Análise do custo de captura de pacotes no ambiente virtualizado Xen

George Teodoro e Tiago Macambira

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais, Brasil
{george,tmacam}@dcc.ufmg.br

1 Introdução

A virtualização apareceu como uma ótima solução para compartilhamento de sistemas computacionais, criando a possibilidade de promover uma melhor taxa de utilização dos recursos de hardware e conseqüentemente um melhor retorno dos investimentos. Além dos ganhos em utilização, a forma como o software é abstraído do hardware oferece várias vantagens, dentre elas podemos destacar a maior disponibilidade e confiabilidade para as aplicações, pois cada um dos ambientes operam independentemente e a falha um deles não afeta os outros.

Dada a grande quantidade de oportunidade de aplicação dessa solução a virtualização vem se popularizando rapidamente. Entretanto, é fácil notarmos que esse tipo de solução apesar de muito interessante acaba introduzindo um custo extra referente ao trabalho de virtualização do sistema. Dessa forma, é importante entender cada um dos fatores que podem resultar em gastos adicionais e tentar minimiza-los. Como esses sistemas virtualizados tem sido especialmente utilizados para por aplicações que utilizam redes intensamente é importante avaliar o custo desse serviço, o que permite a análise de isso pode impactar no desempenho das aplicações.

Existem várias soluções de virtualização que obtiveram grande sucesso, entretanto as três soluções que obtiveram maior destaque são VMware, Qemu e Xen. Existem ligeiras diferenças e vantagens em cada um desses softwares, entretanto nesse trabalho avaliamos somente a ferramenta Xen (3), pois a mesma apresenta algumas vantagens importantes, tais como: é de fonte aberta; é relativamente leve, ou seja, não consome CPU excessivamente; promove um grande nível de isolamento entre as máquinas virtuais; e suporta versões e sistema operativos mistos permitindo instaciamento dinâmico do sistema operativo;

É importante observar que sistemas de virtualização podem usar vários mecanismos para virtualizar interfaces de rede para as suas *virtual machines* (VMs). Entretanto quando analisamos o sistema operacional normal e o sistema virtualizado nota-se que existe uma parte comum no caminho do pacote desde a placa de rede até a aplicação. Isso é interessante, pois a identificação de problemas nessa área permitem que ambos os sistemas possam ser melhorados.

Quando analisamos os sistema operacionais modernos tais como o Linux, FreeBSD e Windows XP observa-se que os mesmos não apresentam um desempenho aceitável para a coleta de pacotes em redes de alta velocidade. Usando PCs *low-end*, pode-se observar taxas consideráveis de queda de pacotes até mesmo em redes mais lentas. Técnicas como *device pooling*, implementadas em versões mais recentes tanto do Linux como do FreeBSD, melhoram esse cenário consideravelmente, mas não eliminam o problema completamente (10; 16).

Existem entretanto algumas iniciativas para melhorar o desempenho da captura de pacotes nesses sistemas que vão desde extensões que otimizam algum aspecto do processo de captura de pacotes, tais

como a arquitetura de captura em si (13; 12; 18) até abordagens que visam paralelizar o processo de captura para aumentar o seu desempenho (19).

Nosso intuito então é avaliar os custos de captura de pacotes em um ambiente virtualizado medindo o *overhead* introduzido por essa solução. A identificação de áreas críticas nesse processo visam identificar os pontos de contenção, o que cria oportunidade de melhoria nesses sistemas.

Assim, é claro que a identificação de áreas críticas nesse processo poderia indicar possíveis pontos de contenção que, uma vez resolvidos, melhorariam o desempenho tanto do processo de captura de pacotes como também de toda a pilha TCP/IP tanto dos sistemas operacionais convencionais como de sistemas virtualizados.

O restante desse artigo se encontra organizado da seguinte forma. Na seção 2 descrevemos alguns dos principais trabalho na área. A seção 3 descreve as ferramentas utilizadas para instrumentar o *kernel* do Linux, vantagens, desvantagens e compromissos encontrados na utilização de cada uma delas. As seções 5 e 6 discutem a metodologia empregada para realização dos experimentos e seus resultados. As considerações finais desse artigo são expostas na seção 7.

2 Trabalhos Relacionados

Trabalhos de caracterização de desempenho de pilhas TCP/IP de sistemas operacionais de propósito comuns não são novos. Como trabalho seminal dessa área podemos citar o trabalho de Clack e Jacobson de 1989 (7). Após esse trabalho, foram feitos muitos estudos de aspectos particulares da pilha e propostas de mecanismos que buscassem otimizar o desempenho de vários aspectos e partes das pilhas de rede desses sistemas. Como exemplo, podemos citar o trabalho de Kay, onde comenta-se sobre a importância dos mecanismos de gerência de *buffers* em um sistema operacional, quantificando como tal pode influenciar no desempenho de aplicações de rede em um sistema UNIX e propondo uma abordagem inovadora de gerenciamento de *buffers* (11).

