UNIVERSIDADE FEDERAL DO PARANÁ

ALINE SANTANA CORDEIRO

INTRINSICS-HMC: GERAÇÃO AUTOMÁTICA DE TRAÇOS DE SIMULAÇÃO PARA ARQUITETURAS DE PROCESSAMENTO EM MEMÓRIA

CURITIBA

2017

ALINE SANTANA CORDEIRO

INTRINSICS-HMC: GERAÇÃO AUTOMÁTICA DE TRAÇOS DE SIMULAÇÃO PARA ARQUITETURAS DE PROCESSAMENTO EM MEMÓRIA

Trabalho de Graduação apresentado como requisito parcial à obtenção do grau de Bacharela em Ciência da Computação da Universidade Federal do Paraná.
Orientador: Prof. Dr. Marco Antonio Zanata Alves

CURITIBA

2017

ALINE SANTANA CORDEIRO

INTRINSICS-HMC: GERAÇÃO AUTOMÁTICA DE TRAÇOS DE SIMULAÇÃO PARA ARQUITETURAS DE PROCESSAMENTO EM MEMÓRIA

Trabalho de Graduação apresentado como requisito parcial à obtenção do grau de Bacharela em Ciência da Computação da Universidade Federal do Paraná, pela seguinte banca examinadora:

Prof. Dr. Marco Antonio Zanata Alves
Orientador - Departamento de Informática, UFPR

Prof. Dr. Carlos Alberto Maziero
Departamento de Informática, UFPR

Bacharel Diego Gomes Tomé
Departamento de Informática, UFPR

Curitiba, 7 de dezembro de 2017

Dedico este trabalho a meus pais, à Deyse e ao Rafael, os quais me incentivaram, apoiaram e permitiram que eu chegasse até esta etapa da minha vida e continuam me incentivando a seguir em frente.

AGRADECIMENTOS

Ao meu namorado, Rafael, quem me ajudou a não desistir da computação, pelo apoio, incentivo e compreensão.

Ao Prof. Dr. Roberto Hexsel, pelas incentivadoras aulas de Arquitetura e Organização de Computadores.

Ao Prof. Dr. Marco A. Zanata Alves, pela orientação, dedicação, disposição, incentivo e pelos votos de confiança.

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 ligadas 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. Neste trabalho 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 por quem realiza estas pesquisas durante a geração de traços. Utilizando nossa técnica, estas pessoas que pesquisam podem escrever códigos em alto nível utilizando nossa biblioteca Intrinsics-HMC, sendo que, tais códigos são compiláveis e executáveis em arquiteturas tradicionais x86. Considerando o Simulator of Non-Uniform Cache Architetctures (SiNUCA), nós utilizamos o SiNUCA-Tracer, para efetuar a tradução das funções HMC para instruções HMC interpretáveis pelo simulador, fornecendo, assim, uma solução prática para projetistas de novas arquiteturas de Processor-in-Memory (PIM). Os resultados obtidos utilizando a técnica proposta para geração automática de traços de simulação de *kernels* de aplicativos de banco de dados mostram a tradução e simulação corretas das novas instruções HMC usando o SiNUCA.

Palavras-chave: *Hybrid Memory Cube*, processamento-em-memória, SiNUCA, simulação.

ABSTRACT

PIM architectures, such as the HMC, are emerging nowadays as a solution for processing large amount of data directly inside the memory. In this area, several researchers are proposing and evaluating new instructions and new PIM architectures. For such evaluations, trace-driven simulators, such as the SiNUCA, are commonly used in order to model these non-existing systems. Such simulators provide fast prototyping of new architectures, while it requires the researcher to write simulation traces manually when evaluating new Instruction Set Architecture (ISA) proposals, which is an error prone task. In this work, we propose a methodology to fast generation of simulation traces focused on HMC architecture, which consists on a high-level Intrinsics-HMC library and a modification in a trace-generator tool from SiNUCA. Our proposal enables the researchers to write high level code in C/C++ languages using our library, which mimics the behavior of HMC instructions. These codes can be compiled and executed in traditional x86 architectures for verification. After ensure the code is correct and working, the user can use our modified version of SiNUCA-Tracer to translate HMC functions into HMC instructions known by the simulator, providing a convenient solution to generate traces and fast simulations of new PIM architectures. Results using the proposed technique to automatic generate simulation traces from database application kernels show the correct translation and simulation of new HMC instructions using SiNUCA.

Keywords: Hybrid Memory Cube, processor-in-memory, SiNUCA, simulation.

LISTA DE FIGURAS

SIGLAS

ASIC Application Specific Integrated Circuit.

CAS Column Address Strobe.

CLAPPS Cycle Accurate Parallel PIM Simulator.

CRC Cyclic Redundancy Check.

CWD Column Write Delay.

DBMS Database Management System.

DDR Double Data Rate.

DRAM Dynamic Random Access Memory.

FPGA Field-Programmable Gate Array.

HBM High Bandwidht Memory.

HIVE HMC Instruction Vector Extensions.

HMC Hybrid Memory Cube.

ISA Instruction Set Architecture.

McPAT Multi-core Power, Area, and Timing.

NoC Network-on-Chip.

NUCA Non-Uniform Cache Architecture.

NUMA Non-Uniform Memory Access.

PIM Processor-in-Memory.

RAS Row Address Strobe.

RP Row Precharge.

SIMD Single Instruction Multiple Data.

SiNUCA Simulator of Non-Uniform Cache Architectures.

TSV Through-Silicon Via.

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Tratando-se da área de arquitetura de computadores, houve avanços significativos desde a década de 1960, época em que a maior preocupação era desenvolver programas eficientes para os computadores, os quais eram extremamente restritos em relação à capacidade de memória e de processamento [Hennessy and Patterson, 2014]. Na mesma década, as estimativas de Gordon Moore, baseadas em poucas gerações de processadores, resultaram na Lei de Moore, a qual prevê que a cada 18 meses (mais tarde passa a se confirmar a cada 24 meses) o número de transistores deve dobrar, resultando em uma maior capacidade de processamento [Moore et al., 1975, Moore, 1998]. Inicialmente, essas melhorias tecnológicas se refletiam diretamente em ganhos de desempenho, contudo, observa-se que, hoje, isso já não é mais verdade devido a problemas como *memory-wall* [Wulf and McKee, 1995], *dark silicon* [Esmaeilzadeh et al., 2011], entre outros. Por isso, surgiram novas abordagens e propostas de arquiteturas Processor-in-Memory (PIM) [Patterson et al., 1997, Elliott et al., 1999], uma iniciativa proposta fora dos padrões clássicos, que começou a surgir em escala comercial nos últimos anos com o avanço das tecnologias 3D, permitindo a integração de dispositivos de DRAM com dispositivos lógicos. Tais arquiteturas visam a melhoria do desempenho dos processadores e memórias. Dentre estas tecnologias, o foco deste trabalho é o Hybrid Memory Cube (HMC) [Hybrid Memory Cube Consortium, 2014], e será discutido a seguir.

O HMC é um novo conceito de memória e processamento [Jeddeloh and Keeth, 2012]. 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 no mesmo dispositivo que a memória, tal capacidade pode mitigar a latência da busca e transmissão de dados entre a memória e o

processador. Desta forma, uma grande quantidade de estudos estão surgindo, tais estudos buscam avaliar o desempenho e o consumo de energia do HMC [Jeddeloh and Keeth, 2012, Khalifa et al., 2013, Hadidi et al., 2017].

