753    COMPROBAR_COND
1754
1755    //Obtiene los operandos
1756    GETOPS_MUL
1757
1758    v = contexto->flags.v;
```

```
1759
1760    //MUL
1761    if (a && !u) {
1762      //Acumula y sin signo
1763      __asm {
1764
1765        mov eax, operando1
1766        mov ebx, operando2
1767        mov edx, 0
1768        mul ebx          ;edx:eax = eax * ebx
1769
1770        movd mm0, eax
1771        movd mm1, edx
1772        punpckldq mm0, mm1  ;Monta los dos registros de
1773                            ;32 bits en uno de 64
1774
1775        movd mm1, acumulador
1776        movd mm2, acumulador2
1777        punpckldq mm1, mm2  ;Monta el registro acumulador
1778
1779        movq uvar1_64, mm0
1780        movq uvar2_64, mm1
1781      }
1782
1783      uvar1_64 += uvar2_64;
1784
1785      __asm {
1786        movq mm0, uvar1_64
1787        movd resultado, mm0  ;Parte baja del resultado
1788        punpckhdq mm0, mm1  ;Pone la parte alta de mm0 en
1789                            ;la parte baja
1790        movd resultado2, mm0 ;Parte alta del resultado
1791        emms
1792      }
1793
1794      contexto->flags.n = (uvar1_64 < 0)?1:0;
```

```
1795      contexto->flags.z = (uvar1_64 == 0)?1:0;
1796
1797    }else if (a && u) {
1798      //Acumula y con signo
1799
1800      __asm {
1801
1802        mov eax, operando1
1803        mov ebx, operando2
1804        mov edx, 0
1805        imul ebx          ;edx:eax = eax * ebx
1806
1807        movd mm0, eax
1808        movd mm1, edx
1809        punpckldq mm0, mm1  ;Monta los dos registros de
1810                            ;32 bits en uno de 64
1811
1812        movd mm1, acumulador
1813        movd mm2, acumulador2
1814        punpckldq mm1, mm2  ;Monta el registro acumulador
1815
1816        movq var1_64, mm0
1817        movq var2_64, mm1
1818      }
1819
1820      var1_64 += var2_64;
1821
1822      __asm {
1823        movq mm0, var1_64
1824        movd resultado, mm0  ;Parte baja del resultado
1825        punpckhdq mm0, mm1  ;Pone la parte alta de
1826                            ;mm0 en la parte baja
1827        movd resultado2, mm0 ;Parte alta del resultado
1828        emms
1829      }
1830
```

```
1831      contexto->flags.n = (var1_64 < 0)?1:0;
1832      contexto->flags.z = (var1_64 == 0)?1:0;
1833
1834    }else if(!a && !u){
1835      //No acumula y sin signo
1836      __asm {
1837        mov eax, operando1
1838        mov ebx, operando2
1839        mov edx, 0
1840        mul ebx          ;edx:eax = eax * ebx
1841        pushf          ;guarda los flags en la fila
1842        mov resultado, eax
1843        mov resultado2, edx
1844      }
1845
1846      //Guarda los flags
1847      GETFLAGS
1848    }else if(!a && u){
1849      //No acumula y con signo
1850      __asm {
1851        mov eax, operando1
1852        mov ebx, operando2
1853        mov edx, 0
1854        imul ebx        ;edx:eax = eax * ebx
1855        pushf          ;guarda los flags en la fila
1856        mov resultado, eax
1857        mov resultado2, edx
1858      }
1859
1860      //Guarda los flags
1861      GETFLAGS
1862    }
1863
1864    contexto->flags.v = v;
1865
1866    contexto->bancoRegistros[rn].u_word = resultado;
```

```
1867    contexto->bancoRegistros[rd].u_word = resultado2;
1868
1869    nCiclos-=6;
1870    goto LecturaInstruccion;
1871
1872
1873  /*-------------------------*/
1874  /*  Rutina MSR (Move Status Register to Register)
1875  /*-------------------------*/
1876  RUT_msr:
1877
1878    //Comprueba el codigo de condicion
1879    COMPROBAR_COND
1880
1881    //Obtiene los operandos
1882    GETOPS_MSR
1883
1884    //CPSR
1885    if (!r) {
1886      operando2 = (contexto->leerCPSR());
1887      //f
1888      if (mascara & 0x8) {
1889        operando2 = (operando1 & 0xFF000000)
1890          | (operando2 & 0x00FFFFFF);
1891      }
1892      if ((contexto->modo != USER_MODE) && (!esMISR)) {
1893        //s
1894        if (mascara & 0x4) {
1895          operando2 = (operando1 & 0x00FF0000)
1896            | (operando2 & 0xFF00FFFF);
1897        }
1898        //x
1899        if (mascara & 0x2) {
1900          operando2 = (operando1 & 0x0000FF00)
1901            | (operando2 & 0xFFFF00FF);
1902        }
```

```
1903        //c
1904        if (mascara & 0x1) {
1905          operando2 = (operando1 & 0x000000FF)
1906            | (operando2 & 0xFFFFFF00);
1907          nCiclos-=2;
1908        }
1909      }
1910
1911      contexto->escribirCPSR(operando2);
1912
1913    //SPSR
1914    } else if ((contexto->modo != USER_MODE)
1915               && (contexto->modo != SYSTEM_MODE)) {
1916      operando2 = (contexto->SPSR[contexto->modo - 1].u_word);
1917      //f
1918      if (mascara & 0x8) {
1919        operando2 = (operando1 & 0xFF000000)
1920          | (operando2 & 0x00FFFFFF);
1921      }
1922      //s
1923      if (!esMISR) {
1924        if (mascara & 0x4) {
1925          operando2 = (operando1 & 0x00FF0000)
1926            | (operando2 & 0xFF00FFFF);
1927        }
1928        //x
1929        if (mascara & 0x2) {
1930          operando2 = (operando1 & 0x0000FF00)
1931            | (operando2 & 0xFFFF00FF);
1932        }
1933        //c
1934        if (mascara & 0x1) {
1935          operando2 = (operando1 & 0x000000FF)
1936            | (operando2 & 0xFFFFFF00);
1937          nCiclos-=2;
1938        }
```

```
1939        }
1940
1941        contexto->SPSR[contexto->modo - 1].u_word = operando2;
1942
1943    }
1944
1945    nCiclos--;
1946    goto LecturaInstruccion;
1947
1948
1949 /*-------------------------*/
1950 /*  Rutina MRS (Move Register to Status Register)
1951 /*-------------------------*/
1952 RUT_mrs:
1953
1954    //Comprueba el codigo de condicion
1955    COMPROBAR_COND
1956
1957    //Obtiene los operandos
1958    rd = instruccion.DPIS.Rd;
1959    r = instruccion.MISR.R;
1960
1961    //CPSR
1962    if (!r) {
1963        contexto->bancoRegistros[rd].u_word = contexto->leerCPSR();
1964
1965    //SPSR
1966    } else if ((contexto->modo != USER_MODE)
1967                && (contexto->modo != SYSTEM_MODE)) {
1968        contexto->bancoRegistros[rd].u_word =
1969                (contexto->SPSR[contexto->modo - 1].u_word);
1970
1971    }
1972
1973    nCiclos--;
1974    goto LecturaInstruccion;
```

```
1975
1976
1977 /*-------------------------*/
1978 /*  Rutina SWI (Software Interrupt)
1979 /*-------------------------*/
1980 RUT_swi:
1981
1982    //Comprueba el codigo de condicion
1983    COMPROBAR_COND
1984
1985    //Obtiene los operandos
1986    operando1 = instruccion.SWI.swiNumber;
1987
1988
1989    core->marcarExcepcion (EXCEP_SWI, operando1);
1990
1991    nCiclos-=3;
1992    goto LecturaInstruccion;
1993
1994
1995 /*-------------------------*/
1996 /*  Rutina CODP (Coprocessor Data Processing)
1997 /*-------------------------*/
1998 RUT_codp:
1999
2000    //Comprueba el codigo de condicion
2001    COMPROBAR_COND
2002
2003    operando1 = instruccion.CODP.cp_num;
2004    coproDP fdp = core->getCoproDP(operando1);
2005
2006    if (fdp != NULL) {
2007       nCiclos -= fdp(instruccion.u_word);
2008    } else {
2009       core->marcarExcepcion (EXCEP_UNDEFINED, operando1);
2010    }
```

```
2011
2012    nCiclos--;
2013    goto LecturaInstruccion;
2014
2015
2016  /*-------------------------*/
2017  /*  Rutina COLS (Coprocessor Load Store)
2018  /*-------------------------*/
2019  RUT_cols:
2020
2021    //Comprueba el codigo de condicion
2022    COMPROBAR_COND
2023
2024    operando1 = instruccion.COLS.cp_num;
2025    coproLS fls = core->getCoproLS(operando1);
2026
2027    if (fls != NULL) {
2028      nCiclos -= fls(instruccion.u_word);
2029    } else {
2030      core->marcarExcepcion (EXCEP_UNDEFINED, operando1);
2031    }
2032
2033    nCiclos--;
2034    goto LecturaInstruccion;
2035
2036
2037  /*-------------------------*/
2038  /*  Rutina CORT (Coprocessor Register Transfer)
2039  /*-------------------------*/
2040  RUT_cort:
2041
2042    //Comprueba el codigo de condicion
2043    COMPROBAR_COND
2044
2045    operando1 = instruccion.CORT.cp_num;
2046    coproRT frt = core->getCoproRT(operando1);
```

