Atenuando a Contenção nas Unidades de Execução com Mapeamento Instruction-Aware*

Matheus S. Serpa[1], Eduardo H. M. Cruz[2], Matthias Diener[3], Antonio C. S. Beck[1], Philippe O. A. Navaux[1]

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970, Porto Alegre – RS – Brasil

{msserpa, caco, navaux}@inf.ufrgs.br

[2]Instituto Federal do Paraná (IFPR)
Paranavaí – PR – Brasil

eduardo.cruz@ifpr.edu.br

[3]Universidade de Illinois em Urbana-Champaign
Champaign, IL, Estados Unidos

mdiener@illinois.edu

Resumo. *Aplicações paralelas executadas em processadores SMT (Simultaneous Multithreading) competem por unidades de execução. O problema fica ainda pior, quando as threads executam instruções semelhantes, como por exemplo de ponto flutuante, inteiro, load e store. Nesses casos, o mesmo tipo de instrução é despachado para execução, o que leva a perdas de desempenho devido a contenção nessas unidades. Este trabalho tem como objetivo fornecer um mecanismo para mapeamento de múltiplas aplicações paralelas em processadores SMT. O mecanismo foca em melhorar o desempenho, mitigando a contenção nas unidades de execução ao executar aplicações paralelas. Para tanto, threads que estressam as mesmas unidades de execução são mapeadas em núcleos diferentes. Os resultados mostram ganhos de desempenho de 29,1% e 17,4%, em média, quando comparado com o escalonador do sistema operacional Linux e com um mapeamento Round-robin.*

1. Introdução

A maior parte das arquiteturas multicore atuais implementa alguma forma de *Simultaneous Multithreading* (SMT). O SMT permite que *threads* independentes executem instruções simultaneamente para várias unidades de execução, melhorando a utilização e o desempenho dos recursos [Tullsen et al. 1995]. A principal vantagem do SMT é que ele pode ocultar a latência de acesso à memória, aumentando a taxa de transferência de execução no núcleo de execução. No entanto, há casos em que essa execução simultânea

*Este trabalho foi parcialmente financiado pelos projetos Petrobras 2018/00263-5 e GREEN-CLOUD: Computação em Cloud com Computação Sustentável (#16/2551-0000 488-9), da FAPERGS e do CNPq, programa PRONEX 12/2014. Os experimentos apresentados neste artigo foram realizados utilizando o ambiente de testes do Grid'5000, sendo desenvolvido no âmbito da ação de desenvolvimento INRIA ALADDIN, com o apoio do CNRS, RENATER e várias universidades, bem como outras organizações de financiamento (mais informações em https://www.grid5000.fr)

causa contenção de recursos (ou seja, várias *threads* competem pelos mesmos recursos), o que pode prejudicar o desempenho geral.

Pesquisas anteriores identificaram a contenção na memória principal e nas memórias *cache* dos processadores SMT como um dos principais gargalos de desempenho [Akturk and Ozturk 2019, Choi and Yeung 2009, Cruz et al. 2018, Feliu et al. 2016, Serpa et al. 2019a, Serpa et al. 2019b]. Assim, utilizando múltiplas aplicações sequenciais, os autores propõem técnicas para mitigar esses efeitos, melhorando o desempenho. Neste trabalho, mostramos a importância de abordar também a contenção nas unidades de execução, principalmente quando se considera aplicações paralelas em vez de aplicações sequenciais. Esse tipo de contenção acontece quando as *threads* de uma mesma ou de outras aplicações emitem instruções de tipos semelhantes (por exemplo, inteiro, acesso à memória, ponto flutuante) que usam as mesmas unidades de execução. Portanto, uma nova metodologia para mapeamento de *threads*, considerando a contenção da unidade de execução, é a chave para melhorar ainda mais o desempenho dos processadores SMT.

Nesse sentido, este trabalho propõe um mecanismo para mapear múltiplas aplicações paralelas, considerando a contenção que ocorre em processadores SMT. Esse mecanismo baseia-se na análise dos padrões de instruções das *threads* e, em seguida, mapeia aquelas que estressam as mesmas unidades de execução em núcleos diferentes. Ao fazer isso, este trabalho apresenta as seguintes contribuições: (i) Proposta do SMT-Bench, um *microbenchmark* que estressa unidades de execução específicas para avaliar seu impacto no compartilhamento de recursos; (ii) Proposta de um mecanismo para mapear múltiplas aplicações paralelas com base nos padrões de instruções das *threads*; (iii) Mostramos que nosso mapeamento baseado nas instruções pode obter ganhos de desempenho de 29,1%, em média, quando comparado com o escalonador nativo do sistema, executando múltiplas aplicações de HPC (*High Performance Computing*) em arquiteturas *multicore* com suporte a SMT.

