Universidade Federal do Amazonas
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de Pós-Graduação em Informática

Cache Comprimido Adaptativo via SOM (Mapas Auto-organizáveis)

Anderson Farias Briglia

Manaus – Amazonas
2009

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

Anderson Farias Briglia

Cache Comprimido Adaptativo via SOM (Mapas Auto-organizáveis)

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Departamento de Ciência da Computação da Universidade Federal do Amazonas, como requisito para obtenção do Título de Mestre em Informática.

Orientador: Dr. Edward Moreno

Anderson Farias Briglia

Cache Comprimido Adaptativo via SOM (Mapas Auto-organizáveis)

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Departamento de Ciência da Computação da Universidade Federal do Amazonas, como requisito para obtenção do Título de Mestre em Informática.

Banca Examinadora

Prof. Edward David Moreno Ordonez, Ph.D. – Orientador
Departamento de Ciência da Computação – UFS/DCOMP

Prof. Raimundo da Silva Barreto, Ph.D. – Co-Orientador
Departamento de Ciência da Computação – UFAM/PPGI

Prof. Carlos Alberto Estombelo, Ph.D. – Banca
Instituto Federal do Amazonas - Manaus

Prof. Gabriel Pereira da Silva, Ph.D. – Banca
Departamento de Ciência da Computação, Instituto de Matemática – UFRJ

Manaus – Amazonas
2009

Aos meus pais Francisco e Irene.

Agradecimentos

Aos meus pais, por sempre acreditarem que o estudo é o melhor caminho para o crescimento.
À minha esposa, Mariana Paraguassu, pela alegria, carinho e total apoio nas horas mais turbulentas durante o curso de mestrado.
Aos meus orientadores, Edward Moreno e Raimundo Barreto, por me ajudarem a terminar esta dissertação.
Ao meu ex-chefe e colega, Ilias Biris, pelo incentivo de ingressar no mestrado e orientação de como iniciar o trabalho.
Ao colega indiano Nitin Gupta, por ter disponibilizado o projeto de Cache Comprimido na Internet, o qual foi baseado esta dissertação.
Aos meus colegas de trabalho Maurício Lin e Francisco Alecrim por me darem total apoio ao usar o trabalho deles afim de auxiliar o desenvolvimento desta dissertação.
A todas as pessoas que me ajudaram de alguma forma na conclusão deste trabalho, muito obrigado.

Só é útil o conhecimento que nos torna melhores.

Sócrates

Resumo

Sistemas embarcados geralmente possuem limitações de memória e processamento. Com o aumento do uso dos dispositivos móveis é mais comum encontrar sistemas operacionais embarcados, como o Linux, sendo usados para executar aplicações multimídia, acessar a Internet, etc. Aumentar a memória física de um dispositivo pode ser custoso e após projetado é inviável fazer mudanças físicas no dispositivo, por esse motivo implementar mecanismos via software pode ser uma possível solução mais acessível e econômica à maioria dos usuários.

O Cache Comprimido é uma técnica que permite comprimir a memória das aplicações, possibilitando um aumento da memória disponível sem a necessidado de trocar o hardware. Porém, a falta de um Cache Comprimido Adaptativo faz com que essa solução tenha um escopo reduzido de benefícios.

Nesta dissertação foi desenvolvido um programa que utiliza a classificação de padrões de consumo de memória usando (SOM - Self Organized Maps – Mapas Auto-Organizáveis), para implementar um Cache Comprimido Adaptativo, que consiga se adequar ao consumo da memória em tempo de execução.

Palavras-chave: Linux, Gerenciamento de Memória, Classificação de Padrões e Redes Neurais, Cache Comprimido.

Abstract

Embedded systems usually have memory and processing limitations. Due to increased use of mobile devices, it is common to see embedded systems being used to run multimedia applications, access the Internet, etc. Increasing amount mobile physical memory could be expensive and after the hardware design it becomes unfeasible, due to this implementing a software solution could be more viable in terms of costs.

Compresse Cache is a solution which provides to applications memory compression, increasing the available memory without making hardware changes. But, the lack of an Adaptive Compressed Cache implementation makes this solution's scope smaller in terms of benefits.

A program which uses memory consumption patterns through SOM – Self Organized Maps was developed in this dissertation, to implement an Adaptive Compressed Cache. This approach makes Cache Compression behavior aligned with memory consumption in real time.

Sumário

1 Introdução **14**
1.1 Motivação . . . 16
1.2 Objetivos . . . 17
1.3 Organização da dissertação . . . 18

2 Cache Comprimido **20**
2.1 A memória virtual do Linux . . . 20
2.1.1 O Swap Cache . . . 21
2.1.2 O Page Cache . . . 21
2.2 Cache Comprimido . . . 22
2.3 Cache Comprimido para Linux *kernel* 2.6.x . . . 24
2.3.1 Projeto da Implementação . . . 24
2.3.2 Armazenamento das Páginas Comprimidas . . . 26
2.3.3 Operações de Inserção e Remoção das Páginas Comprimidas . . . 27
2.4 Experimentos com Cache Comprimido . . . 28
2.4.1 Suite de Testes e Metodologia . . . 29
2.4.2 Ajustando o kernel . . . 30
2.4.3 Testes de Comportamento do Consumo de Memória . . . 31
2.4.4 Testes com o Mibench . . . 37
2.4.5 Testes de Performance . . . 41
2.5 Sumário . . . 42

3 Redes Neurais e Mapas Auto-organizáveis no Cache Comprimido **44**
3.1 Redes Neurais Artificiais . . . 44
3.2 Mapas Auto-Organizáveis . . . 46
3.3 Método de classificação de consumo de memória baseado em SOMs . . . 49
3.4 Coleta dos dados . . . 51
3.5 Treinando a rede neural e criando o SOM . . . 51
3.6 Classificando as aplicações . . . 54
3.7 Sumário . . . 59

4 Cache Comprimido Adaptativo **61**
4.1 Adaptatividade no Cache Comprimido . . . 61
4.1.1 Daemon para adaptar o Cache Comprimido . . . 67
4.2 Sumário . . . 71

5 **Conclusão e Trabalhos Futuros** **73**
5.1 Conclusão 73
5.2 Trabalhos Futuros 75
Referências 76

6 **Apêndices** **80**
6.1 Apêndice - A 80
6.2 Apêndice - B 80

7 **Anexos** **83**
7.1 Anexo - I 83

Lista de Figuras

1.1 Hierarquia de memória com cache comprimido 17

2.1 Campos da estrutura `swp_entry_t` 21
2.2 Estrutura de swaps utilizando ramzswap como dispositivo de swap virtual. 24
2.3 Cabeçalho de um objeto manipulado pelo alocador do Cache Comprimido – xvMalloc. 26
2.4 Cabeçalho de um objeto livre do alocador do Cache Comprimido – xvMalloc. 27
2.5 Radix tree padrão, antes da compressão da página. 28
2.6 Browser: memória livre (MemFree) ao longo do teste. A linha pontilhada representa o consumo quando o CC estava configurado para 10MB de tamanho. 32
2.7 Browser: A linha pontilhada representa o consumo quando o CC estava configurado para 10MB de tamanho e swappiness = 60. 33
2.8 Canola: memória livre (MemFree) ao longo do teste. A linha pontilhada representa o consumo quando o CC estava configurado para 10MB de tamanho. 34
2.9 Canola: memória livre com Cache Comprimido = 10MB e swappiness = 60. 35
2.10 PDF: comportamento de consumo da memória para os 3 cenários. 36
2.11 Teste com o benchmark da categoria Automotive 38
2.12 Teste com o benchmark da categoria Consumer 39
2.13 Teste com o benchmark da categoria Network 39
2.14 Teste com o benchmark da categoria Office Automation 40
2.15 Teste com o benchmark da categoria Security 40
2.16 Teste com o benchmark da categoria Telecom 41
2.17 Tempo de de execução do fillmem para alocar 50 MB. 42

3.1 Exemplo de rede não treinada.[19] 47
3.2 Um exemplo de SOM treinado para classificação de cores.[19] 47
3.3 Os vetores são mapeados em um espaço bidimensional representado pela grade de neurônios [19]. 51
3.4 Cenário Alpha: SOM para rede neural treinada sem Cache Comprimido. . 52
3.5 Cenário Beta: SOM para rede neural treinada com Cache Comprimido de tamanho igual a 10MB. 53
3.6 Cenário Gama: SOM para rede neural treinada com Cache Comprimido = 10MB e swappiness = 60. 53
3.7 Cenário Alpha: regiões onde a aplicação 'Browser' visitou no SOM. 55
3.8 Cenário Beta: regiões onde a aplicação 'Browser' visitou no SOM. 55
3.9 Cenário Gama: regiões onde a aplicação 'Browser' visitou no SOM. 56
3.10 Cenário Alpha: regiões onde o 'Canola' visitou no SOM. 56

3.11 Cenário Beta: regiões onde o 'Canola' visitou no SOM. 57
3.12 Cenário Gama: regiões onde o Canola visitou no SOM. 57
3.13 Cenário Alpha: regiões onde o 'PDF viewer' visitou no SOM. 57
3.14 Cenário Beta: regiões onde o 'PDF viewer' visitou no SOM. 58
3.15 Cenário Gama: regiões onde o 'PDF viewer' visitou no SOM. 58

4.1 Cenário Alpha: imagem que representa o SOM para o Browser e sua matriz de frequências. 62
4.2 Cenário Beta: imagem que representa o SOM para o Browser e sua matriz de frequências. 63
4.3 Cenário Gama: imagem que representa o SOM para o Browser e sua matriz de frequências. 63
4.4 Cenário Alpha: imagem que representa o SOM para o media player Canola e sua matriz de frequências. 64
4.5 Cenário Beta: imagem que representa o SOM para o media player Canola e sua matriz de frequências. 64
4.6 Cenário Gama: imagem que representa o SOM para o media player Canola e sua matriz de frequências. 65
4.7 Cenário Alpha: imagem que representa o SOM para o visualizador de PDF's e sua matriz de frequências. 65
4.8 Cenário Beta: imagem que representa o SOM para o visualizador de PDF's e sua matriz de frequências. 66
4.9 Cenário Gama: imagem que representa o SOM para o visualizador de PDF's e sua matriz de frequências. 66
4.10 Memória livre (MemFree) ao longo do teste com o CC adaptativo. Os picos de memória livre indicam que quando o perfil de consumo de memória é trocado, existe uma liberação maior e um aumento da memória livre total. 71
4.11 Memória livre (MemFree) para cada aplicação sem CC. 72

Lista de Tabelas

2.1 Dados do Cache Comprimido dos cenários Beta e Gama para o Browser. . 33
2.2 Dados do Cache Comprimido dos cenários Beta e Gama para o Canola. . . 35

Capítulo 1

Introdução

A grande maioria dos sistemas embarcados têm como principal característica seu uso para uma tarefa específica, porém, existem sistemas embarcados que não necessariamente são específicos [29].Como é o caso do sistema operacional *Linux* analisado nesta dissertação de mestrado. O avanço da minituarização dos componentes eletrônicos e a queda do preço dos equipamentos, fez com que o poder de processamento de um PDA (*Personal Data Assistance*), por exemplo, caísse consideravelmente, permitindo que esse tipo de dispositivo possa ter sistemas embarcados mais sofisticados.

Com o poder de processamento dos dispositivos móveis aumentando, as aplicações disponíveis aos usuários acompanham este crescimento e se tornam mais complexas. Como características inerentes de qualquer sistema embarcado, pode-se citar o processamento limitado, consumo de energia bem restrito e problemas relacionados à memória disponível para as aplicações. Nesta dissertação é desenvolvida e analisada uma solução para o problema de memória dos sistemas embarcados. Não faz parte do escopo deste trabalho o desenvolvimento de técnicas para melhorar consumo de potência ou aumentar o poder de processamento dos sistemas embarcados.

Segundo Rodrigo Castro [9], uma das possíveis soluções para o problema de escassez de memória nos sistemas embarcados é a utilização de algoritmos de compressão. A compressão tem se mostrado uma técnica eficiente para otimizar o uso da memória em sistemas embarcados[6, 9, 10, 16, 26]. Ela também tem sido utilizada como um meio de melhorar o uso da memória cache, reduzindo o consumo de potência e melhorando a performance do sistema de memória, visto que a memória cache é, geralmente, mais rápida do que a memória principal de um computador ou de um dispositivo móvel. E sendo uma memória de acesso rápido, o processador gasta menos tempo para acessá-la, beneficiando também o consumo de potência total do sistema.

O *Cache* Comprimido (CC) é uma técnica que adiciona um novo nível na hierarquia

de memória do Linux [9, 10, 16]. O CC é usado para aprimorar o tempo de acesso às páginas de memória no *kernel* (ou núcleo) do Linux, armazenando mais páginas na memória RAM (*Random Access Memory*), e reduzindo o número de páginas que vão para a área de *swap*[1] ou que seriam descartadas caso o sistema não a possuísse. É sabido que a área de *swap*, em geral, é muito mais lenta que a memória principal, e custosa com relação ao consumo de energia pois na maioria dos casos está associada a dispositivos de bloco, como por exemplo, um disco rígido. E ainda tem-se o problema de que em sistemas embarcados geralmente essa área não está presente ou não possui o tamanho ideal.

Na implementação usada neste trabalho, o tamanho da área de CC possui influência na quantidade de memória disponível para as aplicações. Se o tamanho do CC, ou seja, da memória alocada para ele, for muito grande, podem ocorrer situações de falta de memória, resultando em travamento do sistema ou das aplicações que estão sendo executadas no momento. Hoje, a escolha de um tamanho para o CC é empírica. Ou seja, é escolhido um tamanho baseado em experimentos (como os vistos nas seções posteriores deste trabalho), ou simplesmente na estimativa de memória total livre após a alocação do CC. Um cenário ideal para calcular o tamanho do CC é que o perfil das aplicações, ou melhor, das alocações de memória das aplicações, seja levando em conta quando o tamanho for escolhido.

Como forma de estimar o comportamento da memória, alinhando assim o tamanho do CC a ser utilizado com o perfil de como as aplicações utilizam a memória, é utilizado o trabalho de mestrado de Maurício Lin [2, 19]. Neste trabalho foi implementado um esquema de classificação de padrões de consumo de memória utilizando Mapas Auto-Organizáveis (do inglês SOM: *Self-Organized Maps*). Basicamente é feito um "mapa" de como a memória foi utilizada por determinada aplicação. Nesta dissertação de mestrado, esse mapa é utilizado como entrada para a ferramenta que irá calcular o tamanho do CC, baseado no perfil das aplicações testadas.

Neste trabalho é implementada uma versão do Cache Comprimido Adaptativo. Utilizando a classificação dos padrões de consumo de memória de aplicações escolhidas, o tamanho da área de memória comprimida é determinado. Espera-se com essa abordagem ter um ganho na memória virtual livre total para as aplicações, possibilitando assim que mais processos possam rodar simultaneamente no sistema.

[1]Área de swap é uma área reservada no sistema de arquivos do Sistema Operacional, a fim de ser utilizada como uma extensão de memória principal.

1.1 Motivação

A memória é um dos componentes críticos que possuem maiores restrições quando usada em sistemas embarcados. Por outro lado, os sistemas têm sido aperfeiçoados através de técnicas sofisticadas, algoritmos complexos e suporte a tempo-real. Como consequência, as aplicações para sistemas embarcados têm se tornado maiores e com um volume de dados manipulados sempre crescente.

Dado esse cenário, é muito importante definir mecanismos que aperfeiçoem a utilização de memória e/ou a performance das aplicações quando estas fazem uso intenso da memória do dispositivo.

Em [9], foi implementada uma versão do Cache Comprimido Adaptativo para a versão 2.4.x do *kernel* do Linux. Os mecanismos de falta de memória encontrados na versão 2.4.x são diferentes do que temos hoje (ex. mapeamento de páginas anônimas, Out-of-memory killer, etc), nas versões mais atuais (2.6.x, por exemplo).

Um ponto motivacional deste trabalho é a realização de uma implementação do Cache Comprimido Adaptativo para as versões mais atuais do *kernel* do Linux, disponibilizando assim seu uso em dispositivos móveis mais atuais, baseados neste sistema operacional. A implementação usada neste trabalho está em desenvolvimento [12], e a idéia desta dissertação é contribuir nesse projeto *Open Source*, além de melhorar o desempenho das aplicações quando executadas em ambiente de Linux embarcado, propondo uma nova metodologia de gerenciamento de memória no *kernel* do Linux.

Na versão do Cache Comprimido Adaptativo apresentada em [12], não foi utilizada nenhuma técnica para estimar o comportamento do tamanho da memória comprimida. A escolha de uma heurística inadequada pode impactar no desempenho de todo o sistema, pois a relação memória comprimida X memória não-comprimida é muito importante quando se trata de adaptatividade. Estudos realizados anteriormente em [9], mostram que uma escolha errada no tamanho da memória comprimida pode criar *overheads* desnecessários ao sistema, acarretando em perda de performance. Assim, a implementação de um esquema de adaptatividade utilizando SOM se torna um outro ponto de motivação do trabalho.

Por último, e não menos importante, um outro ponto de destaque da motivação deste trabalho é que a pesquisa realizada utilizando classificação de padrões de consumo de memória e Cache Comprimido é pioneira nesta arquitetura. Testes com a última versão do Cache Comprimido foram feitos em versões do Linux para desktops e netbooks, mas ainda

não foram realizadas medições em um ambiente de Linux embarcado para arquitetura ARM (*Advance Risc Machine*).

1.2 Objetivos

O propósito principal deste trabalho é desenvolver um sistema de memória comprimida, utilizando como base a implementação *Open Source* encontrada em [12], e que implemente o conceito de adaptatividade do tamanho do CC, baseado em perfis de consumo de memória das aplicações, definidos utilizando uma rede neural e SOMs (mapas auto-organizáveis).

Experimentos realizados em trabalhos anteriores [6, 9], indicaram que o tamanho da área comprimida de memória afeta o desempenho do sistema. Espera-se que seja possível classificar o uso da memória, como mostrado em [19] e utilizar esses dados na heurística de como a área de cache comprimido deve ser dimensionada. A implementação deste trabalho não possui dependências de arquitetura, apesar de ser focada em Linux embarcado para arquitetura ARM (*Advanced RISC Machine*).

A área para memória comprimida será adicionada ao sistema existente como mostrado na Figura 1.1.

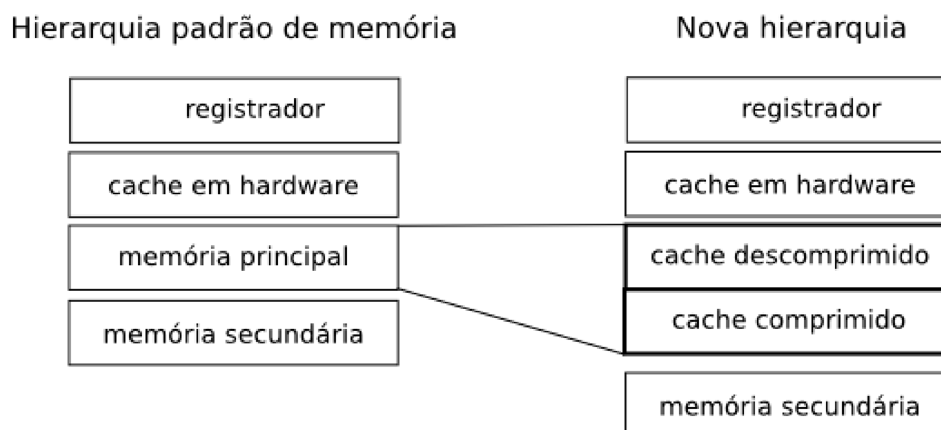

Figura 1.1: Hierarquia de memória com cache comprimido

A versão do *kernel* utilizada é a 2.6.x, e como resultado prático final um *patch* ou uma série de *patches* serão gerados, com modificações relativas à implementação do CC para a última versão do *kernel* disponível. Uma aplicação que extrai os dados providenciados pelo SOM durante a classificação do uso da memória será implementada. Essa aplicação exportará os dados necessários (como os perfis de consumo de memória de aplicações escolhidas), para o ajuste do tamanho do CC. Na Seção 3.3 é apresentada a forma de como os padrões de memória são classificados.

Para a validação da implementação são utilizados dois tipos de testes preliminares: *benchmarks* sintéticos e testes com casos de uso que simulam a utilização real do sistema, utilizando aplicações gráficas tais como: navegador *Web*, leitor de arquivos PDF, tocador de áudio e vídeo. Nos testes com *benchmarks* sintéticos, foram utilizados suites como o MemTest [24] e o MiBench [13]. O MemTest foi utilizado nos testes de performance pois sua principal característica é a realização de operações com a memória virtual do Sistema Operacional Linux, mais especificamente, com alocações e desalocações de porções de memória. O MiBench é bastante utilizado como *benchmarking* de sistemas embarcados e provê um conjunto de módulos que podem ser utilizados como parâmetros de referência tanto para medidas relacionadas à arquitetura de *hardware* quanto a medidas com operações de memória, ou operações multimídia. Basicamente os testes usando essas ferramentas fazem uso intenso da memória RAM. No caso dos testes utilizando casos de uso, é utilizada uma ferramenta de automação chamada XAutomation [25], para criar uma interação com o ambiente gráfico do sistema, simulando um uso real das aplicações mencionadas anteriormente, enquanto dados referentes à utilização da memória livre são armazenados. Com estes dois tipos de testes, espera-se ter dados suficientes para apontar o impacto da utilização de compressão da memória utilizando SOM, com a heurística para adaptar o tamanho da memória comprimida.

1.3 Organização da dissertação

O Capítulo 1 trata da introdução, motivação e objetivos desse trabalho. Neste capítulo o leitor será apresentado ao problema e à solução proposta pelo autor.

O Capítulo 2 descreve o estado atual da implementação do Cache Comprimido utilizada neste trabalho. Detalhes de implementação e testes também são descritos e discutidos.

O Capítulo 3 apresenta o trabalho realizado em [2] e [19]. Estes trabalhos propõem uma classificação de padrões de consumo de memória utilizando redes neurais e mapas auto-organizáveis.

No Capítulo 4 o leitor terá acesso aos resultados obtidos com os testes utilizando Cache Comprimido e Mapas Auto-organizáveis. Também será apresentada a solução encontrada para implementar a adaptatividade do Cache Comprimido quando usado em Linux embarcado.

No Capítulo 5 são apresentados os trabalhos futuros para o Cache Comprimido e a

conclusão encontrada após a relização dos testes e implementação envolvida.

Capítulo 2

Cache Comprimido

Esta seção tem o propósito de apresentar os estudos e as pesquisas realizadas até o momento e que são requisitos relevantes para o desenvolvimento da proposta descrita na Seção 1.2.

Inicialmente é apresentado o estado da arte, onde alguns trabalhos relacionados são analisados. Em seguida, a atual implementação do CC é discutida, para a versão 2.6.x do *kernel* do Linux encontrada em [12]. Também são exibidos e descritos alguns testes realizados com essa versão e apresentados em [6].

2.1 A memória virtual do Linux

Páginas físicas são a unidade básica do gerenciamento de memória [21] e o MMU (*Memory Management Unit*) é o hardware responsável por traduzir endereços virtuais em reais das páginas de memória, e vice-versa.

