Simulações em Memória Cache

Anderson Pilati, César Augusto Graeff

Área do Conhecimento de Ciências Exatas e Engenharias
Universidade de Caxias do Sul (UCS)
Caxias do Sul – RS – Brasil

apilati1@ucs.br,cagraeff@ucs.br

Resumo. *Este trabalho tem por objetivo analisar o desempenho de memórias cache desenvolvendo um software para a simulação utilizando mapeamento associativo por conjunto. Os dados para a simulação são obtidos a partir da leitura de um arquivo texto que é interpretado de acordo com os parâmetros fornecidos pelo usuário.*

Abstract. *This task has as main goal analyze the performance of a CPU Cache creating a simulation software using a set associative cache. Datas are obtained from a text file that will be interpreted according to the parameters provided by the user.*

1. Simulador Desenvolvido

O software para simulação de memórias cache foi desenvolvido utilizando tecnologias web: HTML (*Hypertext Markup Language),* CSS (*Cascading Style Sheets*) e JS (*Javascript*) com a utilização dos frameworks Bootstrap e AngularJS. O simulador pode ser acessado no seguinte endereço: http://cache.megadatasistemas.com.br/.

1.1. Estrutura de Dados Utilizada

Para a simulação da memória cache é utilizado um objeto do JavaScript para o armazenamento dos conjuntos, em cada conjunto um objeto para o armazenamento das tags relativas ao conjunto e outro objeto com as informações da tag como pode ser visto abaixo:

```
vm.cache = {
    '10011101': {                   // CONJUNTO
        '011010101010': {           // TAG
            dirty: 0,               // UTILIZADO PARA WRITE-BACK
            ultima_linha: 12,       // UTILIZADO PARA LRU
            acessos: 5              // UTILIZADO PARA LFU
        }
    }
};
```

1.2. Algoritmos de Simulação

A simulação da memória cache é realizada a partir da leitura de um arquivo texto que possui a informação do endereço e da operação (leitura ou escrita) no seguinte formato:

```
00190fc0 R
0efd0fc0 W
00190fe0 R
```

O usuário informa os seguint6es parâmetros de entrada:

- Memória Cache
    - Tamanho da Linha
    - Número de Linhas
    - Associatividade
    - Tempo de Acesso
    - Politica de Escrita
    - Política de Substituição
- Memória Principal
    - Tempo de Acesso

Com base nos parâmetros informados pelo usuário é feita a leitura do arquivo linha a linha, que é separada em endereço e operação. Com posse do endereço é realizada a decodificação deste que é primeiramente convertido para binário e após separado em três partes: tag, conjunto e palavra.

O código fonte completo do algoritmo pode ser visto na seção 8.

1.2.1. Algoritmo de Leitura

A leitura é realizada buscando o bloco de acordo com a tag e conjunto do endereço requisitado, no caso do bloco já estar na cache o numero de acertos de leitura é atualizado assim como o número de acessos da tag e a informação sobre a última linha a acessar o endereço.

No caso do bloco não estar na memória cache ele é adicionado incrementando o numero de acessos de leitura na memória principal e utilizando a estrutura especificada na seção 1.1 para o armazenamento, caso o tamanho de linhas do conjunto seja igual a associatividade a política de substituição é utilizada antes de adicionar o bloco a cache de acordo com a política definida pelo usuário:

- LRU (*Least Recently Used*) – O conjunto é percorrido em busca da tag que possui o acesso menos recente comparando a variável `ultima_linha` da tag, após encontrada o bloco é removido do conjunto para a adição do novo bloco.
- LFU (*Least Frequently Used*) – O conjunto é percorrido em busca da tag com o menor número de acessos comparando a variável `acessos` da tag após encontrada o bloco é removido do conjunto para a adição do novo bloco.

- Aleatório - Uma tag é escolhida aleatoriamente e o bloco é removido do conjunto para a adição do novo bloco.

1.2.2. Algoritmo de Escrita

A escrita é realizada buscando o bloco de acordo com a tag e conjunto do endereço requisitado, no caso do bloco não estar na cache uma leitura é feita, ver seção 1.2.1, e a tag tem a variável *dirty* mudada para 1 em caso de *write-back* ou uma escrita é feita na memória principal no caso de *write-through*, incrementando o número de acessos de escrita na memória principal.

