Reserva de Processamento: uma abordagem no nível do usuário

Valéria Q. Reis, Renato F. G. Cerqueira

[1]Departamento de Informática – PUC-Rio
Rio de Janeiro, Brasil

{vreis,rcerq}@inf.puc-rio.br

Resumo. *Sistemas Operacionais de propósito geral não apresentam mecanismos eficazes para a reserva de processamento de aplicações. Dessa maneira, algumas iniciativas visam oferecer garantia de processamento através da instrumentação de kernels ou através do isolamento de desempenho por meio da criação de máquinas virtuais. De maneira diferente dessas abordagens, este artigo descreve em detalhes o funcionamento do CPUReserve, um sistema de reserva de processamento que é executado no nível do usuário. Por apresentar uma arquitetura cliente-servidor e significativa escalabilidade, como sugerem os experimentos realizados, o CPUReserve pode ser utilizado em ambientes distribuídos e compartilhados.*

1. Introdução

Ambientes de computação compartilhada, por natureza, apresentam desafios para o gerenciamento de seus recursos. Garantir que usuários não sobrecarreguem máquinas ou violem políticas de uso, assim como garantir aos usuários qualidades mínimas de serviço, consistem em tarefas bastante difíceis quando não existem meios para limitar o uso de recursos no sistema. A implantação de mecanismos de reserva de recursos pode facilitar o gerenciamento ao garantir limites inferiores e superiores de uso de processamento, memória, disco ou rede. O processamento, em especial, consiste em um item central de estudo deste trabalho já que muitas das aplicações submetidas a ambientes compartilhados são limitadas à CPU (*CPU-Bound*).

No caso de grades computacionais, a implantação de reserva de processamento pode evitar a sobrecarga dos nós do sistema ao mesmo tempo que garante aos provedores de máquinas o direito de estabelecer o limite da capacidade de processamento que desejam disponibilizar. No caso de computação sob-demanda, a reserva garantiria ao usuário uma qualidade de serviço mínima, assim como no caso de aplicações multimídia, as quais necessitam de processamento mínimo, periódico e constante.

Sistemas Operacionais de Propósito Geral não tratam classes de aplicações de maneira específica e assim gerenciam de forma ineficiente a Qualidade de Serviço de algumas aplicações. Casos em que o escalonamento é realizado por um algoritmo de compartilhamento de tempo ou espaço não distingüem classes de importância entre as aplicações, além de serem muito conservativos, visto que as fatias de tempo ou espaço alocadas, mesmo em situações em que não estão sendo utilizadas, não podem ser cedidas a outros processos. Por outro lado, em sistemas com escalonamento de processos baseado em prioridade, pode haver uma espera demasiada por parte de aplicações classificadas com baixa prioridade.

Algumas iniciativas, tais como os *Resource Kernels* ou *Resource Containers*, visam o desenvolvimento de mecanismos para a reserva de processamento através do desenvolvimento de extensões de *kernel* com o objetivo de prover acesso garantido e no tempo esperado aos recursos de um Sistema Operacional [Oikawa and Rajkumar 1999, Lee et al. 1996, Banga et al. 1999]. Observa-se, porém, o surgimento de uma tendência ao uso de máquinas virtuais para essa finalidade. Máquinas virtuais disponibilizam ao usuário um ambiente personalizado, onde os recursos computacionais são dedicados somente aos processos do usuário corrente. O isolamento de ambiente resulta em maior controle do uso dos recursos além de aumentar o nível de segurança, pois um processo de uma determinada máquina virtual é incapaz de acessar os dados de outra máquina virtual [Keahey et al. 2004, Santhanam et al. 2005].

De maneira diferente à abordagem dos *Resource Kernels* e das máquinas virtuais, há iniciativas que procuram gerenciar a reserva de recursos no nível do usuário, evitando a necessidade de recompilação de *kernel* ou a sobrecarga do sistema ao instanciar um grande número de máquinas virtuais. Um exemplo desse tipo de solução é o DSRT (*Dynamic Soft Real Time CPU Scheduler*) [Chu and Nahrstedt 1997]. Criado no final da década de 90, visando o tratamento de reservas de processamento para aplicações multimídia, o DSRT foi utilizado em muitos projetos, mas não evoluiu para que fosse possível o seu uso em cenários mais atuais como os encontrados em grades oportunísticas, ambientes de *utility computing* e arquiteturas multiprocessadas. No DSRT, por exemplo, não é possível aumentar a reserva de um processo caso haja CPU ociosa na máquina executora. Também não é possível dividir uma reserva entre todos os processos filhos criados a partir de um processo origem, e nem realizar reservas de processadores a fim de explorar características presentes em arquiteturas multiprocessadas.