No gerenciamento da memória virtual, duas listas do tipo LRU (*Last Recently Used*) são utilizadas afim de classificar as páginas: LRU para páginas ativas e uma LRU para páginas inativas. Quando o sistema precisa alocar novas páginas, elas são inicialmente retiradas da lista LRU de páginas inativas. O algoritmo responsável por selecionar e liberar as páginas é chamado de *Page Frame Reclaiming Algorithm* - PFRA.

Para identificar cada tipo de página, *flags* são utilizadas na estrutura de dados que as implementam. Para diferenciar as páginas comprimidas das páginas comuns, a implementação atual do Cache Comprimido adiciona uma *flag* na estrutura de dados da página.

Quando o sistema está sob pressão de memória, ou seja, há menos memória disponível que o necessário, o PFRA libera as páginas de acordo com a sua classificação:

- Páginas do *Swap-cache* são escritas na área de *swap* disponível.
- Páginas "sujas"do *Page cache* são escritas no *filesystem* utilizando o procedimento específico de escrita.
- Páginas "limpas"do *Page Cache* são simplesmente liberadas da memória, e ficam disponíveis para serem utilizadas novamente por outros processos.

2.1.1 O Swap Cache

Este é o cache para páginas anônimas. Toda as páginas do *swap cache* são parte de um único **swapper_space**, a estrutura que agrupa todas as páginas que podem ir para a área de *swap*. Uma outra estrutura de dados, chamada *radix tree* é utilizada para manter todas as páginas do *Swap cache* e torna a busca por páginas muito mais eficiente. Essa estrutura de dados é a implementação de uma árvore-B em que cada nodo possui um vetor de 64 posições que apontam para o endereço virtual da página. Quando uma página sai do *Swap Cache* e vai para a área de *swap*, esse endereço é atualizado e é assim que o kernel do Linux consegue recuperar uma página quando é requisitada.

O campo **swp_entry_t** da estrutura de dados da área de *swap* é utilizado como chave-de-busca quando uma página do *swap cache* é procurada. Esta estrutura é usada para identificar onde a página requisitada se encontra no dispositivo de bloco utilizado pelo seu *swap*.

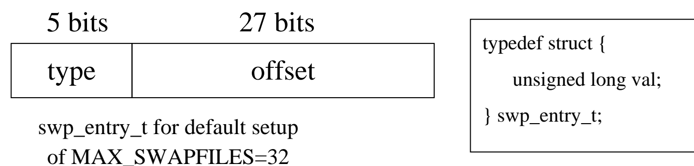

Figura 2.1: Campos da estrutura `swp_entry_t`

Na Figura 2.1, **'type'** identifica em qual área de *swap* a página de se encontra. No sistema operacional Linux, podem haver até 32 *swaps* simultâneos.

2.1.2 O Page Cache

Este é o cache utilizado para armazenar as páginas do *filesystem*. Ou seja, todo arquivo aberto pelas aplicações possui páginas de memória alocadas e armazenadas no *Page Cache*. Assim como o *Swap Cache*, este *cache* também possui uma *radix tree* que armazena as referências, ou melhor, os ponteiros para as páginas de um determinado arquivo presente no sistemas de arquivos, que esteja sendo utilizado. O valor de *offset* dentro do arquivo é utilizado como chave-de-busca para localizar tais páginas. Cada arquivo aberto possui

uma *radix tree* própria, com os nodos da árvore apontando para as páginas pertencentes ao arquivo.

2.2 Cache Comprimido

Em um sistema com cache comprimido, a memória é dividida em duas grandes porções: memória comprimida e não-comprimida [8, 10, 16]. A área de memória comprimida geralmente é alocada onde antes existia memória não-comprimida. A relação dos tamanhos da memória comprimida e não-comprimida deve ser avaliada pois os dois tamanhos interferem em como cada porção de memória se comporta, dependendo do consumo imposto pelas aplicações.

Muitos pesquisadores [16, 26] têm investigado o uso de compressão para reduzir as operações de *paging*, introduzindo um novo nível na hierarquia de memória. Armazenar as páginas de memória em uma área comprimida, naturalmente aumenta o tamanho efetivo da memória e também diminui o acesso à dispositivos de memória secundária [9], que é muito mais lenta que a principal. Apesar dessa diferença de velocidade entre a memória principal e a memória secundária ser grande, quando se leva em conta os sistemas embarcados, ela é menor. Geralmente, os sistemas embarcados não possuem memória secundária armazenada em um disco rígido, outros dispositivos são utilizados nesse caso. Memórias do tipo *compact flash* e cartões MMC, são os principais dispositivos encontrados hoje no mercado que são utilizados como memória secundária em dispositivos móveis [1]. Mesmo tendo a diferença de velocidade de acesso entre a memória principal e a secundária menor, os sistemas embarcados possuem um grande requisito relacionado ao tamanho dessa memória. Assim, o uso de compressão é justificado pois, como será discutido posteriormente, é capaz de aumentar o tamanho efetivo da memória disponível para as aplicações.

As abordagens de cache comprimido baseadas em *software*, ou seja, aquelas que não propõem alteração de *hardware*, podem ser estáticas ou dinâmicas. As abordagens estáticas [7, 28, 30] são caracterizadas por não possuírem uma heurística de redimensionamento do cache comprimido, diferente das abordagens dinâmicas. Nesse segundo tipo, existe um algoritmo que determina quando o cache comprimido deve alterar seu próprio tamanho, geralmente baseado no consumo de memória.

Os primeiros estudos que utilizavam cache comprimido para reduzir a paginação em disco, ou seja, a quantidade de acessos de leitura/escrita, foram feitos por Appel e Li [3] e Paul R. Wilson [27], em 1991. Outro pesquisador, chamado Douglis[10], obteve algumas

[1] A velocidade de leitura de um cartão MMC comum, chega à taxa de 416 Mbits/seg [4]. A velocidade das memórias RAMs variam, mas as mais comuns possuem um tempo de acesso de 80-90 ns.

melhorias na performance do sistema, usando uma implementação de cache comprimido adaptativo no sistema operacional Sprite. Porém, Douglis não conseguiu concluir se o uso de cache comprimido é útil ou não pois seus testes apresentaram resultados divergentes, com melhorias em alguns casos e pioras em outros.

Sendo o trabalho de Douglis inconclusivo, muitos outros pesquisadores se ocuparam em estudar o caso. Em 1999, Kaplan[16] chegou à conclusão que a compressão do cache pode reduzir os custos das operações de *paging*[2]. Kaplan ainda confirmou o que Douglis[10] verificou anteriormente: cache comprimido estático beneficia menos do que um cache comprimido com adaptatividade. Alguns trabalhos correlatos também foram desenvolvidos em 1999. Como apresentado em [7], não se trata especificamente da memória cache comprimida, mas sim da compressão da área de *swap* do sistema. Neste trabalho é apresentada uma versão de compressão da memória voltada às páginas que podem ser selecionadas para o *swap*, salvando algum espaço no disco e diminuindo as operações de E/S no *swap*. Testes concluíram que a compressão da área de *swap* pode aumentar a velocidade das aplicações em 20%. Outros trabalhos também apontaram ganhos na performance de aplicações, como apresentado em [26], no quais a compressão da memória melhorou a performance de aplicações reais, com índices de 130 a 55 por cento. Nos testes com *benchmarks* e algumas aplicações de simulação, como o NS2 (*Network Simulator*), foi verificado que um nível de memória comprimida adicionada ao sistema garante a possiblidade da execução de aplicações que necessitem de um *working set* maior que a memória física disponível.

Um dos últimos trabalhos sobre Cache Comprimido foi implementado por Rodrigo Castro [8, 9]. Utilizando a versão 2.4.x do Linux *kernel*, vários testes foram realizados, com e sem adaptatividade. Os resultados mostraram que os ganhos são maiores quando existe alguma heurística de adaptatividade presente no CC. Como dito anteriormente, um dos objetivos deste trabalho também é propor uma heurística de adaptatividade para o CC. Porém, pretende-se usar um método mais científico e menos empírico. Através dos SOMs e da classificação de padrões de consumo de memória proposta em [19], espera-se traçar um "perfil" de consumo de cada aplicação e usá-lo para definir uma heurística de adaptatividade do CC.

A maioria dos trabalhos discutidos anteriormente, são anteriores às versões mais novas do *kernel* do Linux. Daquela época para os dias atuais, o *kernel* sofreu muitas melhorias e o esquema utilizado para o CC na versão usada nesse trabalho é diferente. Assim faz-se necessário um *overview* da versão do Cache Comprimido usada neste trabalho,

[2]São operações de E/S no *Page Cache*[21]. Este cache é utilizado para reduzir o número de E/S dos discos, aumentando assim a performance. Páginas do *Page Cache* estão na memória RAM.

implementada por Nitin Gupta [6, 12].

2.3 Cache Comprimido para Linux *kernel* 2.6.x

Dados experimentais avaliados em um sistema utilizando Cache Comprimido [6], mostram que não só as taxas de E/S podem ser melhoradas, como também todo o comportamento do sistema, especialmente em situações de memória crítica, como por exemplo, adiando a chamada do *Out-Of-Memory killer* – OOM.

A implementação atual do Cache Comprimido tira vantagem do sistema de *swap*, adicionando uma área de *swap* virtual como área de armazenamento das páginas comprimidas. Utilizando o algoritmo de compressão usando em sistemas JFFS2 (LZO [23, 31]), páginas alocadas na memória selecionadas para irem para o *swap* são comprimidas e enviadas à uma partição de swap virtual através de um dispositivo virtual de bloco chamado *ramzswap*. Basicamente, o *ramzswap* "engana"o *kernel* do Linux e faz com que seja adicionada uma área de *swap*, que na realidade não é uma partição em um disco, mas sim uma área da memória RAM do dispositivo. O tamanho da memória alocada para essa área é o tamanho do Cache Comprimido.

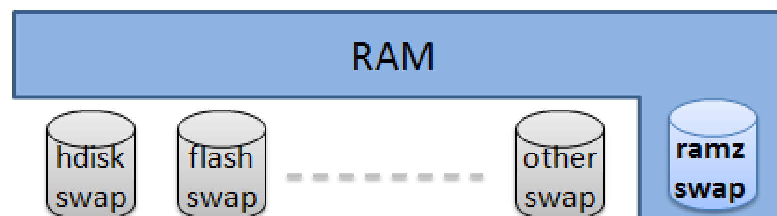

Figura 2.2: Estrutura de swaps utilizando ramzswap como dispositivo de swap virtual.

Uma outra vantagem da versão avaliada do CC é que um *backing-swap device* pode ser configurado, ou seja, pode ser adicionado uma partição real de swap (em disco, por exemplo), a qual as páginas comprimidas podem ir caso falte espaço no *ramzswap*, o dispositivo de bloco virtual criado pelo CC.

2.3.1 Projeto da Implementação

Utilizando um dispositivo virtual de bloco como partição para *swap*, o Cache Comprimido usado neste trabalho aproveita todo o mecanismo de *swapping* implementado no *kernel* do Linux. Dessa forma, o *Swap Cache* (discutido anteriormente), assim como a seleção de páginas para serem comprimidas, são utilizadas de forma natural pelo algoritmo de Gerenciamento de Memória do *kernel* do Linux.

Diferente da versão avaliada em [6], esta nova versão do Cache Comprimido não é intrusiva e não foi necessário alterar o kernel do Linux. Porém, essa não intrusão custou a compressão de páginas não-anônimas, ou seja, aquelas páginas pertencentes ao Page Cache. Somente páginas que podem ir para uma área de swap são comprimidas na versão atual do Cache Comprimido.

Afim de obter maior eficiência e evitar alguns problemas como a fragmentação, foi desenvolvido um escalonador de memória especialmente para o Cache Comprimido. Esse escalonador, chamado xvMalloc[12], é baseado no escalonador TLSF (*Two Level Segregate Fit*)[22], desenvolvido para sistemas de tempo real.

O xvMalloc foi desenvolvido levando em conta que os alocadores de memória de propósito geral, geralmente são projetados para trabalharem com requisições em número de páginas maiores que 4 *kilobytes*, que é o tamanho padrão para páginas de memória no Linux. Dessa forma, quando utilizados para alocarem porções muito menores, como 32 *bytes* ou 3/4 do tamanho de uma página, não se obtém uma boa performance e muitas vezes acontece a fragmentação.

Herdadas do TLSF[22], o xvMalloc[12] possui algumas características que são muito úteis para o Cache Comprimido:

- **Tempo de resposta constante:**[22] O pior caso de tempo de execução (WCET) do xvMalloc (e do TLSF), para alocar e desalocar uma porção de memória é constante, ou seja, $O(1)$.
- **Eficiência no consumo de memória:**[22] TLSF vem sido testado em várias situações de consumo de memória, em sistemas operacionais de tempo-real e
- **Tamanho dos metadados:**[12] em um sistema de 64 bits, o xvMalloc gasta somente 4 bytes por objeto são necessários para localizá-lo e guardar outras informações imprescendíveis.
- **Tamanho do objeto:**[12] Cada objeto (que contém uma página comprimida, ou uma parte dela), possui seu tamanho exato armazenado no cabeçalho, economizando assim mais 2 bytes por objeto. O xvMalloc ainda possui um método chamado *xvGetObjectSize(obj)* que retorna o tamanho do objeto comprimido, esse método é utilizado pelo decompressor.

Essas características são importantes para o Cache Comprimido pois são muito utilizadas pelo sistema proposto. Sendo o *ramzswap* nada mais que uma área de memória, é importante que a velocidade de leitura/escrita (nessa caso, de alocação e liberação),

seja tão rápida quanto o acesso a uma partição de swap real. A baixa fragmentação da memória também é importante pois no caso do Cache Comprimido, estamos falando de porções de memória que podem ser muito menos que o tamanho padrão de uma página (que é de 4 KB, no Linux). Assim, como as páginas de memória possuem tamanhos bastante variados, é importante utilizar técnicas que garatem uma baixa fragmentação, melhorando assim a velocidade na recuperação de páginas comprimidas ou na liberação de espaços vazios.

Na versão anterior do Cache Comprimido apresentada em [6], o tamanho dos metadados era um problema pois gastava-se muitos bytes para endereçar e guardar as informações das páginas comprimidas. Na versão mais atualizada, usada nesse trabalho, os metadados gastos para endereçar as páginas comprimidas ou para indicar espaços vazios na memória, sofreram alterações e a estrutura dos objetos é discutida na próxima seção.

2.3.2 Armazenamento das Páginas Comprimidas

As páginas comprimidas são armazenadas em estruturas de dados que o xvMalloc gerencia. Essas estruturas são projetadas de modo que se agrupem as páginas de memória que estão livres.

O cabeçalho dos objetos que o xvMalloc manipula está representado na Figura 2.3.

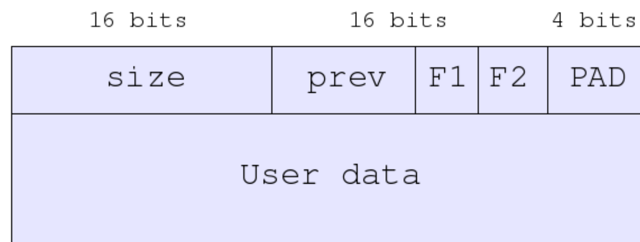

Figura 2.3: Cabeçalho de um objeto manipulado pelo alocador do Cache Comprimido – xvMalloc.

Onde, cada campo significa:

- *Size*: tamanho da página comprimida, passado para o xvMalloc pelo compressor.
- Prev: offset para a posição do bloco anterior (usado ou livre), relativo ao início do *page frame*.
- *Flags*:
  - F1: 1 se o bloco estiver usado 0, caso contrário.

– F2: 1 se o bloco anterior estiver usado 0, caso contrário.

- PAD: Não utilizado se o alinhamento for de 4 bytes.

O tamanho da estrutura de dados que representa um cabeçalho de uma página comprimida ou porção dela é de 4 bytes.

Para objetos que estão livres, ou seja, podem ser alocados pelo alocador xvMalloc para comportar uma página comprimida, ou uma porção dela, a estrutura de dados que os representa é mostrada na Figura 2.4.

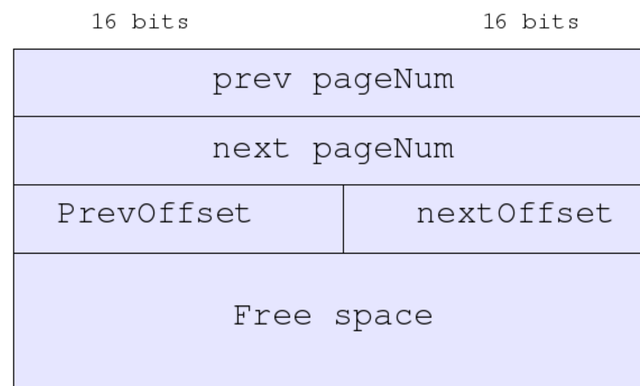

Figura 2.4: Cabeçalho de um objeto livre do alocador do Cache Comprimido – xvMalloc.

Na figura 2.4, existem campos que apontam para a próximo e o anterior endereços de página. Esses campos são utilizados para acessar rapidamente uma página quando é requisitada pelo kernel do Linux. Cada página é identificada pela tupla <*pageNum, offset*>.

2.3.3 Operações de Inserção e Remoção das Páginas Comprimidas

Inserção de páginas no Cache Comprimido: a página descomprimida é primeiramente comprimida numa página de *buffer*. O algoritmo de compressão utilizado é o LZO [23, 31], que já é utilizado em sistemas de arquivo do Linux que utilizam compressão. Após a compressão, o Cache Comprimido requisita ao ramzswap que armazene a página, sendo ω o tamanho da página comprimida. Nesse momento, o ramzswap solicita ao alocador xvMalloc que aloque um espaço de memória de tamanho ω. Por fim, o xvMalloc passa ao ramzswap a tupla <*pageNum, offset*> como identificador daquela página. Essa tupla substitui o endereço no nodo da *radix tree* que o PFRA gerencia.

Remoção de páginas do Cache Comprimido: do ponto de vista do PFRA e do kernel, a página comprimida tem o mesmo tipo que uma página normal. A única diferença entre a página comprimida e as páginas normais é que, a primeira está armazenada em uma

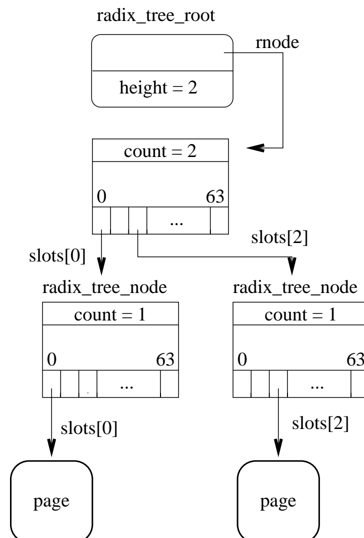

Figura 2.5: Radix tree padrão, antes da compressão da página.

partição de swap chamada ramzswap. Dessa forma, a referência à pagina comprimida, que é passada para a radix tree não se difere da referência das outras páginas, pois o ramzswap é um dispositivo de bloco como os outros. Assim, fica à cargo do Cache Comprimido e do xvMalloc que encontre a página comprimida, descomprime-a e retorne sua referência e valor ao kernel quando o mesmo requisita uma página do ramzswap.

2.4 Experimentos com Cache Comprimido

Nesta seção são apresentados os testes que foram executados utilizando Cache Comprimido em um sistema embarcado com Linux. O principal objetivo dos testes é avaliar os impactos e as características do consumo de memória, quando temos uma área comprimida adicionada ao sistema.

Como dito anteriormente, a atual implementação do Cache Comprimido trata dois tipos de páginas: páginas anônimas e páginas do *Page Cache*. O primeiro tipo de páginas foi utilizado como referência pois pode ser feito *swap*, facilitando a obtenção de dados para os gráficos.

Foram realizados testes com e sem um *swap* real, utilizando uma partição em um cartão MMC[3]. A utilização de um *swap* real se justifica pois é importante comparar o *swap* virtual com o real. Como medidas desta comparação, foram considerados os seguintes itens:

[3] Os cartões MMC são utilizados como memória secundária de dispositivos móveis, câmeras fotográficas, etc. Cartões MMC possuem uma taxa de transferência de até 52 MB/seg. A capacidade de armazenamento varia de alguns *megabytes* até cartões com mais espaço de armazenamento, na ordem de *gigabytes*.

- Quantas páginas são mantidas no Cache Comprimido e não vão para o *swap* real, evitando assim um maior *overhead* por causa das operações de E/S (Entrada e Saída de dados).
- Em quais situações o Cache Comprimido é útil para evitar que o OOM (Out-of-memory killer) seja chamado.
- Qual é a velocidade para que páginas de memória sejam comprimidas e armazenadas no ramzwap?

2.4.1 Suite de Testes e Metodologia

Utilizou-se um dispositivo móvel, parecido com um PDA que possui Linux embarcado como sistema operacional nativo. O Nokia Internet Tablet N810 [20] tem um processador ARM1136 com 400Mhz, 128MB de memória RAM e 256MB de memória *flash*, utilizada como armazenamento secundário. Ainda possui acelerador gráfico 2D/3D e dois leitores de cartões MMC/SD.

Nota-se que este dispositivo móvel, como muitos outros, foi projetado para aplicações multimídia [20]. Este tipo de aplicação demanda muito poder de processamento e quantidade de memória razoável para as aplicações. Levando isso em consideração, os testes foram realizados afim de verificar o comportamento de todo o sistema e das aplicações multimídia, quando a quantidade de memória livre disponível é muito baixa. O objetivo é verificar a performance de todo o sistema, quando se realizam operações de compressão e recuperação das páginas comprimidas.

Os casos de uso dos testes podem ser divididos em duas partes: testes que utilizam *benchmarks* sintéticos e testes que simulam a utilização real do aparelho. No primeiro grupo, existe um maior controle do consumo de memória e assim, pode-se avaliar o comportamento do Cache Comprimido quando é submetido à pressões por falta de memória. No segundo grupo, foi verificado se o Cache Comprimido exerce um *overhead* que afeta a performance das aplicações que o usuário pode executar. Espera-se que a performance das aplicações melhore, visto que o número de E/S resultantes de acessos à um dispositivo de *swap* real (dispositivo de bloco), é diminuído pois mais páginas se encontram na memória principal, através do *swap* virtual utilizando o ramzswap..

Os testes utilizando aplicações, consistem em:

- Executar de 8 a 10 instâncias do navegador WEB, simultaneamente, acessando páginas na Internet através de uma conexão de rede sem-fio.

- Tocar um arquivo de vídeo de 7,5MB.
- Realizar operações com mídias: tocar mp3, indexar novos arquivos, abrir fotos, etc.
- Abrir um documento em formato PDF.

Para fazer a interação entre o *X system* e as aplicações, foi utilizado uma ferramenta chamada Xautomation [25]. Com esta ferramenta foi possível controlar toda a movimentação do cursor na tela e os cliques. Os programas usados para realizar os testes com interação automatizada estão presentes no Anexo - I. Enquanto as aplicações são executadas, outros programas ficam responsáveis de fazer a coleta das informações referentes ao consumo de memória, através de leituras das estatísticas exportadas pelo `procfs`. É com base nesses dados que alguns gráficos foram plotados e discutidos nas seções posteriores deste trabalho.

