Um Simulador Compilado Dinâmico para o ArchC

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Maxiwell Salvador Garcia e aprovada pela Banca Examinadora.

Campinas, 09 de Fevereiro de 2012.

Prof. Dr. Sandro Rigo (Orientador)

Prof. Dr. Rodolfo Azevedo (Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR ANA REGINA MACHADO – CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA – UNICAMP

G165s

Garcia, Maxiwell Salvador, 1986-
Um simulador compilado dinâmico para o ArchC / Maxiwell Salvador Garcia. – Campinas, SP : [s.n.], 2011.

Orientador: Sandro Rigo.
Coorientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Simulação (Computadores). 3. Hardware - Linguagens descritivas. I. Rigo, Sandro, 1976-. II. Azevedo, Rodolfo Jardim de, 1974-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em inglês: Dynamic compiled simulator for ArchC
Palavras-chave em inglês:
Computer architecture
Computer simulation.
Computer hardware description languages
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Sandro Rigo [Orientador]
Marcio Seiji Oyamada
Edson Borin
Data da defesa: 25-11-2011
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 25 de novembro de 2011, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Marcio Seiji Oyamada
UNIOESTE

Prof. Dr. Sandro Rigo
IC / UNICAMP

Prof. Dr. Edson Borin
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Um Simulador Compilado Dinâmico para o ArchC

Maxiwell Salvador Garcia[1]

Fevereiro de 2012

Banca Examinadora:

- Prof. Dr. Sandro Rigo (Orientador)
- Prof. Dr. Edson Borin
  Instituto de Computação – UNICAMP
- Prof. Dr. Marcio Seiji Oyamada
  Colegiado de Ciência da Computação – UNIOESTE
- Prof. Dr. Guido Araújo (Suplente)
  Instituto de Computação – UNICAMP
- Prof. Dr. Cristiano Coêlho de Araújo (Suplente)
  Centro de Informática – UFPE

[1]Suporte financeiro de: Bolsa do CNPq (processo 134749/2009-0) 2009–2010, Projeto Fapesp (processo 2009/13230-9) 2010–2011

ERRATA

Na folha iv - **Onde se lê:** Fevereiro de 2012 **Leia-se:** 25 de Novembro de 2011

Prof. Dr. Paulo Lício de Geus
Coordenador de Pós Graduação
Instituto de Computação UNICAMP
Matr.103268

Resumo

O simulador é uma das ferramentas mais importantes para o desenvolvimento de uma nova arquitetura computacional. Entre as vantagens que ele apresenta destacam-se a flexibilidade e o baixo custo. Os primeiros simuladores eram criados manualmente, uma prática muito propensa a erros. Atualmente, Linguagens de Descrição de Arquiteturas (ADLs) facilitam a geração dessas ferramentas.

O foco deste trabalho é a pesquisa em técnicas de simulação rápida utilizando a ADL ArchC. Partindo do estado da arte nesta área, a simulação compilada, conseguiu-se melhorar ainda mais o desempenho dos simuladores de conjunto de instruções. Duas abordagens compilada foram usadas. A primeira é uma abordagem estática, que analisa e decodifica o binário previamente e especializa o simulador para aquela aplicação, deixando a simulação com um alto desempenho. As simulações ficaram apenas 5 vezes mais lentas, na média, que execuções nativas em máquina Intel, com desempenho atingindo 900 milhões de instruções por segundo.

A segunda abordagem é a dinâmica, que não exige o conhecimento prévio da aplicação, evitando a sobrecarga inicial de se especializar o simulador. Com essa abordagem é possível, também, simular aplicativos que sofrem modificações em seu próprio código, como *boot-loader* e sistemas operacionais. A decodificação e compilação do aplicativo são feitas em tempo de execução, fazendo uso da infraestrutura LLVM. O desempenho de simulação só não superou o estático, alcançando uma média de 140 milhões de instruções por segundo. Considerando-se a sobrecarga de geração do simulador compilado estático, a abordagem dinâmica torna-se mais rápida, mostrando-se uma excelente alternativa ao projetista que não tem o interesse em ficar simulando repetidas vezes a mesma aplicação.

Abstract

The simulator is one of the most important tools to design a new computer architecture. It has many advantages, the most important are flexibility and low cost. The first simulators were written from scratch, which was an error-prone practice. Nowadays, Architecture Description Languages (ADLs) simplify the generation of these tools.

This work focus on the research of new fast simulation techniques using the ArchC ADL. Beginning from the state-of-art in this area, the compiled simulation, is was possible to speed-up the instruction set simulation performance even higher. Two approaches have been used. The first is static compiled simulation, which analyzes and decodes the binary, and specializes the simulator for that application, improving the simulation and reaching high performance. The simulations were only 5 times slower, on average, if compared to native execution on an Intel machine, reaching 900 million instructions per second.

The second approach is a dynamic compiled simulation, which requires no knowledge about the application, avoiding the overhead of specializing the simulator. With this approach it is possible to simulate sef-modifying code, such as in boot-loaders and operating systems. The application is decoded and compiled at runtime, using the LLVM framework. The simulation performance reaches an average of 140 million instructions per second, not overcoming the static approach. However, if you consider the overhead of generating the static compiled simulator, the dynamic approach becomes better, being an excellent alternative to the designer who has no interest in repeating simulations for the same application.

Agradecimentos

Apesar do homem ser um ser sociável, podendo cruzar e interagir com milhões de pessoas, o sentimento de solidão é facilmente experimentado, pela complexa natureza do ser e estar. Por isso, devo muito às pessoas que entraram em minha vida e permitiram que tal sentimento fosse menos presente.

Primeiramente, deixo meu profundo agradecimento aos meus pais, que me ofereceram condições ideais para concluir este curso, apoiando-me sempre nas decisões. Mesmo distantes, os senti mais presentes que antes fora, sendo meu guia e incentivo para continuar a caminhada. À minha mãe, meu exemplo de força de vontade e fé, nunca se deixando abalar, mesmo nos mais difíceis dias.

Quero agradecer aos meus orientadores e amigos Sandro e Rodolfo, que participou diretamente na construção desde trabalho, e "se enxerguei mais longe, foi porque estava sobre os ombros de gigantes". Aos meus amigos, que, neste período de festas e estudos, fizeram parte de minha vida, ajudando-me tanto em caráter pessoal, quanto acadêmico.

Agradeço ainda as agências financiadoras que proporcionaram bolsa de estudos para eu me manter financeiramente, permitindo-me dedicar em tempo integral ao mestrado. Obtive bolsa da CNPq (134749/2009-0) no primeiro ano e da FAPESP (2009/13230-9) no ano seguinte.

Enfim, agradeço a todos que contribuíram para meu crescimento neste período de pós-graduação, e o menino que entrou, hoje se despede como homem.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A crescente demanda por dispositivos com grande poder de processamento fez a indústria introduzir, em uma única pastilha, processadores programáveis embarcados à vários outros elementos de hardware, cunhando-se o termo SoC (*System-on-Chip*). Desenvolver SoC sofisticados pode ser altamente complexo, pois os processadores e os componentes de hardware existentes, muitas vezes, não satisfazem os requisitos estabelecidos, sendo necessário explorar o espaço de projeto [3]. Devido ao alto custo e ao *time-to-market*, fabricar vários novos processadores e componentes para, enfim, encontrar a melhor opção é inviável. Outro agravante é o atraso no desenvolvimento do software embarcado que depende de protótipos para testes e correções. Por conta destes e outros problemas, fica evidente a necessidade de modelos de simulação com rápida adaptabilidade nas etapas de desenvolvimento [3]. Entende-se por modelos de simulação as ferramentas que oferecem mecanismos para que uma máquina hospedeira execute uma aplicação de uma máquina alvo, podendo avaliar o desempenho, estimar a potência média dissipada [10] e oferecer uma plataforma para que testes e correções possam ser feitos no software embarcado muito antes do protótipo real.

Ao simular uma determinada arquitetura almejando uma análise de desempenho,

gasta-se muito tempo de simulação, pois vários detalhes são modelados para que o comportamento do hardware seja priorizado, com precisão de ciclo (do inglês, *cycle-accurate*). Porém, muitas vezes, o objetivo do simulador, inicialmente, é testar ideias com base apenas nas funcionalidades da arquitetura, ou utilizá-lo como plataforma para o desenvolvimento do software embarcado. Nestes simuladores funcionais, um grande ganho de desempenho pode ser alcançado se comparado aos que oferecem precisão de ciclo, pois menos detalhes são modelados e os binários da arquitetura alvo não precisam ser interpretados para angariar estatísticas de desempenho. Técnicas avançadas para se construir simuladores funcionais rápidos foram propostas por diversos trabalhos [52, 24, 22, 30, 16, 39, 23, 47], com o nome de simulação compilada, nas derivações estática ou dinâmica. Ambas as técnicas traduzem blocos de instruções do binário da arquitetura simulada para instruções de máquina da arquitetura hospedeira. Porém, na estática, as traduções são feitas *offline*, isto é, antes de iniciar a simulação é criado um novo simulador com o código traduzido embutido; enquanto que na dinâmica, as traduções são feitas durante a simulação, sendo necessário compilar o simulador apenas uma vez, mesmo que mude o binário da simulação. Para diminuir o *overhead* da tradução em tempo de execução, no caso da simulação com compilação dinâmica, e para permitir a execução de códigos que não são estaticamente conhecidos, no caso da compilação estática, ambas as técnicas podem ser mescladas com a técnica de interpretação.

Modelar um processador em RTL (*Register Transfer Level*) ou em linguagens como C e C++ é complexo e demorado, podendo acumular vários erros no decorrer do processo. Houve, portanto, uma eminente necessidade de se construir Linguagens de Descrição de Arquiteturas (do inglês *Architecture Description Language*, ADL) que atuassem num nível de abstração mais elevado que RTL, e que conseguissem gerar simuladores de maneira automatizada para diferentes propósitos. Além dos simuladores, muitas ADLs atuais,

como ArchC [48] e LISA [40], derivam automaticamente ferramentas auxiliares de extrema importância para a equipe de desenvolvimento, como montadores e ligadores. Com isso, a exploração do espaço de projeto (do inglês *Design Space Exploration*, DSE) é acelerada, pois novas arquiteturas são testadas e suas funcionalidades são examinadas com cuidado pelos programadores do software embarcado, sem gastar tempo e dinheiro com protótipos físicos.

A maioria das ADLs permitem a descrição de processadores em diversos níveis de abstração. Normalmente, inicia-se o modelo dos processadores com poucos detalhes, em um alto nível de abstração, permitindo a geração de simuladores que descrevem as funções das arquiteturas. Com o andamento do projeto, mais detalhes vão sendo modelados para que, apenas com simulação, o desempenho seja previsto com a precisão de ciclos executados.

1.1 Objetivos

Uma das ferramentas mais importantes que o projetista necessita ao desenvolver uma nova arquitetura é o simulador. Ele é o meio mais rápido de testar novos conceitos antes mesmo da realização de um protótipo da arquitetura.

Os primeiros simuladores gerados automaticamente com a ajuda de ADLs utilizaram uma técnica de simulação chamada interpretada, que imita o comportamento do hardware: busca *bytes* na memória, decodifica esperando uma instrução conhecida e executa. Devido ao desempenho insuficiente desta abordagem, uma nova técnica que pré-calcula algumas operações em tempo de compilação e guarda seus resultados foi idealizada, evitando repetir os mesmos cálculos durante a simulação. A esta técnica se deu o nome de simulação compilada estática.

ArchC [2] é uma ADL de código aberto baseado em SystemC [57], concebida para

gerar simuladores de variados propósitos e ferramentas auxiliares, além de possibilitar a construção de ambientes multiprocessados. Devido a grandes mudanças internas de estrutura e lógica, do ArchC 1.6 [47] para o 2.0 [53], o gerador de simulador compilado estático, o **accsim** [5], não foi suportado na versão 2.0. O primeiro foco deste trabalho foi modificar e adaptar o accsim para que a nova versão do ArchC, a 2.1, tivesse novamente um gerador de simulador rápido utilizando a técnica compilada estática. Além disso, melhorias foram implementadas resultando em um simulador compilado muito mais rápido que aquele encontrado na versão anterior. A média do desempenho para os modelos MIPS I, SPARC V8 e PowerPC foram 640, 423 e 356 milhões de instruções por segundo (MIPS), respectivamente, executando em um i7 de 2.8GHz. Devido ao alto desempenho, a velocidade de simulação ficou, na média, apenas 5 vezes mais lenta em uma execução nativa em máquina Intel do mesmo aplicativo. O trabalho referente à simulação compilada estática no ArchC 2.1 foi publicado em 2010 [17].

Com o intuito de sanar as deficiências encontradas pelo simulador compilado estático, como o alto *overhead* na construção e compilação do simulador e a impossibilidade de executar binários que não possuam código estático, foi implementado um gerador de simulador compilado dinâmico, o **acdcsim**. Nestes simuladores, a aplicação da técnica compilada passa a ser feita em tempo de execução e apenas em alguns blocos de códigos escolhidos sob demanda pelo simulador. Para isso, foi utilizado a técnica *Just-in-time* (JIT), da infraestrutura LLVM [29], como suporte de tradução e otimização de código do binário alvo para máquina hospedeira. Os resultados foram satisfatórios, alcançando uma média de 163 milhões de instruções por segundo (MIPS) para a arquitetura MIPS I, com picos de 280 MIPS. Para as arquiteturas SPARC V8 e PowerPC, as médias ficaram 120 e 135 MIPS, respectivamente. Apesar de serem números bem inferiores aos da simulação compilada estática, não há desperdício de tempo na construção do simulador, o que via-

biliza o uso desta abordagem para simulações em que os aplicativos não são reutilizados várias vezes. Além disso, possibilita a simulação rápida de aplicativos que modificam seu próprio código, porém esta funcionalidade, nesta versão do simulador compilado, exige modelos que tenham instruções de tamanho fixo.

1.2 Organização do trabalho

O restante desta dissertação está organizada da seguinte forma:

O Capítulo 2 resume diferentes abordagens relacionadas à simulação de conjunto de instruções. Comenta sobre as primeiras técnicas usadas na geração de simuladores, a facilidade de redirecionamento introduzida com o uso de Linguagens de Descrição de Arquiteturas (ADL) e técnicas recentes para aumento de desempenho dos simuladores.

O Capítulo 3 e 4 discutem os detalhes de implementação da técnica compilada estática e dinâmica do ArchC 2.1, respectivamente, sendo estas as contribuições científicas do presente trabalho de mestrado.

O Capítulo 5 conclui esta dissertação resumindo e mostrando a importância das contribuições deste trabalho, e algumas propostas de trabalhos futuros são sugeridas na última seção.

Capítulo 2

Trabalhos Relacionados

Simulação de arquiteturas tem sido o foco de muitos trabalhos recentes. A ampla gama de técnicas, das mais variadas, permite desde simulações precisas porém lentas a simulações rápidas com restrições. Pesquisas atuais caminham na busca de um equilíbrio entre desempenho, precisão e flexibilidade, resultando em simuladores que atingem bons resultados para variadas situações. Derivar automaticamente estes simuladores a partir de Linguagens de Descrição de Arquiteturas (ADL) permite que o projetista explore novas arquiteturas apoiando-se em técnicas rápidas de experimentação.

Este capítulo está divido em duas partes, sendo a primeira uma revisão dos principais simuladores de arquiteturas e a segunda uma revisão das principais ADLs existentes.

2.1 Simuladores arquiteturais

Antes do advento das ADLs, a construção de simuladores era um processo manual e demorado, que envolvia muita codificação após um estudo detalhado sobre a arquitetura alvo. Depois de concluído, o projetista podia variar inúmeros parâmetros do simulador e optar pela melhor configuração arquitetural — como tamanho da cache e do TLB (*Translation*

Lookaside Buffer), tipo de predição de desvio e número de unidades funcionais.

É grande o número de simuladores encontrados na literatura, tanto funcionais — que executam o programa corretamente mas não modelam qualquer comportamento de hardware — quanto simuladores com precisão de ciclos — que modela partes internas do processador para gerar estatísticas para diferentes propósitos. Por conta da complexidade e do nível de detalhes usado no modelamento, os simuladores da segunda classe possuem um desempenho inferior aos da primeira.

Nas próximas seções, simuladores comportamentais e funcionais serão discutidos. Porém, devido ao foco deste trabalho, os funcionais serão mais explorados.

2.1.1 Simuladores com precisão de ciclos

Quando novas arquiteturas são propostas, é necessário saber quão eficiente elas serão. Para isso, os simuladores devem ser capazes de retornar um conjunto de informações, ao final da simulação, para que os projetistas opinem sobre a arquitetura.

O simulador **sim-outorder** — pertencente ao SimpleScalar Tools [11] — informa os projetistas do desempenho atingido pela arquitetura para um determinado binário de entrada, como número de instruções, número de ciclos, *cache miss*, dentre outras informações. No entanto, não se restringem apenas ao desempenho, podendo ter simuladores que estimam, também, a energia dissipada em cada parte do processador e um somatório final. Ao sim-outorder, foi acoplado um *framework* que, por meio da biblioteca CACTI, calcula a energia média dissipada de várias partes do processador. O *framework* recebeu o nome de Wattch e o simulador passou a se chamar de Sim-Wattch [10]. As ferramentas SimpleScalar podem simular os conjuntos de instruções Alpha, PISA, ARM e x86.

MPARM [9] é um ambiente completo para a exploração do espaço de projeto de MP-SoC em SystemC, composto por modelos de processadores, barramentos (AMBA e NoC)

e memórias. O conjunto de instrução simulado é um ARM descrito em C++ e encapsulado em SystemC. Recentemente, o *framework* MPARM foi estendido para trabalhar também com modelos funcionais em ArchC [35, 34].

Quanto às ADLs, várias permitem um detalhamento do processador para que os simuladores gerados ofereçam precisão de ciclo, como veremos na seção 2.2.

2.1.2 Simuladores funcionais

Quando o objetivo da simulação é apenas funcionalidades, não é necessário simular todos os detalhes da arquitetura, posto que isto consome muito tempo. Projetistas que trabalham no desenvolvimento de aplicativos sofisticados para arquiteturas complexas, não estão interessados, a princípio, em saber estatísticas de desempenho. Eles querem ter uma resposta garantida que seu aplicativo está funcionando corretamente, o mais breve possível. É neste cenário de depuração e de testes de funcionalidades que os simuladores funcionais rápidos tornam-se ferramentas de grande valor. Paralelizar um complexo decodificador de vídeo em 2004, denominado OpenDivX, para executar em um *System-on-Chip* multiprocessado exigiu ferramentas de simulação funcional de alto desempenho [58]. Deste modo, a depuração do software pôde ser feita antes mesmo do hardware estar pronto. Percebe-se que testes de desempenho são dispensáveis nesta etapa de um projeto, pois o foco é o funcionamento correto do aplicativo em uma futura arquitetura que ainda não se tem implementada em hardware.

Alguns dos simuladores abaixo, além de funcionais, também podem ser refinados para simulação com precisão de ciclo. Como o interesse nesta classe de simuladores é o alto desempenho para se testar os binários, muitos trabalhos apresentam otimizações que modificam a abordagem clássica de simulação. A abordagem clássica reproduz o comportamento do hardware: carrega uma instrução da memória, decodifica e executa, e assim

sucessivamente até terminar o programa. Este paradigma é denominado de **simulação interpretada**. Porém, a crescente complexidade dos sistemas tem tornado este modelo tradicional ineficiente. Vários trabalhos têm utilizado a ideia de compilar previamente segmentos de códigos de uma dada arquitetura para códigos de máquina da arquitetura hospedeira para melhorar o desempenho das simulações. A esta técnica é dado o nome de simulação compilada. Quando esta compilação é feita *offline*, isto é, antes do binário ser executado pelo simulador, o nome estático é utilizado. Porém, se essa compilação for feita em tempo de execução do binário, conforme a necessidade do simulador, então denomina-se simulação compilada dinâmica.

O **SimICS** [31] é um simulador *full-system* desenvolvido no Swendish Institute of Computer Science (SICS) e simula vários processadores de arquiteturas comerciais como SPARC V8, Pentium IV e Itanium, no nível funcional. O nome *full-system* vem da possibilidade de simular, além de programas individuais, sistemas operacionais (SO) comerciais completos. O nível de detalhe da simulação pode ser escolhido pelo usuário e, dependendo do nível escolhido o desempenho fica em torno de 20 a 200 vezes mais lento do que o hardware original, atingindo 6.62 MIPS (Milhões de Instruções por Segundo) para carregar o sistema Solaris 8 em um Sparc-U2. SimICS também permite refinamento para precisão de ciclos, com versão *out-of-order*.

Dentro de SimpleScalar Tools [11] há dois simuladores funcionais, o **sim-fast** e o **sim-safe**. O primeiro possui melhor desempenho pois não realiza verificações de alinhamento nem checa as permissões em referências de memória. Como comparativo de velocidade, o sim-outorder, que é um simulador com precisão de ciclo do SimpleScalar Tools, simula 0.3 MIPS, enquanto o sim-fast e o sim-safe simulam 7 e 6 MIPS, respectivamente.

O **Shade** [14] emula um sistema alvo fazendo uma tradução dinâmica do código binário da máquina alvo para o código nativo da máquina hospedeira. Porém, ele não é redire-

cionável, apenas algumas combinações de máquinas hospedeira/alvo foram produzidas dentre arquiteturas SPARC e MIPS. O alto desempenho do Shade faz com que suas emulações sejam apenas 2.3x (para ponto flutuante) a 6.2x (para inteiro) mais lento do que uma execução direta em uma máquina SPARC.