```
2047
2048    if (frt != NULL) {
2049      nCiclos -= frt(instruccion.u_word);
2050    } else {
2051      core->marcarExcepcion (EXCEP_UNDEFINED, operando1);
2052    }
2053
2054    nCiclos--;
2055    goto LecturaInstruccion;
2056
2057
2058  /*-------------------------*/
2059  /*   Rutina INDEFINIDA
2060  /*-------------------------*/
2061  RUT_undefined:
2062
2063    core->marcarExcepcion (EXCEP_UNDEFINED);
2064
2065    nCiclos--;
2066    goto LecturaInstruccion;
2067  }
2068
2069  void ARM9RutinasARM::addDelay(uint32 reg, uint32 address,
2070                                uint32 delay){
2071    addDelay(&delays, reg, address, delay,
2072             this->contexto->PC->u_word);
2073  }
```

A.3 Arquivo de Código Principal

A seguir apresenta-se o código que origina o fluxo de execução do simulador. Ele é responsável pela criação dos objetos, por sua configuração e pelo início da simulação.

Código A.4: Arquivo Principal.h

```
1 #include "ARM9Core.h"
2 #include "ARM9Cache.h"
3 #include "ARM9Memory.h"
```

```
4  #include "ARM9WriteBuffer.h"
5
6  #include <iostream>
7  #include <fstream>
8  using namespace std;
9
10 ARM9Cache *level1, *level2;
11 ARM9Cache* instCache;
12 ARM9WriteBuffer* writeBuffer;
13 ARM9Memory* memory;
14 bool prefetch = false;
15
16 void externa(void){
17
18 }
19
20 uint32 memoryReadHandlerWord(uint32 address, uint32* delay){
21   return level1->readMemoriaWord(address,delay);
22 }
23
24 void memoryWriteHandlerWord(uint32 data, uint32 address,
25                             uint32 *delay){
26   writeBuffer->writeMemoriaWord(address,data,delay);
27 }
28
29 void memoryTick(uint32 cycles){
30   writeBuffer->tick(cycles);
31 }
32
33 uint32 instHandler(uint32 address, uint32* delay){
34   return instCache->readMemoriaWord(address,delay);
35 }
36
37 int main(int argc, char **argv){
38
39   if (argc > 1){
```

```
40      std::string param(argv[1]);
41      prefetch = param.compare("-p") == 0;
42    }
43
44    ARM9MemRegion memoryRegions;
45
46    const int MEMORY_SIZE = 0x4000000;
47
48    ARM9Core core;
49
50    // Memory with 16MB, 100 processor cycles for each
51    // memory access
52    memory = new ARM9Memory(MEMORY_SIZE, 100);
53    memory->id = MEMORY;
54
55    // Cache with 1 MB, 20 cycles of processor for each
56    // cache L2 access
57    level2 = new ARM9Cache(0x100000, 0x40, 16, 20,
58                           memory, core.rutinas());
59    level2->id = DATA_CACHE_2;
60
61    // Cache with 32 KB, 4 cycles of processor for each
62    // cache L1 access
63    level1 = new ARM9Cache(0x8000,0x40,4,4,level2,core.rutinas());
64    level1->id = DATA_CACHE;
65
66    // Instruction cache with 16 words, 0 cycle of processor
67    // for each instruction cache access
68    // because if processor bring the instruction word in advance
69    instCache = new ARM9Cache(0x8000,0x80,1,0,memory,NULL);
70    instCache->id = INST_CACHE;
71
72    // Buffer for writes
73    writeBuffer = new ARM9WriteBuffer(level1);
74
75    memoryRegions.inferior = 0;
```

```
76    memoryRegions.superior = MEMORY_SIZE;
77    memoryRegions.buffer = NULL;
78    memoryRegions.handlerRead32 =
79      memoryRegions.handlerRead16 =
80      memoryRegions.handlerRead8 = memoryReadHandlerWord;
81    memoryRegions.handlerWrite8 =
82      memoryRegions.handlerWrite16 =
83      memoryRegions.handlerWrite32 = memoryWriteHandlerWord;
84    memoryRegions.handlerInstruction = instHandler;
85    memoryRegions.permisos =
86      USR_READ_PERM |
87      USR_WRITE_PERM |
88      PRIV_READ_PERM |
89      PRIV_WRITE_PERM;
90    memoryRegions.memoryTick = memoryTick;
91
92    // Um core apenas
93    core.setCores(1);
94
95    core.addMemoria(1,&memoryRegions);
96
97    core.setFuncionExterna(&externa);
98
99    // 50 Mhz
100   core.setFrecuenciaARM(50.0);
101
102   // Link core and cache
103   core.addCache(level1);
104
105   // Read the program and put it in the memory
106   ifstream prog("./program.in");
107   if (! prog.is_open())
108     throw "Program.in could not be opened";
109   uint32 instr;
110   uint32 address = 0;
111   while (! prog.eof()){
```

