Anderson Farias Briglia
Orientador: Dr. Edward David Moreno
Co-orientador: Raimundo Barreto

Memória Cache Comprimido Adaptativo para o kernel 2.6 do Linux utilizando Mapas Auto-organizáveis

Proposta de Dissertação apresentada ao Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Informática.

Manaus
Novembro de 2007

Sumário

1 Introdução **4**

2 Motivação **5**

3 Objetivos **6**

4 Estado Atual da Pesquisa **7**
4.1 A memória virtual do Linux 7
4.1.1 O Swap Cache 8
4.1.2 O Page Cache 9
4.2 Cache Comprimido 9
4.3 Cache Comprimido para Linux *kernel* 2.6.x 11
4.3.1 Design da Implementação 12
4.3.2 Armazenamento das Páginas Comprimidas 12
4.3.3 Operações de Inserção e Deleção das Páginas 12
4.4 Métodos de Compressão 12

5 Redes Neurais e Mapas Auto-organizáveis **12**
5.1 Treinando a rede neural 12
5.2 Mapas Auto-Organizáveis 13
5.3 Classificação de Padrões de Consumo de Memória baseado em Redes Neurais Auto-Organizáveis 17

6 Cronograma **20**

Referências **20**

Lista de Figuras

1 Hierarquia de memória com cache comprimido 6
2 Campos da estrutura `swp_entry_t` 9
3 Exemplo de rede não treinada. 14
4 Um exemplo de SOM treinado para classificação de cores. . . . 14
5 Vizinhança topológica de um BMU representada pela área sombreada. 17
6 Subintervalos que representam os estados *Low (L)*, *Medium (M)* e *High (H)*. 18
7 Os vetores são mapeados em um espaço bidimensional representado pela grade de neurônios. 19

Lista de Tabelas

1 Cronograma de trabalho. 20

1 Introdução

Segundo [5], 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. Ela também tem sido utilizada como um meio de melhorar o uso da memória cache, reduzindo assim o consumo de potência e melhorando a performance, visto que a memória cache é, geralmente, mais rápida do que a memória principal de um computador ou 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.

Cache Comprimido (CC) é uma técnica que adiciona um novo nível na hierarquia de memória do Linux [5]. 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 e reduzindo o número de páginas que vão para a área de *swap* ou que seriam descartadas caso o sistema não possuísse *swap*. 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.

Como forma de estimar o comportamento da memória, alinhando assim o tamanho do Cache Comprimido a ser utilizado, será utilizado Mapas Auto-Organizáveis com base no trabalho do Msc. Maurício Lin [13]. Os Mapas Auto-Organizáveis (do inglês SOM - *Self-Organized Maps*, como demonstrado em [13], podem ser utilizados afim de classificar determinados padrões de utilização de memória, das aplicações existentes no *file system*. A essa técnica de redimensionamento do Cache Comprimido utilizando SOM, chamaremos de Cache Comprimido Adaptativo.

2 Motivação

A memória é um dos componentes críticos que possuem maiores restrições de recursos em sistemas embarcados. Por outro lado, os sistemas têm sido aperfeiçoados através de técnicas sofisticadas, algoritmos complexos e suporte a *real time*. Como resultado, 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 da memória do dispositivo.

Em [5], 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, 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 Linux *kernel*, disponibilizando assim seu uso em dispositivos móveis mais atuais, baseados em Linux. A implementação do Cache Comprimido atual está em implementação [8], e a idéia desta dissertação é contribuir nesse projeto *Open Source*.

Na versão do Cache Comprimido Adaptativo apresentada em [8], 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 descomprimida é muito importante quando se trata de adaptatividade. Estudos realizados anteriormente em [5], 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.

3 Objetivos

O propósito principal deste trabalho é desenvolver um sistema de memória comprimida, utilizando como base a implementação *Open Source* encontrada em [8], e que implemente o conceito de adaptatividade. Experimentos realizados em trabalhos anteriores como em [5], 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 [13] e utilizar esses dados na heurística de como a área de cache comprimido deve ser dimensionada. A implementação deste trabalho, não deverá ter dependências de arquitetura, apesar de ser focada em Linux embarcado para arquitetura ARM.

A área para memória comprimida será adicionada ao sistema existente como mostrado na figura 1.