O **SimOS** [49] é um ambiente de simulação pra sistemas multiprocessados. Permite a simulação eficiente de SOs comerciais. Técnicas de compilação dinâmicas são utilizadas para melhorar o desempenho da simulação. A simulação dos múltiplos núcleos é feita de forma determinística, isto é, quando os parâmetros são mantidos constantes, a execução de uma simulação sempre atinge os mesmo resultados.

SyntSim [12] é um simulador funcional desenvolvido pela Cornell University que utiliza C como representação intermediária. Parte do código do binário é traduzido, estaticamente, em grandes funções C, podendo, em algumas vezes, causar *overflow* no compilador C. O usuário pode, se preferir, especificar aproximadamente a fração de instruções para ser simulada em modo compilado ou utilizar um *profile* para selecionar as instruções mais frequentemente executadas para traduzir. A parte restante do binário é executada de modo interpretado. Com esta combinação da técnica compilada com a interpretada, o simulador mostrou-se muito versátil e, realizando experimentos com a arquitetura Alpha, seu desempenho foi, na média, 6.6x mais lento que uma execução nativa.

Várias ADLs oferecem recursos para gerar simuladores compilados estáticos e dinâmicos [23, 40, 48], como é o caso da LISA, EXPRESSION e ArchC, comentados na próxima seção.

2.2 ADLs

Linguagens de descrição de hardware (HDL) como VHDL [41], Verilog [51] e, SystemC [55] são bastante utilizadas para modelar e simular processadores, mas com o objetivo

de desenvolver o protótipo em hardware. O uso destes modelos para exploração de arquiteturas e geração de ferramentas de desenvolvimento de software tem duas principais desvantagens: (i) não existe informação sobre o conjunto de instruções do processador, como a sintaxe do montador e (ii) eles cobrem um grande número de detalhes de implementação desnecessários para avaliação de desempenho, simulação de ciclos e verificação de software embarcado, impactando significativamente na velocidade da simulação [50].

Um salto de flexibilidade ocorreu com o surgimento, na década de 90, das Linguagens de Descrição de Arquiteturas, que permitiram uma modelagem em um nível elevado de abstração. Destes modelos, os interpretadores das ADLs derivam automaticamente simuladores e ferramentas auxiliares de programação, como montadores, depuradores e compiladores.

Diversas ADLs foram propostas recentemente com o objetivo de sanar deficiências encontradas nas primeiras, porém nenhuma delas tornou-se um padrão. Por ser um campo de pesquisa relativamente novo, as ADLs devem ser melhor estudadas para realização eficiente da exploração do espaço de projeto para arquiteturas de SoC. Para que uma ADL seja próxima do ideal, alguns aspectos são importantes [4]:

Especificação concisa e natural: o projetista do sistema deve ser capaz de escrever as especificações da arquitetura de maneira simples e natural, reduzindo o tempo de modelagem e facilitando uma futura leitura do código. Ser concisa e não redundante também evita erros difíceis de serem encontrados.

Generalidade na especificação: as especificações suportadas pela a ADL devem ser capazes de modelar uma ampla gama de processadores, desde os mais simples RISCs (*Reduced Instruction Set Computer*) até os mais complexos VLIWs (*Very Long Instruction Word*), superescalares e DSP (*Digital Signal Processor*) com *datapaths* irregulares. Hierarquias de memória compostas por DRAMs, memórias flash e *cache* também devem

ser suportadas.

Geração automática de ferramentas: Uma ADL ideal deve possibilitar a geração automática de ferramentas de software de qualidade, que incluem, pelo menos, um compilador com capacidade de gerar código para arquiteturas paralelas em nível de instrução (VLIW e superescalares), os chamados compiladores ILP, e um simulador com precisão de ciclos. Porém, estas ferramentas requerem informações detalhadas da arquitetura, tipicamente de forma não concisa e de difícil especificação. Portanto, tornam-se necessários procedimentos que geram automaticamente estas informações a partir da especificação ADL. Como exemplo, tem-se a especificação de conflito de recursos entre instruções para compiladores ILP, pois somente assim o compilador pode gerar um código otimizado.

Dentre as ADLs existentes, pode-se verificar três grandes grupos: as ADLs de estruturas, as de comportamento e as híbridas. As primeiras têm o foco na descrição da estrutura de um processador ou como é a ligação das unidades funcionais entre si para formação de pipeline, acessos à memória e aos registradores. Com isso, é possível derivar ferramentas que façam simulações com precisão de ciclo. As ADL de comportamento focam-se no detalhamento do que é visto pelo usuário da arquitetura, como o projeto do conjunto de instruções e a funcionalidade oferecida por essas instruções, permitindo simulações funcionais rápidas. Na classe das híbridas, é dada igual importância à estrutura e ao comportamento do processador e por essa versatilidade, estão atualmente em ascensão. Todas as ADLs descritas abaixo são híbridas.

A **MDes** [21] é uma ADL para DSE de processadores de alto desempenho. Ela captura a estrutura e o comportamento de processadores em forma de seções hierárquicas. As restrições para execução de operações são descritas com tabelas de reservas. MDes é usada em uma infraestrutura integrada de compilação e monitoramento de desempenho chamada Trimaran. O recurso de simulação com compilação dinâmica ou estática não

parece disponível.

FlexWare [38] é utilizada para desenvolvimento de DSPs e ASIPs (*Application Specific Integrated Processor*). A descrição consiste em três componentes: o conjunto de instruções, os recursos e o grafo de interconexão que representa o caminho dos dados. Hierarquia de memória parece não ser possível.

EXPRESSION [23], desenvolvida na Universidade da California em Irvine, é uma ADL focada em arquiteturas RISC e VLIW, permitindo também a modelagem da memória. A estrutura é descrita como componentes e suas interconexões. O *pipeline* pode ser detalhado, informando os estágios de cada unidade funcional e suas temporizações. O comportamento do conjunto de instruções é descrito de forma hierárquica. Os conflitos de recursos não são descritos explicitamente, mas as tabelas de reserva são construídas de forma automática [20] e passadas para um compilador ILP. Além de um simulador a nível de ciclos [27], esta ADL implementa uma técnica para simulação compilada chamada IS-CS (*Instruction-Set Compiled Simulation*) [45], que melhora o desempenho e a flexibilidade de simuladores compilados. Em 2003, tanto a linguagem quanto a ferramenta de simulação interpretada foram colocadas em domínio público. Porém, a ferramenta que apresenta a técnica IS-CS não está disponível no *site* da ADL e não foi possível localizar outras fontes para *download*.

LISA [40, 59] é uma ADL comercial e de código fechado, desenvolvida em *RWTH Aachen University* com parcerias de empresas como Synopsys [56] e IBM [19], provê suporte a diversos modelos de processadores e seus simuladores podem ou não possuir precisão de bits e ciclos. Sua principal característica é a especificação do *pipeline* por operações, o que permite a modelagem de técnicas complexas de *interlock* e *bypass*. Cada instrução consiste em múltiplas operações que são definidas como transferências entre registradores durante um passo de controle, que pode ser uma instrução, um ciclo, ou

uma fase do ciclo, dependendo do nível de precisão desejado. Os conflitos de recursos não são escritos explicitamente, mas através de operações que ativam ações de parada, de deslocamento e de descarga do *pipeline*. Além dos simuladores, ferramentas auxiliares para desenvolvedores, como ligadores, compiladores e depuradores, também podem ser geradas automaticamente. Simulação compilada estática [9] e dinâmica, conhecida como JIT-CSS [36], também são técnicas suportadas pela ADL LISA, sendo uma das primeiras ADLs a oferecer estas abordagens de simulação rápida. Outro modo de simulação suportado pela LISA é a híbrida [28], que oferece ao projetista a opção de simular partes do aplicativo com precisão de ciclo ou de fase, e partes de maneira rápida e sem precisão, explorando apenas a funcionalidade.

RADL [54] é uma extensão da LISA proposta pela empresa RockWell Semiconductor Systems que suporta um modelo de *pipeline* mais detalhado, com *delay slots*, interrupções, *zero-overhead loops*, *hazards* e múltiplos *pipelines*. Simuladores com precisão de bits e de ciclos (ou fases) podem ser gerados para DSPs.

ArchC [48, 47] é uma ADL baseada em C e SystemC, desenvolvida no Laboratório de Sistemas de Computação do IC-UNICAMP e está disponível publicamente sob a licença GPL no site `http://www.archc.org`. Tendo expressividade suficiente para modelar várias classes de arquiteturas (RISC, CISC, DSPs, etc), o projetista explora rapidamente um novo conjunto de instruções, gerando várias ferramentas automaticamente, como simuladores, montadores e depuradores. O diferencial do ArchC está na facilidade de gerar um primeiro modelo funcional, necessitando de poucas informações para que um simulador do conjunto de instruções possa ser gerado para testes iniciais. Outra característica importante é a verificação de modelos em níveis de abstração diferentes, sendo um modelo ArchC comparado com um outro modelo de referência descrito também em ArchC, mas com outro nível de abstração, ou diretamente em SystemC.

Na atual versão, além dos processadores, é possível descrever plataformas inteiras, com barramentos, processadores, memórias, entre outros IPs (*Intellectual Property*), utilizando TLM (*Transaction-level modeling*) e SystemC. Para facilitar a construção destas plataformas, um *framework* denominado ArchC Reference Plataform (ARP) pode ser utilizado [1].

A descrição de um modelo de processador em ArchC é dividido em duas partes, deixando claro o que é informação comportamental e estrutural. A descrição *Instruction Set Architecture* (AC_ISA) contém detalhes sobre o formato, o tamanho e o nome das instruções, combinado com toda informação necessária para decodificá-las. O comportamento de cada instrução é feito em C++, aumentando a familiaridade de projetistas para com a ferramenta. Na descrição *Architecture Resources* (AC_ARCH), o projetista informa sobre os dispositivos de armazenamento, estrutura do *pipeline*, entre outros detalhes. Com essas duas descrições em mãos, o gerador `acsim`, do pacote ArchC, pode gerar um simulador interpretado escrito em SystemC, que pode ser funcional ou com precisão de ciclo, dependendo do nível de abstração utilizado. Simuladores compilados funcionais, estáticos e dinâmicos, também podem ser gerados por meio do `accsim` e do `acdcsim`, respectivamente, sendo estes geradores a contribuição deste trabalho para o pacote de ferramentas do ArchC. O Capítulo 3 está reservado à simulação compilada estática e o Capítulo 4 à simulação compilada dinâmica.

A Tabela 2.1 exibe um resumo comparativo das principais características de ADLs. LISA e EXPRESSION são os dois projetos com maior similaridade com ArchC, e por isso, foram escolhidos.

Característica	LISA	EXPRESSION	ArchC
Conjunto de Instruções	X	X	X
Precisão de Ciclos	X	X	X
Suporte a Multiciclo	X	X	X
Suporte a *Pipeline*	X	X	X
Hierarquia de Memória	X		
Emulação de S.O.	X		X
Hierarquia de Comportamento			X
Co-verificação	X		X
Geração de Simuladores SystemC	X		X
Geração de *Linker*	X	X	X
Simulador Funcional com Compilação Estática	X	X	X
Simulador Funcional com Compilação Dinâmica	X	X	X
Simulador *Cycle-Accurate* com Compilação Estática	X		
Suporte a Multiprocessadores	X		X

Tabela 2.1: Comparação entre as principais ADLs

2.3 Simuladores modernos

Tradicionalmente, a simulação de processadores é feita de uma maneira similar ao que ocorre no hardware. Os passos de busca da instrução, sua decodificação e execução são feitos sequencialmente para cada instrução. Este método é referenciado como simulação interpretada.

O significativo aumento de complexidade das novas arquiteturas provoca um impacto negativo no desempenho dos simuladores. Atualmente, algumas técnicas tornam os simuladores rápidos tão flexíveis quanto os tradicionais. Porém, construí-los ainda exige um grande esforço e altos custos em programação. Alternativamente, as linguagens de descrição de arquitetura (ADLs) oferecem a infraestrutura que o projetista necessita, gerando automaticamente os simuladores, compiladores, montadores e depuradores. Assim, as ADLs ganharam força nesses últimos anos e têm sido utilizadas com sucesso para especificar conjuntos de instruções de novas arquiteturas.

Diversas técnicas para agilizar a simulação, antes encontradas apenas em simuladores que eram implementados manualmente, agora estão presentes em simuladores gerados automaticamente por diversas ADLs. Os primeiros trabalhos trataram de amenizar o custo de decodificação ao inserir no simulador uma *cache* de instruções decodificadas. Esta otimização provoca um grande ganho de desempenho, já que, em laços, as instruções são decodificadas apenas na primeira rodada (se a *cache* for do tamanho do laço ou maior).

Um outro avanço foi constatar que muitas operações do simulador poderiam ser pré-calculadas antes do início da simulação. Assim, cada vez que o simulador precisasse de um resultado, ele já estaria calculado. Este raciocínio é a base para o que se chamou de simulação compilada, que é uma simulação que sofre um pré-processamento para otimização.

2.3.1 Simulação compilada

Quando se utiliza a simulação compilada, a fase de busca e decodificação das instruções é realizada antecipadamente, evitando decodificar cada instrução em tempo de execução. Além disso, blocos de códigos do binário a simular são expandidos e analisados antes de serem simulados, permitindo agressivas otimizações.

Quando a técnica compilada utilizada é estática, a decodificação já é feita no momento da compilação do simulador, necessitando conhecimento prévio do software a ser executado. Na dinâmica, o simulador é criado sem conhecimento do software que será executado, porém, antes de iniciar a simulação, um pré-processamento deve ser realizado para que os blocos de códigos sejam decodificados e otimizados sob demanda, evitando a decodificação de cada instrução no momento da simulação daquele trecho.

Gerar simuladores rápidos, com a técnicas sofisticadas de otimizações, têm sido propostas de algumas ADLs, como FACILE [52], Sim-nML [24], ISDL [22], MIMOLA [30],

ANSIC [16], LISA [39], EXPRESSION [23] e ArchC [47].

Braun *et al.* [9] propuseram uma técnica de escalonamento estático baseado na ADL LISA, em que o grande diferencial é a utilização da técnica compilada para melhorar o desempenho de simuladores com precisão de ciclo. Apesar de ser a única ADL a oferecer este recurso, muito trabalho ainda deve ser feito para que o desempenho seja suficiente para atender as necessidades do mercado.

Várias plataformas utilizam a simulação interpretada mesclada com código gerado em tempo de execução para balancear desempenho com flexibilidade. Como exemplo tem-se o simulador SimICS [31], que permite codificar vários aspectos do conjunto de instruções, fazendo avaliações parciais para melhorar o desempenho da simulação. O decodificador se torna o principal gargalo de desempenho. O simulador SyntSim [12] melhora o desempenho da simulação combinando cuidadosamente simulação compilada com interpretada e implementando algumas otimizações, como *dead code elimination*, *inlining*, propagação de constantes entre outras. SyntSim seleciona as instruções para aplicar a técnica compilada baseado-se no perfil de execução, se disponível, ou por meio de algumas heurísticas que tentam balancear ganho de desempenho com custo de compilação. No entanto, esta seleção de instruções para o modo compilado nem sempre é aplicável.

Uma técnica de compilação dinâmica para LISA está presente em Nohl *et al.* [36], denominada *Just-in-Time Cache-Compiled Simulation* (JIT-CCS). A compilação de uma instrução ocorre em tempo de execução, imediatamente antes de ser executada. Logo, a informação extraída é armazenada em uma *cache* para possível reuso, caso esse trecho de código seja executado novamente. O simulador reconhece se o código do programa de um endereço previamente executado foi modificado e inicia uma recompilação. Fazendo uso desta *cache*, a técnica melhora o desempenho da simulação interpretada reduzindo o *overhead* de decodificação. Porém, ele traduz as instruções para uma estrutura sem

otimizações adicionais. Utilizando um Athlon 1.2GHz com 768MB de memória principal, o desempenho médio desta técnica foi de 7 milhões de instruções por segundo (MIPS) simulando uma arquitetura ARM.

Brandner *et al.* [8] apresenta uma ferramenta implementada em uma ADL baseada em XML [7], que gera um simulador com precisão de ciclo com compilação dinâmica. Para a tradução de código, utiliza as bibliotecas da infraestrutura LLVM [29]. A velocidade de simulação média é 43 MHz, com picos de 800MHz, simulando um MIPS com 5 estágios de *pipeline*, porém, o artigo não deixa claro qual a relação entre ciclos e instruções executadas. A ausência de detalhes e a forma como os resultados foram apresentados não permite maiores comentários. O artigo também não informa se a ferramenta está disponível publicamente.

Qin *et al.* [44] propuseram uma abordagem multiprocessada para acelerar a simulação. A simulação, que utiliza compilação dinâmica, é executada em uma *thread* enquanto várias outras ficam traduzindo segmentos do binário. Se um segmento for solicitado para execução e este ainda não está traduzido, ele é executado de modo interpretado e é adicionado na fila para tradução para futuras solicitações. Se modificações acontecerem em segmentos já traduzidos, interrupções são acionadas e estes segmentos são encaminhados para tradução novamente. Executando em apenas um processador, a simulação de um MIPS32 obteve 138 MIPS, 3.55 vezes mais rápida do que a interpretada. Com 4 processadores, atingiu 197 MIPS, sendo 5.06 vezes mais rápida que a simulação interpretada. A técnica de compilação estática possui maior desempenho, alcançando 282 MIPS no mesmo simulador. Porém, se considerar o *overhead* da compilação, os *Speedups* alcançados são de 1.18 e 3.17, para 1 e 4 processadores, respectivamente. Este trabalho não foi incorporado a nenhuma ADL e encontra-se implementado nos simuladores de código aberto SimIt-ARM [42] e SimIt-MIPS [43].

Utilizando a ADL EXPRESSION, Reshadi *et al.* [45, 46] desenvolveram uma técnica de simulação compilada híbrida denominada *Instruction-Set-Compiled Simulation* (IS-CS). Nesta abordagem, diferente do JIT-CSS, a decodificação volta a ser feita em tempo de compilação, como a simulação compilada estática, mas com a possibilidade de re-decodificação no caso de uma instrução ser modificada em tempo de execução. Porém, essa nova decodificação utiliza uma função genérica para a classe da instrução e não uma função otimizada, como ocorre em tempo de compilação. O artigo supõe que isto não impacta em perda considerável de desempenho, pelo fato de o número de instruções modificadas em tempo de execução ser muito pequeno. Mas esta suposição depende da área de aplicação. Outra diferença é o uso de uma técnica batizada de abstração de instrução para gerar instruções decodificadas com otimizações agressivas, explorando o fato de certas classes de instruções possuírem um valor constante para um dado campo. Utilizando um Pentium 3 de 1GHz com 512 MB de memória principal, a simulação teve um desempenho médio de 12 milhões de instruções por segundo (MIPS).

O trabalho de Bartholomeu *et al.* [5], fazendo uso de ArchC 1.6, resultou em um gerador de simulador compilado estático denominado `accsim`. Esta implementação foi uma das primeiras a ter código aberto desta técnica de simulação rápida. O binário, compilado por um compilador *cross-compiler* para a arquitetura alvo, juntamente com descrição da arquitetura em ArchC, são as entradas do gerador. Como se conhece o binário com antecedência, é possível fazer a decodificação das instruções em um pré-processamento, eliminando da simulação. Outra invariante é a ordem das instruções, que permite criar um simulador que escalona o comportamento de cada instrução do programa na mesma ordem original. Cada instrução decodificada do aplicativo corresponde a uma chamada de função no simulador gerado. Estas funções podem ser acessadas aleatoriamente, posto que o destino dos saltos pode ser qualquer instrução. Apenas com estas diretivas básicas de

compilação estática, o desempenho médio em um Athlon de 1GHz foi de 24 MIPS, contra 12 MIPS do IS-CS (*Instruction-Set Compiled Simulation*), da ADL EXPRESSION. Além deste simulador compilado básico gerado pelo `accsim`, duas novas técnicas foram implementadas, denominadas de FSCS (*Fast Static Compiled Simulation*) [5], que atingem de 45 a 200 MIPS em um Pentium 4 de 2.4GHz. Como comparação, a técnica JIT-CCS (*Just-in-Time Cache Compiled Simulation*), desenvolvida para LISA, alcança desempenho máximo de 30 MIPS em um Athlon de 2.5GHz. No próximo capítulo será descrito o esforço realizado neste trabalho para que o `accsim` continuasse funcionando na atual versão do ArchC (2.1), alcançando o desempenho médio de 230 MIPS no Pentium 4 de 2.4GHz e 640 MIPS em um i7 2.8GHz.

Além da técnica compilada estática, o ArchC 2.1, por meio deste presente trabalho, passou a possuir um gerador de simulador compilado dinâmico, denominado `acdcsim`, e seu resultado está exposto no Capítulo 4. Como comparação, em um Pentium 4 de 2.4GHz, a média da simulação da arquitetura MIPS I foi de 86 MIPS.

A Tabela 2.2 exibe um pequeno resumo comparativo entre as técnicas modernas de simulação rápida. Algumas técnicas não estão implementadas em ADLs, mas sim em simuladores específicos, como é o caso do trabalho proposto por Qin *et al.* [44], que está implementado no simulador SimIt-ARM e SimIt-MIPS. Na tabela, as técnicas estão separadas em três tipos de simulação. Na simulação estática todo o binário é decodificado e otimizações são feitas no momento da geração do simulador. A simulação dinâmica otimiza e compila trechos de códigos durante a execução do simulador, sendo mais rápido que a técnica interpretada e mais flexível que a técnica estática. A simulação híbrida é uma combinação de ambas as técnicas, podendo decodificar e otimizar o binário no momento da criação do simulador com a possibilidade de compilar e otimizar alguns blocos de códigos durante a execução.

