Replicação de Máquinas Virtuais Xen com *Checkpointing* Adaptável

Marcelo P. da Silva, Rafael R. Obelheiro*, Guilherme P. Koslovski*

[1]Programa de Pós Graduação em Computação Aplicada (PPGCA)
Universidade do Estado de Santa Catarina (UDESC) – Joinville, SC

pereira@joinville.udesc.br, *firstname.lastname@udesc.br

Abstract. *Remus is a virtual machine (VM) replication mechanism that provides high availability despite crash failures. Replication is performed by checkpointing the VM at fixed intervals. However, there is an antagonism between processing and communication regarding the optimal checkpointing interval: while longer intervals benefit processor-intensive applications, shorter intervals favor network-intensive applications. Thus, any chosen interval may not always be suitable for the hosted applications, limiting Remus usage in many scenarios. In this paper, we introduce adaptive checkpointing for Remus, dynamically adjusting the replication frequency according to the characteristics of running applications. Experimental results indicate that our proposal improves performance for applications that require both processing and communication, without harming applications that use only one type of resource.*

Resumo. *Remus é um mecanismo de replicação de máquinas virtuais (MVs) que fornece alta disponibilidade diante de faltas de parada. A replicação é realizada através de checkpointing, seguindo um intervalo fixo de tempo predeterminado. Todavia, existe um antagonismo entre processamento e comunicação em relação ao intervalo ideal entre checkpoints: enquanto intervalos maiores beneficiam aplicações com processamento intensivo, intervalos menores favorecem as aplicações cujo desempenho é dominado pela rede. Logo, o intervalo utilizado nem sempre é o adequado para as características de uso de recursos da aplicação em execução na MV, limitando a aplicabilidade de Remus em determinados cenários. Este trabalho apresenta uma proposta de checkpointing adaptativo para Remus, ajustando dinamicamente a frequência de replicação de acordo com as características das aplicações em execução. Os resultados indicam que a proposta obtém um melhor desempenho de aplicações que utilizam tanto recursos de processamento como de comunicação, sem prejudicar aplicações que usam apenas um dos tipos de recursos.*

1. Introdução

Aplicações computacionais estão sujeitas a ter sua execução parcial ou totalmente comprometida pela ocorrência de faltas[1] nos recursos de processamento, comunicação ou armazenamento por elas utilizados. Para superar esse cenário, aplicações críticas recorrem a soluções (algoritmos, protocolos e infraestruturas) para garantir o seu funcionamento correto mesmo na ocorrência de faltas. Paralelamente, os centros de processamento de

[1]Neste trabalho utilizamos a nomenclatura falta (*fault*), erro e falha (*failure*).

dados têm explorado a virtualização de recursos computacionais para consolidação de servidores, economia no consumo de energia, e gerenciamento facilitado. Em um ambiente virtualizado, os usuários executam suas aplicações sobre um conjunto de máquinas virtuais (MVs), através da introdução de uma abstração entre o hardware físico (que hospeda a MV) e a aplicação final. A camada de abstração, chamada de monitor de máquinas virtuais (MMV), tem acesso a todo o estado de execução de uma MV, e por isso pode ser explorada para introduzir tolerância a faltas de forma transparente para o software hospedado. Motivado por tal oportunidade, surgiu Remus [Cully et al. 2008], um mecanismo para replicação de MVs residente no MMV Xen [Barham et al. 2003] e que é baseado no modelo de replicação primário-*backup* [Budhiraja et al. 1993].

Remus encapsula as aplicações e o sistema operacional do usuário em uma MV, salvando dezenas de *checkpoints* por segundo. Os *checkpoints* salvos no hospedeiro primário são transferidos para um segundo hospedeiro, chamado de *backup*, enquanto prossegue a execução no primário. Normalmente, os usuários interagem com a MV hospedada no primário, porém, quando uma falta de parada ocorre neste hospedeiro, a MV residente no *backup* é ativada e passa a responder pelo serviço em poucos milissegundos. Cada *checkpoint* replicado compreende um conjunto de alterações no estado da MV desde o último *checkpoint* salvo.

A alta frequência de replicação reduz o trabalho que deverá ser refeito no *backup* quando esta réplica tornar-se ativa [Elnozahy et al. 2002], propiciando rápido *failover*. A influência causada pela replicação no desempenho das aplicações hospedadas não é desprezível; estudos anteriores [da Silva et al. 2014, Gerofi and Ishikawa 2011] mostram que, quanto maior a frequência de *checkpointing*, menor a latência na comunicação com o cliente que interage com a MV protegida. Por outro lado, frequências menores de *checkpointing* causam menos interrupções na execução da MV, reduzindo o tempo de execução de aplicações dependentes apenas de processamento. Frente a estas características antagônicas, soma-se o fato que o intervalo utilizado por Remus é fixo. Embora as características das aplicações hospedadas em relação ao uso de recursos possam ser conhecidas, não é trivial a escolha de um intervalo de *checkpointing* ideal.

Aplicações com características mistas de uso de recursos aumentam a complexidade da escolha. Uma frequência alta de *checkpointing* priorizará a parte da aplicação voltada à comunicação, porém, prejudicará a outra que necessita de mais tempo de processamento. A escolha de um intervalo médio não representa uma boa opção, uma vez que a aplicação nunca experimentará uma frequência que seja a melhor possível para suas características durante todo o período de execução. Ademais, Remus não garante que o intervalo de *checkpointing* seja sempre respeitado: um novo *checkpoint* só é gerado depois que o anterior tiver sido transferido para o *backup* e devidamente salvo, o que pode levar mais tempo que o intervalo estipulado, consequentemente estendendo o tempo de execução da MV e assim prejudicando as aplicações comunicantes.

Frente a essas limitações, o presente trabalho traz como contribuição uma abordagem para adaptar a frequência de *checkpointing* guiada pelo fluxo de rede produzido pela MV. A solução introduz o uso de intervalo adaptativo, alternando entre alta e baixa frequência. O objetivo é priorizar a diminuição da latência na comunicação sempre que houver tráfego de saída de rede na MV, aumentando o tempo de processamento na ausência de tráfego.

Os resultados experimentais indicam que o uso de *checkpointing* adaptativo para a replicação de MVs com Remus, principalmente para aplicações com características mistas de uso de recursos, é justificado. Para aplicações cujo desempenho é influenciado pela rede, o *checkpointing* adaptativo obteve uma redução de aproximadamente 30% no tempo de resposta em relação à versão original de Remus. Para aplicações dependentes exclusivamente de processamento, o desempenho foi sempre semelhante à versão original com uma baixa frequência de *checkpointing*, que favorece este tipo de aplicação.

Este trabalho está organizado da seguinte forma: a Seção 2 detalha o mecanismo de replicação implementado pelo Remus e salienta suas limitações. A Seção 3 apresenta a solução implementada, enquanto os resultados experimentais são discutidos na Seção 4. A Seção 5 revisa os trabalhos relacionados. Por fim, a Seção 6 conclui o trabalho.

2. O Mecanismo de Replicação Remus e suas Limitações

Remus é um mecanismo para replicação de MVs agnóstico em relação ao sistema operacional em execução na MV, o qual é executado sem precisar de configuração adicional [Cully et al. 2008]. No cenário de execução de Remus, cada hospedeiro possui duas interfaces de rede, sendo uma exclusiva para o tráfego de replicação. É preciso que os discos das MVs estejam sincronizados nos dois hospedeiros antes do processo de replicação iniciar. Com os discos sincronizados, uma MV em estado de pausa é criada no hospedeiro *backup*, e uma cópia da memória da MV em execução no primário é transferida para o *backup*. Após esse processo inicial, começa o período de *checkpointing*. Em intervalos definidos (tipicamente algumas dezenas ou centenas de milissegundos), a MV sofre uma pausa para salvar um novo *checkpoint*. Nesta etapa, somente o conteúdo da memória que foi alterado desde o último *checkpoint* salvo no *backup* será transferido (isso corresponde à fase *stop-and-copy* do mecanismo de *live migration* de MVs [Clark et al. 2005]). Porém, antes, estes dados são salvos em um *buffer* local no hospedeiro primário, permitindo que a MV retome a execução enquanto o *buffer* é transferido para o *backup*. Assim que o *buffer* estiver salvo no *backup*, é enviada uma confirmação para o primário. A replicação do disco é realizada por uma adaptação da ferramenta DRBD (*Distributed Replicated Block Device*) [Reisner and Ellenberg 2005], onde a cada *checkpoint* o disco da MV residente no hospedeiro primário é sincronizado com o disco da MV no *backup*. Quando ocorre uma falta de parada no hospedeiro primário, a MV residente no *backup* é ativada, assumindo o endereço IP da MV protegida.

Durante o período que compreende a retomada da execução até a próxima pausa para salvar um novo *checkpoint*, e até que este estado seja completamente transferido, não há garantia de que o que foi ou está sendo processado na MV esteja protegido. Ocorrendo uma eventual falta de parada no primário, os dados processados na MV antes ou durante a transferência de um *checkpoint* serão perdidos, ou seja, o *backup* retoma a execução considerando apenas o último *checkpoint* completamente recebido. Devido a este risco, diz-se que a execução da MV no primário está constantemente em modo especulativo.

2.1. Visão do Cliente: Linearizabilidade

Remus permite que os clientes percebam um comportamento consistente do sistema em caso de faltas. Essa consistência, chamada de linearizabilidade, define que somente será processado no primário o que já foi previamente salvo em armazenamento

estável [Guerraoui and Schiper 1997], no caso o disco ou memória do hospedeiro *backup* [Elnozahy et al. 2002]. Remus considera apenas o tráfego de saída de rede como forma de garantir a linearizabilidade. A MV recebe e processa o tráfego de entrada sem qualquer restrição, o que, além de reduzir a sobrecarga, também preserva a semântica das aplicações em execução [Guerraoui and Schiper 1997]. Durante a execução, o tráfego de rede que sai da MV fica retido em um *buffer* no primário, chamado *buffer* de rede. Somente após o *backup* confirmar o recebimento do *checkpoint* (ou seja, o *buffer* local ter sido totalmente recebido e o disco sincronizado) é que este tráfego é liberado. O *buffer* de rede se faz necessário para garantir, na visão do cliente, um estado idêntico em relação às conexões de rede caso ocorra uma falta do primário [Cully et al. 2008]. É uma forma de garantir que somente estados salvos no *backup* sejam visualizados pelos clientes. Contudo, este *buffer* é responsável pelo aumento da latência na comunicação. Em [da Silva et al. 2014] foi apresentado um estudo sobre o desempenho da rede sob várias frequências de *checkpointing*, mostrando que, quanto maior a frequência, menor a latência da comunicação. Em relação ao tráfego de entrada, não há influência deste *buffer*, ou seja, a MV está permanentemente apta a receber dados sempre que estiver em execução.

2.2. Frequência de *Checkpointing*

O tempo para transferência de um *checkpoint* é variável e depende da carga de trabalho em execução na MV. A Figura 1 apresenta Remus em uma situação típica, com um intervalo estipulado em 50 ms. Tomando como exemplo o período 3, observa-se que o tempo de execução da MV antes de salvar um novo *checkpoint* foi exatamente de 50 ms. Isso ocorre porque o tempo de transferência do *checkpoint* referente ao período 2 foi inferior ao intervalo estipulado. O mecanismo então aguarda o intervalo estipulado ser atingido, garantindo o tempo de execução de 50 ms, e inicia novo *checkpoint*. Com o estado confirmado no *backup*, o *buffer* de rede é liberado no primário. A seta vertical número 3 indica a visão do cliente em relação ao que foi executado na MV durante o período 3. Nota-se que o cliente tem uma visão atrasada em relação ao que foi processado na MV.

A Figura 1 ilustra o intervalo de *checkpointing* sendo respeitado em todos os períodos, porém, nem sempre isso ocorre: quando o tempo de transferência de um *checkpoint* entre primário e *backup* for maior que o intervalo determinado, o intervalo passará a ser o tempo de transferência (para o *checkpoint* em questão) [Petrovic and Schiper 2012]. Esta característica pode ser observada na Figura 2, na qual o intervalo determinado de 50 ms foi superado nos períodos 2, 3 e 6. Por exemplo, o período 2 teve seu tempo de execução aumentado para 200 ms porque a transferência do *checkpoint* relativo ao período 1 foi exatamente de 200 ms. Quanto maior o tempo de execução da MV, maiores as chances do tempo de transferência do próximo *checkpoint* exceder o intervalo estipulado, uma vez que a quantidade de dados a serem replicados aumenta. Isso afeta diretamente a latência da comunicação percebida pelo cliente, uma vez que o *buffer* de rede depende da confirmação de recebimento do *checkpoint* no *backup* para ser liberado.

