Geração de traços de simulação para instruções de processamento em memória

Aline Santana Cordeiro, Marco Antonio Zanata Alves
Departamento de Informática – Universidade Federal do Paraná
asc12@inf.ufpr.br, mazalves@inf.ufpr.br

***Resumo*—Diversas novas arquiteturas de processamento em memória, como o Hybrid Memory Cube (HMC), estão surgindo com o intuito de reduzir a movimentação de dados entre memória e processador. O principal foco das pesquisas nesse contexto são a proposta e avaliação de novas técnicas arquiteturais para garantir e melhorar esse processamento em memória. Entretanto, para tal, utiliza-se simuladores que, muitas vezes, dependem de traços de execução real. A geração desses traços para arquiteturas/instruções inexistentes não é automatizada e envolve a escrita manual de códigos em linguagem de montagem interpretáveis pelo simulador, o que é uma tarefa propensa a erros. Nesse artigo apresentamos uma solução para a simulação de novas arquiteturas, focando em processamento em memória, através da tradução de código em linguagem x86 para instruções definidas pelo pesquisador durante a geração de traços. Utilizando esta técnica, os pesquisadores podem escrever códigos em alto nível a partir de nossa biblioteca Intrinsics-HMC, tais códigos são compiláveis e executáveis em arquiteturas tradicionais x86. Considerando o Simulator of Non-Uniform Cache Architectures (SiNUCA), utilizamos o SiNUCA-Tracer para efetuarmos a tradução das funções HMC para instruções HMC interpretáveis pelo simulador, fornecendo, assim, uma solução prática para os projetistas de novas arquiteturas de Processor-in-Memory (PIM).**

I. Introdução

Seguindo o lançamento de memórias 3D [1], surgiram propostas de arquiteturas fora dos padrões clássicos, como as iniciativas de Processor-in-Memory (PIM) [2], [3], que permitem a melhoria do desempenho de processadores e memórias. Dentre estas tecnologias inovadoras, gostaríamos de ressaltar o Hybrid Memory Cube (HMC) [4], que é um novo conceito de memória e processamento em memória [5]. Enquanto memória, substituindo as memórias DDR-3, por exemplo, o HMC provê um grande paralelismo entre bancos Dynamic Random Access Memory (DRAM), garantindo uma baixa latência média durante alta pressão na memória. Por outro lado, o HMC possui também capacidade de efetuar processamento, podendo mitigar a latência da busca e transmissão de dados entre a memória e o processador, já que o processamento é feito no mesmo *chip* que a memória. Desta forma, uma grande quantidade de estudos estão surgindo em busca de avaliar o desempenho e o consumo de energia do HMC [5], [6], [7].

Porém, mesmo que memórias HMC já sejam comercializadas e tenham uma especificação bastante completa, tais dispositivos são extremamente caros e estão fora de alcance de muitos pesquisadores. Além disso, tais memórias são de pouca utilidade para pesquisas arquiteturais que visem propostas de novas organizações, uma vez que não se pode experimentar novas mudanças na arquitetura ou organização, de forma a avaliar e entender seu impacto.

De forma geral, os projetistas de processadores e pesquisadores dependem de simuladores para avaliar o desempenho de novas organizações e novos componentes arquiteturais, porém, tais simuladores precisam receber programas completos ou traços de execução em formatos específicos para terem sua execução simulada por tais ferramentas. Nesse sentido, um dos principais problemas encontrados é a geração destes traços. Muitas vezes os projetistas precisam escrever códigos em linguagem de montagem (código *assembly*) ao se propor uma nova Instruction Set Architecture (ISA), o que é o caso das novas instruções HMC. Tal tarefa demanda tempo e é bastante propensa a erros, devido a grande quantidade de detalhes envolvida no processo de escrita de código em baixo nível.

Nesse contexto, o objetivo principal deste trabalho é, a partir de um código fonte de alto nível (em linguagem C ou C++), permitir a geração de traços utilizando instruções específicas do HMC para que estas possam ser simuladas e estimadas por projetistas.

Para tal, foi desenvolvida uma biblioteca chamada Intrinsics-HMC, baseada na especificação HMC-2.1 [4], que dispõe de funções que emulam o comportamento das instruções HMC, utilizando apenas o conjunto x86. Dessa forma, podemos escrever um programa completo em linguagem C ou C++ utilizando tais funções. Este programa poderá ser linkado, compilado e executado normalmente para garantir seu correto funcionamento. Entretanto, durante a geração de traços para serem utilizados no simulador, o gerador de traço irá identificar as funções HMC disponíveis na biblioteca e convertê-las para instruções HMC simuláveis. Neste trabalho iremos utilizar o Simulator of Non-Uniform Cache Architectures (SiNUCA) [8], [9]. Para auxiliar na geração dos traços, é utilizado o instrumentador binário Pin da Intel em conjunto com a ferramenta SiNUCA-Tracer que é capaz de instrumentar o código binário de tal modo a extrair o traço de simulação que representa a completa execução de uma determinada aplicação.

II. Conceitos Gerais

Nesta seção, serão introduzidos os conceitos básicos sobre o Hybrid Memory Cube (HMC) e sobre os simuladores *trace-driven*, abordando especificamente o Simulator of Non-Uniform Cache Architectures (SiNUCA).

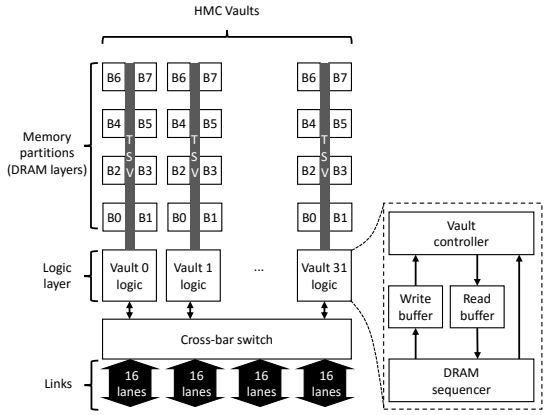

Figura 1. Diagrama de blocos do HMC com 32 *vaults* com 8 bancos cada. Adaptado de [10].

A. Hybrid Memory Cube – HMC

O HMC é um dispositivo de memória composto de até 8 camadas de Dynamic Random Access Memory (DRAM) empilhadas, integradas à base que é a camada lógica, conforme ilustrado na figura 1. O HMC é particionado em 32 *vaults*, sendo que, cada *vault* tem um controlador de memória dedicado e é composto de até 16 bancos (distribuídos nas 8 camadas DRAM) que são conectados entre si e com o controlador de memória através de Through-Silicon Vias (TSVs) [1]. O HMC é capaz de esconder sua latência interna de acesso a DRAM devido ao seu alto paralelismo de acesso aos dados [10], [5], [11].

Teoricamente, o HMC pode buscar dados de 32 bancos diferentes no mesmo ciclo em paralelo, sendo um por *vault*, e copiá-los para os *buffers* internos de leitura, permitindo uma largura de banda de até 320 GB/s [5]. Diferente das memórias Double Data Rate (DDR) 3, que transmitem 64 bits por canal, o HMC utiliza serialização para transmitir dados por 16 *full-duplex lanes* por *link*. Estes *links* podem alcançar altas frequências com menos interferência durante as transmissões, com foco na transmissão serial [10], [12].

O HMC trabalha com requisições simples enviadas pelo processador (leitura, escrita, instrução), onde os sinais internos da DRAM são gerados pela camada lógica dos *vaults*. Tais requisições são transmitidas utilizando um protocolo de pacotes, tal que, cada pacote é dividido em *flits* os quais tem capacidade para alocar no máximo 128 bits de dados e, como o HMC suporta requisições de leitura e escrita de 8 bytes até 128 bytes, são necessários no mínimo 2 e no máximo 17 *flits* para transmissão das instruções.

Durante a execução de instruções em memória, ou seja Processor-in-Memory (PIM), o processador deve tratar as instruções HMC como instruções de memória comum, da mesma forma que leituras ou escritas são manipuladas. Ou seja, o processador deverá efetuar a busca, decodificação, execução do cálculo de endereço e enviar a instrução para o HMC. Tal instrução poderá ter um campo adicional que diz respeito ao imediato a ser utilizado na operação. Quando a instrução chega no HMC, a mesma é encaminhada para o *vault* responsável pelo endereço de memória indicado. A camada lógica do *vault* deverá interpretar cada instrução, procedendo a busca dos dados e posterior operação sobre os dados buscados. Dependendo da instrução HMC, o valor deverá ser atualizado na mesma posição de memória da busca de dados, ou ainda os dados resultantes da operação deverão ser enviados ao processador como resposta à instrução enviada.

Trabalhos que avaliam o HMC mencionam que a arquitetura do HMC pode resultar em uma economia de 70% de energia comparado com a memória DDR3-1333 e, teoricamente, aumento da velocidade do sistema em 15 vezes [10], [5], [11], contudo, não está claro se todas as aplicações podem ser beneficiadas pelas atuais instruções do HMC. Dessa forma, estudos sobre extensões no conjunto de instruções, ou ainda, que avaliem a inclusão de novos componente arquiteturais, são de grande importância. Para tais avaliações, grande parte dos pesquisadores utilizam simuladores.

B. Simulator of Non-Uniform Cache Architectures – SiNUCA

Ferramentas de simulação baseadas em traço representam a escolha de grande parte dos projetistas de arquitetura de computadores, pois não precisam executar as instruções da aplicação durante a simulação, apenas precisam considerar os detalhes comportamentais (algorítmico) e latências da microarquitetura. Assim, recebem como entrada traços que descrevem os passos de execução de uma determinada aplicação ou *benchmark*, que são instruções já organizadas de forma adequada para a interpretação destes, de forma que sejam estimados. Os traços podem ser gerados manualmente pelo pesquisador ou automaticamente por ferramentas específicas, como por exemplo, instrumentadores binários.

Assim, neste trabalho foi utilizado o simulador SiNUCA [8], [9], que foi desenvolvido para suprir a demanda por um simulador que, além do desempenho, também estimasse o consumo de energia dos componentes da arquitetura simulada. Foi concebido considerando que os simuladores já existentes possuem diversas limitações, tais como falta de detalhes sobre a memória principal (sinais DRAM), incapacidade de modelagem de memórias cache *multi-banked*, entre outros detalhes da microarquitetura. Assim, o SiNUCA recebe traços de execução de aplicações de interesse e, simulando os detalhes e latências microarquiteturais, estima a quantidade de ciclos decorridos na execução de cada instrução, dependendo das latências dos componentes internos ao processador e hierarquia de memória que foram estressados pelo conjunto de instruções da carga de trabalho.

O SiNUCA foi desenvolvido baseado na arquitetura x86 e simula a execução de aplicações *mono* e *multi-threaded*, formado de um número configurável de processadores de execução fora-de-ordem, modelando tecnologias como Non-Uniform Cache Architecture (NUCA), Non-Uniform Memory Access (NUMA), Network-on-Chip (NoC) e DDR.

Os traços de simulação usados pelo SiNUCA são divididos em três tipos: estático, dinâmico e de memória. Os traços estáticos consistem de instruções formadas por código *assembly*,

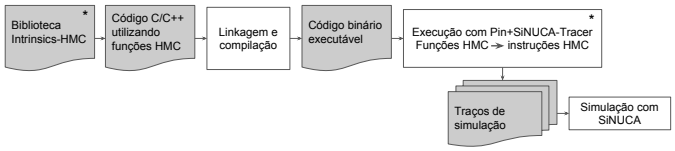

Figura 2. Sequência de passos para a geração dos traços de simulação (* indica nossas principais contribuições).

opcode do SiNUCA, tamanho, registradores de leitura e escrita e demais *flags*. As instruções são agrupadas em blocos básicos, indicados por "@". O traço dinâmico contém a sequência de chamadas dos blocos básicos (do traço estático) realizado pela aplicação durante a execução. Os traços de memória contém o endereço de memória e o tamanho da instrução para cada acesso de memória realizado pela aplicação.

Os traços utilizados pelo simulador SiNUCA, podem ser escritos manualmente respeitando os campos e formato de cada traço. Também podemos gerar os traços de simulação através de ferramentas de instrumentação de código binário de forma automatizada. Atualmente, o SiNUCA distribui junto ao seu código fonte, uma ferramenta de geração de traço, chamada SiNUCA-Tracer que deve ser utilizada junto a ferramenta de instrumentação binária Pin da Intel para gerar os traços de simulação automaticamente a partir da execução de uma aplicação real. Tal ferramenta será descrita na próxima seção.

III. Gerador de Traços

Para prover a simulação eficiente de processamento em memória, precisamos criar alternativas para a rápida e correta geração de traços de simulação. Nesse sentido, ao longo desta seção, serão abordadas a criação da biblioteca Intrinsics-HMC e a geração dos traços a partir do SiNUCA-Tracer que utiliza o instrumentador binário Pin da Intel, especificando de que forma estes dois conceitos se complementam.

A. Visão Geral da Proposta

A visão geral de nossa metodologia de geração de traços de simulação para instruções HMC é ilustrada na figura 2, na qual, inicialmente, apresentamos a biblioteca Intrinsics-HMC que estamos propondo, com as funções que descrevem o comportamento das instruções do HMC [4]. Estas funções são chamadas em um programa em C ou C++, de forma que este possa ser compilado corretamente, gerando o código binário. Tal aplicação é executada utilizando o SiNUCA-Tracer (também parte de nossa proposta) com o instrumentador Pin, para que seja possível instrumentar o código e gerar os traços de execução. Durante a geração dos traços todas as chamadas às funções disponíveis na biblioteca Intrinsics-HMC serão convertidas em instruções HMC simuláveis que servirão de entrada no simulador SiNUCA, que pode avaliar o desempenho computacional da arquitetura.

B. Biblioteca Intrinsics-HMC

Os dispositivos HMC possuem uma camada lógica que, segundo sua especificação, é capaz de executar operações aritméticas e binárias atômicas. Por isso, neste trabalho foram desenvolvidas funções para a criação da biblioteca Intrinsics-HMC, as quais reproduzem o comportamento das instruções do HMC. Abaixo, estão listados os tipos de funções que foram implementados com a descrição de suas funcionalidades, sendo que, neste caso, as instruções de leitura e escrita não foram implementadas nesta biblioteca, pois, terão seus traços gerados a partir das instruções tradicionais (leitura/escrita) normalmente decodificadas e executadas pelo processador da máquina.

Aritmética: Funções de soma de operandos de memória com imediatos e de incremento em uma unidade em um operando (endereço) de memória específico. No primeiro caso, os operandos podem ter tamanho de 8 bytes (com a possibilidade de processar dois valores imediatos em paralelo) ou 16 bytes (operando sobre apenas um valor imediato).

Binárias: Funções para escrever bits específicos do operando imediato nas mesmas posições do operando de memória, selecionando-os através de uma máscara de bits e para comparar os operandos de memória e imediato e, caso forem iguais, o operando de memória é retornado para o processador que enviou a instrução.

Booleanas: Funções que executam operações de *AND*, *NAND*, *OR*, *NOR* e *XOR* entre os operandos de memória e imediato.

Comparação: Funções de comparação entre os operandos de memória e imediato para que, dependendo da funcionalidade desejada, seja escrito o maior ou o menor dentre os dois na memória ou para verificar se são iguais entre si ou iguais a zero.

Para implementar as funções da biblioteca Intrinsics-HMC que reproduzem as instruções HMC, algumas adaptações e padronizações foram consideradas, desde o tipo dos dados até a estrutura das funções. Assim, a biblioteca foi escrita em C++ e os dados foram padronizados em quatro tipos específicos, descritos na tabela I.

Tabela I
Padronização dos tipos de dados para a biblioteca Intrinsics-HMC.

Tipo de Dado	Descrição
__h16s1	Tipo de dado equivalente a um *unsigned short*
__h64l1	Tipo de dado equivalente a um *unsigned long*
__h64l2	Tipo de dado equivalente a um vetor de 2 *unsigned long*
__h128l1	Tipo de dado equivalente a um *unsigned long long*

Partindo dos tipos de dados, as funções para o HMC foram desenvolvidas inspiradas nos moldes das funções *intrinsics* da Intel, que são funções em C que, quando chamadas durante a execução do código, direcionam a execução para trechos específicos de código em *assembly* x86 direto no compilador, sendo que são trechos otimizados e bastante específicos, tanto que o *assembly* gerado durante a compilação de códigos com a mesma funcionalidade, mas sem a chamada das *intrinsics*, dificilmente será igual [13]. Assim, permitem a fina otimização do código fonte através das chamadas de funções obtendo-se, muitas vezes, a vetorização do código, induzindo à execução de instruções Single Instruction Multiple Data (SIMD). Con-

tudo, nem todo processador permite a execução das funções *intrinsics* e alguns processadores permitem um número limitado delas, dependendo do modelo de microarquitetura presente em cada processador. Normalmente as *intrinsics* são utilizadas por programadores experientes que buscam obter o máximo que o conjunto de instruções da arquitetura tem a oferecer, sem a necessidade de depender de otimizações do compilador para obter tais melhorias.

Código 1. Exemplo de chamada de função da biblioteca Intrinsics-HMC.

```
1  void _hmc64_saddimm_d(uint64_t *mem_op, uint64_t *imm_op)
       {
2      mem_op[0] = mem_op[0] + imm_op[0];
3      mem_op[1] = mem_op[1] + imm_op[1];
4  }
```

Uma vez pronta a biblioteca proposta, basta que o programador desenvolva um programa em linguagem C ou C++ que realize chamadas das funções referentes à biblioteca Intrinsics-HMC. Sendo assim, o código 2 apresenta um exemplo de trecho de código que realiza a chamada da função ***_hmc64_saddimm_d(*mem_op, *imm_op)***, correspondente ao código 1. Tal função efetua a soma paralela dos operandos de memória com os valores imediatos. Uma vez que o código exemplo esteja pronto, este deve ser compilado, para que se possa gerar o código binário. O código binário será utilizado como entrada para o gerador de traços SiNUCA-Tracer. Ressaltando que o programador poderá efetuar testes e depuração do código normalmente, para garantir a corretude do programa antes da geração dos traços.

Código 2. Exemplo de chamada de função.

```
1  hmc_lib.hpp
2
3  int main(){
4      uint64_t imm_op[2];
5      imm_op[0] = rand();
6      imm_op[1] = rand();
7      _hmc64_saddimm_d(&mem_op, &imm_op);
8  }
```

C. Detector de funções HMC

Após gerado e depurado o código binário utilizando a biblioteca Intrinsics-HMC, este servirá de entrada para o gerador de traços SiNUCA-Tracer. O gerador de traços irá, primeiramente, instrumentar o código binário com auxilio da ferramenta de instrumentação Pin. Após a etapa de instrumentação, o código será executado, onde a geração dos traços irá acontecer, considerando o comportamento real da execução do programa. Em nosso caso, o SiNUCA-Tracer deverá identificar todas as funções provenientes da biblioteca Intrinsics-HMC, e substituí-las por simples instruções HMC, ou seja, a cada chamada a função será substituída por uma execução de instrução HMC válida para o simulador. A seguir veremos mais detalhes sobre a detecção e geração dos traços.

A ferramenta Pin é desenvolvida pela Intel e foi integrada ao SiNUCA para realizar a instrumentação e análise de código, pois permite a criação de *Pintools*, que são programas desenvolvidos a partir de rotinas disponibilizadas pela própria ferramenta da Intel que podem ser integradas a um programa, permitindo determinar quais trechos do código serão analisados e inspecionados e, na sequência, qual o tipo de análise/operação deverá ser realizada nestes trechos, por exemplo, questões de memória, execução, custo, desempenho e entre outros fatores podem ser o foco de tais inspeções.

A ferramenta Pin deve ser aplicada em programas já compilados (binários), sem a necessidade de modificações no código fonte. As ferramentas de análise devem ser desenvolvidas em C ou C++ para fazer a análise durante a execução do programa, por isso é conhecido por ser um instrumentador *just-in-time compiler*. Logo, a instrumentação deve ser realizada antes da análise, priorizando-se os trechos mais significativos do código, para evitar queda de desempenho ou demora na execução.

Para realizar a análise do código, deve ser especificada a forma como a *Pintool* deve percorrer (inspecionar) o código. A forma mais abrangente é abrir a imagem do código binário para iniciar a inspeção, a qual pode ser feita de uma das formas citadas abaixo:

Por instrução: a menor unidade da imagem, percorre cada uma das instruções na sequência em que são executadas;

Por blocos básicos: são criados durante a execução, analisando o fluxo do programa, sempre que haja uma entrada e uma saída específicas para um trecho de código. Caso dentro deste trecho haja alguma condição de decisão, podem ser criados mais de um bloco básico para ele;

Por traços: um traço tem início em um salto tomado e termina no fim de um salto incondicional, isto inclui a chamada e retorno de funções. Normalmente são quebrados em rotinas, blocos básicos ou instruções;

Por rotina: cada rotina da imagem pode ser percorrida separadamente e, caso seja necessário analisá-la, ela pode ser quebrada em blocos básicos ou instruções;

Por seções: cada seção da imagem pode ser percorrida separadamente e, caso seja necessário analisá-la, ela pode ser quebrada em rotinas, traços e em suas demais subseções.