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, a obtenção de tais memórias é 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, projetistas e pesquisadores de arquitetura de processadores dependem basicamente de simuladores para avaliar o desempenho de novas organizações e novos componentes arquiteturais. Simuladores *full-system* necessitam de binários compilados especificamente para a Instruction Set Architecture (ISA) simulada, precisando, assim, de um compilador para estas arquiteturas. De outro lado, simuladores dirigidos por traços são mais flexíveis, necessitando somente de traços de execução em formatos específicos e que utilize instruções reconhecidas pelo simulador, registrando a ordem dinâmica de execução. Nesse sentido, um dos principais problemas que tais cientistas enfrentam no contexto de simuladores é durante a geração de códigos binários ou traços de simulação, pois acabam por escrever manualmente um código ou traço em linguagem de montagem (assembly), uma atividade que, além de tomar um tempo considerável, é bastante propensa a erros, devido a grande quantidade de detalhes envolvida no processo de escrita de código em baixo nível. Essa dificuldade é típica em ambientes de grande inovação onde novas ISAs estão sendo avaliadas, antes mesmo que haja suporte para tais extensões arquiteturais.

Nesse contexto, o objetivo principal deste trabalho é propor um método que permita a geração automática de traços de simulação utilizando instruções HMC considerando a atual ISA e também futuras mudanças no conjunto de instruções. Além disso, tal método deverá permitir que os traços possam ser gerados a partir de um código fonte de alto nível (escrito em linguagem C ou C++), compilável e executável. Para tal, foi desenvolvida a biblioteca Intrinsics-HMC, baseada na especificação HMC-

2.1[Hybrid Memory Cube Consortium, 2014], que providencia um conjunto de funções que emulam o comportamento do HMC. Esta biblioteca foi desenvolvida em C++ e utiliza somente instruções x86, permitindo, assim, ser ligada, compilada e executada normalmente facilitando a validação e verificação de corretude do código escrito. Depois de validado, o pesquisador poderá utilizar o gerador de traços, que identifica as funções da biblioteca Intrinsics-HMC e as converte para instruções HMC identificáveis pelo simulador, gerando assim um traço de simulação que utiliza a nova ISA.

A fim de implementar a geração dos traços com as traduções das funções HMC, foi adaptado o gerador de traços do Simulator of Non-Uniform Cache Architectures (SiNUCA) [Alves, 2014], [Alves et al., 2015], SiNUCA-Tracer que utiliza o instrumentador binário Pin da Intel, 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.

O restante deste trabalho está organizado da seguinte forma: No capítulo 2, serão explicados alguns conceitos sobre o HMC, sobre o funcionamento básico do SiNUCA e sobre as intrinsics e a ferramenta Pin, ambas da Intel. No capítulo 3, apresenta-se a explicação sobre como os traços são gerados, abordando o desenvolvimento da biblioteca de funções do HMC e uma breve explicação da relação desta com as funções *intrinsics* da Intel, seguindo com a forma como as funções são detectadas pelo SiNUCA-Tracer para a geração de traços e a relação da ferramenta Pin da Intel com o gerador de traços do simulador. Por fim, no capítulo 4, serão mostrados alguns exemplos de uso e os resultados de validação serão analisados. O capítulo 5, traz a conclusão deste trabalho, finalizando com as propostas de pesquisas futuras.

CAPÍTULO 2

CONCEITOS GERAIS

2.1 HYBRID MEMORY CUBE

O HMC é um dispositivo de memória composto de até 8 camadas de DRAM empilhadas e integradas à uma camada lógica na base, conforme ilustrado na figura 2.1. O HMC é particionado em 32 *vaults*, sendo que, cada *vault* tem uma fila de operações e um controlador de memória dedicado, localizado na camada lógica, e até 16 bancos independentes de memória DRAM (distribuídos nas 8 camadas DRAM), na qual são conectados entre si e com o controlador de memória através de Through-Silicon Vias (TSVs) [Olmen et al., 2008].

Devido à esta tecnologia de integração 3D, é possível a integração de células de memória (DRAM) e processamento (transistores de alto desempenho) no mesmo dispositivo, mas diferentes *chips*. O HMC é capaz de esconder sua latência interna de acesso a DRAM devido ao seu alto paralelismo de acesso aos dados [Pawlowski, 2011, Jeddeloh and Keeth, 2012, Hybrid Memory Cube Consortium, 2013] e, 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 [Jeddeloh and Keeth, 2012].

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* (cada *lane* é um par de linhas de sinal invertido). Esse conjunto de lanes é chamado de *link*, os 4 *links* disponíveis no HMC não estão conectados a uma porção específica da memória, isto significa que qualquer *link* pode ser usado para transferir dados para qualquer *vault* do HMC. Com o uso desta técnica de comunicação resulta em uma área menor para fios (transmissão de sinais), além disto, estes *links* podem alcançar altas frequências (até 30 GHz) com menos interferência durante as transmissões e foco na transmissão serial

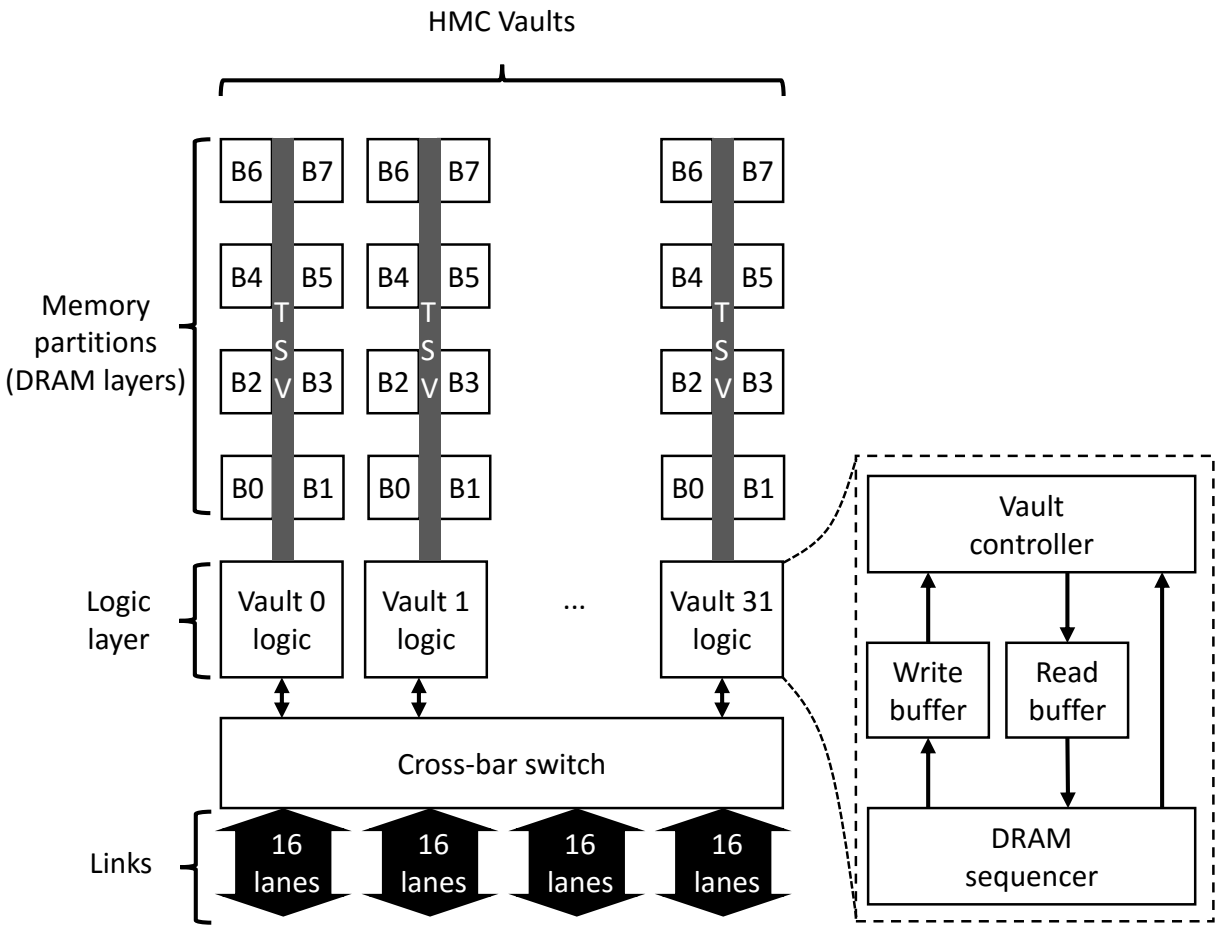

Figura 2.1: Diagrama de blocos do HMC com 32 *vaults* com 8 bancos cada. Adaptado de [Hybrid Memory Cube Consortium, 2013]

[Hybrid Memory Cube Consortium, 2013, Thanh-Hoang et al., 2014].

