Simulação de Arquiteturas de Hardware com Memórias Não-Voláteis

Mauricio G. Palma, Emilio Francesquini, Rodolfo Azevedo

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, Nº 1251, CEP 13083-852, Cidade Universitária, Campinas/SP - Brazil

mauricio.gagliardi@students.ic.unicamp.br, {francesquini,rodolfo}@ic.unicamp.br

Abstract. *Emerging non-volatile memory technologies, known as NVMs, have great potential to replace DRAM as the main memory technology. This may open opportunities to manipulate persistent data directly, without transient copies. However, NVMs still need to overcome some limitations such as lower performance when compared to DRAM. Taking these limitations into consideration, hybrid memory architectures using DRAM for transient data and NVMs for persistent data have been proposed. However, these hardware architectures are not readily available on the market, making it difficult to assess the performance of the proposed solutions. This paper describes a simulator for multi-memory technology architectures, which enables hardware and software designers to evaluate the performance of their solutions on these new architectures. We employ this simulator to evaluate the performance impact of a complete substitution of DRAM for NVMs. We also describe a programming API designed to make use of hybrid DRAM-NVM architectures and detail new research opportunities.*

Resumo. *As novas tecnologias de memórias não voláteis, conhecidas coletivamente como NVMs, prometem rivalizar com a DRAM na disputa pela escolha da tecnologia da memória principal. As NVMs possibilitam, por exemplo, a manipulação de dados persistentes sem o uso de cópias transientes das mesmas. Apesar disso as NVMs ainda não são capazes de oferecer um desempenho superior à DRAM. Levando isso em consideração, a configuração onde se usa uma memória principal híbrida, composta da tradicional DRAM e de NVMs auxiliando na manipulação de dados persistentes, se torna uma solução interessante. Este artigo descreve um simulador que é capaz de simular um sistema onde a memória principal é composta por uma ou mais tecnologias distintas. Utilizamos o simulador para comparar a substituição total de DRAM por NVMs e mostramos que atualmente isso levaria à uma considerável perda de desempenho. Também descrevemos uma API que é capaz de fazer uso de NVM, focando no caso da memória híbrida, onde demonstramos as possibilidades que podem ser melhor exploradas.*

1. Introdução

A maneira pela qual a manipulação de dados é atualmente feita pelos programas computacionais foi fortemente influenciada pela arquitetura de hardware. Dados transientes que não necessitam existir além do tempo de execução do programa são armazenados na memória principal, também chamada de memória primária ou de trabalho. A tecnologia volátil DRAM é tipicamente utilizada para este fim. Já os dados persistentes, que necessitam ter sua existência estendida além do tempo de execução do programa,

são armazenados em uma memória secundária, também chamada de memória de armazenamento. Neste caso tecnologias não voláteis como *Hard Disk Drive* (HDD) ou *Solid State Drive* (SSD) são utilizadas. Apesar dessa divisão presente no armazenamento dos dados, os programas são comumente limitados à manipulação de dados que se encontram na memória principal. Assim, a manipulação de dados persistentes ocorre indiretamente, trabalhando-se apenas com suas cópias transientes armazenadas temporariamente na memória de trabalho.

Apesar de diversas soluções terem sido propostas para facilitar e otimizar a manipulação de dados persistentes, em sua maioria elas focam na arquitetura convencional em que os computadores são construídos. Novas tecnologias de memórias não-voláteis, conhecidas como *Non-Volatile Memories* (NVMs) [Greengard 2015], devem alterar significativamente a arquitetura atual. Essas novas memórias têm potencial para rivalizar o desempenho da DRAM e se tornarem as novas tecnologias utilizadas para a memória principal. Numa arquitetura deste tipo será possível manter os dados persistentes na memória principal e portanto manipulá-los diretamente sem a necessidade de cópias. Já há trabalhos que visionam uma memória composta apenas por NVM [Jung and Cho 2013, Narayanan and Hodson 2012] e também outros que consideram uma memória híbrida [Kannan et al. 2016]. Apesar disso, no momento, as NVMs ainda não são amplamente comercializadas. Por esta razão pesquisas relacionadas ao uso de NVMs são feitas através de protótipos, como em Whole-System Persistence (WSP) [Narayanan and Hodson 2012], ou através de simulação, como em [Jung and Cho 2013, Kannan et al. 2016].

Simulações são cotidianamente empregadas na pesquisa de arquiteturas de computadores [Yi and Lilja 2006]. Seguindo esta mesma linha, este trabalho propõe um simulador capaz de reproduzir o comportamento de uma arquitetura que utiliza NVM como memória principal. Para isso, integramos o simulador da arquitetura x86, ZSIM [Sanchez and Kozyrakis 2013], ao simulador de memórias NVMain [Poremba et al. 2015] que é capaz de simular tanto a tecnologia DRAM quanto algumas diferentes tecnologias de NVM. Mostramos também como o simulador proposto já é capaz de reproduzir comportamentos de arquiteturas com memórias de trabalho completamente não voláteis ou de arquiteturas híbridas que utilizam combinações de DRAM e NVM. Para isto apresentamos resultados de desempenho comparando DRAM com STTM, RRAM e PCM, além de um exemplo de como uma aplicação criada nos moldes tradicionais pode ser alterada para utilizar memórias NVM de maneira eficiente.

