UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FRANCIS BIRCK MOREIRA

Desenvolvimento de Memórias Scratchpad para Arquiteturas Multi-core

Trabalho de Graduação.

Prof. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, Junho de 2011

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

CPU	Central Processing Unit
MMU	Memory Management Unit
SPM	Scratchpad Memory
L1/L2	Level 1/Level 2 cache memory
API	Application Programming Interface
SPARC	Scalable Processor Architecture
RAM	Random Access Memory
IC	Instruction Cache
DC	Data Cache
IDS	Instruction-Data Splitter
TLB	Translation Look-Aside Buffer
MMU	Memory Management Unit

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Durante a execução de programas paralelos em arquiteturas com múltiplos núcleos, diversas vezes torna-se necessária a manipulação de uma quantidade razoável de dados compartilhados entre as múltiplas *threads* paralelas, as quais podem estar em diferentes núcleos de processamento. Portanto, dados os problemas de coerência de memória, torna-se interessante o uso de uma memória única para dados compartilhados entre os núcleos, especialmente se o programador puder controlar quais dados vão para essa memória. Para isso, é necessário o uso de uma memória não transparente, ou seja, uma memória da qual o processador tem conhecimento: o modelo clássico de uma memória *scratchpad* (SPM).

Sendo o *scratchpad* um tipo de memória bastante usado em processadores embarcados, a ideia já foi atacada de várias formas. Geralmente, tais soluções implicam nos seguintes problemas: falta de retrocompatibilidade, o que leva à recompilação do código para tamanhos diferente de *scratchpad*; *hardware* adicional, além da interconexão entre a memória *scratchpad* e o processador; e complexidade para o programador, uma vez que a responsabilidade na escolha do conteúdo do *scratchpad* ficará a seu cargo.

Este trabalho apresenta o desenvolvimento de uma implementação de uma memória *scratchpad*. Fora utilizado o simulador Simics, modelando uma arquitetura *multi-core* baseada em núcleos UltraSPARC II. O objetivo final do trabalho é alcançar ganhos de desempenho, demonstrando maneira diferente de implementação e uso de uma memória *scratchpad*, com seus benefícios e desvantagens, e avaliar o desempenho obtido.

Durante a fase de avaliação foram editados dispositivos de hierarquia da memória presente no Simics, a fim de simular a presença da memória *scratchpad* na arquitetura, considerado restrições como latência e armazenamento da memória *scratchpad*. Para acessar a memória *scratchpad* no nível do programador, foi desenvolvida uma biblioteca que, juntamente com uma função *callback* do simulador, implementa funções para o programador manipular a *scratchpad*.

Como resultados são mostrados ganhos de até 45% na execução da carga de trabalho avaliada. Sendo que para a maioria das aplicações obteve-se ganhos de desempenho de, na média, 26%, dado que a memória *scratchpad* foi bem utilizada. Tornou-se evidente durante os experimentos um dos principais problemas do uso do scratchpad, que é a escolha de seu conteúdo. Tal problema é solucionado através do uso de compiladores e ferramentas de *profiling*.

Palavras Chave: Memórias *scratchpad*, Arquiteturas Paralelas, Ambiente de Simulação Simics.

1 INTRODUÇÃO

Arquiteturas *multi-core* são o foco do estado da arte no desenvolvimento das novas arquiteturas de processadores. Nestas arquiteturas, geralmente encontra-se o problema de compartilhamento e validação das memórias privadas dos diferentes núcleos. A memória *scratchpad* (Scratchpad Memory – SPM) [1] é uma memória especializada, geralmente usada em sistemas embarcados, da qual o processador e o sistema operacional tem conhecimento. Nas arquiteturas *multi-core*, o *scratchpad* pode ser usado como uma memória compartilhada de rápido acesso disponível a vários núcleos, oferecendo possibilidades de ganho com uma seleção de dados inteligente.

Muitas pesquisas com o *scratchpad* ganham principalmente em termos de área e energia [2]. Isto deve-se ao fato de a maioria das implementações de memórias *scratchpads* não utilizarem *tags* e comparadores, comuns em memórias *cache*. Tais características tornam o uso do *scratchpad* interessante em sistemas embarcados, onde o consumo de potência e área do *chip* são muitas vezes mais importantes que o desempenho.

O objetivo deste trabalho é avaliar o desempenho de sistemas *multi-core* com o uso de memórias *scratchpad*. Para isto, desenvolveu-se uma simulação de memória *scratchpad* em um ambiente *multi-core*, usando esta memória para guardar dados compartilhados frequentemente acessados pelos múltiplos núcleos. O *scratchpad* foi implementado no simulador Simics [3], integrando-se como um componente da hierarquia de memória para ambientes multi-core. Como forma de integração, foi proposto um mecanismo de funcionamento para o *scratchpad* baseado em memória virtual, de maneira a diminuir problemas de retrocompatibilidade com o *software* e *hardware* existentes.

Em uma modelagem inicial, com modelos de tempo mais flexíveis, é previsto um ganho de desempenho, devido à falta de detalhamento em relação aos possíveis *overheads* gerados devido à chamada de sistema (necessária para a manipulação da memória virtual), e quanto o mecanismo de *Direct Memory Access*(DMA) restringiria tal *overhead*. Com a adição de tal chamada de sistema, uma estrutura de DMA, e uma alocação estática, gerando chamadas de sistema apenas no início da execução do aplicativo, é possível obter uma precisão próxima ao mundo físico, com ganhos em relação ao modelo clássico com *caches*.

O ganho de desempenho mostra-se possível devido à garantia de obtenção dos dados alocados no *scratchpad*, o que evita faltas de dados nas *caches* de primeiro nível privadas e na *cache* de nível 2 compartilhada, a qual seria alvo de grande parte destas faltas, devido a ser o mecanismo de compartilhamento mais rápido entre os núcleos. Os casos mais explícitos são de programas com grandes estruturas de dados maiores que a

cache de nível 2, a qual, em um laço que passe por tal estrutura, acaba sendo ocupada inteiramente por partes desta estrutura, mesmo que cada endereço seja acessado uma única vez. Com o uso da memória *scratchpad* para armazenar partes destas estruturas, é possível evitar o uso inadequado da hierarquia de *cache* normal, tornando-a mais eficiente.

Este trabalho está organizado da seguinte maneira. O capítulo 2 menciona os trabalhos relacionados a área. O capítulo 3 contém a descrição de como fora implementada a simulação da presença do *scratchpad* na arquitetura alvo. O capítulo 4 demonstra a arquitetura e o simulador para permitir a criação de uma biblioteca usada na adição de blocos de memória no *scratchpad* pelos aplicativos dentro da arquitetura simulada. O capítulo 5 apresenta um modelo mais realístico de como a memória *scratchpad* poderia ser implementada em *hardware*. O capítulo 6 contém a descrição da arquitetura simulada e mostra os experimentos e resultados obtidos. O capítulo 7 analisa os resultados obtidos e conclui o trabalho.

2 TRABALHOS RELACIONADOS

A base da idéia de utilização de uma memória *scratchpad* em um ambiente multi-core vem de Yanamandra et al.[4], o qual utiliza uma memória *scratchpad* para cálculos envolvendo matrizes esparsas. O artigo demonstra a comparação de eficiência substituindo partes de ambos níveis da hierarquia cache por memória *scratchpad,* obtendo ganhos tanto em desempenho quanto em energia. Na descrição do método de simulação da memória *scratchpad* é citado o uso do simulador Simics, as latências e as modificações de tamanhos nas *caches*, mas não há explicação alguma de como realmente a memória fora simulada e de como era feito o acesso a ela.

Um dos primeiros modelos de utilização de memória *scratchpad* pode ser observado no trabalho de Banakar et al. [5], onde a alternativa de uso da memória *scratchpad* é comparada ao uso da memória *cache* para sistemas embarcados. Os quesitos mais importantes foram área e energia, e demonstrou-se superioridade relevante da memória *scratchpad*. A programação dos aplicativos para uso, porém, precisa ser feita em nível baixo e o programador necessita gerenciar diretamente a memória *scratchpad*.

A partir do artigo de Poletti et al.[6] foram estudados modos eficazes de implementar a arquitetura da memória *scratchpad* em um ambiente multi-core, através do uso de um mecanismo de *Direct Memory Access,* o qual permite alocação eficiente da memória *scratchpad* em tempo de execução. Neste texto, fora descrita uma interface para o uso da memória *scratchpad,* ajudando na programação envolvendo o mesmo. Porém, o endereçamento da memória ainda dependia do tamanho da memória *scratchpad* e era feito de modo direto, o que implicava em dificuldade para usar funções e bibliotecas já prontas para dados armazenados na memória *scratchpad* usada.

Nguyen et al.[7], é demonstrada uma técnica para resolução do problema de retrocompatibilidade ao criar-se um instalador de *software* customizado, o qual decide a alocação da memória *scratchpad* exatamente antes da primeira execução do programa. Obtêm-se o tamanho da memória *scratchpad* e o instalador modifica o executável de acordo com a alocação escolhida. Esta técnica foi usada para sistemas embarcados, e o uso de um instalador é característico deste tipo de sistema, onde não existe a necessidade de modificação de *software*. O escopo do trabalho apresentado aqui é outro, onde a necessidade de alocação dinâmica de memória e recompilações tornam-se interessantes para uso genérico do poder de computação utilizado.

A pesquisa e o uso de *caches* não deveriam ser restritos apenas aos modelos clássicos. Novas técnicas, como as demonstradas em [8], estão aumentando o desempenho das hierarquias de memória para uso de arquiteturas heterogêneas, as quais representam a vanguarda da computação de alto desempenho. Tais técnicas envolvem

caches gerenciadas por *software*, uma opção semelhante à ideia da memória *scratchpad*, porém, com o objetivo de facilitar o papel do programador quando usando tais arquiteturas. No futuro, seria interessante pesquisar se tais técnicas poderiam ser otimizadas com o uso de uma memória *scratchpad,* ou se uma memória *scratchpad* poderia ser gerenciada mais facilmente com tais técnicas.

Portanto, o diferencial deste trabalho é o objetivo de obter-se uma memória *scratchpad* de tamanho grande, afim de oferecer espaço de memória com acesso eficiente e garantido para vários processos. Ao mesmo tempo, procura-se obter uma implementação que permita retrocompatibilidade e facilidade de programação, características indispensáveis para a indústria de *software*.

3 SIMULAÇÃO DE ARQUITETURAS NO SIMICS

Neste capítulo introduz-se o leitor ao simulador Simics[4], suas abstrações de hierarquia de memória e os conceitos destas que foram usados para simular uma memória *scratchpad*. O objetivo é prover noções sobre o ambiente de simulação usado e, juntamente com o capítulo 4, o método utilizado para modelar uma memória *scratchpad.*

A simulação de arquiteturas é uma importante tecnologia a favor da pesquisa e desenvolvimento de novas idéias na área. Um dos problemas da simulação de arquiteturas é obter alto desempenho para prover um retorno em tempo hábil, tentando manter com fidelidade às características físicas de cada arquitetura. Para isto, o Simics é usado mundialmente devido à sua capacidade de simular sistemas completos, executando sistemas operacionais sem modificações, possuindo também detalhes da arquitetura alvo que podem ser analisados e algumas vezes modificados.

O Simics simula desde a hierarquia de memória até redes específicas entre vários nodos com arquiteturas diferentes, sendo aconselhável usá-lo focando-se em um conjunto restrito de dispositivos e métricas a serem observados, tais como, por exemplo, quantas unidades de memória terão suas transações observadas. Tal restrição visa obter um desempenho satisfatório na execução da simulação, ou seja, uma resposta em tempo hábil.

Tal propriedade do Simics de simular apenas o necessário permite que ele mantenha interatividade com o usuário de forma que este possa usar a arquitetura simulada, contanto que não se exceda na quantidade de características e objetos simulados. Isto deve-se ao fato da simulação ser baseada em eventos, ao invés de ciclos. Cada dispositivo é um objeto, e ele simula apenas os objetos conectados à configuração principal e ao processador.

