Uma Proposta para Migrac ¸˜ao de P´aginas no Linux

Article

2 authors, including:

Avelino F. Zorzo
Pontifícia Universidade Católica do Rio Grande do Sul
138 PUBLICATIONS **1,212** CITATIONS

SEE PROFILE

Uma Proposta para Migração de Páginas no Linux

Guilherme A. A. Tesser[1], **Avelino F. Zorzo**[2]

[1]HP P&D Brasil

[2]PPGCC-FACIN-PUCRS
Av. Ipiranga, 6681, CEP: 90619-900 – Porto Alegre – RS – Brasil

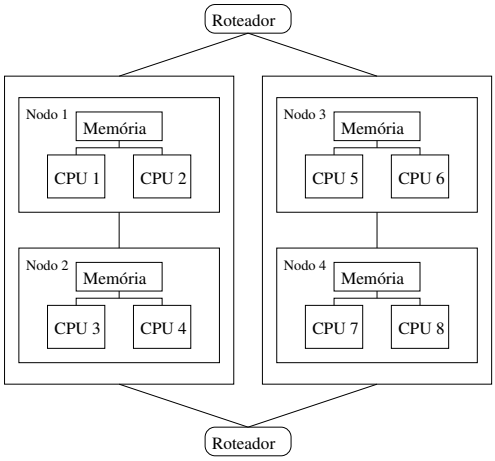
guilherme.tesser@hp.com, zorzo@inf.pucrs.br

Abstract. *This paper presents two algorithms to deal with memory page migration in the Linux Operating System. Currently Linux does not migrate memory pages when the load balancing algorithm migrates processes in a NUMA machine. This can degrade the overall system performance. One algorithm presented here is based on the idea of migrating the full process memory when the process migrates to a different computer node. The second algorithm migrates only memory pages when they are accessed by the process. The paper presents a thoroughly simulation analysis of the three strategies, i.e. full migration, on demand and the Linux current strategy.*

Resumo. *Este trabalho apresenta dois algoritmos para migração de páginas no Sistema Operacional Linux. Atualmente, o Linux não migra páginas da memória quando o algoritmo de balanceamento de carga migra um processo em uma máquina NUMA. Isto pode prejudicar o desempenho de todo sistema. Um algoritmo apresentado aqui é baseado na idéia de migrar toda a memória do processo quando o mesmo migra para um outro nodo da máquina. O segundo algoritmo migra somente as páginas de memória quando estas são acessadas pelo processo. Este trabalho apresenta uma análise com base em simulação das três estratégias: full migration, on demand e a estratégia atual do Linux.*

1. Introdução

A complexidade de cálculos para se obter resultados satisfatórios para várias questões, que nos dias de hoje são fundamentais para diferentes tipos de negócio exigem cada vez mais que tenhamos máquinas com grande poder computacional. Para questões como a previsão do tempo, simulação de trânsito, entre outras que se encaixam nesse perfil, o mercado oferece supercomputadores, *clusters* e máquinas multiprocessadas como algumas das possíveis soluções para esses usuários que necessitam de alta capacidade de processamento. Dentre essas soluções, as máquinas multiprocessadas são um tema de bastante estudo e pesquisa [Bircsak et al. 2000, Chapin et al. 1995, Tao Mu, Jie Tao, Martin Schulz 2003]. Máquinas multiprocessadas são computadores formados por um conjunto de processadores. Estas podem ser classificadas em dois tipos segundo o seu tempo de acesso à memória [Paterson and Hennessy 1998]: UMA (*Uniform Memory Access*), também conhecida como SMP (*Symetric Multiprocessor*), onde o tempo de acesso à memória é igual para todos os processadores, e NUMA (*Non-uniform Memory Access*), onde o tempo de acesso à memória depende do processador e da posição de memória acessada [Paterson and Hennessy 1998]. A Figura 1 mostra uma máquina



SLIT

Nodo	1	2	3	4
1	10	20	30	40
2	20	10	40	30
3	30	40	10	20
4	40	30	20	10

Figura 1. Máquina NUMA

NUMA formada por 4 nodos (nesse caso, quatro máquinas SMP) onde qualquer um dos oito processadores tem acesso ao banco de memória de qualquer um dos nodos.

A arquitetura das máquinas multiprocessadas propicia uma série de características que proporcionam melhor desempenho. Em máquinas multiprocessadas, por exemplo, podemos ter diversos processos executando em paralelo, cada um em um processador. Estes processos são colocados na memória e cabe ao sistema operacional alocá-los de forma a garantir maior desempenho possível ao sistema. A parte do sistema operacional responsável pelo tratamento da memória é o gerente de memória. Em máquinas SMP, as funções desse gerente são facilitadas devido à máquina ter apenas um banco de memória. Para máquinas NUMA, esta gerência não mais se dá de forma tão simples, já que o sistema operacional tem de lidar com os diversos bancos de memória, em que o tempo de acesso a eles já não é o mesmo para todos os processadores (ou conjunto de processadores). Dessa forma, diversas questões como balanceamento de carga, por exemplo, tem de ter um tratamento diferenciado para que a máquina seja melhor explorada e tenha melhor desempenho.

Atualmente, existem diversos paradigmas em uso para o balanceamento de carga em máquinas multiprocessadas [Chandra et al. 1994]. A maioria deles baseia-se em ambientes de programação paralela que decidem onde cada processo irá rodar no momento em que o processo é criado. Nestes ambientes, uma vez que um processo inicia em um nodo, ele deve se manter no mesmo nodo até o seu término. Mesmo se a carga do sistema, desta forma, não for a melhor, não se poderá voltar atrás sobre decisões de escalonamento já tomadas. Geralmente, é muito difícil prever como a carga do sistema irá variar no tempo, e, sendo assim, aumenta a dificuldade de otimização do escalonamento de processos em processadores [Chandra et al. 1994]. Em função disso, uma série de esforços foram feitos a fim de se desenvolver uma classe diferente de ambientes de programação paralela que permitisse mover um processo de um nodo para outro dinamicamente, durante qualquer momento da vida do processo. Essa mudança de local de execução de um processo durante o seu tempo de vida é chamada “migração de processo” [Milojicic et al. 2000]. Através de mudança dinâmica de processos durante o seu tempo de vida, o sistema pode se adaptar às mudanças de carga que não podiam ser previstas no início da execução das tarefas. Essa migração pode ser interessante do ponto de vista

de balanceamento de carga, mas, novamente considerando arquiteturas em que o tempo de acesso à memória não é igual para todos os processadores (NUMA, por exemplo), a migração pode não beneficiar o desempenho do sistema, já que o custo do acesso de um processo migrado ao seu espaço de endereçamento (que pode estar em um banco de memória mais "distante", devido à migração do processo) pode não pagar o custo (compensar) de um balanceamento de carga. Para o melhor aproveitamento da migração de processos, um sistema de migração de páginas é importante pois fará com que as áreas de memória do processo fiquem mais próximas ao processador em que este está executando, para fazer com que estes dois custos sejam os menores possíveis e resultem em um maior desempenho do sistema global.

Visto a importância de um sistema de migração de páginas, este trabalho apresenta duas estratégias para migração de páginas e mostra um estudo, através de um processo de simulação, sobre o desempenho destas estratégias se implementadas no sistema operacional Linux. A seguir, serão apresentadas algumas questões relevantes sobre gerenciamento de memória em máquinas NUMA (Seção 2), e em seguida, o gerenciamento de memória feito pelo sistema operacional Linux (Seção 2.2). Os modelos de migração de páginas propostos no trabalho, então, são apresentados, primeiramente com o modelo de migração completa (*full migration*) e, em seguida, o modelo de migração sob demanda (*on demand*), ambos mostrados na Seção 2.3. Para a avaliação do desempenho do sistema foi utilizado um modelo simulado, que será apresentado ao final na Seção 3.

2. Gerência de Memória em Sistemas Operacionais com suporte à NUMA

A fim de garantir a eficiência em um sistema, a memória deve ser muito bem gerenciada [Haeberlen and Elphinstone 2003]. Em um sistema operacional, o gerente de memória é responsável por gerenciar o uso de toda a memória disponível no sistema. Gerência de memória em máquinas monoprocessadas tem sido assunto comum de pesquisa nos últimos anos. Contudo, gerência de memória ainda apresenta alguns desafios quando aplicada a máquinas multiprocessadas, embora já tenha sido estudada por um tempo considerável.

Com o objetivo de proporcionar o correto gerenciamento da memória, o sistema operacional precisa conhecer a estrutura de memória do computador. Atualmente, grande parte dos computadores implementam o padrão ACPI [ACPI 2004] (*Advanced Configuration and Power Interface*), o qual provê diversas informações para o sistema operacional.

2.1. ACPI

ACPI [ACPI 2004] é uma especificação de inteface que provê informações sobre a configuração do *hardware*, permitindo ao sistema operacional realizar o gerenciamento de energia dos dispositivos de um computador. Todos os dados da ACPI estão hierarquicamente organizados em tabelas de descrição que são montadas pelo *firmware* do computador. A estrutura base do padrão ACPI é o *Root System Description Pointer* (RSDP), o qual é carregado na memória em um endereço padrão e aponta para a *Root System Description Table* (RSDT). A RSDT contém ponteiros para todas as outras tabelas de descrição que provém informações a respeito da configuração da máquina, como por exemplo, informações de dispositivos *plug & play*, temporizadores, informação sobre a memória, etc. Uma destas tabelas, chamada *System Locality Information Table* (SLIT), descreve a

distância relativa (latência de memória) entre localidades ou domínios de proximidade. Especificamente no caso de computadores tipo NUMA, cada nodo é uma localidade. Assim, a distância entre os nodos está disponível na SLIT. Em uma SLIT, cada entrada representa a distância de um nodo a outro. A distância entre um nodo e ele mesmo é chamada distância SMP e possui o valor padrão de 10. Todas outras distâncias são relativas à distância SMP. Como processadores e blocos de memória estão dentro de um nodo, a SLIT provê a distância entre processadores e áreas de memória, isto é, diferentes distâncias na SLIT representam diferentes níveis de acesso à memória. A Figura 1 mostra a SLIT para a máquina NUMA da mesma figura. De acordo com esta tabela, a distância entre o nodo 1 e o nodo 2 é duas vezes a distância SMP. Isto significa que um processador no nodo 1 acessa a área de memória do nodo 2 duas vezes mais lento que uma área de memória no nodo 1 [ACPI 2004].

2.2. O Gerente de Memória do Linux

O gerente de memória implementado pelo Linux (versão 2.6) trabalha de forma similar para máquinas NUMA e SMP. Em máquinas NUMA, contudo, a memória é organizada em bancos chamados nodos, e, o gerente de memória do Linux aloca memória para um processo no banco mais perto do nodo no qual o processo está executando. Esta estratégia se mostra interessante se um processo não muda de nodo durante seu tempo de execução. Entretanto, o escalonador do Linux tenta manter a carga de todos os processadores o mais balanceada possível, reatribuindo processos a processadores em três diferentes situações: (i) quando um processador entra no estado ocioso; (ii) quando um processo executa a chamada de sistema *exec* ou *clone*; e (iii) periodicamente em intervalos de tempo específicos definidos para cada domínio de escalonamento [Corrêa et al. 2006]. Esta migração de processos pode causar o "afastamento" da área de memória do processo em relação ao nodo no qual o processo está executando. Atualmente, o gerente de memória do Linux não implementa qualquer tipo de migração de páginas, isto é, depois da migração de um processo, nenhum esforço é feito para trazer o seu espaço de endereçamento para um banco de memória mais próximo do processador no qual ele foi reescalonado para rodar. O único esforço feito pelo Linux para trazer as páginas mais perto do nodo onde o processo está executando ocorre se as páginas são trazidas à memória em função de uma falta de página (*page fault*).

2.3. Estratégias para Migração de Páginas

Em sistemas SMP, a escolha de migrar processos de um processador sobrecarregado para um processador livre (*idle*) não causa nenhum efeito maior. Como a distância entre todos os processadores e a memória é a mesma, migrar um processo de qualquer processador para outro não diminui o desempenho global do processo. Isto não acontece em máquinas NUMA; migrar um processo de um processador para outro no mesmo nodo é melhor do que migrá-lo de um processador para outro nodo. Isto se deve às diferentes distâncias entre processadores que estão em diferentes nodos.

Este trabalho analisa duas abordagens que consideram a migração do espaço de endereçamento de um processo migrado para o banco de memória mais próximo ao processador para o qual o escalonador migrou o processo: (i) migrar todo o conteúdo da memória do processo no mesmo momento em que o processo migra (*full memory migration*); e (ii) migrar do conteúdo da memória do processo conforme o processo continua a executar (*on-demand memory migration*).

Full Memory Migration. Esta estratégia considera a possibilidade de migrar toda a memória do processo no momento em que o processo está sendo migrado. Normalmente o Linux migra um processo quando um processador está livre, portanto, esta estratégia não irá diminuir o poder de processamento global do sistema. O maior *overhead* trazido ao sistema está relacionado ao acesso ao barramento da memória, isto é, mover páginas de um processo migrado de um nodo para outro pode fazer com que processos executando tenham de esperar até esta migração ser concluída.

A Figura 2 demonstra o algoritmo em questão. No passo 1 o escalonador verifica que existe um processador livre ($P3$), enquanto $P1$ encontra-se "sobrecarregado". O escalonador, então migra o processo $S1$ para $P3$ no passo 2 e em seguida, no passo 3, o algoritmo de migração detecta que o espaço de endereçamento de $S1$ não se encontra o mais próximo possível do processador no qual ele está executando e assim, move o espaço de endereçamento para o banco de memória mais próximo de $P3$ (passo 4), neste exemplo.

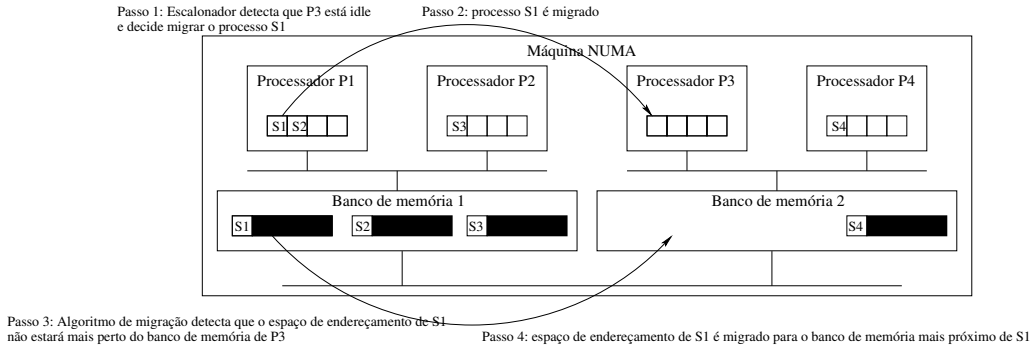


Figura 2. Migração do espaço de endereçamento de um processo

On-Demand Memory Migration. Embora migrar todo o espaço de endereçamento do processo ao mesmo tempo que o processo está migrando pode aumentar o desempenho do sistema, mover todo o espaço de endereçamento do processo toda vez que um processo é migrado pode causar a diminuição do desempenho se este migra diversas vezes durante seu tempo de execução ou ainda se a migração ocorre quando o mesmo está próximo de terminar. A segunda abordagem analisada neste trabalho considera a mesma estratégia usada pelo sistema de memória virtual do Linux quando este faz *swap* de páginas do disco. Nossa estratégia marca todas as páginas do processo como não presentes na memória (*swapped out*) e quando uma falta de página ocorre é verificado se esta falta foi gerada por um processo migrado ou por um processo que tinha, de fato, páginas armazenadas no disco. Se a falta de página for relacionada a um processo migrado, nosso sistema move as páginas do nodo remoto para o nodo atual do processo.

3. Resultados Numéricos

Para comparar o desempenho do algoritmo do gerente de memória do Linux quando ocorre migração de processos em máquinas NUMA com o mesmo algoritmo usando os modelos propostos, foi implementado um modelo simulado utilizando a ferramenta de simulação *JavaSim* [Javasim 2005]. Em [Tsafrir and Dror 2006] pode-se verificar que generalizar a caracterização do *workload* do sistema através da coleta dos *logs* de carga

da máquina, e criar modelos estatísticos baseados nos atributos desse *workload* pode ser questionado se os dados não são estáveis (bastante variáveis), que é o caso do contexto de supercomputadores paralelos. Simulação é um método de análise de desempenho bastante flexível, no que diz respeito a possibilidade de configuração de diversos ambientes reais de maneira simplificada. Assim, este tipo de estratégia faz com que os resultados não sejam sobre uma ou poucas configurações específicas. Para obter os resultados apresentados neste trabalho, as simulações foram feitas usando-se uma máquina baseada nas arquiteturas Altix da SGI [M.Woodacre et al. 2005]. A máquina simulada possui quatro nodos, oito processadores e quatro níveis de acesso à memória. A Figura 1 ilustra a máquina descrita.

Considerando os resultados das simulações, estes foram baseados no tempo total de execução de todos os processos submetidos ao sistema. A configuração do ambiente foi variada da seguinte forma[1]: tempo médio de execução dos processos, número de processos no sistema, tamanho do processo, taxa de acesso à memória do processo e taxa de repetição de acesso a uma mesma área de memória já acessada pelo processo (localidade de referências). Para cada caso de teste foram feitas 200 simulações. As variáveis que configuram o ambiente foram definidas a fim de cobrir a maioria dos casos possíveis.

Dentre as variáveis anteriormente citadas, o tempo médio de execução dos processos foi utilizado com o objetivo de se analisar o comportamento do sistema quando submetido a uma carga de processos que ficam um curto espaço de tempo executando em comparação a uma carga de processos que possuem maior tempo de processamento. Isto, devido ao fato de que em um ambiente com grande quantidade de processos sendo submetidos, quanto maior o tempo que os processos tem de executar, maior a chance destes sofrerem o processo de migração. Desta forma, configura-se um cenário interessante para o estudo, já que pode-se ter, assim, a área de memória do processo migrado em um banco de memória "remoto", ou seja, em um banco de memória que não é o mais próximo ao processador para qual este processo migrou.

A Figura 3 mostra o resultado das simulações com a variação do tempo médio de execução de cada processo nos seguintes valores: 500, 1000, 2000 até 15000ms com variação de 1000ms para cada simulação. Estes valores foram simulados em um ambiente com 50 processos, todos de 50KB, que ficam 60% do seu tempo fazendo acessos à memória sendo 40% desses acessos a posições de memória já alguma vez acessadas.

Como podemos observar, o algoritmo *full migration* teve um desempenho menor comparado aos demais. Os algoritmos do Linux e o *on demand migration* apresentam um desempenho muito similar, sendo o ganho do segundo algoritmo ficando entre 9 e 13%. Este ganho inicia em 10,8% e cresce atingindo o pico de 12,7% com processos de tempo médio de execução de 3000ms. A partir de então, este ganho decresce até 3% com processos de 100000ms. Pode-se explicar este comportamento pelo fato de que quanto mais tempo os processos ficam executando, mais páginas da memória estes podem acessar, causando, portanto, uma maior quantidade de migrações de páginas no caso do algoritmo *on demand*. Assim, o ganho deste algoritmo acaba se "suavizando", pois na medida em que esse número de migrações é maior, e, sendo as páginas migradas acessadas poucas vezes, o custo do acesso a uma página fica cada vez mais similar ao do algoritmo

[1]Algumas destas informações são baseadas em dados do Linux ou de máquinas reais

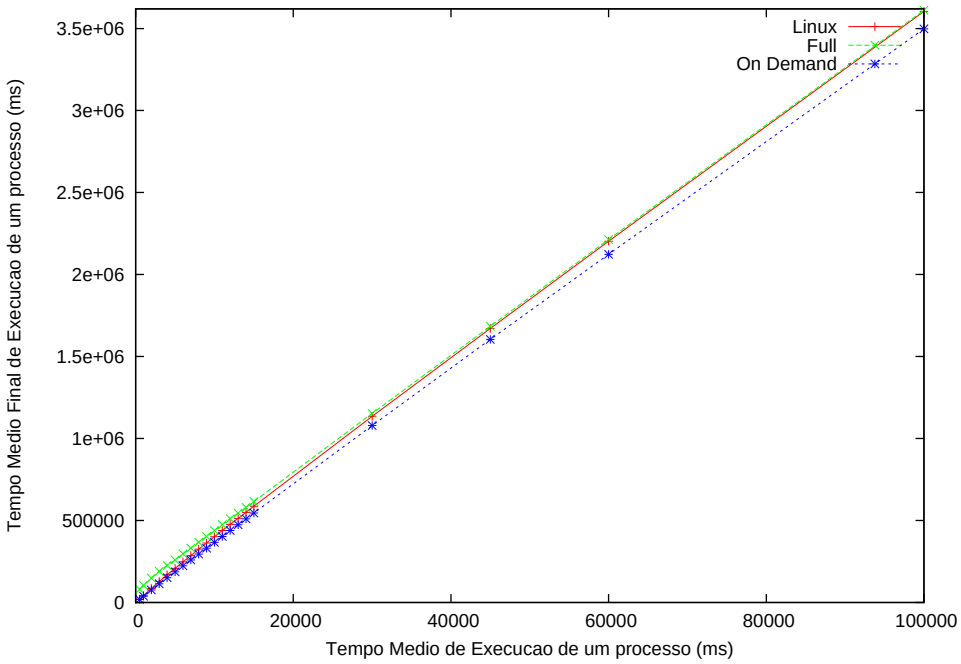


Figura 3. 50 processos de 50KB, taxa de acesso a memória de 60% e taxa de acesso a posições de memória já acessadas de 40%