Os testes utilizando *benchmarks* sintéticos, foram executados usando um conjunto de utilitários chamado MemTest [24]. MemTest foi implementado para avaliar a estabilidade e consistência do sistema de gerência de memória do Linux *kernel*. Do MemTest foi utilizado um utilitário chamado `fillmem`. Este utilitário faz várias alocações de memória, ou seja, de páginas de memória, disparando a necessidade de mandar algumas páginas para o ramzswap, ou seja, o *swap* virtual.

Ambos os tipos de testes, utilizaram cenários pré-definidos, dependendo do que se queria medir. Basicamente, os cenários apresentavam diferentes tamanhos da memória principal, se possuíam uma área de *swap* real ou não, ou se estavam com o Cache Comprimido habilitado ou não.

Os testes de comportamento do consumo de memória visam avaliar como a memória é consumida ao longo do tempo de duração do teste. Assim, espera-se identificar os pontos de falta de memória e a atuação do OOM *killer*. Como o Cache Comprimido se utiliza da memória principal para armazenar as páginas comprimidas no *swap* virtual, a quantidade de memória não-comprimida disponível para as aplicações é decrescida, e isso pode levar a uma chamada precoce do OOM killer.

2.4.2 Ajustando o kernel

Antes dos testes serem iniciados, alguns parâmetros do gerenciamento de memória do *kernel* devem ser ajustados, afim de garantir que os testes sejam executados com sucesso.

O sistema de *swap* do *kernel* possui alguns parâmetros que podem interferir nos valores medidos, e até na execução das aplicações. Durante os testes, foram alterados dois desses

parâmetros:

- **swappiness** [21] é um parâmetro que configura um fator de balanço que o *kernel* utiliza para manter mais ou menos páginas no *Page Cache* antes de mandar para a área de *swap*. Seu valor *default* é 60.
- **min_free_kbytes** [21] é usado como um limite mínimo de memória livre para que o sistema de gerenciamento de memória virtual do *kernel* comece a mandar páginas para o *swap*.

Se o usuário quer que o *kernel* mande mais páginas para o *swap*, o que significa em mais páginas comprimidas, quando o CC está rodando com o *swap* virtual, o parâmetro `swappiness` deve ser acrescido. Uma outra forma de mandar as páginas mais cedo para o *swap*, é aumentar o valor do `min_free_kbytes`. Durante os testes, foram utilizados valores diferentes desses dois parâmetros em cada cenário, afim de prover diferentes situações de falta de memória.

2.4.3 Testes de Comportamento do Consumo de Memória

O principal objetivo dos testes realizados nessa seção é classificar e identificar padrões de consumo de memória ao longo do tempo para determinadas aplicações.

As aplicações escolhidas para os testes foram: um navegador de Internet ou *browser*, um *media player* chamado Canola e um programa para ler arquivos no formato PDF. O ambiente de testes é um dispositivo com Linux embarcado e 128MB de memória RAM total. Para realizar as coletas em diferentes situações e ter um comparativo, foram ainda definidos três cenários:

- Cenário Alpha: Cache Comprimido desligado e kernel sem área de swap.
- Cenário Beta: Cache Comprimido ligado e configurado com área de swap virtual aproximadamente igual a 10 MB.
- Cenário Gama: Cache Comprimido ligado, configurado com área de swap virtual aproximadamente igual a 10 MB e com *swap balance* igual a 60.

Os cenários servem para identificar em quais situações é bom ter um CC configurado ou não, ou ainda qual o impacto de inserir uma área de swap na memória e comprimir as páginas. Também servem para dar um panorama e ajudar na definição de uma heurística boa para implementar a adaptatividade do CC, um dos objetivos desse trabalho.

Para padronizar os testes e simular a mesma situação nos três cenários dados, foram desenvolvidos programas para automatizar a interação com as aplicações utilizadas nos testes. O XAutomation [25], é um utilitário para Linux que serve para interagir com o sistema gráfico do sistema, fazendo o papel dos cliques de mouse, por exemplo. Assim é possível automatizar a abertura de páginas da Internet nos testes envolvendo o *browser*, automatizar também a interação com o Canola e o visualizador de PDF's.

Resultados dos testes

Os experimentos, como dito anteriormente, foram divididos em três cenários: Alpha, Beta e Gama. Os resultados para cada teste com as aplicações estão representados nos gráficos abaixo (Figuras 2.6, 2.7, 2.8, 2.9, 2.10).

Testes com o browser

Cenário Alpha versus cenário Beta

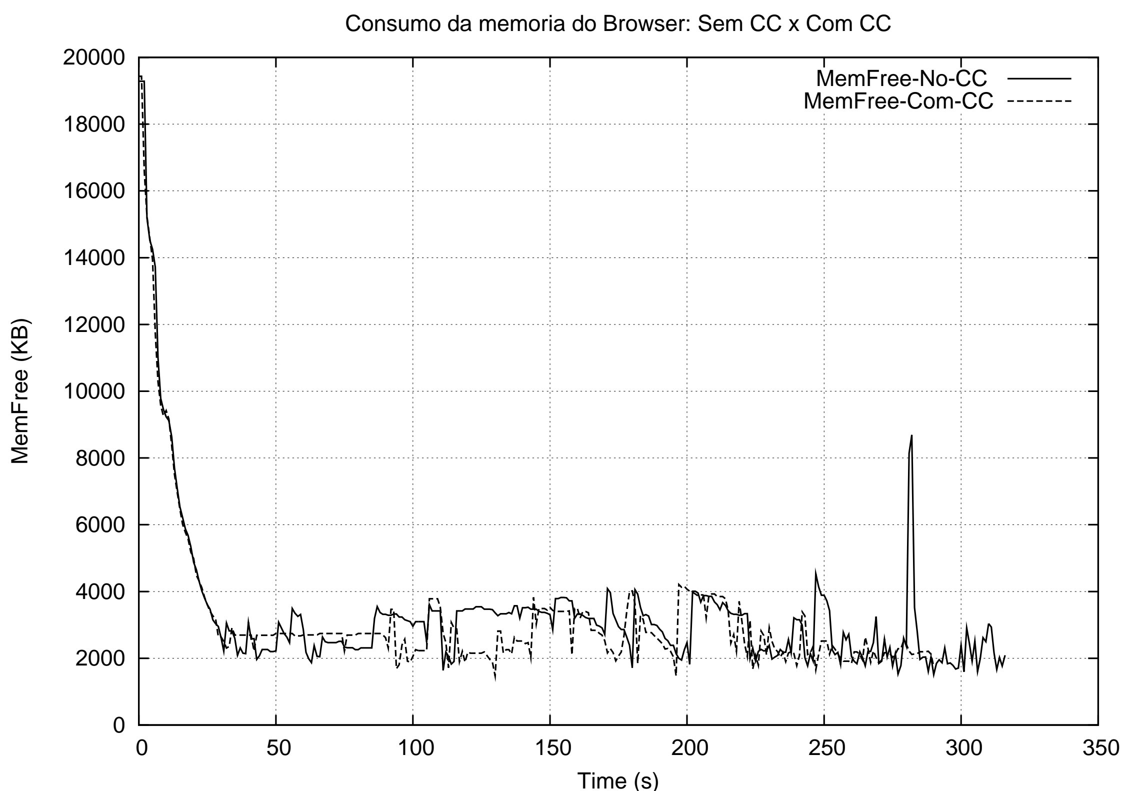

Figura 2.6: Browser: memória livre (MemFree) ao longo do teste. A linha pontilhada representa o consumo quando o CC estava configurado para 10MB de tamanho.

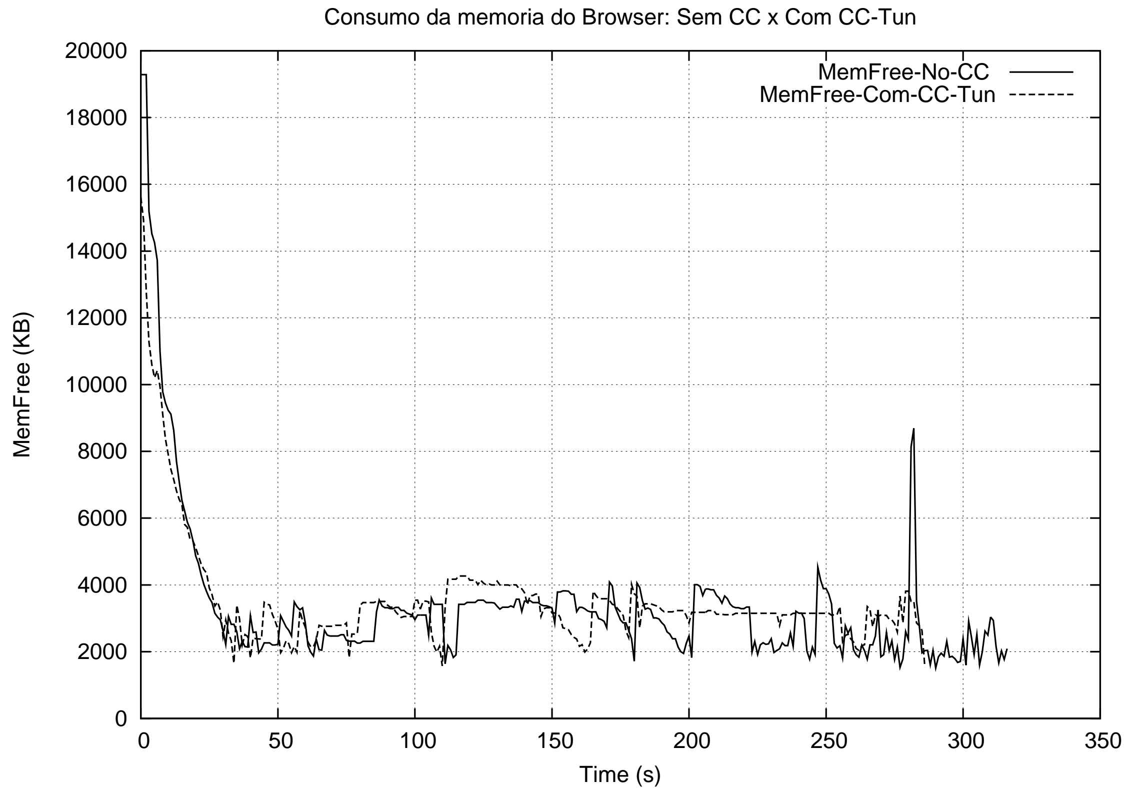

Figura 2.7: Browser: A linha pontilhada representa o consumo quando o CC estava configurado para 10MB de tamanho e swappiness = 60.

A memória livre no segundo teste é maior pois mais páginas de memória são enviadas mais cedo para a área de swap, nesse caso, o *ramzswap* do Cache Comprimido. Isso acontece pois o *kernel* foi configurado com *swappiness* = 60, e no gráfico anterior 2.7, esse valor era 0. A Tabela 2.1 mostra estatísticas lidas do Cache Comprimido, comparando o cenário com *swappiness* = 0 e o outro com *swappiness* = 60.

swappiness	Num. páginas compr.	Tam. original (KB)	Tam. comprimido (KB)	Mem. usada (KB)
= 0	632	2528	857	880
= 60	2544	10176	3337	3520

Tabela 2.1: Dados do Cache Comprimido dos cenários Beta e Gama para o Browser.

A Tabela 2.1, mostra que quando o swappiness é maior, a quantidade de páginas arma-

zenadas e comprimidas pelo CC também é maior. Assim, em situações de grande consumo de memória é bom ter um swappiness com um valor maior que zero pois mais páginas são comprimidas. Nos testes podemos ver que o tamanho original da memória ocupada pelas páginas, quando o swappiness é igual a 60, é de 10176 KB, após a compressão, somente 3337 KB é armazenado na área de swap virtual do CC, o ramzswap. Esse dado mostra que no teste foi possível armazenar as páginas de memória gastando-se 1/3 da memória sem compressão. Mesmo quando o swappiness é zero, a taxa de compressão continua muito boa pois temos 2528 KB de páginas sem compressão e apenas 857 KB de memória após a compressão.

Testes com o Canola

Cenário Alpha versus cenário Beta

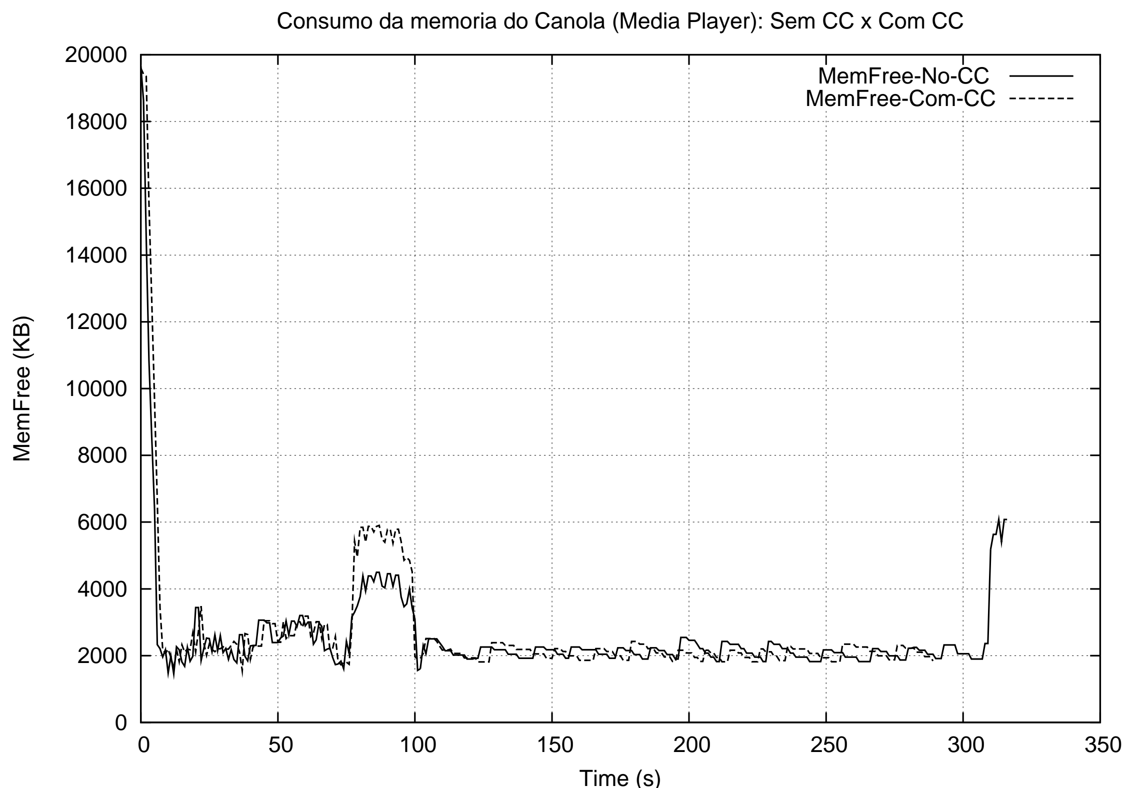

Figura 2.8: Canola: memória livre (MemFree) ao longo do teste. A linha pontilhada representa o consumo quando o CC estava configurado para 10MB de tamanho.

Cenário Alpha versus cenário Gama

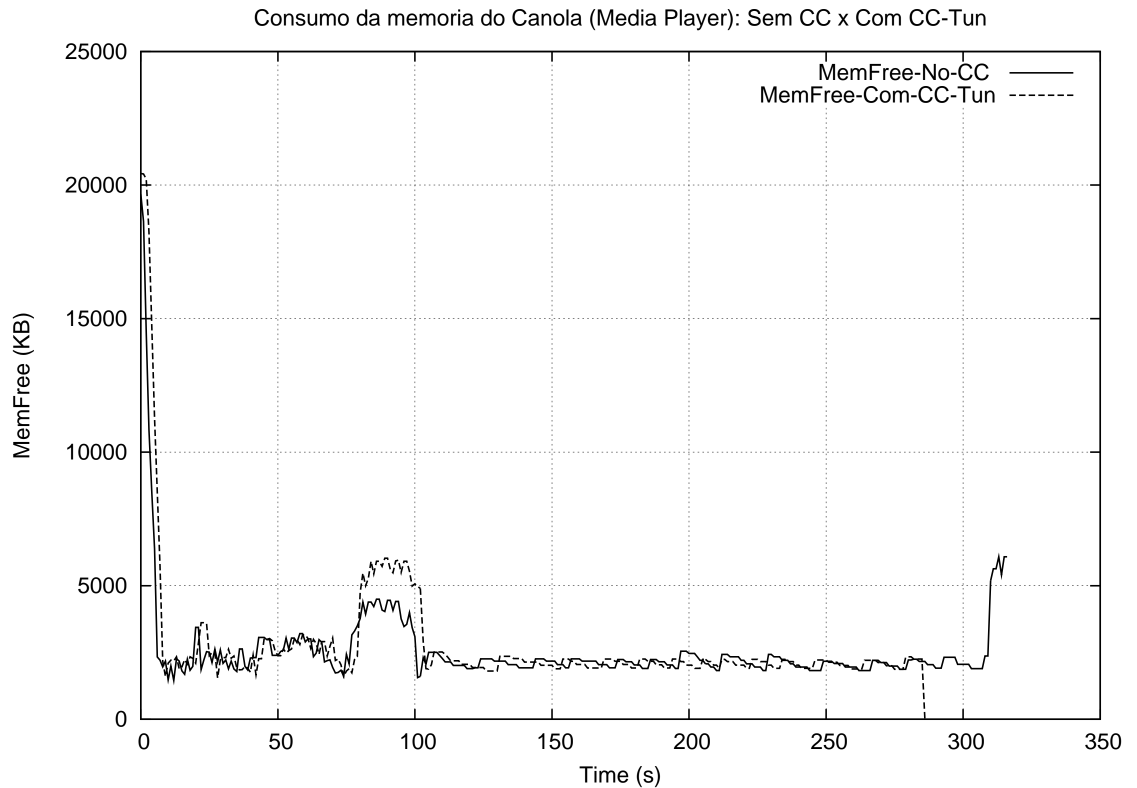

Figura 2.9: Canola: memória livre com Cache Comprimido = 10MB e swappiness = 60.

Nos gráficos do Canola podemos notar que a memória livre se manteve estável para os 2 cenários testados. Também podemos concluir que isso aconteceu porque o Canola não aloca muitas páginas que podem ir para o swap ou ainda, não alocou uma grande quantidade de memória.

Analisando os dados do CC para os cenários Beta e Gama, temos:

swappiness	Num. páginas compr.	Tam. original (KB)	Tam. comprimido (KB)	Mem. usada (KB)
= 0	15	60	1	4
= 60	33	132	1	4

Tabela 2.2: Dados do Cache Comprimido dos cenários Beta e Gama para o Canola.

Na Tabela 2.2, os dados indicam que a área de memória comprimida do CC não foi muito utilizada, nem quando o *kernel* foi configurado com swappiness igual a 60.

Testes com o PDF viewer

Nos dados coletados para o leitor de PDF's, os gráficos de consumo de memória se apresentaram de forma idêntica para os 3 cenários (veja Figura 2.10). Isso se deve ao fato da aplicação testada não ter atingido um consumo alto de memória, acionando assim o uso da área de swap.

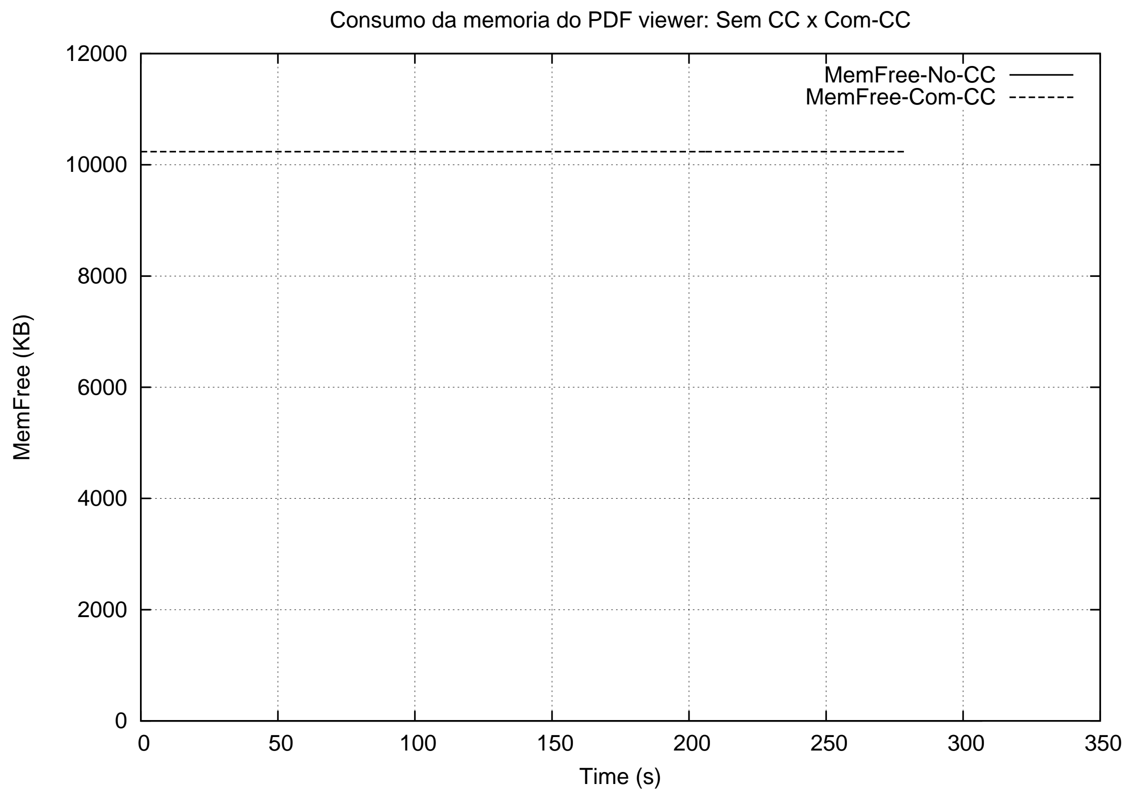

Figura 2.10: PDF: comportamento de consumo da memória para os 3 cenários.

Comparando os resultados

Apresentados todos os resultados para cada aplicação, podemos tirar algumas conclusões:

- O browser é a aplicação que mais consumiu a memória nos testes.
- O Canola teve um consumo mediano. O uso do CC foi baixo, com poucas páginas comprimidas.
- Foi confirmado que o parâmetro de *swappiness* influencia o consumo de memória das aplicações.

- Não foi verificada qualquer diferença no consumo da aplicação de leitura de PDF's.

Com esses dados, podemos ainda concluir que o CC traz mais benefícios para o sistema em situações de alto consumo de memória e com swappiness igual a 60, fazendo com que mais páginas de memória sejam comprimidas. Para aplicações que possuem consumo de memória mediano, o CC pode ter um tamanho menor pois não é tão utilizado. Para aplicações com consumo de memória muito baixo, o CC não traz nenhum benefício.

2.4.4 Testes com o Mibench

MiBench [13] é um conjunto de *benchmarks* sintéticos que simulam vários tipos de operações afim de testar a performance de uma arquitetura ou de um sistema embarcado.