O HMC trabalha com requisições simples enviadas pelo processador (leitura, escrita, instrução), onde os sinais complexos da DRAM, como Row Precharge (RP), Column Address Strobe (CAS), Row Address Strobe (RAS), Column Write Delay (CWD) entre outros, são gerados por cada um dos controladores de memória de cada *vault*. Tais requisições são feitas pelo processador e transmitidas pela rede de HMCs utilizando um protocolo de pacotes, tal que, cada pacote tem 64 bits de cabeçalho e 64 bits de cauda para controlar os dados e fluxo destes na memória do HMC. Cada pacote é formado por *flits*, sendo que cada *flit* tem capacidade para alocar no máximo 128 bits de dados. Desta forma, um pacote nulo tem um *flit* de tamanho, apenas para alocar os dados referentes ao cabeçalho e cauda, enquanto que um pacote de dados pode ter entre 2 e 17 *flits* de

tamanho. Assim, para transmitir uma requisição, algumas instruções são inseridas em um *flit* e o HMC origem as empacota e serializa para transmiti-las através dos *links* seriais de alta vazão para o HMC destino, o qual recebe os dados e desempacota todas as instruções para serem executadas, retornando um pacote de resposta para o processador.

Para garantir a corretude dos dados dentro dos *flits*, antes de transmiti-los, o solicitante (ex. processador) faz o cálculo do Cyclic Redundancy Check (CRC) de todos bits de cada pacote, exceto dos bits referentes à cauda do pacote, e envia ao destinatário (ex. HMC) o CRC junto dos *flits* de dados. Assim que o destinatário receber os *flits*, serão recalculados o CRC destes e comparado com o CRC recebido. Caso os valores sejam iguais, o destinatário executa as instruções recebidas e retorna os resultados para o solicitante. Caso os CRCs não coincidam, os *flits* recebidos têm seus dados invalidados e são marcados como envenenados, e informado ao solicitante, para que este reenvie os dados corretos.

Independente do tamanho dos pacotes, o HMC suporta requisições de leitura ou escrita de 16 bytes até 256 bytes e instruções aritméticas e binárias que operam sobre 8 ou 16 bytes.

Durante a execução de instruções em memória, ou seja no PIM, o processador deve tratar as instruções HMC como operações 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 [Hybrid Memory Cube Consortium, 2013,

Jeddeloh and Keeth, 2012, Pawlowski, 2011], 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 das pessoas que trabalham com estas pesquisas utilizam simuladores, como os descritos na próxima seção.

2.2 SINUCA

Considerando que, durante a avaliação de novas arquiteturas, apenas a prototipação de *hardware*, simulação ou modelagem analítica suprem as demandas de pessoas que realizam estas pesquisas, podemos ponderar que a prototipação de *hardware* apresenta a melhor precisão, mas com um custo muito alto em termos de tempo de projeto e complexidade. Por outro lado, a formulação analítica apresenta maior flexibilidade representando o comportamento geral do sistema, porém devido a grande complexidade dos sistemas computacionais, formados de diversos níveis e componentes, tal formulação costuma ser difícil de ser deduzida e muitas vezes muito imprecisa [Jain, 1990].

Dessa maneira, as ferramentas de simulação representam a escolha de grande parte de projetistas de arquitetura de computadores. Nesse sentido, os simuladores baseados em traços possuem boa aceitação e não necessitam efetivamente executar as instruções da aplicação durante a simulação, necessitando apenas considerar os detalhes comportamentais (algorítmico) e latências da microarquitetura.

Esses simuladores utilizam traços de execução reais, ou seja, instruções já organizadas de forma adequada para interpretação destes. Os traços são formados por vários arquivos que contém o fluxo de instruções observado durante a execução do programa e podem ser gerados manualmente por quem conduz a pesquisa ou automaticamente por ferramentas de instrumentação binária e sistemas específicos para isto. Desta forma, estes simuladores não executam as instruções em si, mas recebem como entrada estes traços que descrevem os passos de execução de uma determinada aplicação ou *benchmark*, para que sejam interpretados e estimados.

Neste trabalho, foi utilizado o simulador SiNUCA [Alves, 2014], que é um simulador com precisão de ciclos e dirigido a traços, desenvolvido para suprir a demanda por um simulador que, além do desempenho, também estimasse o consumo de energia dos componentes da arquitetura simulada. Esse simulador 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. Por isso, é considerado bastante flexível e completo quando comparado com outros simuladores atuais. O consumo de energia pode ser estimado com a ferramenta Multi-core Power, Area, and Timing (McPAT). Para que os resultados do simulador sejam mais realistas, foram integrados componentes bastante usados nas arquiteturas atuais como pré-buscadores de dados, memórias cache não-bloqueantes, controlador detalhado de memória DRAM e preditores de salto.

Os traços de simulação usados pelo SiNUCA são divididos em três tipos, estático das instruções, dinâmico e de memória, ilustrados na tabela 2.2 e descritos abaixo:

Traço Estático: Consiste de instruções em linguagem de montagem divididas em blocos básicos, tal traço é carregado em memória no início da execução do simulador. Este traço é muito parecido com o código binário do programa. O início de cada bloco básico é marcado por um *"@"* e cada linha contida nestes blocos é uma instrução que contém o nome da operação em *assembly*, o *opcode* do tipo da instrução (código interno do simulador), o tamanho da instrução, o número de registradores de leitura utilizados

e a identificação de cada um destes, o número de registradores de escrita utilizados e a identificação de cada um destes, os registradores base e índice, que servem para diferenciar os registradores usados em memória dos registradores usados pelas instruções. Por fim, os indicadores de leitura e escrita de memória, descritores de tipo de saltos e indicadores de predicados e *prefetch*.

Traços Dinâmicos: Contém as chamadas dos blocos básicos efetuadas durante a execução da aplicação, os quais se referem aos blocos do traço estático. Assim, o traço dinâmico segue o fluxo de execução do programa pela *thread* e não contém informações sobre caminhos especulados erroneamente durante predições erradas de saltos, apresentando apenas os blocos efetivamente graduados pelo processador durante a execução. Além disso, este traço não sofre nenhuma interferência do sistema operacional ou demais aplicações em execução durante a geração dos traços. Em geral, utiliza-se apenas um traço estático e diversos traços dinâmicos, um para cada *thread* a ser simulada.

Traços de Memória: Apresenta informações sobre as operações de memória geradas durante a execução do programa, contendo endereço de memória, o tamanho da instrução e o bloco básico referente ao registro de memória. Este arquivo é gerado em paralelo ao traço dinâmico, de forma que, para cada bloco básico analisado, todas as informações sobre as operações de memória são registradas separadamente. Note que o traço de memória é fundamental, pois uma mesma instrução de leitura pode efetuar a carga de dados sobre um endereço diferente a cada vez que for executada. Assim, esse traço conterá todos os endereços lidos ou escritos na memória.

Os traços utilizado pelo simulador SiNUCA, podem ser escritos manualmente respeitando os campos e formato de cada traço. Também é possível 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 traços, 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 utilizando a ISA x86. Tal ferramenta será adaptada para suportar nosso mecanismo, e será descrita no próximo capítulo.

Traço Estático

```
1  #main
2  @1
3  MOV 1 0x95727 4 1 14 1 34 14 0 1 0 0 0 0 0
4  #main
5  @2
6  MOV 8 0x95717 3 1 14 1 65 14 0 1 0 0 0 0 0
7  ADD 1 0x95720 3 2 14 65 1 34 14 0 1 0 1 0 0 0
8  ADD 1 0x95723 4 1 14 1 34 14 0 1 0 1 0 0 0
9  CMP 1 0x95727 4 1 14 1 34 14 0 1 0 0 0 0 0
10 JBE 7 0x95731 2 2 35 34 1 35 0 0 0 0 0 1 0 0
```

Traço de Memória

```
1  R 4 0x1701448 1
2  #
3  R 4 0x1701448 2
4  R 4 0x1701452 2
5  W 4 0x1701452 2
6  R 4 0x1701448 2
7  W 4 0x1701448 2
8  R 4 0x1701448 2
9  #
10 R 4 0x1701448 2
11 R 4 0x1701452 2
12 W 4 0x1701452 2
13 R 4 0x1701448 2
14 W 4 0x1701448 2
15 R 4 0x1701448 2
```

Traço Dinâmico

```
1  1
2  2
3  2
```

Tabela 2.1: Formato dos traços de entrada do SiNUCA.

2.3 TRABALHOS RELACIONADOS

A chegada do HMC motivou várias pesquisas recentemente [Alves et al. 2016], [Hadidi et al. 2017], [Oliveira et al. 2017]. Junto com o HMC, surgiu o High Bandwidht Memory (HBM), uma memória de arquitetura 2.5D, semelhante ao HMC contudo, a camada lógica com controlador de memória e conjunto de instruções específico deve ser solicitado junto a quem as comercializa. Desta forma, podemos aplicar nosso método ao HBM também, porém, o HMC foi escolhido neste trabalho justamente por ter um conjunto de instruções mais simples e melhor documentado. Além disso, o HMC tem largura de banda de 320 GB/s, comparado com o HBM que alcança somente 128 GB/s em sua primeira versão (HBM1) e 256 GB/s com a segunda versão (HBM2) [Kim and Kim 2014].

A maior parte das pesquisas em PIM foca em melhorar a eficiência destas arquiteturas, utilizando simuladores para realizar a análise. Assim, o trabalho desenvolvido por [Alves et al., 2016] 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. Após as simulações, constatou-se que, comparado à memória do HMC e às operações de vetorização implementadas no processador, é obtido um ganho de desempenho de até 17,3× com 9,4× em média. 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 conjunto de HMCs reais utilizando um Field-Programmable Gate Array (FPGA) que gera e envia requisições customizadas para os HMCs [Hadidi et al., 2017], 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 a avaliação sobre projetos de novas instruções demandaria modificações no compilador, que gera código HMC, e também novas prototipações de hardware (Application Specific Integrated Circuit (ASIC)), o que é uma tarefa custosa.

Um dos primeiros simuladores de HMC surgiu baseado no simulador Gem5, o *SMC Simulation Environment* (SMCSim) [Azarkhish et al., 2016] é um módulo integrado ao simulador que visa melhorar a capacidade da camada lógica do HMC, modelando todas as camadas de software e hardware, além de considerar a sobrecarga causada pelo sistema operacional, coerência de cache e gerenciamento de memória. Contudo, o Gem5 não atinge as especificações de largura de banda do HMC.

No trabalho [Oliveira et al., 2017] foi desenvolvido o Cycle Accurate Parallel PIM Simulator (CLAPPS) que pode modelar arquiteturas PIM customizadas para simulação. Comparado ao simulador Gem5 com SMCSim, o CLAPPS foi desenvolvido para prover uma arquitetura PIM mais precisa. Desta forma, os autores criaram uma arquitetura similar ao HMC, baseando-se na especificação 2.0 [Hybrid Memory Cube Consortium, 2013], 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. Além disso, o CLAPPS simula somente a memória do HMC, precisando, assim, de um simulador de processador para obter resultados realísticos.

Por fim, o CasHMC [Jeon and Chung, 2017] é um simulador com precisão de ciclos que

foi desenvolvido em C++ e abrange os detalhes mais específicos da arquitetura do HMC, descritos na especificação. Diferente do SiNUCA, o CasHMC modela o HMC como uma memória simples, sem capacidade PIM. Por isso, a proposta deste trabalho é incrementar as características do SiNUCA, permitindo a geração automática de traços de simulação com instruções do HMC do código binário, enquanto os tamanhos das operações também são customizadas. Além disso, este método é adaptável a qualquer outro simulador e pode ser estendido para suportar diferentes conjuntos de instruções.

CAPÍTULO 3

MECANISMO DE GERAÇÃO DE TRAÇOS

3.1 VISÃO GERAL

A partir do momento que se deseja avaliar arquiteturas das quais não podem ser facilmente obtidas ou ainda inexistentes (novas arquiteturas ou organizações), os simuladores dirigidos por traços se tornam uma solução interessante, já que são capazes de realizar estimativas de desempenho e energia, entre outros fatores computacionais.

Para a geração de traços para tais simuladores, muitas vezes são utilizados emuladores ou instrumentadores binários. Os instrumentadores binários, como o Pin da Intel, possuem a vantagem da alta velocidade de execução e baixo sobrecusto. Entretanto, tanto as ferramentas de emulação quanto de instrumentação binária dependem da completa execução da aplicação em uma máquina real ou emulador para que seja gerado o traço de execução, impossibilitando ou dificultando, dessa forma, a geração de traços de aplicações que utilizem instruções inexistentes nas arquiteturas atuais.

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.

Desta forma, a visão geral desta metodologia de geração de traços de simulação para instruções HMC é ilustrada na figura 3.1, na qual, inicialmente, serão apresentadas a biblioteca Intrinsics-HMC com as funções que descrevem o comportamento das instruções do HMC [Hybrid Memory Cube Consortium, 2014]. Estas funções são chamadas em um programa em C++, de forma que este possa ser compilado corretamente, gerando o código binário. Tal aplicação é executada utilizando o SiNUCA-Tracer com o instrumentador Pin, para que seja possível instrumentar o código e gerar os traços de execução. Nossa

idéia é que durante a geração dos traços, todas as chamadas às funções disponíveis na biblioteca Intrinsics-HMC sejam convertidas em instruções HMC simuláveis que servirão de entrada no simulador SiNUCA, que pode avaliar o desempenho computacional da arquitetura.

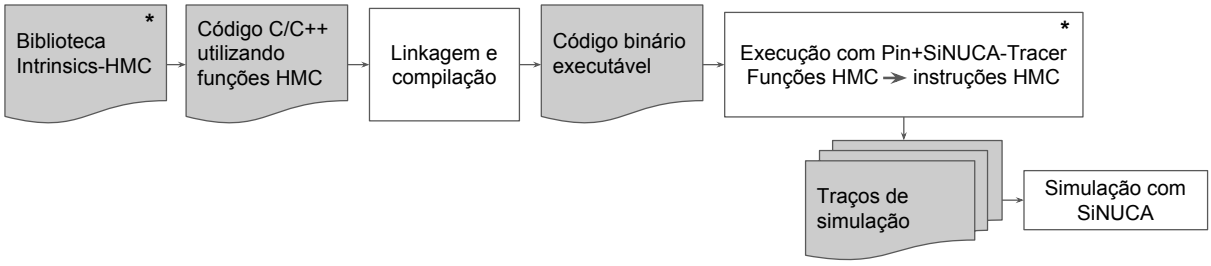

Figura 3.1: Sequência de passos para a geração dos traços de simulação (* indica nossas principais contribuições).

A biblioteca foi desenvolvida na linguagem C++, está disponível no Github[1] e pode ser baixada para ser utilizada em uma arquitetura x86 convencional, de forma que o código desenvolvido fazendo usa dessa biblioteca pode ser compilado, debugado e executado normalmente. Para a geração de traços, são necessários o SiNUCA-Tracer que depende das ferramentas Pin e PinPlay.

3.2 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.

[1] `https://github.com/AlineS/intrinsics-hmc`.

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.