```
112         prog >> hex >> instr;
113         memory->initializeMemory(address, instr);
114         address += 4;
115     }
116
117     // Reiniciar o core e deixa-lo pronto para a execucao
118     core.reset();
119
120     core.run(1000000);
121
122     return 0;
123 }
```

Referências Bibliográficas

Angiolini, F.; Benini, L. & Caprara, A. (2003). Polynomialtime algorithm for onchip scratchpad memory partitioning. *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems.*

Banakar, R.; Steinke, S.; Lee, B.-S.; Balakrishnan, M. & Marwedel, P. (2002). Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. Em *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pp. 73--78, New York, NY, USA. ACM.

Burks, A. W.; Goldstine, H. H. & von Neumann, J. (1946). Preliminary discussion of the logical design of an electronict computing instrument.

Cook, H.; Asanovi, K. & Patterson, D. A. (2009). Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments. Relatório técnico UCB/EECS-2009-131, Electrical Engineering and Computer Sciences University of California at Berkeley.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271. ISSN 0029-599X.

Francesco, P.; Marchal, P.; Atienza, D.; Benini, L.; Catthoor, F. & Mendias, J. M. (2004). An integrated hardware/software approach for run-time scratchpad management. *DAC '04 Proceedings of the 41st annual Design Automation Conference.*

Francillon, A. & Castelluccia, C. (2008). Code injection attacks on harvard-architecture devices. Em *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pp. 15--26, New York, NY, USA. ACM.

FREUND, J. E. (Porto Alegre: 2004). *Estatística Aplicada Economicamente. 11. ed.* Bookman.

Hennessy, J. L. & Patterson, D. A. (2007). *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann Publishers.

Hiser, J. D. & Davidson, J. W. (2004). Embarc: An efficient memory bank assignment algorithm for retargetable compilers. *SIGPLAN Not.*, 39(7):182--191. ISSN 0362-1340.

Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321--. ISSN 0001-0782.

Kandemir, M.; Ramanujam, J.; Irwin, M. J.; Vijaykrishnan, N.; Kadayif, I. & Parikh, A. (2001). Dynamic management of scratch-pad memory space. *Design Automation Conference, 2001. Proceedings.*

Kilburn, T.; Edwards, D. B. G.; Lanigan, M. J. & Sumner, F. H. (1962). One-level storage system.

Moreira, F. B.; Alves, M. A. Z. & Navaux, P. O. A. (2010). Simulating scratchpad memories for general purpose processors. *WSPPD 2010 - VIII Workshop de Processamento Paralelo e Distribuído.*

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers.

Nguyen, N.; Dominguez, A. & Barua, R. (2005). Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. pp. 115--125.

ARM Limited (2005). Arm architecture reference manual. Relatório técnico, ARM Limited.

Intel Corporation (2013). 4th generation intel core i7 processors.

LLVM Developer Group (2015). The llvm compiler infrastructure project.

Samsung Electronics (2011). Samsung exynos 4210 - risc microprocessor. Relatório técnico, Samsung Electronics.

Panda, P. R.; Dutt, N. & Nicolau, A. (1999). *Memory Issues in Embedded System-On-Chip - Optimizations and Exploration.* Kluwer Acadamic Publishers.

Panda, P. R.; Dutt, N. D. & Nicolau, A. (1997). Efficient utilization of scratch-pad memory in embedded processor applications. *Proceedings of the 1997 European Design and Test Conference.*

Patterson, D. A. & Hennessy, J. L. (2012). *Computer organization and design: the hardware/software interface 4th ed. p. cm.* Morgan Kaufmann Publishers.

Serrano, S. H.; Aranda, A. H. & Vacas, D. S. (2007). Arm9core - un emulador del procesador arm9tdmi para pc. Relatório técnico, Universidad Complutense de Madrid.

Udayakumaran, S. & Barua, R. (2003). Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. Em *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pp. 276--286, New York, NY, USA. ACM.

Wilkes, M. (1965). Slave memories and dynamic storage allocation. IEEE Trans. Electronic Computers EC.

Williams, S.; Carter, J.; Oliker, L.; Shalf, J. & Yelick, K. (2008). Lattice boltzmann simulation optimization on leading multicore platforms. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.*

Williams, S.; Shalf, J.; Oliker, L.; Kamil, S.; Husbands, P. & Yelick, K. (2007). Scientific computing kernels on the cell processor. *International Journal of Parallel Programming.*