O trabalho apresentado por Qin *et al.* possui mais de um núcleo de execução, no qual um núcleo fica responsável apenas pela simulação do binário enquanto os outros vão otimizando e compilando blocos de códigos conforme necessidade. A última coluna apresenta o desempenho médio em milhões de instruções por segundo (MIPS) das técnicas e suas respectivas máquinas.

	ADL	**Tipo de simulação**	**Núcleos de execução**	**Desempenho médio em MIPS**	
SyntSim	-	Híbrida	1	-	
Qin *et al.*	-	Dinâmica	1, 2 ou 4	138 (1xP4 2.8Ghz)	197 (4xP4 2.8Ghz)
JIT-CSS	LISA	Dinâmica	1	7 (Athlon 1.2GHz)	21 (Athlon 2.5GHz)
IS-CS	EXPRESSION	Híbrida	1	12 (Athlon 1GHz)	
accsim	ArchC	Estática	1	230 (P4 2.4GHz)	640 (i7 2.8GHz)
acdcsim	ArchC	Dinâmica	1	86 (P4 2.4GHz)	163 (i7 2.8GHz)

Tabela 2.2: Resumo de técnicas modernas de simulação

Capítulo 3

Simulação Compilada Estática

A implementação da técnica de simulação compilada no Projeto ArchC [5], foi importante por ter sido a primeira implementação de código aberto desta técnica de simulação rápida. Além disso, a técnica foi implementada juntamente com duas novas otimizações, chamadas de FSCS (*Fast Static Compiled Simulation*), para que a simulação compilada estática ficasse ainda mais rápida.

Porém, com o advento do ArchC 2.0, o gerador deste simulador compilado estático, *accsim*, parou de ter suporte, e o Projeto ArchC voltou a oferecer apenas os simuladores interpretados. Isto porque o código interno do projeto foi reestruturado para que ferramentas como SystemC e TLM fossem utilizadas com maior facilidade. Neste capítulo está descrito uma visão geral de como a simulação compilada estática foi implementada no ArchC 1.6 [5, 4], e serão detalhadas as modificações que o gerador de simulador compilado estático teve que sofrer para ser compatível com o novo ArchC. Com o uso de agressivas otimizações do GCC [18], juntamente com as duas otimizações FSCS herdadas, a simulação compilada estática do ArchC 2.1, para o modelo MIPS I, ficou apenas 4 vezes mais lenta na média que uma execução nativa em máquina Intel.

3.1 Simulação interpretada em ArchC

A primeira ferramenta desenvolvida no Projeto ArchC foi o simulador interpretado. A técnica de simulação interpretada segue exatamente os mesmos passos do hardware e é o modo natural de se pensar em um simulador de arquiteturas.

```
1  void sparcv8::behavior() {
2
3     /* Procura instrucao decodificada na cache */
4     if (dec_cache[ac_pc]->valid)
5        decoded = dec_cache[ac_pc]->entry;
6
7     /* Nao esta na cache, decodifica */
8     else {
9        fetch = IM->read(ac_pc);
10       decoded = Decode(fetch);
11    }
12
13    /* Executa comportamentos */
14    ISA->instruction->behavior();        // generico
15    decoded->format->behavior();         // do formato
16    decoded->instruction->behavior();    // da instrucao
17
18 }
```

Figura 3.1: A rotina principal *simplificada* da simulação interpretada

A Figura 3.1 mostra, de forma simplificada, os passos tomados no núcleo da simulação interpretada de ArchC. A descrição de um comportamento de instrução, em ArchC, é feita de forma hierárquica. Primeiro, um comportamento genérico é executado para qualquer instrução. Em seguida, executa-se um comportamento específico ao grupo daquela instrução. Finalmente, o comportamento específico da instrução é invocado. As linhas 14 a 16 executam esses comportamentos. Doravante, essa hierarquia será apresentada como função comportamento da instrução.

Como já comentado em capítulos anteriores, a simulação interpretada é lenta e ine-

ficiente, e a simulação rápida, com técnicas de simulação compilada, tomou a cena em trabalhos recentes. Note que na Figura 3.1, mesmo sendo interpretada, possui uma otimização que é característica da simulação compilada, (linhas 4 e 5): as instruções são decodificadas somente uma vez, pois os campos decodificados são armazenados em uma *cache*.

3.2 Simulação compilada estática em ArchC

A implementação de um gerador de simulador compilado estático para o ArchC, com duas novas técnicas denominadas FSCS, trouxe um ganho de desempenho excelente, superior a todos os outros trabalhos relacionados.

O conjunto de otimizações proposto na técnica de simulação compilada é baseado no fato da aplicação a ser simulada ser conhecida com antecedência, permitindo especializar o simulador com uma vasta gama de otimizações.

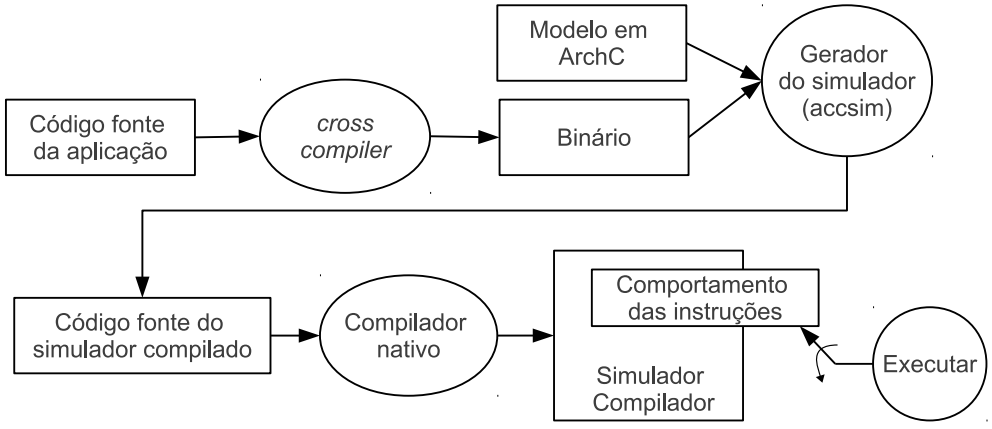

Figura 3.2: Fluxo do simulador compilado estático

A Figura 3.2 apresenta o fluxo seguido pelo simulador compilado estático. A aplicação é compilada utilizando um *cross-compiler* (GCC [18] neste trabalho) configurado para gerar um arquivo binário para a máquina alvo. Esse arquivo, juntamente com a descrição

do modelo em ArchC, serve de entrada para o gerador de simulador compilado *accsim*. As instruções do programa são decodificadas e armazenadas em uma estrutura de memória. Cada instrução decodificada corresponde a uma chamada para a função de comportamento na linguagem C++. A estrutura escolhida para armazenar essas instruções decodificadas é um `switch`, sendo indexado pelo `ac_pc` (*Program Counter*).

Na Figura 3.3 apresenta um exemplo de como três instruções MIPS I são decodificadas e transformadas para a linguagem intermediária C++. Dentro de cada `case`, há funções que correspondem à instrução do código *assembly*. A função `ac_bhv_instr` é o comportamento genérico das instruções, portanto, todas executam esta função. Como cada instrução pertence a um tipo e cada tipo possui comportamentos particulares, então a função `ac_bhv_TypeR` é responsável por executar os comportamentos associados. Por fim, é executado o comportamento específico da instrução. A implementação dessas funções, que fazem parte do modelo, é feita pelo projetista, codificando a ação de cada instrução. Na geração do simulador compilado, tem-se a opção de fazer *inlining* de todas essas funções, aumentando o desempenho significantemente. O `ac_pc` é atualizado dentro dessas funções, normalmente em `ac_bhv_instr`, e quando um desvio é decodificado, seu comportamento específico atribui ao `ac_pc` o valor correto.

```
0x4000:     lw $1, 0($8)
0x4004:     bnez $1, 4000
0x4008:     addi $2, $1, 4
```

```
void mips1::Region3(){
   while (...){
      switch (ac_pc){
      ...
         case 0x4000:
            ac_bhv_instr(...);
            ac_bhv_TypeI(...);
            ac_bhv_lw(r1, 0, r8, ...);
            ac_instr_counter++;
            break;
         case 0x4004:
            ac_bhv_instr(...);
            ac_bhv_TypeR(...);
            ac_bhv_bnez(r1, 0x4000, ...);
            ac_instr_counter++;
            break;
         case 0x4008:
            ac_bhv_instr(...);
            ac_bhv_TypeR(...);
            ac_bhv_addi(r2, r1, 4, ...);
            ac_instr_counter++;
            break;
      ...
      }
   }
}
```

Figura 3.3: Tradução de 3 instruções MIPS para código em C++

O número de entradas no `switch` pode ter impacto no tempo de compilação e no tempo de simulação. A configuração que ofereceu melhor resultado foi 512 instruções decodificadas em um único `switch`, chamado de região. Logo, são necessários vários `switchs`, um para cada região e um controle para poder saltar entre eles. O método que controla a simulação principal selecionará a região em que se encontra a próxima instrução a ser executada. Este método está presente na Figura 3.4.

```
1  void mips1::Execute(...){
2     while (!ac_stop_flag){
3        switch (ac_pc >> 9){
4           case 0: Region0(); break;
5           case 1: Region1(); break;
6           ...
7           case 14: Region14(); break;
8           default:
9              AC_ERROR(...);
10             ac_stop(EXIT_FAILURE);
11             break;
12       }
13    }
14 }
```

Figura 3.4: Rotina principal da simulação

Esta implementação é o simulador compilado básico, que ainda será otimizado na próxima seção. Mesmo assim, comparado com simuladores similares presentes na literatura, os resultados obtidos foram superiores. A técnica IS-CS [46], da ADL EXPRESSION, apresentou média de 12 milhões de instruções por segundo (MIPS) em um Athlon de 1GHz, contra 24 MIPS do simulador compilado básico do ArchC em máquina equivalente. Já a JIT-CCS [36], da ADL LISA, apresentou desempenho máximo de 30 MIPS com um Athlon 2.5GHz. Após as duas otimizações a seguir, o desempenho do simulador compilado estático do ArchC alcança, no mínimo, 160 MIPS em um Pentium de 2.4 GHz. Porém, este processador foi utilizado apenas para normalização. Os resultados apresentados a seguir foram gerados em um i7 com 2.8GHz.

3.3 Otimizações FSCS

Além da técnica básica da simulação compilada estática, o simulador da ADL ArchC 1.6 implementou duas novas otimizações, denominadas de FSCS (*Fast Static Compiled Simulation*) [4, 5]. Para fazer uso dessas técnicas, foram incluídas novas construções na

linguagem de descrição ArchC, fornecendo mais informações sobre o processador.

É um grande desafio ter um modelo que é eficiente na qualidade da descrição e ao mesmo tempo no desempenho do simulador gerado. A qualidade na descrição consiste capturar informações da arquitetura de uma maneira natural e concisa, sem sobrecarregar o projetista de informações irrelevantes. Por outro lado, gerar simuladores de alto desempenho requer o máximo possível de informações estáticas sobre a arquitetura e seu conjunto de instruções. Logo, modificar uma linguagem de descrição para aumentar o desempenho da simulação exige muita cautela. Assim, essas novas construções são obrigatórias apenas se o projetista desejar fazer uso das técnicas FSCS.

3.3.1 Primeira otimização

Reduzir a computação realizada em tempo de execução proporciona ganhos de desempenho. Uma das formas foi reduzir a frequência de avaliação do fluxo de execução do programa simulado. Assim, um bloco básico (um conjunto de instruções com apenas um ponto de entrada e um ponto de saída) pode ser simulado sem nenhuma verificação por mudanças no contador de programa (`ac_pc`).

Na seção anterior, foi comentado que as instruções decodificadas são transformadas em uma estrutura `switch`, onde cada `case` é uma instrução do programa. Logo, ao executar uma instrução, um `break` é encontrado, o `ac_pc` é atualizado para a nova instrução e entra no `switch` novamente. Se a simulação estiver no meio de um bloco básico, ou seja, a instrução atual não é de desvio, obviamente a próxima será executada. Então, ao invés de sair do `switch` com o `break`, pode-se fazer uma análise na fase de pré-processamento para retirar os `breaks`, deixando a execução continuar sem causar *overhead*. Para isso, na descrição do modelo, é necessário uma informação adicional sobre quais instruções causam desvio, e se é condicional ou incondicional.

Em casos onde há presença de *delay slot*, o fluxo de execução pode mudar somente após a execução da instrução no *delay slot*. Por esta razão, o número de *delay slots* também precisa ser fornecido na descrição do conjunto de instruções.

3.3.2 Segunda otimização

A segunda otimização da FSCS consiste em reduzir ainda mais o número de cálculos efetuados em tempo de execução, delegando à fase de pré-processamento o cálculo do maior fluxo de execução possível. Apenas cálculos dependentes dos dados de entrada são postergados para o tempo de execução. Esta abordagem é denominada Avaliação Parcial [25].

Esta otimização foi a mais ambiciosa, delegando a tarefa de controlar e manipular o Contador de Programa para o simulador. Para isso, a descrição das instruções de controle teve que ser ainda mais refinada do que foi para a primeira otimização, agora deixando explícito o modo de manipulação do Contador de Programa nas instruções de controle de fluxo. O custo de inserir novas construções e aumentar o tamanho da descrição ArchC é compensado pelos excelentes resultados obtidos.

Além das informações acrescentadas para a primeira otimização, mais quatro são necessárias: (1) a condição para o desvio no caso de saltos condicionais; (2) como instruções de salto calculam o seu alvo; (3) a condição para execução de *delay slots* (permitindo assim sua anulação); e (4) o comportamento adicional realizado pelas instruções de controle, se existir (por exemplo, instruções de chamada de rotinas devem salvar o endereço de retorno antes do salto).

Para mais esclarecimentos sobre FSCS, consultar os trabalhos de Bartholomeu *et al.* [4, 5].

3.4 Evoluções do accsim

Com o lançamento do ArchC 2.0 [53], ferramentas que existiam na versão 1.6 deixaram de funcionar. Os arquivos de descrições sofreram melhorias e um novo *parser* teve que ser implementado. Os simuladores interpretados gerados tiveram uma reorganização nas estruturas do código, melhorando o entendimento e facilitando a inserção dos processadores em plataformas com vários componentes, permitindo a utilização de TLM (*Transaction-Level Modeling*) e SystemC, por exemplo.

Porém, nesta nova versão, o gerador de simulador compilado estático, `accsim`, ficou sem suporte. Várias estruturas importantes para seu funcionamento deixaram de existir e as informações das descrições do modelo foram modificadas. Para oferecer esta abordagem na versão 2.1, foi necessário adaptar o código desta ferramenta para utilizar algumas estruturas presentes na versão 2.0 e implementar outras que foram descontinuadas. Além disso, várias funções tiveram que ser criadas ou reimplementadas para que o simulador compilado gerado fosse compatível com a recente versão do ArchC. A biblioteca SystemC passou a ser utilizada pelos simuladores gerados pelo `accsim` 2.1, o que necessitou de cuidados para que isso não impactasse no desempenho.

Durante o processo de adaptação, estudou-se qual seria o melhor caminho e, utilizar várias classes com muitas heranças e auto-referencias, como é a proposta do simulador interpretado da versão 2.0, não é uma boa alternativa quando o objetivo é desempenho. Porém, criar um objeto que encapsulasse tudo que fosse particular a um processador é importante pois possibilita a simulação de vários núcleos homogêneos ou heterogêneos. Por isso, decidiu-se criar uma única classe, pois, apesar de não ser boa prática de programação, permite que várias otimizações sejam feitas pelo compilador. No entanto, ao se encapsular alguns recursos que eram globais, o desempenho chegava a cair 60% e, por isso, foram mantidos como globais *a priori*. Na próxima seção, têm-se uma alternativa

para resolver esta queda de desempenho.

Outro problema encontrado foi com as versões utilizadas do GCC. No ArchC 1.6, a compilação do simulador gerado era feito com o gcc-3.3. Ao gerar o simulador no ArchC 2.1, a compilação passou a ser feita pelo gcc-4.4. Com o gcc-3.3, conseguia-se 270 MIPS (Milhões de Instruções por Segundo) com o aplicativo `sha` e o tempo de compilação do simulador demorava 30 segundos. Ao utilizar o gcc-4.4, um pouco mais de 270 MIPS era atingido, mas a compilação demorava 33 minutos. A Tabela 3.1 exibe estes números, onde a primeira coluna informa qual foi o gerador de simulador utilizado e a segunda coluna diz a versão do GCC que compilou o simulador gerado.

Versão accsim	**Compilador**	**Tempo de compilação**	**Desempenho**
accsim-1.6	gcc-3.3	30seg	270 MIPS
accsim-1.6	gcc-4.4	33min	80 MIPS
accsim-2.1	gcc-3.3	30seg	270 MIPS
accsim-2.1	gcc-4.4	33min	300 MIPS

Tabela 3.1: GCC *vs* Tempo de Compilação

Após relatar o problema para a equipe de desenvolvimento do GCC, passamos a utilizar o último *branch* do gcc-4.4, conseguindo 39 segundos no tempo de compilação e com o desempenho chegando a 618 MIPS. Este ganho extra de 128% de desempenho se deve, principalmente, à otimização `inline-small-functions`, que foi implementada nos últimos *branches* do gcc-4.4 e está presente tanto em `-O2` quanto em `-O3`. Esta otimização é baseada em heurísticas que decidem se uma função é simples o suficiente para ser transformada em *inline*.

3.5 Otimizações no accsim 2.1

Uma vez que o simulador compilado foi devidamente testado com várias descrições de arquiteturas, algumas melhorias foram implementadas para que mais desempenho fosse

alcançado.

As duas otimizações da técnica FSCS, já comentadas, foram adaptadas para funcionar nesta nova versão, que é orientada a objeto. Porém, os recursos utilizados pelo processador eram variáveis globais e, ao encapsulá-los em uma classe, o desempenho diminuiu 60%.

O principal responsável é o contador do número de instruções (`ac_instr_counter`), que era incrementado milhares de vezes durante a simulação, um por `case`. Uma alternativa foi eliminar esses incrementos excessivos, salvando o alvo do último desvio e, ao encontrar um outro desvio, fazer a diferença. Com isso, consegue-se contar o número de instruções executadas entre os desvios.

Esta solução permitiu deixar todos os recursos encapsulados sem grande perda de desempenho (na ordem de 7%), preparando a ferramenta para que, futuramente, seja possível instanciar vários processadores, homogêneos ou não, em plataformas SystemC. Como esse suporte a multiprocessadores ainda não está implementado na abordagem compilada estática, a opção padrão deixa todos os recursos como globais. Se o projetista desejar gerar um simulador compilado estático com os recursos encapsulados, basta utilizar a opção `--wrapper` quando executar o accsim.

3.6 Resultados

Com o objetivo de avaliar o simulador compilado estático gerado pela ADL ArchC 2.1, selecionou-se três modelos: MIPS I [26], SPARC V8 [37] e PowerPC [15]. Os aplicativos utilizados são dos *benchmarks* Mediabench [32] e Mibench [33], abordando vários domínios de aplicação. Estes programas não sofrem modificações no código durante a simulação, requisito necessário para o simulador compilado estático. Os aplicativos conseguem ler e escrever em arquivos do disco na máquina hospedeira utilizando emulações de chamadas de sistemas [6].

Todos os experimentos foram executados em um Core i7 860, cuja frequência é de 2.80GHz com 4Gb de memória RAM. O sistema operacional utilizado foi o Linux Ubuntu 10.04 32bits e para a compilação dos simuladores utilizou-se o gcc-4.4.

Nas figuras que exibem o desempenho das simulações, o simulador base não possui nenhuma otimização, o `opt1` simboliza a primeira otimização e o `opt2` a segunda otimização do FSCS. A otimização de postergar o incremento do contador de instruções para um desvio é ativado também pela `opt2`. Em todos os casos utilizou-se a opção `-O3` do GCC em conjunto com a *flag* `inline`, para aumentar o desempenho da simulação.

Para cada aplicativo, seis execuções redundantes foram conduzidas, coletando-se o tempo gasto na compilação do simulador compilado estático, o tempo de simulação e o número total de instruções executadas. Com essas seis execuções, calculou-se o desvio padrão referente ao número de instruções executadas por segundo (MIPS) para a segunda otimização, que deu origem ao intervalo de confiança presente nos gráficos a seguir. O grau de confiança utilizado foi de 95%.

A Figura 3.5 mostra o resultado dos *benchmarks* para o modelo MIPS I. A segunda otimização aliada às otimizações do GCC renderam ótimos resultados, alcançando *speedup* de 221% (para `patricia`) até 626% (para `rijndael enc`) em comparação com o base. A marca de 1 bilhão de instruções por segundo (BIPS) para o programa `sha` quase foi alcançada, sendo o aplicativo que apresentou o melhor desempenho. A média do desempenho ficou em 642 MIPS (milhões de instruções por segundo).

Aplicativos que consumiram mais tempo na simulação apresentaram um comportamento mais estável, com um menor desvio padrão, e isto é refletido no intervalo de confiança. Isso acontece porque aplicativos simples, com poucas instruções, alcançam um alto desempenho na simulação compilada estática, e centésimos de segundo causam um grande impacto na unidade de medida MIPS (milhões de instruções por segundo). O

tempo de simulação do aplicativo `sha` com `opt2` variou entre 0.13s a 0.18s e obteve um desvio padrão de 53 milhões de instruções por segundo. Já o aplicativo `lame`, com `opt2`, teve seu tempo de simulação variando entre 212.19s a 216.24s, com um desvio padrão de apenas 3 milhões de instruções por segundo.

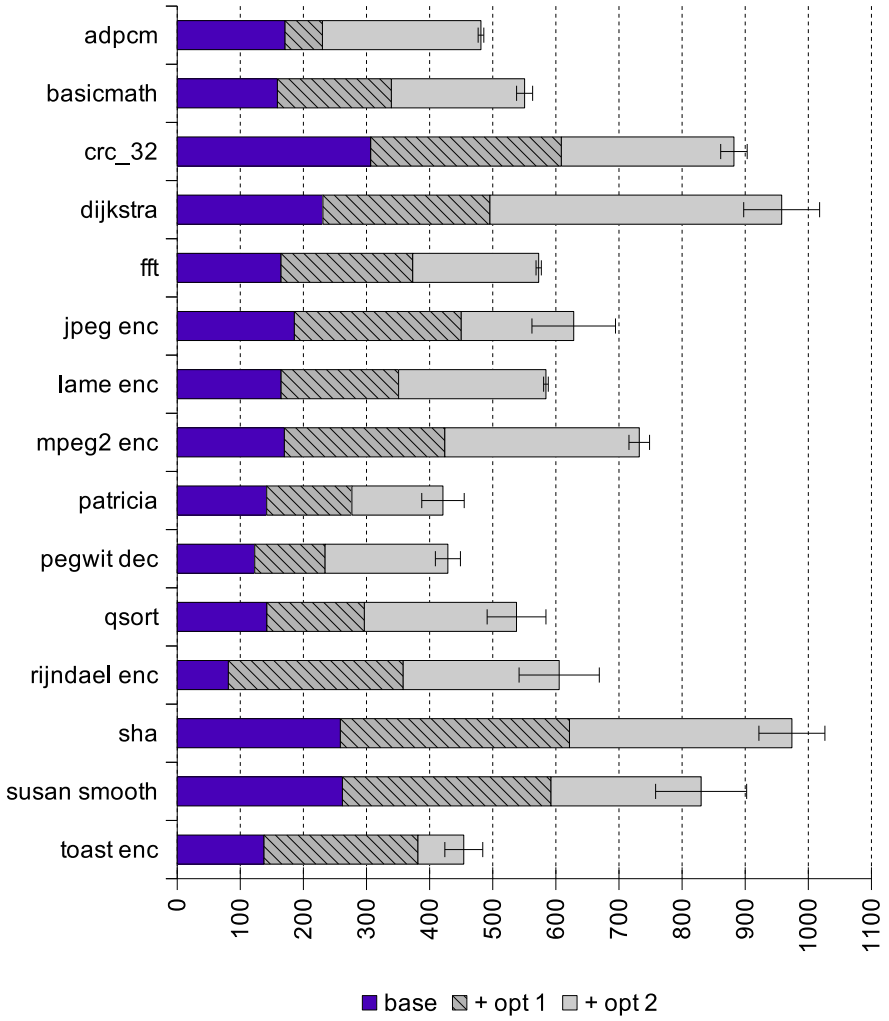

Figura 3.5: Desempenho do MIPS I em milhões de instr. por seg. (mips)

Figura 3.6 mostra o resultado para o modelo SPARC V8. A média das execuções ficou em 423 MIPS, sendo o `sha` e o `crc_32` os aplicativos com melhores desempenhos, 862

e 699 MIPS, respectivamente. A queda de desempenho, em comparação ao modelo do processador MIPS, foi devido à complexidade do modelo SPARC V8.

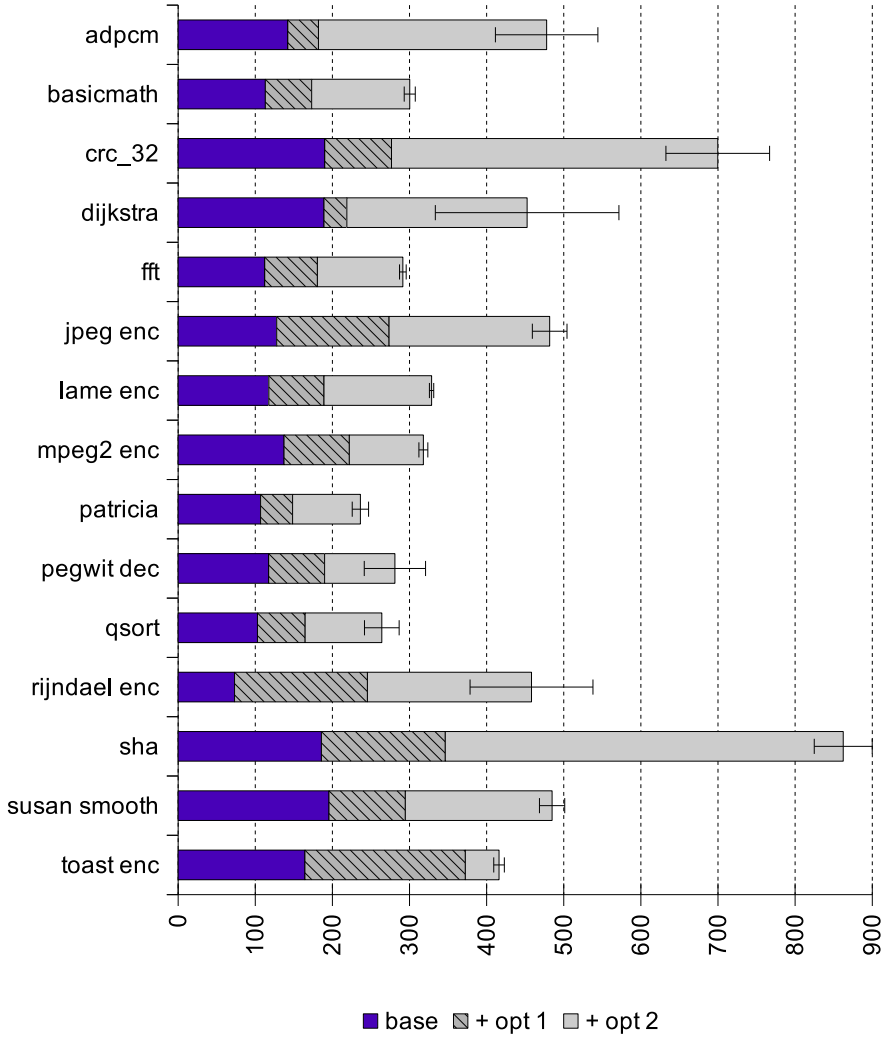

Figura 3.6: Desempenho do SPARC V8 em Milhões de Instr. por Seg. (MIPS)

Os testes executados com o PowerPC, exibidos na Figura 3.7, apresentaram uma média levemente superior ao SPARC V8, 438 MIPS. PowerPC possui uma descrição complexa, tanto quanto a SPARC V8, e por isso sua média também se manteve abaixo da descrição MIPS I.

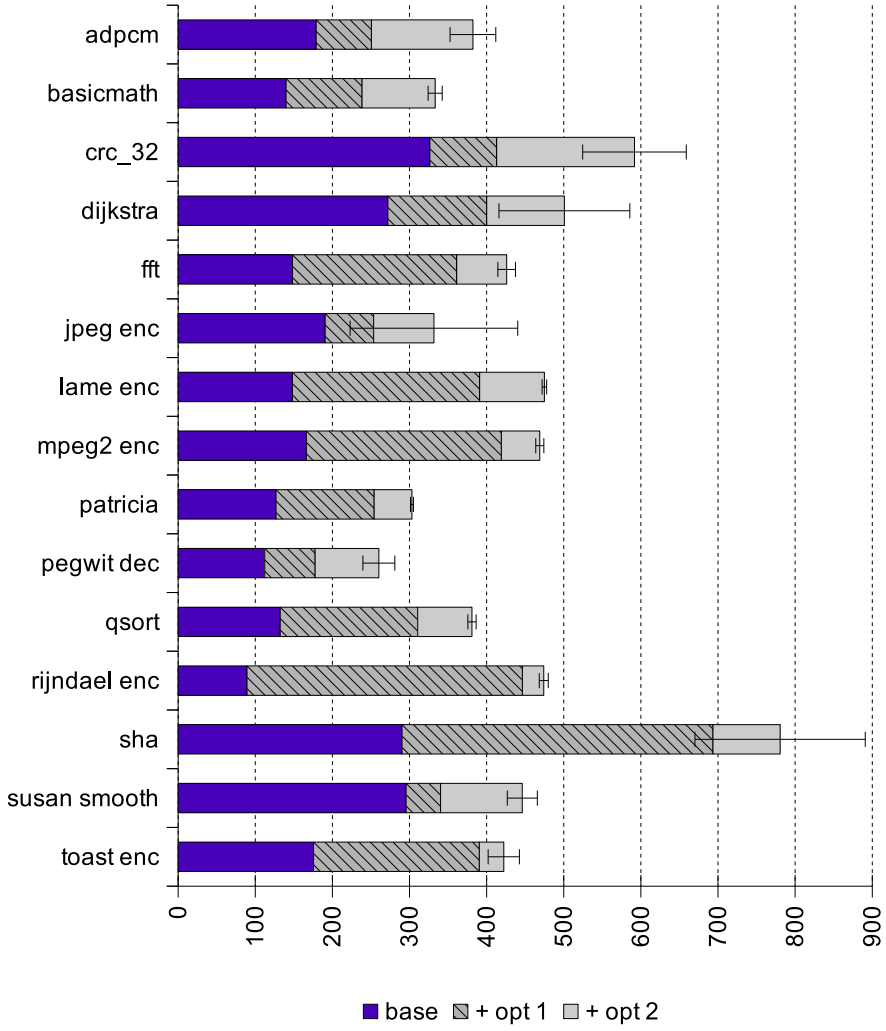

Figura 3.7: Desempenho do PowerPC em Milhões de Instr. por Seg. (MIPS)

Além da otimização `opt2` aumentar o desempenho do simulador, ela gera códigos mais simples que o código do simulador com `opt1`, deixando a compilação mais rápida, como mostra a Figura 3.8. Logo, o `opt1` foi inserido nos experimentos apenas para comparação, posto que sua escolha é inviável. Já o simulador `base` teve um tempo de compilação próximo ao alcançado pelo simulador gerado com `opt2`. Porém, sua escolha também é inviabilizada pelo baixo desempenho da simulação. A otimização `opt2`, conclusivamente,

é a melhor escolha. A Figura 3.8 compara o tempo de compilação dos simuladores do modelo MIPS I sem otimização (modo `base`), com `opt1` e com `opt2`.

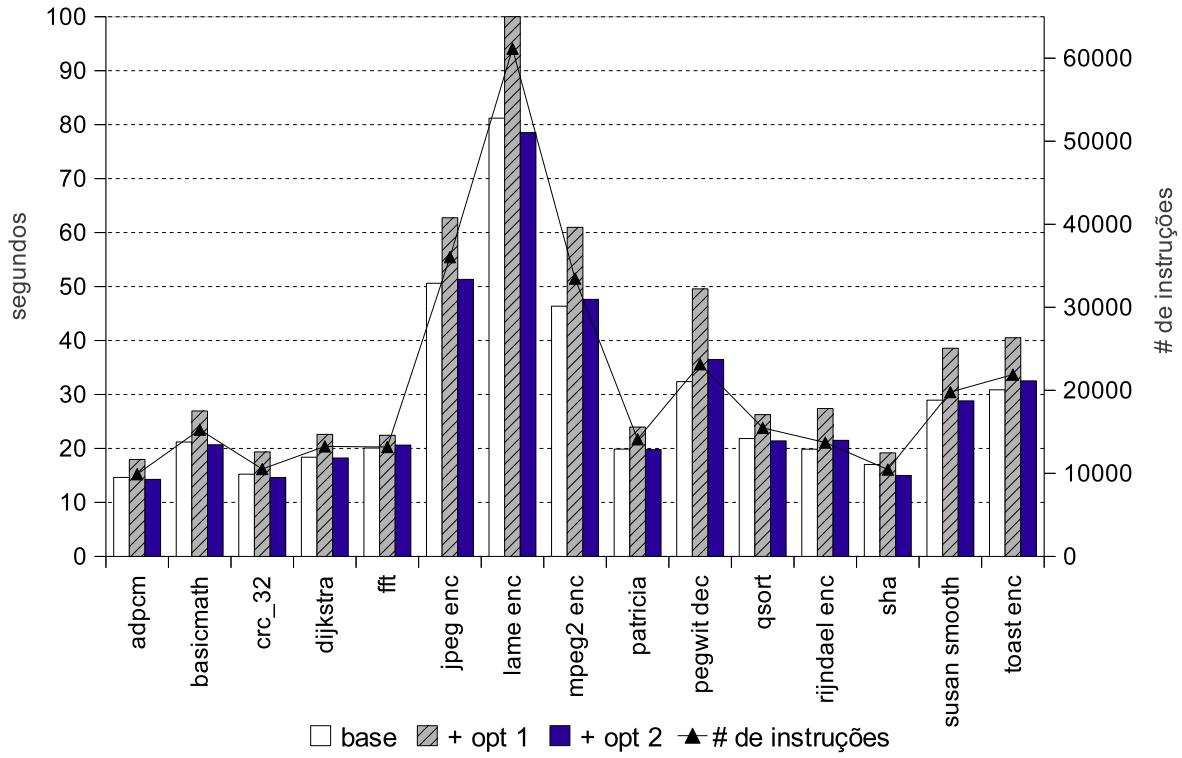

Figura 3.8: Tempo de compilação dos simuladores MIPS I

A linha no gráfico mostra o número de instruções que um determinado aplicativo possui, orientada pelo eixo Y à direita. Percebe-se que o tempo de compilação cresce linearmente com o número de instruções do aplicativo a ser executado. Nos experimentos deste trabalho, o aplicativo `lame` foi o mais demorado a compilar, chegando a 78s com a `opt2` ligado no MIPS I, 107s no PowerPC e 127s no SPARC V8. Obviamente, de acordo com a arquitetura modelada, o código gerado pode ser mais complexo que outros, impactando no tempo de compilação. A Figura 3.9 faz um comparativo do tempo de compilação com simuladores dos três modelos, utilizando apenas a segunda otimização, que é a mais agressiva. Os tempos de compilação do modo `base` e do `opt1` para os

outros modelos apresentam o mesmo comportamento do modelo MIPS I e, portanto, foram omitidos nesta figura. A diferença de complexidade das arquiteturas fica claro neste gráfico, sendo a PowerPC e a SPARC V8 muito mais custosas para se compilar.

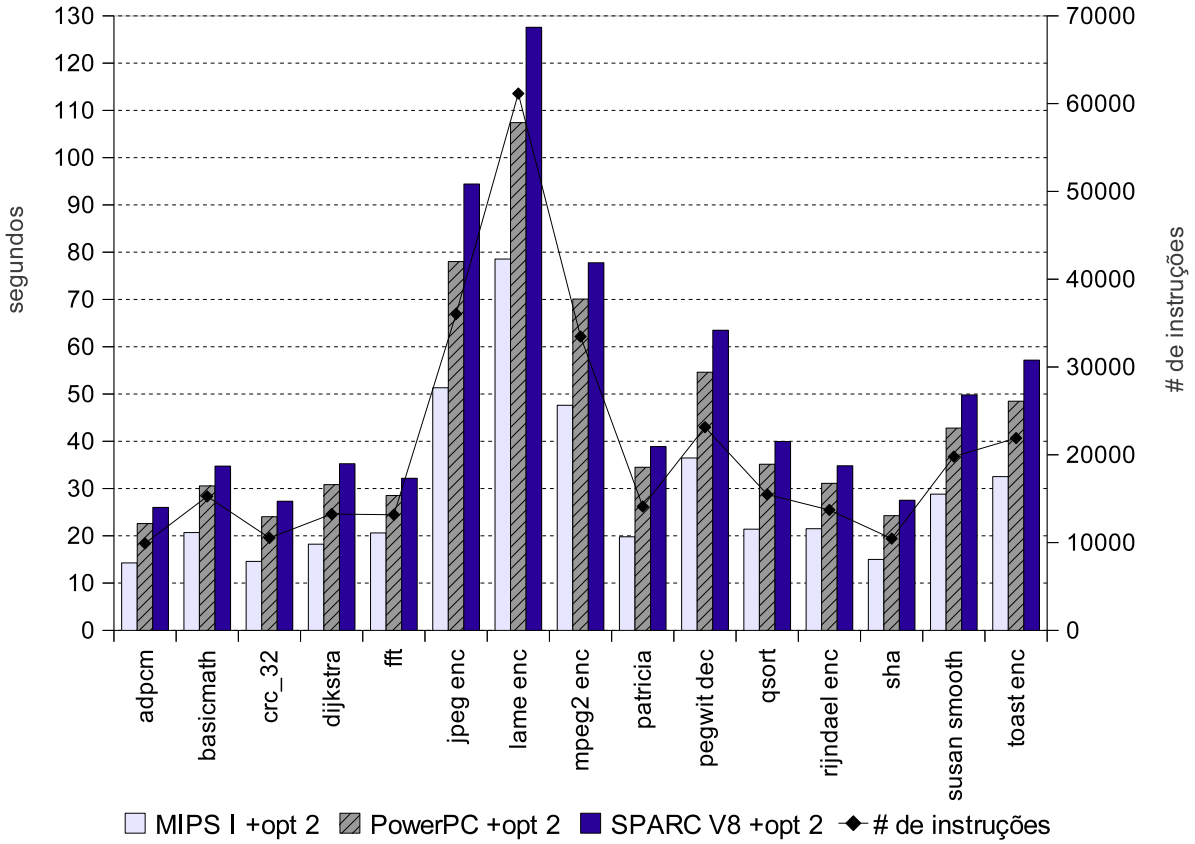

Figura 3.9: Tempo de compilação dos simuladores com `+opt2`

Para mostrar a relação entre o tempo gasto compilando contra o tempo gasto simulando, o gráfico da Figura 3.10 utiliza o modelo MIPS I como exemplo, porém os outros dois modelos apresentaram o mesmo comportamento. A otimização utilizada foi a `opt2`.

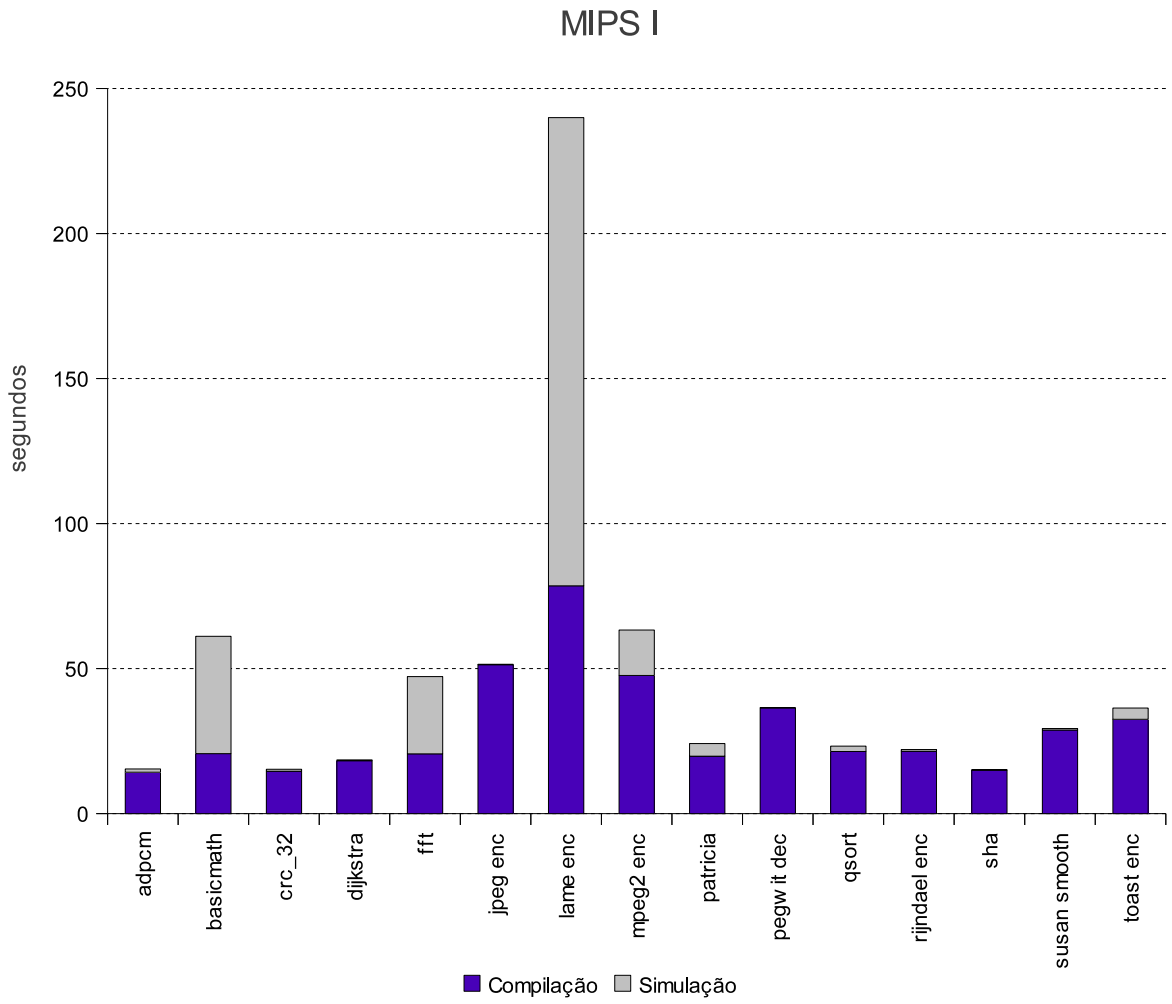

Figura 3.10: Tempo de compilação e de simulação para MIPS I com `+opt2`

Ao normalizar o tempo total (compilação + simulação) de todos os aplicativos, deixando o eixo Y do gráfico da Figura 3.11 como porcentagem de tempo gasto, percebe-se que para a maioria dos aplicativos, menos de 10% do tempo é gasto na simulação. Os quatro aplicativos que mais consumiram tempo de simulação, `lame enc`, `basicmath`, `fft` e `mpeg2 enc` fazem uso de ponto flutuante.

