Técnicas de Emulação de Operações de Ponto Flutuante em Sistemas Operacionais Modernos

Lucas C. Villa Real[1] **, Edson T. Midorikawa**[2] **, Marcelo K. Zuffo**[1]

[1] Escola Politécnica da Universidade de São Paulo
LSI - Laboratório de Sistemas Integráveis

{lucasvr,mkzuffo}@lsi.usp.br

[2]Escola Politécnica da Universidade de São Paulo
LAHPC - Laboratório de Arquitetura e Computação de Alto Desempenho

edson.midorikawa@poli.usp.br

Abstract. *The diversity of platforms based on dedicated processors with different computation capabilities has motivated a great effort in the software market to make it possible to use modern operating systems on such devices. These efforts include the adequation and generalization of many internal sub-systems, such as memory and process management and timers. This article presents an analysis of the features available today in low level software to emulate floating point operations on FPU-less hardware, focusing the study on the implementation of the GCC compiler and on the Linux kernel port for ARM processors.*

Resumo. *A diversidade de plataformas baseadas em processadores dedicados e com diferentes poderes computacionais propiciou um grande esforço no mercado de software para tornar possível o seu uso em sistemas operacionais modernos. Estes esforços incluem a adequação e a generalização de sub-sistemas de gerenciamento de memória, processos, timers e diversos outros. Este trabalho apresenta uma análise do suporte existente hoje em softwares básicos para emular operações de ponto flutuante em arquiteturas não dotadas de uma unidade de processamento deste tipo, com um estudo focado no compilador GCC e no kernel Linux para processadores ARM.*

1. Introdução

No mercado de sistemas embarcados, a demanda por computadores dedicados e especializados fez crescer a indústria de sistemas integrados em um único *chip* (SoC) [Lahiri et al. 2000] nos últimos anos. Neste processo surgiram diversas variantes de processadores com diferentes capacidades computacionais, como unidades desprovidas de gerenciamento de memória, DSPs dedicados ou suporte estendido de gerenciamento de energia para lidar com a economia de recursos. Muitos deles apresentaram também variantes quanto à existência ou não de uma unidade de processamento de ponto flutuante (FPU), visto que muitas aplicações dedicadas não lidam com este tipo de dado.

Integram hoje essa variedade de processadores nomes conhecidos como o Alpha, ARM, MIPS, PowerPC, S390 e SPARC, presentes em diversos eletrônicos de consumo como *set-top-boxes*, celulares e terminais de cartão de crédito, entre outros[1]. Garantir o

[1]Na linha x86, os processadores desprovidos de FPU foram apenas o Intel i386/i486SX e o Cyrix 486DLC, que delegavam todo o processamento de ponto flutuante ao co-processador 387.

suporte à essa variedade de processadores sem FPU torna-se, assim, um fator importante para que um sistema operacional moderno se adeqüe às plataformas embarcadas e tenha uma maior aceitação entre seus usuários e desenvolvedores.

Entretanto, ao considerar que a maior parte das linguagens de programação permite o tratamento de dados em ponto flutuante (*reais*), torna-se difícil garantir que qualquer software ou algoritmo que se deseja rodar nesta plataforma seja desprovido de tais instruções – salvo casos nos quais eles tenham sido escritos especialmente para este hardware [Iordache and Tang 2003]. Essa garantia torna-se ainda mais difícil quando o processador prevê, em seu conjunto de instruções, extensões para lidar com ponto flutuante, deixando, entretanto, sua implementação a cargo dos fabricantes de hardware, que podem adotá-las ou não.

O tratamento deste problema é especialmente importante quando se deseja gerenciar a distribuição e a instalação de pacotes binários de software para este tipo de hardware, visto que uma mesma família de processadores pode ter variantes com e sem FPU. Desta forma, torna-se necessário detectar ou simular as instruções de ponto flutuante, evitando a parada do programa que as apresentou devido à execução de uma instrução desconhecida. Este é o ponto principal abordado por este trabalho, que se organiza conforme apresentado na próxima seção.

2. Organização

O artigo, na seção 3, apresenta os processadores ARM e as características dos seus co-processadores de ponto flutuante, usados como referência neste trabalho. Estas seções introduzem termos técnicos e conceitos apresentados no restante do texto e darão uma base para a compreensão do modelo de implementação de unidades de FPU em processadores modernos.

A seção 4 trata de emuladores de ponto flutuante orientados a exceções, disparadas quando o processador decodifica uma instrução não suportada pelo mesmo. Este tipo de evento acarreta a troca de contexto de execução do processador, que passa a rodar no modo privilegiado (de sistema), exigindo que a implementação do emulador se concentre no *kernel* do sistema operacional. A referência para esta análise foi o emulador NWFPE do kernel Linux 2.6.12, com a execução de experimentos sendo feita em um processador ARM XScale (PXA255).

A próxima seção do artigo lida com a solução do problema no modo usuário, na implementação do compilador. Nesta situação, o compilador é instruído a não gerar instruções de ponto flutuante no código objeto, substituindo-as por chamadas para funções internas que as implementam eficientemente através de operações sobre inteiros. Em tempo de execução do programa estas funções são resolvidas através de uma biblioteca dinâmica, criada no processo de instalação do compilador, que se torna uma dependência do executável gerado. A implementação de referência para esta emulação foi o código do compilador GCC 3.4.5 para processadores ARM.

Em seguida, nas seções 6 e 7, são apresentados resultados de desempenho e prototipações para cada uma das metodologias abordadas. Uma análise dos impactos de cada uma delas no sistema acompanha a discussão, mostrando os efeitos colaterais observados e suas origens.

O artigo é então concluído, na seção 8, apresentando um resumo das técnicas abordadas e buscando definir questões de *design* de um sistema operacional que tragam o menor sobrecusto de implementação, gerência e tratamento dos problemas discutidos.

3. A Arquitetura ARM

O ARM é um processador RISC projetado para permitir implementações de hardware muito pequenas e de alto desempenho [Seal 2000]. Em função deste foco, os processadores da sua família geralmente apresentam um baixíssimo consumo de energia, sendo este um dos principais motivos pela sua presença dominante entre os eletrônicos de consumo.

Um processador ARM possui 7 possíveis modos de execução, que regulam e protegem o acesso aos diversos recursos do sistema. São eles:

- *User:* modo normal de execução de programas, executa em modo não-privilegiado;
- *FIQ:* usado no tratamento rápido de interrupções, para transferências de dados de alta velocidade;
- *IRQ:* usado para o tratamento de interrupções de propósito geral;
- *Supervisor:* modo protegido do sistema operacional;
- *Abort:* usado na implementação de memória virtual e/ou na proteção de memória;
- *Undefined:* utilizado para a emulação em software de co-processadores de hardware ausentes;
- *System:* executa tarefas privilegiadas do sistema operacional (disponível na arquitetura ARMv4 e superiores).

Quanto aos seus registradores, a arquitetura disponibiliza um total de 31 para propósito geral. Destes, apenas 16 encontram-se visíveis em um dado instante; o restante é usado para acelerar o processamento de exceções. Estes registradores dividem-se nos seguintes grupos:

- *Unbanked registers:* são os registradores R0-R7. Eles sempre referenciam o mesmo endereço de memória real, independente do modo de execução do processador;
- *Banked registers:* são os registradores R8-R14. Diferentemente dos registradores R0-R7, os endereços físicos referenciados por eles *variam* de acordo com o modo de execução do processador. Isto permite que cada modo de execução mantenha um conjunto de trabalho individual, não compartilhado com os outros modos;
- *Program counter:* associado ao registrador R15.

O conjunto de instruções do ARM é dividido em duas partes: uma responsável pelo *core*, com instruções aritméticas, de condições, saltos, *status*, *load/store*, movimentação de dados, semáforos e de geração de exceções, e outra que define um conjunto de instruções para a conversação com co-processadores, escalável até um máximo de 16. Com isso, a arquitetura ganha flexibilidade, permitindo que ela cresça sem que o conjunto de instruções se torne muito extenso com a adição de instruções especializadas [Sloss et al. 2004].

Assim, fica a cargo de cada co-processador definir seu conjunto de instruções, mantendo a arquitetura de instruções bastante modular. O ARM disponibiliza 5 instruções

como base para este protocolo de comunicação, que permitem realizar a escrita e leitura de registradores do co-processador, operações sobre dados nele armazenados e a movimentação de dados entre o co-processador e registradores do ARM. Entre os aceleradores de ponto flutuante beneficiados por esta arquitetura estão o ARM FPA10, o FPA11 e o VFP, que representam as opções de co-processadores de FPU disponíveis nas implementações de processadores ARM.

A execução de instruções de co-processador funciona com base em um simples protocolo: ao executar uma delas, o ARM aguarda uma confirmação (ACK) vinda do co-processador para que ela possa ser executada. Caso nenhuma resposta seja recebida, uma *trap* de instrução indefinida ocorre, ocasionando a seguinte seqüência de operações:

1. O endereço da próxima instrução é armazenado no registrador R14 no banco do modo *Undefined*;
2. O processador entra no modo *Undefined*;
3. O estado do processador é definido para o modo ARM[2];
4. As interrupções normais são desabilitadas (*fast interrupts*, ou FIQs, podem ocorrer);
5. A execução é desviada para a função definida no vetor de interrupções;
6. A instrução não suportada é emulada;
7. O fluxo da execução continua a partir do endereço salvo previamente.

Nota-se que neste processo as interrupções são desabilitadas; elas geralmente permanecem assim até que o tratador da *trap* decodifique o *opcode* da instrução, chame a rotina pertinente e trate-a. Desta forma, este processo traz uma grande penalidade consigo: o aumento da latência no tratamento de requisições vindas de periféricos integrados à plataforma.

4. Emuladores Orientados a Exceções

O NWFPE (*NetWinder Floating Point Emulator*) é um emulador de operações matemáticas com ponto flutuante para processadores ARM [Bambrough 2006]. Sua implementação segue a especificação IEEE 754 [IEEE Standards Committee 754 1985], dando suporte às operações de ponto flutuante com precisões simples, dupla e dupla estendida. O emulador se situa no *kernel* Linux, funcionando a partir da geração de exceções, conforme a seqüência de operações apresentada na seção anterior.

A rotina configurada no vetor de interrupções do ARM Linux para instruções indefinidas verifica, primeiramente, se existe suporte para emulação da operação solicitada. Caso haja, ela repassa a instrução completa para a função de emulação, que irá processá-la em software através do NWFPE, armazenando o resultado no registrador de destino informado. Neste ponto, o *kernel* repassa a execução para o modo usuário através de uma nova troca de contexto, e o programa pode continuar a executar do ponto onde parou.

O retorno do emulador é feito através de dois possíveis pontos configurados pelo *kernel*. Caso a emulação ocorra com sucesso, a execução é desviada para a função `ret_from_exception()`, e o *kernel* se encarrega de retornar o controle da trap para o código de usuário. Caso o emulador não a tenha emulado com sucesso, ele retorna através da função `fpundefinstr()`, e o *kernel* encerra o processo com um *core dump*.

[2]O ARM tem dois modos de operação: o modo ARM e o THUMB, que operam em 32 e 16 bits, respectivamente.

Na entrada do emulador, o registrador `r10` aponta para uma área privada do processo, onde o *workspace* de ponto flutuante se encontra – é nesta área que o emulador salva seus registradores durante as diversas chamadas do processo. O primeiro *byte* desta área é usada como uma *flag* para detectar a primeira vez que o processo usa ponto flutuante. Com isso, o emulador não precisa iniciar sua máquina de emulação novamente, visto que já existe um contexto salvo e que pode ser restaurado rapidamente.

Para reduzir os custos de troca de contexto, o NWFPE busca por instruções de ponto flutuante subseqüentes à executada, caso ela tenha sido emulada com sucesso. Esse laço de busca e emulação se repete até que seja encontrada uma instrução que não envolva ponto flutuante, fazendo com que a execução retorne para o espaço de usuário. As chances de que haja um agrupamento de instruções de ponto flutuante são grandes, devido aos esforços do compilador GCC para otimizar a emulação.

