Paralelismo em nível de thread

Camila Satsu de Amorim Yokoigawa
(107006)
Instituto de Computação da UNICAMP
Av. Albert Einstein, 1251, Cidade Universitária
Campinas, SP, Brasil
+55 19 3521 5838
camila.satsu@gmail.com

RESUMO
A tendência da indústria pela predileção por sistemas computacionais de alto desempenho, econômicos em relação a gasto de energia e custo baixo de projeto resultou em pesquisas, desenvolvimento e lançamento de sistemas computacionais com vários núcleos de processamento de arquitetura simples, baixa frequência e alto desempenho. Desta forma, tornando a exploração de paralelismo em nível de thread (TLP) atraente para se alcançar altas performances. São exemplos os processadores da família Intel Xeon, processadores que exploram TLP por meio da tecnologia *Hyper-Threading*, e família Sun UltraSPARC, que exploram TLP por meio da tecnologia *Chip Multithreading*. Além, disso, técnicas de exposição de TLP em aplicações que oferecem um alto grau de abstração para programadores, como o OpenMP, auxiliam ainda mais no aumento de performance por meio de exploração de TLP.

Categorias e Descritores de Assunto
C.1.2 [**Arquitetura de Computadores**]: Arquiteturas de múltiplos *streams* de dados.

Termos Gerais
Modelagem (*Design*).

Keywords
Paralelismo em nível de *threads*, multiprocessadores.

1. INTRODUÇÃO
O aumento no tamanho de transistores, o crescimento de taxa de transferência de memória, políticas de cachês, necessidade nos gasto com energia, exploração de paralelismo em nível de instrução (ILP – *Instruction Level Parallelism*) chegam a um limite no qual não é mais possível crescer o desempenho do processamento dentro das máquinas.

Neste sentido, houve a necessidade de se explorar o paralelismo que se encontra em um nível acima do nível de instrução, o paralelismo em nível de thread (TLP - *Thread-Level Parallelism*).

IC-UNICAMP MO401A, Month 5-7, 2010, Campinas, SP, Brazil.

Um *thread* pode ser definido como um processo separado com suas próprias instruções e dados, ou seja, tanto podem representar um processo paralelo em um programa, como em programas independentes [1]. A exploração de paralelismo em nível de *thread* visa expor, reorganizar e distribuir *threads* para executá-las simultaneamente em processadores.

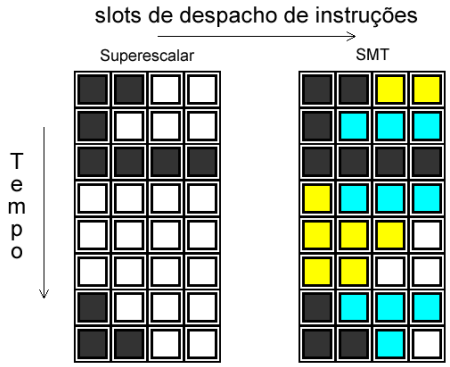

Figura 1. Duas técnicas diferentes que utilizam slots de despacho de um processador superscalar.

A figura 1 apresenta duas técnicas para comparação. No lado esquerdo da figura é representada uma máquina superescalar sem suporte a TLP, no lado direito é representada uma máquina superescalar com suporte a TLP por meio da técnica de SMT, técnica que explora simultaneamente TLP e ILP [1], permitindo um melhor aproveitamento dos recursos, pois, se o ILP for baixo, é despachada instruções de múltiplos *threads*. A dimensão horizontal representa a capacidade de instrução em cada ciclo de *clock*, a dimensão vertical representa uma sequência de *clocks*. Os quadrados brancos representam *slots* ociosos e os *slots* coloridos representam *threads* distintos.

É visível na figura 1 que o ILP explorado na máquina superescalar consegue oferecer ganho de desempenho, pois, utiliza vários recursos de *pipeline*, previsão de desvio, o escalonamento dinâmico, o desdobramento de *loop* entre outros. Porém, a existência de pouco paralelismo para exploração e dependências verdadeiras desnecessárias, por exemplo, podem causar longos *stalls*, deixando ociosos os recursos do processador. Por tanto, permitir que outro *thread* possa utilizar recursos ociosos causados por *stalls* de instruções resulta em aumento de desempenho.

Além disso, a tendência de aumentar desempenho das máquinas através da inserção de múltiplos processadores e o crescente interesse por servidores *web* e de banco de dados que recebem inúmeras requisições independentes [2], geram ambientes onde o paralelismo em nível de *thread* é bem mais alto do que o paralelismo em nível de instrução. E, por tanto, a exploração de TLP tendo peso maior no ganho de desempenho.

Nas seções seguintes são abordados os paradigmas SMT e CMP, a comparação de desempenho entre eles, a tecnologia *Hyper-Threading* da Intel, o processador *Niagara* da Sun, e aspectos gerais de ganho de desempenho e abordagens de exposição de TLP em alto nível de abstração em aplicações.

2. ARQUITETURAS COMERCIAIS

Em processadores comerciais, duas arquiteturas que exploram TLP destacam-se: *Multithreading* simultâneo (SMT – *Simultaneous Multithreading*) e Multiprocessamento em *chip* (CMP – *Chip Multiprocessor*).

2.1 SMT

A arquitetura SMT [3] consiste em explorar o paralelismo em nível de *thread* e em nível de instrução, resultando em processadores que despacham múltiplas instruções de múltiplos *threads* em cada ciclo.

SMT explora o ILP por seleção de instruções de *threads* que possam ser potencialmente despachadas. O processador esquematiza dinamicamente os recursos através das instruções, promovendo uma maior chance para um maior uso do *hardware*, ou seja, se um *thread* tiver um alto ILP, o paralelismo será satisfeito; se vários *threads* apresentarem baixo ILP, eles poderão ser executados juntos para compensar. A figura 1 traz uma representação simples de SMT.

A arquitetura SMT tem sua vantagem de ganho de desempenho devido ao melhoramento o *throughput* dos *threads*, pois, devido ao chaveamento dos *threads* e escolhas das instruções que usaram os recursos ociosos, há um aumento do *speedup* das instruções.

O *insight* [1] de SMT está no fato dos processadores atuais possuírem a implementação da maioria dos mecanismos necessários para a exploração dinâmica de TLP. Como, por exemplo, nas máquinas superescalares escalonadas dinamicamente, que possui um grande conjunto de registradores virtuais que mantém um conjunto de registradores das instruções dos *threads* independentes.

Entretanto, na literatura [1], é abordada a existência de desafios que a arquitetura SMT enfrenta, como: (a) lidar com um arquivo de registrador maior necessário para manter os múltiplos contextos; (b), não afetar o ciclo de *clock* devido a situações de escolha de próxima instrução a ser executada, por exemplo; (c) garantir a integridade dentro de um desempenho significativo, apesar dos conflitos de cachê e TLB gerados pela execução simultânea de múltiplos *threads*.

2.2 CMP

A arquitetura CMP (*Chip Multiprocessor*) é definida como um grupo de uniprocessadores integrados dentro de um mesmo *chip* de processamento atuando em conjunto [4].

A existência de vários núcleos dá suporte a paralelismo em nível de *thread* nativo aos processadores. Isso significa que, se o CMP dispuser de 8 núcleos escalares, é possível a existência de processamento de 8 *thread* simultâneos. Desta forma, não há impedimento que os núcleos utilizem superescalaridade em até mesmo, SMT, porém, provocando o aumento considerável na área de CMP.

A literatura [5] apresenta uma relação de vantagens da arquitetura CMP: (a) design simples, que permite rápido *clock* em cada unidade de processamento; (b) melhor aproveitamento da área de silício; (c) na perspectiva de software, é a plataforma ideal para a execução de *workload* (carga de trabalho) multiprogramado ou aplicações *multithreaded*.

2.3 COMPARAÇÃO

Embora ambos os paradigmas citados anteriormente promovam aumento de *throughput* para aumento de desempenho. Observou-se [6], por meio de comparações entre CMP e SMT, que, em termos de performance de CPU, a arquitetura CMP apresenta vantagem. Isto acontece devido à típica falta de contenção e execução entre *threads* existentes em SMT. Em contrapartida, SMT apresenta maior eficiência em relação à energia e memória, pois, SMT de único núcleo, tem suporte a cachês L2 de tamanho maior do que em chips múltiplos núcleos, além do fato da arquitetura CMP, devido à replicação de núcleos, apresentarem maior área e *overhead* de energia, além disso, a alta dissipação de energia faz com que o comportamento térmico e custos extras de refrigeração sejam fonte de grande preocupação.

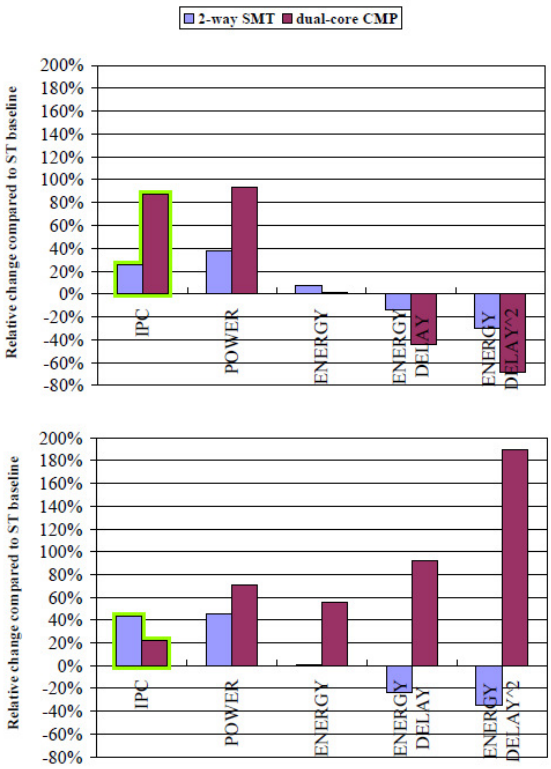

Figura 2. Comparação de performance e eficiência de energia entre SMT e CMP para *workload* com baixo (gráfico de cima) e alto (gráfico de abaixo) taxa de miss na L2.

A figura 2, extraída da literatura [6], representa bem a afirmação de que um paradigma pode superar outro dependendo de cada

situação. Nela é apresentado um estudo comparativo dos ganhos de desempenho e eficiência de energia entre SMT e CMP ambientados em uma microarquitetura POWER4. Onde ambos os paradigmas são analisados por meio de dois grupos de *benchmarks*: um grupo com um *miss rate* baixo na L2 (gráfico de cima) e, outro grupo, com um *miss rate* alto na L2 (gráfico de baixo).

Na comparação de desempenho, destacada em verde nos gráficos da figura 2, demonstra que, para *workloads* de baixo *miss rate* na L2, CMP impulsiona o *throughput* na ordem de 87% contra 27% do SMP. Entretanto, no gráfico de baixo, para *workloads* de alto *miss rate* na L2, CMP impulsiona *throughput* na ordem de 22% comparado com 42% do SMP. Isto acontece porque o *chip* CMP possui metade do tamanho de cachê L2 do SMP.

3. PROCESSADORES COMERCIAIS

Esta seção aborda duas tecnologias de processadores comerciais: O *Hyper-Threading* da Intel e o *Niagara* da Sun.

3.1 Hyper-Threading

Hyper-Threading [7] é uma tecnologia de arquiteturas criada pela Intel que se baseia em SMP. O conceito de *Hyper-Threading* foi primeiramente implementado nos processadores Intel Xeon, voltados para servidores, inicio dos anos 2000. Atualmente, está presente nos processadores Itanium, Core i7, Pentium IV, Xeon 5500, entre outros.

Nesta tecnologia, um único processador afigura-se como vários processadores para o nível lógico, que compreende aos sistemas operacionais e aplicações. Para tanto, cada processador lógico tem uma cópia do estado arquitetural e compartilha os recursos físicos, ou seja, compartilham os registradores de propósito gerais e de controle, alguns de máquinas de estado e controlador de interrupção, enquanto que, os recursos compartilhados, compreendem a cachês, unidades funcionais, barramentos, lógica de controle e previsão de desvios. Ou seja, do ponto de vista lógico, *threads* do mesmo programa ou de programas diferentes são alocados em diferentes processadores, enquanto que, no ponto de vista físico, esses *threads* são alocados em unidades funcionais compartilhadas.

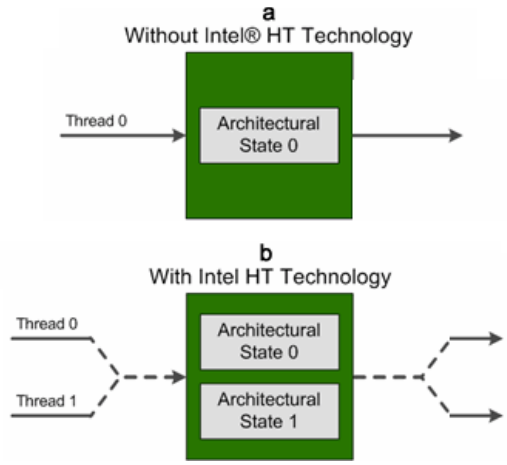

Figura 3. Ilustração de processadores sem Hyper-Threading (a) e com Hyper-Threading (b).

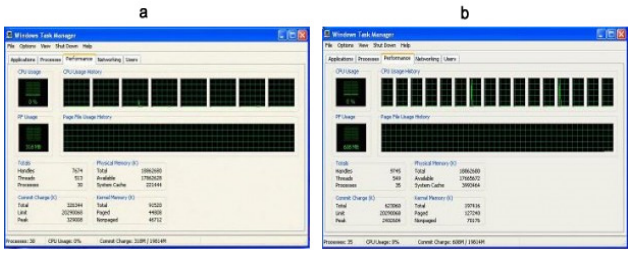

Figura 4. CPUs na mesma plataforma, com Hyper-Threading desativado (a) e ativado (b).

A figura 3, selecionada da literatura, mostra dois processadores. O processador (a) representa o ponto de vista físico, enquanto o processador (b) representa o ponto de vista lógico. Isto pode ser melhor visualizado através da figura 4, retirada da literatura [8], onde é exibida a tela *Task Manager* do *Windows* estando em uso processadores dual Xeon 5500 *series server*. Quando o *Hyper-Threading* está ativado (b), vê-se 16 CPUs, ou seja, 16 *threads* sendo executadas em 8 *cores* físicos, sendo 2 *threads* executados por *core*.

Para poder organizar os processadores lógicos, a tecnologia *Hyper-Threading* é estruturada por meio de um *front-end* e um *out-of-order execution engine* [7]. A organização desta estrutura é representada graficamente na figura 5.

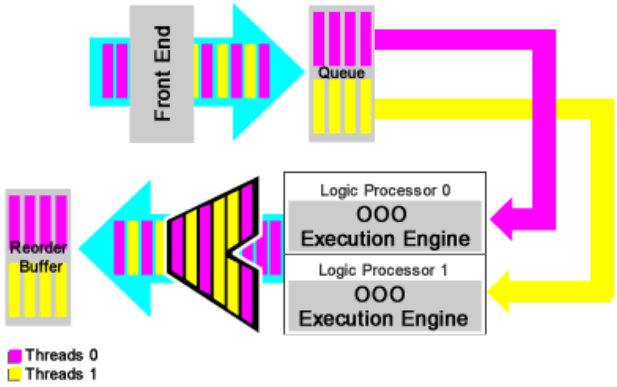

Figura 5. Representação, com alta abstração, da estrutura Hyper-Threading.

O *front-end* é responsável por entregar as instruções para o *pipeline*. É por meio do *front-end* que cada processador acessa uma posição do cachê primário alternadamente, além disso, para maior controle, cada posição possui uma *tag* que identifica o processador lógico que o possui.

Assim que as instruções são extraídas do cachê, o *front-end* os coloca numa fila que é dividida entre os processadores. Desta fila, chamada de *Uops Queue*. O *out-of-order execution engine* de cada processador lógico busca a instrução direcionada para si e preparam recursos e renomeiam registradores para despachá-las. São despachadas, no máximo, 6 instruções por meio de 5 despachantes de instruções para serem executadas misturadas em um mesmo processador físico. Após execução, as instruções são enviadas para uma fila de ordenação para serem manipuladas pelos seus respectivos processadores lógicos.

Ao comparar o Xeon com seu antecessor, Pentium 3, obteve-se um ganho de 65% em performance. Boa parte desse ganho se deve ao conceito de *Hyper-Threading* [7].

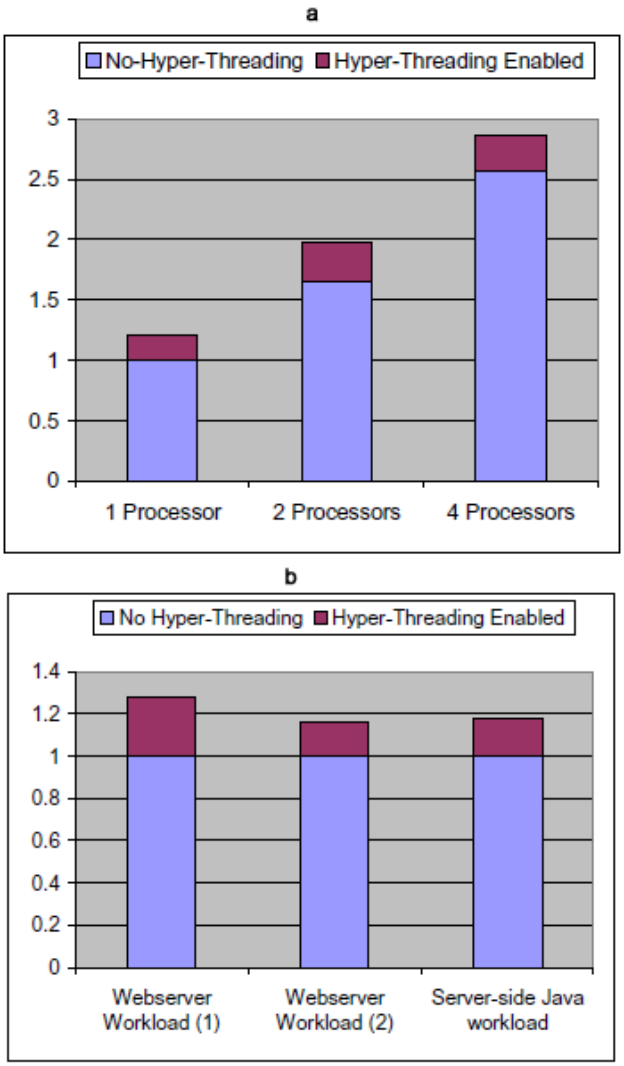

Figura 6. Comparação de performance obtida por meio da execução de *benchmarks* quando o Hyper-Threading está ativo e inativo considerando um Power4.

Na figura 6, é apresentado gráficos que comparam a performance obtida por meio da execução de *benchmarks* de OLTP ou processos de transações *online* (a), e *benchmarks* de servidor (b), em situações onde o *Hyper-Threading* é ativo ou inativo. Em ambos os gráficos é visto que o ganho gira em torno dos 25%.

É visível o fato de aplicações servidoras e de OLTP terem ILP alto, aproveitando melhor os recursos que o conceito *Hyper-Threading* oferece.

Sistemas Operacionais atuais, como o *Windows* 7, já oferecem suporte para trabalharem com o conceito *Hyper-Threading*, além das abordagens como OpenMP, Intel *Threading Building Blocks*, que otimizam o paralelismo em nível de *threads* de aplicações [9].

3.2 Niagara

Niagara é o codinome dado ao processador UltraSPARC T1, pertencente a família de processadores UltraSPARC. Este processador foi concebido para trazer uma abordagem diferente das tecnologias dual *core* implementadas nos processadores Intel e AMD [10].

A abordagem adotada pelo processador *Niagara* e por toda a extensa família UltraSPARC, chama-se *Chip Multithreading* (CMT) que consiste em tornar os processadores *multicore* e *multthreaded,* ou seja, unir vários cores (CMP) capazes de explorar TLP (SMT) e chavear threads para evitar ociosidade do *core* por causa do tempo de espera do acesso à memória (VMT – *Vertical Multi-Threading*) [10] [11], como ilustrado na figura 7.

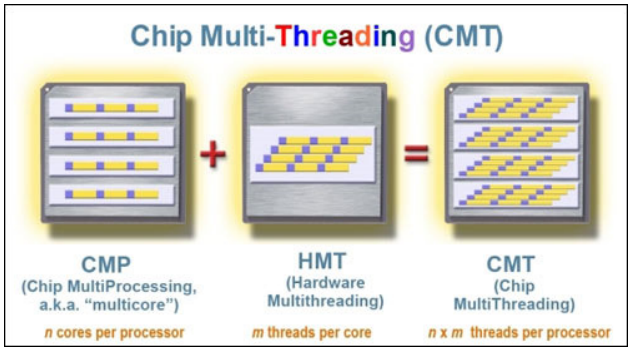

Figura 7. Representação gráfica do conceito CMT.

CMT melhora o desempenho de aplicações *multithreaded* porque executa seus *threads* paralelamente em cada ciclo sem a necessidade de mudança no código da aplicação. Entretanto, para tirar melhor proveito de CMT é necessário que os sistemas operacionais ofereçam suporte para ele, assim, sendo capazes de minimizar conflitos de recursos, e, maximizar o paralelismo e o *throughput* [11].

O processador *Niagara*, lançado no ano de 2005, é principalmente voltado para servidores, dentre as aplicações chaves estão as aplicações servidoras em Java e aplicações *Enterprise*, como o ERP [13]. Possui 8 núcleos que implementam o conjunto de instruções SPARC v9 [14] e são capazes de executar 4 *threads* por ciclo, totalizando 32 *threads* executados concorrentemente. O CPU usa em média 72W de força e trabalha a 1.4GHz. Possui a limitação de possuir apenas uma única unidade de ponto flutuante, sendo compartilhados por seus 8 núcleos resultando na capacidade de processamento de operações de ponto flutuante em torno de 1 a 3% [12].

A família de processadores UltraSPARC é extensa. Após o lançamento do *Niagara*, UltraSPARC T1, foi lançado em 2007 o *Niagara* 2 ou UltraSPARC T2, seu sucessor. Com novidades como a capacidade de executar simultaneamente 64 *threads*, 2 ALUs por core, 1 unidade flutuante por core, maior cachê L2, criptografia integrada ao core, consequêntemente, maior gasto de energia, 95 W, porém, maior desempenho [11] [12].

Anterior ao processador *Niagara*, existe ainda o processador *Cheetah* (UltraSPARC III) lançado em 2001, além de suas variações lançadas até o ano de 2005: *Cheetah+* (UltraSPARC III Cu), *Jalapeno* (UltraSPARC IIIi) e *Serrano* (UltraSPARC IIIi). Depois, ainda foram lançados em 2004 o *Jaguar* (UltraSPARC

IV) e, logo em seguida, sua variação, o *Panther* (UltraSPARC IV+).

Na literatura [15] foram realizados testes para a comparação de performance entre o Niagara (UltraSPARC T1) e outros processadores usando os *benchmarks* Specjbb2005, Specweb2005 e SAP 2-Tier.

System	CPU	Power Dissipation CPUs (Estimated)	Number of cores	Number of Active threads	Score	Percentage score
Sun Fire T2000	1x 1.2GHz UltraSPARC T1	72-79 W	8	32	63,378	160%
Sun Fire X4200	2x 2.4GHz DC Opteron	150-180 W	4	4	45,124	114%
IBM p5 550	2x 1.9GHz POWER5+	320-360 W	4	8	61,789	156%
IBM xSeries 346	2x 2.8GHz DC Xeon	270-300 W	4	8	39,585	100%

Figura 8. Comparação de performance entre processadores por meio do *benchmark* Specjbb2005.

O *benchmark* Specjbb2005 representa uma aplicação que processa pedidos de um fornecedor escrito em Java. A figura 8 mostra que o UltraSPARC T1 obteve uma performance alta. O próprio *benchmark* é ideal para este processador, pela presença de várias theads Java. Entretanto, o Power 5+ obteve uma performance muito próxima, ele também executa simultaneamente 8 *threads* como o UltraSPARC T1, porém, sua dissipação de energia é quase 4 vezes maior que a o UltraSPARC T1.

System	Processors	Power Dissipation CPUs (Estimated)	Number of cores	Number of Active threads	Score	Percentage score
Sun Fire T2000	1x 1.2GHz UltraSPARC T1	72-79 W	8	32	14,001	289%
IBM p5 550	2x 1.9GHz POWER5+	320-360 W	4	8	7,881	162%
IBM xSeries 346	2x 3.8GHz Xeon	220-260 W	4	4	4,348	90%
Dell 2850	2x 2.8GHz DC Xeon	260-300 W	4	8	4,85	100%

Figura 9. Comparação de performance entre processadores por meio do *benchmark* Specweb2005.

O *benchmark* Specweb2005 simula *workload* de sites de banco, *e-commerce* e suporte, que, naturalmente possui alto grau de independência entre *threads*, sendo um ambiente que favorece o processador UltraSPARC T1, que, como ilustrado na figura 9, obteve uma performace dramaticamente superior em relação aos outros processadores. Afinal, o UltraSPARC T1 é o processador que consegue executar o maior número de *threads* simultaneamente.

System	Processors	Power Dissipation CPUs (Estimated)	Number of cores	Number of Active threads	Score	Percentage score
Sun Fire T2000	1x 1.2GHz UltraSPARC T1	72-79 W	8	32	4780	97%
IBM p5 550	2x 1.9GHz POWER5+	320-360 W	4	8	5020	102%
HP DL580	4x 3.33GHz Xeon MP	440-520 W	4	8	4700	96%
HP DL385	2x 2.2GHz DC Opteron	140-180 W	4	4	4920	100%

Figura 10. Comparação de performance entre processadores por meio do *benchmark* SAP 2-Tier.

O SAP 2-Tier é baseado em aplicações ERP, que são tipicamente de baixo IPC (Instrução por ciclo), não permitindo que o UltraSPARC T1 tenha ganhos de desempenho maiores que processadores que possuem núcleos mais robustos, conforme visto na figura 10. Ainda assim, a performance por watt é imbatível.

4. SOFTWARES

Como citado nas seções anteriores, as aplicações voltadas para ambientes de servidores, como *e-commerce*, *internet banking* e gerenciadores de banco de dados, conseguem aproveitar o ganho de performance que os processadores oferecem explorando TLP [6][7].

Porém, não apenas em aplicações servidoras, como também, aplicações *desktop* conseguem aproveitar os ganhos de performance obtidos pela exploração de TLP.

Estudos [16] mostraram que o uso de dois processadores ao invés de um reduziu em média 22% do tempo de execução. A resposta das aplicações teve um aumento médio de 29% com um MP3 player executando em background, de um máximo possível de 50%. Esta melhora de desempenho em dois processadores se deve a exploração da taxa de TLP existentes nas aplicações.

Apesar dos compiladores conseguirem detectar e explorar o paralelismo, não é em todos os casos que o fazem de maneira eficiente. Na maioria deles é necessário que o paralelismo seja exposto para os compiladores.

Por não ser uma tarefa trivial, sistemas de programação paralela foram desenvolvidos. Os principais são: *Shared Memory, Distributed Memory, Data-parallel programming, Threads* [18].

Interfaces de programação (API) surgiram para deixar mais simples a tarefa de expor o paralelismo em nível de threads, colocando uma camada de abstração acima, e, consequentemente, deixando o programador livre de criação, gerenciamento e destruição de *threads*. Um exemplo é a API OpenMP[17][18], que provê uma interface flexível para o desenvolvimento de aplicações paralelas em multiprocessadores *shared memory*. OpenMP insere um aviso, chamando uma "diretiva", dentro de uma aplicação escrita em uma linguagem de programação convencional, C, C++ ou Fortran. O compilador OpenMP gera o código paralelizado. Recentemente, vários compiladores para computadores *desktop* e servidores tem suporte a OpenMP.

Como teste de performance, em [19] foi criado um programa que calcula π em duas versões: uma com implementação de diretivas OpenMP e outra sem executadas no compilador ICC (Intel) e GCC (Gnu).

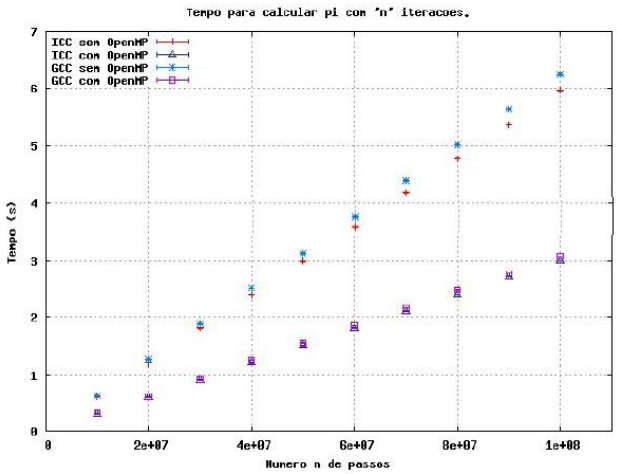

Figura 11. Desempenho de OpenMP no cálculo de π.

Como apresentado na figura 11, o resultado foi uma redução do tempo de execução pela metade quando implementado OpenMP.

Outros exemplos de APIs para exposição de paralelismo para compiladores com alto nível de abstração são: Threading Building Block [20], Intel, e Podix [21], IEEE.

5. CONCLUSÃO

Explorar o paralelismo em nível de *thread* das aplicações é muito vantajoso, trazendo ganhos de desempenho não apenas em servidores, que possui um ambiente que recebe muitas requisições independentes, como também em aplicações *desktop*. Em estudos [17], a redução do tempo de execução obtido por executá-los apenas em processadores com dois núcleos, sem qualquer otimização para exposição de paralelismo, foi em média 22%. Contudo, escrever o código de aplicações usando recursos que exponham o paralelismo em nível de thread para compiladores, e assim, aproveitar melhor as tecnologias suportada pelos processadores, como o OpenMP, é esperado que a redução do tempo de execução de aplicações reduza ainda mais. No trabalho [20] foi construído uma aplicação de cálculo de PI com e sem otimização OpenMP, com o intuito de testar a eficiência da exposição de TLP por meio de OpenMP. O resultado foi um ganho em performance por volta de 50% para a aplicação com otimização OpenMP.

Atualmente, as empresas de tecnologia têm oferecido ao mercado de computadores, desde *desktop* a robustos servidores, processadores *multithreading* e *multicore*, pois, perceberam a possibilidade de ganhos de performance a baixo custo energético e de projeto. Tudo isto graças a evolução dos *chips*, que estão sendo capazes de comportar cada vez mais núcleos, o que possibilitou o surgimento de arquiteturas, como SMT e CMP. Que deram origem a tecnologias importantes como o Hyper-Threading da Intel e CMT da Sun, uma tecnologia mais conhecida através do processador Niagara Sun que o implementa.

6. REFERENCES

[1] Hennessy, J. L. and Patterson, D. A. 2007. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc. Sao Mateo, CA. Forth Edition, 1995.

[2] Freitas, H. C., Alves, M. A. Z., Maillard N. B., Navaux P. O. A. 2008. Ensino de arquiteturas de processadores Multi-Core através de um sistema de simulação completo e da experiência de um projeto de pesquisa. In Workshop sobre Educação em Arquitetura de Computadores (Campo Grande, Brasil, Outubro – 29, 2008).

[3] Eggers S., Emer J., Levy H., Lo J., Stamm R., Tullsen D. 1997. Simultaneous Multithreading: A platform for next-generation processor. IEEE Micro. (Sep. 1997), 12-19.

[4] Olukotun K., Hammond L., Laudon J. 2007. Chip multiprocessor architecture: Tehcniques to improve throughput and latency. Morgan & Claypool publishers, Inc. San Rafael, CA. First Edition, 2007.

[5] Krishnan V., Torrellas J. A Chip-Multiprocessor architecture with speculative multithreading. IEEE transaction on computer. Vol. 48. No. 9. (Sep - 1999), 866-880.

[6] Skadron Y. L., Brooks K., Zhigang Hu D. Performance, energy, and thermal considerations for SMT and CMP architectures. High-performance computer architecture. 2005. HPCA-11. 11th International Symposium, pages 71-82, February 2005.

[7] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., Upton, M. Hyper-Threading Technology Architecture and Microarchitecture. 2002. Intel Technology Journal, vol. 6, issue 1. ITJ 2002.

[8] INTEL. 2010. Hyper-Threading technology: Your questions answered. DOI= Artigo de site. DOI= http://software.intel.com/en-us/blogs/2009/06/02/intel-hyper-threading-technology-your-questions-answered. Acessado em 10/06/2010.

[9] Valles, A., Gillespie, M., Drysdale, G. Performance insights to Intel Hyper-Threading Tecnology. DOI= http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/. Acessado em 10/06/2010.

[10] Lobo, S. Understanding Chip Multithreading (CMT) Technology. DOI=http://www.cxotoday.com/India/Features/Understanding_Chip_Multitreading_CMT_Technology/551-106442-932.html. Acessado em 11/06/2010.

[11] Nagarajayya, N. Improving application efficiency through chip multi-threading. DOI=http://developers.sun.com/solaris/articles/chip_multi_thread.html. Acessado em 11/06/2010.

[12] Sun Microsystems. OpenSparc T1 microarchitecture specification. DOI=http://opensparc-t1.sunsource.net/specs/OpenSPARCT1_Micro_Arch.pdf. Acessado em 11/06/2010.

[13] ORACLE. Sun Microelectronics. DOI=http://www.oracle.com/us/products/servers-storage/microelectronics/index.html. Acessado em 11/06/2010.

[14] Sparc International Inc. 1994. The sparc architecture manual. Prentice Hal Publishersl, Inc Englewood Cliffs, NJ. Frist Edition. 1994.

[15] De Gelas, J. 2005. SUN's UltraSparc T1 – the next generation Server CPUs. DOI= http://www.anandtech.com/print/1909. Acessado em 11/06/2010.

[16] Flautner, K., Uhlig, R., Reinhardt, S., and Mudge, T. 2000. Thread-level parallelism and interactive performance of desktop applications. SIGPLAN Not. 35, 11 (Nov. 2000), 129-138. DOI= http://doi.acm.org/10.1145/356989.357001.

[17] Küsel, R. A. M., Programação OpenMP. DOI=http://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/apostila_openmp.pdf. Acessado em 12/06/2010.

[18] Hanawa, T., Sato, M., Lee, J. Imada, T., Kimura, H., Boku, T. Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP. 2009. 5th International workshop on OpenMP, IWOMP 2009. Pages 15 - 27, June 2009.

[19] Gallina, L. Z. Avaliação de desempenho do OpenMP em arquiteturas paralelas. DOI=http://www.inf.ufrgs.br/~nicolas/pdf/tcc_gallina.pdf. Acesso em 13/06/2010.

[20] THREADING BUILDING BLOCKS. DOI=http://www.threadingbuildingblocks.org/documentation.php Acessado em 13/06/2010.

[21] OpenGroup. DOI=http://www.opengroup.org/austin/papers/posix_faq.html. Acessado em 13/06/2010.