Conforme descrito em [Petrovic and Schiper 2012], dependendo do momento em que um pacote de saída é gerado na MV, a latência adicional na comunicação pode ser maior ou menor. Por exemplo, se um pacote a ser transferido para um cliente for gerado próximo à pausa para salvar um novo *checkpoint*, a latência imposta na comunicação será o somatório dos tempos de pausa e transferência. Por outro lado, se o pacote for gerado imediatamente após um novo *checkpoint* salvo, ou seja, logo no início de um

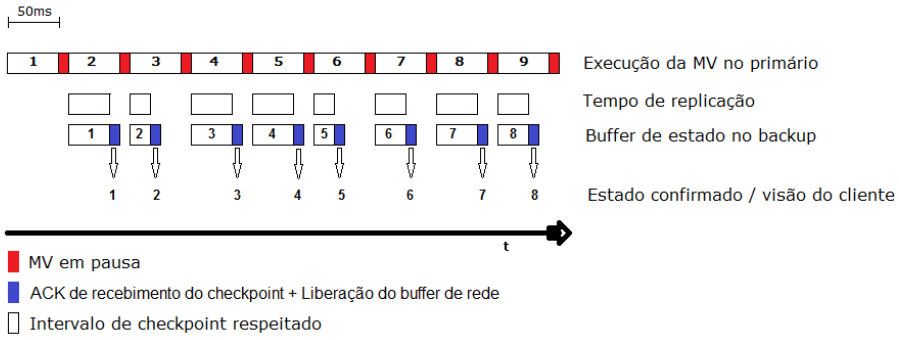


Figura 1. O processo nativo de replicação em Remus.

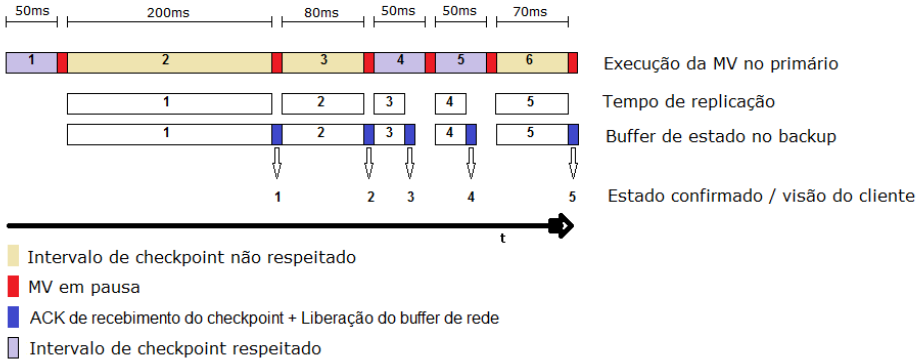


Figura 2. O tempo de execução em modo especulativo varia de acordo com o tempo de replicação sempre que o intervalo definido for ultrapassado.

período de execução da MV, a latência na comunicação será o resultado do somatório do tempo de execução até a próxima pausa, pela pausa e pelo tempo de transferência deste *checkpoint*. Este pode ser caracterizado como o pior caso em relação à latência percebida pelo cliente. Em suma, é difícil predizer a latência que um pacote de saída pode sofrer, uma vez que a liberação do *buffer* de rede é proporcional ao tempo de transferência do *checkpoint* anterior. Isso significa que Remus, em sua configuração padrão, não obedece a frequência estipulada de *checkpointing*, como observado na Figura 2.

3. Replicação com *Checkpointing* Adaptável

Enquanto um tempo de execução maior beneficia o tempo de processamento das aplicações hospedadas na MV, já que intervalos maiores entre *checkpointings* resultam em menos interrupções, por outro lado, as aplicações comunicantes são prejudicadas pela existência do *buffer* de rede. Esse antagonismo dificulta a escolha de um intervalo adequado, principalmente quando a MV reúne uma carga mista de uso de recursos, com uso variável de CPU, memória, E/S e rede.

A proposta deste trabalho alterna entre dois modos de operação, rede e processamento, guiado pelo fluxo de saída da rede (FSR) da MV:

- **Modo rede**: este modo mantém o sistema em alta frequência de *checkpointing*

sempre que for detectado tráfego de saída de rede na interface da MV, visando a diminuição da latência na comunicação com o cliente.
- **Modo processamento**: mantém o sistema em baixa frequência de *checkpointing* quando não houver FSR na interface da MV, aumentando o tempo de execução da MV (TEMV) para priorizar seu tempo de processamento.

O fluxo de saída de rede (FSR) da MV, quantificado em bytes, é medido no próprio Remus.

A solução proposta não introduz otimizações de baixo nível (como compressão de *checkpoint*, *cache* LRU e mapeamento das áreas de leitura e escrita em memória no hipervisor), mas utiliza os mecanismos já adotados por Remus (descritos em [Cully et al. 2008, Rajagopalan et al. 2012]). O foco está no ajuste dinâmico da frequência de *checkpointing* em função da carga de trabalho exercida sobre a MV.

No nosso mecanismo (denominado Remus Rede – RR), o sistema opera normalmente no modo processamento. Nesse modo, após cada *checkpointing*, contabiliza-se o FSR; se houver tráfego, o sistema alterna para o modo rede, aumentando a frequência de *checkpointing*. Nesse modo, a verificação repetida de bytes na interface da MV incorre uma sobrecarga para o mecanismo, aumentando o TEMV, e consequentemente atrasando o salvamento de um novo *checkpoint*. Logo, o sistema permanece no modo rede por 30 *checkpoints* consecutivos, sem contabilizar FSR, e após volta ao modo processamento, retomando a verificação de FSR. O número de *checkpoints* consecutivos foi estabelecido empiricamente (tipicamente a permanência em modo rede é inferior a 1s).

