Projeto 3 - Sistemas Multicore e Offload para Hardware

André Nakagaki Filliettaz - RA104595
Gregori Yuji Mandelli Inagaki 105006
Lucas Noriyuki Yabuta 106120

Introdução

O desempenho na execução de tarefas tem sido uma das preocupações da computação desde o seu início. Hoje em dia, todas as máquinas possuem microprocessadores, com altos níveis de desempenho e hardwares modernos extremamente rápidos. Por mais que o poder computacional esteja aumentando, também aumenta a complexidade dos problemas que os computadores devem resolver.

Problemas antes impossíveis, como a decodificação do genoma humano, tem sido resolvidos com computação paralela, e outros problemas ainda mais complexos esperam para serem estudados.

Quando a técnologia utilizada para desenvolver microprocessadores começou a esbarrar nos limites físicos (microprocessadores estavam alcançando temperaturas elevadas demais, tornando-se não confiáveis), visando o aumento de performance, surgiu o que se denomina multiprocessador!

Os multiprocessadores nada mais são do que chips que contém mais do que um core (núcleo de processamento). Individualmente, esses núcleos tendem a ter um desempenho inferior aos tradicionais microprocessadores, mas sua grande vantagem advém do processamento paralelo! Ao invés de se investir em processadores mais rápidos, hoje as empresas de tecnologia investem em processamento paralelo.

Para este projeto, foi proposto a modelagem de um sistema com até oito cores, que executariam um software que tiraria proveito de todo o paralelismo nele. Além disso, um módulo em hardware, especializado em executar algum trecho de software, deveria ser desenvolvido (software offloading). Software Offloading não é algo incomum em computação, e já havia sido usado previamente em unidades responsáveis pelo cálculo de ponto flutuante (FPU)[2]. Com isso deve-se observar e medir o impacto que o processamento paralelo e software offloading tem sobre o desempenho do processador na execução do programa.

Objetivos

Buscou-se medir o aumento de desempenho que cada core, juntamente com o módulo de software offloading, trazia para os tempos de execução finais do processador. Para realizar os testes, foi utilizado um programa que implementava a fórmula de Bailey–Borwein–Plouffe (Algoritmo de BBP)[3] para o cáuculo dos dígitos de π.

Com o uso desse algoritmo simples, foi possível obter-se uma quantidade muito boa de resultados. Para a unidade de Software Offloading, foi implementado um hardware que realizava de maneira independente o calculo de potências.

Metodologia

Construindo o Sistema

Medir o impacto dessas mudanças no processador exigiu que fossem feitas modificações nos códigos na plataforma ARP.

O código fornecido era capaz de executar o programa Hello World de forma simples, e contava apenas com um processador e uma memória, que se comunicavam através do padrão TLM. Como o TLM pressupõe uma comunicação ponto a ponto por portas, na qual uma delas deve ser a mestre a outra deve ser escrava da mesma, era impossível simplesmente instanciar multiplos processadores e fazer com que os mesmos acessassem a memória ao mesmo tempo.

Para que os multicores tivessem acesso, foi necessária a criação de um barramento de memória. O barramento funciona de forma a ser escravo dos processadores, mas mestre da memória, servindo para realizar as requisições

de acesso dos processadores a memória. O barramento possui oito portas que se comunicam com os cores e uma porta que se comunica com a memória.

Isso resolvia o problema de um sistema com multiplos processadores, entretanto, gerou-se um outro problema, comum a processamento paralelo, no caso, a concorrência e disputa por acesso a um dado. Sempre que houvesse uma escrita na memória, haveria também um potencial risco, já que poderiamos ter problemas de multiplas escritas na mesma posição de memória.

O problema de sincronização é muito comum em processamento paralelo, e os processadores MIPS fornecem duas instruções, **load-linked** e **store-conditional** (LL/SC), ambas utilizadas para garantir esse controle. Se um processador realiza uma instrução LL, ele carrega o que está naquele endereço de memória, e o endereço de memória então é marcado. Uma instrução subsequente de SC só escreverá naquela posição se não houver nenhuma alteração naquele endereço de memória desde que a instrução LL marcou a posição.

Obtém-se assim uma operação de read-modify-write que é livre de locks e atômica (ela não pode ser interrompida, ou ela escreve por inteiro ou não escreve nada). Essas instruções não constavam no ISA da plataforma, de maneira que também foram implementadas.

Obtendo as Medidas

Para obter uma medida de quanta melhora o processamento paralelo trás para a execução do programa, foi contabilizada a quantidade de ciclos que o simulador utilizava para executar todo o programa. O tempo final de execução($\mathbf{T}_f$) do programa pode ser visto como:

$T_f = N_c * T_c$ *(1)*

Aonde $\mathbf{N}_c$ e $\mathbf{T}_c$ são, respectivamente, o número total de ciclos gasto pelo processador para executar o programa e o tempo por ciclo. A melhora no desempenho (**i**) pode ser vista como quantas vezes menor o tempo de execução em paralelo é da execução serial. Assim:

$i = \frac{T_f 1}{T_f 2} = \frac{N_c 1 * T_c 1}{N_c 2 * T_c 2}$ *(2)*

Considerando que todos os cores são idênticos, podemos dizer que o tempo de ciclo de todos os core é igual, de forma que podemos simplificar a expressão para:

$i = \frac{N_c 1}{N_c 2}$ *(3)*

Logo, o programa foi executado em três sistemas diferentes. Um simples (**Sistema A**), com apenas um core e sem nenhum offloading (sem barramento de dados, o processador é ligado diretamente com a memória), outro com quatro processadores e um offloading (**Sistema B**) e por fim um terceiro sistema com todos os oito cores e sistema de offloading (**Sistema C**).

Foram obtidas então medidas da quantidade de ciclos gastos nos 3 sistemas. Utilizando a equação (3) estimou-se qual o peso do multicore e software offloading no desempenho do sistema.

Observa-se que o número de ciclos gastos no caso em que o sistema possui multicores é sempre o do processador que gastou mais ciclos, pois a execução só terminaria quando este core terminar, fazendo todos os outros esperarem por ele.

Por números de ciclos foi considerado a execução do pipeline de 5 estágios, com bypassing e branch predicor (saturating counter com distinção de instruções), e no caso os efeitos da cache foram desconsiderados, bem como os efeitos de segunda ordem. O código utilizado para os cores foi o do projeto anterior.

Análise de Dados

Depois da execução do programa, arquivos auxiliares foram gerados aonde estavam marcados o total de ciclos gastos por todos os cores que estavam executando o programa, contabilizandos junto com os stalls que eram inseridos no pipeline para execução do programa.

Com os dados destas execuções a tabela a seguir foi montada:

Tabela 1: Ciclos por Core

core	**Sistema A**	**Sistema B**	**Sistema C**
1	383589	17215	7946
2	-	18423	9417
3	-	19295	10312
4	-	24971	10652
5	-	-	11307
6	-	-	11386
7	-	-	11427
8	-	-	14668

Desta tabela, muitas coisas podem ser observadas. Primeiramente, nota-se que o total de ciclos gastos (apresentado na Tabela 2, a seguir) difere grandemente entre os sistemas. Em especial, são gastos no mínimo 4 vezes mais ciclos para se executar o programa de maneira serial. Uma das possíveis explicações seria a divisão de recursos por parte do Sistema A, já que ele possui menos recursos de hardware e precisa gerenciá-los mais. Além disso, o Sistema A não possui unidade de Software Offloading, o que fazia com que o processador tivesse que realizar esse cálculo sozinho.

Tabela 2: Total de Ciclos por Sistema

Sistema A	**Sistema B**	**Sistema C**
383589	79904	87115

Além disso, considere que o tempo gasto para a execução do programa tende a ser determinado pela equação *(1)*. No caso do processamento paralelo ideal, aonde não existem depêndencias de dados entrem os cores, e aonde todos os cores executam instruções a todo ciclo (nenhum core entra em estado *idle*), o tempo de processamento gasto por um sistema paralelo será determinado apenas pelo core que utilizou mais ciclos. Dessa forma, os ciclos máximos de execução são apresentados pela *Tabela 3*.