do Linux. O custo do primeiro acesso a uma página depois da migração do processo é o mesmo tanto para o algoritmo do Linux quanto para o *on demand*. O ganho do algoritmo *on demand* se dá através dos acessos repetidos à páginas já migradas, assim, quanto menos acessos repetidos temos a uma página, menor esse ganho, e quanto mais páginas são acessadas, mais parecido fica o desempenho dos dois algoritmos pois o custo do primeiro acesso após a migração, como já dito, é o mesmo. Com o mesmo raciocínio, podemos verificar o desempenho do algoritmo *full migration*. Quanto mais páginas são acessadas pelo processo, maior o desempenho deste algoritmo quando comparado aos outros dois. E como já dito, quanto mais tempo o processo fica executando maior a chance disto acontecer, refletindo na diferença do ganho do algoritmo do Linux quando comparado a este. O ganho começa com quase 200%, e decresce (146%, 74%, 48%, 33%, 24%, ...) até se aproximar de 0% com processos de tempo médio de execução de 100000ms.

A fim de verificar outro fator variável no sistema, um caso interessante de se analisar é a influência do tamanho dos processos no desempenho dos algoritmos de migração. Isto porque com processos de tamanhos maiores, temos mais espaço da memória sendo alocado para os processos, e portanto, em caso de migração, temos maior probabilidade de se ter o espaço de endereçamento do processo distribuído entre os bancos de memória da máquina, diferenciando o custo do acesso à mesma dependendo de onde o processo precisa buscar dados.

A Figura 4 apresenta o resultado das simulações com a variação do tamanho do processo de 50, 100, 200, 300, 400 e 500KB. Estes valores foram simulados em um ambiente com 50 processos, todos com tempo médio de execução de 10000ms, que ficam 60% do seu tempo fazendo acessos à memória sendo 40% desses acessos a posições de

memória já alguma vez acessadas. O algoritmo de migração *full migration* apresentou um resultado já esperado. Como neste algoritmo, toda vez que ocorre migração do processo, toda sua área de memória é migrada, quanto maior o tamanho do processo, mais tempo este leva para migrar de um banco de memória para outro, sendo assim, maior proporcionalmente será o seu tempo de execução. A curva de desempenho dos algoritmos do Linux e o *on demand* ficaram muito parecidas, com um ganho novamente do algoritmo *on demand*. O ganho inicial de 9,3% com processos de 50KB aumentou gradativamente até 13,6% com processos de 300KB, onde se estabilizou. A estabilização deste ganho pode ser entendida, neste cenário, pensando-se que o fato de se aumentar o tamanho do processo não causa diferença no desempenho se o processo não tem tempo de acessar todos os seus dados. Assim, pode-se pensar que, aumentando-se o tamanho do processo e da mesma forma, aumentando-se o seu tempo de execução, pode-se ter um ganho diferente do que o visto neste cenário.

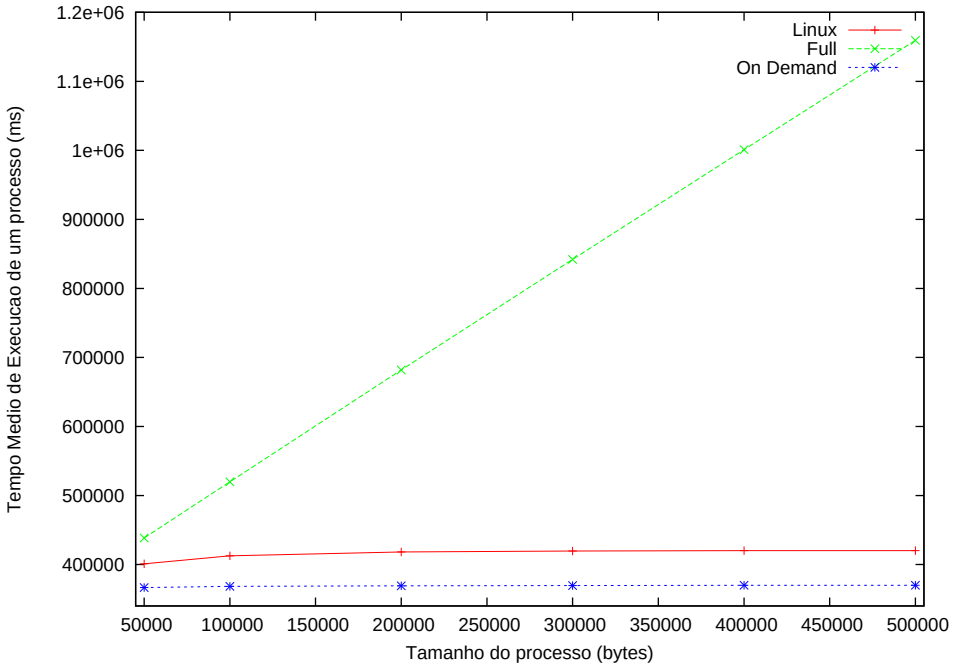


Figura 4. 50 processos de tempo médio de execução de 10000ms, taxa de acesso à memória de 60% e taxa de acesso a posições de memória já acessadas de 40%