A principal diferença do MiBench para os outros *benchmarks* para sistemas embarcados é que o primeiro é composto por programas que possuem o código aberto, ou seja, são *Open Source*. Uma outra diferença é que o MiBench é dividido em vários pacotes [13], ou módulos, divididos de acordo com o tipo de benchmark que deseja-se realizar, como mostrado a seguir:

- *Automotive and Industrial Control*: possui programas que visam a avaliação de microprocessadores dedicados ao controle de sistemas. Os programas usados nessa categoria avaliam as operações matemáticas, contagem de bits, ordenação e processamento de imagens.

- *Network*: possui programas destinados a testarem microprocessadores existentes em roteadores e *switches*. Os testes encontrados nesse módulo visam avaliar algoritmos e operações referentes à cálculos para encontrar rota mais curta, busca em árvores e tabelas de entrada/saída de dados.

- *Security*: este módulo inclui programas que avaliam os algoritmos de encriptação, decriptação e *hash* de um microprocessador. É mais voltado para operações matemáticas.

- *Consumer Devices*: esta categoria é a mais parecida com o propósito deste trabalho. Possui programas que avaliam o processamento de imagem, áudio e HTML. Os programas incluídos nesta categoria fazem uso do processador e da memória para tocar uma música ou aplicar transformações em uma fotografia, por exemplo.

- *Office Automation*: esta categoria, utiliza programas que avaliam a manipulação de texto afim de realizar testes de performance em equipamentos de escritório, tais como impressoras, aparelhos de fax e processadores da fala.

- *Telecommunications*: neste módulos estão presentes programas que avaliam a capacidade dos microprocessadores dedicados à comunicação, codificação e decodificação de voz.

Os testes realizados neste trabalho visam avaliar se o uso do Cache Comprimido é importante ou prejudicial na execução de alguns *benchmarks* presentes no MiBench. Para a realização dos testes, foi escolhido um programa de cada categoria apresentada anteriormente.

A seguir encontram-se os gráficos do consumo da memória e do consumo da área de swap para cada categoria do MiBench. Foi escolhido um *benchmark* de cada categoria:

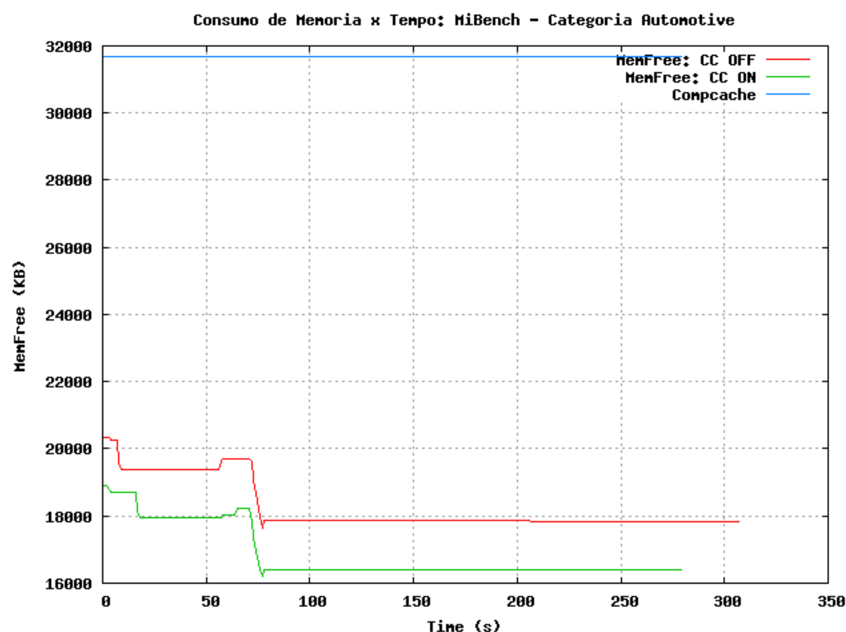

Figura 2.11: Teste com o benchmark da categoria Automotive

Analisando os gráficos de consumo de memória dos testes com o MiBench (Figuras 2.11, 2.12, 2.13, 2.14, 2.15, 2.16), a utilização do Cache Comprimido nesse caso não trouxe nenhum benefício, ou seja, o consumo de memória não foi melhorado pelo uso do CC. A diferença da memória livre inicial vista nos gráficos, deve-se ao fato de que, quando o sistema possui área do Cache Comprimido configurada, a memória total livre é menor do que se o Cache Comprimido estivesse desligado. Isso deve-se ao fato do ramzswap alocar uma área da memória principal para armazenar as páginas comprimidas.

Os gráficos com testes do MiBench indicam que os benchmarks do MiBench não são ideais para avaliarem consumo de memória quando não é utilizado um simulador de arquitetura. Por utilizarem muito mais o processador para operações, os benchmarks são projetados para avaliarem número de instruções efetuadas em várias regiões do microprocessador e não o consumo de memória do dispositivo.

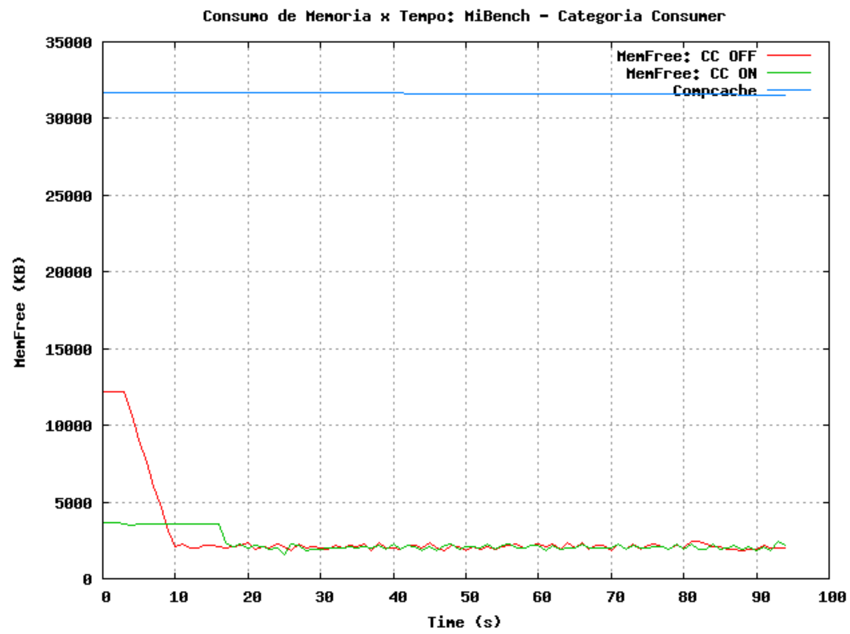

Figura 2.12: Teste com o benchmark da categoria Consumer

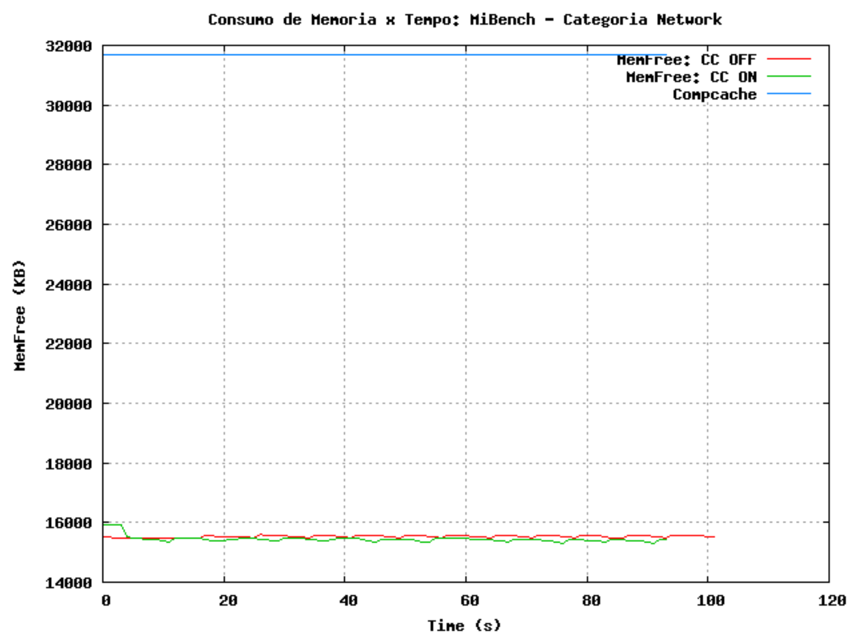

Figura 2.13: Teste com o benchmark da categoria Network

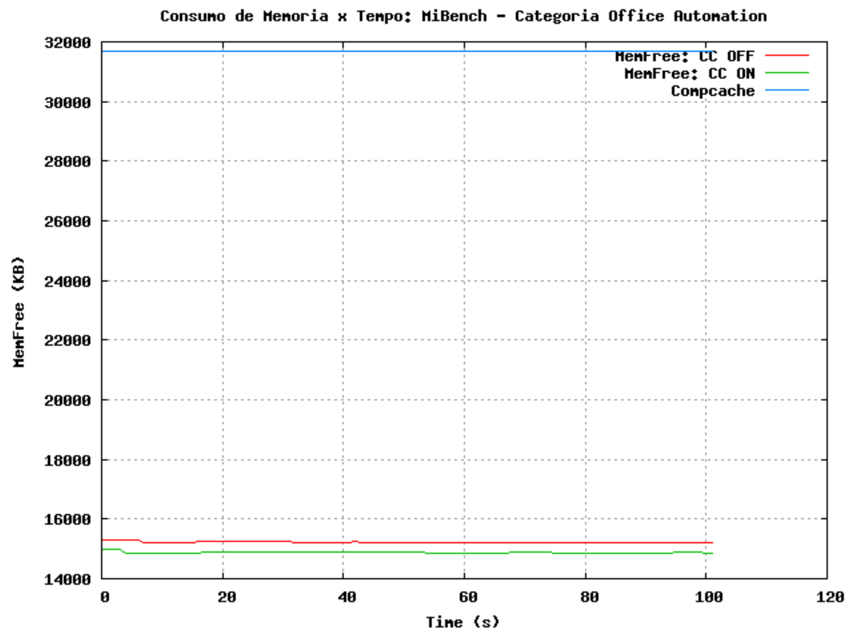

Figura 2.14: Teste com o benchmark da categoria Office Automation

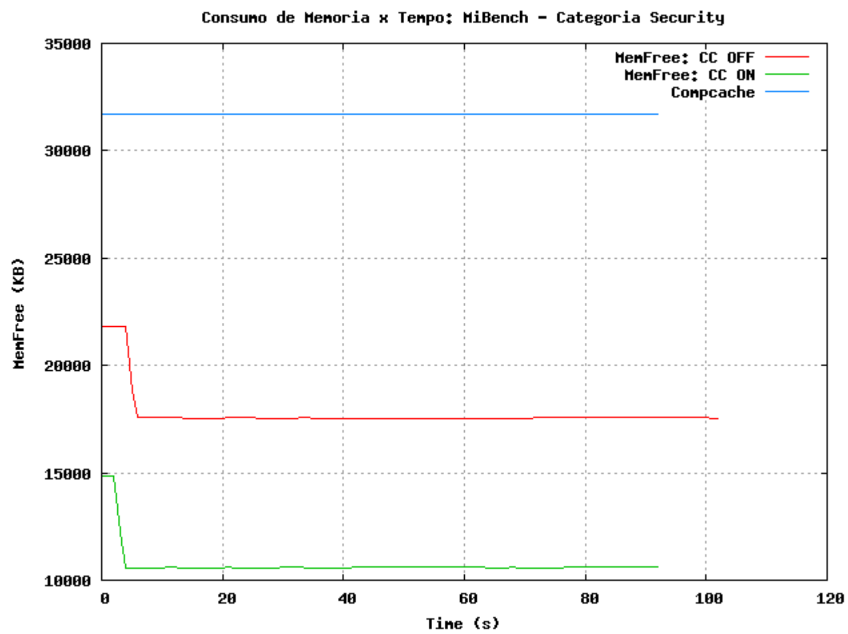

Figura 2.15: Teste com o benchmark da categoria Security

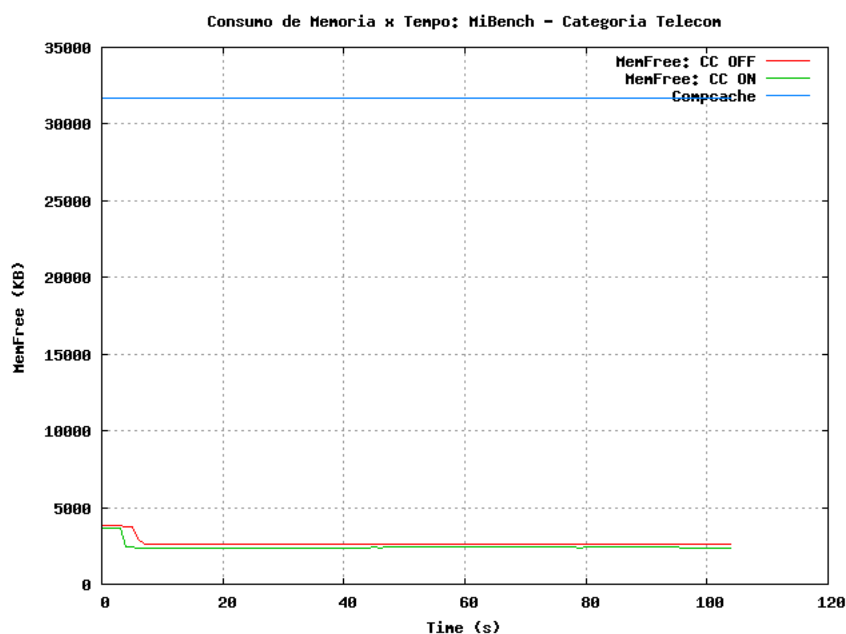

Figura 2.16: Teste com o benchmark da categoria Telecom

2.4.5 Testes de Performance

Nesse tipo de teste foi utilizado um *benchmark* sintético que realize a alocação de uma grande área de memória. O objetivo é analisar o overhead imposto pelo Cache Comprimido quando as páginas são destinadas à área de *swap* virtual (ramzswap).

Para testar a alocação de memória, foi utilizado um programa da suíte de testes MemTest[24]. O programa em questão é o **fillmem** e ele foi usado para alocar uma grande quantidade de memória de forma muito rápida. Na figura 2.17 estão os tempos de execução do fillmem para alocar 50 MB de memória.

No gráfico mostrado na Figura 2.17, nota-se que o tempo de alocação do fillmem é cerca de três vezes maior quando se usa o Cache Comprimido. Isso deve-se ao fato de haver overheads para a compactação das páginas. A plataforma de *hardware* usada [20], como visto anteriomente, não possui disco rígido e sua memória secundária é composta por memória do tipo flash e cartões MMC. Esses dois tipos de memória, por suas características, possuem um tempo de acesso muito maior do que o tempo dos discos rígidos convencionais. Por essa razão, a diferença entre ter uma memória de *swap* virtual, e ter um swap real, tanto na memória *flash*, quanto no MMC, não garante que o acesso ao *swap* virtual do Cache Comprimido será mais rápido. Como pode-se notar no gráfico da figura 2.17, o acesso às páginas de memória armazenadas no *swap* real, em um cartão MMC, é até um pouco maior do que o tempo de acesso no Cache Comprimido. Isso deve-se ao fato de que o acesso ao *swap* no MMC não possuir os tempos de descompactação da página.

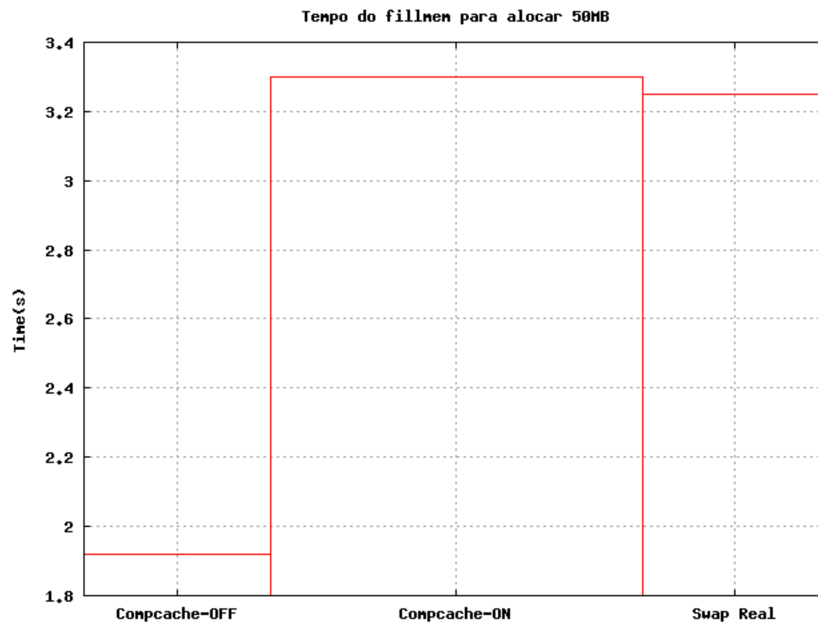

Figura 2.17: Tempo de de execução do fillmem para alocar 50 MB.

2.5 Sumário

Neste Capítulo foram apresentados os detalhes de implementação e decisões de projeto do Cache Comprimido utilizado neste trabalho. Também foram apresentadas as diferenças entre essa versão e as versões utilizadas nos trabalhos relacionados ao Cache Comprimido. Ainda foram apresentados os resultados dos testes com o Cache Comprimido quando executados em Linux embarcado e com cenários de uso pré-definidos, que representassem o uso normal do Linux embarcado. Esses cenários serão utilizados no Capítulo 4, no treinamento da rede neural e fabricação do SOM.

Um dado constatado no teste de consumo de memória é que o Cache Comprimido é mais utilizado quando o kernel é pré-configurado para mandar as páginas de memória mais cedo para o swap. Para o kernel, é transparente o uso do swap virtual do Cache Comprimido, desta forma não se interfere no algoritmo utilizado para selecionar as páginas e enviá-las. Configurando o swapness com o valor padrão de 60, as páginas de memória são enviadas para o Cache Comprimido antes do sistema atingir uma situação crítica de falta de memória.

Um outro dado, não menos importante, é que o *overhead* de comprimir e descomprimir as páginas e localizá-las dentro do swap virtual é bastante próximo do *overhead* padrão de utilização de um swap real, ou seja, de um swap utilizando uma partição em dispositivo de bloco (como um MMC card). Verificando o gráfico apresentado na seção de Testes de Performance, pode-se notar uma pequena diferença entre o tempo de execução do teste

em cada cenário.

Capítulo 3

Redes Neurais e Mapas Auto-organizáveis no Cache Comprimido

Este capítulo apresenta uma introdução sobre redes neurais e mapas auto-organizáveis. Também apresenta como funciona o método de classificação de consumo de memória proposto em [19], e utilizado neste trabalho nos testes com o Cache Comprimido.

3.1 Redes Neurais Artificiais

As *redes neurais artificiais* podem ser caracterizadas como modelos computacionais, com certas propriedades particulares como a habilidade de aprender, de identificar e classificar padrões [18]. Desta maneira, uma rede neural é capaz de extrair regras básicas a partir de dados reais, ou seja, a sistemática do problema, diferindo assim da computação programada, onde é necessário um conjunto de regras rígidas pré-fixadas e algoritmos [14].

Uma rede neural artificial consiste em [18]:

- Um conjunto de unidades de processamento.
- Um estado de ativação para cada uma dessas unidades, que equivale aos dados de saída.
- Conexões entre as unidades (ou neurônios). Essas conexões possuem pesos que interferem nos sinais de saída das unidades vizinhas.
- Uma regra de propagação que ajusta a entrada de dados nas unidades de processamento. Esses dados são provenientes de uma amostra externa.

- Uma função de ativação, que determina o novo nível de ativação baseada na entrada efetiva e no nível corrente de ativação.

- Um conjunto de dados externos ou amostra para a entrada de dados de cada unidade de processamento.

- Uma regra de aprendizagem.

- Um ambiente onde a rede neural deve ser inserida. Esse ambiente deve prover os dados de entrada, sinais e códigos de erros se necessário. Geralmente esse ambiente é o sistema operacional ou um sistema de simulação.

Existem duas abordagens de treinamento para as redes neurais: o supervisionado e o não-supervisionado [14].

- Treinamento supervisionado: nesse tipo de treinamento é fornecido um par de entrada e saída desejada de acordo com os dados da amostra. É calculado também um erro que é utilizado como retro-alimentação da rede afim de ajustar os pesos sinápticos dos neurônios.

- Treinamento não-supervisionado: nesse tipo de treinamento somente são fornecidos os dados de entrada. O próprio mecanismo da rede deve ser capaz de extrair informações estatísticas das amostras afim de ajustar os pesos sinápticos os neurônios da rede.

Independente do tipo de aprendizagem ou treinamento selecionado, toda rede é alimentada por uma seqüência de valores $x_1, x_2, \cdots, x_n$ na entrada e os pesos são ajustados de acordo com um modelo matemático durante a fase de treinamento. O processo de treinamento pode ser organizado nas etapas seguintes [14, 18]:

1. O primeiro padrão de entrada é apresentado para a rede.

2. Os pesos são ajustados para capacitar a rede e reconhecer o padrão fornecido.

3. O segundo padrão de entrada é apresentado para a rede e a etapa 2 é efetuada novamente.

4. O mesmo é aplicado para todos os outros padrões.

5. O procedimento de 1 até 4 é executado novamente centenas ou milhares de vezes até encontrar uma configuração de pesos sinápticos capaz de reconhecer todos os padrões fornecidos no treinamento.

Este trabalho utiliza-se de uma rede neural com treinamento não-supervisionado para encontrar padrões de consumo de memória. No trabalho encontrado em [19], pode-se verificar que é possível identificar padrões referentes à quantidade de páginas de memória que são requisitadas ao *kernel* do Linux.

Como cada página de memória é atômica, ou seja, está ou não está completamente ocupada, é possível gerar amostras de entrada para a rede neural com base na velocidade que as páginas são requisitadas ou ocupadas. Cada programa requisita memória de acordo com o seu fluxo de execução e é com base nessa caracaterística que foi comprovado que é possível classificar o consumo de memória [19].

Após encontrados os perfis de consumo de memória das aplicações testadas nos casos de uso, espera-se adaptar o uso do Cache Comprimido de acordo com o comportamento das mesmas. Para aplicações que requisitam mais memória, o Cache Comprimido deve ser um pouco maior, esperando aumentar a capacidade de armazenamento das páginas de memória. Para aplicações que possuem um consumo menor, o Cache Comprimido pode até ser descartado. Porém, não podemos esquecer de uma importante variável: a velocidade de consumo da memória.

Como visto na seção de testes inciais deste trabalho (ver detalhes capítulo 2), aplicações que requisitam memória muito rapidamente, podem levar o sistema à falta de memória, mesmo que o Cache Comprimido esteja ativo. Para evitar isso, é necessário que se identifique esse perfil antes mesmo que a memória seja requisitada, de forma adequada para otimizar o consumo de memória de uma aplicação.

3.2 Mapas Auto-Organizáveis

Mapas Auto-Organizáveis ou simplesmente *SOM (do inglês Self Organizing Maps)* é uma rede neural artificial auto-organizável, de aprendizagem não supervisionada, baseada em grades de neurônios artificiais onde os pesos são adaptados em conformidade com os vetores de entrada fornecidos durante o treinamento. Foi desenvolvido pelo professor Teuvo Kohonen e às vezes é chamado de mapa de Kohonen [1, 5, 11, 15, 17].