A emulação propriamente dita é feita a partir da função C `EmulateAll()`, chamada pela sub-rotina `emulate`. Esta função primeiramente verifica o *opcode* e, a partir de sua decodificação, a emulação é direcionada para a rotina do co-processador identificado ou retorna um erro de instrução inválida caso não haja suporte para o co-processador solicitado.

O processo de emulação para uma operação de soma usando precisão simples pode ser resumido pelos seguintes passos, já dentro da função de emulação chamada pela `EmulateAll()`:

1. Verificação de validade do tamanho do operador (simples, duplo ou duplo estendido);
2. Configuração da máscara de arredondamentos, caso tenha sido solicitado;
3. Armazenamento da precisão solicitada;
4. Comparação do tamanho dos operandos, usando o maior para fazer uso de toda a precisão possível;
5. Verificação do tipo de precisão solicitada, encaminhando a emulação para a função correspondente – por exemplo, para a `SingleCPDO()` no caso de emulação simples.

Esta última rotina faz uso de um vetor de ponteiros para funções, indexado pela instrução solicitada. Após passar por mais uma indireção, a função de emulação de soma é finalmente invocada, recebendo os operandos e uma *flag* informando se a soma deve ser negada antes de ser retornada.

O processo de emulação através do NWFPE pode ser acompanhado pela Figura 1, que descreve os blocos principais do emulador.

5. Emuladores Implementados em Compiladores

Para arquiteturas que não suportam ponto flutuante, é possível instruir o compilador GCC para que ele converta as intruções de ponto flutuante em chamadas de função que as simulem. Estas funções são implementadas pelo próprio compilador, e ficam disponíveis na `libgcc` depois que o compilador é instalado. Desta forma, a emulação, antes feita através de exceções e tratada pelo *kernel*, pode ser realizada totalmente em espaço de usuário, sem o sobrecusto de trocas de contexto e decodificações extras de *opcodes*.

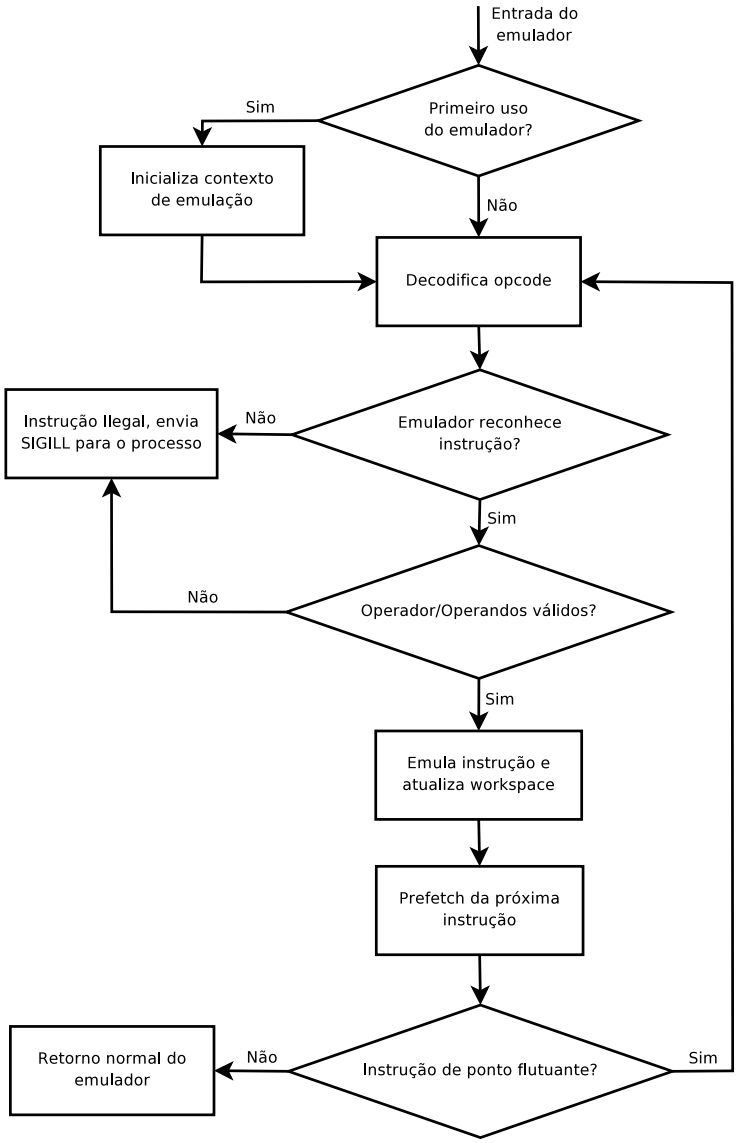

Figura 1. Blocos principais do emulador NWFPE.

Este suporte no GCC para o ARM conta hoje com duas implementações: uma inteiramente feita na linguagem C e uma versão otimizada, escrita em *assembly*. A implementação é escolhida durante a execução do *script* de configuração do GCC, que incorpora à sua compilação fragmentos da máquina alvo. Estes fragmentos são descritos em arquivos texto e permitem, entre outros parâmetros, definir a emulação de ponto flutuante [Stallman 1999].

Ambas as implementações suportam operações com precisão simples e dupla, representando os tipos *float* e *double*, respectivamente. Assim como o NWFPE, a implementação do GCC segue o formato estabelecido pelo IEEE 754 [IEEE Standards Committee 754 1985], exceto para a geração e manipulação de exceções e para arredondamento de valores. Os segmentos de código a seguir ilustram trechos de um executável gerado neste modo, para um programa composto apenas de uma operação de soma entre dois valores de ponto flutuante [Chamberlain et al. 2004].

```
0000848c <main>:
    848c:       e1a0c00d        mov     ip, sp
    8490:       e92dd810        stmdb   sp!, {r4, fp, ip, lr, pc}
    8494:       e24cb004        sub     fp, ip, #4
    8498:       e24dd010        sub     sp, sp, #16
    849c:       e50b0014        str     r0, [fp, #-20]
    84a0:       e50b1018        str     r1, [fp, #-24]
    84a4:       e59f3048        ldr     r3, [pc, #72]
    84a8:       e50b301c        str     r3, [fp, #-28]
    84ac:       e59f3044        ldr     r3, [pc, #68]
    84b0:       e50b3020        str     r3, [fp, #-32]
    84b4:       e51b001c        ldr     r0, [fp, #-28]
    84b8:       e51b1020        ldr     r1, [fp, #-32]
    84bc:       eb0000e4        bl      8854 <__addsf3>
    ( ... )
```

Após iniciar a função `main()`, o programa armazena os dois operandos nos registradores `r0` e `r1`, que em chamadas de função no ARM representam o primeiro e segundo argumentos, respectivamente. Ao invés de apresentar uma operação de ponto flutuante, o executável faz uma chamada à função `__addsf3()`, que irá conter as intruções de emulação correspondentes à implementação selecionada durante a configuração do compilador:

```
00008504 <__adddf3>:
    8504:       e1310003        teq     r1, r3
    8508:       0020c002        eoreq   ip, r0, r2
    ( ... )
```

Desta forma, o único sobrecusto além da emulação propriamente dita é uma chamada de função. Este modo de emulação é muito atraente do ponto de vista de desempenho. Os programas compilados com este recurso, entretanto, não ganham vantagem quando são executados em uma máquina dotada de uma FPU, visto que todos os pontos nos quais haveria instruções deste tipo foram substituídos por chamadas de função para emulá-las através de inteiros.

6. Medições e Resultados de Desempenho

Muitos dos softwares que compõem um sistema operacional orientado ao *desktop* realizam operações em ponto flutuante de alguma forma. Com a exceção de bibliotecas específicas para o processamento de sinais, programas de simulação e de cálculo, a grande maioria as utiliza de maneira bastante simples, envolvendo apenas as operações básicas com o intuito de calcular médias, porcentagens, funções de *hash*, alimentar sementes aleatórias, etc. Esta classe de programas é o alvo das medições de desempenho realizadas, apresentadas logo a seguir.