Tabela 3: Número Máximo de Ciclos e Melhora de Desempenho

Sistema A	**Sistema B**	**Sistema C**
383589	24971	14668
Improvement	15.36	26.15

Como esperado, o Sistema C é o que apresenta melhor desempenho, justamente por tirar muito mais proveito do paralelismo e do software offloading.

Conclusão

Este projeto acabou por unir grande parte daquilo que havia sido trabalhado em projetos anteriores. Em especial, para ser possível contabilizar as diferentes execuções, foi utilizado grande parte do projeto 2.

O controle de concorrência em hardware não é feito de maneira simples. Em especial, em todos os Sistemas, haviam multiplos cores (sem nenhum sistema de cache) que acessavam uma única memória. Haviam duas saídas para evitar uma concorrência, ou instalar um lock na posição de memória, ou implementar instruções especiais, que serviriam para limitar o acesso dos cores a escrita em determinada posição (as instruções **ll** e **sc**). A segunda abordagem acabou por economizar recurso em hardware, pois não foi necessário criar um sistema de lock para o barramento.

Isso se provou um grande desafio, pois a inserção de duas instruções no simulador não foi uma tarefa trivial, e muitas coisas a respeito do funcionamento da plataforma ArchC tiveram de ser entendidas. O funcionamento da instrução, seu tipo, os registradores que usava, os registradores auxiliares incluídos, tudo isso teve de ser entendido e manipulado.

Com o auxilio do processador desenvolvido no projeto anterior obteve-se a quantidade de ciclos gastos por cada processador. Muitas coisas podem ser observadas daí. Primeiro de tudo, os processadores não dividem o workload de forma igualitária. Um core era sempre responsável por mais trabalho do que os outros, de forma que gasta muito mais ciclos que os outros. Isso é normal, já que um dos cores deve dividir o trabalho entre todos os outros.

Além disso, o Sistema que utiliza menos ciclos no total não necessariamente é aquele que possui o menor tempo de execução. O Sistema B possui a menor quantidade de ciclos totais, no entanto, os cores executam aproximadamente o dobro dos ciclos do Sistema C individualmente. Isso faz com que o tempo de execução do Sistema C seja menor que a metade do tempo de execução do Sistema B.

Por fim, podemos analisar a influência do Software Offloading. No caso desse projeto, foi criada uma unidade que calculava a potência de um determinado número.

É razoavel pensar (mesmo considerando uma distribuição desigual de workloads) que os ciclos gastos por B sejam 4 vezes menor do que A, já que este possui 4 cores idênticos aos de A trabalhando em paralelo. Entretanto não é esse comportamento o observado, o tempo gasto é bem menor do que o esperado. Essa melhora pode ser atribuida a essa unidade de offloading. Em especial, as funções expoênte consomem grande quantidade de processamento, pois são implementadas como laços, o que em nível de hardware equivalem a condicionais e desvios.

Com uma unidade fazendo isso de maneira separada, os cores podiam fazer outras tarefas, de forma que o processo final acabou-se por acelerar de maneira significativa.

Tendo como enfoque o desempenho dos Sistemas Computacionais, o projeto provou que o Paralelismo é sem dúvida o próximo caminho a ser dado para o aumento de desempenho. Hoje em dia, o processamento paralelo está presente em todos os tipos de sistema, desde grandes centros computacionais, a smartphones. Apesar da grande dificuldade intrínseca ao desenvolvimento de um sistema que opere de forma eficiênte em paralelo, os resultados obtidos aqui mostram de maneira clara que paralelismo é uma solução compensadora.

Bibliografia

1 http://oxent.ic.unicamp.br/sites/oxent.ic.unicamp.br/files/ch1$_0$.*pdf*

2 http://www.princeton.edu/ achaney/tmve/wiki100k/docs/Floating$_p$*oint*$_u$*nit.html*

3 http://mathworld.wolfram.com/BBPFormula.html