Projeto 3^+
Processador Multicore

Gabriel Borges
RA: 116909
Grupo 5

30 de junho de 2015

Sumário

Professor: Rodolfo Jardim de Azevedo
MC723 - Laboratório de Projeto de Sistemas Computacionais
Turmas A e B

1 Introdução

O objetivo deste projeto é desenvolver e estender o funcionamento de um processador de múltiplos núcleos. Para isso, se utilizou o simulador ArchC (uma linguagem de descrição de arquiteturas de processadores baseada na linguagem SystemC) para simular a execução de um programa relevante na arquitetura de processador MIPS, com o intuito de observar o funcionamento da arquitetura *multicore*, bem como de constatar os ganhos atingidos com o módulo de *harware* criado para acelerar o desempenho do sistema. Finalmente, foi também necessário implementar em *hardware* recursos que permitissem a execução concorrente de trechos críticos de código.

2 Programa de Testes

O programa de testes utilizado tinha o objetivo de calcular, com a melhor precisão possível, uma aproximação para o número π. Para tanto, nos valemos das seguintes relações:

$$\pi = 4 * atan(1) \tag{1}$$

$$atan(x) = \int_0^x \frac{1}{1+x^2} dx \rightarrow \pi = 4 * \int_0^1 \frac{1}{1+x^2} dx \tag{2}$$

O método usa somas de *Riemann* com intervalos definidos pelo usuário para fazer o cálculo da aproximação para a integral. Em sua versão *multithread*, o programa subdivide essas tarefas entre os processos, somando os subcálculos para chegar no valor final ao término da execução.

O código foi adaptado para certas especificidades do projeto de `http://cs.calvin.edu/curriculum/cs/374/homework/threads/01/pi.c`.

3 Decisões de projeto

Para corretamente projetar e analisar a arquitetura *multicore* do processador, consideramos os seguintes fatores:

3.1 Controlador de processadores

Como já evidenciado no roteiro inicial deste projeto e empregado no sistema apresentado, é comum que, num regime *multicore* de processadores, assim que houver *boot* no sistema ou *hard reset*, os núcleos sejam inicialmente inicializados e colocados num estado à espera do sistema operacional. O *bootstrap processor* (processador a carregar o sistema operacional em primeira instância) então começa a executar instruções, que eventualmente farão uso dos demais processadores. Quando isso ocorre, este ou estes sai(em) do modo de espera e é(são) colocado(s) à executar instruções definidas.

Neste projeto, a plataforma de intercâmbio entre os núcleos *per se* e o sistema operacional é um controlador de processadores, representado por um periférico chamado *cores_controller*. A comunicação entre o periférico e o sistema é feita através de três interfaces:

- A função `number_of_cores` retorna a quantidade de núcleos do processador disponíveis ao sistema operacional;
- A função `is_core_on` recebe um índice de núcleo do processador (esses índices são naturais de 0 a `number_of_cores()`-1), e retorna se o núcleo explicitado está ou não ativo (ou "ligado");
- A função `set_core` recebe um *status* (*on/off*) e um índice de núcleo do processador, e associa ao processador correspondente àquele índice o *status* recebido. Essa função é utilizada para "ligar" e "desligar" núcleos quando se fizer necessário.

Em termos de implementação na arquitetura *ArchC*, o controlador foi feito de maneira análoga à memória fornecida no repositório original do projeto[1]. Desta sorte, seu funcionamento pode ser resumidamente explicado da seguinte forma:

1. Requisições tratadas pelo *bus* através de seu método `transport` que outrora eram inevitavelmente direcionadas ao `transport` equivalente na memória agora podem ser levadas ao do referido controlador, de acordo com a posição do endereço acessado;

2. Na lógica do controlador, essas requisições podem ser tanto de leitura (caso das funções `number_of_cores` e `is_core_on`) como de gravação (caso da função `set_core`). Outros mecanismos de controle são empregados para determinar qual a função desejada, bem como seus argumentos;

3. A requisição é atendida (especificamente, nos módulos disponíveis em `cores_controller.cpp`) e os efeitos colaterais, realizados (caso de ser necessário que núcleos sejam ligados/desligados);

4. Para o sistema operacional, a maneira de acessar tais funcionalidades é de ler de (caso de requisições de leitura) ou gravar em (caso de requisições de gravação) endereços específicos em memória. Essas solicitações são então levadas até o controlador de processadores, no processo descrito acima.

3.2 Controle de concorrência

O controle de concorrência foi implementado em *hardware*. Com o intuito de manter a coerência com operações da arquitetura *MIPS*, as instruções implementadas foram as seguintes:

- **Load Linked** (`ll`): Instrução do tipo I, de *opcode* 30_{hex}.
  Faz a operação `R[rt] = M[R[rs]+SignExtImm]` (idêntica à instrução `lw`), e que seta um campo especial com o valor `R[rt]` (doravante denominado `history`);

- **Store Contiditional** (`sc`): Instrução do tipo I, de *opcode* 38_{hex}. Faz as seguintes operações:
  a) `M[R[rs]+SignExtImm] = R[rt]`; *b)* `R[rt] = R[Rt] == history ? 1 : 0`.
  A operação *"a)"* é idêntica à `sw`, e o item *"b)"* é o que determina se a operação é atômica, registrando essa condição em `R[rt]`.

Com as instruções `ll` e `sc` disponíveis em *hardware*, podemos implementar as operações da biblioteca `pthread` utilizadas no *software* de teste. As funções implementadas são as seguintes:

- `pthread_create`: cria uma nova *thread*, se possível, e destina um processador a trabalhar nela. Também recebe o ponteiro para uma função – a execução do processador designado deve iniciar nessa função. Sua implementação verifica se existe processador ocioso, e em caso afirmativo, associa a ele a *thread* requisitada;

- `pthread_join`: aguarda até que a *thread* correspondente tenha terminado sua execução (se isto ainda não tiver ocorrido), interrompendo a execução da *thread* atual até que a condição se concretize. Sua implementação interrompe a execução até que o processador correspondente esteja ocioso;

- `pthread_mutex_init`: inicializa o *mutex* correspondente, permitindo que ele seja bloqueado ou desbloqueado;

- `pthread_mutex_lock`: operação de bloqueio do *mutex* (aguarda até que o *mutex* esteja liberado, quando o bloqueia e faz uso do recurso crítico). Implementado com o seguinte loop:
  `while (load_linked(mutex) || !store_conditional(mutex, 1));`

- `pthread_mutex_unlock`: operação de desbloqueio do *mutex*. Meramente altera o valor da chave para zero.

[1] disponível em `/home/staff/rodolfo/mc723/base_platform.git`.

3.3 *Hardware Offloading*

Ao analisar a estrutura do código principal de testes, `pi.c`, é evidente perceber que o gargalo na execução está no loop principal da função `computePI()`. Portanto, essa foi a seção em que se fez o *hardware offloading*.

Para tanto, criamos mais um componente do dispositivo, de maneira similar à criação do controlador de processadores, descrito acima. A função desse dispositivo seria de efetuar o cálculo da aproximação para π, para uma *thread* específica, de acordo com o número de *threads* total e o número de intervalos de cálculo.

Foi criada a função `offload_arctan()` com esse objetivo. Se o *offloading* não está ativo, a computação segue em software conforme esperado. Em caso contrário, cada *thread* escreve em um arquivo os parâmetros necessários para o cálculo da função desejada e chama uma operação de leitura, que sinaliza ao sistema no *hardware* que o arquivo gravado está disponível para leitura, e que a *thread* está aguardando a resposta. O componente em *hardware*, por sua vez, lê os argumentos do arquivo, efetua o cálculo desejado e grava o resultado noutro arquivo, que é, por fim, lido novamente pelo programa principal.

4 Resultados

É uma propriedade da soma de *Riemann* que quanto maior o número de intervalos utilizados para aproximar uma integral, maior será a precisão da aproximação. Também é esperado que quanto maior o número de processadores utilizados para o desempenho da tarefa, mais rápida deve ser sua execução, descontado o *overhead* de criação e sincronização dos processos. Temos a seguinte tabela com os resultados experimentados:

Tabela 1: Performance do programa de testes

Offload	T	I	E	Tempo gasto (s)
Não	1	100	5.6	0.54
Sim	1	100	5.6	1.67
Não	1	1000	7.6	4.93
Sim	1	1000	7.6	1.64
Não	2	1000	7.6	3.76
Não	7	1000	7.6	2.87
Não	7	5000	9.0	14.09
Sim	1	5000	9.0	1.65
Sim	1	10^5	11.6	1.67
Sim	1	10^6	13.6	1.67
Sim	1	10^7	15.6	1.83
Sim	1	10^8	15.7	3.40
Sim	2	10^8	15.7	5.78
Sim	1	10^9	15.3	19.03

Legenda:

- **Offload**: Determina se o *hardware offload* foi utilizado;
- **T**: Número de *threads* utilizadas no cálculo. Esse número deve variar entre 1 e 7, pois uma thread é utilizada para a sincronização dos dados;
- **I**: Número de intervalos;
- **E**: $-log_{10}(\pi_{medido}/\pi)$, aproximado para uma casa decimal. Quanto maior este valor, mais precisa é a aproximação;
- **Tempo gasto (s)**: Tempo "real" medido em segundos, aproximado para duas casas decimais.

Os resultados nos permitem tomar as seguintes conclusões:

- Há um certo significativo *overhead* para todos os modos. Na versão sem *offload*, ele começa a perder importância a partir de $I \approx 1000$. Já na com offload, ele parece deixar de se manifestar para $I \approx 10^8$;
- A influência da paralelização no modo sem *offload* não é diretamente proporcional à redução do tempo gasto: de fato, os programas são executados em menos tempo, mas o fator de redução é bem menor que o de aumento no número de núcleos;
- A introdução do *offload* permite reduções extremamente significativas nos tempos de execução do programa;
- Como o hardware de *offload* não é paralelizado, não é vantajoso paralelizar a execução do programa de testes em modo paralelo (já que todo o seu *hot spot* foi transportado para o hardware).

5 Conclusão

Os objetivos do projeto foram plenamente alcançados: o controlador de processadores está completamente funcional, o controle de concorrência foi implementado segundo as especificações e a implementação de código em *hardware* foi realizada. As metas atingidas nos permitiram fazer as seguintes observações:

- Ficou bastante clara uma simples e eficiente maneira de inicializar processadores *multicore*, recorrendo à criação de um periférico para o gerenciamento das chamadas de suporte aos núcleos, e encarregando o sistema operacional de definir os momentos de inicialização dos núcleos além do principal (*bootstrap processor*);
- Uma etapa anterior deste projeto envolveu uma implementação das rotinas de `lock` e `unlock` em *software*, sem recorrer a instruções atômicas. Embora funcional, a técnica era bastante lenta e com alto *overhead*. Ficava evidente a necessidade da implementação desse recurso em *hardware*. Uma vez tendo se alcançado esse objetivo, pudemos perceber a mudança em performance das rotinas de teste. Finalmente, uma vez que fizemos *wrappers* para funções da biblioteca `pthread`, transportar outras rotinas de teste que também utilizem a biblioteca para o sistema *MIPS* desenvolvido deve ser pouco trabalhoso, se requerir algum trabalho;
- A implementação dos "gargalos" da rotina em *hardware* nos permitiu observar ganhos consideráveis no desempenho da aplicação – devido aos longos tempos de espera na configuração *multithread* sem *offload*, a comparação entre os modos não é necessariamente estatisticamente relevante. Se extrapolando os resultados observados, no entanto, notamos ganhos entre $8x$ e $148,000x$. Uma vez não tendo o processador sido concebido para realizar as operações características do "gargalo", a concepção de um componente em *hardware* com essa função efetivamente elimina o *overhead* experimentado e torna a computação extremamente rápida. Exemplos de dispositivos reais que fazem uso dessa estratégia são *GPUs*: essas resolvem uma área de problemas que *CPUs* são muito menos eficientes em resolver. Isso se deve à diferença arquitetural dessas unidades de hardware.

Link para o repositório com o código fonte do projeto: `https://github.com/borgesgabriel/mc723_p3`.