O restante deste artigo está organizado da seguinte maneira. A Seção 2 discute a nossa motivação e trabalhos relacionados. Em seguida, na Seção 3 apresentamos o desenvolvimento e as características do simulador construído. Mostramos também um exemplo de código de uma aplicação adaptada para a utilização eficiente de NVMs. Na Seção 4 descrevemos alguns resultados obtidos com o simulador, comparando DRAM com algumas tecnologias de NVMs. Por fim, a Seção 5 conclui o artigo.

2. Motivação e Contexto

As memórias secundárias, que normalmente são dispositivos de bloco, são acessadas através de chamadas ao sistema operacional (SO). Isso causa um aumento da latência na manipulação dos dados persistentes, devido à camada adicional do SO intermediando o acesso e a necessidade da serialização/desserialização dos dados [Huan et al. 2015]. Uma solução comumente aplicada para este problema é a utilização de arquivos mapeados em memória [Tevanian et al. 1987]. Logo, programas podem mapear os arquivos e

acessá-los diretamente no seu espaço de endereçamento. Esta solução, contudo, também não evita a cópia dos dados da memória persistente para a transiente, ainda que ela seja feita de maneira quase transparente. Versões mais recentes do *kernel* do Linux trazem a possibilidade de utilização de uma extensão chamada DAX, que evita que páginas de dados de arquivos mapeados sejam colocadas em um *cache* na memória de trabalho pelo SO [Wilcox 2014].

Restrições impostas pelo funcionamento do hardware direcionaram o desenvolvimento de linguagens de programação de modo a oferecer suporte eficiente à manipulação de dados transientes. Acessos aos dados persistentes são então feitos por interfaces não tão diretas quanto aquelas utilizadas por memórias transientes. Tais interfaces são tipicamente baseadas em sistemas de arquivos ou bancos de dados. Assim, os programadores são obrigados a efetuar constantemente conversões entre dados persistentes e transientes. Visando facilitar essas conversões, várias extensões às linguagens de programação foram propostas ao longo dos anos, como em Pascal [Schmidt 1977] ou mais recentemente em Java [Atkinson et al. 1996]. Todas estas soluções, no entanto, não são capazes de contornar a necessidade imposta pelo hardware de manipulações de dados apenas na memória de trabalho.

Com o intuito de facilitar o uso das novas tecnologias de NVM e oferecer uma maneira consistente para a sua utilização, várias propostas envolvendo novos modelos de programação e ambientes de execução foram feitas. Mnemosyne [Volos et al. 2011] e NV-Heaps [Coburn et al. 2011], por exemplo, utilizam conceitos de memória transacional para a solução de problemas de concorrência e atomicidade. Outros como pmem.io [Intel 2015] e Atlas [Chakrabarti et al. 2014] se concentram na utilização do mais recente suporte em hardware para a utilização correta e eficiente destas novas memórias. Todas estas propostas envolvem maneiras alternativas para a declaração de dados persistentes e transientes além de lidarem automaticamente com a persistência dos dados. Tais soluções são baseadas ou em APIs próprias ou em modificações nas próprias linguagens de programação. A princípio, todas essas soluções poderiam ser utilizadas em conjunto com o simulador proposto, entretanto, por hora, ele dá suporte apenas a uma API própria construída como um subconjunto da API já oferecida pelo Atlas.

A utilização de simulação já é uma realidade nas pesquisas de arquiteturas que utilizam NVM. Memorage [Jung and Cho 2013] e pVM [Kannan et al. 2016] por exemplo modificam o *kernel* do Linux para simular NVMs utilizando arquivos, discos simulados em memória ou nós NUMAs. Outros trabalhos fazem modificações em algum simulador existente, como em NVM Duet [Liu et al. 2014] e THyNVM [Ren et al. 2015], que modificam respectivamente os simuladores MARSSx86 [Patel et al. 2011] e gem5 [Binkert et al. 2011] para avaliarem suas propostas. Um fato negativo desse cenário é que nem sempre as modificações de um trabalho podem ser utilizadas para outros trabalhos, por realmente não estarem adaptadas a outras propostas ou simplesmente pelo fato de as modificações não terem sido publicadas. A criação de um simulador focado na exploração de soluções que utilizam NVM pode impulsionar a continuidade dessas pesquisas, de tal maneira que essas futuras pesquisas não demandem muito esforço para desenvolverem uma ferramenta para avaliarem suas soluções.

3. O Simulador

Com o objetivo de possibilitar a exploração de novas arquiteturas de hardware baseadas em NVMs e para permitir a avaliação do desempenho de aplicações quando

executadas nestas arquiteturas, neste trabalho desenvolvemos um simulador com suporte à utilização de NVMs. Tal simulador foi construído tendo como base o ZSIM [Sanchez and Kozyrakis 2013]. O ZSIM é um simulador da arquitetura x86 que visa simular uma máquina com um grande número de processadores. Por esse fato, é conhecido por ter um ótimo desempenho em velocidade de execução quando comparado a outros simuladores, principalmente aqueles que se dispõem a fazer uma simulação completa do sistema, como por exemplo o gem5. O ZSIM permite não apenas a simulação do processador, como também permite a simulação de uma hierarquia de memória até a memória principal. No nível das caches o ZSIM possui modelos detalhados com configurações, por exemplo, para o ajuste de capacidade, latência e políticas de substituição das linhas de cache entre outros. No entanto, para a memória principal o ZSIM conta apenas com algumas configurações de DRAM e não dá suporte a nenhuma tecnologia de NVM.

O NVMain é um simulador de memória principal que conta com modelos de memória DRAM e de alguns tipos de NVM, como por exemplo *Phase Change Memory* (PCM) e *Spin-Transfer Torque Memory* (STTM). Em contrapartida, a exemplo de outros simuladores de memória como o DRAMSim2, é comumente usado em conjunto com simuladores de arquitetura como o ZSIM e o gem5. A integração entre o ZSIM e o NVMain foi feita baseando-se nos resultados do projeto AXLE [AXL 2016]. Além disso, uma série de modificações adicionais, descritas a seguir, foram incorporadas ao código.

Arquiteturas híbridas Apesar de o ZSIM possuir suporte para a simulação de múltiplos controladores de memória, ele exige que todas as tecnologias de memória ligadas a estes controladores sejam idênticas. Assim, embora o ZSIM integrado ao NVMain já possa ser utilizado para a simulação de uma arquitetura somente com memórias não voláteis, ele não poderia ser utilizado para a simulação de uma arquitetura híbrida.

Para permitir que tais configurações fossem feitas, nós primeiramente alteramos o formato do arquivo de configuração do ZSIM. Além de especificar o número de controladores de memória da arquitetura simulada, este arquivo também especifica a tecnologia de memória a ser utilizada em cada um dos controladores individualmente. Neste sentido, a especificação das várias tecnologias de memória principal se torna similar àquela utilizada para a especificação das caches, com a ressalva de que todas as memórias de trabalho estão no mesmo nível hierárquico. A Listagem 1 mostra o trecho relevante do arquivo de configuração no formato original e a versão com nossas modificações.

Dependendo da configuração escolhida e quando utilizado com múltiplos controladores de memória, o ZSIM pode empregar duas estratégias para a determinação do controlador responsável por atender cada uma das requisições. O primeiro modo, que é o padrão, consiste em fazer um entrelaçamento (*interleaving*) dos endereços entre os controladores. Assim, é esperado que o número de requisições por controlador seja equilibrado. O segundo modo consiste em dividir o espaço de endereçamento em faixas de igual tamanho e atribuir cada uma das faixas a um controlador. O número de faixas é, portanto, igual ao número de controladores.

As tecnologias de NVM atualmente disponíveis indicam que as futuras arquiteturas híbridas provavelmente funcionarão com faixas de endereço para definição do controlador responsável por cada requisição à memória, de maneira bem similar ao modelo atualmente disponível no ZSIM [NVD 2015]. Indicam também que, muito provavelmente, os tamanhos dessas memórias não voláteis serão muito maiores do que as disponíveis atualmente com a tecnologia DRAM (por exemplo, o SanDisk's ULLtraDIMM tem capa-

```
mem = {
 controllers = 2;
 type = "NVMain";
 hasDRAMCache = false;
 latency = 75;
 techIni = "pcm-nvmain.config";
 envVar = "ZSIMPATH";
 outputFile = "nvmain.out";
 traceName = "trace.out";
};
```

(a)

```
mem = {
 mem_controllers = {
  DRAM = {
   type = "DDR";
   latency = 75;
  };
  NVM = {
   type = "NVMain";
   latency = 75;
   hasDRAMCache = false;
   techIni = "pcm-nvmain.config";
   outputFile = "nvmain.out";
   envVar = "ZSIMPATH";
   traceName = "nvmain.trace";
   addrRange=0x100000000-0x200000000
  };
 };
 splitAddrs = false;
 nvmallocIntegration = true;
};
```

(b)

Listagem 1. Arquivos de configuração do simulador. (a) Arquivo original utilizado pelo ZSIM. (b) Arquivo de configuração com as novas opções.

cidade de até 400GB). Para acomodar tais arquiteturas criamos um terceiro modo para a escolha do controlador responsável por cada requisição de memória no ZSIM. O usuário pode agora especificar, por meio de um arquivo de configuração, as faixas de endereço que serão de responsabilidade de cada controlador. Adicionalmente, o usuário pode especificar apenas a faixa de endereço pela qual o controlador da NVM é responsável, neste caso todas as requisições fora desta faixa serão atendidas usando o modo de entrelaçamento entre os demais controladores.

