UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

CÁSSIO CHAVES MELLO

IMPACTO DA HIERARQUIA DE MEMÓRIA NO DESEMPENHO E CONSUMO ENERGÉTICO DE APLICAÇÕES PARALELAS EM SISTEMAS EMBARCADOS E DE PROPÓSITOS GERAIS

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Antônio Carlos S. Beck Filho
Co-orientador: Me. Arthur Lorenzon

Porto Alegre
2014

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a minha família que sempre me apoiou e fazem parte importante dessa conquista.

Agradeço também ao meu orientador Prof. Dr. Antônio Carlos S. Beck Filho e meu co-orientador Me. Arthur Lorenzon pela ótima orientação que tive durante todo este ano de Trabalho de Graduação, estando sempre presentes e dispostos a me ajudar.

Agradeço à UFRGS, funcionários e colaboradores pela excelente infra-estrutura e ao corpo docente pelo conhecimento obtido nestes 5 anos de faculdade. Agradeço também ao Ciências sem Froteiras pela oportunidade de intercâmbio e ao PET Computação pela experiência obtida.

Ademais, agradeço aos colegas do Laboratório de Sistemas Embarcados (LSE-UFRGS) por terem me acolhido, pela grande ajuda prestada e pelos momentos de lazer. Aos servidores do LSE e do BALI, agradeço imensamente por terem trabalho dia e noite nas simulações, sem eles não seria possível a realização deste trabalho.

RESUMO

A necessidade por desempenho afeta tanto processadores de propósitos gerais quanto de sistemas embarcados, uma vez que as aplicações de ambos tendem a se tornar cada vez mais complexas. Por outro lado, o consumo energético de tais processadores também deve ser levado em conta, principalmente para os sistemas embarcados que, muitas vezes, dependem de bateria para operar. Os processadores *multicore* surgiram neste contexto e da necessidade da exploração do paralelismo em outras granularidades, trazendo consigo uma hierarquia de memória projetada para suprir as necessidades das aplicações paralelas. Cada aplicação pode ser atribuída a um grupo de aplicações conforme seu nível de compartilhamento de dados. Cada grupo fornece diferentes implicações tanto no desempenho quanto no consumo energético das aplicações. Assim, este trabalho mostra que, para aplicações com uso intensivo do processador e baixo compartilhamento de dados, não há grande impacto em relação à organização da hierarquia de memória utilizada pelo processador. Por outro lado, aplicações com alto compartilhamento de dados dependem diretamente da hierarquia de memória – por consequência, a microarquitetura do processor perde bastante importância. A diferença entre compartilhar dados através da memória principal ou através de uma memória *cache* compartilhada pode causar uma diferença de 23% no desempenho de uma aplicação e 30% em seu consumo energético. Ademais, os resultados mostram que o processador Intel Atom N2600 (representando os processadores embarcados) obteve melhor *speedup*, consumo de energia e EDP com o aumento no número de *threads* em aplicações com alto compartilhamento de dados quando comparado com o processador Intel Core i7 860 (representando os processadores de propósitos gerais). Dessa forma, este trabalho mostra que, para este tipo de aplicações com um grande número de *threads*, a hierarquia de memória é mais importante que o processador.

Palavras-chave: Hierarquia de Memória, Programação Paralela, Processadores Embarcados e de Propósitos Gerais, Desempenho, Consumo Energético.

Memory Hierarchy Impacts on Performance and Energy Consumption of Parallel Applications in Embedded and General Purposes Systems

ABSTRACT

The need for performance affects both general purpose processors and embedded systems, since both applications tend to become increasingly complex. Moreover, the energy consumption of such processors must also be taken into account, especially for embedded systems that often rely on battery to operate. Multicore processors have emerged in this context and the need for parallelism exploitation in other granularities, bringing with it a memory hierarchy designed to meet the needs of parallel applications. Each application can be assigned to a group of applications as its data sharing level. Each group provides different implications both in performance and energy consumption of applications. This work shows that for applications with intensive processor use and low sharing data, there is little impact on the organization of the memory hierarchy used by the processor. On the other hand, high data sharing applications depend directly on the memory hierarchy. The difference between share data through the main memory or through a shared cache memory can cause a difference of 23% in the performance of an application and 30% in their energy consumption. Furthermore, the results show that the Intel Atom N2600 processor (representing embedded processors) got better speedup, energy consumption and EDP with the increase in the number of threads of high data sharing applications when compared to the Intel Core i7 860 processor (representing general-purpose processors). Thus, this work shows that, for this kind of applications with a large number of threads, the memory hierarchy is more important than the processors.

Keywords: Memory Hierarchy, Parallel Programming, Embedded and General Purpose Processors, Performance, Power Consumption.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CMP	Chip Multiprocessor
DRAM	Dynamic Random Access Memory
FIFO	First-In First-Out
GPU	Graphic Processing Units
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
IPP	Interface de Programação Paralela
LLC	Last Level Cache
LPDDR	Low Power Double Data Rate
LRU	Least Recently Used
MCM	MultiChip Module
RAM	Random Access Memory
TDP	Thermal Design Power
TLP	Thread Level Parallelism

LISTA DE SÍMBOLOS

n	Número de Cores
i	Variável de Iteração
i_T	Número de Threads
c_i	Fração de Tempo Ativo do Sistema
c_0	Fração de Tempo Ocioso do Sistema
N_{SE}	Valor da Série Harmônica
S_n	Soma da Série Harmônica
E_T	Energia Total Consumida
E_I	Energia Consumida pelas Instruções
E_M	Energia Consumida pela Memória
E_E	Energia Estática Consumida
I_{exe}	Número de Instruções Executadas
E_{inst}	Energia Consumida por uma Instrução
AC	Número de Acessos
E	Energia Consumida
T_{exe}	Tempo de Execução da Aplicação
S	Consumo Estático

SUMÁRIO

1 INTRODUÇÃO

Aplicações no geral tendem a ficar cada vez mais complexas exigindo, assim, maior poder computacional. Simulações de mecânica dos fluidos, por exemplo, são executadas em processadores de propósitos gerais a fim de fornecer soluções para problemas complexos de aerodinâmica (aeronaves, navios, edificações), hemodinâmica e biofísica. Sistemas embarcados (e.g., *smartphones, tablets*) também passaram a executar aplicações mais complexas como, por exemplo, o reconhecimento de voz, que utiliza diversas técnicas para realizar a recepção do som e o tratamento das palavras e frases a fim de transformá-las em comandos. No entanto, da mesma forma que é necessário melhorar o poder computacional de tais processadores, deve-se fazer isto com o mínimo impacto no consumo de energia.

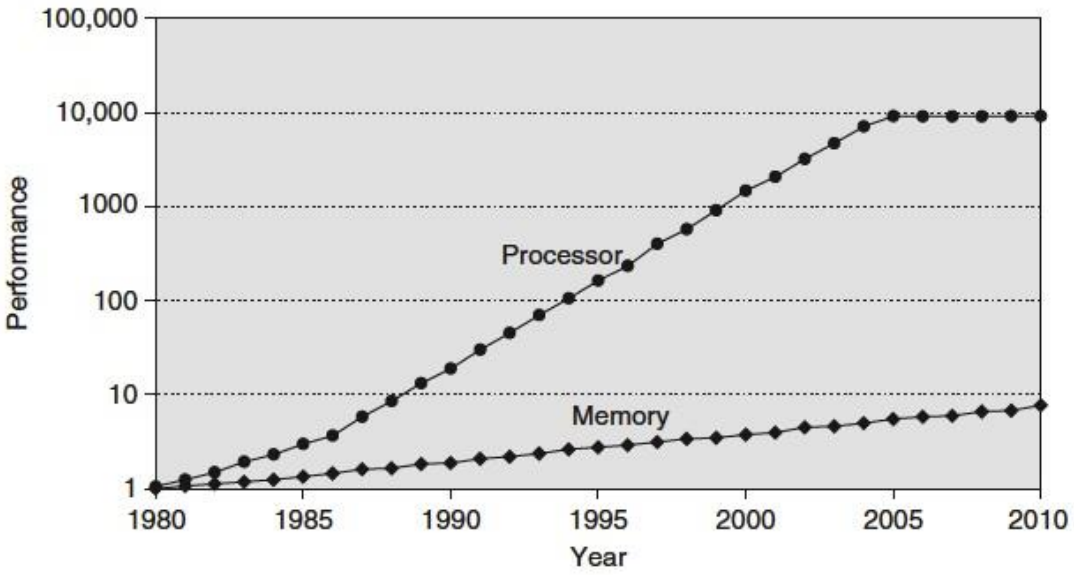
Inicialmente, a fim de melhorar o desempenho dos processadores, diferentes tecnologias foram utilizadas como, por exemplo, o *pipeline* e o superescalar. Elas exploram o paralelismo no nível de instrução (*Instruction Level Parallelism* – ILP), em que diferentes instruções são executadas concorrentemente. No entanto, devido à dependência de dados entre as instruções, existe um limite na exploração deste tipo de paralelismo (WALL, 1991). Por outro lado, o aumento da frequência de *clock* do processador também tem sido uma alternativa para melhorar o desempenho das aplicações, uma vez que reduz o tempo gasto na execução das instruções. Contudo, quanto maior for a frequência de *clock* do processador, maior tende a ser a potência dissipada, aumentando assim, o consumo energético (GEPNER, 2006).

Neste contexto, surgiram os processadores *multicore*, que caracterizam-se por possuírem vários núcleos de execução, chamados de *cores*. Dessa forma, tornou-se mais comum a exploração do paralelismo no nível de *threads* (*Thread-Level Parallelism* - TLP). Nele, a aplicação é dividida em várias tarefas (ou ainda, conjunto de instruções), que podem ser executadas concorrentemente, possibilitando assim, reduzir o tempo total de execução das aplicações.

Além do aquecimento dos processadores, existe outra limitação, que é relacionada ao sistema de memória. Enquanto as tecnologias empregadas em processadores visavam ganho de desempenho, as tecnologias usadas no sistema de memória buscavam aumentar a capacidade de armazenamento. Na Figura 1.1 é apresentado o crescimento exponencial da diferença de desempenho entre os *cores* e as memórias ao longo dos anos, onde a linha *Processor* representa o número de requisições à memória por sergundo e a linha *Memory* representa o número de acessos à DRAM por segundo. É possível observar que enquanto o desempenho dos *cores* melhorou a uma taxa de 60% ao ano, o tempo de acesso à DRAM melhorou menos de 10% ao

ano, provocando assim um gargalo entre memória e processador (CARVALHO, 2002). Nota-se também que, a partir de 2005, com o advento da tecnologia *multicore*, não houve aumento significativo em desempenho por *core*, embora exista aumento no desempenho total do processador.

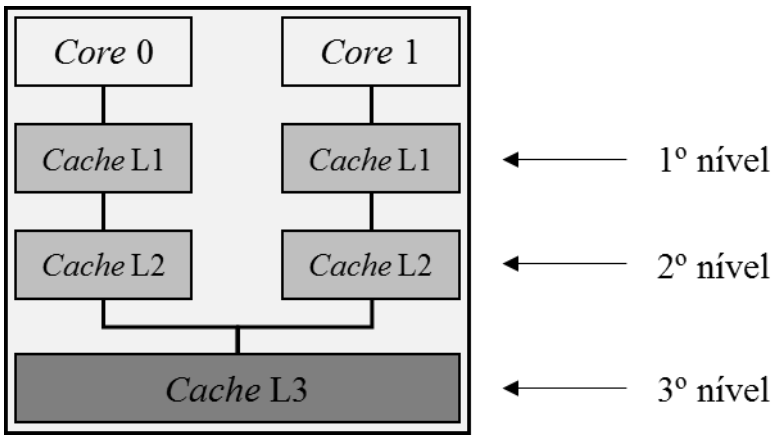
Figura 1.1 – Gargalo no Desempenho entre Memória e Processador



Fonte: Hennessy (2012, p. 73)

Para diminuir o impacto da diferença de desempenho entre memória e processador, ocorreu, dentre outras técnicas, a introdução das memórias *cache* multiníveis. Elas são memórias de maior velocidade (utilizando tecnologia diferente, porém mais cara) e com menor espaço de armazenamento. Seu objetivo é manter os dados mais utilizados pelo processador a fim de reduzir o número de acessos à memória principal, que é mais lenta. A Figura 1.2 mostra um exemplo de arquitetura *multicore* com dois *cores* (*Core* 0 e *Core* 1) e três níveis de memória *cache* (*Caches* L1, L2 e L3), onde os dois primeiros níveis são privados a cada *core* enquanto o terceiro nível é compartilhado entre os *cores*. Como pode ser observado na Figura 1.2, em uma arquitetura *multicore*, memórias *cache* multiníveis ocupam uma área significativa do *chip* do processador, já que elas têm papel relevante na determinação de seu desempenho: sua capacidade e velocidade determinam a vazão de dados disponibilizados aos *cores*.

Figura 1.2 – Exemplo de Arquitetura *Multicore*



Fonte: Mello (2004, p. 12)

Tradicionalmente, processadores com um único *core* já incluíam níveis de memória *cache intra-chip* para diminuir a latência média do acesso à memória. Entretanto, quando múltiplos *cores* estão presentes, existe uma limitação relacionada à comunicação entre as *threads* que executam sobre estes *cores*. Neste caso, o acesso aos dados se dá em regiões da memória que estão hierarquicamente mais distantes do processador (i.e.: memória *cache* L3 e principal), e que possuem maior consumo de energia e tempo de acesso quando comparado ao acesso à memória privada de cada processador (memória *cache* L1 e L2) (KORTHIKANTI, 2010).

Portanto, embora o desempenho das aplicações em arquiteturas *multicore* possa ser melhor devido à exploração do TLP, a comunicação entre as *threads* pode levar a perda de desempenho e maior consumo energético. Assim, a hierarquia de memória *cache* atua de forma decisiva nestes quesitos, uma vez que são as memórias *cache* que têm o papel de fornecer os dados para os *cores* de maneira rápida e, ao mesmo tempo, reduzir ao máximo o consumo energético do sistema.

1.1 *Objetivos*

Considerando o cenário apresentado na seção anterior, infere-se que embora a paralelização permita ganhos de desempenho, pode levar a um maior consumo energético, já que faz-se necessário trocar informações entre as *threads* em regiões da memória que estão hierarquicamente mais distantes dos *cores* e que possuem maior consumo de energia. Ademais, o sistema de memória

de processadores embarcados é diferente quando comparada a de propósito geral, em termos de tamanho e consumo de energia devido a sua portabilidade e dependência de baterias.

Assim, este trabalho tenta mostrar que a eficiência energética das aplicações paralelas varia conforme as características da hierarquia de memória, o número de *threads*, a característica da aplicação e o tipo do processador (e.g.: propósito geral ou sistema embarcado). Além disso, será avaliada a influência da microarquitetura do processador para aplicações paralelas com diferentes necessidades de comunicação/sincronização entre as *threads*. Portanto, os objetivos deste trabalho correspondem a:

- Analisar o comportamento e o impacto de diferentes hierarquias de memória no desempenho e consumo de energia considerando processadores de propósito geral e embarcados;

- Para cada hierarquia de memória, verificar se a escalabilidade (i.e.: comportamento quando o número de *cores* aumenta em relação à execução sequencial), em termos de desempenho e consumo de energia, é a mesma em ambos os tipos de processadores;

- Confirmar que a microarquitetura do processador e a hierarquia de memória influenciam no desempenho de forma diferenciada para cada tipo de aplicação paralela.

Para tanto, foram implementadas cinco aplicações classificadas em três grupos de acordo com a necessidade de comunicação, pontos de sincronização e dependência de dados. Todas elas foram paralelizadas através da Interface de Programação Paralela OpenMP (*OpenMulti Processing*). O simulador Multi2Sim foi utilizado para modelar e simular dois processadores: Intel Core i7 860, representando os processadores de propósito geral, e Intel Atom N2600, representando os processadores para sistemas embarcados; além de quatro hierarquias de memória com diferentes características.

1.2 *Estrutura do Texto*

Este trabalho está organizado como segue:

- No Capítulo 2, as arquiteturas *multicore* são discutidas com ênfase nas hierarquias de memória e nas principais diferenças entre processadores de propósito geral e de sistemas embarcados. A seguir são introduzidos conceitos referentes à memória *cache* e, por fim, é explicado o principal desafio referente às hierarquias de memória em arquiteturas *multicore*: a comunicação entre *threads*.

- O Capítulo 3 apresenta os trabalhos relacionados a esta dissertação, os quais analisam o impacto no desempenho de diversos *benchmarks* através de variações na hierarquia de

memória, na organização dos *cores* e na rede de interconexões. Ademais, o escopo deste trabalho é apresentado, destacando suas principais contribuições com relação aos trabalhos já realizados.

- O Capítulo 4 descreve a metodologia utilizada no desenvolvimento deste trabalho. Inicialmente, são contextualizadas as principais características do OpenMP, que é a IPP utilizada neste trabalho. A seguir, o conjunto de *benchmarks* escolhido é apresentado com a estratégia de paralelização adotada. Na sequência, o simulador Multi2Sim é descrito, juntamente com as modificações realizadas e sua validação. Ao final, contextualiza-se o ambiente de execução, destacando os processadores e as principais características de cada hierarquia de memória escolhida.

- Os resultados obtidos são apresentados e discutidos no Capítulo 5. Para tanto, será inicialmente analisado o comportamento das hierarquias de memória em termos de *hits*, *misses* e acessos para cada grupo de *benchmarks*. A seguir será realizado uma análise de desempenho e consumo energético em relação aos processadores, *benchmarks*, número de *threads* e hierarquias de memória. Por fim, será analisado o custo-benefício destas configurações.

- O Capítulo 6 apresenta as conclusões obtidas com este trabalho e reforça sua contribuição. Além disso, trabalhos futuros são discutidos.

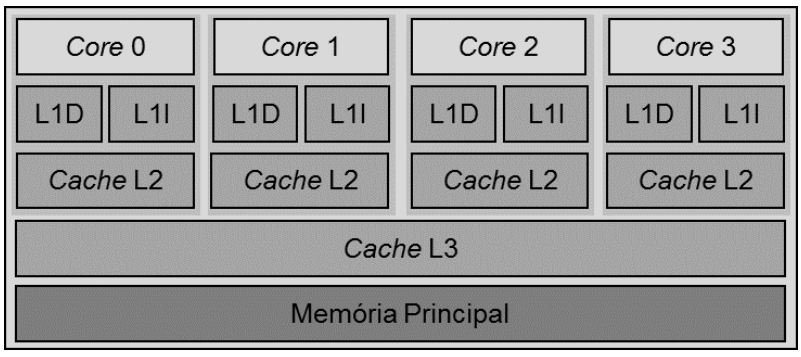
2 ARQUITETURAS MULTICORE

Buscando aumentar o desempenho e tirar vantagem do número crescente de transistores disponíveis devido à melhora no processo de manufatura de circuitos integrados, a indústria passou a investir na tecnologia *multicore*, tornando-se uma alternativa às limitações do ILP – *Instruction Level Parallelism* que processadores do tipo *superescalar* demonstravam; além de também apresentar-se como uma solução para a alta potência que eles consomem, uma vez que pode-se aumentar o desempenho através de um maior número de *cores*, sem aumentar a frequência de operação.

As arquiteturas *multicore* possuem, em um mesmo *chip*, mais de uma unidade de processamento chamada *core*, cada um com seus próprios componentes como, por exemplo, registradores e unidades de execução. Arquiteturas *multicore* são capazes de realizar a execução paralela de diferentes fluxos de instruções em unidades de processamento independentes, através da exploração do TLP. No entanto, para que exista a cooperação entre estes diferentes fluxos que executam concorrentemente, há a necessidade de prover troca de dados entre eles. Assim, a comunicação entre estes fluxos ocorre em um espaço de endereçamento compartilhado, através de instruções do tipo *load* e *store* (HENNESSY, 2012), ou através de trocas de mensagens embora este segundo tipo não seja abordado neste trabalho.

O sistema de memória de um processador *multicore* é organizado em diferentes níveis, formando uma hierarquia de memória. Este sistema é fundamental para o desempenho dos *cores*, uma vez que ele determinará a capacidade de armazenamento de instruções e dados, além da velocidade com que estes dados/instruções serão fornecidos para os *cores*. A Figura 2.1 apresenta um exemplo de arquitetura *multicore*.

Figura 2.1 – Exemplo de Arquitetura Multicore com três níveis de memória *cache*



Fonte: Mello (2014, p.15)

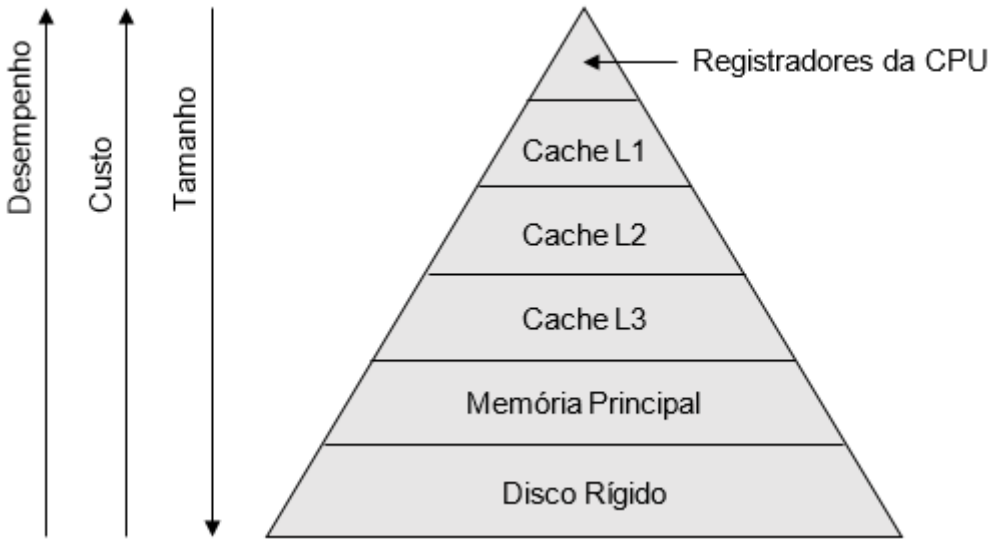
Na Figura 2.1, pode-se notar que os dois primeiros níveis de memória *cache* (L1 e L2) são privados a cada *core*. Assim, sempre que as *threads* que executam em diferentes *cores* precisarem comunicar-se, elas irão acessar o terceiro nível da hierarquia (a *cache* L3), a qual é compartilhada entre todos os *cores*.

2.1 *Hierarquia de Memória*

Enquanto que um sistema de memória único é atrativo por sua simplicidade, uma hierarquia com vários níveis e tecnologias permite que um sistema de memória se aproxime tanto do desempenho dos componentes mais rápidos quanto do custo por *bit* dos componentes mais baratos (JACOB, 2007).

De acordo com a Figura 2.2, a hierarquia de memória *cache* possibilita conciliar as memórias de alto desempenho (situadas próximas ao *core*), com as de alta capacidade de armazenamento (situadas próximas da memória principal). Memórias de alto desempenho são muito caras, porém necessárias para reduzir o tempo de espera do processador pela busca de dados. Nos níveis mais baixos da hierarquia, memórias com menor desempenho, e mais baratas, preenchem a falta de espaço disponível que as memórias de alto desempenho não conseguem suprir por serem pequenas. Dessa forma, um equilíbrio entre desempenho e armazenamento de dados/instruções é obtido.

Figura 2.2 – Relação Desempenho x Custo x Tamanho para Hierarquia de Memória



Fonte: Adaptado de Jacob (2007, p. 4)

Como pode ser observado no exemplo de arquitetura *multicore* da Figura 2.1, os registradores representam o primeiro nível do sistema de memória, sendo os mais próximos ao *core*. Entretanto, em relação à hierarquia de memória *cache*, a *cache* L1 representa o primeiro nível, recebendo requisições diretamente do *core* a qual está associada e sendo a mais rápida da hierarquia; ela tem por objetivo armazenar os dados/instruções da janela atual de execução da aplicação para que possam ser acessados rapidamente, reduzindo o tempo de espera do processador. As *caches* L2 e L3, que representam respectivamente o segundo e terceiro nível, permitem aumentar a disponibilidade dos dados/instruções sem comprometer o desempenho da aplicação; além disso, se o último nível de *cache* (*cache* L3 no exemplo da Figura 2.1) for compartilhada entre os *cores*, ela será a responsável pela troca de informações entre as *threads* de *cores* distintos.

2.2 *Memória Cache*

As memórias *cache*, localizadas entre o processador e a memória principal, têm como função principal otimizar o acesso aos dados e instruções provenientes dos dispositivos de armazenamento. Com a premissa inicial de ser rápida e passar a impressão de possuir grande espaço para armazenamento de dados, as memórias cache, segundo Jacob (2007), baseiam-se no principio da localidade dos dados, que pode ser dividido em dois tipos:

- Localidade Espacial: quando um item (instrução ou dado) da memória é acessado, existe uma tendência de que os itens contíguos na memória também venham a ser acessados dentro de um curto espaço de tempo. Isso ocorre porque uma aplicação, ou parte dela, geralmente segue um fluxo sequencial, assim como vetores de dados, que são armazenados sequencialmente na memória. A Figura 2.3 apresenta o pseudocódigo de um algoritmo que soma dois elementos de posições diferentes do vetor e armazena em uma terceira posição do mesmo vetor. Como pode ser visto internamente ao laço `for`, a localidade espacial está presente na estrutura de uma dimensão `vetorA`, a qual é carregada sequencialmente na memória, estando disponível para um grande intervalo de posições desta estrutura.

- Localidade Temporal: quando um item da memória é acessado, existe uma tendência de que este item possa ser acessado novamente dentro de um curto espaço de tempo. Isso ocorre, pois os dados podem ser utilizados mais de uma vez durante a execução da aplicação, como no exemplo destacado na Figura 2.3. Nela, nota-se que a variável de controle do *loop* (variável `i`)

é reutilizada em cada iteração do laço `for`, beneficiando-se assim da característica de localidade temporal.

Figura 2.3 – Exemplo do Princípio de Localidade em uma estrutura de repetição

```
for(i = 0; i < 100; i++)
    vetorA[i] = vetorA[i+1]+vetorA[i+2];
```

Fonte: Mello (2014, p. 18)

Aproveitando-se da localidade espacial, o carregamento de itens na memória *cache* é realizado em blocos, onde cada bloco possui um conjunto de instruções ou dados. A forma de alocação destes blocos na memória *cache* é definida através de mecanismos de mapeamento de memória: Mapeamento Direto, onde um determinado conjunto de blocos da memória de origem possui uma posição fixa na memória *cache;* Conjunto-Associativo, no qual existem subdivisões na memória *cache*, chamadas de conjuntos, que agrupa um conjunto de blocos e *tags;* e Totalmente Associativo, em que qualquer bloco da memória de origem pode ser carregado em qualquer posição da memória *cache.*

Quando um bloco é carregado em um conjunto já completamente preenchido, é necessário utilizar uma política de substituição de blocos da *cache*, a qual irá definir qual bloco deverá ser retirado da memória *cache* a fim de carregar o novo bloco. As políticas mais utilizadas são: *Least Recently Used* (LRU - *Menos Recentemente Usado*), a qual retira da memória *cache* o bloco menos recentemente utilizado, ou seja, dos blocos pertencentes ao conjunto, retira o bloco que a mais tempo não foi requisitado pelo processador; *First-In First-Out* (FIFO), que funciona como uma pilha, onde o bloco mais antigo do conjunto será retirado da memória *cache*; e Substituição Aleatória, em que escolhe um bloco do conjunto de maneira aleatória. Para uma discussão mais aprofundada sobre memória *cache*, consultar Jacob (2007).

O tempo entre a requisição de um bloco da memória e seu recebimento é chamado de latência. Analisando como referência o processador Intel Core i7 860, *caches* L1, que precisam ser rápidas para acompanhar o *clock* do processador, possuem latência de 4 ciclos de *clock*; já *caches* L2 e L3 possuem latências maiores, de 10 e 38 ciclos, respectivamente (INTEL, 2010). Ainda sim, a latência de memórias *cache* é muito inferior à da memória principal (na casa das centenas de ciclos de *clock*), tornando-as fundamentais para suprir a necessidade de dados do processador sem grande perda de desempenho.

Conforme pode ser acompanhado na arquitetura *multicore* mostrada na Figura 2.1, as memórias *cache* podem ser de dois tipos: privadas, como o caso das *caches* L1 e L2, ou compartilhadas, como no caso da *cache* L3. Na memória *cache* privada, todos os dados/instruções carregados nesta memória são definidos por requisições do único *core* ao qual a memória *cache* está associada. Já a memória *cache* compartilhada responde requisições de vários *cores.* Por este motivo, são as memórias compartilhadas que realizam a comunicação entre *threads* em aplicações paralelas.

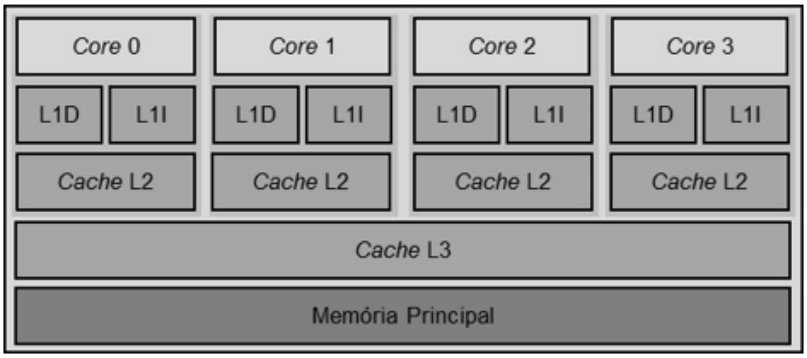
2.3 *Transferência de Dados em Memória Compartilhada*

Uma vez que diferentes *threads* de uma aplicação compartilham o mesmo espaço de endereçamento da memória principal, utiliza-se o modelo de variáveis compartilhadas como forma de comunicação de dados entre estas *threads*. Cada *thread* pode possuir dois tipos de variáveis: privadas, que são acessíveis somente pela *thread* a qual está associada e localizadas em regiões privadas da memória; e compartilhadas, acessíveis por todas as *threads* e localizadas em regiões compartilhadas da memória.

Conforme pode ser inferido através da hierarquia de memória mostrada na Figura 2.4, as variáveis privadas geralmente estão armazenadas nas *caches* privadas L1 e L2. A comunicação entre as *threads* que estão executando em *cores* distintos é realizada na *cache* L3, que é a *cache* compartilhada de menor nível na hierarquia de memória e que possui, portanto, as variáveis compartilhadas da aplicação. No caso de não existir memória *cache* compartilhada, a comunicação é realizada pela memória principal.

Os dados em memória compartilhada devem ser controlados quanto ao acesso de duas ou mais threads simultâneas uma vez que poderá provocar inconsistências no valor deste endereço de memória dependendo da ordem em que ele é lido ou gravado pelas *threads*. Para controlar e coordenar o acesso às variáveis compartilhadas por múltiplas *threads*, operações de sincronização são utilizadas para garantir que os acessos concorrentes a mesma variável ocorram de forma sincronizada.

Figura 2.4 – Exemplo de Hierarquia de Memória



Fonte: Mello (2014, p. 20)

A Figura 2.5 apresenta dois exemplos de pseudocódigos que realizam comunicação entre duas *threads*. Na Figura 2.5a cada *thread* computará a função `soma()` e armazenará o resultado em uma variável privada, chamada de `SPriv`. Após, cada *thread* irá atualizar o valor da variável compartilhada `SComp` e seguir seu fluxo de execução. No entanto, a atualização simultânea da variável `SComp` pode gerar inconsistência no resultado final, pois as duas *threads* podem tentar acessar e alterar o valor de `SComp` ao mesmo tempo, caracterizando uma seção crítica.

Figura 2.5 – Exemplo de Troca de Dados por Variáveis Compartilhadas

```
#define NUM_THREADS 2

main(){
     ...
     SPriv = soma(idThread);
     SComp = SComp + Spriv;
     ...
}
```

a) Sem Ponto de Sincronização

```
#define NUM_THREADS 2

main(){
     ...
     SPriv = soma(idThread);
     inicioSecaoCritica
          SComp = SComp + Spriv;
     fimSecaoCritica
     ...
}
```

b) Com Ponto de Sincronização

Fonte: Mello (2014, p. 20)

Para que o resultado final da execução permaneça correto, somente uma *thread* pode acessar o endereço desta variável a cada momento. Este comportamento é chamado de exclusão

mútua e possui dois estados: bloqueado, que significa que já existe uma *thread* acessando a região crítica, e desbloqueado, que indica que uma nova *thread* pode acessar a região crítica. Um exemplo de exclusão mútua é apresentado na Figura 2.5b, que utiliza duas *threads*. Nele, foram inseridas duas operações de sincronismo: `inicioSecaoCritica` e `fimSecaoCritica`, que indicam respectivamente o início e fim de uma seção crítica. Assim, quando uma *thread* estiver acessando e atualizando a variável `SComp`, a outra *thread* ficará aguardando o desbloqueio da seção crítica, para então acessá-la. Assim, garante-se que o valor final da variável `SComp` estará correto.

2.4 *Processadores de Propósito Geral e Processadores Embarcados*

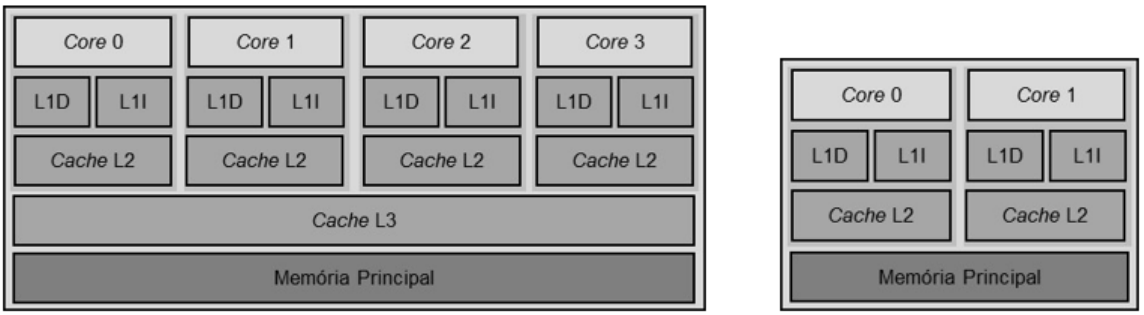
Os processadores diferem-se entre si pelo nicho de mercado em que atuam. Processadores de propósito geral buscam principalmente alto desempenho, mesmo que ao custo de alta dissipação de energia e grande espaço físico. Por outro lado, processadores embarcados (e.g.: *tablets* e *smartphones*) possuem restrições mais rígidas em relação ao gasto de espaço físico e dissipação energética, pois muitas vezes utilizam uma fonte limitada de energia (baterias).

Uma característica a ser destacada nestes processadores é o TDP (*Thermal Design Power*) máximo, que representa a quantidade máxima de energia que o processador poderá dissipar. Os processadores para sistemas embarcados possuem TDP máximo que varia em torno de 0.8 e 10 W, o que representa menos de 10% do consumo máximo do processador de propósito geral Intel Core i7 860, por exemplo, com TDP de 95 W (INTEL, 2010). Outra diferença significante está no sistema de memória. Dispositivos móveis normalmente possuem memórias desenvolvidas com tecnologia *Low Power* (e.g.: LPDDR, LPDDR2, LPDDR3). Conforme dados obtidos através da simulação de tais memórias no CACTI Tool (BERRY et al., 2013), a diferença de consumo entre uma memória RAM DDR3 para uma LPDDR3 chega a ser de 7 vezes (15,6 *nJ* para 2,4 *nJ*).

Na Figura 2.6 são apresentadas duas hierarquias de memória: o processador Intel Core i7 860 (Figura 2.6a), representando a classe de propósito geral, e o processador Intel Atom N2600 (Figura 2.6b), de sistemas embarcados. O processador Intel Core i7 860 utiliza quatro *cores* com memórias *cache* L1 e L2 privadas por *core*, além de *cache* L3 compartilhada entre os *cores*, a qual realiza o compartilhamento de dados entre as *threads*. Por outro lado, o

processador Intel Atom N2600 possui dois *cores* com *caches* L1 e L2 privadas e sem *cache* L3, realizando o compartilhamento de dados entre *threads* pela memória principal.

Figura 2.6 – Hierarquias de Memória dos Processadores de Referência



a) Processador Intel Core i7 860 b) Processador Intel Atom N2600

Fonte: Mello (2014, p. 22)

Pode-se notar que as principais diferenças de processadores de sistemas embarcados em relação aos processadores de propósito geral estão relacionadas à redução da configuração de componentes responsáveis por consumir mais energia. Conforme Molka (2009), para um processador *Intel Core i7* similar ao apresentado na Figura 2.6a, o tempo de um acesso à memória *cache* L1 e L2 corresponde a 1,3 *ns* e 2,4 *ns* respectivamente. Já para as memórias compartilhadas, *cache* L3 e principal, o tempo de acesso sobe para 13 *ns* e 65,1 *ns* respectivamente. Isto implica que o acesso à memória principal é aproximadamente 50 vezes mais lento que o acesso a *cache* L1. Quando é considerada a energia consumida em cada acesso a memória, esta diferença é ainda maior. Por exemplo, conforme Berry et al. (2013) e Ji (2008), enquanto que um acesso de leitura a *cache* L1 consome aproximadamente 0,014 *nJ*, o acesso de leitura a memória principal consome aproximadamente 800 vezes mais (~15 *nJ*).

3 TRABALHOS CORRELATOS

O advento das arquiteturas *multicore* abriu espaço para diversas pesquisas relacionadas ao impacto de cada componente de *hardware* no desempenho das aplicações paralelas. Nesta seção, são expostos alguns trabalhos dessa linha de pesquisa. Além disso, são comentadas as principais contribuições deste trabalho.

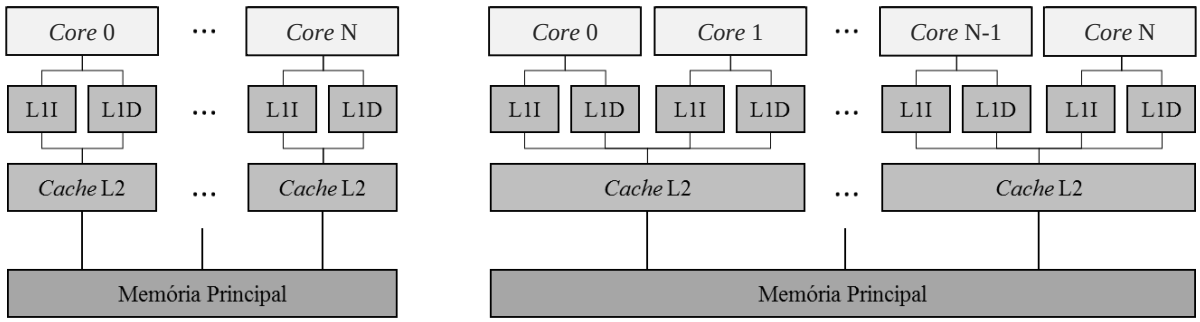
Em Nayfeh et al. (1996) verificou-se o impacto da organização de *clusters* em memórias compartilhadas de pequeno tamanho para arquiteturas *multicore.* Para tal, utilizou-se um processador único separado em *clusters* de 2, 4 e 8 *cores* com memória *cache* L1 privadas e L2 compartilhadas entre os *cores* do *cluster*. Para a interconexão entre *clusters* e a memória principal, foi utilizada a tecnologia MCM (*Multichip Module*), que é um pacote eletrônico especializado que fornece alta largura de banda com baixa latência entre interconexões. Através dos resultados, notou-se que a rede de interconexões possui uma parcela significativa do tempo de execução para sistemas de oito *cores* e aplicações de alta comunicação. Ainda conforme o autor, utilizar processadores em *clusters* de quatro ou oito *cores* com uma *cache* L2 compartilhada em cada *cluster*, conectando-os através da tecnologia MCM, reduz efetivamente o gargalo na comunicação.

Em Jaleel (2005) foi apresentado uma análise da organização de memória *cache* de último nível (LLC – *Last Level Cache*) da hierarquia de memória de um processador *multicore* para aplicações com alto compartilhamento de dados. O estudo mostrou que uma LLC compartilhada tem um desempenho em torno de 40 a 60% melhor que uma LLC privada por *core*. Além disso, também mostrou que, para algumas aplicações com alto compartilhamento de dados, 98% do uso do LLC é referente a compartilhamento entre *cores*, e quanto maior for o tamanho do LLC, menor será sua taxa de *miss*. Dessa forma, diminui-se o uso dos barramentos, melhorando assim o desempenho da aplicação.

O trabalho de Kumar et al. (2005) foca no impacto das interconexões *intra-chip* na hierarquia de memória em relação à área, potência, desempenho e problemas de projeto. O estudo apresentou que as interconexões entre memórias são tão importantes quanto o projeto da hierarquia de memória e da arquitetura interna de cada *cache*, uma vez que podem ocupar a área de três *cores* e dissipar a potência de um *core*. Também apontou que memórias *cache* que ocupam bastante área podem diminuir o desempenho das aplicações, pois o espaço restante disponível para a rede de interconexões torna-se pequeno e, portanto, mais simples, criando gargalos na comunicação.

O trabalho de Marino (2006a) testa diferentes organizações de memória em um processador de 32 *cores* com memórias *cache* L2 privadas, ou compartilhadas em grupos de 2 ou 4 *cores*. Na Figura 3.1a, é apresentada a organização de memória com *caches* L1 e L2 privadas, sendo a *cache* L1 separada em *cache* de instruções (L1I) e *cache* de dados (L1D). Na Figura 3.1b, é mostrado a organização em *clusters* de dois *cores* com a *cache* L1 privada e a cache L2 compartilhada entre os *cores* do *cluster*. Segundo o autor, o compartilhamento da *cache* L2 por mais de 2 *cores* contribuiu para o ganho de desempenho dos *benchmarks* utilizados.

Figura 3.1 – Organizações de memória do CMP analisado pelo trabalho de Marino (2006a)



Fonte: Adaptado de Marino (2006a, p. 2)

Em Marino (2006b), estendeu-se o trabalho de Marino (2006a) ao adicionar uma variação no tamanho de memória *cache* L2 para 1 MB, 2 MB e 4 MB. O estudo mostrou que, para um processador de 32 *cores* de memórias *cache* L2 compartilhadas por grupo, o aumento no tamanho da *cache* L2 melhora o tempo de execução dos *benchmarks* em 18,9% para o *Ocean*, 88,8% para o *Raytrace* e 31,8% para o *Volrend*, por causa da diminuição dos conflitos e quantidade de *misses*.

O trabalho de Alves (2007) estuda o impacto de aplicações com conjunto de dados contíguos e não contíguos para diferentes agrupamentos de *cores*. Para isso, foi utilizada uma memória *cache* L2 compartilhada em um CMP (*Chip Multiprocessor*) homogêneo, isto é, com todos os *cores* idênticos, de até 32 *cores*. Os resultados obtidos mostraram que dados contíguos na memória promovem ganho de desempenho à medida que se aumenta o número de *cores* para a aplicação. Entretanto, para dados não-contíguos, a baixa proximidade dos dados implica em uma maior taxa de falhas na memória compartilhada (*cache* L2), reduzindo o ganho de desempenho a medida em que se aumenta o número de *cores*.

Em Zhao et al. (2007) é apresentado uma configuração de memória *cache* L2 híbrida, separada em uma parte privada para o *core* e uma parte compartilhada entre os *cores* do processador. Os testes foram realizados com os *benchmarks radix, FFT, LU, cholesky, ocean, barnes* e *water*. Para estes *benchmarks*, o trabalho mostra que a arquitetura de *cache* L2 híbrida possui uma redução no tempo de execução, em média, de 10,6% contra memórias *cache* L2 totalmente compartilhadas e 3% contra totalmente privadas.

Em Alves (2009) foi estudado diferentes organizações de memória a fim de avaliar o impacto do compartilhamento de *cache* L2 em um processador de 32 *cores*. A partir dos resultados, foi observado que o aumento no número de *cores* compartilhando a mesma *cache* L2 diminuiu o desempenho das aplicações devido ao alto número de conflitos. Entretanto, segundo o autor, aumentar o tamanho da linha da *cache* reduz em 32% o número de *cache miss* e em 27% a área total da *cache*, com uma melhora no desempenho de 2% em relação ao tamanho de linha original para o *NAS Parallel Benchmark*.

3.1 *Contexto deste Trabalho*

Conforme pode ser notado com os trabalhos acima, a maioria deles converge para o uso de hierarquias de memória com *cache* L2 compartilhada como a melhor opção para ganho de desempenho. Jaleel (2005) mostrou que *caches* de último nível compartilhadas possuem desempenho melhor que privadas, o que é verdade também para *clusters* conforme apresentado em Marino (2006a). Além disso, para os *benchmarks radix, FFT, LU, cholesky, ocean, barnes* e *water*, utilizados em Zhao et al. (2007), uma memória *cache* L2 híbrida (parte privada e parte compartilhada) possui desempenho médio melhor que uma memória *cache* L2 compartilhada.

Outros trabalhos voltaram-se para a organização interna da memória *cache*. Alves (2007) descobriu que aplicações com dados contíguos possuem maior ganho de desempenho que aplicações com dados não-contíguos; assim como Marino (2006b) e Jaleel (2005) conseguiram maior desempenho ao aumentar o tamanho da *cache* compartilhada.

Além do já citado, existem outras variáveis em relação à hierarquia de memória que afetam diretamente o desempenho como, por exemplo, área do *chip*, rede de interconexões e conflitos na memória compartilhada (i.e.: sequência de acessos à memória que repetidamente substitui a mesma entrada da memória, ocorrendo quando, por exemplo, dois blocos de dados mapeados para a mesma entrada são necessários simultaneamente). Kumar et al. (2005) mostrou que memórias que ocupam grande espaço podem reduzir o desempenho do sistema

devido a limitar a área do *chip* e, portanto, a área disponível para a rede de interconexões. Assim como uma rede de interconexões limitada pode provocar gargalos na comunicação por reduzir a largura de banda, isso também é verdade para a associação de um alto número de *cores* em uma mesma memória *cache* compartilhada, conforme apontado por Alves (2009) e Nayfeh et al. (1996). Dessa forma, mostra-se que determinadas modificações, que em tese melhorariam o desempenho das aplicações, podem afetar de forma negativa outros componentes, criando gargalos que degradam o desempenho.

Este trabalho analisa o comportamento, o consumo energético e o impacto de hierarquias de memória de 2 e 3 níveis com LLC privada, compartilhada ou compartilhada por pares (análoga aos *clusters* de Marino (2006a), embora *intra-chip*). Ao contrário dos trabalhos correlatos, realiza uma análise em processadores tanto de propósitos gerais quanto de embarcados, utilizando *benchmarks* dos tipos *CPU-Bound*, *Weakly Memory-Bound* e *Memory-Bound*. Além disso, expande os trabalhos correlatos ao realizar simulações sequenciais e paralelas com 2, 4, 8 e 16 *threads*, avaliando a escalabilidade das hierarquias de memória para os diferentes tipos de processadores. Principalmente, avalia a influência da microarquitetura para cada um dos *benchmarks* utilizados, traçando uma relação entre microarquitetura, hierarquia de memória e características das aplicações para diferentes números de *threads*.

4 METODOLOGIA

Esta seção descreve a metodologia empregada neste trabalho. Inicialmente, na Seção 4.1 é apresentada a IPP OpenMP utilizada para a paralelização dos *benchmarks* explicados na Seção 4.2. Na seção 4.3 é exposto o simulador Multi2Sim, suas métricas, as modificações realizadas e sua validação. O ambiente de execução escolhido para a análise é apresentado na Seção 4.4.

4.1 Interface de Programação Paralela *OpenMulti Processing*

Embora as arquiteturas tenham evoluído para a execução paralela de tarefas por meio de múltiplos *cores*, em muitos casos, é necessário modificar o programa para que as *threads* sejam executadas em paralelo. Assim, a exploração do paralelismo é explicita por parte do programador. Ou seja, ele tem a tarefa de definir a comunicação entre processos/*threads* e pontos onde um programa pode usufruir do paralelismo sem provocar erros de execução como, por exemplo, dependência de dados ou sincronização de tarefas. Por este motivo, utilizam-se as IPPs, que fornecem recursos para que o programador possa especificar tais necessidades e, por consequência, facilitam a paralelização.

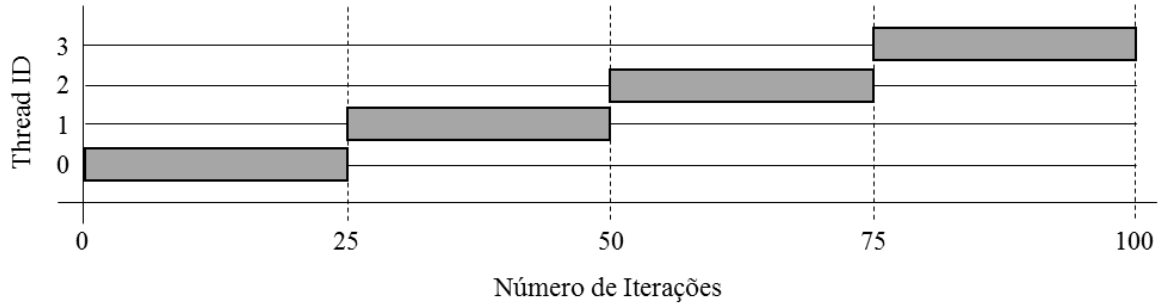
O OpenMP é uma IPP multi-plataforma, voltada a processamento paralelo em memória compartilhada para as linguagens C/C++ e Fortran, que consiste de um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente (CHAPMAN et al., 2008). O paralelismo em OpenMP é extraído através do uso de diretivas de compilação, as quais possuem significado especial para o compilador. Elas são divididas em construtor paralelo, construtores de compartilhamento de trabalho e diretivas de sincronização.

O **construtor paralelo** é a diretiva que informa ao compilador qual região do código será executada em paralelo, seguindo o modelo *Fork/Join*. Assim, inicialmente o código segue um fluxo sequencial principal (*thread master*). Ao executar-se uma diretiva de construtor paralelo, iniciando uma região paralela, criará um novo grupo de *threads* (*Fork*). Ao término da região paralela existirá uma barreira implícita que sincronizará todas as *threads* (*Join*). Após este ponto, somente a *thread master* continuará a execução do código (AYGUADÉ et al., 2009).

Os **construtores de compartilhamento de trabalho** são diretivas responsáveis por distribuir a carga de trabalho entre as *threads* da região paralela. Tais construtores são divididos em: laços paralelos, seções paralelas e construtor único de trabalho.

No construtor de laços paralelos, as iterações da estrutura de repetição *for* são distribuídas entre as *threads*. A granularidade da distribuição da carga de trabalho dos laços paralelos é determinada pelo *chunk*: um subconjunto contíguo de iterações (e.g.: uma iteração de um laço *for* ou múltiplas iterações). Os *chunks* podem ser escalonados entre as *threads* através de diferentes escalonadores: *static*, *dynamic*, *guided* e *runtime*. Para a paralelização dos *benchmarks* deste trabalho, foi utilizado o escalonador *static*, o qual, conforme Lorenzon (2014), mostrou melhores resultados comparado aos demais escalonadores. Ele funciona através da atribuição dos *chunks* para as *threads* de foma estática, por meio da política *round-robin*, ordenados pelo número da *thread*. A Figura 4.1 ilustra o comportamento deste escalonador. Nela, 100 iterações de um laço (eixo X) são distribuídas entre quatro *threads* (eixo Y) com *chunk* definido em 25 iterações. Assim, cada *thread* irá computar o *chunk* que lhe foi atribuído.

Figura 4.1 – Exemplo de Laço Paralelo com o Escalonador *Static*



Fonte: Mello (2014, p. 28)

No construtor de seções paralelas, são atribuídas partes diferentes de trabalho entre as *threads*. Ele permite especificar diferentes regiões de código para serem executadas em paralelo por diferentes *threads*, assim, cada *thread* é responsável pela execução de cada bloco.

Já no construtor único de trabalho, um bloco de execução de código é atribuído para ser executado por uma única *thread*. Enquanto a *thread* escolhida está executando este bloco de trabalho, as demais aguardam em uma barreira implícita, no final desta região.

Por fim, as **diretivas de sincronização** servem para evitar condições de corrida, nas quais duas ou mais *threads* tentam atualizar ao mesmo tempo a mesma variável, causando inconsistências nos resultados. Portanto, as diretivas de sincronização garantem acesso exclusivo a estas variáveis por meio das diretivas: *critical*, *atomic* e *barrier*.

A sincronização entre as *threads* também pode ser implícita, como ocorre no início e fim de uma região paralela. Por exemplo, sempre que ocorre a criação de novas *threads* (*fork*),

elas são sincronizadas entre si antes de iniciar a computação. O mesmo ocorre na finalização destas *threads* (*join*), que são todas sincronizadas antes de serem finalizadas. Sempre que houver ponto de sincronização (explícito ou implícito) entre as *threads*, no OpenMP elas entram em estado de *busy-waiting*, ou seja, elas executam instruções de acessos a memória para checar a variável de controle repetidamente até o final da sincronização.

4.2 *Conjunto de Benchmarks*

Para este trabalho, foram implementadas cinco aplicações classificadas em três categorias: *CPU-bound*, *Weakly Memory-Bound* e *Memory-Bound*, que são diferenciadas pelo número de acessos à memória (endereços compartilhados e privados), pela dependência de dados e pela sincronização entre *threads*.

Embora existam diferentes conjuntos de *benchmarks* como, por exemplo, o *NAS Parallel Benchmark* (BAILEY et al., 1994) e o *ParMibench* (IQBAL et al., 2010), a classificação aqui proposta, levando em conta a quantidade de comunicação entre as *threads*, não é abordada em tais conjuntos. Assim, como este trabalho busca avaliar o impacto de diferentes organizações de memória *cache* em *benchmarks* relacionados com aplicativos de um usuário regular, foi implementado um conjunto que satisfaça essa necessidade.

A Tabela 4.1 apresenta as principais características de cada *benchmark* (explicados posteriormente), os quais foram obtidos através de dados de acesso a memória coletados do Pin Tool (LUK et al. 2005) e de análise prévia das aplicações paralelizadas. Nela, podem ser observados os números de acesso às memórias compartilhada e privada, considerando leitura, escrita e o total para cada *benchmark*. Dependência de dados significa que existe pelo menos uma *thread* que somente irá iniciar sua computação quando o resultado da computação de uma ou mais *threads* estiverem prontos, o que implica na existência de comunicação entre *threads*/processos. Pontos de sincronização determinam que em certos pontos de execução da aplicação todas as *threads*/processos devem ser sincronizadas antes de iniciar uma nova tarefa. Por fim, a métrica TLP é usada conforme definida pelos autores em (BLAKE et al., 2010) e (GAO et al., 2014). Ela é usada para medir o nível de concorrência da aplicação e a taxa de utilização dos processadores durante a execução da aplicação. Quanto mais perto o valor é do número de *threads*, maior é o nível de exploração do TLP fornecido pela aplicação. Por exemplo, um TLP de 3 para 3 *threads* significa que durante a execução da aplicação, as três

threads estiveram executando concorrentemente durante 100% do tempo. O cálculo do TLP é efetuado conforme a Equação 4.1.

$$TLP = \frac{\sum_{i=1}^{N_c} c_i i_T}{1 - c_0} \quad 4.1$$

Onde c_i é a fração de tempo que N_c *cores* estão executando concorrentemente i_T *threads*, *n* é o número de *cores*, e c_0 é a fração de tempo ocioso do sistema (nenhuma *thread* está executando a aplicação). Pode-se notar ainda que as aplicações escolhidas possuem alto grau de exploração de paralelismo no nível de *threads*, o que possibilita melhor obtenção de desempenho através da paralelização. Portanto, neste trabalho, consideraremos aplicações que são paralelizáveis e que possuem ganho de desempenho com relação à versão sequencial.

Tabela 4.1 – Classificação dos Benchmarks

Benchmarks		**Escrita (%)**		**Leitura (%)**		**Total (%)**		**TLP**			**Depend. de Dados**	**Pontos de Sincr.**
		Priv	**Comp**	**Priv**	**Comp**	**Priv**	**Comp**	**2**	**3**	**4**		
CPU-B	**PI**	99.99	0.01	96.30	3.70	98.51	1.49	2.00	3.00	3.92	Não	Não
	SE	99.99	0.01	93.28	6.71	94.91	5.09	2.00	2.98	3.92	Não	Não
WMEM-B	**DJ**	99.99	0.01	71.94	28.06	73.86	26.14	2.00	2.96	3.86	Não	Não
	JV	66.63	33.37	68.34	31.66	68.25	31.75	1.91	2.75	3.33	Sim	Sim
MEM-B	**GS**	70.49	29.51	53.98	46.02	56.01	43.99	1.99	2.98	3.84	Sim	Sim

4.2.1 CPU-Bound (CPU-B)

Composta por aplicações que possuem uso intensivo do processador, esta categoria abrange os seguintes *benchmarks*:

Cálculo do Pi (PI): Este programa calcula o valor de Pi, que é definido como a relação entre o diâmetro e a circunferência de um círculo. Diversos métodos são disponíveis para calcular este valor. Neste trabalho, nós utilizamos o método de Leibniz (ANDREWS et al., 1999), que dado um número total de iterações, consiste na aproximação do valor de Pi através de sucessivos cálculos internos à uma estrutura de repetição (controlada pelo número de iterações).

Série Harmônica (SE): Consiste de uma série finita que calcula a soma de precisão arbitrária depois do ponto decimal (GOLDSTON, 2000). Tal valor é obtido através da Equação 4.2.

$$S_n = \sum_{i=1}^{N_{SE}} \frac{1}{i} \qquad 4.2$$

Onde N_{SE} representa o valor da série harmônica a ser calculado. Contém ainda um vetor que armazena a soma da precisão em cada *i* do algoritmo, e seu principal processamento consiste de sucessivas operações de divisão e cálculo do módulo de cada valor de *i*.

4.2.2 Weakly Memory-Bound (WMEM-B)

Categoriza aplicações que utilizam bastante o processador, embora possua acessos de leitura a endereços de memória compartilhados entre as *threads*, e que possuem pouca dependência de dados. Das cinco aplicações propostas, estão nesta categoria as seguintes:

Dijkstra (DJ): Este algoritmo computa o caminho mais curto entre vértices de um grafo, com custos não negativos para as arestas. Para um dado vértice fonte no grafo, o algoritmo encontra o caminho com menor custo (i.e., o menor caminho) entre este vértice e qualquer outro vértice (DIJKSTRA, 1959). Ele consiste em encontrar a distância entre cada vértice de uma matriz de adjacência de tamanho *N x N*. A saída consiste de um vetor contendo a distância mínima entre cada um dos vértices.

Jogo da Vida (JV): Simula a evolução da vida em uma sociedade representada por uma estrutura bidimensional. A evolução é baseada em leis genéticas definidas por Conway, levando em consideração o estado das células vizinhas (GARDNER, 1970). O algoritmo consiste de percorrer posição por posição uma matriz de tamanho *N x N* computando as leis genéticas. Este processo é repetido até que a evolução da sociedade esteja satisfeita e a saída do algoritmo consiste da sociedade evoluída após *i* gerações.

4.2.3 Memory-Bound (MEM-B)

Categoria para aplicações que possuem dependência de dados, sincronização e comunicação entre *threads*/processos, impactando em maior número de acessos à memória

compartilhada conforme explicado na seção 2.3. Abaixo, é exposto o *benchmark* escolhido para esta categoria:

Gram-Schmidt (GS): Consiste de um método para ortonormalizar um conjunto de vetores em um espaço de produto interno, chamado de espaço Euclidiano R_n (CHENEY, 2009). O algoritmo possui uma matriz de entrada (onde cada linha representa um vetor), e através de sucessivas computações em *N* etapas (as quais devem ser computadas uma após a outra, pois a computação de N+1 depende do resultado de N), produz uma nova matriz de saída.

4.2.4 Paralelização dos Benchmarks

Conforme Rauber (2010), o processo de desenvolvimento de uma aplicação paralela consiste de três etapas:

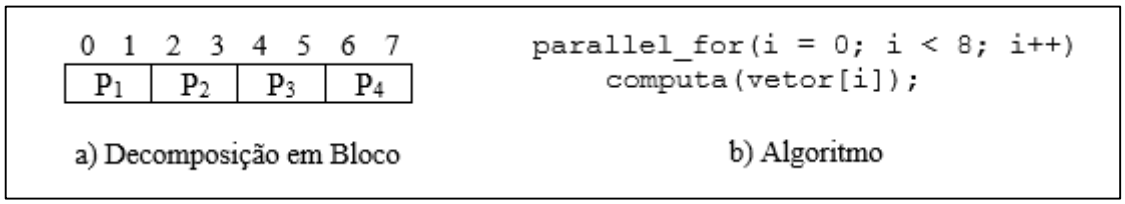
- Decomposição da Computação: consiste em decompor um programa sequencial em diferentes instruções que possam ser executadas concorrentemente;

- **Atribuição das tarefas para *threads*:** determina qual *thread* irá computar cada tarefa (carga de trabalho) definida na etapa de decomposição;

- **Mapeamento dos *threads* em unidades físicas de processamento**: realizada pelo algoritmo de escalonamento do sistema operacional.

A paralelização dos *benchmarks* apresentados nesta seção foi realizada por meio de laços paralelos utilizando as instruções informadas por Chapman (2008) e Foster (1995). Para a aplicação Série Harmônica realizou-se a computação sobre estruturas de dados com uma única dimensão (vetores). Portanto, para a sua paralelização foi utilizada a decomposição de domínio em blocos de uma dimensão (1D), conforme apresentado na Figura 4.2. Neste modelo, cada *thread* computa sobre diferentes elementos do vetor, que são distribuídos em forma de blocos de iterações (Figura 4.2a). Um possível algoritmo para este modelo de decomposição é apresentado na Figura 4.2b. Nele, existe um laço `parallel_for` que define quais as iterações do laço `for`, que percorre as posições do vetor, serão distribuídas entre as *threads* e computadas de forma concorrente.

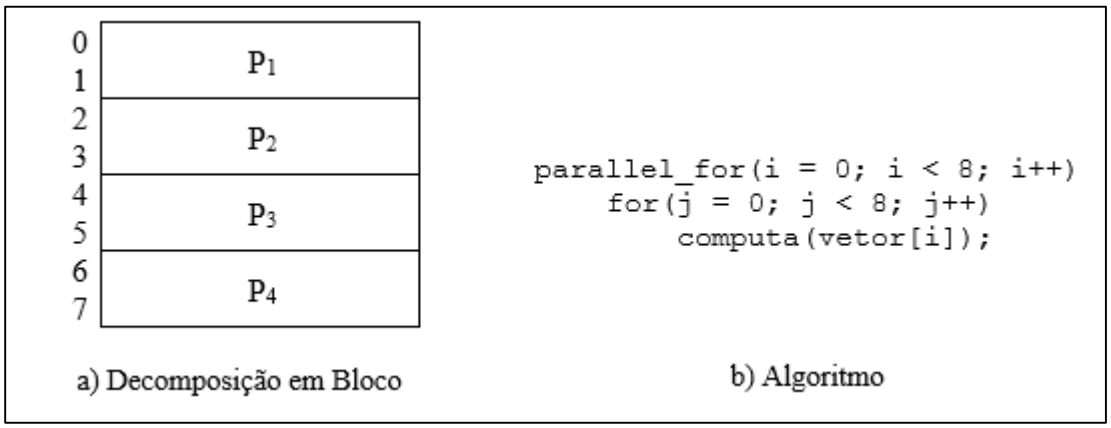
Figura 4.2 – Decomposição de Domínio 1D



Fonte: Mello (2014, p. 33)

As aplicações *Dijkstra* e Jogo da Vida computam sobre estruturas de dados bidimensionais (matrizes). Assim, a estratégia de paralelização utilizada foi a decomposição de domínio em blocos de duas dimensões (2D), apresentada na Figura 4.3a. O algoritmo deste modelo é apresentado na Figura 4.3b. Ele mostra que o laço `for` a ser paralelizado é o mais externo (`parallel_for`), ou seja, o que controla a distribuição das linhas entre as *threads.*

Figura 4.3 – Decomposição de Domínio 2D



Fonte: Mello (2014, p. 33)

Embora a aplicação Gram-Schmidt também compute sobre matrizes, ela foi paralelizada utilizando abordagens definidas por Wilson (1996) e Rauber (2010), devido a suas limitações de dependência de dados e sincronização entre as *threads*. Por fim, a paralelização envolvendo o Cálculo do Pi ocorreu através da divisão do número de iterações pelo número de *threads*, de forma análoga a decomposição de domínio 1D.

4.3 *Simulador Multi2Sim*

Conforme consta em (BARTON et al., 2013), o simulador Multi2Sim é um framework para computação heterogênea CPU-GPU implementado em linguagem *C*, que divide suas simulações em dois tipos:

- Simulação Funcional: também chamado de emulador, tem por objetivo reproduzir o comportamento do programa em execução, como se estivesse sendo executado nativamente na microarquitetura escolhida.

- Simulação Detalhada: é o modo de simulação que modela estruturas de hardware e mantém o registro dos tempos de acesso. O hardware modelado inclui estágios de *pipeline*, registradores de *pipe*, filas de instruções, unidades funcionais, memórias *cache* e outros.

O Multi2Sim utiliza simulações *Application-Only*, a qual se concentra somente na execução de aplicações no nível de usuário, removendo o sistema operacional e os *drivers* de dispositivos. Suporta simulações detalhadas de CPU *multicore*, *multithreaded* e superescalar na arquitetura x86, e funcional nas arquiteturas MIPS 32 e ARM, para aplicações de propósitos gerais. Além disso, também permite simulações de GPUs (*Graphic Processing Units*), para as arquiteturas AMD Evergreen, AMD Southern Islands (em modo detalhado) e NVIDIA Fermi (em modo funcional), utilizada para processamento gráfico. A partir do *framework* OpenCL, o simulador também possibilita computações heterogêneas consistindo de CPUs e GPUs. Neste trabalho, as simulações realizadas restringem-se a computações homogêneas na arquitetura x86 em modo detalhado com processamento *multicore*, *multithreaded* e superescalar no simulador Multi2Sim de versão 4.2.

4.3.1 Configuração do Processador

Para configurar um processador no simulador, usa-se um arquivo de configuração, o qual possui opções como:

- Frequência: frequência do *clock* do processador em Mhz.

- *Cores*: número de *cores* do processador.

- *Threads*: número de *threads* por *core*.

- *Pipeline*: modelado por 6 estágios, sendo eles: *fetch*, *decode*, *dispatch*, *issue*, *writeback* e *commit*. É possível configurar o número de instruções x86 por ciclo na etapa de *decode* e o número de microinstruções por ciclo nas etapas de *dispatch*, *issue* e *commit*. Em todas as etapas

também é possível configurar se as instruções em diferentes *threads* serão repassadas no mesmo ciclo ou se utilizarão política de *round-robin*.

- *Branch Predictor*: previsão de desvios do processador. Uma série de opções permitem configurar o tamanho das tabelas e dos registradores para cada tipo de modo de previsão de desvios. Também é possível configurar o número de ciclos de penalidade por erro na previsão de um desvio não tomado.

- *Queues*: fornece opções para configurar o tamanho das filas de *fetch*, microinstruções, instruções, *load-store* e tamanho do *reorder buffer*, dentre outras filas que podem ser encontradas em Barton et. al (2013). Também é possível escolher se as filas serão privadas ou compartilhadas entre *threads*, além de poder informar o número de registradores inteiros, de ponto flutuante e XMM.

4.3.2 Configuração do Sistema de Memória

O simulador possui uma grande flexibilidade quanto à configuração da hierarquia de memória. É possível criar sistemas de memória com qualquer número de *caches* e qualquer hierarquia entre elas, com configurações específicas em cada uma. Dentre suas configurações de *cache*, podemos destacar: número de conjuntos, associatividade, tamanho do bloco, latência e política de realocação.

Através de um conjunto de variáveis é possível configurar a hierarquia das memórias *cache*, além de definir a rede de interconexão entre as *caches* e associar cada *cache* a cada *core* e *thread* específico do processador.

4.3.3 Estatísticas de Saída

No final de cada simulação, o Multi2Sim fornece um conjunto básico de resultados estatísticos. Dentre eles, podemos citar:

- ***SimTime:*** Tempo correspondente à execução da aplicação.
- ***Cycles:*** Número total de ciclos de *clock* de simulação.
- ***Contexts:*** Máximo número de *threads* executando simultaneamente.
- ***Memory:*** Máxima quantidade de memória usada por todos os contextos.
- ***CommittedInstructions:*** Número de instruções x86 executadas.
- ***CommittedMicroInstructionsPerCycle:*** Número de microinstruções x86 executadas.
- ***BranchPredictionAccuracy:*** Taxa de acerto da previsão de desvios.

Para cada *core*, é possível analisar diversas informações como IPC (*Instructions Per Cycle*), número de desvios tomados e não tomados e número de microinstruções para cada tipo de instrução (*add*, *mov*, *mult*, *div*, etc.). Também é possível, dentre outros dados, obter informações de cada tipo de fila (*reorder buffer*, fila de instruções, fila de *load-store*, etc.) tais como: ocupação, número de vezes que a fila ficou cheia e número de leituras e escritas.

Já para o sistema de memória, é possível analisar, para cada *cache*, o número de acessos (leitura e escrita), *hits*, *misses*, taxa de acerto, dentre outros dados. Também é mostrado, por cada interconexão (entre *caches*), o número de mensagens, a quantidade de *bytes* transferidos, além da utilização, número de *bytes* por ciclo e o número de ciclos de *clock* em que a interconexão estava ocupada.

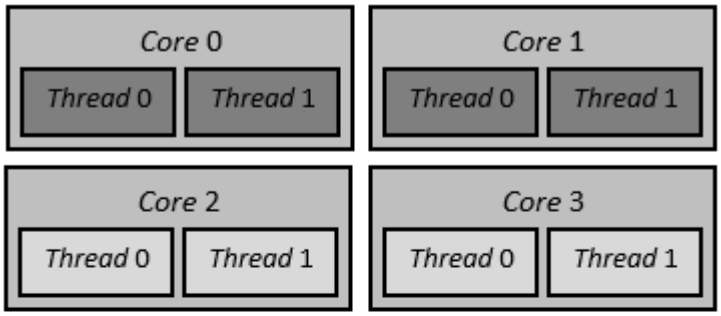
Para este trabalho, utilizaremos principalmente os dados de *SimTime*, *Contexts* e *CommitedInstructions*, além do número de acessos, *hits* e *misses* de cada memória *cache*.

4.3.4 Alocação das *Threads* em *Cores*

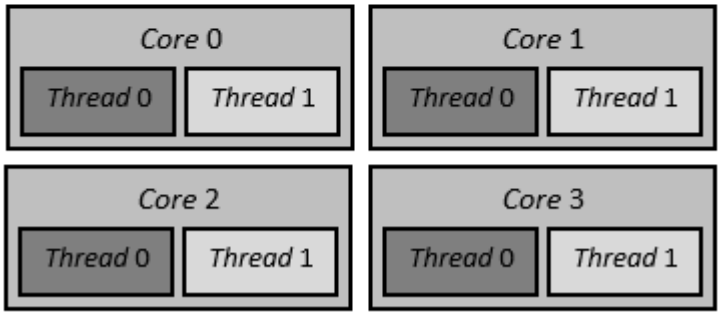
A fim de executar uma aplicação paralela, as *threads* devem ser alocadas nos diferentes *cores* do processador. O simulador Multi2Sim 4.2 realiza a alocação das *threads* em *cores* de forma estática no início da execução da aplicação. Este escalonamento visa maximizar o número de *cores* cheios, ou seja, alocando o máximo possível em um *core* para depois começar a alocar no próximo *core*. Utilizando como exemplo um processador com quatro *cores* e duas *threads* por *core*, conforme a Figura 4.4a, uma aplicação que utiliza somente quatro *threads* será alocada nos dois primeiros *cores*, isto é, duas *threads* no *Core* 0 e duas *threads* no *Core* 1 (conforme os blocos de *threads* mais escuros); os demais *cores* ficarão ociosos. Dessa forma, tanto as *threads* no *Core* 0 quanto as *threads* no *Core* 1 irão concorrer pelo tempo de execução na unidade de processamento e pelos demais recursos como as memórias *cache* privadas.

Uma vez que aplicações paralelas se beneficiam justamente por estarem em *cores* separados e poderem executar paralelamente (ao invés de concorrerem pelo tempo de processamento em um mesmo *core*), foi necessário ajustar o ponto do código que realiza a atribuição das *threads* nos *cores* do processador para que a alocação de *threads* seja feita de forma a maximizar o número de *cores* utilizados pela aplicação.

Figura 4.4 – Alocação de Threads do Multi2Sim 4.2 e da Versão Modificada



a) Alocação de Threads pelo Multi2Sim 4.2



b) Alocação de Threads pelo Multi2Sim Modificado

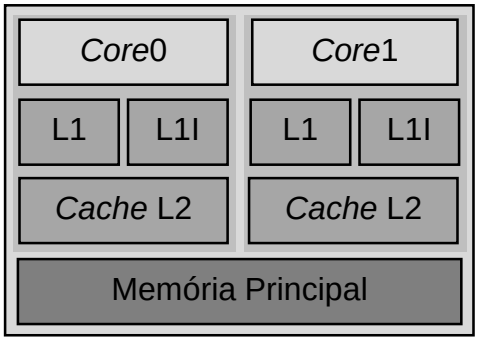
Fonte: Mello (2014, p. 37)

Por meio do mesmo exemplo anterior, com este novo tipo de alocação, nota-se na Figura 4.4b que as *threads* estão sendo alocadas separadamente em cada *core*. Dessa forma, somente após todos os *cores* possuírem uma *thread*, poderá ser alocado uma segunda *thread* para o mesmo *core*. Por este motivo, utilizou-se a versão que modificamos do simulador Multi2Sim para este trabalho.

4.3.5 Validação do Simulador

Com a finalidade de validar o simulador, analisando a proximidade entre a arquitetura simulada no Multi2Sim e a arquitetura real, foi modelado um processador similar a um Intel Core i5 2450M de microarquitetura Sandy Bridge. A Figura 4.5 ilustra a organização do sistema de memória do processador Core i5 2450M, que utiliza três níveis de *cache*: L1 (dividida em dados e instruções) e L2, privadas para cada um dos dois *cores*, e L3, compartilhada entre os *cores*. Os dados sobre o sistema de memória deste processador são mostrados na Tabela 4.2.

Figura 4.5 – Hierarquia de Memória do Processador Intel Core i5 2450M



Fonte: Mello (2014, p. 38)

Tabela 4.2 – Informações Detalhadas das *Caches* da Arquitetura Sandy Bridge

	***Cache* L1 Dados**	***Cache* L1 Instruções**	***Cache* L2**	***Cache* L3**
Tamanho Total	32 kB	32 kB	256 kB	4 MB
Conjuntos	64	64	512	4096
Associatividade	8	8	8	16
Tamanho de Bloco	64 bytes	64 bytes	64 bytes	64 bytes
Latência	3 ciclos de *clock*	3 ciclos de *clock*	9 ciclos de *clock*	21 ciclos de *clock*

Assim, realizou-se uma análise comparativa entre o sistema real e o sistema simulado pelo Multi2Sim no modo detalhado (conforme citado na seção 4.3) em uma execução sequencial do *benchmark* Multiplicação de Matrizes. A aplicação foi compilada utilizando o GCC 4.7.3. Tanto a execução real, sob o Sistema Operacional Debian, quanto a simulação com o Multi2Sim foram realizadas com os seguintes tamanhos de matrizes de entrada: 64x64, 128x128, 256x256, 512x512, 768x768 e 1024x1024. Desta forma, é possível avaliar o comportamento do simulador perante aplicações com poucos dados de entrada, onde ocorre poucos *misses* de memória, e com bastante dados de entrada, fornecendo vários *misses* e transferência de dados entre os níveis de memória da hierarquia.

Para cada tamanho de entrada, foram realizadas 30 execuções no sistema real, obtendo o tempo total de cada execução. Após, foram retirados os 5 piores e os 5 melhores tempos para calcular a média e o desvio padrão dos restantes. A Tabela 4.3 apresenta o tempo gasto pela execução no sistema real, para cada dado de entrada, além do desvio padrão referentes às 20 execuções, do tempo obtido pela simulação da arquitetura no Multi2Sim e da diferença no tempo de execução entre o sistema real e o simulado.

Tabela 4.3 – Resultados comparativos entre o sistema real e a simulação no Multi2Sim

Multiplicação de Matrizes	Tempo Real do Core i5 [s]	Desvio Padrão [%]	Tempo Simulado pelo Multi2Sim [s]	Diferença [%]
64x64	0.008801	0.894737	0.002701	325.81
128x128	0.024489	2.864285	0.055202	44.36
256x256	0.098971	0.956664	0.728486	13.58
512x512	1.018277	0.374804	8.129828	12.52
768x768	4.381140	1.168006	27.207533	16.10
1024x1024	27.764502	1.722568	105.428140	26.33

Nota-se, a partir dos resultados expostos, que ocorreu uma grande diferença entre o tempo real da aplicação e sua simulação no Multi2Sim para valores pequenos (matrizes de 64x64 elementos). Isso indica possíveis simplificações no simulador na carga inicial dos dados para a memória que não são computadas. À medida que o tamanho da entrada de dados aumenta, essa diferença torna-se negativa, ou seja, o tempo real da aplicação é menor que o tempo simulado pelo Multi2Sim. Neste caso, as tecnologias empregadas no processador que não são modeladas no simulador como o *Intel Turbo Boost*, que aumenta dinamicamente a frequência do processador conforme necessário até seu limite de energia, corrente e temperatura (INTEL, 2012); e o *pipeline* de 13 estágios em relação ao simulador que modela somente 6 estágios (BARTON et al., 2013) têm maior relevância e acabam por otimizar a execução da aplicação. Entretanto, para valores razoavelmente grandes, nota-se uma estabilidade na diferença entre os resultados para cada dado de entrada, indicando que o simulador consegue manter uma proporcionalidade entre os resultados, tornando-o válido para os experimentos deste trabalho.

4.4 *Ambiente de Simulação*

Conforme explicado na seção 2.1, a hierarquia de memória de um processador afeta diretamente seu desempenho. Tendo isso em vista, foram propostos quatro diferentes tipos de hierarquias com diferentes características a fim de analisar o impacto da hierarquia de memória na execução das aplicações paralelas em processadores de propósitos gerais e de sistemas embarcados.

4.4.1 Processadores para Referência

Como já discutido, devido à popularização dos sistemas embarcados *multicore*, as arquiteturas *multicore* podem ser divididas em duas grandes classes: processadores de propósito geral e de sistemas embarcados. As principais diferenças entre estes sistemas estão relacionadas à restrição do consumo de energia por parte dos processadores e ao sistema de memória.

Para este trabalho, serão utilizados dois processadores da Intel para referência: Intel Core i7 860, cuja arquitetura é voltada para processadores de propósito geral, e Intel Atom N2600, com arquitetura voltada para sistemas embarcados. O processador Intel Core i7 860, lançado em 2009, de microarquitetura *Nehalem* e núcleo *Lynnfield* possui quatro *cores*, onde cada um gerencia até duas *threads* por meio da tecnologia *Hyper-Threading*, permitindo a execução concorrente de até oito *threads* (INTEL, 2010). Já o processador Intel Atom N2600 de microarquitetura *Bonnell* e núcleo *Cedarview*, lançado em 2011, possui dois *cores*, onde cada um gerencia duas *threads*, por meio da tecnologia *Hyper-Threading*, em um total de quatro *threads* concorrentes (INTEL, 2012).

Para os experimentos propostos, será considerado que ambos os processadores possuirão a mesma frequência de operação de 2,8 Ghz. Isso permite avaliar o impacto da arquitetura interna dos processadores de maneira comparável, uma vez que se anula a grande diferença de tempo de execução que as frequências originais dos processadores provocariam. Além disso, possuirão oito *cores* com duas *threads* cada, a fim de permitir simulações com até 16 *threads*. Ademais, serão utilizados quatro sistemas de memória, baseados nestes processadores de referência e nos trabalhos correlatos, os quais serão apresentados mais em detalhes na seção 4.4.2. Maiores detalhes sobre a configuração dos processadores podem ser analisados no Apêndice E.

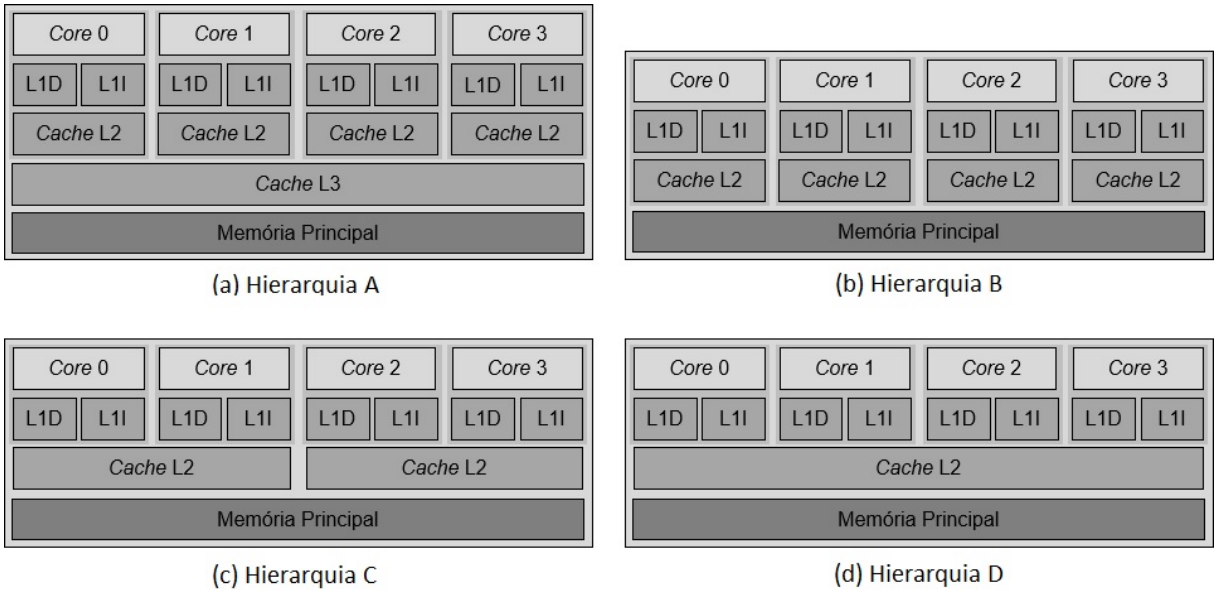
4.4.2 Organizações de Memória

As hierarquias de memória apresentadas a seguir foram escolhidas por possuírem diferentes características em relação à comunicação entre *threads* (explicadas posteriormente) e por serem encontradas no mercado e em trabalhos correlatos.

A hierarquia de memória A, apresentada na Figura 4.6a, baseia-se na hierarquia do Intel Core i7 860, o qual possui três níveis de *cache*. Conforme pode ser acompanhado na Figura

4.6a, cada *core* (*Core* 0 ao *Core* 7) possui uma *cache* L1 privada dividida entre dados (L1D) e instruções (L1I) assim como as outras hierarquias B, C e D. A *cache* L2 é privada, porém unificada entre instruções e dados e a *cache* L3 é unificada e compartilhada entre todos os *cores*. A hierarquia B, apresentada na Figura 4.6b, difere-se da hierarquia A por não utilizar *cache* L3, assemelhando-se à hierarquia de memória do processador Intel Atom N2600. A hierarquia C, apresentada na Figura 4.6c, também não possui *cache* L3, porém a *cache* L2 é compartilhada em pares de *cores*, representando uma hierarquia de *clusters* conforme o trabalho de Nayfeh et al. (1996). Já a hierarquia D, apresentada na Figura 4.6d, não apresenta *cache* L3 e a *cache* L2 é compartilhada entre todos os *cores*, assim como os trabalhos de Alves (2007 e 2009) e Marino (2006a e 2006b).

Figura 4.6 – Hierarquias de Memória Avaliadas



Fonte: Mello (2014, p. 41)

Na Tabela 4.4 é possível analisar a configuração interna de cada *cache* para todas as hierarquias trabalhadas, a qual baseia-se na configuração do Intel Core i7 860 (INTEL, 2010). As *caches* L1 para cada hierarquia possuem os mesmos parâmetros tanto para dados (L1-D) quanto para instruções (L1-I). Já as *caches* L2 e L3, embora possuam valores iguais ao do Core i7 860, diferem-se entre hierarquias pelo tamanho total de cada *cache*, uma vez que cada hierarquia possui um número diferente de *caches* e a soma do tamanho total de cada *cache* para o mesmo nível de hierarquia mantém-se o mesmo. Para as simulações, a latência da memória principal foi escolhida para 135 ciclos de *clock* de processador (com frequência de 2,8 Ghz), o

que corresponde à latência de um modelo comercial comum (Memória RAM G.Skill DDR3 1600Mhz).

Tabela 4.4 – Parâmetros internos das memórias *cache* para cada hierarquia

	Hierarquia A			**Hierarquia B**		**Hierarquia C**		**Hierarquia D**	
	L1-D/I	**L2**	**L3**	**L1-D/I**	**L2**	**L1-D/I**	**L2**	**L1-D/I**	**L2**
Tamanho Total [kB]	32/32	256	8192	32/32	256	32/32	512	32/32	2048
Conjuntos	64/128	512	8192	64/128	512	64/128	1024	64/128	4096
Associatividade	8/4	8	16	8/4	8	8/4	8	8/4	8
Tamanho de Bloco [bytes]	64/64	64	64	64/64	64	64/64	64	64/64	64
Latência [ciclos]	4/4	10	38	4/4	10	4/4	10	4/4	10

O motivo da escolha dessas hierarquias está na forma em que é feita a comunicação entre *threads* em aplicações paralelas. A Tabela 4.5 agrega os dados de latência apresentados na Tabela 4.4 para cada hierarquia de memória a fim de mostrar a latência total em uma requisição de compartilhamento de dados entre threads para cada hierarquia de memória proposta. Como pode ser acompanhado pela Tabela 4.5, na hierarquia A, todo o compartilhamento de dados entre *cores* deve ser realizado através da *cache* L3, passando pelas *caches* L1 e L2, resultando em um gasto de 52 ciclos de *clock* de processador. Ainda sim, a *cache* L3 permite que a comunicação não tenha de ser realizada pela memória principal cuja latência é de 135 ciclos. Ao contrário da hierarquia A, a hierarquia B não possui *cache* L3 e, pela *cache* L2 ser privada, o compartilhamento de dados entre *cores* será realizado pela memória principal, com um gasto de 149 ciclos de *clock*, ou seja, a latência da memória principal somada à latência das *caches* L1 e L2. Para a hierarquia C, existem duas formas de compartilhamento. A primeira (*intra-cluster*) é pela *cache* L2 entre *cores* no mesmo *cluster* (*Core* 0 e *Core* 1, *Core* 2 e *Core* 3, e assim por diante), resultando em uma latência de 14 ciclos; a segunda (*extra-cluster*) é pela memória principal entre *cores* fora de seu *cluster* (*Core* 0 e *Core* 2, por exemplo), com um gasto de 149 ciclos de *clock*. É importante salientar que, na organização de memória C, existe um grande impacto na forma em que a aplicação é distribuída para execução entre os *cores*, uma vez que o melhor caso é quando a dependência de dados entre *cores* está internamente no *cluster*, ao invés de estar entre *cores* de *clusters* distintos. Já para a hierarquia D, uma vez que a *cache* L2 é compartilhada entre todos os *cores*, todo o compartilhamento entre *cores* é realizado por essa *cache*, com uma latência total de 14 ciclos.

Tabela 4.5 – Latência no Compartilhamento de Dados em cada Hierarquia de Memória

	Hierarquia A	**Hierarquia B**	**Hierarquia C [intra-cluster/extra-cluster]**	**Hierarquia D**
Cache L1	4 ciclos	4 ciclos	4/4 ciclos	4 ciclos
Cache L2	10 ciclos	10 ciclos	10/10 ciclos	10 ciclos
Cache L3	38 ciclos	-	-	-
Memória Principal	-	135 ciclos	-/135 ciclos	-
Total	52 ciclos	149 ciclos	14/149 ciclos	14 ciclos

Através destas hierarquias de memória, deseja-se analisar qual o impacto que estas organizações provocam no desempenho de aplicações paralelas para cada um dos tipos de *benchmarks* expostos na seção 4.2, tanto em processadores de alto desempenho (como o Intel Core i7 860) quanto em processadores de alta eficiência energética (como o Intel Atom N2600). Desse modo, pode-se identificar a relação entre número de *threads*, desempenho, hierarquia de memória e microarquitetura de processamento.

4.4.3 Cálculo de Consumo Energético

A fim de calcular a energia gasta pelo sistema na execução de uma aplicação, serão utilizados os dados fornecidos por Blem et al. (2013) e pela ferramenta CACTI (BERRY, 2013), apresentados nas Tabelas 4.6 e 4.7.

Tabela 4.6 – Consumo por Acesso e Consumo Estático do Sistema de Memória

	Hierarquia A		**Hierarquia B**		**Hierarquia C**		**Hierarquia D**	
	Acesso [nJ]	**Estático [nW]**	**Acesso [nJ]**	**Estático [nW]**	**Acesso [nJ]**	**Estático [nW]**	**Acesso [nJ]**	**Estático [nW]**
***Cache* L1 Dados**	0,022472	1306910	0,022472	1306910	0,022472	1306910	0,022472	1306910
***Cache* L1 Inst.**	0,022472	1306910	0,022472	1306910	0,022472	1306910	0,022472	1306910
***Cache* L2**	0,13427	5175330	0,13427	5175330	0,282671	12252300	0,607157	37205400
***Cache* L3**	1,18196	62365000	0	0	0	0	0	0
RAM Atom	3,944	149000000	3,944	149000000	3,944	149000000	3,944	149000000
RAM i7	15,6	429000000	15,6	429000000	15,6	429000000	15,6	429000000

Tabela 4.7 – Custo por Instrução e Potência Estática dos Processadores

	i7	**Atom**
Custo médio por instrução [nJ]	1,26	0,413
Potência Estática [nW]	1097500000	242000000

Para estimar o consumo energético total (E_T), considerou-se o consumo energético das instruções executadas (E_I), da *cache* e da memória principal (E_M) e da energia estática (E_E), conforme a Equação 4.3.

$$E_T = E_I + E_M + E_E \quad (4.3)$$

Para descobrir a energia consumida pelas instruções foi utilizada a Equação 4.4, onde I_{exe} representa o número de instruções executadas multiplicado pela energia média gasta por cada instrução (E_{inst}).

$$E_I = I_{exe} * E_{inst} \quad (4.4)$$

O consumo energético total do sistema de memória foi obtido a partir da Equação 4.5, onde (AC_{L1} * E_{ACL1}) é a energia gasta pelos acessos à memória cache L1 (considerando dados e instruções); (AC_{L2} * E_{ACL2}) é a energia gasta pelos acessos à memória cache L2; (AC_{L3} * E_{ACL3}) é a energia gasta pelos acessos à memória cache L3; e (AC_{MP} * E_{ACMP}) é a energia gasta pelos acessos à memória principal.

$$E_M = (AC_{L1} * E_{ACL1}) + (AC_{L2} * E_{ACL2}) + (AC_{L3} * E_{ACL3}) + (AC_{MP} * E_{ACMP}) \quad (4.5)$$

Por último, a estimativa do consumo estático de cada componente é dado pela Equação 4.6. Como potência estática é consumida enquanto existe atividade no circuito, ela deve ser considerada durante todo o tempo de execução da aplicação, ou seja, o número de ciclos de *clock* do processador (N_{ciclos}) dividido pela sua frequência de operação (*freq*). O consumo estático total considera o consumo estático do processador (S_{CPU}), da *cache* L1 (S_{L1}), da *cache* L2 (S_{L2}), da *cache* L3 (S_{L3}) e da memória principal (S_{MP}).

$$E_E = \frac{N_{ciclos}}{freq} * (S_{CPU} + S_{L1} + S_{L2} + S_{L3} + S_{MP}) \quad (4.6)$$

5 RESULTADOS

Nesta seção, serão apresentados os resultados dos experimentos propostos neste trabalho. Serão abordadas as hierarquias de memória, em relação a número de acessos, *hits* e *misses* em cada nível de *cache*, e os processadores Intel Core i7 860 e Intel Atom N2600. Os dados de entrada para cada *benchmark* são mostrados na Tabela 5.1. Todos os *benchmarks* utilizados foram paralelizados através da interface de programação paralela OpenMP e compilados pelo GCC 3.7.4 no Sistema Operacional Debian. As aplicações foram simuladas através do Multi2Sim em sua versão 4.2, com a modificação na alocação de *threads* em *cores* discutida na seção 4.3.4.

Os resultados apresentados a seguir correspondem a análise dos seguintes tópicos: as características de cada hierarquia de memória proposta na seção 4.4.2 em termos de acessos à memória, *hits* e *misses* (Seção 5.1); a comparação dos tempos de execução (Seção 5.2) e do consumo energético (Seção 5.3) entre os processadores utilizados para cada hierarquia, verificando sua escabilidade para aplicações sequenciais e paralelas de 2, 4, 8 e 16 *threads*.; e, por fim, uma análise de EDP (*Energy-Delay Product*) será apresentada na seção 5.4.

Tabela 5.1 – Dados de Entrada para os *Benchmarks* Utilizados

Benchmarks		**Tamanho de Entrada**
CPU-Bound	**Cálculo do Pi (PI)**	1 Bilhão de pontos
	Série Harmônica (SE)	Vetor com 25000 elementos
Weakly Memory-Bound	**Dijkstra (DJ)**	Matriz de 2048 x 2048 com 1000 vértices
	Jogo da Vida (JV)	Matriz de 2048 com 50 gerações
Memory-Bound	**Gram-Schmidt (GS)**	Matriz de 1000 x 1000

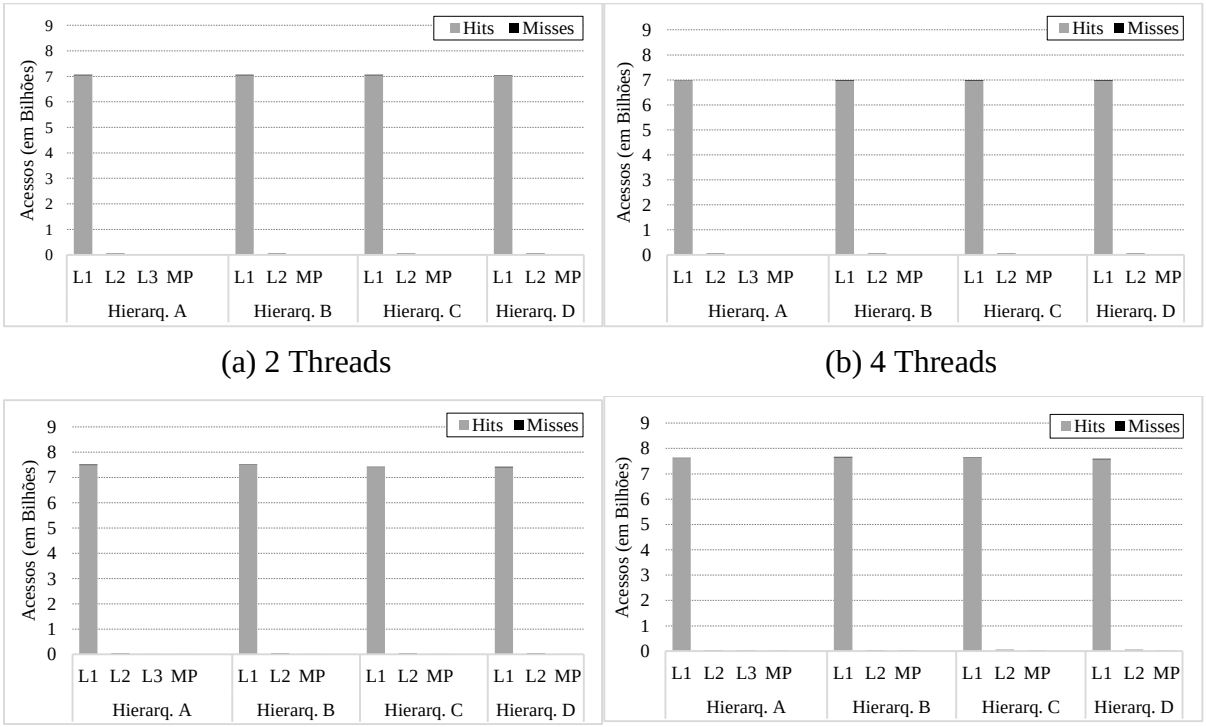
5.1 *Análise das Hierarquias de Memória*

A fim de verificar a classificação dos *benchmarks* e o comportamento de cada hierarquia de memória proposta na seção 4.4.2, obtiveram-se os resultados apresentados nas tabelas do Apêndice A, os quais são separados por grupo de *benchmarks*: *CPU-Bound* (Tabela A.1), *Weakly Memory-Bound* (Tabela A.2) e *Memory-Bound* (Tabela A.3). Os dados apresentados referem-se à média aritmética dos resultados de cada *benchmark* em um mesmo grupo. Nas tabelas é possível analisar o número de acessos, *hits* e *misses* da *cache* L1 de dados (L1), *cache* L2 (L2), *cache* L3 (L3) e memória principal (MP) para cada hierarquia (A, B, C e D) em diferentes números de *threads*. Como o objetivo desta seção é analisar o comportamento das

hierarquias de memória em relação ao compartilhamento de dados de aplicações paralelas, foram ignorados os acessos à memória *cache* L1 de instruções. Além disso, como o sistema de memória é o mesmo para ambos os processadores, será mostrado somente os dados para o Intel Core i7 860, sendo omitido os dados do Intel Atom N2600.

A Figura 5.1 apresenta os gráficos resultantes dos dados fornecidos pela Tabela A.1, que refere-se ao grupo de *benchmarks CPU-Bound*. Os gráficos apresentam o número de acessos, subdivididos em *hits* e *misses*, de cada componente do sistema de memória para cada hierarquia de memória avaliada.

Figura 5.1 – Comportamento das Aplicações *CPU-Bound*



(a) 2 Threads (b) 4 Threads

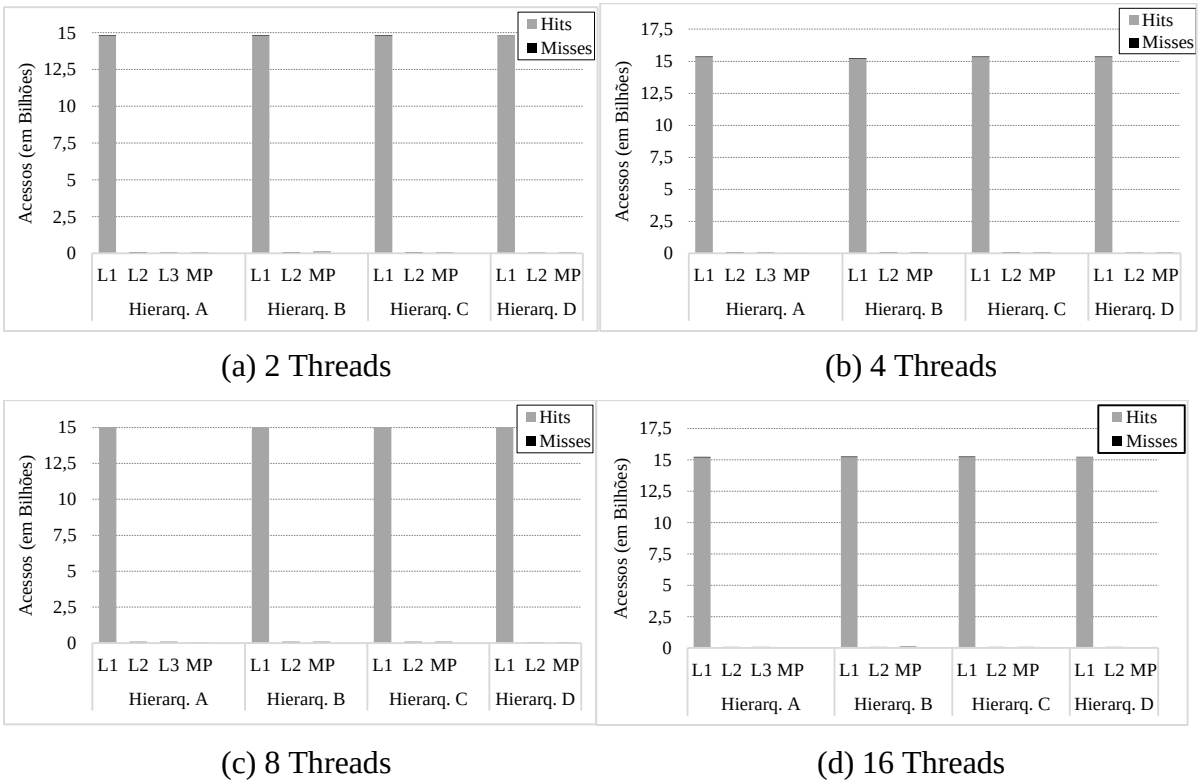
(c) 8 Threads (d) 16 Threads

Assim, nota-se que, para o grupo de *benchmarks CPU-Bound*, em 99% dos acessos a memória o dado estava presente na *cache* L1, uma vez que este grupo caracteriza-se por possuir pouco compartilhamento de dados entre *threads*. Dessa forma, as variáveis necessárias para a execução de cada trecho do programa estavam mantidas localmente na *cache* L1, tornando desnecessário o acesso constante às demais memórias do sistema. Pelo mesmo motivo, não nota-se grande diferença entre as hierarquias e nem entre diferentes números de *threads*.

A Figura 5.2 apresenta os gráficos resultantes dos dados fornecidos pela Tabela A.2, que refere-se ao grupo de benchmarks *Weakly Memory-Bound*. Os gráficos apresentam o número de acessos, subdivididos em *hits* e *misses*, de cada componente do sistema de memória

para cada hierarquia de memória avaliada. Para o grupo de aplicações *Weakly Memory-Bound*, nota-se um pequeno aumento no número de acessos às memórias *cache* compartilhadas em relação ao grupo de *benchmarks CPU-Bound*. Isso se deve ao fato de as aplicações Weakly *Memory-Bound* necessitarem de acessos de leitura aos dados compartilhados entre os *cores* dispostos na memória compartilhada. Tais acessos não são muito significativos em relação aos acessos à *cache* L1 uma vez que, como são acessos somente de leitura, não há dependência entre as *threads* e, portanto, cada *thread* pode possuir sua cópia do dado em memória local sem problemas de coerência.

Figura 5.2 – Comportamento das Aplicações *Weakly Memory-Bound*

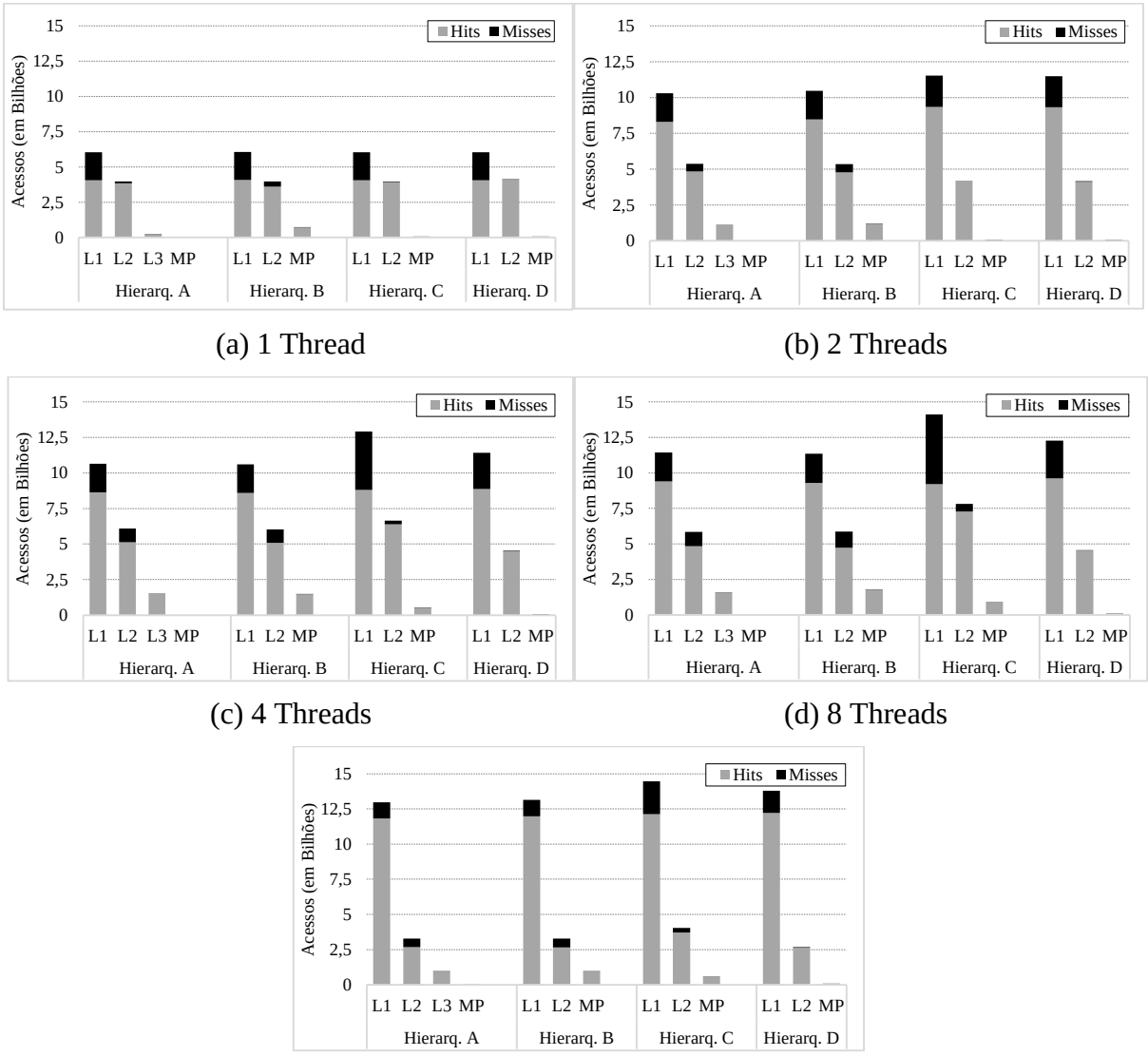


(a) 2 Threads (b) 4 Threads

(c) 8 Threads (d) 16 Threads

A Figura 5.3 apresenta os gráficos resultantes dos dados fornecidos pela Tabela A.3, que refere-se ao grupo de *benchmarks Memory-Bound*. Os gráficos apresentam o número de acessos, subdivididos em *hits* e *misses*, de cada componente do sistema de memória para cada hierarquia de memória avaliada. Nota-se, a partir dos gráficos, um aumento no número de *misses* das memórias *cache* privadas nas aplicações *Memory-Bound* em relação aos demais grupos. Uma vez que este grupo possui um alto compartilhamento de dados entre *threads*, os dados têm de ser constantemente obtidos da memória compartilhada, causando os *misses*.

Figura 5.3 – Comportamento das Aplicações *Memory-Bound*



(a) 1 Thread

(b) 2 Threads

(c) 4 Threads

(d) 8 Threads

(e) 16 Threads

No caso das hierarquias A e D, a memória *cache* L3 e a memória *cache* L2, respectivamente, realizam o compartilhamento de dados, evitando acessos à memória principal. A hierarquia B, que possui somente memórias *cache* privadas, possui maior utilização da memória principal, uma vez que é a única memória compartilhada do sistema. Já a hierarquia C, que possui uma memória *cache* L2 compartilhada em pares de *cores*, evita acessos à memória principal, exceto quando o compartilhamento de dados é realizado entre *cores* de pares distintos, o que ocorre a partir de 4 *threads*.

Importante salientar que, para o caso de 16 *threads*, houve uma redução nos acessos às memórias compartilhadas, o que se deve ao fato de os processadores utilizados possuírem 8 *cores*, conforme a seção 4.4.1, e portanto, cada *core* compartilhar seu espaço de endereçamento

com duas *threads*. Dessa forma, a comunicação entre *threads* de um mesmo *core* não necessita de acessos a memória compartilhada, podendo ser realizada diretamente pela *cache* L1. Uma vez que a carga de trabalho de cada *thread* na simulação em 16 *threads* foi dividida pela metade em relação à simulação com 8 *threads*, o número de compartilhamentos de dados entre diferentes *cores* também foi reduzido, o que explica essa diminuição nos acessos à memória compartilhada.

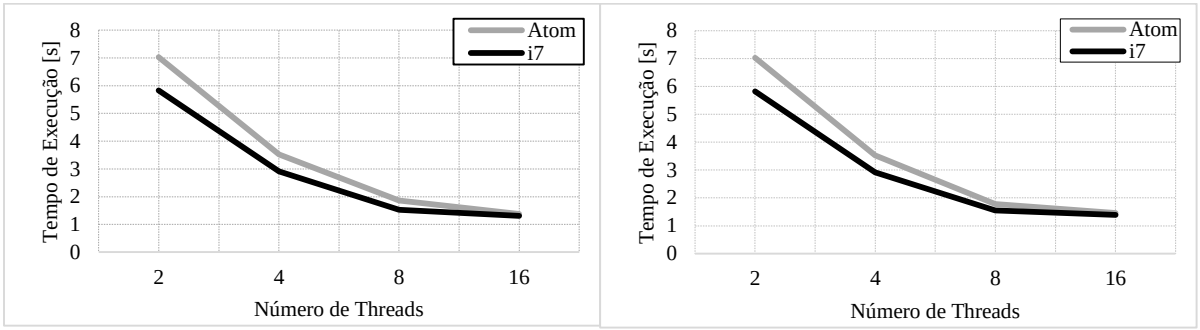
Por fim, observa-se um aumento de 57% no número de acessos à memória *cache* L1 na hierarquia A (por exemplo) em relação à versão sequencial, por causa do *overhead* de aplicações paralelas em relação a sincronização e comunicação entre *threads*. Além disso, o fato do OpenMP utilizar *busy-waiting* para sincronizar as *threads*, conforme explicado na seção 4.1, explicaria esse aumento que não é observado nos demais grupos de *benchmarks*.

5.2 *Análise de Desempenho*

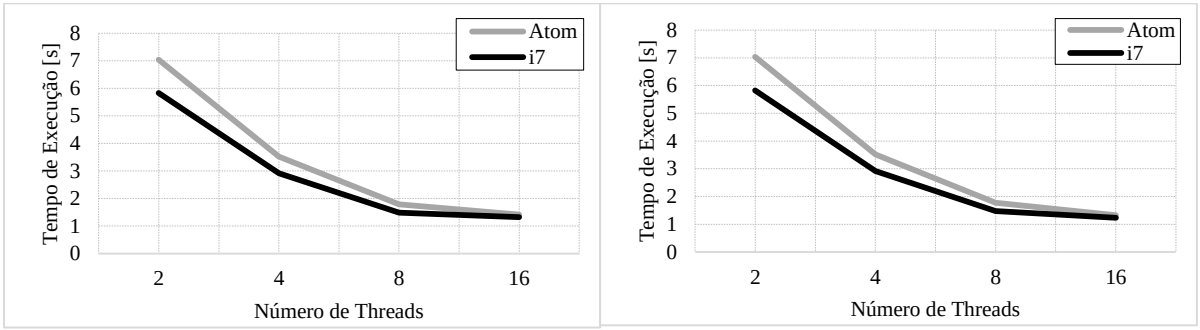
A partir dos experimentos apresentados na seção 4 deste trabalho foram obtidos os tempos de execução, em segundos, para cada *benchmark* proposto. Os dados obtidos são apresentados no Apêndice B, divididos em *CPU-Bound* (Tabela B.1), *Weakly Memory-Bound* (Tabela B.2) e *Memory-Bound* (Tabela B.3). A fim de manter as características de cada aplicação, são mostrados nas tabelas os tempos de execução referente a média geométrica dos resultados de cada *benchmark* em um mesmo grupo. Os dados foram divididos entre processadores (*Atom*, representado o Intel Atom N2600, e *i7*, representando o Intel Core i7 860), número de *threads* e hierarquias de memória (A, B, C e D).

A Figura 5.4 fornece os gráficos referentes aos dados de aplicações *CPU-Bound*, fornecidos na Tabela B.1. Nos gráficos é apresentada a relação entre o tempo de execução dos *benchmarks*, medido em segundos, e o número de *threads* para cada processador (Atom e i7). Observa-se que o sistema de memória não causa impacto significativo nas aplicações *CPU-Bound*, uma vez que os dados necessários para cada *core* mantém-se localmente em sua respectiva memória *cache* privada conforme mostrado na seção 5.1, não necessitando, portanto, acessar de modo relevante as demais memórias da hierarquia. Dessa forma, as variações na organização das hierarquias de memória, que estão focadas na *cache* L2 e L3, não influenciam de modo significativo os *benchmarks* deste grupo.

Figura 5.4 – Gráficos de Desempenho de Aplicações *CPU-Bound*



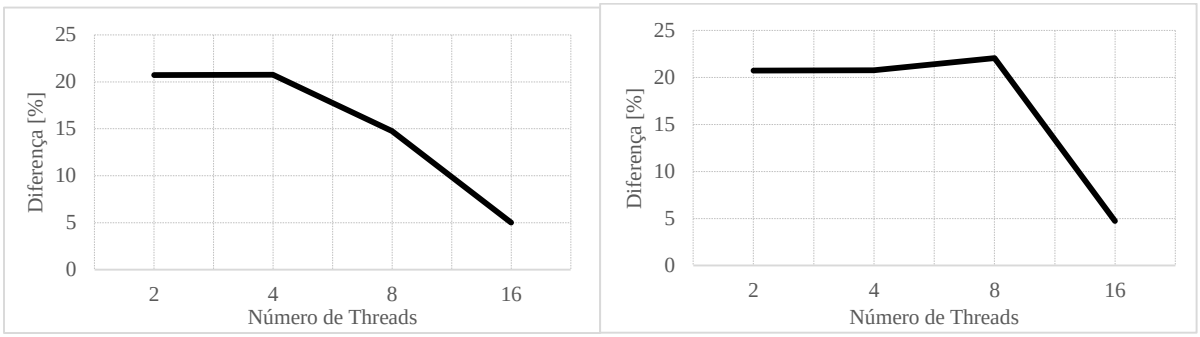
(a) Hierarquia A (b) Hierarquia B



(a) Hierarquia C (b) Hierarquia D

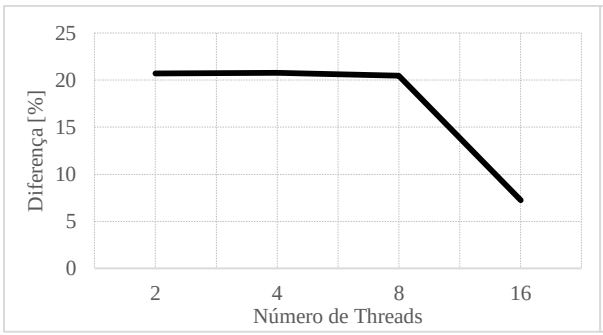
A Figura 5.5 mostra o gráfico da diferença em porcentagem entre o tempo de execução do processador Intel Core i7 860 em relação ao Intel Atom N2600 para cada número de *threads*. Nota-se que a diferença entre o i7 e o Atom mantém-se praticamente constante, em torno de 20%, exceto para o caso de 16 *threads*, mostrando o predomínio da microarquitetura dos processores neste grupo de *benchmarks* para a obtenção de desempenho. No caso de 16 *threads*, o parâmetro de entrada para as aplicações não foi grande o suficiente para permitir que todas as *threads* executassem com todo seu potencial devido ao *overhead* da paralelização, explicando assim o fato do Atom ter se aproximado subitamente do tempo de execução do i7; além disso, as limitações causadas pelo *hyper-threading*, como o compartilhamento de recursos, também podem ter influenciado esse resultado.

Figura 5.5 – Gráficos da Diferença de Desempenho entre os Processadores (CPU-B)

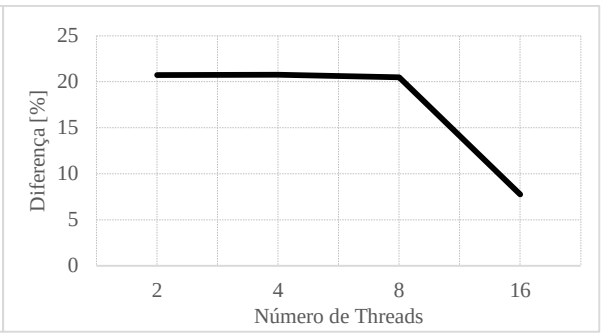


(a) Hierarquia A

(b) Hierarquia B



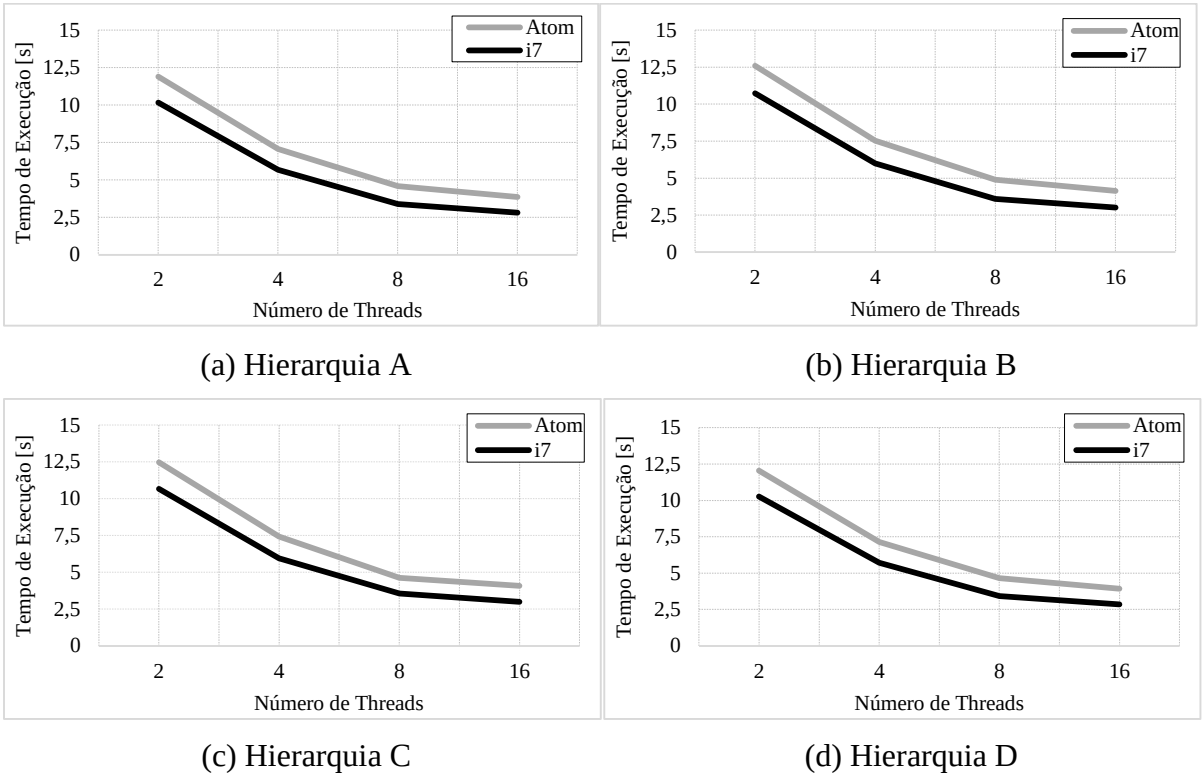
(c) Hierarquia C



(d) Hierarquia D

A Figura 5.6 apresenta os gráficos referente aos dados da Tabela B.2, para aplicações do grupo *Weakly Memory-Bound*. Nos gráficos, é apresentada a relação entre o tempo de execução dos *benchmarks*, medido em segundos, e o número de *threads* para cada processador (Atom e i7). Os resultados apontam que, para as aplicações *Weakly Memory-Bound* utilizadas, não houve grande divergência de comportamento em relação às aplicações *CPU-Bound*. Devido aos acessos à memória compartilhada, como esperado, as hierarquias A e D obtiveram os melhores resultados. As hierarquias B e C apresentaram resultados 5% piores em média, resultante dos acessos à memória principal, mostrado na seção 5.1.

Figura 5.6 – Gráficos de Desempenho de Aplicações *Weakly Memory-Bound*

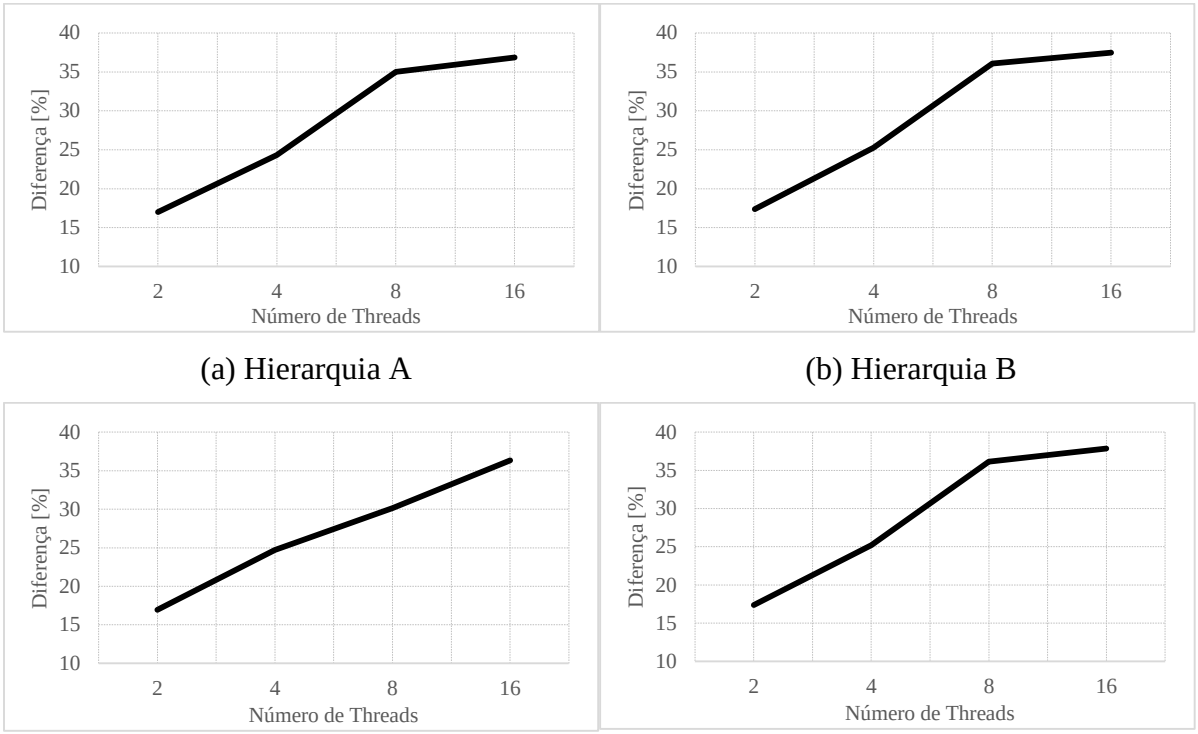


(a) Hierarquia A (b) Hierarquia B

(c) Hierarquia C (d) Hierarquia D

A Figura 5.7 mostra os gráficos da diferença em porcentagem entre o tempo de execução do processador Intel Core i7 860 em relação ao Intel Atom N2600 para cada número de *threads*. Nas aplicações *Weakly Memory-Bound*, houve um aumento de 20% na diferença entre os processadores i7 e Atom, mostrando uma melhor escalabilidade do i7 na paralelização de aplicações com acessos à dados da memória compartilhada sem dependência.

Figura 5.7 – Gráficos da Diferença de Desempenho entre os Processadores (WMEM-B)



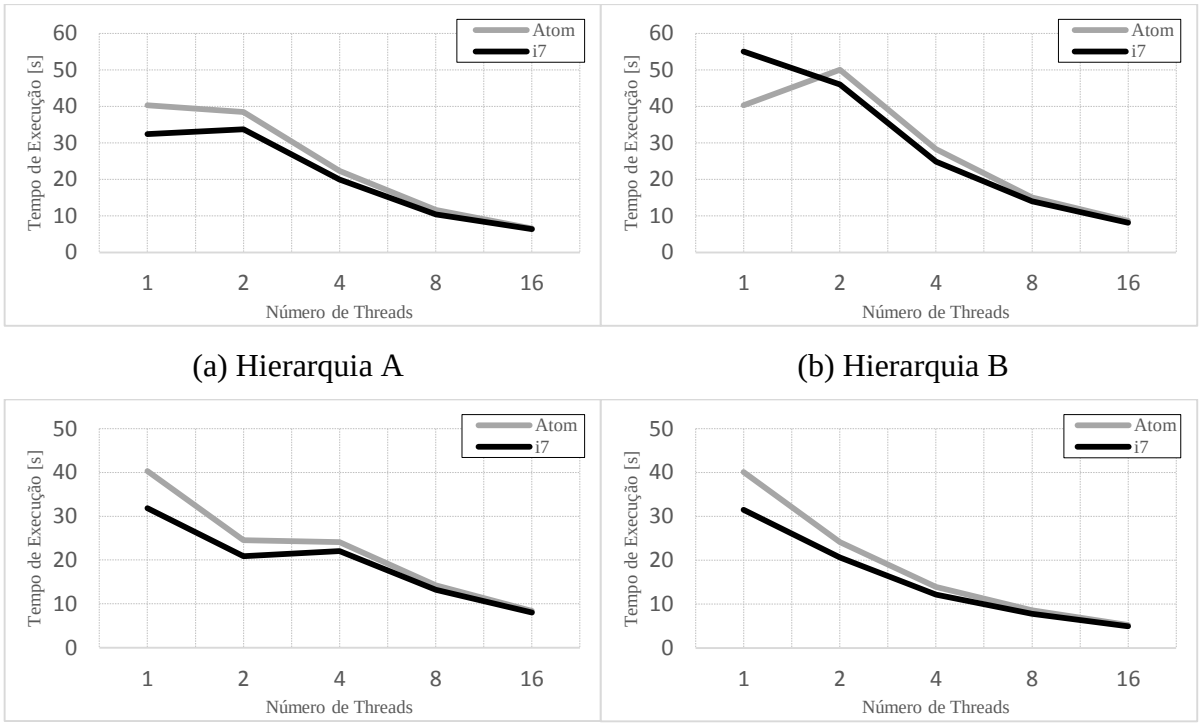
(a) Hierarquia A (b) Hierarquia B

(c) Hierarquia C (d) Hierarquia D

A Figura 5.8 mostra os gráficos dos dados contidos na Tabela B.3, referente ao desempenho de aplicações do grupo *Memory-Bound*. Nos gráficos é apresentada a relação entre o tempo de execução dos *benchmarks*, medido em segundos, e o número de *threads* para cada processador (Atom e i7). Nota-se uma diferença significativa do tempo de execução das aplicações em relação ao sistema de memória utilizado. A hierarquia D (Figura 5.8d), com *cache* L2 compartilhada entre todos os *cores*, obteve melhor desempenho, obtendo uma média geral de 12,18 segundos. A hierarquia B (Figura 5.8b), cuja comunicação entre *threads* tem de passar pela memória principal, obteve os piores resultados para este grupo de *benchmarks*, com uma média geral de 24,35 segundos, uma vez que a memória principal possui uma latência muito maior que as memórias *cache*. A hierarquia C (Figura 5.8c), por possuir *cache* L2 compartilhada em pares de *cores*, obteve resultados semelhantes à hierarquia D até duas *threads* (por serem hierarquias semelhantes até dois *cores*), entretanto, conforme o número de *threads* crescia, as comunicações entre *cores* de pares distintos começam a ocorrer, causando acessos à memória principal, o que reduziu seu desempenho em relação à hierarquia D, obtendo uma média geral de 16,89 segundos. A hierarquia A (Figura 5.8a), que possui *cache* L3 compartilhada, obteve uma média de 18,69 segundos, sendo pior que a hierarquia C em média. Entretanto, na faixa de 2 a 16 *cores*, a hierarquia A possui maior redução no tempo de execução

que a hierarquia C (81% para a hierarquia A e 61% para a hierarquia C), uma vez que esta passa a utilizar a memória principal para a comunicação entre *threads*, reduzindo seu desempenho.

Figura 5.8 – Gráficos de Desempenho de Aplicações *Memory-Bound*

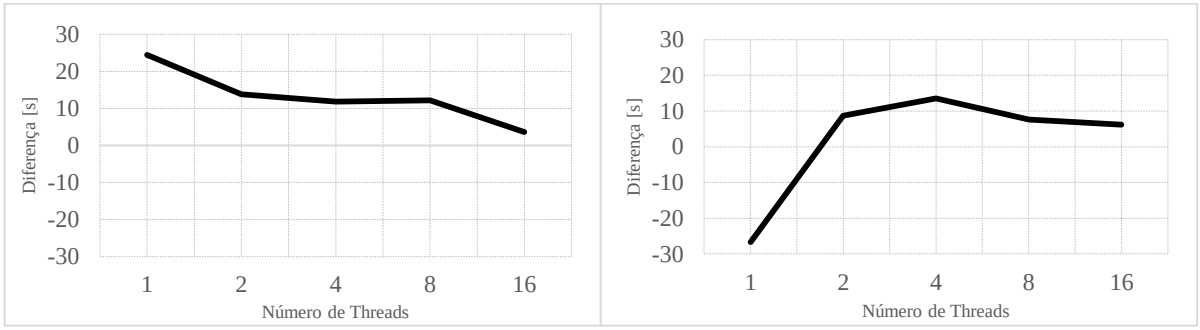


(a) Hierarquia A (b) Hierarquia B

(c) Hierarquia C (d) Hierarquia D

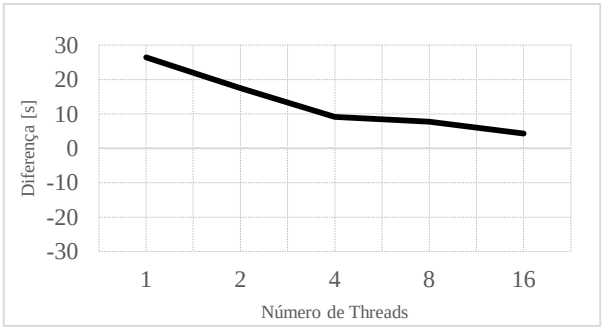
A Figura 5.9 mostra o gráfico da diferença entre os tempos de execução dos processadores para cada número de *threads*. Para o grupo de *benchmarks Memory-Bound*, a diferença do tempo de execução entre os processadores *i7* e *Atom*, mostrada nos gráficos da Figura 5.9, decai a medida que aumenta o número de *threads*. Isso ocorre, provavelmente, pela alta dependência de dados compartilhados na hierarquia de memória que reduz o potencial de execução das instruções uma vez que ocorrem muitas sincronizações e requisições em níveis de memória com alta latência. O que reforça essa teoria é o fato das hierarquias A e D, as quais possuem memórias *cache* compartilhadas e, portanto, acesso mais rápido aos dados compartilhados, apresentarem maior diferença no tempo de execução entre os processadores que a hierarquia B e a hierarquia C (para um número de threads maior que 2), que utilizam a memória principal para comunicação entre *threads*, onde a latência de uma requisição por dados compartilhados é maior. Portanto, conclui-se que, para um grande número de *threads*, o sistema de memória passa a ser o fator principal no desempenho das aplicações e não mais a microarquitetura dos processadores.

Figura 5.9 – Gráficos da Diferença de Desempenho entre os Processadores (MEM-B)

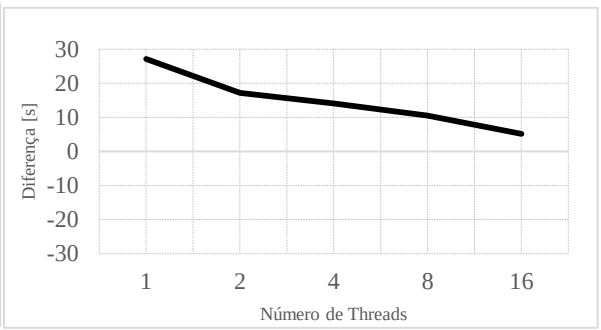


(a) Hierarquia A

(b) Hierarquia B



(c) Hierarquia C



(d) Hierarquia D

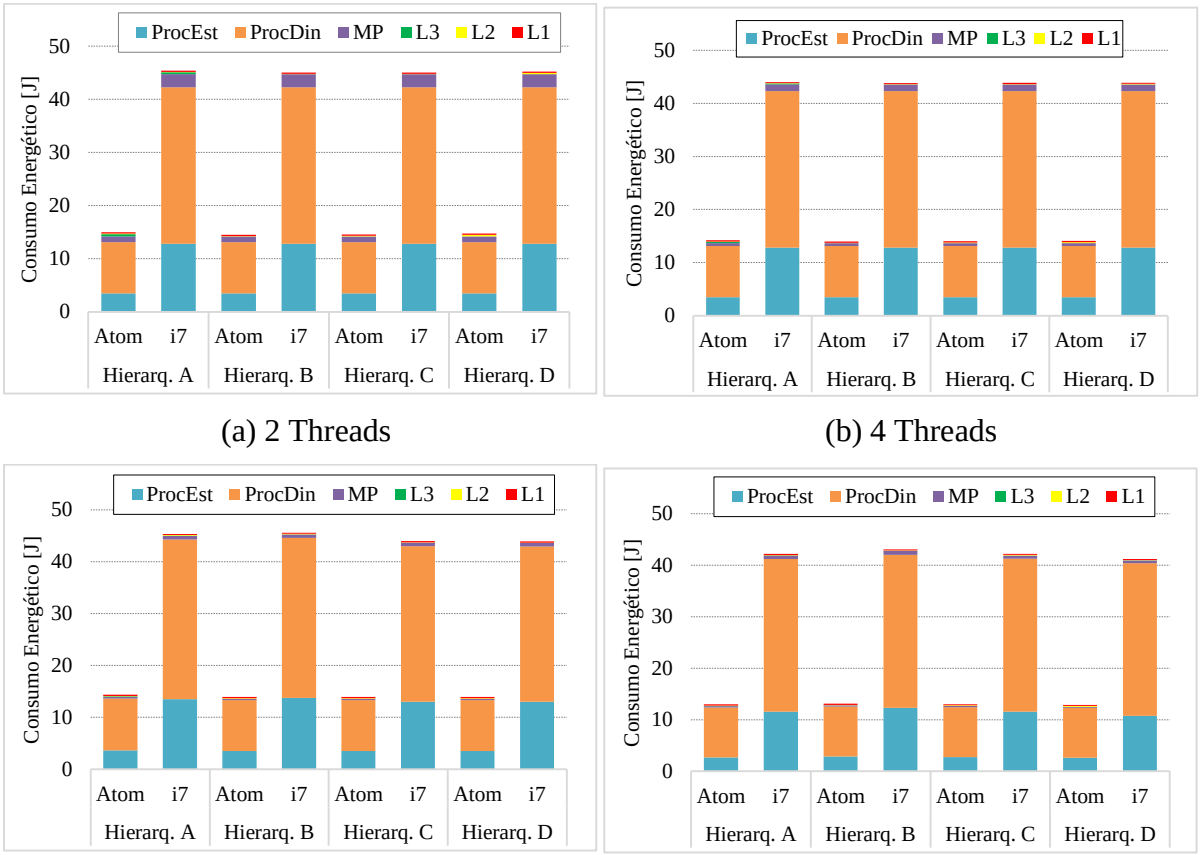
5.3 *Análise de Consumo Energético*

Por meio dos tempos de execução, do número de instruções executadas, do número de acesso para cada nível de memória e dos dados apresentados nas Tabelas 4.6 e 4.7, foi obtido o consumo energético dos *benchmarks* tanto para o processador Intel Core i7 860 (*i7*) quanto para o Intel Atom N2600 (*Atom*). No Apêndice C, são apresentados os dados referente ao consumo energético das memórias *cache* L1 (L1), *cache* L2 (L2), *cache* L3 (L3) e memória principal (MP), além do consumo estático (ProcEst), consumo dinâmico (ProcDin) e consumo total (Et) de cada processador para as hierarquia de memória A, B, C e D. As tabelas foram separadas pelo número de *threads* (2, 4, 8 ou 16 *threads*) e pelo grupo de *benchmarks*: *CPU-Bound* (Tabelas C.1), *Weakly Memory-Bound* (Tabelas C.2) e *Memory-Bound* (Tabelas C.3).

A Figura 5.10 apresenta os gráficos referente aos dados das Tabelas C.1, para aplicações do grupo *CPU-Bound*. Os gráficos apresentam o consumo energético total e de cada componente do sistema, para cada hierarquia de memória tanto para o processador Atom quanto para o i7. Nota-se, a partir dos gráficos, uma predominância no consumo dinâmico do processador (*ProcDin*) como já era esperado, uma vez que este grupo de *benchmarks* executa sobre dados sem dependência significativa entre *threads*. O consumo da memória principal, no

processador i7, de 5% na simulação com duas *threads* e 2,5% na simulação com quatro *threads* ocorre principalmente pelo consumo estático da memória principal, uma vez que, para estas quantidades de *threads*, o tempo de execução da aplicação ainda é grande. Tal comportamento não ocorre no consumo estático do processador uma vez que a redução pela metade no tempo de execução da aplicação ocorre através da duplicação do número de *cores* do sistema, mantendo o consumo estático total do processador constante. Por fim, embora este grupo de *benchmarks* tenha alto número de acessos à memória *cache* L1, conforme apresentado na seção 5.1, não houve impacto significativo no consumo energético total, uma vez que esta memória *cache* possui um gasto energético por acesso reduzido (de 0.022472×10^{-9} J) se comparado à memória principal (de 3944×10^{-9} J) por exemplo.

Figura 5.10 – Consumo Energético das Aplicações *CPU-Bound*



(a) 2 Threads

(b) 4 Threads

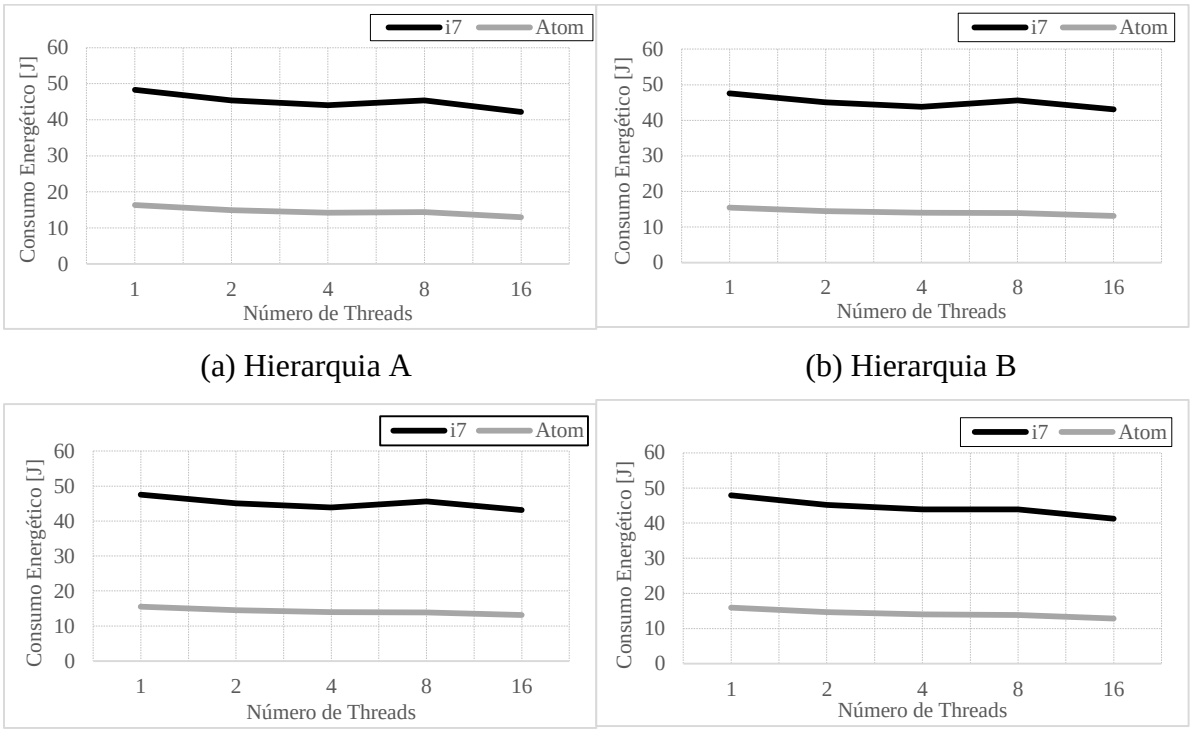
(c) 8 Threads

(d) 16 Threads

A Figura 5.11 apresenta os gráficos referentes ao consumo energético total para cada hierarquia de memória em relação ao número de *threads* e aos processadores i7 e Atom. Observa-se que, em relação ao número de *threads*, não há aumento significativo no gasto energético, uma vez que, embora o número de *cores* dobre, há a redução pela metade no tempo

de execução causado pelo alto paralelismo das aplicações. Além disso, nota-se uma economia de 66% do processador Atom em relação ao i7.

Figura 5.11 – Consumo Energético Total das Aplicações *CPU-Bound*



(a) Hierarquia A

(b) Hierarquia B

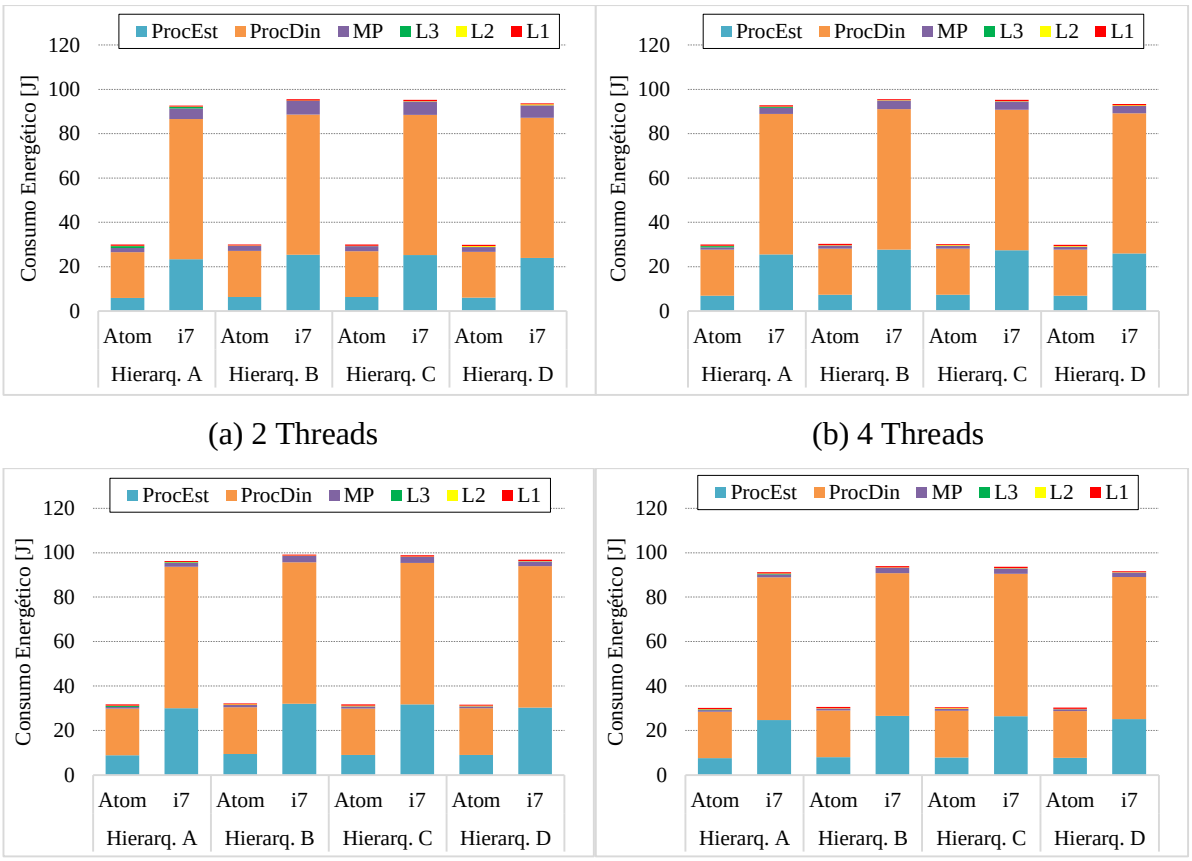
(c) Hierarquia C

(d) Hierarquia D

A Figura 5.12 apresenta os gráficos referente aos dados das Tabelas C.2, para aplicações do grupo *Weakly Memory-Bound*. Os gráficos apresentam o consumo energético total e de cada componente do sistema, para cada hierarquia de memória tanto para o processador Atom quanto para o i7. Nota-se como o aumento no número de acessos à memória compartilhada analisado na seção 5.1 teve efeito no consumo energético das hierarquias de memória. Embora o consumo total, assim como nas aplicações *CPU-Bound*, sejam diretamente dependentes do consumo do estático e dinâmico do processador, o consumo energético relativo da memória principal para cada hierarquia de memória teve variações consideráveis; a hieraquia A, por exemplo, reduziu cerca de 30% no consumo energético da memória principal em relação à hierarquia B pela utilização da memória *cache* L3 compartilhada. Além disso, como o aumento nos acessos à memória principal na hierarquia B e C também influenciou o tempo total de execução destas aplicações, conforme a seção 5.2, ocorreu um aumento de 7% no consumo estático do processador destas hieraquias em relação as demais. Para os parâmetros de entrada utilizados, não houve mudança significativa no tempo de execução e consumo energético devido a estes

fatores, entretanto, para simulações mais longas, é provável que tais variações impactem de forma relevante no resultado final de cada hierarquia de memória.

Figura 5.12 – Consumo Energético das Aplicações *Weakly Memory-Bound*



(a) 2 Threads

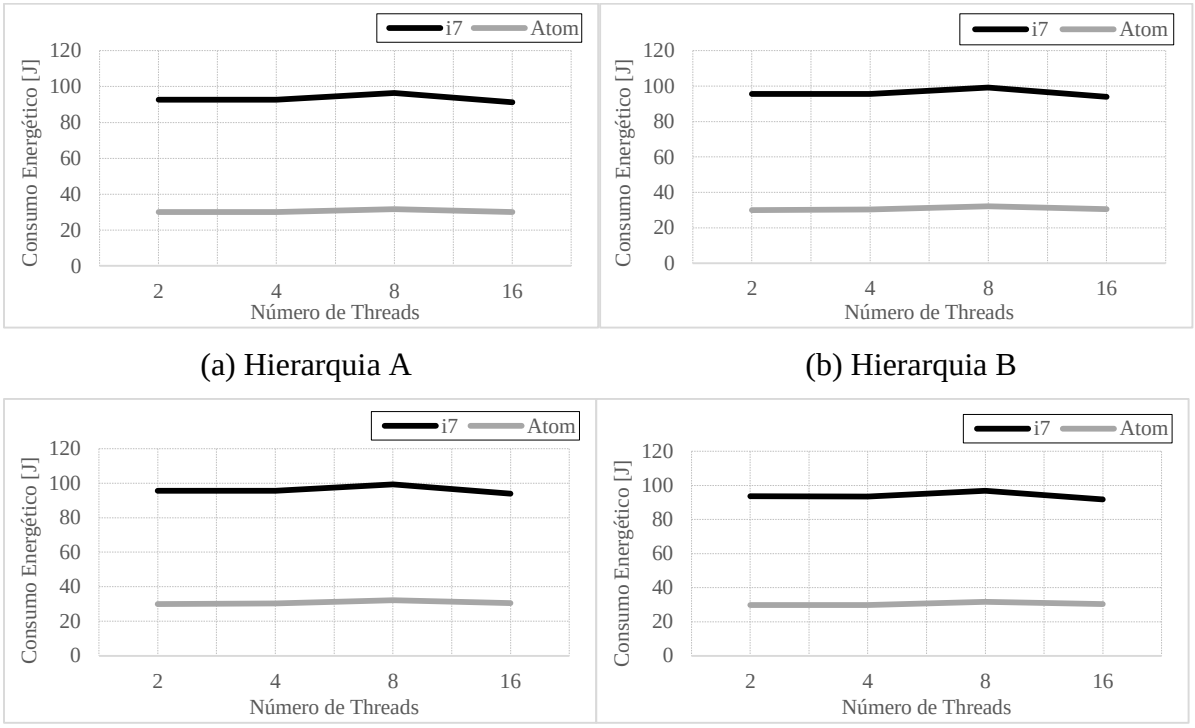
(b) 4 Threads

(c) 8 Threads

(d) 16 Threads

A Figura 5.13 apresenta os gráficos referente ao consumo energético total para cada hierarquia de memória em relação ao número de *threads* e aos processadores i7 e Atom. Observa-se, nos gráficos de consumo energético total da Figura 5.13, que a diferença entre o consumo energético do Intel Atom N2600 e do Intel Core i7 860 é praticamente constante assim como nas aplicações *CPU-Bound*. Os acessos à memória compartilhada para leitura de dados compartilhados não afetou significativamente o consumo energético total das aplicações para os parâmetros de entrada utilizados nos *benchmarks* deste grupo. Nota-se, contudo, uma economia em torno de 68% do Atom em relação ao i7 para todas as quantidades de *threads* analisadas, o que é bem semelhante ao encontrado para as aplicações *CPU-Bound* (de 66%).

Figura 5.13 – Consumo Energético Total das Aplicações *Weakly Memory-Bound*



(a) Hierarquia A

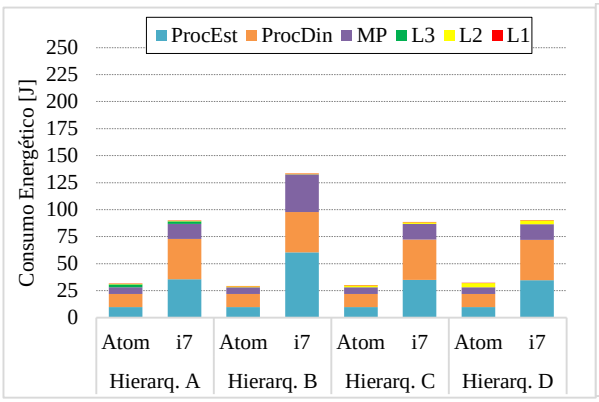
(b) Hierarquia B

(c) Hierarquia C

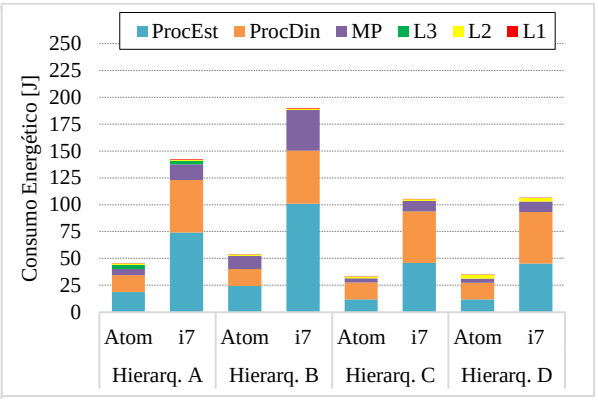
(d) Hierarquia D

A Figura 5.14 apresenta os gráficos referente aos dados das Tabelas C.3, para aplicações do grupo *Memory-Bound*. Com o aumento do compartilhamento de dados entre *threads*, pode-se observar uma maior significância da hierarquia de memória no consumo energético total das aplicações do grupo *Memory-Bound*. Até duas threads, nota-se como a hierarquia C e D possuem melhores resultados por utilizar a cache L2 para o compartilhamento de dados; a hierarquia B, por utilizar a memória principal para o compartilhamento, consome cerca de 20% do consumo energético total para esta memória, enquanto as demais consomem cerca de 10%, considerando o caso de duas *threads*. A partir de duas threads, o compartilhamento da hierarquia C passa a ser também pela memória principal, aumentando o consumo nesta memória. Com a diminuição do tempo de execução das aplicações por causa do aumento no número de threads, nota-se também uma gradual redução no consumo energético da memória principal nas hierarquias A e D devido à diminuição do consumo estático.

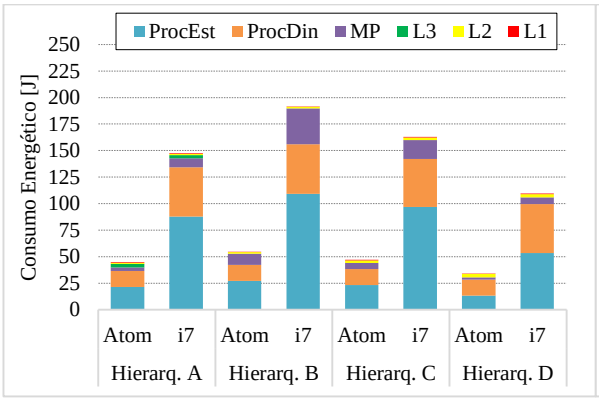
Figura 5.14 – Consumo Energético das Aplicações *Memory-Bound*



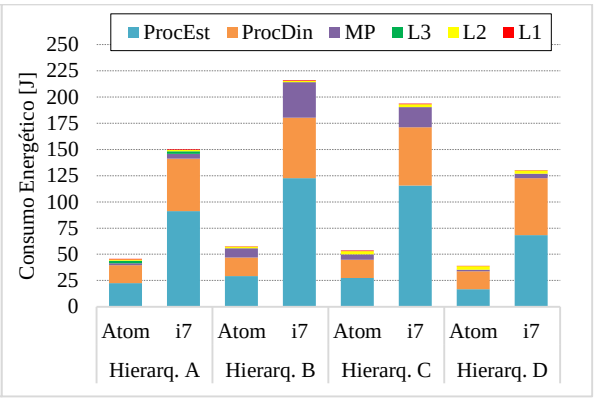
(a) 1 Thread



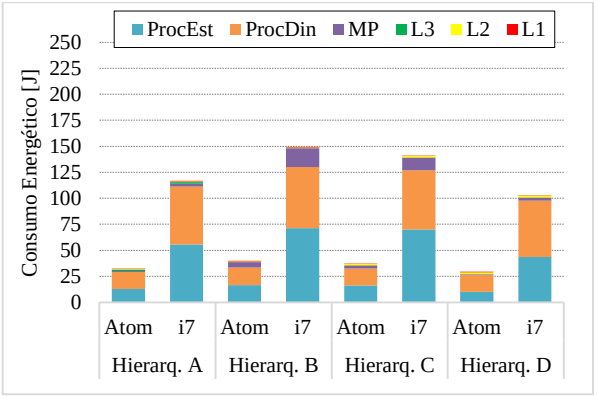
(b) 2 Threads



(c) 4 Threads



(d) 8 Threads



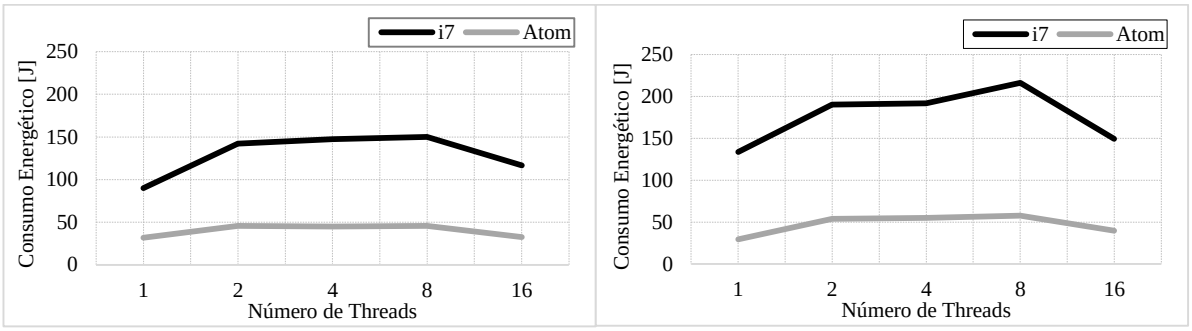
(e) 16 Threads

Em relação ao crescimento no número de *threads*, apresentado nos gráficos da Figura 5.15, nota-se um aumento no consumo energético total devido ao alto número de compartilhamentos, em contraste aos demais grupos que mantiveram seu consumo total praticamente constante. As hierarquias B e C, que utilizaram a memória principal, obtiveram, respectivamente, picos de 216 e 194 J no consumo energético total, enquanto as demais hierarquias obtiveram picos de 150 J (hierarquia A) e 130 J (hierarquia D). Quanto aos processadores, como esperado, o Intel Atom N2600 mostrou maior economia em relação ao

Intel Core i7 860, chegando a uma diferença de 73% para o caso de maior consumo energético (8 *threads* na hierarquia B). Na execução sequencial, onde não há compartilhamento de dados, a diferença entre os processadores mantem-se praticamente constante, exceto pela hierarquia B que possui memória *cache* L2 privada por *core*, causando *misses* que resultaram em acessos à memória principal e, consequentemente, em maior consumo energético.