Normalmente, o Simics nem mesmo possui memória *cache*, possuindo um módulo que lhe permite simular a memória *cache* detalhadamente ao ser conectado ao sistema. A maioria dos traços detalhados está desconectada, e ao iniciar o simulador é necessário passar a opção *-stall* pra ele contar penalidades e atrasos durante a execução.

Outra vantagem do Simics é que, ao fornecer tais módulos relacionados à hierarquia de memória, tal como os módulos *id-splitter* e *g-cache*, o pacote do simulador também fornece guias[9] para o usuário escrever seus próprios módulos e editar os já existentes, fornecendo liberdade para criação de novos módulos e testes da execução da arquitetura com tais módulos.

A modelagem de uma memória *scratchpad* pode ser feita através de vários mecanismos dentro do ambiente de simulação do Simics. Após a avaliação de três métodos, foi escolhido modificar módulos da hierarquia de memória que pudessem

armazenar dados para análise posterior de forma eficiente.

3.1 Modelagem de Scratchpad Através de *Triggers*

O Simics permite, através da definição de gatilhos e chamadas de funções relacionadas a estes, executar scripts e mudar o estado do simulador durante a execução do mesmo. Isto permite simular um *scratchpad* simplesmente mantendo uma estrutura de dados que armazene os endereços escolhidos pelo programador. Quando alguma posição de memória contida nesta estrutura for acessada, ocorre um acerto no *scratchpad*.

Porém, a simulação com *scripts* não funciona na prática. Não há garantias na ordem de execução do *script* e da simulação, devido à estrutura de fila das instruções executadas na arquitetura simulada, o que faz com que a precisão do modelo simulado por tal implementação de *scripts* seja precária. Além disso, o desempenho da simulação rodando com tantos *scripts* e pausas para tais funções sofre perdas consideráveis.

3.2 Modelagem de Scratchpad Através de *Memory Spaces*

O objeto *memory space,* conforme usado no Simics, mapeia uma porção de endereços definidos pelo usuário da memória para o próprio objeto e para seu campo *timing_model.* Assim, seria possível simular um scratchpad ao mapear os endereços desejados da memória principal para um objeto memory space, e com um dispositivo que retorne apenas um número fixo de ciclos como seu *timing_model.*

O problema de tal solução se resume ao fato de que não se sabe em que área da memória principal nosso programa está alocado para obter as páginas do mesmo. É arriscado alocar qualquer espaço da memória pois não se sabe o que o sistema operacional está usando, e no que está se interferindo, inclusive correndo-se o risco de gerar um *crash* do sistema operacional.

3.3 Módulos *Id-splitter, G-cache* e a Hierarquia de Memória

A hierarquia de memória do Simics é composta por 3 tipos de componentes: *trans-staller*, *g-cache* e *id-splitter*(IDS). Como o sistema possui uma memória principal, para adicionar a hierarquia de *cache*, é preciso fazer algo um pouco fora do convencional: ao invés de conectar a hierarquia de memória *cache* com os níveis mais baixos no processador, conectamos ela na memória principal. O módulo *id-splitter* é conectado para observar as transações de memória, e divide estas em instruções ou acessos a dados.

É conectado então um módulo *g-cache* modelado para funcionar como uma *instruction cache* (IC) para receber as instruções do módulo *id-splitter*, e um módulo *g-cache* modelado para funcionar como uma *data cache* (DC) para receber os acessos a dados do módulo *id-splitter*. O IDS retornará a latência para o sistema, sendo que todos conectados a ele (neste exemplo, IC e DC) receberão latências de quem estiver conectados neles, e passarão suas latências acumuladas para o IDS. A Figura 3.1 exemplifica uma hierarquia de memória para um núcleo.

Todas estas conexões são feitas através do campo *timing model* de cada dispositivo,

portanto também da memória principal, cujo *timing model* é o IDS no caso de apenas um núcleo. Assim é possível conectar uma *cache* L2 às *caches* modeladas em IC e DC, gerando uma hierarquia que retorne a penalidade ao chamar sua função de operação e, caso necessário, a função de operação do próximo nível.

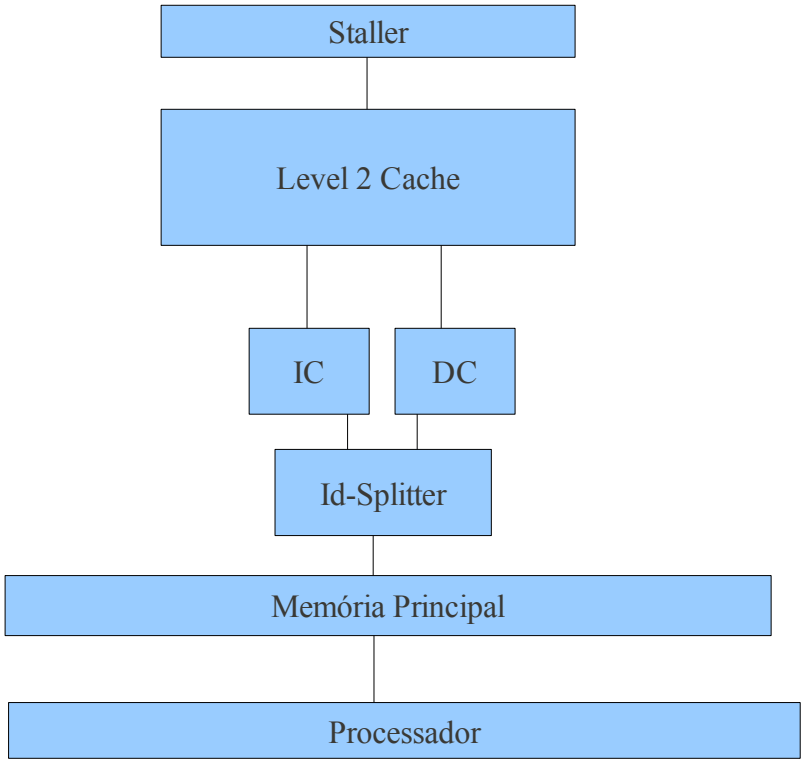

Figura 3.1 - Hierarquia da Memória com Abstrações do Simics.

Naturalmente, deve-se conectar a *cache L2* na memória principal, à qual estamos conectados pelo *id-splitter*. Porém, não é possível simular um *loop* nestas conexões, e caso o dado não esteja na *cache L2*, devemos retornar a latência da memória principal. Este é o papel do módulo *trans-staller,* o qual retorna uma certa latência especificada pelo usuário para o dispositivo ao qual ele está conectado.

Portanto, caso ocorra uma falta de dados na *cache L2* e o simulador espere resposta do próximo nível, receberá o valor especificado pelo usuário para o *trans-staller*. Este valor representa o número de ciclos que o processador irá atrasar a execução, portanto simulando um acesso a memória principal. O valor justo para tal latência deve levar em conta a quantidade de memória simulada, através de ferramentas como CACTI[10].

3.4 Modelagem de *Scratchpad* Usando o Módulo *Id-splitter*

O módulo *id-splitter* fora editado para simular a presença de uma memória *scratchpad*, o qual receberia as transações de uma *Memory Management Unit*(MMU) estendida que pode armazenar endereços especiais para redirecionar as transações para a SPM. Como o Simics não simula um módulo de MMU separadamente, e portanto não fornece ao usuário um modo simples de editar a MMU, preferiu-se simular a SPM com

as seguintes modificações no módulo *id-splitter*.

3.4.1 Código de Adição de Novos Campos

O módulo *id-splitter* possui normalmente os seguintes campos: *ibranch*, *dbranch*. Tais campos são interfaces para o módulo *trans-staller*, no qual uma *g-cache* se conecta. Da forma como o módulo *id-splitter* fora implementado, é necessário o módulo *trans-staller* para fazer a interface entre a *g-cache* e o módulo *id-splitter*.

Os novos campos adicionados foram:

- *integer_t *spm_base*: lista que possui os endereços base de cada bloco de memória na SPM.
- *integer_t *spm_offset*: lista que possui o tamanho de cada bloco de memória que começa no endereço correspondente em *spm_base*.
- *integer_t spm_size*: tamanho da SPM. Caso tente se adicionar blocos com tamanho maior ou a soma dos tamanhos dos blocos seja maior, o bloco não será adicionado.
- *int block_count*: número de blocos de memória na SPM.
- *integer_t spm_penalty*: penalidade que o *id-splitter* retornará caso o acesso seja na SPM.
- *integer_t spm_write*: número de acessos de escrita.
- *integer_t spm_read*: número de acessos de leitura.

Com tais campos, é possível observar se o endereço de memória da transação passada está dentro de algum destes blocos, e portanto, retornar a latência da SPM ao invés de procurar pela latência do próximo nível da hierarquia.

3.4.2 Código de Operação do Módulo e suas Modificações

Cada módulo de memória do Simics tem uma função fundamental chamada *operate*. Tal função é chamada para a execução mais fundamental daquele módulo, assemelhando-se ao seu comportamento em um sistema normal. O código da função *operate* modificado para o módulo *id-splitter* está ilustrado na Figura 3.2.

```
static cycles_t
id_splitter_operate(conf_object_t *mem_hier, conf_object_t *space,
     map_list_t *map, generic_transaction_t *mem_trans){

 id_splitter_t *ids = (id_splitter_t *)mem_hier;
 if (SIM_mem_op_is_data(mem_trans)) {
  if (ids->dbranch){
  /*Código inserido*/
   int spm_index;
   for(spm_index=0;spm_index < ids->block_count;spm_index++{
   /*percorre lista de blocos*/
    if (((integer_t)mem_trans->logical_address>=
     ids->spm_base[spm_index]) &&
     (mem_trans->logical_address <
     ((integer_t)ids->spm_base[spm_index] + (integer_t)ids->spm_offset[spm_index]))){
      if (SIM_mem_op_is_read(mem_trans))
       ids->spm_read++; /*inc. hits de leitura na SPM */
      else  ids->spm_write++;/*inc. hits de escrita na SPM*/
       return ids->spm_penalty;/*retorna a penalidade definida*/
    }
   }
/*fim do código inserido*/
   return ids->dbranch_if.operate(ids->dbranch, space, map, mem_trans);
  } else {
   if (ids->ibranch)
    return ids->ibranch_if.operate(ids->ibranch, space, map, mem_trans);
 }
 return 0;
}
```

Figura 3.2 - Código modificado do id-splitter

Como pode-se ver na Figura 3.2, o código do módulo serve simplesmente de interface para as memórias *cache* observarem as transações da memória. Com a modificação, estabelece-se comunicação com o simulador ao definirmos os métodos *get* e *set* dos campos criados, de modo a permitir a manipulação dos campos pelo usuário do Simics utilizando o módulo, e depois, com uma biblioteca e funções *callback* apropriadas do Simics, o programador da arquitetura simulada pode manipular o conteúdo da memória *scratchpad*.