Teoricamente, durante a execução de um programa em uma máquina que possui capacidade de executar instruções no HMC, vários *flits* são transmitidos, carregando as instruções que devem ser executadas e especificando os endereços de memória dos HMCs. Quando a instrução exige, os operandos imediatos também são transmitidos, para que esta possa ser executada corretamente. Dependendo do tipo da instrução, ela pode retornar ou não algum sinal ou valor para o processador solicitante. A figura 3.2 ilustra a forma como os operandos são alocados nos *flits* para serem enviados do processador ao HMC, de acordo com a especificação [Hybrid Memory Cube Consortium, 2014].

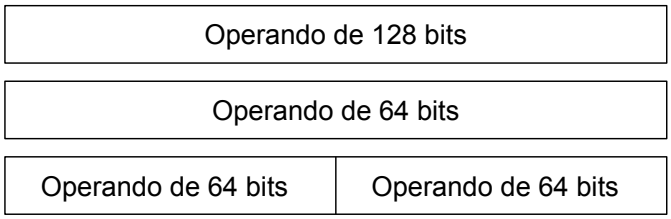

Figura 3.2: Ilustração de possíveis formas de carregar os operandos nos *flits*.

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, os dados foram padronizados em quatro tipos

específicos, descritos na tabela 3.2.

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

Tabela 3.1: Padronização dos tipos de dados para a biblioteca Intrinsics-HMC.

Partindo dos tipos de dados, as funções para o HMC foram desenvolvidas inspiradas nos moldes das funções *intrinsics* da Intel, que são funções escritas em C/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 [Coorporation, 2009]. Assim, permitem a fina otimização do código fonte através das chamadas de funções obtendo-se, muitas vezes, a vetorização do código através da execução de instruções Single Instruction Multiple Data (SIMD). Contudo, cada processador possui suporte variado para as instruções contidas dentro das funções *intrinsics*, dependendo do modelo de microarquitetura presente em cada processador. Normalmente as *intrinsics* são utilizadas por pessoas que já têm experiência em programação e 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.

Uma vez pronta a biblioteca proposta, basta que o programador desenvolva um programa em linguagem C++ e realize chamadas às funções referentes à biblioteca Intrinsics-HMC. Sendo assim, o código 3.1 apresenta um exemplo de trecho de código que realiza a chamada da função ***_hmc64_saddimm_d(*mem_op, *imm_op)***, correspondente ao código 3.2. 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 quem programa pode efetuar testes e depuração do código normalmente, para garantir a corretude do programa antes da geração dos traços.

```
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 }
```

Código 3.1: Exemplo de chamada de função.

```
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 }
```

Código 3.2: Exemplo de chamada de função da biblioteca Intrinsics-HMC.

3.3 FERRAMENTA-PIN: SINUCA-TRACER

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 única instrução HMC válida para o simulador. A seguir veremos mais detalhes sobre a detecção e geração dos traços.

Pin é uma ferramenta desenvolvida pela Intel para a instrumentação e análise de código e permite a criação de ferramentas-Pin de análise, ou *Pintools*, que são programas desenvolvidos a partir de rotinas disponibilizadas pela própria ferramenta da Intel que permitem determinar quais trechos do código serão analisados e inspecionados e, na sequência, qual o tipo de análise/operação deverá ser realizada nestes trechos.

A ferramenta-Pin deve ser aplicada em programas já compilados (binários), sem a necessidade de modificações no código fonte. Estas ferramentas de análise devem ser desenvolvidas em C ou C++ para que possam fazer a análise durante a execução do programa, por isso, a ferramenta Pin é conhecida 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. Neste trabalho, foi usada a versão 3.2 (revisão 81205) da ferramenta Pin.

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 utilizado três tipos de instrumentação: i) 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; ii) por blocos básicos, responsável pela criação dos traços dinâmicos, que descreve todos os blocos básicos executados pelo programa; iii) 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.

```
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 }
```

Código 3.3: Código referente à geração dos traços nos arquivos de saída.

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 3.3. 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 3.4.

```
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  }
```

Código 3.4: Código referente à inserção dos traços artificiais das funções Intrinsics-HMC.

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 e escritos no traço de memória.

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.

Resumindo, o SiNUCA-Tracer identifica todas as funções HMC, suprime a geração de código x86 destas, substituindo-as por instruções HMC, como se os traços estivessem sendo simulados por um computador com uma arquitetura com suporte para o conjunto de instruções do HMC, como ilustra a figura 3.5.

```
1  lw
2  add
3  or
4  sw
5  CALL hmc_function ...    ;;;;
6  add                        ;; a chamada x86 sera interceptada e,
7  and                        ;; no lugar, sera inserida uma unica
8  sw                         ;; instrucao atomica HMC:
9  lw                         ;; hmc_atomic_instruction
10 RET hmc_function ...     ;;;;
11 nop
12 beq
```

Código 3.5: Código ilustrando a substituição da função x86 da biblioteca Intrinsics-HMC por uma instrução HMC atômica.

É importante perceber que o compilador deve ter gerado dependências reais entre os registradores externos e internos à cada função da biblioteca Intrinsics-HMC. Depois da tradução destas funções para instruções HMC, tais dependências devem ser mantidas. Durante a instrumentação do binário são analisadas todas as dependências de entrada e saída. Dependências de entrada são todos os registradores que são lidos dentro da função,

mas foram escritos antes da chamada da mesma. Dependências de saída correspondem a todos os registradores escritos dentro da função e que podem ser lidos depois do retorno da função. Para mantermos essa dependencia, cada instrução HMC contém registradores de leitura, ou seja, uma lista com todos os registradores de entrada, e registradores de escrita, que é uma lista com todos os registradores de saída.

A figura 3.3 ilustra a análise realizada para manter as dependências corretas durante cada tradução de funções HMC. Neste exemplo, observa-se o código do traço estático para a função de soma *add* (linhas 1∼27). As linhas 28∼30 contém as instruções HMC traduzidas para executar a operação *add*. Nesta figura são mostrados os registradores de leitura (negrito com fundo cinza), escrita (negrito), base e deslocamento (sublinhado) para cada instrução. É possível observar que os registradores 5 e 6 (linha 3) representam as primeiras leituras antes de qualquer escrita, isso significa que estes registradores são dependências reais, e devem estar nas listas de dependências da instrução HMC. O mesmo não ocorre para o registrador de leitura número 10 (linha 10), pois ele foi escrito anteriormente pela instrução da linha 9, então ele não representa uma dependência real para a instrução HMC. Desta forma, quando gerado o traço de simulação para o HMC, o mecanismo utilizado analisa todos os registradores de entrada e saída (representados em círculos na figura) da função HMC e as utilizamos como registradores de leitura e escrita, respectivamente, nas instruções HMC simuláveis.

Durante a execução com o SiNUCA-Tracer, é necessário registrar o traço de memória de cada instrução de memória, para tal, são usados os endereços de memória reais, seus tamanhos, e o número do bloco básico no qual eles estão contidos. Para registrar as instruções de memória x86 (ins), são utilizada as rotinas Pin *INS_IsMemoryRead(ins)*, *INS_HasMemoryRead2(ins)* e *INS_IsMemoryWrite(ins)*, de forma que, para conseguir o endereço efetivo do operando de memória de uma função HMC, é utilizado, como parâmetro, o comando *IARG_FUNCARG_ENTRYPOINT_VALUE, 0* que, durante a execução, assume como valor o endereço de memória do parâmetro real passado para a função HMC, sendo assim, registrado no traço de memória.

