Universidade Estadual de Campinas
Instituto de Computação

Mario Mikio Hato

Análise de Desempenho e Otimização dos Simuladores ArchC

CAMPINAS
2017

Mario Mikio Hato

Análise de Desempenho e Otimização dos Simuladores ArchC

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Edson Borin
Coorientador: Prof. Dr. Rodolfo Jardim de Azevedo

Este exemplar corresponde à versão final da Dissertação defendida por Mario Mikio Hato e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): CAPES, 01-P-3951/2011; CAPES, 01-P-1965/2012
ORCID: http://orcid.org/0000-0001-5568-2211

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

H286a

Hato, Mario Mikio, 1987-
Análise de desempenho e otimização dos simuladores ArchC / Mario Mikio Hato. – Campinas, SP : [s.n.], 2017.

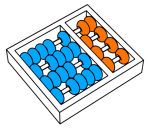
Orientador: Edson Borin.
Coorientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Hardware – Linguagens descritivas – Métodos de simulação. I. Borin, Edson,1979-. II. Azevedo, Rodolfo Jardim de,1974-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Performance analysis and optimizations of the ArchC simulators
Palavras-chave em inglês:
Computer architecture
Hardware description languages computer - Simulation methods
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Alexandro José Baldassin
Sandro Rigo
Data de defesa: 03-02-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação



Mario Mikio Hato

Análise de Desempenho e Otimização dos Simuladores ArchC

Banca Examinadora:

- Prof. Dr. Edson Borin
  IC-UNICAMP
- Prof. Dr. Alexandro José Baldassin
  IGCE-DEMAC-UNESP
- Prof. Dr. Sandro Rigo
  IC-UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 03 de fevereiro de 2017

Dedicatória

Dedico este trabalho aos meus pais, Shuji e Margarida, e toda minha família.

If you find a path with no obstacles,
it probably doesn't lead anywhere.
(Frank A. Clark)

Agradecimentos

Inicialmente, agradeço a Deus pela vida.

Aos meus pais, irmãs e a toda minha família que, com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa da minha vida.

Aos meus orientadores, por seus ensinamentos, paciência e confiança ao longo das supervisões das minhas atividades.

Aos meus amigos, pelas alegrias e tristezas compartilhadas. Com vocês, tudo o que tenho produzido na vida é muito melhor.

À CAPES e ao FAEPEX, pelo apoio financeiro que possibilitou a realização deste trabalho.

Por fim, a todos aqueles que de alguma forma estiveram e estão próximos de mim, fazendo esta vida valer cada vez mais a pena.

Resumo

Geração automática possui a grande vantagem de automatizar um processo, reduzir o tempo que seria gasto nesta etapa e evitar que erros comuns aconteçam. Porém, de que adianta reduzir o tempo de uma etapa se existe a possibilidade de aumentar o tempo das demais etapas. Em projetos de circuitos digitais, foram desenvolvidas as linguagens de descrição de arquitetura, que possibilitaram o surgimento de ferramentas capazes de gerar automaticamente simuladores, compiladores, etc., que são utilizados para avaliar uma arquitetura sem que esta tenha um *hardware* propriamente dito. Simuladores gerados automaticamente são utilizados para executar aplicações e averiguar o comportamento destas e da arquitetura sendo projetada. No entanto, caso o simulador gerado não seja eficiente, o tempo de simulação aumenta, podendo superar o ganho obtido pela geração automática, cancelando suas vantagens. Neste caso, como verificar a eficiência do simulador gerado? Uma forma bastante usada é comparar com outros simuladores existentes ou gerar o simulador manualmente para comparação. Comparar com simuladores existentes exigem que estes sejam similares, já gerar manualmente o simulador elimina o propósito da geração automática. Nesse contexto, desenvolvemos uma metodologia para se avaliar os simuladores gerados automaticamente através de perfilamento de código. Isto permitiu a identificação dos gargalos de desempenho e, consequentemente, o desenvolvimento de otimizações na geração de código. Com as otimizações, conseguimos gerar um simulador do modelo MIPS 1,48 vezes melhor.

Abstract

Automatic generation has a great advantage of automating a process. This reduces the time taken in this step and avoiding common mistakes. However, what is the advantage of reducing the time of a step if there is the possibility of increasing the time of the remaining steps? In digital circuit design, the architecture description languages emerged to make possible the development of tools that automatically generate simulators, compilers, and others tools, that we use to evaluate an architecture without it having a hardware itself. Automatically generated simulators run applications and verify their behavior and the architecture in design. But if the generated simulator is not efficient, the simulation time increases and can exceed the gain achieved by automatic generation, canceling its benefits. How to check the efficiency of the generated simulator in this case? A common option compares the generated simulator with other existing simulators. The other alternative is generating manually a simulator for comparison. The first choice requires that the simulators are similar and the second possibility eliminates the purpose of automatic generation. In this context, we have developed a methodology to evaluate the simulators automatically generated using code profiling. This allowed the identification of performance bottlenecks and, consequently, the development of optimizations on code generation. With the optimizations, we generated a MIPS simulator 1.48 times better.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O projeto de circuitos digitais é um trabalho bastante complexo e demorado. Para facilitar o desenvolvimento destes projetos e reduzir o tempo gasto foram desenvolvidas diversas ferramentas. No âmbito do projeto de novas arquiteturas é necessário o desenvolvimento não apenas de ferramentas para o *hardware* em si, mas de uma série de ferramentas para gerar também o *software* a ser utilizado nesta nova arquitetura. Para acelerar o fluxo de desenvolvimento, são utilizados simuladores de instruções e isso permite verificar o funcionamento de uma arquitetura antes de se ter um *hardware* propriamente dito. Alterações na arquitetura normalmente exigem alterações em todas as etapas envolvidas no projeto, e isto gera bastante retrabalho. Neste cenário foram desenvolvidas as linguagens de descrição de arquitetura (ADL, no inglês) que vieram para complementar as informações que as linguagens de descrição de *hardware* (HDL, no inglês) não possuíam e permitir a geração automática das ferramentas utilizadas no projeto da arquitetura.

Neste contexto, a geração automática de simuladores, compiladores, montadores, entre outras ferramentas, apresenta grandes vantagens por ser atualizada automaticamente a cada modificação na especificação da arquitetura. Porém, para se gerar ferramentas automaticamente normalmente adota-se dois tipos de abordagem: A primeira consiste em gerar ferramentas genéricas o suficiente para abranger qualquer especificação e a segunda consiste em gerar ferramentas específicas exclusivamente para determinada arquitetura. A primeira abordagem normalmente sofre de problemas de desempenho, visto que, a abrangência pelos quais seus métodos/funções devem respeitar obriga a geração de código para tratar todas as diferenças gerando uma sobrecarga que não ocorre na segunda abordagem. Além disso, como essas ferramentas devem ser genéricas, é bastante difícil de se explorar as vantagens proporcionadas por recursos específicos que cada arquitetura possui. Logo, mesmo com a economia de tempo proporcionado graças à geração automática, de nada adianta se as ferramentas geradas não apresentam desempenho satisfatório.

Neste caso, como avaliar o desempenho dos simuladores gerados automaticamente? Uma forma bastante comum de se fazer isso é realizando comparações com outros simuladores. Porém, neste trabalho, apresentamos uma metodologia para avaliar os simuladores gerados automaticamente a partir de perfilamento de código. Através dessa técnica é possível identificar os pontos de ineficiência nos simuladores gerados automaticamente e propor otimizações na geração de código.

1.1 Contribuições

Nossas principais contribuições com este trabalho são:

- uma metodologia para identificação de problemas de desempenho que consiste nos seguintes passos:
  1. Seleção/Filtro de *benchmarks*
  2. Seleção de instruções importantes
  3. Geração de *microbenchmarks* com as instruções selecionadas
  4. Perfilamento dos *microbenchmarks* e análise dos resultados
  5. Análise do código selecionado e levantamento de hipóteses/otimizações
  6. Implementação das otimizações
  7. Retorna ao passo 4 até os resultados serem satisfatórios
  8. Perfilamento dos *benchmarks* selecionados e análise dos resultados
  9. Escolha dos melhores conjuntos caso existam otimizações combinadas
  10. Análise do código selecionado e levantamento de hipóteses/otimizações
  11. Combinação/União de otimizações
  12. Implementação das otimizações
  13. Retorno ao passo 8 enquanto houver otimizações identificadas
- otimização dos simuladores gerados automaticamente pelo ArchC com a implementação de melhorias no gerador automático.

1.2 Organização Textual

Esta dissertação está organizada da seguinte forma. O Capítulo 2 apresenta os conceitos básicos utilizados e os trabalhos relacionados com esta dissertação. O Capítulo 3 mostra a metodologia desenvolvida para a análise de desempenho realizado sobre o simulador MIPS sem qualquer alteração. O Capítulo 4 detalha os pontos de ineficiência do código fonte do simulador e descreve as possíveis otimizações identificadas. Já o Capítulo 5 descreve os resultados obtidos com as otimizações que foram implementadas no gerador automático ACSIM. Por fim, o Capítulo 6 apresenta as conclusões deste trabalho, juntamente com os pontos/melhorias levantados que não foram possíveis de se realizar nesta dissertação e que ficaram para trabalhos futuros.

Capítulo 2

Conceitos Básicos e Trabalhos Relacionados

Este capítulo apresenta os conceitos básicos utilizados nos demais capítulos, bem como os trabalhos relacionados relevantes da literatura.

2.1 ArchC

O ArchC [1, 45] é uma linguagem de descrição de arquitetura (do inglês, *Architecture Description Language*, ADL) [12, 40] *open source* baseada em SystemC [38] e um conjunto de ferramentas desenvolvidos no Laboratório de Sistemas de Computação (LSC) da Unicamp. Essas ferramentas permitem a geração automática de simuladores, montadores [2], ligadores, depuradores e outros utilitários [3,4] a partir da descrição da arquitetura, acelerando o fluxo de desenvolvimento de novos sistemas.

Entre estas diversas ferramentas, o foco deste trabalho está nos geradores automáticos de simuladores de conjunto de instruções (ISS) [44]. Os geradores do ArchC permitem desde a criação de simuladores interpretados, quanto a criação de simuladores com tradução de binários [36]. Tradução de binários consiste em compilar a aplicação a ser simulada substituindo-se as instruções da arquitetura alvo para as respectivas instruções da arquitetura hospedeira. Esta abordagem possui um grande custo inicial, porém, após a tradução das instruções, sua execução é bastante rápida. Este processo de tradução pode ser executado de forma estática – antes de iniciar a execução da aplicação – ou de forma dinâmica – durante a execução. O ArchC possui ferramentas para as duas formas. A ferramenta ACCSIM [5, 24] é responsável por gerar simuladores com tradução estática e a ferramenta ACDCSIM [23], responsável pelos simuladores utilizando tradução dinâmica.

Por outro lado, os simuladores interpretados são gerados pela ferramenta ACSIM e esta será a ferramenta analisada neste trabalho. A simulação interpretada [9], diferente da compilada, funciona de forma similar ao comportamento do *hardware*. Inicialmente busca-se a instrução a ser simulada, depois decodifica-se a mesma e por fim é realizado um despacho para a rotina responsável por simular o comportamento da instrução. Esta abordagem permite que o simulador seja bastante simples e portável, porém o desempenho da simulação é um dos pontos fracos.

A Figura 2.1 ilustra o processo de geração automática dos simuladores interpretados utilizando a ferramenta ACSIM. Inicialmente é necessário criar dois arquivos para descrever a arquitetura. No arquivo `AC_ARCH` são definidas as características da arquitetura, como quantidade e tamanho dos registradores, tamanho da palavra, quantidade de memória endereçável, entre outras informações. Já no arquivo `AC_ISA` são definidas as informações acerca do conjunto de instruções, como os formatos e campos das instruções, o nome e outras informações usadas na decodificação das instruções em si. Com estes dois arquivos, o ACSIM consegue emitir o código do simulador, que, junto do código das rotinas de comportamento fornecidas pelo usuário pode ser compilado pelo GCC. Após a compilação, tem-se um simulador interpretado como produto final.

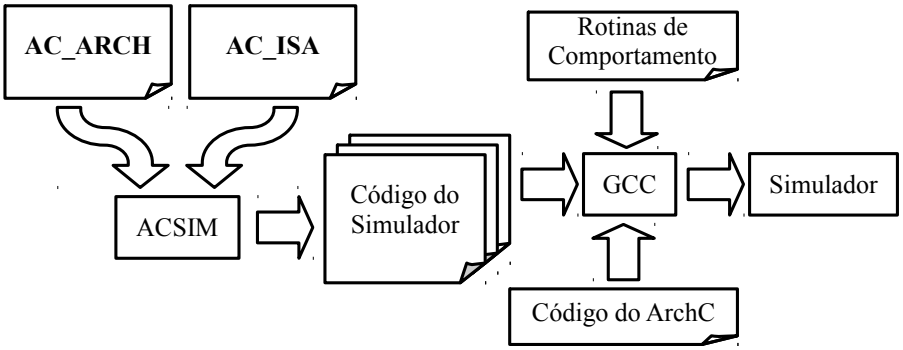


Figura 2.1: Processo de geração de um simulador utilizando a ferramenta ACSIM.

Os simuladores ArchC são formados por 3 partes. A primeira parte consiste nas bibliotecas disponibilizadas pelo ArchC que possuem as rotinas de tratamento de chamadas de sistema e de acesso à leitura e escrita em memória, o decodificador de instruções, a *cache* de instruções decodificadas entre outros componentes que são utilizados em comum pelos simuladores. A segunda parte é relacionada à ferramenta responsável por realizar a emulação das instruções. Nesta parte, dependendo da ferramenta utilizada, o código emitido será diferente, podendo variar desde um simulador interpretado (ACSIM), até um simulador com tradução de binários (ACCSIM ou ACDCSIM). Por fim, a última parte consiste em código fornecido pelo usuário, que define as rotinas comportamentais da arquitetura a ser simulada. Estas partes são resumidas na Figura 2.2, que mostra a estrutura dos simuladores tendo como base a ferramenta ACSIM.

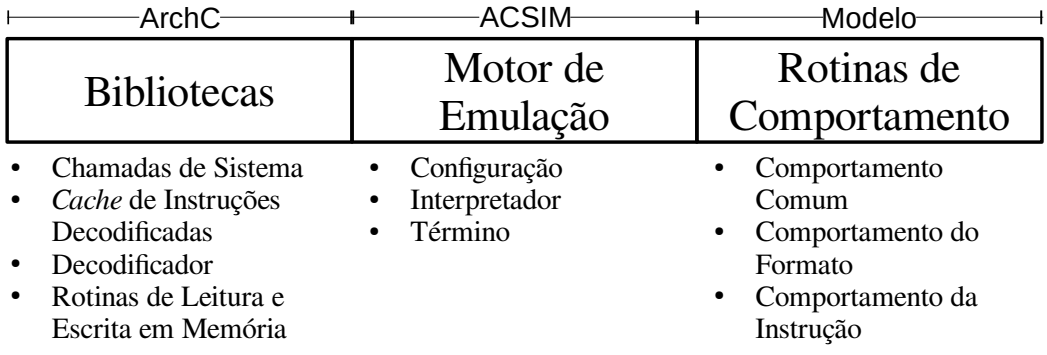


Figura 2.2: Estrutura dos simuladores gerados pelo ACSIM.

Como o ACSIM trata da geração de simuladores interpretados, o motor de emulação gerado pela ferramenta faz uso de uma técnica de interpretação. A técnica implementada atualmente nesta ferramenta é a interpretação clássica com tabela de *opcodes*, também conhecida como abordagem clássica, *Instruction Dispatch Using Switch* [47], *Switch dispatch* [11, 14] ou *decode-and-dispatch* [50]. Esta técnica foi inicialmente apresentada por Klint [28] e consiste em um laço com um comando *switch* que seleciona uma rotina de emulação de instrução em função do *opcode* da instrução. A Figura 2.3 ilustra o processo de interpretação clássica.

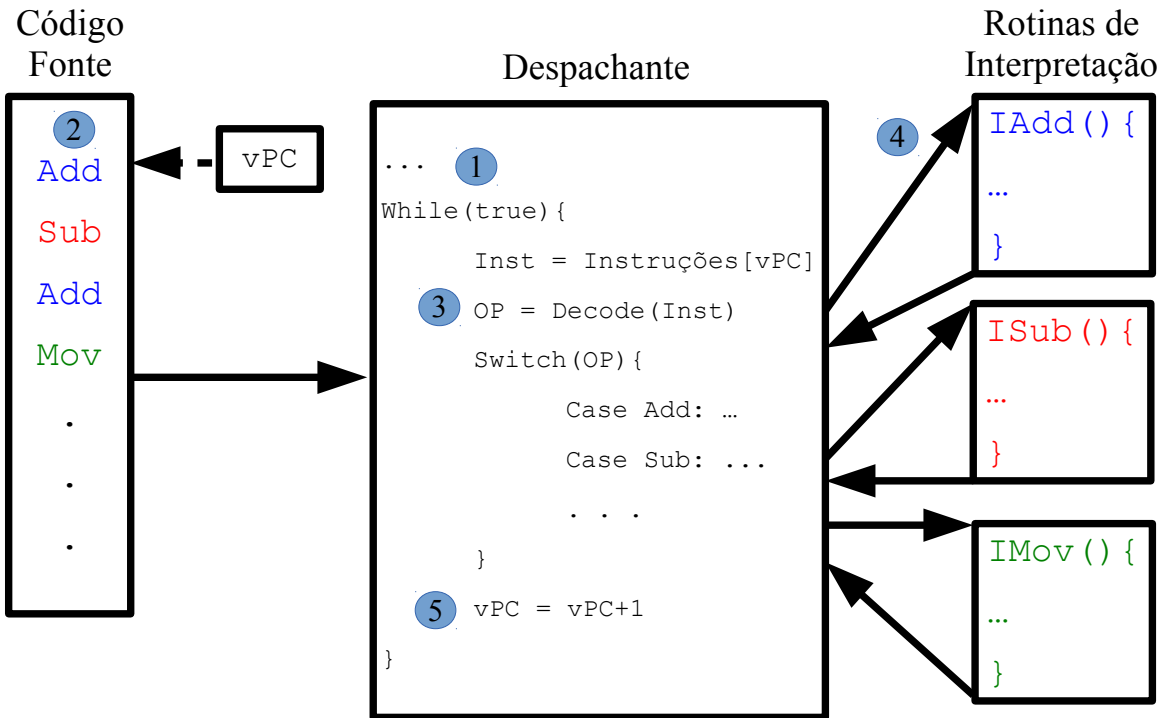


Figura 2.3: Técnica de Interpretação Clássica. Adaptado de Smith e Nair [50].

De forma geral, o simulador inicia o laço da simulação (1), depois busca-se no código fonte (2) a instrução a ser emulada. Com o *opcode* (3) é realizado o direcionamento do fluxo de execução para a rotina de interpretação correspondente (4). Por fim, é realizado o incremento para a próxima instrução (5) e o processo é executado novamente. Esta técnica possui como ponto forte a facilidade de implementação, simplicidade e grande portabilidade, e como ponto fraco um desempenho inferior às técnicas de tradução e compilação de binários.

2.2 Trabalhos Relacionados

Existem diversas ADLs descritas em trabalhos na literatura que, assim como o ArchC, realizam a geração automática de simuladores. Entre os principais, pode-se citar o LISA [34, 54], o EXPRESSION [26], o nML [21], entre outros. Porém a comparação entre estes e o ArchC não é o foco deste trabalho. Estes foram citados para reforçar que

o uso de ADLs é bastante comum.

Por outro lado, também exitem trabalhos em que são utilizados simuladores específicos e especializados para determinada arquitetura, como o BrouHaHa, o Cint e o Talisman, que são simuladores interpretados. O BrouHaHa [37] é um intepretador de Smalltalk-80 e utiliza a técnica de interpretação *decode-and-dispatch*. Ele está escrito quase todo na linguagem C e utiliza uma *cache* de contexto para melhorar o desempenho e armazenar blocos de instruções. A cada novo bloco criado ou quando ocorre alteração de algum bloco a *cache* de contexto é atualizada. Já o Cint [13] é um interpretador que converte o código fonte em linguagem C para um código em representação intermediária e depois realiza interpretação para executar o código gerado. Ele utiliza a técnica de interpretação clássica e possui como grande vantagem a portabilidade para várias arquiteturas sem a necessidade de modificação. Por fim, o Talisman [6] é um simulador para multiprocessadores composto de um tradutor de instruções, um interpretador, modelos de *cache*, modelos de TLB, entre outros e utiliza a técnica *threaded code* [7] de interpretação.

Existem também trabalhos com geradores de interpretadores, como o Vmgen [20], que permitem a comparação de diversas técnicas de interpretação. O Vmgen é um gerador de interpretadores para máquinas virtuais baseadas em pilha. Ele utiliza como técnicas de interpretação as técnicas *threaded code, stack caching [17] e superinstructions [11, 19, 20]*, e realiza otimizações no escalonamento e na busca das próximas instruções para prover maior precisão na predição de desvios. Nos testes, os interpretadores gerados pelo Vmgen mesmo sem a técnica de *superinstructions* se mostraram bastante competitivos em relação a outros interpretadores. Diferentemente do Vmgen, o ACSIM gera simuladores baseados em registradores e era capaz somente de gerar simuladores contendo a técnica clássica, porém, neste trabalho foi incluído a opção de geração de simuladores com a técnica *Direct Threaded Code*, aproximando-se um pouco do Vmgen.

Uma forma bastante comum de avaliar o desempenho de simuladores é realizando comparações entre eles [46]. Quando se trata de simuladores interpretados, o mais comum e bastante encontrado nos trabalhos presentes na literatura é a comparação envolvendo as diferentes técnicas de interpretação desenvolvidas com o tempo [19, 29, 47, 52]. Entre as diversas técnicas de interpretação existentes, as mais comuns são a técnica clássica – descrita anteriormente – e a técnica de *Threading* – ilustrada na Figura 2.4 – que serve de base para diversas outras técnicas, como *Indirect Threaded Code* [15, 28, 50], *Context Threading* [8] e *Replication* [11].

A técnica Threading foi inicialmente apresentada por Bell [7] em 1973 como *Threaded Interpretation* e consiste em remover o laço e o comando *switch* que realiza o despacho na abordagem clássica. Para realizar este processo, a aplicação a ser emulada (1) é carregada em memória e durante este processo o *loader* (2) substitui o campo de *opcode* pelo endereço da rotina de interpretação (4) correspondente, gerando o que é chamado de código intermediário (3). Como cada instrução do código intermediário agora tem o endereço de sua respectiva rotina de interpretação, isto permite descentralizar o fluxo de controle, possibilitando que ao final da execução (5, 6 e 7) de qualquer rotina de interpretação a próxima instrução seja buscada e o fluxo seja encaminhado para a próxima rotina de interpretação sem a necessidade de ter que voltar para o despachante.

Posteriormente esta técnica passou a ser conhecida como *Direct Threaded Code* [11,15,

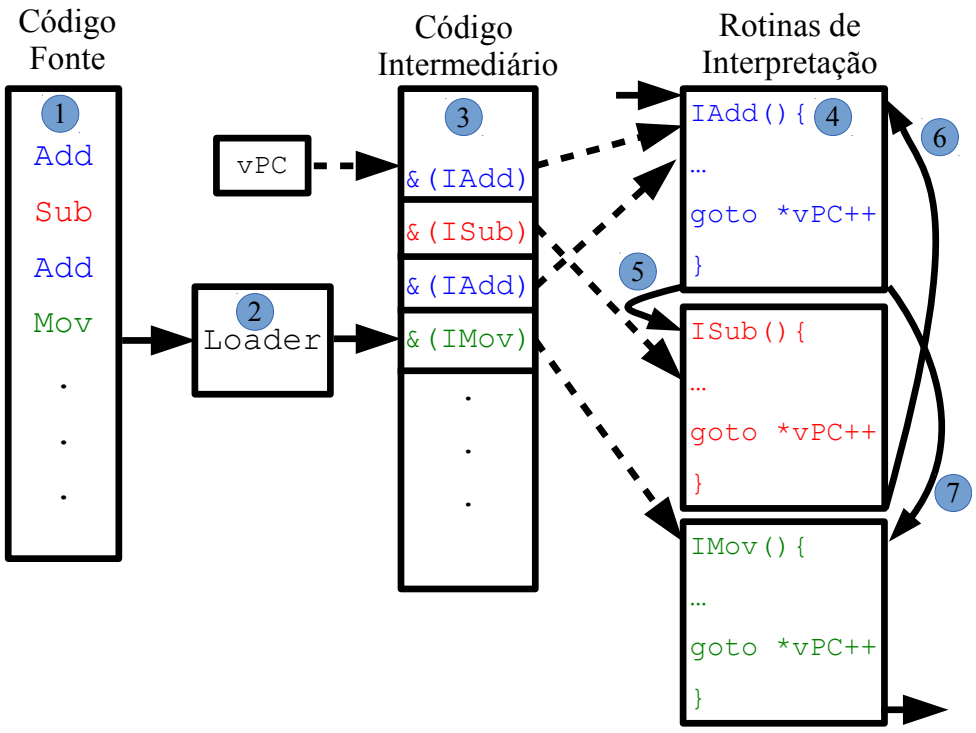


Figura 2.4: Técnica de Interpretação *Direct Threaded Code*. Adaptado de Berndl e outros [8].

28] ou *Threaded Dispatch* [14]. Smith e Nair [50] apresentam esta mesma técnica com a adição de pré-decodificação, onde as instruções são armazenadas no código intermediário de forma decodificada para melhorar a velocidade de interpretação. Rossi e Sivalingam [47] descrevem uma variação desta técnica, denominada *Direct Threading Using Tail-Calls* ou *Token Threaded Dispatch* [14]. No padrão ANSI C e em outras linguagens que não possuem rótulos de primeira classe, a técnica DTC não pode ser implementada diretamente, logo uma solução é utilizar *tail-calls*[1]. Já no padrão GNU C, como possui suporte a rótulos como valor, a técnica DTC pode ser implementada diretamente.

Existem também na literatura trabalhos descrevendo formas eficientes de geração de simuladores [31, 43]. Porém, a maioria deles tratam do problema de trocar a arquitetura alvo de simulação, ou seja, trocar a máquina hospedeira em que será executado o simulador. Além disso, estes trabalhos são realizados em simuladores compilados, visto que, diferente destes simuladores, uma das grandes vantagens da simulação interpretada é a flexibilidade e enorme portabilidade dos simuladores interpretados.

Já em relação à análise de desempenho existem diversas metodologias para identificar os pontos de gargalo do sistema a ser analisado. Uma das mais conhecidas e utilizada por diversos trabalhos [42, 48, 53] é a técnica de análise de caminho crítico. A outra forma de identificar os gargalos é realizando o perfilamento da aplicação para conhecer melhor seu comportamento e, assim, possibilitar o desenvolvimento de otimizações apropriadas [10, 22, 51], e foi esta a abordagem adotada neste trabalho.

[1] *Tail-Calls* são sub-rotinas que são chamadas por alguma rotina e ao retornar valores, realizam a chamada de outra sub-rotina.

Uma forma bem comum de perfilamento consiste em utilizar os contadores de evento contidos no processador para se amostrar a quantidade de instruções executadas, acessos e faltas ocorridas na *cache* ou o total de desvios e erros de predição, por exemplo, que ocorreram durante a execução de determinada aplicação. Em [10] é realizado o perfilamento de um simulador de tráfico para identificar os pontos críticos – que estavam nas funções de escalonamento de eventos – e sugerir formas de otimizá-los. Já em [51] foi desenvolvido um sistema capaz de auxiliar na identificação de problemas no desempenho de aplicações em Java através da análise dos contadores de desempenho do *hardware* em sistemas *multithread* e multiprocessados. Diferentemente destes, nesta dissertação o foco não foi em um simulador/aplicação específico, pois as otimizações propostas abrangem todos os simuladores gerados automaticamente, e além de identificar os gargalos, também foram implementados as otimizações propostas. Por fim, em [22] é realizado o perfilamento e otimização do LAMMPS(*Large-scale Atomic/Molecular Massively Parallel Simulator*), que é um simulador molecular dinâmico que também é utilizado como *benchmark* para avaliar sistemas de computação de alto desempenho. Neste caso, o perfilamento serviu para apontar diversos segmentos no código para otimização. As otimizações consistiram principalmente em remoção de desvios em laços críticos e buscaram melhorar o paralelismo do mesmo.