Para avaliar o desempenho das operações aritméticas emuladas no ARM, foram utilizados *benchmarks* do software LMbench[3] [McVoy and Staelin 1996]. O LMbench consiste em um conjunto de ferramentas de teste básicos do sistema operacional e do

[3]http://www.bitmover.com/lmbench

processador, medindo com bastante precisão dados como latência e largura de banda de aspectos como leitura/escrita de memória, troca de contexto, manipulação de sinais, comunicação na rede, entre outros. Ele é bastante popular entre os desenvolvedores de sistemas operacionais livres, sendo este o outro motivo pela sua escolha para conduzir estes experimentos [Brown and Seltzer 1997].

Dos testes relacionados a ponto flutuante, foram avaliadas as operações de soma, multiplicação, divisão e uma combinação destas (*combo*). As seguintes operações são realizadas pelo laço principal de cada um deles, sendo todas as variáveis do tipo *float* e armazenadas em registradores (`register float`):

- soma: `f += f; f += g;`
- multiplicação: `f *= f; f *= g;`
- divisão: `f = g / f; g = f / g;`
- combo: `x[i] = (1.0f + x[i]) * (1.5f - x[i]) / x[i];`, em um laço expandido que manipula 10 posições do vetor `x` a cada iteração do laço principal. O laço expandido é executado 100 vezes a cada iteração, na tentativa da simulação de aumentar a carga computacional para estimar a quantidade de operações por segundo que o processador é capaz de executar.

Todos os testes foram executados com zero milissegundos de tempo de *warmup*. O ambiente de execução continha apenas *threads* de sistema como processos concorrentes em um sistema mono-usuário (com apenas um processo interativo), no qual cada teste foi executado 50 vezes, com 500 iterações no laço principal.

A plataforma de *hardware* consistiu em um ARM PXA 255, composto por um processador Intel XScale sem unidade de ponto flutuante. O sistema operacional usado para os testes foi um GoboLinux embedded[4], baseado no *kernel* 2.6.12-mm2 e na Glibc 2.3.6. Esta CPU teve sua freqüencia calculada pelo LMbench em 396 MHz, com uma precisão no *clock* de 2,5253 nano-segundos. Outras informações relevantes incluem sua TLB e o tamanho da linha de cache, composta de 32 páginas e 32 bytes, respectivamente.

Os resultados obtidos das execuções realizadas sobre o NWFPE e as implementações do GCC são vistos na Tabela 1, representados em unidades de nano-segundos. Os desvios padrão obtidos ficaram dentro de 0,08 nano-segundos para as operações de soma, divisão e multiplicação na biblioteca do GCC, contra 1,7 nano-segundos no NWFPE. Os desvios constatados para as operações de combo foram de 2,15 e 8,94 nano-segundos para a libgcc e o NWFPE, respectivamente.

Tabela 1. Tempos de execução no XScale, representados em nano-segundos

	Soma	Multiplicação	Divisão	Combo
LibGCC/ASM	182,68	124,75	355,84	866,14
LibGCC/C	650,92	573,94	1013,45	3098,50
NWFPE(Emulação)	700,08	939,05	1382,33	6284,00
NWFPE(Total)	1090,08	1329,50	1772,33	6674,00

A primeira linha representa as medidas obtidas com a implementação em *assembly* da biblioteca do GCC, seguida pela implementação original em C. A terceira linha indica

[4]http://embedded.gobolinux.org

os tempos de emulação para o NWFPE, *desconsiderando* o tempo médio de uma troca de contexto, estimado em 0,39 micro-segundos (390 nano-segundos) através da chamada de sistema `getppid()`. A última linha apresenta o tempo total (real) obtido com o NWFPE, ou seja, *considerando* os tempos de troca de contexto. A Tabela 2 apresenta o ganho de desempenho das estratégias implementadas em compilador em relação ao NWFPE. Este ganho foi calculado pela diferença dos tempos de execução em relação ao tempo total do NWFPE em pontos percentuais. Os valores obtidos mostram a diferença de desempenho em cada uma das implementações.