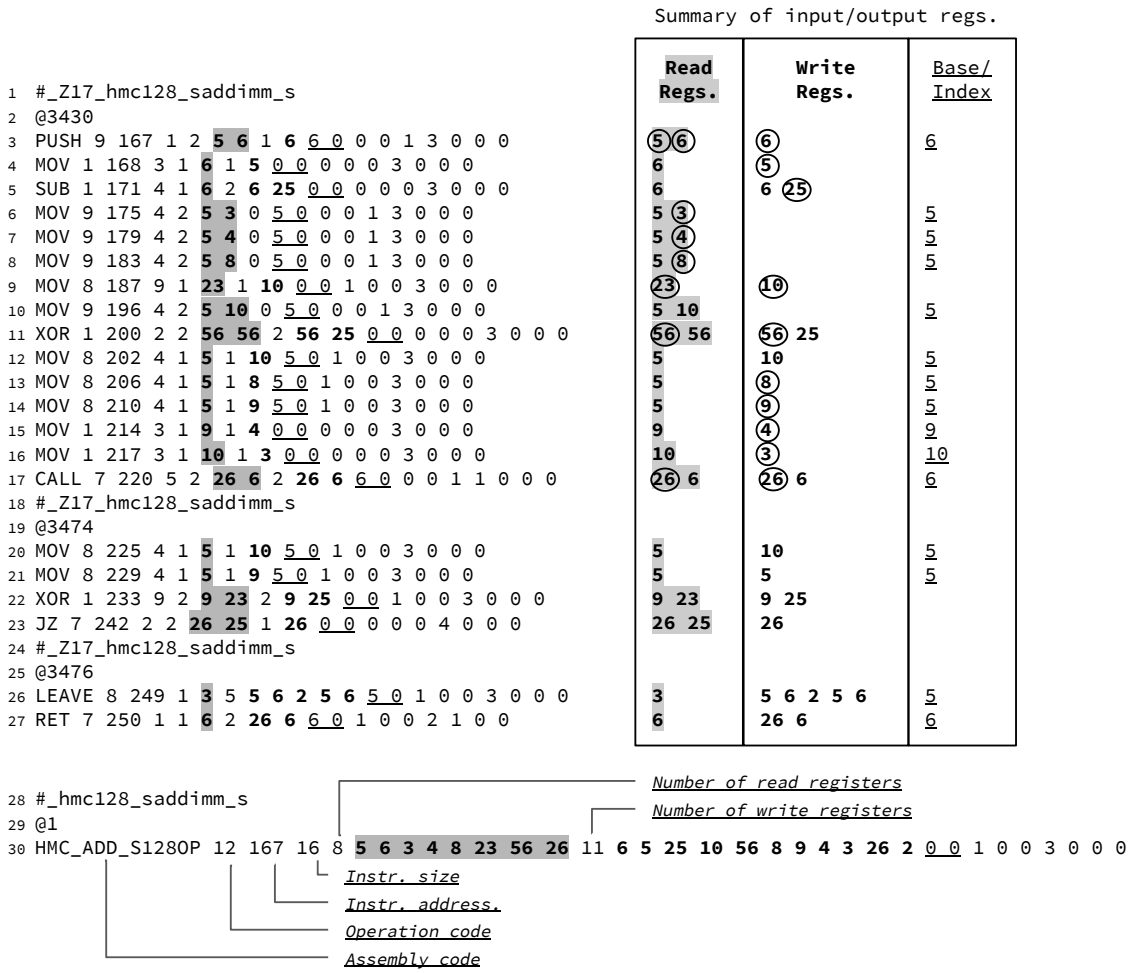

Figura 3.3: Exemplo ilustrando a função x86 traduzida para uma instrução HMC e as dependências entre registradores de leitura e escrita.

CAPÍTULO 4

APLICAÇÕES E RESULTADOS

Para validação deste trabalho, foram feitas simulações com dois algoritmos de banco de dados, *join*, de uma consulta sintética, e *select scan*, de uma *query* real tirada do TPC-H[1]. Estes algoritmos foram escolhidos devido ao comportamento de *streaming* de dados que é bastante apropriado para explorar a capacidade de processamento em memória do HMC.

```
1  #include "../hmc.hpp"
2
3  void nljoin(vector<__h64l1> &outer, vector<__h64l1> &inner,
4  vector<__h64l1> &join_index) {
5    for(size_t i=0; i < outer.size(); ++i) {
6      for(size_t j=0; j < inner.size(); ++j) {
7        if( _hmc64_equalto_s(outer[i], inner[j])) {
8          join_index[i] = j;
9          break;
10       }
11     }
12   }
13 }
```

Código 4.1: *Join* com laços encadeados usando Intrinsics-HMC.

Os algoritmos de *join* são o centro das operações de junção do Database Management System (DBMS), os quais combinam duas tabelas (relações) comparando os atributos de junção e gerando um conjunto de tuplas que coincidem com esses atributos. Normalmente, os atributos de junção são chaves primárias e estrangeiras. Um dos algoritmos precursores de *join*, é o *nested loop join* (NLJoin), como segue o código 4.1. Este algoritmo percorre dois vetores, sendo que cada vetor representa uma relação. O laço externo interage na relação maior e o laço interno na menor. Dentro do laço interno é realizada a comparação entre os atributos de junção, na qual a função *_hmc64_equalto_s* da biblioteca Intrinsics-HMC é usada, caso estes atributos coincidam, o índice de junção[2] é realizado.

[1]Um padrão de referência para suporte de decisão em sistemas de banco de dados: `http://www.tpc.org/tpch`.

[2]Mais explicações sobre o índice de junção podem ser encontradas em `http://cs-www.cs.yale.edu/homes/dna/papers/vldb.pdf`.

Para a operação de *select scan* foi selecionada, do TPC-H, o algoritmo *query 6* que examina algumas colunas e seus valores são acessados somente uma vez, por exemplo, os valores, depois de usados, não são mais acessados nesta consulta. A *query 6* está apresentada no código 4.2, ela executa varredura e seleção aplicando predicados em três colunas (cláusula *WHERE*), projetando a soma de duas colunas somente para as tuplas que passaram pela avaliação de predicados.

```
1 SELECT
2   sum(l_price * l_disc) as revenue
3 FROM
4   lineitem
5 WHERE
6   l_date >= date '1994-01-01'
7   AND l_date < date '1994-01-01' + interval '1' year
8   AND l_disc between 0.05 AND 0.07
9   AND l_quant < 24;
```

Código 4.2: Query 6 do TPC-H em código SQL.

O código 4.3 é uma implementação da *query 6*, usando Intrinsics-HMC nativo, de forma que, um laço percorre quatro colunas armazenadas em vetores, sendo que, as colunas incluídas na cláusula *WHERE* são avaliadas pelas seguintes funções Intrinsics-HMC: *_hmc64_cmpswapgt_s* (compara e troca se maior) e *_hmc64_cmpswaplt_s* (compara e troca se menor). Somente as tuplas que passam pela avaliação de predicado são adicionadas à variável de resultado. Quando diz-se que as funções HMC utilizadas são da biblioteca Intrinsics-HMC nativa, significa que estas funções foram implementadas de acordo com as especificações documentadas do HMC. Neste caso, podemos observar que simples comparadores de menor ou maior seriam mais propícios na execução do *select scan*, porém tais instruções não existem na especificação do HMC e portanto não são consideradas na biblioteca Intrinsics-HMC nativa.

```
1  void query6() {
2      for(size_t i=0; i < l_date.size(); ++i) {
3          if( _hmc64_cmpswapgt_s(&l_date[i],19940101)>=19940101 &&
4              _hmc64_cmpswaplt_s(&l_date[i],19950101)<19950101 &&
5              _hmc64_cmpswapgt_s(&l_disc[i],5 )>=5 &&
6              _hmc64_cmpswaplt_s(&l_disc[i],7 )<=7 &&
7              _hmc64_cmpswaplt_s(&l_quant[i],24)<24)
8          {
9              res += l_price[i] * l_disc[i];
10         }
11     }
```

```
12 }
```

Código 4.3: Query 6 usando Intrinsics-HMC no código em C.

4.1 SIMULAÇÃO DOS TRAÇOS GERADOS

Para simulação, foram gerados os traços de ambas as aplicações nas versões x86 e HMC nativo, que são ilustrados nas figuras 4.1 e 4.2. As simulações mostram, respectivamente, o número de μops executados e o tempo de execução de cada uma das versões e aplicações do DBMS citadas acima, sendo que, de cada tabela com dados de entrada utilizada, foram selecionadas 100 mil linhas, ou seja, um total aproximado de 4 KB por coluna da tabela. Foi utilizado o número de μops, pois, a intenção era obter um número mais realista de operações de fato executadas pelas unidades funcionais, sendo que o número de *opcodes* não traz esta informação, pois estamos trabalhando com uma arquitetura CISC x86.

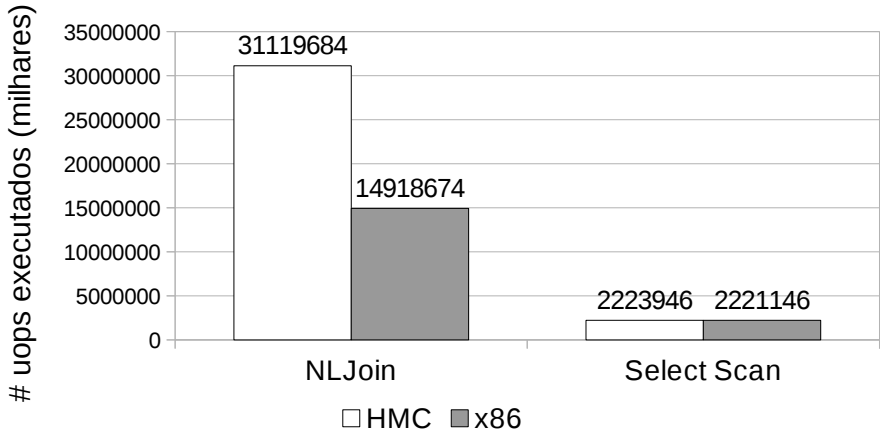

Figura 4.1: Número de μops executadas para as operações de *join* e *select scan* comparando x86 e HMC nativo.

A figura 4.1 apresenta o número total de μops executadas, enquanto que a figura 4.2 apresenta o tempo total de execução para cada experimento, sendo que cada figura compara os resultados das operações de *join* e *select scan* utilizando duas implementações, uma que utiliza instruções HMC com x86 e outra que utiliza instruções x86 somente.

A operação *NLJoin*, fazendo uso do conjunto de instruções do HMC, executa o dobro de μops que a versão x86 da mesma aplicação, enquanto que a operação de *select scan*, para o mesmo caso, não gerou tamanha diferença. A hipótese mais plausível para este

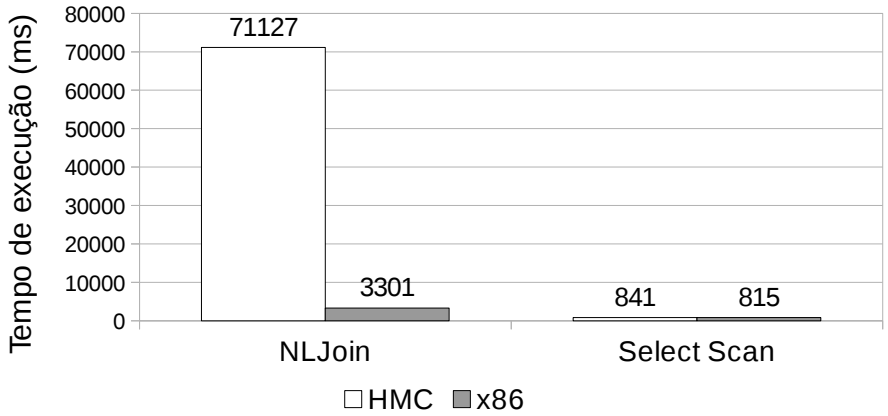

Figura 4.2: Tempo de execução das operações de *select scan* e *join* comparando x86 e HMC nativo.

valor discrepante é que, apesar da instrução HMC realizar exatamente a mesma operação de comparação que a instrução do conjunto x86, acreditamos que o compilador pode estar executando operações adicionais antes da chamada da função e após o retorno da mesma. Entretanto, tal hipótese necessita de investigações mais aprofundadas para verificar se a mesma é verdadeira.

Notamos ainda que o tempo de execução cresce proporcionalmente ao número de μops executadas para o caso do *select scan*. Porém o mesmo não é observado para o algoritmo *NLJoin*, que demora 21 vezes comparado à sua versão com instruções x86 somente. Tal discrepância no algoritmo *NLJoin* advém do fato que durante a execução do laço aninhado, as estruturas de dados necessárias cabem na cache, enquanto que na execução dentro do HMC, as instruções enfrentam as altas latências da DRAM.

Este aumento no tempo de execução quando a biblioteca Intrinsics-HMC é utilizada, foi obtido em trabalhos antigos [Alves et al., 2016], [Hadidi et al., 2017], isso ocorre pois o conjunto de instruções do HMC ainda é bastante restrito, assim, não existem instruções HMC equivalentes a instruções x86, por exemplo, para cada comparação da Intrinsics-HMC realizada, é feita também uma operação de leitura, uma de troca (*swap*) e outra de escrita, mas ainda assim é utilizada pois é a única instrução do conjunto de instruções do HMC que pode ser utilizada para tal finalidade. Além disso, os trabalhos anteriores descrevem que as instruções HMC operam apenar sobre 16 bytes por vez, o que é uma segunda causa de gargalo na arquitetura.

Instrução	Funcionalidade
_hmc64_cmpgteq_s	Se o operando do endereço de memória (8-byte) for maior ou igual ao o operando imediato(8-byte), retorna 1, caso contrário, retorna 0.
_hmc64_cmplteq_s	Se o operando do endereço de memória (8-byte) for menor ou igual ao o operando imediato(8-byte), retorna 1, caso contrário, retorna 0.
_hmc64_cmplt_s	Se o operando do endereço de memória (8-byte) for menor que o operando imediato(8-byte), retorna 1, caso contrário, retorna 0.

Tabela 4.1: Novas instruções adicionadas na Biblioteca Intrinsics-HMC.

4.2 EXTENSÃO DA INTRINSICS-HMC

A partir dos resultados obtidos utilizando as aplicações de *join* e *select scan* com a biblioteca Intrinsics-HMC nativa, não se pode dizer que foi obtido melhor desempenho, pois o número de μops executadas é maior que as do conjunto x86, enquanto que o tempo de execução teve um aumento significativo. Desta forma, é possível diminuir a diferença entre os tempos de execução das aplicações adicionando novas instruções HMC na biblioteca Intrinsics-HMC nativa. Assim, a tabela 4.2 apresenta as novas instruções para a biblioteca Intrinsics-HMC, para que, agora que adaptadas, sejam melhor aproveitadas no algoritmo da *query 6*, a qual obteve tempo de execução disparado, evitando assim, operações extras, como citado anteriormente.

Desta forma o código 4.2 é reescrito como apresentado no código 4.4, fazendo uso das novas instruções da biblioteca Intrinsics-HMC, de forma que, tendo seus traços de simulação gerados e sendo simulado com o SiNUCA, os resultados a seguir são obtidos.

```
1  void query6() {
2      for(size_t i=0; i < l_date.size(); ++i) {
3          if(_hmc64_cmpgteq_s(&l_date[i], 19940101) &&
4              _hmc64_cmplt_s(&l_date[i], 19950101) &&
5              _hmc64_cmpgteq_s(&l_disc[i], 5) &&
6              _hmc64_cmplteq_s(&l_disc[i], 7) &&
7              _hmc64_cmplt_s(&l_quant[i], 24));
8          {
9              res += l_price[i] * l_disc[i];
10         }
11     }
12 }
```

Código 4.4: Query 6 usando novas instruções da biblioteca Intrinsics-HMC no código em C.