Também não são novidade o uso de mecanismos de virtualizacao, pois os mesmos já são utilizados desde o início da década de 80 pelos sistemas IBM VM/370 para prover compatibilidade com binários antigos (17). Entretanto existe várias técnicas de virtualização: virtualização por imagem única de SO (Virtuozo, VServers etc), virtualização completa (VMWare, QEmu etc) e para-virtualização (Xen, UML etc). Soluções de para-virtualização têm se mostrado bastante populares, sendo que o Xen é a mais comumente utilizada e estudada (3).

A disponibilidade de interfaces para mecanismos de captura de pacotes em vários sistemas operacionais de propósito geral bem como a flexibilidade desses mecanismos de captura promoveram a sua disseminação e aplicação para os mais diversos fins. Como exemplo destes, podemos citar a identificação, monitoração e análise de tráfego, serviços de detecção de intrusão em rede (NIDS), verificação de políticas de utilização de rede, entre outros (13).

Tradicionalmente, sistemas operacionais modernos tais como o Linux, FreeBSD e Windows XP não apresentam um desempenho aceitável para a coleta de pacotes em redes de alta velocidade. Usando PCs *low-end*, pode-se observar taxas consideráveis de queda de pacotes até mesmo em redes mais lentas. Técnicas como *device pooling*, implementadas em versões mais recentes tanto do Linux como do FreeBSD, melhoram esse cenário consideravelmente, mas não eliminam o problema completamente (10; 16).

Outra forma de melhorar o desempenho da captura nesses sistemas é através de extensões que otimizam algum aspecto do processo de captura de pacotes, tais como a arquitetura de captura em si (13; 12; 18), o processo de filtragem (20; 1; 9; 4), ou diminuindo o custo de recuperar o pacote da placa de rede para a área de memória do *kernel* e o custo da cópia de pacotes dessa área para a área de memória das aplicações (16; 8), em trazer parte do processamento feito pelas aplicações para dentro do *kernel* (6; 10) e até em paralelizar o processo de captura para aumentar o seu desempenho (19).

Essas modificações, no entanto, não fazem parte do *kernel* do Linux, que é utilizado pelo Xen como sistema base para seu gerente de VMs. Além

disso, muitas destas modificações não têm nenhuma previsão de serem incorporadas a ele. Finalmente, a despeito da amplitude dos trabalhos de caracterização de desempenho no Xen, nenhum destes aborda o custo que essa solução de virtualização acrescenta na captura de pacotes, que é o foco desse trabalho.

3 Ferramentas para Instrumentação do Kernel

No processo de medida do custo de captura de pacotes pelo Xen, observando o custo dos vários componentes envolvidos nessa atividade, é necessário, antes de mais nada, usar ferramentas que nos permitam obter métricas e informações sobre o funcionamento desse sistema com a menor intrusão possível.

Existem algumas técnica bastante usadas para obter informações do *kernel*, tais como `printk()`, `top`, `iostat`, `vmstat` e `oprofile`. Todas essas ferramentas são muito valiosas na obtenção de informações sobre problemas de desempenho específicos, entrentantao elas também possuem limitações graves (14). Alguns dos maiores problemas estão relacionados ao carater estático das mesmas, que não permitem por exemplo que o usuário adicione novas medidas em tempo de execução.

Afim de resolver esses problemas, foram proposta várias ferramenta, entre elas destacamos o Kprobes que permite uma instrumentação dinâmica e ativa do *kernel* do Linux. Entretanto, essa ferramenta também apresenta algumas deficiências graves relativas à dificulade de utilização, porém ela foi muito importante, pois permitiu que ferramentas como o SystemtTap (14), que sana a maioria dos seus problemas, fossem construidas utilizando os recursos providos pelo Kprobe.

3.1 O SystemTap

A proposta do SystemtTap é a de facilitar o desenvolvimento e uso de pequenos programas de instrumentação para o *kernel* do Linux. Essa ferramenta utiliza da infra-estrutura provida pelo Kprobes para permitir a escrita, em uma linguagem similar a AWK, de scripts para monitorar e instrumentar um *kernel* em execução. Esses scripts, por sua vez, serão convertidos pelo SystemtTap em módulos Kprobes e inseridos no *kernel*.

Devido à facilidade e da flexibilidade do uso do SystemTap, optamos por utiliza-lo como ferramenta para instrumentação do *kernel*. No entanto, é importante ressaltar que, como toda ferramenta, essa também possui limitações, tais como: (1)as funções do SystemTap são similares à `gettimeofday()`, tendo a maior precisão na ordem de microssegundos. Isso foi resolvido com a utilização da função `get_cycles()`. Essa função retorna o número de ciclos do processador desde a sua inicialização e, por isso, possui uma precisão muito maior e (2) a linguagem de script do SystemTap não tem suporte a ponto flutuante. Assim, algumas medidas tiveram que ser arredondadas. Devido ao uso da função `get_cycles()`, pudemos contornar o impacto de arredondamento para um único ciclo de *clock*.

3.2 Impacto da ferramenta

Com o intuito de quantificar o impacto da utilização do SystemTap, realizamos alguns experimentos. O objetivo principal era o de medir o impacto do uso dessa ferramenta no desempenho de chamada de sistemas, a fim de que pudéssemos contabilizar o custo adicional que o uso dessa ferramenta impõe no custo total observado.

Para medir o custo da chamada de sistema, fizemos um programa que executa 1 milhão de vezes a chamada de sistema `umask()` e mede o tempo para a exceção das chamadas. Foi utilizada a chamada `umask()` pois ela apenas modifica uma máscara com as permissões que os arquivos criados pelo programa terão, tendo praticamente nenhum código. Dessa forma, o tempo medido pelo programa é o tempo gasto nas trocas de contexto do modo usuário para o modo *kernel* e de volta.

No primeiro experimento, rodamos o programa sem nenhuma escuta no *kernel*. O programa foi executado 100 vezes e calculamos os intervalos de confiança entre as execuções. O intervalo com 95% de confiança do custo da chamada de sistema foi de

0,47102 $\pm$ 0,00009 μs.

No segundo e últimos experimento, medimos o custo do ponto de vista do programa em modo usuário de uma chamada de sistema com uma escuta que realiza medições semelhantes às que iremos realizar no processamento de pacotes no *kernel*. Essa escuta mede o tempo gasto pelo processador para executar a chamada `umask()`. O tempo é medido em ciclos de máquina, obtido do registrador TSC (*TimeStamp Counter*) através da função `get_cycles()`, disponível no SystemTap.

Essa escuta utiliza quatro variáveis: o momento de entrada na chamada se sistema cujo valor será subtraído do TSC na saída da chamada para se obter o tempo gasto nela, o número de medidas feitas, a soma dos valores das medidas e a soma dos quadrados dos mesmos. Essas variáveis serão utilizadas para calcular a média e a variância do tempo gasto na chamada de sistema monitorada e todas são inteiros de 64 bits para evitar erros causados por arredondamentos. O intervalo com 95% de confiança do tempo da chamada de sistema com essa escuta é de 2,37719 $\pm$ 0,00209 μs.

Apesar de a escuta aumentar a variação das medidas, esse aumento é finito e com um grande número de medidas podemos ter resultados confiáveis. Os experimentos confirmam essa hipótese: executando o programa 100 vezes, obtivemos com confiança de 99%, o custo da chamada de sistema com todo a escuta está no intervalo entre 2.37444 e 2.37994 μs.

4 Rede no Xen

Para que possamos retirar métricas interessantes, é imprescindível conhecer o caminho pelo qual um pacote[1] passa, desde o momento em que ele foi recebido pela interface de rede ou NIC (de *Network Interface Card*), até o programa do usuário rodando em um domínio não-privilegiado do Xen (domU). Essa seção busca apresentar esse percurso, detalhando os pontos nos quais instalaremos as "escutas" do SystemTap. Esboçaremos apenas os detalhes desses sistemas que forem pertinentes à nossa análise. Para um visão mais detalhada, sugerimos a consulta e outros trabalhos existentes na literatura (15; 2; 5).

No Xen, apenas o domínio privilegiado (*dom0*) possui acesso direto ao *hardware* da máquina. O acesso a este pelos domínios não-privilegiados (*domU*) precisa da intermediação do *dom0*. No caso de dispositivos de rede e de bloco, esse mecanismo de intermediação é feito através da separação dos drivers desses dispositivos em duas partes: uma residindo no *dom0* e outra residindo no *domU*. No caso específico da rede, esses drivers são chamados de *netback* e *netfront*.

A comunicação desses drivers com mundo exterior ou mesmo com outros domínios, no entanto, não ocorre diretamente. Ao invés disso, algum mecanismo no *dom0* fará a cópia ou o roteamento de pacotes entre os domínios e o mundo exterior e inter-domínios. No caso mais comum, uma *bridge* virtual é utilizada para ligar o driver da placa de rede real existente no *dom0*com as placas de rede dos outros domínios.

4.1 Captura de pacotes

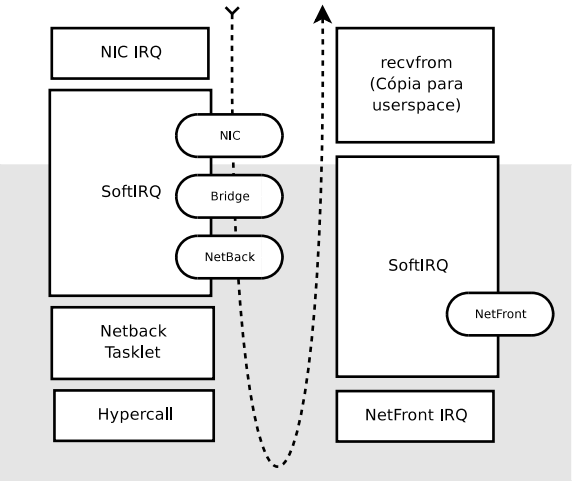


Figura 1. Percurso de um pacote capturado no Xen

[1] No nosso caso, seria mais correto afirmar um quadro, já que, sendo mais preciso, realizamos a captura de quadros Ethernet, e não de pacotes IP propriamente.

A captura de pacotes no *kernel* virtualizado possui vários estágios que são executados de forma assíncrona dentro do *kernel*. Não é possível então analisar o custo dessas partes em conjunto, sendo necessário analisá-las separadamente e contabilizar seus custos isoladamente. Esses estágios, compreendendo tanto aqueles que executam no *dom0* quanto aqueles que executam no *domU* são:

IRQ O primeiro estágio é a interrupção de hardware gerada pela NIC quando um pacote chega. Ela agenda a execução de uma *SoftIRQ* e desativa as interrupções geradas pela NIC quando chegam pacotes. A placa voltará a gerar interrupções apenas quando o a *SoftIRQ* for executada. Esse estágio está presente tanto no dom0, que possui acesso direto ao hardware, quanto no domU. Neste último, essa interrupção está relacionada a uma *hypercall* enviada pelo *dom0*para notificá-lo que existem novos pacotes para serem recebidos.

SoftIRQ *SoftIRQs* são interrupções de baixa prioridade. O tratador de SoftIRQ para redes aciona todos os drivers de dispositivos de redes registrados (virtuais ou não) para que estes realizem os processamentos necessários para tratar os dados existentes nos seus dispositivos. Para a placa de rede, isso significa retirar os pacotes do driver interno da NIC para a memória da placa. Para a *bridge*, receber e tratar o pacotes que chegaram na bridge. Para as placas virtualizadas, isso significa agendar os pacotes para serem enviados aos seus respectivos domínios.

Cópia entre Domínios A cópia de pacotes entre domínios, agendada durante o tratamento da *SoftIRQ* é efetivamente realizada.

HyperCall Feita a cópia, uma *hypercall* do *dom0*para o domU é feita através do Xen. Isso acarreta a recepção, no domU, de uma interrupção de rede, que ocasionará a chamada no *domU*do tratador de IRQs da placa de redes virtualizada e, em seguida, do seu tratador de SoftIRQs e daí por diante.

Cópia para modo usuário Finalmente o programa em modo usuário faz uma chamada de sistema no *domU*para copiar o pacote para o modo usuário. Cada chamada de sistema copia apenas um pacote.

O percurso completo, mostrando todas as partes encontra-se representado na Figura 1.

5 Metodologia

Nesta seção, descrevemos a metodologia de execução e análise de experimentos adotada neste trabalho. As subseções a seguir detalham cada uma das etapas seguidas durante a fase de preparação, execução e análise dos resultados obtidos.

5.1 Instrumentando o *kernel*

Na fase de instrumentação do *kernel* utilizamos a ferramenta de SystemTap, descrita em detalhes na seção 3. Essa ferramenta permiti que coloquemos "escutas"nas chamadas de função do *kernel*, assim podemos fazer medidas de tempo cada vez que uma das funções envolvidas no processo de captura de pacotes é invocada.

Assim, utilizávamos a função `get_cycles()` no início e no fim de cada função que desejamos saber o tempo de execução para que pudéssemos medir o tempo gasto dentro de cada trecho de código de interesse. Essa função foi utilizada por nos prover uma precisão maior que as disponibilizadas pela ferramenta SystemTap.

5.2 Definindo contornos dos experimentos

O experimento em si consistiu em ligar dois micros através de um cabo direto e, num deles, realizar captura de pacotes e no outro o envio. Durante os teste devemos isolar esses computadores de qualquer rede a eles conectada, afim de evitar que qualquer outro tipo de comunicação influencie nos resultados coletados.

O processo de captura utilizou-se da infraestrutura padrão do kernel do linux e do Xen para

tanto, que consiste no uso de *sockets* da família `AF_PACKET` e tipo `SOCK_RAW`. Os pacotes eram capturados pelo uso da chamada `recvfrom()` e com eles não era realizado nenhum tipo de processamento. Dessa forma, o custo observado pela captura consistia tão somente dos custos envolvidos pelo kernel e suas facilidades para captura, não possuindo então nenhum custo relativo ao modo usuário além dos mínimos possíveis. Sendo que no caso do Xen adiciona-se o custo de copiar o pacote para o *kernel* virtualizado.

O processo de envio utilizou-se de uma infraestrutura similar mas destinado ao envio de pacotes "crus". Desta forma, tinha-se flexibilidade para montar os quadros Ethernet como se desejasse. Isso era necessário para evitar que a pilha TCP/IP da máquina usada no envio fosse um ponto de contenção e de acréscimo de variabilidade no experimento. A aplicação usada montava um quadro com o tamanho especificado para o experimento e passava a enviar esse mesmo quadro continuamente. Se fosse desejado, entre o envio de quadros consecutivos seria feita uma pequena pausa. Os quadros criados foram marcados como sendo quadros Ethernet contendo pacotes X.25, o que também impediria que a pilha TCP/IP do lado receptor pudesse acrescentar alguma variabilidade, já que não existiam receptores de pacotes X.25 habilitados na máquina.

5.3 Medidas e análise de resultados

Durante a realização dos experimento utilizamos apenas um processo de captura de pacotes, como descrito na seção anterior. Assim, instrumentamos todo o caminho que os pacotes percorrem desde a placa de rede até seu destino, pois dessa forma podemos identificar onde se encontram as etapas mais custosas desse processo.

Após a identificação do caminho seguido pelo pacote e das variáveis envolvidas envolvidas em cada uma dessas etapas, precisamos também avaliar como tamanho, intervalo entre envio e quantidade de pacotes pode influenciar no desempenho do sistema.

A avaliação desses fatores prosseguiu da seguinte forma:

- **Tamanho do quadro ethernet enviado**: para esse fator estabelecemos apenas dois níveis: quadros com pacotes com 64 bytes e quadros com 1500 bytes.
- **Intervalo entre envios de quadros consecutivos**: para essa variável estabelecemos apenas dois valores: não esperar entre o envio consecutivo de quadros e esperar 1 μ, isso foi feito depois da análise de que o comportamento esperando mais que o últimos valor é semelhante ao mesmo.
- A quantidade de pacotes foi definida com o valor de 10.000, pois valores maiores não acrescenta muito na confiabilidade dos resultados.

6 Resultados Experimentais

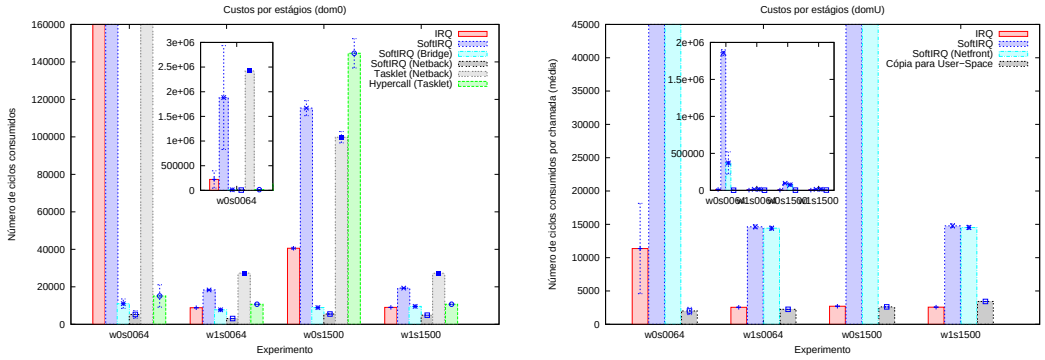
Os resultados das medições realizadas, tanto no *dom0* como no *domU* encontram-se representados na figura 2. Observe que os custos apresentados referem-se ao valor médio do custo de execução das respectivas funções monitoradas e não ao custo amortizado por pacotes.

Comentaremos resultados apresentados estágio-a-estágio. Nas figuras das subseções seguintes, as variações das medidas para um intervalo de confiança de 95% está representado pelas barras azuis.

6.1 Domínio Privilegiado

6.1.1 Tratamento da Interrupção da NIC

Na Figura 3 podemos observar os custos em número de ciclos da CPU que foram utilizados pelo tratador de interrupção (IRQ) do kernel do domínio privilegiado para os 4 cenários observados. Nela, vê-se que existe uma grande variabilidade no tempo para tratara a IRQ no cenário sem espera e com envio de pacotes de 64 bytes (w0S0064). Além disso, ao contrário do que seria intuitivo, o custo nesse cenário foi superior àquele em que os pacotes enviados foram de 1500 bytes. Isso pode ser explicado



(a) (b)

Figura 2. Ciclos médios gasto por chamadas para todos os estágios

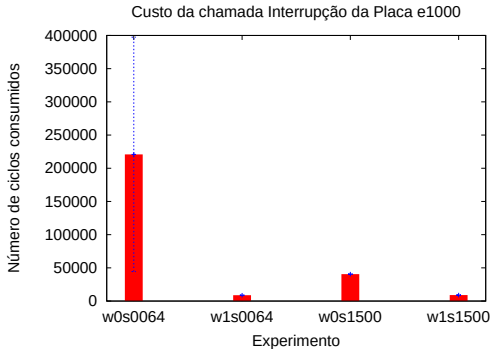


Figura 3. Ciclos usados para tratar a interrupção da NIC

6.1.2 Tratamento da SoftIRQ

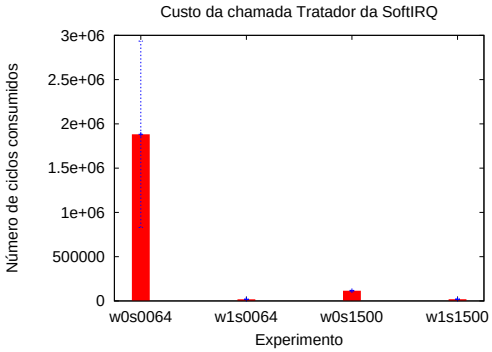


Figura 4. Ciclos usados para tratar a SoftIRQ

devido ao fato de que pacotes menores estressam muito mais a NIC: ao recebimento do primeiro pacote, logo segue-se uma rajada de pequenos pacotes, de tal forma que o desempenho do sistema fica degradado.

Ainda nessa figura, vê-se que nos cenários com espera entre envios sucessivos de pacotes, o desempenho no recebimento de pacotes pequenos e pacotes grandes é praticamente o mesmo, apesar de não serem estatisticamente equiparáveis com 95 % de confiança. Esse comportamento se repete nos estágios seguintes.

Uma característica interessante desse estágio da captura de um pacote é que ele ele é o primeiro a manipular diretamente os dados dos pacotes e, com isso, a possuir custos que variam com o tamanho destes.

Os valores apresentados na Figura 4 compreendem os valores agregados para tratar os drivers da NIC, da bridge e da NetBack. Nas Figuras 5 e 6 apresentamos os valores relativos apenas aos drivers da *bridge* e da *netback*. O custo do tratamento do driver da NIC pela softIRQ não está presente como uma figura isolada mas pode ser derivado das

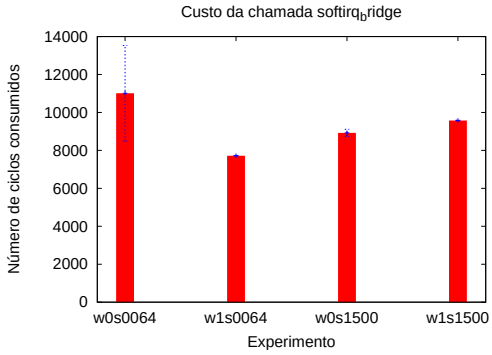


Figura 5. Ciclos usados para tratar a SoftIRQ (driver da *bridge*)

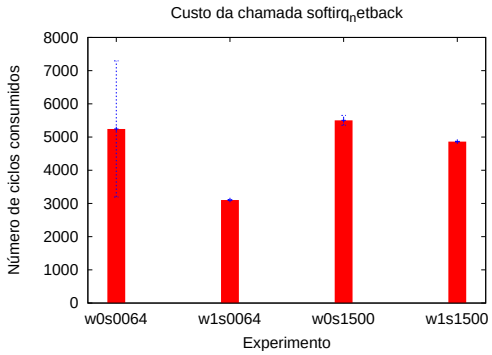


Figura 6. Ciclos usados para tratar a SoftIRQ (driver da *netback*)

3 figuras desta seção. Este fato, aliado com a análise destas figuras permite-nos observar que a maior parcela de custo e o maior responsável pela variabilidade dos tempos observados na Figura 4 é o driver da NIC. Isso pode ser explicado observando que este é primeiro driver a tratar dos dados recebidos da rede. Todos os outros recebem apenas os dados que puderam ser obtidos dele. Além disso, após a execução do driver da NIC, os pacotes já se encontram na memoria do sistema, onde poderão ser operados com mais rapidez.

6.1.3 Tratamento da tasklet da NetBack e da HyperCall

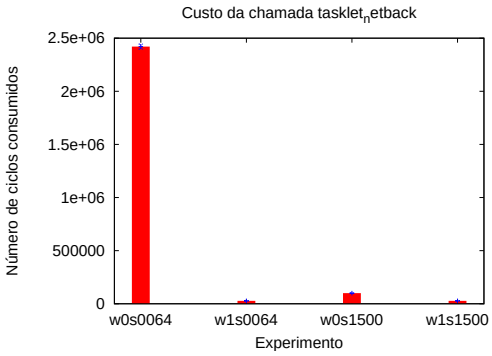


Figura 7. Ciclos usados para tratar a *tasklet* do driver *netback*

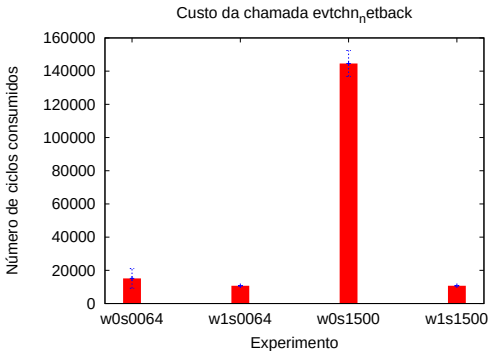


Figura 8. Ciclos usados para realização da *hypercall*

Nesses dois cenários podemos observar como um sistema operando em sobrecarga se comporta, e o efeito desta em diferentes partes do sistema.

Na Figura 7, observamos os custos para efetivamente copiar os pacotes entre domínios. Dissolvidos nesse custo estão os custos para a realização de algumas *hypecalls* de gerência, necessários devido a movimentação de páginas de memória entre os domínios. Como podemos observar, para os cenários sem espera o custo dessa cópia foi

maior do que nos cenários com espera. Além disso, no cenário sem espera e com envio de pacotes de 64 bytes, o desempenho do sistema foi lastimável: a cópia de pacotes conseguiu ter um desempenho pior do que todo o processo de captura físico junto. Observe que os custos representados nessa figura são custos exclusivamente devido a virtualização.

Já na Figura 8 observamos o custo para realizar a notificação do recebimento de pacotes pelo *dom0* ao *domU*. Essa chamada, por sua vez, será recebida o domínio não privilegiado como uma interrupção de *hardware*. Mais uma vez observamos que nos cenários com carga extrema houve uma perceptível degradação desses valores. Ao contrário do que ocorreu no tratamento com a *tasklet*, foi o cenário sem espera e com pacotes de 1500 bytes aquele a apresentar o pior desempenho. Isso pode ser parcialmente explicado devido ao fato de que nesse cenário houve uma maior quantidade de pacotes recebidos comparativamente ao cenário sem espera e com recebimento de pacotes 64 bytes.

Em trabalhos futuros, realizaremos análise mais aprofundada desses sintomas.

6.2 Domínio não-privilegiado

6.2.1 Tratamento da Interrupção da NIC virtual (netfront)

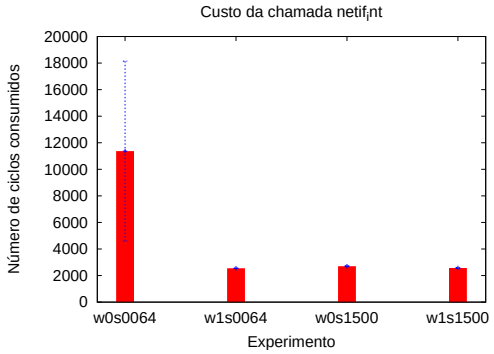


Figura 9. Ciclos usados para tratar a interrupção da NIC

De maneira similar ao que aconteceu no tratador de interrupções de hardware da NIC no domínio privilegiado, no *domU*, o tratador da interrupções da placa virtual também apresentou valores maiores para os cenários sem espera entre envios e, dentre estes cenários, valores maiores e com maior variabilidade para aquele cenário com recebimento de pacotes de 64 bytes.

Deve-se observar que a faixa valores para os tratadores do domínio não privilegiado é bem menor do que as suas similares no domínio privilegiado.

6.2.2 Tratamento da SoftIRQ

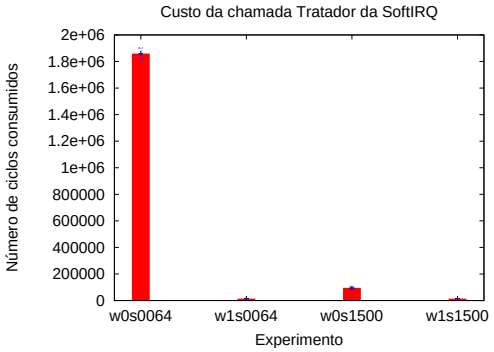


Figura 10. Ciclos usados para tratar a SoftIRQ

Na Figura 10 observamos o custo para o tratamento da SoftIRQ no domínio não privilegiado. Observamos novamente reflexos do comportamento existente no domínio privilegiado no *domU*, mas em escala reduzida. É importante salientar a diferença brutal no número de chamadas realizadas para as SoftIRQs dos dois domínios. Além de efeitos de aglomeração de pacotes, isso deve-se também à perdas de pacotes entre domínios, particularmente nos cenários sem espera entre envios.

6.2.3 Cópia para o modo usuário

A Figura 11 apresenta os valores para realizar a cópia de pacotes do *kernel-mode* para o *user-mode* do domínio não-privilegiado.

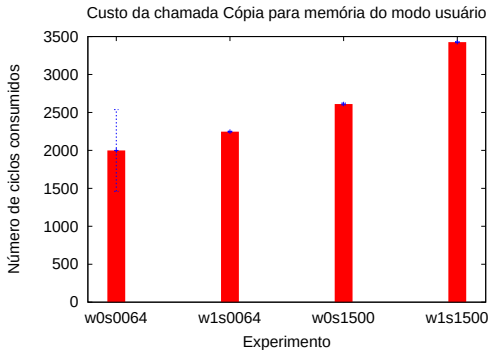


Figura 11. Ciclos usados para tratar a cópia do kernel-mode para o user-mode

A variabilidade e o custo de lidar com tráfego intenso foram amenizados pelos estágios anteriores. A variabilidade do custo nesse estágio deve-se sobretudo, ao processo de copiar os pacotes recebidos para o processo em modo usuário. Por esse motivo é possível afirmar que, descontando-se a variabilidade do experimento $w0s64$, o tamanho dos pacotes é fator que determina o custo desse estágio.