Modelo de Programação Para que as aplicações possam utilizar NVMs de maneira eficiente são necessárias algumas modificações em seu funcionamento. Por exemplo, uma aplicação tradicional que lida com persistência de dados normalmente funciona seguindo os seguintes passos: abertura de arquivo, leitura e desserialização de dados nele contidos, reconstrução em memória das estruturas de dados, processamento, serialização das estruturas de dados e escrita no arquivo. Em uma arquitetura com NVMs, não há a necessidade de serializar ou desserializar os dados, já que eles já estão na memória persistente no formato esperado pela aplicação. Também é desnecessário fazer o carregamento dos dados e, a princípio, o seu salvamento. No entanto, por conta das memórias caches do processador, algumas escritas podem acabar sendo feitas de maneira incompleta ou em ordem não cronológica para a memória principal. Em caso de execuções sem falha, isto é irrelevante. Contudo, caso haja falhas, escritas incompletas e fora de ordem podem corromper os dados. Neste trabalho não abordamos o tratamento destas falhas. Da mesma maneira que uma escrita incompleta em um arquivo pode corrompê-lo, assumimos que é responsabilidade das aplicações garantir que as alterações sejam feitas de uma maneira segura. Para isto, a exemplo do pmem.io e do Atlas, o desenvolvedor da aplicação pode utilizar barreiras de memória e instruções como `CLFLUSH`, `CLWB` e `PCOMMIT` (x86) para garantir que as escritas foram completamente feitas e alcançaram a memória principal na ordem desejada.

Apesar do salvamento e carregamento serem desnecessários, ainda é preciso que os ponteiros para as raízes das estruturas de dados da aplicação sejam reestabelecidos du-

```
1 //Aloca size bytes em NVM
2 void *pmalloc (size_t size);
3 //Libera a regiao de memoria apontada por p
4 void  pfree (void *p);
5 //Define p como a raiz dos dados. p deve apontar para uma das regioes
6 //previamente alocadas em NVM com pmalloc ou suas variacoes
7 void  pset_root(void *p);
8 //Devolve um ponteiro para a raiz dos dados definida por pset_root
9 void *pget_root();
```

Listagem 2. Protótipos de funções contidas em `nvmalloc.h`.

rante a sua inicialização. De maneira equivalente, também é necessário que a aplicação informe ao mecanismo de execução durante a sua execução sobre o ponteiro raiz para os dados da sua aplicação além das regiões de memória que devem ser persistentes. Neste trabalho definimos um subconjunto da API definida pelo Atlas que contém funções voltadas especificamente para a alocação e liberação de memória persistente e para a definição e recuperação do ponteiro raiz. Os protótipos das funções mais relevantes são mostrados na Listagem 2. Diversas outras funções presentes nas implementações mais comuns de `malloc` como, por exemplo, `realloc` e `calloc` também têm a sua versão persistente (tipicamente prefixadas com "`p`") e foram aqui omitidas por motivos de espaço.

Para simular o comportamento esperado das NVMs quando da utilização destas interfaces, nós adicionamos ao ZSIM a capacidade de salvar e recuperar o estado da memória persistente simulada ao final e início da execução. Isto é feito de maneira totalmente transparente à aplicação. À aplicação basta definir as regiões de memória persistente (usando as funções descritas) e o ponteiro para a raiz. Um exemplo de uso completo é mostrado na Listagem 3. Nele demonstramos como uma lista encadeada persistente pode ser implementada utilizando a API disponibilizada. A Linha 2, inclui o cabeçalho que contém as funções para controle de memória persistente. Em seguida, após ler um número da entrada padrão (Linha 12), faz a inclusão ou exclusão de um elemento da lista. Se o número lido for 0 é feita a exclusão do primeiro elemento da lista. Em ambos os casos, as rotinas de manipulação da lista fazem uso da função `pget_root` (linhas 15 e 21). Esta função devolve o ponteiro que foi definido com a chamada `pset_root` e funciona mesmo entre em execuções distintas da mesma aplicação. Quando um nó é adicionado ao início da lista, a função `pset_root` é usada para definí-lo como sendo a cabeça da lista enquanto que quando o nó é removido é usada para definir a cabeça como sendo o elemento seguinte da lista (Linha 23). A alocação persistente de novos nós é feita pela função `pmalloc` (Linha 19) e a liberação da memória pela função `pfree` (Linha 17).

É interessante notar que, neste exemplo, tanto memória volátil quanto memória não volátil são utilizadas da mesma maneira, o que permite que dados voláteis possam ser facilmente copiados entre as duas memórias. Por exemplo, o valor da variável `v` que é volátil é copiado para a variável `val` (persistente) presente em cada nó da lista encadeada diretamente sem a necessidade de conversões ou chamadas de API (Linha 20). Como regra geral todas as alocações que não forem feitas através das novas interfaces (pilha de execução, variáveis temporárias e alocações dinâmicas) são mantidas na DRAM enquanto aquelas usando a nova interface são mantidas na NVM.

4. Avaliação Experimental

Utilizamos o simulador para analisar a substituição total da DRAM por NVMs, assim como para avaliar o caso de uso de memórias híbridas. A memória DRAM foi simulada

```c
#include <stdio.h>
#include "nvmalloc.h"

struct ll_node {
    int val;
    struct ll_node *next;
};

int main () {
    int v;
    printf ("Digite um numero (0 remove o 1o. elemento da lista):\n");
    fscanf (stdin, "%d", &v);
    struct ll_node *curr = NULL;
    if (v == 0) {//Remove o primeiro elemento da lista
        struct ll_node *head = pget_root();
        if (head) curr = head->next;
        pfree (head);
    } else { //Adiciona um elemento no inicio da lista
        curr = pmalloc (sizeof (struct ll_node));
        curr->val = v;
        curr->next = pget_root ();
    }
    pset_root (curr);
    printf("Lista: ");
    while (curr) {
        printf("%d ", curr->val);
        curr = curr->next;
    }
    printf("\n");
    return 0;
}
```

Listagem 3. Adição e remoção de nós a uma lista encadeada persistente.

com parâmetros que equivalem aos de uma DDR3 de 1600 MHz. Foram simuladas três tipos de NVMs, uma PCM que equivale à descrita em [Choi et al. 2012], uma RRAM que equivale à descrita em [Kawahara et al. 2012] e uma STTM que equivale à descrita em [STT 2007]. Em todos os experimentos, o processador simulado continha caches L1, L2 e L3 de 64K (32KB dados + 32KB instruções), 256KB e 12MB respectivamente, e a frequência do mesmo foi estipulada em 2,27 GHz. Essa configuração reproduz as características do processador Intel Xeon L5640.

As execuções foram feitas em um cluster de computadores gerenciados pelo Condor [Litzkow et al. 1988]. Todos os nós deste cluster executavam Linux 64 bits (Ubuntu 14.04) com o *kernel* na versão 3.13. A versão do PIN [Luk et al. 2005], utilizado pelo ZSIM, foi a 2.14, e a versão do NVMain utilizada foi a mais recente obtida do repositório do projeto no dia 1 de julho de 2016. Pelo fato do ZSIM não ser um simulador funcional, podem ocorrer pequenas variações no tempo de execução estimado pelo simulador para um dado benchmark. Essas variações foram, entretanto, limitadas a 1% do valor observado em todos os benchmarks para todas as execuções.

4.1. DRAM vs. NVMs

A Tabela 1 mostra a perda de desempenho (*Slowdown*) obtida ao substituir totalmente a DRAM por uma NVM na execução do benchmark STREAM [McCalpin 1995], que é um benchmark de acesso irregular à memória que visa verificar a taxa de transferência da

	STTM	RRAM	PCM
Slowdown	1,5	1,5	5,3
Slowdown TMA	1,7	1,8	6,8

Tabela 1. Slowdown executando o benchmark STREAM com NVMs em relação à execução com DRAM

memória principal. Neste caso, a utilização total de qualquer uma das três NVMs mostra uma perda de desempenho de ao menos 50%, enquanto o tempo médio de acesso (TMA) à memória prncipal mostra perdas de ao menos 70%. É, contudo, esperado que um programa como o STREAM tenha seu desempenho vinculado diretamente ao desempenho da memória utilizada já que estimar o desempenho da memória é justamente a sua função.

Com intuito de investigar programas que tenham uma menor dependência do desempenho da memória principal, avaliamos também a utilização total de NVMs executando alguns benchmarks do SPEC2006. A Figura 1 mostra, em ciclos do processador, o tempo médio de acesso (TMA) utilizando as entradas de referência do SPEC2006 para cada uma das NVMs analizadas. Apesar de não possuir o maior valor de TMA entre os benchmarks, o benchmark `soplex` sofreu os maiores *slowdowns*, de 1,4, 1,4 e 2,8 quando executado em STTM, RRAM e PCM respectivemente. A Figura 2 mostra o *Slowdown*, total e também do TMA, na utilização das NVMs em relação à utilização de DRAM. Note que o maior *Slowdown* no TMA (`hmmer`) não está presente no benchmark onde houve o maior *Slowdown* de desempenho (`soplex`). Outro ponto interessante é que metade dos benchmarks tiveram *Slowdowns* de no máximo 1,1, valor distante do pior caso mostrado na Tabela 1, de 5,3. Isso mostra que uma aplicação bem comportada com relação às caches é capaz de mascarar os maiores tempos de acesso à memória principal.