As limitações encontradas no DSRT, aliadas à dificuldade de inserção de novas políticas de compartilhamento de recursos nesse sistema, motivaram o desenvolvimento de um novo gerenciador de reservas, o CPUReserve, objeto de estudo deste artigo. O CPUReserve consiste em uma reimplementação das idéias propostas pelo DSRT acrescida de novas funcionalidades inseridas para que o sistema pudesse ser utilizado em cenários de computação compartilhada, oportunística e de arquitetura multiprocessada. No CPUReserve, a comunicação é realizada através de *sockets*, o que facilita o seu uso em ambientes distribuídos. Quando há processamento ocioso, as reservas ativas dos clientes podem ser expandidas para que esse recurso seja melhor utilizado. O CPUReserve também permite reservar parte dos processadores de máquinas multiprocessadas. Por fim, a forma como o CPUReserve foi projetado priorizou a modularização do código de forma que o mecanismo de implementação de reserva fosse independente das políticas de reserva. A separação entre mecanismo e política de reserva abre espaço para que novos parâmetros de priorização sejam inseridos no CPUReserve.

As próximas seções apresentam as características do CPUReserve. Na Seção 2, são descritos os trabalhos relacionados ao CPUReserve. Na Seção 3 são apresentados os detalhes de implementação desse sistema e sua arquitetura, seguidos da Seção 4, onde são descritas a avaliação experimental, casos de uso e limitações do CPUReserve. Na Seção 5 é apresentada uma proposta de integração das abordagens de reserva de processamento no nível do usuário e com o uso de máquinas virtuais. Por fim, na Seção 6, a conclusão do trabalho é exposta.

2. Trabalhos Relacionados

A reserva de processamento está freqüentemente relacionada aos *Resource Kernels*. Um *resource kernel* é um *kernel* modificado para gerenciar recursos através de modelos de reserva. Com ele é possível garantir o acesso a uma determinada parte dos recursos de uma máquina durante a execução de uma aplicação através de funcionalidades providas pelo núcleo do SO. RT-Mach, Linux/RK consistem em dois dos principais *Resource kernels* encontrados na literatura [Oikawa and Rajkumar 1999, Lee et al. 1996].

Iniciativas recentes têm utilizado máquinas virtuais como meios para assegurar a reserva de processamento. Isso porque máquinas virtuais apresentam ambientes fechados de execução garantindo segurança e isolamento de desempenho das aplicações [Keahey et al. 2004, Santhanam et al. 2005]. O Xen consiste em um dos monitores de maior relevância na literatura [Barham et al. 2003].

Resource Kernels e máquinas virtuais gerenciam a reserva de processamento de maneira eficiente, mas apresentam limitações de uso, seja pela necessidade de recompilação do Sistema Operacional, seja pela sobrecarga para a criação de novos domínios. De maneira diferente a essas duas abordagens, o DSRT (*Dynamic Soft Real Time CPU Scheduler*), um sistema para o gerenciamento de aplicações desenvolvido no Departamento de Ciência da Computação da Universidade de Illinois em Urbana Champaign, gerencia a reserva de processamento no nível do usuário [Chu and Nahrstedt 1997].

No DSRT, as reservas de processamento são garantidas por meio de temporizadores que, ao expirarem, manipulam as prioridades das aplicações clientes junto ao Sistema Operacional corrente. Nesse sistema, as aplicações clientes e servidor devem residir em uma mesma máquina visto que a comunicação dessas entidades é feita por compartilhamento de memória. Essa implementação dificulta o uso do DSRT em ambientes distribuídos.

O DSRT foi projetado para atender as necessidades de aplicações multimídia. Assim, ele possibilita a classificação das aplicações de acordo com o perfil de uso de processamento, que pode seguir diferentes padrões de periodicidade. Em função dessa classificação, o DSRT vai fazer a reserva de recursos apropriada. Ele também oferece uma API de reserva que pode ser inserida no código das aplicações para que o controle dos recursos seja feito de forma mais precisa (mecanismo de *probe*).

Por outro lado, o DSRT não explora o processamento ocioso de máquinas, dificultando o seu uso em cenários de computação oportunística. Também não existe no DSRT mecanismos para a reserva de processadores, uma funcionalidade importante para o gerenciamento dos recursos de arquiteturas multiprocessadas.

3. CPUReserve

As desvantagens encontradas na reserva de processamento realizadas pelos *Resource Kernels* e pelas máquinas virtuais motivaram a implementação do CPUReserve, um sistema para reserva de processamento no nível do usuário. Assim como o DSRT, o CPUReserve gerencia processos através de chamadas ao sistema que alteram dinamicamente a prioridade dos processos para consumirem mais ou menos processamento dentro de um período de tempo.

O CPUReserve segue o modelo cliente-servidor com comunicação entre os processos sendo feita via *socket* o que permite seu uso em ambientes distribuídos. Para executar o servidor é necessário disparar o processo em uma determinada porta, especificando em quais processadores da máquina deseja-se gerenciar as reservas[1]:

```
./server <porta> <cjto processadores decimal>
```

Para executar o cliente, é necessário executar um comando do tipo

```
./client <maq:porta> <periodo> <fatia> <cons> <exec> <params...>
```

onde `maq:porta` corresponde ao nome da máquina juntamente com o número da porta onde o servidor escuta, `fatia`, o tempo de processamento necessário à aplicação a cada período de ms especificado. O campo $cons$ (*conservação*), se configurado com 0, limita a quantidade de CPU alocada ao processo àquela especificada na mensagem. Caso esse campo seja igual a 1, a quantidade de processamento por período pode ser maior caso haja processamento ocioso na máquina executora. Os campos `exec` e `params` correspondem juntos à linha de comando da aplicação a ser executada.

Se o pedido do cliente for aceito, a aplicação receberá a porcentagem de CPU especificada na requisição. A alteração dos parâmetros de reserva pode ser feita através de um comando do tipo

```
./adapt <maq:porta> <periodo> <fatia> <cons> <pid>
```

onde `pid` é o identificador do processo sendo executado pelo servidor.

Para que não haja inanição dos processos executados por fora do servidor CPUReserve, incluindo o próprio servidor, um limite de reserva é estipulado pela variável RESERVATION_LIMIT que, por padrão é de 0.8*num_processadores_reservados[2].

Tanto a reserva quanto a adaptação, quando executadas pelo servidor, se estendem a todo processo criado pelo processo em execução (via *fork*, por exemplo). Os novos processos compartilham a fatia de tempo reservada ao processo que os criou. Essa prática evita que processos filhos burlem o acordo de uso de processamento.

3.1. Arquitetura

O servidor CPUReserve é composto por duas *threads* principais: uma de monitoração de uso da CPU (CPU ociosa e tempo de CPU ocupado por cada processo) e outra de espera de conexões dos clientes. A *thread* de espera de conexões escuta, na porta determinada no momento de início do servidor, por requisições de reserva de processamento ou adaptação de reserva.

A figura 1 ilustra a arquitetura do servidor e a interação entre seus componentes. Ao receber uma mensagem do cliente na *thread* de tratamento de IO, o servidor inicia outra *thread* para tratar da requisição. Essa *thread* trata de maneira diferente pedidos de reserva e de adaptação. No caso dos pedidos de reserva, ela verifica se os parâmetros da mensagem são consistentes e se podem ser atendidos com a qualidade desejada. No

[1]Cada processador é representado por um *bit*. Os processadores a serem reservados têm o seu *bit* configurado para 1. Em seguida, o valor binário é convertido para decimal. No caso de se desejar reservar os processadores 0 e 2, por exemplo, deve-se passar o valor 5 no terceiro parâmetro já que $5 = 0101$ em binário.

[2]Esse valor foi determinado por meio de experimentos descritos na Seção 4.

caso afirmativo, o processo especificado é criado e o mesmo passa a ter o consumo de processamento monitorado. Monitora-se também novos filhos desse processo para que os mesmos compartilhem a fatia de tempo estipulada para o processo pai. O controle da reserva de processamento é realizado por meio de alarmes que expiram após uma fatia de tempo de processamento.

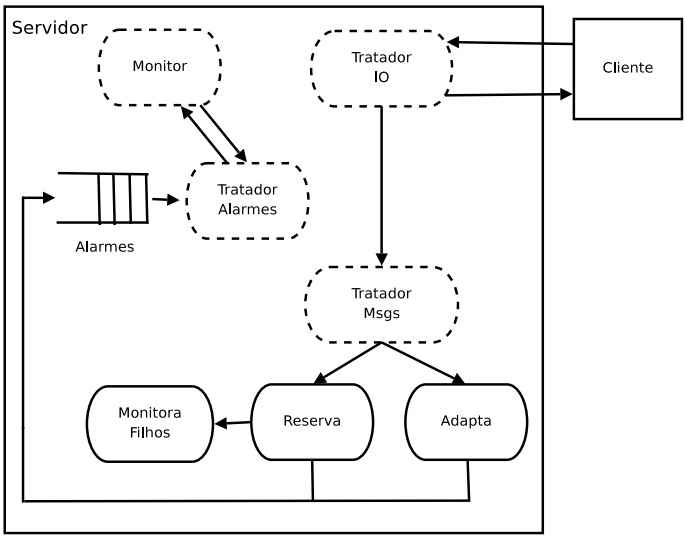

Figura 1. Arquitetura do servidor de reserva.

No caso do pedido de adaptação, a *thread* tratadora da requisição verifica se o processo requisitado existe. Se o processo existir, ela verifica se os parâmetros de adaptação são válidos e possíveis de serem atendidos. Nesse caso, os novos parâmetros de reserva do processo são atualizados e novos alarmes são configurados.

Os alarmes são ordenados em uma fila, onde os primeiros são os alarmes que expiram mais cedo. Quando um alarme expira, cria-se uma *thread* para tratá-lo. Essa *thread* verifica o tempo de processamento do processo que gerou o alarme e toma decisões quanto à continuidade ou não de execução do processo em questão. Trocas de informações acontecem entre as *threads* de monitoração e a tratadora de alarme. Elas ocorrem nos momentos em que a *thread* tratadora de alarme deseja saber se há capacidade de processamento ociosa na máquina a fim de cedê-la a um processo do tipo conservativo.

Dependendo da configuração da propriedade TRANSFER_OUTPUT do servidor, os clientes podem ficar bloqueados em espera até que os seus pedidos sejam finalizados e os arquivos de saída (*output* e erro), gerados na execução dos processos, sejam transferidos para os seus sistemas de arquivo locais.

3.2. Detalhes de Implementação

Ao ser iniciado, o servidor tem a sua execução ligada a um determinado conjunto de processadores. A ligação de processadores restringe não só a execução do servidor, mas também a dos processos gerenciados por ele. Ela é garantida por meio de chamadas ao Sistema Operacional as quais realizam a afinidade de processadores.

Para gerenciar o escalonamento dos processos solicitados pelos clientes, o servidor é executado com a máxima prioridade para processos de tempo real. Ao colocar

um processo cliente em execução, o servidor o torna de tempo real com a segunda maior prioridade do sistema. Em seguida, esse processo cliente é monitorado e, caso tenha superado a fatia de tempo estipulada para o período, ele pode ser parado ou então ter a sua prioridade reduzida ao mínimo permitido para classes de processos comuns do Linux até o próximo período. A decisão de ser parado ou ter a prioridade reduzida é feita com base no parâmetro conservação informado pelo pedido de reserva ou adaptação.

Toda a monitoração dos processos é realizada por meio de alarmes de tempo real que, quando expirados, invocam uma *thread* de decisão. Essa *thread* apresenta implementação guiada pelas decisões ilustradas na figura 2.

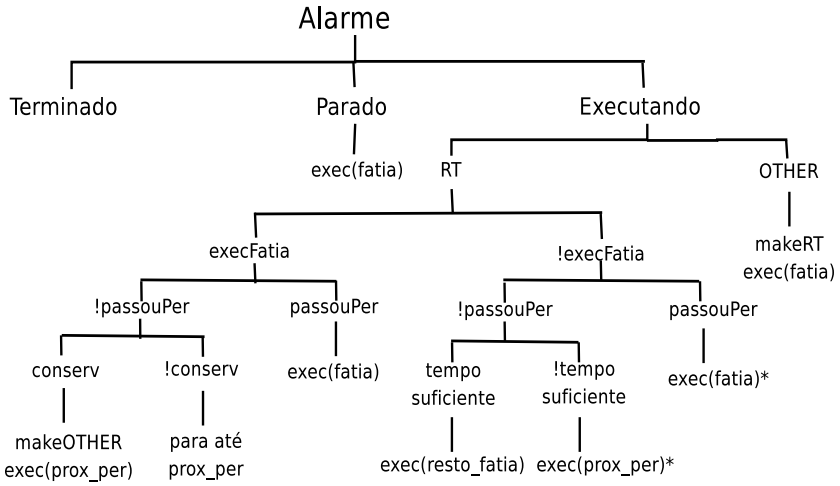

Figura 2. Etapas de decisão da *thread* tratadora dos alarmes.

Ao expirar um alarme, a *thread* verifica qual é o estado do processo que solicitou o alarme. Se o estado for terminado, então o processo é retirado da fila de processos do servidor e, dependendo da configuração do sistema, pode ter os seus arquivos de saída transferidos para a máquina do cliente. Se o estado for parado, é sinal que no período anterior, o processo executou a sua fatia de tempo de processamento, foi parado até o fim do período por não ser conservativo e, no novo período, deve executar uma nova fatia de tempo. Se o estado do processo for executando, então é preciso verificar se ele vem sendo executado como processo de tempo real(RT) ou comum(OTHER). Se for processo do tipo OTHER, então o alarme expirou porque o processo teve um período finalizado. Dessa maneira, é preciso transformar o processo em tempo real novamente e dar a ele uma nova fatia de tempo para execução.

Caso o processo venha executando em tempo real, então é preciso verificar se ele já executou a sua fatia de tempo. Caso tenha executado, verifica-se se o limite de período já passou. Se o limite estiver sido atingido, o processo pode executar por mais uma fatia de tempo. Se o limite de período não tiver sido ultrapassado, então verifica-se se o processo é do tipo conservativo. Se o processo for conservativo, ele é transformado em um processo comum de baixa prioridade e deixado em execução nesse estado. Por outro lado, se o processo não for conservativo, ele é parado até o próximo limite de período.

Se o processo vem executando em tempo real, mas não executou a sua fatia de tempo ainda, então verifica-se se o período já foi ultrapassado. Caso afirmativo, ocorreu

um erro (indicado na figura 2 com um asterisco). Esse erro indica que há mais reserva do que os processadores disponíveis podem suportar. Nesse caso, deve-se reduzir o valor da variável RESERVATION_LIMIT. Para que não haja pausa na execução do servidor, permite-se que o processo, mesmo com um erro, possa continuar a executar reiniciando uma fatia de tempo. Caso o período do processo já tenha expirado, permite-se que o processo execute pelo restante da fatia de tempo, se ainda for possível executar o que falta da fatia antes do período expirar; ou pelo restante do período, situação onde ocorreu um erro semelhante ao anteriormente descrito.

4. Avaliações Experimentais

Testes foram realizados a fim de avaliar a escalabilidade do CPUReserve e o seu uso em um cenário real de computação compartilhada distribuída. Os testes mostraram que o servidor apresenta significativa escalabilidade, mas também evidenciaram algumas limitações do sistema proposto assim como será exposto nas próximas seções.

4.1. Testes de Escalabilidade

Os testes de escalabilidade foram realizados em um computador Intel Centrino Duo de 1,66 GHZ, 1G de memória executando o Sistema Operacional Linux, distribuição Ubuntu 7.10 padrão, com *kernel* 2.6.22. O objetivo foi estimar o quanto o servidor consome de recursos para atender a um número crescente de clientes. Para isso, o servidor CPUReserve foi inicializado para gerenciar os dois processadores da máquina. Os clientes consistiram em uma aplicação de espera ocupada. As especificações dos clientes eram de que o servidor reservasse a eles $10ms$ a cada $1000ms$, ou seja, 1% da CPU para cada processo cliente. A escolha de uma porcentagem tão pequena de processamento foi feita com o objetivo de que muitos alarmes fossem disparados no menor intervalo de tempo possível[3]. Com isso, o servidor trabalharia em um estado mais próximo ao de sua saturação pois teria que gerar muitas *threads* para os tratamentos das reservas. Os experimentos mostraram que o servidor por si só, sem atender a nenhum cliente, consome valor muito próximo a 0% de CPU, 18 MB de memória virtual e 732 KB de memória residente. Os resultados dos experimentos, divididos pelo número de clientes, para as taxas de porcentagem de CPU, quantidade de *soft page fault* por segundo, memória virtual e memória residente do servidor são apresentados na tabela 1.

Nota-se que o consumo de CPU, assim como o número de *page faults* por segundo, cresce linearmente com o aumento do número de clientes. A quantidade de CPU por processo nos testes oscilou entre 0.1 e 0.2%, enquanto a taxa de *page faults* manteve-se em torno de 30 por segundo. O baixo consumo de CPU era esperado visto que o servidor apresenta pouco processamento no seu código.

Como conseqüência do aumento do compartilhamento de memória quando o número de clientes foi aumentado, notou-se uma discreta redução na quantidade de memória virtual alocada por cliente. Com 5 clientes, essa quantidade era próxima de $15MB$ por processo, já com 60 clientes, essa quantidade passou para aproximadamente $9MB$.

[3]O tempo de execução de um processo no CPUReserve é obtido a partir do seu arquivo /proc/pid/stat. Esse tempo é dado em *jiffies* que, no ambiente utilizado para testes, corresponde a $10ms$.

Num. Clientes	%CPU	Page Fault/ sec.	Mem. Virtual(KB)	Mem. Residente(KB)
1	1	51	34.77	784
5	0.2	29.6	15.35	165.6
10	0.2	20.9	11.78	86.8
15	0.13	29.33	10.58	61.07
20	0.15	29.6	9.99	48
25	0.12	22.08	9.96	40.16
30	0.13	27.4	9.66	34.67
35	0.11	27.26	9.45	30.97
40	0.13	27.08	9.4	28.3
45	0.13	24.84	9.17	25.96
50	0.14	27.82	9.16	24.32
55	0.13	32.07	9.14	22.84
60	0.12	31.35	9.08	21.8

Tabela 1. Resultados dos testes de escalabilidade do servidor.

A quantidade de memória residente alocada para cada cliente no servidor é reduzida com o aumento do número de processos. Essa redução acontece por três motivos: primeiro porque o gerenciamento de novos processos pelo servidor acarreta na criação de processos leves do tipo *thread* que consomem poucos recursos do sistema; segundo porque o montante de memória necessário para carregar o servidor foi sendo dividido entre um número crescente de processos; e terceiro porque o curto espaço de tempo entre cada período dos clientes faz que alguns alarmes de monitoração expirem ao mesmo tempo, sendo então tratados todos por uma única *thread*.

Apesar dos testes com o servidor mostrarem que o CPUReserve consome poucos recursos do sistema, não foi possível realizar experimentos com mais de 60 clientes. Nesses casos, aconteceram retardos no tempo de resposta do Sistema Operacional e, conseqüentemente, mal comportamento das aplicações sendo executadas nele, inclusive na execução de alguns processos clientes que, em vez de executarem com apenas 1% de CPU executavam com 3 ou 4%. Esse mal funcionamento da reserva é causado por dois motivos: o alto tempo gasto para a criação de novas *threads* para o tratamento dos alarmes; e o excesso de trabalho que o servidor tem que realizar a cada intervalo de $10ms$ - quando a quantidade de processos se aproxima de 60, o servidor não é mais capaz de gerenciar as prioridades e os temporizadores de todos os processos com período/fatia de tempo expirados dentro do intervalo de $10ms$, fazendo com que alguns processos permaneçam em execução por mais tempo do que deveriam.

À medida que o servidor CPUReserve torna-se saturado, a *thread* receptora de novos pedidos de clientes, implementada como um processo de baixa prioridade, deixa de receber novas requisições. Essa prática é importante para evitar que o servidor fique ainda mais saturado em situações de alta carga de trabalho a ser gerenciada.

Como o servidor consumiu menos de 10% de CPU para o gerenciamento limite de 60 clientes, pode-se garantir, nesse caso, que um valor limite para a variável RESERVATION_LIMIT esteja em torno de 0.9.

4.2. Casos de Uso

No CSBase [Lima et al. 2006], um *framework* para gerenciamento de recursos e execução de aplicações em ambientes distribuídos, estuda-se a inserção de um mecanismo de reserva de processamento nos servidores de execução com o objetivo de limitar a quantidade de CPU que uma aplicação utiliza. Essa funcionalidade, provida pelo CPUReserve, possibilitaria a divisão mais justa de recursos entre os diversos usuários dos nós de execução que compõem a infra-estrutura do sistema.

Ainda no projeto CSBase, cada servidor executor de aplicações tem a sua capacidade de processamento medida por meio de um *benchmark* baseado no Whetstone. O resultado desse *benchmark*, denominado CSBench, é utilizado como métrica no escalonamento das aplicações. Como não há garantia de que o *benchmark* será executado sem nenhuma interferência de cargas de trabalho no servidor, procura-se atribuir uma relação entre a taxa de CPU ocupada e o resultado gerado pelo *benchmark*.

Para verificar se tal relação acontecia no CSBench, foram realizadas execuções desse *benchmark* com reservas no CPUReserve variando de 10 a 100%. Os resultados obtidos são apresentados nas segunda e terceira colunas da tabela 2. Nota-se que não há relação entre a quantidade de *clocks* totais com a quantidade gasta em processamento. Ao utilizar 1117 *clocks* com 100% de CPU, o *benchmark* deveria utilizar cerca de 10 vezes mais *clocks* com 10% de processamento, mas utilizou cerca de 14 vezes mais. Além disso, o número de *clocks* de processamento deveria se manter mais ou menos constante, mas variou cerca de 50%.

Reserva (%)	CSBench		Linpack		Espera Ocupada	
	Clock Total	Clock Proc.	Clock Total	Clock Proc.	Clock Total	Clock Proc.
100	1117	1090	1982	1977	3516	3504
90	1391	1267	2177	1966	3654	3292
80	2048	1646	2813	2253	4348	3483
70	2215	1556	3101	2173	5004	3499
60	2616	1576	3716	2238	5744	3461
50	3266	1664	4413	2217	6918	3459
40	3521	1480	5506	2205	8724	3503
30	5430	1648	7003	2106	11617	3493
20	7816	1573	10108	2037	17506	3503
10	15935	1625	22303	2251	35011	3514

Tabela 2. Resultados de reserva do CSBench.

Para confrontar dados do CSBench, foi realizado o mesmo teste com o Linpack, um outro *benchmark* de CPU. Os resultados, ao contrário do CSBench, foram consistentes, como apresentado nas colunas 4 e 5 da tabela 2.