3.4.3 Códigos de Acesso e Escrita aos Campos do Módulo

Para cada campo criado em um módulo, é preciso oferecer funções para acesso e escrita, mais comumente conhecidas como *get* e *set*. Para todos os campos de tipo inteiro ou *integer_t*, isto é relativamente simples, como pode observar-se na Figura 3.3.

```
static set_error_t
set_spm_penalty(void *dont_care, conf_object_t *obj,
      attr_value_t *val, attr_value_t *idx){
 id_splitter_t *ids = (id_splitter_t *) obj;
 ids->spm_penalty = val->u.integer;
 return Sim_Set_Ok;
}

static attr_value_t
get_spm_penalty(void *dont_care, conf_object_t *obj,
      attr_value_t *idx){
 id_splitter_t *ids = (id_splitter_t *) obj;
 return SIM_make_attr_integer(ids->spm_penalty);
}
```

Figura 3.3 - Código para get e set do campo spm_penalty

Porém, para campos cujo valor é uma lista, as funções de acesso tornam-se mais complexas. A função de *get* simplesmente retorna a lista existente, porém a função *set* pode ser problemática, pois deve ser tratada de modo diferente para os dois campos, *spm_base* e *spm_offset*, e como só dispõe-se da função *set* para a interface do Simics, ela foi modificada para oferecer opções de manipulação. Para isto, foi definida uma codificação com quatro tipos de valores possíveis:

- Uma lista, a qual será concatenada com a lista já existente;
- Um inteiro negativo, o qual, se possuir correspondente positivo na lista *spm_base*, eliminará aquele bloco do *scratchpad*;
- Zero, o que limpará a lista, deixando ela com tamanho zero/nulo;
- Um inteiro positivo, o qual, caso não possuir correspondente na lista *spm_base*, será adicionado como bloco.

Tais definições servem para a lista de endereços iniciais de cada bloco, ou seja, *spm_base*. A lista *spm_offset* precisa de tratamento diferente, pois é possível ter blocos de tamanhos iguais, portanto adicionar blocos ou remover blocos não deve ser feito na lista de *spm_offset*. Esta lista, portanto, receberá três tipos de valores:

- Uma lista que, se de tamanho igual à lista *spm_base*, substituirá a lista existente em *spm_offset*, ocorrendo apenas caso a soma das posições da lista não superem o campo *spm_size*;
- Zero, o que zera todas as posições da lista, mas mantém seu tamanho;
- Um inteiro positivo, o qual, caso não faça a soma das posições da lista superarem o campo *spm_size*, substituirá o valor da última posição da lista *spm_offset*.

As listas *spm_base* e *spm_offset* sempre terão o mesmo tamanho, dada a correspondência entre endereço inicial e tamanho do bloco. O campo *block_count* contém a informação do tamanho das listas, e é usado constantemente nas operações para manter a coerência entre as listas. Com isto, é possível mexer livremente no *scratchpad*, mesmo durante a execução através do uso da biblioteca, como demonstrado no terminal da Figura 3.4. Nesta, são passados valores para os campos que contém as listas e é possível observar as listas resultantes.

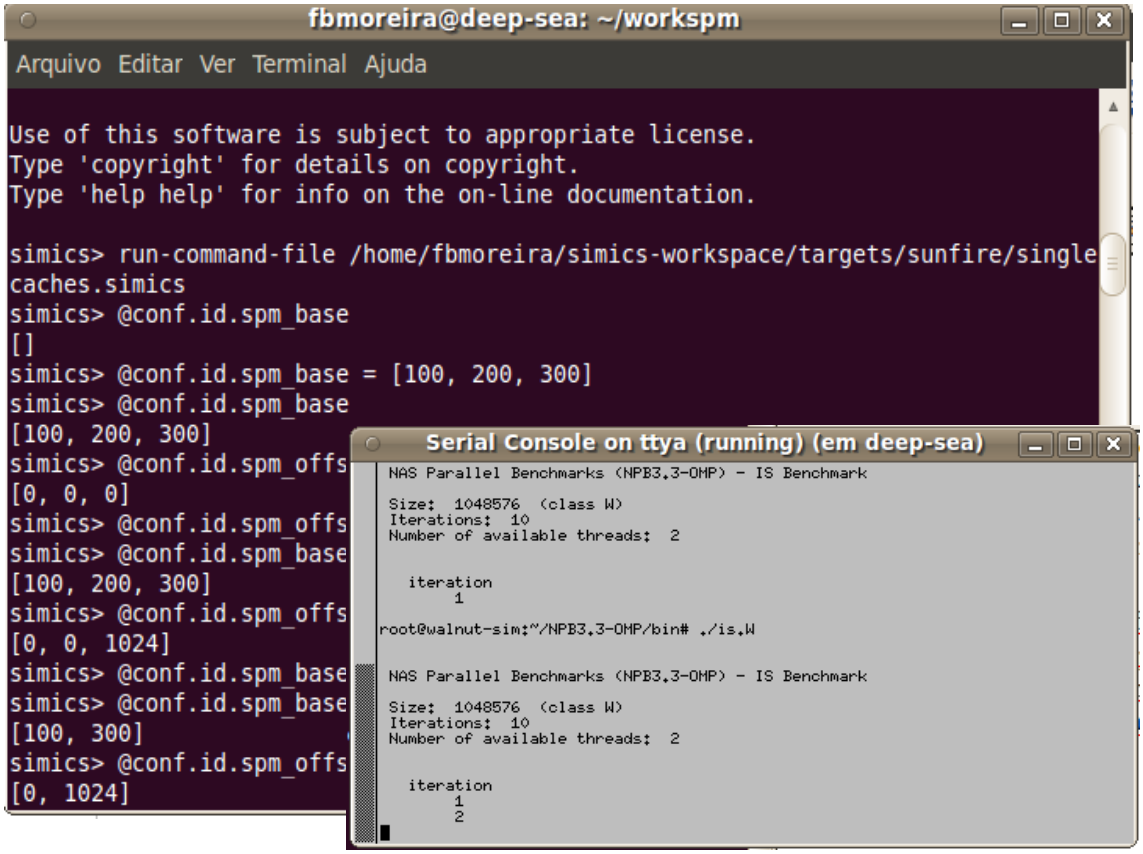

Figura 3.4 - Manipulação da SPM no simulador

4 COMUNICAÇÃO ENTRE SIMULADOR E ARQUITETURA SIMULADA

Conforme o simulador executa a simulação, é importante observar certas informações do sistema operacional e da arquitetura. O Simics registra qualquer parte das instruções executadas que sejam especificadas, mantém informação referente a *traps*, e, para interesse deste trabalho, registradores. Não somente registradores especiais, mas todos os registradores da arquitetura, dando livre acesso ao usuário do simulador para observar estes durante a simulação e, se assim desejar, modificar seu conteúdo, correndo risco de obter um estado de simulação inconsistente e em breve uma exceção na arquitetura, travando o sistema operacional simulado.

Pode-se ainda criar definições de eventos a serem tratados com uma função definida, como, por exemplo, caso a posição de memória 103094 seja acessada , pare a simulação, imprima a posição acessada, adicione 4 ao valor do registrador *g1*, e continue a simulação. O acesso à posição demarca o evento, e os outros 4 passos definem a função a ser executada caso o evento ocorra. Isto é possível devido às chamadas funções de *callback* e os gatilhos de evento *haps* no Simics.

4.1 Função *Callback*

No Simics pode-se definir um evento usando uma das definições de *haps* da API do Simics, podendo-se observar praticamente qualquer transação ou mudança de estado no simulador para denotar um evento. Para um evento definido, a função *callback* relacionada a ele, se existente, será chamada e rodará em *background.* Por padrão, esta função primeiramente para a simulação sem retornar à interface do Simics, executa, e então permite o simulador voltar à execução, embora tal parada seja opcional, nem sempre garantida conforme citado no Capítulo 3. Entre os eventos, há um evento especial, chamado de *magic instruction*: uma instrução que não faz nada na arquitetura, mas dispara o evento *Core_Magic_Instruction* no Simics, devolvendo algum valor, o qual pode ser usado pelo usuário.

No conjunto de instruções do UltraSPARC II, tal instrução é a *sethi* no registrador *g0*. O registrador *g0* é fixo em 0, portanto tentar mudar os seus bits equivale a um *nop*. Porém, ao detectar tal instrução com o registrador *g0*, o Simics dispara o evento *Core_Magic_Instruction*, e o argumento que seria passado a *g0* é passado ao simulador, permitindo definir uma função diferente para cada valor.

```
def a_callback(user_arg, cpu, breakpoint_number):
 if breakpoint_number == 1:
  for cpux in conf.system_cmp0.cpu_list:
   cpux.physical_memory.timing_model.spm_base = cpu.global_registers[0][1]
   cpux.physical_memory.timing_model.spm_offset = cpu.global_registers[0][3]
 elif breakpoint_number == 2:
  for cpux in conf.system_cmp0.cpu_list:
   cpux.physical_memory.timing_model.spm_base= -1*cpu.global_registers[0][1]
 elif breakpoint_number == 3:
  for cpux in conf.system_cmp0.cpu_list:
   cpux.physical_memory.timing_model.spm_base = 0
SIM_hap_add_callback("Core_Magic_Instruction", a_callback, None)
```

Figura 4.1 - Código da função callback

Segue na Figura 4.1, a função de comunicação entre simulador e simulação. O que este código em Python faz é, definir a função, e então adicionar ela ao evento, para que sempre que este ocorra, esta função seja chamada. Esta função callback se divide em 3 funções específicas: *add_block(1), erase_block(2)* e *clear_spm(3)*.

Observa-se, na condição *breakpoint_number == 1,* a criação de um bloco com o valor passado com o valor no registrador global 1, ou seja, *g1*, como endereço inicial do bloco. Logo após, recebe-se o valor do registrador global 2, *g2*, e passa este para *spm_offset*, ou seja, o tamanho do bloco.

Isto é feito para todos os núcleos da configuração do sistema, ou seja, é estabelecido um *scratchpad* idêntico para simular o compartilhamento do mesmo entre todos os núcleos. Os registradores globais foram usados para passar os valores para o simulador, e nele, adicionar um bloco de endereço inicial em g1 e tamanho em g2.

Em *breakpoint_number == 2*, o *buffer* recebe um valor e multiplica por -1, ou seja, é feita a remoção do bloco correspondente ao endereço passado em *g1*. No caso de *breakpoint_number == 3*, não é necessário nenhum registrador, pois a memória *scratchpad* será limpada.

4.2 Biblioteca Para Uso da SPM

Como pode ser observado na Seção 4.1 deste capítulo, tudo que o programador precisa para manipular os conteúdos da memória *scratchpad* são funções apropriadas que alterem os valores dos registradores para obter os efeitos desejados. Para tal, demonstra-se como exemplo as seguintes funções usando *assembly* do conjunto de instruções da arquitetura (ISA) *sparc v9* na Figura 4.2.

```
int add_block(void *ptr,int size){
 __asm__ __volatile__("ldx %0, %%g1\n\t"
       "lduw %1, %%g2\n\t"
       "sethi 1, %%g0"
       :
       :"m"((int)ptr),"m"(size)
       :"g1","g2"
      );
 return 0;
}

int erase_block(void *ptr){
 __asm__ __volatile__("ldx %0, %%g1\n\t"
       "sethi 2 ,%%g0\n\t"
       :
       :"m"((int)ptr)
       :"g1"
 );
 return 0;
}

int clear_spm(){
 __asm__ __volatile__("sethi 3 ,%%g0");
 return 0;
}
```

Figura 4.2 - Código da biblioteca para uso da SPM

Tais funções, escritas na linguagem C, utilizam a habilidade do GCC de executar *inline assembly*, permitindo que sejam alterados os conteúdos dos registradores, utilizando neste caso, as instruções *ldx* e *lduw*. Após, é executado o *sethi* em *g0* com o número correspondente na função *callback*. Assim, cria-se a comunicação e a manipulação da SPM em 3 funções sucintas.

No caso da arquitetura possuir *n* núcleos, é necessário definir *n* interfaces *id-splitter* (cada núcleo tem suas *caches* L1 próprias), portanto, obtém-se *n* interfaces para *n scratchpads* diferentes. Simula-se um *scratchpad* único fazendo com que cada interface possua um *scratchpad* com o tamanho original, mas também com que toda modificação seja executada em todos *scratchpads.*

Assim, o conteúdo deles é essencialmente o mesmo a qualquer momento, como demonstrado na função *callback* da Figura 4.1. Devido aos vários núcleos, conforme a frequência de acessos à memória *scratchpad,* podem tornar-se necessários mais de um *write-read port*, o que pode pesar em termos de latência como observado no Cacti[10]. A responsabilidade de sincronizar as *threads* e gerenciar o compartilhamento de memória cai sobre o programador, o qual, ao fazer modificações no *scratchpad*, deve evitar problemas de coerência para execução correta de seu aplicativo.

Em uma arquitetura com dois núcleos, demonstrada na Figura 4.3, não utiliza-se o objeto da memória para escutar as transações de memória, mas sim os espaços de mapeamento de transações dos núcleos. Estes espaços definem que transações da memória estão indo para cada núcleo. O objeto *Cpu0_space,* do tipo *memory space*, serve para mapear transações e espaços específicos da memória.

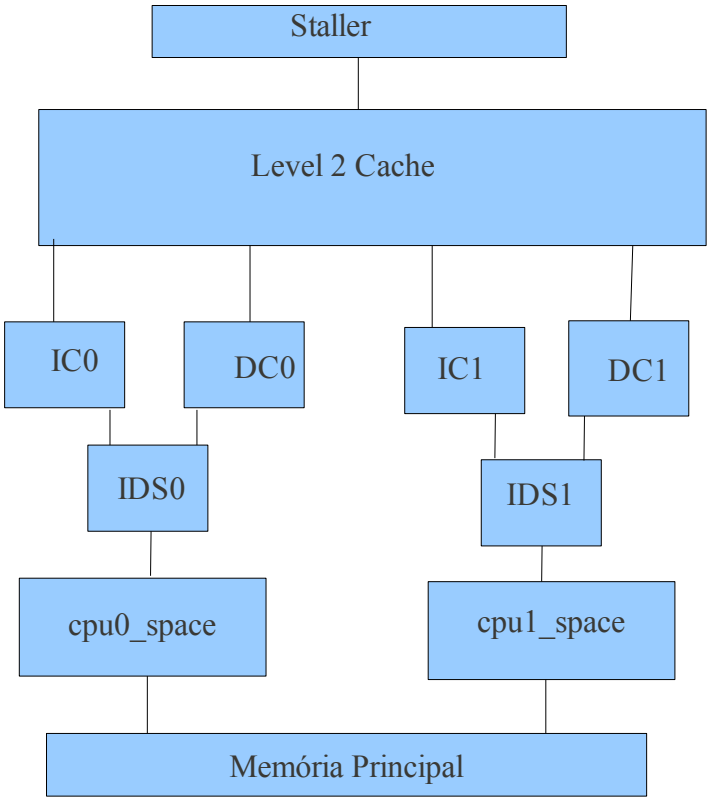

Figura 4.3 - Hierarquia da Memória para Arquitetura multi-core com 2 Núcleos

Como o Simics não é paralelizável, simulando apenas um ciclo e uma instrução por vez, cada núcleo de cada vez, o mapeamento dos dois *memory spaces* é o mesmo. A cada ciclo, um núcleo executará uma instrução. A instrução é passada aos dois memory spaces, mas como apenas um dos núcleos executará a instrução, no caso do núcleo cpu0, apenas o *cpu0_space* terá seu *timing_model* executado. Assim simula-se uma divisão e distinguem-se as transações de memória de cada processador, pois é possível replicar as unidades de observação (*id-splitters*, 0 e 1 para os respectivos núcleos), e destinar as operações às *caches* específicas.

5 MODELO DE IMPLEMENTAÇÃO DA MEMÓRIA SCRATCHPAD

Neste capítulo, é descrito o modelo de implementação da memória *scratchpad* que pretende-se avaliar. A arquitetura de um núcleo é demonstrada na Figura 5.1, feita conforme o modelo de [11]. Com o método de simulação proposto, espera-se modelar uma memória *scratchpad* que armazene dados em páginas, de modo a evitar o uso de tags e substituir eficientemente uma parcela da memória *cache* L2.

Tal decisão de projeto tem como base os resultados de [3], onde pode ser observado maior ganho de desempenho na substituição da memória *cache* L2 por memória *scratchpad*, dado que é difícil de competir com a memória *cache* L1 em termos de desempenho. Espera-se também que a hierarquia de memória cache supra a necessidade de instruções, uma vez que é rara a necessidade e frequência de uso de um determinado conjunto de instruções que não caiba na memória cache L1 de instruções.

Assim, será feita uma modificação na arquitetura da Figura 5.1, para a adição de uma memória *scratchpad* que serve como *buffer* de páginas virtuais de dados, transportadas conforme requisições do sistema operacional através de um canal DMA.Como espera-se que a memória *scratchpad* seja explicitamente controlada pelo pro gramador ou pelo compilador, utiliza-se um modelo de chamadas de sistema para expressar a inserção ou remoção de uma determinada página na memória *scratchpad*.

O interessante em levar páginas inteiras para a memória *scratchpad* e deixar o gerenciamento para o sistema operacional é evitar problemas de compatibilidade de funções e recompilação de código, pois não é preciso utilizar endereços específicos da memória *scratchpad*. Isto deve-se ao motivo de que será feita uma seleção e tradução dos endereços que se encontram na memória *scratchpad* através da manipulação dos descritores de tradução de páginas que se encontram na unidade de gerenciamento de memória.

5.1 Proposta da Arquitetura da Memória Scratchpad Simulada

Tal manipulação trata-se da adição de um bit aos descritores de tradução, o qual sinaliza a presença daquela página na memória *scratchpad*. Conforme a Figura 5.1 e a Figura 5.3, ao detectar a presença do bit em 1, os dados que normalmente seriam passados para a *cache* de dados são desviados para a memória *scratchpad*. A página traduzida, juntamente com o *block offset* (deslocamento dentro da página), seleciona a página correta na memória *scratchpad*, assim como o endereço correto dentro de tal página.

Tal descritor de tradução deve, portanto, ser manipulado pelo sistema operacional, para que obtenha-se a posição correta da página dentro da memória *scratchpad*. Faz-se

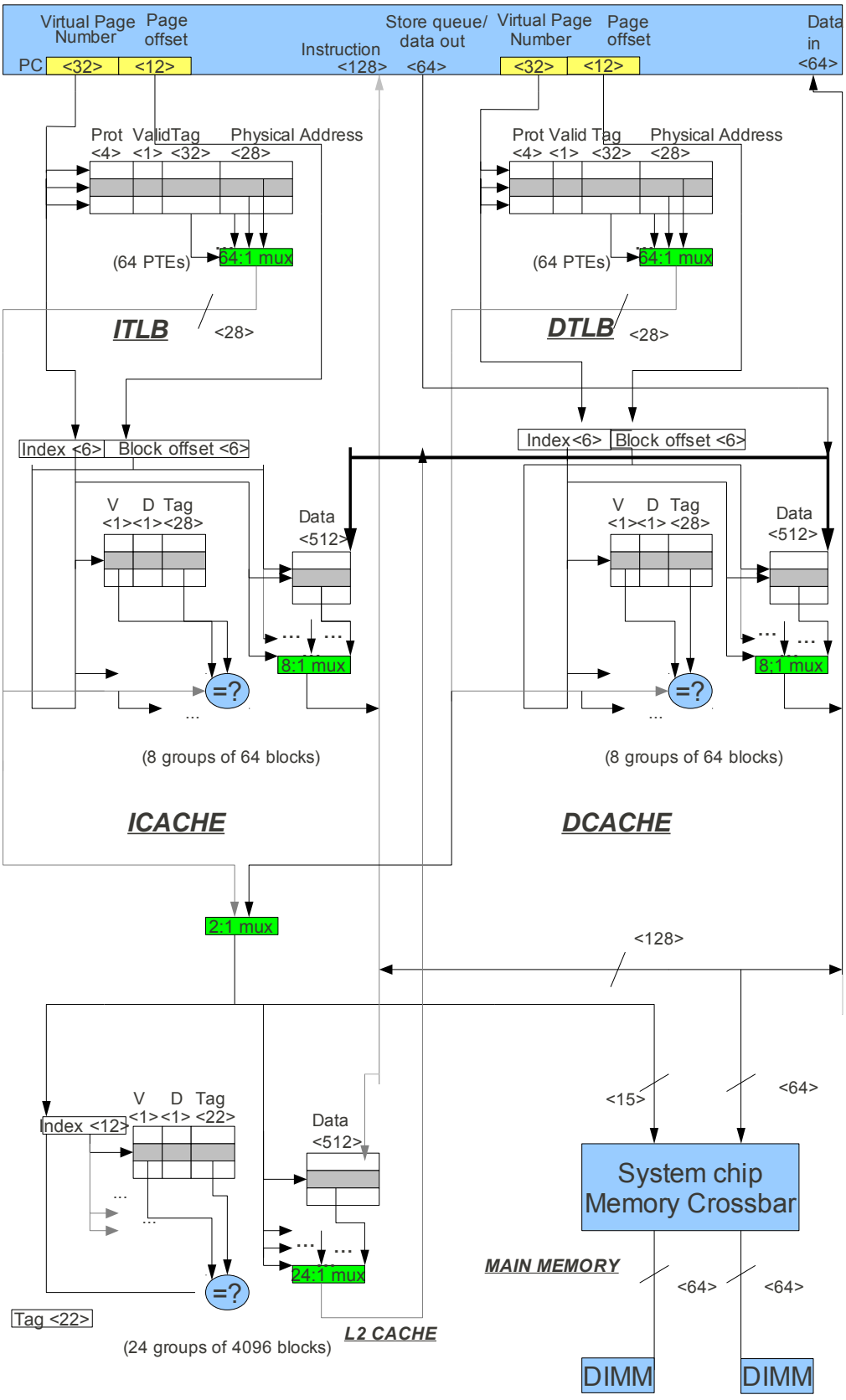

Figura 5.1 - Visualização Física da Hierarquia de Memória Original de um Núcleo

isto na chamada de sistema ocorrida para inserir a página na memória *scratchpad*. Inicialmente, o descritor da página é inserido em uma tabela de páginas da memória *scratchpad* (a qual deverá ser mantida pelo sistema operacional). Em seguida, atribui-se ao campo de tradução deste descritor nesta tabela um endereço livre da memória *scratchpad*, indicando onde a página começa, com tamanho seguinte suficiente para armazenar a página. A Figura 5.2 demonstra os passos para inserção de uma página.

Marca-se o descritor com o bit "spm" em 1 para sinalizar que aquela tradução está com uma página na memória *scratchpad*. Este endereço portanto é estabelecido como a tradução para o descritor daquela página virtual, ou seja, quando a MMU fizer a tradução de uma página que está na memória *scratchpad* (bit "spm" em 1), acessará a memória *scratchpad* a partir do endereço que fora inserido no descritor, adicionando-se o *block offset*, a fim de obter o endereço dentro da página, e assim acessando o endereço dentro da memória *scratchpad*.

É viável utilizar vários algoritmos, e optou-se por *First Come, First Served,* ou seja, quem faz a requisição primeiro obtém o recurso do armazenamento na memória *scratchpad*. Esta decisão de projeto confia ao programador a competência em alocar estruturas frequentemente acessadas e de tamanho considerável para a memória *scratchpad,* já que um algoritmo que tentasse recorrer a um *profiling* que expressasse a frequência de acesso a cada página seria custoso. Caso a memória esteja totalmente ocupada, a chamada de sistema não retorna nada(ou retorna um 0), e o programa pode continuar normalmente, pois o fato da página estar ou não na memória *scratchpad* é indiferente ao programa.

Portanto, podemos usar a memória *scratchpad* em qualquer código para uma memória *scratchpad* de tamanho desconhecido, e teremos assim portabilidade entre plataformas, desde que o sistema operacional suporte o uso da memória scratchpad. Mesmo que não suporte, é fácil escrever uma chamada de sistema que retorne 0, conforme padrão ANSI-C, assim permitindo que o código compilado rode mesmo sem uma memória *scratchpad* no sistema.

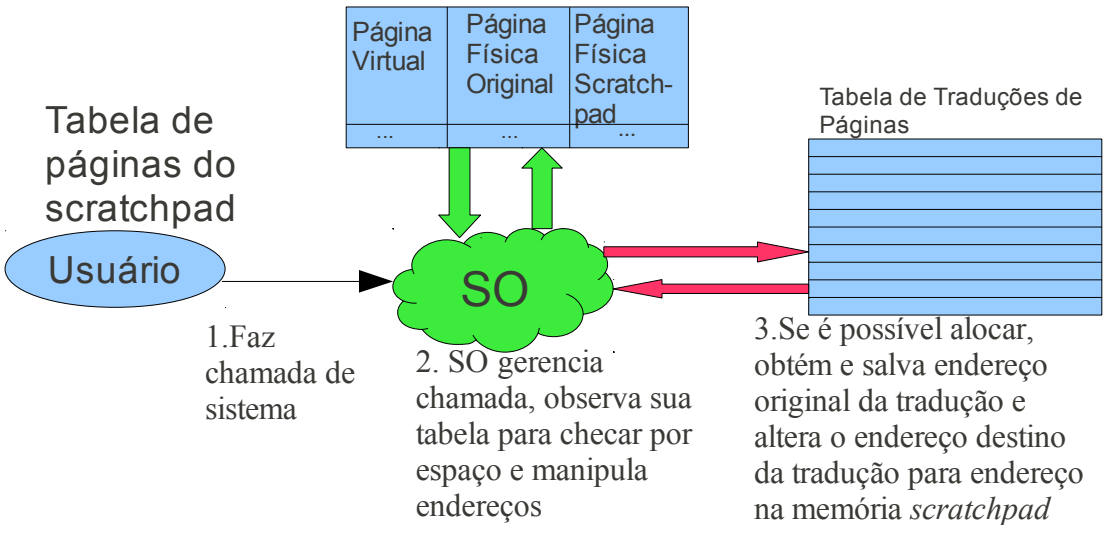

Figura 5.2 – Abstração do modelo usado

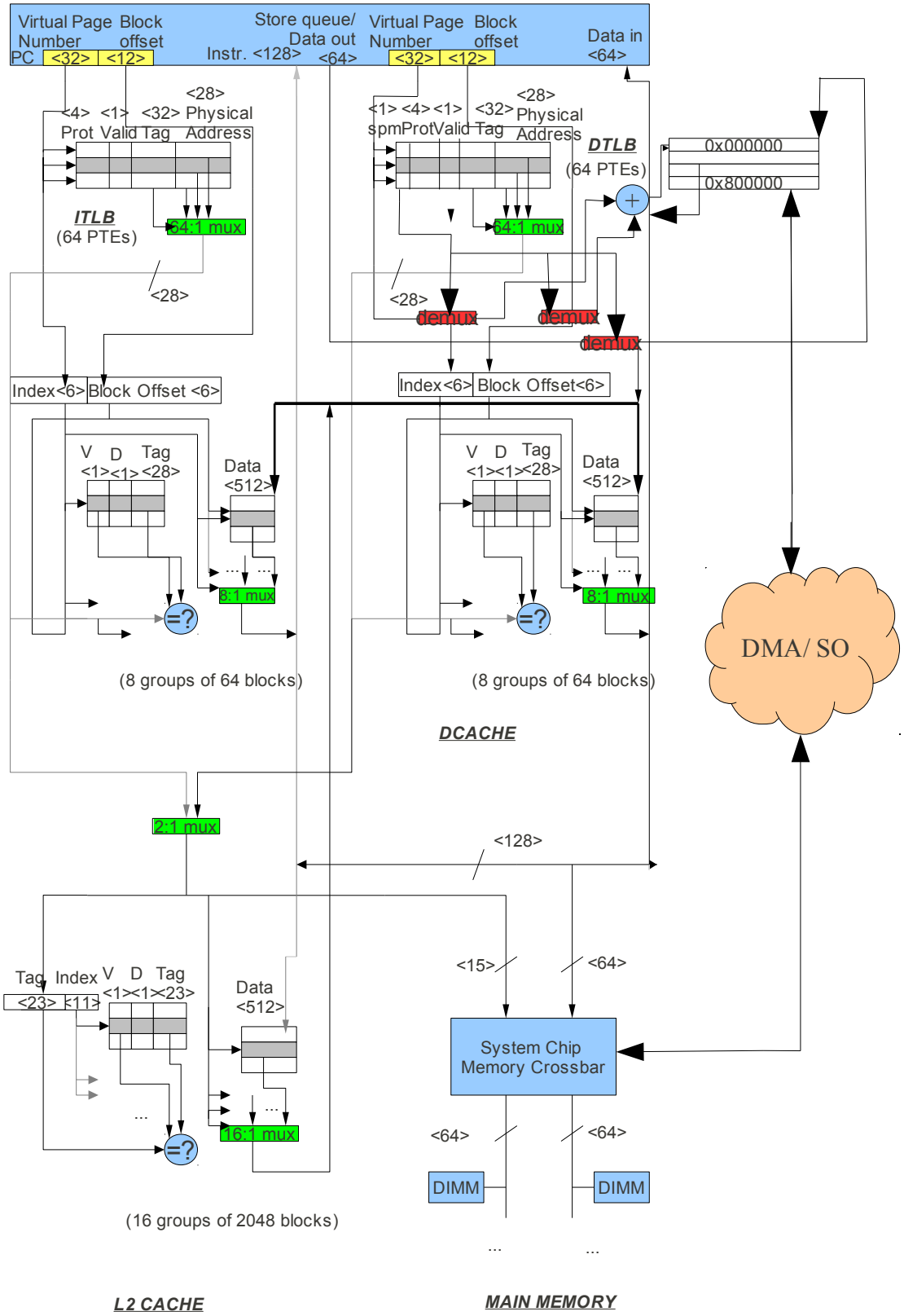

Figura 5.3 - Visualização Física da Hierarquia de Memória do Núcleo Simulado

6 RESULTADOS DE DESEMPENHO

6.1 Modelo da Arquitetura Usada

Para comparação de performance dos métodos e arquiteturas, foram realizadadas simulações com nove dos dez *benchmarks* do NAS *Parallel Benchmarks* (NPB)[12], todos executados com tamanho *W* (segundo menor tamanho disponível para o conjunto), devido ao tempo necessário para simulação. Foram executados com configurações usando apenas *cache*, e configurações usando de uma a quatro portas para a memória *scratchpad*, de modo a observar a influência e necessidade do número de portas maior para os acessos vindos dos quatro núcleos.

A latência adicional dado o número maior de portas não foi considerada, pois o foco do teste era prever a necessidade de portas adicionais dado a frequência de acesso obtida, considerando-se como desvantagem principal o preço do *hardware* adicional, e não a latência. As simulações foram executadas com a configuração demonstrada na tabela 6.1, simulada dentro do Simics versão 3.0.31.

Tabela 6.1 - Configuração usada para simulação com quatro núcleos.

OS	Ubuntu 6.06.2 LTS – SMP	
Núcleos de Processamento	IPC(sem *cache*)	1
	Pipeline	Não Modelado
	Número de Núcleos	4 Núcleos
	Modelo do Núcleo	UltraSPARC II – V9
	Frequência do Relógio	2 Ghz
	Tecnologia de Integração	45 nm
Memória Principal	Tamanho	1 GB
	Tempo de Acesso	78 ciclos
	Tecnologia de Integração	65 nm
Caches	L1 Instrução	32KB 4-way associativa
	L1 Dados	32 KB 8-way associativa
	L2	6 MB 16-way associativa

Para arquitetura *multi-core*, foi usado o núcleo UltraSPARC II, e modelou-se uma hierarquia de memória semelhante ao de um processador da família Harpertown da Intel. A arquitetura do UltraSPARC, conforme usada na simulação, não bate com o de um UltraSPARC no mundo físico, pois ele não possui L2 normalmente, utilizando uma *cache* externa. Ele possui uma *cache* de dados mapeada diretamente, e uma *cache* de instruções com associatividade 2. O modelo de *cache* usado foi escolhido para comparar ganhos com hierarquias de memória mais atuais, embora mantendo com uma arquitetura relativamente simples para melhor visualização, modelagem, e posterior modificação ao adicionar o *scratchpad*.

6.2 Resultados do Uso da SPM

Nesta seção foi modelada uma memória *scratchpad* de 8MB, substituindo 8MB de memória L2 em uma hierarquia de memória usada no processador Harpertown da Intel: 4 núcleos, cada um com 32KB de L1 de instruções e 32KB de L1 de dados, sendo que cada par de núcleos compartilha uma memória L2 de 6MB. As *caches* L2 serão reduzidas para 1MB cada uma, devido ao espaço necessário para a memória *scratchpad*, a reorganização dos núcleos para permitir o compartilhamento da memória *scratchpad* entre todos os núcleos, e a estrutura de DMA (“Direct Memory Access”) necessária para a comunicação eficiente entre a memória *scratchpad* e a memória principal, conforme modelo apresentado no Capítulo 5.

A decisão de projeto que leva a uma memória *scratchpad* tão grande se deve à necessidade de alocação desta por vários programas em um sistema operacional genérico. O tamanho de estruturas usadas é usualmente grande, e a diferença de latência e energia entre memórias *cache* e *scratchpad* parece escalar conforme testes no CACTI[10]. Seria possível utilizar um tamanho menor de memória *scratchpad* com maior quantidade de memória *cache* L2*,* mas tendo como base o número de processos rodando em um sistema operacional tradicional e o tamanho das estruturas de dado usadas, o tamanho escolhido demonstra-se eficiente. Adicionalmente, a escolha de uma memória *scratchpad* compartilhada entre todos núcleos tenta abusar de qualquer compartilhamento de memória entre 2 núcleos para permitir comunicação mais eficiente entre 2 *threads.*

Para a alocação das estruturas de dados dos programas foi necessário modificar o código dos *benchmarks* usados, para fazer as chamadas ao Simics descrita nos capítulos 3 e 4 e adicionar as estruturas desejadas à memória *scratchpad.* Para selecionar as estruturas foi usado o programa Vtune[13] da Intel, o qual utiliza de *hardware counters* nos processadores da Intel para fazer *profiling* de aplicações. Com isto, foi possível saber quais estruturas de dados demoravam mais tempo na execução, ou seja, quais estruturas seria aconselhável colocar na memória *scratchpad*.

Todas as aplicações tiveram a alocação da memória no *scratchpad* estaticamente, no início das aplicações, o que prejudicou alguns *benchmarks*. Foram obtidos resultados de todas aplicações do NPB, exceto da aplicação DC (*Data Cube Operator*), a qual não executaria em tempo viável dentro de uma simulação. São fornecidas as diferenças de tempo e transações de memórias executadas para cada *benchmark* e um histograma apresentando a frequência de acesso a cada página dentro da memória *scratchpad*, a ponto de analisar o uso da memória. As memórias estão expressas da seguinte lmaneira:

"dc0" a "dc3" são as memórias cache de dados dos respectivos núcleos (de 0 a 3); "l2c0" e "l2c1" são as memórias cache L2 compartilhadas entre os pares de núcleos 0-1 e 2-3, respectivamente; e "SPM" representa a memória scratchpad, com acessos apenas nas configurações quando usada.

6.2.1 Desempenho com Benchmark BT (Block Tridiagonal)

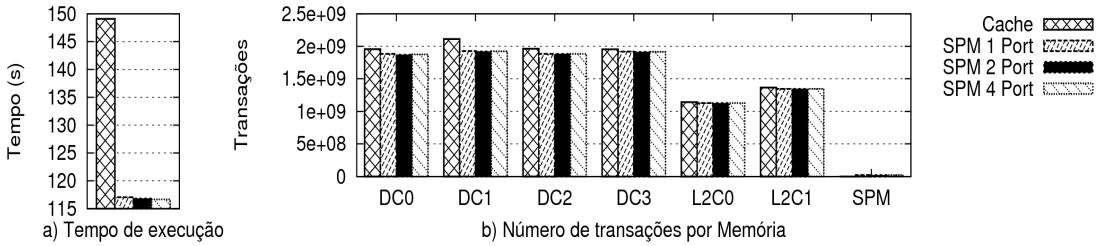

Figura 6.1 – Desempenho do BT.

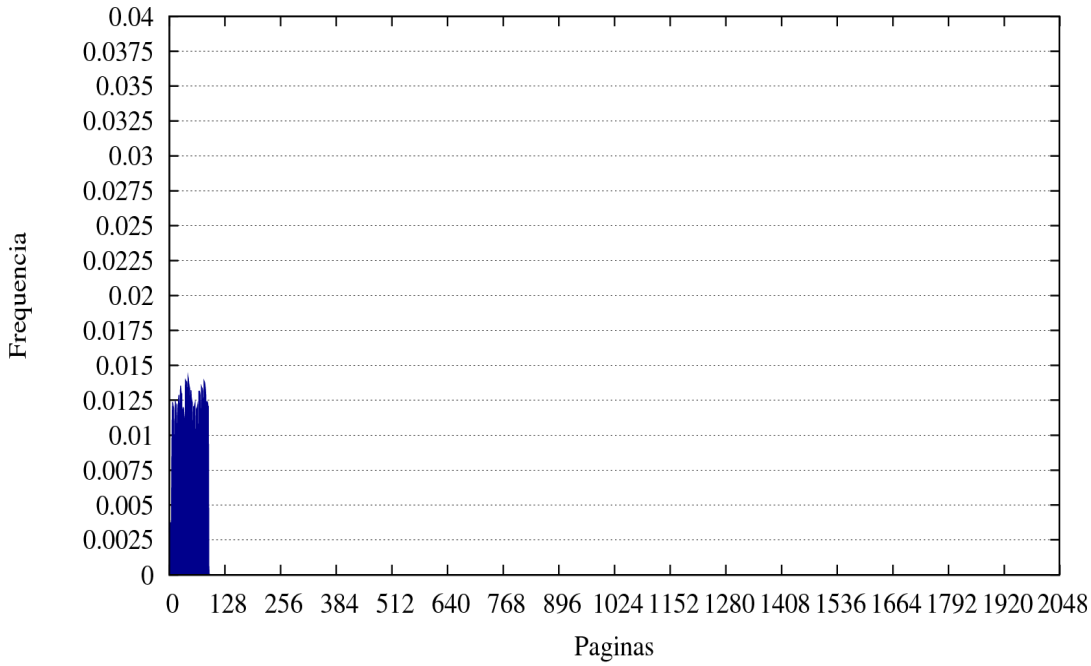

Figura 6.2 – Frequência de Acesso às Páginas da SPM para BT.

Observa-se, no *benchmark* BT, o uso de uma parcela pequena da memória *scratchpad*, usando apenas 92 páginas da memória *scratchpad,* conforme a Figura 6.2. Ainda assim, os resultados apresentam ganhos de 21,75% conforme a Figura 6.1, devido à alta concentração de acessos nesta parcela de páginas. Observando a estrutura do programa, é possível assumir ganhos maiores conforme adiciona-se uma parte maior da estrutura principal do programa (para tamanho W, uma matriz tridimensional de 33x33x33) à memória *scratchpad*.

6.2.2 Desempenho com Benchmark CG (Conjugate Gradient)

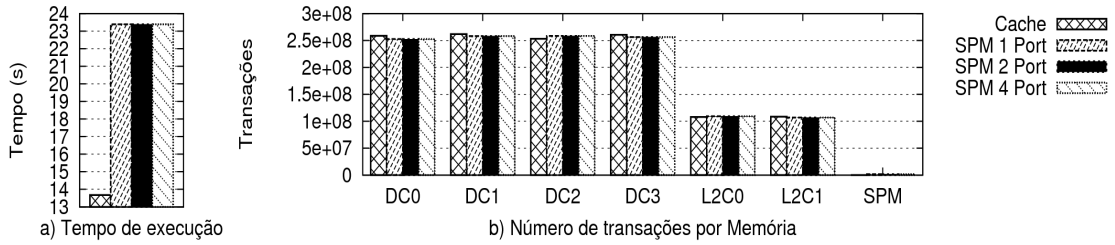

Figura 6.3 – Desempenho do CG.

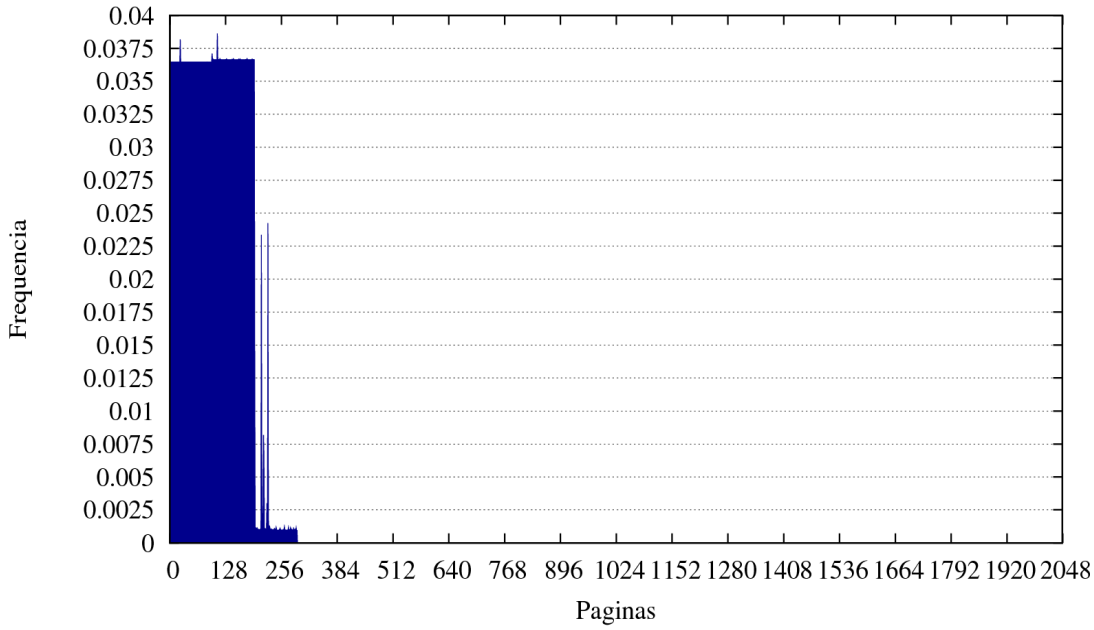

Figura 6.4 – Frequência de Acesso às Páginas da SPM para CG.

O padrão de acessos à memória randômico no *benchmark* CG e a falta de informação em relação às estruturas do programa fez com que o uso da memória *scratchpad* fosse insuficiente em comparação ao necessário. Como a *cache* L2 fora reduzida, também não fora possível armazenar as estruturas nela, o que gerou perda de desempenho. Os tempos de execução do benchmark apresentam perda de 71,03% , na Figura 6.3. A frequência do uso das páginas se encontra na Figura 6.4. Para o tamanho W, o benchmark utiliza uma matriz esparsa positiva de 7000 posições.

6.2.3 Desempenho com Benchmark EP (Embarrassingly Parallel)

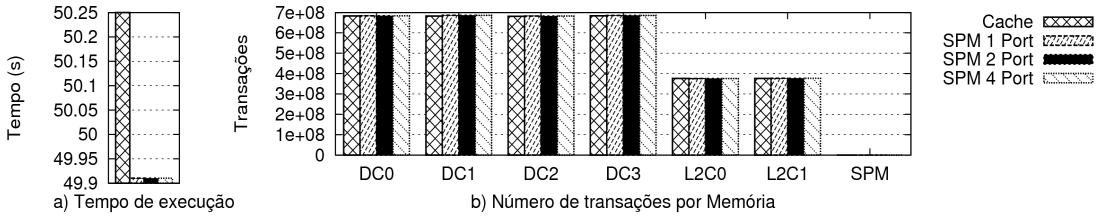

Figura 6.5 – Desempenho do EP.

O *benchmark* EP praticamente não usa memória, pois usa o método polar Marsaglia para gerar pares de números randômicos. Na Figura 6.5, é demonstrado que não existe diferença significante entre os tempos de execução. Foi colocada uma página que poderia ser considerada relevante, a qual, na figura resultante, estaria colada ao eixo de frequência. Portanto, por irrelevância, tal figura não fora adicionada. Mesmo esta página praticamente não tem acessos, gerando nenhum ganho ou perda de desempenho.

6.2.4 Desempenho com Benchmark FT (Fast Fourier Transform)

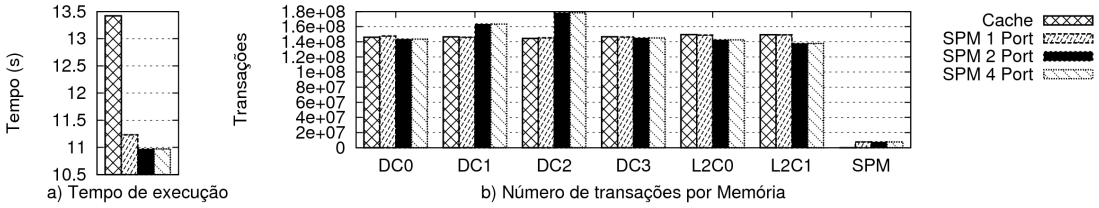

Figura 6.6 – Desempenho do FT.

O *benchmark* FT realiza a transformada rápida de Fourier em uma grande estrutura matricial com dimensões de 128, 128 e 32. Os acessos são bem distribuídos e regulares, conforme observável na Figura 6.7, e devido à alocação de grande parte da estrutura, obteve-se ganho de desempenho de 18,3%, como observado na Figura 6.6.

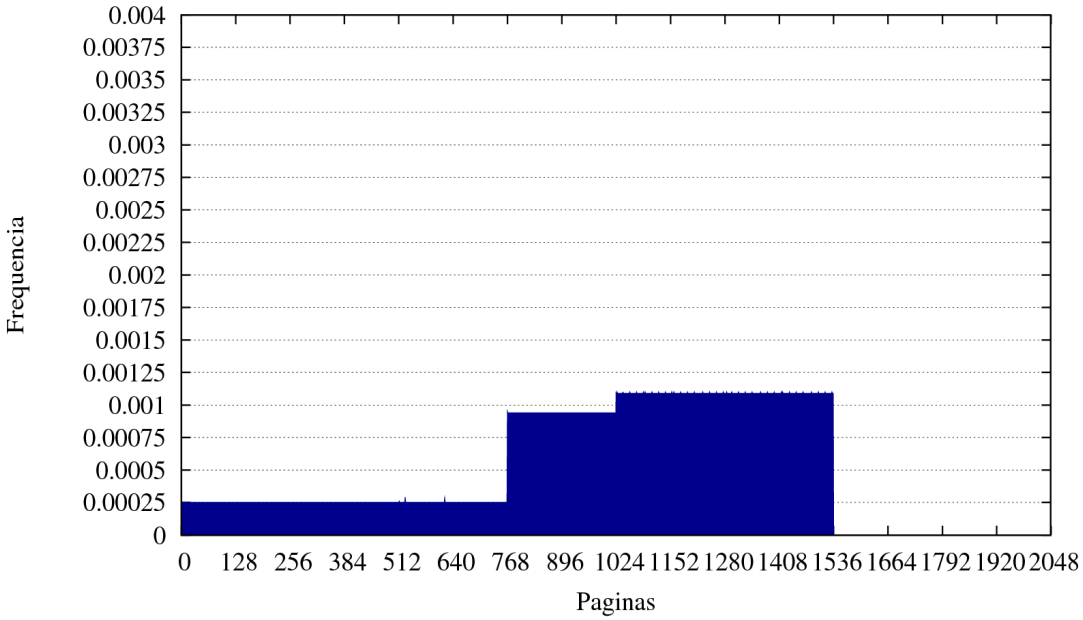

Figura 6.7 – Frequência de Acesso às Páginas da SPM para FT.