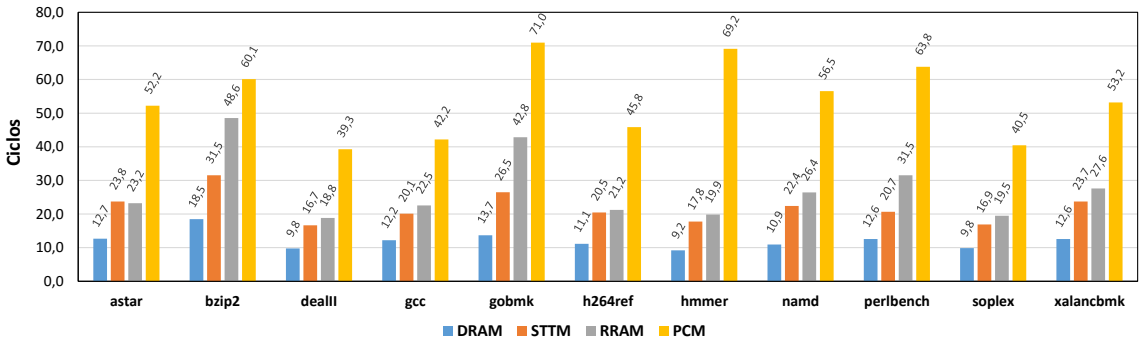

Figura 1. Latência média das requisições do SPEC2006

Os resultados da substituição total de DRAM por NVMs, apesar de mostrarem que tal configuração causa uma perda de desempenho, mostra também que essa perda pode ter grande variação dependendo da aplicação em execução, podendo inclusive ser irrelevante.

4.2. Arquitetura Híbrida

Em contrapartida à substituição total, podemos utilizar DRAM e NVMs em conjunto na memória principal. A avaliação do desempenho de arquiteturas híbridas quando comparadas às arquiteturas tradicionais é fortemente dependente da aplicação em questão. Por exemplo, se a aplicação sendo testada tiver uma pequena quantidade de dados persistidos mas por outro lado fizer uma quantidade alta de manipulações com esses dados, ela

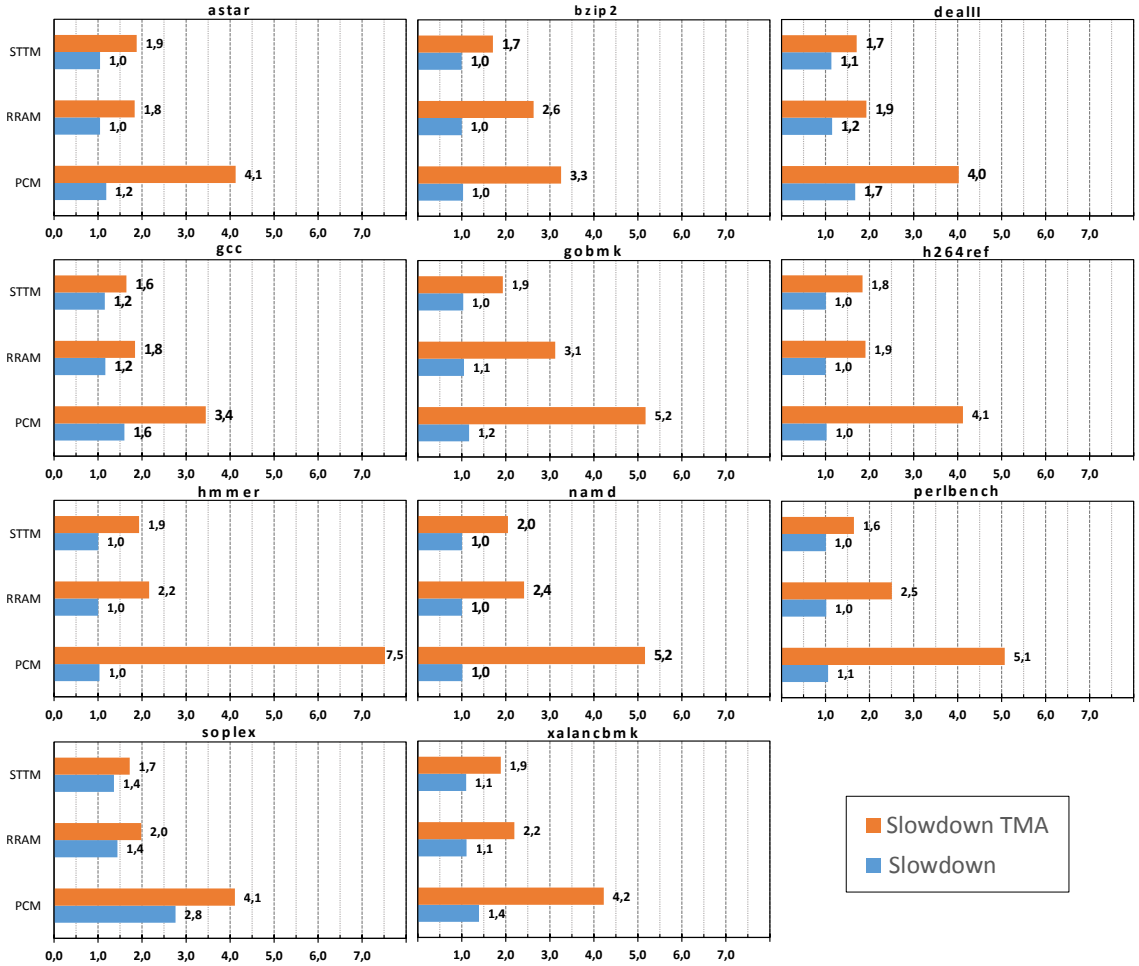

Figura 2. *Slowdown* do tempo de execução e do TMA em relação à DRAM

provavelmente terá um desempenho melhor em uma arquitetura tradicional. Esse é o caso típico das aplicações presentes no SPEC2006 cujos resultados foram apresentados na seção anterior. O inverso também é verdadeiro, ou seja, uma aplicação com muitos dados persistentes e poucas manipulações tende a ter um resultado de desempenho melhor em arquiteturas híbridas. Em outras palavras, se a relação entre o número de operações por byte lido/escrito de forma persistente for alta, arquiteturas tradicionais baseadas em DRAM e HD/SSD levam vantagem. Contudo, conforme a relação entre o número de operações por byte lido/escrito de dados persistentes diminui, arquiteturas híbridas passam a ter um desempenho mais competitivo. Está além do escopo deste trabalho fazer uma comparação de desempenho envolvendo todo o espectro de aplicações com variadas relações entre o número de operações e bytes lidos/escritos. Por esta razão descrevemos o uso de uma aplicação que varia o seu comportamento conforme a entrada.

A aplicação em questão, chamada *Escadas*, funciona tomando como entrada duas palavras e um dicionário. Para cada par de palavras contidas no dicionário é calculada a distância de Levenshtein entre elas. Essas informações são utilizadas na montagem de um grafo, mantido em memória, onde cada vértice representa uma palavra. Existe uma aresta que conecta dois vértices quando a distância entre as palavras que eles representam é inferior a 1. A saída do programa consiste no caminho mínimo, caso exista, entre as duas palavras dadas como entrada. Por exemplo, com esta aplicação é possível transformar

"girafa" em "zebra": *girafa* → *girara* → *gerara* → *zerara* → *zerar* → *zera* → *zebra*.

A maior parte do tempo de execução é utilizado durante o cálculo das distâncias entre as palavras. Faz sentido, portanto, que este grafo seja salvo de maneira persistente após ser calculado. A primeira execução é limitada pela velocidade do processamento e se comporta muito bem com relação ao cache. Logo, o uso de memórias híbridas (de qualquer uma das tecnologias testadas) não causa nenhuma variação significativa ($<$0,1%) no tempo de execução.

Por outro lado, nas execuções seguintes onde o processamento já foi feito o desempenho é limitado pelo acesso aos dados. Nós verificamos experimentalmente que, conforme o tamanho do dicionário cresce, nesta aplicação a razão entre o número de acessos à memória DRAM em relação à NVM (que contém o grafo) converge para $\sim$1,4%. Usando um dicionário de 2.732.533 palavras, com a razão entre o número de acessos DRAM/NVM em 1,43% temos um aumento médio no tempo de execução de 2,8X para PCM enquanto que para RRAM e STTM observamos um leve *speedup* de $\sim$2% em relação ao tempo de execução baseado no carregamento de um arquivo (arquivo binário mapeado em memória) a partir de um SSD. Por começar com o dicionário em memória e o grafo já calculado, as NVMs passam a ter uma vantagem sobre a DRAM+Disco/SSD que precisa ler o grafo serializado do disco e remontar as estruturas em memória.

5. Conclusão e trabalhos futuros

Neste trabalho, apresentamos uma ferramenta, baseada no simulador ZSIM, para simular e avaliar arquiteturas que utilizem NVMs, de maneira total ou híbrida, na memória principal. Também mostramos uma nova API que consegue aproveitar os benefícios de tal configuração. Utilizamos o simulador para verificar o aumento no tempo médio de acesso à memória quando a DRAM é totalmente substituída por NVMs, e também verificamos o *slowdown* causado por esse aumento. Mostramos que o *slowdown* pode ser irrelevante (1,0) ou muito significativo (valor máximo encontrado de 5,3). Com o uso do simulador também mostramos um caso de uso de memória híbrida, onde fomos capazes de encontrar um limiar onde essa configuração se torna vantajosa.

Ainda há funcionalidades a serem adicionadas ao simulador, como permitir que haja NVMs compondo as memórias cache, como explorado por [Zhao et al. 2013]. Pretendemos continuar aprimorando o simulador e utilizá-lo para desenvolver soluções que façam uso de NVMs. Como visto, não necessariamente o aumento no tempo médio de acesso à memória tem grande impacto na execução dos programas. Com isso, pretendemos explorar com o simulador tanto soluções de substituição completa da DRAM por NVMs como com memórias hibridas. À medida que as tecnologias de memória forem evoluindo, configurações inovadoras podem surgir o que destaca a importância do contínuo aprimoramento deste simulador.