Também foram realizados testes com uma aplicação que consistia em um laço que realizava multiplicações por 1. Os resultados são apresentados nas duas últimas colunas da tabela 2. Nota-se que há uma relação entre a quantidade de reserva de CPU e a quantidade total de *clocks* gastos pela aplicação, enquanto que a quantidade de *clocks* gastos com processamento sofre pequenas variações.

Para que houvesse garantia de que o CSBench não estava sendo prejudicado por criar processos em número igual ao de processadores, a aplicação de multiplicação foi alterada para também criar processos, mas os resultados foram bastante parecidos com os apresentados pela aplicação sem a criação de filhos. Uma possível explicação para o mau comportamento do CSBench nos experimentos pode ser a alta influência da *cache* de dados/instruções na execução do programa. Entender esse comportamento, porém, é objetivo de outros estudos que fogem ao escopo deste trabalho.

4.3. Limitações

Por ser desenvolvido no nível do usuário, o CPUReserve é limitado pelo Sistema Operacional da máquina. Cabe ao sistema operacional, por exemplo, prover interfaces para chamadas ao sistema responsáveis por configurar a afinidade de processadores, alterar a prioridade de processos e a política de escalonamento em atividade. A impossibilidade de atender aplicações com período/fatia de tempo menores que a de um *jiffy*[4] consiste em uma das limitações impostas pelo Sistema Operacional[5].

Ainda por executar no nível do usuário, o CPUReserve pode sofrer a interferência de outros processos na mudança de prioridade das aplicações sendo gerenciadas (comandos do tipo *nice*, por exemplo). Para que a reserva seja feita de forma consistente, é preciso garantir que o servidor CPUReserve seja executado como o de maior prioridade no sistema, sendo seguido em prioridade pelas aplicações as quais ele gerencia.

A execução do servidor é imprevista em situações em que há muita carga de trabalho na máquina. Nesses casos, pode acontecer de um alarme expirar sem que o tratador possa ser chamado naquele momento. Para que isso não aconteça, é importante configurar a varíavel RESERVATION_LIMIT de modo que o Sistema Operacional não sofra limitações para a sua execução.

5. CPUReserve e Máquinas Virtuais

Durante a fase de criação de uma máquina virtual, é possível especificar o quanto de memória, disco e processamento que a máquina conterá. Com essa especificação, monitores de máquinas virtuais são capazes de prover isolamento de desempenho entre os diferentes domínios presentes no *hardware* através da multiplexação dos recursos das máquinas.

Apesar do isolamento de desempenho ser garantido com o uso de máquinas virtuais, o custo para o gerenciamento das mesmas é alto e pode inviabilizar a prática de isolar a execução de aplicações colocando-as em diferentes máquinas virtuais [Gupta et al. 2006]. Por outro lado, a execução de mais de uma aplicação por máquina virtual deixa de garantir o isolamento de desempenho. Como alternativa, é possível visualizar ambientes de isolamento híbridos onde coexistem máquinas virtuais e gerenciadores de reserva no nível do usuário.

A figura 3 apresenta um ambiente híbrido contendo uma hierarquia de reservas, onde uma máquina física é representada com duas máquinas virtuais. Na primeira delas, existe uma instância do CPUReserve responsável por gerenciar de forma mais fina

[4]Um *jiffy* corresponde à duração de um *tick* do relógio do sistema. Tipicamente, esse valor é igual a $10ms$ em sistemas Linux.

[5]Os tempos de execução dos processos nos arquivos /proc/pid/stat são dados em *jiffies*.

a reserva de processamento entre as aplicações de uma mesma máquina virtual. Nessa máquina, é possível que diversas aplicações executem concorrentemente, sem que o desempenho entre elas seja afetado. Pode-se, por exemplo, especificar que a primeira máquina será executada com 50% de uma CPU e a aplicação1 com 30% desses 50%, ou seja, com aproximadamente 16% da capacidade de processamento de uma CPU. Já na segunda máquina virtual, o isolamento de desempenho é garantido se somente uma aplicação for executada, banalizando assim a criação de novas máquinas virtuais quando novas execuções forem solicitadas.

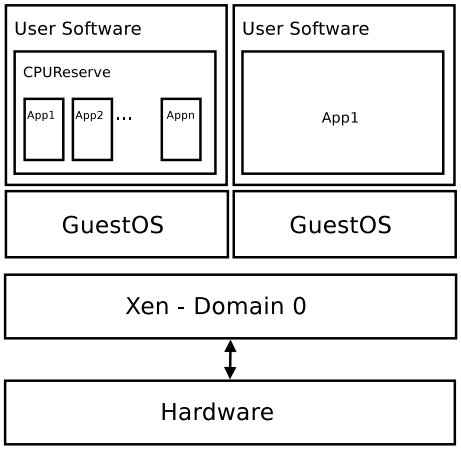

Figura 3. Hierarquia de reserva de processamento em máquinas virtuais.