Capítulo 3

Análise de Desempenho dos Simuladores ArchC

Neste capítulo é apresentada a análise de desempenho realizada nos simuladores ArchC. Para isso, inicialmente será descrito todo o ambiente computacional envolvido nesta avaliação e posteriormente será detalhado como foi realizada a análise de desempenho. Como o foco deste trabalho são os simuladores interpretados, a análise irá cobrir somente os simuladores gerados automaticamente pela ferramenta ACSIM.

Os simuladores gerados pelo ACSIM realizam a simulação em três fases: Configuração, emulação e término. Na primeira fase, o aplicativo a ser emulado é carregado para a memória e é iniciado o ambiente de simulação. A fase de emulação consiste na busca e despacho das instruções da aplicação hóspede e execução da rotina responsável por emular o comportamento da instrução. Por fim, a última fase consiste na impressão das informações da simulação e liberação dos recursos utilizados.

As etapas de configuração e término dependem do tamanho estático do programa sendo emulado e geralmente executam em um tempo relativamente rápido, gastando apenas alguns millisegundos. O tempo de execução da etapa de emulação, por outro lado, depende do tamanho dinâmico (número de instruções executadas) da aplicação sendo emulada. Como este tamanho é geralmente muito maior do que o tamanho estático da aplicação, o tempo de execução da emulação é geralmente muito maior do que o tempo das outras etapas. Dessa forma, o foco do trabalho está na análise e otimização da emulação, tornando importante garantir que esta ocupe a maior parte do tempo de simulação. Assim sendo, inicialmente foi realizado a análise do custo de configuração e término dos simuladores e os *benchmarks* que possuem tempo de emulação abaixo dos tempo de configuração e término foram removidos para realizar as demais análises.

3.1 Ambiente Experimental

Esta seção apresenta o ambiente em que foram realizados os experimentos deste trabalho. Será listado o *hardware* que foi utilizado, bem como os aplicativos que auxiliaram na experimentação. O ArchC possui como dependência o SystemC, logo, foi utilizado o SystemC 2.3 que é compatível com a versão oficial 2.2 do ArchC. Para a compilação do

ArchC, dos modelos e demais aplicações foi utilizado o gcc 4.4.3. Para coleta de dados e perfilamento, foram utilizados o `Perf` 0.0.2 e o `PIN` 2.13. Além disso, foi utilizada a versão 1.0 dos modelos ARM, MIPS, PowerPC e SPARC.

Os experimentos foram executados em um sistema operacional Ubuntu Server Linux LTS 10.4 x64 - kernel 2.6.32. Para testes de desempenho foi utilizado um processador Intel Core i7 980X com 6 núcleos com HT que possuem clock de 3.33 GHz e com 16 GB de memória RAM. Para os demais testes foi utilizado um processador Core 2 Quad de 2,4 GHz com 4 GB de memória RAM.

3.2 Modelos ArchC Utilizados

O portal do ArchC disponibiliza 4 modelos prontos de processadores, juntamente com os *benchmarks* e ferramentas de apoio já compiladas. Os modelos são: ARM [49], MIPS [27], PowerPC [16] e SPARC [39]. O modelo MIPS, dentre os quatro, é o simulador mais simples, já o modelo ARM é o simulador mais complexo, pois utiliza basicamente todas as rotinas de comportamento disponíveis pelo ArchC.

A Tabela 3.1 apresenta os 4 modelos selecionados do ArchC, seu *Endianness*, a quantidade de formatos de instrução e a quantidade de instruções que cada arquitetura possui. Por ser o modelo mais simples, visto que, o comportamento das instruções é bastante simples e possui bem menos instruções que os demais modelos, o modelo MIPS será utilizado como base para os demais experimentos. Os demais modelos serão utilizados somente no final para testes de desempenho e verificação de funcionamento.

Tabela 3.1: Modelos do ArchC.

Modelo	*Endianness*	# de Formatos	# de Instruções
ARM	*little*	21	100
MIPS	*big*	3	59
PowerPC	*big*	49	181
SPARC	*big*	6	119

3.3 Descrição dos *Benchmarks*

Esta seção apresenta o conjunto de *benchmarks* que foram pré-selecionados para medição/análise de desempenho dos simuladores. Foram pré-selecionadas as aplicações das suítes MiBench e MediaBench descritas a seguir.

3.3.1 MiBench

O MiBench [25, 35] é um conjunto de aplicações utilizadas para avaliar processadores embarcados. Estes aplicativos são classificados em seis categorias, de forma a diferenciar e restringir o foco do que está sendo medido. As seis categorias são: automação, consumo,

rede, escritório, segurança e telecomunicação. Abaixo segue uma breve descrição de cada uma destas categorias.

1. **Automação:** Esta categoria engloba os *benchmarks* que melhor caracterizam o ambiente presente na automação e controle industrial. Este ambiente requer desempenho em cálculos matemáticos, manipulação de *bits* e movimentação e organização de dados.

2. **Consumidor:** Esta categoria representa os *benchmarks* utilizados em diversos dispositivos de consumo - como câmeras, celulares, tocadores de MP3, entre outros - e seu foco concentra-se em aplicações multimídias.

3. **Rede:** Esta categoria apresenta os *benchmarks* responsáveis por medir o desempenho de processadores embarcados utilizados em dispositivos de rede, como *switches* e roteadores.

4. **Escritório:** Esta categoria aborda os *benchmarks* de aplicações que representam o maquinário utilizado em escritório. Estas aplicações tratam de itens relacionados à manipulação de textos e à organização de dados.

5. **Segurança:** Esta categoria traz os *benchmarks* que realizam encriptação / decriptação de dados utilizados em dispositivos cujo foco é a segurança dos dados.

6. **Telecomunicação:** Esta categoria abrange os *benchmarks* responsáveis por avaliar o desempenho de processadores embarcados utilizados em telecomunicação de dados. Consiste em aplicações que realizam análise de frequência, algoritmos de checagem de dados e codificação/decodificação de voz.

A Tabela 3.2 apresenta os *benchmarks* que compõem a suíte MiBench e estão separados por suas respectivas categorias. Estes *benchmarks* listados são os que estão disponíveis/-compilados para os simuladores gerados pelo ACSIM.

Tabela 3.2: *Benchmarks* da suíte MiBench separados por categoria.

Automação	Consumidor	Rede	Escritório	Segurança	Telecomunicação
basicmath	jpeg (encode)	dijkstra	stringsearch	rijndael (encode)	adpcm (encode)
bitcount	jpeg (decode)	patricia		rijndael (decode)	adpcm (decode)
qsort	lame			sha	crc32
susan (corners)					fft
susan (edges)					fft (inversa)
susan (smoothing)					gsm (encode)
					gsm (decode)

3.3.2 MediaBench

O MediaBench [30, 33] é uma suíte de *benchmarks* que possui foco em aplicações multimídia, processamento de imagens e comunicações. É bastante utilizado em análise de

processadores em sistemas embarcados, pois foi elaborado com o objetivo de capturar os elementos essenciais para estes ambientes.

Serão utilizados os *benchmarks* da versão I da suíte MediaBench disponíveis no ArchC, por já estarem compilados e testados para as respectivas arquiteturas dos principais simuladores gerados automaticamente pelo ArchC. Entre os *benchmarks* da suíte completa, o ArchC disponibiliza os seguintes: adpcm; gsm; jpeg; mpeg e pegwit.

3.4 Análise do Custo de Configuração e Término

Esta seção descreve as medições do custo das etapas de configuração e término dos simuladores gerados pelo ACSIM. Isto é importante para se ter uma ideia se a quantidade de instruções da aplicação hóspede são suficientes para analisar o desempenho da etapa de emulação ou se o tempo de execução é dominado pelas demais etapas. Esta análise nos auxiliará na seleção dos *benchmarks* a serem utilizados na análise de desempenho.

3.4.1 Metodologia

Para realizar as medições, foi implementado um *microbenchmark* contendo um laço com uma quantidade pré-definida de instruções de soma. A quantidade de iterações e instruções do laço eram alteradas a cada execução para verificar o efeito que surtia na quantidade de instruções executadas pela máquina hospedeira. Para verificar como essa alteração se comporta em relação às etapas de configuração e término do simulador, foram geradas duas versões. A primeira executa toda a aplicação e a segunda possui uma instrução de finalização logo no início do programa. Assim é possível mensurar o custo das etapas de configuração e término. Para simulação foi utilizado o modelo MIPS, por este ser o mais simples e possuir menos instruções que os demais modelos.

Os códigos-fonte a seguir exemplificam como os *microbenchmarks* foram implementados. O Código 3.1 exibe o *microbenchmark* que executa as instruções na quantidade pré-definida e o Código 3.2 mostra o mesmo *microbenchmark* com uma instrução – na linha 2 – para finalizar a simulação no início da emulação. É importante ressaltar que os códigos devem ter o mesmo tamanho, pois a etapa de configuração inclui o carregamento da aplicação para a memória, logo o tempo desta etapa varia de acordo com o tamanho do código.

```
1  main:
2    nop
3    nop
4    li $2,98
5  $L9:
6    addu $3,$3,1
7    addu $3,$3,-1
8    ...
9    addu $3,$3,1
10   addu $3,$3,-1
11   bgez $2,$L9
12   addu $2,$2,-1
13   j $31
14   move $2,$0
```

Código 3.1: *Microbenchmark* com execução total.

```
1  main:
2    j $31
3    nop
4    li $2,98
5  $L9:
6    addu $3,$3,1
7    addu $3,$3,-1
8    ...
9    addu $3,$3,1
10   addu $3,$3,-1
11   bgez $2,$L9
12   addu $2,$2,-1
13   j $31
14   move $2,$0
```

Código 3.2: *Microbenchmark* com finalização no início.

3.4.2 Resultados Experimentais

As Tabelas 3.3 e 3.4 apresentam a quantidade de instruções executadas durante a simulação do *microbenchmark*. Na primeira tabela a medição foi realizada com o programa `Perf` [41] e a segunda com o programa `PIN` [32].

Nestas tabelas, a primeira coluna(Tam. Laço) apresenta a quantidade de instruções contidas no laço, a segunda coluna (# de Iterações) mostra a quantidade de vezes que o laço foi executado e na terceira coluna(Inst. Hóspede) está a quantidade total de instruções da aplicação emulada. Nas colunas seguintes são apresentadas a quantidade de instruções da máquina hospedeira executadas nas etapas de configuração e término do simulador e a quantidade total de instruções executadas na simulação. Por fim é apresentada a porcentagem que as etapas de configuração e término ocupam em relação à simulação completa.

Tabela 3.3: Custo das etapas de configuração e término da simulação utilizando o `Perf`.

Tam.	# de	Inst.	Inst. Hospedeira		
Laço	Iterações	Hóspede	Config./Térm.	Total	%
100	100	10.216	5.879.912	6.649.410	88,43%
100	1.000	102.016	5.874.881	12.454.108	47,17%
100	10.000	1.020.016	5.873.277	70.535.522	8,33%
100	100.000	10.200.017	5.875.843	651.324.526	0,90%
1.000	100	100.216	5.931.929	13.374.861	44,35%
1.000	1.000	1.002.016	5.933.812	70.398.016	8,43%
1.000	10.000	10.020.016	5.934.334	640.623.949	0,93%
1.000	100.000	100.200.017	5.929.719	6.342.494.972	0,09%

Tabela 3.4: Custo das etapas de configuração e término da simulação utilizando o `PIN`.

Tam. Laço	# de Iterações	Inst. Hóspede	Inst. Hospedeira		
			Config./Térm.	Total	%
100	100	10.216	3.909.199	4.678.436	83,56%
100	1.000	102.016	3.909.199	10.485.870	37,28%
100	10.000	1.020.016	3.909.199	68.555.644	5,70%
100	100.000	10.200.017	3.909.299	649.254.833	0,60%
1.000	100	100.216	3.921.803	11.322.190	34,64%
1.000	1.000	1.002.016	3.921.803	68.335.615	5,74%
1.000	10.000	10.020.016	3.921.803	638.467.715	0,61%
1.000	100.000	100.200.017	3.921.819	6.339.789.118	0,06%

A aplicação hóspede utiliza 16 ou 17 instruções para inicialização e finalização do programa (I_0). Este número varia de acordo com a quantidade de iterações do laço. Quando esta quantidade é pequena utiliza-se apenas 16 instruções, já quando esta quantidade é grande, é necessário mais uma instrução para inicializar a variável de controle do laço. No laço de repetição, além das instruções de soma, executam-se mais 2 instruções relacionadas com o controle do laço - uma instrução de desvio condicional (`bgez`) para finalizar o laço e uma instrução de decremento (`addu`) para controlar as iterações do laço. Logo, a quantidade de instruções da aplicação hóspede executadas pode ser calculado da seguinte forma:

$$\textbf{Inst. Hóspede} = I_0 + \textbf{(Tam. Laço + 2) * \# de Iterações}$$

Inicialmente nota-se que há uma diferença entre as medidas do `PIN` com as do `Perf`. O `PIN` realiza a contagem das instruções através de instrumentação de código, já o `Perf` realiza a contagem através de amostragem. Logo, esta diferença pode ser justificada pelo fato do `Perf` incluir fatores externos na contagem, como as intervenções do Sistema Operacional. No entanto, ao comparar os resultados isoladamente, nota-se que a relação entre as Instruções da aplicação hóspede e da máquina hospedeira são muito similares, indicando que esta diferença não atrapalha na confiabilidade das duas medições.

Conclui-se que o custo das etapas de configuração e término dos simuladores gerados pelo ACSIM está na casa de alguns milhões de instruções da máquina hospedeira. Logo, para que um *benchmark* não seja afetado por estes custos, esta aplicação hóspede deve executar pelo menos 100 milhões de instruções. Com este requisito, pôde-se realizar um filtro na lista de *benchmarks*. A Tabela 3.5 apresenta a lista contendo todos os *benchmarks* das suítes Mibench e MediaBench, junto com a quantidade de instruções que cada aplicação executa. Dentre estes 55 *benchmarks*, somente 27 atendem ao requisito de executarem 100 milhões de instruções e estão marcados na tabela.

Tabela 3.5: Lista de *benchmarks* da suíte MiBench e MediaBench.

Suíte	N	*Benchmark*	# de Instruções	Selecionado
MiBench	0	basicmath_small	1.361.274.783	✓
	1	basicmath_large	22.269.199.867	✓
	2	bitcount_small	45.593.674	
	3	bitcount_large	684.250.382	✓
	4	qsort_small	14.412.623	
	5	qsort_large	989.374.482	✓
	6	susan_small_corners	3.458.872	
	7	susan_small_edges	6.887.633	
	8	susan_small_smoothing	35.320.189	
	9	susan_large_corners	44.586.473	
	10	susan_large_edges	177.422.307	✓
	11	susan_large_smoothing	423.392.198	✓
	12	jpeg_small_encode	29.474.823	
	13	jpeg_small_decode	8.697.311	
	14	jpeg_large_encode	109.076.355	✓
	15	jpeg_large_decode	29.353.751	
	16	lame_small	8.033.703.484	✓
	17	lame_large	94.314.753.652	✓
	18	dijkstra_small	59.353.158	
	19	dijkstra_large	285.280.151	✓
	20	patricia_small	289.205.289	✓
	21	patricia_large	1.830.947.169	✓
	22	stringsearch_small	279.725	
	23	stringsearch_large	6.967.897	
	24	rijndael_small_encode	33.715.298	
	25	rijndael_small_decode	34.684.744	
	26	rijndael_large_encode	351.060.637	✓
	27	rijndael_large_decode	361.155.494	✓
	28	sha_small	13.036.287	
	29	sha_large	135.696.013	✓
	30	adpcm_small adpcm	34.628.836	
	31	adpcm_small pcm	27.256.674	
	32	adpcm_large adpcm	688.972.768	✓
	33	adpcm_large pcm	538.659.721	✓
	34	CRC32_small	31.642.633	
	35	CRC32_large	615.051.421	✓
	36	FFT_small	760.568.629	✓
	37	FFT_small_inv	1.822.953.522	✓
	38	FFT_large	15.244.218.367	✓
	39	FFT_large_inv	14.750.402.915	✓
	40	gsm_small_encode	32.662.402	
	41	gsm_small_decode	9.612.897	
	42	gsm_large_encode	1.763.290.739	✓
	43	gsm_large_decode	523.196.626	✓
MediaBench	44	adpcm_encode	7.491.531	
	45	adpcm_decode	5.902.015	
	46	gsm_encode	220.748.262	✓
	47	gsm_decode	61.040.935	
	48	jpeg_encode	16.795.923	
	49	jpeg_decode	5.205.413	
	50	mpeg2_encode	11.473.635.447	✓
	51	mpeg2_decode	3.771.530.154	✓
	52	pegwit_generate	12.611.338	
	53	pegwit_encrypt	30.667.604	
	54	pegwit_decrypt	17.497.246	

3.5 Seleção de Instruções

Esta seção descreve a metodologia adotada na seleção das instruções dos modelos gerados pelo ACSIM. Os 27 *benchmarks* das suítes Mibench e Mediabench selecionados na Tabela 3.5 foram utilizados, resultando em um levantamento das instruções mais utilizadas por estas aplicações. Isto é importante para descobrir quais são as instruções mais importantes e permitir a análise e otimização de suas respectivas rotinas de emulação.

3.5.1 Metodologia

Inicialmente os modelos foram gerados pelo ACSIM com a opção `-stats` habilitada. Esta opção permite que os simuladores coletem a informação de quantas e quais instruções a aplicação emulada executa. Como não será realizada nenhuma alteração nestas aplicações, a quantidade de instruções a serem executadas pelos simuladores para cada aplicação será determinístico, logo, será realizada somente uma execução para cada *benchmark*.

Com os dados de utilização das instruções, foi criado um *script* para separar as instruções que faziam parte do grupo necessário para cobrir 90% da execução do *benchmark*. Este valor foi escolhido por abranger grande parte da execução sem apresentar uma enorme quantidade de instruções. O objetivo desta seleção é limitar a quantidade de instruções que devem ser analisadas a fim de se obter um melhor ganho de desempenho. Pois, analisar uma quantidade elevada de instruções gasta um tempo muito grande e se estas instruções não forem bastante executadas, o ganho de desempenho obtido em suas otimizações será mínimo.

Este *script* calcula a porcentagem de execução de cada tipo de instrução em determinado aplicativo e depois ordena as instruções de forma decrescente, realizando a soma dessas porcentagens até atingir os 90% de cobertura. Estas instruções selecionadas são separadas em listas para cada um dos aplicativos, sendo ao final contabilizadas, resultando na quantidade de *benchmarks* dos quais a instrução faz parte e que compreende os 90%. No final é gerado como saída uma lista contendo as instruções mais utilizadas e ordenadas em relação à sua participação nestes grupos de 90% de cobertura. Logo, as instruções que possuem cobertura de 90% em mais aplicativos ficam em posições mais elevadas, indicando uma importância em relação as demais instruções. Esta abordagem foi escolhida para evitar que as aplicações que demoram mais tempo e, por consequência, executam mais instruções dominem e determinem quais são as instruções mais executadas.

3.5.2 Resultados Experimentais

A Tabela 3.6 apresenta as instruções e a quantidade de aplicações (`N`) em que estas instruções fazem parte do grupo que cobre 90% da execução total da aplicação para cada um dos modelos analisados. Nesta tabela aparecem somente as instruções que fizeram parte de algum grupo de cobertura de algum *benchmark*.

Resumindo os dados da tabela 3.6, tem-se que: o modelo ARM possui 100 instruções e somente 27 instruções são necessárias para cobrir 90% da execução dos *benchmarks*. Já o modelo MIPS possui 59 instruções e somente 34 aparecem. Para o modelo PowerPC

Tabela 3.6: Instruções necessárias para cobrir 90% da execução dos *benchmarks*.

ARM		MIPS		PowerPC		SPARC	
Instrução	N	Instrução	N	Instrução	N	Instrução	N
b	23	addu	27	addi	27	add_imm	26
ldr1	22	lw	25	bc	27	add_reg	26
mov1	22	addiu	24	rlwinm	26	subcc_imm	25
add3	20	sll	24	lwz	23	or_reg	24
cmp3	19	beq	23	cmpi	22	sll_imm	24
add1	18	nop	22	ore	22	subcc_reg	23
sub3	18	sw	22	stw	22	be	20
str1	17	bne	20	add	16	bne	20
mov3	16	srl	18	cmp	15	ld_imm	20
orr1	13	and	15	bclr	14	or_imm	20
cmn3	11	or	15	stwu	14	sethi	19
stm	11	lui	14	bl	13	srl_imm	19
ldm	10	sltu	14	cmpl	13	st_imm	19
cmp1	9	ori	8	mfcr	13	and_imm	18
ldr2	6	andi	7	srawi	8	ba	17
ldrb1	6	sra	7	lbz	7	bleu	17
and3	5	subu	7	lwzx	7	jmpl_imm	15
eor1	4	sltiu	6	b	6	and_reg	14
rsb1	4	j	5	xxor	6	call	14
strb1	4	lbu	5	mullw	5	ldd_imm	13
mla	2	mflo	5	subf	5	std_imm	13
bic3	1	xor	4	addis	4	save_imm	12
ldrb2	1	lh	4	cmpli	4	ld_reg	9
mul	1	mult	4	lhax	4	sra_imm	7
orr3	1	slt	4	extsh	3	addcc_imm	6
sub1	1	bltz	3	lha	3	bgu	6
tst3	1	slti	3	mtspr	2	ldub_reg	6
		xori	3	sth	2	bl	5
		jr	2	adde	1	ble	5
		lhu	2	addic	1	xor_reg	5
		sh	2	ande_	1	ldub_imm	4
		bgez	1	andi_	1	restore_reg	4
		jal	1	lbzx	1	smul_reg	4
		lb	1	mfspr	1	ldsh_reg	3
				neg	1	lduh_reg	3
				ore_	1	st_reg	3
				ori	1	sth_reg	3
				rlwinm_	1	sub_reg	3
				slw	1	subx_imm	3
				sraw	1	addx_imm	2
				subfc	1	andcc_imm	2
				subfe	1	bg	2
				subfic	1	ldsh_imm	2
				xori	1	lduh_imm	2
						andcc_reg	1
						bneg	1
						bpos	1
						ldsb_reg	1
						nop	1
						orcc_reg	1
						sra_reg	1
						stb_reg	1
						sth_imm	1

temos 181 instruções disponíveis mas só 44 são mais utilizadas. Por fim, o modelo SPARC disponibiliza 119 instruções e somente 53 compõem a cobertura.

3.6 Análise de Desempenho via *MicroBenchmarks*

Esta seção apresenta a metodologia e os dados obtidos na análise das instruções selecionadas na Seção 3.5 para o modelo MIPS. Estes dados serviram de apoio para diagnosticar o comportamento dos simuladores gerados automaticamente pelo ACSIM.

3.6.1 Metodologia

Inicialmente foi desenvolvido um conjunto de *microbenchmarks* com as 34 instruções selecionadas. O *microbenchmark* consiste em executar 100 milhões de vezes a mesma instrução. Desta forma, é possível obter os dados em relação se determinada instrução está sendo emulada de forma eficiente, bem como se o simulador encontra-se otimizado. Após a criação destes *microbenchmarks*, foi realizado a execução para cada uma das instruções selecionadas e foi utilizado o `Perf` para coleta de dados.

3.6.2 Resultados Experimentais

A Tabela 3.7 apresenta os dados coletados com a execução destes *microbenchmarks*. Cada linha da tabela apresenta a instrução que foi executada 100 milhões de vezes; em quantos ciclos a simulação foi executada; a quantidade de instruções que a máquina hospedeira (IH) executou durante a simulação; o tempo gasto e as quantidades totais e de faltas na *cache* e, por fim, a quantidade total e de erros de predição de desvios. Pelos dados desta tabela, nota-se que a maioria das instruções da aplicação hóspede comportam-se de forma similar, porém, há alguns itens que se destacam e serão explicados nas próximas figuras.

A Figura 3.1 apresenta o gráfico da velocidade da simulação obtido por cada uma das instruções selecionadas. Nesta figura, pode-se ver que o desempenho das instruções está muito próximo um dos outros, exceto pelas instruções de leitura e escrita em memória e instruções de desvio condicionais. Outro item a ser notado é a instrução `nop` que apresenta a maior velocidade, representando a sobrecarga do mecanismo de busca, decodificação e despacho de rotinas de emulação do simulador, visto que esta instrução não executa nenhum comportamento em si, ou seja, não faz nada em específico.

Já a Figura 3.2 apresenta o gráfico da taxa de erros de predição de desvio na execução das instruções. Neste caso, pode-se ver que emular instruções de desvio condicionais (`beq, bgez, bltz, bne`) são uma das principais causas dos erros de predição. Isto já era esperado por causa do fato da aplicação emulada em execução acrescentar um novo desvio além dos desvios que o simulador já executa em seu próprio funcionamento na simulação, prejudicando o preditor de desvio. Um fato interessante constatado neste gráfico é acerca das instruções `sltu` e `sltiu` que não são instruções de desvio condicionais e apresentaram mais erros que as demais instruções e, neste caso, serão analisadas posteriormente.

A Figura 3.3 mostra o gráfico com o percentual de faltas na *cache* física do processador em relação à quantidade total de acessos ao executar os *microbenchmarks* das instruções

Tabela 3.7: Resultados dos *microbenchmarks* das instruções selecionadas.

Instrução	IH	Tempo (s)	Acessos à *Cache*		Desvios	
			Total	Faltas	Total	Erros
addiu	6.342.518.394	0,91	696.112	1.997	1.529.826.345	1.242.416
addu	6.442.949.475	0,94	1.253.194	306	1.529.925.106	1.237.407
and	6.443.055.134	0,93	511.380	1.733	1.529.844.006	1.237.966
andi	6.442.604.659	0,90	573.660	1.780	1.529.957.244	1.240.596
beq	6.079.101.014	1,11	1.712.261	963	1.460.969.544	23.424.752
bgez	5.979.164.435	1,04	1.482.401	630	1.460.984.026	23.480.072
bltz	5.979.119.534	1,08	1.813.940	203	1.460.981.070	23.165.968
bne	6.079.122.274	1,06	2.473.426	285	1.460.973.401	23.536.401
j	5.992.850.521	0,88	2.411.855	760	1.429.930.408	1.601.383
jal	6.192.912.786	0,94	7.786.614	407	1.429.922.639	1.499.735
lb	9.842.590.428	1,40	223.969	302	2.229.811.143	1.238.556
lbu	9.842.593.735	1,34	487.612	248	2.229.805.369	1.238.331
lh	12.142.700.988	1,46	592.788	322	2.829.839.434	1.238.809
lhu	12.142.755.834	1,48	595.188	324	2.829.829.806	1.238.680
lui	6.242.640.834	0,86	1.238.156	199	1.529.865.924	1.239.789
lw	10.242.647.859	1,35	854.150	212	2.129.828.153	1.237.973
mflo	6.243.637.637	0,87	472.726	424	1.530.112.841	1.446.533
mult	6.843.598.707	0,96	961.948	168	1.530.089.490	1.248.916
nop	5.443.148.136	0,79	2.280.601	167	1.329.993.165	1.245.093
or	6.443.091.944	0,93	215.024	215	1.529.849.864	1.236.508
ori	6.442.608.249	0,89	825.332	242	1.529.966.576	1.248.029
sh	12.542.900.604	1,54	417.367	285	2.829.875.536	1.239.243
sll	6.443.382.845	0,90	1.155.231	194	1.530.031.950	1.246.432
slt	6.643.020.289	0,93	341.073	202	1.629.841.017	5.542.684
slti	6.542.535.612	0,91	171.578	221	1.629.830.840	1.237.811
sltiu	6.542.572.969	0,99	198.479	199	1.629.958.142	10.185.544
sltu	6.643.086.860	1,00	701.185	150	1.629.945.032	1.237.292
sra	6.443.436.113	0,90	316.184	186	1.530.047.283	1.297.309
srl	6.443.415.458	0,93	532.632	229	1.530.039.312	1.251.872
subu	6.442.984.294	0,95	701.397	183	1.529.930.397	1.236.287
sw	9.742.737.938	1,28	624.784	211	2.129.850.990	1.237.733
xor	6.443.134.180	0,98	475.722	198	1.529.856.456	1.236.626
xori	6.442.638.769	0,90	151.371	217	1.529.976.006	1.248.006

selecionadas. Neste caso, não é possível tirar nenhuma conclusão, visto que os *microbenchmarks* são pequenos e a taxa de faltas apresentadas foram baixas (menores que 0,35%).

Por fim, a Figura 3.4 mostra o gráfico da relação entre a quantidade de instruções executadas na máquina hospedeira pela quantidade de instruções da aplicação hóspede – que será chamado de relação IH/IG – e a sobrecarga causada pelo despachante do simulador. Neste gráfico, pode-se notar que, no geral, a maioria das instruções utilizam, em média, a mesma quantidade de instruções para serem simuladas. Porém, duas diferenças devem ser notadas neste gráfico. A primeira é em relação às instruções de leitura e escrita, que

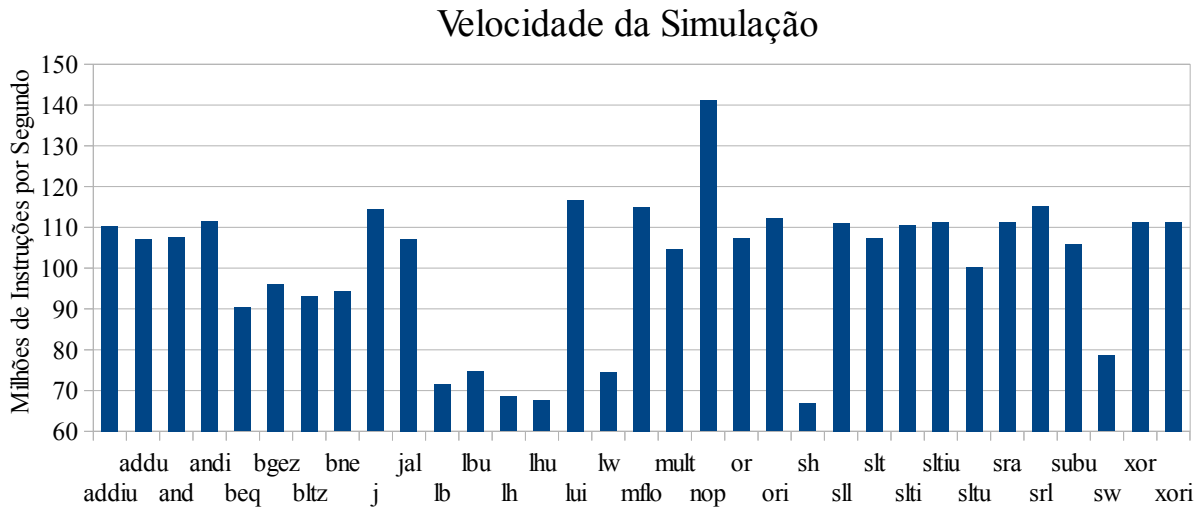


Figura 3.1: Desempenho das instruções selecionadas do modelo MIPS.

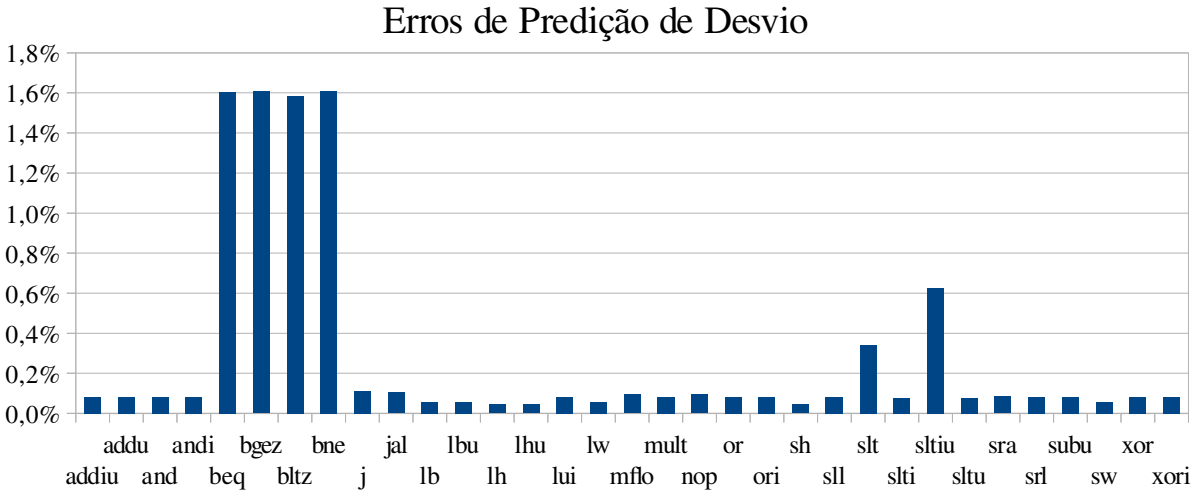


Figura 3.2: Taxa de erros de predição das instruções selecionadas do modelo MIPS.

apresentam um número muito maior que as demais e devem ser analisadas separadamente. A outra refere-se à instrução `nop` que apresentou a menor relação IH/IG e representa a quantidade de instruções gasta pelo despachante para gerenciar a simulação.

A instrução `nop` não simula nenhum comportamento, logo, os dados apresentados por esta instrução representam somente a execução do código do simulador que realiza a busca, decodificação e despacho para a rotina de emulação da instrução. Neste caso, esta parte do código do simulador utiliza em média 54 instruções. Portanto, para contabilizar de forma correta a quantidade de instruções da máquina hospedeira utilizada somente para emular o comportamento da instrução da aplicação hóspede, deve-se subtrair do valor mostrado no gráfico o valor obtido na instrução `nop`. Por exemplo, conforme mostrado pelo gráfico, a instrução `j` utiliza 60 instruções em média, logo, subtraindo-se 54 tem-se

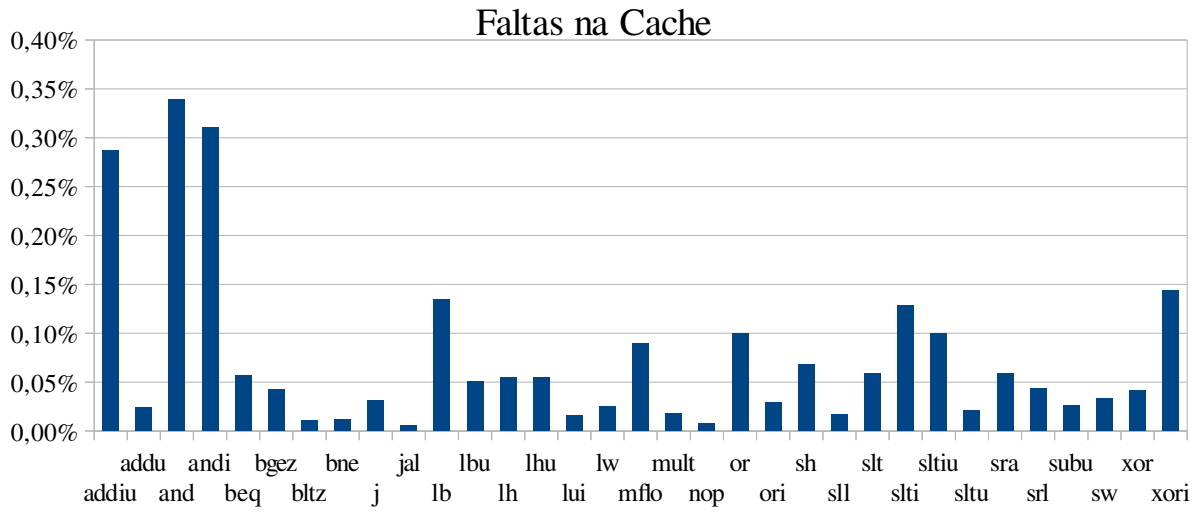


Figura 3.3: Taxa de faltas na *cache* das instruções selecionadas do modelo MIPS.

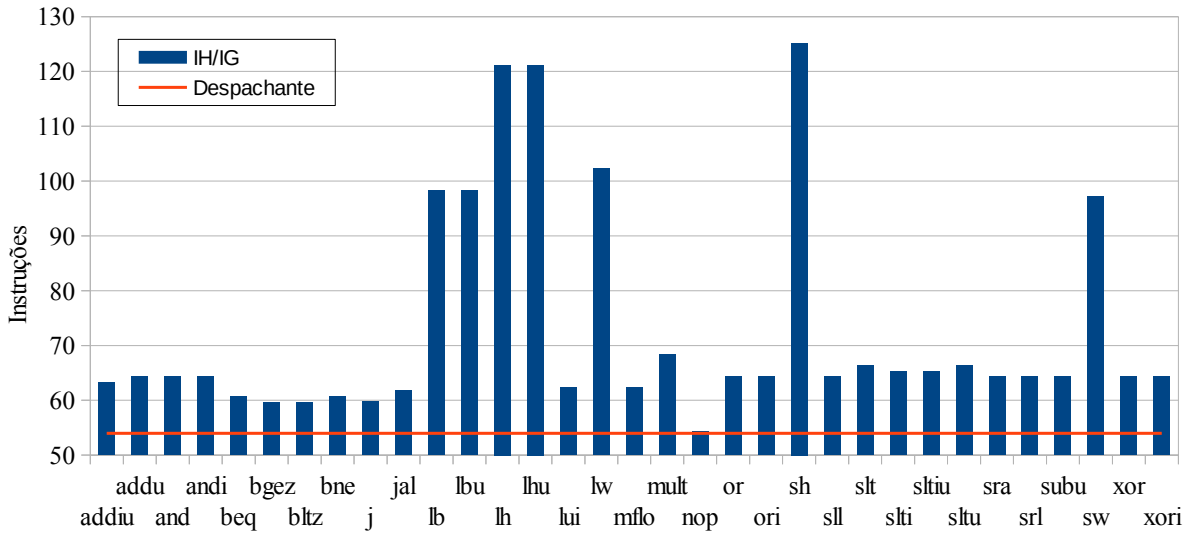


Figura 3.4: Quantidade de instruções da máquina hospedeira para emular uma instrução da aplicação hóspede e a sobrecarga do despachante.

que somente 6 instruções são utilizadas efetivamente na emulação da instrução `j`.

A maioria das instruções selecionadas do modelo MIPS do ArchC apresentaram um desempenho similar entre si. Isto indica que suas implementações não possuem grandes diferenças, sendo que estas instruções apresentaram baixa relação IH/IG, conclui-se que não há grandes problemas em suas implementações. Ao se retirar a sobrecarga produzida pela busca, decodificação e despacho (aproximadamente 54 instruções da máquina hospedeira) das instruções a serem emuladas do simulador, pode-se ver que a maioria das instruções gastam entre 5 a 14 instruções da máquina hospedeira para emular o comportamento da instrução da aplicação hóspede. Porém, alguns itens devem ser investigados.

Primeiramente, devem ser analisadas as instruções de leitura e escrita, pois apresen-

taram uma elevada relação IH/IG – de 43 a 71 instruções – e, consequentemente, um pior desempenho. Outro item que deve ser analisado é referente à alta ocorrência de erros de predição de desvio de algumas instruções que não emulam desvios condicionais. Por fim, será verificado o código do simulador, a fim de avaliar se as 54 instruções utilizadas na busca, decodificação e despacho das instruções a serem emuladas são realmente necessárias na simulação.

3.7 Análise dos *Benchmarks* MiBench e MediaBench

Esta seção apresenta os dados obtidos com o perfilamento das suítes MiBench e MediaBench sendo executadas no modelo MIPS.

3.7.1 Metodologia

Os *benchmarks* selecionados na Tabela 3.5 da Seção 3.4 foram executados no simulador do modelo MIPS e a partir disso, utilizando-se a ferramenta `Perf`, foram coletados os dados de perfilamento destas execuções. Cada um dos *benchmarks* foram executados pelo menos 3 vezes e os resultados apresentam a média destas execuções.

3.7.2 Resultados Experimentais

A Tabela 3.8 apresenta os dados obtidos com o perfilamento. Esta tabela apresenta o *benchmark* (B) executado; a quantidade de instruções que a máquina hospedeira executou (IH); a quantidade de instruções da aplicação hóspede emuladas (IG); a quantidade total de acessos e de faltas na *cache* L1 de dados e a quantidade total e de erros de predição de desvios e, por fim, o tempo total gasto na simulação. Todas as colunas estão em milhões, exceto as colunas do *benchmark* e do tempo.

A partir dos dados da Tabela 3.8 foram plotados os gráficos a seguir. O primeiro – Figura 3.5 – apresenta a velocidade da simulação de cada um destes *benchmarks* no modelo MIPS. Pelo gráfico, observa-se que este simulador apresenta uma velocidade média de 75 milhões de instruções por segundo. O *benchmark* 3 (bitcount_large) foi o que apresentou a maior velocidade (95,33), no entanto, os *benchmarks* 26 e 27 (rijndael_large_encode/decode) foram os que apresentaram as menores velocidades – 62,65 e 63,33 milhões de instruções por segundo, respectivamente. Mas, no geral, a maioria dos *benchmarks* ficaram próximos a 70 milhões de instruções por segundo em termos de velocidade.

Já a Figura 3.6 apresenta a relação entre as instruções da máquina hospedeira e as instruções da aplicação hóspede. Neste gráfico fica claro porque o *benchmark* 3 foi o que obteve a maior velocidade. A maioria dos *benchmarks* executa em torno de 70 ou mais instruções nativas para cada instrução da aplicação hóspede, já o *benchmark* 3 é um dos que executam menos de 70 instruções. Os resultados indicam que este *benchmark* utiliza, em média, instruções mais eficientes que os demais, proporcionando uma velocidade maior. Ainda neste gráfico, observa-se que o *benchmark* 32 (adpcm_large_encode) apresenta a menor relação IH/IG dentre todos os *benchmarks* testados, porém, somente isto não foi

Tabela 3.8: Dados do perfilamento das suítes MiBench e MediaBench.

B	IH	IG	Acessos à *Cache* L1		Desvios		Tempo
			Total	Faltas	Total	Erros	(s)
0	95.718	1.361	58.021	1.343,81	22.266	771	18,68
1	1.601.015	22.269	963.963	20.066,45	369.865	12.983	313,05
3	46.818	684	28.807	0,40	11.021	152	7,18
5	70.869	989	42.780	878,55	16.424	532	13,53
10	12.569	177	7.677	178,09	2.934	94	2,42
11	29.795	423	18.468	10,38	7.005	193	5,29
14	8.064	109	4.846	24,85	1.859	35	1,32
16	569.583	8.034	344.025	8.578,20	132.101	4.744	111,51
17	6.702.872	94.315	4.045.041	99.191,90	1.553.293	56.093	1315,72
19	20.300	285	12.263	30,21	4.685	94	3,30
20	20.958	289	12.589	266,55	4.833	173	4,18
21	132.604	1.831	79.660	1.671,16	30.581	1.092	26,25
26	27.119	351	16.305	720,89	6.150	242	5,60
27	27.689	361	16.711	721,66	6.314	247	5,70
29	9.784	136	5.980	6,50	2.258	34	1,52
32	46.793	689	28.512	13,73	11.148	293	8,18
33	37.751	539	22.892	7,97	8.912	240	6,74
35	47.381	615	28.202	8,21	10.808	170	7,32
36	54.180	761	32.692	876,45	12.552	436	10,58
37	129.752	1.823	78.313	2.108,58	30.063	1.046	25,31
38	1.085.953	15.244	655.225	17.258,86	251.531	8.753	211,57
39	1.049.743	14.750	633.572	16.143,15	243.214	8.453	203,41
42	138.013	1.763	81.378	2.064,68	31.881	889	24,66
43	37.896	523	22.753	52,82	8.959	184	6,25
46	17.315	221	10.203	260,22	3.999	112	3,13
50	829.159	11.474	499.655	9.847,44	191.529	6.702	159,90
51	271.314	3.772	163.477	3.785,75	62.623	2.278	53,16

o suficiente para torná-lo o *benchmark* mais rápido, visto que este conseguiu também ser superado pelos *benchmarks* 29 (sha_large) e 19 (dijkstra_large).

Continuando, a Figura 3.7 apresenta o gráfico com o percentual de faltas em relação à quantidade total de acessos na *cache* L1 de dados. Neste gráfico pode-se notar porque os *benchmarks* 26 e 27 foram os que tiveram as menores velocidades. Estes *benchmarks* foram os que apresentaram as maiores taxas de faltas – respectivamente, 4,42% e 4,32%. Além disso, este dois também estão no grupo dos *benchmarks* que apresentaram relação IH/IG acima de 76 instruções – vide Figura 3.6. Voltando para as falhas na *cache* L1, pode-se separar os *benchmarks* em 3 grupos: O primeiro formado pelos *benchmarks* 26 e 27 que superam 4,0%, o segundo com 17 *benchmarks* que ficaram entre 1,5% e 3,0% e, por último, os 9 restantes que não superam 0,5% de faltas. No geral, aparentemente não há grandes problemas no acesso à *cache* L1 dos *benchmarks*, porém será realizada uma análise mais detalhada na Seção 4.2 por conta dos *benchmarks* 26 e 27.

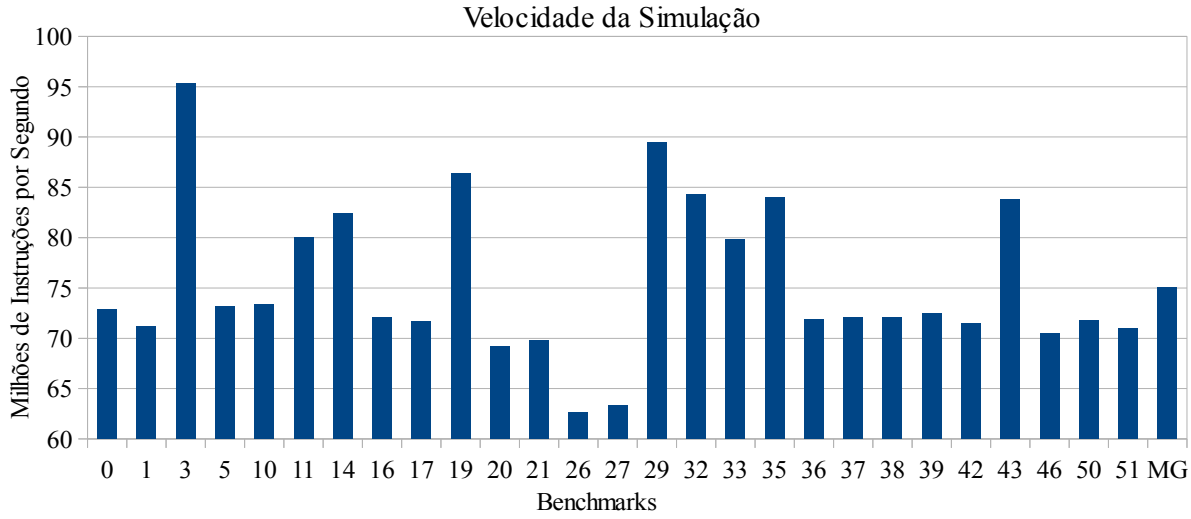


Figura 3.5: Desempenho do modelo MIPS nas suítes MiBench e MediaBench.

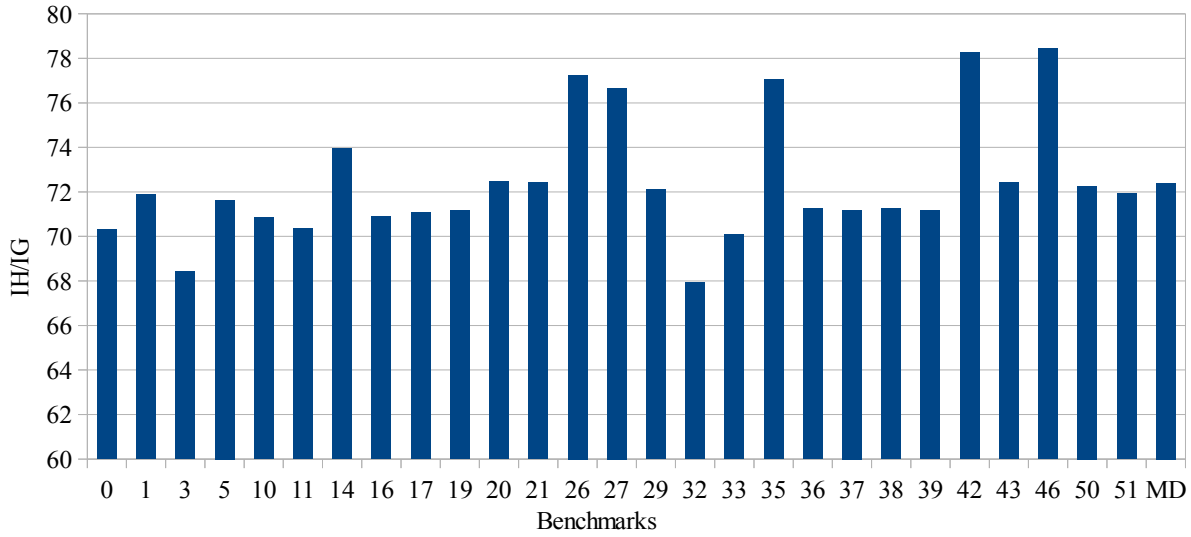


Figura 3.6: Relação IH/IG do modelo MIPS.

Por fim, a Figura 3.8 mostra o gráfico contendo o percentual de erros de predição de desvio em relação à quantidade total de desvios executados. Neste gráfico, observa-se que cada *benchmark* possui um comportamento distinto, porém no geral, apresentam baixa taxa de erros de predição (abaixo de 4%), indicando não haver grandes problemas neste item. Os *benchmarks* 26 e 27 aparecem novamente com as maiores taxas de erros de predição – 3,93% e 3,92%, respectivamente. Já em relação ao *benchmark* 32 que apresentou a menor relação IH/IG e não apresentou velocidade compatível com esse indicador, pode-se ver que, em comparação com o *benchmark* 3, este apresentou uma taxa maior de erros de predição, limitando, assim, sua velocidade.

Concluindo a análise, tem-se que a execução dos *benchmarks* das suítes MiBench e MediaBench mostraram que alguns pontos precisam ser analisados nos simuladores

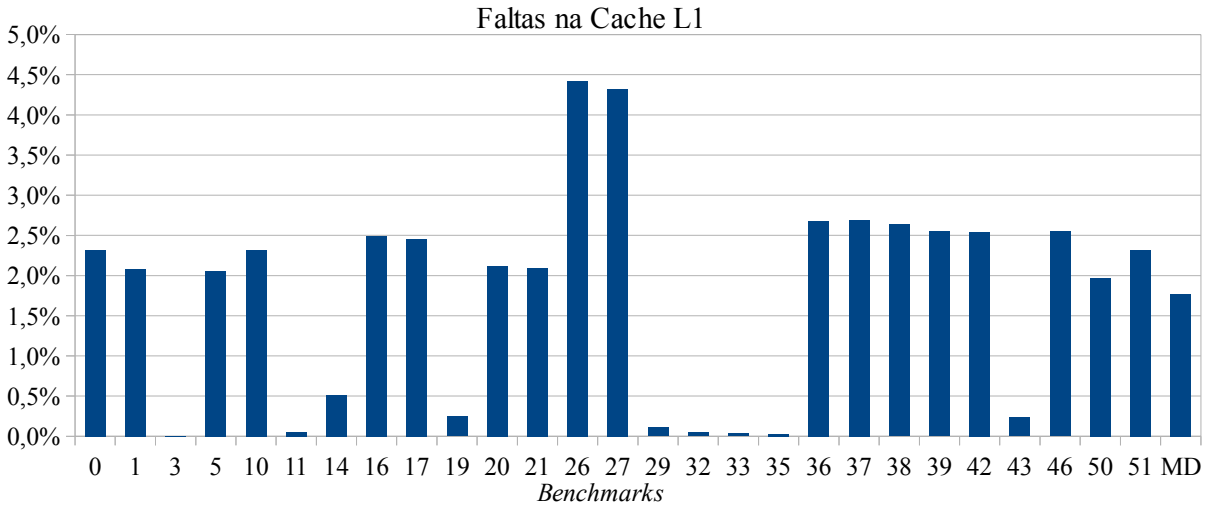


Figura 3.7: Taxa de faltas na *cache* L1 de dados na execução do simulador do modelo MIPS.

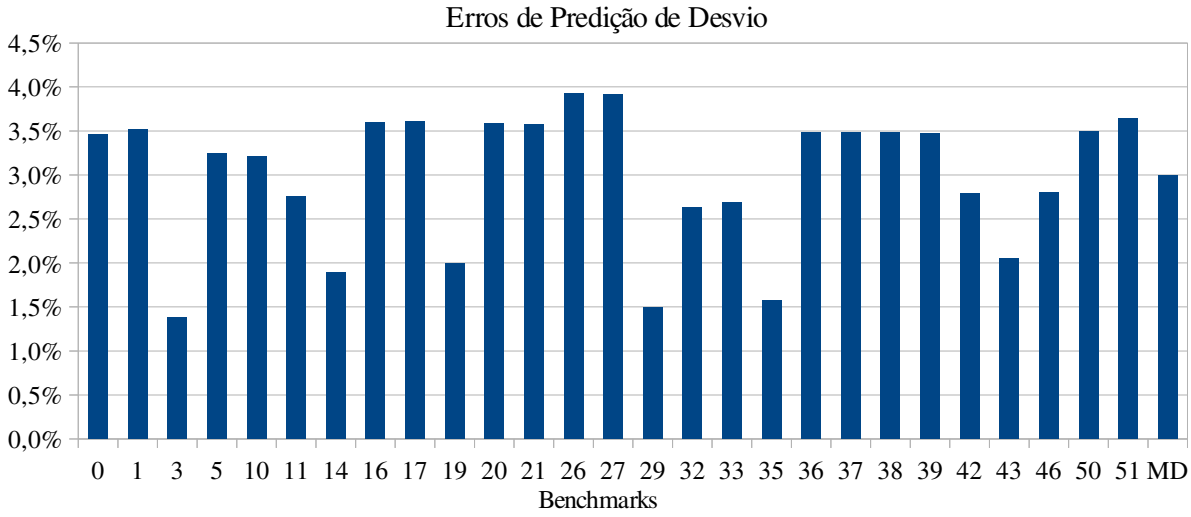


Figura 3.8: Taxa de erros de predição de desvio na execução do simulador do modelo MIPS.

gerados pelo ACSIM. O primeiro ponto refere-se ao elevado valor da relação IH/IG que, em média, apresentam 72 instruções da arquitetura hospedeira para cada instrução da aplicação hóspede. O segundo ponto refere-se aos *benchmarks* 26 e 27 que apresentaram os piores resultados em quase todos os indicadores medidos, com destaque para a elevada taxa de faltas na *cache* L1 em relação aos demais *benchmarks*. Logo, será necessário a inspeção destes itens no código do simulador.

3.8 Conclusões

Neste capítulo foi realizado uma análise sobre o custo das etapas de configuração e término dos modelos gerados pelo ACSIM, chegando-se a conclusão de que são necessários pelo menos 100 milhões de instruções da aplicação hóspede para avaliar o desempenho da etapa de emulação. A partir desse resultado, foram selecionados 27 dos 55 *benchmarks* e foram identificadas as instruções que são executadas com mais frequência. Com a análise dessas instruções e das 27 aplicações das suítes MiBench e MediaBench foi constatado que a relação IH/IG das instruções estava muito elevada, especialmente em instruções de leitura e escrita em memória, e que o simulador do modelo MIPS gasta 54 instruções em média para gerenciar a simulação, ou seja, para realizar a busca, decodificação e o despacho da instrução a ser emulada. O capítulo a seguir realiza uma análise mais detalhada no código fonte dos simuladores, focando nestes pontos levantados.

Capítulo 4

Análise e Otimizações do Código dos Simuladores

No Capítulo 3 foram levantados alguns problemas de desempenho no simulador do modelo MIPS. Este simulador apresentou elevada relação IH/IG em instruções de leitura e escrita em memória, muitos erros de predição de desvio para algumas instruções e altas taxas de faltas na *cache* para alguns *benchmarks*. Isto torna importante uma análise mais detalhada do código que compõe o simulador.

O código dos simuladores gerados pelo ACSIM são compostos por 3 partes. A primeira parte consiste dos códigos das bibliotecas do ArchC e que são utilizados por todos os simuladores gerados. Esta parte de códigos será referenciado como código do ArchC e inclui as rotinas de leitura e escrita de dados em memória, a decodificação da instrução, a estrutura da *cache* de instruções decodificadas e demais códigos que funcionam de forma similar para todos os modelos. Já a segunda parte é composta pelo código que o ACSIM emite – composta principalmente pelo interpretador – e será denominado como código do simulador. A última parte refere-se aos códigos escritos pelo usuário que compõem os comportamentos comum, de formato e de instrução, que são específicos para cada modelo e será chamado de código de modelo.

Alterações no código ArchC são mais críticas, pois trata-se das bibliotecas utilizadas por todos os modelos, sendo que sua estabilidade é fundamental para o funcionamento geral. Logo, a prioridade são as análises em cima dos pontos identificados que pertencem principalmente à parte de código do ArchC e posteriormente será priorizado as demais partes do código do simulador. Inicialmente foi realizada a análise nos pontos identificados pelo perfilamento, começando pelas rotinas de leitura e escrita em memória e depois pela estrutura da *cache* de instruções decodificadas, que são códigos do ArchC. Por fim, foi analisado o código do simulador para avaliar possíveis otimizações na geração de código.

4.1 Rotinas de leitura e escrita em memória

Esta seção apresenta as análises e resultados obtidos sobre os experimentos realizados nos métodos de leitura e escrita utilizados pelos simuladores gerados pelo ACSIM. Pelos dados coletados na Seção 3.6, observou-se que as operações de escrita e leitura não estavam sendo

realizadas de forma eficiente, portanto, era necessário analisá-las detalhadamente.

O ArchC implementa instruções de leitura e escrita em memória de forma similar entre os modelos gerados. As rotinas que implementam o comportamento da instrução devem solicitar à classe `ac_memport` a ação que desejam efetuar sobre a memória através das funções `read` e `write`. Esta, por sua vez, efetua uma chamada à classe `ac_storage` das respectivas rotinas de leitura e escrita. Por fim, estas rotinas efetuam a ação requerida na memória. Isto é ilustrado na Figura 4.1 que exemplifica o funcionamento da instrução `lw` do modelo MIPS.

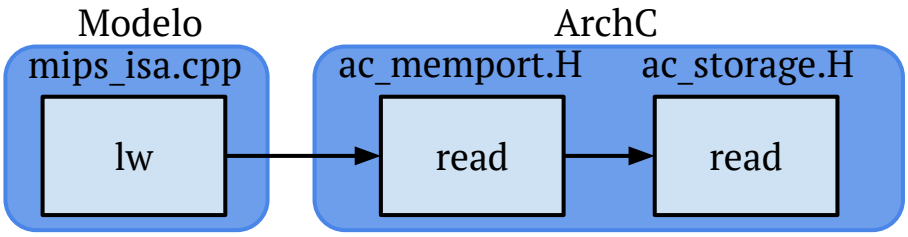


Figura 4.1: Diagrama de funcionamento da instrução `lw` do MIPS.

A seguir, serão apresentados os códigos fonte das classes envolvidas na leitura e escrita em memória. Estes códigos serão utilizados como apoio para explicar os pontos que foram identificados na análise de desempenho efetuada. O Código 4.1 mostra trechos de código da classe `ac_memport`. Já os Códigos 4.2 e 4.3 apresentam, respectivamente, a classe `ac_storage` e a implementação das suas rotinas de `read` e `write`.

```
1  template<typename ac_word, typename ac_Hword> class ac_memport :
2    public ac_arch_ref<ac_word, ac_Hword> {
3
4  private:
5    ac_inout_if* storage;
6    ac_word aux_word;
7    ac_Hword aux_Hword;
8    uint8_t aux_byte;
9
10 public:
11   ///Reads a word
12   inline ac_word read(uint32_t address) {
13     storage->read(&aux_word, address, sizeof(ac_word) * 8);
14     if (!this->ac_mt_endian) aux_word = byte_swap(aux_word);
15     return aux_word;
16   }
17   //!Writing a word
18   inline void write(uint32_t address, ac_word datum) {
19     aux_word = datum;
20     if (!this->ac_mt_endian) aux_word = byte_swap(datum);
21     storage->write(&aux_word, address, sizeof(ac_word) * 8);
22   }
23 }
```

Código 4.1: Código fonte da classe `ac_memport`.

```
1  class ac_storage : public ac_inout_if {
2
3  private:
4    ac_ptr data;
5
6  public:
7    void read(ac_ptr buf, uint32_t address, int wordsize);
8    void write(ac_ptr buf, uint32_t address, int wordsize);
9  }
```

Código 4.2: Código fonte da classe `ac_storage`.

```
1  void ac_storage::read(ac_ptr buf, uint32_t address, int wordsize) {
2    switch (wordsize) {
3    case 8: // unsigned char
4      *(buf.ptr8) = (data.ptr8)[address]; break;
5    case 16: // unsigned short
6      *(buf.ptr16) = *((uint16_t*) (data.ptr8 + address)); break;
7    case 32: // unsigned int
8      *(buf.ptr32) = *((uint32_t*) (data.ptr8 + address)); break;
9    case 64: // unsigned long long
10     *(buf.ptr64) = *((uint64_t*) (data.ptr8 + address)); break;
11   default: /* weird size */ break;
12   }
13 }
14 void ac_storage::write(ac_ptr buf, uint32_t address, int wordsize) {
15   switch (wordsize) {
16   case 8:  // unsigned char
17     (data.ptr8)[address] = *(buf.ptr8); break;
18   case 16: // unsigned short
19     *((uint16_t*) (data.ptr8 + address)) = *(buf.ptr16); break;
20   case 32: // unsigned int
21     *((uint32_t*) (data.ptr8 + address)) = *(buf.ptr32); break;
22   case 64: // unsigned long long
23     *((uint64_t*) (data.ptr8 + address)) = *(buf.ptr64); break;
24   default: /* weird size */ break;
25   }
26 }
```

Código 4.3: Rotinas `read` e `write` da classe `ac_storage`.

Analisando os códigos apresentados, é possível levantar alguns pontos interessantes. Primeiro, nas linhas de 6 a 8 do Código 4.1, pertencentes à classe `ac_memport`, são declaradas variáveis auxiliares que guardam cópias dos dados que foram lidos/escritos. Porém, estas cópias não são reutilizadas por nenhuma outra função. Isto faz com que o compilador tenha que gerar código para guardar esta cópia em memória de forma desnecessária. Segundo, nas linhas 14 e 20 do mesmo código, é verificada a necessidade de executar a

função `byte_swap` para tratamento de *endianness*. Esta computação também é desnecessária, visto que, os modelos não possuem instrução para trocar de *endianness* durante a execução.

Já no Código 4.2, da classe `ac_storage`, pode-se notar alguns possíveis problemas nos métodos `read` e `write`. Primeiro na própria declaração (linha 7) em que ao invés de retornar o dado lido através do retorno da função, o dado é retornado por parâmetro. Isto impede o compilador de fazer otimizações como manter o dado de retorno em registradores ao invés de guardá-lo na pilha. Por fim, no Código 4.3, referente às rotinas de leitura e escrita, observa-se que, como o tamanho dos dados varia de acordo com o tamanho da palavra definida no modelo, este método trata estas variações realizando um *switch case* dividindo em 4 tamanhos pré-definidos. Esta generalidade das rotinas `read/write` faz com que mais instruções sejam geradas.

O Código 4.4 apresenta o código em linguagem de montagem gerado pelo compilador gcc para a rotina de emulação da instrução `lw`. Neste código, o registrador `r12` é o ponteiro para o local onde estão armazenadas as variáveis auxiliares. Na linha 16 é realizada a chamada à rotina `read` da classe `ac_storage`. Nas linhas 20 e 42 ocorrem leituras no valor retornado pela rotina `read` e na linha 33 o valor é atualizado após a execução da função `byte_swap`. Pode-se ver que por se tratar de uma variável global no contexto da classe `ac_memport` o valor lido deve ficar em memória, prejudicando possíveis otimizações por parte do compilador.

```
1  00000000004078f0 <_ZN10mips_parms8mips_isa11behavior_lwEjjji>:
2    4078f0: mov    QWORD PTR [rsp-0x10],rbp
3    4078f5: mov    QWORD PTR [rsp-0x18],rbx
4    4078fa: movsxd rdx,edx
5    4078fd: mov    QWORD PTR [rsp-0x8],r12
6    407902: sub    rsp,0x18
7    407906: mov    r12,QWORD PTR [rdi+0xa0]
8    40790d: mov    rbx,QWORD PTR [rdi+0xa8]
9    407914: mov    ebp,ecx
10   407916: mov    ecx,0x20
11   40791b: mov    rdi,QWORD PTR [r12+0x98]
12   407923: lea    rsi,[r12+0xa0]
13   40792b: add    r8d,DWORD PTR [rbx+rdx*4+0x8]
14   407930: mov    rax,QWORD PTR [rdi]
15   407933: mov    edx,r8d
16   407936: call   QWORD PTR [rax]
17   407938: mov    rax,QWORD PTR [r12+0x38]
18   40793d: cmp    BYTE PTR [rax],0x0
19   407940: jne    407990
20   407942: mov    edx,DWORD PTR [r12+0xa0]
21   40794a: movzx  eax,dl
22   40794d: mov    ecx,edx
23   40794f: movzx  esi,dh
24   407952: shl    eax,0x8
25   407955: shr    ecx,0x10
26   407958: shr    edx,0x18
27   40795b: or     eax,esi
28   40795d: and    ecx,0xff
```

```
29   407963: shl    eax,0x8
30   407966: or     eax,ecx
31   407968: shl    eax,0x8
32   40796b: or     eax,edx
33   40796d: mov    DWORD PTR [r12+0xa0],eax
34   407975: movsxd rbp,ebp
35   407978: mov    DWORD PTR [rbx+rbp*4+0x8],eax
36   40797c: mov    rbx,QWORD PTR [rsp]
37   407980: mov    rbp,QWORD PTR [rsp+0x8]
38   407985: mov    r12,QWORD PTR [rsp+0x10]
39   40798a: add    rsp,0x18
40   40798e: ret
41   40798f: nop
42   407990: mov    eax,DWORD PTR [r12+0xa0]
43   407998: jmp    407975
44   40799a: nop    WORD PTR [rax+rax*1+0x0]
```

Código 4.4: Código gerado para a rotina de emulação da instrução `lw`.

Os processadores da família de arquiteturas x86, a partir do 486, passaram a conter uma instrução específica (`bswap`) para realizar a inversão de *bits*. O ArchC possui macros específicas para a inclusão ou não desta instrução. Porém, observa-se que isto não está funcionando de forma correta, pois são emitidas instruções no código em linguagem de montagem do método genérico de inversão de *bits* – linhas 21 a 32. Isto contribui muito para alta taxa da relação IH/IG das instruções de leitura e escrita – acréscimo de 12 instruções só nesta função. Além deste fato, nas linhas 17 a 19 é realizado a computação para verificar a necessidade de executar este tratamento, contribuindo com mais 3 instruções para o aumento da relação IH/IG.

Já o Código 4.5 apresenta o código em linguagem de montagem da rotina `read`. Neste código, pode-se ver que o grande problema são as comparações realizadas para determinar o código que deve ser executado. No pior dos casos são executadas 8 instruções para depois realizar a leitura do dado requisitado. Outro ponto confirmado foi a utilização de ponteiros para retornar o resultado.

```
1  0000000000417a40 <_ZN10ac_storage4readE6ac_ptrji>:
2    417a40: cmp    ecx,0x10
3    417a43: je     417a80
4    417a45: jle    417a58
5    417a47: cmp    ecx,0x20
6    417a4a: je     417a90
7    417a4c: cmp    ecx,0x40
8    417a4f: nop
9    417a50: je     417a70
10   417a52: repz ret
11   417a54: nop    DWORD PTR [rax+0x0]
12   417a58: cmp    ecx,0x8
13   417a5b: jne    417a52
14   417a5d: mov    rax,QWORD PTR [rdi+0x8]
15   417a61: mov    edx,edx
```

```
417a63: movzx  eax,BYTE PTR [rax+rdx*1]
417a67: mov    BYTE PTR [rsi],al
417a69: ret
417a6a: nop    WORD PTR [rax+rax*1+0x0]
417a70: mov    rax,QWORD PTR [rdi+0x8]
417a74: mov    edx,edx
417a76: mov    rax,QWORD PTR [rax+rdx*1]
417a7a: mov    QWORD PTR [rsi],rax
417a7d: ret
417a7e: xchg   ax,ax
417a80: mov    rax,QWORD PTR [rdi+0x8]
417a84: mov    edx,edx
417a86: movzx  eax,WORD PTR [rax+rdx*1]
417a8a: mov    WORD PTR [rsi],ax
417a8d: ret
417a8e: xchg   ax,ax
417a90: mov    rax,QWORD PTR [rdi+0x8]
417a94: mov    edx,edx
417a96: mov    eax,DWORD PTR [rax+rdx*1]
417a99: mov    DWORD PTR [rsi],eax
417a9b: ret
417a9c: nop    DWORD PTR [rax+0x0]
```

Código 4.5: Código gerado para a rotina `read` da classe `ac_storage`.

A instrução `lw` foi escolhida por se tratar de uma instrução que retrata bem a maior parte dos problemas relacionados à generalidade presente nos métodos de leitura e escrita em memória dos simuladores ArchC. Além disso, as demais instruções de manipulação de dados na memória funcionam de forma bastante similar, exceto pelos seguintes detalhes: Instruções que fazem manipulação em *byte* – `lb`, `lbu` e `sb` – não fazem uso das funções de tratamento de *endianness*, logo, as otimizações nestes métodos de tratamento não surtirão efeito nestas instruções. Já a otimização de especialização do retorno da função trata somente dos métodos de leitura, logo, instruções de escrita não sofrem alterações, exceto as instruções `swl` e `swr` que realizam uma leitura antes de realizar a escrita.

4.1.1 Metodologia

Após a identificação dos problemas nos métodos de leitura e escrita do ArchC, foram geradas diferentes versões e, com isso, foi analisado o desempenho de cada uma destas otimizações. Para análise desses métodos serão utilizados os mesmos *microbenchmarks* descritos na seção 3.6, porém foram selecionados somente os *microbenchmarks* das instruções de leitura (`lb`, `lbu`, `lh`, `lhu`, `lw`, `lwl` e `lwr`) e escrita (`sb`, `sh`, `sw`, `swl` e `swr`) em memória do modelo MIPS.

A primeira versão, denominada `v0`, trata-se do ArchC versão oficial 2.2 executado em conjunto com o modelo MIPS v1.0 sem qualquer alteração. As demais versões são derivadas desta configuração e foram realizadas de forma incremental. Portanto, tudo que foi implementado em uma determinada versão será base para a próxima versão.

A segunda versão, denominada `v1`, buscou atacar o problema da cópia local que é realizada na classe `ac_memport`. Esta versão consiste na remoção dos atributos `aux_word`, `aux_Hword` e `aux_byte` – linhas 6 a 8 – e na instanciação de variáveis locais nos respectivos métodos. Já a versão `v2` ataca o problema com o tratamento de *endianness*. Nesta versão ao invés de continuar utilizando os métodos específicos do ArchC foi utilizado os métodos da classe "endian.h". Além disso, foi removido a computação da comparação do *endianness* e incluído uma diretiva de pré-compilação para incluir ou não a instrução de tratamento.

Continuando, a versão `v3` ataca o problema da generalidade das rotinas `read` e `write`. Nesta versão ao invés de uma rotina para tratamento de variados tamanhos de dados, tem-se várias rotinas para tratamento de um tamanho de dado específico (8, 16, 32 ou 64 *bits*). Por fim, a última versão (`v4`) trata o caso específico de utilizar o retorno da função ao invés de retornar via parâmetro o dado a ser lido. Todas estas versões geradas estão resumidas na Tabela 4.1.

Tabela 4.1: Versões geradas para os métodos de leitura e escrita em memória

Otimização	v0	v1	v2	v3	v4
Versão Oficial	✓	✓	✓	✓	✓
Sem Cópia Local		✓	✓	✓	✓
Endianness			✓	✓	✓
Rotinas Especializadas				✓	✓
Retorno da Função					✓

4.1.2 Resultados Experimentais

O Código 4.6 mostra o código gerado para instrução `lw` com todas as otimizações aplicadas (`v4`). Neste código, observa-se que houve uma grande redução no número de instruções. Um dos grandes contribuidores para esta redução foi a utilização da instrução `bswap` para tratamento de *endianness*, e pode ser observada na linha 15. Ainda nesta linha, observa-se que foi utilizado o registrador `eax` como retorno da função read chamada na linha 14, ao invés de um apontador para a memória.

```
1  00000000004073d0 <_ZN10mips_parms8mips_isa11behavior_lwEjjji>:
2    4073d0: mov    QWORD PTR [rsp-0x8],rbp
3    4073d5: mov    QWORD PTR [rsp-0x10],rbx
4    4073da: sub    rsp,0x18
5    4073de: mov    rax,QWORD PTR [rdi+0xa0]
6    4073e5: mov    rbx,QWORD PTR [rdi+0xa8]
7    4073ec: movsxd rdx,edx
8    4073ef: mov    ebp,ecx
9    4073f1: movsxd rbp,ebp
10   4073f4: mov    rdi,QWORD PTR [rax+0x98]
11   4073fb: mov    esi,DWORD PTR [rbx+rdx*4+0x8]
12   4073ff: mov    rax,QWORD PTR [rdi]
13   407402: add    esi,r8d
```

```
14   407405: call   QWORD PTR [rax+0x10]
15   407408: bswap  eax
16   40740a: mov    DWORD PTR [rbx+rbp*4+0x8],eax
17   40740e: mov    rbx,QWORD PTR [rsp+0x8]
18   407413: mov    rbp,QWORD PTR [rsp+0x10]
19   407418: add    rsp,0x18
20   40741c: ret
21   40741d: nop
22   40741e: xchg   ax,ax
```

Código 4.6: Código otimizado gerado para a instrução lw.

Já o Código 4.7 mostra o código para a rotina `read` que trata a leitura de um dado de 8 *bits*. Este código representa a versão `v3` que realiza o retorno do valor lido através de parâmetro utilizando ponteiro. A diferença deste código para a versão `v4` consiste na exclusão da linha 5 que realiza a escrita no ponteiro armazenado em `rsi`. As demais rotinas que realizam a leitura de dados de 16, 32 e 64 *bits* são idênticas a este código, diferindo apenas no tamanho do dado lido na linha 4 e escrito na linha 5.

```
1  0000000000417360 <_ZN10ac_storage4readEPhj>:
2    417360: mov    rax,QWORD PTR [rdi+0x8]
3    417364: mov    edx,edx
4    417366: movzx  eax,BYTE PTR [rax+rdx*1]
5    41736a: mov    BYTE PTR [rsi],al
6    41736c: ret
7    41736d: nop
8    41736e: xchg   ax,ax
```

Código 4.7: Código otimizado gerado para a rotina `read` da classe `ac_storage` que realiza a leitura de 8 *bits*.

A Tabela 4.2 apresenta os dados de perfilamento das instruções de leitura e escrita. Nas linhas desta tabela são exibidas a instrução simulada, a versão em que a instrução foi executada, a quantidade de ciclos utilizada na simulação, a quantidade de instruções da arquitetura hospedeira utilizadas para simular 100 milhões de instruções da aplicação hóspede e o tempo de execução em segundos. Analisando cada instrução individualmente, observa-se que a cada versão os dados vão ficando menores, indicando que as otimizações propostas estão funcionando.

A Figura 4.2 apresenta a relação IH/IG. Neste gráfico, pode-se ver que a versão `v1`, como esperado, reduz em pouco a relação IH/IG – de 1 a 2 instruções. Porém, nas instruções (`sh`, `swl` e `swr`) houve um aumento nesta relação. Ao analisar o código gerado para estas instruções, foi identificado a causa deste aumento. Ao remover a cópia local, o compilador conseguiu alocar o resultado na pilha, porém, isto alterou o escalonamento de registradores. Ao alocar mais registradores, o compilador teve que salvar mais dados na pilha, gerando mais instruções para isso.

Tabela 4.2: Dados das instruções de leitura e escrita.

Instrução		Ciclos	IH	Tempo
lb	v0	4.683.754.035	9.842.574.043	1,31
	v1	4.618.967.246	9.642.546.696	1,29
	v2	4.640.324.718	9.642.532.585	1,30
	v3	4.200.834.151	7.742.429.945	1,18
	v4	3.989.038.754	7.542.404.419	1,12
lbu	v0	4.732.835.856	9.843.632.209	1,32
	v1	4.579.349.030	9.642.567.024	1,28
	v2	4.637.358.303	9.642.538.281	1,30
	v3	4.197.864.210	7.742.440.953	1,17
	v4	3.987.160.236	7.542.423.660	1,12
lh	v0	5.123.351.656	12.142.681.658	1,43
	v1	5.255.362.513	12.042.759.892	1,47
	v2	4.709.012.615	9.542.573.226	1,32
	v3	4.622.524.984	9.242.558.348	1,29
	v4	4.370.607.564	8.942.517.956	1,22
lhu	v0	5.197.832.959	12.142.720.497	1,45
	v1	5.237.783.517	12.042.822.131	1,46
	v2	4.617.147.850	9.542.631.666	1,29
	v3	4.651.241.002	9.242.575.550	1,30
	v4	4.406.679.233	8.942.553.858	1,23
lw	v0	4.679.369.387	10.242.621.531	1,31
	v1	4.599.880.205	10.242.610.931	1,29
	v2	4.210.817.320	8.542.509.117	1,18
	v3	4.203.334.193	7.942.503.954	1,18
	v4	4.006.856.164	7.642.481.112	1,12
lwl	v0	5.185.895.875	12.642.747.770	1,45
	v1	5.140.870.403	12.542.748.018	1,44
	v2	4.902.989.626	10.942.669.012	1,37
	v3	4.846.835.949	10.342.648.721	1,36
	v4	4.628.260.721	10.042.634.122	1,29
lwr	v0	5.298.657.565	12.942.899.867	1,48
	v1	5.211.984.993	12.842.775.241	1,45
	v2	4.943.163.194	11.242.696.344	1,38
	v3	4.858.920.687	10.642.681.603	1,36
	v4	4.575.870.296	10.342.633.276	1,28
sb	v0	4.186.573.426	8.942.613.463	1,17
	v1	4.184.060.722	8.942.609.463	1,17
	v2	4.192.439.299	8.942.579.931	1,17
	v3	3.814.573.869	7.142.513.048	1,07
	v4	3.820.154.253	7.142.522.423	1,07
sh	v0	5.350.730.700	12.542.863.543	1,50
	v1	5.373.752.515	12.642.861.969	1,50
	v2	4.213.996.851	8.742.607.379	1,18
	v3	4.063.531.392	8.042.573.534	1,14
	v4	4.046.977.666	8.042.577.408	1,13
sw	v0	4.454.419.770	9.742.702.043	1,25
	v1	4.443.663.471	9.742.705.504	1,24
	v2	4.174.902.491	8.142.622.961	1,17
	v3	3.841.495.034	7.242.553.239	1,08
	v4	3.804.217.934	7.242.661.604	1,06
swl	v0	6.837.239.520	16.343.204.764	1,91
	v1	7.297.661.575	16.543.292.506	2,03
	v2	6.142.226.335	13.443.037.572	1,71
	v3	5.654.403.552	11.642.927.316	1,58
	v4	5.119.884.944	11.442.853.554	1,43
swr	v0	6.797.562.426	16.343.209.758	1,90
	v1	7.191.036.092	16.543.282.465	2,01
	v2	6.185.929.938	13.543.170.903	1,73
	v3	5.637.111.042	11.742.947.766	1,57
	v4	5.092.683.342	11.542.851.668	1,42

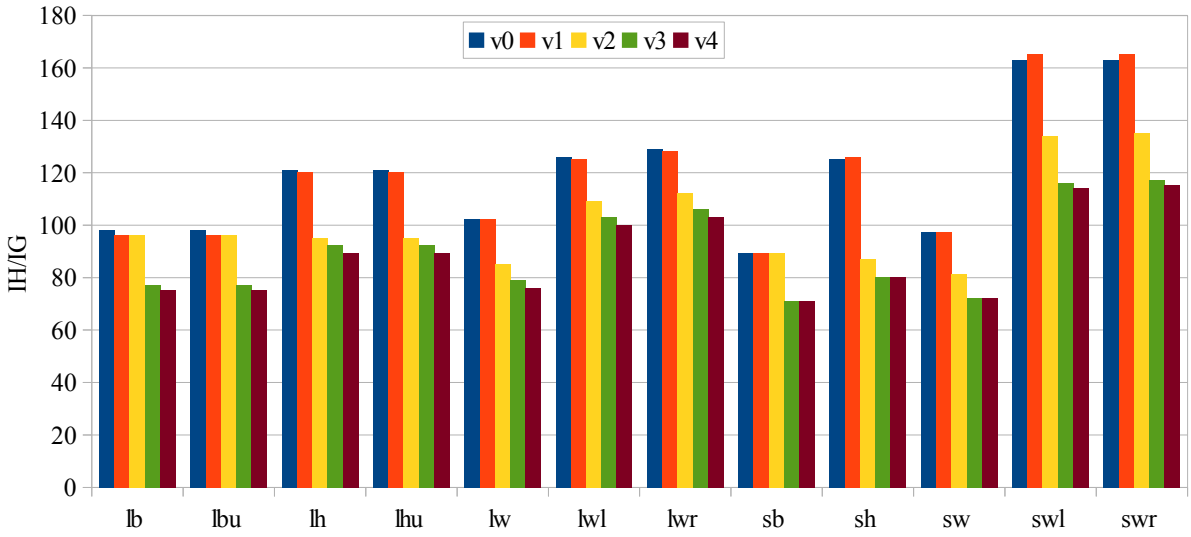


Figura 4.2: Relação IH/IG das instruções de leitura e escrita.

Já em relação a versão `v2`, observa-se que a utilização de instruções especializadas em inversão de *bits* consegue reduzir bastante a relação IH/IG. Conforme a análise de código, era esperado uma redução de pelo menos 15 instruções, porém, esta otimização conseguiu uma redução de 16 a 38 instruções. Esta redução se deve por causa da remoção da necessidade de preparar os dados para a execução da função genérica de inversão de *bits* e pelo menor uso de registradores, reduzindo assim a necessidade de ter que salvar dados na pilha. Por fim, as versões `v3` e `v4` que tratam de especialização de código não apresentaram nada a mais do que era esperado destas versões. A versão `v3` apresentou redução em todos os casos e a versão `v4` apresentou redução somente nas instruções que fazem uso da rotina `read`.

O gráfico da Figura 4.3 apresenta o desempenho, em milhões de instruções por segundo, obtido pelas instruções de leitura e escrita com as otimizações implementadas. Neste gráfico, nota-se que as instruções `lb`, `lbu` e `sb` possuem comportamento similar. Pouco ganho de desempenho na versão `v1` e nenhum ganho na versão `v2` – pelo simples motivo desta versão atacar um problema no tratamento de *endianess* e este tratamento não ser utilizado nestas instruções. Algum ganho de desempenho na versão `v3` para essas 3 instruções e um maior ganho na versão `v4` somente para as instruções de leitura. A instrução `sb` não teve nenhum ganho de desempenho na versão `v1` por que o método de leitura já não utilizava a cópia local. As instruções `swl` e `swr` foram as que obtiveram os maiores ganhos de desempenho – acima de 30%. Porém como executam tanto leitura quanto escrita em memória, foram as que obtiveram as menores velocidades.

A Figura 4.4 mostra quantas instruções da máquina hospedeira são executadas por ciclo de execução (IPC). Neste caso, um número baixo pode significar que o processador não conseguiu escalonar as instruções de forma a executá-las paralelamente em suas unidades de execução, ou seja, algumas unidades de execução ficaram sem ter o que executar, ou as instruções demoram bastante tempo para serem executadas. Normalmente isso ocorre quando há dependência de dados entre as instruções e estas não podem ser executadas

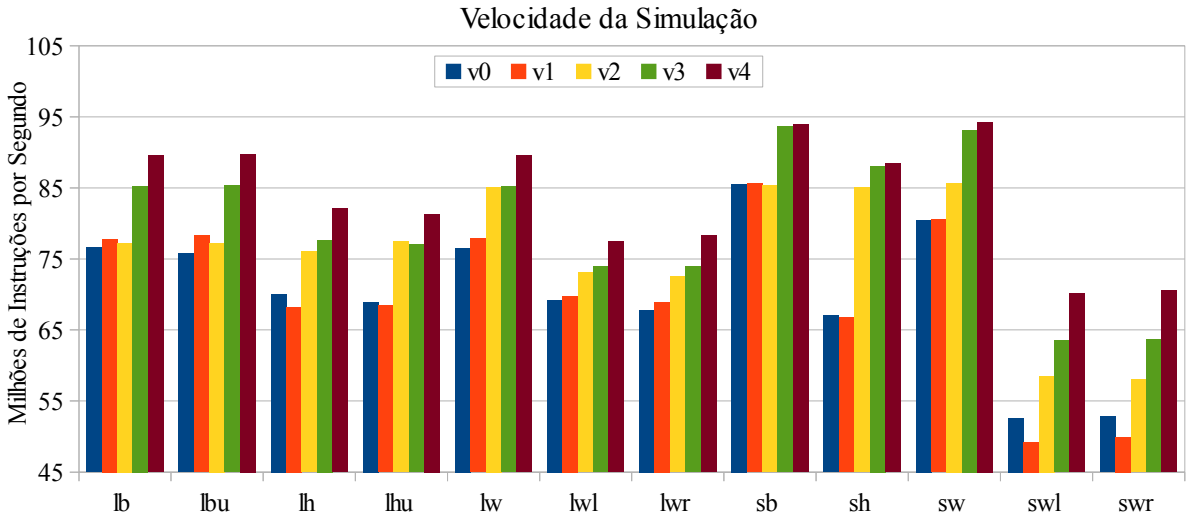


Figura 4.3: Desempenho das instruções de leitura e escrita.

no mesmo momento.

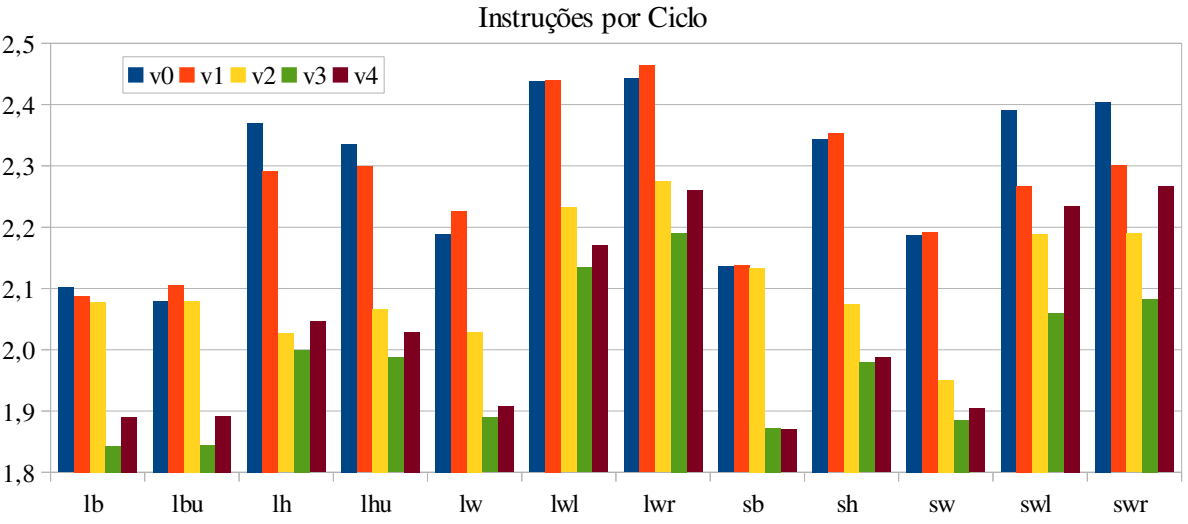


Figura 4.4: Instruções por ciclo das instruções de leitura e escrita.

Neste gráfico, observa-se que a redução na relação IH/IG impacta diretamente no IPC. Como menos instruções serão executadas, mais difícil tornar-se-a o escalonamento destas instruções. Por se tratar de instruções de operações de leitura e escrita em memória, estas instruções apresentam alta dependência de dados em memória, logo, quando diminui-se a quantidade de instruções a serem executadas, o número de instruções por ciclo também é reduzido. Um fato interessante neste gráfico é em relação à versão `v4`, que mesmo reduzindo a relação IH/IG, conseguiu um aumento no IPC em relação à v3. Isto é justificado pelo fato desta versão utilizar uma otimização que proporciona uma redução de acessos à memória e, consequentemente, uma diminuição na contenção por dependência de dados.

Com a análise do código foram identificados diversos problemas nos métodos de leitura e escrita em memória. O principal ponto consistia em um elevado número de instruções que eram executados por cada uma das instruções de leitura e escrita. Em média eram executadas 120 instruções por instrução de leitura/escrita e foi possível reduzir esta média para 88 instruções. Com as otimizações, obteve-se um ganho médio de desempenho de 19,20% na versão `v4`.

4.2 *Cache* de Instruções Decodificadas

Como visto na Seção 3.7, alguns *benchmarks* da suíte MiBench apresentaram altas taxas de faltas na *cache*. Nos simuladores ArchC, uma das estruturas que mais utiliza a *cache* física do processador é a *cache* de instruções decodificadas. Logo, este é o mecanismo que foi escolhido para análise.

A *cache* de instruções decodificadas tem o objetivo de possibilitar a reutilização das instruções já decodificadas, eliminando a necessidade de realizar novamente a decodificação da instrução e, com isso, proporcionar um ganho de desempenho. Porém, esta *cache* pertence à parte do código ArchC, fazendo com que ela seja genérica. Isto gera alguns problemas que serão detalhados a seguir.

O ArchC implementa uma classe para representar a instrução chamada `ac_instr`. Esta classe consiste em um vetor de inteiros de 32 *bits* não sinalizados. A quantidade de campos que este vetor irá possuir é definida pela quantidade de campos que os formatos das instruções do modelo possui acrescido de mais um campo que será utilizado como identificador da instrução. Já um item, ou linha da *cache* de instruções decodificadas, consiste em uma estrutura contendo uma *flag* (`valid`) que serve para indicar se a linha contém uma instrução já decodificada e um ponteiro para um objeto da classe `ac_instr`. Por fim, no código do simulador, fica a instância da *cache* que consiste de um vetor dessa estrutura.

Para ficar mais claro, será utilizado o modelo MIPS como exemplo. Na Figura 4.5 são apresentados os formatos das instruções do modelo MIPS. Pela figura, observa-se que este modelo possui 3 formatos distintos e estes formatos possuem 12 campos no total. Logo, uma instrução decodificada será representada por um vetor contendo 13 campos – que pode ser visto na Figura 4.6 – encapsulado em uma estrutura com uma *flag*, formando uma linha da *cache* de instruções decodificadas.

Pela Figura 4.6, nota-se 2 problemas neste tipo de abordagem. O primeiro problema é relacionado aos espaços vazios – campos 9 à 12 – que são gerados em decorrência de campos repetidos (`op`, `rs`, `rt`) nos formatos de instrução do modelo. Já o segundo problema é referente a alocar mais *bits* para se armazenar cada um destes campos. Como cada campo da instrução decodificada possui um tamanho fixo (32 *bits*), cada linha da *cache* de instruções decodificadas irá ocupar 416 *bits*. Se o tamanho da *cache* variar de acordo com a aplicação a ser executada, que é o caso dos simuladores gerados pelo ACSIM, isto irá ocasionar, de forma desnecessária, em um consumo elevado de memória.

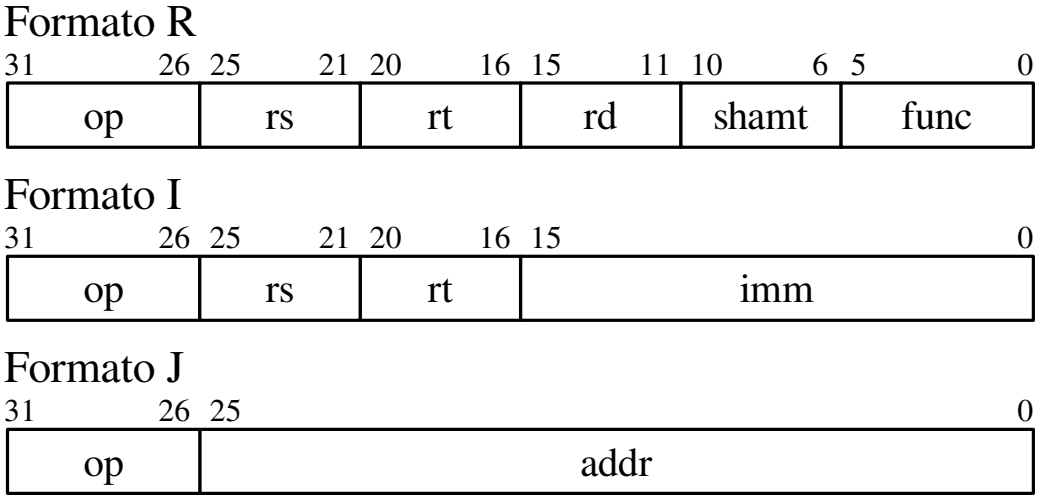


Figura 4.5: Formato das instruções do modelo MIPS.

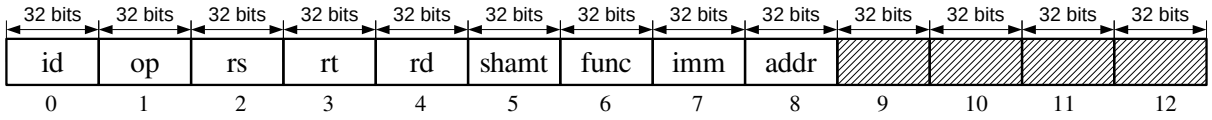


Figura 4.6: Estrutura de uma instrução decodificada do modelo MIPS.

4.2.1 A Nova *Cache* de Instruções Decodificadas

Como observado, a quantidade de campos das instruções decodificadas varia de acordo com os formatos de instruções de cada modelo. A definição da linha da *cache* fica no código do ArchC e a instância da *cache* fica no código do simulador. Porém, pelos problemas apresentados anteriormente, será necessário rever este conceito.

O primeiro problema pode ser solucionado de forma simples e sem alterações no conceito, reduzindo-se o valor que é utilizado na instância da *cache* de instruções decodificadas. Já o segundo, como trata-se de um problema na estrutura da linha da *cache*, exige uma abordagem mais complexa. Como a definição da estrutura de *cache* fica no código ArchC, esta estrutura tem que ser genérica, sendo esta a fonte do problema. Para solucionar este problema será necessário especializar a *cache* de instruções decodificadas, logo, esta estrutura deverá ser removida do código ArchC, passando a ser gerada no código do simulador.

Para especializar a *cache* de instruções decodificadas foi criada uma estrutura com os campos definidos a partir dos campos dos formatos da instrução do modelo. Isto permite alocar somente o espaço necessário para cada campo, sem ter os problemas de desperdício já citados anteriormente. Para compactar ainda mais a linha de *cache*, ao invés de tratar individualmente cada campo e armazená-los em posições diferentes, foi realizado uma sobreposição dos formatos das instruções utilizando-se do modificador `union`. Esta nova estrutura da *cache* de instruções decodificadas pode ser observada no Código 4.8, que exemplifica como ficaria o código da nova linha de *cache* para o modelo MIPS.

```
1  typedef struct {
2    bool valid;
3    unsigned id;
4    unsigned op:8;
5    union {
6      struct {
7        unsigned rs:8;
8        unsigned rt:8;
9        union {
10           struct {
11             unsigned rd:8;
12             unsigned shamt:8;
13             unsigned func:8;
14           };
15           int imm:16;
16       };
17     };
18     unsigned addr:32;
19   };
20 } lcache;
```

Código 4.8: Nova estrutura de linha da *cache* de instruções decodificadas.

Já a Figura 4.7 apresenta a nova disposição dos campos na nova linha de *cache* de instruções decodificadas. Nesta figura, observa-se que ainda há um desperdício nos formatos I e J (8 *bits*), porém é bem menor que na abordagem anterior. Na abordagem anterior, cada linha de *cache* ocupa 417 *bits*, esta nova abordagem ocupa 81 *bits*.

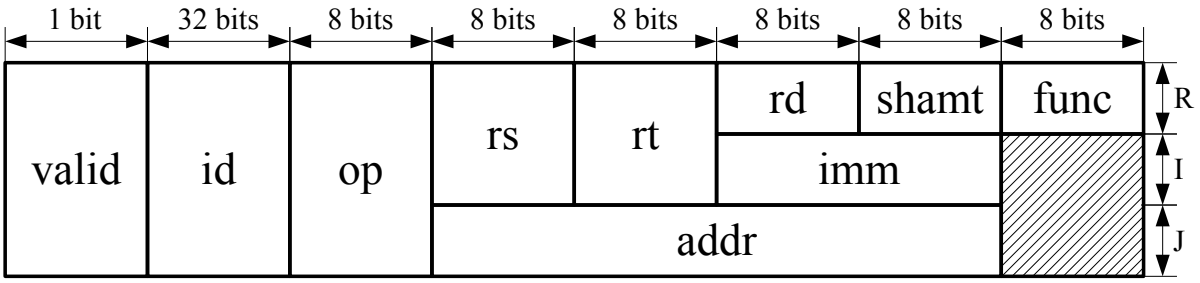


Figura 4.7: Nova linha da *cache* de instruções decodificadas.

Ao realizar alterações estruturais na linha de *cache* de instruções decodificadas, a forma que os dados são lidos e escritos nesta *cache* também sofrem impactos. Logo, os métodos que fazem uso da *cache* de instruções decodificadas também deverão ser alterados. Um dos pontos críticos envolvidos nesta alteração é o método de decodificação.

A decodificação de instruções dos simuladores ArchC é realizada por um método genérico pertencente ao código ArchC. Portanto, retorna-se à questão de realizar ou não a portabilidade de código do ArchC para código de simulador. Porém, neste caso a complexidade envolvida no método de decodificação de instruções é bem maior que o da

estrutura de *cache*. Logo, optou-se por manter a decodificação no código do ArchC e tratar o retorno desta função.

A função de decodificação de instruções retorna a estrutura apresentada na Figura 4.6. O tratamento deste retorno consiste em realizar um *parser* deste vetor com os campos da nova linha de *cache* de instruções decodificadas. O Código 4.9 apresenta o trecho do código fonte implementado para isso.

```
1  unsigned *aux = (ISA.decoder)->Decode(reinterpret_cast<unsigned char*>(
       buffer), quant);
2  if (aux) {
3    instr_vec->id = aux[IDENT];
4    instr_vec->op = aux[1];
5    //Type J
6    if (instr_vec->id == 46 || instr_vec->id == 47) {
7      instr_vec->addr = aux[8];
8    }
9    else {
10     instr_vec->rs  = aux[2];
11     instr_vec->rt  = aux[3];
12     //Type I
13     if (instr_vec->id < 21 || (instr_vec->id > 49 && instr_vec->id < 58)
           ) {
14       instr_vec->imm = aux[7];
15     }
16     //Type R
17     else {
18       instr_vec->rd    = aux[4];
19       instr_vec->shamt = aux[5];
20       instr_vec->func  = aux[6];
21     }
22   }
23 }
24 instr_vec->valid = true;
```

Código 4.9: *Parser* do retorno da função de decodificação de instruções.

Além da função de decodificação, as rotinas de interpretação – rotinas que implementam o comportamento das instruções a serem emuladas – também acessam a *cache* de instruções decodificadas, logo, estas também foram alteradas. Portanto, como se trata de código de simulador, sua alteração foi bastante simples. A alteração consistiu em substituir as chamadas à função `get` da antiga estrutura pelo campo requisitado da nova *cache*.

4.2.2 Metodologia

Para testar a nova *cache* de instruções decodificadas, foi realizada a implementação manualmente desta no modelo MIPS e foram geradas duas versões: `v1` e `v2`. No Capítulo 5 será realizado a implementação desta nova *cache* no gerador automático do ArchC. Como

versão `v0` foram utilizados os dados dos *benchmarks* selecionados das suítes MiBench e MediaBench executados na versão oficial no modelo MIPS – seção 3.7. Já a `v1` trata a implementação da nova *cache* de instruções decodificadas. Por fim, a `v2` trata-se da versão `v1` com a otimização nos métodos de leitura e escrita (`v4`), que foi descrita na Seção 4.1.

4.2.3 Resultados Experimentais

A Tabela 4.3 apresenta os resultados da execução dos *benchmarks* com a nova estrutura de *cache* de instruções decodificadas. Nesta tabela, cada linha apresenta: O *benchmark* (`B`) executado; a versão testada (`V`); a quantidade de instruções em milhões executada na máquina hospedeira (`IH`); quantos acessos e faltas em milhões ocorreram na *cache* L1 de dados; a quantidade em milhões do total e de erros de predição de desvios e o tempo de execução em segundos.

Comparando os dados desta tabela com os dados da versão oficial – Tabela 3.8 – observa-se que a quantidade de instruções executadas na máquina hospedeira e a quantidade de desvios teve um aumento na `v1` e uma redução na `v2`. Este aumento na `v1` é decorrente do acréscimo do *Parser* utilizado para tratamento da instrução após sua decodificação. A `v2`, por sua vez, sofre com o mesmo acréscimo de instruções do *Parser*, porém, as otimizações realizadas nos métodos de leitura e escrita em memória proporcionaram uma redução superior ao acréscimo feito pelo *Parser*. Estas otimizações tinham o objetivo de eliminar as instruções desnecessárias e dentre as instruções removidas, a maioria eram instruções de desvio, explicando, assim, a redução ocorrida na `v2`. Isto fica claro no gráfico da Figura 4.8.

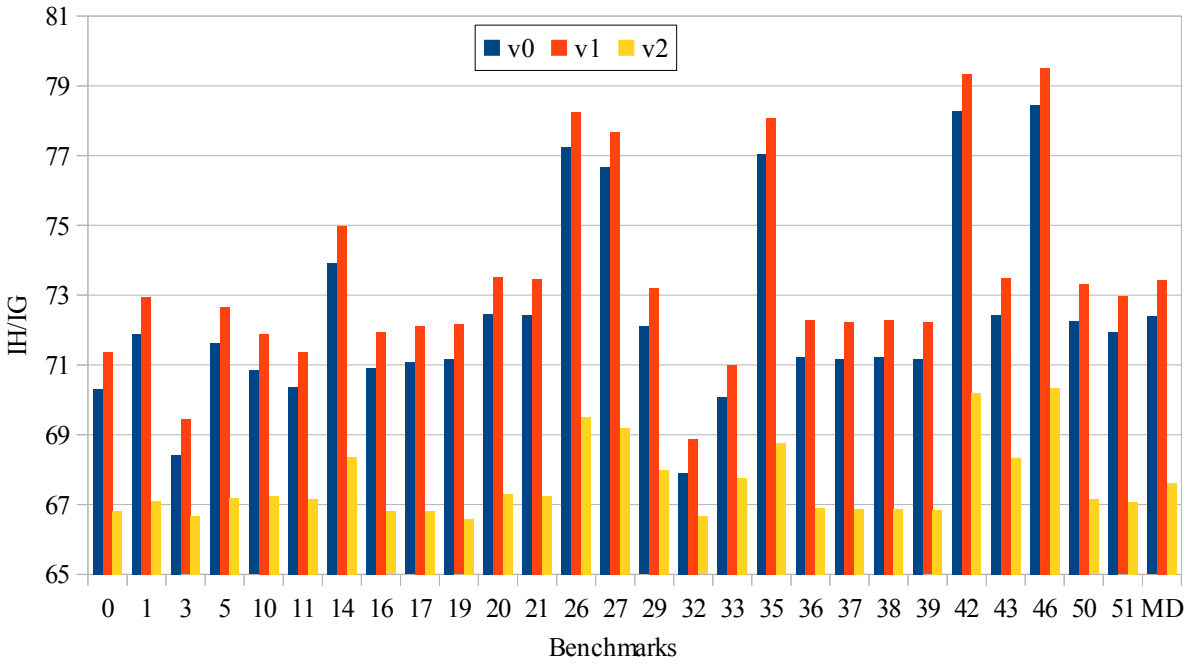


Figura 4.8: Relação IH/IG da nova *cache* de instruções decodificadas.

Já o gráfico da Figura 4.9 apresenta o desempenho da nova estrutura de *cache* de

Tabela 4.3: Dados de perfilamento da nova *cache* de instruções decodificadas.

B	V	IH	Acessos à *cache* L1		Desvios		Tempo
			Total	Faltas	Total	Erros	(s)
0	v1	97.150	57.966	608,10	22.266	747	17,14
	v2	90.942	56.600	627,13	21.305	729	16,65
1	v1	1.624.354	962.784	10.082,20	369.864	12.708	286,76
	v2	1.494.443	933.257	9.792,03	349.673	11.686	271,26
3	v1	47.515	28.794	0,31	11.021	207	7,28
	v2	45.619	28.257	0,35	10.683	136	6,66
5	v1	71.886	42.699	432,98	16.424	505	12,36
	v2	66.489	41.467	412,93	15.551	507	12,01
10	v1	12.753	7.665	98,45	2.934	94	2,25
	v2	11.929	7.437	99,78	2.787	90	2,15
11	v1	30.217	18.437	4,69	7.004	97	4,40
	v2	28.440	17.779	4,69	6.621	113	4,33
14	v1	8.179	4.835	7,59	1.859	32	1,22
	v2	7.455	4.650	7,96	1.731	34	1,18
16	v1	577.916	343.633	3.447,94	132.100	4.696	103,24
	v2	536.631	334.511	3.463,07	125.736	4.568	99,20
17	v1	6.800.596	4.040.345	41.207,78	1.553.291	55.777	1212,44
	v2	6.302.241	3.929.999	41.212,37	1.476.452	53.391	1165,41
19	v1	20.587	12.215	10,28	4.684	80	3,05
	v2	18.995	11.862	11,54	4.439	95	3,06
20	v1	21.260	12.570	139,82	4.833	169	3,82
	v2	19.467	12.156	141,10	4.551	153	3,59
21	v1	134.519	79.542	860,27	30.580	1.070	24,06
	v2	123.127	76.908	889,56	28.791	960	22,61
26	v1	27.469	16.297	465,69	6.150	236	5,12
	v2	24.403	15.468	465,45	5.630	235	4,83
27	v1	28.052	16.708	464,05	6.314	245	5,26
	v2	24.989	15.869	465,05	5.785	238	4,93
29	v1	9.933	5.980	1,63	2.258	40	1,50
	v2	9.227	5.804	1,88	2.144	36	1,42
32	v1	47.463	28.384	5,92	11.148	267	7,63
	v2	45.923	28.109	5,96	10.788	285	7,63
33	v1	38.250	22.858	6,94	8.912	211	6,24
	v2	36.506	22.386	6,57	8.539	191	5,97
35	v1	48.023	28.176	4,16	10.808	223	7,32
	v2	42.300	26.732	3,48	9.845	64	5,89
36	v1	54.976	32.655	367,56	12.552	431	9,71
	v2	50.883	31.743	348,96	11.921	411	9,29
37	v1	131.660	78.225	820,23	30.062	1.035	23,31
	v2	121.903	76.049	809,13	28.559	989	22,31
38	v1	1.101.914	654.490	6.796,97	251.530	8.632	195,06
	v2	1.019.386	636.064	6.215,14	238.820	8.186	185,92
39	v1	1.065.187	632.861	6.101,95	243.213	8.339	187,83
	v2	986.157	615.245	6.552,46	231.045	7.930	179,79
42	v1	139.901	81.304	854,27	31.881	831	22,68
	v2	123.797	77.112	766,59	28.448	898	21,84
43	v1	38.449	22.738	26,42	8.959	132	5,58
	v2	35.748	21.945	26,19	8.305	186	5,72
46	v1	17.550	10.194	114,31	3.999	106	2,87
	v2	15.530	9.668	100,99	3.567	115	2,75
50	v1	841.153	499.051	3.930,44	191.528	6.627	147,50
	v2	770.401	481.978	3.920,12	180.222	5.898	138,21
51	v1	275.178	163.267	1.094,54	62.622	2.263	48,90
	v2	252.948	158.299	1.143,16	59.172	2.207	47,00

instruções decodificadas. Neste gráfico, pode-se ver claramente que, apesar do aumento na relação IH/IG, na maioria dos *benchmarks* a nova *cache* obteve velocidades maiores – em média, a `v1` obteve 81,22 e a `v2` 85,14 – deixando evidente o ganho de desempenho proporcionado por esta nova estrutura. Porém, existem casos em que não houve nenhuma melhora na `v1` – *benchmarks* 3 e 35 – e casos em que a `v1` conseguiu desempenho similar ou superior à `v2` – *benchmarks* 19, 32 e 43.

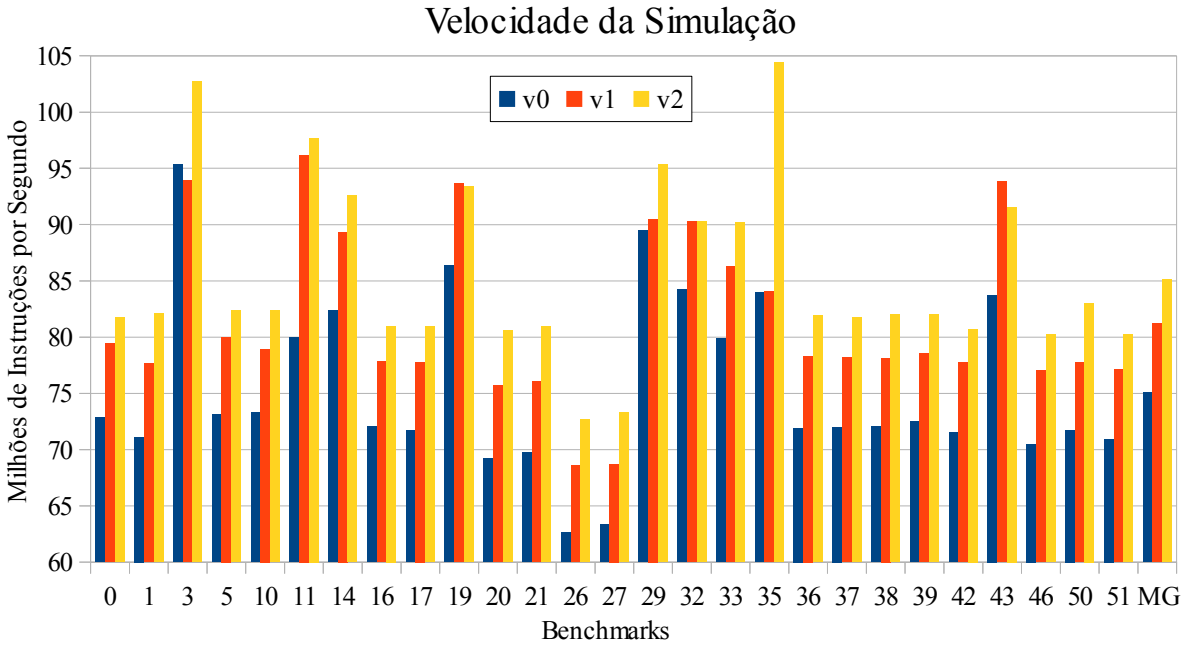


Figura 4.9: Desempenho da nova *cache* de instruções decodificadas.

Continuando a análise, a Figura 4.10 apresenta o percentual de faltas que ocorreram na *cache* L1 de dados do processador em relação à quantidade total de acessos executados. Neste gráfico pode-se ver que a nova *cache* de instruções decodificadas consegue reduzir as faltas que ocorriam na *cache* L1 – em média, houve uma redução de 52,94% na `v1` e 53,06% na `v2`. A taxa média de faltas da `v2` (0,86%) mostrada nesta figura é um pouco maior que a da `v1` (0,83%) por conta de uma redução de 3,39% na quantidade total de acessos na `v2` contra uma redução de 0,13% apresentada pela `v1`, ambas em relação ao total de acessos à *cache* na `v0`. O *benchmark* 3 quase não apresentava faltas na `v0`, logo, este não irá ter ganhos na `v1` como já confirmado na Figura 4.9.

Por fim, na Figura 4.11 é apresentado o gráfico com as taxas de erros de predição de desvio. Com este gráfico, fica claro o motivo pelo qual os *benchmarks* apresentarem os desempenhos vistos no gráfico da Figura 4.9. Os *benchmarks* 3 (bitcount_large) e 35 (CRC32_large) não tiveram ganhos de desempenho na `v1` por causa do aumento de erros de predição de desvio nesta versão. Já na versão `v2`, o *benchmark* 35 obteve uma grande redução de erros de predição de desvio na `v2`, e consequentemente um grande ganho de desempenho nesta versão. Por fim, em relação aos *benchmarks* que obtiveram ganhos similares ou superiores na `v1` em relação à `v2`, pode-se ver que foram os mesmos que sofreram uma redução na taxa de erros de predição na `v1`.

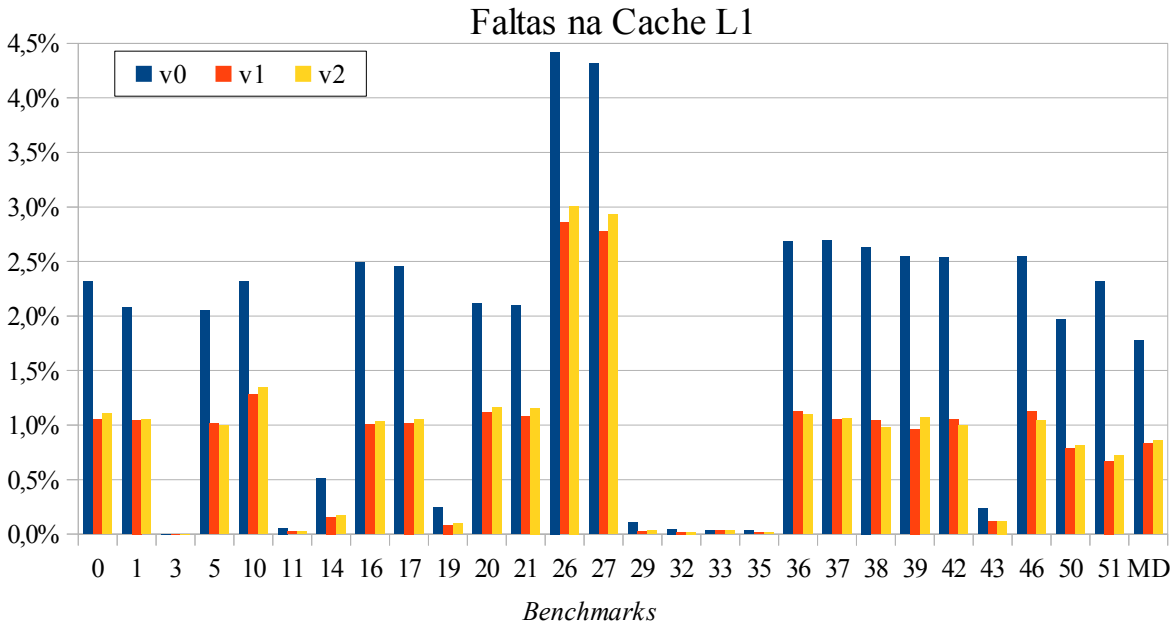


Figura 4.10: Taxa de faltas na *cache* L1 de dados da nova *cache* de instruções decodificadas.

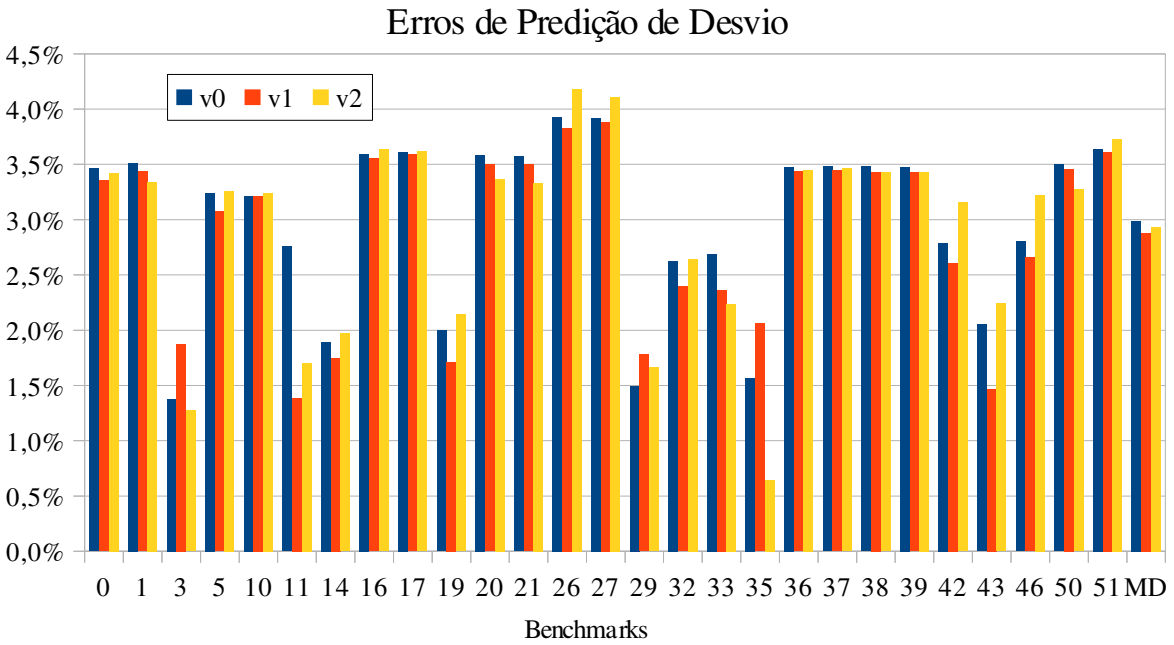


Figura 4.11: Taxa de erros de predição de desvio da nova *cache* de instruções decodificadas.

Concluindo, a análise de desempenho mostrou que a estrutura original da *cache* causava uma quantidade excessiva de faltas na *cache* em função do tamanho dos campos da linha da *cache* estarem superdimensionados. Uma análise mais aprofundada no código fonte detectou 2 grandes problemas na abordagem genérica desta estrutura e foi proposta

uma solução que proporcionou um ganho de desempenho, em média, de 8,14%. Combinado com as otimizações nos métodos de leitura e escrita em memória este ganho de desempenho foi para 13,36%.

4.3 Código do Simulador

Esta seção apresenta as análises realizadas no código dos simuladores gerados automaticamente pela ferramenta ACSIM. Pelas medições realizadas no Capítulo 3, um item ainda pendente refere-se ao alto número de instruções gastas pelo simulador na busca, decodificação e despacho da instrução a ser emulada. Portanto, esta análise do código do simulador tem como foco identificar os pontos que não são necessários e propor otimizações a serem feitas na emissão de código do simulador.

4.3.1 Funcionamento dos Simuladores ArchC

Como visto anteriormente no Capítulo 3, os simuladores ArchC executam 3 fases – configuração, emulação e término. Durante a fase de emulação são executadas 7 etapas para simular uma instrução. A primeira etapa consiste no tratamento de chamadas de sistema. Nesta etapa o PC é verificado e, caso o endereço esteja associado a uma chamada de sistema, o fluxo de execução é direcionado para as rotinas de tratamento de chamadas de sistemas. Se a verificação não identificar como sendo uma chamada de sistema, o fluxo é direcionado para a próxima etapa, que realiza a busca da instrução na *cache* de instruções decodificadas. Caso a instrução não esteja decodificada é realizado a próxima etapa que consiste na decodificação da instrução. Depois da decodificação da instrução é executada a etapa de comportamento comum, que consiste em código de execução comum para todas as instruções – normalmente trata-se do incremento do PC. A etapa seguinte realiza o despacho da rotina de emulação da instrução, que consiste em direcionar o fluxo de execução para as rotinas específicas para cada instrução. Antes de executar o comportamento específico da instrução, é executada a etapa de comportamento do formato, que reúne o código específico para tratar especificações requeridas pelos formatos de instrução. Por fim, é executada a etapa de comportamento da instrução, também conhecida como rotina de interpretação, e o ciclo volta para o início. Estas etapas são ilustradas na Figura 4.12.

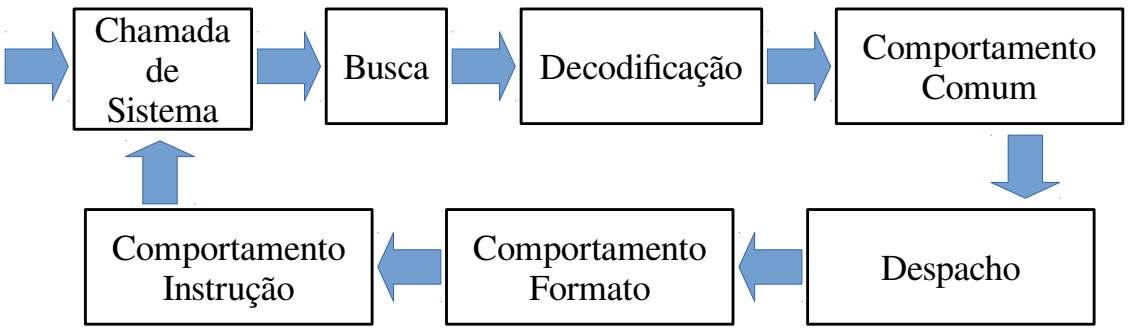


Figura 4.12: Etapas da simulação das instruções.

Como se trata de um laço que se repete até que a última instrução a ser emulada seja executada, qualquer código inserido em uma destas 7 etapas influencia bastante no número de instruções gastos na simulação. Logo, a análise deverá focar-se em verificar a necessidade e eficiência dos métodos presentes neste ciclo.

4.3.2 Computação de *Flags*

Ao analisar o código da fase de emulação nos simuladores gerados pelo ACSIM, um dos maiores problemas concentra-se na verificação de *flags*. Existem várias *flags* e estas são computadas em variados pontos do ciclo de emulação. O problema aqui é gerado pela não utilização da maior parte destas *flags* e isto é agravado pelo fato delas pertencerem ao código ArchC e, por causa disso, serem alocadas em memória e, consequentemente, não permitirem a sua remoção na compilação.

O código 4.10 apresenta a *flag* `ac_annul_sig` que é a *flag* mais utilizada. Esta *flag* é utilizada para tratar a anulação da instrução em tempo de execução.

```
1  if (!ac_annul_sig) ISA._behavior_instruction(instr_vec->get(1));
2  switch (ins_id) {
3  case 1: // Instruction lb
4    if (!ac_annul_sig) ISA._behavior_mips_Type_I(instr_vec->get(1),
       instr_vec->get(2), instr_vec->get(3), instr_vec->get(7));
5    if (!ac_annul_sig) ISA.behavior_lb(instr_vec->get(1), instr_vec->get
       (2), instr_vec->get(3), instr_vec->get(7));
6    break;
7  case 2: // Instruction lbu
8    if (!ac_annul_sig) ISA._behavior_mips_Type_I(instr_vec->get(1),
       instr_vec->get(2), instr_vec->get(3), instr_vec->get(7));
9    if (!ac_annul_sig) ISA.behavior_lbu(instr_vec->get(1), instr_vec->get
       (2), instr_vec->get(3), instr_vec->get(7));
10   break;
```

Código 4.10: Trecho de código com a computação da *flag* `ac_annul_sig` para o modelo MIPS.

Como pode-se ver no Código 4.10, a computação desta *flag* ocorre em todas as etapas de comportamento – comum (linha 1), formato (linhas 4 e 8) e instrução (linhas 5 e 9) – e isto adiciona pelo menos duas instruções para cada computação, uma instrução de comparação e uma de desvio. O grande problema não consiste somente no aumento de instruções, mas sim pelo fato de serem instruções de desvio, aumentando a pressão no preditor de desvios e, consequentemente, aumentando as chances de ocorrer um erro de predição.

O ArchC possui tanto modelos que fazem uso deste recurso – o ARM, por exemplo – como modelos que não utilizam ele, como o MIPS. Em modelos que não fazem uso deste recurso esta computação torna-se desnecessária e sua remoção soluciona o problema. Já em modelos que fazem uso deste recurso, é necessário verificar o funcionamento mais profundamente.

No modelo ARM, a *flag* `ac_annul_sig` pode ser alterada em duas etapas: na etapa de comportamento comum e na etapa de comportamento de formato. Nestas etapas são realizadas verificações e a partir de determinadas condições ocorre a alteração da *flag* `ac_annul_sig`. Após a alteração da *flag*, o ciclo de emulação continua com o despacho da instrução – linha 2 – sem realizar nenhuma etapa de comportamento seguinte. No fim o ciclo é reiniciado com a busca da próxima instrução.

Analisando este funcionamento, uma solução mais eficiente seria efetuar o desvio do fluxo de execução ao invés de ter que continuar a execução do ciclo de emulação e computar a *flag* `ac_annul_sig` para cada etapa. Porém, para implementar esta solução é necessário saber como desviar o fluxo e para onde.

Quando uma instrução é anulada, por definição, a próxima deve ser executada. Logo, é necessário descobrir como direcionar o fluxo para a próxima instrução. Quem controla qual instrução está sendo executada é o contador de programa (`PC`) e este é incrementado na etapa de comportamento comum para apontar para a próxima instrução. Como este incremento do `PC` ocorre antes da verificação de anulação, isto permite direcionar o fluxo de execução diretamente do ponto em que se identificou a anulação para o início do ciclo de emulação sem realizar nenhum tratamento.

A biblioteca da linguagem C++ possui uma função chamada `longjmp` que permite o desvio do fluxo de execução para pontos fora da função. Para isso, é necessário utilizar a função `setjmp` que guarda em uma variável o ponto em que se deseja ir e ao chamar a função `longjmp` passar esta variável juntamente com um valor de retorno. Ao retornar para o ponto setado com `setjmp`, pode-se analisar o valor de retorno e tomar uma ação diferente.

Com a abordagem utilizando `longjmp`, ao invés de realizar a computação da *flag* `ac_annul_sig` em cada etapa sujeita à anulação, é possível tratar somente no ponto em que se identifica a anulação e remover todas as demais computações. Esta otimização foi chamada de *New_ Annul* e irá proporcionar três melhorias. A primeira será a redução na quantidade de instruções executadas pelo simulador e a segunda trata-se de aliviar a carga do preditor de desvio, visto que serão removidas instruções de desvio condicionais. Por fim, como o tratamento só ocorre no momento em que é identificado a anulação, modelos que não utilizam este recurso não serão mais impactados com a computação deste mecanismo.

Outra *flag* utilizada no ciclo de emulação é a *flag* `ac_wait_sig`. Esta *flag*, na teoria, seria utilizada para tratar *stalls* – momentos em que o processador não tem o que executar e fica parado – na simulação do processador. Porém, ao realizar uma análise mais detalhada neste recurso, foi identificado que o mesmo não está funcionando. Portanto, por questão de desempenho, foi optado pela remoção desta *flag* e esta otimização foi chamada de *No_ Wait_ Sig*.

Por fim, a *flag* `ac_stop_flag` trata a finalização da etapa de emulação e funciona de forma similar à anulação de instruções. A cada ciclo de emulação, esta *flag* é computada e caso esteja setada a simulação é terminada. Assim como no tratamento da *flag* `ac_annul_sig`, é possível utilizar a função `longjmp` e remover esta computação do ciclo. Ao retornar para o início do ciclo, antes de iniciá-lo novamente, é possível verificar o valor retornado e finalizar a simulação. Esta otimização será referenciada como *New_ Stop* e sua

inclusão será controlada por parâmetro com o objetivo de testar a eficiência da abordagem usando a função `longjmp`.

4.3.3 Métodos de Configuração

Os simuladores gerados pelo ACSIM executam 2 métodos de configuração para iniciar a emulação. O primeiro realiza o carregamento da aplicação hóspede para memória e o segundo realiza a alocação da *cache* de instruções decodificadas, ambos métodos são executados somente uma vez. Porém, o primeiro método é executado antes de iniciar o laço de emulação – o ciclo descrito na Figura 4.12, já o segundo método é executado dentro deste laço. Ter métodos dentro do laço de emulação implica em aumentar a quantidade de instruções utilizadas para cada instrução emulada. O segundo método de configuração é executado somente na primeira iteração do laço, porém mesmo sem executar a alocação da *cache* de instruções, ainda é necessário computar a *flag* que controla sua execução, resultando em 2 instruções a mais em cada iteração. Se, em tempo de execução, for necessário executar novamente o segundo método de configuração, basta alterar a *flag* de controle deste método e o mesmo será reexecutado, porém, este não é o caso dos simuladores ArchC. Logo, para resolver este problema, basta mover este método para fora do laço de emulação. Esta otimização foi denominada como *New_ Config*.

4.3.4 Interpretador

Os simuladores ArchC utilizam como técnica de emulação a interpretação clássica [11, 14, 28, 47, 50]. Estudos realizados [18, 46] apontaram que esta técnica não é tão eficiente quanto a técnica *Direct Threading Code* [7, 15]. Além disso, analisando ambas as técnicas, pode-se obter uma redução no número de instruções executadas na emulação com a técnica *Direct Threading Code*. Nesta nova técnica é realizado um salto para a próxima rotina de interpretação ao final de cada rotina enquanto que na técnica clássica é necessário retornar para o código de despacho [50] para então saltar para a rotina de interpretação. Como há uma redução na quantidade de desvios, a pressão no preditor de desvio também será aliviada.

Para implementar a técnica *Direct Threading Code* é necessário algumas alterações no código do simulador, porém estas são alterações simples. Portanto, é uma boa opção analisar o desempenho desta nova técnica de interpretação e aumentar as opções que o ACSIM oferece. A partir daqui esta nova técnica será referenciada como otimização *Threading*.

4.3.5 Rotinas de Interpretação

Ao analisar o código das rotinas de interpretação – etapa de comportamento da instrução – foi identificado algumas diferenças entre rotinas de instruções semelhantes. Para exemplificar isto, o Código 4.11 apresenta o código em linguagem de montagem do comportamento da instrução `addu` e o Código 4.12 apresenta a instrução `add`. A diferença conceitual entre o comportamento dessas duas instruções consiste em um tratamento de erro existente na instrução `add` que não existe na instrução `addu`.

```
1   40fb01: cmp    BYTE PTR [rbx+0x102],0x0
2   40fb08: jne    40f3d0 <_ZN4mips8behaviorEv+0x120>
3   40fb0e: movsxd rsi,DWORD PTR [rax+0x8]
4   40fb12: movsxd rdi,DWORD PTR [rax+0xc]
5   40fb16: mov    rdx,QWORD PTR [rbx+0x410]
6   40fb1d: movsxd rcx,DWORD PTR [rax+0x10]
7   40fb21: mov    eax,DWORD PTR [rdx+rdi*4+0x8]
8   40fb25: add    eax,DWORD PTR [rdx+rsi*4+0x8]
9   40fb29: mov    DWORD PTR [rdx+rcx*4+0x8],eax
10  40fb2d: jmp    40f3d0
```

Código 4.11: Código em linguagem de montagem gerado para instrução addu do modelo MIPS.

```
1   40fad0: cmp    BYTE PTR [rbx+0x102],0x0
2   40fad7: jne    40f3d0 <_ZN4mips8behaviorEv+0x120>
3   40fadd: mov    r9d,DWORD PTR [rax+0x14]
4   40fae1: mov    r8d,DWORD PTR [rax+0x10]
5   40fae5: mov    rdi,r14
6   40fae8: mov    ecx,DWORD PTR [rax+0xc]
7   40faeb: mov    edx,DWORD PTR [rax+0x8]
8   40faee: mov    esi,DWORD PTR [rax+0x4]
9   40faf1: mov    eax,DWORD PTR [rax+0x18]
10  40faf4: mov    DWORD PTR [rsp],eax
11  40faf7: call   407270
12  40fafc: jmp    40f3d0
13
14  0000000000407270 <_ZN10mips_parms8mips_isa12behavior_addEjjjjjj>:
15  407270: push   rbx
16  407271: mov    rax,QWORD PTR [rdi+0xa8]
17  407278: movsxd rdx,edx
18  40727b: movsxd rcx,ecx
19  40727e: movsxd r8,r8d
20  407281: mov    esi,DWORD PTR [rax+rcx*4+0x8]
21  407285: add    esi,DWORD PTR [rax+rdx*4+0x8]
22  407289: mov    DWORD PTR [rax+r8*4+0x8],esi
23  40728e: mov    ebx,esi
24  407290: xor    ebx,DWORD PTR [rax+rdx*4+0x8]
25  407294: js     40729c
26  407296: xor    esi,DWORD PTR [rax+rcx*4+0x8]
27  40729a: js     4072a0
28  40729c: pop    rbx
29  40729d: ret
30
31  4072a0: mov    rdi,QWORD PTR [rip+0x21b7e9]
32  4072a7: mov    edx,0x41b3f0
33  4072ac: mov    esi,0x1
34  4072b1: xor    eax,eax
35  4072b3: call   403ac8 <__fprintf_chk@plt>
36  4072b8: mov    edi,0x1
37  4072bd: call   403b28 <exit@plt>
```

Código 4.12: Código em linguagem de montagem gerado para instrução add do modelo MIPS.

Em ambos os códigos, as linhas 1 e 2 tratam a computação da *flag* `ac_annul_sig`. Após esta computação, pode-se ver que, no primeiro código, o compilador conseguiu realizar o *inline* da rotina de interpretação da instrução `addu` – linhas 3 à 9 – no código do simulador, possibilitando diversas otimizações. Já no segundo código o mesmo não ocorreu, e, por causa desse fato, o código gerado possui muito mais instruções além das instruções gastas no tratamento de erro – linhas de 24 à 27 e de 31 à 37.

O compilador gcc possui regras para decidir se irá realizar ou não o *inline* a partir da quantidade de instruções que a função possui. Neste caso, o tratamento de erro fez com que a rotina da instrução `add` não cumprisse com os requisitos para realizar o *inline*, excluindo assim a possibilidade das otimizações realizadas para a instrução `addu`. No entanto, é possível forçar o compilador a realizar o *inline* em determinadas funções utilizando o atributo `always_inline`, e isto será analisado como otimização *Force_ Inline*.

4.3.6 Tratamento de Chamadas de Sistema

A emulação de chamadas de sistema nos simuladores ArchC consiste em um *Switch Case* que realiza a comparação do PC com alguns valores pré-definidos. Caso uma igualdade seja encontrada é executado uma rotina específica que implementa a chamada de sistema em si. Somente no caso de não haver nenhuma correspondência é que será buscada e executada a instrução apontada pelo PC. A lista de chamada de sistema pré-definida do ArchC ocupa os endereços de 0x3c a 0x88, logo, endereços acima destes endereços não necessitam entrar nesta comparação. Uma possível otimização é realizar um desvio para endereços acima de 0x100 para deixar de executar as comparações com a lista de chamadas de sistema. Esta será a otimização conhecida como *Syscall_ Jump*.

Com a implementação da otimização *Threading* e ao utilizar a *cache* de instruções decodificadas é possível utilizar o salto realizado pelo despachante e elaborar um novo mecanismo de tratamento de chamadas de sistema. Inicialmente deve ser criado rotinas de interpretação para realizar as devidas chamadas de sistema e ao inicializar a simulação é necessário preencher os espaços destinados às chamadas de sistema na *cache* de instruções decodificadas com os endereços dessas rotinas. Com isso é possível remover o *Switch Case* e a verificação descritos acima. Esta otimização será chamada de *New_ Syscall* e como já citado, só poderá ser aplicado em uma determinada configuração e resulta também na desativação da otimização *Syscall_ Jump*.

4.3.7 Execução em Plataforma

Para permitir a simulação em plataformas, os simuladores ArchC implementam uma sincronização entre as threads que emulam os diferentes periféricos da plataforma, e isto funciona da seguinte forma: o simulador executa um número configurável de instruções

e depois que atinge um limite realiza a chamada da função `wait` do SystemC e fica esperando novamente sua vez para continuar a execução. Isto é necessário para possibilitar o escalonamento da execução entre os diversos dispositivos da plataforma. O Código 4.13 apresenta a implementação deste recurso.

```
1 if (instr_in_batch < instr_batch_size) {
2   instr_in_batch++;
3 }
4 else {
5   instr_in_batch = 0;
6   wait(1, SC_NS);
7 }
```

Código 4.13: Trecho de código contendo a pausa para permitir a execução de outros dispositivos.

Em casos em que é simulado somente o processador, esta sincronização torna-se desnecessária. Logo, assim como ocorre no segundo método de configuração, a remoção deste código também irá ajudar na redução da alta relação IH/IG do simulador apresentada no Capítulo 3. O ACSIM já permite remover este trecho na geração dos simuladores utilizando o parâmetro `-no-wait` e o ganho de desempenho proporcionado por esta remoção será analisado como otimização *No_ Wait*.

4.3.8 Verificação do PC e da instrução

Os simuladores ArchC possuem duas verificações que são executadas durante o ciclo de emulação. A primeira consiste na comparação do PC com o limite de memória e a segunda trata de verificar se a instrução foi identificada.

A verificação do PC é realizada para garantir que a aplicação hóspede não execute instruções fora da área de memória reservada para a mesma. Neste caso, não foi encontrado nenhuma opção mais eficiente, porém, para teste de desempenho optou-se por modificar o ACSIM para permitir que esta verificação seja incluída ou não e isto será referenciado como otimização *No_ PC_ Ver*.

Já a identificação da instrução é executada em todo ciclo de emulação e isto é totalmente desnecessário ao utilizar a *cache* de instruções decodificadas. Como os simuladores ArchC não tratam de código auto modificável (*Self-Modifying Code*), uma vez que a instrução é decodificada, esta é armazenada na *cache* de instruções decodificadas e não será decodificada novamente. Logo, é necessário verificar a instrução somente quando ocorre a etapa de decodificação. *No_ Ident_ Inst* foi o nome dado à esta otimização.

4.3.9 Índice da *Cache* de Instruções Decodificadas

Os simuladores ArchC realizam a indexação da *cache* de instruções decodificadas utilizando como índice o PC e este é incrementado de forma a realizar um salto para a próxima instrução. Este incremento possui o tamanho da instrução decodificada, logo, isto faz com

que a *cache* de instruções decodificadas armazene as instruções de forma esparsa. Esta abordagem é a mais simples e é a abordagem que possui o melhor desempenho, visto que não é necessário realizar traduções de endereços, porém, isto faz com que seja alocado muito mais espaço do que o efetivamente utilizado. Em sistemas com pouca memória esta utilização demasiada e desnecessária é um grande problema.

Uma solução bastante simples e fácil de se implementar, consiste em realizar uma divisão do índice pelo tamanho da instrução ao indexar a *cache* de instruções decodificadas e o mesmo deve ser feito ao realizar a alocação do espaço em memória. Porém, em arquiteturas com instruções de tamanho variável, esta solução é inviável pois não é possível determinar o comportamento do incremento do PC. Logo, esta otimização foi denominada de *Index_Fix* e, por causa deste impacto, será implementado e utilizado no ACSIM através de parâmetro para ativar ou desativar tal solução.

4.3.10 Decodificação da Instrução

Os simuladores gerados pelo ACSIM realizam a decodificação das instruções por demanda. Durante a emulação, se uma instrução não está na *cache* de instruções decodificadas, a mesma então é decodificada e armazenada nesta *cache*. Como a *cache* de instruções decodificadas já possui espaço suficiente para armazenar toda a aplicação hóspede e não possui tratamento para *self-modifying code*, pode-se realizar a decodificação de toda a aplicação para depois iniciar o ciclo de emulação. Nesta nova abordagem, denominada de *Full_Decode* tem-se a vantagem de eliminar a verificação de validade da linha da *cache* de instruções decodificadas do ciclo de emulação. Por outro lado, será gasto tempo decodificando a aplicação hóspede por completo, que inclui tanto as instruções que não são utilizadas durante a emulação quanto a área de dados.

4.3.11 Informações da Simulação

Durante o ciclo de emulação, a variável `cur_instr_id` é utilizada para salvar o identificador da instrução a ser executada. Esta informação é armazenada em memória e é utilizada por alguns métodos para efeitos de informação. Por ser armazenada em memória, o compilador não tem como saber se está sendo utilizada e por isso não consegue removê-la. Realizando uma verificação no código, foi constatado que os métodos que fazem uso desta informação não estão sendo utilizados, logo, para efeitos de análise de desempenho a melhor solução é removê-la. Esta otimização será chamada de *No_Save_ID* e será necessário implementar no ACSIM uma opção que permita configurar a remoção ou não desta variável e dos métodos que a utilizam.

4.4 Conclusões

Neste capítulo foi realizada uma análise detalhada sobre as rotinas de leitura e escrita em memória e após identificar os problemas e corrigi-los, obteve-se uma redução de 32 instruções na relação IH/IG e um ganho de desempenho de 19,20% nos *microbenchmarks* destas instruções. Depois foi realizada uma análise profunda na *cache* de instruções

decodificadas que também apresentou problemas. Nesta análise, a otimização na *cache* sozinha obteve um ganho de desempenho de 8,14% e combinada com a otimização das rotinas de escrita e leitura em memória o ganho foi de 13,36% nos *benchmarks* das suítes MiBench e MediaBench. Por fim, foi realizada uma análise detalhada sobre o código gerado automaticamente para os simuladores ArchC e foram identificados diversos pontos que podem ser otimizados. O impacto destas otimizações no desempenho do simulador será analisado no próximo capítulo.

Capítulo 5

Geração Automática de Simuladores Otimizados

Como apresentado no Capítulo 4, foram levantados pontos no código do ArchC de possíveis otimizações. Neste capítulo são apresentados os resultados e análises realizadas nesses pontos. Para alguns pontos, foi possível elaborar soluções sem prejudicar as funcionalidades do simulador, já para outros foi necessária a criação de parâmetros para controlar a inclusão ou não da otimização.

As otimizações realizadas no código ArchC – uma sobre as instruções de leitura e escrita em memória que será referenciada como *LD_ST_Mem* e a outra sobre a *cache* de instruções decodificadas que será chamada de *New_Dec_Cache* – junto com as otimizações referentes à computação de *flags* (*New_Annul* e *No_Wait_Sig*), as otimizações nos métodos de configuração (*New_Config*) e as otimizações de verificação de instrução indefinida (*No_Ident_Inst*) foram aplicadas de forma fixa para simplificar os testes. Já as otimizações do novo método de parada (*New_Stop*) e as demais otimizações aplicadas nas rotinas de interpretação (*Force_Inline*), no tratamento de chamadas de sistema (*Syscall_Jump*), no interpretador (*Threading*), na execução em plataforma (*No_Wait*), na verificação do PC (*No_PC_Ver*), no índice da *cache* de instruções decodificadas (*Index_Fix*) e no decodificador de instruções (*Full_Decode*) foram configuradas a partir de parâmetros fornecidos à ferramenta ACSIM, que é encarregada de gerar o simulador a partir do modelo da arquitetura.

Por fim, as otimizações sobre as informações da simulação (*No_Save_ID*) e o novo mecanismo de chamadas de sistema (*New_Syscall*) foram identificados somente após a geração e testes do conjunto inicial de otimizações e, por este motivo, serão analisados posteriormente. A Tabela 5.1 apresenta um resumo das otimizações aplicadas e dos impactos/restrições identificados. Nesta tabela, cada linha apresenta a letra atribuída a cada otimização (`OPT`), seu nome, se foi aplicado de forma fixa ou parametrizada e o impacto/restrição relacionado à ela.

OPT	Otimização	Fixo	Impacto/Restrição
A	*LD_ST_Mem*	✓	
B	*New_Dec_Cache*	✓	
C	*New_Annul*	✓	
D	*No_Wait_Sig*	✓	
E	*New_Config*	✓	
F	*No_Ident_Inst*	✓	
G	*New_Stop*		
H	*Force_Inline*		
I	*Syscall_Jump*		
J	*Threading*		
K	*No_Wait*		Não funciona em Simuladores de Plataforma.
L	*No_PC_Ver*		Redução da segurança da simulação.
M	*Index_Fix*		Não aplicável em arquiteturas com formato variado.
N	*Full_Decode*		*Self-Modifying Code*.
O	*No_Save_ID*		Perda de algumas informações sobre a execução.
P	*New_Syscall*		Requer a otimização J com *cache* e desativa a I.

Tabela 5.1: Resumo das Otimizações e Impactos/Restrições.

5.1 Materiais e Métodos

Para realizar a análise, foi adotado o modelo MIPS e inicialmente será realizada a comparação da versão oficial 2.2 do ArchC com a versão A-F que será chamada de Otimização Base. Foi adotado este nome por ser a versão que contêm todas as otimizações fixas e que fazem parte de todas as demais versões geradas. As demais versões tratam-se de acréscimos realizados em cima da Otimização Base. O próximo passo será a realização de uma análise individual sobre as otimizações que são parametrizáveis – no caso as otimizações de G à N. Depois será realizada a análise combinando estas otimizações (G-N) e, por fim, será analisado as duas últimas otimizações O e P em conjunto com as melhores combinações identificadas na análise anterior.

Para a análise das versões geradas, para cada versão é realizada a execução dos 27 *benchmarks* selecionados das suítes MiBench e MediaBench. Cada um destes *benchmarks* é executado 3 vezes e é computada a média aritmética do tempo de execução. A computação do ganho de desempenho de uma determinada versão é realizada da seguinte forma: inicialmente seleciona-se as médias dos tempos de execução de cada um dos *benchmarks* da determinada versão. Com cada uma dessas médias é realizada a divisão pela média obtida por este mesmo *benchmark* na versão que foi utilizada como base – no caso da Otimização Base será a versão oficial 2.2 e no caso das otimizações individuais será a Otimização Base e assim por diante. Com isto, tem-se o ganho de desempenho em relação à versão base. Com os ganhos de desempenho dos *benchmarks* realiza-se agora uma média geométrica para obter-se o ganho de desempenho médio da versão em todos os *benchmarks* selecionados.

5.2 Análise da Otimização Base

A Figura 5.1 apresenta um gráfico resumindo os resultados obtidos pela Otimização Base em relação à versão Oficial. Neste gráfico, observa-se que a Otimização Base conseguiu reduzir, em média, a quantidade de instruções executadas na máquina hospedeira em 29%, proporcionando um ganho de desempenho médio de 20%. Além disso, nota-se que houve uma grande redução nos acessos/faltas na *cache* L1 de dados – redução de 27% nos acessos e 67% nas faltas – e na quantidade total de desvios executados (redução de 43%). Porém a quantidade total de erros de predição teve um aumento de 6% e o número de instruções por ciclos (IPC) sofreu uma redução de 15%.

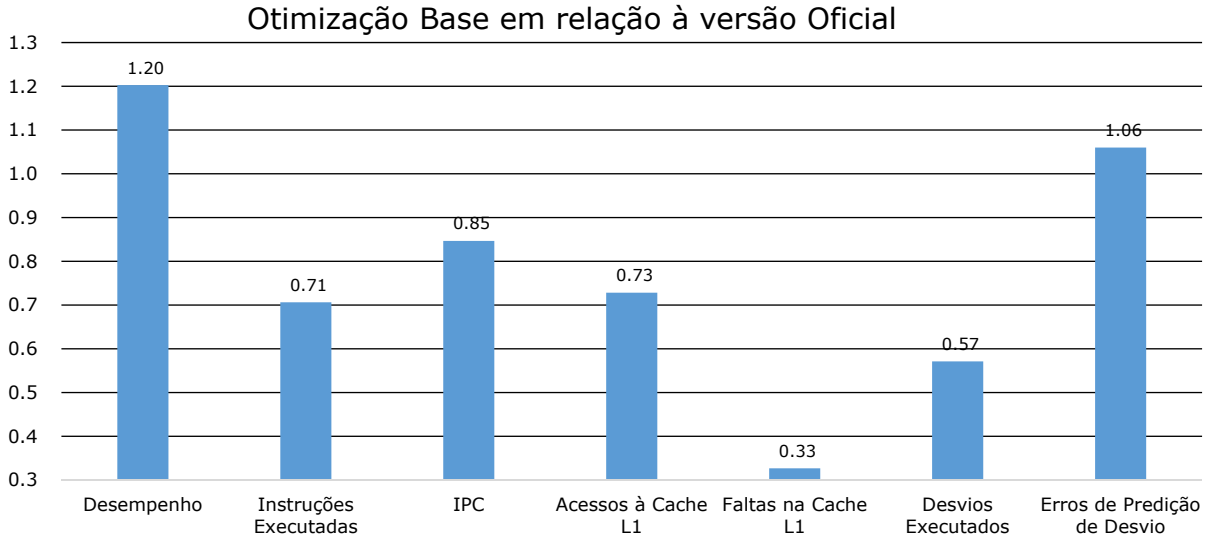


Figura 5.1: Comparação da Otimização Base com a versão Oficial.

Para analisar com mais detalhes a Otimização Base, as Figuras 5.2, 5.3, 5.4 e 5.5 apresentam, em ordem, o desempenho, a relação IH/IG, o percentual de faltas na *cache* L1 e, por fim, o percentual de erros de predição de cada um dos *benchmarks* na versão Oficial e na Otimização Base.

No geral, os *benchmarks* apresentam resultados semelhantes se comparado o mesmo *benchmark* nas duas versões. No entanto, pode-se descatar alguns pontos. Por exemplo, ao analisar o *benchmark* 11, observa-se que este não foi o *benchmark* que obteve a maior velocidade de simulação da Otimização Base, porém foi o *benchmark* com o maior ganho de desempenho (26,82%) entre versões. Isto se deve ao fato deste *benchmark* ter a maior taxa de redução na quantidade de erros de predição de desvios (25,76%) entre versões, mesmo apresentando a menor redução na relação IH/IG (25,7%) entre versões. No caso das faltas na *cache* L1 de dados, este já não possuia grande taxa de faltas e isto melhorou um pouco na Otimização Base.

Já o *benchmark* 03 foi o que apresentou a maior velocidade na Otimização Base – 110,39 milhões de instruções por segundo – porém foi o segundo pior ganho de desempenho (15,8%) entre versões. Este *benchmark* já não apresentava problemas na *cache* L1 na

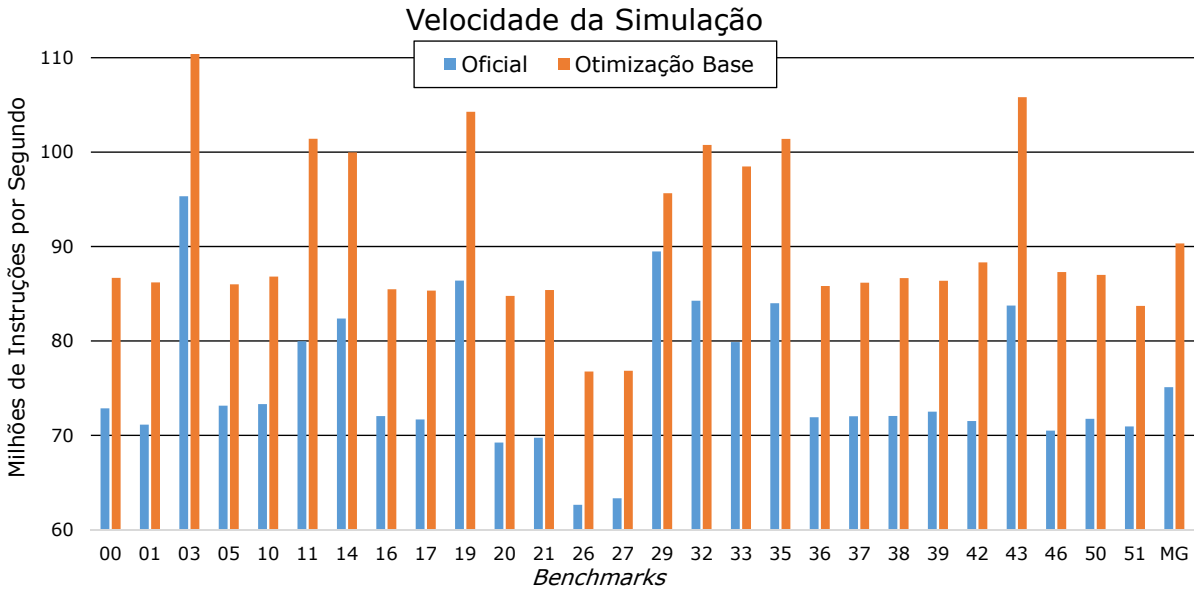


Figura 5.2: Desempenho dos *benchmarks* na versão Oficial e na Otimização Base.

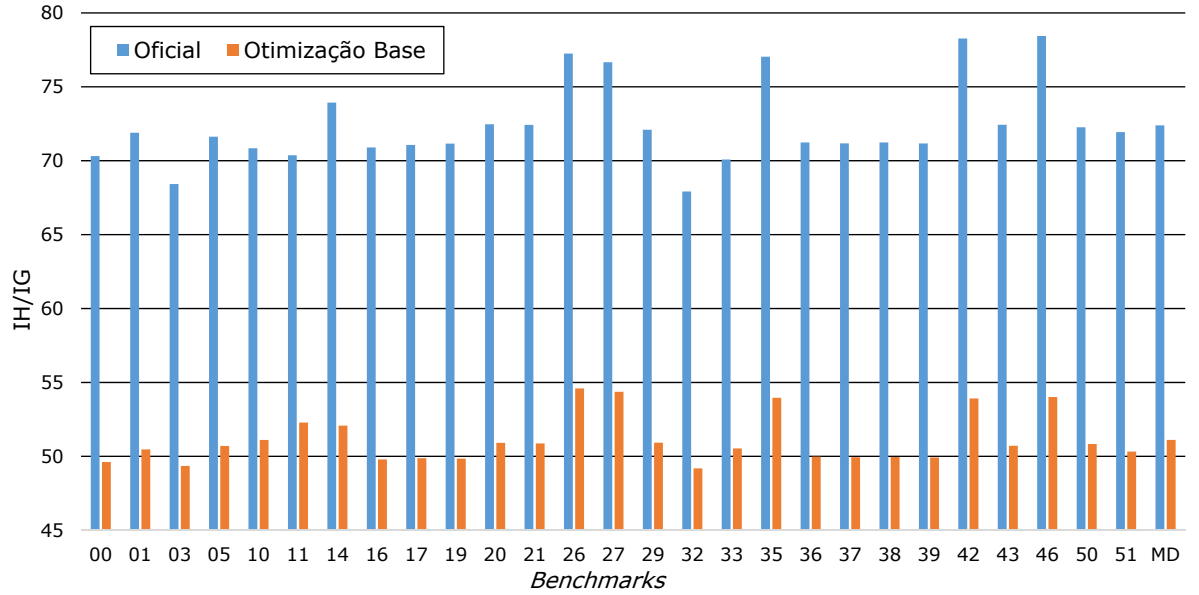


Figura 5.3: Relação IH/IG dos *benchmarks* na versão Oficial e na Otimização Base.

versão Oficial e isto continuou na Otimização Base, logo, as oportunidades de ganhos em otimizações de *cache* são mínimas. O ganho de desempenho se deve a uma redução de 27,87% nas instruções executadas, e poderia ter sido maior se não fosse pelo aumento de 34,21% na quantidade de erros de predição.

Resumindo, a Otimização Base conseguiu atingir uma velocidade, em média, de 90,33 Milhões de instruções por segundo, graças à uma relação IH/IG de 51,11 instruções em média com baixa taxa de faltas (0,78%) na *cache* L1. Porém a taxa de erros de predição aumentou de 2,99% para 5,5% e isto se deve a dois fatores. Primeiro, a redução

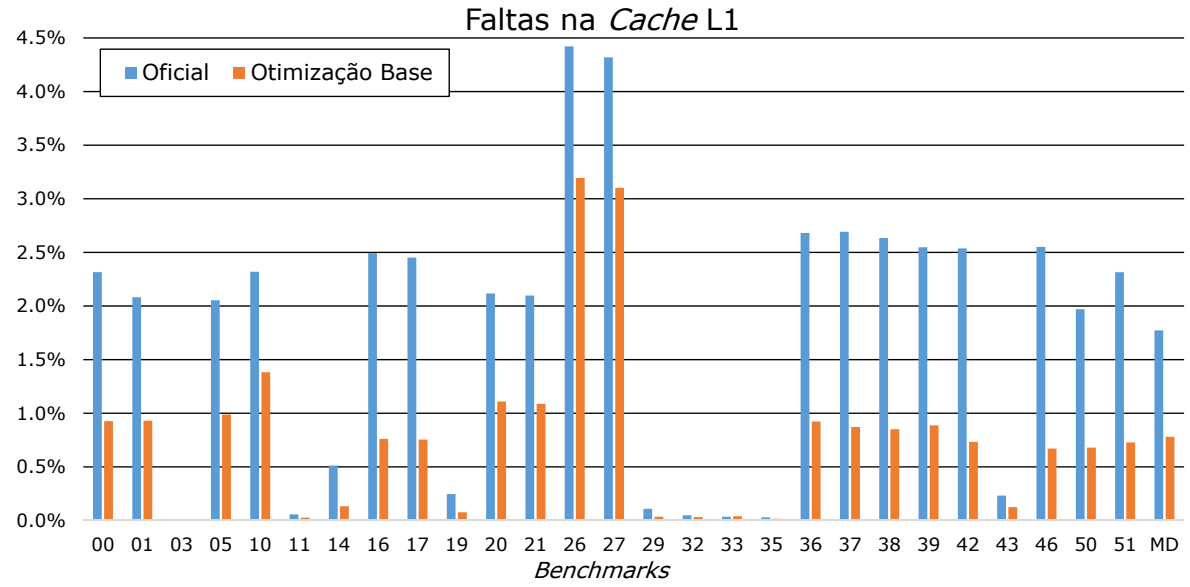


Figura 5.4: Taxa de faltas na *cache* L1 de dados dos *benchmarks* na versão Oficial e na Otimização Base.

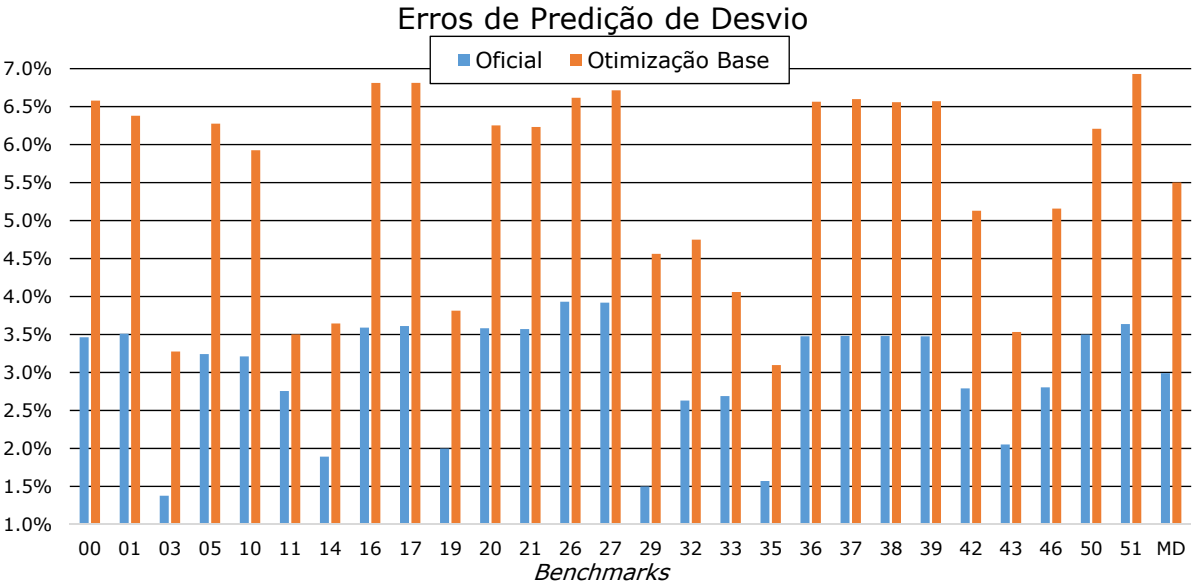


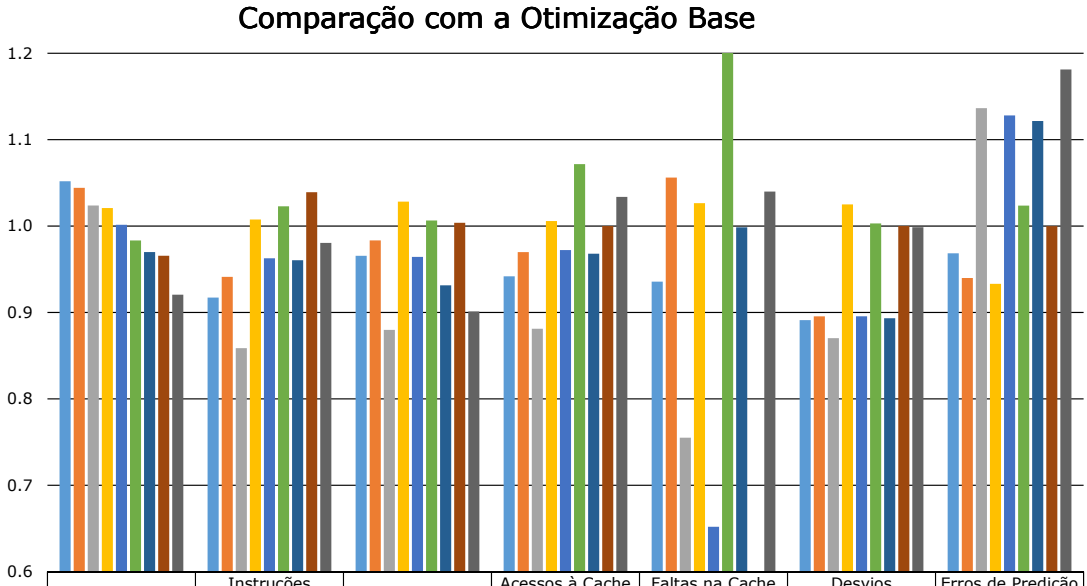
Figura 5.5: Taxa de erros de predição dos *benchmarks* na versão Oficial e na Otimização Base.

de 43% na quantidade total de desvios concentrou-se principalmente em desvios que o preditor normalmente acertava, ou seja, houve a remoção de desvios que contabilizavam como acertos, fazendo com que a taxa de erros aumentasse. Segundo, quanto menor a relação IH/IG significa que menos instruções são necessárias para se realizar a emulação de uma instrução, logo, a distância entre desvios também será reduzida e consequentemente a carga sobre o preditor de desvios aumenta. Desvios próximos fazem o preditor

sobreescrever constantemente o histórico de saltos realizados, dificultando assim o acerto na predição.

5.3 Análise Individual das Otimizações

A Figura 5.6 mostra o desempenho, a quantidade de instruções executadas, o número de instruções por ciclo, bem como os acessos e faltas na *cache* L1 e a quantidade de desvios e erros de predição de cada uma das otimizações em relação à Otimização Base. As otimizações estão ordenadas a partir de seu desempenho e foi adicionado um Oráculo que apresenta a melhor configuração para cada um dos *benchmarks*.



	Desempenho	Instruções Executadas	IPC	Acessos à Cache L1	Faltas na Cache L1	Desvios Executados	Erros de Predição de Desvios
Oráculo	1.052	0.917	0.966	0.942	0.936	0.891	0.968
New_Stop	1.044	0.941	0.983	0.970	1.056	0.896	0.940
No_Wait	1.024	0.859	0.880	0.881	0.755	0.870	1.137
Force_Inline	1.021	1.008	1.028	1.006	1.026	1.025	0.933
Full_Decode	1.002	0.963	0.964	0.972	0.652	0.896	1.128
Threading	0.983	1.023	1.006	1.072	2.217	1.003	1.024
No_PC_Ver	0.970	0.960	0.931	0.968	0.999	0.893	1.122
Index_Fix	0.966	1.039	1.004	1.000	0.147	1.000	1.000
Syscall_Jump	0.921	0.981	0.901	1.034	1.040	0.999	1.181

Figura 5.6: Comparação Individual das Otimizações em relação à Otimização Base.

Vale lembrar que estas otimizações são acréscimos realizados sobre a Otimização Base, logo, na teoria, deveriam obter ganhos de desempenho. Porém, através deste gráfico, observa-se que 4 otimizações – *New_ Stop*, *No_ Wait*, *Force_ Inline* e *Full_ Decode* – apresentam desempenho similar ou superior à Otimização Base, já as demais 4 otimizações tiveram perda de desempenho. A otimização *New_ Stop* apresentou o melhor desempenho (4,4% de ganho em relação à Otimização Base) e ficou bastante próxima do Oráculo (5,2%).

As otimizações *New_ Stop, No_ Wait, Full_ Decode, No_ Ver_ PC e Syscall_ Jump* tem como objetivo reduzir a quantidade de instruções executadas e, pode-se ver pelo gráfico

da Figura 5.6 que este objetivo é alcançado – reduções de 5,9%; 14,1%; 3,7%; 4% e 1,9% respectivamente. Como as remoções tratam-se de instruções de desvio, a quantidade de desvios executados, por consequência, também tiveram redução – 10,4%, 13%, 10,4%, 10,7% e 0,1% –, que já era esperado. Além das instruções de desvio, nas 4 primeiras otimizações dessas 5 houve a remoção de instruções de leitura em memória de *flags* globais. Isto explica a redução de 3%; 11,9%; 2,8% e 3,2%, respectivamente, nos acessos à *cache* L1. A única otimização que não há a remoção de instruções de leitura em memória é a *Syscall_Jump*, e neste caso, ocorreu um aumento de 3,4% nos acessos à *cache* L1. O inesperado nestas otimizações foi o aumento nas faltas na *cache* L1 na otimização *New_Stop* e a crescente ocorrência de erros de predição nas demais 4 otimizações.

No caso da otimização *Force_Inline* o foco é forçar o *inline* das rotinas de interpretação para permitir que o compilador possa ter mais possibilidades de otimizar o código do simulador. O esperado neste caso é a remoção das instruções de chamada às rotinas de interpretação e, consequentemente, a redução na quantidade de instruções executadas, porém, houve um aumento de 0,8%. O mais provável, neste caso, foi o *inline* ter sobrecarregado o escalonamento de registradores e forçado o compilador a gerar *spill code*[1].

Já a otimização *Threading* objetiva reduzir a quantidade de erros de predição no salto realizado pelo despachante, porém, observa-se que houve um aumento no número total de erros de predição. A causa disso pode ser atribuída à ocorrência de erros de predição em desvios fora do foco desta otimização. O aumento nos acessos à *cache* L1 (7,2%) já era esperado, uma vez que é realizada a replicação do código do despachante para cada rotina de interpretação, fazendo com que o tamanho do código do simulador cresça, aumentando assim a carga no escalonador de registradores e, consequentemente, fazendo o compilador gerar mais *spill code*, justificando também, o aumento de 2,3% na quantidade de instruções executadas.

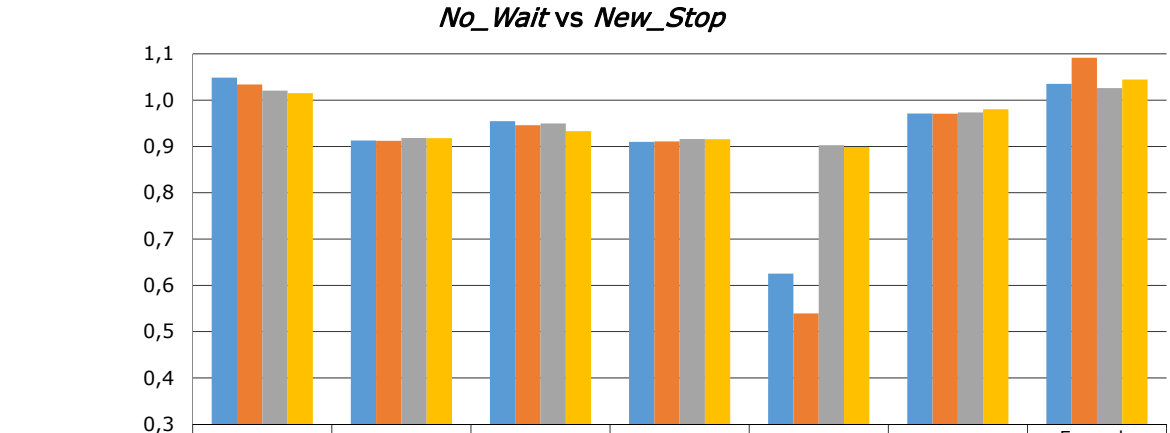
Por fim a otimização *Index_Fix* tem como objetivo reduzir o consumo de memória através da compactação dos dados da *cache* de instruções decodificadas. Neste caso, a compactação não altera a quantidade de acessos à memória, porém provoca uma melhoria na localidade dos dados e, consequentemente, uma redução nas faltas na *cache* – redução de 85,3% na L1. Houve um aumento de 3,9% no número de instruções executadas, por conta da inclusão das instruções para ajuste do índice da *cache* de instruções decodificadas, porém, como não são instruções de desvio, a quantidade total de desvios não sofreu alteração.

Dos 27 *benchmarks* selecionados, a otimização *New_Stop* proveu o melhor resultado em 15 *benchmarks*, ficou a até 1% de distância do melhor desempenho em outros 7 *benchmarks* e com distância acima de 1% nos 5 *benchmarks* restantes. Nestes 5 *benchmarks*, a otimização *No_Wait* apresentou o melhor desempenho em 4 *benchmarks* e a otimização *Full_Decode* foi a melhor no *benchmark* restante.

As Figuras 5.7 e 5.8 apresentam as informações de perfilamento das otimizações *No_Wait* e *Full_Decode* ao executar os *benchmarks* em que foram melhores que a otimi-

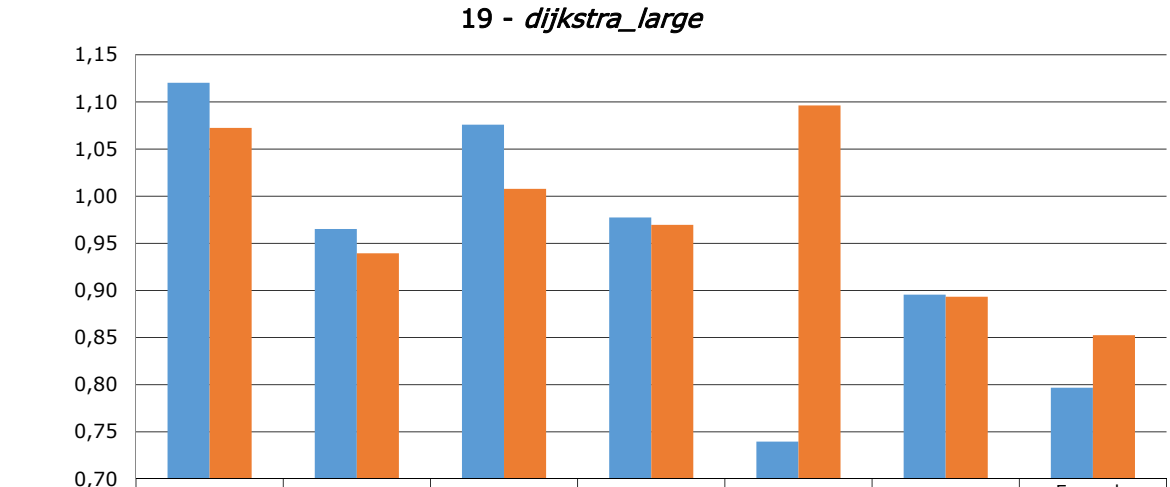
[1]Quando todos os registradores estão em uso e é necessário alocar mais uma variável em registrador, o compilador tem que gerar código para liberar temporariamente algum registrador. Este código gerado para armazenar/restaurar os valores de registradores em memória para liberá-los para novas alocações é conhecido como *spill code*.

zação *New_Stop*. Na primeira figura os dados da otimização *No_Wait* são comparados com os dados obtidos pela otimização *New_Stop* em cada um destes *benchmarks*. Já na segunda figura, são apresentados os dados em relação à otimização base.



	Desempenho	Instruções Executadas	IPC	Acessos à Cache L1	Faltas na Cache L1	Desvios Executados	Erros de Predição de Desvios
jpeg_large_enc	1,049	0,913	0,955	0,910	0,625	0,971	1,035
sha_large	1,034	0,912	0,946	0,911	0,539	0,971	1,092
rijndael_large_enc	1,021	0,918	0,950	0,916	0,903	0,973	1,026
rijndael_large_dec	1,015	0,918	0,933	0,916	0,899	0,980	1,045

Figura 5.7: Dados de perfilamento da otimização *No_Wait* em relação à *New_Stop*.



	Desempenho	Instruções Executadas	IPC	Acessos à Cache L1	Faltas na Cache L1	Desvios Executados	Erros de Predição de Desvios
Full_Decode	1,120	0,965	1,076	0,977	0,740	0,896	0,797
New_Stop	1,072	0,939	1,008	0,970	1,096	0,893	0,853

Figura 5.8: Dados de perfilamento do *benchmark 19-dijkstra_large* das otimizações *Full_Decode* e *New_Stop* em relação à otimização base.

Analisando estes gráficos, observa-se que a otimização *No_Wait* apresenta ganhos de desempenho de 4,9%, 3,4%, 2,1% e 1,5% em relação à otimização *New_Stop* nestes

4 *benchmarks*. Isto se deve a uma redução de quase 10% na quantidade de instruções executadas em conjunto com uma redução nos acessos/faltas na *cache* L1. O ganho de desempenho só não foi maior por causa do aumento nos erros de predição de desvios.

No caso da otimização *Full_ Decode*, a mesma apresentou um desempenho 4,8% melhor que a otimização *New_ Stop* no *benchmark 19-dijkstra_ large*, mesmo tendo um aumento de 2,6% no número de instruções executadas. Neste caso, o principal fator foi a ocorrência de 35,6% menos faltas na *cache* L1 junto com uma redução de 5,6% nos erros de predição de desvios.

5.4 Análise de Otimizações Combinadas

Foram geradas 256 versões combinando as otimizações parametrizáveis – otimizações de `G` à `N` da Tabela 5.1 – e os conjuntos de otimizações que apresentaram o melhor desempenho em pelo menos um *benchmark* são apresentadas na Tabela 5.2. Nesta tabela, a primeira linha apresenta o identificador do conjunto gerado, as próximas linhas apresentam quais otimizações fazem parte de cada conjunto, depois é apresentado a porcentagem da média geométrica do desempenho obtido por cada um deles em relação à Otimização Base. Por fim, é mostrado a quantidade de *benchmarks* em que cada conjunto apresentou o melhor desempenho. Vale ressaltar que os 5 primeiros conjuntos (`C0-C4`) são os conjuntos que apresentaram os melhores desempenhos médios, já para os demais (`C5-C8`) existem conjuntos com melhor média de desempenho que estes, porém, não foram apresentados pelo fato de não serem os primeiros em pelo menos 1 *benchmark*.

Tabela 5.2: Melhores conjuntos de otimizações combinadas.

Versão	C0	C1	C2	C3	C4	C5	C6	C7	C8
G-*New_ Stop*	✓	✓	✓	✓	✓	✓	✓	✓	✓
H-*Force_ Inline*		✓	✓	✓	✓		✓	✓	✓
I-*Syscall_ Jump*	✓	✓	✓	✓		✓			
J-*Threading*	✓	✓			✓	✓	✓	✓	✓
K-*No_ Wait*	✓	✓	✓	✓	✓	✓	✓	✓	✓
L-*No_ PC_ Ver*	✓	✓	✓	✓	✓	✓		✓	✓
M-*Index_ Fix*				✓		✓		✓	
N-*Full_ Decode*	✓	✓	✓	✓		✓			✓
Desempenho (%)	20,55	20,54	20,47	19,31	19,20	18,91	17,94	17,75	17,72
1^0 Lugar em:	3	8	7	1	3	1	1	2	1

Através desta tabela, pode-se ver que as otimizações *New_ Stop* e *No_ Wait* são bastante importantes, visto que aparecem em todos os conjuntos, mostrando que suas contribuições são importantes tanto individualmente quanto combinadas com outras otimizações. Por outro lado, a otimização em cima do interpretador (*Threading*) que não tinha apresentado bons resultados individualmente, conseguiu reduzir a diferença e passou a liderar ao ser combinado com outras otimizações (ganho de desempenho de 0,07% entre o melhor conjunto com e sem esta otimização – `C0` e `C2`). Este mesmo comportamento pode

ser notado nas demais otimizações (*No_ Pc_ Ver* e *Syscall_ Jump*), que individualmente não faziam sentido em ser aplicadas por apresentar perdas de desempenho, mas combinadas com outras, além de apresentarem ganhos de desempenho, também fazem parte dos 4 melhores conjuntos.

O conjunto `C1` não foi o que apresentou o melhor desempenho médio, porém, por ficar em primeiro em mais *benchmarks* (8), além do fato da diferença de desempenho em relação à `C0` ser bem pequeno (0,01%), este conjunto será considerado como sendo o melhor conjunto de otimizações. A diferença deste conjunto com o conjunto `C0` está na aplicação ou não da otimização *Force_ Inline*. Esta otimização depende muito do compilador e pelo fato de aparecer na maioria dos melhores conjuntos, sua aplicação foi considerada vantajosa.

Estes conjuntos de resultados apresentados na Tabela 5.2 levam em conta somente o melhor desempenho proporcionado pela combinação das otimizações e não considera as restrições que algumas otimizações possuem. Como mencionado na Tabela 5.1 as otimizações de `K` à `N` possuem algumas condições para serem aplicadas. Levando-se em conta que o simulador a ser gerado pelo ArchC possua alguma restrição, por exemplo, precise simular em um ambiente com plataforma, nenhum destes conjuntos poderiam ser usados, visto que todos utilizam-se da otimização `K` (*No_ Wait*). Neste caso, a Tabela 5.3 apresenta os melhores conjuntos excluindo-se alguma dessas otimizações restritivas.

Tabela 5.3: Melhores conjuntos excluindo alguma otimização restritiva.

Versão	CR0	CR1	CR2	CR3
G-*New_ Stop*	✓	✓	✓	✓
H-*Force_ Inline*		✓	✓	✓
I-*Syscall_ Jump*		✓	✓	
J-*Threading*	✓	✓	✓	
K-*No_ Wait*	✓			
L-*No_ PC_ Ver*		✓		
M-*Index_ Fix*				
N-*Full_ Decode*		✓	✓	
Desempenho (%)	18,89	6,45	4,66	4,50

É importante ressaltar que existem conjuntos na Tabela 5.2 que já apresentam algumas configurações sem alguma destas otimizações restritivas. É o caso dos conjuntos `C0` e `C4`, que excluem, respectivamente, as otimizações *Index_ Fix* e *Full_ Decode*. Neste caso, estes conjuntos não são apresentados novamente na Tabela 5.3.

Com isso evidencia-se a importância da otimização *No_ Wait* no ganho de desempenho. Com esta otimização ativa é possível atingir ganhos de desempenho de até 20,55% (`C0`), porém, os ganhos de desempenho ficam em 6,45% (`CR1`) com a mesma desativada. Já no caso em que ocorram todas as quatro restrições, o ganho de desempenho fica limitado em 4,5%.

5.5 Outras Otimizações

Como já explicado anteriormente no início deste capítulo, duas otimizações foram identificadas durante a execução dos testes. Ao invés de realizar a execução novamente de todas as combinações possíveis, optou-se por selecionar as melhores combinações e ativar as duas novas otimizações. Diferentemente da otimização *No_Save_ID*, a otimização *New_Syscall* não foi implementada com a utilização de um novo parâmetro. Pelo fato desta última apresentar restrições de uso, enquanto que a primeira apresenta possíveis impactos em seu uso. Por causa disso, todo conjuto que satisfaça seu pré-requisito terá esta otimização ativa – é o caso dos conjuntos `C0` e `C1` que possuem a otimização *Threading* ativa. Os resultados destes novos testes são apresentados na Tabela 5.4.

Tabela 5.4: Outros conjuntos de otimização.

Versão	C3+OPT	C0+OPT	C2+OPT	C1+OPT
O-*No_Save_ID*	✓	✓	✓	✓
P-*New_Syscall*		✓		✓
Desempenho (%)	23,18	22,02	21,49	14,34
1^0 Lugar em:	14	7	2	1

Nesta tabela, observa-se que o conjunto `C3` acrescido da otimização *No_Save_ID* é o que apresenta o melhor desempenho e fica em primeiro em mais *benchmarks*, logo, esta é a melhor combinação. Diferentemente do apresentado na Tabela 5.2, o conjunto que foi escolhido como melhor combinação (`C1`) foi o que apresentou o pior desempenho ao ser combinado com estas novas otimizações. Para explicar o que aconteceu com a versão `C1+OPT` será necessário uma análise mais detalhada em seu dados de perfilamento. Para simplificar esta análise, foi realizado a verificação apenas nos *benchmarks* em que houve grande diferença em relação à versão `C1` e os resultados são apresentados na Tabela 5.5.

Tabela 5.5: Dados de Perfilamento das versões `C1` e `C1+OPT`.

B	V	IH	MIPS	Desvios		*Cache* L1		*Cache* L2	
				Total	Erros	Total	Faltas	Total	Faltas
19	C1	10.123	134,31	1.532	76	7.043	10	592.660	127.296
	C1+OPT	8.475	123,44	1.215	91	6.313	11	596.393	126.111
26	C1	13.600	91,37	2.070	233	9.494	364	14.521.184	197.394
	C1+OPT	11.782	81,96	1.718	238	8.726	383	8.421.233	220.541
27	C1	13.904	92,41	2.117	238	9.748	359	14.364.624	213.825
	C1+OPT	12.025	82,88	1.751	240	8.954	385	3.359.005	223.143

A Tabela 5.5 apresenta o *benchmark* (`B`); a versão (`V`) testada; a quantidade (em milhões) de instruções executadas pela máquina hospedeira (IH); o desempenho (MIPS); a quantidade total e de erros de predição de desvio em milhões; a quantidade de acessos total e de faltas à *cache* L1 em milhões e, por fim, a quantidade absoluta de acessos e

de faltas na *cache* L2. Nesta tabela, nota-se que apesar da versão `C1+OPT` ter executado menos instruções e realizado menos desvios e acessos às *caches* L1 e L2 em relação à versão `C1`, a mesma apresentou mais erros de predição de desvios e mais faltas em ambas as *caches*, justificando a queda de desempenho.

A abordagem de selecionar as melhores combinações e aplicar as novas otimizações nem sempre apresenta os melhores resultados. Como a abordagem não cobre todas as combinações de otimizações possíveis, há a possibilidade de existirem combinações com melhor desempenho. Um caso selecionado ao acaso (`C5+OPT`) conseguiu apresentar um desempenho de 23,70% em relação à otimização base, ultrapassando assim o desempenho obtido por `C3+OPT` (23,18%). Isto torna importante a verificação das demais combinações de otimizações, porém, isto será alvo de trabalho futuro.

Finalizando, tem-se que a otimização *New_ Stop* é a melhor otimização, pois apresentou o melhor desempenho individual e está presente em todas as melhores combinações, tanto as sem restrições quanto nas com restrições. E, portanto, fica como recomendação estar sempre ativada. Já em simulações de processadores sem plataforma, recomenda-se que a otimização *No_ Wait* seja sempre usada, visto que, apresentou o segundo melhor desempenho individual e combinado com outras otimizações teve uma grande diferença entre o melhor desempenho (20,55%) utilizando esta otimização em frente a sem utilizá-la (6,45%).

Capítulo 6

Conclusão

Esta dissertação apresentou uma análise de desempenho realizada sobre os simuladores gerados automaticamente a partir de modelos descritos em ArchC. A partir do perfilamento e análise do código dos simuladores foram identificados diversos pontos de ineficiência nestes simuladores e foram desenvolvidas otimizações para cada um destes pontos. A ferramenta ACSIM, responsável pela geração dos códigos do simulador interpretado dos modelos, foi modificada e conseguimos um desempenho médio 48,18% melhor no modelo MIPS em relação ao mesmo modelo gerado na versão oficial 2.2.

Inicialmente, o foco deste trabalho era sobre a técnica de interpretação clássica implementada nos simuladores ArchC, pois, segundo a literatura [18, 46] esta técnica não é tão eficiente quanto a técnica de interpretação *Threading*. No entanto, ao implementar esta outra técnica não conseguimos obter os mesmos resultados – esperáva-se obter ganho de desempenho com a técnica de *Threading*, porém obtivemos o oposto. Isto fez com que a análise sobre os simuladores gerados automaticamente virasse prioridade e os resultados estão apresentados nesta dissertação.

Com a otimização sobre os simuladores gerados automaticamente, pode-se ver que, na atual situação, as técnicas de interpretação analisadas (clássica e *Threading*) não proporcionam mais os ganhos já documentados em outras épocas. A técnica de *Threading* tenta otimizar o simulador a partir da tentativa de facilitar os desvios realizados no dispachante, que na época eram o grande problema para o preditor de desvios. Com a evolução do preditor de desvios, a diferença que era grande se tornou irrisória, fazendo com que a técnica de *Threading* perca a vantagem que possuia na sua concepção.

Concluimos que atualmente a técnica de interpretação no contexto dos simuladores ArchC não exerce grande influência no desempenho da simulação. Visto que ambas as técnicas implementadas não proporcionam entre si grandes diferenças nos tempos de execução. Técnicas mais avançadas de interpretação [8, 11, 19, 20] podem ser implementadas, porém a maior parte destas técnicas se aproximam da tradução dinânima de binários, tornando-as menos portáveis e mais complexas, indo contra o fundamento de simplicidade objetivado nos simuladores interpretados do ACSIM.

Realizamos a análise de desempenho através do perfilamento do código do simulador e desenvolvemos uma metodologia para auxiliar na identificação de possíveis pontos de ineficiência. Nesta metodologia, o custo de configuração e término serve de filtro para selecionar os *benchmarks* a serem utilizados nos testes e a seleção de instruções auxilia

na identificação das instruções mais utilizadas, e, consequentemente, mais importantes. Por fim, os *MicroBenchmarks* propostos nos permitem identificar grandes variações de desempenho entre diferentes instruções emuladas pelo simulador.

Como contribuições pode-se destacar o estudo e elaboração de métricas realizadas para se avaliar o desempenho do simulador gerado e do código ArchC, que exigiu uma análise comportamental destes, o desenvolvimento de uma metodologia para interpretar os dados de perfilamento e identificar os gargalos de desempenho e, por fim, o desenvolvimento de otimizações que podem ser controladas a partir de parâmetros na geração do código dos simuladores do ArchC.

A seguir apresentamos uma lista com os trabalhos que achamos relevantes para um futuro próximo.

1. Separar a busca da instrução da decodificação: Atualmente o processo de busca de instrução está encapsulado junto com o processo de decodificação da instrução. Isto inflexibiliza e dificulta a análise destas etapas e de possíveis otimizações sobre elas. A proposta aqui seria isolar estas duas etapas, pois da forma que está sendo realizada é bastante complicado diferenciar e identificar como é realizada a busca.

2. Especializar a decodificação de instrução para o Modelo: A decodificação das instruções faz parte do Código ArchC, por causa disso, esta deve ser genérica o suficiente para atender todos os modelos de simulador. Isto faz com que seja complicado realizar otimizações e torna necessário a utilização do *parser* para armazenar a instrução decodificada na nova *cache* de instruções decodificadas que é específica para cada modelo.

3. Tratar *Self-Modifying Code*: Este é um recurso que atualmente os simuladores ArchC não possuem e é fundamental para realizar simulações que possuem código automodificável – sistema operacional, por exemplo. Para isso, será necessário alterar a *cache* de instruções decodificadas, criando um mecanismo para identificar quando houver alterações em áreas contendo instruções.

4. Realizar as demais combinações de otimizações: Este é um item que ficou pendente e, como já citado no texto, é importante para confirmar se existe alguma combinação com ganho superior aos resultados obtidos e documentados nesta dissertação.

5. Analisar o ganho de desempenho nos outros modelos: Neste trabalho, as análises foram realizadas sobre o modelo MIPS, porém, as otimizações criadas e implementadas no gerador automático são genéricas o suficiente para serem aplicadas em qualquer modelo – com excessão de algumas que possuem restrições. Logo, é interessante analisar o comportamento destas otimizações nos demais modelos e é possível que o melhor conjunto final de otimizações para cada modelo seja diferente.

6. Validar a metodologia nos demais modelos/geradores: Como se trata de uma metodologia genérica, ela pode ser aplicada nos demais modelos (ARM, PowerPC e SPARC) e também nos outros geradores (ACCSIM e ACDCSIM).

Referências Bibliográficas

[1] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33, 2005.

[2] A. Baldassin, P. Centoducatte, and S. Rigo. Extending the archc language for automatic generation of assemblers. In *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, pages 60–67, Oct 2005.

[3] A. Baldassin, P. Centoducatte, S. Rigo, D. Casarotto, L.C.V. Santos, M. Schultz, and O. Furtado. Automatic retargeting of binary utilities for embedded code generation. In *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, pages 253–258, March 2007.

[4] A. Baldassin, P. Centoducatte, S. Rigo, D. Casarotto, L.C.V. Santos, M. Schultz, and O. Furtado. An open-source binary utility generator. *ACM Trans. Des. Autom. Electron. Syst.*, 13(2):27:1–27:17, April 2008.

[5] M. Bartholomeu, R. Azevedo, S. Rigo, and G. Araujo. Optimizations for compiled simulation using instruction type information. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 74–81, Oct 2004.

[6] Robert C. Bedichek. Talisman: fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '95/PERFORMANCE '95, pages 14–24, New York, NY, USA, 1995. ACM.

[7] James R. Bell. Threaded code. *Commun. ACM*, 16:370–372, June 1973.

[8] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society.

[9] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(12):1625–1639, Dec 2004.

[10] M. Bumble, L. Coraor, and L. Elefteriadou. Exploring corsim runtime characteristics: profiling a traffic simulator. In *Simulation Symposium, 2000. (SS 2000) Proceedings. 33rd Annual*, pages 139–146, 2000.

[11] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, 29(6), October 2007.

[12] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA, 1996. IEEE Computer Society.

[13] J. W. Davidson and J. V. Gresh. Cint: a risc interpreter for the c programming language. In *Papers of the Symposium on Interpreters and interpretive techniques*, SIGPLAN '87, pages 189–198, New York, NY, USA, 1987. ACM.

[14] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, IVME '03, pages 41–49, New York, NY, USA, 2003. ACM.

[15] Robert B. K. Dewar. Indirect threaded code. *Commun. ACM*, 18:330–331, June 1975.

[16] K. Diefendorff and E. Silha. The powerpc user instruction set architecture. *Micro, IEEE*, 14(5):30–, Oct 1994.

[17] M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 315–327, New York, NY, USA, 1995. ACM.

[18] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 403–412, London, UK, 2001. Springer-Verlag.

[19] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, IVME '04, pages 7–14, New York, NY, USA, 2004. ACM.

[20] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Softw. Pract. Exper.*, 32(3):265–294, March 2002.

[21] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *European Design and Test Conference, 1995. ED TC 1995, Proceedings.*, pages 503–507, Mar 1995.

[22] J. Fischer, V. Natoli, and D. Richie. Optimization of lammps. In *HPCMP Users Group Conference, 2006*, pages 374–377, June 2006.

[23] M. Garcia, R. Azevedo, and S. Rigo. Optimizing simulation in multiprocessor platforms using dynamic-compiled simulation. In *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*, pages 80–87, Oct 2012.

[24] M.S. Garcia, R. Azevedo, and S. Rigo. Optimizing a retargetable compiled simulator to achieve near-native performance. In *Computing Systems (WSCAD-SCC), 2010 11th Symposium on*, pages 33–39, Oct 2010.

[25] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[26] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 485–490, March 1999.

[27] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. Mips: A microprocessor architecture. *SIGMICRO Newsl.*, 13(4):17–22, October 1982.

[28] Paul Klint. Interpretation techniques. *Software — Practice & Experience*, 11(9), 1981.

[29] P. M. Kogge. An architectural trail to threaded-code systems. *Computer*, 15:22–32, March 1982.

[30] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[31] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators. In *Design Automation Conference, 1999. Proceedings of the ASP-DAC '99. Asia and South Pacific*, pages 339–342 vol.1, Jan 1999.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI'05*, New York, NY, USA, 2005.

[33] Mediabench. http://euler.slu.edu/ fritts/mediabench/mb1/, 11 2013.

[34] M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer. Compiler/interpreter generator system lisa. In *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*, pages 10 pp.–, Jan 2000.

[35] Mibench. http://www.eecs.umich.edu/mibench/index.html, 11 2013.

[36] Christopher Mills, Stanley C. Ahalt, and Jim Fowler. Compiled instruction set simulation. *Software: Practice and Experience*, 21(8):877–889, 1991.

[37] Eliot Miranda. Brouhaha - a portable smalltalk interpreter. *SIGPLAN Not.*, 22:354–365, December 1987.

[38] Preeti Ranjan Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

[39] Richard P. Paul. *SPARC Architecture, Assembly Language Programming, and C.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1999.

[40] D.A. Penry and K.D. Cahill. Adl-based specification of implementation styles for functional simulators. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 165–173, July 2011.

[41] Perf. https://perf.wiki.kernel.org/index.php, 10 2013.

[42] Hongguang Ren, Zhiying Wang, Wei Shi, and D. Edwards. Critical path analysis in data-driven asynchronous pipelines. In *Computer Communication and Informatics (ICCCI), 2012 International Conference on*, pages 1–9, Jan 2012.

[43] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An efficient retargetable framework for instruction-set simulation. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 13–18, Oct 2003.

[44] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Design Automation Conference, 2003. Proceedings*, pages 758–763, June 2003.

[45] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. Archc: a systemc-based architecture description language. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 66–73, Oct 2004.

[46] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. *SIGPLAN Not.*, 31:150–159, September 1996.

[47] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical report, Seminar on Mobile Code, Number TKO-C-79, Laboratory of Information Processing Science, Helsinki University of Technology, 1995, 1996.

[48] A.G. Saidi, N.L. Binkert, S.K. Reinhardt, and T. Mudge. Full-system critical path analysis. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 63–74, April 2008.

[49] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[50] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[51] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of java applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[52] Nguyen T. Thanh and E. Walter Raschner. Indirect threaded code used to emulate a virtual machine. *SIGPLAN Not.*, 17:80–89, May 1982.

[53] E. Tune, Dongning Liang, D.M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 185–195, 2001.

[54] V. Zivojnovic, S. Pees, and H. Meyr. Lisa-machine description language and generic machine model for hw/sw co-design. In *VLSI Signal Processing, IX, 1996., [Workshop on]*, pages 127–136, Oct 1996.