Visto que a taxa de acessos repetidos a posições de memória já acessadas é um fator que diferencia o algoritmo *on demand*, algumas simulações foram executadas, a fim de verificar a influência desta taxa no desempenho final do algoritmo. A taxa de acesso a posições de memória já acessadas foi variada nos seguintes valores: 20, 40, 60 e 80%. Estes valores foram simulados em diferentes ambientes, onde variou-se a taxa de acesso à memória dos processos simulados.

As figuras a seguir apresentam o resultado das simulações com a taxa de acesso à memória variando nos seguintes valores: 20% (Tabela 3 - a), 40% (Tabela 3 - b), 60% (Tabela 3 - c) e 80% (Tabela 3 - d).

Pode-se, desta forma, verificar, que o ganho do algoritmo *on demand* aumenta conforme aumenta-se a taxa de acessos a posições de memória já acessadas, quando

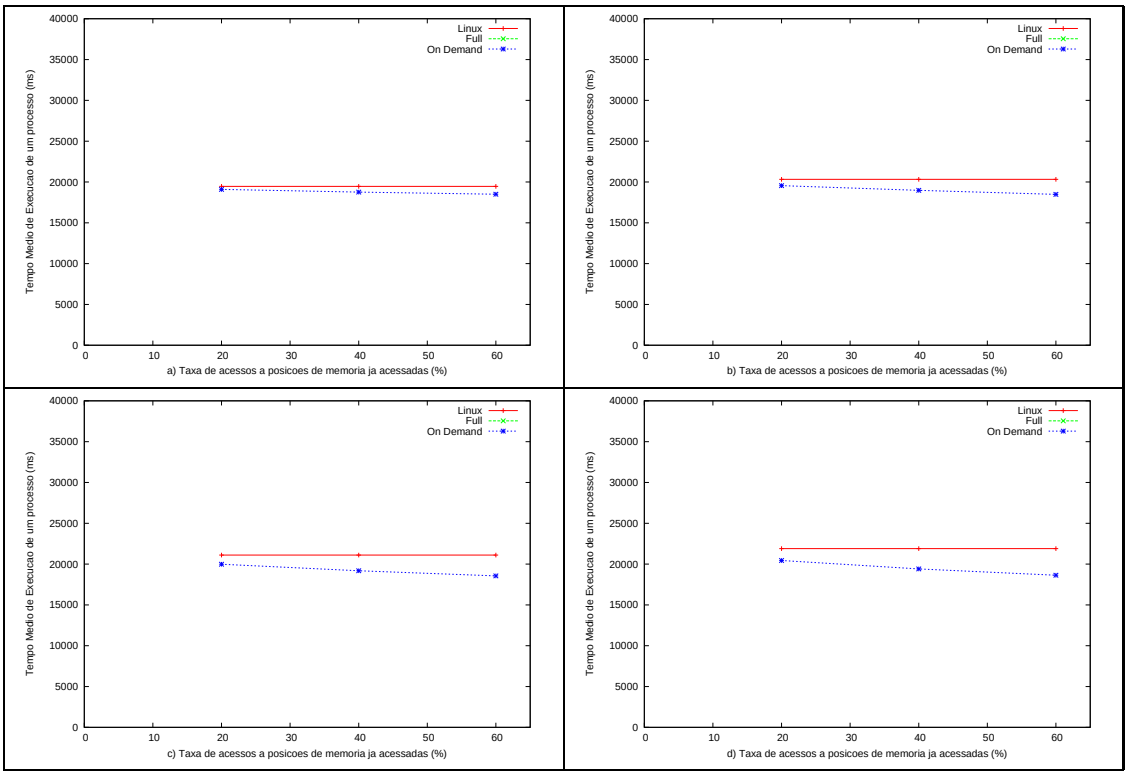


Tabela 1. Ambiente com 200 processos de 50KB, tempo médio de execução de 500ms e taxa de acesso à memória de 20%, 40%, 60% e 80%.

comparando-o com o algoritmo do Linux. A Tabela 2 apresenta o ganho do algoritmo proposto (*on demand*) quando comparado ao implementado pelo Linux.

Tx Acesso Memória / Tx Acesso Rep. Memória	20%	40%	60%
20%	1,94%	3,69%	5,18%
40%	3,94%	7,1%	9,94%
60%	5,59%	10,04%	13,72%
80%	7,12%	12,83%	17,5%

Tabela 2. Ganho do algoritmo *on demand* comparado ao do Linux em um cenário com 200 processos de 50KB e de tempo médio de execução de 500ms.

Além disso, podemos verificar que, conforme aumenta-se a taxa de acesso à memória, mesmo mantendo-se constante a taxa de acessos a posições de memória já acessadas, o ganho do algoritmo *on demand* sob o do Linux também aumenta. A Tabela 2 mostra esse ganho.

Interessante notar que ao aumentarmos em 20% tanto à taxa de acessos a posições de memória já acessadas, quanto à taxa de acesso à memória, o aumento no ganho final do algoritmo *on demand* é de no mínimo 27%, chegando a atingir quase 100% em alguns casos.

Aumentando-se a carga do sistema, submetendo-se 400 processos para serem executados, obtem-se os resultados, em que novamente verifica-se que, aumentando-se em 20% tanto à taxa de acesso à memória, quanto à taxa de acessos a posições de memória

já acessadas, o ganho do algoritmo *on demand* é de no mínimo 30%.

Pôde-se ainda verificar, com estes resultados que, quanto maior a taxa de acesso à memória e quanto maior a taxa de acesso a posições de memória já acessadas, maior o ganho do algoritmo *on demand* comparado ao algoritmo do Linux. O ganho quando aumenta-se a taxa de acesso a posições de memória já acessadas pode ser explicado pelo fato de que, quanto mais o processo acessa páginas que já acessou, maior a chance dessas páginas já estarem no banco de memória mais próximo do processador onde o processo está executando no momento. Assim sendo, o custo do acesso a essas páginas é menor para o algoritmo *on demand* em relação ao implementado pelo Linux. A explicação do ganho ao se aumentar a taxa de acesso à memória vem do fato de que quando um processo migra para um processador que o deixa "longe" do banco de memória onde foi carregado, quanto mais acesso à memória este processo faz, mais se aplica o ganho causado pelo acesso a posições de memória já acessadas, descrito anteriormente.

Pode-se notar também que este ganho diminui conforme a carga do sistema, como pôde-se ver comparando a Tabela 2 com os resultados da mesma configuração de simulação com 400 processos de carga no sistema. Além disso, podemos verificar que ao aumentarmos a taxa de acessos a posições de memória já acessadas, o ganho do algoritmo *on demand* é proporcionalmente maior do que o valor desse aumento. Isso mostra o que já havia sido dito anteriormente (Seção 2.3), de que o diferencial deste algoritmo com relação ao do Linux se dá na taxa de acessos a posições de memória já acessadas.

Como visto nas simulações anteriores, o número de processos no sistema é outro fator que influencia o desempenho dos algoritmos estudados. A fim de verificar esta influência, foram feitas simulações submetendo o sistema a diferentes cargas. Para isso, variou-se o número de processos de de 10 à 750. A Figura 5 mostra o desempenho dos algoritmos no cenário descrito.

Pode-se notar uma discrepância bastante grande do desempenho do algoritmo *full migration* comparado aos outros dois, quando temos um cenário com pequeno número de processos. No cenário mostrado na Figura 5, em um sistema com 10 processos tem-se um desempenho quase que 800% pior do que os outros dois algoritmos avaliados. Isto pode ser explicado pelo alto impacto que uma *full migration* causa em um sistema com poucos processos, pois o custo de migrar toda área de memória de um processo de um banco para outro, é bastante superior ao custo de execução do processo. Como exemplo, pode-se imaginar o cenário simulado, em que temos 10 processos de tempo médio de execução de 500ms. Supondo-se, então que 9 desses processos executem no cenário em exatos 500ms e que um desses processos seja migrado (*full migration*), temos que o tempo de execução do processo migrado aumentará consideravelmente. Como temos poucos processos no sistema, é grande a chance de que após se passarem 500ms, tenhamos somente o processo migrado executando. Com o aumento do número de processos, o tempo da migração acaba se diluindo no paralelismo do processamento, pois temos mais processos a executar. Naturalmente que o "fenômeno" mencionado pode continuar a acontecer, mas o fato de existir mais processos no sistema, faz com que o impacto causado no desempenho final se "suavize".

O ganho do algoritmo *on demand* sobre o do Linux é maior em um sistema com poucos processos (10) que com um número maior (60). Desta vez, pode-se explicar este

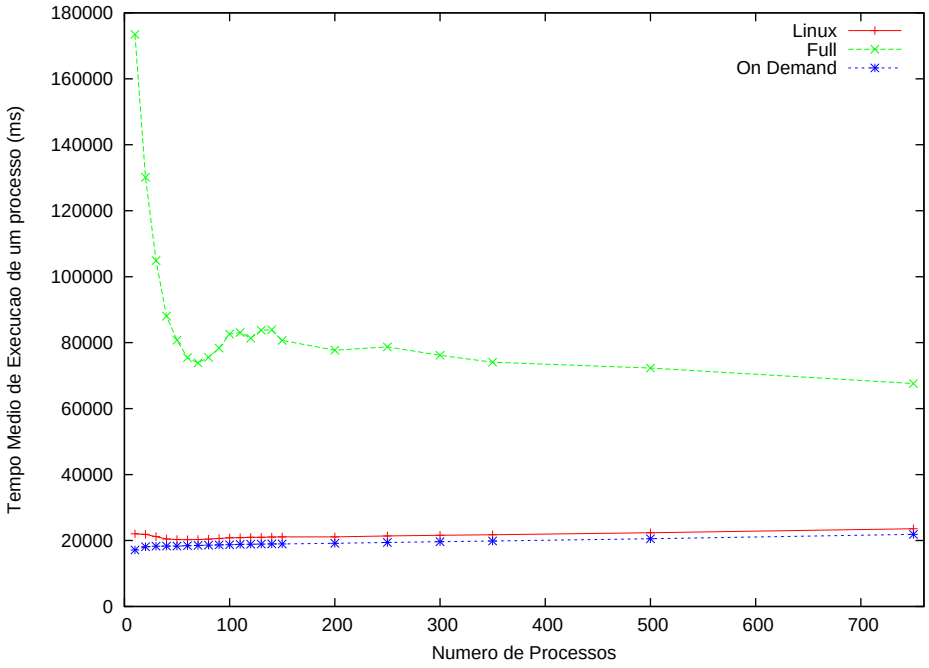


Figura 5. Processos de 50KB, tempo médio de execução de 500ms, taxa de acesso à memória de 60% e taxa de acesso a posições de memória já acessadas de 40%

comportamento devido o grande impacto que um acesso "remoto" à memória traz quando temos poucos processos no sistema. A idéia é a mesma do "fenômeno" descrito anteriormente: em um sistema com poucos processos, quando um é migrado, o seu custo de acesso à memória aumenta, aumentando, também, o seu tempo de execução. Assim sendo, teremos o cenário descrito no parágrafo anterior novamente, em que transcorrido um tempo, teremos apenas o processo migrado executando. Com um maior número de processos, esse custo é "suavizado" pelo paralelismo da execução dos demais processos.

4. Conclusão

Dada a escalabilidade de máquinas multiprocessadas do tipo NUMA, tem-se máquinas com arquiteturas cada vez mais diferentes, o que torna o trabalho do sistema operacional bastante complexo quando se objetiva o melhor desempenho da máquina. A gerência de memória é um dos fatores que influenciam bastante no desempenho do sistema neste tipo de máquinas. Neste trabalho foram apresentadas duas estratégias para a migração de páginas da memória. A forma de estudo das mesmas foi através de simulação, visto que é uma abordagem flexível e permite que sejam analisados diferentes cenários, evitando o estudo de apenas um conjunto de casos que aparentemente cobrem grande parte dos casos de uso desse tipo de máquina, mas que é uma forma questionável de se fazê-lo [Tsafrir and Dror 2006]. Neste trabalho foram mostrados alguns resultados obtidos com as simulações. Diversos outros cenários foram simulados e apresentaram resultados semelhantes. Em [Corrêa et al. 2006] foi estudada uma proposta para a migração de processos no Linux que mostrou através de simulação e posterior implementação, o desempenho do sistema aumenta quando processos são migrados entre processadores. Em [Chanin et al. 2005] utilizando modelos estocásticos, verificou-se da mesma forma este

ganho. Este trabalho mostrou que, levando-se em conta a migração de páginas juntamente com a migração dos processos pode-se obter um desempenho ainda maior. O algoritmo *on demand* mostrou ter um desempenho superior ao algoritmo implementado pelo Linux em todos os cenários simulados. Por ser um trabalho feito em parceria com a Hewlett-Packard, o desenvolvimento do algoritmo de migração *on demand* apresentado está sendo feito por um HP *Lab*. Diversos outros aspectos relacionados com as características de processos não foram mostrados neste artigo e são parte de um trabalho futuro, por exemplo, compartilhamento de páginas entre processos.

Agradecimentos. A. Zorzo possui bolsa de produtividade em pesquisa do CNPq. G. Tesser possui bolsa de mestrado da HP Brasil. Os autores agradecem também aos revisores do artigo.

Referências

ACPI (2004). Advanced configuration and power interface specification. In *http://www.acpi.info/DOWNLOADS/ ACPIspec30.pdf*.

Bircsak, J., Craig, P., Crowell, R., et al. (2000). Extending OpenMP for NUMA machines. *Scientific Programming*, 8(3):163–181.

Chandra, R., Devine, S., Verghese, B., Gupta, A., and Rosenblum, M. (1994). Scheduling and page migration for multiprocessor compute servers. In *ASPLOS*, pages 12–24.

Chanin, R., Corrêa, M., Fernandes, P. H. L., Sales, A., Scheer, R., and Zorzo, A. (2005). Analytical modeling for operating system schedulers on NUMA systems. In *Electronic Notes In Theoretical Computer Science*, pages 1–20.

Chapin, J., Herrod, S. A., Rosenblum, M., and Gupta, A. (1995). Memory system performance of UNIX on CC-NUMA multiprocessors. In *Joint International Conference on Measurement & Modeling of Computer Systems*, pages 1–13.

Corrêa, M., Zorzo, A., and Scheer, R. (2006). Operating system multilevel load balancing. In *ACM Symposium on Applied Computing*.

Haeberlen, A. and Elphinstone, K. (2003). User-level management of kernel memory. In *Asia-Pacific Computer Systems Architecture Conference*, volume 2823 of *Lecture Notes in Computer Science*, pages 277–289.

Javasim (2005). Javasim web site. In *http://javasim.ncl.ac.uk*.

Milojicic, D., Dougles, F., Paindaveine, Y., Wheeler, R., and Zhou, S. (2000). Process migration. In *ACM Computing Surveys*, pages 241–299.

M.Woodacre, Robb, D., and Feind, K. (2005). The SGI AltixTM 3000 global shared-memory architecture.

Paterson, D. and Hennessy, J. (1998). *Computer Organization and Design: the hardware/software interface*. Morgan Kaufmann.

Tao Mu, Jie Tao, Martin Schulz, S. A. M. (2003). Interactive locality optimization on NUMA architectures. In *ACM Symposium on Software visualization*, pages 133–213.

Tsafrir, D. and Dror, G. F. (2006). Instability in parallel job scheduling simulation: The role of workload flurries. In *IEEE International Parallel & Distributed Processing Symposium*.