6. Agradecimentos

Este trabalho contou com financiamento da FAPESP, CAPES e CNPq.

Referências

[STT 2007] (2007). *256K x 16 MRAM Memory*. Everspin Technologies. Rev. 6.

[NVD 2015] (2015). LIBNVDIMM: Non-Volatile Devices v13. https://www.kernel.org/doc/Documentation/nvdimm/nvdimm.txt.

[AXL 2016] (2016). AXLE project: Advanced analytics for extremely large european databases. https://axleproject.eu/.

[Atkinson et al. 1996] Atkinson, M. P., Daynes, L., Jordan, M. J., Printezis, T., and Spence, S. (1996). An orthogonally persistent java. *ACM Sigmod Record*, 25(4):68–75.

[Binkert et al. 2011] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

[Chakrabarti et al. 2014] Chakrabarti, D. R., Boehm, H.-J., and Bhandari, K. (2014). Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA. ACM.

[Choi et al. 2012] Choi, Y., Song, I., Park, M.-H., Chung, H., Chang, S., Cho, B., Kim, J., Oh, Y., Kwon, D., Sunwoo, J., et al. (2012). A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 46–48. IEEE.

[Coburn et al. 2011] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. (2011). NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA. ACM.

[Greengard 2015] Greengard, S. (2015). Better memory. *Communications of the ACM*, 59(1):23–25.

[Huan et al. 2015] Huan, J., Badam, A., Qureshi, M. K., and Schwann, K. (2015). Unified Address Translation for Memory-Mapped SSDs with FlashMap. In *International Symposium on Computer Architecture (ISCA'15)*.

[Intel 2015] Intel (2015). The NVM library. http://pmem.io/.

[Jung and Cho 2013] Jung, J.-Y. and Cho, S. (2013). Memorage: Emerging persistent ram based malleable main memory and storage architecture. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 115–126. ACM.

[Kannan et al. 2016] Kannan, S., Gavrilovska, A., and Schwan, K. (2016). pvm: persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 13. ACM.

[Kawahara et al. 2012] Kawahara, A., Azuma, R., Ikeda, Y., Kawai, K., Katoh, Y., Tanabe, K., Nakamura, T., Sumimoto, Y., Yamada, N., Nakai, N., Sakamoto, S., Hayakawa, Y., Tsuji, K., Yoneda, S., Himeno, A., i. Origasa, K., Shimakawa, K., Takagi, T., Mikawa, T., and Aono, K. (2012). An 8mb multi-layered cross-point reram macro with 443mb/s write throughput. In *2012 IEEE International Solid-State Circuits Conference*, pages 432–434.

[Litzkow et al. 1988] Litzkow, M. J., Livny, M., and Mutka, M. W. (1988). Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111. IEEE.

[Liu et al. 2014] Liu, R.-S., Shen, D.-Y., Yang, C.-L., Yu, S.-C., and Wang, C.-Y. M. (2014). Nvm duet: Unified working memory and persistent store architecture. In *ACM SIGPLAN Notices*, volume 49, pages 455–470. ACM.

[Luk et al. 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200.

[McCalpin 1995] McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25.

[Narayanan and Hodson 2012] Narayanan, D. and Hodson, O. (2012). Whole-system persistence. *ACM SIGARCH Computer Architecture News*, 40(1):401–410.

[Patel et al. 2011] Patel, A., Afram, F., and Ghose, K. (2011). Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1st International Qemu Users' Forum*, pages 29–30.

[Poremba et al. 2015] Poremba, M., Zhang, T., and Xie, Y. (2015). Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143.

[Ren et al. 2015] Ren, J., Zhao, J., Khan, S., Choi, J., Wu, Y., and Mutlu, O. (2015). Thynvm: enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 672–685. ACM.

[Sanchez and Kozyrakis 2013] Sanchez, D. and Kozyrakis, C. (2013). Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *SIGARCH Comput. Archit. News*, 41(3):475–486.

[Schmidt 1977] Schmidt, J. W. (1977). Some high level language constructs for data of type relation. *ACM Transactions on Database Systems (TODS)*, 2(3):247–261.

[Tevanian et al. 1987] Tevanian, A., Rashid, R. F., Young, M., Golub, D. B., Thompson, M. R., Bolosky, W. J., and Sanzi, R. (1987). A unix interface for shared memory and memory mapped files under mach. In *USENIX Summer*, pages 53–68.

[Volos et al. 2011] Volos, H., Tack, A. J., and Swift, M. M. (2011). Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA. ACM.

[Wilcox 2014] Wilcox, M. (2014). DAX: Page cache bypass for filesystems on memory storage. Linux Kernel Mailing List. https://lwn.net/Articles/618064/.

[Yi and Lilja 2006] Yi, J. and Lilja, D. (2006). Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. *Computers, IEEE Transactions on*, 55(3):268–280.

[Zhao et al. 2013] Zhao, J., Li, S., Yoon, D. H., Xie, Y., and Jouppi, N. P. (2013). Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432. ACM.