Os SOMs são um tipo de rede neural um pouco diferente pois utilizam a distância topológica entre os neurônios como parâmetro de entrada do sistema de aprendizagem. Os SOMs são muito utilizados para fazer mapas em que os padrões podem ser visualizados, ou seja, características semelhantes dos dados de entrada (ou amostra), são agrupados topologicamente próximos. Portanto um SOM é caracterizado pela formação de um mapa topográfico dos padrões de entrada, baseado nas características estatísticas intrínsecas

contidas nesses dados, por isso o nome *mapa auto-organizável* [14].

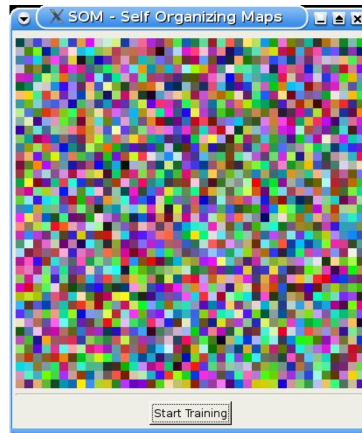

Figura 3.1: Exemplo de rede não treinada.[19]

Em [2] e [19] foi usado um exemplo comum de SOM que utiliza um mapeamento de cores a partir de seus 3 componentes: vermelho, verde e azul ou RGB (do inglês *red, green and blue*) em um espaço bidimensional. Na Figura 3.1, pode-se ver um exemplo de SOM que ainda não foi treinado. Já a Figura 3.2 ilustra um exemplo de SOM treinado para reconhecer padrões de cores RGB. As cores são fornecidas para a rede como vetores de 3 dimensões, uma dimensão para cada componente de cor, e a rede foi treinada para representá-los em um espaço de 2 dimensões. Observe que além do agrupamento de cores em regiões distintas, as regiões de propriedades similares estão localizadas de forma adjacente [19].

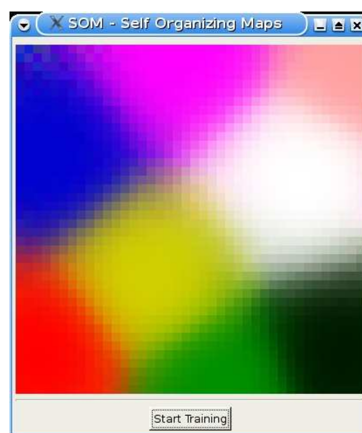

Figura 3.2: Um exemplo de SOM treinado para classificação de cores.[19]

A rede SOM ilustrada na Figura 3.2 apresenta uma grade de tamanho 40×40. Cada nó da grade possui 3 pesos, cada um representando um componente RGB. Além disso, cada nó é representado por uma célula retangular, quando desenhado na interface gráfica

do aplicativo utilizado para treinar a rede e mostar o SOM.

A formação do SOM é feita primeiramente iniciando os pesos sinápticos da grade, com valores obtidos de um gerador de números randômicos. Assim, nenhuma organização prévia é imposta ao mapa de características (veja Figura 3.1. Depois que a grade é feita, são executados 6 processos importantes [14]:

Iniciação: Cada neurônio tem os seus pesos sinápticos inicializados aleatoriamente. Geralmente os pesos são inicializados entre 0 e 1, ou seja, $0 < w < 1$.

Seleção de Vetor de Entrada: Um vetor de entrada é escolhido do conjunto de dados de treinamento e apresentado para a grade de neurônios.

Competição de Neurônios: Todos os pesos de todos os neurônios são calculados para determinar o neurônio mais semelhante em relação ao vetor de entrada. O neurônio mais semelhante é selecionado e é considerado como o neurônio vencedor é chamado de Unidade de Melhor Casamento ou BMU (do *inglês Best Matching Unit*).

Cooperação de Neurônios: O *raio da vizinhança topológica* do BMU é calculado. O valor do raio assume inicialmente um valor elevado, geralmente tem o mesmo raio da grade, mas diminui a cada iteração do treinamento. Os neurônios localizados dentro deste raio são considerados os vizinhos do BMU.

Adaptação Sináptica: Os pesos sinápticos de cada neurônio vizinho do BMU são ajustados para torná-los similares ao vetor de entrada. Os neurônios vizinhos mais próximos do BMU têm os seus pesos alterados de modo mais significativo.

Repetição: O segundo passo é retomado novamente selecionando um novo vetor de entrada e os passos subseqüentes são então executados.

A forma para determinar o BMU é acessar todos os neurônios da grade e calcular a *distância Euclidiana* entre o vetor peso de cada neurônio e o vetor de entrada atual. O neurônio de vetor peso mais próximo do vetor de entrada (menor distância Euclidiana), é considerado como o neurônio vencedor ou BMU. Neste caso a distância Euclidiana é a função discriminante neste processo competitivo de neurônios e calculado como ilustra a Equação 3.1.

$$dist_j = \sqrt{\sum_{i=1}^{n}(x_i - w_i)^2} \tag{3.1}$$

Os pesos sinápticos dos neurônios são ajustados de acordo com o BMU a cada iteração do treinamento. No decorrer do treinamento, o raio de ação do BMU diminui, modificando menos neurônios vizinhos e especializando ainda mais as áreas com relação aos padrões

de entrada da amostra.

Os algoritmos usados para a iniciação dos pesos sinápticos, cálculo para achar o BMU e o cálculo da distância euclidiana são os mesmos que foram usados em [19] e estão presentes no Apêndice A.

3.3 Método de classificação de consumo de memória baseado em SOMs

Ainda baseado em [19], essa seção mostra como os SOMs podem ser utilizados para classificar o consumo de memória e também como é feita a coleta dos dados de entrada para o treinamento das redes neurais e conseguinte geração dos SOMs.

As coletas consistem em executar casos de uso de aplicações (Browser, Canola e PDF viewer), afim de realizar algumas medições do consumo de memória das mesmas. Os casos de uso foram executados no sistema operacional Linux e seguindo a metodologia apresentada na seção 2.4.1.

Como utilizado nos trabalhos [2] e [19], o algoritmo de treinamento da rede espera dados referentes ao consumo de memória de uma determinada aplicação. Ou seja, o programa de treinamento da rede neural espera um vetor com x entradas, sendo x o número de leituras da memória consumida ao longo do tempo.

Para cada aplicação testada, foram feitas leituras durante um tempo t. Essas leituras formaram o vetor de entrada para treinamento da rede neural e geração do SOM. Assim, o SOM gerado indica o padrão de consumo de memória de todas as aplicações juntas. Cada aplicação ainda tem o seu próprio padrão, agora identificável pela rede neural e SOM gerado. Esse padrão é classificado de acordo com a metodologia desenvolvida em [19].

Lembrando que [19]:

- *memória* (mem) é a quantidade de páginas físicas alocadas que representa a *quantidade de memória física consumida*;
- *variação do consumo de memória (VCM)* que é utilizada para indicar o ritmo em que o consumo de memória está aumentando ou diminuindo. Esta variação é calculada usando a razão entre a diferença da quantidade de memória física consumida e o intervalo de tempo decorrido, conforme a Equação 3.2;

$$VCM = \frac{mem_2 - mem_1}{t_2 - t_1} \tag{3.2}$$

- *taxa de variação do consumo de memória (TVCM)* que é utilizada para indicar o ritmo em que a variação de consumo de memória está aumentando ou diminuindo. Esta taxa é calculada usando a razão entre a diferença da variação do consumo de memória e o intervalo de tempo decorrido, conforme a Equação 3.3.

$$TVCM = \frac{vcm_2 - vcm_1}{t_2 - t_1} \tag{3.3}$$

Ainda como visto em [19], essas três dimensões ainda possuem subintervalos que podem ser caracterizados como Low (L), Medium (M) e High(H). Esses subintervalos são utilizados para representarem o comportamento de uma determinada propriedade de uma forma simples e abstrata.

A quantidade de memória consumida está localizada em um desses subintervalos, assumindo um dos três estados $\{Low, Medium, High\}$ apresentados. Sendo assim a quantidade de memória consumida é classificada como *baixo consumo* de memória quando o seu valor está no estado L, *médio consumo* de memória quando o seu valor está no estado M e *alto consumo* de memória quando localizado no estado H. A quantidade de memória consumida abrange somente valores inteiros positivos, visto que consumo de memória de valor negativo é logicamente inexistente no mundo real. A mesma analogia é utilizada para os valores de VCM e TVCM.

Assim, a tripla <memória,vcm,tvcm> pode ser mapeada facilmente para uma estrutura de dados que seja aceita no algoritmo do SOM, que foi usado em [19]. Já que o SOM utiliza vetores tridimensionais como valores de entrada, a tripla pode facilmente ser utilizada, contendo cada valor como sendo uma dimensão desse vetor.

Cada vetor no espaço tridimensional é uma instância específica da tripla que é mapeado na grade de neurônios do SOM. Cada célula da grade de neurônios armazena então um vetor tridimensional do tipo <memória, vcm, tvcm>, conforme a Figura 3.3.

A idéia de utilizar o mapa de neurônios auto-organizáveis é agrupar topologicamente as classes apresentadas de consumo de memória em áreas ou regiões distintas. Dessa forma, cada região representa uma configuração ou conjunto de configurações similares capaz de representar o estado do consumo de memória de uma aplicação em um determinado instante.

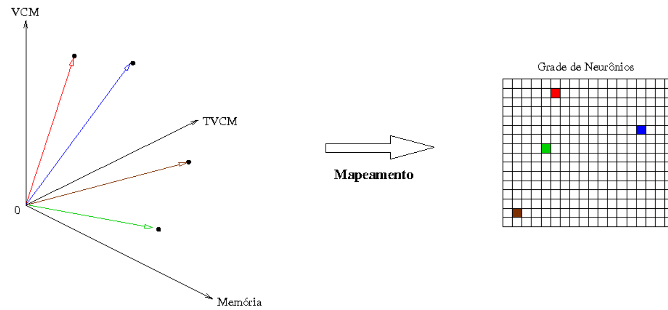

Figura 3.3: Os vetores são mapeados em um espaço bidimensional representado pela grade de neurônios [19].

3.4 Coleta dos dados

Como dito anteriormente, os dados coletados para a entrada da rede neural foram obtidos de leituras constantes da memória livre do sistema. Alguns casos de uso foram definidos com base nos mesmos casos apresentados na seção 2.4.3. Lá, foram utilizadas as seguintes aplicações: browser, media player e visualizador de arquivos no formato PDF.

Todas as leituras foram realizadas no dispositivo móvel e foram desenvolvidos programas que automatizassem a coleta dos dados. Os passos para gerar as amostras foram os seguintes:

1. Executa-se um caso de uso, ou seja, somente o browser, ou o media player, etc.
2. Durante a execução são coletadas as amostras de memória total livre × tempo.
3. As amostras são convertidas em triplas para o tipo usado no treinamento da rede neural: <memória, vcm, tvcm>.

No final, temos 3 arquivos de log contendo a memória total livre durante a execução das 3 aplicações: browser, media player e visualizador de PDF's. Ainda foi adicionado mais um arquivo de log contendo a memória livre × tempo durante a execução dos 3 programas simultaneamente.

Esses arquivos de log são concatenados e utilizados como dados de entrada para o treinamento da rede neural.

3.5 Treinando a rede neural e criando o SOM

Foram treinadas diferentes redes neurais com seus SOMs respectivos. A idéia é treinar uma rede neural e gerar o SOM para cada cenário usado nos testes de comportamento de memória (ver seção 2.4.3):

- Cenário Alpha: Cache Comprimido desligado e kernel sem área de swap.
- Cenário Beta: Cache Comprimido ligado e configurado com área de swap virtual aproximadamente igual a 10 MB.
- Cenário Gama: Cache Comprimido ligado, configurado com área de swap virtual aproximadamente igual a 10 MB e com *swappiness* igual a 60.

Para cada cenário apresentado foram realizadas medidas da memória consumida pelas aplicações durante a execução dos testes. Os testes seguem a mesma metodologia apresentada na seção 2.4.3 e essas amostras fizeram parte do vetor de entrada para treinamento das redes e geração de cada SOM. No final do treinamento das redes, temos um SOM para o consumo de memória das aplicações quando são executadas no cenário Alpha, um SOM que representa o consumo de memória das aplicações para o cenário Beta e por último um SOM que representa o consumo da memória para o cenário Gama. Essas 3 redes neurais e seus respectivos SOMs são capazes de identificar o consumo de cada uma das aplicações utilizadas nos testes (Browser, Canola ou PDF viewer), de acordo com o cenário usado.

Nas Figuras 3.4, 3.5, 3.6, podemos visualizar que os mapas são levemente diferentes em cada um dos três cenários apresentados. Essa diferença nos SOMs indica que o uso do Cache Comprimido altera as classes de consumo de memória das aplicações. De acordo com [19], uma área mais avermelhada, ou seja, com mais elemento *Red* indica que a quantidade de memória medida é grande (mem) e uma área mais azulada (*Blue*), indica que taxa de variação do consumo (tvcm) é alta e o consumo é baixo. Porém, o mais importante é a área de visitação que cada aplicação possui. É esta área que será utilizada para identificar qual aplicação está sendo executada no momento e com isso gerar uma tomada de decisão para o Cache Comprimido (ver seção 3.6).

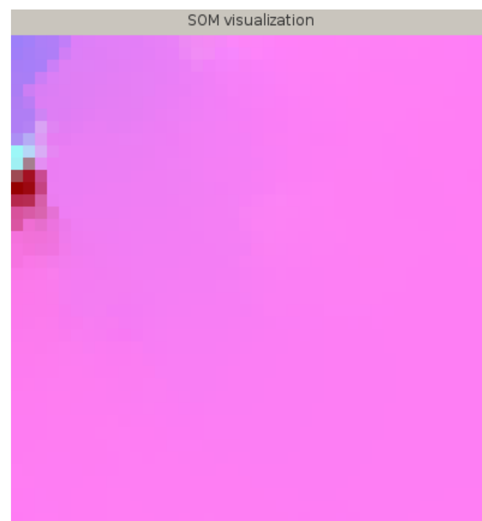

Figura 3.4: Cenário Alpha: SOM para rede neural treinada sem Cache Comprimido.

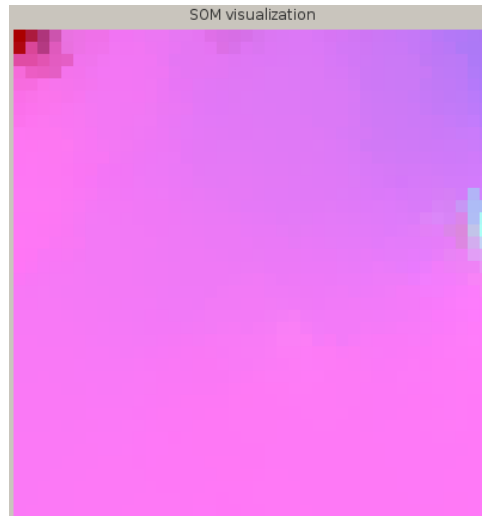

Figura 3.5: Cenário Beta: SOM para rede neural treinada com Cache Comprimido de tamanho igual a 10MB.

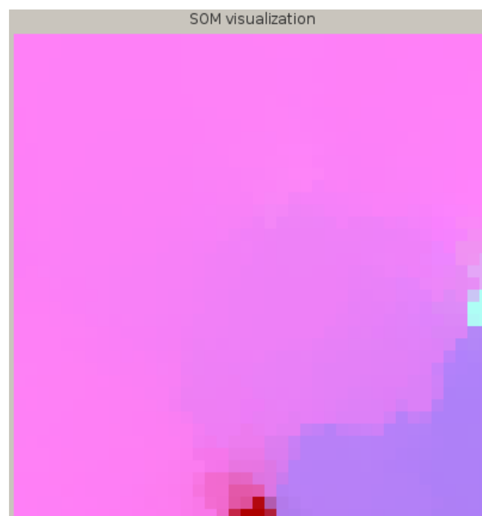

Figura 3.6: Cenário Gama: SOM para rede neural treinada com Cache Comprimido = 10MB e swappiness = 60.

Nos testes com o cenário Alpha, nenhuma página de memória era comprimida e podemos notar que a parte azulada do gráfico, que indica uma baixa aceleração no consumo e um baixo consumo de páginas de memória, é bem pequena. Nos cenários seguintes, Beta e Gama, a área azulada já é ligeramente maior. Isso se deve ao fato do Cache Comprimido armazenar mais páginas de memória, deixando assim páginas livres na área não-comprimida, indicando um consumo menor da memória, por parte das aplicações.

Também podemos notar que de acordo com que o Cache Comprimido é mais utilizado, ou seja, mais páginas de memória são comprimidas e armazenadas, temos um aumento da área azulada no SOM. Isso pode ser verificado no cenário Gama, o qual, através da configuração do parâmetro de *swappiness* em `/proc/sys/vm/swappiness`, apressamos o envio de páginas de memória para a área de *swap virtual*, deixando assim mais memória livre.

3.6 Classificando as aplicações

As triplas <memória, vcm, tvcm>, são mapeadas para valores RGB das cores de cada pixel que compõe o SOM, sendo que:

- a memória é mapeada para R ou vermelho.
- o vcm é mapeado para G ou verde.
- o tvcm é mapeado para B ou azul.

Como mostrado na seção 3.5, a rede neural que gerou cada SOM é treinada com coletas da memória livre feitas durante a execução de cada aplicação testada. Assim, o SOM agrupa os BMU's (Best Matching Units), de acordo com a classe de consumo, ou seja, de acordo com os valores das triplas <memória, vcm, tvcm>, mapeadas como cores dos pixels.

O SOM representa o padrão de consumo de todas as aplicações testadas juntas (Browser, Canola e PDF viewer). Se quisermos saber qual é o SOM de uma determinada aplicação (somente do Browser, por exemplo), devemos passar para a rede um vetor contendo as amostras de consumo de memória da aplicação e determinar quais são os BMUs dada a amostra.

Assim, uma aplicação que visita uma área mais avermelhada (pixels com mais vermelho do que outras cores), possui um consumo de memória maior do que a velocidade e a aceleração (vcm e tvcm, respectivamente), visto que possui uma porção maior de vermelho na composição da cor. Uma aplicação que visita uma área mais azulada, possui uma

variação na aceleração maior do que as outras duas dimensões, e assim por diante. Com base nessas características e no programa desenvolvido em [2], é possível identificar as áreas visitadas do SOM dado um log de entrada, ou seja, dado um vetor de entrada com os valores da memória consumida da aplicação testada no decorrer do tempo. Espera-se com isso conseguir identificar qual aplicação está executando com base no padrão de comportamento de áreas visitadas do SOM, ou ainda definir o perfil que o consumo de memória irá seguir, dado o consumo instantâneo.

Nas Figuras 3.7, 3.8 e 3.9, temos a visualização das áreas visitadas do SOM de acordo com as amostras coletadas durante a execução do Browser, ou seja, quais foram os BMU's encontrados de acordo com o valor da memória consumida pelo Browser. Note que as áreas são diferentes dado um cenário (Alpha, Beta ou Gama). Isso confirma o fato de que o comportamento do consumo de memória do Browser é influenciado quando o Cache Comprimido está presente ou ainda quando o *swappiness* é configurado com valor igual a 60.

Figura 3.7: Cenário Alpha: regiões onde a aplicação 'Browser' visitou no SOM.

Figura 3.8: Cenário Beta: regiões onde a aplicação 'Browser' visitou no SOM.

Figura 3.9: Cenário Gama: regiões onde a aplicação 'Browser' visitou no SOM.

Como visto nas Figuras 3.7, 3.8 e 3.9, nos cenários Alpha e Beta, o consumo de memória do Browser, mapeado no SOM de cada rede mostra que o mesmo é maior do que a velocidade e a aceleração, indicando um valor alto. No cenário Gama, quando o Cache Comprimido e o swappiness estavam configurados, o consumo de memória do Browser visitou mais áreas azuladas do SOM, indicando um consumo de memória mais baixo.

As áreas visitadas para o Canola e o PDF viewer são mostradas nas Figuras 3.10, 3.11, 3.12, 3.13, 3.14 e 3.15.

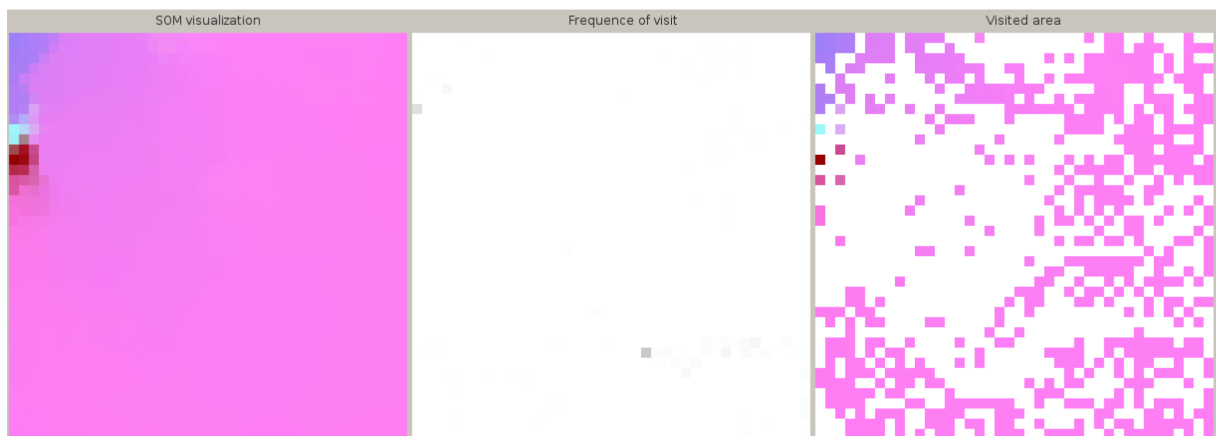

Figura 3.10: Cenário Alpha: regiões onde o 'Canola' visitou no SOM.

Nas áreas visitadas pelo PDF e pelo Canola, não notamos uma diferença quando os cenários são mudados. Conclui-se que o Cache Comprimido não possui muita influência no consumo de memória dessas aplicações. Porém, no caso do Browser, houve um consumo mais baixo da memória quando o Cache Comprimido estava ativo e o *swap virtual* configurado para receber as páginas de memória mais cedo (swappiness = 60). Isso pode ser visualizado no gráfico do cenário Gama, Figura 3.9.