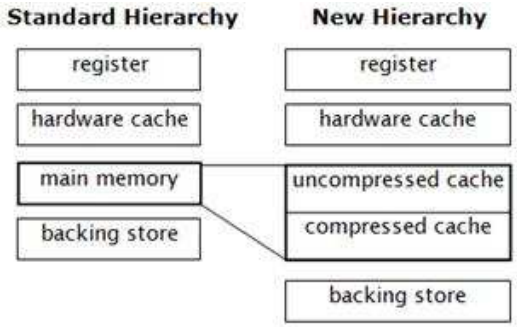

Figura 1: Hierarquia de memória com cache comprimido

A versão do *kernel* utilizada será 2.6.x, e como trabalho final será gerado um *patch* ou uma série de *patches* que aplicados à última versão disponível deverá implementar o Cache Comprimido Adaptativo. Também será gerada uma aplicação que extrairá os dados providenciados pelo SOM, durante a classificação do uso da memória. Essa aplicação exportará os dados necessários para o ajuste do tamanho do cache comprimido.

Para a validação da implementação, serão utilizados dois tipos de testes: *benchmarks* sintéticos e testes com casos de uso reais. Nos testes com *bench-*

marks, serão utilizados programas como o MemTest [16], e o LLCbench [15], que são uma suíte de testes para avaliar a performance da memória principal e da memória *cache*. No caso dos testes utilizando casos de uso, será utilizado uma ferramenta de automação chamada XAutomation [18], para criar uma interação com o sistema, simulando um uso real das aplicações, enquanto dados referentes à utilização da memória cache comprimida 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.

4 Estado Atual da Pesquisa

Esta seção tem o propósito de apresentar os estudos e as pesquisas realizadas até o momento, que são requisitos relevantes para o desenvolvimento da proposta descrita na Seção 3.

Inicialmente será apresentado o estado da arte, onde serão apresentados alguns trabalhos relacionados. Em seguida, será apresentada a atual de implementação do Cache Comprimido para a versão 2.6.x do *kernel* do Linux encontrada em [8]. Também serão apresentados alguns testes realizados com essa versão e apresentados em [4]. Por fim, será demonstrado o esquema de classificação de padrões de consumo de memória apresentado em [13] e como essa metodologia poderá ajudar na implementação da adaptatividade no Cache Comprimido atual.

4.1 A memória virtual do Linux

Páginas físicas são a unidade básica do gerenciamento de memória [14] e o MMU 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 retiradas da lista LRU de páginas inativas. O algoritmo responsável por selecionar e liberar as páginas é chamado de Page Frame Reclaiming Algorithm - PFRA.

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

Quando o sistema está sob pressão de memória, ou seja, há pouca 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.

4.1.1 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. 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. Este identificador identifica onde a página se encontra no dispositivo de bloco utilizado pelo *swap*..

Na figura 2, **'type'** identifica páginas que podem ser *swapped*.

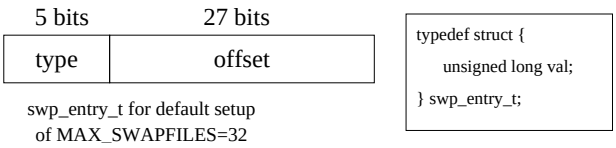

Figura 2: Campos da estrutura `swp_entry_t`

4.1.2 O Page Cache

Este é o cache utilizado para armazenar as páginas do *filesystem*. Ou seja, todo arquivo aberto pelas aplicações, possuem 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 do cache. O valor de *offset* dentro do arquivo é utilizado como chave-de-busca. Cada arquivo aberto possui uma *radix tree* própria. Paga páginas presentes na memória, o nodo correspondente na *radix tree* aponta para a página que contém uma parte do arquivo e um valor de *offset* da mesma dentro do arquivo o qual pertence.

4.2 Cache Comprimido

Em um sistema com cache comprimido, a memória é dividida em duas grandes porções: memória comprimida e não-comprimida [5], [6], [11]. 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 *workload* imposto pelas aplicações.

Muitos pesquisadores 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 [5], muito mais lenta que a principal. Apesar dessa diferença de velocidade entre a memória principal e a memória se-

cundária ser grande, quando se leva em conta os sistemas embarcados, não tem-se uma diferença muito grande. Geralmente os sistemas embarcados não possuem memória secundária armazenada em um disco rígido, outros dispositivos são utilizados nesse caso. Memórias do tipo *compact flash*, cartões MMC, são os principais dispositivos encontrados hoje no mercado, que são utilizados como memória secundária. 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 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 *workload* da memória.