Já no caso do bloco não estar na cache o numero de acertos de escrita é atualizado assim como o número de acessos da tag e a informação sobre a última linha a acessar o endereço, em caso de *write-back* a tag tem a variável `dirty` mudada para 1 e em caso de *write-through* é incrementando o numero de acessos de escrita na memória principal.

Com a utilização de write-back a escrita na memória principal ocorre no momento da substituição do bloco (ver seção 1.2.1) ou na finalização do processamento do arquivo, assim no caso da variável `dirty` ser 1 o numero de acessos de escrita na memória principal é incrementado.

2. Impacto do Tamanho da Cache

Para medir o impacto do tamanho da memória cache foram feitos experimentos utilizando blocos com 128 bytes, política de escrita *write-through*, associatividade de 4 blocos por conjunto e política de substituição LRU (*Least Recently Used*) variando o numero de blocos entre 16 e 1024. Os resultados podem ser visualizados no gráfico abaixo:

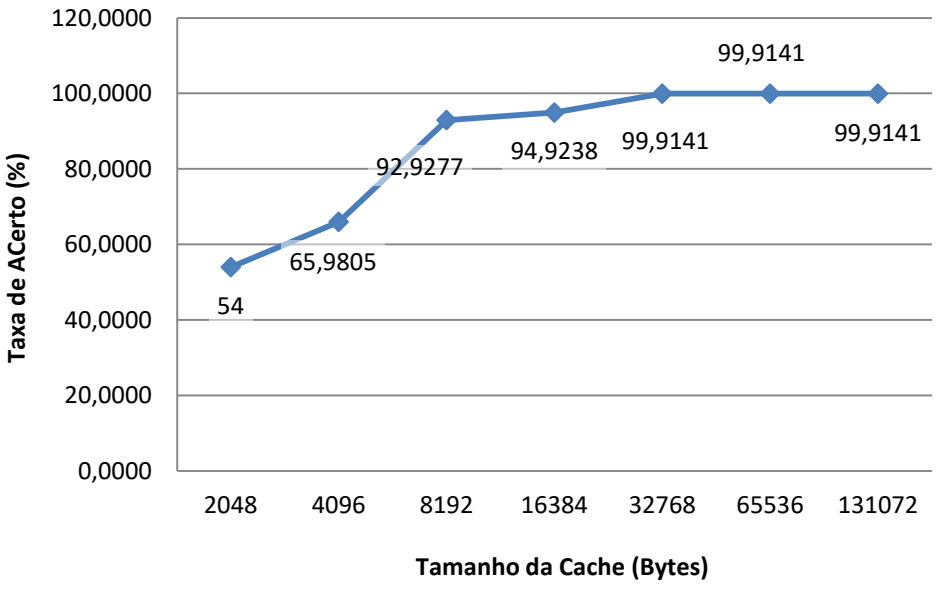


Gráfico 1. Tamanho da Cache x Taxa de Acerto

Podemos verificar no gráfico que a taxa de acerto aumenta proporcionalmente ao tamanho da cache, ou seja, quanto maior o tamanho da cache maior a taxa de acertos e menor o tempo de acesso, porém acarreta em um maior custo de hardware.

O gráfico possui este formato pois quanto maior o número de blocos da cache maior o numero de blocos armazenados e menor a chance de ocorrer uma substituição de um bloco aumentando assim a taxa de acerto e diminuindo o tempo de acesso.

3. Impacto do Tamanho do Bloco

Para medir o impacto do tamanho do bloco foram feitos experimentos utilizando uma memória cache de 8 *Kbytes*, política de escrita *write-through*, associatividade de 2 blocos por conjunto e política de substituição LRU (*Least Recently Used*) variando o tamanho do bloco entre 4 e 4096 *Bytes*. Os resultados podem ser visualizados no gráfico abaixo:

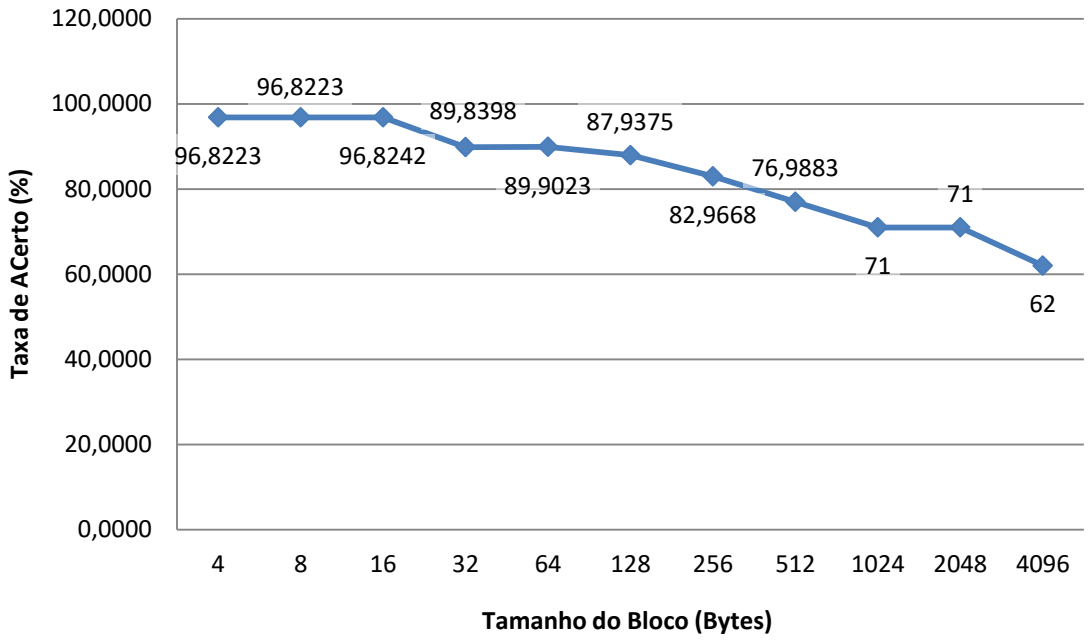


Gráfico 2. Tamanho do Bloco x Taxa de Acerto

Podemos verificar no gráfico que a taxa de acerto diminui com o crescimento do tamanho do bloco, ou seja, quanto maior o tamanho do bloco menor a taxa de acerto, isto ocorre por causa do principio da localidade espacial que diz que há uma probabilidade de acesso maior para dados e instruções em endereços próximos àqueles acessados recentemente, já que as instruções de um programa são armazenadas de forma sequencial na memória, assim como as variáveis do programa que são armazenadas próximas umas das outras, porém, com o tamanho da linha muito grande a chance do próximo endereço ser utilizado é reduzida diminuindo a taxa de acerto.

De acordo com o gráfico o tamanho ideal de bloco para esta situação seria 16 Bytes onde encontramos a maior taxa de acerto: 96,8242%.

4. Impacto da Associatividade

Para medir o impacto da associatividade foram feitos experimentos utilizando blocos com 128 bytes, política de escrita *write-back*, política de substituição LRU (*Least Recently Used*) utilizando uma memória cache de 8 *Kbytes* e variando o numero de blocos por conjunto entre 1 e 64. Os resultados podem ser visualizados no gráfico abaixo:

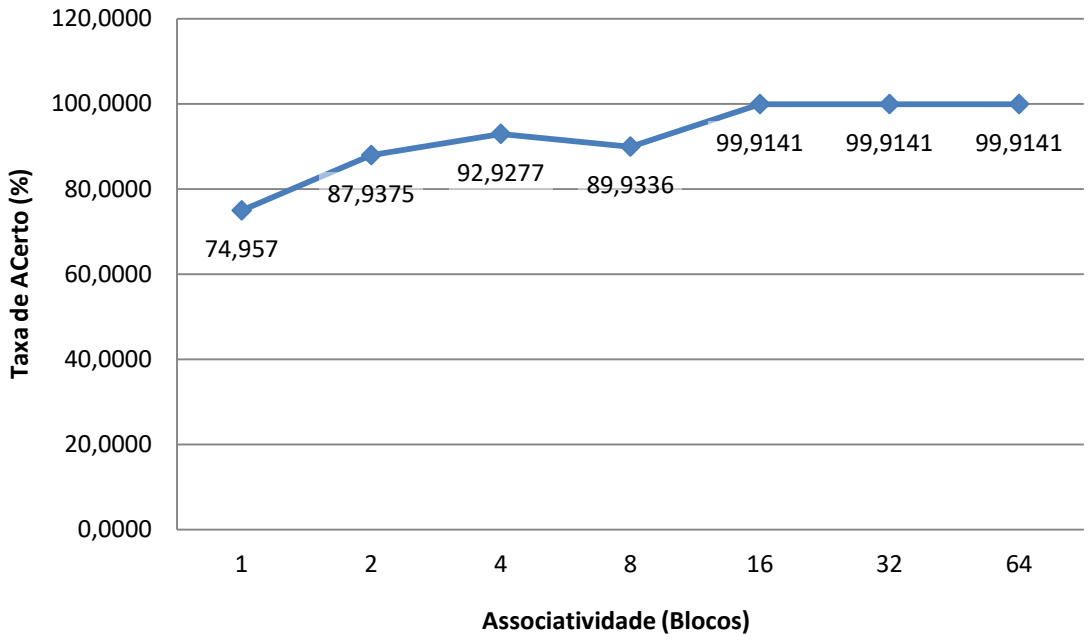
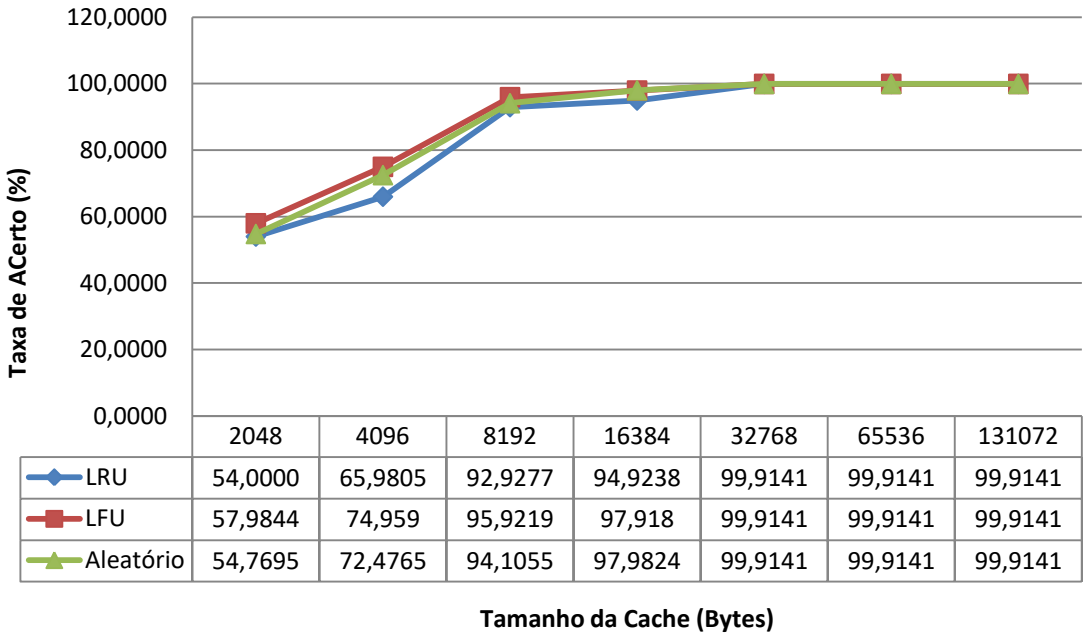


Gráfico 3. Associatividade x Taxa de Acerto

Podemos verificar no gráfico que a taxa de acerto aumenta com o crescimento da associatividade, ou seja, quanto maior o numero de blocos por conjunto maior a taxa de acerto, isto ocorre, pois quanto maior o numero de linhas por conjunto maior a flexibilidade em relação à localização do bloco na cache, diminuindo o numero de substituições necessárias e aumentando a taxa de acerto.

5. Impacto da Política de Substituição

Para medir o impacto da política de substituição foram feitos experimentos utilizando blocos com 128 bytes, política de escrita *write-through*, associatividade de 4 blocos por conjunto e variando o tamanho da cache entre 2048 e 131072 *Bytes* para cada um dos métodos de substituição: LRU (*Least Recently Used*), LFU (*Least Frequently Used*) e Aleatório. Os resultados podem ser visualizados no gráfico abaixo:



	2048	4096	8192	16384	32768	65536	131072
LRU	54,0000	65,9805	92,9277	94,9238	99,9141	99,9141	99,9141
LFU	57,9844	74,959	95,9219	97,918	99,9141	99,9141	99,9141
Aleatório	54,7695	72,4765	94,1055	97,9824	99,9141	99,9141	99,9141