7 Conclusões

É interessante observar como as variações dos fatores intensidade do tráfego e tamanho de pacotes influenciam no comportamento do processo de captura.

Uma surpresa interessante é como tráfegos intensos, apesar de mostrarem os benefícios de *device pooling*, acarretam uma degradação generalizada no processo de captura de pacotes. Essa generalização, como pode ser observado, foi mais acentuada quando o tráfego era composto por pacotes pequenos.

No que diz respeito aos custos da captura de pacotes no ambiente de virtualização Xen, nossos resultados nos surpreenderam. Era esperado em em situações de tráfego intenso o desempenho do *kernel* fosse aquém do esperado. Todavia, o que se observou foi que nessa situações o Xen contribuía com uma boa parcela do custo total. Além disso, as soluções utilizadas para ligar máquinas virtuais entre si (no caso do nosso experimento, a *bridge* virtual) contribuíram de maneira significativa no processo de degradação da capacidade do sistema de capturar pacotes: ao total, no cenário $w0s0064$, dos 10000 pacotes enviados apenas 26 foram capturados com sucesso pelo programa em modo usuário rodando no *domU*.

Como trabalho futuro pretendemos aprofundar a nossa análise, observando as causas das grandes variações encontradas na *tasklet* da SoftIRQ da *netback* e na invocação da *hypercall* entre domínios para notificação de recebimento de pacotes. Além disso, uma comparação mais minuciosa entre o desempenho do sistema virtualizado e o sistema não-virtualizado poderia nos dar maiores informações sobre como optimizar ambos os sistemas.

Referências

[1] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. In *Proc. of the 1st Symposium on Operating System Design and Implementation*, pages 115–123, Nov 1994. USENIX Association.

[2] L. G. C. Barbato and A. Montes. Técnicas de Ocultação de Tráfego de Rede em Honeypots de Alta Interatividade. In *Anais do VI Simpósio sobre Segurança em Informática (SSI'2004)*, São José dos Campos, SP, Novembro 2004. ISBN 85-87978-05-5.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield". Xen and the Art of Virtualization. In *Proc. of 19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[4] A. Begel, S. McCanne, and S. L. Graham. BPF+: exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 123–134, New York, NY, USA, 1999. ACM Press.

[5] R. Bolla and B. Bruschi. RFC 2544 performance evaluation for a linux based open router, 2006.

[6] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly fast packet filters. In *Proceedings of 6th Symposium on Operating Sys-*

tem Design and Implementation (OSDI 2004), Dec 2004.

[7] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27:23–29, jun 1989.

[8] L. Deri. Improving passive packet capture: Beyond device polling. In *4th International System Administration and Network Engineering Conference*, 2004.

[9] D. R. Engler and M. F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 53–59, New York, NY, USA, 1996. ACM Press.

[10] S. Ioannidis, K. Anagnostakis, J. Ioannidis, and A. Keromytis. xPF: packet filtering for lowcost network monitoring. In *IEEE Workshop on High-Performance Switching and Routing (HPSR)*, pages 121–126, May 2002.

[11] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in tcp/ip. *SIGCOMM Computer Communication Review*, 23(4):259–268, 1993.

[12] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for usert-level packet capture. In *Proc. of the Winter 1993 USENIX Conference*, pages 259–270, San Diego, California, 1993.

[13] J. Mogul, R. Rashid, and M. Accetta. The packer filter: an efficient mechanism for user-level network code. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 39–51, New York, NY, USA, 1987. ACM Press.

[14] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Ottawa Linux Symposium*, pages 49–64, 2005.

[15] M. Rio, M. Goutelle, T. Kelly, R. Hughes-Jones, J. Martin-Flatin, and Y. Li. A map of the networking code in linux kernel 2.4.20. Technical report, DataTAG Project, March 2004. Technical Report DataTAG-2004-1, FP5/IST.

[16] L. Rizzo. Device polling support for FreeBSD. In *BSDConEurope Conference*, 2001.

[17] L. Seawright and R. MacKinnon". VM/370 - a study of multiplicity and usefulness. In *IBM Systems Journal*, pages 4–17, Oct 1979.

[18] J. van der Merwe, R. Cáceres, Y. hua Chu, and C. Sreenan. mmdump: a tool for monitoring internet multimedia traffic. *SIGCOMM Comput. Commun. Rev.*, 30(5):48–59, 2000.

[19] G. Varenni, M. Baldi, L. Degioanni, and F. Risso. Optimizing packet capture on symmetric multiprocessing machines. In *SBAC-PAD '03: Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, page 108, Washington, DC, USA, 2003. IEEE Computer Society.

[20] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter*, pages 153–165, 1994.