Inicialmente, o modo processamento utiliza um intervalo entre *checkpoints* de 100 ms, que em [da Silva et al. 2014] se mostrou adequado para aplicações que dependem de processamento. Este limiar pode ser aumentado, beneficiando o tempo de processamento, porém, quanto maior o TEMV, maior a quantidade de dados e consequentemente o tempo para salvar um *checkpoint* localmente. Essa sobrecarga adicional, antes da transmissão, pode levar o *backup* a detectar erroneamente uma falta de parada no primário, fazendo com que sua réplica da MV seja ativada de forma indevida. Quando executando no modo rede, não há um valor mínimo, ou seja, um novo *checkpoint* é salvo imediatamente após o anterior estar confirmado no *backup*. Ressalta-se que, para os limiares aqui utilizados, quando o tempo de duração de um *checkpoint* (TDC) for maior que 100 ms, o mecanismo perde sua capacidade de atuação e passa a operar no modo padrão do Remus, ou seja, o intervalo para a frequência de *checkpointing* passa a ser regido pelo TDC.

4. Análise Experimental

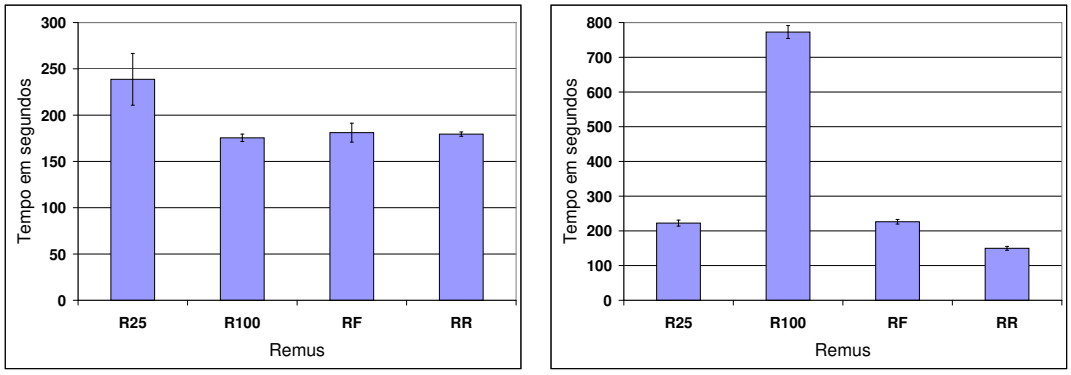
Para verificar a eficácia da proposta, as linhas de base para comparação foram definidas pela execução da versão original de Remus com intervalos fixos de 25 ms e 100 ms (respectivamente chamadas de R25 e R100). Ainda, uma terceira abordagem, guiada somente pelo tempo de processamento observado na MV a cada *checkpointing*, sem considerar o FSR, foi implementada. Esta abordagem – Remus Flutuante (RF) – alterna entre dois intervalos de *checkpointing* (25 e 100 ms). A alternância é definida pelo TDC anterior: se inferior a 25 ms, o intervalo deve se manter em 25 ms; se igual ou superior a 25 ms, deve ser alternado para 100 ms. O uso destes limiares foi baseado em testes de desempenho efetuados em [Cully et al. 2008] e [da Silva et al. 2014]. A verificação do TDC ocorre ao final de cada *checkpointing*, e diferentemente de RR, que é baseado no FSR, é o fator responsável por aumentar ou diminuir a frequência da replicação.

Os experimentos compreendem aplicações que utilizam exclusivamente rede (transferência de arquivos), processamento (compilação e *benchmark* DaCapo Tomcat) e mistas (*benchmarks* NAS e RUBiS). Em [da Silva et al. 2014] o estudo indicou que operações de E/S não interferem significativamente na definição do intervalo de *checkpointing*. Exceto para o *benchmark* RUBiS, no qual o número de operações efetuadas pelo cliente é utilizado como métrica de comparação, para os outros cenários o menor tempo de execução é o indicativo de melhor desempenho. Paralelamente, identificou-se a sobrecarga de processamento imposta ao hospedeiro primário para as 4 versões avaliadas.

O ambiente de testes foi composto por 5 computadores com processador AMD Phenom II X4 (4 núcleos) de 2,8 GHz, RAM de 4 GB, disco 500 GB SATA, executando o sistema operacional Ubuntu 12.10 (*kernel* 3.5.0-17-generic). Um equipamento foi exclusivamente utilizado como origem das requisições para os testes de transferência e RUBiS. Os hospedeiros primário e de *backup* possuem duas interfaces de rede, sendo uma para a conexão com a máquina cliente, através de um comutador de 1 Gbps, e outra exclusiva para a replicação da MV (1 Gbps via cabo *crossover*). Cada hospedeiro foi virtualizado com o MMV Xen 4.2.1, executando o DRBD versão 8.3.11-remus. A MV a ser protegida foi composta por duas vCPUs, 20 GB de disco e 1 GB RAM, executando o sistema operacional OpenSUSE 12.2 (x86_64). Para cada experimento, foram realizadas dez rodadas, e as barras de erros nos gráficos representam o intervalo de confiança de 95%.

4.1. Compilação

Neste experimento foi efetuada a compilação do BIND[2] versão 9.9.4.p2 em uma MV protegida pelo Remus. Este processo envolve uso de CPU, memória e acesso a disco, ou seja, simula uma aplicação dependente do tempo de processamento, já que não envolve comunicação. A Figura 3(a) ilustra o experimento comparativo entre as quatro versões.



(a) Compilação do BIND. (b) Transferência um arquivo de 50 MB.

Figura 3. Comparativo entre as versões para a compilação do BIND e a transferência um arquivo de 50 MB da MV para a máquina cliente.

Aplicações dependentes exclusivamente de processamento apresentam um melhor desempenho quando a frequência de *checkpointing* é menor, uma vez que intervalos maiores diminuem a interferência na execução da MV. As versões RF e RR se adaptaram à natureza do experimento, obtendo resultados semelhantes a R100, que serve como base

[2]`http://www.isc.org/software/bind`

de melhor desempenho para este cenário. Analisando RR e R25, pode-se afirmar que RR teve um tempo médio de compilação 25% menor que o de R25. Para R25, sempre que o TDC for maior que 25 ms, o TEMV será sempre igual a TDC. Para as outras três versões isso não ocorre, ou seja, sempre que TDC for menor que 100 ms, o TEMV será de no mínimo 100 ms. Tomando como exemplo um TDC de 30 ms, em R25 o TEMV será de 30 ms, enquanto que, para RR, RF e R100, será de 100 ms.

4.2. Transferência de um arquivo

Para avaliar o desempenho das quatro versões de Remus em relação à latência na comunicação e seu impacto nas aplicações comunicantes, foi efetuada a transferência de um arquivo de 50 MB da MV para a máquina cliente. O tempo de transferência deste arquivo é de aproximadamente 4,5 s quando Remus não está em execução. O gráfico da Figura 3(b) apresenta os resultados obtidos. Um intervalo de 100 ms a cada *checkpoint* é desejável para aplicações que necessitam de tempo de processamento, mas o mesmo não ocorre com as aplicações comunicantes, ou seja, a versão R100 torna-se impraticável neste cenário. Como esperado, os resultados de R25 e RF foram semelhantes (já que ambas têm como intervalo padrão 25 ms). RR mostrou-se a melhor opção, uma vez que não espera um tempo mínimo para salvar um novo *checkpoint*. O *buffer* de rede é liberado e um novo e um novo *checkpoint* é salvo imediatamente após o anterior ser confirmado pelo *backup* (o TDC passa a reger a frequência da replicação), aumentando a frequência e diminuindo a latência na comunicação. O tempo de transferência de RR foi aproximadamente 33% menor que o de R25 e RF.

4.3. DaCapo Tomcat

O *benchmark* DaCapo Tomcat[3] simula a execução de um conjunto de consultas a um servidor Tomcat[4]. Este teste não envolve uma conexão entre clientes e servidores, sendo totalmente dependente de tempo de processamento da MV. A Figura 4(a) ilustra os resultados obtidos para as quatro versões de Remus. Considerando o intervalo de confiança, RR, cuja frequência padrão de *checkpointing* é 100 ms quando não há tráfego de rede na MV, obteve um melhor desempenho somente em relação a R25, que utiliza uma frequência padrão de 25 ms. R25, por sua vez, manteve um desempenho semelhante a R100 e RF.

Em resumo, todas a versões apresentaram um desempenho semelhante devido à carga de processamento constante imposta pelo *benchmark*: o TDC superou 100 ms em praticamente todos *checkpoints*. Essa análise pode ser comprovada na Figura 6(b): o processamento imposto ao hospedeiro primário foi igual para todas as versões de Remus avaliadas. A captura dos bytes na interface da MV, a cada *checkpoint*, elevou ligeiramente o TDC na versão RR, o que indica sua pequena vantagem em relação a R25.

4.4. NAS

NAS pode ser definido como um conjunto de aplicações destinadas a avaliar o desempenho em computação paralela [Bailey et al. 1991]. Dentre os *benchmarks* disponíveis na versão NPB3.3-MPI, escolheu-se o MG (*multi-grid*), por possuir um comportamento misto em relação ao uso de recursos (intervalos definidos de processamento e

[3] `http://www.dacapobench.org/`

[4] `http://tomcat.apache.org/`

comunicação). Para este experimento, além da MV residente no hospedeiro primário, outros três hospedeiros continham uma MV configurada com o *benchmark* NAS, totalizando quatro MVs para o teste de computação paralela. Das quatro, a única MV protegida por Remus era a residente no hospedeiro primário.

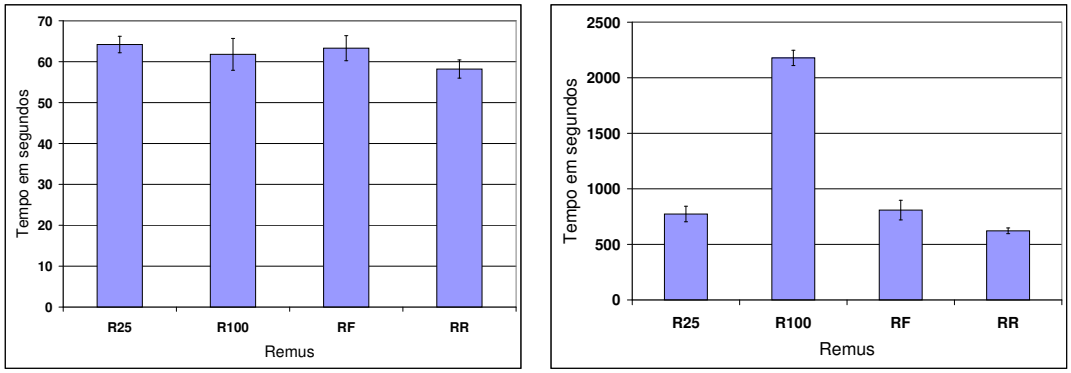
Como observado na Figura 4(b), RR apresentou o melhor resultado em relação ao tempo de execução total do experimento. Dentre as sete aplicações disponíveis no *benchmark* NAS, MG apresenta o maior consumo de memória, com processamento médio de 59,6%, e o terceiro menor tráfego de rede entre os nós (MVs) hospedados [Subhlok et al. 2002]. Entretanto, o resultado obtido indica que a latência na comunicação é o fator preponderante. RF e R25 apresentaram resultados semelhantes: em aplicações que envolvem transferência de rede, é preferível que a frequência de *checkpointing* seja regida pelo TDC, como faz RR, e não por intervalos mínimos ou fixos, como é o caso de RF, R25 e R100. Intervalos predeterminados elevam a latência da comunicação sempre que o TDC for menor que o intervalo definido, pois atrasam a liberação do *buffer* de rede.

4.5. RUBiS

Assim como NAS, este experimento visa avaliar as versões do Remus quando a aplicação em execução na MV requer características mistas de recursos, ou seja, processamento e rede, mas em proporções distintas. Para execução do RUBiS[5], o cenário foi dividido em i) *Servidor*: simula as funcionalidades básicas de um *site* de leilão como o Ebay, como navegação, busca e negociações de compra e venda; e ii) *Cliente*: um emulador de clientes para servidor, sendo que cada cliente gera um sequência de interações com o *site*. O servidor RUBiS está em execução na MV residente no hospedeiro primário, enquanto o emulador na máquina cliente. A métrica escolhida para o analisar o comportamento da aplicação foi o número de operações atendidas pelo *site*. Cada operação compreende a execução de pesquisas (por categoria, região) e a visualização dos itens encontrados. No total, foi simulada a interação de 300 clientes com o *site* de leilão.

A Figura 5 mostra que a latência da comunicação, assim como observado na transferência do arquivo e na execução do *benchmark* NAS-MG, é o fator responsável pelo desempenho final da aplicação. R100 conseguiu atender a apenas 1/3 das operações de

[5] `http://rubis.ow2.org/`



(a) *Benchmark* DaCapo Tomcat.

(b) *Benchmark* NAS-MG.

Figura 4. Tempo de execução dos *benchmarks* DaCapo Tomcat e NAS-MG.

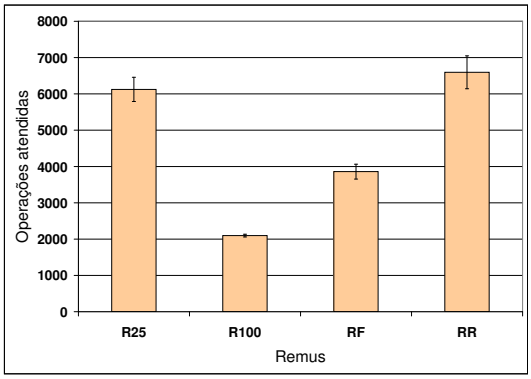


Figura 5. Resultados do *benchmark* RUBiS para as quatro versões do Remus em relação ao número de operações realizadas pelos clientes.

R25 e RR, uma vez que seu intervalo para salvar um novo *checkpoint* é sempre de, no mínimo, 100 ms, o que aumenta a latência percebida na comunicação. Por sua vez, RF alcança um desempenho intermediário: em média, o número de operações de RF é 41% inferior ao de RR, o melhor desempenho, e 84% superior ao de R100, o pior. Como R25 e RR estão dentro do mesmo intervalo quando analisamos o desvio padrão, infere-se que o TDC médio da versão RR ficou em torno de 25 ms. Assim como na transferência e no NAS-MG, é preferível que a frequência de *checkpointing* seja regida pelo TDC.

4.6. Utilização de CPU no Hospedeiro Primário

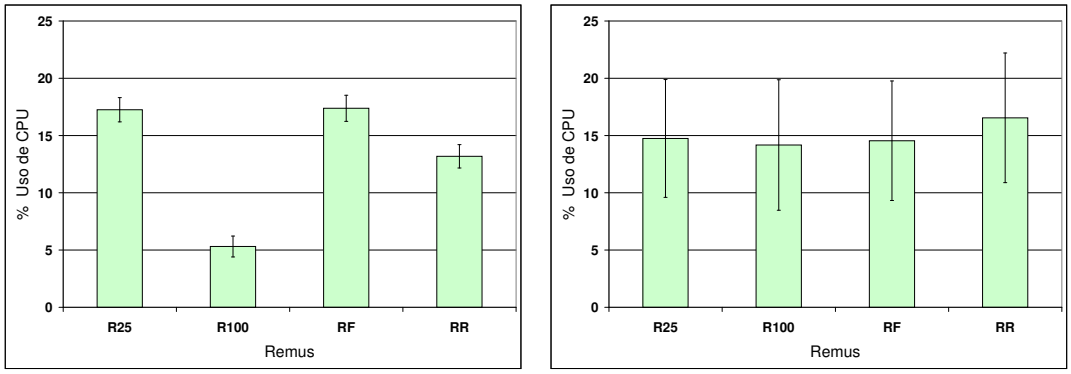
Para verificar a aplicabilidade da solução proposta, é necessário quantificar a sobrecarga imposta pelas abordagens implementadas. Assim, observou-se a utilização média de processador do hospedeiro primário, uma vez que é o hospedeiro responsável por salvar e transferir os *checkpoints* para o *backup*. Dois experimentos foram realizados: o primeiro com a MV ociosa, conforme ilustra a Figura 6(a), e o segundo com o Dacapo Tomcat em execução, conforme ilustra a Figura 6(b). A utilização de CPU foi capturada com intervalos de 1 segundo com a ferramenta mpstat[6].

Em nenhum dos casos houve uma sobrecarga perceptível no processamento. Verifica-se que o processamento aumenta conforme aumenta a frequência de *checkpointing* (Figura 6(a)). A taxa de utilização não chegou a 20% em nenhuma das versões. R25 e RF apresentaram um percentual maior e idêntico devido ao seu estado padrão, que utiliza um intervalo de 25 ms. Entre R100 e RR (com 100 ms como intervalo padrão), existe uma diferença em relação ao uso de CPU, motivada pela coleta de métricas realizada por RR para identificar o tráfego de saída da MV. A diferença do percentual de processamento entre as versões desaparece quando a MV sofre uma carga de uso de processamento maior e constante. Isto é, quando TDC ultrapassa **constantemente** os 100 ms (limiar superior para as quatro versões aqui avaliadas), **todas as versões passam a ter o mesmo TDC**, e consequentemente a mesma frequência de *checkpointing*. Devido a este motivo, a média de uso de processamento foi semelhante para todas as versões avaliadas (Figura 6(b)).

4.7. Discussão

O modo de adaptação RR obteve um desempenho superior ou semelhante ao melhor caso em todos os experimentos, adaptando-se às características da aplicação que está

[6] `http://sebastien.godard.pagesperso-orange.fr/`



(a) MV ociosa. (b) *Benchmark* DaCapo em execução.

Figura 6. Análise do uso de CPU imposta ao hospedeiro primário.

em execução na MV. Ainda, ao contrário de R25, R100 e RF, RR tem a capacidade de aumentar, ao máximo possível, a frequência de *checkpointing*, salvando um novo *checkpoint* imediatamente após o anterior estar devidamente salvo e confirmado no hospedeiro *backup*. Em relação ao processamento, como RR adota um intervalo de 100 ms quando não há comunicação envolvida, seu desempenho mostra-se semelhante a R100 e superior a R25 para as aplicações que necessitam de maior tempo de execução da MV. A justificativa é dada pela variação do TDC: por exemplo, um TDC de 26 ms fará com que o TEMV em R25 seja exatamente de 26 ms. Já em RR, o TEMV será de 100 ms. Ademais, sempre que o TDC superar o intervalo fixo estipulado, Remus original (R25 e R100) adota TDC como intervalo padrão, e passa a indiretamente priorizar o processamento em detrimento da comunicação. Quando a aplicação é mista e dependente tanto de rede quanto de processamento, como é o caso de RUBiS e NAS, aguardar seguidamente 100 ms para salvar um novo *checkpoint* torna-se prejudicial. Nesses casos, o aumento da frequência de *checkpointing* resulta em um melhor desempenho.

5. Trabalhos Relacionados

A adaptação da frequência de *checkpointing* já foi explorada em ambientes não virtualizados. Por exemplo, [Ziv and Bruck 1997] buscam minimizar a sobrecarga de *checkpointing* de um programa, aumentando a frequência quando o estado a ser salvo é pequeno e reduzindo-a quando o custo de *checkpointing* aumenta. [Zhang and Chakrabarty 2003] adaptam a frequência de *checkpointing* em sistemas embarcados de tempo real brando, minimizando o consumo de energia enquanto tolera um número fixo de faltas para uma dada tarefa. [Chtepen et al. 2009] propõem o uso de *checkpointing* adaptativo em grades computacionais, ajustando o intervalo entre *checkpoints* de acordo com o tempo estimado de execução do *job* e da frequência de falhas de recursos; o objetivo é balancear o *overhead* de *checkpointing* e o custo de reiniciar *jobs* interrompidos por falhas. Em comum, esses trabalhos têm como foco controlar o tempo de execução limitando o tempo de recuperação após uma falha, sem se preocupar diretamente com a responsividade.

No contexto de MVs, raramente o desempenho das métricas de rede é um fator determinante. Processos auxiliares, como compressão, visam diminuir o tamanho de um *checkpoint* e por consequência diminuir a sobrecarga imposta na MV. Porém, na maioria dos casos, a frequência de *checkpointing*, que influencia diretamente na latência da

comunicação, ainda é regida pelo tempo de duração do *checkpoint* para o *backup*. Os trabalhos relacionados podem ser organizados em migração e replicação de MVs.

Migração de máquinas virtuais: CloudNet [Wood et al. 2011] propõe a migração de MVs sobre uma WAN (*wide-area network*) visando o balanceamento de carga. A tomada de decisão para tal migração depende da latência na comunicação observada pelo cliente final e/ou limites de capacidade. O processo final de migração da MV pode levar o cliente a perceber um alto período de indisponibilidade, ferindo a linearizabilidade. O principal objetivo da migração em tempo real é minimizar o tempo de interrupção de serviço quando uma MV precisa ser movida para outro hospedeiro físico, seja por motivos de manutenção no hospedeiro, balanceamento de carga ou até por questão de economia de energia. O trabalho realizado por [Hu et al. 2013] aborda o desempenho de diferentes MMVs (KVM, XenServer, VMware e Hyper-V) para a migração em tempo real. Dentre os resultados observados, o tempo total de migração e de indisponibilidade da MV e o tráfego de rede entre os hospedeiros são considerados. O mecanismo apresentado em nosso trabalho pouco contribuiria com soluções de migração em tempo real, já que a latência na comunicação percebida pelo cliente ocorrerá somente durante o período de indisponibilidade da MV, quando a migração entra na fase *stop-and-copy*.

Replicação de máquinas virtuais: [Koppol et al. 2011] fez um estudo sobre Remus avaliando a qualidade de serviços que utilizam pacotes de voz. O atraso causado pelo uso do *buffer* de rede, somado ao fato de que a liberação ocorre de uma só vez ao final de um *checkpoint*, levou a uma explosão de tráfego, degradando a qualidade do áudio.

Por sua vez, SecondSite [Rajagopalan et al. 2012] utiliza as mesmas técnicas de replicação do Remus diferenciando o ambiente alvo: propõe a replicação completa de um *site* entre *datacenters* distribuídos. O objetivo é migrar as MVs em caso de catástrofes naturais, faltas de *hardware* ou energia para outro *datacenter*. Nesta solução, a manutenção da conectividade das MVs é obtida através de configurações de roteamento previamente realizadas. Assim como no Remus, o tamanho do *checkpoint* a ser transferido para o *backup* é o fator determinante para o tempo de execução da MV. Técnicas de compressão para diminuir o tamanho de um *checkpoint* foram utilizadas, porém dependendo da aplicação, nem sempre são suficientes. SecondSite pode se beneficiar da proposta deste trabalho, uma vez que o enlace WAN existente entre os hospedeiros primário e *backup* pode aumentar a latência na comunicação percebida pelo cliente da MV.

[Tamura et al. 2008] implementou uma solução para replicação de MVs que visa tolerância a faltas de parada utilizando o modelo primário-*backup*. O mecanismo de replicação não utiliza *buffer* de rede, replicando o estado da MV apenas na ocorrência de eventos de leitura ou escrita em disco. Um mecanismo para tolerar faltas de parada no MMV KVM foi implementado em [Cui et al. 2009], realizando uma migração contínua entre dois hospedeiros. Diferentemente de Remus, a MV utiliza um disco compartilhado e não há um *buffer* de rede implementado no hospedeiro, diminuindo a latência na comunicação percebida pelo cliente. Essas abordagens diminuem a sobrecarga imposta ao tempo de processamento na MV, porém não garantem a linearizabilidade do sistema.

Uma solução para diminuir a sobrecarga do processo de *checkpointing* é apresentada em [Zhu et al. 2010]. Nesse estudo, o tempo para salvar um *checkpoint* é guiado pelo comportamento de acesso a memória no MMV. Embora o procedimento possa diminuir

o tempo de transferência de um *checkpoint*, a alteração da frequência não é abordada. Em outra solução, porém baseada no MMV VMware, a replicação do estado da MV não é baseada em *checkpoints*, e sim por *logs*, que são enviados ao *backup* imediatamente após uma instrução ser executada no primário [Scales et al. 2010]. Semelhante ao Remus, o sistema utiliza um *buffer* de rede, inovando através da busca automática por um servidor com recursos disponíveis para atuar como *backup* (para tal, a solução utiliza armazenamento em disco compartilhado). Nesse cenário, o tempo de processamento no primário varia de acordo com a capacidade de processamento do hospedeiro *backup* selecionado. Além de ser uma solução fechada, aplica-se a servidores e não a máquinas comuns que utilizam hardware de prateleira, e limita a utilização de apenas um processador por MV.

[Gerofi and Ishikawa 2011] mescla duas abordagens: adaptação do intervalo e diminuição do volume transferido. Em Remus, o volume (memória e disco) é otimizado com técnicas de compressão e identificação de páginas recentemente alteradas. Enquanto esta proposta abordou 3 variáveis em sua formulação (memória, largura de banda e intervalo) o modelo proposto no presente artigo confiou na otimização já existente em Remus, simplificando o processo de adaptação. Ainda, o limite superior para o intervalo de *checkpointing* foi definido pelos autores em 2 s, ou seja, 20 vezes o limite adotado pela presente proposta (os limiares adotados visam a manutenção da confiabilidade da MV protegida).

6. Conclusão

Com o advento da virtualização, técnicas economicamente acessíveis para provimento de tolerância a faltas tornaram-se possíveis. Nesse contexto, destaca-se Remus, um mecanismo de replicação de MVs Xen que proporciona tolerância a faltas de parada. Embora promissora, a aplicação de Remus em produção enfrenta algumas limitações: a frequência de *checkpointing* influencia no desempenho das aplicações hospedadas. Este trabalho introduziu uma proposta para *checkpointing* adaptativo em Remus, que busca a identificação dinâmica do melhor intervalo entre *checkpoints* com base no fluxo de saída de rede da MV. Os resultados experimentais mostraram que a abordagem guiada pelo fluxo de rede – quando não há um tempo de execução mínimo para a MV e um novo *checkpoint* é salvo imediatamente após o anterior estar confirmado no hospedeiro *backup* – oferece uma melhora da ordem de 30% no desempenho de aplicações que usam a rede, sem prejudicar o desempenho de aplicações que necessitam de tempo de processamento. Ademais, a sobrecarga do mecanismo de adaptação mostrou-se negligenciável na avaliação. Na continuação deste trabalho pretende-se investigar a aplicabilidade da solução quando primário e *backup* estão ligados em ambientes WAN, bem como a possibilidade de replicação de múltiplas MVs hospedadas no mesmo primário.

Referências

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, D., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The NAS Parallel Benchmarks. *Int. Journal of Supercomputer Applications*.

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. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The primary-backup approach. *Distributed Systems (2Nd Ed.)*, pages 199–216.

Chtepen, M., Dhoedt, B., De Turck, F., Demeester, P., Claeys, F., and Vanrolleghem, P. (2009). Adaptive checkpointing in dynamic grids for uncertain job durations. In *Information Technology Interfaces, 2009. ITI '09. Proceedings of the ITI 2009 31st International Conference on*, pages 585–590.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *Proc. of the 2Nd Conference on Symposium on Networked Systems Design & Implementation*, NSDI'05, pages 273–286.

Cui, W., Ma, D., Wo, T., and Li, Q. (2009). Enhancing reliability for virtual machines via continual migration. In *Proc. of the 15th International Conference on Parallel and Distributed Systems*, ICPADS '09, pages 937–942, Washington, DC, USA. IEEE Computer Society.

Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. (2008). Remus: High availability via asynchronous virtual machine replication. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174.

da Silva, M., Koslovski, G., and Obelheiro, R. (2014). Uma análise da sobrecarga imposta pelo mecanismo de replicação de máquinas virtuais Remus. In *XV Workshop de Testes e Tolerância a Falhas (WTF)*, Florianópolis, Brasil.

Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408.

Gerofi, B. and Ishikawa, Y. (2011). Workload adaptive checkpoint scheduling of virtual machine replication. In *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*, pages 204–213.

Guerraoui, R. and Schiper, A. (1997). Software-based replication for fault tolerance. *Computer*, 30(4):68–74.

Hu, W., Hicks, A., Zhang, L., Dow, E. M., Soni, V., Jiang, H., Bull, R., and Matthews, J. N. (2013). A quantitative study of virtual machine live migration. In *Proc. of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 11:1–11:10.

Koppol, P., Namjoshi, K., Stathopoulos, T., and Wilfong, G. (2011). The inherent difficulty of timely primary-backup replication. In *Proc. of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 349–350. ACM.

Petrovic, D. and Schiper, A. (2012). Implementing virtual machine replication: A case study using Xen and KVM. In *Proc. of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, AINA '12, pages 73–80.

Rajagopalan, S., Cully, B., O'Connor, R., and Warfield, A. (2012). SecondSite: Disaster tolerance as a service. *SIGPLAN Not.*, 47(7):97–108.

Reisner, P. and Ellenberg (2005). DRBD v8 – replicated storage with shared disk semantics. In *Proc. of the 12th International Linux System Technology Conference*.

Scales, D. J., Nelson, M., and Venkitachalam, G. (2010). The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev.*, 44(4):30–39.

Subhlok, J., Venkataramaiah, S., and Singh, A. (2002). Characterizing NAS benchmark performance on shared heterogeneous networks. In *Proc. of the 16th Int. Parallel and Distributed Processing Symposium*, IPDPS '02, pages 91–, Washington, USA. IEEE Computer Society.

Tamura, Y., Sato, K., Kihara, S., and Moriai, S. (2008). Kemari: VM Synchronization for Fault Tolerance. In *USENIX '08 Poster Session*.

Wood, T., Ramakrishnan, K. K., Shenoy, P., and van der Merwe, J. (2011). CloudNet: Dynamic pooling of cloud resources by live wan migration of virtual machines. In *Proc. of the 7th ACM SIGPLAN/SIGOPS Int. Conference on Virtual Execution Environments*, VEE '11.

Zhang, Y. and Chakrabarty, K. (2003). Energy-aware adaptive checkpointing in embedded real-time systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 918–923.

Zhu, J., Dong, W., Jiang, Z., Shi, X., Xiao, Z., and Li, X. (2010). Improving the performance of hypervisor-based fault tolerance. *Int. Parallel and Distributed Processing Symposium*, 0:1–10.

Ziv, A. and Bruck, J. (1997). An on-line algorithm for checkpoint placement. *Computers, IEEE Transactions on*, 46(9):976–985.