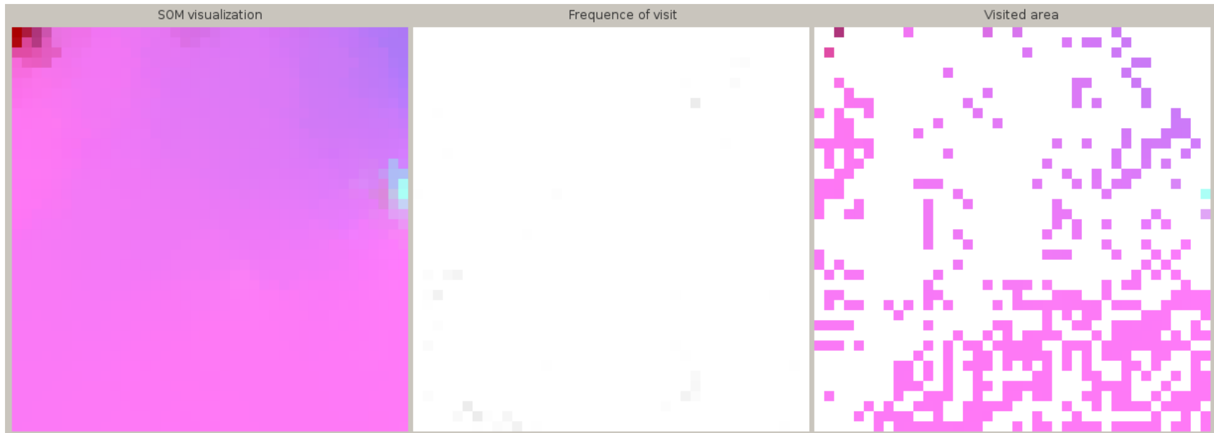

Figura 3.11: Cenário Beta: regiões onde o 'Canola' visitou no SOM.

Figura 3.12: Cenário Gama: regiões onde o Canola visitou no SOM.

Figura 3.13: Cenário Alpha: regiões onde o 'PDF viewer' visitou no SOM.

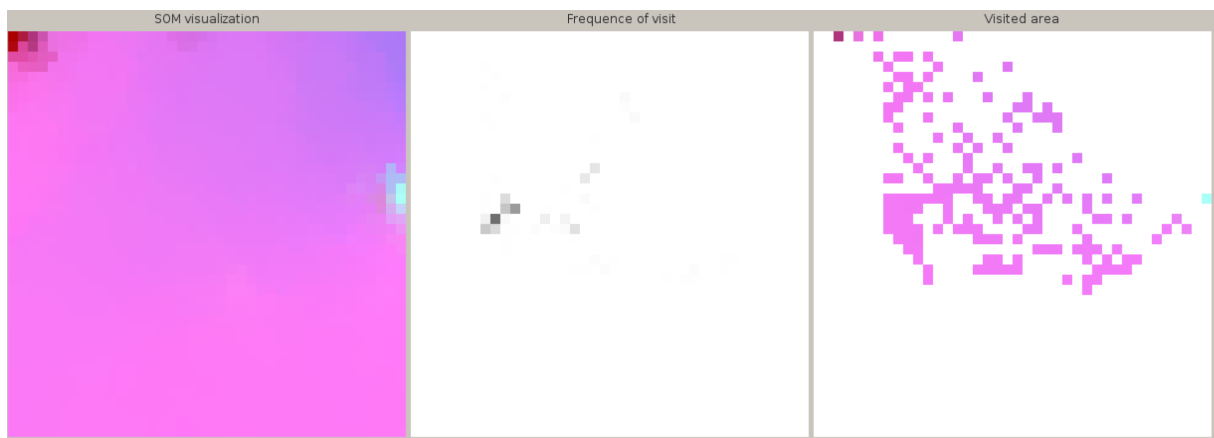

Figura 3.14: Cenário Beta: regiões onde o 'PDF viewer' visitou no SOM.

Figura 3.15: Cenário Gama: regiões onde o 'PDF viewer' visitou no SOM.

Uma outra medida, não menos importante, é a frequência de visita dos pontos (visualizada aqui no retângulo central de cada figura). Esse gráfico mostra quais pontos foram mais visitados durante a execução da aplicação avaliada. A frequência é calculada pela quantidade de vezes que um BMU é encontrado quando comparado a um valor dos dados de entrada. Cada vez que um pixel é visitado, ele ganha 1 unidade de cor preta. Assim, pontos mais escuros indicam mais visitas.

Novamente pode-se notar que quando o Cache Comprimido estava ligado, os pontos mais visitados estavam na área azulada do SOM, indicando um baixo consumo da memória principal do sistema.

Analisando as áreas do SOM, visitadas por cada aplicação testada, podemos chegar à conclusão de 3 perfis para o consumo de memória:

- Alto consumo de memória: são aquelas aplicações que tiveram seus perfis visitando áreas no SOM em tons de vermelho.
- Baixo consumo de memória: são aquelas aplicações que tiveram seus perfis visitando áreas no SOM em tons de azul.
- Mediano consumo de memória: são aquelas aplicações que possuem seus perfis como uma mistura dos outros dois, não prevalecendo nenhuma área.

Dado cada perfil, é possível desenvolver um programa capaz de identificar qual aplicação está sendo executada ou em qual perfil de consumo ela se encaixa, com base na rede neural já treinada de acordo com o cenário utilizado: Alpha, Beta ou Gama. Esse programa é usado para tomar uma decisão: ligar ou não o Cache Comprimido e ainda, qual será a configuração do mesmo. Esse sistema é apresentado no capítulo a seguir.

3.7 Sumário

Neste Capítulo foram apresentados conceitos de redes neurais e mapas auto-organizáveis. Também foi apresentado um resumo dos trabalhos usados como referência [19, 2], os quais também demonstram resultados utilizando SOMs para classificar padrões de consumo de memória.

Neste Capítulo também foram mostradas as redes neurais e seus respectivos SOMs, treinadas para identificar o consumo de memória das aplicações testadas em diferentes cenários, entre eles, o cenário com o Cache Comprimido ligado. Com isso, foi observado que o Cache Comprimido exerce uma influência positiva no consumo de memória de aplicações que requerem uma quantidade grande de memória. Com o Cache Comprimido

o comportamento do consumo de memória de tais aplicações cai, liberando assim mais memória para outras aplicações.

Capítulo 4

Cache Comprimido Adaptativo

Neste Capítulo será apresentada uma solução para implementar o Cache Comprimido Adaptativo, ou seja, a possibilidade do Cache Comprimido ser ligado ou desligado e ser configurado em tempo real de acordo com o perfil de consumo de memória atual e em um futuro próximo.

Os testes com o Cache Comprimido (CC) são executados em Linux embarcado na plataforma ARM OMAP. O equipamento não possui área de swap configurada e possui memória total igual a 128MB. Para mais detalhes sobre a configuração de hardware do dispositivo, ver seção 2.4.1.

4.1 Adaptatividade no Cache Comprimido

Implementar a adaptatividade ao CC é adaptá-lo de acordo com o consumo de memória instantâneo e futuro. Para implementar essa nova característica alguns requisitos são necessários:

- Ter uma amostra do consumo de memória durante um tempo pré-definido.
- Ter um padrão que confronte Configurações do CC versus Padrão de Consumo de memória. Esse padrão vale para decidir o que fazer quando determinado nível de memória é atingido.
- Implementar uma heurística para definir a configuração do CC que será adotada, de acordo com a amostra de memória.

Como visto na seção 3.6, cada aplicação possui um SOM característico. Além disso, o SOM de cada aplicação testada depende do cenário adotado: se existe ou não CC ativo, por exemplo. O programa utilizado em [2], além de gerar o SOM para um log de consumo de memória, também gera uma matriz de frequências, como visto nas Figuras

4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8 e 4.9 (área sombreada). Cada quadrado representa um neurônio na grade e sua cor varia de acordo com as características de consumo do perfil de memória analisado. Nas figuras ainda estão representadas as matrizes de frequência e de visitação dos perfis de memória. Na primeira, quanto mais escuro é o quadrado, mais visitas aconteceram àquele neurônio. Na área de visitação são representados os neurônios e regiões do SOM que foram visitadas durante a execução da aplicação testada.

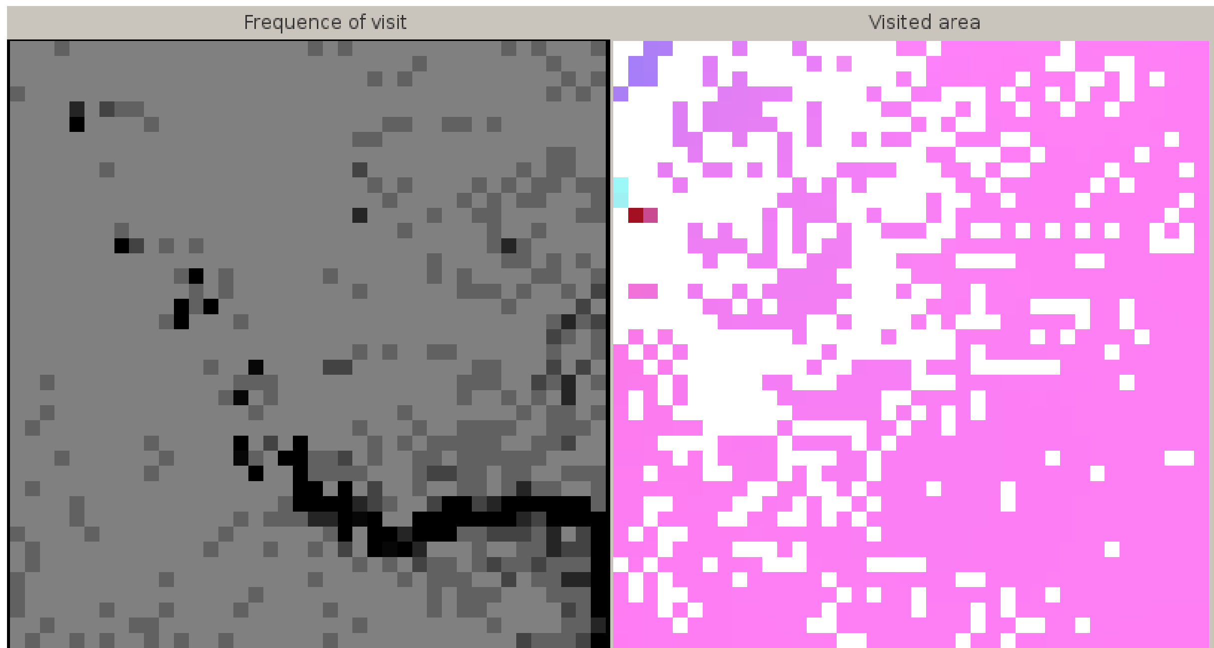

Figura 4.1: Cenário Alpha: imagem que representa o SOM para o Browser e sua matriz de frequências.

A matriz de frequência de cada aplicação mostra quantas vezes aquele BMU foi visitado. Quanto mais escuro é o pixel, mais vezes o BMU foi selecionado.

Se for possível determinar os BMUs em tempo de execução e compará-los com as matrizes de frequência das três aplicações testadas, é possível determinar qual aplicação está sendo executada e tomar medidas quanto ao Cache Comprimido assim que essa identificação é feita. O algoritmo seria o seguinte:

1. Ler o saved_som.txt contendo a rede neural treinada para o cenário dado (o código está disponível no Apêndice B). A rede neural é representada por uma matriz quadrática de ordem 40 em que cada célula representa um **som_node** com os pesos sinápticos.

2. Armazenar em matrizes de ordem 40 a tabela de frequência para cada aplicação: browser, pdf e canola.

3. Realizar leituras periódicas da memória livre em um intervalo de tempo ($t = 1$s). Guardar os valores em um vetor de inteiros.

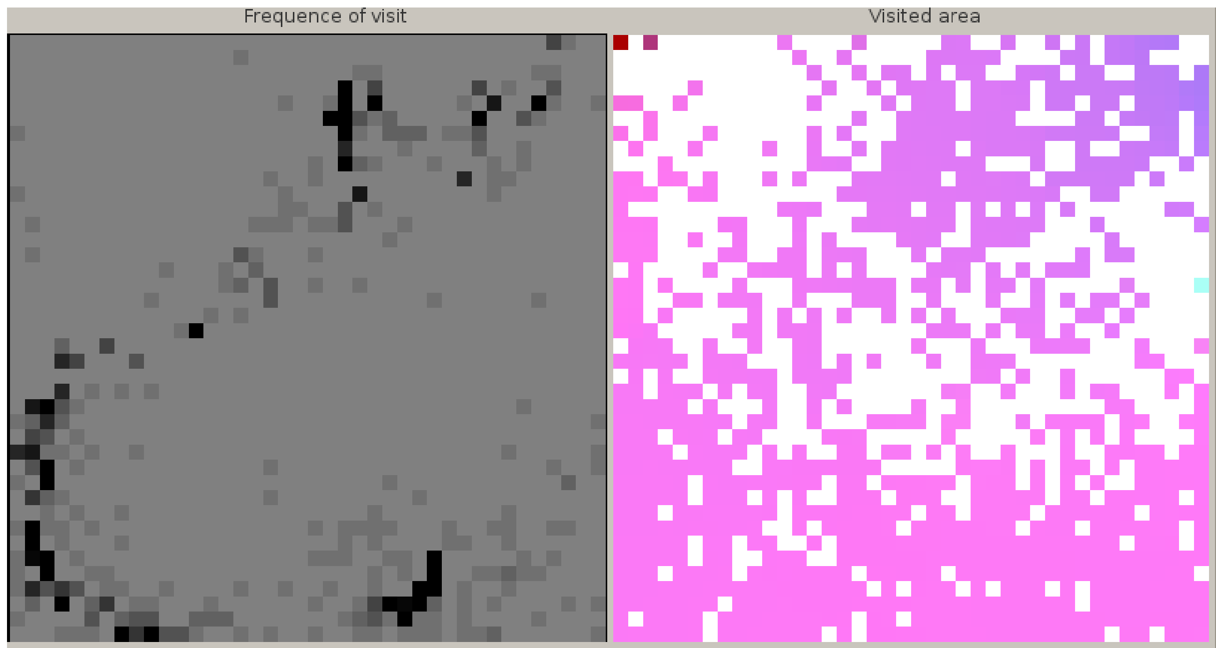

Figura 4.2: Cenário Beta: imagem que representa o SOM para o Browser e sua matriz de frequências.

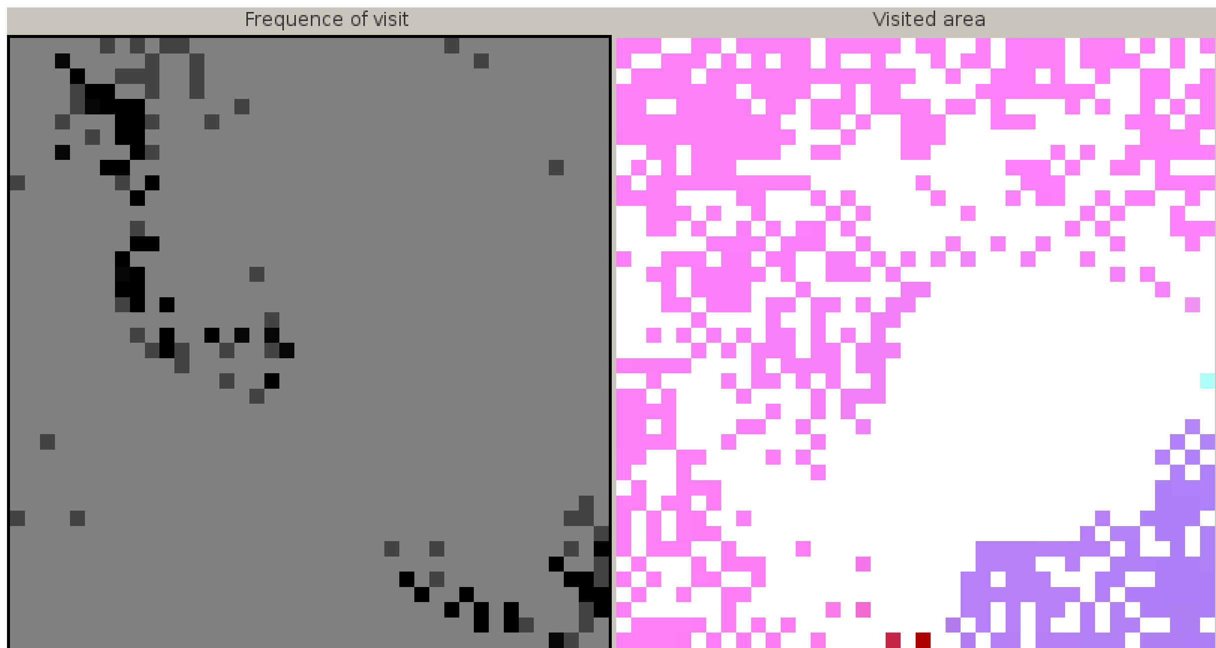

Figura 4.3: Cenário Gama: imagem que representa o SOM para o Browser e sua matriz de frequências.

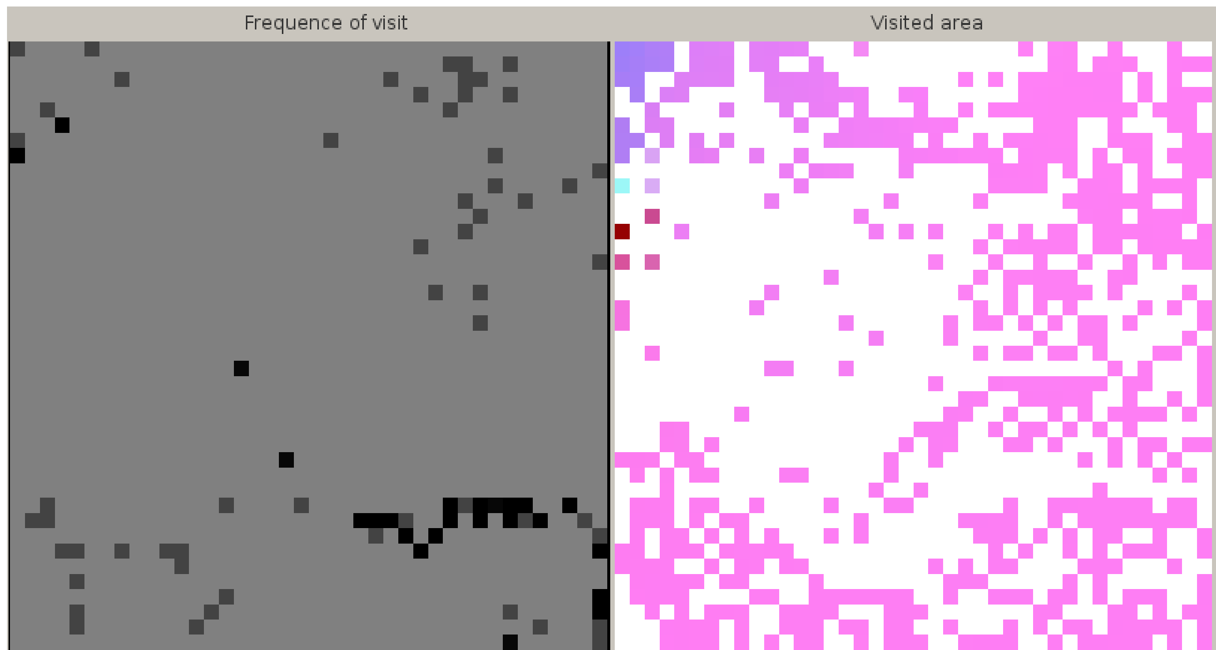

Figura 4.4: Cenário Alpha: imagem que representa o SOM para o media player Canola e sua matriz de frequências.

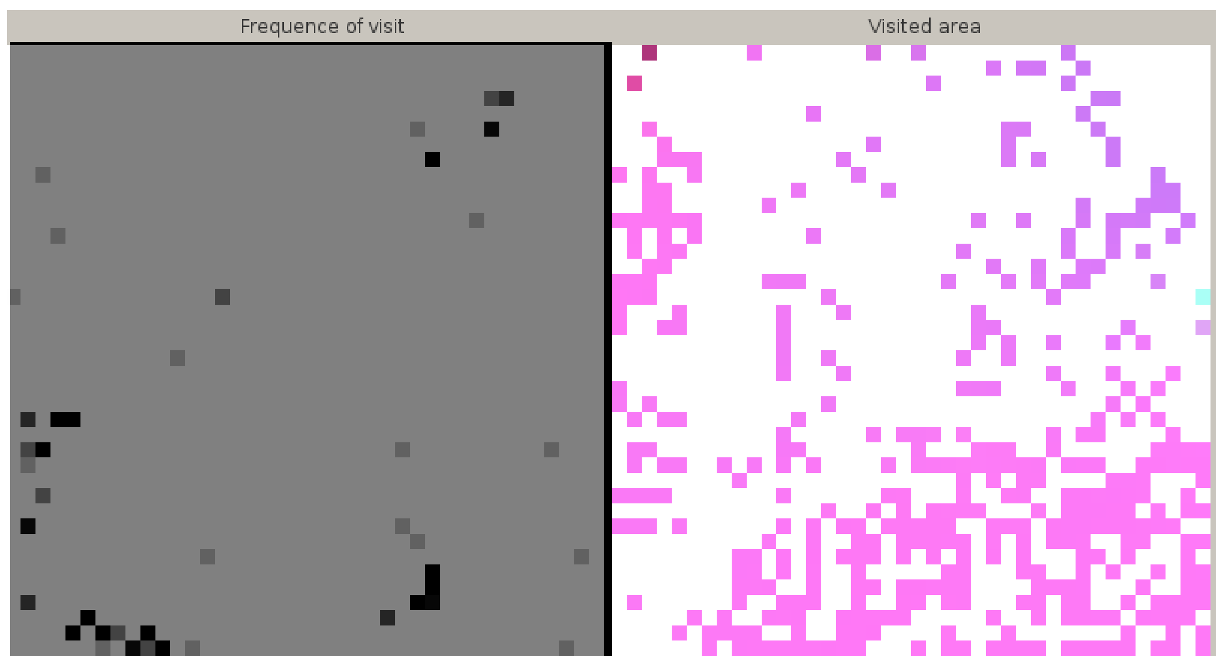

Figura 4.5: Cenário Beta: imagem que representa o SOM para o media player Canola e sua matriz de frequências.

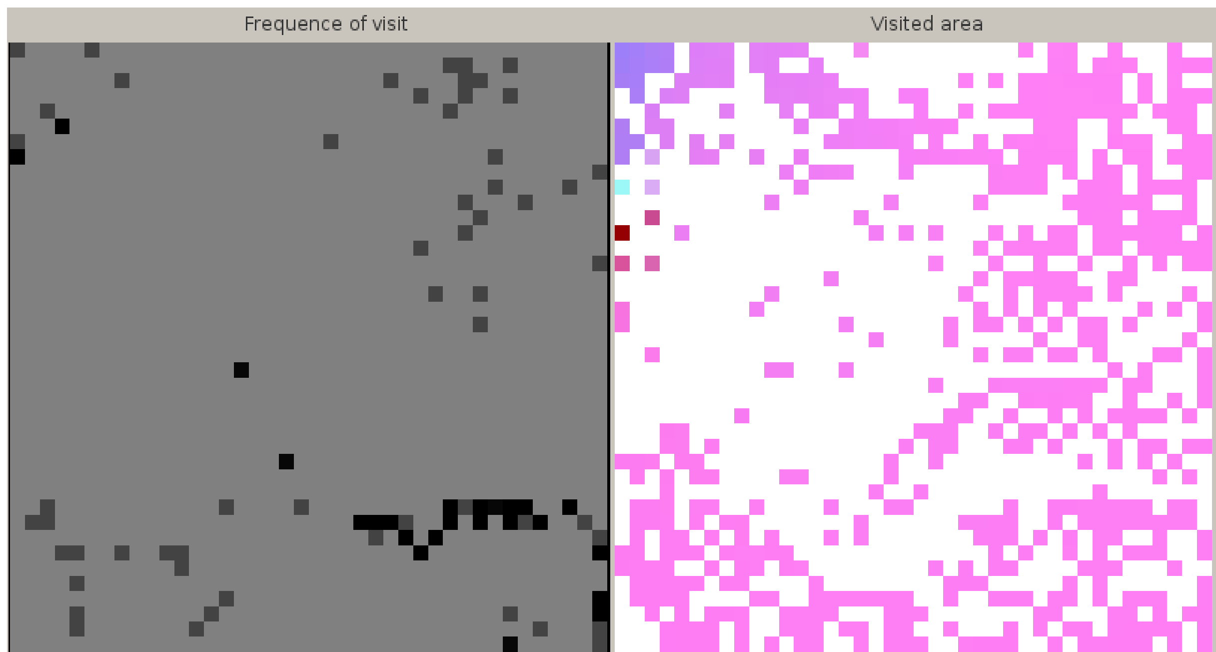

Figura 4.6: Cenário Gama: imagem que representa o SOM para o media player Canola e sua matriz de frequências.