Os modelos dos processadores utilizados neste trabalho não tem instruções de ponto flutuante e, para compilar os aplicativos para a máquina alvo, foi necessário que a

opção `-msoft-float` fosse habilitada no compilador *cross-compiler*. Isto faz com que as operações em ponto flutuante sejam executadas em software, ao invés de hardware, diminuindo o desempenho na simulação. Se estas instruções fizessem parte do modelo, o tempo de simulação diminuiria drasticamente, atingindo desempenho similar aos outros aplicativos.

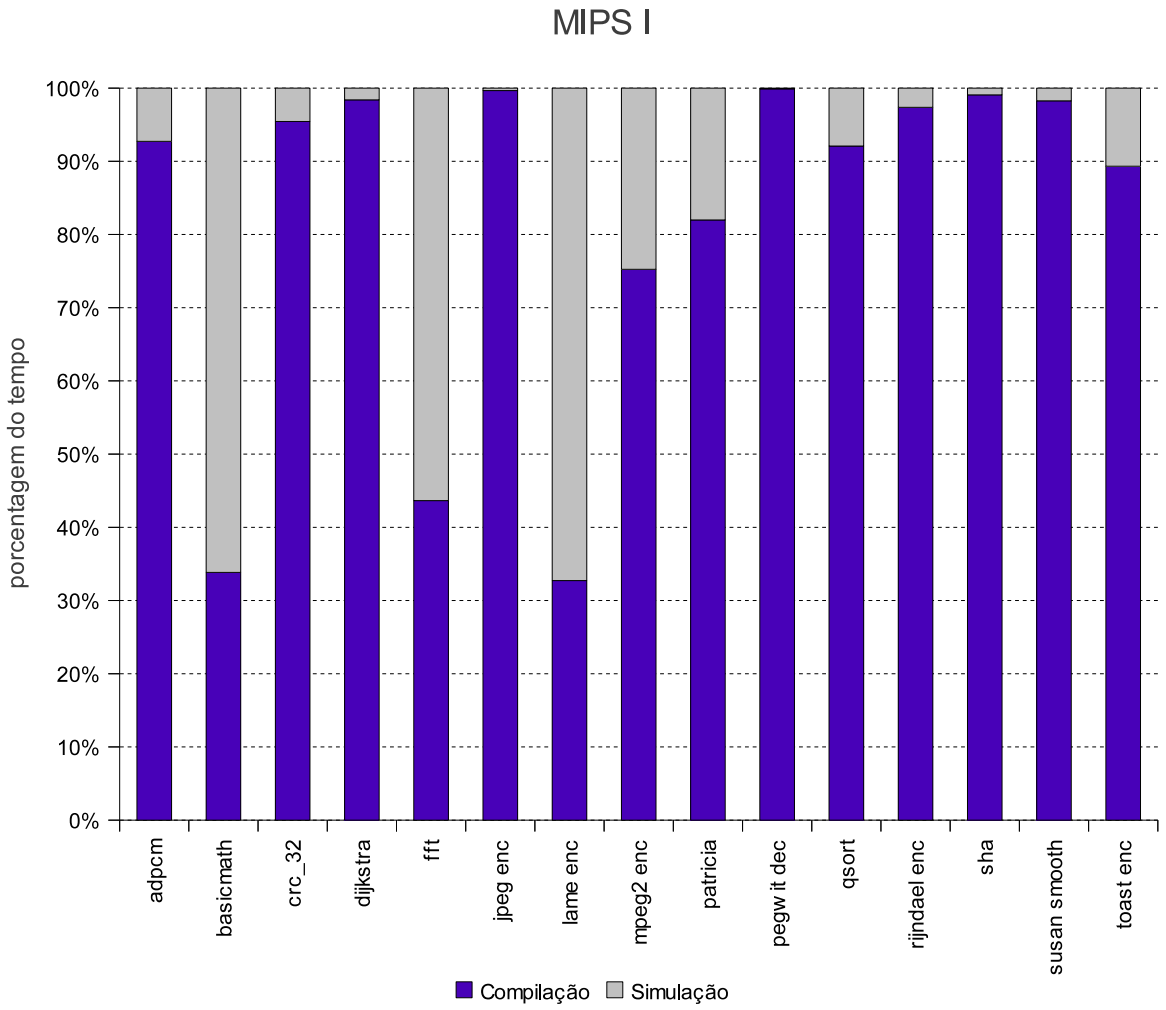

Figura 3.11: Porcentagem do tempo de compilação e de simulação para MIPS I com `+opt2`

Testes foram realizados comparando o desempenho dos simuladores compilados em máquina Intel com os aplicativos compilados nativamente para Intel. A Figura 3.12 mostra

quantas vezes a simulação é mais lenta comparada com a execução nativa. Experimentos feitos em 2004, no `accsim` do ArchC 1.6 [4], mostram que o fator de lentidão do `adpcm` era de 14,1x para o modelo SPARC V8, que foi o modelo com melhor desempenho exposto no trabalho. No `accsim` atual, o `adpcm` possui o fator de lentidão de 3,6x comparando com o modelo com melhor desempenho. Esta comparação é válida, pois no computador simulado em 2004 [5], o tempo de execução do código nativo foi de 0.90s contra os 0.51s atual, *speedup* de 70%. No entanto, o tempo de simulação em 2004 [5], foi de 12.69s contra os 1.86s atual, *speedup* de 582%.

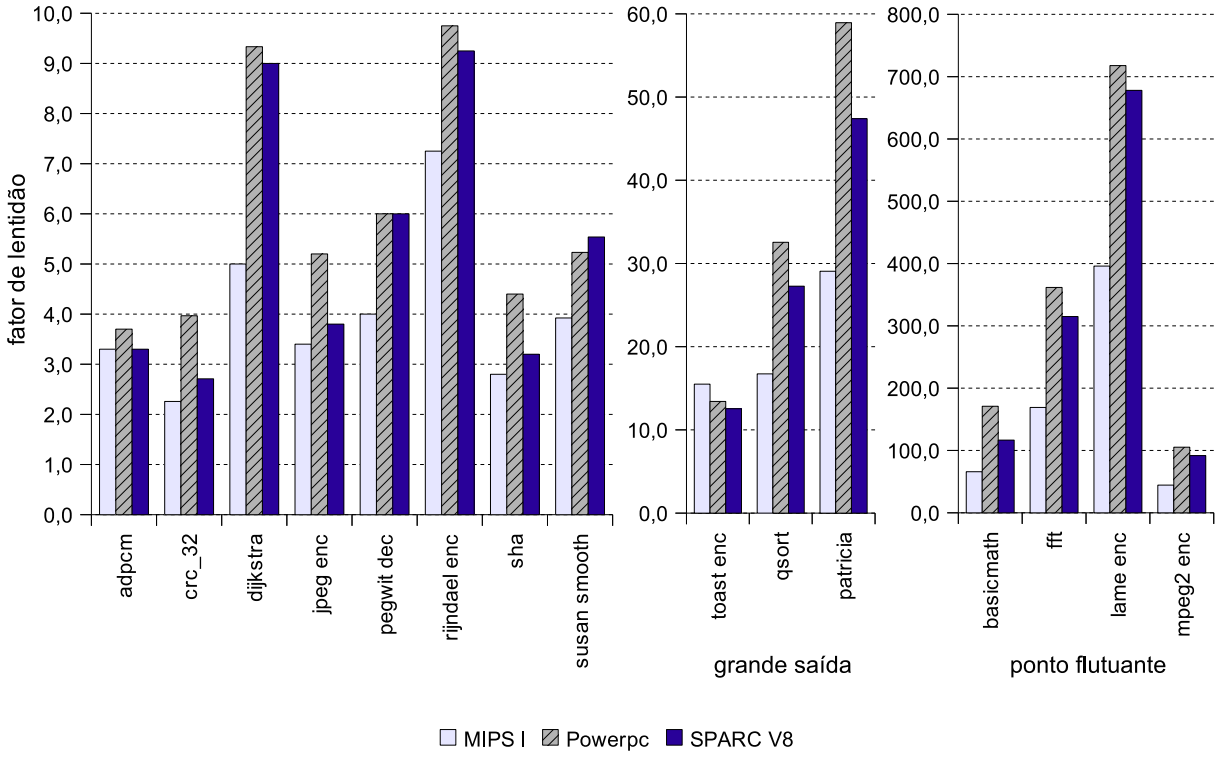

Figura 3.12: Fator de lentidão relativo à execução nativa Intel

Há três grupos de dados na Figura 3.12: os aplicativos que atingiram alto desempenho de simulação e ficaram próximo à execução nativa, o segundo grupo que contém aplicativos que tiveram seu desempenho afetado devido a grandes escritas de saída e o terceiro grupo,

que são aplicativos com alta carga de operações em ponto flutuante. Como já comentado, instruções de ponto flutuante não fazem parte dos modelos utilizados neste trabalho. Por isso, a maioria dos aplicativos deste grupo foram, no mínimo, 100x mais lentos que a execução nativa.

A média do fator de lentidão foi de 51x para o modelo MIPS I, 88x para o SPARC V8 e 100x para o PowerPC. Porém, ao analisar apenas o primeiro grupo, a média melhora drasticamente, alcançando 4x para MIPS I, 5.3x para SPARC V8 e 5.9x para PowerPC.

O trabalho apresentado nesta seção fez com que o ArchC 2.1 voltasse a oferecer um simulador rápido, com um desempenho superior aos trabalhos correlatos. Essa tendência é importante para que arquiteturas mais complexas possam ser simuladas dentro do curto espaço de tempo disponível para o projeto.

Capítulo 4

Simulação Compilada Dinâmica

Com o aumento da complexidade das novas arquiteturas, simuladores puramente interpretados tornaram-se lentos e variadas técnicas foram propostas para aumentar o desempenho. Uma melhoria foi inserir uma *cache* de instruções já decodificadas nos simuladores interpretados, melhorando a execução de códigos repetitivos. Outro avanço foi permitir que os simuladores realizassem um pré-processamento do aplicativo a ser executado, armazenando o resultado das operações para uso durante a simulação. Esta abordagem é a base do que se chamou de **simulação compilada**. Quando o aplicativo é conhecido antes da geração do simulador, uma opção é realizar o pré-processamento completo do binário e gerar um simulador especializado para sua execução. Essa técnica é conhecida como simulação compilada **estática** e como o ArchC a implementa está descrito no Capítulo 3. Caso o projetista não queira especializar um simulador para cada aplicativo mas deseja, ainda, fazer uso da simulação compilada devido ao desempenho, o simulador pode dividir o aplicativo em vários blocos e, durante a execução, reservar um tempo para processar previamente o bloco que será simulado. Isso melhora o desempenho daquele trecho, quando comparado ao simulador interpretado e oferece maior flexibilidade que o simulador compilado estático. Essa alternativa é chamada de simulação compilada **dinâmica** e

sua implementação no ArchC será abordada neste capítulo.

4.1 A infraestrutura LLVM

As aplicações modernas necessitam de tecnologias de compilação que excedam o modelo tradicional do compilador estático, que realize análises e otimizações do código fonte original e simplesmente gere o executável final. É preciso ter ferramentas que permitam a otimização de um programa ao longo do seu ciclo de vida, realizando transformações dinâmicas durante a execução, além de colher informações para melhorar o desempenho em execuções futuras. Neste cenário encontra-se a infraestrutura LLVM (*Low Level Virtual Machine*) [29], que contém um conjunto completo de componentes reutilizáveis para implementação de compiladores e de ferramentas para análise e otimização de código executável. Inicialmente, seu objetivo era substituir o *back-end* do GCC por um mais moderno, porém seu sucesso influenciou a construção de novos *front-ends* de variadas linguagens. Iniciou-se como um projeto de pesquisa na Universidade de Illinois e hoje em dia conta com um número bastante significativo de contribuintes internacionais e de projetos vinculados, sendo um destaque em termos de pesquisa acadêmica na área de compiladores e uma referência entre as ferramentas de compilação tanto de código aberto como proprietárias.

Os componentes do LLVM participam de todos os passos de análise de código ao longo do processo de desenvolvimento, provendo ferramentas de código aberto que podem ser utilizadas e estendidas livremente. Além de possibilitar a construção de compiladores estáticos, o projeto dá suporte a ferramentas de perfilamento em tempo de execução, máquinas virtuais, ferramentas de análise estática, geração de código e otimizações em tempo de execução (conhecido como *Just-in-Time* - JIT), dentre outras opções. Resumidamente, os usuários finais podem ver o LLVM de três formas: (i) um compilador

estático, que utiliza o GCC [18] ou Clang [13] como *front-end* com recursos de otimização configuráveis e *profiling*; (ii) uma ferramenta que executa programas em formato *bitcode*, vindo do *front-end*, fazendo uso do compilador JIT, quando possível, ou por meio de interpretação; ou (iii) uma infraestrutura para construção de compiladores e ferramentas de análise de código, permitindo o desenvolvimento de novos recursos.

Os usuários em (i) e (ii) utilizam o LLVM como ferramenta final, sem necessidade de entender códigos internos do projeto. Caso o usuário se enquadre em (iii), é necessário um estudo dos módulos internos do LLVM para que as modificações resultem em novos recursos. Outra abordagem para usuário em (iii) é utilizar o LLVM como uma API (*Application Programming Interface*) e implementar os novos recursos e ferramentas como projetos separados. Se possível, é aconselhável esta segunda abordagem por ser menos intrusiva, mantendo estáveis as estruturas de dados e os módulos do LLVM, utilizadas por vários projetos. Mesmo ferramentas internas e que fazem parte da API do LLVM são implementadas neste molde.

O recurso do LLVM citado, em que trechos de códigos em representação intermediária são compilados e otimizados em tempo de execução (JIT), é de grande valia para a implementação de um simulador compilado dinâmico, como veremos no decorrer deste capítulo. Porém, não se pode confundir usuários que utilizam o LLVM como compilador JIT com usuários que queiram utilizar o recurso JIT internamente em seus projetos. No primeiro caso, todo o arcabouço do LLVM é utilizado, desde o início da compilação (*front-end*) até a execução do programa, não necessitando de nenhum conhecimento extra do usuário para com o código interno do LLVM. No segundo caso, apenas módulos do LLVM responsáveis pelo JIT serão invocados dentro do projeto do desenvolvedor, por meio de APIs. Logo, bibliotecas do LLVM serão ligadas ao projeto do usuário, que deve ter domínio dos códigos envolvidos. É neste segundo cenário que se encontra o

desenvolvimento do simulador compilado dinâmico do ArchC.

4.1.1 A Arquitetura LLVM

O processo de compilação de código com ferramentas do projeto LLVM é representado pela Figura 4.1 [29]. Neste cenário, todo o arcabouço LLVM é utilizado, desde o *front-end* até a execução do programa, seja da maneira convencional ou por meio de JIT.

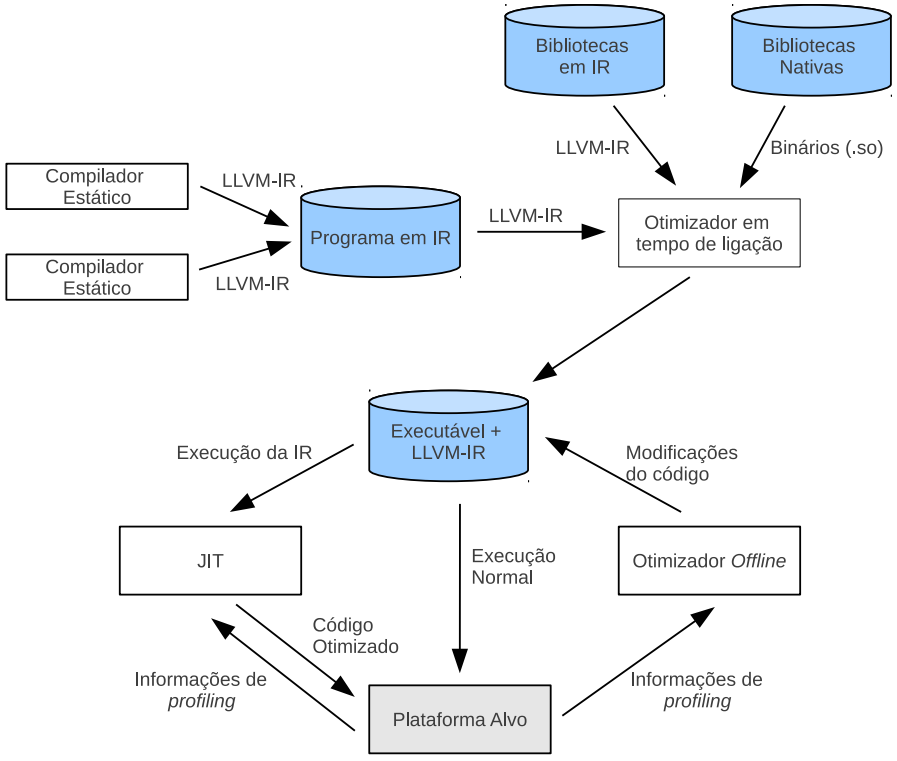

Figura 4.1: Arquitetura da infraestrutura LLVM

O usuário desenvolve o programa em uma linguagem de alto nível e depois envia para um *front-end* de sua preferência, desde que este seja capaz de construir a representação intermediária (IR) do LLVM, chamada de LLVM-IR. Inicialmente, o projeto LLVM tinha como objetivo substituir o *back-end* do GCC por um mais moderno e que suprisse as necessidades atuais de compilação. Logo, o *front-end* GCC foi o primeiro a ser utilizado

com geração de código em representação intermediária LLVM-IR. Porém, o sucesso do projeto resultou em iniciativas para desvincular o GCC do LLVM, surgindo outros *front-ends* de variadas linguagens, como o **Clang** [13].

Em outro passo, um otimizador em tempo de ligação pode unir o código desenvolvido com bibliotecas LLVM-IR ou em código nativo, descobrindo as dependências entre as unidades para obter mais informações para guiar as otimizações. O executável gerado, junto com a IR equivalente, é produzido. Com este código, existem duas opções de execução [29]. É possível executar o código binário normalmente, e utilizar as informações de uma execução em particular para guiar um otimizador em tempo de compilação (*offline*) capaz de melhorar a eficiência do código original. A outra opção é executar diretamente a IR do programa por meio de uma máquina virtual, que compila e otimiza trechos da IR para código nativo, em tempo de execução. Esta abordagem é denominada de JIT – *Just-in-Time*. A máquina virtual pode ser vista como um simples interpretador, caso o recurso JIT não esteja disponível (normalmente por falta de suporte à arquitetura).

4.2 Simulação compilada dinâmica em ArchC

Com a implementação da técnica de simulação compilada **estática** no ArchC, apresentada no capítulo anterior, notou-se uma notável melhora de desempenho quando comparado com o simulador interpretado. Porém, devido à necessidade de decodificar todo o binário para especializar o simulador a essa determinada aplicação, o projeto ArchC necessitava de uma segunda ferramenta de simulação rápida, que gerasse um simulador genérico por arquitetura, postergando a decodificação do binário para o tempo de execução. Assim, esta seção apresenta a técnica de simulação compilada **dinâmica** que foi implementada no ArchC. Apesar da nova abordagem, o algoritmo proposto possui raízes no algoritmo do cenário estático, mostrado no capítulo anterior.

A nova ferramenta de geração de simulador compilado dinâmico foi denominada `acdcsim` e seu fluxo está apresentado na Figura 4.2. A título de exemplificação, foi tomado como entrada o modelo ArchC referente à arquitetura **mips1**, porém, o fluxo é o mesmo para outros modelos, alterando-se apenas o nome dos arquivos.

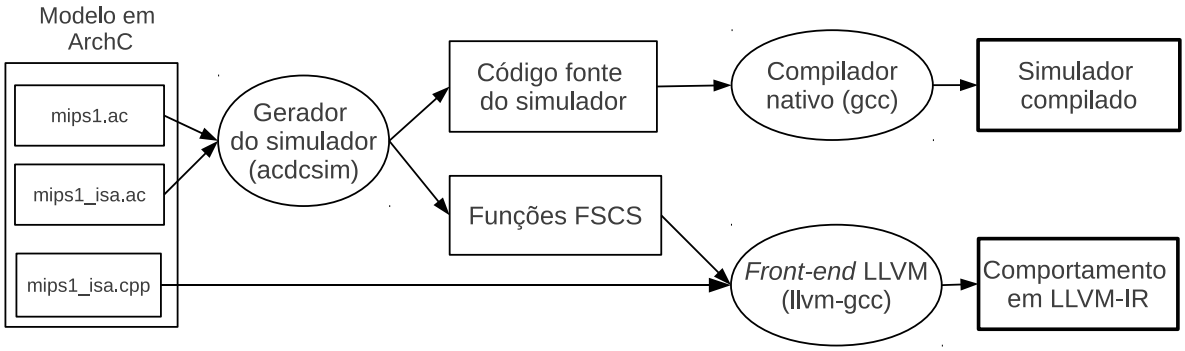

Figura 4.2: Fluxo da geração do simulador compilado dinâmico

A entrada do simulador é a descrição do modelo ArchC, que contém informações sobre os recursos do processador (`mips1.ac` e `mips1_isa.ac`) juntamente com código do comportamento de cada instrução (`mips1_isa.cpp`). O gerador realiza o processamento dessas informações e gera dois tipos de saída: o código fonte do simulador do modelo alvo, em C++, e o código das otimizações FSCS de cada instrução de desvio, separadas em funções, também em C++. Com o código do simulador em C++, um compilador nativo é invocado para gerar o executável do simulador. As funções FSCS, juntamente com o comportamento das instruções, passa por um *front-end* LLVM (neste caso `llvm-gcc`), resultando em um arquivo em linguagem intermediária LLVM-IR. Cabe reforçar que o código fonte do comportamento das instruções (`mips1_isa.cpp`) não é gerado pelo `acdcsim`, mas sim pelo projetista, na descrição ArchC. O código das otimizações FSCS também faz parte da descrição do modelo, como acontece no simulador compilado estático (Capítulo 3), e o `acdcsim` estrutura essas otimizações em funções para serem utilizadas *a posteriori*, como biblioteca LLVM-IR. A Seção 4.4 traz detalhes de como as otimizações FSCS foram

implementadas na abordagem dinâmica.

Com o simulador pronto para o determinado modelo , este recebe o aplicativo a ser simulado, que advém de um *cross-compiler* configurado para gerar binário para esta arquitetura. Além do binário, o simulador também espera o comportamento das instruções em LLVM-IR como entrada. A Figura 4.3 mostra o fluxo de execução de um binário na abordagem dinâmica em ArchC.

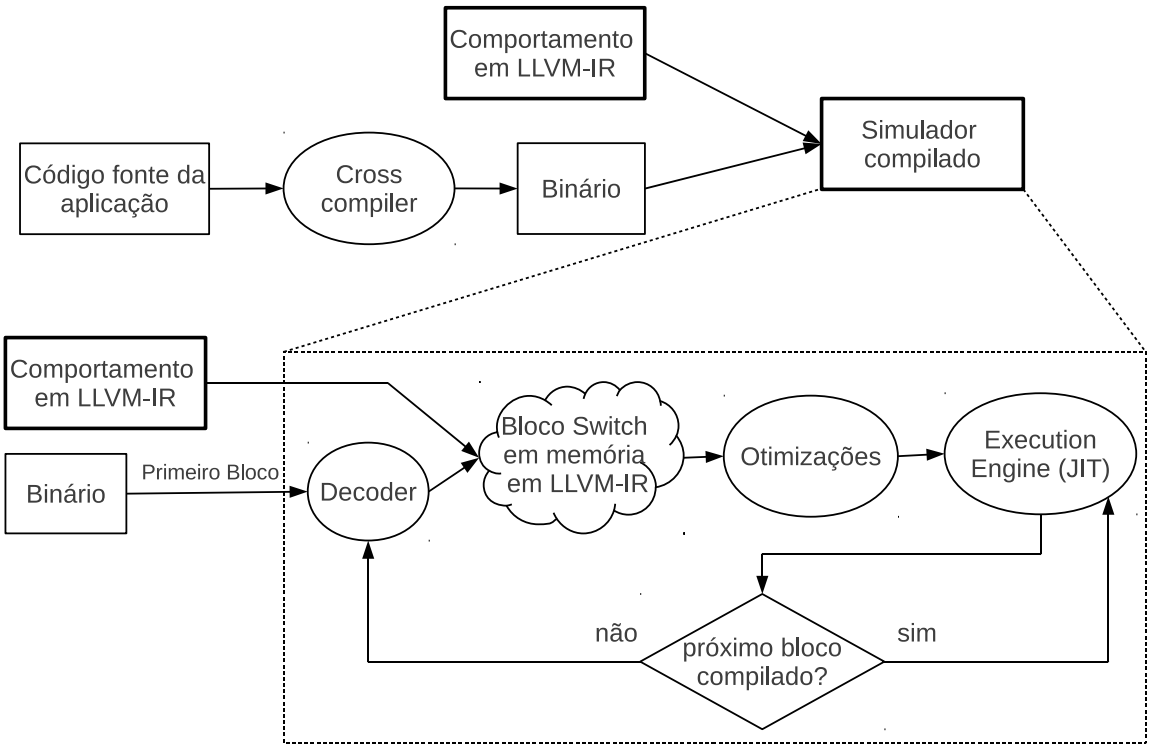

Figura 4.3: Fluxo de execução do simulador compilado dinâmico

A primeira diferença em relação à abordagem estática é a decodificação do binário em tempo de execução. Porém, diferente do que acontece na técnica interpretada, em que cada instrução é decodificada e executada, aqui um bloco de instruções é analisado e decodificado para depois enviar este bloco para execução. Cada instrução decodificada corresponde a uma chamada de função comportamento, semelhante ao que acontece na abordagem estática. Estas funções podem ser acessadas aleatoriamente devido aos saltos,

e a construção `switch` da linguagem C++ foi escolhida para indexar estas funções de comportamento. Cada instrução recebe um rótulo `case` e a seleção é feita por meio do contador de programa (`ac_pc`).

No capítulo anterior foi explicado que a construção `switch`, contendo as instruções decodificadas, era dividida em regiões para melhorar o desempenho da simulação. Nesta parte do trabalho, esta ideia continua sendo utilizada, porém, cada região, também chamada de bloco de código, é compilada e executada conforme a necessidade. Como mostra a Figura 4.3, após as instruções do bloco serem decodificadas, o `switch` é montado em memória e todas as chamadas de funções comportamento sofrem *inlining*, deixando o `switch` pronto para passar por um otimizador. O último passo é a *engine* de execução, responsável por executar o bloco montado e compilado. Ao encerrar a execução, o novo `ac_pc` indicará qual será o próximo bloco a ser executado. Se ele já está compilado, o simulador simplesmente chama a *engine* de execução para este bloco. Se não, o bloco deverá passar por todo o processo.

Uma vantagem da tradução e otimização em tempo de execução é a capacidade de simular programas que modificam seu próprio código (do inglês, *self-modifying code*), como é o caso de *boot-loaders* e sistemas operacionais. Para simular aplicativos com estes comportamento foi implementado um mecanismo tratador de exceção utilizando as funções `setjmp` e `longjmp` da biblioteca ANSI C. Todas as instruções de escrita na memória são monitoradas, de maneira que quando alguma escreve em um bloco de código já compilado, uma exceção faz com que a simulação compilada seja interrompida. O simulador, então, retira o bloco modificado da lista de blocos compilados e retoma a simulação. Assim, este bloco que sofreu a modificação será tratado como qualquer outro bloco que ainda não foi compilado. Porém, algumas melhorias precisam ser feitas para que o simulador compilado dinâmico do ArchC 2.1 tenha suporte completo a essa classe de

aplicativos. Um exemplo é o erro do cálculo de endereço quando modelos com instruções de tamanho variável são utilizados.

Na técnica compilada estática, a estrutura `switch`, criada pelo decodificador do `accsim`, é armazenada em um arquivo em disco e, ao compilar o simulador especializado, as chamadas das funções de comportamento sofrem *inlining* e o compilador nativo aplica suas otimizações. Nesta abordagem dinâmica, o `switch` oriundo do decodificador fica armazenado em memória e, com auxílio de um compilador dinâmico, este código é otimizado, transformado em linguagem de máquina e executado.

Na seção anterior, foi comentado sobre o recurso JIT, contido no arcabouço LLVM. Este recurso permite a execução em alto desempenho de códigos em linguagem intermediária LLVM-IR, compilando e otimizando trechos da IR dinamicamente, isto é, em tempo de execução. Por isso, também é encontrado na literatura o termo compilador JIT. É neste momento do simulador, em que o bloco `switch` decodificado precisa ser otimizado e executado, que as bibliotecas do LLVM tornam-se essenciais.

Como mostrado na Figura 4.3, o código em memória não fica estruturado na linguagem C++, como na abordagem anterior, mas sim em LLVM-IR. Isso por que a API responsável por otimizar um código em memória e executá-lo, recebe apenas códigos LLVM-IR como entrada. Pelo mesmo motivo, foi necessário gerar um arquivo em disco com todos os comportamentos em LLVM-IR, para que o otimizador fizesse *inlining* do arquivo para com as chamadas contidas no `switch` em memória.

Cabe lembrar que o LLVM é utilizado como biblioteca e seus recursos são invocados por meio de API. Apesar de existir uma ferramenta no LLVM que faz o JIT de um determinado código (a `lli`), o simulador ArchC incorpora essas funcionalidades dentro de seu código, tornando-as transparente ao projetista. O uso da ferramenta `lli` também não seria possível pois ela recebe um código intermediário, executa, e mostra o resultado

– útil quando o usuário deseja fazer JIT de todo um programa. No caso da simulação, o JIT deve ser invocado de dentro do simulador em execução.

4.3 Implementando a LLVM-IR

A principal rotina de execução está exposta na Figura 4.4. Dado o `ac_pc`, é possível definir em qual região está a instrução a ser executada. Cada região de código é um `switch`, com um número de `cases` configurável pelo projetista (sendo o padrão 256), e cada `switch` é construído dentro de uma função JIT. Assim, a linha 4 resgata uma função JIT já construída e compilada ou a constrói, se a função especificada não estiver armazenada. Logo, existirá uma função JIT por região de código e é esta função que será otimizada e colocada em execução. A linha 5 transforma o ponteiro da função JIT recebido em uma função nativa executável e a linha 6 a executa.

```
1  void mips1::Execute(){
2      ...
3      while (!ac_stop_flag) {
4          void *NativePtr=getOrInsertJITFunction(ac_pc>>REGION_SIZE);
5          void (*NativeFunc)() = (void (*)())(intptr_t)NativePtr;
6          NativeFunc();
7      }
8  }
```

Figura 4.4: Rotina principal da simulação compilada dinâmica

A Figura 4.5 exibe a rotina da criação de uma função JIT. Após o teste que verifica se já existe uma função JIT da região interessada armazenada em um *pool* de funções, na linha 4, o código segue chamando um inicializador padrão para funções JIT, linha 7, que estabelece a estrutura `switch` e define todos os blocos básico necessários, em LLVM-IR. Após, a linha 8 instancia um ponteiro para o primeiro bloco básico da função JIT. Com este ponteiro, é possível caminhar por entre os blocos básicos e ir alocando as instruções

vindas do decodificador. O laço da linha 12 exibe este comportamento, com a linha 14 lançando códigos LLVM-IR no bloco básico apontado.

```
1  void *mips1::getOrInsertJITFunction(int region){
2
3     /* Testa se ja existe funcao JIT para este Bloco */
4     if (FunctionPool[region] == 0) {
5        ...
6        /* Constroi as estruturas basicas para a Funcao JIT */
7        Function *JIT = initJITFunction(region);
8        Function::iterator iBB = JIT->begin();
9        ...
10       // KERNEL: emit instructions
11       int j = (region << REGION_SIZE);
12       for (; j < end_region; j++) {
13          if (decoder(j)) {
14             acdc_EmitInstr(JIT, iBB++, j, region);
15          }
16          ...
17       }
18       BranchInst::Create(iBB->getNextNode(), iBB);
19       ReturnInst::Create(mod->getContext(), ++iBB);
20
21       TheFPM->run(*JIT);
22       inlineFPM(*JIT);
23       FunctionPool[region] = EE->getPointerToFunction(JIT);
24    }
25    return FunctionPool[region];
26 }
```

Figura 4.5: Método responsável por criar as funções JIT

Seguindo, ao sair do laço, tem-se toda a estrutura `switch` formada em LLVM-IR, bem como as chamadas das funções comportamento de cada endereço. As linhas 21 e 22 fazem com que a função JIT criada passe pelas fases de otimização; a linha 23 liga esta função à *Engine* de Execução do LLVM para que esta possa ser executada e também armazena seu ponteiro no *pool* de funções JIT.

O objeto `TheFPM`, da linha 21, é uma instância da classe `FunctionPassManager`, res-

ponsável por aplicar otimizações dado o ponteiro de uma função. Existe uma imensa variedade de otimizações possíveis nesta classe e, ao implementar o simulador, foi tomado como base as otimizações presentes em `-O3`, do gcc-4.4, com várias modificações empíricas, até encontrar o conjunto de otimizações que alcançasse o melhor desempenho. Uma otimização fundamental para a função JIT deste trabalho e que a classe `FunctionPassManager` não implementa, é o *inlining*. Devido ao grande desempenho que iria alcançar, a linha 22 faz uma varredura na função em busca de chamadas de funções e, por meio de recursos da própria API do LLVM, busca o código alvo para fazer parte da função JIT, eliminando a chamada.

Ao sair do laço que emite as instruções (linhas 12 à 17), o código resultante fica similar ao código montado pelo `accsim`, porém em linguagem intermediária LLVM-IR e em memória. A Figura 4.6 exibe um *dump* deste código em memória, no qual o `switch` contém as instruções decodificadas da região 0 e está inserido dentro da função JIT denominada `Region0`. O código desta figura ainda não passou pelo otimizador responsável por fazer *inlining*.

Similarmente ao que acontece em linguagens de baixo nível, diversos blocos básicos devem ser instanciados para se implementar a estrutura `switch`, um para cada `case`. Para facilitar a depuração, todo bloco básico criado para receber as instruções decodificadas recebe o nome de `case` mais o endereço da instrução, como `case8`.

```
1  define void @Region0(i32) {
2      ...
3      entrySwitch:
4      %1 = load i32* getelementptr inbounds (...@ac_pc...)
5      switch i32 %1, label %default [
6          i32 0, label %case0
7          i32 4, label %case4
8          i32 8, label %case8
9          ...
10         ]
11
12     case0:
13         call void @ac_behavior_instruction(...)
14         call void @ac_behavior_Type_I(...)
15         call void @ac_behavior_lui(...)
16         br label %case4
17
18     case4:
19         call void @ac_behavior_instruction(...)
20         call void @ac_behavior_Type_I(...)
21         call void @ac_behavior_addi(...)
22         br label %case8
23
24     case8:
25         call void @ac_behavior_instruction(...)
26         call void @ac_behavior_Type_I(...)
27         call void @ac_behavior_lui(...)
28         br label %case12
29     ...
30     default:
31         br label %return
32
33     return:
34         ret void
35 }
```

Figura 4.6: Exemplo de código em LLVM-IR do `switch` da região 0

4.4 Otimizações FSCS

Na abordagem estática, a adição de informações ao modelo com relação a saltos condicionais e incondicionais, deu origem a duas importantes otimizações que melhoraram drasticamente o desempenho de simulação. A primeira foi retirar os `breaks` após as instruções que não causam desvio pois, obviamente, a próxima instrução será executada. A segunda consiste em reduzir ainda mais os cálculos feitos em tempo de execução, delegando à fase de pré-processamento o cálculo do maior fluxo de execução possível. Apenas cálculos dependentes dos dados de entrada são postergados para o tempo de execução.

A primeira otimização pode ser implementada na abordagem dinâmica sem grande esforço, pois ao lançar o código em LLVM-IR na função JIT, basta não inserir um `branch` para o bloco básico da entrada do `switch`. Porém, o LLVM-IR não permite que haja um bloco básico sem um ponto de saída, ou seja, sem desvio ao final. Logo, como está mostrado na Figura 4.6, o bloco básico de uma instrução que não causa desvio possui um `branch` para o próximo bloco básico e não para a entrada do `switch`.

Já a segunda otimização foi um pouco mais trabalhosa para ser adaptada na abordagem dinâmica. Isso porque o projetista detalha o comportamento de um determinado salto em uma *string* em C++. Na abordagem estática, isto é resolvido copiando esta *string* do arquivo de configuração do ArchC e colando no código fonte do simulador, especificamente no lugar em que aparece o salto em questão. Assim, ao compilar o simulador, esta *string* é resolvida normalmente. A Figura 4.7 possui o detalhamento do comportamento da instrução `jal`, do modelo MIPS1, escrito pelo projetista (acima) e como este código é inserido no simulador (abaixo). Logo, apenas é necessário substituir algumas variáveis, como `ac_pc` e `addr`, pelo valor vigente no momento.

```
1  jal.is_jump(((ac_pc+4) & 0xF0000000) | (addr<<2));
2  jal.delay(1);
3  jal.behavior(RB[31] = (ac_pc+4)+4;);
```

```
1  void Region0(){
2      ...
3      case 0x14:
4          //instr: jal
5          PRINT_TRACE;
6          if (1) {
7              tmp_pc = ((20+4) & 0xF0000000) | (60325<<2);
8              RB[31] = (20+4)+4;
9          }
10         else {
11             tmp_pc = 0x1c;
12         }
13         ac_instr_counter++;
14         // delay=1
15         if (1) {
16             PRINT_TRACE;
17             ac_behavior_instruction(32 , 0);
18             ac_behavior_Type_R(32, 0, 0, 0, 0, 0, 0);
19             ac_behavior_nop(32, 0, 0, 0, 0, 0, 0);
20             ac_instr_counter++;
21         }
22         ac_pc = tmp_pc;
23     break;
24     ...
25 }
```

Figura 4.7: Exemplo de código gerado para a instrução `jal` do mips1 no `accsim`

Na abordagem dinâmica, todo código do `switch` deve ser em LLVM-IR e soluções que compilam *strings* C++, na memória, em LLVM-IR envolvem várias partes do *front-end* e uma codificação não trivial. Uma outra alternativa foi, no momento de gerar o simulador, montar funções de otimização para cada salto e inseri-las no arquivo de comportamento, como exibido na Figura 4.8. Assim, dentro dos `switchs` tem apenas chamadas para essas funções, que retornam um *booleano* que informa se o *delay slot* será ou não executado. Como o `Makefile` guia o arquivo com o comportamento das instruções

para ser transformado em um arquivo em LLVM-IR, estes códigos com o comportamento detalhado dos saltos também são traduzidos para LLVM-IR e sofrem *inline* na fase de otimização.

```
1  int jal_optfunc(int ac_pc, int _tmp_pc , int addr ) {
2     if (1){
3        tmp_pc = ((ac_pc+4) & 0xF0000000) | (addr<<2);
4        RB[31] = (ac_pc+4)+4;
5     }
6     else {
7        tmp_pc = _tmp_pc;
8     }
9     if (1)
10       return 1;
11    return 0;
12 }
```

Figura 4.8: Exemplo de código gerado para a instrução `jal` do mips1 no **acdcsim**

4.5 Evoluções no acdcsim

Durante o processo de implementação do gerador de simuladores `acdcsim`, várias dificuldades foram enfrentadas e o trabalho exposto nas seções anteriores foi fruto de várias modificações ao longo do projeto.

Uma dificuldade encontrada foi trabalhar com a linguagem C++ junto com a *engine* de execução do JIT. Como no ArchC 2.1 tudo é método de classe, inclusive os comportamentos, o esforço inicial foi na tentativa de executar métodos de classe em modo JIT. Assim, os métodos JIT, contendo os blocos de código, invocariam os métodos de comportamento, otimizando logo em seguida. Porém, a ideia fracassou e a *engine* de execução JIT se mostrou complexa ao ir por este caminho. Se funções independentes fossem construídas ao invés de métodos, era necessário apenas passar o ponteiro da função para a *engine* e deixar os recursos em uso na função com o escopo global. Para evitar modi-

ficações no modelo ArchC, elaboradas macros foram codificadas para adaptar o código escrito pelo projetista à abordagem dinâmica.

Estando o `switch` em função JIT, em LLVM-IR, ainda faltava resolver como colocar os comportamentos das instruções em execução. Seguindo orientações da comunidade, o arquivo contendo os comportamentos foi compilado para código objeto e, de dentro da função JIT, era feito uma chamada externa ao código objeto compilado, correspondente ao comportamento da instrução. Apesar de funcional, esta solução não trouxe ganho de desempenho, pois o *inlining* era impossível e as otimizações do módulo JIT não tinham efeito. O programa `sha`, que não possui complexidade, demorava 684s para ser executado e após melhorias no código, o desempenho máximo atingido foi de 137s.

Um esforço, então, foi realizado para que todos os comportamentos fossem traduzidos para LLVM-IR durante a compilação do simulador permitindo, assim, *inlining* na função JIT no momento da execução, junto com todas as otimizações necessárias. Além disso, diversas estruturas foram criadas para evitar que buscas repetitivas fossem feitas, como o `getFunction()`, que retorna um ponteiro para a função JIT. Armazenar esses ponteiros em uma estrutura e chamar este método de busca em apenas um momento trouxe um aumento significativo de desempenho. Assim, o tempo de execução do `sha` chegou a 6s, contra 3,9s do simulador interpretado, e do `lame` chegou a 367s, contra 261s do interpretado. Uma grande melhoria quando comparado à versão com os comportamentos em código objeto.

O módulo JIT é uma estrutura que contém todas as funções JIT geradas dinamicamente. Assim, existe a classe que otimiza cada função JIT, denominada `FunctionPass Manager` e a que otimiza o módulo inteiro, contendo todas as funções JIT, denominada `PassManager`. O *inlining* é uma otimização presente apenas nesta segunda classe. A cada nova função JIT adicionada ao módulo, todo o módulo era mandado para *inline*

novamente, causando muita sobrecarga. Por isso, essa otimização foi reescrita para que apenas a nova função fosse varrida em busca de chamadas de função para *inline*. Devido a essa adaptação, no código da Figura 4.5 há uma chamada separada, na linha 22, para esta otimização.

Para executar uma função JIT, pode-se invocar o objeto `ExecutionEngine` e solicitar o método `runFunction()`. Esta forma foi a primeira implementada no simulador e muito difundida na comunidade. Porém, o desempenho do simulador estava aquém do que se esperava, com resultados piores que o simulador interpretado. Foi então que passou-se a implementar um *pool* de funções, que armazenava o ponteiro da função JIT já compilada. Na Figura 4.5, linha 23, o método `getPointerToFunction()` realiza este trabalho de compilar e retornar um ponteiro. Para executar, é necessário fazer um `cast` e invocar este ponteiro como uma função nativa, como está no código da Figura 4.4, linhas 5 e 6, na rotina principal de simulação.