Gráfico 4. Política de Substituição x Taxa de Acerto

Podemos verificar no gráfico que a taxa de acerto varia de acordo com o método de substituição utilizado e que neste caso o LFU tem um desempenho maior que o LRU e Aleatório isto ocorre, pois cada método substitui de forma diferente os blocos da cache:

- LRU (*Least Recently Used*) – A substituição ocorre removendo o bloco menos recentemente utilizado do conjunto.
- LFU (*Least Frequently Used*) - A substituição ocorre removendo o bloco menos frequentemente utilizado do conjunto.
- Aleatório – A substituição ocorre acontece escolhendo aleatoriamente um bloco do conjunto.

6. Largura de Banda da Memória

Para medir a largura de banda da memória foram feitos experimentos utilizando a política de substituição LRU (*Least Recently Used*) e variando o tamanho da cache: 8 *Kbytes* e 16 *Kbytes*, o tamanho do bloco: 64 *Bytes* e 128 *Bytes*, e a associatividade: 2 e 4 Blocos para cada uma das políticas de escrita: *Write-Through* e *Write-Back*. Os resultados podem ser visualizados na tabela baixo:

A.1 Largura de Banda da Memória

Tabela 1 – Parâmetros da Simulação

Parâmetro	**Valor**
Algoritmo de Substituição	LRU (Last Recently Used)

Tabela 2 – Resultados da Simulação

		Politica de Escrita	**Write-Through**		**Write-Back**	
Tamanho da Cache (Kb)	Tamanho do Bloco (Bytes)	Associatividade	Leituras na MP	Escritas na MP	Leituras na MP	Escritas na MP
8	64	2	5170	6144	5170	1026
8	64	4	3126	6144	3126	515
8	128	2	6176	6144	6176	1026
8	128	4	3621	6144	3621	515
16	64	2	2615	6144	2615	1026
16	64	4	2615	6144	2615	515
16	128	2	3621	6144	3621	1026
16	128	4	2599	6144	2599	515
		Média	3692,88	6144	3692,88	770,5

Podemos verificar na tabela acima que a política de escrita *write-back* possui um tráfego de memória muito menor que a política *write-through* quando comparados em relação a escrita, isto ocorre pois enquanto a política *write-through* realiza a cada escrita na memória principal enquanto a política *write-back* somente realiza a escrita na memória cache e realiza a escrita na memória principal somente ao ser substituído o bloco ou a execução ter terminado, fazendo com que o numero de escritas seja reduzido visto que de acordo com a localidade espacial as escritas normalmente são feitas em sequencia escrevendo múltiplas vezes no mesmo bloco.

7. Avaliação Global

Apesar de o simulador ser para memórias cache associativas por conjunto ele também pode ser utilizado para simular memórias cache com mapeamento direto e totalmente associativo caso os parâmetros sejam configurados corretamente, assim, se configurarmos o simulador para somente um conjunto teremos um cache totalmente associativa enquanto se configurarmos a cache para somente uma linha por conjunto teremos uma cache com mapeamento direto.

Analisando os gráficos e informações reunidas neste trabalho podemos verificar que as melhores configurações para os testes realizados para a memória cache são as seguintes:

- **Tamanho do Bloco:** Entre 4 e 64 Bytes
- **Número de Conjuntos:** 16 ou mais conjuntos
- **Associatividade:** 4 ou mais blocos por conjunto
- **Política de Escrita:** *Write-Back*
- **Política de Substituição:** LFU (*Least Frequently Used*)

Analisando o desenvolvimento do projeto podemos concluir que a maior dificuldade foi na hora de fazer o projeto do sistema e reunir todas as informações necessárias para sua construção, visto que a parte de implementação ocorreu facilmente após o projeto estar concluído.

8. Código Fonte

O código fonte completo pode ser encontrado no GitHub pelo seguinte endereço:

https://github.com/cesargraeff/simulador-cache

Abaixo encontra-se o código fonte da função responsável por fazer a simulação do memória cache que recebe como parâmetros um objeto com os dados informados pelo usuário:

```
function executa(params){
    vm.resultados = {
        linhas_arquivo: 0,
        linhas_processadas: 0,
        escritas_arquivo: 0,
        leituras_arquivo: 0,
        escritas: 0,
        leituras: 0,
        cache: {
            leituras: 0,
            escritas: 0
        },
        memoria: {
            leituras: 0,
            escritas: 0
        }
    };

    vm.cache = {};

    var linhas = params.file.split('\n');
    vm.resultados.linhas_arquivo = linhas.length;
    if(linhas[linhas.length - 1] == '')  vm.resultados.linhas_arquivo--;
```

```
linhas.forEach(function(value){
    if(value){
        // DECODIFICAÇÃO DA LINHA
        var operacao = value.substr(9,1);
        var endereco = converte_bin(value.substr(0,8),32);

        // DECODIFICAÇÃO DO ENDEREÇO
        var conjunto = params.num_linhas - params.linhas_conjunto;
        var tag = 32 - ((params.num_linhas - params.linhas_conjunto) + params.tam_linha);
        conjunto = endereco.substr(tag,conjunto);
        tag = endereco.substr(0,tag);

        if(!vm.cache[conjunto]) vm.cache[conjunto] = {};

        if(operacao == 'W'){
            vm.resultados.escritas_arquivo++;
        }else{
            vm.resultados.leituras_arquivo++;
        }

        var write = 0;
        if(operacao == 'W'){
            vm.resultados.escritas++;

            if(vm.cache[conjunto][tag]){
                vm.resultados.cache.escritas++;
                vm.cache[conjunto][tag].dirty = 1;
                vm.cache[conjunto][tag].ultima_linha = vm.resultados.linhas_processadas;
                vm.cache[conjunto][tag].acessos++;
            }else{
                write = 1;
                operacao = 'R';
            }
            // WRITE-THROUGTH
            if(params.politica_escrita == 0)
                vm.resultados.memoria.escritas++;
        }
        if(operacao == 'R'){
            if(write == 0)
                vm.resultados.leituras++;

            if(!vm.cache[conjunto][tag]){
                vm.resultados.memoria.leituras++;
                if(Object.keys(vm.cache[conjunto]).length >= Math.pow(2,params.linhas_conjunto)){
                    if(params.politica_substituicao == 2) {
                        // RANDOM
                        var tamanho = Object.keys(vm.cache[conjunto]).length;
```

```
            var indice = Math.floor((Math.random() * (tamanho - 1)));
            var i = 0;
            angular.forEach(vm.cache[conjunto],function(v,t){
                if(i == indice){
                    // WRITE-BACK
                    if(params.politica_escrita == 1
                        && vm.cache[conjunto][t].dirty == 1){
                        vm.resultados.memoria.escritas++;
                    }
                    delete vm.cache[conjunto][t];
                }
                i++;
            });
        }else{
            var rotulo,menor;
            angular.forEach(vm.cache[conjunto], function (v, t) {
                if (params.politica_substituicao == 1) {
                    // LRU
                    if(!menor){
                        menor = v.ultima_linha;
                        rotulo = t;
                    }
                    if(v.ultima_linha < menor) {
                        menor = v.ultima_linha;
                        rotulo = t;
                    }
                } else {
                    // LFU
                    if(!menor){
                        menor = v.acessos;
                        rotulo = t;
                    }
                    if(v.acessos < menor) {
                        menor = v.acessos;
                        rotulo = t;
                    }
                }
            });
            if (rotulo){
                if(params.politica_escrita == 1 && vm.cache[conjunto][rotulo].dirty == 1){
                    vm.resultados.memoria.escritas++;
                }
                delete vm.cache[conjunto][rotulo];
            }
    }
}
vm.cache[conjunto][tag] = {
    acessos: 0,
    ultima_linha: vm.resultados.linhas_processadas,
```

```
                        dirty : write
                    };
                }else{
                    vm.cache[conjunto][tag].ultima_linha = vm.resultados.linhas_processadas;
                    vm.cache[conjunto][tag].acessos++;
                    vm.resultados.cache.leituras++;
                }
            }
            vm.resultados.linhas_processadas++;
        }
    });

    if(params.politica_escrita == 1){
        angular.forEach(vm.cache,function(v,conjunto){
            angular.forEach(vm.cache[conjunto],function(v,tag){
                // WRITE-BACK
                if(vm.cache[conjunto][tag].dirty == 1){
                    vm.resultados.memoria.escritas++;
                }
            });
        });
    }
}
```