Os primeiros estudos que utilizavam cache comprimido foram feitos por Wilson [24], Appel e Li [2], em 1990. Outro pesquisador, chamado Douglis[6], obteve algumas melhorias na performance do sistema, usando uma implementação de cache comprimido adaptativo, no sistema operacional Sprite. Porém, Douglis também teve alguns problemas e não conseguiu concluir se o uso de cache comprimido é útil ou não.

Tendo o trabalho de Douglis ser inconclusivo, muitos outros pesquisadores se ocuparam em estudar o caso. Em 1999, Kaplan[11] chegou à conclusão que a compressão do cache pode reduzir os custos das operações de *paging*. Kaplan ainda confirmou o que Douglis[6] 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 [17], não se trata especificamente da memória cache comprimida, mas sim da compressão da área de *swap* do sistema. No trabalho apresentado em [17], foi implementado 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 [22].

Os trabalhos discutidos anteriormente, são antigos e foram implementados em sistemas operacionais da época. Os que foram implementados em Linux, utilizava a versão 2.4.x do *kernel*. Daquela época para dos dias atuais, o *kernel* do Linux sofreu muitas melhorias e o esquema utilizado para o cache comprimido nessa versão do *kernel* é diferente. Por essa razão, as próximas seções fazem um *overview* da versão do cache comprimido para o *kernel* 2.6.x, implementada por Nitin Gupta [8] e apresentada em [4].

4.3 Cache Comprimido para Linux *kernel* 2.6.x

Dados experimentais[4] avaliados em um sistema utilizando Cache Comprimido, nos mostram que nós podemos não só melhorar as taxas de E/S (Entrada e Saída), 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 estocagem das páginas de memória comprimida. Usando um algoritmo de compressão baseado em dicionários, páginas do *page cache*, ou seja, do *filesystem* e páginas anônimas são comprimidas e distribuídas em porções de memória de tamanho variável - ***chunks***[23]. Com essa abordagem, tem-se o mínimo de framentação e uma rápida recuperação das páginas, quando estas são requisitadas pelo sistema. O tamanho do Cache Comprimido pode ser ajustado

separadamente para páginas do *page cache* e páginas anônimas. Este ajuste é efetuado através de escritas em variáveis exportadas no `procfs`, dando maior flexibidade ao usuário em relação aos casos de uso da memória.

4.3.1 Design da Implementação

4.3.2 Armazenamento das Páginas Comprimidas

4.3.3 Operações de Inserção e Deleção das Páginas

4.4 Métodos de Compressão

5 Redes Neurais e Mapas Auto-organizáveis

As *redes neurais artificiais* consistem em um método para solucionar problemas de inteligência artificial, armazenando um conhecimento experimental do problema através de técnicas de treinamento, como discutido anteriormente.

As redes neurais podem ser definidas como um grupo interconectado de neurônios artificiais que usa um modelo matemático ou computacional baseado no comportamento do cérebro humano para processamento de informações [20, 19, 9, 21]. 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 [9].

Existem duas abordagens de treinamentos para as redes neurais: o supervisionado e o não-supervisionado.

5.1 Treinando a rede neural

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 abaixo.

1. O primeiro padrão de entrada é apresentado para a rede.
2. Os pesos são ajustados para capacitar a rede de 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.

5.2 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 as vezes é chamado de mapa de Kohonen [7, 1, 10, 3].

No SOM, os neurônios estão colocados em nós de uma grade unidimensional ou bidimensional, outras dimensões também são possíveis, embora não sejam tão comuns. Os neurônios são seletivamente sintonizados a vários padrões de entrada ou classes de padrões de entrada no decorrer de um processo de aprendizagem ou treinamento.

As localizações dos neurônios assim sintonizados se tornam ordenadas entre si de modo que um sistema de coordenadas significativas para características diferentes de entrada é criado sobre a grade. Portanto um mapa

auto-organizável é caracterizado pela formação de uma mapa topográfico dos padrões de entrada, baseado nas características estatísticas intrínsecas contidas nos padrões de entrada, por isso o nome *mapa auto-organizável* [9].

Figura 3: Exemplo de rede não treinada.

Um exemplo comum usado para mostrar os princípios de SOM é o mapeamento de cores a partir de seus 3 componentes: vermelho, verde e azul ou RGB (red, green and blue) em um espaço bidimensional. A Figura 4 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.

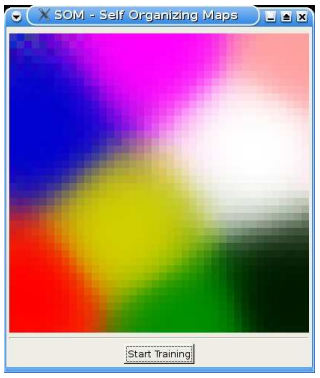

Figura 4: Um exemplo de SOM treinado para classificação de cores.

A rede SOM ilustrada na Figura 4 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.

Cada nó da grade tem uma posição topológica específica representada por uma coordenada (x, y) e contém um vetor de pesos sinápticos de dimensão igual ao vetor de entrada. Ou seja, caso os dados de treinamento consistem de vetores x de dimensão n:

$$x = [x_1, x_2, x_3, ..., x_n]$$

Então cada nó ou neurônio j da grade contém um vetor correspondente de peso w_j com a mesma dimensão n:

$$w_j = [w_{j1}, w_{j2}, w_{j3}, ..., w_{jn}]$$

A formação do SOM é feita primeiramente inicializando 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. Depois que a grade é feita, são executados 6 processos importantes[9]:

Inicialização: 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 vetor de entrada. O neurônio mais semelhante é selecionado e é considerado como o neurônio vencedor e 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 o 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 1.

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

A cada iteração do treinamento, após o BMU ter sido determinado, o passo seguinte consiste em calcular quais são os neurônios localizados na vizinhança topológica do BMU. Tais neurônios terão os seus pesos sinápticos alterados posteriormente durante o processo de adaptação sináptica. A figura 5 ilustra um exemplo de vizinhança topológica determinada no início do treinamento da rede.

Uma característica da rede SOM é que a área da vizinhança topológica diminui ao longo do treinamento da rede. A cada iteração t, a função de decaimento exponencial é calculada para permitir o reajuste do tamanho da vizinhança do BMU como mostrada na Equação 2:

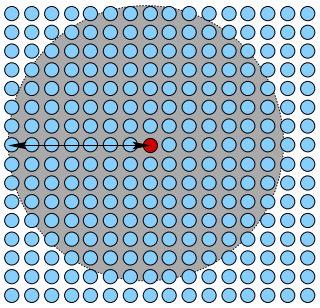

Figura 5: Vizinhança topológica de um BMU representada pela área sombreada.

$$\sigma(t) = \sigma_0 e^{(-\frac{t}{\lambda})} \tag{2}$$

onde σ_0 representa o raio σ no tempo t_0 que tem o mesmo valor do raio da grade, e λ é uma constante de tempo. A variável t representa um dado instante do treinamento, ou melhor, um passo de tempo da iteração.

5.3 Classificação de Padrões de Consumo de Memória baseado em Redes Neurais Auto-Organizáveis

Ainda baseado em [13], essa seção mostrará um pouco dos testes realizados afim de classificar padrões de consumo de memória utilizando um SOM. Os testes consistem em executar casos de uso de aplicações afim de realizar algumas medições do consumo de memória das aplicações envolvidas. Os casos de uso foram executados no sistema operacional Linux. Após a execução de alguns casos de uso, foram observadas várias variáveis afim de compor os dados de entrada para o algoritmo de SOM. A seguinte estrutura foi escolhida para representar os dados de entradas:

- 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 através da razão entre a diferença da quantidade de memória física consumida e o intervalo de tempo decorrido, conforme a Equação 3;

$$VCM = \frac{mem_2 - mem_1}{t_2 - t_1} \tag{3}$$

- *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 através da razão entre a diferença da variação do consumo de memória e o intervalo de tempo decorrido, conforme a Equação 4.

$$TVCM = \frac{vcm_2 - vcm_1}{t_2 - t_1} \tag{4}$$

Os seguintes estados podem ser atribuídos para cada propriedade apresentada: Low (L), Medium (M) e High (H). Os estados L, M e H representam respectivamente um conjunto de números pequenos, médios e grandes em um intervalo de valores estabelecidos. A definição de L, M e H são usadas para representar o comportamento de uma determinada propriedade de uma forma simples e abstrata.

Assumindo que os valores para uma determinada propriedade está em um intervalo positivo $[limite_{minimo}, limite_{maximo}]$, então o mesmo pode ser dividido em 3 subintervalos que representam os estados L, M e H, como ilustrado na Figura 6. Observe que os valores de L, M e H são dependentes do limite mínimo e máximo de um intervalo estabelecido.

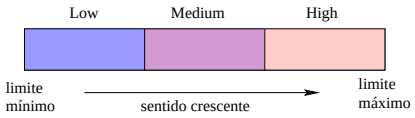

Figura 6: Subintervalos que representam os estados *Low (L)*, *Medium (M)* e *High (H)*.

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.

A tripla ¡memória,vcm,tvcm¿ pode ser mapeada facilmente para uma estrutura de dados que seja aceita algoritmo do SOM. Já que o SOM utiliza vetores tridimensionais como valores de entrada, a tripla pode facilmente ser calculada 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 7.

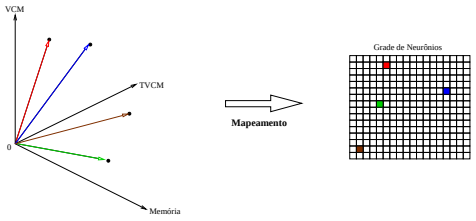

Figura 7: Os vetores são mapeados em um espaço bidimensional representado pela grade de neurônios.

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 em um determinado instante.

Para treinar a rede, utilizou-se um caso de uso que tentava reproduzir o uso normal de um computador. Neste caso de uso foram utilizadas várias aplicações gráficas como: browsers, leitores de pdf, video players e editores de texto.

6 Cronograma

O trabalho aqui proposto será dividido em etapas distintas conforme detalhadas nos itens seguintes:

1. Levantamento Bibliográfico - Coleta, organização e análise da literatura técnica relacionada com os assuntos abordados no trabalho.
2. Etapa01 -
3. Etapa02 -
4. Etapa03 -
5. Etapa04 -
6. Etapa05 -

Tarefas	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
Levantamento Bibliográfico	■	■	■	■	■				■			
Etapa01	■	■	■									
Etapa02			■	■	■							
Etapa03					■	■	■	■	■			
Etapa04									■	■	■	
Etapa05	■	■	■	■	■	■	■	■	■	■	■	■

Tabela 1: Cronograma de trabalho.

Referências

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

[2] A. W. Appel and K. Li. Virtual memory primitives for user programs. 1991. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.

[3] 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.

[4] Anderson F. Briglia, Allan Bezerra, Nitin Gupta, and Leonid Moiseichuk. Evaluating effects of cache memory compression on embedded systems, 2007. http://linuxsymposium.org/2007.

[5] Rodrigo Souza de Castro. Cache comprimido adaptativo: projeto, estudo e implementaćão. 2003.

[6] Douglis F. The compression cache: Using on-line compression to extend physical memory, 1993. http://www.lst.inf.ethz.ch/research/publications/publications/USENIX_2005/USENIX_200

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

[8] Nitin Gupta. Compressed caching for linux project's site, 2006. http://linuxcompressed.sourceforge.net/.

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

[10] 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.

[11] S. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, University of Texas at Austin, 1999.

[12] 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.

[13] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005. ISBN 0-672-32720-1.

[14] John Thurman Philip J. Mucci, Kevin London, 2007. http://icl.cs.utk.edu/projects/llcbench/index.html.

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

[16] T. Cortes R. Cervera and Y. Becerra. Improving application performance through swap compression, 1999. http://www.usenix.org/events/usenix99/full$_p$*apers/cervera/cervera.pdf*.

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

[18] Leslie Smith. *An Introduction to Neural Networks*. Department of Computing and Mathematics - University of Stirling. http://www.cs.stir.ac.uk/˜lss/NNIntro/InvSlides.html.

[19] Christos Stergiou and Dimitrios Siganos. Neural networks. Technical report, Department of Computing - Imperial College London. http://www.doc.ic.ac.uk/˜nd/surprise_96/journal/vol4/cs11/report.html.

[20] Cassia Yuri Tatibana and Deisi Yuki Kaetsu. *Uma Introducão às Redes Neurais*. Departamento de Informática - Universidade Estadual de Maringá. http://www.din.uem.br/ia/neurais/.

[21] Tuduce and Gross. Adaptative main memory compression, 2005. USENIX 2005.

[22] Irina Chihaia Tuduce and Thomas Gross. Adaptive main memory compression, 2005. http://www.lst.inf.ethz.ch/research/publications/publications/USENIX_2005/USENIX_200

[23] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems., 1999. USENIX 1999.