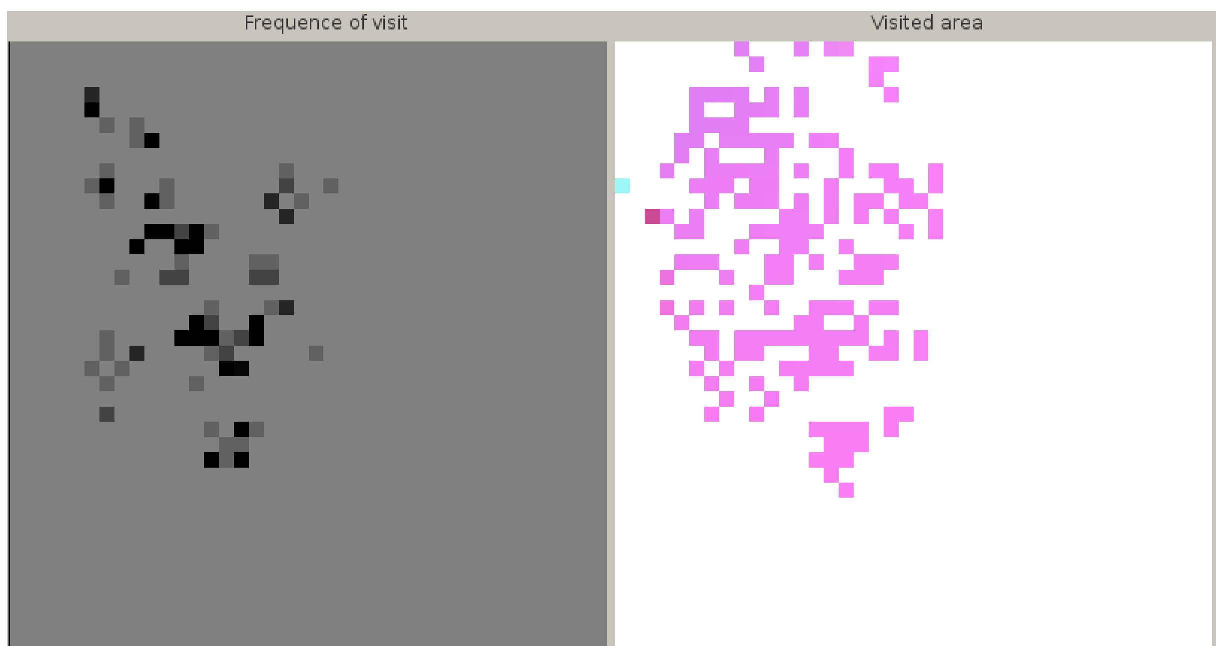

Figura 4.7: Cenário Alpha: imagem que representa o SOM para o visualizador de PDF's e sua matriz de frequências.

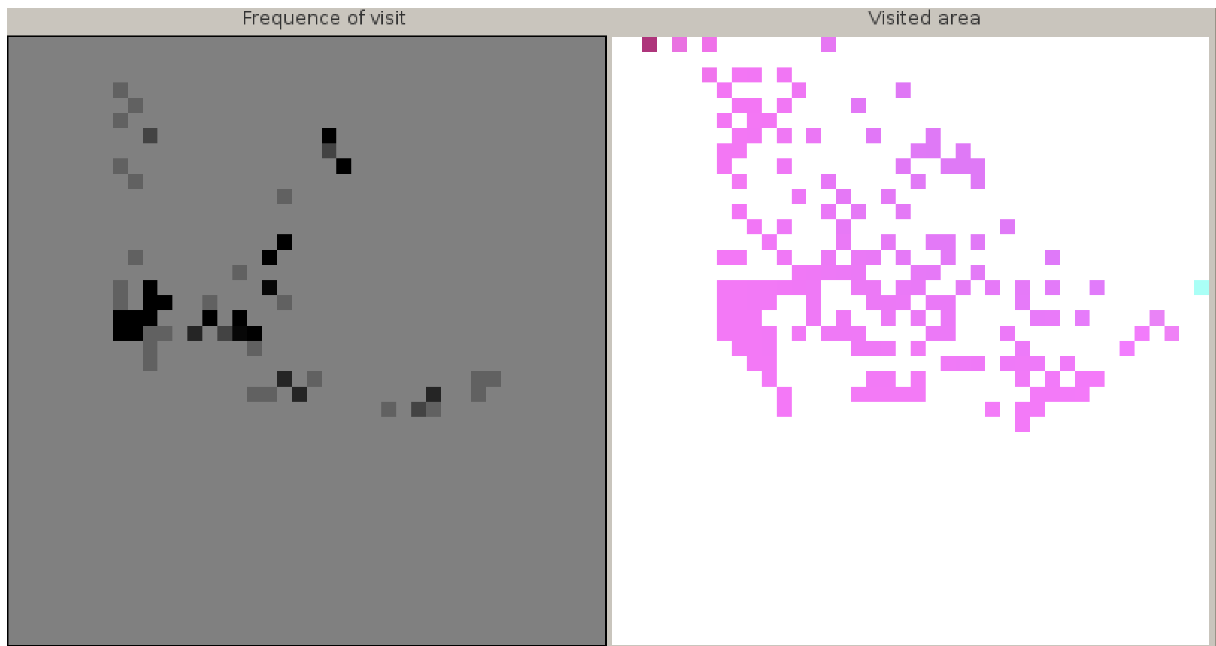

Figura 4.8: Cenário Beta: imagem que representa o SOM para o visualizador de PDF's e sua matriz de frequências.

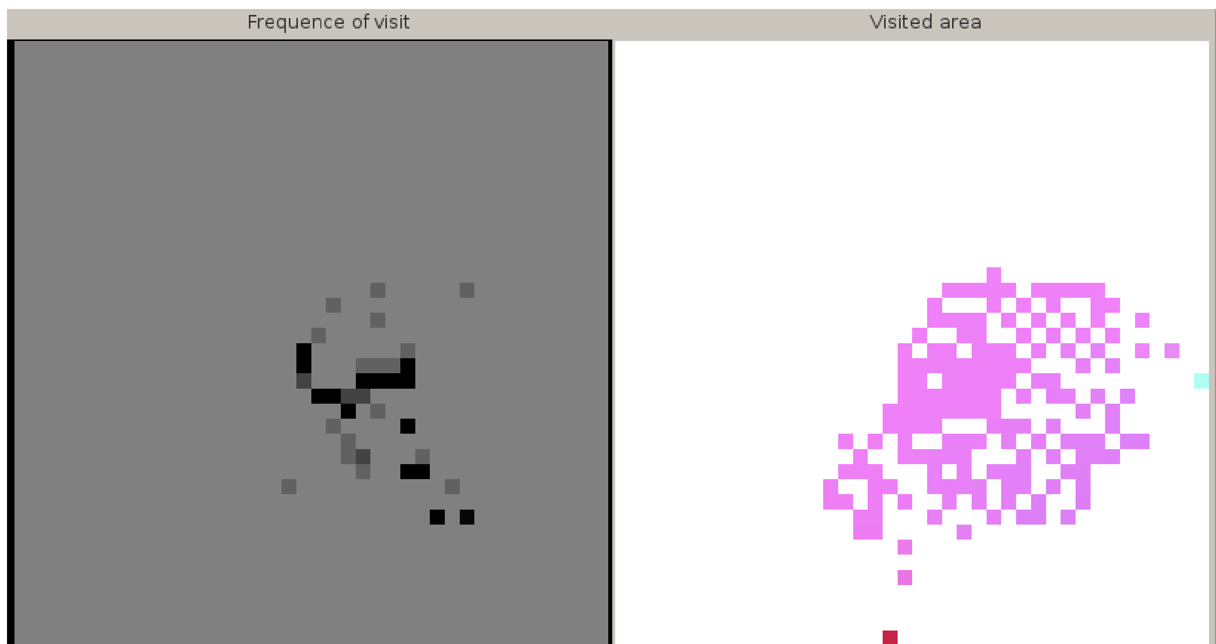

Figura 4.9: Cenário Gama: imagem que representa o SOM para o visualizador de PDF's e sua matriz de frequências.

4. Calcular o BMU para cada valor do vetor construído no item anterior. Comparar esse BMU com a matriz de frequências de cada aplicação.

O algoritmo descrito foi implementado em um programa e testado na plataforma ARM, mais detalhes na seção 2.4.1. O código mostrado em 4.1, escrito em linguagem C, ilustra o algoritmo apresentado:

A variável `MAX_INPUTS` possui o tamanho do vetor das amostras de memória livre em tempo real. O programa foi feito para realizar leituras no `/proc/sys/vm/memfree` a cada segundo. Assim, o programa possui uma latência de `MAX_INPUTS` segundos, tirando o tempo de processamento, para calcular os **fatores de semelhança**. Esse fator de semelhança é a quantidade de vezes que o BMU calculado em tempo real é igual ao BMU de uma das 3 matrizes de frequências, dado um dos cenários Alpha, Beta ou Gama.

O objetivo dessa função é calcular um *fator de semelhança*, ou seja, o maior valor entre `pdf_factor`, `browser_factor` e `canola_factor`, indica ainda qual matriz de frequências teve mais BMUs em comum em relação à coleta de memória livre do sistema. Assim, o maior valor entre as 3 variáveis indica qual aplicação está sendo executada em um determinado momento.

Identificando qual aplicação está sendo executada, e conhecendo seu comportamento de consumo de memória, pode-se decidir qual configuração passar para o CC, em termos de tamanho, e ainda qual valor para o `swappiness` e `min_free_kbytes`, ambas variáveis de configurações do kernel do Linux. Isso é feito em tempo de execução.

4.1.1 Daemon para adaptar o Cache Comprimido

No Linux, programas do tipo *daemon* são programas que possuem sua execução sendo feita em *background*, ou seja, são programas que não esperam uma interação direta do usuário.

Para adaptar o CC ao consumo de memória, foi desenvolvido um *daemon* que, dentre outras operações, utiliza o Algoritmo 4.1 para determinar qual perfil executar do CC. Esse *daemon* também é responsável por decidir que configuração realizar no CC, ou ainda, quando ligá-lo ou desligá-lo. O Algoritmo 4.2 mostra a função *main* do *daemon*.

Note que cada perfil adotado executa ações diferentes no sistema. Para o perfil `ALPHA`, o CC é desligado e a área de swap virtual é retirada através do comando do Linux, `swapoff`. Para o perfil `BETA`, o CC é ligado com tamanho aproximado de 5MB, nesse caso espera-se um consumo mediano da memória. Para o perfil `GAMA`, o CC é ligado com tamanho aproximado de 10MB e `swappiness` = 60, o que faz as páginas irem mais cedo

Algoritmo 4.1 Algoritmo usado para calcular o fator de semelhança entre os BMU's.

```
int calculate_profile() {
    struct rss_list *head = NULL;
    struct rss_list *iterator = NULL;
    int x, i, j;
    unsigned int current_rss = 0;
    unsigned int new_rss;
    struct som_node *bmu = NULL;
    int veloc = 0;
    int accel = 0;
    int browser_factor = 0;
    int canola_factor = 0;
    int pdf_factor = 0;

    i = j = 0;
    /* Collect memFree for x seconds */
    populate_mem_free_array();

    /* Get the head pointer for rss_list */
    head = read_mem_free_log();
    iterator = head;

    for (x = 0; x < MAX_INPUTS; x++) {
        new_rss = iterator->rss_pages;
        iterator = iterator->next;

        /* Get the closest bmu */
        bmu = get_bmu_xy(grids,
                         &new_rss,
                         &current_rss,
                         &veloc,
                         &accel);
            i = bmu->xp;
            j = bmu->yp;

            if (freq_pdf[i][j] != 0) {
                pdf_factor = pdf_factor + freq_pdf[i][j];
            } else if (freq_browser[i][j] != 0) {
                browser_factor = browser_factor + freq_browser[i][j];
            } else if (freq_canola[i][j] != 0) {
                canola_factor = canola_factor + freq_canola[i][j];
            }
    }
    if ((pdf_factor > browser_factor) && (pdf_factor > canola_factor)) {
        return ALPHA_PROFILE;
    } else if (browser_factor > canola_factor)
        return GAMA_PROFILE;
    else
        return BETA_PROFILE;
}
```

Algoritmo 4.2 Algoritmo usado no daemon de adaptatividade.

```
int main()
{
    /* Starts profile as ALPHA_PROFILE: CC is not configured */
    current_profile = ALPHA_PROFILE;

    do {
        if (current_profile != previous_profile) {
            if (current_profile == ALPHA_PROFILE) {
                printf("ALPHA PROFILE\n");
                system("sh unuse_compcache.sh");

                previous_profile = ALPHA_PROFILE;

                /* Save trained som in grids struct */
                read_trained_som(NO_CC_DIR"saved_som.txt", grids);

                init_freq_tables();

                read_freq_log(NO_CC_DIR"freq-pdf.log", freq_pdf);
                read_freq_log(NO_CC_DIR"freq-browser.log", freq_browser);
                read_freq_log(NO_CC_DIR"freq-canola.log", freq_canola);

                read_max_min_values(NO_CC_DIR"max_min_values.txt");
            } else if (current_profile == BETA_PROFILE) {
                printf("BETA PROFILE\n");
                system("sh unuse_compcache.sh");
                system("sh use_compcache.sh 5120");

                previous_profile = BETA_PROFILE;

                /* Save trained som in grids struct */
                read_trained_som(WITH_CC_DIR"saved_som.txt", grids);

                init_freq_tables();

                read_freq_log(WITH_CC_DIR"freq-pdf.log", freq_pdf);
                read_freq_log(WITH_CC_DIR"freq-browser.log", freq_browser);
                read_freq_log(WITH_CC_DIR"freq-canola.log", freq_canola);

                read_max_min_values(WITH_CC_DIR"max_min_values.txt");
                } else if (current_profile == GAMA_PROFILE) {
                    printf("GAMA PROFILE\n");
                    system("sh unuse_compcache.sh");
                    system("sh use_compcache.sh 10240");
                    system("echo 60 > /proc/sys/vm/swappiness");

                    previous_profile = GAMA_PROFILE;

                    /* Save trained som in grids struct */
                    read_trained_som(WITH_CC_DIR"saved_som.txt", grids);

                    init_freq_tables();

                    read_freq_log(WITH_CC_DIR"freq-pdf.log", freq_pdf);
                    read_freq_log(WITH_CC_DIR"freq-browser.log", freq_browser);
                    read_freq_log(WITH_CC_DIR"freq-canola.log", freq_canola);

                    read_max_min_values(WITH_CC_DIR"max_min_values.txt");

                    }
            }

                current_profile = calculate_profile();
        } while(1);

        return 0;
}
```

para o swap virtual onde são comprimidas e armazenadas. Nesse perfil esperamos um consumo intenso da memória ou a possível falta da mesma.

Cada perfil é encontrado usando a função apresentada no Algoritmo 4.1. Esse algoritmo utiliza a rede neural treinada, as matrizes de frequência do BMU de cada aplicação (Browser, Canola e PDF viewer) e as coletas da memória livre durante 20 segundos (valor da variável MAX_INPUTS). A cada 20 segundos, um vetor contendo 20 leituras da memória consumida é passado para a função apresentada no Algoritmo 4.1. Essa função determina qual das 3 matrizes de frequência teve BMUs mais visitados. A matriz de frequência que teve mais BMUs visitados indica que aplicação está sendo executada naquele instante, ou ainda, qual é o perfil de consumo da memória. O tempo de 20 segundos foi escolhido com base na quantidade de dados necessária para se identificar qual perfil de consumo de memória. Utilizando um tempo de leitura menor que 20 segundos, ou 20 iterações, observou-se que a rede neural não conseguia identificar o comportamento tendencioso do consumo.

O acerto em determinar o perfil de memória depende da quantidade de coletas que serão feitas antes da análise nas matrizes de frequências. Porém, coletar a memória leva, como dito anteriormente, `MAX_INPUT` segundos, onde esse valor é o tamanho do vetor de coletas da memória livre. Assim, existe um ajuste para que o *daemon* não leve muito tempo para determinar em que perfil está ou para qual perfil o consumo de memória está caminhando.

Na Figura 4.10, está representada a memória livre ao longo de um teste feito para checar se a adaptatividade do CC trás benefícios ou não. O teste consiste em abrir 8 instâncias do Browser, tocar um vídeo no Canola, simultaneamente e ainda abrir um arquivo PDF. De acordo com que a memória vai sendo consumida, os BMUs das leituras feitas em tempo real visitam mais as áreas avermelhadas do SOM treinado para o perfil (Alpha, Beta ou Gama) atual. A cada 20 segundos é feita a avaliação do consumo de memória e determinado o perfil do mesmo. De acordo com o perfil determinado, o CC é ligado ou desligado, o swappiness é configurado ou não. (mais detalhes, ver Algoritmo 4.2). Se o padrão de consumo da memória se mantém constante, não há mudança de perfil.

Ainda na Figura 4.10, pode-se notar que a memória livre (MemFree), varia bastante ao longo do tempo do teste. Também podemos notar que existem alguns picos de liberação da memória. Isso se deve ao fato de que quando o perfil Alpha é selecionado, o swap virtual é retirado da memória e acontece um aumento na memória livre. Também podemos notar que o uso do *daemon* mantém uma média de memória livre maior do que em casos onde ele não é utilizado (Ver Figura 4.11, para comparação). Mas a performance das aplicações

quando o Cache Comprimido é utilizado sofre uma pequena degradação pois o tempo de compressão/descompressão das páginas é somado ao tempo de iniciação da aplicação.

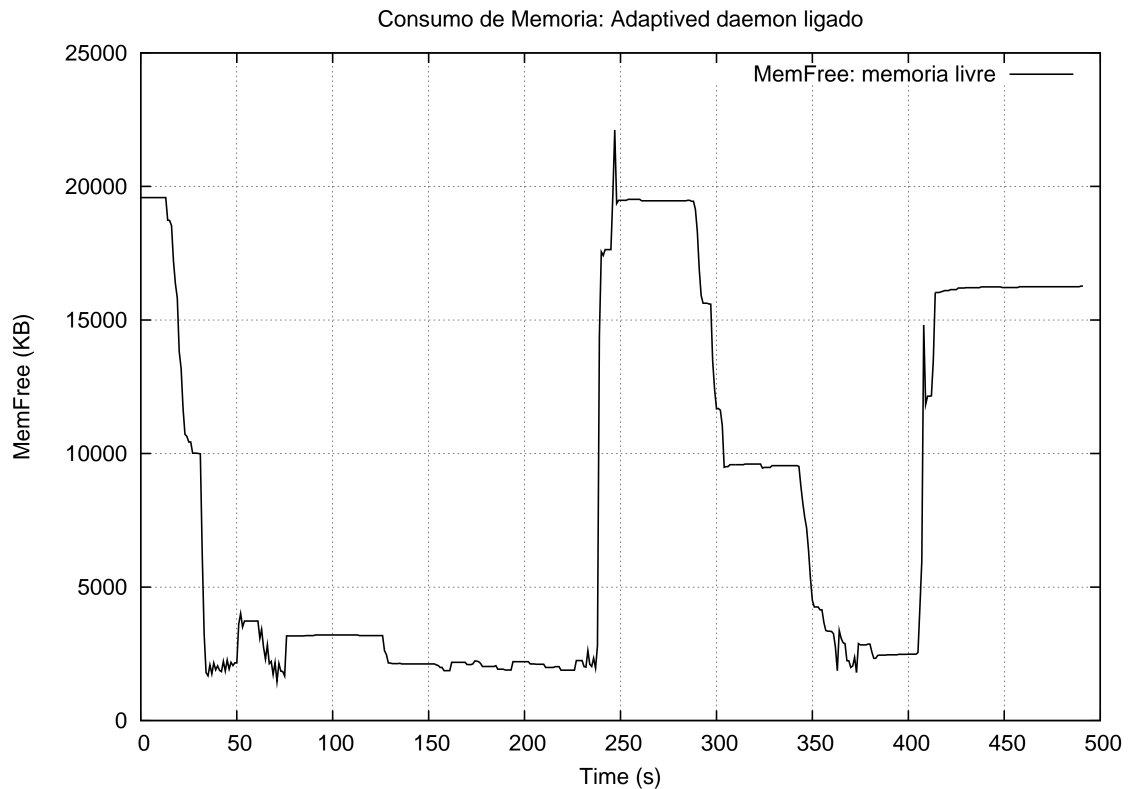

Figura 4.10: Memória livre (MemFree) ao longo do teste com o CC adaptativo. Os picos de memória livre indicam que quando o perfil de consumo de memória é trocado, existe uma liberação maior e um aumento da memória livre total.

Na Figura 4.11, está plotado um gráfico que mostra a variação na memória livre para cada uma das aplicações testadas. Os dados mostram como a memória é consumida ao longo do tempo quando o CC está desligado. Comparando com a Figura 4.10, pode-se notar que a memória média é maior quando o *daemon* desenvolvido (chamado de adaptived), está sendo executado.

4.2 Sumário

Neste Capítulo foi apresentada uma heurística e a solução encontrada para utilizar redes neurais e perfis de consumo de memória na adaptatividade do Cache Comprimido. Também foi mostrado que um *daemon* foi implementado para configurar o CC e a área de swap virtual afim de obter um uso mais eficiente da memória total do sistema.