4.6 Resultados

Com o objetivo de avaliar o simulador compilado dinâmico gerado pela ADL ArchC 2.1, selecionou-se três modelos: MIPS I [26], SPARC V8 [37] e PowerPC [15]. Os aplicativos utilizados são dos *benchmarks* Mediabench [32] e Mibench [33], abordando vários domínios de aplicação. Os aplicativos conseguem ler e escrever em arquivos do disco na máquina hospedeira utilizando emulações de chamadas de sistemas [6].

Todos os experimentos foram executados em um Core i7 860, cuja frequência é de 2.80GHz com 4Gb de memória RAM. O sistema operacional utilizado foi o Linux Ubuntu 10.04 32bits e, para a compilação dos simuladores, utilizou-se o GCC versão 4.4.

Além do desempenho da abordagem em interesse, a compilada dinâmica, os gráficos abaixo exibem o desempenho do simulador interpretado `acsim` para todos os *benchmarks*

nos três modelos, juntamente com os dados do simulador compilado estático, `opt1` e `opt2`. Em todos os casos utilizou-se a opção `-O3` do GCC em conjunto com a *flag* `inline`, para aumentar o desempenho da simulação.

Para cada aplicativo, seis execuções redundantes foram conduzidas para cada simulador, coletando-se o tempo de simulação e o número total de instruções executadas. No simulador compilado estático, também houve a coleta do tempo de compilação do simulador, dispensável nas outras abordagens. Com essas seis execuções, calculou-se o desvio padrão referente ao número de instruções executadas por segundo (MIPS), dando origem ao intervalo de confiança presente nos gráficos a seguir. O grau de confiança utilizado foi de 95%. Os dados da abordagem estática são equivalentes aos apresentados no capítulo anterior.

A Figura 4.9 exibe um gráfico comparando, em milhões de instruções por segundo (MIPS), o desempenho no modelo MIPS I das três abordagens, sendo a abordagem estática separada pelas duas otimizações. A média de execução do simulador dinâmico ficou em 163 MIPS, com simulações atingindo 280 MIPS, no `toast enc`.

Aplicativos que consumiram mais tempo na simulação apresentaram um comportamento mais estável, com um menor desvio padrão, e isto é refletido no intervalo de confiança. Na abordagem compilada estática, aplicativos simples, com poucas instruções, alcançam um alto desempenho na simulação, e centésimos de segundo causam um grande impacto na unidade de medida MIPS (milhões de instruções por segundo). Já as abordagens interpretada e compilada dinâmica, por consumirem mais tempo nas simulações, ficaram com um baixo desvio padrão, deixando o intervalo de confiança quase imperceptível na escala em que se encontra o gráfico.

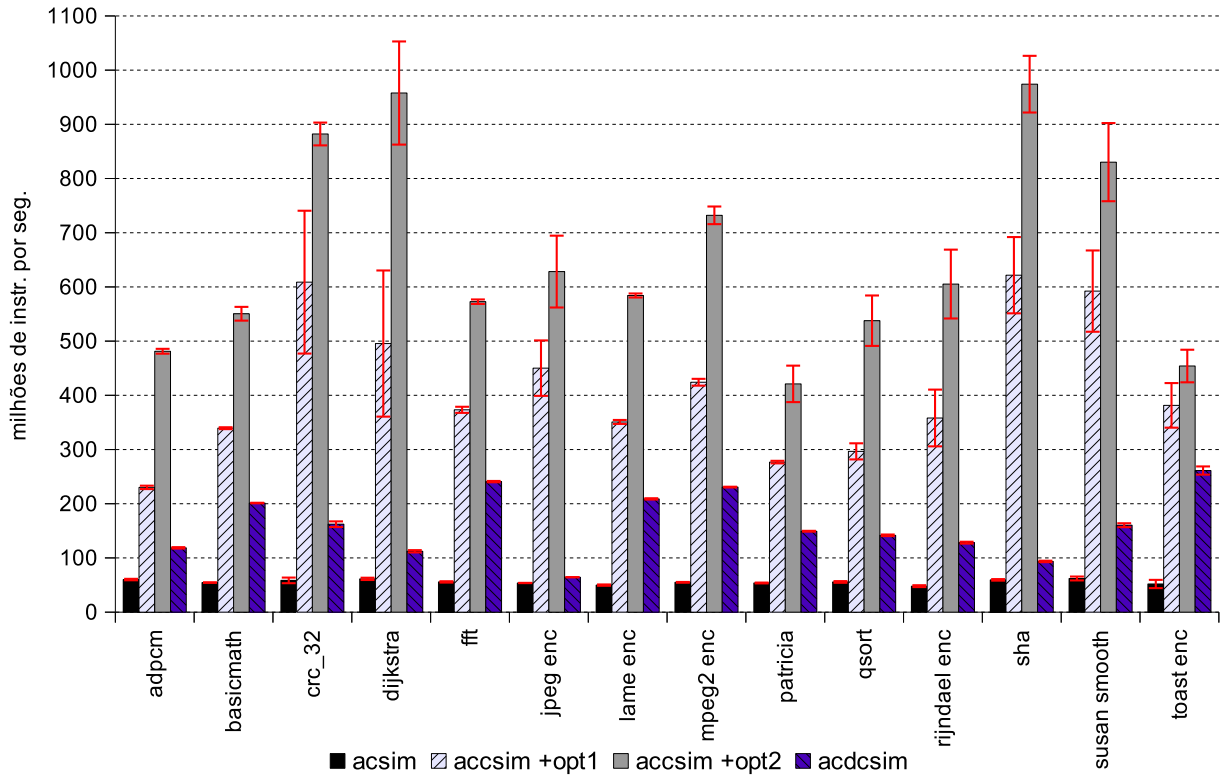

Figura 4.9: Simulação das três abordagens para o modelo MIPS I

Diferente do simulador interpretado e do simulador compilado dinâmico, a abordagem estática requer que o simulador seja especializado para o binário em questão. Ou seja, para cada binário alvo a simular, uma instância do simulador precisa ser construída e compilada. A construção envolve a decodificação do binário integralmente e compilar as estruturas geradas consome um tempo considerável. Por causa desse processo, o simulador estático pode se tornar menos conveniente, sendo útil apenas quando um único binário estará sendo testado por um longo tempo ou quando o *overhead* da construção e compilação do simulador ainda compense quando comparado com outras abordagens. O gráfico da Figura 4.10 adiciona este *overhead* de compilação do simulador compilado estático com `opt2` para o modelo MIPS I, podendo comparar com o tempo gasto na si-

mulação da abordagem compilada dinâmica. Como o simulador compilado dinâmico é gerado uma única vez para todas as aplicações, o *overhead* de compilação foi ignorado. A escala logarítmica foi utilizada para facilitar a leitura dos dados de aplicativos que atingiram alto desempenho.

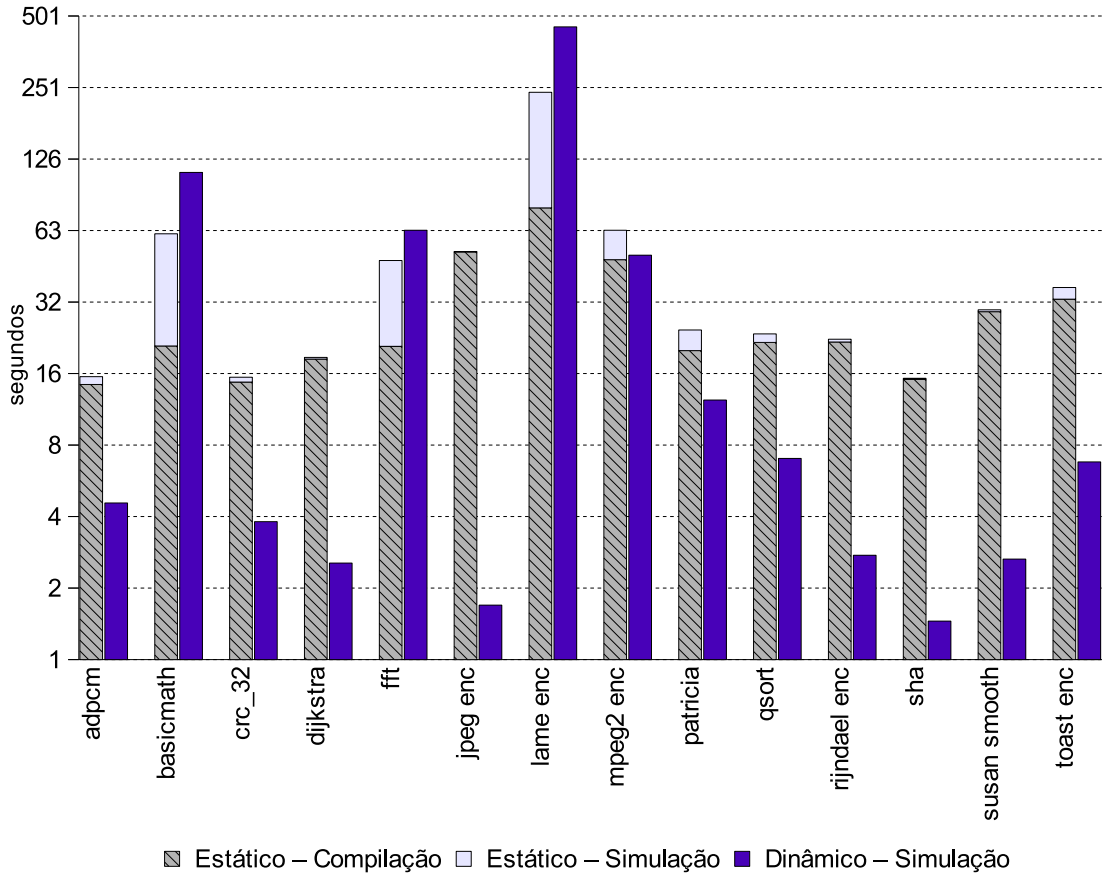

Figura 4.10: Comparação da abordagem estática e dinâmica em relação ao tempo para o modelo MIPS I

De acordo com o gráfico, na abordagem compilada estática, mesmo que o tempo de simulação do aplicativo `crc_32` seja menos de 1 segundo, o projetista precisaria esperar 16 segundos para concluir o experimento, pois a compilação do simulador especializado gasta 15 segundos. Fica evidente, portanto, que grande parte do tempo é destinado à compilação do simulador. Logo, o gráfico da Figura 4.11 inclui este tempo de compilação

do simulador compilado estático em um gráfico de desempenho similar ao mostrado na Figura 4.9. O *overhead* de compilação do simulador interpretado e do simulador compilado dinâmico foi ignorado, pois eles são gerados uma única vez para todas as aplicações.

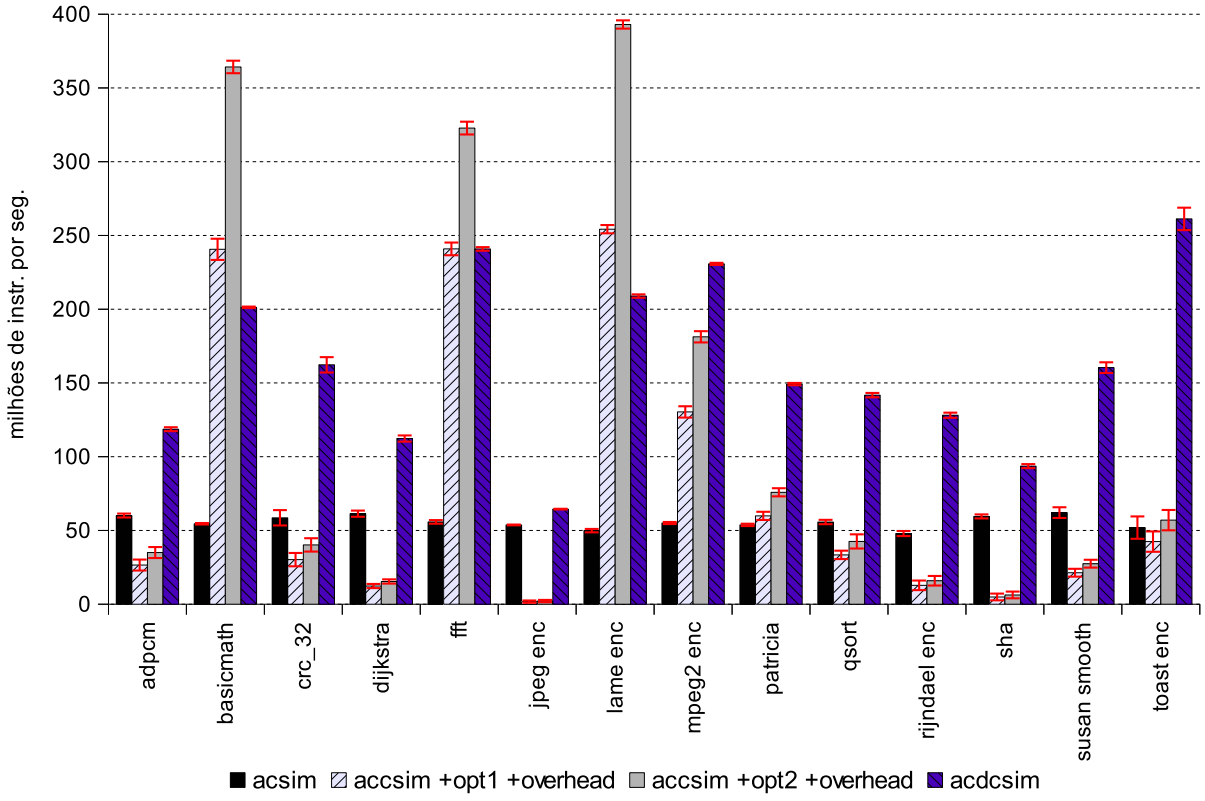

Figura 4.11: Simulação do modelo MIPS I com *overhead* no `accsim`

Neste cenário, o simulador compilado estático supera o dinâmico em apenas três aplicativos, `basicmath`, `fft` e `lame`, sendo que todos estes fazem uso de operações com ponto flutuante. Adicionando o *overhead*, o simulador interpretado também supera o simulador compilado estático para algumas aplicações, servindo de alerta ao projetista sobre equívocos na hora de escolher a melhor abordagem para as suas simulações. Ao inserir o tempo de compilação do simulador compilado estático no cálculo da unidade MIPS (milhões de instruções por segundo), o desvio padrão abaixa significantemente, como

pode-se perceber por meio do intervalo de confiança.

Há aplicativos em que a simulação interpretada possui um desempenho próximo ao da simulação compilada dinâmica, como é o caso do `sha` e do `jpeg`. De acordo com a Figura 4.12, estes dois aplicativos possuem os menores números de instruções executadas, e a alta carga de bibliotecas e estruturas do simulador compilado dinâmico, faz com que aplicativos pequenos tenham simulações ineficientes, com desempenho semelhante ao interpretado.

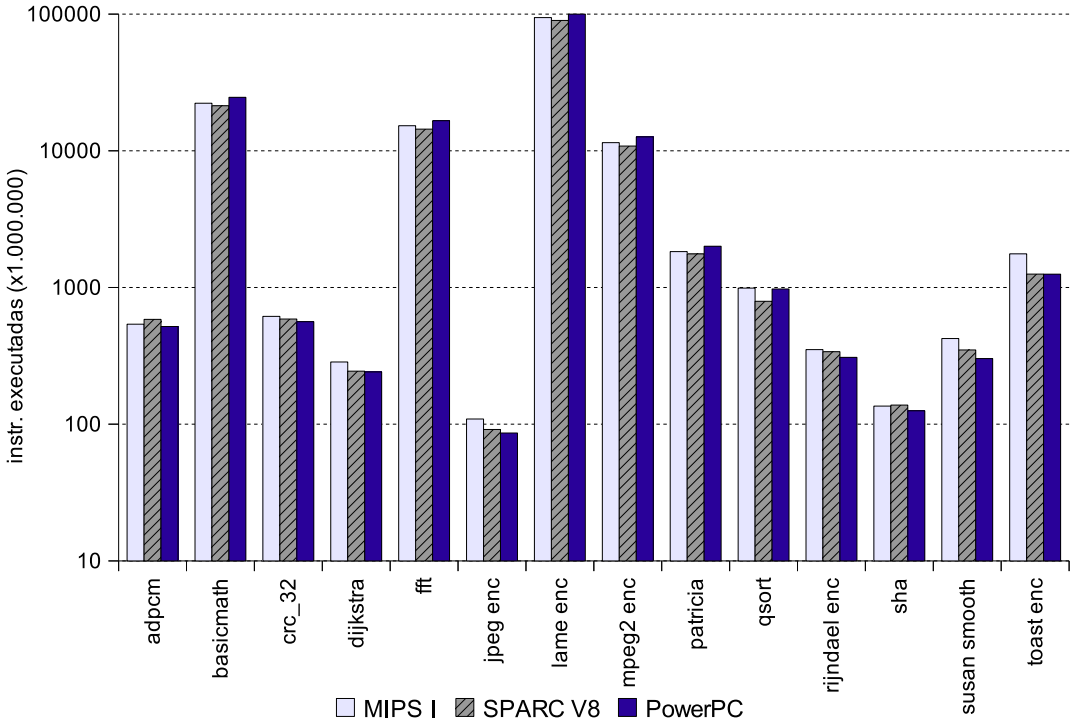

Figura 4.12: Número de instruções executadas nas simulações

Os experimentos com os outros modelos apresentaram o mesmo comportamento que o MIPS I, sendo a abordagem estática com `opt2`, sem considerar o *overhead* da geração do simulador, a que obteve o melhor resultado, conforme ilustra a Figura 4.13, referente a simulações com o modelo SPARC V8 e PowerPC.

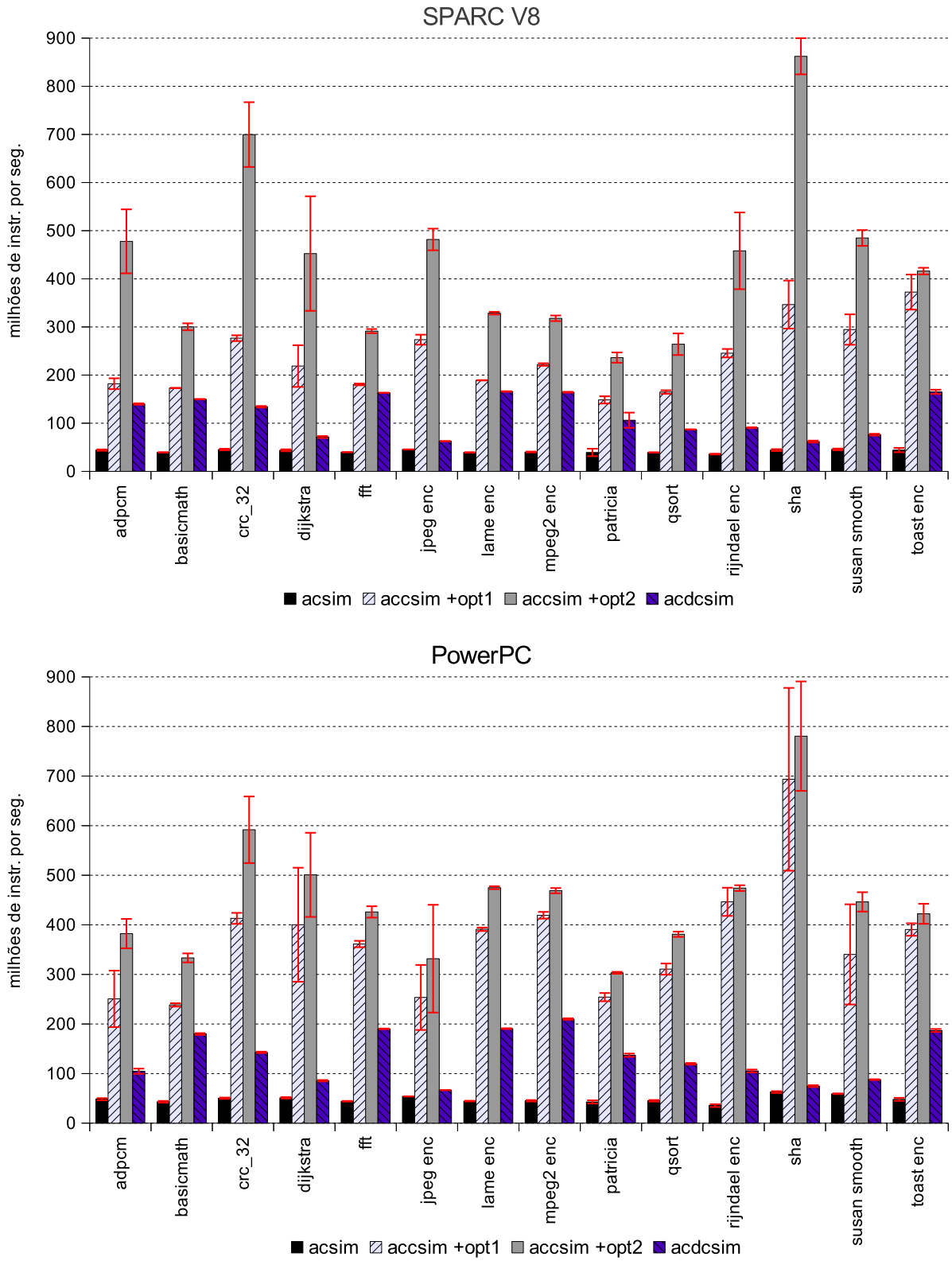

Figura 4.13: Simulação das três abordagens para os modelos SPARC V8 e PowerPC

Ao inserir o *overhead* da geração do simulador compilado estático nesses dois modelos, Figura 4.14, o cenário fica semelhante às simulações com o modelo MIPS I, com a abordagem dinâmica alcançando desempenho superior às outras para a maioria dos aplicativos.

O modelo SPARC V8 apresentou média de 120 MIPS para a simulação dinâmica, com picos de 165 MIPS para o aplicativo `toast enc`. Como aconteceu no modelo MIPS I, o desempenho desta abordagem não superou a estática (com *overhead*) nos aplicativos `basicmath`, `fft` e `lame`.

Nas simulações feitas no modelo PowerPC, a abordagem dinâmica teve uma execução média de 135 MIPS, com picos de 210 para `mpeg2 enc`. Novamente, apenas os aplicativos `basicmath`, `fft` e `lame` apresentaram desempenhos superiores ao utilizar a simulação compilada estática.

Há também, nesses dois modelos, proximidades de desempenho entre a abordagem dinâmica e a interpretada para os aplicativos que possuem um pequeno número de instruções executadas, como `jpeg` e `sha`. Comparando a Figura 4.12, que informa o número de instruções executadas durante a simulação, com os gráficos de desempenho, percebe-se que a abordagem dinâmica possui melhor desempenho quando programas maiores são simulados, pois a carga de toda a infraestrutura LLVM é feita em tempo de simulação, prejudicando o desempenho de aplicativos pequenos.

Cenários onde um mesmo aplicativo é simulado repetidas vezes por muito tempo também faz a abordagem dinâmica ser ineficiente, pois o *overhead* inicial para se construir o simulador compilado estático é compensado pelo alto desempenho na simulação. Isto se o aplicativo não modificar seu próprio código durante a simulação, pois o simulador estático não é capaz de resolver este tipo de comportamento. Neste caso, o projetista usuário do ArchC pode optar entre a abordagem interpretada e a compilada dinâmica.

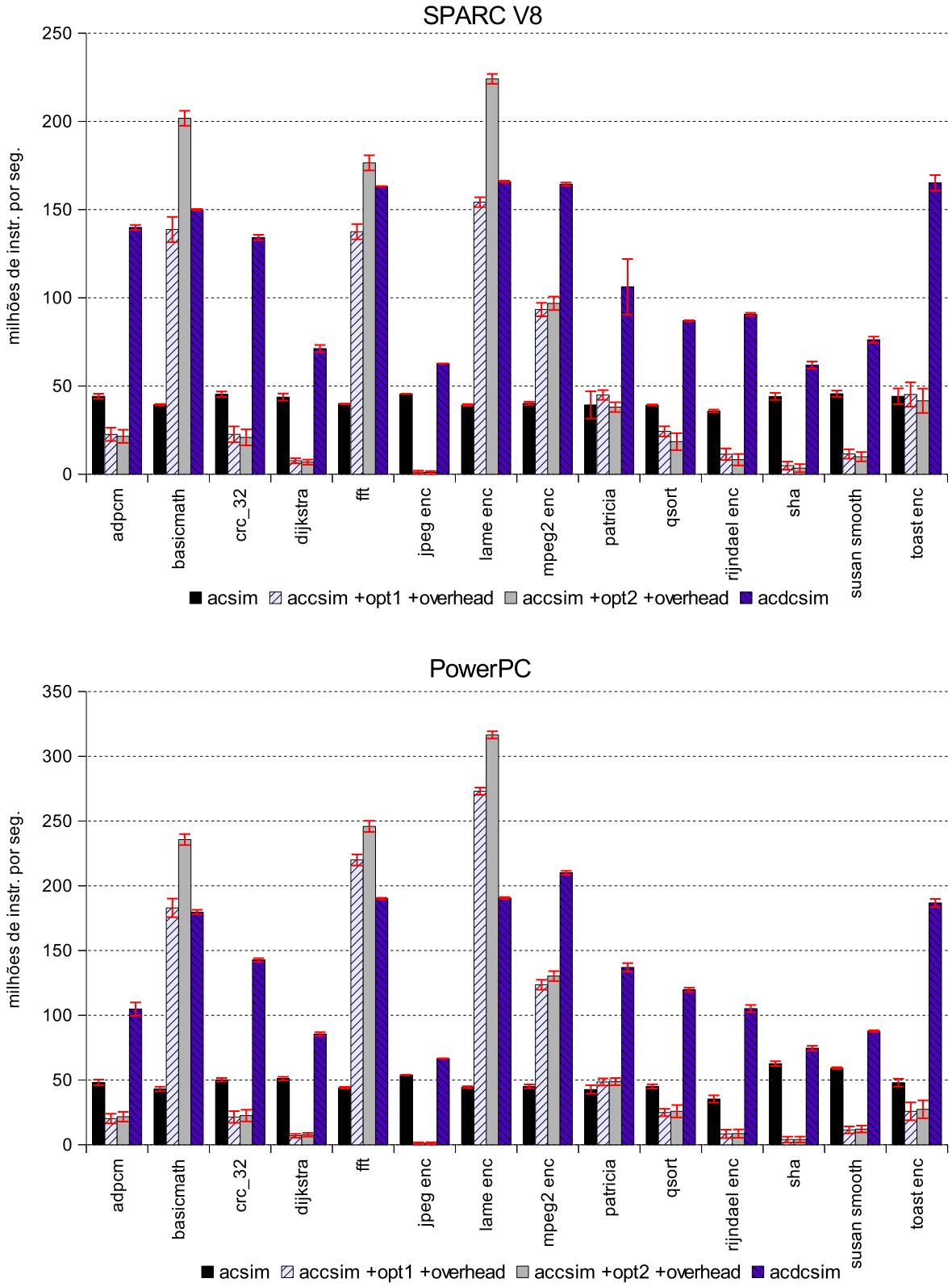

Figura 4.14: Simulação dos modelos SPARC V8 e PowerPC com *overhead*

Capítulo 5

Conclusão

Este trabalho de mestrado dotou o Projeto ArchC, atualmente na versão 2.1, com duas novas ferramentas de simulação, ambas mais rápidas que o simulador interpretado existente. Estas novas ferramentas fazem uso da técnica de simulação compilada, em que o aplicativo, ou parte dele, é pré-processado antes da simulação. Quando se conhece o aplicativo previamente, o gerador de simulador pode analisar o binário por completo e especializar um simulador para a aplicação, aumentando muito o desempenho. Este modelo de simulação compilada recebe o nome de estática. A falta de flexibilidade desta abordagem requer que um outro modelo seja implementado, pois nem sempre é interessante especializar um simulador para cada aplicativo, mesmo com seu alto desempenho. Assim, no modelo de simulação compilada dinâmica, o simulador recebe o aplicativo, divide-o em vários blocos e, antes de simular um determinado bloco, faz um processamento para armazenar os resultados das operações. Assim, a simulação de um bloco será mais rápida que interpretando cada instrução; identificar que um bloco necessita ser simulado novamente, estando ele já processado, faz o desempenho aumentar.

Possuir simuladores rápidos no conjunto de ferramentas do ArchC é importante para reduzir o ciclo de desenvolvimento de uma nova arquitetura. Muitas são as variáveis a

serem consideradas na criação de um novo projeto de arquitetura computacional e uma simulação rápida proporciona uma melhor exploração do espaço de desenvolvimento sem que o tempo de projeto seja comprometido.

O projeto ArchC, em versões anteriores, já contava com um gerador de simuladores compilados estático. Porém, depois de algumas melhorias internas no ArchC, essa ferramenta de simulação, que possuía alto desempenho, parou de ter suporte e não estava presente nas últimas versões do projeto. Logo, a primeira etapa deste trabalho, descrita no Capítulo 3, foi reimplementar esta ferramenta para novamente ser compatível ao ArchC, otimizando algumas estruturas para obter mais desempenho.

Os resultados obtidos com o simulador compilado estático gerado pelo novo `accsim` mostraram que esta ferramenta pode se tornar uma excelente opção ao projetista que deseja simular aplicativos com alta velocidade, alcançando a média de 642 milhões de instruções por segundo (MIPS) para a arquitetura MIPS I. A simulação de algumas aplicações, como o `sha`, `crc_32` e o `jpeg`, ficaram apenas 3x mais lenta que uma execução nativa em máquina Intel, com pico de desempenho de 974 milhões de instruções por segundo (MIPS).

O maior programa executado nos testes foi o `lame`, um codificador de áudio para formato MP3. No simulador interpretado para a arquitetura MIPS I, seu desempenho foi de 55 milhões de instruções por segundo (MIPS), levando cerca de 30 minutos para executar a entrada de teste. No simulador compilado estático para a mesma arquitetura, a execução alcançou a marca de 584 MIPS, demorando 3 minutos para simular a mesma estrada de teste. Porém, ao utilizar um simulador compilado estático, o tempo de geração do simulador especializado deve ser levado em consideração no tempo total de simulação, que no caso do `lame` foi de 4 minutos. Este tempo de geração se mostrou linearmente proporcional ao tamanho da aplicação.

O segundo objetivo deste trabalho foi implementar um gerador de simuladores que fosse mais flexível que a abordagem compilada estática, mas que ainda atingisse um desempenho superior aos simuladores interpretados. Assim foi desenvolvido o gerador **acdcsim**, que concebe simuladores compilados dinâmicos. Sua maleabilidade se dá por não precisar informar o aplicativo a ser simulado no momento da criação do simulador, ou seja, não há especialização. Assim, um único simulador é necessário por arquitetura, como acontece na técnica interpretada. Outra vantagem de levar para o momento da execução a decodificação e compilação do binário é a possibilidade de executar códigos que se auto modificam, como *boot-loaders* e sistemas operacionais.

O desempenho dos simuladores gerados pelo `acdcsim` superou o simulador interpretado e o simulador compilado estático com a primeira otimização da técnica FSCS ativada, atingindo média de 163 milhões de instruções por segundo (MIPS) para a arquitetura MIPS I, com picos de 280 MIPS. Não obstante, os simuladores mais rápidos do ArchC 2.1 ainda são aqueles gerados pelo `accsim`, com todas as otimizações ativada. Mas se considerar o tempo de construção e compilação do simulador compilado estático no cálculo do desempenho, a média para a arquitetura MIPS I salta de 640 MIPS para 103 MIPS, ou seja, inferior à abordagem dinâmica. Assim, o projetista tem duas possibilidades: um simulador muito rápido com um alto custo em sua construção – interessante quando se deseja simular diversas vezes o mesmo aplicativo, com variadas entradas – e um simulador flexível, sem custo de construção e com bom desempenho.

A primeira parte do trabalho aqui exposto, referente à ferramenta `accsim` e à simulação compilada estática no ArchC 2.1, foi publicada em Garcia *et al.* [17], no ano de 2010. A segunda parte do trabalho, que diz respeito à simulação compilada dinâmica no ArchC 2.1, no momento em que esta dissertação foi finalizada ainda não havia publicação.

Com base no que foi apresentado aqui, vários trabalhos interessantes podem sur-

gir futuramente, tanto na parte da simulação compilada estática quanto na dinâmica. Incluir um módulo de codificação e compilação em tempo de execução na abordagem estática, tornando-a híbrida, é uma opção interessante, pois deixaria a simulação mais flexível. Estender o simulador compilado dinâmico para se trabalhar com vários núcleos pode fazer com que a simulação aumente o desempenho, onde um núcleo fica responsável pela simulação, enquanto os outros preparam blocos de código para a simulação. Esta abordagem já está presente em outros trabalhos da área e apresentou um bom aumento de desempenho. Amadurecer o suporte ao *self-modifying code* no simulador compilado dinâmico permitiria que aplicativos mais complexos, como um sistema operacional, fossem simulados rapidamente. Uma grande contribuição seria melhorar a seleção dos blocos a se compilar na técnica dinâmica, colocando algumas heurísticas ou *profile* para saber se há vantagem em se compilar o bloco todo, ou apenas executar aquelas instruções de maneira interpretada.

Com relação a plataformas multiprocessadas, tanto o simulador compilado estático quanto o dinâmico requerem futuras implementações para que seja possível inserí-los em uma plataforma SystemC com TLM. E vivendo em um mundo cada vez mais multiprocessado, em que até *smartphones* possuem 4 núcleos, é essencial poder simular ambientes com vários processadores.

Referências Bibliográficas

[1] Bruno Albertini. Um framework de desenvolvimento de plataformas e um mecanismo de depuração baseado em reflexão computacional. Master's thesis, Instituto de Computação - Universidade Estadual de Campinas, 2005.

[2] ArchC. http://www.archc.org, 2011.

[3] Rodolfo Azevedo, Sandro Rigo, and Guido Araújo. *Projeto e Desenvolvimento de Sistemas Dedicados Multiprocessados*. Livro das Jornadas de Atualização em Informática, Campo Grande, MS, Brasil, 2006.

[4] Marcus Bartholomeu. *Simulação Compilada para Arquiteturas Descritas em ArchC*. PhD thesis, Instituto de Computação - Universidade Estadual de Campinas, 2005.

[5] Marcus Bartholomeu, Rodolfo Azevedo, Sandro Rigo, and Guido Araujo. Optimizations for compiled simulation using instruction type information. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '04, pages 74–81, Foz do Iguaçu, PR, Brasil, 2004. IEEE Computer Society.

[6] Marcus Bartholomeu, Sandro Rigo, Rodolfo Azevedo, and Guido Araujo. Emulating operating system calls in retargetable isa simulators. Technical Report IC-03-29,

Emulating Operating System Calls in Retargetable ISA Simulators de Computação - Universidade Estadual de Campinas.

[7] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 13–22, Salzburg, Austria, 2007. ACM.

[8] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. Fast and accurate simulation using the llvm compiler framework. In *Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '09, Paphos, Cyprus, 2009.

[9] Gunnar Braun, Andreas Hoffmann, Achim Nohl, and Heinrich Meyr. Using static scheduling techniques for the retargeting of high speed, compiled simulators for embedded processors from an abstract machine description. In *Proceedings of the 14th international symposium on Systems synthesis*, ISSS '01, pages 57–62, Montreal, PQ, Canada, 2001. ACM.

[10] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 83–94, Vancouver, British Columbia, Canada, 2000. ACM.

[11] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25:13–25, June 1997.

[12] Martin Burtscher and Ilya Ganusov. Automatic synthesis of high-speed processor simulators. In *Proceedings of the 37th annual IEEE/ACM International Symposium on*

Microarchitecture, MICRO 37, pages 55–66, Portland, Oregon, 2004. IEEE Computer Society.

[13] Clang. http://clang.llvm.org/, 2011.

[14] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '94, pages 128–137, Nashville, Tennessee, United States, 1994. ACM.

[15] Keith Diefendorff and Ed Silha. The powerpc user instruction set architecture. *IEEE Micro*, 14:30–41, October 1994.

[16] Frank Engel, Johannes Nührenberg, and Gerhard P. Fettweis. A generic tool set for application specific processor architectures. In *Proceedings of the eighth international workshop on Hardware/software codesign*, CODES '00, pages 126–130, San Diego, California, United States, 2000. ACM.

[17] Maxiwell S. Garcia, Rodolfo Azevedo, and Sandro Rigo. Optimizing a retargetable compiled simulator to achieve near-native performance. In *Proceedings of the 11th Symposium on Computing Systems*, WSCAD-SCC '10, pages 33–39, Petrópolis, RJ, Brasil, 2010. IEEE Computer Society.

[18] GCC. http://gcc.gnu.org/, 2011.

[19] IBM Genesys. http://www.research.ibm.com/haifa/projects/verification/genesys.html, 2011.

[20] Peter Grun, Ashok Halambi, Nikil Dutt, and Alex Nicolau. Rtgen: an algorithm for automatic generation of reservation tables from architectural descriptions. *IEEE Trans. Very Large Scale Integr. Syst.*, 11:731–737, August 2003.

[21] John C. Gyllenhaal, Wen mei W. Hwu, and B. Ramabriohna Rau. Optimization of machine descriptions for efficient use. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 349–358, Paris, France, 1996. IEEE Computer Society.

[22] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: an instruction set description language for retargetability. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 299–302, Anaheim, California, United States, 1997. ACM.

[23] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '99, pages 485–490, Munich, Germany, 1999.

[24] Mark R. Hartoog, James A. Rowson, Prakash D. Reddy, Soumya Desai, Douglas D. Dunlop, Edwin A. Harcourt, and Neeti Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 303–306, Anaheim, California, United States, 1997. ACM.

[25] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[26] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[27] Asheesh Khare, Nicolae Savoiu, Ashok Halambi, Peter Grun, Nikil Dutt, and Alex Nicolau. V-sat: A visual specification and analysis tool for system-on-chip explo-

ration. In *Proceedings of the 25th Euromicro Conference*, EUROMICRO '99, pages 1196–1203, Milan, Italy, 1999.

[28] Stefan Kraemer, Lei Gao, Jan Weinstock, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Hysim: a fast simulation framework for embedded software development. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 75–80, Salzburg, Austria, 2007. ACM.

[29] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 0–75, Palo Alto, California, 2004. IEEE Computer Society.

[30] Rainer Leupers, Johann Elste, and Birger Landwehr. Generation of interpretive and compiled instruction set simulators. In *Proceedings of the Asia and South Pacific Design Automation Conference*, ASP-DAC '99, pages 339–342, Wanchai, Hong Kong, 1999.

[31] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, February 2002.

[32] Mediabench. http://euler.slu.edu/ fritts/mediabench, 2011.

[33] Mibench. http://www.eecs.umich.edu/mibench, 2011.

[34] João Moreira, Felipe Klein, Alexandro Baldassin, Paulo C. Centoducatte, Rodolfo Azevedo, and Sandro Rigo. Using multiple abstraction levels to speedup an mpsoc

virtual platform simulator. In *Proceedings of the 22nd IEEE International Symposium on Rapid System Prototyping*, RSP '11, pages 99 –105, Karlsruhe, Germany, 2011.

[35] João B. C. G. Moreira. Análise do consumo de energia em stms e uma plataforma de simulação multiprocessada com abstração híbrida. Master's thesis, Instituto de Computação - Universidade Estadual de Campinas, 2010.

[36] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 22–27, New Orleans, Louisiana, USA, 2002. ACM.

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

[38] Pierre G. Paulin, Clifford Liem, Trevor C. May, and Shailesh Sutarwala. Flexware: A flexible firmware development environment for embedded systems. In Peter Marwedel and Gert Goossens, editors, *Code Generation for Embedded Processors*, pages 67–84. Kluwer, 1994.

[39] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargeting of compiled simulators for digital signal processors using a machine description language. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '00, pages 669–673, Paris, France, 2000. ACM.

[40] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisamachine description language for cycle-accurate models of programmable dsp architectures. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 933–938, New Orleans, Louisiana, United States, 1999. ACM.

[41] Douglas Perry. *VHDL*. McGraw-Hill, 3rd edition, 1998.

[42] Wei Qin. http://simit-arm.sourceforge.net, 2011.

[43] Wei Qin. http://simit-mips.sourceforge.net, 2011.

[44] Wei Qin, Joseph D'Errico, and Xinping Zhu. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 193–198, Seoul, Korea, 2006. ACM.

[45] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 758–763, Anaheim, CA, USA, 2003.

[46] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. *ACM Trans. Embed. Comput. Syst.*, 8:20:1–20:27, April 2009.

[47] Sandro Rigo, Guido Araujo, Marcus Bartholomeu, and Rodolfo Azevedo. Archc: a systemc-based architecture description language. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '04, pages 66 – 73, Foz do Iguaçu, PR, Brasil, 2004.

[48] Sandro Rigo, Rodolfo Azevedo, and Guido Araújo. The archc architecture description language. Technical Report IC-03-15, Instituto de Computação - Universidade Estadual de Campinas, 2003.

[49] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the simos machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7:78–103, January 1997.

[50] James A. Rowson. Hardware/software co-simulation. In *Proceedings of the 31st annual Design Automation Conference*, DAC '94, pages 439–440, San Diego, California, United States, 1994. ACM.

[51] Vivek Sagdeo. *The Complete VERILOG Book*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[52] Eric C. Schnarr, Mark D. Hill, and James R. Larus. Facile: a language and compiler for high-performance processor simulators. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 321–331, Snowbird, Utah, United States, 2001. ACM.

[53] Thiago M. Sigrist. Reestruturação de archc para integração a metodologias de projeto baseadas em tlm. Master's thesis, Instituto de Computação - Universidade Estadual de Campinas, 2007.

[54] Chuck Siska. A processor desription language supporting retargetable multi-pipeline dsp program development tools. In *Proceedings of the 11th international symposium on System synthesis*, ISSS '98, pages 31–36, Hsinchu, Taiwan, China, 1998. IEEE Computer Society.

[55] Stuart Swan and Cadence Design Systems. An introduction to system level modeling in systemc 2.0. Technical report, Cadence Design Systems, Inc., 2001.

[56] Synopsys. http://www.synopsys.com, 2011.

[57] SystemC. http://www.systemc.org, 2011.

[58] Mohamed-Wassim Youssef, Sungjoo Yoo, Arif Sasongko, Yanick Paviot, and Ahmed A. Jerraya. Debugging hw/sw interface for mpsoc: video encoder system design case study. In *Proceedings of the 41st Design Automation Conference*, DAC '04, pages 908–913, San Diego, CA, USA, 2004. ACM.

[59] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description language and generic machine model for hw/sw co-design. In *in Proceedings of the IEEE Workshop on VLSI Signal Processing*, VLSISP '96, pages 127–136, San Francisco, USA, 1996.