2. Trabalhos Relacionados

Muitos trabalhos relacionados identificaram a contenção de recursos em processadores SMT como um gargalo de desempenho. No entanto, eles se concentraram apenas na comunicação e nas memórias *cache*. Diferente deles, também consideramos unidades de execução, como *branch*, inteiro e ponto flutuante.

Akturk et al. [Akturk and Ozturk 2019] propõem um escalonador que mapeia *threads* de uma forma que minimiza o número de acessos a *cache* L1 e reduz o número de *evictions* em *caches* compartilhadas que eventualmente limitam o desempenho. Eles melhoram o desempenho do *benchmark* PARSEC [Bienia 2011] em até 12,6%. Apenas a contenção nas *caches* foi levada em consideração, o que não abrange as aplicações CPU-bound que estressam as unidades de pontos flutuantes, por exemplo.

Choi et al. [Choi and Yeung 2009] propõem uma abordagem que se concentra em otimizar o desempenho total do sistema. Em tempo de execução, eles observam o impacto que as decisões de distribuição de recursos têm no desempenho. Eles melhoram o desempenho de cargas de trabalho multiprogramadas em até 11,5%. Nesse trabalho, os autores não levam em consideração aplicações paralelas.

Feliu et al. [Feliu et al. 2016] propõem um escalonador *bandwidth-aware* para processadores SMT. Eles usam o *Instructions Per Cycle* (IPC) para estimar a largura de

banda da *cache Last level cache* (LLC) e da memória principal. Seu escalonador melhora o desempenho do SPEC CPU 2006 [Henning 2006] em até 6,7% em relação ao escalonador do Linux. No entanto, os autores não levam em consideração operações de ponto flutuante e não avaliam aplicações paralelas como o benchmark NAS.

A maioria das propostas anteriores concentra-se em múltiplas aplicações sequenciais. Existe uma carência de literatura que estude e proponha técnicas para mitigar a degradação de desempenho causada pelo compartilhamento de diferentes unidades de execução na execução de aplicações paralelas.

3. Impacto da Contenção nas Unidades de Execução

Nesta seção, mostramos o impacto sobre o desempenho ao compartilhar diferentes tipos de unidades de execução em processadores SMT. Inicialmente, apresentamos o *microbenchmark* que desenvolvemos para estressar unidades de execução específicas, demonstrando que o tipo de operação que cada *thread* executa afeta o desempenho e a contenção. Em seguida, mostramos que o mapeamento de *threads* que compartilham os mesmos recursos em um núcleo causa contenção, reduzindo o desempenho, e que é melhor mapeá-las em diferentes núcleos.

3.1. SMT-Bench: Um Microbenchmark para Avaliar o Compartilhamento de Recursos em Processadores SMT

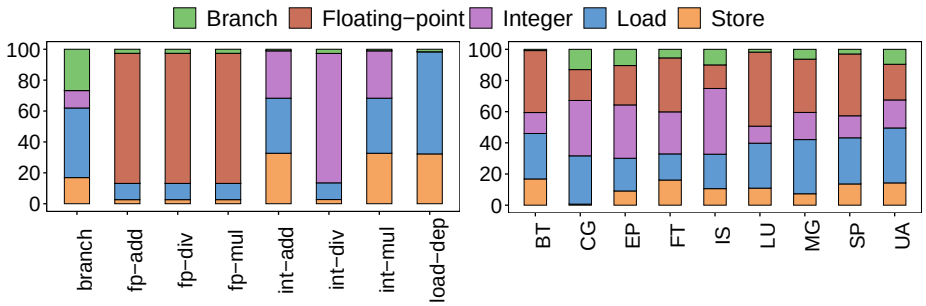
Em um processador SMT, as unidades de execução e as memórias *cache* são compartilhadas entre as *threads* executadas em um mesmo núcleo. Mesmo sendo transparente para as aplicações, na camada de *hardware*, há um número limitado de recursos. Quando duas *threads* precisam usar a mesma unidade de execução, o escalonador de *hardware* tem que decidir entre instruções de diferentes *threads*, indicando qual executará primeiro e qual aguardará. Para avaliar melhor o impacto do compartilhamento de recursos nos processadores SMT, desenvolvemos o SMT-Bench[1], que é um conjunto de *kernels* que estressa unidades de execução específicas.

Desenvolvemos o SMT-Bench principalmente devido à dificuldade de medir a influência da execução de *threads* com o mesmo tipo de instruções em diferentes cenários: *benchmarks* existentes enviam várias instruções que estressam diferentes unidades de execução. Com o SMT-Bench, podemos analisar os padrões de instrução e depois o desempenho com diferentes estratégias de mapeamento.

SMT-Bench utiliza o *Performance Application Programming Interface* (PAPI) [Johnson et al. 2012], uma ferramenta que fornece acesso a vários contadores de *hardware* do processador. A distribuição de instruções de cada um dos oito *kernels* do SMT-Bench é ilustrada na Figura 1a. Na Figura, o eixo y indica a distribuição de instruções de cada *kernel*, enquanto o eixo x mostra o nome dos *kernels*. Estes *kernels* foram escolhidos por representarem várias unidades de execução de um processador SMT. As barras empilhadas indicam o tipo de instrução.

O *kernel* `branch` executa as construções `if-else` e `switch-case` a fim de criar um comportamento de salto. Como podemos ver na Figura 1a, 26,8% de suas instruções são *branches*. `fp-add, fp-div, e fp-mul` são três *kernels* que estressam as unidades de ponto flutuante executando 32 operações independentes dentro de um

[1] `https://github.com/msserpa/SMT-bench`



(a) *Kernels* do SMT-Bench. **(b) Aplicações do NPB.**

Figura 1. Distribuição de instruções para diferentes aplicações.

loop, com um número reduzido de operações de memória, controle e dependências de dados, permitindo um *throughput* próximo ao ideal. Nestes *kernels*, 84,2% das instruções são do tipo ponto flutuante. Em seguida, `int-add, int-div e int-mul` estressam as unidades de inteiro em diferentes proporções, variando de 30,7% a 83,8%. Finalmente, o *load-dep* executa um *loop* que percorre uma lista encadeada, esperando que cada *load* seja completado antes de iniciar o próximo. Este *kernel* estressa tanto a unidade de *load* quanto a de *store*.

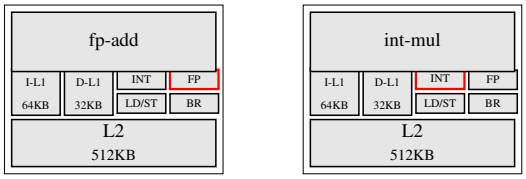
Também analisamos o padrão de instruções do NAS Parallel Benchmarks (NPB) [Bailey 2011], mostrado na Figura 1b. Nosso mapeamento *instruction-aware* pode usar esta análise para decidir onde cada *thread* deve ser mapeada. A aplicação BT tem a maioria de suas instruções como ponto flutuante e *loads*. Mapear aplicações como LU e SP para o mesmo núcleo de BT degradariam o desempenho, já que estas aplicações também se concentraram em ponto flutuante. Entretanto, se a mapearmos IS e CG, aplicações que se concentram em operações sobre inteiros, isso melhoraria o tempo total de execução do sistema.

A maioria das instruções de MG são *loads* e ponto flutuante. UA apresenta características de acesso aos dados de forma aleatória. Mapear estas aplicações no mesmo núcleo degradaria o desempenho de ambas aplicações. Para MG e UA, aplicações tais como EP e FT apresentariam o melhor desempenho. EP e FT concentram-se em inteiros e em ponto flutuante, respectivamente. Estas aplicações executam um baixo número de instruções de *load* em comparação com MG e UA, características que as identificam como boas candidatas a ser mapeadas no mesmo núcleo.

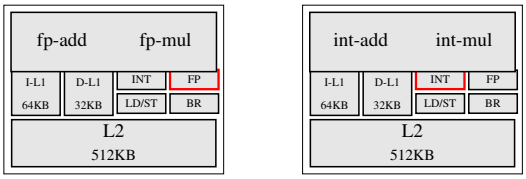
3.2. Análise da Degradação de Desempenho

A análise da degradação de desempenho é realizada utilizando o SMT-Bench com três cenários diferentes. Para ilustrar como a contenção de diferentes unidades de execução influencia o desempenho da aplicação, a Figura 2 mostra três possibilidades de mapeamento em processadores baseados em SMT, como a arquitetura AMD Zen:

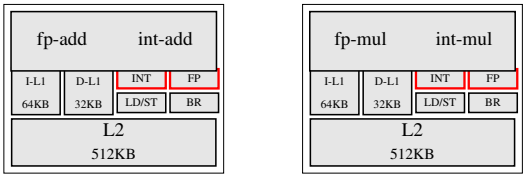
Cenário A: Uma única *thread* executa em um núcleo (Figura 2a). Este cenário evita a interferência de co-aplicações, todas as unidades de execução estão totalmente disponíveis.



(a) Uma *thread* executando em cada núcleo.



(b) Duas *threads* em cada núcleo competindo pelas mesmas unidades de execução.



(c) Duas *threads* em cada núcleo usando unidades de execução diferentes.

Figura 2. Diferentes cenários avaliados.

Cenário B: Duas *threads* executam no mesmo núcleo, estressam as mesmas unidades de execução (Figura 2b) e têm que compartilhar o maior número possível de unidades de execução a fim de equilibrar o *throughput* de cada *thread*.

Cenário C: Duas *threads* executam no mesmo núcleo, estressam unidades de execução diferentes (Figura 2c) e têm as unidades do núcleo alocadas dinamicamente entre as duas *threads*, dependendo do tipo de carga de trabalho que cada uma está executando.

Esperamos que os cenários A e C, que têm mais unidades de execução disponíveis, sejam menos prejudiciais do que o cenário B, onde a contenção devido ao uso das mesmas unidades de execução pode causar execução sequencial. Além disso, os cenários B e C executam o dobro do número de *threads* em comparação com o cenário A.

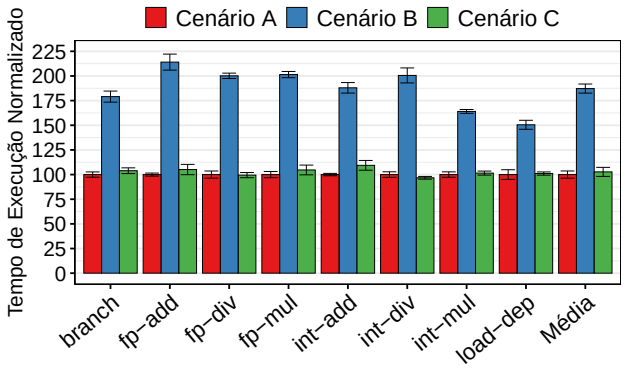


Figura 3. Tempo de execução para os diferentes cenários normalizados para o cenário A.

A Figura 3 mostra a degradação de desempenho devido à contenção de recursos em diferentes unidades de execução. Na Figura, o eixo x mostra os *kernels*, enquanto o

eixo y indica o tempo de execução normalizado ao cenário A. A média geométrica para os cenários A, B e C foi de 100%, 187,2% e 102,7%, respectivamente. Os resultados seguiram a mesma tendência para todos os *kernels*. O desempenho no cenário B é o pior devido à contenção nas unidades de execução. Além disso, o cenário C, onde executamos o dobro de *threads* do cenário A, tem quase o mesmo desempenho por *thread*. O cenário A tem o melhor desempenho. Entretanto, ele limita o número de aplicações e *threads* que o sistema pode executar. O cenário B é o pior, uma vez que, neste caso, as *threads* que estressam as mesmas unidades são mapeadas para o mesmo núcleo, aumentando a contenção da unidade de execução. O cenário C mostrou ser o melhor cenário, no qual aplicações que estressam diferentes unidades são alocadas no mesmo núcleo. Diferentemente do cenário A, no cenário C, conseguimos executar duas *threads* por núcleo e não apenas uma. A partir destes resultados, podemos fazer várias observações:

- O tipo de operação que cada *thread* executa em um núcleo afeta diretamente o desempenho do núcleo e da aplicação como um todo.
- Embora tenhamos o melhor desempenho por *thread* no cenário A, ele permite apenas a execução de uma *thread* por núcleo, o que não é desejável em sistemas baseados em SMT (ou qualquer outro tipo de núcleo *multithread*).
- O cenário C apresenta o mapeamento *thread-to-core* mais eficiente, com uma perda de desempenho por *thread* próxima de 0 e utilizando todos os núcleos virtuais disponíveis, fazendo uso de todo o poder de computação.
- O cenário B tem uma degradação de desempenho de até 120%, o que é inaceitável para aplicações que exigem alto desempenho. Isso nos mostra que o tipo de instrução que cada *thread* executa é um fator que deve ser considerado para mapeamento em processadores SMT.

Portanto, devemos nos esforçar para gerar um mapeamento de *thread* para núcleo o mais próximo possível do cenário C para reduzir a contenção de unidades de execução e, assim, melhorar o desempenho das aplicações. Os resultados com o Benchmark NAS serão apresentados na Seção 5.

4. Mapeamento *Instruction-Aware* para Aplicações Paralelas

O mapeamento proposto neste artigo é realizado em cinco etapas. A Figura 4 detalha essas etapas. Primeiro, detectamos os padrões de instrução das aplicações usando contadores de *hardware*. Esta etapa é realizada executando cada aplicação uma vez antes da execução real. Posteriormente, diferentes aplicações e combinações de topologias de arquiteturas podem ser executados e mapeados. Na segunda etapa, coletamos informações sobre a topologia da máquina e calculamos o mapeamento. Finalmente, as *threads* são mapeadas e as aplicações são executadas. Todas as etapas foram implementadas pré-carregando a biblioteca no binário (via `LD_PRELOAD`) e, portanto, nenhuma modificação do código-fonte é necessária para usar nosso mecanismo.

4.1. Detectando o Padrão de Instruções

Como discutimos na Seção 3, dentro de um processador SMT, *threads* mapeadas para o mesmo núcleo competem por recursos. Nosso mecanismo requer a detecção do tipo de instruções que cada *thread* executa porque melhoramos o desempenho mapeando *threads* que estressam a mesma unidade de execução em núcleos diferentes. A primeira etapa

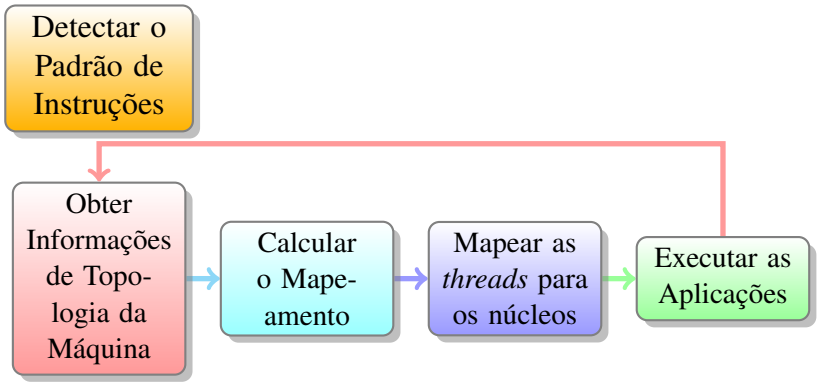


Figura 4. Etapas usadas para realizar o mapeamento.

do nosso mecanismo detecta o padrão de instrução de cada aplicação. Usamos a Performance Application Programming Interface (PAPI) [Johnson et al. 2012], uma ferramenta que fornece acesso a vários contadores de *hardware* do processador, como o número de instruções de cada tipo. O PAPI separa as instruções em de *branch*, *floating-point*, *integer*, *load*, e *store*. Esta etapa é realizada apenas uma vez para cada aplicação. Depois disso, as outras etapas podem ser repetidas várias vezes para diferentes aplicações e combinações de arquiteturas.

4.2. Coletando Informações de Topologia da Máquina

A segunda etapa do nosso mecanismo é detectar a topologia da máquina, que depende do tipo de arquitetura, número de processadores, núcleos e *threads* SMT. Para fazer isso, usamos o hwloc [Broquedis et al. 2010], uma ferramenta que reúne várias informações de topologia hierárquica, incluindo nós *non-uniform memory access* (NUMA), *sockets*, *caches* compartilhadas, núcleos e *simultaneous multithreading*.

4.3. Calculando o Mapeamento

Desenvolvemos um algoritmo de mapeamento para múltiplas aplicações paralelas executadas em processadores SMT. O algoritmo equilibra a carga nos núcleos e unidades de execução usando o padrão de instrução obtido na etapa 1, considerando diferentes tipos de instrução. Como mostramos na Seção 3, o cenário C, onde *threads* que estressam diferentes unidades de execução são mapeadas no mesmo núcleo, apresenta o melhor desempenho por *thread*. Nesse sentido, desenvolvemos um algoritmo baseado na teoria da soma de Gauss que equilibra o número de instruções de um mesmo tipo que cada *thread* em um núcleo executa de forma que a soma seja quase a mesma para todos os núcleos. O algoritmo é mostrado no Algoritmo 1, e recebe como entrada um vetor representando os núcleos da máquina, um vetor contendo as aplicações e o número de instruções de um determinado tipo, o número de núcleos e *threads*. Nosso algoritmo primeiro define o número de *threads* em execução em todos os núcleos como zero. Depois disso, classificamos as *threads* pelo número de instruções de um tipo e, em seguida, colocamos no mesmo núcleo uma *thread* que tem o número máximo e uma que tem o mínimo de instruções. Isso se repete até que todas as *threads* sejam mapeadas.

4.4. Mapeando as *Threads*

Após detectar o padrão de instruções, reunir a topologia da máquina e calcular o mapeamento, a etapa final do nosso mecanismo é executar a aplicação com o mapeamento de-

Algorithm 1: Algoritmo de Mapeamento para Processadores SMT.

```
   Input: cores[], nCores, loads[], nt
   Output: map[]
 1 begin
 2     for i ← 0; i < nCores; i ← i+1 do
 3         cores[i].nt ← 0;
 4     sortThreadsByInstructionPattern(loads);
 5     i ← 0;
 6     head ← 0;
 7     tail ← nt - 1;
 8     while head <= tail do
 9         map[head] ← cores[i].virtualCore[cores[i].nt];
10         head ← head + 1;
11         cores[i].nt ← cores[i].nt + 1;
12         map[tail] ← cores[i].virtualCore[cores[i].nt];
13         tail ← tail − 1;
14         cores[i].nt ← cores[i].nt + 1;
15         i ← i + 1;
16     return map
```

terminado. Para fazer isso em plataformas reais, precisamos de algum suporte do sistema operacional. O Linux fornece a chamada de sistema `sched_setaffinity` para definir a afinidade de uma *thread*. Nenhuma modificação no código-fonte é necessária porque desenvolvemos uma ferramenta que faz uso da variável de ambiente `LD_PRELOAD` para injetar a biblioteca ao carregar a aplicação. A ferramenta executa todas as etapas anteriores e, em seguida, chama `sched_setaffinity` para mapear cada *thread* para o respectivo núcleo.

5. Resultados de Desempenho dos Mapeamentos

Primeiro apresentamos a avaliação de desempenho da nossa abordagem. Depois disso, mostramos que aplicações diferentes, ou seja, aplicações que executam um tipo diferente de instrução, têm melhorias de desempenho superiores usando nosso mapeamento. Em nossa avaliação, usamos o tempo de execução normalizado como uma métrica de desempenho. Além disso, para cada experimento, duas aplicações foram executadas juntas, cada um com uma *thread* por núcleo. Como as aplicações possuem diferentes tempos de execução, mostramos o desempenho baseado no tempo da aplicação mais lenta.

Os experimentos foram realizados no *cluster* `chiclet` do Grid 5K [Bolze et al. 2006]. Cada nó do `chiclet` é composto por dois processadores AMD EPYC 7301, onde cada processador consiste em 16 núcleos físicos, permitindo a execução de 64 *threads* com SMT [Tullsen et al. 1995]. Ele está executando a versão 4.19 do *kernel* Linux. Informações sobre a topologia de *hardware* foram coletadas usando a ferramenta hwloc [Broquedis et al. 2010].

Como carga de trabalho, usamos o SMT-Bench descrito na Seção 3 e a implementação OpenMP do NAS Parallel Benchmarks [Bailey 2011], v3.4. Os resultados foram obtidos por meio da média de 30 execuções aleatórias com acesso exclusivo à máquina. Os resultados comparam o desempenho obtido com a nossa solução aos obtidos com o escalonador do Linux e com o Round-robin. Normalizamos os resultados

do tempo de execução ao escalonador do Linux. Barras menores nas figuras indicam melhores resultados de desempenho.

Calculamos o desvio padrão usando a distribuição t-Student com um intervalo de confiança de 95%. Selecionamos o tamanho de entrada B do NPB, com o objetivo de fornecer um tempo de execução viável. Também investigamos outros contadores de *hardware* utilizando o PAPI [Terpstra et al. 2010].

Comparamos nosso mecanismo com o escalonador do Linux e o Round-robin. O atual escalonador do Linux, o *Completely Fair Scheduler* (CFS) [Pabla 2009], mapeia *threads* dando a cada *thread* uma parcela igual do poder de processamento da CPU. Se um único processo estiver em execução, ele receberá 100% do processador. Com dois processos, cada um teria exatamente 50%. Portanto, isso seria justo para todas as tarefas em execução. Ele não leva em consideração nenhuma informação sobre a carga de trabalho, como o padrão de instrução, como fazemos. O mapeamento Round-robin distribui as *threads* para os núcleos na ordem de 0 ao número de núcleos virtuais menos 1. Para duas aplicações, Round-robin mapeia *threads* da mesma aplicação no mesmo núcleo.

5.1. Resultados de Desempenho de *Benchmark* NAS

Nosso mapeamento *instruction-aware* usa similaridade das instruções para decidir onde mapear as *threads*, devido a isso, separamos a discussão de acordo com o tipo de instrução que é predominante na execução de cada aplicação. A Figura 5 mostra o grupo de aplicações que mais executam operações de ponto flutuante, BT, FT, LU e SP, onde as instruções de ponto flutuante representam de 34,6% a 39,8% do total de instruções. Essas aplicações podem ter melhor desempenho quando colocadas no mesmo núcleo que aplicações que se concentram em inteiros ou instruções de *load*, por exemplo.

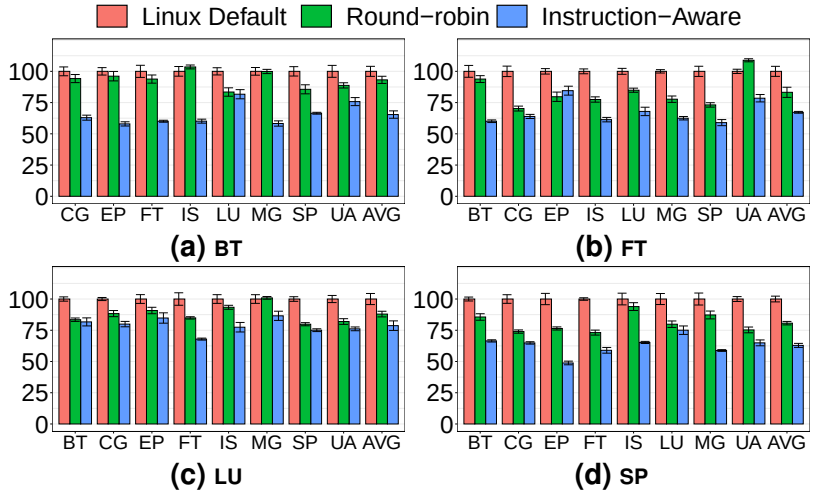


Figura 5. Tempo de Execução do Sistema Normalizado (%) Executando Aplicações do NAS que se Concentram em Instruções de Ponto Flutuante.

Mostramos o tempo de execução obtido com cada uma dessas aplicações rodando com diferentes co-aplicações na Figura 5. A melhoria de desempenho do BT é em média 34,6% em comparação com o escalonador do Linux e 27,8% em comparação com Round-robin. Quando a co-aplicação é a LU, que também executa várias operações de ponto flutuante e *load*, o desempenho é apenas 1,8% melhor que Round-robin. Nesse caso, as aplicações compartilham as mesmas unidades de execução e, portanto, o mapeamento

delas no mesmo ou em núcleos diferentes resulta em quase a mesma melhoria de desempenho. Porém, quando a co-aplicação é o IS, uma aplicação em que a maioria de suas operações são inteiros, o desempenho é 43,4% melhor. Isso mostra que o tipo de operação que cada *thread* executa é um fator chave ao mapear aplicações paralelos.

As instruções de LU e SP são principalmente de ponto flutuante. Esperamos as melhorias maiores quando as aplicações que estressam unidades de execução diferentes são mapeadas no mesmo núcleo. Quando IS é a co-aplicação de LU, o desempenho do nosso mecanismo é 22,6% e 16,0% melhor do que o escalonador do Linux e o Round-robin. Isso acontece porque o IS executa instruções de inteiros na maioria das vezes. O oposto acontece quando BT é a co-aplicação de LU. Nesse caso, a contenção não é reduzida porque ambas as aplicações estressam as mesmas unidades. Para SP, os melhores resultados ocorrem quando as instruções da co-aplicação têm baixa similaridade com o padrão de instruções de SP. Por exemplo, EP foca em instruções sobre inteiros, enquanto, SP, em ponto flutuante. Nosso mecanismo melhora o desempenho em 51,2% em comparação com escalonador do Linux e 27,6% em comparação com o Round-robin quando a contenção nas unidades de execução é reduzida. Temos resultados semelhantes para FP, que está no mesmo grupo de aplicações, com melhorias de 32,8% em comparação com o escalonador do Linux e 16,0% em comparação com Round-robin.

O segundo grupo de aplicações é formado por CG, EP e IS, aplicações que executam operações de inteiros em 35,5%, 34,2% e 42,2% de suas instruções. Mostramos o tempo dessas aplicações na Figura 6. Para CG, o tempo de execução do sistema é melhorado em média em 32,7% em comparação com escalonador do Linux e 12,1% em comparação com Round-robin. Para co-aplicações que compartilham as mesmas unidades, como IS, o ganho é baixo. Isso acontece porque tanto CG quanto IS apresentam uma alta proporção de instruções de inteiros e, portanto, qualquer mapeamento ainda estressa as unidades de inteiros. Para BT, que executa principalmente instruções de ponto flutuante, o desempenho é melhorado em 37,1% quando comparado com o escalonador do Linux e 31,3% em comparação com o mapeamento Round-robin. Os ganhos em BT acontecem porque nosso mecanismo reduz a contenção em unidades de ponto flutuante executando no mesmo núcleo uma aplicação com predominância de operações inteiros.

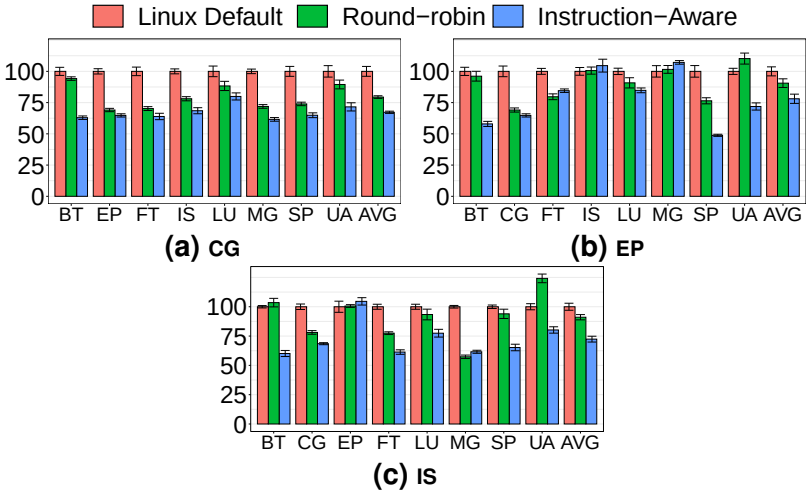


Figura 6. Tempo de Execução do Sistema Normalizado (%) Executando Aplicações do NAS que se Concentram em Instruções de Inteiro.

O mesmo comportamento ocorre para EP e IS, aplicações que também executam várias instruções de inteiro. Para IS, a melhoria média de desempenho de nosso mecanismo é 27,6%. Já para EP, quando a co-aplicação é, por exemplo, SP, uma aplicação que possui um pequeno número de instruções de inteiro, o desempenho do nosso mecanismo é 51,2% melhor que o escalonador do Linux e 27,6% melhor que o Round-robin.

MG e UA formam o terceiro grupo, onde a maior parte das instruções é de *load*. Apresentamos os resultados desse grupo na Figura 7. Para MG, nosso mecanismo melhora o tempo de execução do sistema em 28,4% e 17,6% em média, em comparação com escalonador do Linux e Round-robin. Enquanto para UA, nosso ganho é de 25,5% em comparação com o escalonador do Linux e 25,0% em comparação com Round-robin. Quando a co-aplicação é SP, uma aplicação que também realiza vários *loads*, nosso mecanismo melhora o desempenho em 35,1% em comparação com o escalonador do Linux e 10,4% com Round-robin. No entanto, quando a aplicação estressa um tipo diferente de unidade de execução, como IS, que se concentra em instruções de inteiros, o desempenho é 44,0% melhor do que com o mapeamento Round-robin.

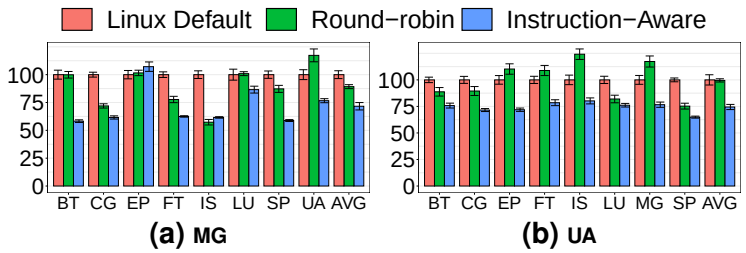


Figura 7. Tempo de Execução do Sistema Normalizado (%) Executando Aplicações do NAS que se Concentram em Instruções de *Load*

6. Conclusão e Trabalhos Futuros

Espera-se que o efeito da contenção da unidade de execução cresça em processadores futuros, onde um número crescente de núcleos continuará compartilhando diferentes unidades de execução. Dessa forma, técnicas de mapeamento como as que apresentamos, terão papel fundamental para futuras arquiteturas.

Neste artigo, apresentamos um *microbenchmark* para avaliar o impacto do compartilhamento de recursos. Além disso, diferentemente do estado da arte, que se concentrava apenas na contenção de memória, propusemos uma técnica para mapear *threads* de várias aplicações paralelos em arquiteturas multicore baseadas em SMT.

Os resultados mostraram melhorias de desempenho de até 51,2% em comparação com o escalonador do Linux e 44% para o Round-robin. Essas melhorias vêm da redução na contenção das unidades de execução, as quais nosso mecanismo foca em diminuir, mapeando *threads* que estressam as mesmas unidades de execução em núcleos diferentes.

Como trabalho futuro, pretendemos investigar o impacto da contenção das unidades de execução em outras arquiteturas, como AMD Rome e Intel Kabylake.

Referências

Akturk, I. and Ozturk, O. (2019). Adaptive thread scheduling in chip multiprocessors. *International Journal of Parallel Programming*, 47(1):1–31.

Bailey, D. H. (2011). Nas parallel benchmarks. *Encyclopedia of Parallel Computing*, 1(1).

Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University.

Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., et al. (2006). Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications*, 20(4):481–494.

Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186, Pisa, Italy. IEEE.

Choi, S. and Yeung, D. (2009). Hill-climbing SMT processor resource distribution. *ACM Transactions on Computer Systems*, 27(1):1–47.

Cruz, E. H., Diener, M., Serpa, M. S., Navaux, P. O. A., Pilla, L., and Koren, I. (2018). Improving communication and load balancing with thread mapping in manycore systems. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*.

Feliu, J., Sahuquillo, J., Petit, S., and Duato, J. (2016). Bandwidth-aware on-line scheduling in SMT multicores. *IEEE Transactions on Computers*, 65(2).

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Johnson, M., McCraw, H., Moore, S., Mucci, P., Nelson, J., Terpstra, D., Weaver, V., and Mohan, T. (2012). Papi-v: Performance monitoring for virtual machines. In *International Conference on Parallel Processing Workshops*.

Pabla, C. S. (2009). Completely fair scheduler. *Linux Journal*, 2009(184):4.

Serpa, M. S., Cruz, E. H., Diener, M., Krause, A. M., Navaux, P. O., Panetta, J., Farrés, A., Rosas, C., and Hanzich, M. (2019a). Optimization strategies for geophysics models on manycore systems. *The International Journal of High Performance Computing Applications*, 33(3):473–486.

Serpa, M. S., Moreira, F. B., Navaux, P. O., Cruz, E. H., Diener, M., Griebler, D., and Fernandes, L. G. (2019b). Memory performance and bottlenecks in multicore and gpu architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–236. IEEE.

Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*. Springer.

Tullsen, D. M., Eggers, S. J., and Levy, H. M. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95. ACM.