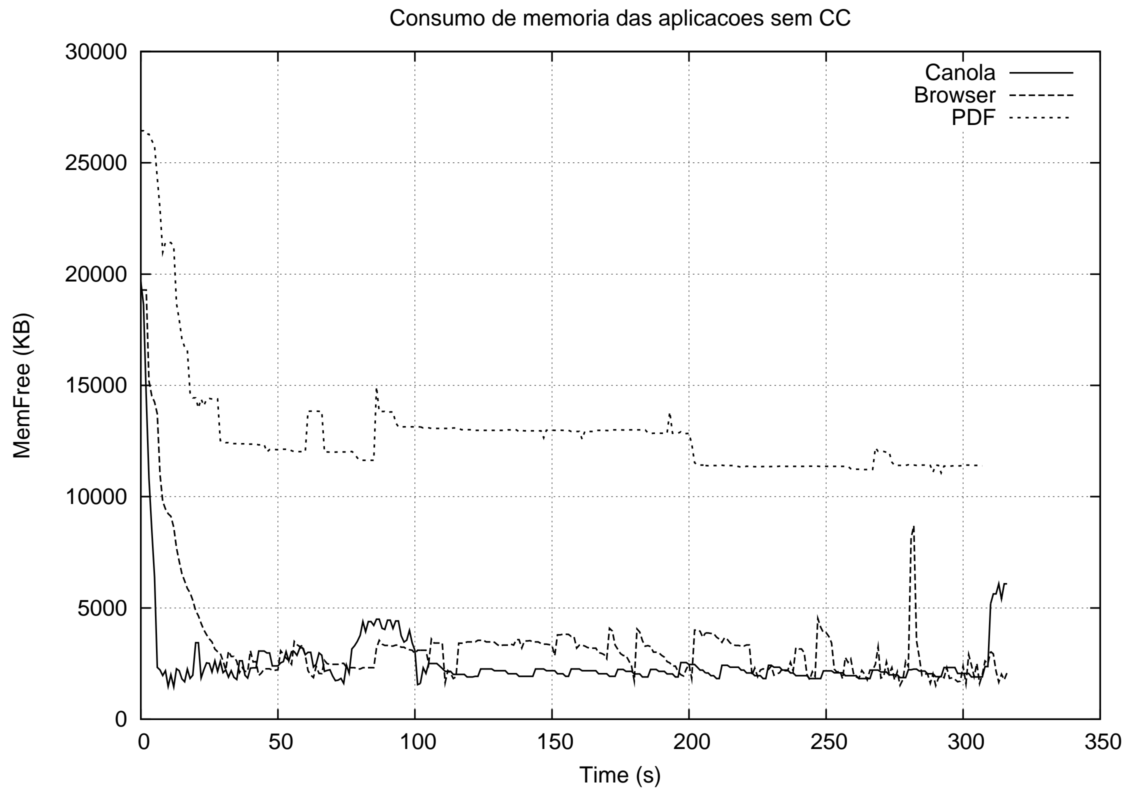

Figura 4.11: Memória livre (MemFree) para cada aplicação sem CC.

Nos gráficos de consumo de memória das aplicações, notamos que o daemon deixa uma memória livre média maior do que quando o CC está desligado, indicando assim um consumo de memória mais otimizado.

Capítulo 5

Conclusão e Trabalhos Futuros

Neste Capítulo são apresentados algumas ideias para trabalhos futuros envolvendo o Cache Comprimido e a adaptatividade. Também é apresentada a conclusão deste trabalho de mestrado.

5.1 Conclusão

Neste trabalho de mestrado foi mostrada uma solução para otimização da memória livre no Linux através da compressão de dados. Os dados estão armazenados em páginas de memória e são comprimidos quando são enviados para uma área de swap virtual. Essa solução é chamada Cache Comprimido.

Nesta dissertação foram realizados vários testes com o Cache Comprimido. Todos os testes foram realizados em uma máquina real, ou seja, sem virtualização ou simulação. Essa máquina compreende um dispositivo móvel com Linux embarcado, 128MB de RAM e um processador ARM1136 com 400Mhz de *clock*. O dispositivo móvel possui tela sensível ao toque e um conjunto de aplicações pré-instaladas. Essa aplicações são, na sua maioria, para acessar os serviços da Internet e também prover acesso ao conteúdo multimídia instalado.

Os testes com o Cache Comprimido visam identificar se é benéfico ou não o uso desse mecanismo no consumo de memória das aplicações e na performance das mesmas. Foi verificado que, com o uso do Cache Comprimido, existe um aumento da memória livre disponível, resultando assim na possibilidade de abrir outras aplicações simultaneamente ou melhorar a velocidade de respostas das aplicações que já estão em execução.

Porém, também foi verificado que em situações onde o consumo de memória é altíssimo, a memória alocada para o Cache Comprimido se torna prejudicial e o ideal seria que seu tamanho fosse diminuído, ou ainda que existisse um mecanismo para liberar a

memória consumida pelo próprio Cache Comprimido.

O Cache Comprimido é estático. Ou seja, seu tamanho não varia de acordo com o consumo de memória, ou ainda, de acordo com alguma heurística que busque a otimização da sua utilização. Foi mostrado que um Cache Comprimido de tamanho errado pode gerar mais situações de falta de memória, chamando um dispositivo do kernel chamado *Out-of-memory killer* (OOM killer).

Após os testes com o Cache Comprimido, redes neurais foram treinadas para gerar os SOMs e assim encontrar um padrão de consumo para cada aplicação testada: Browser, Canola (*media player*) e PDF *viewer*. Essas aplicações foram escolhidas pois representam ações reais que um usuário comum poderia executar nesse tipo de dispositivo móvel.

Foi verificado um padrão de consumo para cada aplicação testada dentro de cada cenário: Alpha – sem Cache Comprimido, Beta – com Cache Comprimido e Gama – Cache Comprimido e configuração extra para swap. Também mostramos que o consumo de cada aplicação é diferente dado um cenário.

Como proposto nesta dissertação, a classificação dos padrões de consumo de memória utilizando redes neurais e Mapas Auto-organizáveis (ou SOMs), desenvolvida em [2] e [19], é bastante útil para determinar padrões de consumo de memória e pré-determinar um comportamento nesse consumo. Utilizando essa ferramenta, foram determinados perfis de consumo de memória com o propósito de implementar a adaptatividade do CC. Essa adaptatividade consiste em implementar um programa que identifique o padrão de consumo e configure o CC de acordo com esse padrão, tudo feito em tempo de execução. Utilizando os mapas gerados pela análise de cada aplicação, pode-se avaliar em quais momentos é necessário que o tamanho do CC seja alterado para suprir a necessidade de memória livre do sistema.

Com a criação de um mapa com casos de uso que simulem a utilização do sistema por um usuário final, é possível determinar quais são as áreas mais acessadas por determinadas aplicações. Nesse caso, é possível verificar qual é o comportamento de consumo de memória para um determinado caso de uso e assim, tomar medidas para que não falte memória. Essas medidas podem ser o aumento ou diminuição do tamanho do CC, a mudança para um algoritmo de compressão mais eficaz (porém mais lento), etc. A tripla <memória, vcm, tvcm>, pode ser utilizada para "prever"o consumo de memória baseado no consumo anterior e no perfil (mapa) da aplicação analisada. O parâmetro "tvcm"é especialmente importante pois pode ser utilizado como a "velocidade"que uma aplicação aloca mais memória. Muito útil para evitar que situações que o OOM *killer* seja invocado.

A heurística proposta para implementar a adaptatividade ao Cache Comprimido utilizando SOMs e redes neurais mostrou-se eficiente, aumentando a quantidade média de memória livre mesmo quando o consumo de memória é alto. Essa conclusão leva a acreditar que é possível executar as aplicações com mais memória disponível, ou ainda, executar um número maior de aplicações com a mesma quantidade de memória, sem a necessidade de alterações de *hardware* (aumentando a capacidade da memória, por exemplo).

5.2 Trabalhos Futuros

Quanto ao Cache Comprimido, existem alguns pontos, apresentados em [12] de melhorias e implementação de novas características:

- *Swap free notification*: hoje, o swap virtual (ramzswap) não possui um mecanismo que libere as páginas que não são mais necessárias, como páginas de processos que foram finalizados.
- Implementar uma heurística que também contemple uma área de swap real, ou seja, quando o CC estiver sem capacidade de armazenas mais páginas, redirecioná-las para um swap real. No caso de sistemas embarcados essa pode não ser uma solução ótima pois esses sistemas geralmente são *swapless* (sem área de swap real).

No daemon implementado com a adaptatividade, o SOM representando a rede neural treinada é carregado para cada perfil adotado. Isso é um problema se tivermos muitos perfis ou se usarmos uma rede neural com uma grade maior de neurônios (a grade atual tem tamanho $40x40$).

Uma característica ao daemon que pode ser implementada é a identificação dos perfis em um tempo menor. Assim o tempo de reação do daemon ao consumo de memória pode ser menor, evitando casos em que a memória atinja um nível mínimo entre uma leitura e outra. Hoje o tempo mínimo para identificar um perfil é de 25 segundos, que é o tempo gasto para coletar as informações de memória livre. Esse tempo não está levando em conta o tempo de processamento para achar o BMU.

Um outro trabalho futuro que visa a otimização do consumo da memória em Linux embarcado é alterar o algoritmo de seleção das páginas de memória que vão para a área de swap (o PFRA, *Page Frame Reclaiming Algorithm*). Implementando flags de identificação nas páginas afim de definir quais são comprimidas, seria possível tratá-las de forma diferente e adiar ou antecipar a compressão e descompressão, otimizando o overhead de

leitura e escrita das páginas.

O Cache Comprimido usado neste trabalho faz uso da memória RAM do Linux. Existem soluções mais especializadas que fazem uso de uma memória em cache (L1 ou L2), ou ainda um chip especial para armazenar dados comprimidos, sem necessariamente utilizar a memória RAM.

Referências Bibliográficas

[1] Ai-Junkie. Kohonen's self organizing feature maps. http://www.ai-junkie.com/ann/som/som1.html.

[2] Francisco Alecrim. Análise de padrões de consumo de memória no linux baseado em mapas auto-organizáveis. Monografia. Departamento de Ciência da Computacão - UFAM, 2007.

[3] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *ASPLOS*, pages 96–107, 1991.

[4] Multimedia Card Association, 2008. http://www.mmca.org/technology/about/.

[5] Christian Borgelt. Self-organizing map training visualization. http://fuzzy.cs.uni-magdeburg.de/˜borgelt/doc/somd/, 2000. School of Computer Science - Otto-von-Guericke-University of Magdeburg.

[6] Anderson F. Briglia, Allan Bezerra, Nitin Gupta, and Leonid Moiseichuk. Evaluating effects of cache memory compression on embedded systems. In *Proceedings of the 2007 Linux Symposium*, 2007.

[7] R. Cervera, T. Cortes, and Y. Becerra. Improving application performance through swap compression. In *Proceedings of the USENIX Technical Conference (Freenix track)*. USENIX Association, jun 1999.

[8] Rodrigo S. de Castro, Alair Pereira do Lago, and Dilma Da Silva. Adaptive compressed caching: Design and implementation. In *SBAC-PAD*, pages 10–18. IEEE Computer Society, 2003.

[9] Rodrigo Souza de Castro. Cache comprimido adaptativo: projeto, estudo e implementacão. Dissertacão de Mestrado. Instituto de Matemática e Estatística. Universidade de São Paulo, 2003.

[10] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *USENIX Winter*, pages 519–529, 1993.

[11] Tom Germano. Self organizing maps. http://davis.wpi.edu/˜matt/courses/soms/, 1999.

[12] Nitin Gupta. Compressed caching for linux, site do projeto, 2009. http://code.google.com/p/compcache.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, dec 2001.

[14] Simon Haykin. *Redes Neurais - Princípios e prática*. Bookman, 2nd edition, 1999.

[15] Timo Honkela. *Self-Organizing Maps in Natural Language Processing*. PhD thesis, Helsinki University of Technology - Neural Networks Research Centre, P.O. Box 2200 FIN-02015 HUT, FINLAND, 1997. http://www.mlab.uiah.fi/˜timo/som/thesis-som.html.

[16] Scott Frederick Kaplan. *Compressed caching and modern virtual memory simulation*. PhD thesis, University of Texas at Austin, 1999.

[17] Teuvo Kohonen. An introduction to neural computing. *Neural Networks*, 1:3–16, 1988.

[18] Ben Krose, Ben Krose, Patrick van der Smagt, and Patrick Smagt. *An introduction to Neural Networks*. The University of Amsterdam, 1993.

[19] Maurício Tia Ni Gong Lin. Metodologia para classificacão de padrões de consumo de memória no linux baseado em mapas auto-organizáveis. Dissertacão de Mestrado. Departamento de Ciência da Computacão - UFAM, 2006.

[20] LinuxDevices.com. Nokia unveils linux-powered n810 internet tablet, 2007. http://linuxdevices.com/news/NS3669465936.html.

[21] Robert Love. *Linux kernel development*. Novell Press, Indianapolis, IN, USA, second edition, 2005.

[22] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *ECRTS*, pages 79–86. IEEE Computer Society, 2004.

[23] Markus Franz Xaver Johannes Oberhumer. Implementaćão e código-fonte do algoritmo de compressão lzo, 2005. http://www.oberhumer.com/opensource/lzo/lzodoc.php.

[24] Juan Quintela. Memtest, 2006. http://carpanta.dc.fi.udc.es/ quintela/memtest/.

[25] Steve Slaven. Xautomation, 2006. http://hoopajoo.net/projects/xautomation.html.

[26] Irina Chihaia Tuduce and Thomas R. Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference, General Track*, pages 237–250. USENIX, 2005.

[27] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. *SIGPLAN Notices*, 26(3):45–52, 1991.

[28] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 101–116, 1999.

[29] Wayne Wolf. *Computer as Components*. O'Reilly Associates, 1st edition, 2000.

[30] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat T. Chakradhar. CRAMES: compressed RAM for embedded systems. In Petru Eles, Axel Jantsch, and Reinaldo A.Bergamaschi, editors, *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2005, Jersey City, NJ, USA, September 19-21, 2005*, pages 93–98. ACM, 2005.

[31] Ziv and Lempel. A universal algorithm for sequential data compression. *IEEETIT: IEEE Transactions on Information Theory*, 23, 1977.

Capítulo 6

Apêndices

6.1 Apêndice - A

Algoritmos utilizados em [19], para iniciação dos pesos sinápticos dos neurônios da grade, encontrar o BMU e cálculo da distância euclidiana. Esses algoritmos são parte do treinamento das redes neurais e geração dos SOM's.

Algoritmo 6.3 Inicialização dos pesos sinápticos dos neurônios da grade.

```
void init_grid(struct som_node * grids[][GRIDS_YSIZE]) {
    int i, j, k;
    srand(time(0));
    for (i=0; i<GRIDS_XSIZE; i++) {
        for (j=0; j<GRIDS_YSIZE; j++) {
            grids[i][j] = (struct som_node *)
                    malloc(sizeof(struct som_node));
            grids[i][j]->xp = i;
            grids[i][j]->yp = j;
            for (k=0; k<WEIGHTS_SIZE; k++) {
                grids[i][j]->weights[k] = rand_float();
            }
        }
    }
}
```

6.2 Apêndice - B

}

Algoritmo 6.4 Pesquisa o neurônio vencedor (BMU) durante o processo competitivo.

```
struct som_node * get_bmu(double input_vector[],
                      int len_input,
                      struct som_node * grids[][GRIDS_YSIZE]
                      ) {
    struct som_node *bmu = grids[0][0];
    double best_dist = euclidean_dist(input_vector,
                                      bmu->weights,
                                      len_input);
    double new_dist;
    int i, j;
    for (i=0; i<GRIDS_XSIZE; i++) {
        for (j=0; j<GRIDS_YSIZE; j++) {
            new_dist = euclidean_dist(input_vector,
                                      grids[i][j]->weights,
                                      len_input);
            if (new_dist < best_dist) {
                bmu = grids[i][j];
                best_dist = new_dist;
            }
        }
    }
    return bmu;
}
```

Algoritmo 6.5 Cálculo da distância Euclidiana.

```
double euclidean_dist(double *input, double *weights, int len) {
    double summation = 0;
    double temp;
    int i;
    for (i=0; i<len; i++) {
        temp = (input[i]-weights[i]) * (input[i]-weights[i]);
        summation += temp;
    }
    return summation;
}
```

Algoritmo 6.6 Função que lê os dados do saved_som.txt.

```
int read_trained_som(char filename[], struct som_node * grids[][GRIDS_YSIZE]) {
    FILE * fp;
    ssize_t bytes_read;
    size_t len = 0;
    char * line = NULL;
    char * input[WEIGHTS_SIZE];
    int i, j;

    fp = fopen(filename, "r");

    if (fp == NULL) {
        return 0;
    }

    for (i=0; i<WEIGHTS_SIZE; i++)
        input[i] = (char *)malloc(10);

    while((bytes_read = getline(&line, &len, fp)) != -1) {
        sscanf(line, "(%d, %d) %s %s %s",
                     &i, &j, input[0], input[1], input[2]);

        grids[i][j] = (struct som_node *)
                     malloc(sizeof(struct som_node));
        grids[i][j]->xp = i;
        grids[i][j]->yp = j;
        grids[i][j]->weights[0] = strtod(input[0], NULL);
        grids[i][j]->weights[1] = strtod(input[1], NULL);
        grids[i][j]->weights[2] = strtod(input[2], NULL);
    }

    for (i=0; i<WEIGHTS_SIZE; i++)
        free(input[i]);

        if (line)
            free(line);

        if (fclose(fp)) {
            fprintf(stderr,
                         "error closing file %s: %s\n",
                         filename, strerror(errno));
            exit(EXIT_FAILURE);
        }
        return 1;
}
```

Capítulo 7

Anexos

7.1 Anexo - I

Programas em linguagem Python utilizados para interagir com o sistema gráfico e as aplicações, automaticamente.

Algoritmo 7.7 Programa para automatizar o teste com o Browser.

```
#!/usr/bin.python

import time
from os import system

class TestBrowser:

    def __init__(self):

        self.send_url_cmd = 'run-standalone.sh dbus-send --type=method_call --dest=com.nokia.osso_browser
        /com/nokia/osso_browser com.nokia.osso_browser.load_url string:'
        self.urls = ['www.uol.com.br', 'www.acritica.com.br', 'www.kibeloco.com.br',
                'www.wikipedia.org', 'www.humortadela.com.br', 'www.google.com',
                'www.ufam.edu.br', 'www.gmail.com',
                'sanguedelpredador.briglia.net', 'techblog.briglia.net']
        self.wcount = 0
        self.__steps = []

    def close_window(self):
        cmd = ("xte -x :0.0 'mousemove 776 25' 'mouseclick 1'", 2)
        self.add_step(cmd)

    def open_new_browser(self):
        '''
        This is necessary to open first browser window.
        '''
        if self.wcount == 0:
            self.add_step((self.send_url_cmd, 5))
        else:
            cmd = ("xte -x :0.0 'mousemove 150 30' 'mouseclick 1'", 1)
            self.add_step(cmd)
            cmd = ("xte -x :0.0 'mousemove 150 85' 'mouseclick 1'", 1)
            self.add_step(cmd)
            cmd = ("xte -x :0.0 'key Page_Down' 'key Page_Down' 'key Down' 'key Return'", 10)
            self.add_step(cmd)
        self.wcount = self.wcount + 1

    def open_url(self, url):
        self.open_new_browser()
        cmd = self.send_url_cmd + url
        self.add_step((cmd, 20))

    def add_step(self, cmd):
        self.__steps.append(cmd)

    def close_all(self):
        for i in xrange(0, self.wcount):
            self.close_window()

    def play_steps(self):
        '''
        Run each command stored in self.__steps.
        Uses time.sleep() to make an interval between each command.
        '''
        for cmd in self.__steps:
            #print 'CMD: ', cmd[0]
            system(cmd[0])
            time.sleep(cmd[1])

    def begin_test(self):
        for url in self.urls:
            self.open_url(url)
        self.close_all()
        self.play_steps()

if __name__ == '__main__':
    print 'Initiating tests...'
    test = TestBrowser()
    test.begin_test()
    print 'Test Finished...'
```

Algoritmo 7.8 Programa para automatizar o teste com o Canola (Parte 1/2).

```
#!/usr/bin.python

import time
from os import system

BT_Y = 222 # Since they are in horizontal, y is the same
BT_OFFSET = 150 # Position offset

'''
Positions for main buttons when they are in even layout
'''
EV_FIRST_BT_X = 170 # X position for first bt
AUDIO_BT = 0
PHOTOS_BT = 1
VIDEOS_BT = 2
SETT_BT = 3
MY_PHOTOS_BT = 1
MY_VIDEOS_BT = 1

'''
Positions for buttons when they are in uneven layout
'''
UNEV_FIRST_BT_X = 248
MY_MUSIC_BT = 0
PODCASTS_BT = 1
RADIO_BT = 2

class TestCanola:

    def __init__(self):
        self.__steps = []

    def click(self, pos, sleep):
        cmd = ("xte -x :0.0 'mousemove " + str(pos[0]) + " " + str(pos[1]) + "' 'mouseclick 1'", sleep)
        self.add_step(cmd)

    def click_back(self):
        '''
        Click back button.
        '''
        self.click((45, 455), 3)

    def click_bt(self, button, first_bt_pos):
        '''
        Click buttons presented in main window.
        '''
        x = (button * BT_OFFSET) + first_bt_pos
        self.click((x, BT_Y), 3)

    def click_list(self, pos):
        '''
        Click in a list position
        Since just seven elements are showed each time, pos must be
        checked against these values. This method does not implement
        list rolling.
        '''
        if (pos >= 0) and (pos < 7):
            self.click((160, 68 + (pos * 63)), 3)

    def play_pause_music(self):
        self.click((759, 227), 2);
```

Algoritmo 7.9 Programa para automatizar o teste com o Canola (Parte 2/2).

```
    def photo_click(self, photo):
        '''
        photo = photo number, starting in 0.
        '''
        x1 = 58
        y1 = 118
        x_offset = 146
        y_offset = 115
        if photo < 5:
            self.click(x1 + (photo * x_offset), y1)
        elif (photo >= 5) and (photo < 10):
            self.click((x1 + ((photo - 6) * x_offset), y1 + y_offset), 4)
        elif (photo >= 10) and (photo < 15):
            self.click((x1 + ((photo - 11) * x_offset), y1 + (2 * y_offset)), 4)

    def click_yes(self):
        self.click((508, 297), 2)

    def add_delay(self, delay):
        self.add_step(('', delay))

    def add_step(self, cmd):
        self.__steps.append(cmd)

    def play_steps(self):
        '''
        Run each command stored in self.__steps.
        Uses time.sleep() to make an interval between each command.
        '''
        for cmd in self.__steps:
            #print cmd
            if cmd[0] != '':
                system(cmd[0])
            time.sleep(cmd[1])

    def begin_test(self):
        # Playing a music
        self.click_bt(AUDIO_BT, EV_FIRST_BT_X)
        self.click_bt(MY_MUSIC_BT, UNEV_FIRST_BT_X)
        self.click_list(0)
        self.click_list(1)
        self.add_delay(10)
        self.play_pause_music()
        self.click_back()
        self.click_back()
        self.click_back()
        self.click_back()
        #Opening a photo
        self.click_bt(PHOTOS_BT, EV_FIRST_BT_X)
        self.click_bt(MY_PHOTOS_BT, EV_FIRST_BT_X)
        self.click_list(0)
        self.photo_click(6)
        self.click_back()
        self.click_back()
        self.click_back()
        self.click_back()
        #Opening a video
        self.click_bt(VIDEOS_BT, EV_FIRST_BT_X)
        self.click_bt(MY_VIDEOS_BT, EV_FIRST_BT_X)
        self.click_list(0)
        self.click_list(3)
        self.add_delay(210)
        self.click_back()
        self.click_back()
        self.click_back()
        self.click_back()
        # End
        self.click_back()
        self.click_yes()
        self.play_steps()

if __name__ == '__main__':
    print 'Initializing tests...'
    test = TestCanola()
    test.begin_test()
    print 'Test finished...'
```

Livros Grátis

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

Milhares de Livros para Download:

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

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo