Optimizing a Retargetable Compiled Simulator to Achieve Near-native Performance

Maxiwell Salvador Garcia, Rodolfo Azevedo, Sandro Rigo
Instituto de Computação, UNICAMP
maxiwell.garcia@students.ic.unicamp.br, {rodolfo, sandro}@ic.unicamp.br

Abstract

The design of new architectures can be simplified with the use of retargetable instruction set simulation tools, which can validate the decisions in the design exploration cycle with high flexibility and reduced cost. The increasing system complexity makes the traditional approach to simulation inefficient for today's architectures. The compiled simulation technique makes use of a priori knowledge about the application to accelerate the simulation with high efficiency. This paper presents a retargetable compiled simulator with three optimization techniques and taking advantage of new GCC optimizations to improve the performance. Three architectures were modeled and tested, MIPS, SPARC and PowerPC. Our MIPS model achieved the best results, with average of 651 million instruction per second, and only 2.8 times slower than native execution.

1. Introdução

Simulação é um dos métodos mais utilizados para avaliação de desempenho de um sistema, tendo como principal vantagem a flexibilidade e o custo reduzido. A utilização de simulação em arquiteturas de computadores iniciou-se na década de 70, com a técnica *threaded* code [4]. Estas ferramentas oferecem mecanismos para que uma maquina hospedeira execute uma aplicação de uma máquina alvo.

Na abordagem clássica, a simulação é feita reproduzindo o comportamento do hardware: carrega as instruções da memória, decodifica e executa. Este paradigma é denominado de simulação interpretada. Porém, a crescente complexidade dos sistemas tem tornado este modelo tradicional ineficiente. Além disso, devido à pressão imposta pelo *time-to-market,* desempenho na simulação é um requisito fundamental no projeto das modernas e complexas arquiteturas atuais.

Algumas alternativas foram propostas para melhorar o desempenho. Primeiramente, o *overhead* da decodificação foi atenuado inserindo uma cache com as instruções já decodificadas. Com isso, a execução de laços foi melhorada, pois suas instruções são decodificadas apenas uma vez. Outra melhoria foi usar o conhecimento prévio do aplicativo a ser executado no simulador e mover operações frequentes da execução para a compilação. Esta trouxe ganhos significativos, chegando a até duas ordens de magnitude na velocidade, se comparada com a simulação interpretada, sem perder a precisão [18]. Esta abordagem passou a ser chamada de simulação compilada e consiste em gerar um simulador otimizado para um aplicativo específico, necessitando de um passo de pré-processamento antes da simulação. Este passo é feito na compilação do simulador, e o aplicativo é inteiro decodificado e suas instruções são instanciadas estaticamente. Os simuladores gerados para este trabalho utilizam este princípio, implementando algumas otimizações e utilizando recursos do gcc-4.4 para alcançar o melhor desempenho.

Devido ao *overhead* da compilação, o foco principal destes simuladores são projetistas de ASIPs e DSPs, que executam apenas uma aplicação no sistema. Porém, com o poder de processamento atual, o pequeno *overhead* na compilação é compensado pelo desempenho alcançado na simulação. Os aplicativos, por sua vez, devem ser estáticos durante a execução.

Devido à complexidade e à demora para modelar um processador para simulação, tornou-se importante ferramentas que derivassem simuladores e ferramentas auxiliares de forma automática, baseando-se em uma Linguagem de Descrição de Arquitetura (do inglês, *Architecture Description Languages, ADL*). Todos os simuladores utilizados neste trabalho foram implementados com base na ADL ArchC [15].

No ArchC 1.6, de 2004, a primeira versão do simulador compilado foi lançada e algumas técnicas, denominadas de *Fast Static Compiled Simulation* (FSCS), foram implementadas para aumentar o desempenho da simulação [2]. Com o lançamento do ArchC 2.0, devido à mudanças internas na ADL, o simulador compilado parou de ter suporte. No ArchC

978-0-7695-4274-4/10 $26.00 © 2010 IEEE
DOI 10.1109/WSCAD-SCC.2010.17

2.1, de 2010, o gerador de simulador compilado, doravante *accsim,* foi incluído novamente no projeto e detalhes de otimizações e codificações são apresentadas neste artigo.

Este artigo segue, assim, organizado: a seção 2 comenta sobre os trabalhos relacionados à simulação compilada e a seção 3 introduz, sumariamente, a ADL ArchC. Nas seções 4 e 5 estão as evoluções e otimizações necessárias para que o *accsim* atingisse um desempenho superior ao encontrado na literatura. A seção 6 apresenta os resultados de algumas simulações utilizando as arquiteturas MIPS I, SPARC V8 e POWERPC. Finalmente, a seção 7 resume a conclusão do trabalho e descreve os trabalhos futuros.

2. Trabalhos relacionados

É amplo o número de simuladores e ADLs sendo pesquisados atualmente. Inicialmente, é importante dividir em duas grandes classes de simuladores. De um lado encontram-se aqueles que modelam o comportamento do hardware, simulando ciclo a ciclo (*cycle-accurate*), necessário quando se almeja uma análise de desempenho com alta precisão. Porém, isso leva muito tempo de simulação. De outro lado, têm-se os simuladores funcionais, que oferecem um ambiente que simula o comportamento da arquitetura, ideal para desenvolvedores e simulações rápidas. Algumas ADLs, dependendo do nível de detalhes descritos, podem gerar tanto simuladores *cycle-accurate* quanto funcionais.

A linguagem LISA [12] pode ser utilizada para gerar simuladores rápidos com a técnica compilada. Nohl *et al* [10] apresenta a técnica *Just-in-Time Cache Compiled Simulation* (JIT-CCS) que combina a flexibilidade da técnica interpretada com a velocidade da compilada. Para isso, durante a simulação, algumas instruções que estão à frente são decodificadas, antes da sua execução, e armazenadas em uma cache. Esta alternativa é comumente chamada de simulação compilada dinamicamente. Em Qin *et al* [13], esta mesma técnica é apresentada, mas em um ambiente *muticore*, onde um núcleo é responsável pela simulação e os outros pela decodificação das instruções posteriores.

EXPRESSION [5] é uma ADL que tem como foco a geração de simuladores e compiladores. Reshadi *et al* [14] apresenta a técnica chamada de *Instruction-Set Compiled Simulation* (IS-CS), desenvolvida para melhorar o desempenho de simulação e foi implementada na EXPRESSION. O IS-CS foi 40% melhor que JIT-CCS. Este ganho de desempenho foi atingido pois levou a decodificação das instruções da simulação para a compilação e utilizou estruturas em C, permitindo uma melhor otimização dos compiladores.

A ADL MiADL [8] também é capaz de gerar um simulador compilado e seu resultado é cerca de 90% superior ao IS-CS do EXPRESSION. Em [2], o simulador compilado base do ArchC 1.6, *i.e.* apenas com as otimizações base, conseguia um desempenho de 90% à 170% superior ao IS-CS. Se todas as otimizações fossem ligadas, o *speedup* chegava a 244% comparando com o base.

3. ArchC

ArchC [15] é uma linguagem de descrição de arquiteturas baseado em C e SystemC e, após algumas versões sem oferecer suporte ao simulador compilado, a recém lancada versão 2.1 trouxe esta abordagem novamente. O ArchC está disponível publicamente sob a licença GPL no site `www.archc.org`.

ArchC foi desenvolvida inicialmente para descrever arquiteturas de processadores, tendo expressividade suficiente para modelar várias classes de arquiteturas (RISC, CISC, DSPs, etc). Com isso, o projetista explora rapidamente um novo conjunto de instruções, gerando várias ferramentas automaticamente, como simuladores, montadores e depuradores. Na atual versão, além dos processadores, é possível descrever plataformas inteiras, com barramentos, processadores, memórias, entre outros IPs (*Intellectual Property*), utilizando TLM e SystemC. Para facilitar a construção destas plataformas, um *framework* denominado *ArchC Reference Plataform* (ARP) pode ser utilizado [1].

A descrição de uma arquitetura 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 ArchC pode gerar um simulador interpretado escrito em SystemC, que pode ser funcional ou *cycle-accurate*, dependendo do nível de abstração utilizado. Um simulador compilado puramente funcional também pode ser gerado, necessitando informar o aplicativo a ser executado no momento da geração.

3.1 Simulação Compilada

Na ADL ArchC, é possível gerar simuladores compilados estáticos. Utiliza-se a palavra estática porque toda decodificação é feita antes da simulação,

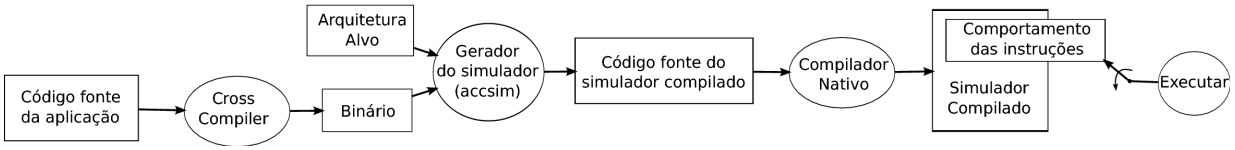

Figura 1. Fluxo do simulador compilado estático

não sendo possível executar aplicativos que sofrem mudanças no código binário *on-the-fly*.

A Figura 1 apresenta o fluxo seguido pelo simulador compilado estático. A aplicação é compilada utilizando um *cross-compiler* (GCC neste trabalho) configurado para gerar um arquivo binário para a máquina alvo. Esse arquivo, juntamente com a descrição da arquitetura, 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 *Program Counter* (`ac_pc`).

Na Figura 2, tem 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 *inline* de todas essas funções, aumentando o desempenho drasticamente. O `ac_pc` é atualizado dentro dessas funções, normalmente em `ac_bhv_instr`, e quando um *branch* 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
```

(a) segmento com 3 instruções MIPS

```
void mips1::Region3(){

  while (1){
     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;
     ...
     }
  }
}
```

(b) Código em C++ equivalente

Figura 2. Tradução de código para 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.

Ao executar uma instrução, um *break* é encontrado, o *Program Counter* é atualizado para a nova instrução e entra no *switch* novamente. Se uma instrução não for de desvio, obviamente a próxima será executada. Então, ao invés de sair do *switch* com

```
void mips1::Execute(...){

  while (!ac_stop_flag){
     switch (ac_pc >> 9){

        case 0: Region0(); break;
        case 1: Region1(); break;
         ...
        case 14: Region14(); break;

        default:
            AC_ERROR(...);
            ac_stop(EXIT_FAILURE);
            break;
     }
  }
}
```

Figura 3. Rotina principal da simulação

o *break,* pode-se fazer uma análise na fase de pré-processamento para retirar os *breaks* e deixar a execução continuar sem causar *overhead.* Para isso, na descrição da arquitetura, é necessário uma informação adicional sobre quais instruções causam desvio. Esta é a primeira otimização presente na técnica FSCS [2]. 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. Para isso, algumas informações adicionais referentes a saltos condicionais e incondicionais devem fazer parte da descrição da arquitetura.

4. Evoluções do *accsim*

Com o lançamento do ArchC 2.0 [16], 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, podendo utilizar TLM (*Transaction-Level Modeling*), por exemplo.

Porém, o *accsim,* nesta versão, ficou sem suporte. Várias estruturas importantes para seu funcionamento deixaram de existir e as informações das descrições da arquitetura foram modificadas. Para oferecer esta abordagem na versão 2.1, foi necessário adaptar o código do *accsim* 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 verão 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 em perda de desempenho.

Durante o processo de adaptação, estudou-se qual seria o melhor caminho e, utilizar várias classes com muitas heranças e *self-references*, 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, portanto, tais recursos 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 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 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 *branch* 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 sofrer *inline.*

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 orientado 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 número da última instrução desviada 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, dando a opção ao projetista de instanciar vários processadores em sua plataforma. Porém, 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 `--multicore` quando executar o *accsim*.

6. Resultados

Com o objetivo de avaliar o simulador compilado estático gerado pela ADL ArchC 2.1, selecionou-se três arquiteturas: MIPS I [6], SPARC V8 [11] e POWERPC [17].

Os aplicativos utilizados são dos *benchmarks* Mediabench [7] e Mibench [9], 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 [3].

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 do FSCS e o *opt2* a segunda. A otimização de postergar o incremento do contador de instruções para um desvio é ativado também pela *opt2*. Apenas na *opt2* utiliza-se a opção `-O3` para ganho de desempenho, deixando a compilação do *base* e do *opt1* mais rápidas (apenas com `-O`). Porém, em todos os casos a opção *inline* é ligada, aumentando o desempenho de execução sem aumentar *overhead* na compilação.

A Figura 4 mostra o resultado dos *benchmarks* para a arquitetura MIPS I. A segunda otimização aliada a otimizações do GCC renderam ótimos resultados,

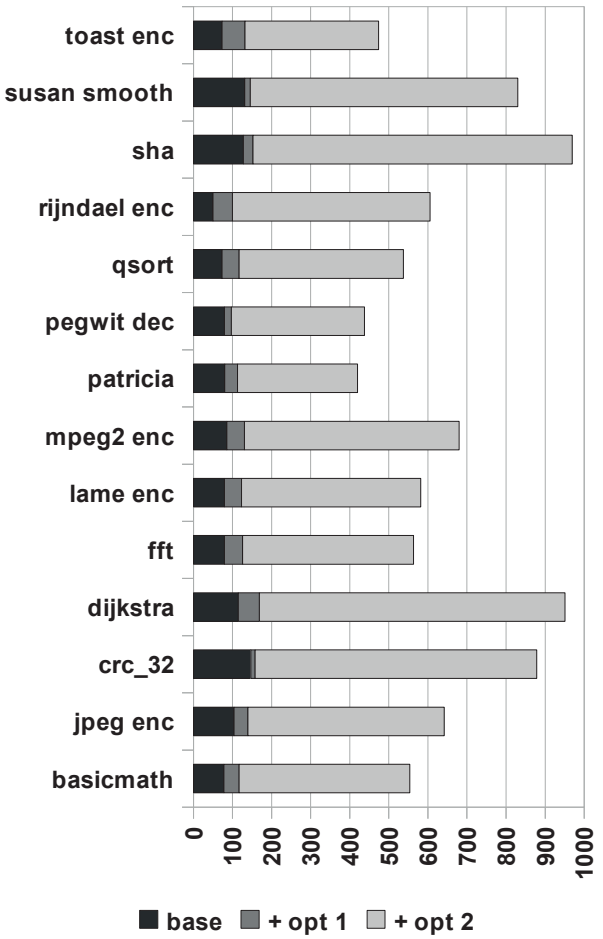

Figura 4. Desempenho do MIPS I (em MIPS)

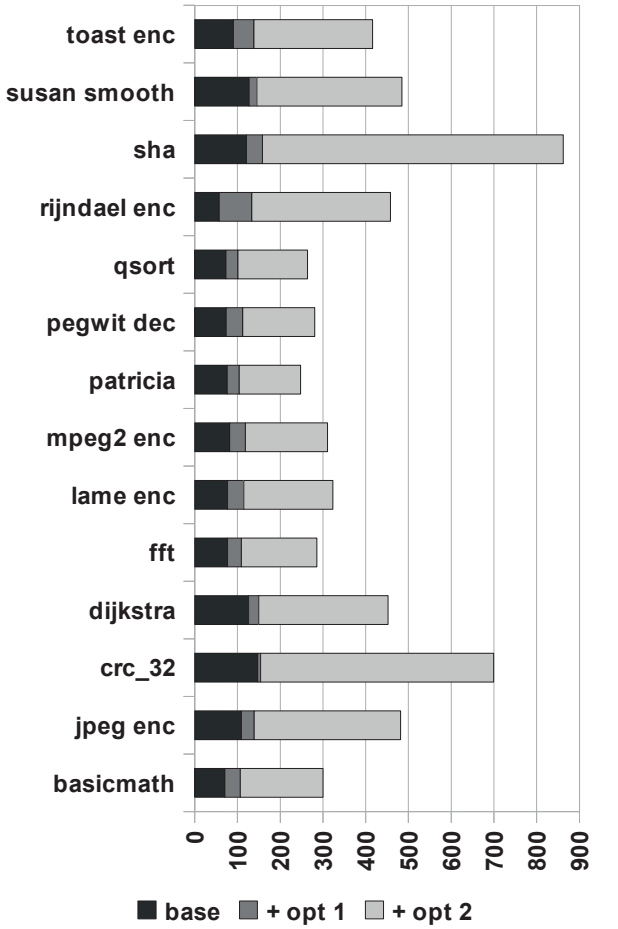

Figura 5. Desempenho do SPARC V8 (em MIPS)

alcançando *speedup* de 423% (para *patrícia*) até 657% (para *sha*) em comparação com o base. A marca de 1KMIPS para o programa SHA quase foi alcançada, sendo o aplicativo que apresentou o melhor desempenho. A média do desempenho ficou em 651 MIPS (milhões de instruções por segundo).

A Figura 5 mostra o resultado para a arquitetura SPARC V8. A média das execuções ficou em 419 MIPS, sendo o SHA e o CRC32 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.

Os testes executados com o POWERPC, exibidos na Figura 6, apresentaram a pior média, 354 MIPS. Isto se deve a complicações na descrição da arquitetura, sendo uma descrição bem mais elaborada que MIPS I e SPARC V8.

Ambas as otimizações (*opt 1* e *2*) geram códigos mais simples, tornando a compilação do simulador mais rápida. Porém, como já dito, o uso da opção `-O3` na segunda otimização, aumenta o desempenho da simulação com um impacto negativo na compilação.

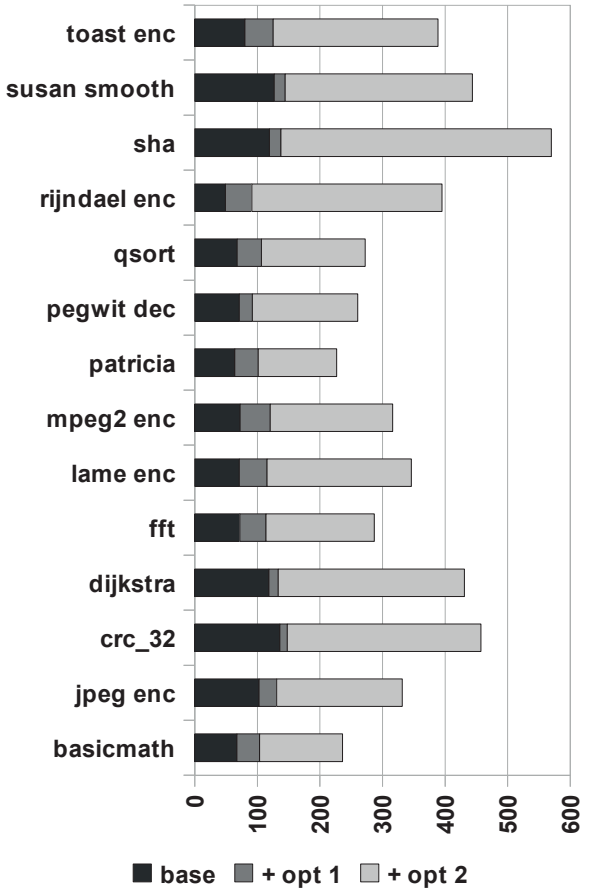

Figura 6. Desempenho do POWERPC (em MIPS)

Na Tabela 2 é exibido o tempo de compilação do simulador MIPS I com alguns aplicativos. Percebe-se que o tempo de compilação cresce linearmente com o número de instruções do aplicativo a ser executado. Em nossos testes, o aplicativo LAME foi o mais demorado a compilar, chegando a 86 segundos no MIPS I, 132 no POWERPC e 138 no SPARC V8. Obviamente, de acordo com a arquitetura modelada, o código gerado pode ser mais complexo que outros, impactando no tempo de compilação.

Aplicação	# de Instruções	base	+opt 1	+opt 2
sha	10454	9,5s	9,6s	20,6s
susan	19794	15,7s	14,6s	34,8s
toast enc	21900	15,9s	15,8s	37,8s
lame enc	61149	38,8s	37,4s	86,8s

Tabela 2. Tempo de compilação de simuladores compilados MIPS I

Testes foram realizados comparando o desempenho dos simuladores compilados em máquina INTEL com os aplicativos compilados nativamente para INTEL. Como os simuladores compilados da arquitetura MIPS foram os mais eficientes, eles foram utilizados para comparação na Tabela 3. Em [2], os experimentos do *accsim* do ArchC 1.6 mostram que o fator de lentidão do ADPCM era de 14,1x para a arquitetura SPARC V8, que foi o modelo com melhor desempenho exposto no artigo. 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 [2], 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 [2] foi de 12.69s contra os 1.86s atual, *speedup* de 582%.

Aplicação	Nativo Intel	Simulador MIPS I	Fator de Lentidão
sha	0,05	0,14	2,8 x
susan	0,10	0,51	5,1 x
crc_32	0,25	0,70	2,8 x
adpcm	0,51	1,86	3,6 x
jpeg enc	0,05	0.17	3,4 x
fft	0,16	27,05	169,0 x

Tabela 3. Código nativo vs tempo de simulação para um *host* INTEL

Dos aplicativos escolhidos, o FFT trabalha com ponto flutuante e isto causou impacto no tempo de simulação. Como os modelos dos processadores apresentados neste artigo não tem instruções de ponto flutuante, para compilar os aplicativos para a máquina alvo, é necessário que a opção `-msoft-float` seja habilitada no compilador *cross-compiler*. Isto faz com que as operações de ponto flutuante sejam executadas em software, ao invés de hardware, diminuindo o desempenho na simulação.

7. Conclusão e trabalhos futuros

Neste trabalho foi apresentado o simulador compilado presente na recente versão da ADL ArchC. Novas informações foram necessárias na descrição da arquitetura para que as otimizações presentes na técnica FSCS fosse implementada. Na primeira, requer a declaração explícita de quais instruções causam desvio e na segunda, precisa de informações suficientes para permitir que o *accsim* consiga controlar o máximo possível do fluxo de execução, gerando simples códigos C++ para melhorar o desempenho da simulação. Evitar o incremento do contador de programas também melhorou o desempenho e permitiu que os recursos fossem encapsulados na classe do processador. Ao ativar estas otimizações e compilar com o último *branch* do GCC 4.4, o desempenho de 969 MIPS para o aplicativo SHA foi alcançado, ficando apenas 2,8 vezes mais lento que uma execução nativa.

Nos próximos trabalhos, um gerador de simulador compilado dinâmico com as otimizações apresentadas será implementado, evitando o *overhead* que o estático possui na compilação e permitindo a execução de aplicativos que modificam seu código binário (*self-modifying code*). Estuda-se, também, utilizar vários núcleos para decodificação e um para simulação, aumentando o ganho de desempenho.

8. Agradecimentos

Agradeço à FAPESP (*Grants* 2009/13230-9) e ao CNPq (*Grants* 134749/2009-0) pelo apoio financeiro recebido para este trabalho e aos estudantes e professores que utilizam o ArchC para estudo ou pesquisa, contribuindo com melhorias e *feedback.*

9. Referências

[1] B. Albertini. “Um framework de desenvolvimento de plataformas e um mecanismo de depuração baseado em reflexão computacional”, *Dissertação de Mestrado*. Instituto de Computação, UNICAMP, 2007.

[2] M. Bartholomeu, R. Azevedo, S. Rigo, G. Araujo. “Optimizations for Compiled Simulation Using Instruction Type Information”, In: *16th Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD, 2004.

[3] M. Bartholomeu, S. Rigo, R. Azevedo, e G. Araujo. “Emulating operating system calls in retargetable isa simulators”. *Technical Report IC-03-29*, Institute of Computing, University of Campinas, December 2003.

[4] J. R. Bell. “Threaded code”. *Commun. ACM,* 1973, pp. 370-372.

[5] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt and A. Nicolau, “EXPRESSION: A language for architecture exploration through compiler/simulator retargetability”, In: *Proceedings of Design Automation and Test in Europe*, DATE, Munich, Germany, 1999, pp. 485-490.

[6] G. Kane and J. Heinrich, “MIPS RISC Architecture”, Prentice Hall, Englewood Cliffs, N.J., 1992.

[7] Mediabench, http://euler.slu.edu/~fritts/mediabench/

[8] J. C. Metrôlho, C. A. Silva, C. Couto, A. Tavares. “A language for automatic generation of fast instruction-set compiled simulators”, In: *International Symposium on Industrial Embedded System*, Le Grande Motte, June, 2008, pp. 111-117,

[9] MiBench, 2010. www.eecs.umich.edu/mibench/

[10] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, and H. Meyr, "A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation", in proceedings of 39th Design Automation Conference, New Orleans, EUA, 2002. pp. 22- 27.

[11] R. P. Paul, “SPARC Architecture, Assembly Language Programing, and C*”,* Prentice Hall, 2000.

[12] S. Pees, A. Hoffmann, V. Zivojnovic, H. Meyr. “LISA – Machine Description language for Cycle-Accurate Models of Programmable DSP Architectures”, In: *Proceedings of the 36th Annual Conference on Design Automation*, 1999, pp. 933 – 938.

[13] W. Qin, J. D'errico, X. Zhu. “A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation”, In: *International Conference on Hardware/software Codesign and System Synthesis* , Seoul, Korea, 2006. pp 193-198.

[14] M. Reshadi, P. Mishra, and N. Dutt. “Instruction set compiled simulation: A technique for fast and flexible instruction set simulation”, In: *Proceedings of Design Automation Conferece,* DAC, 2003.

[15] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. “ArchC: A systemc-based architecture description language”, In: *Proceedings of the XVI Symposium on Computer Architecture and High Performance Computing,* SBAC-PAD, Foz do Iguaçu, Outubro, 2004.

[16] T. M. Sigrist. “Reestruturação de ArchC para integração a metodologias de projeto baseadas em TLM ”, *Dissertação de Mestrado*, Instituto de Computação, UNICAMP, 2007.

[17] E. Silha, “The PowerPC Architecture”, *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, Austin, Tex., 1993.

[18] V. Zivojnovc, A. Ropers, and H. Meyr. “Fast simulation of the TITMS320C54xDSP”. In *Proc. Int. Conf. on Signal Processing Application and Technology,* ICSPAT*,* San Diego, Setembro, 1999. pp. 995-999.