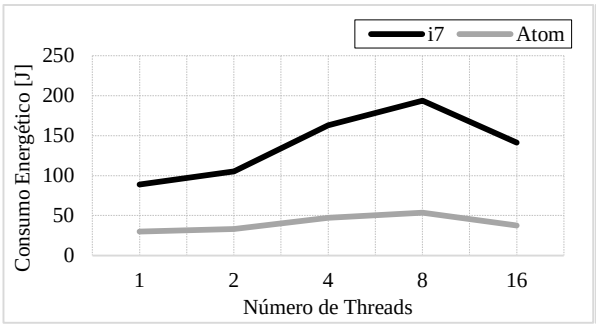
Nota-se, portanto, que o aumento no número de *threads* influencia diretamente o consumo energético das aplicações, uma vez que cresce o número de dependências de dados. Para o caso de 16 *threads*, há uma redução no consumo energético causada pelo fato de que comunicações entre *threads* de mesmo *core* são efetuadas na própria *cache* L1, uma vez que ambas as *threads* compartilham o mesmo espaço de endereçamento.

Figura 5.15 – Consumo Energético Total das Aplicações *Memory-Bound*

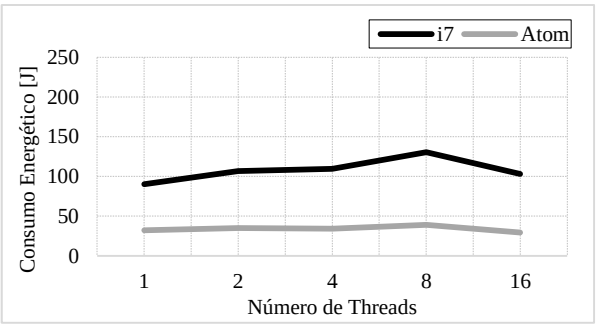


(a) Hierarquia A

(b) Hierarquia B



(c) Hierarquia C



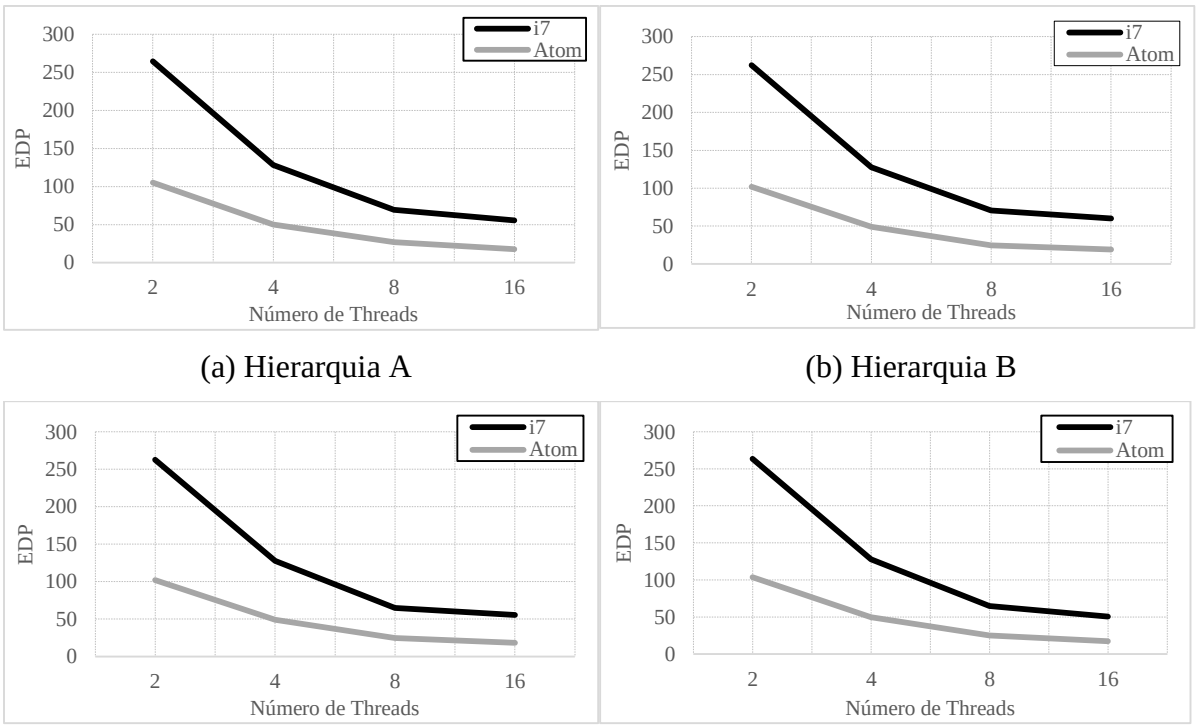
(d) Hierarquia D

5.4 *Análise de EDP (Energy-Delay Product)*

Através do produto entre o tempo de execução e o consumo energético dos *benchmarks*, conforme introduzido por Laros et. al (2013), é possível analisar o custo-benefício dos processadores Intel Core i7 860 e Intel Atom N2600 para cada hierarquia de memória trabalhada. No Apêndice D são apresentados os dados referente ao EDP para cada grupo de *benchmarks*: *CPU-Bound* (Tabelas C.1), *Weakly Memory-Bound* (Tabelas C.2) e *Memory-Bound* (Tabelas C.3), divididos entre processadores (*Atom*, representado o Intel Atom N2600, e *i7*, representando o Intel Core i7 860), número de *threads* e hierarquias de memória (A, B, C e D).

A Figura 5.16 apresenta os gráficos referente aos dados das Tabelas D.1, para aplicações do grupo *CPU-Bound*. Os gráficos apresentam o EDP para cada número de *threads*, separados em cada hierarquia de memória, tanto para o processador Atom quanto para o i7.

Figura 5.16 – EDP das Aplicações *CPU-Bound*



(a) Hierarquia A (b) Hierarquia B

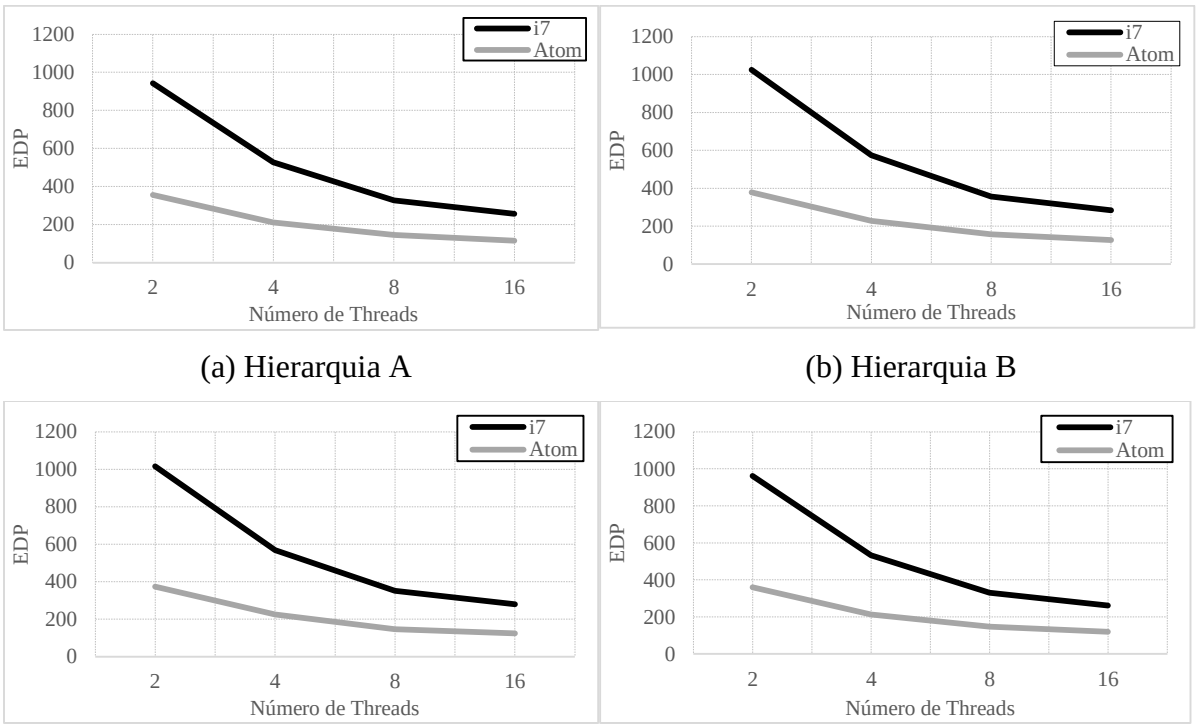
(c) Hierarquia C (d) Hierarquia D

Conforme ilustrado nos gráficos, o processador Atom possui melhor EDP que o processador i7 em todas as quantidades de *threads* analisadas uma vez que, embora os tempos de execução permaneçam com uma diferença em torno de 20%, o custo energético do Atom é 66% menor que o do i7. Entretanto, nota-se que consideramos a frequência de operação do

Atom como de 2,8Ghz ao invés de 1,6 Ghz como é originalmente, o que poderia impactar no consumo energético do Atom de forma negativa. Por fim, observa-se que a diferença no EDP permanece em torno dos 61% para todas as quantidades de *threads*, sem variação significativa entre hierarquias de memória.

A Figura 5.17 apresenta os gráficos referente aos dados das Tabelas D.2, para aplicações do grupo *Weakly Memory-Bound*. Os gráficos apresentam o EDP para cada número de *threads*, separados em cada hierarquia de memória, tanto para o processador Atom quanto para o i7. Nota-se que o EDP do Atom em relação ao i7 na execução com duas threads mantém-se em torno de 62% menor em todas as hierarquias. Entretanto, na execução com 16 *threads*, o EDP do Atom em relação ao i7 baixa para 54%, devido ao ganho em desempenho do i7 para os *benchmarks* utilizados neste grupo, conforme analisado na seção 5.2.

Figura 5.17 – EDP das Aplicações *Weakly Memory-Bound*



(a) Hierarquia A (b) Hierarquia B

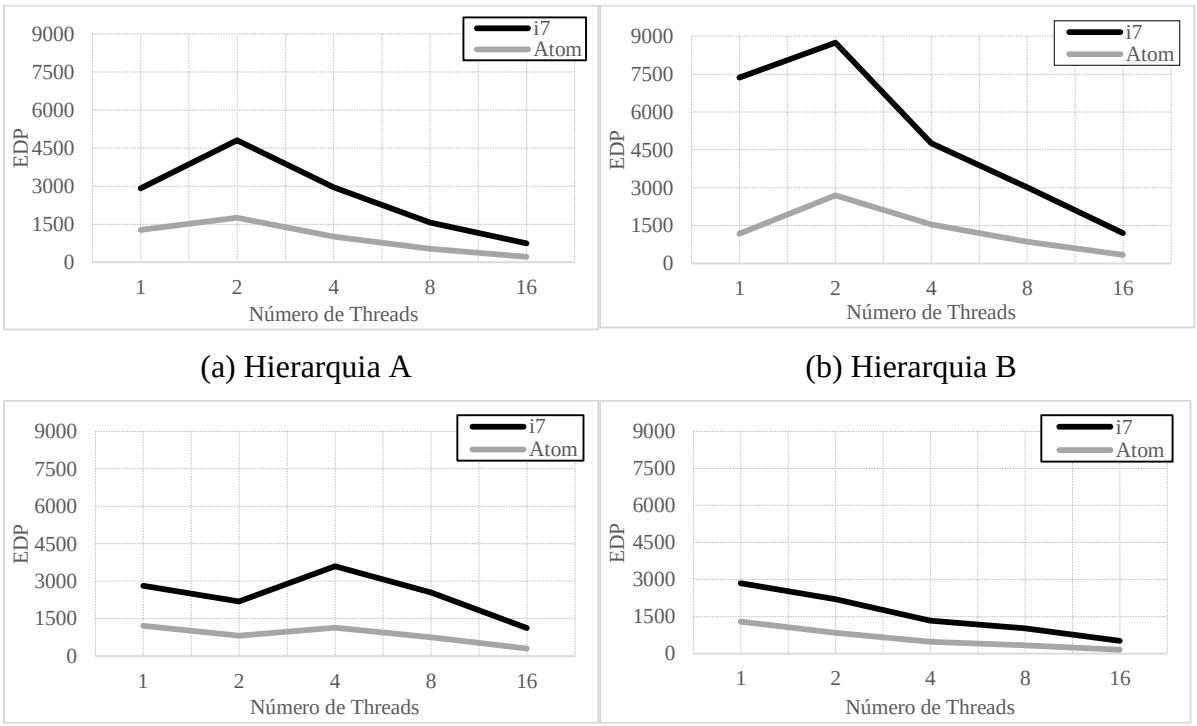
(c) Hierarquia C (d) Hierarquia D

A Figura 5.18 apresenta os gráficos referente aos dados das Tabelas D.3, para aplicações do grupo *Memory-Bound*. Os gráficos apresentam o EDP para cada número de *threads*, separados em cada hierarquia de memória, tanto para o processador Atom quanto para o i7. Observa-se que, para este grupo de aplicações, houve uma diferença significativa no EDP para cada hierarquia de memória. Na versão sequencial, o alto número de acessos à memória principal pela hierarquia B fizeram com que tanto seu tempo de execução quanto seu consumo

energético subissem, fazendo com que seu EDP chegasse a marca de 7356 para o i7, enquanto as demais hierarquias mantiveram-se em torno de 2850 neste mesmo processador (61% menor). Na execução em duas threads, onde ocorreu o maior pico de EDP da hierarquia B (igual a 8742), o Atom obteve uma diferença de 84% em relação ao i7, mostrando que a tecnologia utilizada pelo Atom voltada ao baixo consumo energético consegue amortizar o consumo da memória principal, que é de $15{,}6 \times 10^{-9}$ J por acesso para o i7 e $3{,}944 \times 10^{-9}$ J por acesso para o Atom, conforme a Tabela 4.6.

A medida em que se aumenta o número de *threads*, nota-se que a taxa de redução de EDP do i7 é menor que a do Atom, devido a alta utilização da hierarquia de memória (principalmente da memória principal). Comparando-se a diferença de EDP entre execução sequencial e paralela com 16 threads, há um ganho de 15% na diferença entre os processadores. Portanto, para aplicações com alto compartilhamento de dados, não somente o desempenho do Intel Atom N2600 é melhor, conforme analisado na seção 5.2, mas ele também possui melhor custo-benefício que o Intel Core i7 860.

Figura 5.18 – EDP das Aplicações *Memory-Bound*



(a) Hierarquia A (b) Hierarquia B

(c) Hierarquia C (d) Hierarquia D

6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho objetivou analisar o comportamento, consumo energético e desempenho de aplicações para processadores embarcados e de propósitos gerais com diferentes hierarquias de memória e números de *threads*. Para tanto, utilizou-se cinco *benchmarks* divididos igualmente em três grupos: *CPU-Bound*, com uso intensivo do processador e baixo compartilhamento de dados entre *threads*, *Weakly Memory-Bound*, com bastante uso do processador porém com acessos de leitura à memória compartilhada, e *Memory-Bound*, que possui dependência de dados e alto compartilhamento entre *threads*. Além disso, utilizou-se o simulador Multi2Sim para executar as aplicações nos processadores Intel Core i7 860, representando os processadores de propósitos gerais, e Intel Atom N2600, representando os processadores de sistemas embarcados, em quatro hierarquias de memória de características distintas. Por fim, variou-se o número de *threads* utilizadas para analisar a escalabilidade das hierarquias de memória para os diferentes tipos de processadores.

Os resultados obtidos mostraram que as hierarquias de memória não possuem impacto significativo em aplicações com baixa dependência e compartilhamento de dados. Para aplicações com alto compartilhamento, como previsto, houve um grande aumento nos acessos à memória principal das hierarquias B e C, o que cresceu a medida em que aumentava-se o número de *threads*. As hierarquias A e D minimizaram significativamente o uso da memória principal, para todos os números de *threads* utilizados, por possuírem, respectivamente, memória *cache* L3 e L2 compartilhada.

Em relação ao desempenho, a hierarquia D apresentou melhores resultados em aplicações com alto compartilhamento, uma vez que utilizava uma memória *cache* com menor latência para a comunicação dos dados entre *threads*; o pior resultado foi da hierarquia B, o qual realizava compartilhamento pela memória principal. Para aplicações *CPU-Bound*, entretanto, não houve diferença significativa entre as hierarquias de memória trabalhadas. Além disso, de modo geral, a escalabilidade das aplicações foi melhor para o processador Intel Atom N2600 do que para o Intel Core i7 860, o que apontou que a microarquitetura dos processadores perde significância a medida em que se aumenta o número de *threads* no sistema.

O consumo energético foi diretamente influenciado pelo número de acessos aos diferentes níveis da hierarquia de memória. Hierarquias como a B e a C, as quais utilizam a memória principal para o compartilhamento de dados, obteve um maior consumo em relação as demais hierarquias que possuem memória *cache* compartilhada. Em relação aos processadores, o Intel Atom N2600 garantiu um consumo energético 3 vezes menor para

aplicações *CPU-Bound* e 4 vezes menor para aplicações *Memory-Bound* em relação ao Intel Core i7 860.

Por fim, por meio do EDP, analisou-se o custo-benefício dos processadores Intel Core i7 860 e Intel Atom N2600 para cada hierarquia de memória e grupo de *benchmarks*. Em aplicações *CPU-Bound*, o Atom obteve EDP em torno de 61% menor que o i7, não havendo variação tanta para a hierarquia utilizada quanto para o número de *threads* da aplicação. Em aplicações *Weakly Memory-Bound*, a diferença entre Atom e i7 ficou praticamente a mesma das aplicações *CPU-Bound*, entretanto, como houve ganho de desempenho do i7 em relação ao Atom a medida que se aumenta o número de *threads* neste grupo de aplicações, a diferença de EDP baixou para 54% em execuções de 16 *threads*. Já em aplicações *Memory-Bound*, tanto a hierarquia de memória quanto o número de *threads* influenciaram no EDP. A hierarquia B obteve o maior EDP devido à alta utilização da memória principal, enquanto a hierarquias D obteve o EDP mais baixo por causa da memória *cache* L2 compartilhada entre todos os *cores*. Em relação ao número de *threads* neste grupo de aplicações, observou-se um aumento de 15% na diferença entre Atom e i7, mostrando que o custo-benefício do Atom aumenta em relação ao i7 para maiores quantidades de *threads* em aplicações *Memory-Bound*.

Portanto, conclui-se com este trabalho que hierarquias de memória com *cache* L2 compartilhadas possuem melhores resultados em questão de desempenho e consumo energético para os *benchmarks* e ambientes avaliados. Notou-se que o uso da memória principal como forma de compartilhamento de dados entre *threads* reduz significativamente o desempenho das aplicações *Memory-Bound*, além de aumentar seu consumo energético. Já para aplicações *CPU-Bound* e *Weakly Memory-Bound*, a hierarquia de memória não teve impacto significativo para ambos os quesitos. Por fim, mostrou-se que o processador Intel Atom N2600 possui um consumo energético muito inferior ao Intel Core i7 860; e, embora seu desempenho seja inferior para a maioria dos *benchmarks* utilizados, a medida em que se aumenta o número de *threads* no sistema, a diferença de desempenho reduz drasticamente, fazendo com que tenha melhor custo-benefício que o Intel Core i7 860.

É importante ressaltar, como última análise, que embora o processador Intel Atom N2600 utilizado neste trabalho tenha obtido um consumo energético muito superior ao Intel Core i7 860, o fato de a frequência de operação utilizada no Atom ser de 2,8Ghz e não de 1,6Ghz (frequência original), conforme informado na seção 4.4.1, pode acarretar mudanças no consumo energético deste processador e em seu respectivo EDP.

Como trabalhos futuros, pode-se destacar a execução de uma maior gama de *benchmarks* a fim de tornar os resultados menos tendenciosos a uma determinada característica

da aplicação. A extensão para simulações em 32 *threads* em conjunto com execuções com dados de entrada maiores também permitiriam analisar melhor a tendência tanto das hierarquias quanto da relação entre o desempenho, número de *threads* e tipo de processador. Além disso, a utilização de processadores de outras marcas e modelos forneceriam resultados mais genéricos, evitando assim que características específicas de processadores influenciem os resultados. Outros componentes que afetam diretamente as execuções como, por exemplo, a rede de interconexões e a área do *chip* também possui grande relevância aos resultados por criar um ambiente mais realista. Por fim, a utilização do simulador McPAT para análise do consumo energético seria de grande importância aos trabalhos futuros a fim de obter dados mais precisos e confiáveis.

REFERÊNCIAS

ALVES, M.; FREITAS, H.; FLÁVIO, W.; NAVAUX, P.: **Influência do Compartilhamento de Cache L2 em um Chip Multiprocessado sob Cargas de Trabalho com Conjuntos de Dados Contíguos e Não Contíguos.** In: Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD), 2007.Anais. ..[S.l.:s.n.], 2007.

ALVES, M.; FREITAS, H.; NAVAUX, P.: **Investigation of Shared L2 Cache on Many-Core Processors.** In: Workshop on Many-Cores, 2009, Berlin. Proceedings. .. VDE Verlag GMBH, 2009. p.21-30.

ANDREWS, G.E.; ASKEY, R.; ROY, R.: **Special Functions.** Cambridge University Press. 1999.

AYGUADÉ, E.; COPTY, N. C.; TERUEL, X.: **The Design of OpenMP Tasks.** IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 3, pp. 404-418. 2009.

BAILEY, D.; BARSZCZ, E.: **The NAS Parallel Benchmarks.** RNR Technical Report.1994.

BARTON, C.; CHEN, S. AND CHEN, Z.: **The Multi2Sim Simulation Framework.** Manual do Multi2Sim v.4.2, revisão 357.2013.

BERRY, I.; ROMAN, T.; ADAMS, L.: **The Cacti Manual.** Manual do Cacti Tool. 2013.

BLAKE, G.; DRESLINSKI, R.G.; MUDGE, T.; FLAUTNER, K.: **Evolution of Thread-Level parallelism in Desktop Applications.** In: 37th Annual International Symposium on Computer Architecture, 2010, Proceedings. New York, NY, USA: ACM, 2010. p.302-313.

BLEM, E. R.; MENON, J.; SANKARALINGAM, K.: **Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures.** In: HPCA, pp. 1-12. 2013.

CARVALHO, C.: **The Gap between Processor and Memory Speeds.** In Proc. of IEEE International Conference on Control and Automation. ICCA, 2002.

CHAPMAN, B.; JOST, G.; VAN DER PAS, R.: **Using OpenMP: Portable Shared Memory Parallel Programming.** Cambridge, MIT Press, 2008.

CHENEY, W.; KINCAID, D.: **Linear Algebra: Theory and Applications.** [S.l:s.n.], pp. 544-558.

DIJKSTRA, E. W.: **A Note on two Problems in Connection with Graphs.** Numerische Mathematik, [S. l.], vol. 1, nº 1, p.269-271. Dec. 1959.

FOSTER, T.: **Designing and Building Parallel Programs – Concepts and Tools for Parallel Software Engineering.** Addison-Wesley Press. 1995

GAO, C.; GUTIERREZ, A.; DRESLINSKI, R.G.: **A Study of Thread Level Parallelism on Mobile Devices.** In: IEEE ISPASS, March 2014, Proceedings. Monterey, CA: IEEE, 2014 pp. 126-127.

GARDNER, M.: **Mathematical Games – The Fantastic Combinations of John Conway's new Solitaire Game of Life.** Scientific American, [S.l.] vol. 223, nº 5, p. 120-123. Oct. 1970.

GEPNER, P; KOWALIK, M.: **Multi-Core Processors: New Way to Achieve High System Performance.** In.: Parallel Computing in Electrical Engineering (PARELEC) 2006, pp. 9-13. 2006

GOLDSTON, D.; YILDIRIM, C. Y.: **On the Second Moment for Primes in an Arithmetic Progression.** Mathematics – Number Theory. 2000.

HENNESSY, J. L; PATTERSON, D. A.: **Computer Architecture – A Quantitative Approach** (5ª ed.). Morgan Kaufmann. 2012.

INTEL: **Intel® Atom™ Processor D2000 and N2000 Series.** Datasheet, Volume 1. 2012.

INTEL: **Intel® Core™ i7-800 and i5-700 Desktop Processor Series.** Datasheet, Volume 1. 2010.

IQBAL, S. M.; YUCHEN, L.; GRAHN, H.: **ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems.** IEEEComputerArchitecture Letters; 9, 2; p.45-48. 2010.

JALEEL, A.; MATTINA, M.; JACOB, B.; **Last Level Cache (LLC) Performance of Data Mining Workloads On a CMP - A Case Study of Parallel Bioinformatics Workloads.** In: Int. Symp.on High-Performance Computer Architecture, 2005.Proceedings. .. IEEE, 2005.p.88-98.

JI, J.; WANG, C.; ZHOU, X.: **System-Level Early Power Estimation for Memory Subsystem in Embedded Systems.** Fifth IEE International Symposium on Embedded Computing, pp. 370-375, 2008.

KORTHIKANTI, V. A.; AGHA, G.: **Towards Optimizing Energy Costs of Algorithms for Shared Memory Architectures.** In: 22nd ACM SPAA, 2010, Proceedings. New York, NY, USA: ACM, 2010. p.157-165.

KUMAR, R.; ZYUBAN, V.; TULLSEN, D.: **Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling.** In: ISCA: Int. Symp. on Computer Architecture,2005.Proceedings. .. IEEE, 2005.p.408-419.

LAROS, J.H.I; PEDRETTI, K.; KELLY, S.M.: **Energy-efficient high performance computing.** London: Springer. 2013.

LORENZON, A.; CERA, M.; BECK, A.: **Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General Purpose Systems.** Journal of Signal Processing Systems, 2014.

LUK, C-K.; COHN, R.; MUTH, R.; PATIL, H.; KLAUSER, A.; LOWNEY, G.; WALLACE, S.; REDDI, V. J.; HAZELWOOD, K.: **Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation.** In PLDI, 2005, Proceedings. New York, NY, USA: ACM, 2005. p. 190-200.

MARINO, M. D.: **32-core CMP with multi-sliced L2: 2 and 4 cores sharing a L2 slice.** In: SBAC-PAD: Int. Symp. on Computer Architecture and High Performance Computing, 2006a. Proceedings. .. IEEE, 2006a p.141-150.

MARINO, M. D.: **L2-Cache hierarchical organizations for multi-core architectures.** In: ISPA: Int. Symp. On Parallel and Distributed Processing and Applications, 2006b. Proceedings. .. IEEE, 2006b p.74-83.

MOLKA, D.; HACKENBERG, D.; SCHONE, R.; MULLER, M.S.: **Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System.** In 18th

International Conference on Parallel Architectures and Compilation Techniques. pp. 261-270, IEEExplore, 2009.

NAYFEH, B. A.; OLUKOTUN, K.; SINGHT, J. P.: **The Impact of shared-Cache Clustering in Small-Scale Shared-Memory Multiprocessors.** In: HPCA: Second Int. Symp. on high-Performance Computer Architecture, 1996. Proceedings. .. IEEE, 1996.p.74-84.

PORTERFIELD, A. K.; OLIVIEAR, S. L.; BHALACHANDRA, S.; PRINS, J.: **Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs.** In: IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum, May, 2013, Proceedings.Cambridge, MA: IEEE, 2013. p.884-891.

RAUBER, T.; RUNGER, G.: **Parallel Programming for Multicore and Cluster Systems.** Springer, 2010.

WALL, D.: **Limits of instruction-level parallelism.** In.: ASPLOS IV Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. New York, NY, USA. pp.176-188. 1991.

WILKES, M.: **The Memory Gap and the Future of High Performance Memories.** Technical Report, AT&T Laboratories Cambridge, April, 2001.

WILSON, G., V.; BAL, H. E.: **Using the Cowichan Problems to Assess the Usability of Orca.** IEEE Parallel and Distributed Technology: Systems and Applications, [S.l.], vol. 4, nº 3, p. 36-44, Fall 1996.

ZHAO, X. et al.: **Split Private and Shared L2 Cache Architecture for Snooping-based CMP.** In: ICIS: Int. Conf. on Computer and Information Science, 2007. Proceedings. .. IEEE, 2007.p.900-905.

APÊNDICE A – COMPORTAMENTO DAS HIERARQUIAS DE MEMÓRIA

A.1 – APLICAÇÕES CPU-BOUND

		Hierarquia A				Hierarquia B			Hierarquia C			Hierarquia D		
		L1	**L2**	**L3**	**MP***	**L1**	**L2**	**MP***	**L1**	**L2**	**MP***	**L1**	**L2**	**MP***
2 Threads	**Hits**	7,04E+09	3,90E+07	1,05E+04	0,00E+00	7,04E+09	3,90E+07	1,08E+04	7,04E+09	3,90E+07	3,95E+03	7,04E+09	3,90E+07	3,00E+00
	Misses	1,95E+07	1,21E+04	8,99E+03	8,99E+03	1,95E+07	1,22E+04	9,00E+03	1,95E+07	9,40E+03	9,00E+03	1,95E+07	9,00E+03	9,00E+03
	Acessos	7,06E+09	3,91E+07	1,95E+04	8,99E+03	7,06E+09	3,91E+07	1,98E+04	7,06E+09	3,90E+07	1,29E+04	7,06E+09	3,90E+07	9,01E+03
4 Threads	**Hits**	6,96E+09	3,91E+07	1,64E+04	0,00E+00	6,97E+09	3,91E+07	1,60E+04	6,97E+09	3,91E+07	8,16E+03	6,97E+09	3,90E+07	3,50E+01
	Misses	1,95E+07	1,70E+04	1,06E+04	1,06E+04	1,95E+07	1,67E+04	1,06E+04	1,95E+07	1,31E+04	1,06E+04	1,95E+07	1,06E+04	1,06E+04
	Acessos	6,98E+09	3,91E+07	2,71E+04	1,06E+04	6,99E+09	3,91E+07	2,66E+04	6,99E+09	3,91E+07	1,88E+04	6,99E+09	3,91E+07	1,07E+04
8 Threads	**Hits**	7,49E+09	3,65E+07	4,95E+06	0,00E+00	7,51E+09	3,83E+07	1,32E+06	7,41E+09	3,91E+07	1,98E+04	7,40E+09	3,90E+07	2,06E+02
	Misses	1,95E+07	2,49E+06	1,37E+04	1,37E+04	1,95E+07	6,73E+05	1,37E+04	1,95E+07	2,22E+04	1,37E+04	1,95E+07	1,37E+04	1,37E+04
	Acessos	7,51E+09	3,90E+07	4,96E+06	1,37E+04	7,53E+09	3,90E+07	1,34E+06	7,42E+09	3,91E+07	3,36E+04	7,42E+09	3,91E+07	1,39E+04
16 Threads	**Hits**	7,61E+09	2,86E+07	2,10E+07	0,00E+00	7,64E+09	3,29E+07	1,25E+07	7,64E+09	3,67E+07	4,87E+06	7,57E+09	3,91E+07	3,56E+03
	Misses	1,96E+07	1,05E+07	2,04E+04	2,04E+04	1,96E+07	6,24E+06	2,03E+04	1,96E+07	2,45E+06	2,04E+04	1,95E+07	2,10E+04	2,04E+04
	Acessos	7,63E+09	3,91E+07	2,10E+07	2,04E+04	7,66E+09	3,91E+07	1,25E+07	7,66E+09	3,92E+07	4,89E+06	7,59E+09	3,91E+07	2,39E+04

* MP = Memória Principal

A.2 – APLICAÇÕES WEAKLY MEMORY-BOUND

		Hierarquia A				Hierarquia B			Hierarquia C			Hierarquia D		
		L1	**L2**	**L3**	**MP***	**L1**	**L2**	**MP***	**L1**	**L2**	**MP***	**L1**	**L2**	**MP***
2 Threads	**Hits**	1,48E+10	4,06E+07	7,28E+07	1,32E+07	1,48E+10	4,06E+07	7,95E+07	1,48E+10	4,11E+07	7,49E+07	1,48E+10	5,00E+07	5,45E+07
	Misses	3,98E+07	3,98E+07	6,73E+06	1,67E+05	3,98E+07	3,98E+07	1,67E+05	3,99E+07	3,76E+07	1,67E+05	3,88E+07	2,74E+07	1,67E+05
	Acessos	1,48E+10	8,03E+07	7,95E+07	1,34E+07	1,48E+10	8,04E+07	7,97E+07	1,48E+10	7,87E+07	7,51E+07	1,48E+10	7,74E+07	5,47E+07
4 Threads	**Hits**	1,53E+10	4,33E+07	7,30E+07	1,32E+07	1,52E+10	4,33E+07	7,94E+07	1,53E+10	4,47E+07	7,49E+07	1,53E+10	5,29E+07	4,98E+07
	Misses	3,99E+07	3,99E+07	6,72E+06	1,67E+05	3,98E+07	3,98E+07	1,67E+05	3,99E+07	3,76E+07	1,67E+05	3,91E+07	2,50E+07	1,67E+05
	Acessos	1,54E+10	8,32E+07	7,98E+07	1,33E+07	1,52E+10	8,32E+07	7,96E+07	1,54E+10	8,22E+07	7,51E+07	1,54E+10	7,78E+07	4,99E+07
8 Threads	**Hits**	1,56E+10	4,60E+07	7,30E+07	1,32E+07	1,55E+10	4,60E+07	7,95E+07	1,56E+10	4,80E+07	7,51E+07	1,56E+10	5,46E+07	4,85E+07
	Misses	3,99E+07	3,99E+07	6,72E+06	1,67E+05	3,99E+07	3,99E+07	1,67E+05	4,00E+07	3,76E+07	1,67E+05	3,98E+07	2,44E+07	1,67E+05
	Acessos	1,57E+10	8,58E+07	7,97E+07	1,33E+07	1,56E+10	8,59E+07	7,97E+07	1,57E+10	8,57E+07	7,53E+07	1,57E+10	7,89E+07	4,87E+07
16 Threads	**Hits**	1,52E+10	6,11E+07	7,33E+07	1,32E+07	1,52E+10	6,11E+07	7,99E+07	1,52E+10	6,32E+07	7,56E+07	1,52E+10	6,89E+07	5,02E+07
	Misses	4,83E+07	4,01E+07	6,72E+06	1,68E+05	4,83E+07	4,01E+07	1,68E+05	4,85E+07	3,79E+07	1,68E+05	4,80E+07	2,52E+07	1,68E+05
	Acessos	1,52E+10	1,01E+08	8,00E+07	1,33E+07	1,53E+10	1,01E+08	8,01E+07	1,53E+10	1,01E+08	7,58E+07	1,52E+10	9,42E+07	5,04E+07

* MP = Memória Principal

A.3 – APLICAÇÕES MEMORY-BOUND

		Hierarquia A				Hierarquia B			Hierarquia C			Hierarquia D		
		L1	**L2**	**L3**	**MP***	**L1**	**L2**	**MP***	**L1**	**L2**	**MP***	**L1**	**L2**	**MP***
1 Thread	**Hits**	4,06E+09	3,86E+09	2,34E+08	0,00E+00	4,09E+09	3,61E+09	7,24E+08	4,06E+09	3,94E+09	6,16E+07	4,06E+09	4,12E+09	5,69E+07
	Misses	1,99E+09	1,17E+08	6,52E+04	6,52E+04	1,99E+09	3,62E+08	6,52E+04	1,99E+09	3,08E+07	6,52E+04	1,99E+09	2,95E+07	6,52E+04
	Acessos	6,05E+09	3,97E+09	2,34E+08	6,52E+04	6,08E+09	3,98E+09	7,24E+08	6,04E+09	3,97E+09	6,17E+07	6,04E+09	4,15E+09	5,69E+07
2 Threads	**Hits**	8,31E+09	4,83E+09	1,07E+09	0,00E+00	8,48E+09	4,77E+09	1,18E+09	9,35E+09	4,14E+09	6,30E+07	9,33E+09	4,12E+09	5,89E+07
	Misses	1,99E+09	5,34E+08	6,63E+04	6,63E+04	2,00E+09	5,86E+08	6,63E+04	2,18E+09	3,15E+07	6,63E+04	2,16E+09	2,95E+07	6,63E+04
	Acessos	1,03E+10	5,37E+09	1,07E+09	6,63E+04	1,05E+10	5,36E+09	1,18E+09	1,15E+10	4,17E+09	6,31E+07	1,15E+10	4,15E+09	5,89E+07
4 Threads	**Hits**	8,64E+09	5,14E+09	1,49E+09	0,00E+00	8,58E+09	5,09E+09	1,48E+09	8,81E+09	6,39E+09	5,33E+08	8,87E+09	4,49E+09	5,89E+07
	Misses	2,00E+09	9,55E+08	6,64E+04	6,64E+04	2,01E+09	9,32E+08	6,65E+04	4,10E+09	2,67E+08	6,65E+04	2,54E+09	2,95E+07	6,64E+04
	Acessos	1,06E+10	6,09E+09	1,49E+09	6,64E+04	1,06E+10	6,02E+09	1,48E+09	1,29E+10	6,65E+09	5,33E+08	1,14E+10	4,52E+09	5,90E+07
8 Threads	**Hits**	9,41E+09	4,83E+09	1,56E+09	0,00E+00	9,31E+09	4,72E+09	1,78E+09	9,21E+09	7,29E+09	8,61E+08	9,62E+09	4,53E+09	5,64E+07
	Misses	2,03E+09	1,02E+09	6,66E+04	6,66E+04	2,05E+09	1,14E+09	6,65E+04	4,90E+09	5,34E+08	6,65E+04	2,65E+09	2,83E+07	6,65E+04
	Acessos	1,14E+10	5,85E+09	1,56E+09	6,66E+04	1,14E+10	5,86E+09	1,78E+09	1,41E+10	7,82E+09	8,61E+08	1,23E+10	4,56E+09	5,65E+07
16 Threads	**Hits**	1,18E+10	2,68E+09	9,35E+08	0,00E+00	1,20E+10	2,66E+09	9,48E+08	1,21E+10	3,71E+09	5,57E+08	1,22E+10	2,64E+09	5,95E+07
	Misses	1,15E+09	6,00E+08	6,68E+04	6,68E+04	1,17E+09	6,17E+08	6,68E+04	2,32E+09	3,32E+08	6,68E+04	1,58E+09	2,98E+07	6,68E+04
	Acessos	1,30E+10	3,28E+09	9,35E+08	6,68E+04	1,32E+10	3,28E+09	9,48E+08	1,45E+10	4,04E+09	5,57E+08	1,38E+10	2,67E+09	5,95E+07

* MP = Memória Principal

APÊNDICE B – DESEMPENHO DAS APLICAÇÕES

B.1 – APLICAÇÕES CPU-BOUND

	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
Nº Threads	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
2	7,033	5,826	7,033	5,826	7,033	5,825	7,032	5,825
4	3,521	2,916	3,521	2,916	3,521	2,915	3,520	2,915
8	1,864	1,527	1,782	1,553	1,781	1,479	1,781	1,478
16	1,376	1,313	1,461	1,391	1,411	1,316	1,323	1,228

* Medido em segundos.

B.2 – APLICAÇÕES WEAKLY MEMORY-BOUND

	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
Nº Threads	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
2	11,892	10,164	12,599	10,733	12,476	10,667	12,050	10,267
4	7,070	5,688	7,522	6,006	7,428	5,956	7,148	5,709
8	4,593	3,402	4,882	3,589	4,631	3,558	4,656	3,419
16	3,858	2,819	4,140	3,011	4,074	2,988	3,936	2,855

* Medido em segundos.

B.3 – APLICAÇÕES MEMORY-BOUND

	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
Nº Threads	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
1	40,342	32,414	40,337	55,026	40,276	31,839	40,092	31,547
2	38,431	33,758	50,015	46,019	24,497	20,845	24,168	20,632
4	22,353	19,994	28,252	24,878	24,088	22,067	13,903	12,192
8	11,666	10,403	15,027	13,959	14,183	13,164	8,602	7,782
16	6,575	6,343	8,610	8,110	8,343	7,997	5,222	4,966

* Medido em segundos.

APÊNDICE C – CONSUMO ENERGÉTICO DAS APLICAÇÕES

C.1 – APLICAÇÕES CPU-BOUND

1 Thread								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,26	0,23	0,26	0,23	0,26	0,23	0,26	0,23
L2	0,08	0,07	0,08	0,07	0,19	0,16	0,56	0,47
L3	0,90	0,74	0,00	0,00	0,00	0,00	0,00	0,00
MP	2,14	5,12	2,14	5,12	2,14	5,12	2,14	5,12
ProcEst	3,48	13,09	3,48	13,09	3,48	13,09	3,48	13,09
ProcDin	9,52	29,05	9,52	29,05	9,52	29,05	9,52	29,05
Et	16,38	48,30	15,49	47,55	15,60	47,64	15,97	47,95

* Medido em Joules.

2 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,24	0,22	0,24	0,22	0,24	0,22	0,24	0,22
L2	0,08	0,07	0,08	0,07	0,10	0,08	0,29	0,24
L3	0,45	0,36	0,00	0,00	0,00	0,00	0,00	0,00
MP	1,07	2,50	1,07	2,50	1,07	2,50	1,07	2,50
ProcEst	3,47	12,81	3,47	12,81	3,47	12,81	3,47	12,81
ProcDin	9,65	29,45	9,65	29,45	9,65	29,45	9,65	29,45
Et	14,96	45,41	14,51	45,04	14,53	45,06	14,72	45,22

* Medido em Joules.

4 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,24	0,22	0,24	0,22	0,24	0,22	0,24	0,22
L2	0,08	0,07	0,08	0,07	0,10	0,08	0,16	0,13
L3	0,22	0,18	0,00	0,00	0,00	0,00	0,00	0,00
MP	0,53	1,25	0,53	1,25	0,53	1,25	0,53	1,25
ProcEst	3,47	12,82	3,47	12,82	3,47	12,82	3,47	12,82
ProcDin	9,66	29,48	9,66	29,48	9,66	29,48	9,66	29,48
Et	14,21	44,02	13,99	43,84	14,01	43,85	14,07	43,89

* Medido em Joules.

8 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,26	0,23	0,26	0,23	0,26	0,23	0,26	0,23
L2	0,08	0,07	0,08	0,07	0,10	0,08	0,09	0,08
L3	0,13	0,10	0,00	0,00	0,00	0,00	0,00	0,00
MP	0,28	0,66	0,27	0,69	0,27	0,64	0,27	0,64
ProcEst	3,64	13,48	3,51	13,73	3,51	13,01	3,51	13,01
ProcDin	9,97	30,80	9,79	30,87	9,79	29,98	9,78	29,93
Et	14,37	45,35	13,91	45,59	13,92	43,95	13,91	43,88

* Medido em Joules.

16 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,23	0,22	0,23	0,22	0,23	0,22	0,23	0,22
L2	0,06	0,06	0,07	0,06	0,08	0,08	0,17	0,07
L3	0,10	0,11	0,00	0,00	0,00	0,00	0,00	0,00
MP	0,21	0,57	0,26	0,80	0,24	0,64	0,20	0,53
ProcEst	2,68	11,57	2,83	12,31	2,74	11,58	2,59	10,78
ProcDin	9,71	29,68	9,72	29,71	9,71	29,70	9,70	29,64
Et	12,98	42,20	13,10	43,11	13,00	42,22	12,89	41,24

* Medido em Joules.

C.2 – APLICAÇÕES WEAKLY MEMORY-BOUND

1 Thread								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,10	0,09	0,10	0,09	0,10	0,09	0,10	0,09
L2	0,03	0,03	0,03	0,03	0,08	0,07	0,23	0,20
L3	0,41	0,36	0,00	0,00	0,00	0,00	0,00	0,00
MP	1,00	2,56	0,96	2,45	0,96	2,45	0,96	2,45
ProcEst	1,54	6,03	1,47	5,73	1,47	5,72	1,47	5,72
ProcDin	6,03	18,38	6,03	18,38	6,03	18,38	6,03	18,38
Et	9,11	27,46	8,59	26,68	8,64	26,71	8,79	26,84

* Medido em Joules.

2 Threads								
	Hierarquia A		**Hierarquia B**		**Hierarquia C**		**Hierarquia D**	
	Atom	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
L1	0,50	0,48	0,51	0,48	0,51	0,48	0,50	0,48
L2	0,14	0,12	0,15	0,13	0,18	0,16	0,51	0,45
L3	0,86	0,76	0,00	0,00	0,00	0,00	0,00	0,00
MP	1,87	4,77	2,28	6,22	2,24	6,10	2,08	5,54
ProcEst	5,91	23,36	6,39	25,44	6,32	25,23	6,05	23,97
ProcDin	20,72	63,25	20,72	63,25	20,72	63,25	20,72	63,25
Et	30,00	92,74	30,05	95,53	29,98	95,23	29,87	93,68

* Medido em Joules.

4 Threads								
	Hierarquia A		**Hierarquia B**		**Hierarquia C**		**Hierarquia D**	
	Atom	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
L1	0,51	0,50	0,52	0,49	0,52	0,50	0,51	0,50
L2	0,16	0,13	0,17	0,14	0,21	0,18	0,32	0,27
L3	0,54	0,46	0,00	0,00	0,00	0,00	0,00	0,00
MP	1,11	2,71	1,45	3,94	1,42	3,85	1,27	3,31
ProcEst	6,89	25,60	7,39	27,65	7,30	27,36	6,98	25,92
ProcDin	20,79	63,36	20,79	63,37	20,79	63,37	20,78	63,35
Et	30,00	92,76	30,32	95,60	30,23	95,25	29,87	93,35

* Medido em Joules.

8 Threads								
	Hierarquia A		**Hierarquia B**		**Hierarquia C**		**Hierarquia D**	
	Atom	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
L1	0,53	0,51	0,54	0,51	0,53	0,51	0,53	0,51
L2	0,20	0,15	0,21	0,16	0,25	0,20	0,22	0,18
L3	0,38	0,31	0,00	0,00	0,00	0,00	0,00	0,00
MP	0,74	1,68	1,04	2,81	0,99	2,72	0,89	2,24
ProcEst	8,90	30,03	9,45	32,02	8,97	31,70	9,02	30,29
ProcDin	20,98	63,67	20,99	63,70	20,99	63,70	21,01	63,63
Et	31,74	96,34	32,23	99,20	31,73	98,84	31,67	96,85

* Medido em Joules.

16 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,51	0,48	0,51	0,48	0,51	0,48	0,51	0,48
L2	0,18	0,13	0,19	0,14	0,23	0,18	0,21	0,16
L3	0,34	0,27	0,00	0,00	0,00	0,00	0,00	0,00
MP	0,63	1,42	0,94	2,55	0,91	2,47	0,80	2,01
ProcEst	7,57	24,75	8,05	26,60	7,92	26,38	7,69	25,11
ProcDin	20,91	64,12	20,90	64,13	20,90	64,13	21,06	63,96
Et	30,13	91,17	30,58	93,90	30,47	93,63	30,27	91,73

* Medido em Joules.

C. .3 – APLICAÇÕES MEMORY-BOUND

1 Thread								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,24	0,24	0,24	0,27	0,24	0,23	0,24	0,23
L2	0,74	0,70	0,74	0,82	1,62	1,51	3,90	3,70
L3	2,65	2,30	0,00	0,00	0,00	0,00	0,00	0,00
MP	6,01	13,91	6,26	34,89	6,25	14,62	6,21	14,42
ProcEst	9,76	35,57	9,76	60,39	9,75	34,94	9,70	34,62
ProcDin	12,24	37,33	12,24	37,33	12,24	37,33	12,24	37,33
Et	31,64	90,04	29,24	133,70	30,09	88,64	32,29	90,30

* Medido em Joules.

2 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,42	0,40	0,45	0,44	0,40	0,39	0,40	0,39
L2	1,12	1,07	1,24	1,20	1,46	1,43	3,38	3,29
L3	3,80	3,37	0,00	0,00	0,00	0,00	0,00	0,00
MP	5,73	14,48	12,05	38,09	3,91	9,93	3,83	9,77
ProcEst	18,60	74,10	24,21	101,01	11,86	45,75	11,70	45,29
ProcDin	15,89	48,85	15,98	49,24	15,67	47,90	15,67	47,93
Et	45,55	142,28	53,93	189,98	33,30	105,41	34,98	106,67

* Medido em Joules.

4 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,45	0,43	0,48	0,45	0,51	0,49	0,41	0,40
L2	1,29	1,23	1,40	1,32	2,47	2,42	3,17	3,20
L3	3,30	3,01	0,00	0,00	0,00	0,00	0,00	0,00
MP	3,33	8,58	10,58	33,78	5,80	17,79	2,31	6,15
ProcEst	21,64	87,77	27,35	109,22	23,32	96,87	13,46	53,52
ProcDin	14,94	46,42	15,02	46,86	15,07	45,35	14,91	46,17
Et	44,94	147,45	54,83	191,63	47,16	162,92	34,25	109,45

* Medido em Joules.

8 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,48	0,46	0,53	0,50	0,58	0,55	0,47	0,45
L2	1,28	1,22	1,40	1,36	2,88	2,86	3,19	3,06
L3	2,62	2,50	0,00	0,00	0,00	0,00	0,00	0,00
MP	1,74	4,46	8,84	33,79	5,32	19,08	1,51	4,22
ProcEst	22,59	91,34	29,09	122,56	27,46	115,58	16,65	68,33
ProcDin	16,94	50,02	17,84	57,79	17,31	55,63	17,17	54,35
Et	45,65	150,00	57,70	215,99	53,56	193,69	39,00	130,40

* Medido em Joules.

16 Threads								
	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
	Atom	i7	Atom	i7	Atom	i7	Atom	i7
L1	0,42	0,43	0,45	0,46	0,48	0,49	0,42	0,44
L2	0,68	0,70	0,77	0,78	1,55	1,53	1,82	1,81
L3	1,49	1,50	0,00	0,00	0,00	0,00	0,00	0,00
MP	0,98	2,72	4,98	18,26	3,15	12,11	1,01	3,06
ProcEst	12,73	55,69	16,67	71,20	16,15	70,21	10,11	43,60
ProcDin	16,27	55,70	16,91	58,72	16,25	56,87	15,96	54,11
Et	32,58	116,75	39,78	149,42	37,58	141,22	29,33	103,02

* Medido em Joules.

APÊNDICE D – EDP (*ENERGY-DELAY PRODUCT*)

D.1 – APLICAÇÕES CPU-BOUND

	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
Nº Threads	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
1	231,3619	574,3982	218,6945	565,5565	220,2099	566,619	225,4544	570,301
2	105,2159	264,5273	102,0742	262,4168	102,2027	262,4902	103,5466	263,4027
4	50,04593	128,3304	49,26429	127,8158	49,32202	127,8387	49,51951	127,9579
8	26,79347	69,26726	24,78429	70,78896	24,80169	64,97585	24,76043	64,84648
16	17,85988	55,42299	19,14617	59,97158	18,35526	55,5588	17,0571	50,62389

D.2 – APLICAÇÕES WEAKLY MEMORY-BOUND

	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
Nº Threads	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
1	115,8707	301,6932	104,5114	278,3631	105,0051	278,6159	106,8344	279,8011
2	356,6955	942,6107	378,5678	1025,26	374,0422	1015,816	359,9346	961,8195
4	212,0615	527,5779	228,0839	574,1751	224,5752	567,3651	213,4996	532,9201
8	145,7663	327,7722	157,3717	355,9883	146,9353	351,6525	147,44	331,1648
16	116,2233	256,9923	126,6041	282,7627	124,118	279,7535	119,1704	261,8936

D.3 – APLICAÇÕES MEMORY-BOUND

	Hierarquia A		Hierarquia B		Hierarquia C		Hierarquia D	
Nº Threads	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**	**Atom**	**i7**
1	1276,588	2918,634	1179,514	7356,757	1211,857	2822,104	1294,566	2848,788
2	1750,635	4802,939	2697,303	8742,642	815,6862	2197,233	845,4176	2200,815
4	1004,615	2948,001	1549,176	4767,365	1135,877	3595,154	476,2102	1334,345
8	532,5378	1560,419	867,0741	3015,04	759,6363	2549,625	335,4855	1014,746
16	214,1746	740,4853	342,5114	1211,795	313,5057	1129,262	153,1276	511,6182

APÊNDICE E – CONFIGURAÇÃO INTERNA DOS PROCESSADORES

	Intel Core i7 860	**Intel Aton N2600**
Frequência	2,8 Ghz	2,8 Ghz
Número de *Cores*	4	4
Número de *Threads* por *Core*	2	2
Tamanho de Página	4 kB	4 kB
Unidades de *Decode*	4	2
Unidades de *Dispatch*	6	4
Unidades de *Issue*	4	2
Tamanho da Fila de Microinstruções	28	32
Tamanho do *Reorder Buffer*	128	64
Tamanho da Fila de Instruções	36	32
Tamanho da Fila de *Load-Store*	30	20
Tipo de Previsão de Desvios	Dois Níveis	Dois Níveis
Número de Conjuntos da BTB	1024	32
Associatividade da BTB	16	4
Número de Entradas do Preditor	4096	1024
Penalidade na Previsão de Desvios Incorreta	17 ciclos	13 ciclos
Tipo de Recuperação de Previsão Incorreta	Write Back	Write Back