Para a geração da ferramenta de análise SiNUCA-Tracer, foi utilizada três tipos de instrumentação: 1) por instrução, para geração do traço estático e também inserção de *hooks* para geração de traços de memória caso a instrução efetue leitura ou escrita de memória; 2) por blocos básicos, responsável pela criação dos traços dinâmicos, que descreve todos os blocos básicos executados pelo programa; 3) por rotina que será responsável, entre outras coisas, em traduzir as funções da biblioteca Intrinsics-HMC, para instruções HMC.

Assim, inicialmente, o gerador de traços SiNUCA-Tracer abre a imagem do programa e a percorre por traços. Em seguida, para cada traço, são identificadas todas as rotinas, estas são registradas no arquivo de saída de traços estáticos junto com as instruções referentes a cada bloco básico contido nestas rotinas. Cada rotina pode conter vários blocos básicos, de forma que, cada bloco básico pode conter várias instruções. Durante esta inspeção de blocos e instruções, são identificadas as instruções de memória, que são registradas no arquivo de saída de memória. Por fim, os blocos básicos são escritos no arquivo de saída dinâmico seguindo a ordem em que foram executados.

Código 3. Código referente à geração dos traços nos arquivos de saída.

```
1  trace_instruction(TRACE trace){
2      register_routine_name(static_trace);
3      for(all basic_blocks in routine){
4          register_basicblock_id(static_trace);
5          for(all instructions in basic_block){
6              if(!HMC_Intrinsics){
7                  if(mem_operation){
8                      register_mem_op(mem_trace);
9                  }
10             }
11         }
12         if(!HMC_Intrinsics){
13             register_dyn_block(dyn_trace);
14         }
15     }
16 }
```

Durante a execução, quando uma função da biblioteca Intrinsics-HMC é chamada, ela é identificada no SiNUCA-Tracer e as saídas nos traços dinâmicos e de memória que ela iria gerar são suprimidos, como apresentado no código 3. Neste caso, deseja-se simular traços como se eles tivessem sido executados por uma máquina com uma arquitetura que emprega o HMC, por isso, o SiNUCA-Tracer suprime momentaneamente a geração dos traços x86 e insere traços artificiais que utilizam instruções HMC como mostrado no código 4.

Código 4. Código referente à inserção dos traços artificiais das funções Intrinsics-HMC.

```
1  routine = FindByName(HMC_Intrinsics name);
2  if (Valid(routine){
3      register_routine_name(static_trace);
4      register_basicblock_id(static_trace);
5      register_mem_op1(mem_trace);
6      register_mem_op2(mem_trace);
7      register_mem_opn(mem_trace);
8      register_dyn_block(dyn_trace);
9  }
```

Uma função da biblioteca Intrinsics-HMC é identificada como uma rotina no código binário pelo Pin, entretanto, durante toda a execução da rotina, o SiNUCA-Tracer irá suprimir temporariamente a geração de traços.

Para que possamos substituir a chamada à função pela instrução HMC simulável, acrescentamos no traço estático blocos básicos novos, contendo apenas a instrução HMC referente à cada função fornecida pela biblioteca. Durante a geração do traço dinâmico, substituímos cada chamada à função HMC por uma chamada a um bloco básico contendo a instrução HMC referente. O traço de memória, por sua vez, irá conter os endereços específicos de leitura e eventual escrita na memória ocasionada pela instrução HMC. Tais endereços são extraídos dos parâmetros da rotina pelo gerador de traços.

Ao final da execução, os arquivos de traços irão conter o registro das rotinas e seus blocos básicos, identificando cada uma das instruções em cada bloco, as operações de memória realizadas em cada bloco e o fluxo de execução do programa, com a sequencia das chamadas destes blocos. Estes traços podem então servir como entrada para o SiNUCA, para que este possa simular a execução do programa pois no lugar das funções da Intrinsics-HMC haverá instruções HMC.

IV. Resultados Obtidos

Nesta seção iremos apresentar resultados de validação do gerador de traços, mostrando o caminho entre o código de alto nível até a geração do traço simulável.

Seguindo os passos descritos nas seções anteriores para a geração dos traços, foi escrito um programa em linguagem C que realiza a chamada de duas das funções da biblioteca Intrinsics-HMC: ***_hmc64_saddimm_d(*mem_op, *imm_op)*** e ***_hmc64_incr_s(*mem_op)*** tal que a primeira, citada na seção anterior, soma paralelamente dois operandos de memória com dois imediatos e a segunda incrementa o operando de memória em uma unidade.

Ao compilar este programa, é gerado seu código objeto que é executado com o gerador de traços SiNUCA-Tracer, obtendo-se, assim, os traços referentes ao programa escrito. Para obter o traço, durante a execução do programa com o SiNUCA-Tracer, a *Pintool* identifica no código binário as funções da biblioteca Intrinsics-HMC e as converte em um conjunto de instruções HMC, no formato requerido pelo simulador, escrevendo-os no arquivo de saída.

Na tabela II, são apresentados o código em *assembly* x86 e o traço estático (simulável pelo SiNUCA), ambos referentes ao programa exemplo. O código assembly x86, traz o código compilado utilizando as funções HMC. O traço estático já apresenta uma tradução das funções HMC para instruções HMC interpretáveis pelo SiNUCA. Comparando os resultados, segundo o número de instruções *assembly*, fica clara a completa tradução do código através de nossa proposta.

Atualmente, apenas as instruções previstas na especificação do HMC, estão disponíveis na biblioteca Intrinsics-HMC. Entretanto, tal biblioteca é de fácil extensão, possibilitando assim a criação de novas funções, referentes a instruções ainda inexistentes. Para isso, basta que seja alterada a biblioteca Intrinsics-HMC e a ferramenta SiNUCA-Tracer, para que seja feita a correta tradução da função para uma nova instrução suportada pelo simulador.

V. Trabalhos Relacionados

Nesta seção, analisaremos os trabalhos correlatos que tratam de simulação e geração de traços para o HMC.

O trabalho desenvolvido por [14] propõe um mecanismo para execução de instruções vetoriais dentro do HMC, chamado HMC Instruction Vector Extensions (HIVE). Este modelo faz uso do paralelismo que a arquitetura do HMC fornece e permite vetorizar as operações lógicas e aritméticas, favorecendo aplicações que fazem uso de grandes quantidades de dados. Contudo, os pesquisadores tiveram que gerar traços de simulação manualmente para avaliar o HIVE, devido a falta de emuladores e compiladores capazes de gerar códigos vetoriais de execução em memória.

Outro trabalho que faz uso do HMC realiza as execuções em um HMC real utilizando um FPGA que gera e envia requisições customizadas ao HMC [7], tendo como foco, avaliar e associar as variáveis de desempenho, temperatura, gasto de energia e latência de acesso. Mesmo de posse de um hardware real, percebemos que avaliação sobre projetos de novas instruções demandaria modificações no compilador que gera código HMC e também novas prototipações de hardware.

Por fim, o trabalho [15] desenvolveu o simulador *Cycle Accurate Parallel PIM Simulator* (CLAPPS) que pode modelar

Tabela II
COMPARAÇÃO DO RESULTADO DO *assembly* X86 E DO TRAÇO DE EXECUÇÃO REFERENTE ÀS FUNÇÕES DA BIBLIOTECA INTRINSICS-HMC.

Chamada das funções Intrinsics-HMC

```
#include "hmc_lib.hpp"

int main(){
  uint64_t imm_op[2];
  uint64_t mem_op[2];

  scanf("%lu", &imm_op[0]);
  scanf("%lu", &imm_op[1]);

  srand (imm_op[0]);
  mem_op[0] = rand() % imm_op[1];
  mem_op[1] = rand() % imm_op[1];

  _hmc64_saddimm_d(mem_op, imm_op);
  _hmc64_incr_s(mem_op[0]);
}
```

Código Assembly x86

```
_hmc64_saddimm_d:
  mov -0x8(%rbp),%rax
  mov (%rax),%rdx
  mov -0x10(%rbp),%rax
  mov (%rax),%rax
  add %rax,%rdx
  mov -0x8(%rbp),%rax
  mov %rdx,(%rax)
  mov -0x8(%rbp),%rax
  add $0x8,%rax
  mov -0x8(%rbp),%rdx
  add $0x8,%rdx
  mov (%rdx),%rcx
  mov -0x10(%rbp),%rdx
  add $0x8,%rdx
  mov (%rdx),%rdx
  add %rcx,%rdx

_hmc64_incr_s:
  mov -0x8(%rbp),%rax
  mov (%rax),%rax
  lea 0x1(%rax),%rdx
  mov -0x8(%rbp),%rax
  mov %rdx,(%rax)
```

Traço estático resultante

```
#_hmc64_saddimm_d
@1
HMC_ADD 12 648 8 2 10 6 2 10 25 0 0 0 0 0 3 0 0 0
#_hmc64_incr_s
@2
HMC_INC 12 672 8 2 12 8 2 12 15 0 0 0 0 0 3 0 0 0
```

arquiteturas PIM customizadas para simulação. Desta forma, os autores criaram uma arquitetura similar ao HMC, baseando-se na especificação 2.0 [10], tendo desenvolvido todas as estruturas e mecanismos descritos na arquitetura e implementado todas as instruções especificadas, a fim de validar e comparar os resultados de largura de banda obtidos com os resultados observados na especificação. Entretanto, notamos novamente o problema da geração de código de maneira eficiente para ser utilizado pelo simulador CLAPPS o qual foi planejado para trabalhar com código binário.

VI. CONCLUSÕES E TRABALHOS FUTURO

Sistemas com capacidade de processamento em memória estão se tornando realidade, e dessa forma, projetistas e pesquisadores de novas arquiteturas precisam de ferramentas para analisar a proposta de novos módulos arquiteturais que possam alavancar o desempenho desses sistemas.

Neste artigo, apresentamos uma metodologia que auxilia a simulação de arquiteturas emergentes e novas instruções. Através da biblioteca Intrinsics-HMC apresentada e das modificações do gerador de traços SiNUCA-Tracer, podemos escrever códigos completos em linguagens de alto nível utilizando funções que emulam novas instruções de processamento em memória. Após avaliada a corretude do código, possibilitamos a geração de traços simuláveis através do SiNUCA-Tracer que traduzem as funções comportamentais HMC em instruções simuláveis HMC. Nesse artigo focamos na proposta de uma solução utilizando o simulador SiNUCA como caso de exemplo, porém outros simuladores podem ser beneficiados pela metodologia apresentada. Fora isto, o gerador de traços SiNUCA-Tracer também pode ser modificado, permitindo simular arquiteturas pouco acessíveis ou inexistentes.

Como trabalho futuro, consideramos revalidar trabalhos que efetuaram a avaliação de propostas de novas arquiteturas PIM os quais utilizaram traços de simulação escritos manualmente em linguagem de máquina.

REFERÊNCIAS

[1] J. V. Olmen, A. Mercha, G. Katti *et al.*, "3D stacked IC demonstration using a through silicon via first approach," in *Int. Electron Devices Meeting*, 2008.

[2] D. Patterson, T. Anderson, N. Cardwell *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar. 1997.

[3] D. G. Elliott, M. Stumm, W. M. Snelgrove *et al.*, "Computational RAM: Implementing Processors in Memory," *Design and Test of Computers*, vol. 16, no. 1, pp. 32–41, Jan. 1999.

[4] Hybrid Memory Cube Consortium, "Hybrid memory cube specification 2.1," 2014, http://www.hybridmemorycube.org/.

[5] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Symp. on VLSI Technology*, 2012.

[6] K. Khalifa, H. Fawzy, S. El-Ashry, and K. Salah, "Memory controller architectures: A comparative study," in *Int. Design and Test Symp.*, 2013.

[7] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the characteristics of 3d-stacked memories: a case study for hybrid memory cube," *arXiv preprint arXiv:1706.02725*, 2017.

[8] M. Alves, "Increasing energy efficiency of processor caches via line usage predictors," Ph.D. dissertation, Universidade Federal do Rio Grande do Sul, May 2014.

[9] M. A. Z. Alves, M. Diener, F. B. Moreira *et al.*, "SiNUCA: a validated micro-architecture simulator," in *High Performance Computation Conf.*, 2015.

[10] Hybrid Memory Cube Consortium, "Hybrid memory cube specification rev. 2.0," 2013, http://www.hybridmemorycube.org/.

[11] J. Pawlowski, "Hybrid memory cube (hmc)," *Hot Chips*, vol. 23, 2011.

[12] T. Thanh-Hoang, A. Shambayati, C. Deutschbein, H. Hoffmann, and A. Chien, "Performance and energy limits of a processor-integrated fft accelerator," in *High Performance Extreme Computing Conf.*, 2014.

[13] I. Coorporation, "Intel 64 and ia-32 architectures optimization reference manual," 2009.

[14] M. A. Z. Alves, M. Diener, P. C. Santos, and L. Carro, "Large vector extensions inside the HMC," in *Conf. on Design, Automation & Test in Europe*, 2016.

[15] G. F. Oliveira, P. C. Santos, M. A. Z. Alves, and L. Carro, "A generic processing in memory cycle accurate simulator under hybrid memory cube architecture," in *Int. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation*, 2017.