6.2.5 Desempenho com Benchmark IS (Integer Sort)

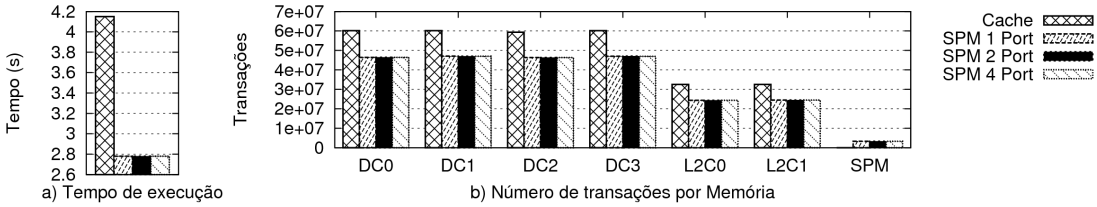

Figura 6.8 – Desempenho do IS.

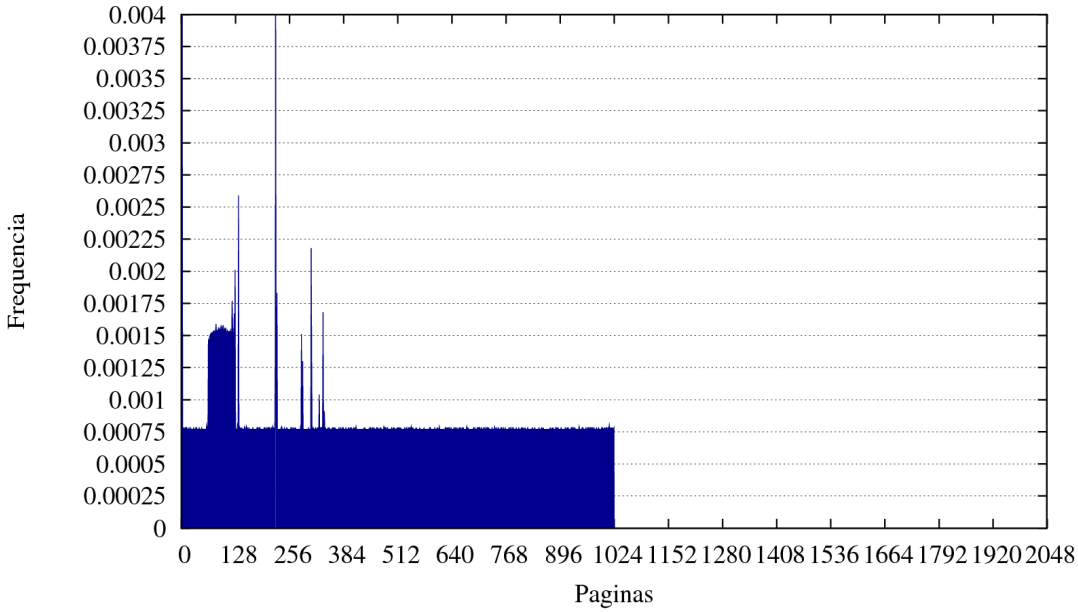

Figura 6.9 – Frequência de Acesso às Páginas da SPM para IS.

A princípio, o *benchmark* IS, o qual realiza a classificação de um vetor de inteiros de 1048576 posições, apresenta acessos randômicos à memória tal qual o *benchmark* CG. Porém, devido ao conhecimento prévio das estruturas principais, foi possível armazenar a estrutura inteira dentro da memória *scratchpad*, resultando em ganhos de desempenho de 33,12% conforme a Figura 6.8. A frequência de acesso às páginas é constante na maior parte da estrutura, como pode ser observado na Figura 6.9.

6.2.6 Desempenho com Benchmark LU (Lower Upper Symetric Gauss Seidel)

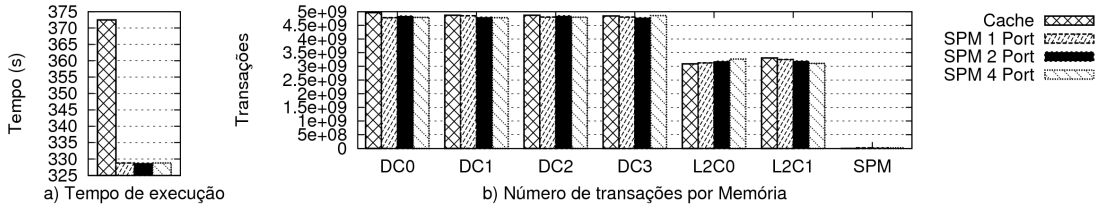

Figura 6.10 – Desempenho do LU.

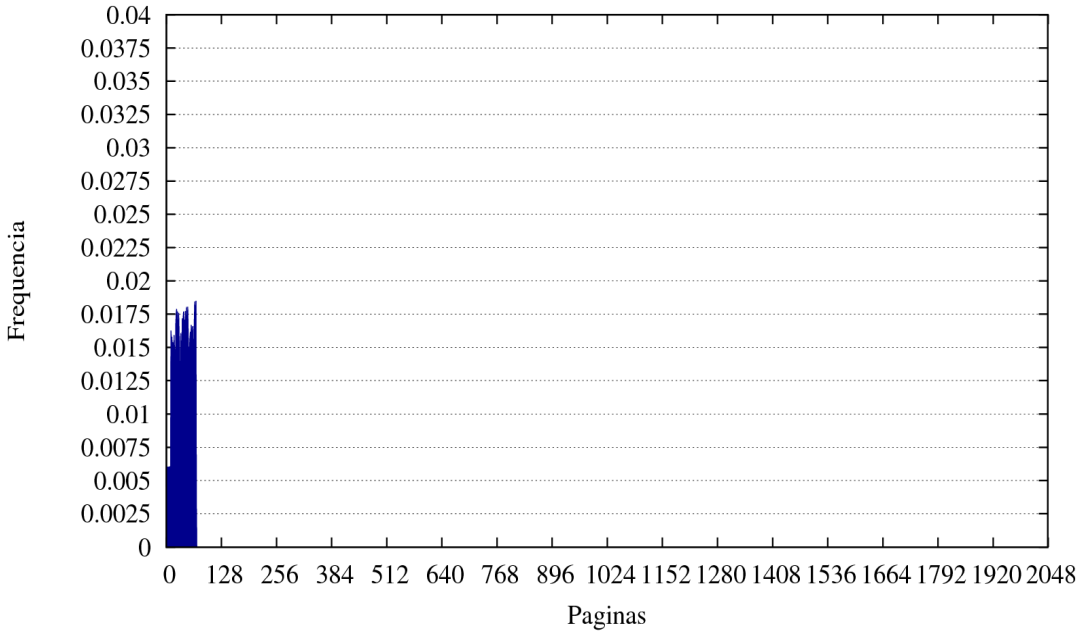

Figura 6.11 – Frequência de Acesso às Páginas da SPM para LU.

Algoritmo de resolução de sistemas tal qual o *benchmark* BT, o *benchmark* LU resolve um sistema de equações não lineares de uma matriz 33 por 33 por 33. Apresentou ganho de desempenho de 12,15% conforme a Figura 6.10, mesmo usando-se apenas uma parcela pequena da memória *scratchpad*, como pode ser observado na Figura 6.11.

6.2.7 Desempenho com Benchmark MG (MultiGrid)

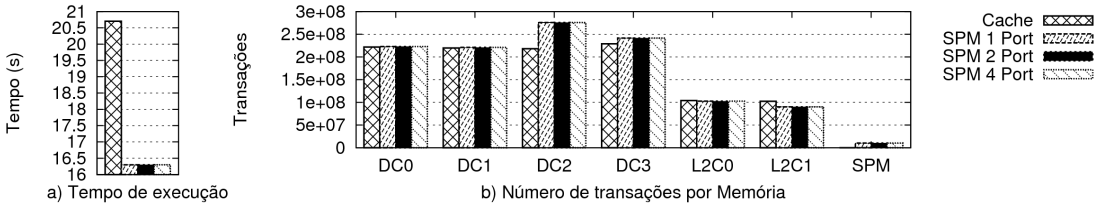

Figura 6.12 – Desempenho do MG.

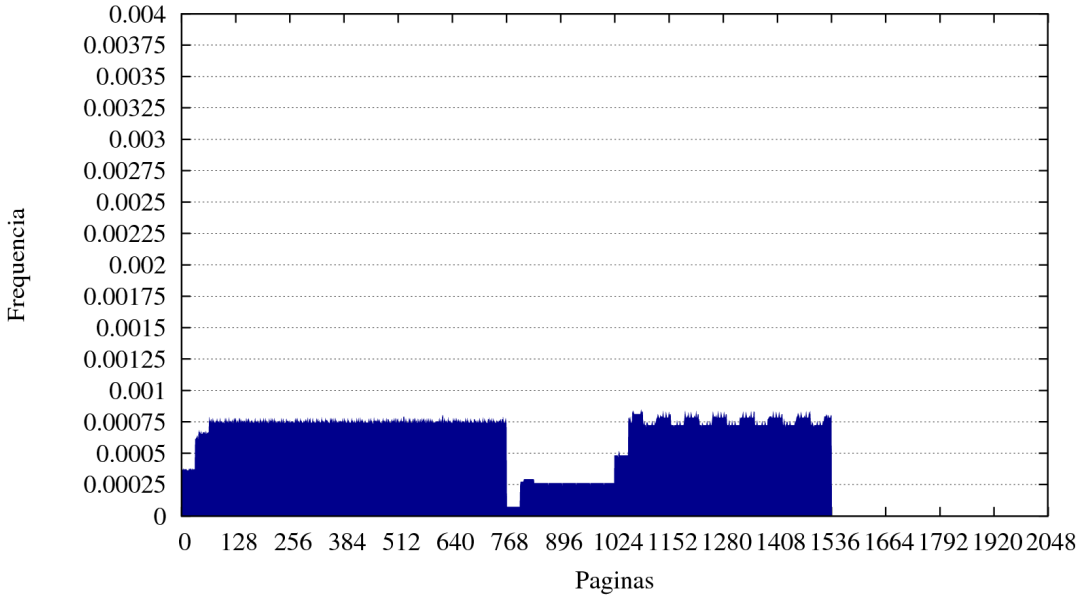

Figura 6.13 – Frequência de Acesso às Páginas da SPM para MG.

O *benchmark* MG apresenta uma estrutura principal clara, uma equação discreta de Poisson tridimensional com dimensões de 128 por 128 por 128, com acessos regulares, facilitando o uso de grande parcela da memória *scratchpad*. Pode-se observar na Figura 6.13 que os acessos distribuem-se uniformemente em duas grandes faixas de página na memória *scratchpad*, porém, obtém-se pouco ganho de desempenho em comparação a outros *benchmarks*, pois a quantidade de memória exigida pelo *benchmark* é maior do que esta hierarquia de memória pode suportar. A Figura 6.12 demonstra uma diferença de desempenho de 20,39%. A falta de memória *cache* de nível 2 mostrou-se irrelevante, pois o ganho obtido pela memória *scratchpad* para tais faixas de memória compensou a perda para o resto do *benchmark*.

6.2.8 Desempenho com Benchmark SP (Scalar Pentadiagonal)

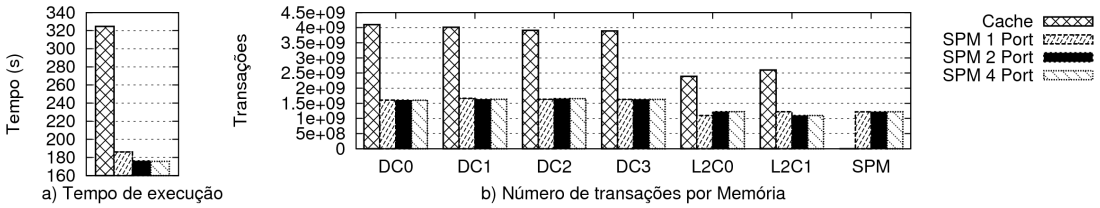

Figura 6.14 – Desempenho do SP.

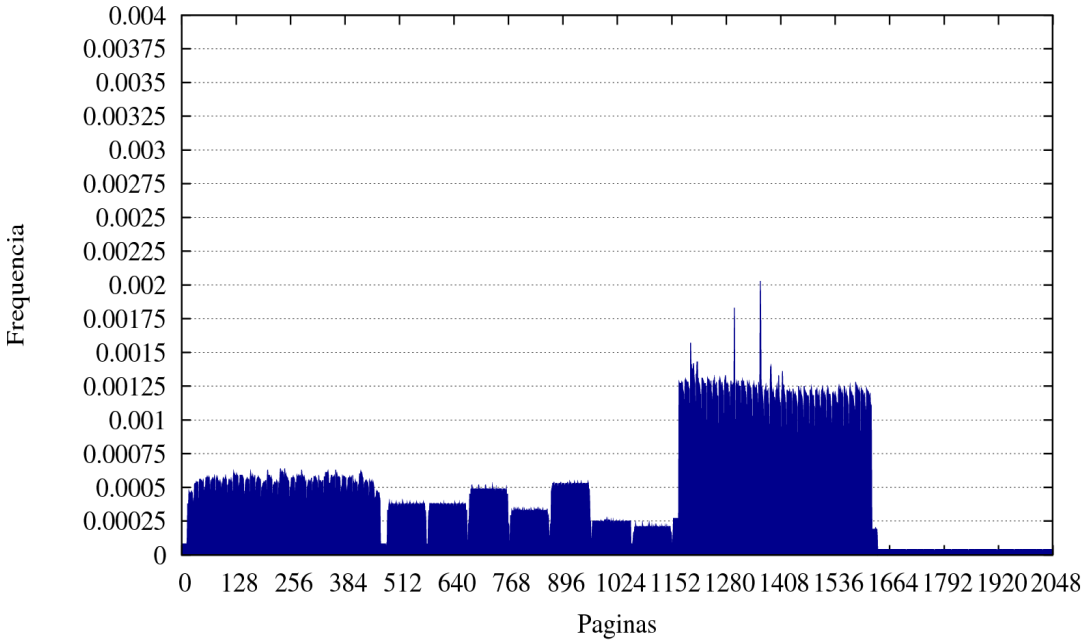

Figura 6.15 – Frequência de Acesso às Páginas da SPM para SP.

Tal qual os *benchmarks* BT e LU, o *benchmark* SP resolve um conjunto de sistemas não lineares de uma matriz 36 por 36 por 36, com acessos regulares. Porém, devido ao código mais claro, foi possível observar quais eram as estruturas mais usadas e alocar parcelas delas na memória *scratchpad*, usando a memória *scratchpad* inteira. O resultado foi um ganho de desempenho superior a qualquer outro *benchmark*, de 45,9% conforme a Figura 6.14. Isto deve-se ao uso intensivo de todas páginas alocadas na memória *scratchpad*, conforme a Figura 6.15.

6.2.9 Desempenho com Benchmark UA (Unstructed Adaptive)

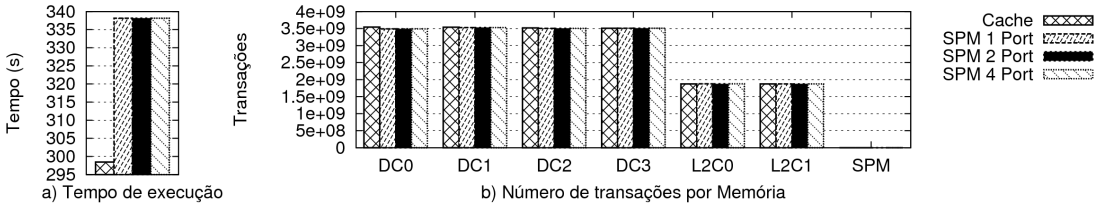

Figura 6.16 – Desempenho do UA.

O *benchmark* UA trata de cálculos de transferência de calor, e assim como os *benchmarks* CG e IS, possui acessos a memória de forma randômica. Porém, este algoritmo também faz a alocação dinâmica da memória, sendo impossível utilizar de um método de alocação estática que consiga obter bom desempenho nele. Obteve-se perda de desempenho de 13,29%, conforme observado na Figura 6.16, pois alocou-se apenas uma página, a qual fica sobreposta ao eixo de frequências, tal qual o *benchmark* EP. Portanto, tal figura não foi demonstrada devido à irrelevância.

Este *benchmark,* UA, foi feito especificamente para evitar ganhos com manipulação de conteúdo de memória através de compiladores e *hardware* especializado[14], o que

torna necessário o suporte à alocação dinâmica da memória scratchpad para ganho de desempenho. Devido à necessidade de uma chamada de sistema para alocação de páginas, atualmente não há forma de ganhar desempenho neste benchmark, pois mesmo com alocação dinâmica o *overhead* da chamada de sistema seria um empecilho.

6.3 Análise dos Resultados

Após observados os resultados de todos os *benchmarks*, foi possível inferir uma média de ganho de desempenho de 7,47%, com ganhos de até 45%. A maioria dos ganhos foi retirada de *benchmarks* onde a memória *scratchpad* servia como garantia da obtenção de dados, técnica já utilizada em compiladores e *hardware* específicos para obter ganhos com os *benchmarks* NPB. A diferença dos resultados aqui apresentados é a viabilidade de programação e manipulação da memória *scratchpad*, o que permite que no futuro use-se de alocação dinâmica da memória (caso achado um mecanismo eficiente o suficiente) para obtenção de ganhos de performance com os *benchmarks* cujos desempenhos pesaram negativamente na média, CG e UA.

7 CONCLUSÕES

Observa-se, à luz dos resultados, um ganho de desempenho proporcional ao uso da memória *scratchpad*, mesmo após uma instalação e modelagem de *overhead* para a chamada de sistema responsável pela inserção de páginas na memória *scratchpad*. Isto deve-se ao fato de que com o uso de uma estrutura de DMA (*Direct Memory Access*) para obtenção de dados para o *scratchpad*, elimina-se a maior parcela do tempo de espera advindo do processador com a chamada de sistema. Consegue-se acessar dados mais eficientemente do que a memória *cache*, pois não é preciso fazer comparação de *tags* como a *cache,* e não é necessário um protocolo de coerência, que geraria invalidações, gerando mais tráfego na hierarquia de memória, portanto, menor eficiência.

Conforme os resultados dos vários *benchmarks*, nota-se que a memória *scratchpad* se comporta como uma solução satisfatória para a grande maioria destes, chegando a ganhos de 45%, sendo que as exceções, CG e UA, podem obter tempos melhores através de *profiling* para o CG, e de alocação dinâmica dentro do programa para o UA.

Em um primeiro momento, foi feita uma pesquisa inicial que demonstrou as variadas metodologias de implementação e uso do *scratchpad*, tanto em hardware como em software. Observou-se a tendência de novas pesquisas rumarem na direção de uso de memórias *scratchpad* em ambientes *multi-core*, onde as soluções achadas tratavam do uso de memórias *scratchpad* como o uso em sistemas embarcados, com dificuldade para programação e aplicação específica. As publicações mais recentes focam no uso de algoritmos especializados para os compiladores decidirem a alocação ótima do *scratchpad* para o programa compilado.

Para obtenção da melhor performance com os *benchmarks*, está sendo analisada a possibilidade de se implementar uma mudança no algoritmo do compilador demonstrado em [15], para utilizar o *scratchpad* da maneira mais efetiva possível sem aumentar o esforço do programador. Também é possível a aplicação das ferramentas de *profiling* descritas em [16], que facilitariam a alocação para todos *benchmarks*, em especial o *benchmark* CG.

REFERÊNCIAS

[1]JACOB, B.; NG, S.; e WANG, D. **Memory Systems: Cache, DRAM, Disk.** [S.l.],Morgan Kaufmann Pub, 2007.

[2]SUHENDRA, V; ROYCHOUDHURY, A; e MITRA, T. (2010). **Scratchpad Allocation for Concurrent Embedded Software**. ACM Trans. Program. Lang. Syst., 32:13:1–13:47.

[3]MAGNUSSON, P. et al **Simics: A Full System Simulation Platform. Computer**, 35(2):50-58, feb 2002.

[4]YANAMANDRA, A; COVER, B; RAGHAVAN, P; IRWIN, M; e KANDEMIR, M, (2008). **Evaluating the Role of Scratchpad Memories in Chip Multiprocessors for Sparse Matrix Computations.** Em: IEEE International Symposium on Parallel and Distributed Processing., IPDPS ’08, pages 1 –10.

[5]BANAKAR, R;STEINKE, S; LEE, B.S;BALAKRISHNAN, M.; e MARWEDEL, P., (2002)**Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems.** Em: Proceedings of the tenth international symposium on Hardware/software codesign, ACM, páginas 73—78

[6]FRANCESCO, P; MARCHAL, P; ATIENZA, D; BENINI, L; CATTHOR, F; e MENDIAS, J. M, (2004). **An Integrated Hardware/software Approach for Run-time Scratchpad Management.** Em: Proceedings of the 41st annual Design Automation Conference, DAC ’04, páginas 238–243, New York, NY, USA. ACM.

[7]NGUYEN, N; DOMINGUEZ, A; e BARUA, R, (2009). **Memory Allocation for Embedded Systems with a Compile-time-unknown Scratchpad Size**. ACM Trans. Embed. Comput. Syst., 8:21:1–21:32.

[8]SANGMIN SEO; JAEJIN LEE; SURA, Z.;(2009) **Design and Implementation of Software-Managed Caches for Multi-cores with Local Memory**. Em: **High** Performance **Computer Architecture**, HCPA, páginas 55--66

[9] SIMICS Programming Guide. [S.l.:s.n], 1998.

[10]TARJAN, D.; THOZIYOOR, S.; JOUPPI, NORMAN P. **CACTI 4.0 HP** Laboratories, Palo Alto, HPL-2006-86, June 2, 2006 Disponível em: <http://www.hpl.hp.com/techreports/2006/HPL-2006-86.html>. Acesso em dez. 2010.

[11]HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture : a quantitative approach**. San Francisco: Morgan Kaufmann, 4th edition, 2007.

[12]JIN, H.; FRUMKIN, M.; YAN, J., The **OpenMP Implementation of NAS *Parallel Benchmarks* and Its Performance,** NAS Technical Report NAS-99-011, NASA Ames Research Center, Moffett Field, CA.

[13]R. K. MALLADI, **Using Intel VTune Performance Analyzer Events/Ratios and Optimizing Applications**, Jan. 2009, http://software.intel.com.

[14]FENG, H; VAN DER WIJNGAART, F; BISWAS, R; MAVRIPLIS, C. (July 2004) **Unstructed Adaptive (UA) NAS Parallel Benchmark,** Version **1.0,** NAS Technical Report NAS-04-006, NASA Ames Research Center, Moffet Field, CA.

[15] LI, L; GAO, L.; e XUE, J. **Memory Coloring: a Compiler Approach for Scratchpad Memory Management**. Em: *Parallel* Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, páginas 329 – 338, 17-21 2005.

[16]CRUZ, E. H. M. ; ALVES, M. A. Z. ; NAVAUX, P. O. A. . **Process Mapping Based on Memory Access Traces**. Em: XI Simpósio em Sistemas Computacionais (WSCAD-SSC 2010), 2010, Petrópolis - RJ. WSCAD-SSC 2010, 2010.