Tabela 2. Ganhos de desempenho em relação ao tempo total do NWFPE (em %)

	Soma	Multiplicação	Divisão	Combo
LibGCC/ASM	83,24	90,61	79,92	87,02
LibGCC/C	40,29	56,82	42,82	53,57
NWFPE(Emulação)	35,78	29,36	22,00	5,84

Destacam-se nestas tabelas as grandes disparidades entre os modos de emulação. Nota-se, a partir da comparação da versão em C e *assembly* do emulador na LibGCC, que o processo de otimização de código do compilador deixa muito a desejar, mesmo quando configurado para utilizar rotinas especiais do processador alvo. Essa implementação em C, por sua vez, contrasta com a implementação NWFPE do kernel Linux, escrita na mesma linguagem, mesmo quando desconsiderados os tempos de trocas de contexto. Isto explica-se de duas formas: o NWFPE tem o custo extra de decodificação dos *opcodes* da instrução, além de apresentar uma implementação diferente da adotada no GCC.

A última linha da Tabela 2 demonstra qual seria o ganho do emulador NWFPE caso ele estivesse implementado em espaço de usuário. Nela, a melhora apresentada em cada operação (soma, multiplicação, divisão e combo) indica a granularidade do cálculo realizado pelo emulador: quanto menor ela for, maior se torna o impacto das trocas de contexto no tempo total da emulação. A exemplo da emulação de soma em ponto flutuante feita pelo NWFPE, 35,78% do tempo total de execução constitui-se puramente de sobrecusto com trocas de contexto.

Considerando uma implementação em *kernel* da LibGCC otimizada, somente o tempo de trocas de contexto (390 nano-segundos) já seria maior do que o da própria emulação para as operações de soma, multiplicação e divisão; o esforço no porte desta infra-estrutura para o *kernel* acaba por não compensar os possíveis ganhos em relação ao NWFPE.

7. Prototipações

Durante a concepção deste trabalho foi implementada uma camada de instrumentação do NWFPE, permitindo o acesso às estatísticas de seu uso através do sistema de arquivos `sysfs`[5]. O registro, acessível através do arquivo `/sys/nwfpe/counter`, apresenta a quantidade de inicializações do estado da FPU (revelando o número de processos que já utilizaram o emulador) e as instruções simuladas, concentradas em três grupos:

[5]Disponível para download em http://lucasvr.gobolinux.org/arm/2.6.12-nwfpe-sysfs.patch

- `CPDO`: incrementado a cada *opcode* aritmético monádico e diádico encontrado, indicando operações com um e dois operandos, respectivamente.
- `CPDT`: incrementado a cada *opcode* de load/store;
- `CPRT`: incrementado quando um *opcode* de conversão, comparação ou de transferência entre registradores é encontrado.

Através da monitoração deste arquivo foi possível detectar a quantidade de operações envolvidas nas simulações. Ele também permitiu identificar um problema interessante – mesmo quando um programa não fazia uso explícito de ponto flutuante, a quantidade de exceções geradas aumentava durante sua execução.

A razão para esta ocorrência estava na Glibc, que não havia sido compilada com a *soft-float* do GCC. Assim, além de acessarem o estado de registradores usuais da plataforma, as operações `longjmp()` e `setjmp()` também o faziam para os registradores de ponto flutuante[6]. Como esses registradores não existiam na plataforma de hardware utilizada, o acesso a eles era emulado pelo NWFPE no kernel, gerando as emulações reportadas pelo monitor desenvolvido.

Esta emulação, entretanto, é bastante cara. Por serem muito usadas durante o fluxo de execução de um programa[7], a ocorrência de *traps* e trocas de contexto acaba acontecendo em grande número – impedindo que o processador responda eficientemente às interrupções de hardware em função das proteções que envolvem o emulador NWFPE no kernel.