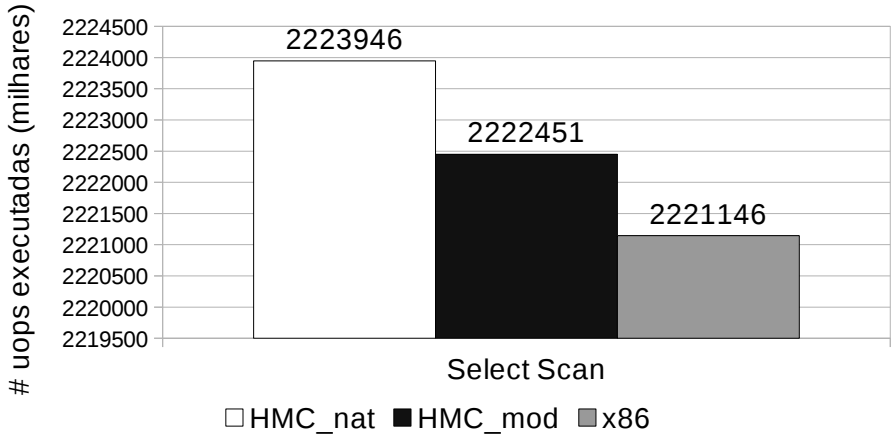

Figura 4.3: Número de μops executadas na operação de *select scan* comparando Intrinsics-HMC com ISA nativa e modificada.

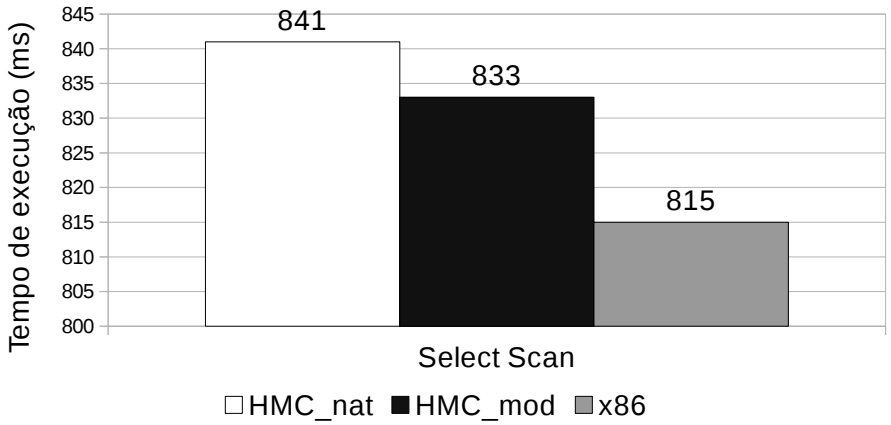

Figura 4.4: Tempo de execução da operação de *select scan* e *join* comparando Intrinsics-HMC com ISA nativa e modificada.

A figura 4.3 apresenta uma comparação entre o número de μops para a aplicação de *select scan* com as arquiteturas x86, HMC nativa e HMC modificada, enquanto que a figura 4.4 apresenta uma comparação entre os tempos de execução das mesmas arquiteturas com a mesma aplicação. Assim, é possível provar que, além da diminuição do número de μops executadas em uma arquitetura HMC modificada, o tempo de execução se aproxime mais ao tempo de execução em uma arquitetura x86. Pode-se observar, também, que a arquitetura x86 tem melhor desempenho em todos os casos, contudo, o propósito do trabalho é permitir a simulação de outras arquiteturas em tempos viáveis comparados com o tempo da arquitetura da máquina.

O uso da arquitetura do HMC, favorece aplicações que utilizem operações atômicas de memória, como, por exemplo, a construção de um *mutex*, que se aproveita integralmente

do conjunto de instruções do HMC, usando *locks* e barreiras de sincronização, tendo ganhos altos em desempenho [Leidel and Chen, 2017]. Por outro lado, em geral arquiteturas PIM possuem grande apelo para aplicações de *streamming* de dados como aplicações de banco de dados. Porém nossos resultados mostram que nesse sentido, a atual ISA do HMC não é adequada para a execução do das operações de *select scan* e *NLJoin*.

CAPÍTULO 5

CONCLUSÕES E TRABALHOS FUTUROS

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 trabalho, foi apresentada 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, pode-se escrever códigos completos em linguagens de alto nível utilizando funções que emulam novas instruções de processamento em memória. Após avaliada a corretude do código, é possibilitada a geração de traços simuláveis através do SiNUCA-Tracer que traduzem as funções comportamentais HMC em instruções simuláveis HMC. Neste trabalho, o foco é a proposta de uma solução utilizando o simulador SiNUCA como caso de exemplo, porém outros simuladores podem ser beneficiados pela metodologia apresentada.

Desta forma, projetistas e pessoas que produzem pesquisas nesta área podem se beneficiar desta ferramenta, poupando-as do trabalho propenso a erros, de escrever e modificar código em linguagem de montagem/simulável. Atualmente, apenas as instruções disponíveis na especificação do HMC foram implementadas na biblioteca Intrinsics-HMC. Contudo, é uma biblioteca de fácil extensão, permitindo, assim, a criação de novas funções para operações recorrentes em *benchmarks*. Fora isto, o gerador de traços SiNUCA-Tracer também pode ser modificado, permitindo simular arquiteturas pouco acessíveis ou inexistentes.

Durante o desenvolvimento deste trabalho de graduação, algumas de suas partes foram publicadas no VII Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC) [Cordeiro and Alves, 2017] e, também, no XVIII Simpósio de Sistemas Compu-

tacionais de Alto Desempenho (WSCAD) [Cordeiro et al., 2017].

Como trabalho futuro, consideramos revalidar trabalhos que efetuaram a avaliação de propostas de novas arquiteturas. Também consideramos a extensão da proposta para suportar instruções vetoriais em memória (HIVE).

REFERÊNCIAS

[Alves, 2014] Alves, M. (2014). *Increasing Energy Efficiency of Processor Caches via Line Usage Predictors*. PhD thesis, Universidade Federal do Rio Grande do Sul.

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

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

[Azarkhish et al., 2016] Azarkhish, E., Rossi, D., Loi, I., and Benini, L. (2016). A case for near memory computation inside the smart memory cube. In *Workshop on Emerging Memory Solutions*.

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

[Cordeiro and Alves, 2017] Cordeiro, A. S. and Alves, M. A. Z. (2017). Geração de traços de simulação para instruções de processamento em memória. *VII Simpósio Brasileiro de Engenharia de Sistemas Computacionais-SBESC*.

[Cordeiro et al., 2017] Cordeiro, A. S., Kepe, T. R., Tomé, D. G., de Almeida, E. C., and Alves, M. A. Z. (2017). Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions. *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho-WSCAD*.

[Elliott et al., 1999] Elliott, D. G., Stumm, M., Snelgrove, W. M., et al. (1999). Computational RAM: Implementing Processors in Memory. *Design and Test of Computers*, 16(1):32–41.

[Esmaeilzadeh et al., 2011] Esmaeilzadeh, H., Blem, E., St Amant, R., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM.

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

[Hennessy and Patterson, 2014] Hennessy, J. L. and Patterson, D. A. (2014). *Computer organization and design: the hardware/software interface*. Elsevier.

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

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

[Jain, 1990] Jain, R. (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

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

[Jeon and Chung, 2017] Jeon, D.-I. and Chung, K.-S. (2017). Cashmc: A cycle-accurate simulator for hybrid memory cube. *IEEE Computer Architecture Letters*, 16(1).

[Khalifa et al., 2013] Khalifa, K., Fawzy, H., El-Ashry, S., and Salah, K. (2013). Memory controller architectures: A comparative study. In *Int. Design and Test Symp.*

[Leidel and Chen, 2017] Leidel, J. D. and Chen, Y. (2017). Hmc-sim-2.0: A co-design infrastructure for exploring custom memory cube operations. *Parallel Computing*, 68:77–88.

[Moore, 1998] Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85.

[Moore et al., 1975] Moore, G. E. et al. (1975). Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13.

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

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

[Patterson et al., 1997] Patterson, D., Anderson, T., Cardwell, N., et al. (1997). A case for intelligent RAM. *IEEE Micro*, 17(2):34–44.

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

[Thanh-Hoang et al., 2014] Thanh-Hoang, T., Shambayati, A., Deutschbein, C., Hoffmann, H., and Chien, A. (2014). Performance and energy limits of a processor-integrated fft accelerator. In *High Performance Extreme Computing Conf.*

[Wulf and McKee, 1995] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24.