A fim de validar a proposta de um ambiente híbrido, um protótipo como o apresentado pela figura 3 foi construído com o Xen 3.2 e *kernel* 2.6.16. Nesse protótipo, porém, as reservas não foram realizadas como esperado dado que o tempo de execução dos processos das máquinas virtuais não espelham o seu uso de CPU dentro dessas máquinas e sim dentro da máquina física como um todo. Um processo de espera ocupada, por exemplo, se executado dentro de uma máquina virtual que tem 50% de CPU dedicada a ela, não aparece nos arquivos de monitoração como consumidor de 100% do processamento da máquina virtual, mas sim de 50%. Entender como as informações de monitoramento de processos são geradas no Xen e adaptar o CPUReserve para gerenciar processos nesse monitor consiste em um objeto de trabalho futuro.

6. Conclusão

Este artigo descreveu a implementação de um sistema para reserva de processamento no nível do usuário baseado nas idéias propostas pelo DSRT [Chu and Nahrstedt 1997]. Ao contrário das abordagens tradicionalmente encontradas na literatura, o sistema resultante, denominado CPUReserve, não necessita de recompilação do *kernel* do Sistema Operacional e não causa sobrecargas no servidor, mesmo quando muitos clientes são monitorados. A escalabilidade apresentada nos testes, assim como a arquitetura cliente-servidor, são características fundamentais para o uso do CPUReserve em ambientes de computação compartilhada distribuída. Um caso de uso em um cenário como esse foi descrito na Seção 4.2, além disso, na Seção 5 uma abordagem híbrida do CPUReserve com máquinas virtuais é proposta.

Além de reservar fatias de tempo, o CPUReserve também permite reservar CPUs em máquinas multiprocessadas. Essa facilidade foi útil para testar as opções de reserva

do próprio sistema quando pôde-se saturar somente um processador da máquina e ver como o servidor se comportava em tal situação sem que o sistema operacional ficasse inoperante por excesso de carga de trabalho. Espera-se que essa característica de gerenciar processadores seja útil em outros testes de escalabilidade.

Por outro lado, algumas características do DSRT ainda não foram implementadas no CPUReserve pela ausência de necessidade imediata. É o caso da API de reserva que pode ser inserida no código das aplicações para que o controle dos recursos seja feito de forma mais precisa. Outra característica é a possibilidade de classificação das aplicações de acordo com o perfil de uso de processamento - periódico ou aperiódico.

Por fim, a forma como o CPUReserve foi implementado, buscando separar as políticas de reserva dos mecanismos em si, facilita a inclusão/substituição de políticas de reserva. Desenvolver novas políticas, assim como portar o CPUReserve para ambientes Windows, consistem em trabalhos futuros a serem desenvolvidos.

Referências

Banga, G., Druschel, P., and Mogul, J. C. (1999). Resource containers: A new facility for resource management in server systems. In *Proceedings of OSDI '99*, pages 45–58, New Orleans, USA. USENIX Association.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of SOSP '03*, pages 164–177, New York, USA. ACM.

Chu, H.-H. and Nahrstedt, K. (1997). A soft real time scheduling server in unix operating system. In *Proceedings of IDMS '97*, pages 153–162, London, UK. Springer-Verlag.

Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. (2006). Enforcing performance isolation across virtual machines in xen. In van Steen, M. and Henning, M., editors, *Proceedings of Middleware '06*, volume 4290 of *Lecture Notes in Computer Science*, pages 342–362. Springer Berlin / Heidelberg.

Keahey, K., Doering, K., and Foster, I. (2004). From sandbox to playground: Dynamic virtual environments in the grid. In *Proceedings of GRID '04*, pages 34–42, Washington, USA. IEEE Computer Society.

Lee, C., Rajkumar, R., and Mercer, C. (1996). Experience with processor reservation and dynamic QoS in real-time mach. In *Proceedings of Multimedia Japan 96*, Japan.

Lima, M. J., Ururahy, C., de Moura, A. L., Melcop, T., Cassino, C., dos Santos, M. N., Silvestre, B., Reis, V., and Cerqueira, R. (2006). Csbase: A framework for building customized grid environments. In *Proceedings of WETICE '06*, pages 187–194, Washington, USA. IEEE Computer Society.

Oikawa, S. and Rajkumar, R. (1999). Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of RTAS '99*, page 111, Washington, USA. IEEE Computer Society.

Santhanam, S., Elango, P., Arpaci-Dusseau, A., and Livny, M. (2005). Deploying virtual machines as sandboxes for the grid. In *Proceedings of WORLDS '05*, San Francisco, USA.