O acompanhamento do arquivo `/sys/nwfpe/counter` indicou que somente no programa `cat` do software BusyBox[8], a emulação do NWFPE é invocada 7 vezes devido a estas rotinas. O comando `ssh --help` exige 38 entradas no NWFPE para acessar os registradores de ponto flutuante, enquanto uma autenticação bem sucedida em um nodo remoto acarreta 435 *traps* para esta emulação. A inicialização do servidor de janelas Xorg 7.0, através do comando `X`, realiza, por sua vez, 54391 acessos deste tipo.

8. Conclusões

Este trabalho analisou o desempenho de duas estratégias alternativas para a implementação do suporte a emulação de instruções de ponto flutuante em processadores sem FPU. Foi utilizado como referência a família do processador ARM, devido a sua grande utilização em sistemas embarcados.

Os resultados obtidos com a emulação de ponto flutuante no ARM mostram que o tratamento de algumas operações aritméticas básicas, quando feitas pelo compilador, pode ser feito em um tempo menor do que o consumido por uma troca de contexto de execução. Isto é especialmente interessante quando se observa a grande quantidade de softwares pré-compilados para esta plataforma que incluem instruções de ponto flutuante, requerendo a existência de um emulador baseado em exceções, como o NWFPE.

A principal contribuição do trabalho foi mostrar que é possível obter uma sensível melhora de desempenho quando a emulação é realizada em modo usuário, com um ganho

[6] Ver glibc-2.3.6/sysdeps/arm/__longjmp.S e arm/fpu/__longjmp.S

[7] As rotinas `longjmp()` e `setjmp()` são utilizadas para efetuar saltos para endereços não locais

[8] http://busybox.net

de desempenho de até 90%. Contudo para usufruir deste resultado é necessário que as aplicações sejam recompiladas de forma a usar a biblioteca de emulação.

Apesar de ser negligenciada em muitos projetos de sistemas operacionais, esta questão mostra-se fundamental quando se pretende garantir um bom desempenho do software frente a essas arquiteturas de processadores dotadas de diferentes capacidades computacionais. Uma má escolha de *design* pode tornar o baixo desempenho perceptível pelos usuários, que poderiam encontrar a resposta para seus problemas em um sistema operacional concorrente.

Como trabalhos futuros, várias alternativas estão sendo estudadas. O primeiro poderia incluir a avaliação de aplicações inteiras que fazem uso de operações de ponto flutuante, como, por exemplo, para o processamento de sinais. Outro trabalho interessante seria uma análise detalhada das rotinas de emulação em C visando estudar pontos de otimização de forma a melhorar o desempenho e se aproximar da versão em *assembly*. Este trabalho também poderia ser realizado no próprio compilador, no *backend* de otimização do código de máquina, na tentativa de melhorar o código gerado para rotinas como as do emulador NWFPE.

Referências

Bambrough, S. (2006). NetWinder Floating Point Emulator. Disponível em http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.17.tar.bz2, acessado em 23 de março de 2007.

Brown, A. B. and Seltzer, M. I. (1997). Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 214–224, New York, NY, USA. ACM Press.

Chamberlain, S., Eichin, M., Wilson, J., Earnshaw, R., and Pitre, N. (2004). LibGCC routines for ARM CPU. Disponível em ftp://ftp.gnu.org/gnu/gcc/gcc-4.0.2/gcc-4.0.2.tar.bz2, acessado em 23 de março de 2007.

IEEE Standards Committee 754 (1985). *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York. Note: Standard 754–1985.

Iordache, C. and Tang, P. T. P. (2003). An overview of floating-point support and math library on the intel "xscale"architecture. In *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, page 122, Washington, DC, USA. IEEE Computer Society.

Lahiri, K., Raghunathan, A., and Dey, S. (2000). Efficient exploration of the soc communication architecture design space. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 424–430, Piscataway, NJ, USA. IEEE Press.

McVoy, L. W. and Staelin, C. (1996). lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294.

Seal, D. (2000). *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Sloss, A., Symes, D., and Wright, C. (2004). *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Stallman, R. M. (1999). *Using and Porting the GNU Compiler Collection, For GCC Version 2.95*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA.