INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

CRM-HLL-VM: Checkpoint, Restore, Migração em Máquinas Virtuais Java

Tiago Garrochinho

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Alberto Manuel Ramos da Cunha
Orientador:	Professor Luis Manuel Antunes Veiga
Vogais:	Professor João Manuel Santos Lourenço

Outubro de 2010

Abstract

Object-oriented programming languages are in current days, the dominant paradigm of application development (mostly Java and .NET languages). Recently, increasingly more Java applications have long (or very long) execution times and manipulate large amounts of data/information, gaining relevance in fields related with e-Science (with Grid and Cloud computing). Significant examples include chemistry, computational biology and bioinformatics, with many available Java-based APIs (e.g., Neobio).

Often, when the execution of one of those applications is terminated abruptly due to failure (regardless of it being caused by hardware or software fault, lack of available resources,...), all of its work already carried out is simply lost and, when the application is later re-executed, it has to restart its work from scratch, wasting resources and time, and being prone to another failure, to delay its completion with no deadline guarantees.

A solution to solve these problems, is through mechanisms of checkpoint and migration. This makes applications more robust and flexible by being able to move to other nodes, without intervention form the programmer. This master thesis provides a solution to Java applications with long execution times, by incorporating such mechanisms in a Java VM (JikesRVM).

The proposed solution was evaluated with very encouraging and positive results. The imposed latency is small relatively to the applications' (long) execution times.

Keywords

Virtual machines, checkpointing, migration, Java VM, e-Science

Resumo

As linguagens de programação orientadas a objectos são nos dias correntes o paradigma de desenvolvimento dominante (na sua maior parte linguagens Java e universo .NET). Hoje em dia, existem cada vez mais aplicações que têm execuções demoradas e possivelmente manipulam muitos dados/informação, ganhando relevância em vários campos relacionados com e-Science (sobretudo no contexto de computação Grid e Cloud). Exemplos significantes incluem química, biologia computacional e bioinformática, com muitas APIs disponíveis baseadas em Java (e.g., Neobio).

Frequentemente, quando a execução duma dessas aplicações termina abruptamente devido a uma falha (independentemente de ter sido causada por uma falta de software ou hardware, indisponibilidade de recursos, entre outras razões), todo o seu trabalho já efectuado é simplesmente perdido e, quando essa aplicação se executar novamente mais tarde, esta tem de reiniciar o seu trabalho a partir do zero, desperdiçando recursos e tempo, estando mais uma vez sujeita a outras falhas, atrasando a sua conclusão sem qualquer garantia de terminação.

Uma solução para resolver estes problemas, é através de mecanismos de checkpoint e migração. Com estes mecanismos, uma aplicação torna-se mais robusta, pois fica tolerante a falhas e ganha flexibilidade, por ser capaz de mover-se para outros nós, sem intervenção do programador. Esta tese de mestrado fornece uma solução para aplicações Java com tempos de execução longos, incorporando os mecanismos indicados na máquina virtual Java (JikesRVM).

A solução proposta foi avaliada com resultados bastante encorajadores e positivos. A latência imposta é pequena relativamente aos tempos de execução (longos) das aplicações.

Palavras Chave

Máquinas virtuais, checkpointing, migração, máquina virtual Java, e-Science

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

Contents

As linguagens de programação orientadas a objectos são nos dias correntes o paradigma de desenvolvimento dominante (na sua maior parte linguagens Java e universo .NET). Estas linguagens prevalecem em aplicações desktop, aplicações de desenvolvimento em si (e.g., Eclipse), aplicações web, componentes, aplicações servidor, e até em jogos, principalmente em cenários móveis.

Hoje em dia, existem cada vez mais aplicações que têm execuções demoradas e possivelmente manipulam muitos dados/informação. Isto está a tornar-se relevante em vários campos relacionados com e-Science (sobretudo no contexto de computação Grid e Cloud) onde Java se está a tornar uma das linguagens de desenvolvimento mais dominantes, muito embora usada por investigadores (programadores), que muitas vezes não são engenheiros/cientistas na área da computação. Exemplos significantes incluem química, biologia computacional e bioinformática [17, 19, 26], com muitas APIs disponíveis baseadas em Java (e.g., Neobio).[1]

Frequentemente, quando a execução de uma dessas aplicações termina abruptamente devido a uma falha (independentemente de ter sido causada por uma falta de software ou hardware, indisponibilidade de recursos, entre outras razões), todo o seu trabalho já efectuado é simplesmente perdido e, quando essa aplicação se executar novamente mais tarde, com os mesmos parâmetros (e.g., aplicação de processamento de dados), esta tem de reiniciar o seu trabalho a partir do zero, desperdiçando recursos e tempo, estando mais uma vez sujeita a outras falhas, atrasando a sua conclusão sem qualquer garantia de terminação.

Uma solução possível para resolver estes problemas, é através de mecanismos de checkpoint e migração. Com estes mecanismos, uma aplicação torna-se mais robusta, pois fica tolerante a falhas e ganha flexibilidade, por ser capaz de mover-se para outros nós, sem intervenção do programador.

Mecanismos tradicionais de checkpoint e migração são suportados a diferentes níveis:

1. **Nível de processo**: quer desencadeado pela aplicação, com código próprio ou através de bibliotecas específicas, quer como um mecanismo oferecido pelo sistema operativo modificado ou estendido [29];

2. **Máquina virtual de sistema**: VM sistema (e.g., Robert Bradford et al. [9]).

Estas abordagens são insuficientes pelas seguintes limitações: ou i) obrigam a que seja guardada / transportada informação que não é relativa apenas à aplicação em si (e.g., informação relativa ao sistema operativo onde se executa), ou ii) limitam a portabilidade da mesma. Portanto, como a maioria das linguagens de programação orientadas a objectos executam as suas aplicações sobre máquinas virtuais para linguagens de alto nível (a partir de agora denominada por VM OO, mas também é conhecida como HLL VM,[2] e.g., Java VM e .NET CLR), propõe-se uma abordagem dos mecanismos de checkpoint e migração a esse nível.

No domínio das soluções de checkpoint e migração ao nível das VM OO, já existe alguma investigação, contudo, as soluções existentes estão inseridas principalmente no contexto de agentes

[1] http://www.bioinformatics.org/neobio/
[2] High-Level Language Virtual Machine.

móveis, e por isso são limitadas, isto é, apenas retratam uma thread sobre um ambiente muito reduzido e controlado (e.g., MobileJikesRVM [11]). De modo idêntico, existem também soluções que: ou têm problemas de eficiência (e.g., JavaGo [39], JavaGoX [38], Brakes [45], ITS [7], que adicionam em média penalizações de desempenho superiores a 300% na execução das aplicações); ou passam responsabilidades ao programador (e.g., WASP [15]), de forma a resolver problemas da própria solução (o que limita a transparência); ou ainda, são soluções cuja completude não é bem retratada, sendo que o problema principal está especificamente relacionado com o estado externo de uma aplicação (e.g., ficheiros e sockets).

1.1 Objectivos

Este documento fornece uma solução para aplicações Java com execuções longas, incorporando mecanismos de checkpoint e migração numa máquina virtual Java (JikesRVM [3]). Os objectivos principais focam-se nos problemas da transparência e completude. Porém, a solução proposta tem em conta o seguinte conjunto de propriedades:

- **Transparência**: Os mecanismos não devem ser construídos de forma a atribuir responsabilidades ao programador. É fornecido um controlador externo (aplicação linha de comandos), que permite qualquer utilizador (para além do próprio programador), usufruir dos mecanismos desenvolvidos, sem que seja necessário alterar qualquer linha de código da aplicação. As aplicações não devem aperceber-se da mudança de ambiente, nem que foram recuperadas usando checkpoint ou transportadas para outro nó utilizando migração.

- **Flexibilidade**: Apesar de não haver imposição de atribuição de responsabilidades ao programador, propõe-se uma API[3] que permita o programador controlar estes mecanismos a partir da sua aplicação.

- **Consistência**: O estado de uma aplicação mantém-se inalterável, livre de inconsistências, mesmo após uma operação de recuperação por checkpoint ou migração. Em termos funcionais a aplicação prossegue a sua execução como se nenhum dos mecanismos tivesse ocorrido (não inclui questões temporais).

- **Completude**: Os mecanismos têm de retratar todo o estado de uma aplicação: código, dados (e.g., heap), estado de execução (e.g., stack, threads) e ligações externas (ficheiros e sockets), incluindo também estado de execução nativo (JNI),[4] e estado de sincronização. De notar que não se pretende guardar/transportar a máquina virtual inteira. É pretendido apenas que se tenha em conta o mínimo estado relativo à aplicação em si, de forma a que esse mínimo seja o suficiente para reconstruir o estado da aplicação na máquina virtual destino.

[3] Application Programming Interface.
[4] Java Native Interface.

- **Portabilidade**: Em parte assegurada por abordagem baseada em VM OO, mas é necessário ter disponível a máquina virtual modificada/estendida em todos os nós. É também desejável que o mesmo código da máquina virtual compilada num dado sistema operativo, e numa dada arquitectura, seja capaz de utilizar checkpoints criados pela mesma máquina virtual compilada noutras plataformas.

- **Eficiência**: O custo adicional de desempenho durante a execução da aplicação deve ser mínimo ou nenhum. O custo de desempenho durante a execução dos mecanismos deve ser proporcional aos recursos utilizados pelas aplicações que as vão utilizar.

- **Robustez**: Os mecanismos no mínimo, não devem prejudicar a aplicação, nem ser fonte de novas excepções que não foram antevistas pelo programador (e.g., quando não é possível efectuar checkpoint ou migração, a aplicação deve continuar normalmente).

1.2 Contribuições

As contribuições oferecidas por este trabalho são as seguintes:

- Um estudo vasto de soluções existentes relacionadas com mecanismos de checkpoint, restore e migração.

- Uma solução de arquitectura para aplicações em execução na máquina virtual JikesRVM, estendida com mecanismos de checkpoint, restore e migração, que assegura as seguintes **funcionalidades distintivas**:

  - Suporta guardar e recuperar o estado de threads anteriormente bloqueadas (e.g., sleep, wait, input read, file read, entre outros).

  - Suporta guardar e recuperar informação de sincronização (bytecodes monitorEnter e monitorExit).

  - Suporta aplicações que façam invocações nativas via JNI.

  - Solução transparente para o programador, sem impor responsabilidades ou alterações nas aplicações.

  - Solução que retrata também as ligações com o estado externo, mesmo quando uma ligação está em uso, isto é, quando existe informação a ser transportada nos canais de comunicação (no entanto, não inclui nos sockets questões relacionadas com mensagens perdidas).

- Realização e avaliação da solução proposta com microbenchmarks e aplicações exemplo.

1.3 Estrutura do documento

Este documento está organizado da seguinte forma: no Capítulo 2 retrataremos o trabalho relacionado, como visão geral das soluções de checkpoint, restore e migração existentes; no Capítulo 3 abordaremos a arquitectura da solução desenvolvida, com foco nas propriedades oferecidas por esta solução; no Capítulo 4 detalharemos os pormenores de implementação para a máquina virtual JikesRVM; no Capítulo 5 mostraremos os resultados qualitativos e quantitativos obtidos na avaliação dos mecanismos desenvolvidos; por fim no Capítulo 6 apresentaremos a conclusão.

No decurso deste documento são realizadas múltiplas menções a termos e/ou conceitos cuja designação comummente aceite é na língua inglesa. Para não criar frequentes descontinuidades na formatação do texto, optámos por não as apresentar em itálico.

1.4 Publicações científicas

Uma versão preliminar deste trabalho, encontra-se parcialmente descrita numa publicação cientifica no MGC 2010, Middleware for Grids, Clouds and e-Science, integrada na conferência Middleware 2010, com o título "CRM-OO-VM: A Checkpoint-enabled Java VM for efficient and Reliable e-Science Applications in Grids".

2

Trabalho Relacionado

Contents

O tema deste trabalho está dividido em três grandes áreas que estão directamente relacionadas com o seu título:

- **Checkpoint/restore**: checkpoint é o mecanismo que permite persistir informação pertencente a um sistema operativo, ou a uma aplicação/processo, ou ainda a uma thread. Restore é o mecanismo que permite efectuar a sua recuperação e recomeçar a execução a partir do ponto em que foi realizado o checkpoint.
- **Migração**: o mecanismo de migração permite transportar durante a execução a informação anteriormente referida entre dois nós/máquinas diferentes (entre nós origem e destino).
- **Log e replay**: logging, no contexto de checkpoint, é um mecanismo que permite guardar acções para serem executadas novamente mais tarde. Essas acções são re-executadas de forma sequencial sobre um estado consistente do sistema (e.g. obtido num checkpoint anterior).

O trabalho relacionado vai ser exposto com a seguinte estrutura: como a grande maioria dos sistemas que suportam migração, necessitam previamente de efectuar um checkpoint, de forma a obter um estado consistente da execução de uma aplicação, apresentamos primeiro de forma conjunta as soluções dos mecanismos de checkpoint/restore e migração por nível de implementação (nível de processo, VM sistema, e VM OO, o assunto principal deste trabalho). No final, abordamos mecanismos de log e replay. Esses mecanismos são relevantes para o trabalho relacionado desta solução porque realizam checkpoint através de técnicas de log e replay e por outro lado, podemos verificar que é frequente existirem sistemas de log e replay que usam checkpoint para melhorar o seu desempenho.

2.1 Mecanismos de checkpoint/restore e migração

As soluções existentes dos mecanismos de checkpoint e migração estão implementadas a diferentes níveis: **nível de processo**, **VM sistema** e **VM OO**, tal como já foi abordado anteriormente no Capítulo 1 (Introdução). Porém, é importante referir que, o nível de implementação influencia directamente o tipo de informação retratada, ou seja, dependendo do nível de implementação, pode-se obter checkpoint e migração de: **sistemas operativos** (isto é, um sistema completo), **aplicações** ou **threads**. No entanto, independentemente do nível de implementação, o estado de execução que os mecanismos têm de suportar pode ser dividido em duas partes: **estado interno** e **externo**. Ao longo do trabalho relacionado, para cada nível de implementação, irá ser detalhado qual é o estado interno e externo correspondente. Relativamente ainda a este ponto, é importante salientar que, o estado interno é mais ou menos bem retratado (pelo menos relativamente ao tipo de informação que a solução se compromete a retratar, seja thread, aplicação ou sistema operativo). Contudo, em relação ao estado externo as soluções já têm alguns problemas que ficam por resolver.

Numa perspectiva geral, soluções de checkpoint detalham melhor como é que o estado de execução é preservado e recuperado. Soluções de migração abordam melhor detalhes relacionados com a

infra-estrutura da solução (e.g. ligação entre os vários nós), e importam-se também com algoritmos que optimizam o transporte de informação entre os diferentes nós, isto porque, o transporte do estado de execução numa rede, diminui significativamente o desempenho dos mecanismos.

De seguida retratamos as soluções existentes por nível de implementação, começando primeiro pelas soluções de checkpoint, e depois pelas soluções de migração (pelas razões anteriormente indicadas). Como a solução proposta é feita ao nível de uma VM OO, soluções relevantes a esse nível irão ser detalhadas.

2.1.1 Nível de processo

O trabalho de Eric Roman [37], dá uma boa perspectiva das soluções de **checkpoint** existentes a este nível de implementação. É importante referir que, este tipo de abordagens exige homogeneidade, na qual dependem da adopção global de uma arquitectura ou sistema operativo específico. As máquinas virtuais (tanto, VM sistema, como VM OO) tornaram-se importantes no contexto de checkpoint e migração por suportarem heterogeneidade, podendo executar a mesma instância da máquina virtual sobre sistemas operativos e arquitecturas físicas diferentes.

Com base numa análise sobre as soluções de checkpoint ao nível de processo, é possível definir o estado de execução que tem de ser retratado a este nível de implementação (inclui como já foi referido, estado interno e externo). O estado interno é definido por sinais pendentes, espaço de endereçamento (que contém grande parte do estado de uma aplicação, sendo este: heap, stack e qualquer região mapeada) e registos de CPU. O estado externo é caracterizado por descritores de ficheiro, o conteúdo dos ficheiros e sockets. Das soluções que conseguem retratar o estado externo, sobressaem apenas as seguintes:

- **Rx** [34] e **Triage** [46], por conseguirem preservar os descritores de ficheiro e o conteúdo dos ficheiros que foram abertos durante a execução. **Libckpt** [33] preserva apenas o conteúdo dos ficheiros que estão abertos. Ambas as soluções podem ser insuficientes, isto porque, um ficheiro que possa ser aberto depois de um restore, pode nunca ter sido referenciado anteriormente.

- **CRAK** [48], por recriar as ligações socket anteriormente abertas.

São poucas as soluções que trabalham o estado externo de uma aplicação, e não existe nenhuma que retrate esse estado na totalidade. Na Tabela 2.1, mostramos um conjunto de soluções em função dos parâmetros do estado de execução indicados anteriormente, para melhor comparação e compreensão do que foi dito.

Algumas das soluções apresentadas na Tabela 2.1, têm algumas propriedades que são importantes mencionar, tais como: **transparência**, **checkpoint para memória** e **checkpoints incrementais**, que vão ser retratadas de seguida nos sistemas que as adoptam.

Libckpt, mencionada anteriormente por retratar algum estado externo, é uma biblioteca para

Nome	Nível implementação	Estado interno, referente à aplicação	Descritores ficheiro	Conteúdo ficheiros	Sockets	Multi thread
libckp [23]	Aplicação (App)	Incompleto (signals)	Sim	Não	Não	Não
libckpt [33]	App	Incompleto (signals)	Sim	Sim	Não	Não
libtckpt [13]	App	Completo	Sim	Não	Não	Não
CRAK [48]	Sistema operativo (SO)	Completo	Sim	Não	Sim	Não
BProc [18]	SO	Completo	Não	Não	Não	Não
Flashback [40]	SO	Completo	Sim	Não	Não	Sim
Rx [34]	SO	Completo	Sim	Sim	Não	Sim
Triage [46]	SO	Completo	Sim	Sim	Não	Sim
Jon Howell [20]	SO	Completo	Sim	Não	Não	Sim

Tabela 2.1: Resumo das soluções de checkpoint/restore ao nível de processo.

o sistema operativo Unix, com o objectivo de tornar os mecanismos de checkpoint mais fáceis de utilizador por qualquer programador. Este objectivo é atingido através da propriedade de **transparência** (Secção 1.1, Objectivos), na qual não obriga quaisquer alterações nas aplicações por parte dos programadores, de forma a usufruir dos mecanismos de checkpoint. Contudo, e embora com muito esforço, este objectivo não foi completamente atingido, isto porque, para libckpt tomar controlo sobre a aplicação (com o objectivo de realizar um checkpoint), precisa primeiro de alterar a rotina de entrada de qualquer programa em C, de `main()` para `ckpt_target()`. Isto faz com que libckpt não seja completamente transparente, mas já foi um passo importante neste tipo de solução.

Flashback, é uma solução para depuração de aplicações, que teve de implementar a sua própria solução de checkpoint ao nível do sistema operativo Linux. Esta solução é um exemplo das soluções de checkpoint que não tem como requisito persistir o estado de execução em dispositivos não voláteis (e.g., disco rígido). Basicamente, os checkpoints são salva-guardados em memória e, em conjunto com mecanismos de log e replay, é possível inspeccionar o estado de execução num determinado ponto de execução. A implementação é muito dependente do sistema operativo Linux e actua da seguinte forma: cada vez que existe um pedido de checkpoint, é invocada a chamada de sistema `fork`, que produz uma cópia do processo inicial. Esse processo cópia é mantido como processo *shadow*, para ficar automaticamente suspenso. No restore, a execução do processo principal é interrompida e é criado um novo processo activo através de um processo *shadow* específico.

Outras soluções, como a libckpt (já referenciada) usufruem também de **checkpoints para memória**, mas com o objectivo de não consumir recursos sobre a aplicação principal. A razão principal pela qual muitos mecanismos de checkpoint sofrem grandes penalizações de desempenho, deve-se ao facto de ficarem à espera que as escritas sejam efectuadas para o disco rígido. Com base nesta motivação, esta solução efectua checkpoint para disco através do checkpoint realizado para memória. Contudo, embora pareça que esta técnica venha a melhorar o desempenho dos mecanismos de checkpoint, os resultados demonstram que, os ganhos não são significativos, isto porque observou-se que qualquer escrita que é feita no disco rígido tem tendência para abrandar a

execução em geral das aplicações.

Ambas as soluções, Libckpt e Flashback, introduzem técnicas para optimizar o tamanho dos checkpoints em disco, através de métodos denominados por **checkpoint incrementais**, mais conhecido por *copy-on-write*. Este técnica faz com que apenas seja copiada a informação que foi modificada entre checkpoints. Desta forma, o tamanho dos checkpoints é diminuído, e o tempo para efectuar um checkpoint fica também mais reduzido, isto porque se copia menos informação.

Até agora, vimos soluções de checkpoint ao nível de processo. De seguida, retratamos soluções de **migração** a esse nível de implementação. Já foi discutido, mas é importante relembrar que, soluções de migração focam-se mais na infra-estrutura e em algoritmos que optimizam o transporte da informação entre nós diferentes.

O trabalho dos autores Dejan Milojicic et al. [29] apresenta um estudo muito vasto de soluções para migração que existem a este nível de implementação. Vão ser abordados os sistemas mais utilizados e reconhecidos (Locus, Condor, Mosix). Os algoritmos para migração de informação entre nós diferentes apresentados nesse trabalho vão ser descritos após esses sistemas, com uma análise detalhada, pois são de levada importância para qualquer mecanismo de migração, pelo facto de existirem algoritmos que se adaptam melhor a soluções com determinados objectivos específicos.

Locus [47], foi desenvolvido para ser um sistema distribuído, tolerante a falhas para o sistema operativo Unix, mas compatível com System V [44] e BSD. Foi desenvolvido um sistema de ficheiros distribuído com suporte para cache de ficheiros, replicação e abstracções das operações para os diferentes sistemas operativos. Os processos podem ser migrados para outros sistemas, salientando que a comunicação entre os processos migrados pode continuar sem interrupções, isto porque, através de uma abstracção criada a partir de "Pipes" Unix, a ligação é mantida.

Condor [25], é um sistema de escalonamento de processos desenvolvido sobre o sistema operativo Unix. Esta solução tenta garantir que as aplicações em execução completam o seu trabalho através de mecanismos de checkpoint. Com estes mecanismos as aplicações podem terminar execução quando na presença de indisponibilidade de recursos e, retomar execução mais tarde quando for possível. Esta solução também suporta migração, com o objectivo de escalonar processos para outro nó disponível. É importante referir que nesta solução, as chamadas de sistema sobre ficheiros são sempre redireccionadas para o nó inicial.

Mosix [4], é um sistema operativo multi-computador (para o sistema operativo Unix), desenhado para ser incorporado num conjunto de nós fracamente ligados, que criem um único ambiente de execução (entre os vários nós). O núcleo desta solução está segmentado em três camadas. A primeira camada fornece as ligações com o sistema de ficheiros e com as estruturas de dados dos processos/aplicações. A última camada disponibiliza o conjunto de chamadas (de sistema) do sistema operativo Unix. A camada intermédia suporta a interligação entre ambas as camadas apresentadas (através de mecanismos de comunicação), e possibilita que uma camada esteja num nó, e outra noutro. Mosix suporta tanto balanceamento de carga, como migração de processos,

sem que seja necessário deixar dependências em nós anteriores.

Ambas as soluções, Locus e Condor, são as únicas conhecidas por serem utilizadas mundialmente, pelo facto de serem uniformes tanto nas interfaces fornecidas, como no ambiente de desenvolvimento. Mosix é a melhor solução para balanceamento de carga.

Os algoritmos para migração de informação são apresentados de seguida com exemplos das soluções que os adaptam. De notar que esses algoritmos são relativos ao espaço de endereçamento de uma aplicação/processo (incluído no já referido estado interno de uma aplicação) e, são um bom resumo das soluções existentes, tanto para soluções ao nível de processo, como para soluções ao nível de uma VM sistema ou VM OO (que também usufruem destes algoritmos para transferir outro tipo de informação, e.g., ficheiros externos à aplicação, ou o sistema de ficheiros de um sistema operativo no caso de uma VM sistema):

- **Eager all** (e.g., Condor [25], LSF [49]): todo o espaço de endereçamento é copiado quando é feita a migração. Não fica nenhuma dependência para trás.

- **Eagir dirty** (e.g., Mosix [4], Locus [47]): apenas as páginas modificadas no espaço de endereçamento é que são copiadas. É idêntico ao copy-on-write (apresentado anteriormente).

- **Copy-on-reference** (e.g., Accent [36], Mach [28]): as páginas do espaço de endereçamento só são transferidas para o nó destino, quando referenciadas.

- **Flushing** (e.g., Sprite [31]): apenas as páginas modificadas no espaço de endereçamento são copiadas para o disco de um servidor central, onde todos os nós consigam aceder.

- **Precopy** (e.g., System V [44]): o processo que está a ser migrado, fica a correr no nó origem, quanto o seu espaço de endereçamento está a ser copiado. Qualquer alteração sobre o espaço de endereçamento durante esta fase, tem de ser enviado numa segunda fase.

Análise dos algoritmos: cada estratégia está desenhada para diferentes objectivos, mas obviamente cada uma tem os seus custos. Sistemas que implementem eager all, eliminam dependências com o nó origem, mas têm um custo de migração altíssimo. Eager dirty reduz esse custo, mas pode diminuir o desempenho dos sistemas durante a sua execução, isto porque, tem de haver um controlo sobre a informação que for sendo modificada. Copy-on-reference é a solução que mais dependências tem com o nó origem, porém, é também a que tem menor custo de migração. Flushing elimina as dependências com o nó origem, mas tem sempre dependências com um servidor central. O custo de migração é muito parecido com o eager dirty. Finalmente, precopy pode ter um custo de migração adicional em relação ao eager all caso hajam alterações ao espaço de endereçamento, mas esta solução tem a vantagem de diminuir o tempo de interrupção dos serviços que possam estar em execução.

2.1.2 VM Sistema

As máquinas virtuais de nível sistema actuais fornecem mecanismos de checkpoint/restore e migração. Estes mecanismos permitem guardar o seu estado de execução de forma persistente, para ser recuperado mais tarde e permitem o transporte do mesmo entre sistemas distintos. Virtualização ao nível de sistema está cada vez mais a ser utilizado devido a soluções de computação em Grid e Cloud.

Actualmente, existem muitas máquinas virtuais de sistema que suportam checkpoint e migração, contudo existem soluções mundialmente conhecidas, como é que o caso de **VMWare**,[1] que não têm qualquer informação no domínio público sobre como estão implementados estes mecanismos. No entanto, existem outras soluções interessantes (e.g., **Xen** [5]), que nos permitem explorar as implementações dos mecanismos de checkpoint e migração a este nível. De seguida, abordamos duas soluções, uma para checkpoint, outra para migração. A solução de checkpoint serve de exemplo para definir o estado de execução para soluções deste tipo. A solução de migração, apresenta um novo conceito, denominado por *live migration* e, esclarece como é que soluções de migração ao nível de uma VM sistema, abordam o problema do transporte do sistema de ficheiros, que em termos gerais, contem grandes quantidades de informação (o que dificulta ainda mais o problema).

Samuel T. King et al. [22] têm como objectivo fazer depuração de sistemas operativos. Para tornar esse objectivo possível, esta solução precisa de criar checkpoints sucessivos para que a reposição do estado de execução seja mais eficiente. A solução foi desenvolvida para a máquina virtual UML[2] e, o estado de execução interno a ter em conta numa solução deste tipo é: registos de CPU, memória fisica da máquina virtual, e qualquer estado que esteja dentro da própria máquina virtual. O estado externo engloba de uma forma geral, apenas a imagem virtual em disco. Para evitar que os checkpoints fiquem com dimensões grandes a longo prazo, são aplicados checkpoints incrementais (com copy-on-write).

Soluções de migração ao nível de uma VM sistema, introduzem o conceito de **live migration**: permite transportar uma máquina virtual entre nós distintos, sem interromper os pedidos (e.g., pedidos num servidor web ou aplicacional) que já estão a ser processados, ou seja, sem interromper a execução do sistema e seus processos.

Robert Bradford et al. [9] exploram esse conceito e preocupam-se também em transportar grandes quantidades de informação de forma eficiente. A solução desenhada baseou-se nos mecanismos de migração fornecidos pela máquina virtual Xen. Segundo os autores, a migração do sistema de ficheiros podia ser realizada de duas formas: *on-demand* (idêntico ao copy-on-reference [29]), ou *pre-copying* (semelhante ao eager all [29]). Soluções *on-demand*, têm dependências a longo prazo e, para soluções de migração ao nível da uma VM sistema, essas dependências tornam-se ainda maiores. Por esta razão, adoptaram uma solução *pre-copying* (em conjunto com live migration,

[1] http://www.vmware.com/
[2] User-mode Linux: http://user-mode-linux.sourceforge.net/

muito parecido a uma solução precopy [29]), para evitar que o desempenho dos mecanismos fosse afectado a longo prazo.

Os detalhes de implementação sobre live migration em [9], são apresentados de seguida. O transporte do sistema de ficheiros e do estado de execução é efectuado em paralelo para um nó destino. Se houver alterações no sistema de ficheiros durante esse transporte, ficam registados à parte (registados como deltas). A partir do momento que o transporte do sistema de ficheiros e do estado de execução é dado por terminado, são aplicados os deltas por ordem de chegada no nó destino. Quando essa operação terminar, o mecanismo de migração inicia a máquina virtual nesse nó. Seguidamente, existe um ajuste do endereço IP no serviço de DNS, para o endereço do nó destino, de forma a que novos pedidos sejam encaminhados para o novo nó. Quando não existirem mais pedidos para processar no nó origem, a execução da máquina virtual nesse nó pode terminar.

2.1.3 VM OO

As soluções de checkpoint e migração ao nível de uma VM OO existentes são de forma geral limitadas e insuficientes, tal como foi abordado no Capítulo 1 (Introdução) deste documento. Porém, muito trabalho já foi desenvolvido, e existem soluções que são importantes mencionar pelo facto de explorarem técnicas que poderão ser úteis na solução proposta deste trabalho.

O estado de execução que se tem de ter em conta é definido pela arquitectura da máquina virtual em questão. Na Figura 2.1, mostramos um exemplo da arquitectura de uma máquina virtual em termos gerais (inspirado na máquina virtual Java).

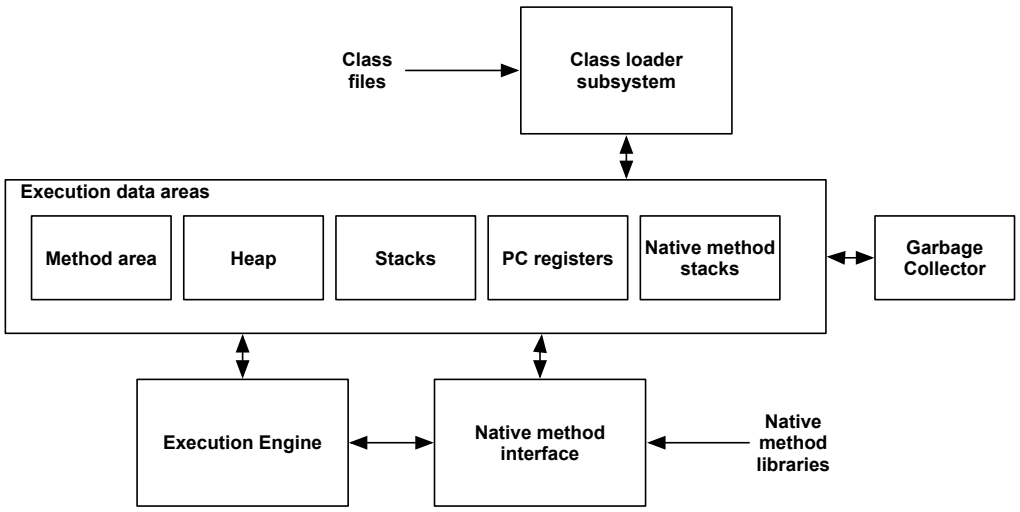

Figura 2.1: Visão geral da arquitectura de uma VM OO (Java).

O estado de execução pode então ser definido. Como estado interno:

- Method area: guarda o código das classes do programa.
- Heap: zona de memória global que guarda objectos e arrays que são alocados dinamicamente.

- Stacks: representa o estado de todos os métodos (não nativos) por thread, que estão em execução. Guarda as variáveis locais, resultados intermédios (operandos) e argumentos de cada método.

- Registos PC (contadores de programa): cada nova thread tem associada automaticamente um contador de programa próprio. Se uma thread está a executar um método não nativo, então esse contador aponta para a próxima instrução (bytecode) que tem de ser executada.

- Native method stacks: o estado de execução de métodos nativos é guardado numa forma dependente da implementação nesta zona.

O estado externo à aplicação, é definido de forma idêntica como nas soluções ao nível de processo através do conteúdo dos ficheiros (incluindo os seus descritores) e sockets. Cada máquina virtual terá a sua arquitectura especifica, mas a arquitectura apresentada é um bom exemplo a ser usado em termos gerais para melhor compreensão do estado de execução que os mecanismos de checkpoint e migração têm de ter em conta.

Soluções de migração a este nível de implementação, definem dois conceitos importantes, que também se mantêm coerentes com soluções de checkpoint: **weak mobility** e **strong mobility** [11, 15, 21]. Estas conceitos começaram a ganhar ênfase em soluções como agentes móveis, isto porque, um agente móvel é um excelente exemplo de weak mobility, onde apenas é transportado código e alguma informação, enquanto que com strong mobility, para além de se transportar essa informação, também é transportado o estado de execução, que costuma ser o desafio principal das soluções existentes.

A grande maioria das máquinas virtuais para linguagens de alto nível suporta mecanismos de serialização e carregamento dinâmico de objectos, o que permite facilmente implementar um sistema com weak mobility. É importante referir que, grande parte das soluções ao nível de uma VM OO têm como ponto de partida um mecanismo de serialização fornecido de base pela própria máquina virtual. Tomando como exemplo o mecanismo de serialização do Java, este permite guardar e recuperar o estado de um objecto e, possibilita também o transporte desse mesmo objecto entre nós diferentes. Com apenas serialização, atinge-se ao mesmo tempo, persistência e transporte de informação.

Relativamente às implementações efectuadas ao nível de uma VM OO, estas podem ser subdivididas em dois níveis:

- **Ao nível da própria VM OO**: verificam o requisito de completude, isto é, ter acesso a todo o estado de execução. No entanto têm problemas de eficiência e portabilidade (as outras máquinas virtuais também têm de ter as alterações de código para que os mecanismos possam fazer efeito). Este tipo de solução normalmente é realizado através de modificações ou extensões do código da própria máquina virtual, introduzindo novas funcionalidades a

partir de bibliotecas que permitam efectuar checkpoint ou migração. Temos como exemplos: Merpati [42], OCVM [2], CIA [21], MobileJikesRVM [11], Sumatra [1], JavaThread [8], Nomads [43], ITS [7], Jessica2 [24]

- **Ao nível das aplicações que se executam na VM OO**: são portáveis, mas têm problemas de eficiência (pelo excesso de novas instruções adicionadas) e não cumprem o requisito de completude. Neste nível o código (código fonte ou bytecode) é transformado por um pré-processador que adiciona novas instruções ao código da aplicação (instruções estas que servem para capturar ou restaurar o estado). Temos como exemplos: WASP [15], JavaGoX [38], Brakes [45], JavaGo [39], M-JavaMPI [27].

De seguida detalhamos um conjunto de soluções que consideramos interessantes pelas respectivas técnicas usadas para efectuar checkpoint ou migração. Essas técnicas focam-se maioritariamente em algoritmos de **type inference**, **dynamic deoptimization**, **exception handling**, entre outras. Para além destas técnicas, existe ainda um conjunto de soluções que se aproveitam de um mecanismo de depuração oferecido de base pela máquina virtual (e.g. JPDA[3] no Java) para retirar e recriar o estado de execução de uma aplicação. Posteriormente, vão ser abordadas algumas questões que ainda não foram mencionadas, mas que são importantes para qualquer mecanismo de checkpoint e migração, estando estas relacionadas com: como se pode parar a máquina virtual (para poder obter um estado consistente); como é que se reconstrói a informação do estado de execução num restore; entre outras.

Merpati [42], é uma solução para checkpoint de aplicações que se executam sobre o Java. Para efectuar um checkpoint consistente, é necessário primeiro garantir que todas as threads se encontrem num ponto seguro, de forma a poder parar a sua execução. Uma thread encontra-se num ponto seguro quando está num estado de preempção, isto é, já terminou de executar a instrução Java corrente e ainda não foi buscar a instrução seguinte e, se não estiver a executar código nativo. É importante salientar que, como a máquina virtual não tem controlo sobre estado nativo, é preciso limitar o tempo de espera dos pedidos de checkpoint, para não haver uma espera indeterminada.

Para obter o estado de execução de uma thread, é preciso analisar os stack frames correspondentes. A informação das variáveis locais e temporárias (operandos), está disponível em cada stack frame, contudo a informação do tipo dessas variáveis é desconhecido para a máquina virtual. Este problema é comum em termos gerais nas máquinas virtuais baseadas em stack (a máquina virtual não tem de conhecer os tipos das variáveis, isto porque, em execução as instruções bytecode indicam como manipular a informação, e.g. iadd para somar dois valores inteiros). Para resolver este problema, é necessário inferir o tipo das variáveis (**type inference**), e pode ser feito de duas maneiras:

- Actualizando durante execução: o tipo das variáveis é actualizado em cada instrução Java.

[3] Java Platform Debugger Architecture.

Consequentemente, quando existe um pedido de checkpoint, a informação do tipo das variáveis já está disponível.

- Calcular quando na acção de um pedido: o tipo das variáveis, é calculado apenas quando existe um pedido de checkpoint. Para determinar o tipo das variáveis desta forma, é necessário realizar os seguintes passos:
  - Primeiro é preciso efectuar uma análise do controlo de fluxo sobre o bytecode, e como resultado é gerado um grafo de controlo de fluxo.
  - De seguida, a máquina virtual avalia o único caminho possível entre o ponto de entrada e o contador de programa (PC), baseado no grafo de controlo de fluxo. Com base nesse caminho, é possível retirar a informação do tipo das variáveis, analisando as instruções bytecode que se encontram nesse caminho. Este último passo é denominado por análise do fluxo da informação.

Como actualizar a informação do tipo das variáveis durante execução reduz drasticamente o desempenho da máquina virtual, então adoptaram a segunda solução.

Como desvantagens desta solução, apenas se releva a seguinte: se uma thread já estiver bloqueada (e.g. para sincronização com outra thread, ou numa chamada nativa bloqueante), então não é possível efectuar checkpoint. Portanto, não poder usufruir do mecanismo quando threads já se encontram bloqueadas, é um problema grave.

OCVM [2], é uma solução de checkpoint desenhada para a máquina virtual OCaml[4]. Desta solução é importante referir dois pontos. Primeiro, é a única solução ao nível de uma VM OO que faz checkpoint para memória (através da cópia de processos) antes de a realizar para disco. Tal como foi visto na solução libckpt (apresentada anteriormente ao nível de processo), e pelas mesmas razões, os ganhos de desempenho não são significativos. O outro ponto importante de referir é o seguinte: a informação do espaço de endereçamento da heap é guardada como um bloco de informação. No restore, os endereços de memória desse bloco, possivelmente serão diferentes. Como tal, é preciso efectuar um passo adicional, para garantir que os apontadores dos endereços de memória antigos, sejam actualizados para os novos endereços.

Esta solução não apresenta muitos detalhes de implementação, o que compromete a sua completude e, o seu âmbito é muito restrito pelo facto de não ser uma máquina virtual usada mundialmente, como é o caso das máquinas virtuais para a linguagem de programação Java.

JavaThread [8], é uma solução para migração de threads sobre Java. Esta solução tem como objectivo principal reduzir os custos de desempenho do mecanismo de migração durante a execução da aplicação. Os resultados mostram que não existem custos de desempenho adicionais durante execução. Para além de usufruírem de um mecanismo de type inference (tal e qual como na solução Merpati), abordam o problema de extrair e reconstruir o estado de execução sobre métodos

[4] Objective Caml - http://caml.inria.fr/

optimizados (JIT'ed).[5] Em Java, métodos optimizados são compilados dinamicamente para código nativo. Isto significa que a execução desses métodos deixa de estar representada nas stacks Java e passa a estar representada nas stacks nativas. Como as soluções de type inference baseiam-se nas stacks Java, então quando existem métodos optimizados, surge um problema. Este problema foi resolvido com base numa análise das estruturas de dados e interfaces disponíveis na máquina virtual em questão (Sun JVM 1.3). Os autores descobriram que essa máquina virtual suportava um mecanismo de **dynamic deoptimization** que transformava frames nativos em frames Java.

Dynamic deoptimization foi introduzido com o objectivo de resolver problemas de inconsistência gerados pelo compilador, quando na presença de métodos *inlined*. A listagem de código 2.1, demonstra o problema. Neste exemplo, o método m1 invoca um método m2 da classe C1. Para optimização, o compilador JIT pode incluir m2 em m1. Mas esta inclusão pode ficar inválida se C2, uma subclasse de C1, que substitui m2, for carregada dinamicamente (o que faria com que m2 não existisse em C1, o que não é verdade). Com isto, dynamic deoptimization é usado para reverter métodos optimizados em métodos normais.

```
public class MyClass {
        void m1() {
                C1 o;
                o = getInstanceOfC1();
                ...
                o.m2();
                ...
        }
}

public class C1 {
        void m2() {
                ...
        }
}

public class C2 extends C1 {
        // Método substituído
        void m2() {
                ...
        }
}
```

Listagem 2.1: Exemplo de código para métodos inlined.

MobileJikesRVM [11], suporta migração de threads (apenas uma thread) sobre Java. Esta solução foi implementada na máquina virtual JikesRVM, a mesma da máquina virtual da solução proposta neste documento, porém, como é uma solução desenhada para agentes móveis, muitos problemas ficaram por resolver. Essa solução releva a importância dos tipos de migração que podem ser realizados num mecanismo de migração de threads:

- **Migração reactiva**: uma thread pode ser interrompida para migração sem qualquer previsão (a thread não invoca nada, é-lhe pedido, e tem de tratar isso a qualquer altura). Obviamente

[5] Just-In-Time: métodos compilados em execução por motivos de optimização.

que vai ter de ser tomada uma acção de migração apenas quando for possível, isto é, não é imediata porque senão podia ser criado um estado inconsistente do sistema.

- **Migração proactiva**: é desencadeada por si mesmo (a thread invoca explicitamente em si própria o método de migração, quando pretende migrar).

Para implementar migração reactiva, esta solução podia usufruir dos pontos seguros definidos em Merpati [42], mas para esta solução, os pontos seguros foram definidos usando outra abordagem. Para garantir que uma thread está num ponto seguro (para efectuar posteriormente uma migração), foram usados *yield points*. Os yield points (na versão 2.4.6 da JikesRVM) são pontos no código em que as threads verificam se podem continuar em execução. Quando esta verificação ocorre, o mecanismo desenvolvido bloqueia a thread em questão para realizar uma migração. Estes yield points são inseridos automaticamente pelo compilador (no inicio e fim de cada método e na iteração de ciclos), quando compila os métodos pela primeira vez. Desta forma, sempre que uma thread for migrada, esta não altera o seu estado durante esse processo, garantindo assim consistência.

CIA [21], é uma solução de migração de threads para a máquina virtual Java. Esta solução aproveita a interface de depuração do Java (JPDA) para extrair e recuperar o estado de execução de uma aplicação. O JPDA quando usado para implementar mecanismos de migração (e checkpoint), tem alguns problemas. Com JPDA não é possível aceder aos valores na pilha de operandos. Para resolver esse problema restringiu-se o programador, para não fazer invocações aninhadas (e.g. variável = invocaA() + invocaB()), obrigando-o a guardar os resultados intermédios dessas invocações, invocaA e invocaB, em variáveis auxiliares temporárias. Outro problema com JPDA é que não permite redefinir o contador do programa (PC) para uma posição específica. Para contornar esse problema, estenderam a máquina virtual com interfaces que suportam modificar o PC.

Para além dos problemas acima indicados, surgiram outros problemas que não conseguiram contornar. Primeiro de tudo, as aplicações têm de executar-se em modo de depuração, o que diminui significativamente o desempenho da execução das aplicações. Adicionalmente, tentaram trabalhar também aplicações com mais do que uma thread, mas uma abordagem de checkpoint ou migração usando o JPDA limita a solução apenas para uma thread.

WASP [15], é uma solução de migração para agentes móveis sobre Java, mas que suporta mais do que uma thread. Esta solução modifica o código fonte Java através de um pré-processador, que adiciona instruções para suportarem migração. Essencialmente, existe uma interface, denominada `saveState`, que pode ser usada pelo programador, de forma a poder guardar o estado de execução do agente móvel. Esse pré-processador quando executado, percorre todas as classes à procura dessa interface, e quando a encontra, instrumenta-a de forma a transformar-se num bloco try-catch (**exception handler**). No bloco de código try é lançado um erro (idêntico a uma excepção, mas não requer que seja declarado na assinatura de um método) para iniciar um pedido de migração. No bloco de código catch, esse erro é interceptado e, tem a responsabilidade principal de guardar

o estado de execução do método correspondente. Quando essa operação termina, o erro é enviado novamente (para cima), para ser apanhado por outros métodos em execução (para que esses também possam guardar o seu estado de execução, de igual forma). Este último passo é possível porque todos os métodos que invocam aqueles que efectuam `saveState`, também são instrumentados com blocos try-catch, e assim sucessivamente.

No restore, foi necessário reajustar o contador de programa, de forma a não executar novamente instruções que já tenham sido executadas. Para tal, definiram formas que permitem saltar partes do código em cada método. Basicamente, são definidas zonas no código, que estão protegidas com instruções if, onde é especificado se uma área do código deve ou não ser executada (quando executada pela primeira vez no restore).

Este tipo de abordagens tem duas grandes desvantagens: não suporta aplicações cujo código fonte Java não é fornecido e pior, são dadas responsabilidades ao programador para resolver problemas internos da solução.

M-Java-MPI [27], é uma solução para migração de aplicações MPI[6] sobre Java. Esta solução combina as técnicas exploradas anteriormente dos sistemas CIA e Merpati. O checkpoint é efectuado através do JPDA e o restore através de exception handlers.

No restore, todos os métodos que precisam de recuperar o estado de execução estão instrumentados com exception handlers. Quando os métodos são executados pela primeira vez, um erro é lançado num block try-catch, e no catch é efectuada a reconstrução do estado a cada método. Esta abordagem é muito idêntica à que vimos no Merpati para guardar o estado de execução, mas neste caso é usado para recuperar o estado de execução.

Como este sistema adopta JPDA como base da solução para extrair o estado de execução, então tem como desvantagem principal apenas conseguir retratar o estado de uma thread. Para além desta, obriga a que as aplicações sejam desenvolvidas sobre MPI, o que não é de forma alguma geral.

Análise das soluções VM OO

Na Tabela 2.2, é apresentado um resumo das soluções anteriormente detalhadas. Os parâmetros comparativos focam-se no estado de execução suportado (interno e externo) e nas propriedades de transparência, portabilidade e eficiência. De notar que, grande parte das soluções são completas em relação ao estado interno, mas apenas para o estado de execução a que se comprometem retratar. M-Java-MPI por exemplo, compromete-se a migração de aplicações, mas apenas suporta aplicações com uma thread.

Ainda relativamente ao estado interno, existe um conjunto de incapacidades das soluções disponíveis que valem a pena serem discutidas. Primeiro é importante referir que, algumas das soluções desenhadas para checkpoint ou migração de aplicações, suportam aplicações com múltiplas thre-

[6] Message Passing Interface.

ads mas no entanto, não se preocupam em retratar o estado de sincronização das mesmas. Isto indicia apenas que essas soluções suportam múltiplas threads que não interagem entre si, o que não costuma ser típico nas aplicações deste género. Outra incapacidade relevante está relacionada com threads que já estejam anteriormente bloqueadas. Não existe nenhuma solução que retrate este problema de forma explicita e no entanto muitas aplicações têm threads que não estão em execução (seja por motivos de sincronização, leituras bloqueantes, entre outros).

Respectivamente aos problemas mais comuns das soluções ao nível de uma VM OO, estes focam-se essencialmente na propriedade de transparência e no estado externo. Em relação à transparência, no pior caso, o programador tem de modificar a sua aplicação de forma a invocar explicitamente os mecanismos suportados, ou então o programador tem de conhecer o modelo de programação, e.g. MPI. O estado externo é um problema ainda com poucas soluções. Tal e qual como para as soluções ao nível de processo, não existe nenhuma que retrate este problema na sua totalidade. Grande parte das soluções não dão suporte a ficheiros ou sockets, e algumas delas aproveitam-se de sistemas de ficheiros distribuídos para resolver os problemas de re-colocação dos ficheiros de uma aplicação. Porém quando se trata de soluções a larga escala (e que retratem grandes quantidades de informação), pode não fazer sentido usar um sistema de ficheiros distribuído [9].

2.2 Mecanismos de log e replay

Os mecanismos de log e replay estão directamente relacionados com a solução proposta por duas razões (referidas anteriormente, mas valem a pena serem relembradas). Por um lado existem soluções que conseguem realizar checkpoint, através de uma vertente de log e replay. Por outro lado, pode-se constatar que as soluções de log e replay actuais usufruem de mecanismos de checkpoint para melhorar o seu desempenho. Ambos os tipos de solução vão ser abordados na sua secção respectiva.

2.2.1 Checkpoint através de log e replay

Alguns dos mecanismos de log e replay, conseguem realizar checkpoint através de sistemas de replicação. Este tipo de abordagem é muito idêntico a replicação passiva [12] (solução de backup primário-secundários). Essencialmente, todas as acções ou operações efectuadas num sistema primário, são interceptadas e replicadas entre os vários servidores secundários. Estes servidores secundários, evoluem globalmente, passando instruções entre primário e secundários, para ficarem todos no mesmo estado. Se ocorrer uma falha no sistema primário, é eleito um secundário para o seu lugar. Os servidores secundários têm cópias da execução do sistema principal, e por este motivo, pode-se afirmar que os secundários contêm um checkpoint do sistema primário. De seguida, abordamos duas soluções que implementam este conceito.

Hypervisor [10] é uma solução de log e replay ao nível de uma VM sistema. Os desafios que tiveram de ser resolvidos estão principalmente relacionados com as interacções com o ambiente:

- Output para o ambiente: Só o sistema primário é que passa a informação para o ambiente.
- Input do ambiente: O sistema primário é que toma todas as decisões não determinísticas. Quando o primário falha, um secundário é eleito para efectuar as mesmas escolhas. É desejável que o sistema como um todo se mantenha consistente (e.g. o relógio do novo primário não pode fornecer uma hora diária mais atrasada que o antigo primário). Este tipo de problemas, foi resolvido recorrendo a mecanismos de sincronização de relógios (com piggyback nas mensagens trocadas entre os vários servidores).

Jeff Naper et al. [30] transpõem a mesma ideia, mas para a máquina virtual Java. Tal como no Hypervisor, foi desenvolvido um sistema de replicação com uma arquitectura de backup primário-secundários. Para além dos problemas que já tinham sido encontrados, sendo estes, efeitos não determinísticos, excepções assíncronas, output para o ambiente, entre outros, também teve de ser revolvido o problema do não determinismo relacionado com múltiplas threads. Mais uma vez é uma solução baseada em máquinas de estado, cujas alterações são efectuadas à custa de comandos ou instruções, que são transportadas entre os vários sistemas.

2.2.2 Log e replay com checkpoint

É frequente existirem soluções de log e replay que usam mecanismos de checkpoint para melhorar o seu desempenho. É importante reter que se essas soluções apenas usassem log e replay, teriam um custo no desempenho muito elevado. A razão principal pela qual isso acontece é porque com log e replay, todos os comandos ou instruções, provocam avanços muito pequenos na recuperação do sistema. Em vez de avançar um sistema, executando todas as instruções guardadas a partir de um ponto inicial, com checkpoint é possível aproximar mais eficientemente o sistema para um ponto que seja pretendido, e a partir deste, aplicar as instruções preservadas no log.

As soluções anteriormente apresentadas [22, 34, 40, 46], são de facto soluções que se encaixam neste contexto de soluções. Porém, como retratavam aspectos importantes respectivos a checkpoint, foram antecipadamente abordadas nessa secção. Em [14, 32] são apresentadas soluções mais recentes, que incidem mais em questões relacionadas com log e replay e, que por essa razão, valem a pena serem detalhadas nesta secção.

SMP-Revirt [14] foi a primeira solução a fazer log e replay em VM sistema com múltiplos processadores. Para detectar corridas de acesso à informação entre os vários processadores virtuais é usado um mecanismo de protecção de páginas no hardware, que está disponível em qualquer computador e processador moderno. Através deste mecanismo, é possível fazer log e replay de toda a máquina virtual, incluindo do núcleo do sistema e de todas as aplicações, sem impor modificações no software. Este mecanismo de protecção de páginas no hardware evita os custos de desempenho adicionais das soluções que manipulam todas as instruções de leitura e escrita ao nível do software, mas requer um controlo adicional e com alguma cautela das zonas de informação partilhadas pelos diversos processadores.

Soyen Park et al. [32] têm como objectivo principal reduzir os custos elevados de desempenho que existem nas soluções de log e replay. Grande parte das soluções que fazem log e replay para repetir falhas dos sistemas, tentam reproduzir uma falha logo na primeira tentativa de replay. Como resultado os sistemas têm um custo de desempenho elevado (10-100 vezes mais lentos), isto porque têm de guardar todos os eventos não determinísticos, enquanto as aplicações estão em execução. Soyen Park et al. propõem como solução, apenas guardar os eventos não determinísticos essenciais, que consigam reconstruir o estado de um sistema. Mesmo que o estado reconstruído não seja exactamente igual ao estado que originou a falha, isso não constitui problema, porque o interesse nem é reconstruir um estado exactamente igual, mas sim reproduzir a falha. A ideia está bem explorada, e a partir do momento em que se consegue reproduzir a falha, este consegue reproduzi-la sempre da mesma forma, em qualquer altura.

Análise global do trabalho relacionado

As soluções de checkpoint e migração existentes estão implementadas a diferentes níveis: nível de processo, VM sistema, e VM OO, o assunto principal deste trabalho. As abordagens realizadas ao nível de processo e VM sistema são insuficientes pelas seguintes razões: ou obrigam a que seja guardada / transportada informação que não é relativa apenas à aplicação em si, ou limitam a portabilidade da mesma. As soluções ao nível de uma VM OO são mais recentes mas também têm alguns problemas. A maioria encaixa-se no contexto de agentes móveis, e por isso, assumem que o ambiente de execução é bem definido e controlado. Outro problema comum está relacionado com o estado de execução que conseguem retratar. São poucas as soluções que trabalham as aplicações como um todo (ou seja, aplicações com múltiplas threads), e ainda são menos aquelas que trabalham o estado externo à aplicação. Em termos globais, o estado externo à aplicação foi sempre a propriedade menos trabalhada. Para além dessa propriedade, as soluções tendem sempre a dar responsabilidades ao programador (diminuindo a transparência), de forma a usufruir destes mecanismos (no pior caso, as interfaces disponíveis têm de ser invocadas explicitamente na aplicação). Isso pode ser um problema muito grave, isto porque, quem utilizar os mecanismos de checkpoint ou migração pode não ser o próprio programador, e como tal, podem surgir resultados inesperados (e.g., erros de compilação) quando uma aplicação é compilada com esses mecanismos. Pode até acontecer que o código fonte de uma aplicação nem seja fornecido, o que não permitiria usufruir destes mecanismos.

Numa outra vertente de checkpoint, abordamos soluções de log e replay, que realizam checkpoint através de sistemas replicados, e são também um bom exemplo recente que demonstra que os mecanismos de checkpoint presentemente são utilizados.

Nome	Nível implementação	Contexto	Estado interno	Descritores ficheiro	Conteúdo ficheiros	Sockets	Multi thread	Transparente	Portável	Eficiente
Merpati [42]	VM OO	Checkpoint Processo	Completo	Não	Não	Não	Sim	Semi	Semi	Semi
OCVM [2]	VM OO	Checkpoint Processo	Completo	Sim	Não	Não	Sim	Semi	Semi	Semi
CIA [21]	VM OO	Migração Thread	Incompleto	Não	Não	Não	Não	Semi	Semi	Não
MobileJikesRVM [11]	VM OO	Migração Thread	Completo	Sim	Parcial	Não	Parcial	Semi	Semi	Sim
JavaThread [8]	VM OO	Migração Thread	Completo	Não	Não	Não	Não	Semi	Semi	Sim
WASP [15]	VM OO App	Migração Processo	Completo	Sim	Parcial	Não	Sim	Não	Semi	Não
M-JavaMPI [27]	VM OO App	Migração Processo	Incompleto	Não	Não	Sim	Não	Semi	Sim	Não

Tabela 2.2: Resumo das soluções de checkpoint e migração detalhadas ao nível da uma VM OO.

3

Arquitectura

Contents

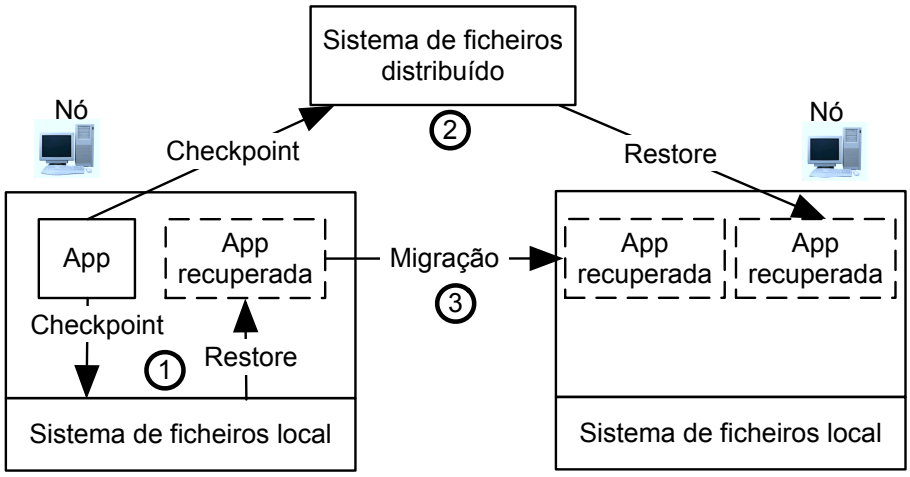

Figura 3.1: CRM-HLL-VM visão geral das funcionalidades suportadas.

Nesta secção descrevemos os aspectos principais da arquitectura da solução desenvolvida. Começamos com uma descrição geral dos mecanismos de checkpoint/restore e migração, e depois abordamos a arquitectura interna da máquina virtual estendida. No fim, apresentamos questões de arquitectura ligadas às propriedades oferecidas da solução desenvolvida (Secção 1.1).

A Figura 3.1 apresenta uma visão das funcionalidades suportadas por este trabalho:

1. Checkpoint para disco local e seu respectivo restore posterior;
2. Checkpoint para um sistema de ficheiros distribuído, e qualquer nó que esteja ligado a esse sistema de ficheiros, consegue fazer restore do estado persistido. Pode ser considerada como uma migração indirecta e/ou replicação indirecta, pois podem ser lançadas mais instâncias com o restore e suas réplicas dos ficheiros;
3. Migração entre dois nós. A migração é efectuada de forma directa entre dois nós.

Na Figura 3.2, apresenta-se a arquitectura deste trabalho, que consiste num conjunto de componentes que se focam principalmente na propriedade de transparência, descrita anteriormente, e/ou no transporte da informação. Existem três componentes primários:

- **Aplicação**: executada no contexto de uma máquina virtual estendida com mecanismos que suportam checkpoint, restore e migração. De seguida, retratamos a interligação dos vários componentes internos (migração, checkpoint, state extraction, *daemon* migração, restore e state restoration, presentes na figura) e suas respectivas funções.

  State extraction captura o estado de execução correspondente a todas as threads presentes na aplicação. Checkpoint tem a obrigação de parar todas as threads em execução (de forma a garantir consistência). De seguida invoca state extraction e finalmente no fim, o estado de execução extraído é guardado de forma persistente para um sistema de ficheiros (local ou distribuído). Migração invoca checkpoint e envia o estado de execução para uma rede.

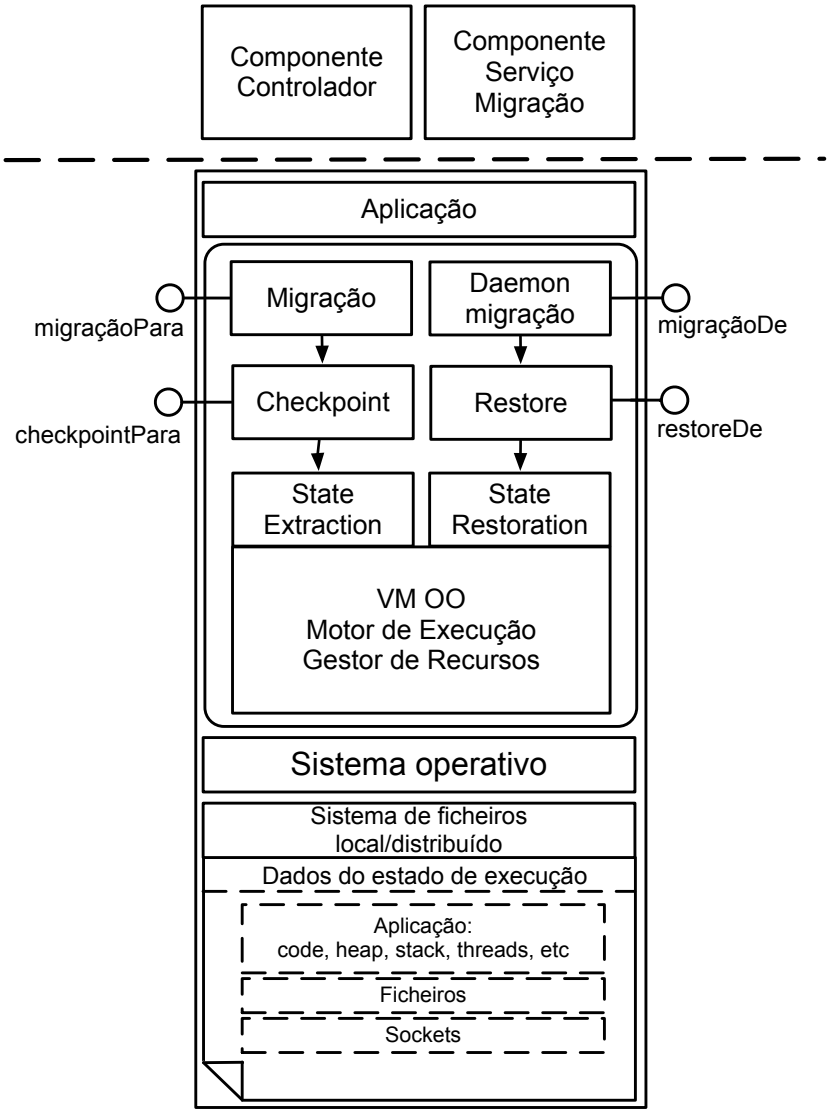

Figura 3.2: CRM-HLL-VM arquitectura num único nó.

State restoration tem como responsabilidade principal recriar o estado de execução numa nova aplicação, o que corresponde a reconstruir e continuar a execução de todos os stack frames,[1] para todas as threads e, quando essa operação terminar, reinicia a execução da aplicação. Restore garante que a nova aplicação criada pode iniciar state restoration e adicionalmente se for pedido, obtém o estado de execução a partir de um sistema de ficheiros. O *daemon* migração é usado numa migração, e recebe o estado a partir da rede. Quando o estado de execução estiver disponível, esse *daemon* invoca o restore sobre esse estado.

Os métodos/serviços checkpointPara e migraçãoPara (disponíveis também para as aplicações), são desencadeados pelo controlador. A aplicação tem uma thread especial que está em escuta num determinado socket. Este socket torna possível receber ordens/comandos vindos

[1] Um stack frame corresponde à invocação de um método que ainda não terminou execução com um return.

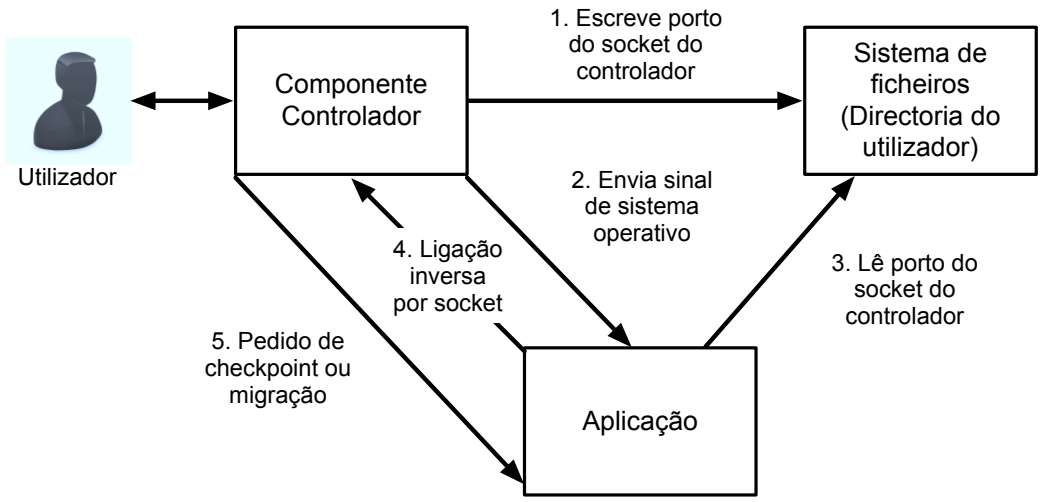

Figura 3.3: Interacções entre componentes controlador e aplicação.

do exterior. Quando desencadeados, os comandos correspondentes são enviados para as componentes internos, checkpoint ou migração. Os métodos/serviços migraçãoDe e restoreDe, são apenas canais de entrada, para receber a informação relativa ao estado de execução.

- **Controlador**: programa linha de comandos, que comunica com a aplicação de forma a tirar partido dos mecanismos desenvolvidos. É a partir deste controlador que um utilizador (não precisa de ser o próprio programador) consegue controlar os mecanismos desenvolvidos sobre a aplicação, parametrizando-o com comandos/instruções dependendo do resultado desejado.
- **Serviço migração**: servidor presente em todos os nós, que recebe as aplicações migradas. Este servidor responde também a pedidos de classes e ficheiros que são requisitados ao longo do tempo.

As interacções entre os vários componentes (em duas situações especificas) vão ser abordadas de seguida. De forma a usar externamente os mecanismos de checkpoint e migração, o controlador tem de descobrir qual é o porto do socket que está em escuta na aplicação (para activar os métodos checkpointPara ou migraçãoPara). Na Figura 3.3 ilustramos como é que se processam essas interacções. Basicamente, o controlador também está em escuta num determinado socket com um porto especifico e, envia um sinal de sistema operativo [41] à aplicação (usando o identificador do processo, e.g. Unix Process ID). De notar que, a aplicação em si corresponde ao processo que corre a instância da máquina virtual, que por sua vez executa a aplicação respectiva. As interacções são de facto efectuadas através do código da máquina virtual. Quando a aplicação recebe este sinal, inicia um processo de ligação no sentido inverso, da aplicação para controlador, via socket, e a partir desse momento é possível haver troca de mensagens entre os dois componentes.

Adicionalmente, para suportar vários controladores (entre diferentes utilizadores no sistema), a informação do socket em que este escuta, pode ser deixada num ficheiro especifico na directoria do

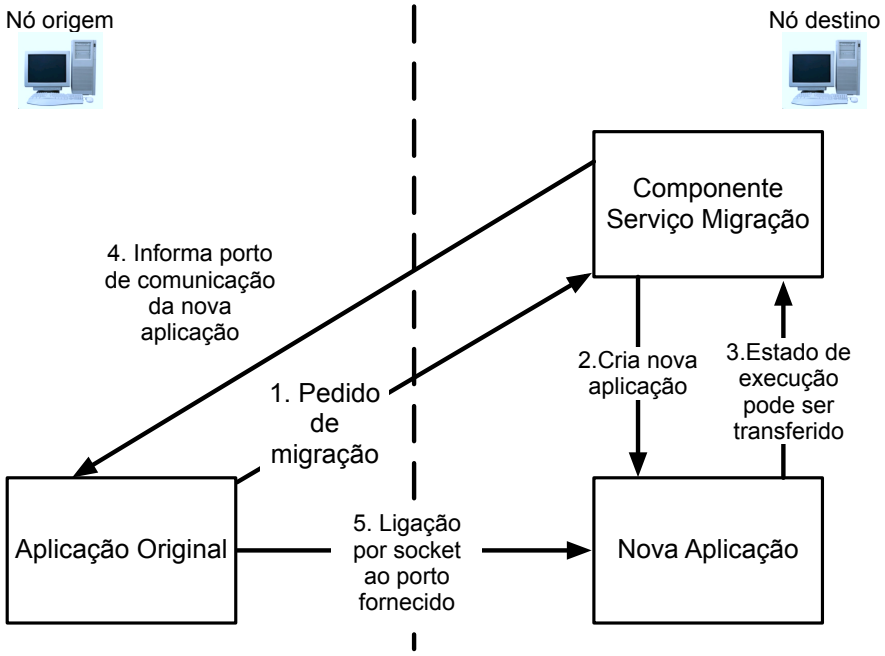

Figura 3.4: Interacções entre componentes na migração de uma aplicação.

utilizador correspondente e, a aplicação antes de efectuar a ligação com o controlador, tem de ler primeiro o socket desse ficheiro. Para suportar situações mais complicadas, em que o mesmo utilizador pode ter vários controladores activos, então teria de haver uma comunicação mais complexa entre ambos os componentes, mas este tipo de abordagem não foi um objectivo a cumprir.

No caso da migração de uma aplicação, a interacção entre os vários componentes é apresentada na Figura 3.4. Essencialmente, a instância da máquina virtual da aplicação comunica com o serviço de migração remoto para iniciar uma migração. Esse serviço é responsável por iniciar uma nova aplicação que vai ficar em escuta num determinado socket (componente interna *daemon* migração). A partir do momento que o estado de execução pode ser transferido, o serviço de migração responde à aplicação original com uma mensagem que indica qual é o porto para o qual onde deve enviar o estado. Daí em diante, o estado pode ser transferido directamente da aplicação original para a nova aplicação criada.

Até ao momento, focámo-nos apenas na arquitectura da solução como um todo e suas ligações com os vários componentes. De seguida abordamos questões de arquitectura ligadas especificamente a algumas das propriedades oferecidas pela solução desenvolvida.

3.1 Propriedade portabilidade: representação intermédia do estado de execução

A implementação das componentes internas state extraction e restoration (Secções 4.5 e 4.6, respectivamente), é muito dependente da máquina virtual onde são desenvolvidas, no entanto,

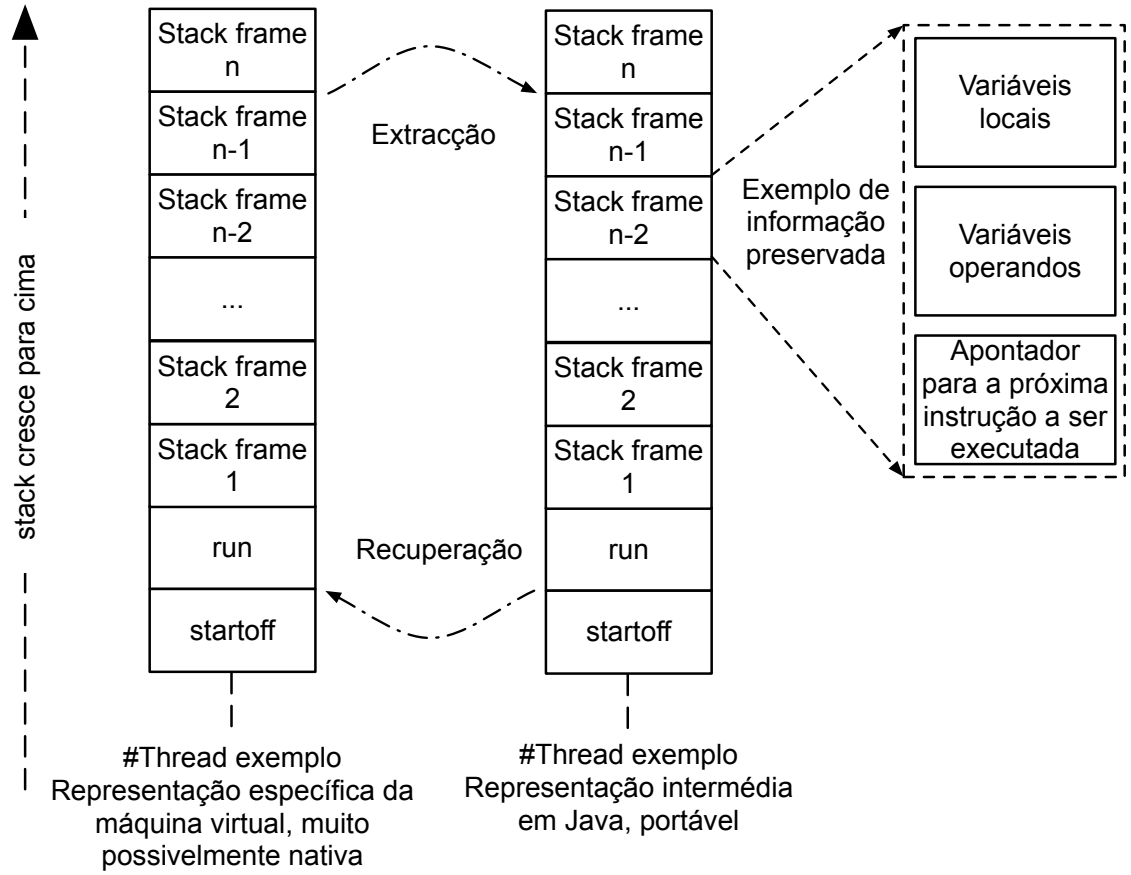

Figura 3.5: Transformação do estado de execução de uma thread para uma representação intermédia em Java para oferecer portabilidade.

de forma a podermos transportar o estado de execução entre sistemas operativos ou arquitecturas diferentes, é necessário criar uma representação intermédia desse estado. Grande parte do estado de execução de uma aplicação já está representado de forma correcta (serializável), contudo, existem alguns casos específicos em que tal não acontece, como é o caso do estado de execução relacionado com uma `Thread`.

A maioria das máquinas virtuais Java têm uma arquitectura baseada em stack (tal como foi apresentado na Figura 2.1, Secção 2.1.3, do trabalho relacionado). Essa stack é constituída por um conjunto de stack frames, que representam o fluxo de execução de cada thread. Independentemente de como o estado de execução é extraído ou recuperado, o estado da stack de uma thread tem de ser transformado numa representação intermédia em Java, isto porque, a implementação da stack de uma thread costuma ter dependências com o ambiente onde se executa (seja dependências das chamadas de sistema do sistema operativo ou de bibliotecas nativas). Na Figura 3.5, demonstramos como é feita essa transformação. De notar que a informação intermédia que é preservada para cada stack frame da stack de uma thread tem de ter em conta as variáveis locais, resultados intermédios (operandos) e o apontador da próxima instrução a ser executada, que possibilitem a recuperação

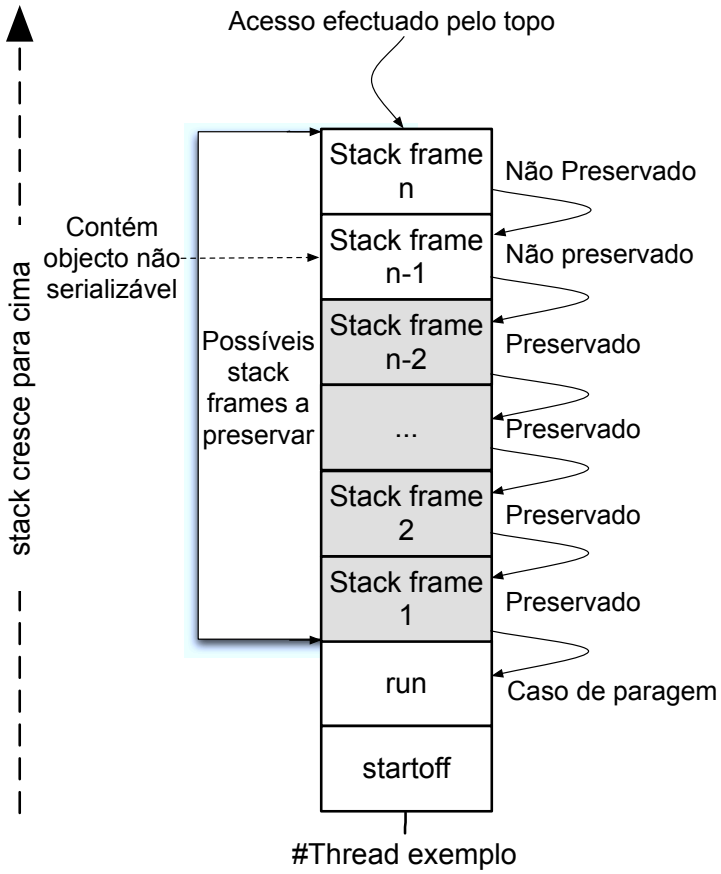

Figura 3.6: Exemplo de stack frames preservados quando na presença de objectos que não são facilmente serializáveis.

(sem perdas de informação) da mesma thread.

3.2 Propriedade transparência: código fonte das aplicações inalterável

Para persistir ou transportar o estado de execução de uma aplicação, pode ser usado um mecanismo de serialização suportado de base pela máquina virtual, onde os mecanismos de checkpoint e migração são desenvolvidos. Grande parte das máquinas virtuais Java existentes, suportam Java `Serialization` de forma embutida, ou através de bibliotecas externas.

Contudo, esse mecanismo de serialização requer que qualquer classe que possa ser serializada implemente a interface `Serializable`. Desta forma, teríamos de confiar que todas as classes da aplicação implementassem essa interface, e consequentemente estaríamos a dar responsabilidades ao programador para o fazer (violando a propriedade de transparência). Porém, esta interface apenas serve para marcar que classes são ou não serializáveis, isto porque, tomando como exemplo a classe `Thread`, esta tem dependências do ambiente onde se executa (como já foi referido na secção anterior) e por isso não pode ser automaticamente/facilmente serializada. Como a solução

proposta neste documento, retrata os problemas da mobilidade desse tipo de objectos, então não é necessário que o código da aplicação implemente de forma explicita essa ou outra interface, que esteja relacionada com este problema.

Adicionalmente, quando for encontrada uma classe/objecto que não é facilmente serializável (e.g., outros objectos internos à maquina virtual que têm dependências sobre o ambiente onde se executam), existem duas soluções possíveis que podem ser adoptadas:

- Criar uma versão especializada ("*externalizada*") do objecto com o mínimo de informação que é necessária (representada por tipos de dados primitivos), que permita a reconstrução desse objecto no restore.

- Evitar serializar o objecto, revertendo a stack de uma thread para a posição de um stack frame onde esse objecto ainda não tivesse sido criado. Isto apenas é possível quando o código que uma thread executou é determinístico e ao mesmo tempo, quando executado novamente, consiga recriar toda a informação como se encontrava anteriormente. Este tipo de abordagem é demonstrada na Figura 3.6 (apenas os stack frames com fundo preenchido são preservados). Essencialmente, o stack frame que contém o objecto não é facilmente serializável, não é preservado e, todos os stack frames acima desse, também não o são.

3.3 Propriedade consistência: pontos seguros e efectivamente seguros

De forma a obter um estado consistente da máquina virtual para realizar um checkpoint, é necessário garantir que todas as threads que não pertençam especificamente à máquina virtual estejam paradas (apenas é preciso ter em conta as threads da aplicação). No entanto, a execução de uma thread só pode ser bloqueada quando se encontra num ponto seguro, isto é, num ponto em que não está a alterar o estado da máquina virtual e ao mesmo tempo, também não está a executar quaisquer instruções (bytecode) da aplicação (Figura 3.7, thread #1). Se uma thread fosse bloqueada enquanto estivesse a executar instruções bytecode da aplicação, podia ficar num estado inconsistente, isto porque, a instrução executada podia não ter terminado completamente e ficar num estado desconhecido.

Este tipo de abordagem seria suficiente para threads que não estão anteriormente bloqueadas. Mas, se uma thread já se encontra bloqueada (e.g., numa leitura de entrada), então não consegue atingir um ponto seguro. Porém, se uma thread já se encontra bloqueada, então está num ponto seguro pelas mesmas razões (denominada por efectivamente segura, Figura 3.7, thread #2). Portanto, se todas as threads se encontram em pontos seguros ou em pontos efectivamente seguros, então a máquina virtual pode ser parada num estado consistente, com alguns cuidados adicionais.

É verdade que threads efectivamente seguras estão realmente em execução, mas no caso de retornarem algum resultado para a maquina virtual (isto é, deixarem de ficar bloqueadas), então

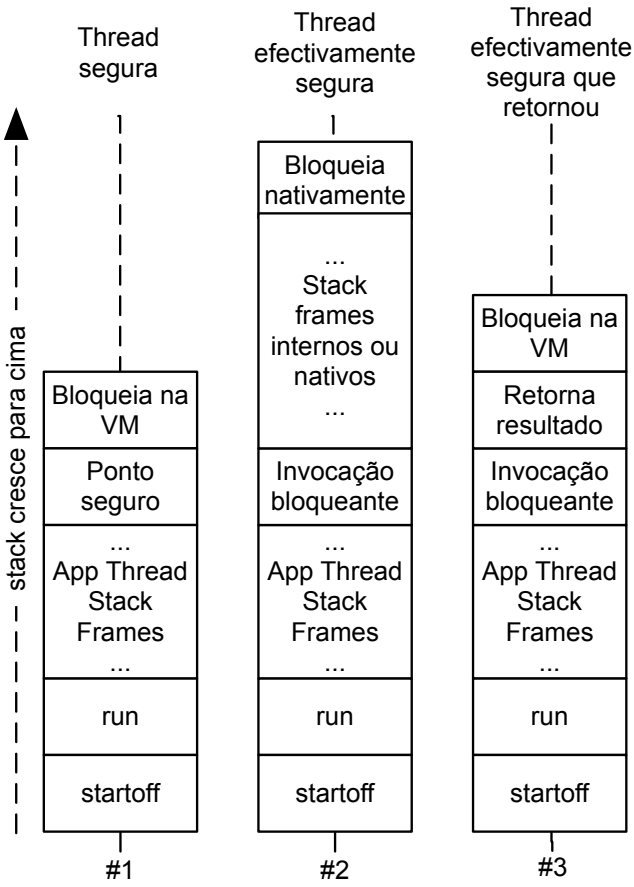

Figura 3.7: Exemplos de threads seguras e efectivamente seguras.

automaticamente são bloqueadas antes de continuarem, reforçando a propriedade de consistência desejada (Figura 3.7, thread #3).

3.4 Propriedade completude: stack frames que têm de ser preservados

Uma thread que se encontre num ponto seguro, tem sempre uma estrutura fixa em termos de construção da stack, isto é, os stack frames de topo vão ser sempre os mesmos, o que faz com que seja facilmente identificável que stack frames são preservados neste caso. Relativamente às threads efectivamente seguras, isso já não acontece. Existem situações variadas para que uma thread seja efectivamente segura, o que faz com que a construção da stack neste caso, seja diferente para cada situação.

Para além disso, é necessário ter em atenção o seguinte. As threads efectivamente seguras têm de ser preservadas de forma a que sejam consistentes para os mecanismos de checkpoint e migração (tendo também em conta que o estado de alguns stack frames, neste tipo de threads, pertence muito possivelmente a estado nativo). Como estas threads se encontram em execução,

os stack frames preservados têm de ser apenas aqueles que não são alterados entre uma transição efectivamente segura (Figura 3.7, thread #2) e uma efectivamente segura que retorna um resultado para a máquina virtual (Figura 3.7, thread #3). Esses stack frames preservados têm ainda que garantir que no restore esse estado efectivamente seguro é repetido, de forma a que não se perca execução da aplicação (para os mecanismos de checkpoint e migração, estas threads efectivamente seguras estão como se nunca chegassem a atingir o seu estado efectivamente seguro). Na Secção 4.4, demonstraremos como é que esta questão foi implementada na máquina virtual JikesRVM.

3.5 Propriedade completude: estado de execução nativo (JNI)

Embora não fosse um objectivo inicial, as técnicas desenvolvidas possibilitaram o suporte de aplicações que fazem invocações JNI. O mecanismo de checkpoint e restore não consegue facilmente inspeccionar e guardar o estado nativo de uma invocação JNI (como já seria de esperar). Contudo, uma invocação JNI encontra-se num estado efectivamente seguro, e essencialmente quando existe um pedido de checkpoint, a stack dessa invocação é revertida (tal como foi mostrado anteriormente, na Figura 3.6), para o último stack frame Java que realiza a própria invocação JNI. Se durante a activação de um mecanismo de checkpoint ou migração, a invocação JNI retornar informação para a máquina virtual, tal como numa thread efectivamente segura, é automaticamente bloqueada. No restore, essa invocação é como se nunca tivesse ocorrido, e é repetida novamente. Isto mantém-se consistente para a máquina virtual.

No caso de haver estado externo à maquina virtual que esteja a ser alterado pela invocação JNI durante um mecanismo de checkpoint ou migração, no restore, na presença de acções deterministas, tudo vai repetir-se da mesma forma como desejado. No caso de acções não deterministas, podem ocorrer problemas, sendo que já estaremos em presença de situações bastante especificas (e.g, uma invocação JNI que escreve num ficheiro de forma aleatória, no restore, essa escrita aleatória vai ser repetida novamente, mas no entanto, já não deveria ocorrer, isto porque, a repetição como é aleatória, irá realizar uma escrita aleatória não desejada). Mas mais uma vez, é importante referir que este tipo de exemplos são raros e só acontecem perante situações não deterministas.

3.6 Propriedade completude: estado de sincronização

O estado de sincronização das threads (e.g., em blocos de código `synchronized` ou métodos `synchronized`) é uma questão muito específica, que está dependente da implementação suportada por cada máquina virtual Java. Para a máquina virtual JikesRVM, os locks de sincronização têm dependências nativas, e como tal evitamos preservar essa informação (usando uma solução idêntica à apresentada na Figura 3.6). Os detalhes de implementação irão ser apresentados na Secção 4.7. É importante referir que, algumas questões da arquitectura relacionadas com o estado de sincronização são muito dependentes da máquina virtual em questão, e como tal são também

discutidas nessa secção da implementação.

3.7 Propriedade completude: estado de execução externo

O estado de execução externo à aplicação não deve ser uma questão deixada para resolver pelo programador, até porque quem usar os mecanismos de checkpoint ou migração pode nem ser o próprio programador que desenvolveu a aplicação. Numa perspectiva geral, a solução proposta para o estado externo tenta ser completa para o tipo de aplicações mais comum (aplicações que manipulam ficheiros e eventualmente comunicam com outras aplicações via socket, incluindo servidores aplicacionais). Para suportar checkpoint ou migração desse tipo de aplicações é necessário preservar a seguinte informação (dos ficheiros e sockets respectivamente):

- Ficheiros: a ligação com o ficheiro (inclui descritor e localização com o nome do ficheiro), conteúdo do ficheiro e a posição (cursor) dentro do ficheiro.
- Sockets: a ligação do socket (inclui descritor), o endereço e porto destino. No caso de um socket servidor, tem de se ter em conta o porto local.

Tal como para o estado de sincronização, algumas questões de arquitectura relacionadas com o estado de execução externo têm dependências fortes com os componentes internos da máquina virtual em questão. Por este motivo, alguns detalhes de arquitectura relacionados com o estado externo, vão ser abordados conjuntamente com a implementação na Secção 4.9.

3.8 Propriedade eficiência

De forma a evitar que haja penalizações no desempenho durante a execução das aplicações, todos os algoritmos que são relativos aos mecanismos de checkpoint e migração têm de ser desenhados de forma a efectuar as suas operações apenas quando existe um pedido de activação desses mecanismos. Pode acontecer que existam operações que tenham de ser mantidas durante a execução das aplicações, mas se minimizarmos isso, as penalizações sobre o desempenho das aplicações durante a execução são reduzidas. Suposto isto, todos os custos de desempenho passam para o ponto em que um mecanismo é activado, sendo que no pior caso, devem ser relativos à dimensão das aplicações em si. O nosso objectivo é evitar que existam penalizações a longo prazo, e o componente que menos tem de ser afectado com essas penalizações é a aplicação.

Resumo

Nesta secção abordámos questões de arquitectura ligadas à solução desenvolvida. Começámos por abordar as funcionalidades oferecidas pela solução desenvolvida: checkpoints locais e remotos com apoio de sistema de ficheiros distribuídos, e migrações directas entre dois nós.

Relativamente aos componentes principais da arquitectura, resumem-se essencialmente aos seguintes: a aplicação, um controlador e um serviço de migração. A aplicação executa-se sobre uma máquina virtual estendida com mecanismos que lhe permitem extrair, recuperar e transportar o estado de execução. O componente controlador permite estabelecer uma relação transparente entre utilizador e aplicação. Finalmente o componente serviço de migração, estará presente em todos os nós, para permitir que o estado de execução possa ser enviado de forma directa entre dois nós.

Questões de arquitectura relacionadas especificamente com as propriedades oferecidas pela solução desenvolvida também foram abordadas. De forma a obter uma solução portável, é preciso criar uma representação intermédia em Java de todo o estado de execução da aplicação. Para serializar em disco ou para uma rede esse estado de execução, os mecanismos de serialização do Java fornecidos pela máquina virtual têm de ser configurados de forma a não dar responsabilidades ao programador. Para poder obter um estado consistente da aplicação, é preciso garantir que todas as threads são paradas em pontos seguros, ou efectivamente seguros. Para que os mecanismos de checkpoint e migração possam ter em conta estado nativo (JNI), esse estado tem de ser revertido (pelo facto de não haver controlo sobre estado nativo), tal e qual como se nunca tivesse ocorrido, e manter-se consistente. O estado de sincronização é muito dependente da implementação, mas como muito possivelmente estamos a trabalhar com locks que têm dependências nativas, esse estado de sincronização também é revertido, com a atenção especial de que no restore tem de ser recriado novamente. Relativamente ao estado externo, não deve ser uma questão deixada para resolver pelo programador, até porque quem usar os mecanismos de checkpoint ou migração pode nem ser o próprio programador que desenvolveu a aplicação. Em termos de eficiência, tentamos reduzir penalizações a longo prazo ligadas directamente com a execução da aplicação.

Questões ligadas directamente com a extracção do estado, e sua respectiva recuperação, são muito dependentes da implementação, e como tal, questões de arquitectura relacionadas com estes componentes, são referidos apenas nessa secção.

4

Implementação

Contents

Os mecanismos de checkpoint e migração foram desenvolvidos na máquina virtual JikesRVM (versão 3.1.0). JikesRVM é uma máquina virtual desenhada para executar aplicações Java, cuja característica distintiva comparada com outras máquinas virtuais Java é que está implementada em Java. Contrariamente a outras máquinas virtuais Java, JikesRVM não precisa ser dependente de uma segunda máquina virtual Java para executar-se. JikesRVM é uma máquina virtual autónoma.

Nesta secção, focamo-nos nos detalhes de implementação que fornecem uma melhor compreensão da solução desenvolvida. Em termos gerais, descrevemos como é que são definidos os pontos seguros na máquina virtual JikesRVM e como é que é possível parar a execução de todas as threads respectivas à aplicação. Para além disso, detalhamos também como é que se retira e reconstrói o estado de execução de uma aplicação, tendo em conta a propriedade de consistência.

4.1 Serialização do estado de execução

A máquina virtual JikesRVM suporta serialização através da biblioteca externa `GNU CLASSPATH`.[1] Estes mecanismos de serialização foram modificados de forma a que as classes Java das aplicações não tenham de implementar especificamente o interface `Serializable` (ou outra relacionada com serialização, e.g., `Externalizable`). Sempre que uma aplicação tem classes que implementem essas interfaces, o comportamento da serialização usado é o das próprias classes. Quando não implementam, o comportamento da serialização é generalizado de forma automática para cada classe.

Quando nos encontramos em situações cujas classes/objectos são de difícil serialização (como é o caso de objectos internos à máquina virtual), as soluções utilizadas, são as apresentadas na Secção 3.2.

4.2 Pontos seguros

Para obter um estado consistente da máquina virtual para realizar um checkpoint, é necessário garantir que todas as threads que são respectivas apenas à aplicação estejam paradas. JikesRVM suporta yield points, que são inseridos automaticamente pelo compilador JIT, no inicio e fim de cada método, e também na reiteração dos ciclos (tal como foi abordado na solução MobileJikesRVM [11], no trabalho relacionado, Secção 2.1.3). Estes yield points, são os pontos seguros (definidos na Secção 3.3) onde a máquina virtual pode tomar controlo sobre uma thread de forma a parar a sua execução, isto porque nestes pontos, as threads não estão a alterar o estado da máquina virtual, nem a sua sincronização ”interna”, nem a executar quaisquer instruções (bytecodes) da aplicação.

[1] http://www.gnu.org/software/classpath/

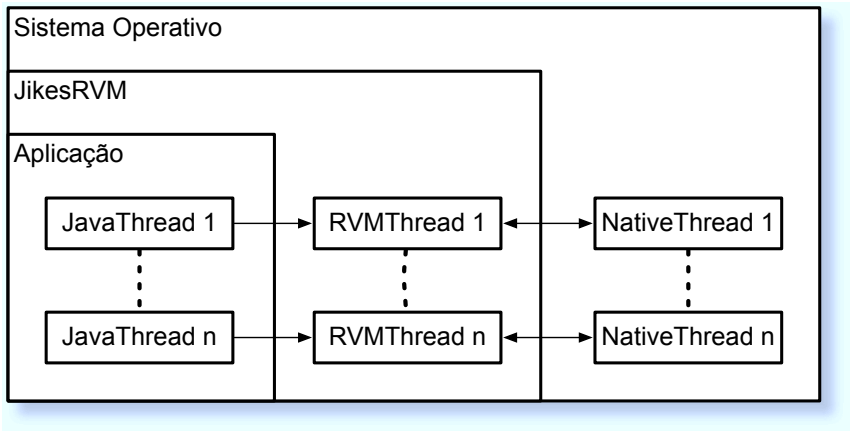

Figura 4.1: Modelo de threading no JikesRVM. Todas as threads Java derivam de RVMThread. Cada RVMThread mapeia directamente com uma thread nativa, que pode estar implementada com qualquer biblioteca de threading (e.g., pthreads [41] ou Apache Harmony threads).

4.3 Suspensão das threads aplicacionais

Para bloquear uma thread (de forma a parar a sua execução), todas as threads que fazem parte dos mecanismos de checkpoint, restore e migração, incluindo também todas as threads internas à máquina virtual (e.g., threads GC)[2] têm uma marca especial. A partir do momento que existe um pedido de checkpoint ou migração, todas as threads que não têm essa marca recebem um sinal, que fica registado num adaptador de bloqueio (criado especificamente para os mecanismos desenvolvidos). Um adaptador de bloqueio regista dois tipos de informação: regista se uma thread tem novos pedidos de bloqueio; e regista se uma thread já está bloqueada por esse adaptador. Existem outros adaptadores de bloqueio (e.g., para o GC, que lhe permite bloquear todas as threads da máquina virtual, de forma a parar a sua execução para efectuar uma reciclagem de memória), e possibilita que uma thread esteja bloqueada em vários adaptadores ao mesmo tempo.

Quando uma thread se encontra num yield point ou numa situação de bloqueio que retornou para a máquina virtual, essa thread executa o método `checkBlock` (listagem de código 4.1). Primeiro verifica se existem novos pedidos em todos os adaptadores de bloqueio existentes. Se houver algum pedido, então o adaptador define em si próprio que a thread tem de bloquear. Quando a thread executa o método `isBlocked`, verifica se está bloqueada em algum adaptador. Se não estiver bloqueada num adaptador, continua a sua execução, caso contrário invoca o método `block`. Esse método tira partido das funcionalidades das threads nativas suportadas (Figura 4.1). Dependendo da biblioteca de threads nativa utilizada, as funções invocadas para bloquear e desbloquear uma thread são respectivamente `sysMonitorWait` e `sysMonitorBroadcast` (implementadas na linguagem de programação C++, listagem de código 4.2).

```
2 void checkBlock()
{
```

[2]Garbage Collector.

```
        for (;;) {
                // verifica se existem novos pedidos de bloqueio
                acknowledgeBlockRequests();

                // bloqueamos num novo pedido?
                if (!isBlocked()) {
                        // não bloqueia, continua execução
                        break;
                }

                // nota: se por exemplo o GC terminar uma limpeza
                // de memória e desbloquear todas as threads,
                // então a thread que está bloqueada no block(),
                // desbloqueia, mas como vai executar novamente o loop,
                // irá automaticamente bloquear-se de novo.
                block(); // invoca método nativo: sysMonitorWait
        }
}

void acknowledgeBlockRequests() {
        for (int i = 0; i < blockAdapters.length; ++i) {
                if (blockAdapters[i].hasBlockRequest(this)) {
                        blockAdapters[i].setBlocked(this, true);
                        blockAdapters[i].clearBlockRequest(this);
                }
        }
}

boolean isBlocked() {
        for (int i = 0; i < blockAdapters.length; ++i) {
                if (blockAdapters[i].isBlocked(this)) {
                        return true;
                }
        }
        return false;
}
```

Listagem 4.1: Algoritmo que verifica se uma thread tem de bloquear.

```
extern "C" void
sysMonitorWait(Word _monitor)
{
#ifdef RVM_FOR_HARMONY
    hythread_monitor_wait((hythread_monitor_t)_monitor);
#else
    vmmonitor_t *monitor = (vmmonitor_t*)_monitor;
    pthread_cond_wait(&monitor->cond, &monitor->mutex);
#endif
}

extern "C" void
sysMonitorBroadcast(Word _monitor)
{
#ifdef RVM_FOR_HARMONY
    hythread_monitor_notify_all((hythread_monitor_t)_monitor);
#else
    vmmonitor_t *monitor = (vmmonitor_t*)_monitor;
    pthread_cond_broadcast(&monitor->cond);
#endif
}
```

Listagem 4.2: Código nativo para bloquear e desbloquear threads.

Adicionalmente, foi necessário garantir que as threads em pontos seguros (yield points), apenas são bloqueadas quando se encontram em yield points de métodos da aplicação. Os yield points que são gerados nos métodos da máquina virtual, não servem para bloquear uma thread da aplicação.

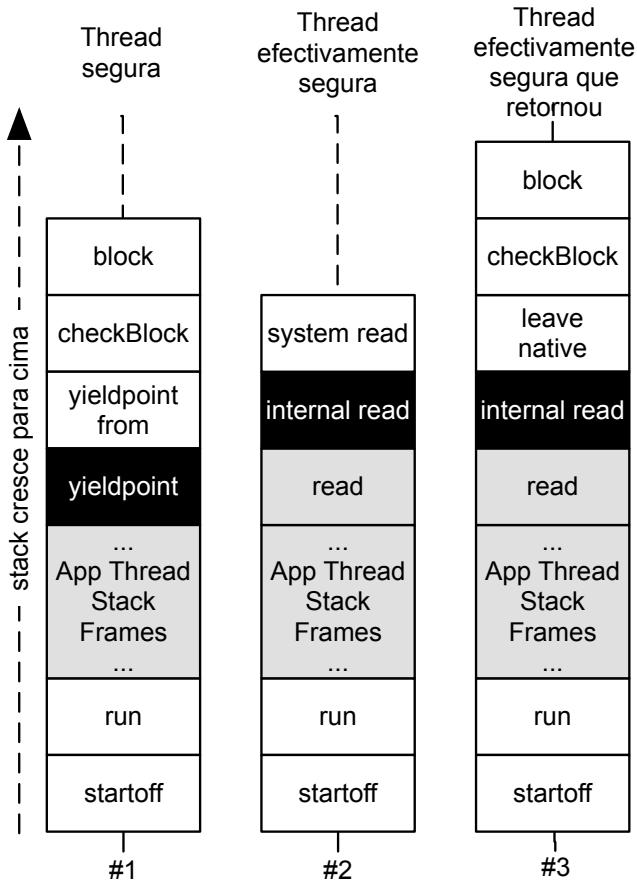

Figura 4.2: Stack frames preservados na JikesRVM para as threads seguras e efectivamente seguras.

Este requisito foi introduzido para garantir que não há perda de controlo nas técnicas que vão ser apresentadas adiante.

4.4 Threads seguras e efectivamente seguras: preservação de stack frames

A Figura 4.2, mostra que stack frames são guardados para cada tipo de thread num pedido de checkpoint ou migração. Mais uma vez, apenas os stack frames com fundo preenchido é que são guardados. Os que estão preenchidos a preto, marcam o primeiro stack frame a ser preservado.

Uma thread num ponto seguro tem sempre o mesmo primeiro stack frame. As threads efectivamente seguras já não se comportam da mesma forma, mas todas as vezes que uma thread está no estado efectivamente seguro, esta passa para estado nativo (código interno à maquina virtual, não é o mesmo que JNI) e, é forçada a guardar os apontadores do FP[3] e IP.[4] O FP guardado marca o primeiro stack frame a ser preservado numa thread efectivamente segura.

Threads efectivamente seguras são sempre recuperadas no mesmo ponto seguro. Se uma th-

[3]Frame pointer, apontador na stack que aponta para o última frame criado.
[4]Instruction pointer, aponta para a próxima instrução a ser executada.

read retorna do seu estado efectivamente seguro enquanto se está a extrair o estado de execução (Figura 4.2, transição da thread #2 para a #3), então no restore vai dar a noção de que nunca avançou na execução (vai ficar exactamente como na thread #2), que é o estado desejado.

4.5 Extracção do estado de execução

Estamos a tirar partido de mecanismos de OSR [16] (on-stack-replacement) para extrair e reconstruir o estado de execução das threads de uma aplicação. OSR torna possível retirar um stack frame de uma stack em execução e substituí-lo por outro.

JikesRVM suporta dois compiladores: baseline e optimizado. Os stack frames do compilador baseline são totalmente observáveis e facilmente extraídos, mas os optimizados não o são. OSR para os métodos optimizados foi construído para dar suporte a optimizações especulativas que possam ser invalidadas em execução (tal como apresentado na solução JavaThread [42], no trabalho relacionado, Secção 2.1.3). Desta forma, o compilador optimizado escolhe pontos no código (pontos OSR, que invocam yield points específicos para este tipo de OSR) que definem quando é que OSR pode ocorrer neste tipo de métodos (são pontos importantes onde são criados os mapeamentos com a informação optimizada, denominados por OSR maps). Como estes mapeamentos não estão sempre disponíveis, a nossa solução por ora, apenas suporta extrair informação de métodos compilados com o compilador baseline.

Numa versão mais evoluída da MobileJikesRVM em [35], a extracção de métodos optimizados foi suportada. Essencialmente, tentaram criar OSR maps para todos os métodos da aplicação, mas como resultado, tiveram de desactivar algumas optimizações do compilador optimizado, isto porque, houve problemas em manter OSR maps para certos níveis de optimização. O resultados de desempenho mostraram que o conjunto de optimizações que foram eliminadas não reduziam significativamente o desempenho do sistema, mas em termos de correcção, tinha alguns problemas, sendo que só funcionava apenas em 60% dos casos. Este tipo de abordagem para a nossa solução tem ainda outros desafios, isto porque, para além de trabalharmos os métodos respectivos apenas às aplicações, existem também um conjunto de métodos preservados que pertencem à maquina virtual em si. Esta delineação teria de ser mais estudada e aprofundada. Portanto, ainda é preciso realizar alguma investigação para dar suporte a métodos optimizados (para além da solução apresentada, tentar explorar outras soluções e compará-las, e.g., tentar transportar código JIT'ed).

De forma a entender como é que o estado de um stack frame pode ser extraído quando compilado pelo compilador baseline, primeiro é necessário analisar a estrutura de um stack frame na JikesRVM (Figura 4.3). Cada stack frame é representado por um array de slots, formalmente declarados como inteiros, que guardam informação de dados primitivos (byte, int, float, entre outros), um apontador para um objecto, um apontador para código máquina (e.g., apontador para um endereço de retorno), ou um apontador para outro slot na mesma stack (e.g., um apontador para outro frame). Da informação apresentada na Figura 4.3, a única informação que está em falta, são os

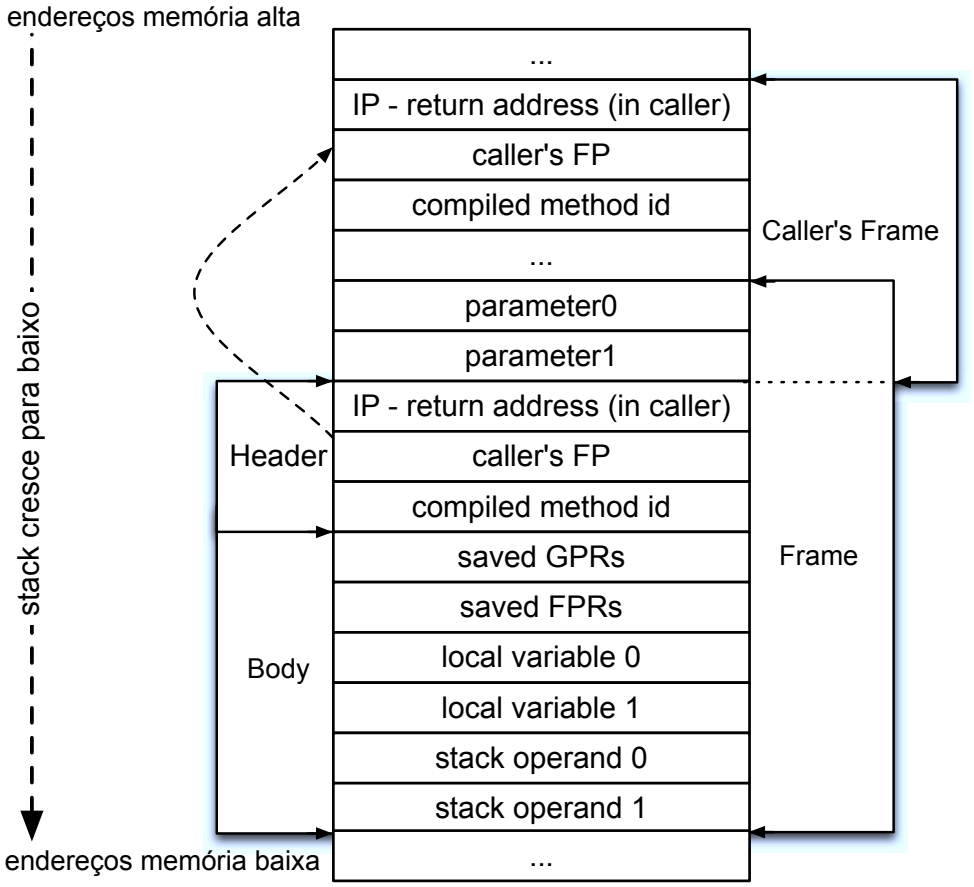

Figura 4.3: Layout de um stack frame na JikesRVM.

tipos das variáveis locais e resultados intermédios (operandos). Essa informação não é mantida pela máquina virtual, tal como já abordamos este problema em outras soluções no trabalho relacionado (e.g., Merpati [42] e JavaThread [8]).

A extracção do estado de execução para métodos baseline é apresentada na Listagem 4.3 (método extractState). Primeiro, é preciso analisar o bytecode de forma a determinar os tipos das variáveis locais e operandos para uma determinada posição do bytecode, tal e qual como um verificador de bytecode comum. O resultado produzido tem de ser ajustado com os mapeamentos do GC, isto porque podem existir variáveis (referências) que eventualmente não estão ainda inicializadas numa determinada posição do bytecode. Adicionalmente, o número das variáveis locais e operandos são contados. Quando os resultados estiverem disponíveis, o estado de um frame é totalmente extraído usando ambos os tipo/número das variáveis. A estrutura da informação retirada é a seguinte (representada na classe `StackFrameExecutionState`):[5]

- Variáveis locais e operandos (incluí os argumentos e referência para o objecto this do método).
- IP (bytecode) nesse método.

[5] A classe `StackFrameExecutionState` é a representação intermédia em Java de um stack frame, tal como foi abordado na arquitectura, Secção 3.1.

- Tipo de compilador (baseline ou optimizado).
- Nome do método (composto por class/descriptor/método (e.g., mypackage.myclass / (I)V /mymethod)).
- Estado de execução do próximo stack frame.

```
// osrFPoff = apontador na stack que aponta para o frame invocado pelo método
// que queremos guardar
// methFPoff = aponta para o frame do próprio método
public StackFrameExecutionState extractState(
        RVMThread thread, Offset osrFPoff, Offset methFPoff, int cmid) {

        byte[] stack = thread.getStack();

        BaselineCompiledMethod fooCM =
                (BaselineCompiledMethod) CompiledMethods.getCompiledMethod(cmid);

        NormalMethod fooM = (NormalMethod) fooCM.getMethod();

        VM.disableGC();
        Address rowIP = Magic.objectAsAddress(stack).loadAddress(
                osrFPoff.plus(STACKFRAME_RETURN_ADDRESS_OFFSET));
        Offset ipOffset = fooCM.getInstructionOffset(rowIP);
        VM.enableGC();

        int bcIndex = -1;

        bcIndex = fooCM.findBytecodeIndexForInstruction(ipOffset);

        // Objecto que guarda o estado de execução
        StackFrameExecutionState state =
                new StackFrameExecutionState(thread, fooM, 'B', cmid, bcIndex);

        // temos que extrair os valores das variáveis locais e operandos,
        // mas primeiro temos de tirar informação do tipo dessas variáveis,
        // para o PC corrente
        BytecodeTraverser typer = new BytecodeTraverser();
        typer.computeLocalStackTypes(fooM, bcIndex);
        byte[] localTypes = typer.getLocalTypes();
        byte[] stackTypes = typer.getStackTypes();

        // consultar os mapeamentos GC de novo, isto porque a informação dos
        // tipos não está completa. A análise de fluxo não consegue distinguir os
        // tipos ref vs non-ref.
        for (int i = 0, n = localTypes.length; i < n; i++) {
                // se um tipo reporta que é uma referência,
                // mas o GC diz que não
                // então alteramos o tipo para 'uninitialized'
                // para mais informação, ver especificação da VM,
                // verificador bytecode
                if (localTypes[i] == ClassTypeCode) {
                        if (!fooCM.referenceMaps.isLocalRefType(
                                fooM, ipOffset.plus(
                                        1 << LG_INSTRUCTION_WIDTH), i)) {
                                localTypes[i] = VoidTypeCode;
                        }
                }
        }
        // percorrer o stack frame e os valores extraídos,
        // para manter a seguinte ordem L0, L1, ..., S0, S1, ...
        // ajustar os apontadores das variáveis locais e operandos
        Offset startLocalOffset = methFPoff.plus(
                BaselineCompilerImpl.locationToOffset(
                        fooCM.getGeneralLocalLocation(0)));

        Offset stackOffset = methFPoff.plus(fooCM.getEmptyStackOffset());
```

```
61
        getVariableValue(stack, startLocalOffset, localTypes, fooCM, LOCAL, state);
63      getVariableValue(stack, stackOffset, stackTypes, fooCM, STACK, state);

65      return state;
}
```

Listagem 4.3: Extracção do estado de execução de métodos baseline.

Grande parte do algoritmo que é usado para retirar o estado de execução de métodos baseline, inspira-se muito em problemas relacionados com type inference. Isto apenas é realizado quando é necessário extrair o estado de um método (ou de uma thread, que irá extrair o estado de todos os seus stack frames). Desta forma, não é esperado que este componente crie penalizações de desempenho durante a execução das aplicações.

Quando todos os stack frames forem extraídos (componente interna state extraction, Capítulo 3), essa informação fica registada no objecto `java.lang.Thread` de cada thread. A componente interna checkpoint (arquitectura, Secção 3) quando activada, percorre todas as threads correspondentes (apenas as da aplicação em si, como já foi referido), e serializa todos os objectos java.lang.Thread, preservando: o estado de execução (lista de `StackFrameExecutionState`'s), nome da thread, se é uma thread daemon, e os locks que que lhe pertencem.

Finalmente, é importante referir que o GC deve estar sempre activo nos processos de extracção e recuperação do estado de execução. Numa fase inicial deste trabalho as threads do GC eram bloqueadas, mas para aplicações muito grandes, o GC era obrigado a intervir (o que fazia com que fossem geradas situações de bloqueio de toda a máquina virtual). Convém que haja de vez em quando uma reciclagem dos objectos que vão sendo criados.

4.6 Recuperação do estado de execução

No restore, todos os objectos java.lang.Thread são recriados com a informação preservada, mas é necessário ainda recuperar o estado de execução guardado (para recriar todas as threads, tal como se encontravam anteriormente).

A recuperação do estado de execução (componente interna state restoration, Capítulo 3) de todas as threads da aplicação é efectuada de forma faseada, para evitar que hajam situações de inconsistência (todas as threads recuperam o seu estado em paralelo). Na arquitectura apresentada no Capítulo 3, já foram abordadas de forma geral as fases que englobam recuperar o estado indicado. Nesta secção abordamos com mais detalhe as várias fases (em cada fase, as threads sincronizam-se entre si):

- Reconstruição dos stack frames: a representação intermédia do estado de execução de cada stack frame (classe `StackFrameExecutionState`), é recompilada com um prólogo especial/adicional e, é gerada uma nova versão compilada do método (stack frame) correspondente (de notar que pode haver várias versões compiladas de um método).

- Recuperação dos locks: os locks que cada thread tinha são readquiridos.
- Re-executação dos stack frames: para criar a execução que existia anteriormente.

De seguida abordamos os detalhes de implementação ligados à reconstrução e re-execução dos stack frames respectivos a cada thread, nas suas secções correspondentes. A recuperação dos locks é auto-explicativa e é uma fase do mecanismo de restore que não tem muitos detalhes significantes.

4.6.1 Reconstrução dos stack frames

A reconstrução dos stack frames de cada thread, implica recompilar o método de cada stack frame com um prólogo adicional, que tem um conjunto de instruções bytecode adicionais com a responsabilidade de:

- Recuperar as variáveis locais e operandos.
- Se existia um método na call stack da thread que estava a ser invocado por esta stack frame (quando foi guardado o estado de execução), então invocamo-lo de novo.
- E finalmente depois disso, recuperar o IP (bytecode) preservado.

Na listagem de código 4.4 mostramos uma aplicação exemplo que demonstra a recuperação do estado de execução de um stack frame. O método main da aplicação PrologueExample (LPrologueExample;.main ([Ljava/lang/String;)V), foi o escolhido para a demonstração. Quando esse método foi extraído da stack, o seu estado de execução éra o seguinte:

- Variáveis locais: L0 -> REF [Ljava.lang.String; L1 -> Int 16; L2 -> Float 17.
- Variáveis operando: S0 -> Int 9.
- IP (bytecode): 10.
- Tipo de compilador: baseline.
- Nome do método: LPrologueExample;.main ([Ljava/lang/String;)V.
- Estado de execução do próximo stack frame: correspondente ao método test, (I)V.

```
public class PrologueExample {
2
        public static int number()
4       {
                int a = 9;
6               return a;
        }
8
        public static void test(int a) throws InterruptedException
10      {
                Thread.sleep(a * 1000);
12      }

14      public static void main(String args[]) throws InterruptedException {
```

```
                int x = 16;
16              long y = 17;

18              test(number());
        }
20
}
```

Listagem 4.4: Aplicação exemplo para demonstrar recuperação do estado de execução de um stack frame.

O bytecode inicialmente gerado pelo compilador para o método main é apresentado na Listagem 4.5. Quando esse método é recuperado, o seu bytecode com o prólogo adicional é apresentado na Listagem 4.6. Tal como já foi explicado anteriormente, primeiro recuperaram-se as variáveis (locais e operandos) do método, de seguida como havia um método que tinha de ser invocado para reconstruir a stack anterior, é invocado com o bytecode pseudo_invoke_cmid (o número 22048 corresponde ao método compilado test que também já foi compilado com o seu prólogo especial). Quando a execução do método test retornar, efectua-se um salto para a posição bytecode seguinte à invocação original desse método (o IP preservado aponta para a instrução que se estava a executar, é preciso incrementá-lo) , para que o código bytecode da aplicação seja executado a partir da posição correcta.

```
1 [0] bipush 16
  [2] istore_1
3 [3] ldc2_w 3
  [6] lstore_2
5 [7] invokestatic -1 < SystemAppCL, LPrologueExample;, number, ()I >
  [10] invokestatic -1 < SystemAppCL, LPrologueExample;, test, (I)V >
7 [13] return
```

Listagem 4.5: Bytecode correspondente ao método LPrologueExample;.main ([Ljava/lang/String;)V.

```
1 // Prologue start
  [0] nop
3 [1] nop
  [2] nop
5 [3] pseudo_load_int 2
  [9] pseudo_load_int 0
7 [15] pseudo_invokestatic < BootstrapCL,
        Lorg/jikesrvm/osr/ObjectHolder; >.getRefAt (II)Ljava/lang/Object;
9 [21] astore 0

11 // inicia recuperação das variáveis
  [23] pseudo_paraminitend
13 [25] pseudo_load_int 16
  [31] istore 1
15 [33] pseudo_load_long 17
  [43] lstore 2
17 [45] pseudo_load_int 9
  [51] pseudo_load_int 2
19 [57] pseudo_invokestatic < BootstrapCL,
        Lorg/jikesrvm/osr/ObjectHolder; >.cleanRefs (I)V
21
  // invoca método compilado test
23 [63] pseudo_invoke_cmid 22048

25 // produz um salto para o bytecode 89
  [73] goto 16 [89]
27 // Prologue end
```

```
29 [78] istore_1
   [79] ldc2_w 3
31 [82] lstore_2
   [83] invokestatic 86 < SystemAppCL, LPrologueExample;, number, ()I >
33 [86] invokestatic 86 < SystemAppCL, LPrologueExample;, test, (I)V >
   [89] return
```

Listagem 4.6: Bytecode (com prólogo adicional que recupera o estado de um stack frame) correspondente ao método LPrologueExample;.main ([Ljava/lang/String;)V.

Ainda relativamente ao prólogo adicional que é gerado para os stack frames na sua recuperação, é importante salientar o seguinte. Apenas é possível utilizar o bytecode `pseudo_invoke_cmid` quando existe uma invocação que é necessária realizar sobre um stack frame que foi anteriormente preservado (para reconstruir a stack que existia) e, quando o IP preservado no estado de execução de uma thread aponta para uma invocação `invokestatic`, `invokevirtual` ou `invokespecial`. A segunda parte é problemática, isto porque existem situações exemplo em que o IP não aponta para esse tipo de instruções (e.g., nos yield points). Um yield point é gerado pela máquina virtual e não fica explicitamente no código bytecode de um método. Sempre que o IP de uma thread aponta para uma instrução que não seja uma das indicadas e, houver algum método que tenha de ser invocado para reconstruir a stack anterior, então a invocação respectiva é forçada e, o IP não é incrementado, isto porque o yield point (ou outra instrução do tipo) é invocado antes de executar a instrução apontada pelo IP.

4.6.2 Re-execução dos stack frames

Quando todos os métodos forem recompilados com os seus prólogos adicionais, uma nova thread é recriada para cada objecto `java.lang.Thread` salvaguardado e, re-executa todos os seus stack frames compilados de forma a ficarem no mesmo estado de execução que estavam anteriormente e, quando esse processo terminar, a thread automaticamente bloqueia-se, para tornar o restore consistente. Quando todas as threads estiverem disponíveis, a máquina virtual pode reiniciar o estado de execução retirado no checkpoint.

O prólogo adicional que foi gerado em cada stack frame, interliga os stack frames de uma thread através do bytecode `pseudo_invoke_cmid`. O problema é que a nova thread gerada (por cada thread que foi salvaguardada), não tem qualquer ligação com esses stack frames (que até agora são apenas métodos compilados, desligados entre si, porque a ligação entre eles só se realizará quando forem executados). De forma a poder invocar o primeiro método compilado, que irá corresponder ao primeiro stack frame preservado na call stack da thread, aproveitamo-nos das capacidades oferecidas pelo `reflection`. Com `reflection` é possível invocar um método pelo seu nome, mas não suporta invocar métodos que já estão compilados na máquina virtual, isto porque parte-se do principio que esses métodos já estão a ser executados, e como tal já foram invocados. Basicamente, usamos uma versão modificada desse mecanismo. Essencialmente, para invocar um método compilado desta forma, é preciso preencher os parâmetros de um método compilado com valores predefinidos (e.g., para inteiros o valor 0, para referências o valor `null`, entre outros). Esses

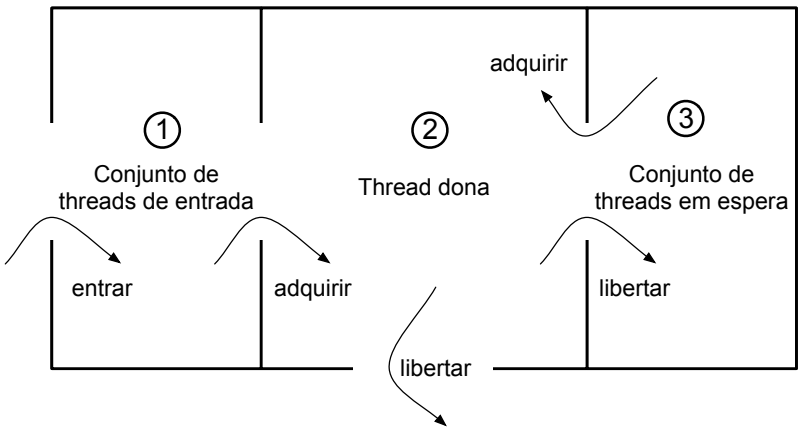

Figura 4.4: Um monitor Java.

parâmetros nunca são utilizados, isto porque, quando um método compilado é re-executado, o seu prólogo adicional irá substituir todas as variáveis dos parâmetros pelos valores preservados, sendo este o resultado pretendido.

4.7 Estado de execução para sincronização

Uma thread para efeitos de sincronização (e.g., blocos de código `synchronized` ou métodos `synchronized`), encontra-se sempre num dos seguintes três estados possíveis (Figura 4.4):

1. Bloqueada no conjunto de threads de entrada do monitor (`locked`);
2. Em execução, sendo a dona do lock;
3. Bloqueadas no monitor à espera de notificação (`wait`).

A implementação dos mecanismos de locking na máquina virtual JikesRVM, está optimizada para as várias combinações de situações que podem ocorrer para sincronização:

- Locks que são adquiridos apenas por uma thread (*biased locking*).
- Locks adquiridos por múltiplas threads mas que raramente estão a ser usados (isto é, não estão a ser alvo de competição) por mais do que uma thread ao mesmo tempo (*thin locking*). A implementação destes locks é relativamente simples. São usados 20 bits no header do objecto para representar o estado actual do lock.
- Locks adquiridos por múltiplas threads e que estão ao mesmo tempo a ser usados (estando a ser alvo de competição) por mais do que uma thread (*contended locks*). Estes locks são implementados com estruturas de dados mais complexas e tiram partido dos seguintes mecanismos:
  - Um mecanismo de lock para proteger as próprias estruturas de dados.

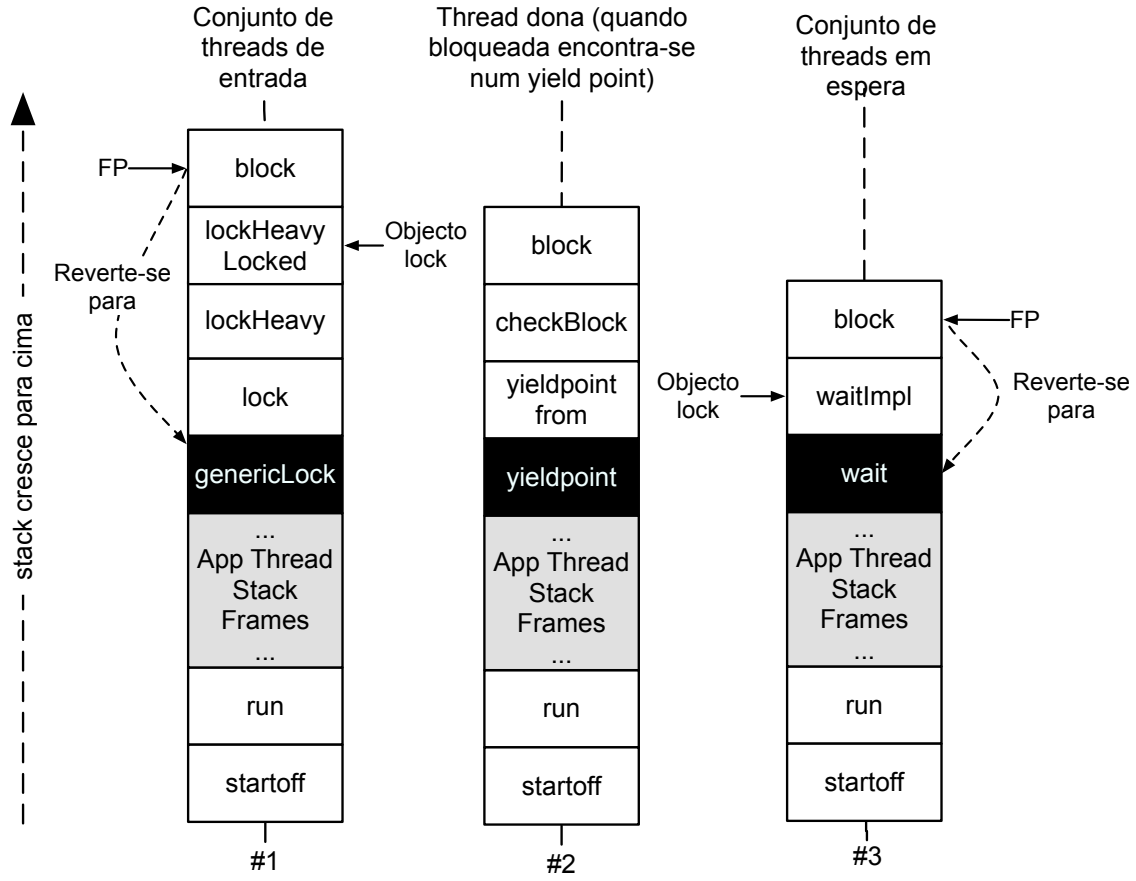

Figura 4.5: Stacks exemplo para os três estados possíveis de uma thread quando se encontra em sincronização.

- Uma lista de threads bloqueadas no lock.
- Um mecanismo para as threads bloqueadas (no conjunto de entrada ou em espera), para adormecerem até serem de novo acordadas, quando retiradas da lista de threads bloqueadas (no seu conjunto respectivo). Este mecanismo de bloqueio é idêntico ao mecanismo apresentado na listagem de código 4.2.

Como os locks usados para sincronização de threads, podem estar em diferentes estados, e cada estado tem a sua implementação específica (sendo que contended locks, podem obrigar que uma thread fique bloqueada em mecanismos oferecidos pelo sistema operativo, e.g., `pthread_cond`), a solução proposta evita guardar esses locks de sincronização internos.

No restore, esses locks terão obviamente de ser recriados e readquiridos. A thread que detém o controlo sobre o lock (que é dona do lock), no restore readquire-o de novo (Secção 4.6, estado de execução: recuperação, segunda fase do processo). Durante a execução da aplicação todos os locks que uma thread é dona são mantidos numa estrutura de dados adicional.

As threads que estão bloqueadas no conjunto de entrada ou em espera, as suas stacks têm presentes os objectos que representam os locks que evitamos preservar. Como tal, revertemos

as stacks respectivas para a posição de um stack frame onde esses locks ainda não tenham sido criados. Isto apenas funciona, porque estamos perante situações deterministas, em que no restore os locks vão ser recriados de novo, e obrigam a que as threads se bloqueiem nos conjuntos (entrada ou espera) correctos. Para melhor compreensão do que foi dito, apresentamos na Figura 4.5, um conjunto de stacks de execução exemplo, na presença de um checkpoint, para os três estados em que uma thread pode ser encontrada por motivos de sincronização (com os respectivos stack frames que são preservados em cada caso, com fundo preenchido).

É verdade que no restore, a ordem pela qual as threads irão ser bloqueadas pode ser diferente daquela que foram bloqueadas no estado anterior, mas a competição entre threads por motivos de sincronização, é muito dependente da implementação que cada máquina virtual fornece e até do escalonamento (*scheduling*) do sistema operativo, e por isso, a ordem pela qual as threads são novamente bloqueadas não foi vista como um requisito (para além de que uma solução que dependa da velocidade da execução das threads para a correcção de uma aplicação, normalmente é vista como má programação). No entanto, cada thread ficará bloqueada no conjunto (fila) correcto e a thread que detinha o lock mantê-lo-á.

4.8 Checkpoint, restore e migração: API suportada

A API fornecida para controlar os mecanismos de checkpoint ou migração foi simplificada e é apresentada na Listagem 4.7. A API para checkpoint recebe como argumento o ficheiro para onde se pretende guardar o checkpoint. A API para migração recebe como argumentos o endereço e o porto destino, para onde irá ser enviado o estado de execução. Para poder invocar essas APIs, é necessário adicionar a classe `VMcrm` ao `CLASSPATH` da aplicação quando compilada (como forma de ligação).

```
public class VMcrm
{
  public static synchronized boolean checkpointVM(String checkpointFile)
  {
        // invoca serviço interno que faz checkpoint e retorna o resultado
  }

  public static synchronized boolean migrateVM(String host, int port)
  {
        // invoca serviço interno que faz migração e retorna o resultado
  }
}
```

Listagem 4.7: API fornecida para os mecanismos de checkpoint e migração.

Para lançar uma aplicação com um checkpoint, a máquina virtual tem um parâmetro especial que recebe como entrada o ficheiro do checkpoint correspondente, como demonstrado de seguida: JikesRVM -X:checkpoint='ficheiro'.

Um detalhe que é importante retratar é o seguinte. A thread que está a salvaguardar o estado de execução da aplicação é a própria thread que faz as invocações `checkpointVM` ou `migraçãoVM` respectivas. Essa thread também está naturalmente em execução, e nem sequer está num estado

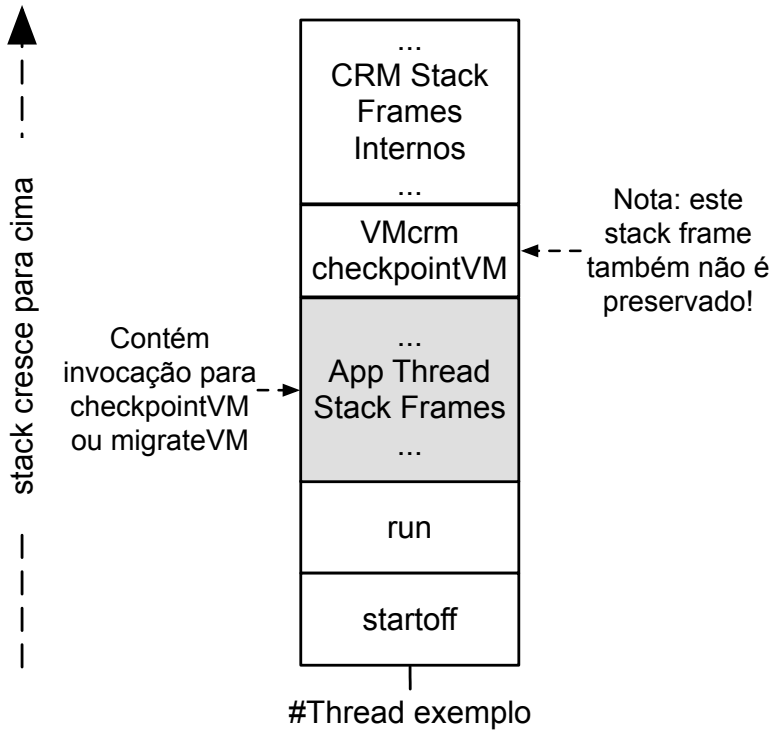

Figura 4.6: Stack exemplo para uma thread que invoca mecanismos de checkpoint ou migração através da API fornecida.

efectivamente seguro. Portanto, este problema teve de ser trabalho de forma especial. Primeiro é preciso determinar se uma thread se incluí nesta situação. Se for verdade, então antes de invocar os métodos respectivos para efectuar um checkpoint ou migração, é obrigada a guardar o seu FP e IP, tal como numa thread efectivamente segura. O FP preservado é o primeiro stack frame a ser guardado. A Figura 4.6, demonstra os stack frames preservados nesta situação especial.

No restore, há que ter em atenção que o prólogo especial que é gerado para o método que invoca `checkpointVM` ou `migrateVM`, posiciona o IP para a posição da invocação respectiva. Isto irá fazer com que o checkpoint ou a migração seja realizada novamente, que não é o pretendido. Para evitar este problema, sempre que uma thread é recuperada nesta situação, é obrigada a saltar a instrução das invocações `checkpointVM` ou `migrateVM` quando recupera o IP.

4.9 Estado de execução externo

O trabalho relacionado com o estado externo, ainda é trabalho em curso. No entanto, algum trabalho já se encontra implementado. Neste momento, é possível preservar a ligação que existe entre a aplicação e o estado externo dos ficheiros (inclui descritor e cursor dentro de um ficheiro). A ligação dos sockets tem questões secundárias, mas a abordagem principal adoptada é muito idêntica à dos ficheiros. O conteúdo dos ficheiros, tanto para classes Java da aplicação, como para os ficheiros que uma aplicação lê ou escreve, também é trabalho em curso. De seguida, abordamos

estas questões na sua secção respectiva.

4.9.1 Ligações com o estado externo dos ficheiros

No Java é possível interagir com ficheiros de várias formas, mas numa perspectiva geral, existem apenas um conjunto de classes (não nativas) que interagem directamente com os ficheiros (e.g, `InputStream`'s ou `OutputStream`'s), e as restantes aproveitam-se dessas para optimizar as leituras e escritas dos ficheiros (e.g., `Buffers`). O `InputStream` que é sempre usado para ler de um ficheiro é o `FileInputStream` e o `OutputStream` para escrever é o `FileOutputStream` (existem outras abstracções, mas mesmo essas criam automaticamente uma interacção com o `FileInputStream` ou `FileOutputStream`). Ambas as classes `FileInputStream` e `FileOutputStream` são as únicas que têm ligações com classes que têm dependências nativas (mais precisamente com o descritor do ficheiro correspondente), as restantes são apenas classes Java para optimização (e.g., os buffers que são normalmente usados, lêem ou escrevem grandes quantidades de informação, e servem de cache à aplicação, para não ter de estar constantemente a aceder ao disco rígido) e, estas são facilmente serializáveis, ao contrário das classes `FileInputStream` e `FileOutputStream`.

Quando existem ligações com o estado externo, como é o caso dos ficheiros, duas situações podem ocorrer:

- Ou estamos numa situação em que existe a ligação, mas não está em uso, ou seja não se está a ler ou a escrever nesse momento (neste caso a thread encontra-se parada num ponto seguro, quando no decurso de um checkpoint).
- Ou estamos numa situação em que existe a ligação e, nesse momento está em curso uma leitura ou escrita para um ficheiro (ou seja encontra-se num ponto efectivamente seguro).

Em ambas as situações, a serialização dos objectos `FileInputStream` e `FileOutputStream` tem de ser tratada de forma especial. Na listagem de código 4.8, demonstramos como exemplo a serialização do objecto `FileInputStream`. Essencialmente guardamos o mínimo de informação que nos possibilita recuperar este objecto: localização com o nome e cursor do ficheiro. De notar que a localização do ficheiro tem de ser relativa à aplicação.

```
public class FileInputStream extends InputStream implements Serializable
{
  // representação nativa do descritor
  transient private FileDescriptor fd;

  transient private FileChannelImpl ch;

  //...

  private void writeObject(ObjectOutputStream out) throws IOException
  {
        // guardamos de forma geral toda a informação
        // que não for nativa
        out.defaultWriteObject();

        // tratamento especial, o path e cursor do ficheiro
        String filePath = ch.getPath();
```

```
18          long fileCursor = ch.getSavedPosition();

20          out.writeObject(filePath);
            out.writeLong(fileCursor);
22          out.flush();
    }
24
    private void readObject(ObjectInputStream in) throws IOException,
26          ClassNotFoundException
    {
28          // lemos toda a informação que não é nativa
            in.defaultReadObject();
30
            // recupera-se o path e o cursor
32          String filePath = (String) in.readObject();
            long fileCursor = in.readLong();
34
            File file = new File(filePath);
36
            SecurityManager s = System.getSecurityManager();
38          if (s != null)
                    s.checkRead(file.getPath());
40           try {
                        // abrimos descriptor de ficheiro
42                  ch = FileChannelImpl.create(file, FileChannelImpl.READ);
             } catch (FileNotFoundException fnfe) {
44                  throw fnfe;
             } catch (IOException ioe) {
46                  FileNotFoundException fnfe = new FileNotFoundException(
                            file.getPath());
48                  fnfe.initCause(ioe);
                    throw fnfe;
50           }

52          // posiciona-se o cursor do ficheiro para a posição correcta
             ch.position(fileCursor);
54  }
```

Listagem 4.8: Serialização do objecto FileInputStream.

Numa situação em que se está num estado efectivamente seguro (e.g., a ler de um ficheiro), a stack de uma thread tem referências para os objectos `FileChannelImpl` e `FileDescriptor`. De forma a evitar que tenhamos de guardar esses objectos no estado de execução de uma thread, a stack dessa thread é revertida para a posição do stack frame do método da aplicação que iniciou a leitura do ficheiro. Desta forma a stack da thread, no pior caso apenas tem presente a referência para o objecto `FileInputStream` (do conjunto de objectos não serializáveis), cuja serialização foi apresentada. No restore, mais uma vez vai parecer que uma leitura ou escrita que tenha ocorrido durante um checkpoint ou migração, nunca ocorreu, e como tal vai ser repetida novamente (sem problemas de inconsistências).

Ainda relativamente a este tipo de situação, é preciso garantir que o cursor do ficheiro é guardado na posição correcta, isto porque, como estamos a reverter a stack, pode acontecer que essa thread efectivamente segura, entretanto retorne algum resultado para a máquina virtual (mas como já vimos, automaticamente bloquear-se-á), o que faz com que o cursor já tenha sido alterado entretanto. Por causa deste problema, o cursor está sempre a ser guardado antes e depois de serem invocadas leituras ou escritas nativas sobre ficheiros.

4.9.2 Conteúdo dos ficheiros

Neste momento, estamos a trabalhar duas soluções para transportar o conteúdo dos ficheiros no estado de execução de uma aplicação (baseadas nas soluções de transporte de informação ao nível de processo abordadas no trabalho relacionado, Secção 2.1.1):

- Uma solução eager all: todos os ficheiros (inclui também ficheiros das classes da aplicação) são completamente transportados em conjunto com o estado de execução interno.

- Uma solução copy-on-reference com algumas modificações: os ficheiros só são transportados quando forem referenciados.

Uma solução eager all tem alguns desafios que têm de ser resolvidos. O primeiro desafio está associado com a identificação dos ficheiros que têm de ser transportados. A máquina virtual não sabe que ficheiros pertencem a uma aplicação, e no máximo, só é possível saber aqueles que foram abertos ao longo do tempo, e aqueles que ainda estão abertos. Os que eventualmente podem ser abertos no futuro é uma incógnita. Para resolver esta questão, simplificámos o problema com uma solução pessimista. Todos os ficheiros que estiverem presentes na directoria da aplicação, são comprimidos num único ficheiro (inclui subdirectorias). É verdade que, mesmo com esta solução pessimista, ainda pode haver ficheiros que são ser abertos no futuro e não são apanhados por esta solução. Para resolver este problema, o utilizador que usufrui dos mecanismos de checkpoint e migração, pode parametrizar ficheiros adicionais que têm de ser tidos em conta, num ficheiro de configuração auxiliar. Nesta última abordagem, pode-se já estar a comprometer a propriedade de transparência, em que no pior caso temos uma solução apenas transparente para o programador.

O segundo desafio está relacionado com ficheiros com dimensões grandes. Os mecanismos de checkpoint e migração quando trabalharem ficheiros com dimensões elevadas podem atingir o limite de memória existente. Para resolver este problema é preciso realizar o transporte do estado externo dos ficheiros de forma faseada, enviando nas várias fases blocos de informação mais pequenos.

Uma solução copy-on-reference é mais sofisticada e transporta os ficheiros apenas quando são referenciados. Portanto, identificar os ficheiros que têm de ser transportados é um problema que está automaticamente resolvido. Porém todos os ficheiros são transportados de forma completa (e para ficheiros muito grandes, pode ser dispendioso, até porque podemos querer apenas interagir com uma porção muito pequena de um ficheiro). Por esta razão, sugerimos uma versão modificada do copy-on-reference. As leituras sobre ficheiros são sempre efectuadas sobre o nó inicial, com optimizações usando caching de blocos dos ficheiros. As escritas, também são sempre enviadas para o nó inicial, para evitar algumas situações: primeiro para evitar que tenhamos de transportar o conteúdo do ficheiro que está a ser escrito; e segundo, para evitar que o conteúdo fique disperso entre os vários nós entre os quais a aplicação pode ser transportada.

4.9.3 Sockets

Na secção 4.9.1, retratámos como é que é possível trabalhar o estado externo das ligações com os ficheiros. Essa solução também pode ser aplicada às ligações que existem com os sockets (mas neste caso preserva-se informação relativa a cada socket, e.g., endereço e porto destino). Contudo, existem outras questões relacionadas com sockets, que dificultam a recuperação dos mesmos:

- Mensagens perdidas: num checkpoint ou migração, pode estar em curso uma leitura de um socket. No restore, essa leitura é como se nunca tivesse ocorrido, e como já foi visto, vai ser repetida novamente. Se as aplicações que estão em comunicação não estiverem preparadas para este tipo de corte na ligação, existe estado que pode ficar perdido e que nunca irá ser recuperado. Essencialmente só temos controlo sobre a aplicação onde estão a ser activados os mecanismos e, se a outra aplicação que enviou a informação não a repetir de novo, é gerado um problema.

- Mudança de rede: se uma aplicação mudar para outro ambiente de rede local, o endereço socket preservado vai deixar de fazer sentido.

As questões apresentadas não têm soluções simples, e grande parte deixam de fazer sentido no contexto da solução inicialmente proposta. Por estas razões, o suporte de sockets no checkpoint ou migração de uma aplicação ainda está numa fase de desenvolvimento muito inicial.

Resumo

Nesta secção abordámos questões relacionadas com detalhes de implementação ligados à solução desenvolvida.

Para bloquear uma thread num ponto seguro, tirámos partido dos yield points suportados pela máquina virtual JikesRVM. De forma a parar a execução das threads da aplicação nesses pontos seguros, invocamos mecanismos nativos, das bibliotecas de threads nativas suportadas (e.g., pthreads ou Apache Harmony threads).

A extracção e recuperação das threads da aplicação foram os mecanismos que foram retratados em maior detalhe. Para extrair uma thread da aplicação, foi preciso analisar todos os stack frames correspondentes, para retirar informação das variáveis locais, operandos, a posição actual no bytecode, entre outros. Adicionalmente foi também necessário inferir os tipos das variáveis, isto porque, este tipo de informação não está presente na máquina virtual JikesRVM. A recuperação de todas as threads da aplicação é realizada de forma faseada. Primeiro reconstruímos todos os stack frames, sendo que o resultado gerado é um conjunto de métodos compilados em memória, que quando executados reconstroem o seu estado de execução anterior. De seguida cada thread readquire todos os locks de sincronização que detinha anteriormente. No fim, uma nova thread é gerada para cada thread que foi preservada, e re-executa todos os métodos compilados pela ordem

que tinham sido anteriormente, sendo apenas nesta fase que a reconstrução dos stack frames é realizada. Depois destes passos, as threads são automaticamente bloqueadas, para que o restore dessas threads seja efectuado ao mesmo tempo.

Relativamente ao estado de sincronização entre várias threads, tem de ser revertido, pelo facto de ser estado que não é facilmente guardado. No restore, como estamos perante situações deterministas, as threads que se encontravam bloqueadas, irão bloquear-se nos mesmos conjuntos (`blocked` ou `wait`), sem problemas de consistência.

A API suportada é muito simples e, sempre que uma thread da aplicação realiza um checkpoint ou migração, essa thread não se encontra num estado seguro, nem efectivamente seguro, e por isso é necessário ter alguns cuidados adicionais.

Finalmente, relativamente ao trabalho relacionado com o estado externo, ainda é trabalho em curso, mas a solução desenvolvida já consegue guardar as ligações com o estado externo dos ficheiros e dos sockets, mesmo quando essas ligações estão a ser utilizadas. Essencialmente, para evitarmos guardar estado interno, muito possivelmente nativo, relativo a essas ligações, tal como na solução para o estado de sincronização, revertemos as threads para uma posição que é mais facilmente controlável. A informação nativa da ligação com ficheiros preservada é caracterizada basicamente pela localização com o nome do ficheiro e o cursor dentro do ficheiro. Para os sockets são preservados, o endereço e porto local/destino, consoante o tipo de socket.

Em relação ao conteúdo dos ficheiros, sugerimos como abordagem ao problema as seguintes soluções: i) uma solução eager all, onde todos os ficheiros são completamente transportados em conjunto com o estado de execução interno (esta solução tem sempre um problema que é saber que ficheiros é que realmente pertencem a uma aplicação). A solução pode ser pessimista para tentar aglomerar o máximo de informação, no entanto, isso terá os seus custos e, pode dar-se o caso na mesma, de não termos em conta todos os ficheiros que uma aplicação possa eventualmente ler ou escrever no futuro. Para resolver estes problemas sugerimos uma segunda solução; ii) uma solução copy-on-reference, em que os ficheiros apenas são transportados quando referenciados, mas com algumas modificações: os ficheiros são lidos sempre do nó original, mas com suporte para caching. A escrita dos ficheiros é sempre enviada também para esse nó, de forma a evitar que a informação fique dispersa entre os vários nós que uma aplicação pode passar.

Os sockets têm questões adicionais, e por isso encontram-se ainda numa fase inicial.

5

Avaliação

Contents

Os mecanismos de checkpoint, restore e migração implementados, foram testados tanto em termos qualitativos, como em termos quantitativos. Os testes qualitativos focam-se em aplicações exemplo, que valorizam o conjunto de propriedades oferecidas (Secção 1.1), e que demonstram a correcção/consistência dessas aplicações com os mecanismos desenvolvidos. Os testes quantitativos avaliam o desempenho dos componentes internos desenvolvidos (state extraction, state restoration, checkpoint, restore e migração),[1] apresentados na arquitectura principal deste trabalho (Capítulo 3), e avaliam também o custo adicional de execução das aplicações com suporte para checkpoint e migração.

5.1 Avaliação Qualitativa

À medida que os componentes foram desenvolvidos, foram também testados de forma individual e gradual, através de um conjunto de testes criados especificamente:

- Como exemplo de teste para aferir a correcção dos mecanismos de checkpoint em presença de paralelismo, sincronização, cooperação e competição entre threads, usámos uma aplicação produtor-consumidor (no contexto de sistemas operativos), com 5 threads produtoras e 5 threads consumidoras.
- Como exemplo de teste para exercitar a correcção na presença de múltiplos stack frames, foram usados testes com recursão para criar stacks com certas dimensões.
- Como exemplo de testes para trabalhar a correcção de aplicações com qualquer tipo/estrutura de dados primitivos ou compostos (e.g., dados inteiros, booleanos, caracteres, strings, entre outros, e estruturas de dados como arrays, listas, entre outros), foram utilizados alguns algoritmos de ordenação (Bubble Sort, Quicksort, e Merge Sort).
- Como exemplo para invocações JNI, alguns programas desenvolvidos na linguagem de programação C, foram executados através de invocações realizadas em Java.
- As escritas e leituras sobre ficheiros e sockets, foram testadas com aplicações simples, sem um contexto específico.

Esses testes individuais não são de forma alguma suficientes para garantir que a solução desenvolvida é totalmente consistente. É preciso combinar de forma exaustiva os vários componentes para detectar outros eventuais problemas que possam existir. Por este motivo, utilizámos aplicações exemplo, fornecidas por bibliotecas usadas mundialmente: Jama (uma biblioteca para álgebra linear)[2] e Neobio (uma biblioteca que implementa um conjunto de algoritmos ligados à bioinformática).[3] Os testes realizados, mostram que as propriedades fornecidas foram de máxima importância para testar esse tipo de aplicações.

[1] Migração é o tempo que demora a transportar o estado de execução entre dois nós diferentes.
[2] http://math.nist.gov/javanumerics/jama/
[3] http://www.bioinformatics.org/neobio/

As propriedades de transparência e completude oferecidas, permitiram que os mecanismos de checkpoint e migração fossem aplicados sobre as aplicações indicadas sem qualquer conhecimento das suas implementações. Para além disso, foram realizados também checkpoints sobre essas aplicações que demonstram que a informação persistida (num checkpoint) é suficiente para que sejam produzidos resultados correctos (quando recuperadas). Relativamente à portabilidade, foram transportados e carregados com sucesso checkpoints entre os sistemas operativos Mac OS X e Linux Kubuntu, com arquitecturas 32bit Intel.

É importante referir, que os testes que demonstraram a consistência da solução implementada, permitiram depurar outros problemas relacionados com erros dos mecanismos. De seguida descrevemos os problemas que foram detectados e modificados em última instância:

- Uma thread está num ponto seguro quando se encontra num yield point (Secção 4.4, Figura 4.2, thread #1). De forma a recuperar o estado de execução no restore (tal como se encontrava anteriormente), existem stack frames dos yield points que são preservados (tal como é demonstrado na figura indicada). Com esses stack frames, no restore, uma thread segura, iria automaticamente bloquear-se de novo num yield point, sem nenhum esforço adicional. Porém em certas condições no restore, realizar as acções de recompilação e re-executação de um yield point, pode gerar um erro na aplicação e terminá-la sem sucesso. Para contornar esta situação, não preservamos qualquer stack frame interno à maquina virtual que corresponda a um yield point, sendo que o último stack frame preservado nestas condições, é o primeiro stack frame existente na stack que ocorra após esses stack frames do yield point. No restore, forçamos uma thread a bloquear-se de novo no yield point correcto (na re-execução dos stack frames, é verdade que os yield points estão a ser invocados, tal como numa execução normal, contudo, a thread só entra num yield point para se bloquear novamente, quando encontra o stack frame correcto), recriando assim a execução anterior.

- A leitura e a escrita do/para o ecrã, nunca foram trabalhadas, e para determinados testes a execução da aplicação suspendia uma thread nesses pontos efectivamente seguros. Contudo, existem descritores nativos, `System.IN` e `System.OUT`, que se encontram presentes na stack dessa mesma thread. Já vimos que quando estamos perante estas situações, a stack é revertida. Essencialmente, a solução adoptada é muito idêntica à solução desenvolvida para a ligação com o estado externo dos ficheiros, com a excepção de que não é preciso preservar esses descritores nativos (ou outra informação interna), pelo facto de serem sempre recriados numa nova instância da máquina virtual.

5.2 Avaliação Quantitativa

Para testar o desempenho dos mecanismos de checkpoint, restore e migração, foram desenvolvidos programas de teste exemplo que combinam três parâmetros, que são manipulados directamente

pelos mecanismos implementados:

- Número de stack frames na aplicação em execução (#Stack Frames).
- Número de objectos na heap (#Objectos) referenciados por esses stack frames (que podem representar átomos individuais, moléculas, partículas, entre outros).
- Tamanho (médio) dos objectos na heap.

Os tempos medidos relativos ao funcionamento interno dos mecanismos implementados são expressos em milissegundos (mseg) e são valores médios extraídos após múltiplas execuções, em computadores com processadores Intel(R) Core(TM) i5 @ 2.4 Ghz, com 4GB RAM. As transferências de dados foram temporizadas numa rede local com uma velocidade de transferência de 100MB/seg.

Foram realizados quatro grupos de teste de forma a tentar descobrir possíveis pontos de degradação, e avaliar o custo associado com a operação de cada componente interno. O primeiro grupo avalia o tipo de aplicação comum, que tem poucos dados de informação (50-300 stacks frames, com os respectivos 50-300 objectos, com dimensões inferiores a 1 Kbyte: Figuras 5.1 e 5.2). Com base nos resultados obtidos neste grupo de teste, não é possível retirar muitos resultados significativos. Por esse motivo, foram gerados testes mais intensivos (restantes três grupos), que se focam de forma individual em cada um dos três parâmetros manipulados directamente pelos mecanismos. O primeiro grupo de teste intensivo, manipula aplicações com muitos stack frames (600, 1200 e 2400 respectivamente, Figuras 5.3 e 5.4). O segundo, instrumenta aplicações com muitos objectos na heap, mas com dimensões pequenas (10000, 100000 e 1000000 objectos, Figuras 5.5 e 5.6). Finalmente o terceiro, trabalha aplicações com poucos objectos, mas com grandes dimensões (10, 100 e 200 MB, Figuras 5.7 e 5.8). Neste último teste, apenas se teve em conta a dimensão dos objectos extraídos na sua totalidade. Estes três últimos testes tentam também estudar o custo adicional causado por aumentar o número dos parâmetros indicados.

Através destes resultados, algumas conclusões podem ser retiradas. Primeiro, os resultados são muito encorajadores, uma vez que a latência imposta é muito pequena relativamente aos tempos de execução (longos) das aplicações em questão. Portanto, os checkpoints podem ser realizados com ganhos significativos na confiabilidade e no desempenho a longo prazo na presença de falhas.

Segundo, aplicações com muitos stack frames, apenas sofrem penalizações no desempenho no componente state extraction (Figura 5.3). A razão pela qual isso acontece, é porque existe um algoritmo de type inference que é realizado sobre os stack frames da aplicação. Para muitos stack frames, isto consome alguns recursos.

Em terceiro lugar, os mecanismos de serialização e deserialização, utilizados nos componentes de checkpoint e restore respectivamente (Figuras 5.4, 5.6 e 5.8), são os que mais prejudicam no desempenho dos mecanismos. Pior, à medida que se vai aumentando o número de objectos, a degradação aumenta significativamente (Figura 5.6). Isto deve-se maioritariamente aos mecanismos

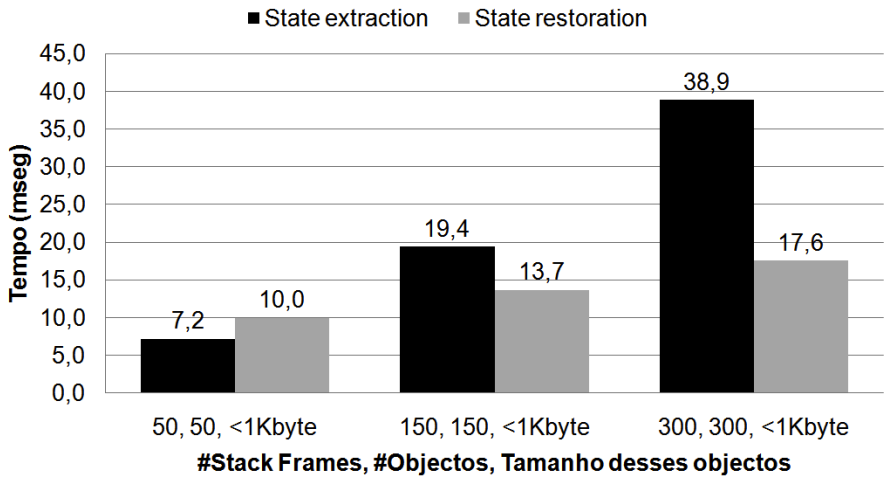

Figura 5.1: Microbenchmarks das componentes internas state extraction e restoration do primeiro grupo de testes.

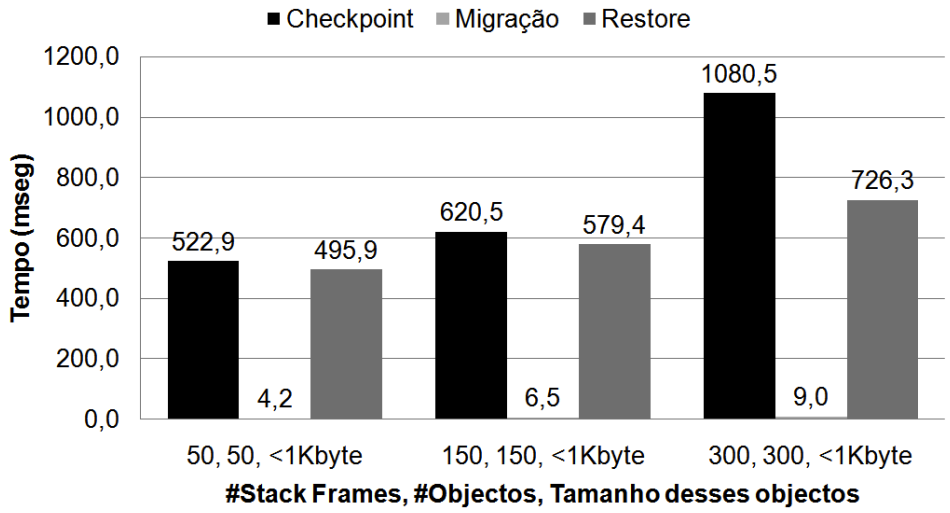

Figura 5.2: Microbenchmarks das componentes internas checkpoint, restore e migração do primeiro grupo de testes.

de serialização que estão a ser utilizados. Neste momento, estamos a tirar partido dos mecanismos de serialização oferecidos pelo `GNU CLASSPATH`. Esses mecanismos quando trabalham com muitos objectos são ineficientes. Relativamente às aplicações com poucos objectos mas com grandes dimensões, o custo adicional provocado pelo aumento desse factor é linear, contudo, embora não seja tão grave em comparação com o aumento do número de objectos, os tempos de execução dos componentes internos checkpoint e restore, já subiram acima dos 2 segundos (fugindo à média relativa aos outros componentes). Por estas razões, vale a pena explorar:

- Checkpoints em *background*.
- Checkpoints diferenciais/incrementais.

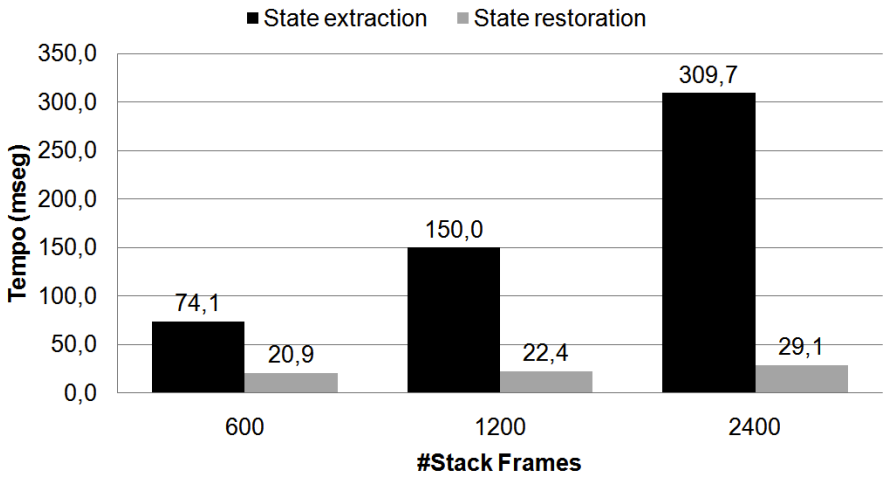

Figura 5.3: Microbenchmarks das componentes state extraction e restoration do primeiro grupo de testes intensivo.

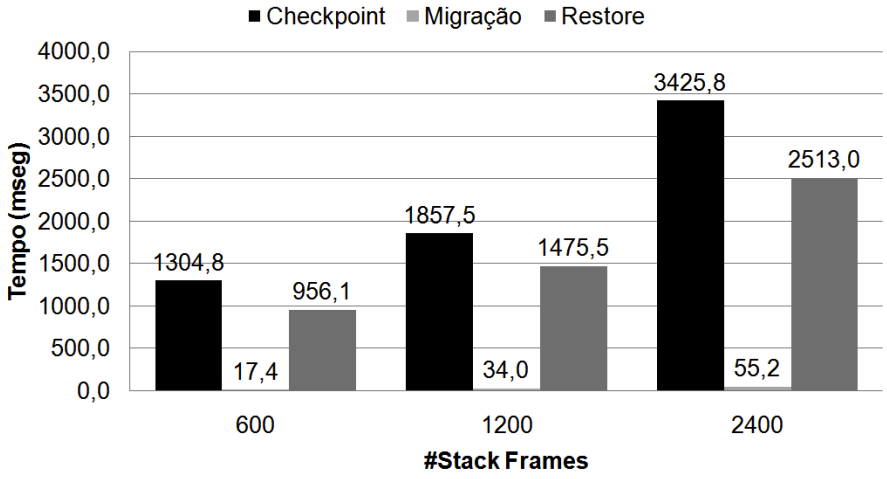

Figura 5.4: Microbenchmarks das componentes internas checkpoint, restore e migração do primeiro grupo de testes intensivo.

Finalmente, é importante salientar que dentro de um cluster, o custo adicional de migração é muito reduzido. É verdade que quando manipulámos grandes quantidades de informação, o tempo de migração já não se comportou dentro dos parâmetros reduzidos, no entanto, para a quantidade de informação que foi preservada, esse custo adicional é perfeitamente plausível. Por outro lado, se as condições de rede no cluster fossem melhores (e.g., aumentando a velocidade no cluster para 1GB/seg), então, já não se punha em causa esta questão relacionada com a migração dentro de um cluster.

Adicionalmente, foram realizados alguns testes com a componente de migração para ilustrar os mecanismos desenvolvidos em cenários de larga escala (como é o caso da computação em Grid e Cloud), de forma a transportar as aplicações para mais nós disponíveis. Como esperado, os resultados mostram que a componente onde se perde mais desempenho é na migração (Figura 5.9). De

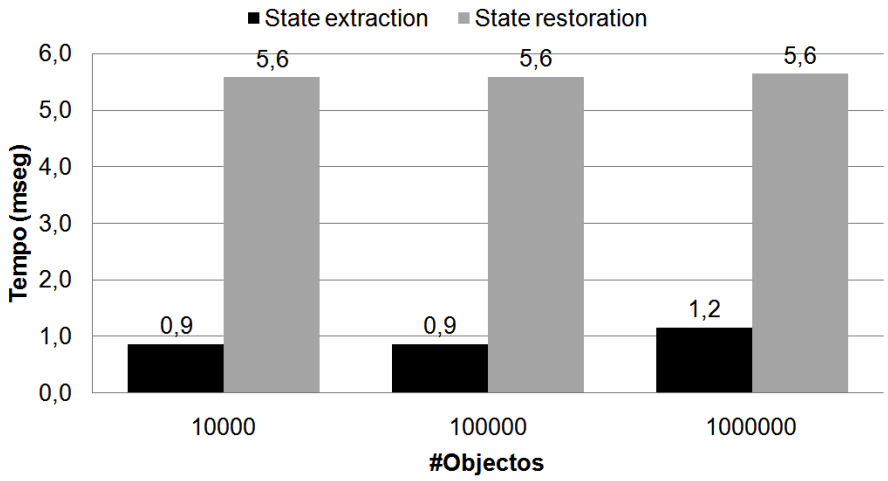

Figura 5.5: Microbenchmarks das componentes state extraction e restoration do segundo grupo de testes intensivo.

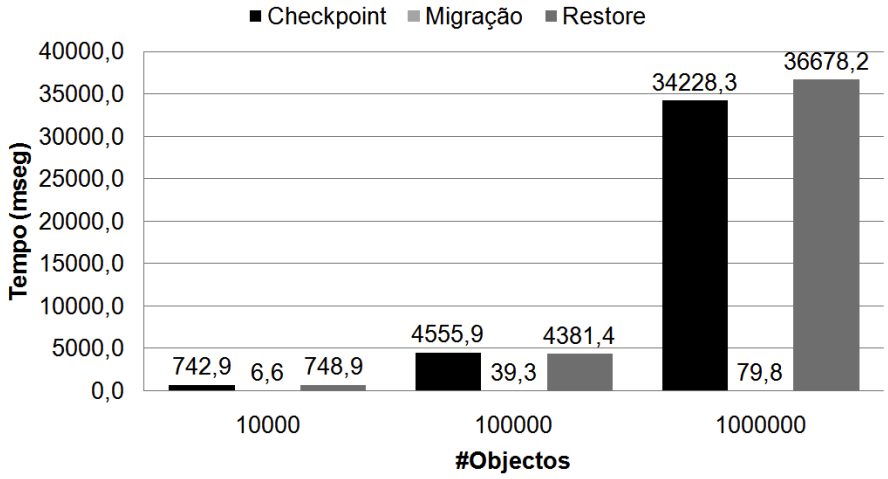

Figura 5.6: Microbenchmarks das componentes internas checkpoint, restore e migração do segundo grupo de testes intensivo.

forma a reduzir isto, comprimimos os checkpoints. Os tempos totais de migração com compressão reduzem em média 25% nos tempos totais de migração, o que é um resultado muito significativo.

Relativamente ao custos adicionais de desempenho durante execução das aplicações, também foram avaliados. Para medir estes custos, foram utilizados os benchmarks DaCapo [6] (Figura 5.10). Os resultados são bastante motivadores, sendo que para o compilador baseline, o prejuízo no desempenho da execução das aplicações é sempre inferior a 10% (muito abaixo das soluções já anteriormente referenciadas [7, 38, 39, 45], que adicionam em média penalizações de desempenho superiores a 300% na execução das aplicações). Com base numa análise, este pequeno custo adicional na execução, está a ser apenas influenciado pelo algoritmo que mantém a estrutura de dados dos locks que uma thread detém por motivos de sincronização. Mais nenhum factor afecta o desempenho das aplicações. Como na abordagem final, também queremos suportar métodos

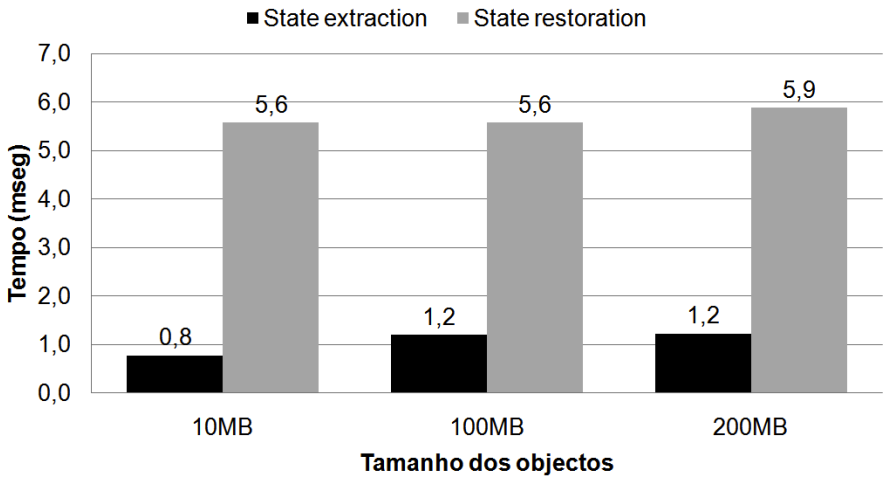

Figura 5.7: Microbenchmarks das componentes state extraction e restoration do terceiro grupo de testes intensivo.

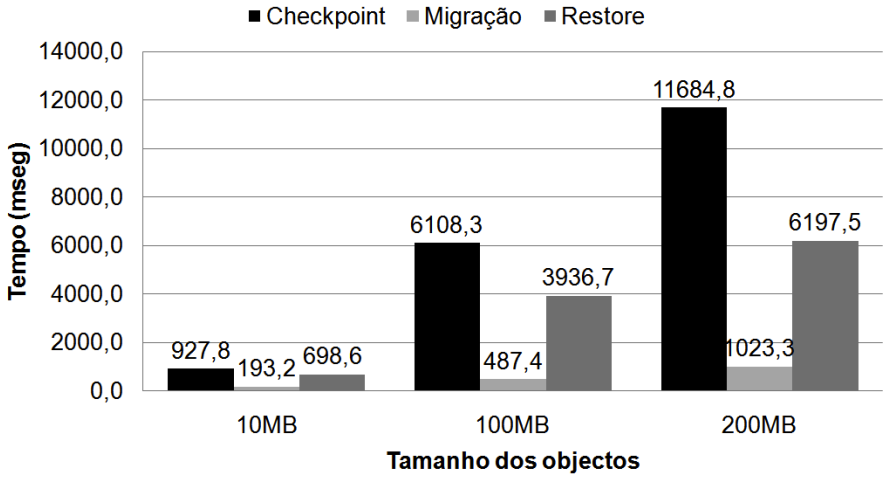

Figura 5.8: Microbenchmarks das componentes internas checkpoint, restore e migração do terceiro grupo de testes intensivo.

optimizados (por ora, ainda não conseguimos extrair da stack métodos optimizados), avaliámos já também o seu desempenho. Os resultados mostram que a execução das aplicações têm melhor desempenho (relativamente aos baseline), não excedendo os 3% no custo adicional, para todos os testes (Figura 5.10).

5.2.1 Benchmarks com aplicações exemplo de cálculo numérico

As medições que foram extraídas com os benchmarks DaCapo, não retratam aplicações no contexto científico e, também não demonstram aplicações de longa duração. Por este motivo, apresentamos os resultados da avaliação sobre duas aplicações de cálculo numérico intensivo (SOR[4] e MonteCarlo PI, com o mesmo tipo de tarefas elementares utilizadas no e-Science), que ilustram que o checkpointing produz ganhos significativos no desempenho a longo prazo na presença de

[4]Successive Over-relaxation.

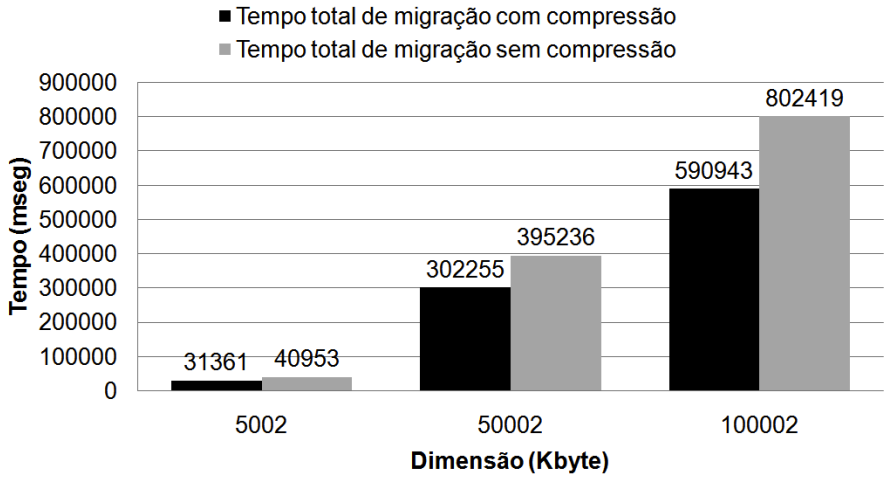

Figura 5.9: Custos de migração dos checkpoints gerados em 5.8, com e sem compressão.

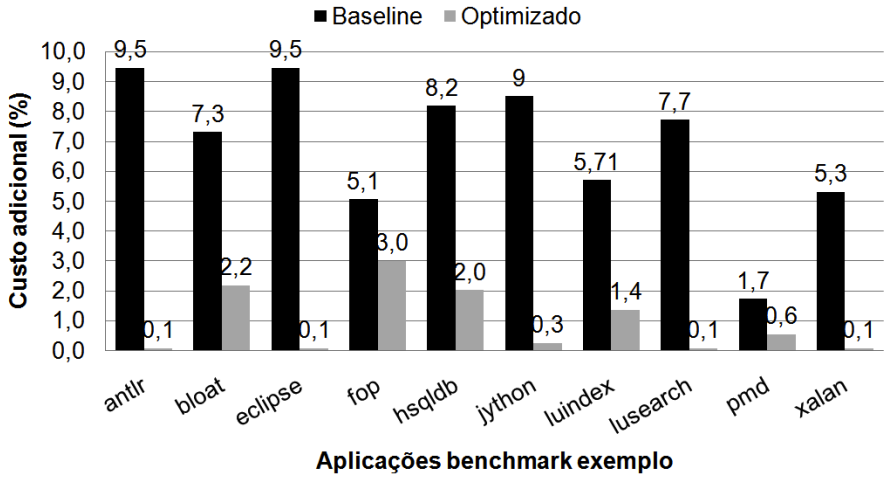

Figura 5.10: Benchmarks DaCapo. Custos de execução adicionais nas aplicações.

falhas.

Os testes realizados foram os seguintes. Para a aplicação SOR, utilizámos como parâmetros uma matriz de 100x100, com um número de 1000000 iterações. Para a aplicação MonteCarlo PI, cujo único parâmetro é o número de iterações, testou-se com 100000000 iterações ao quadrado. Os resultados obtidos são apresentados na Figura 5.11.

Primeiro há que salientar, tal como nos benchmarks DaCapo, o custo adicional de execução das aplicações com suporte para checkpoint e migração foi muito reduzido (inferior a 5%). Segundo, o tempo para realizar um checkpoint foi muito inferior ao tempo total da execução das aplicações. Se aplicássemos checkpoints sucessivos sobre as aplicações respectivas, por exemplo com intervalos de 5 minutos, para o teste realizado na aplicação SOR, adicionávamos no máximo apenas 15 segundos na execução total e para o segundo teste realizado na aplicação MonteCarlo PI, adicionávamos também no máximo apenas 5 segundos na sua execução total. Estes tempos de execução adicionais, são insignificantes comparados com os tempos totais das execuções apresentadas. Se surgisse uma

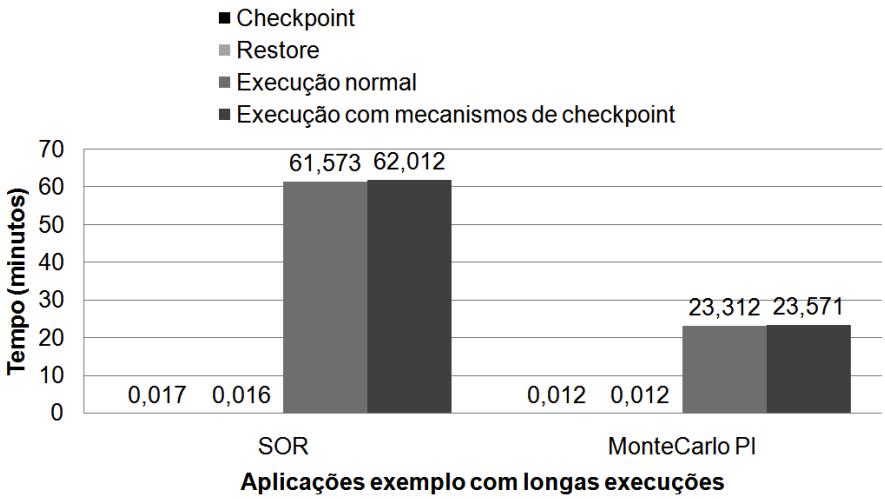

Figura 5.11: Benchmarks para aplicações com longas execuções, SOR e MonteCarlo PI.

falha durante essas longas execuções, com checkpoint, poder-se-ia recuperar o estado de execução, para um ponto mais próximo da falha, sem ter que repetir novamente a execução a partir do zero. Isto comprova mais uma vez o que tem vindo a ser dito: as aplicações com suporte para checkpoint, ganham bastante no desempenho quando na presença de falhas.

5.3 Análise global da solução desenvolvida

Os resultados que foram obtidos, são bastante positivos. Na secção 2.1.3, abordámos um conjunto de soluções para checkpoint e migração ao nível de uma VM OO. Essas soluções foram caracterizadas e sumariadas através de um conjunto de propriedades (Tabela 2.2, na mesma secção). Como até ao momento, a solução proposta ainda não tinha sido globalmente avaliada, nesta secção, analisamos a solução desenvolvida em função dessas características, como forma de avaliação geral, e também comparativa:

- Nível de implementação: VM OO.
- Contexto: checkpoint e migração de aplicações (processo).
- Estado interno: completo. Embora ainda não consigamos extrair da stack métodos optimizados, pretendemos explorar uma solução para esse problema no futuro. No entanto, sem optimizações conseguimos retratar todo o estado interno de execução de uma aplicação (explorando conceitos que outras soluções ainda não tinham abordado: threads efectivamente seguras, estado de sincronização, entre outros).
- Descritores de ficheiro: sim.
- Conteúdo dos ficheiros: parcial, o trabalho relacionado com este problema ainda é trabalho em curso, mas na presença de sistemas de ficheiros distribuídos, tudo funciona como previsto.

- Sockets: sim. Questões ligadas a mensagens perdidas, e mudanças de contexto são problemas que ainda não foram resolvidos. Mas também é verdade que, este tipo de problemas pode já não fazer parte do contexto inicial da solução proposta. A solução [27], é a única solução existente que trabalha os sockets, e no entanto, também tem os mesmos problemas.
- Múltiplas threads: sim, e também na presença de sincronização.
- Transparente: sim.
- Portável: parcial, as alterações realizadas sobre a máquina virtual JikesRVM, têm de estar aplicadas/disponíveis em todos os nós. Este assunto já se encontrava indicado nos objectivos. Contudo, foram já realizados testes que comprovam que a solução é portável entre sistemas operativos diferentes, cumprindo assim o objectivo proposto.
- Eficiente: sim, é verdade que mostrámos algumas condições em que os mecanismos têm de ser melhorados (na serialização em disco e migração em larga escala). Contudo, as soluções existentes não se comprometeram a realizar testes tão intensivos e exaustivos, como os que foram apresentados neste documento. Portanto, essas soluções muito provavelmente, sobre os mesmos testes e condições, iriam obter resultados piores (e há que ter a noção, que o estado de execução trabalhado é muito maior do que qualquer solução apresentada anteriormente na literatura).

6

Conclusões

Contents

Nos dias correntes, existem cada vez mais aplicações em campos ligados a e-Science (química, bioinformática) que são desenvolvidas em Java. Essas aplicações normalmente têm execuções longas e trabalham sobre muitos dados de informação. Quando falham durante execuções longas, todo o trabalho realizado é perdido, a não ser que os programadores implementem explicitamente alguma forma intermédia de guardar os resultados que já foram calculados. Contudo, estas aplicações geralmente são desenhadas por cientistas que não são engenheiros de computação, e por isso, cada nova aplicação tem de implementar repetidamente essas abordagens, sendo também que, um cientista não está preocupado com eventuais problemas das aplicações que estejam relacionados com falhas ligadas ao hardware ou software (situações extremas, que podem ocorrer). Nesta tese de mestrado, descrevemos uma solução para esses problemas, estendendo uma máquina virtual Java com mecanismos de checkpoint, restore e migração.

As soluções de checkpoint e migração existentes estão implementadas a diferentes níveis: nível de processo, VM sistema, e VM OO, o assunto principal deste trabalho. As abordagens realizadas ao nível de processo e VM sistema são insuficientes pelas seguintes razões: ou obrigam a que seja guardada / transportada informação que não é relativa apenas à aplicação em si, ou limitam a portabilidade da mesma. As soluções ao nível de uma VM OO são mais recentes mas também têm alguns problemas. A maioria encaixa-se principalmente no contexto de agentes móveis, e por isso, assumem que o ambiente de execução é bem definido e controlado. Outro problema comum está relacionado com o estado de execução que conseguem retratar. São poucas as soluções que trabalham as aplicações como um todo (ou seja, aplicações com múltiplas threads), e ainda são menos aquelas que trabalham o estado externo à aplicação. Em termos globais, o estado externo à aplicação foi sempre a propriedade menos trabalhada. Para além dessa propriedade, as soluções tendem sempre a dar responsabilidades ao programador (diminuindo a transparência), de forma a usufruir destes mecanismos (no pior caso, as interfaces disponíveis têm de ser invocadas explicitamente na aplicação). Isso pode ser um problema muito grave, isto porque, quem utilizar os mecanismos de checkpoint ou migração pode não ser o próprio programador, e como tal, podem surgir resultados inesperados (e.g., erros de compilação) quando uma aplicação é compilada com esses mecanismos. Pode até acontecer que o código fonte de uma aplicação nem seja fornecido, o que não permitiria usufruir destes mecanismos.

Com base nesta análise do trabalho relacionado, a solução desenhada focou-se principalmente em mecanismos de checkpoint, restore e migração, que fossem transparentes para os programadores, sem obrigar a que fosse necessário realizar alterações nas suas aplicações e, também completos, explorando o tipo de aplicações mais comum (com longas execuções). No entanto, também são suportadas outras propriedades.

A arquitectura deste trabalho, foi desenhada com um foco especial nessas propriedades. Três componentes fazem parte da arquitectura principal da solução desenvolvida: a aplicação, um controlador e um serviço de migração. A aplicação executa-se sobre uma máquina virtual estendida

com mecanismos que lhe permitem extrair, recuperar e transportar o estado de execução. O componente controlador permite estabelecer uma relação transparente entre utilizador e aplicação. Finalmente o componente serviço de migração, estará presente em todos os nós, para permitir que o estado de execução possa ser enviado de forma directa entre dois nós.

Porém, existiram outros detalhes de arquitectura ligados directamente com as propriedades oferecidas, que mereceram alguma atenção. De forma a obter uma solução portável, é preciso criar uma representação intermédia em Java de todo o estado de execução da aplicação. Para serializar em disco ou para uma rede esse estado de execução, os mecanismos de serialização do Java fornecidos pela máquina virtual têm de ser configurados de forma a não dar responsabilidades ao programador. Para poder obter um estado consistente da aplicação, é preciso garantir que todas as threads são paradas em pontos seguros, ou efectivamente seguros. Para que os mecanismos de checkpoint e migração possam ter em conta estado nativo (JNI), esse estado tem de ser revertido (pelo facto de não haver controlo sobre estado nativo), tal e qual como se nunca tivesse ocorrido, e manter-se consistente. O estado de sincronização é muito dependente da implementação, mas como muito possivelmente estamos a trabalhar com locks que têm dependências nativas, esse estado de sincronização também é revertido, com a atenção especial de que no restore tem de ser recriado novamente. Relativamente ao estado externo, não deve ser uma questão deixada para resolver pelo programador, isto porque, como já foi referido anteriormente, quem usar os mecanismos de checkpoint ou migração pode nem ser o próprio programador que desenvolveu a aplicação. Em termos de eficiência, tentamos reduzir penalizações a longo prazo ligadas directamente com a execução da aplicação.

Relativamente à implementação dos mecanismos de checkpoint, restore e migração sobre a máquina virtual Java JikesRVM, houve muitos detalhes que tiveram de ser trabalhados.

Para bloquear uma thread num ponto seguro, tirámos partido dos yield points suportados pela máquina virtual JikesRVM. De forma a parar a execução das threads da aplicação nesses pontos seguros, invocamos mecanismos nativos, das bibliotecas de threads nativas suportadas (e.g., pthreads ou Apache Harmony threads).

A extracção e recuperação das threads da aplicação foram os mecanismos que tiveram mais detalhes de implementação. Para extrair uma thread da aplicação, foi preciso analisar todos os stack frames correspondentes, para retirar informação das variáveis locais, operandos, a posição actual no bytecode, entre outros. Adicionalmente foi também necessário inferir os tipos das variáveis, isto porque, este tipo de informação não está presente na máquina virtual JikesRVM. A recuperação de todas as threads da aplicação é realizada de forma faseada. Primeiro reconstruímos todos os stack frames, sendo que o resultado gerado é um conjunto de métodos compilados em memória, que quando executados reconstroem o seu estado de execução anterior. De seguida cada thread readquire todos os locks de sincronização que detinha anteriormente. No fim, uma nova thread é gerada para cada thread que foi preservada, e re-executa todos os métodos compilados pela ordem

que tinham sido anteriormente, sendo apenas nesta fase que a reconstrução dos stack frames é realizada. Depois destes passos, as threads são automaticamente bloqueadas, para que o restore dessas threads seja efectuado ao mesmo tempo.

Relativamente ao estado de sincronização entre várias threads, tem de ser revertido, pelo facto de ser estado que não é facilmente guardado. No restore, como estamos perante situações deterministas, as threads que se encontravam bloqueadas, irão bloquear-se nos mesmos conjuntos (`blocked` ou `wait`), sem problemas de consistência.

A API suportada é muito simples e, sempre que uma thread da aplicação realiza um checkpoint ou migração, essa thread não se encontra num estado seguro, nem efectivamente seguro, e por isso é necessário ter alguns cuidados adicionais.

Em relação ao trabalho relacionado com o estado externo, ainda é trabalho em curso, mas a solução desenvolvida já consegue guardar as ligações com o estado externo dos ficheiros e dos sockets, mesmo quando essas ligações estão a ser utilizadas. Essencialmente, para evitarmos guardar estado interno, muito possivelmente nativo, relativo a essas ligações, tal como na solução para o estado de sincronização, revertemos as threads para uma posição que é mais facilmente controlável. A informação nativa da ligação com ficheiros preservada é caracterizada basicamente pela localização com o nome do ficheiro e o cursor dentro do ficheiro. Para os sockets são preservados, o endereço e porto local/destino, consoante o tipo de socket.

Para o conteúdo dos ficheiros, sugerimos como abordagem ao problema as seguintes soluções: i) uma solução eager all, onde todos os ficheiros são completamente transportados em conjunto com o estado de execução interno (esta solução tem sempre um problema que é saber que ficheiros é que realmente pertencem a uma aplicação). A solução pode ser pessimista para tentar aglomerar o máximo de informação, no entanto, isso terá os seus custos e, pode dar-se o caso na mesma, de não termos em conta todos os ficheiros que uma aplicação possa eventualmente ler ou escrever no futuro. Para resolver estes problemas sugerimos uma segunda solução; ii) uma solução copy-on-reference, em que os ficheiros apenas são transportados quando referenciados, mas com algumas modificações: os ficheiros são lidos sempre do nó original, mas com suporte para caching. A escrita dos ficheiros é sempre enviada também para esse nó, de forma a evitar que a informação fique dispersa entre os vários nós que uma aplicação pode passar.

Os sockets têm questões adicionais, e por isso encontram-se ainda numa fase inicial.

Finalmente, a solução proposta foi avaliada em termos de correcção e de desempenho. Os resultados são bastante encorajadores e positivos. Embora tenhamos analisado algumas situações que têm de ser melhoradas (serialização em disco e migração em larga escala), no geral, a latência imposta é muita pequena relativamente aos tempos de execução (longos) das aplicações em questão. Portanto, os checkpoints podem ser realizados com ganhos na confiabilidade e no desempenho a longo prazo na presença de falhas.

6.1 Trabalho futuro

Ao longo deste documento, têm sido já abordadas algumas questões que são apontadas como trabalho futuro. Nesta secção, apresentamo-las de forma resumida:

- Embora ainda não tenha sido mencionado, os mecanismos de serialização modificados (para possibilitarem uma solução transparente), ainda precisam de ser trabalhados. Para métodos com campos estáticos, é preciso construir um mecanismo automático que os transporte num checkpoint ou migração. Este comportamento não é suportado de base pelos mecanismos de serialização Java, porque, não está definido na sua especificação se um campo estático que é recuperado usa o seu valor pré-definido ou o valor preservado na serialização. Para a nossa solução, queremos utilizar o valor preservado.

- Investigar e avaliar outros mecanismos de serialização disponíveis (e.g., Apache Harmony) e com melhores resultados, integrar com a solução desenvolvida.

- Suportar métodos optimizados. Um método optimizado tem mapeamentos internos (OSR Maps) que não se encontram disponíveis em todos os casos (a máquina virtual não se encontra desenhada dessa forma). Para além de criar uma versão funcional que extraia métodos optimizados usando os OSR maps, seria interessante explorar também outras alternativas, e compará-las (e.g., tentar transportar código JITed, supondo que os mecanismos são usados em plataformas idênticas).

- Suportar checkpoints em background e live migration.

- Suportar checkpoints diferenciais/incrementais, onde se tenha apenas em conta as modificações que foram realizadas entre checkpoints (para reduzir a informação que é transportada).

- Avaliar e implementar soluções para os problemas da mudança de ambiente e mensagens perdidas relacionados com os sockets.

- Implementar as soluções eager all e copy-on-reference (modificada) para o conteúdo dos ficheiros e, de certa forma retirar resultados com comparações entre ambas.

- Investigar a adopção de uma solução idêntica mas no contexto das máquinas virtuais do universo .NET.

Bibliografia

[1] A Acharya, M Ranganathan, and J Saltz. Sumatra: A language for resource-aware mobile programs. *Lecture Notes in Computer Science*, 1222:111–130, 1997.

[2] Adnan Agbaria and Roy Friedman. Virtual-machine-based heterogeneous checkpointing. *Software: Practice and Experience*, 32(12):1175–1192, Outubro 2002.

[3] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211, 2000.

[4] A Barak and O La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4-5):361–372, Março 1998.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[6] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190. ACM, 2006.

[7] S Bouchenak and D Hagimont. Pickling threads state in the Java system. In *Third European Research Seminar on Advances in Distributed Systems*, 1999.

[8] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, and F. Boyer. Experiences implementing efficient Java thread serialization, mobility and persistence. *Software: Practice and Experience*, 34(4):355–393, 2004.

[9] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. *Proceedings of the 3rd international conference on Virtual execution environments - VEE '07*, pages 169–179, 2007.

[10] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[11] G. Cabri, L. Leonardi, and R. Quitadamo. Enabling Java mobile computing on the IBM Jikes research virtual machine. *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 62–71, 2006.

[12] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Addison-Wesley Longman, 2005.

[13] W R Dieter and J E Lumpp Jr. User-level checkpointing for LinuxThreads programs.

[14] G.W. Dunlap, D.G. Lucchetti, M.A. Fetterman, and P.M. Chen. Execution replay of multiprocessor virtual machines. *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2008.

[15] S. Ffinfrocken. Transparent migration of Java-based mobile agents. *Springer*, Volume 147:26–37, 1998.

[16] S.J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 241–252. IEEE, 2003.

[17] Dominik Gront and Andrzej Kolinski. Utility library for structural bioinformatics. *Bioinformatics*, 24(4):584–585, 2008.

[18] Erik Hendriks. BProc: the Beowulf distributed process space. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 129–136, New York, NY, USA, 2002. ACM.

[19] Richard C. G. Holland, Thomas A. Down, Matthew R. Pocock, Andreas Prlic, David Huen, Keith James, Sylvain Foisy, Andreas Dräger, Andy Yates, Michael Heuer, and Mark J. Schreiber. Biojava: an open-source framework for bioinformatics. *Bioinformatics*, 24(18):2096–2097, 2008.

[20] J. Howell. Straightforward Java persistence through checkpointing. *Advances in Persistent Object Systems: Proceedings of the Int'l Workshop on Persistent Object Systems (POS) and the Int'l Workshop on Persistence \& Java (PJAVA)*, pages 322–334, 1999.

[21] T. Illmann, T. Krueger, F. Kargl, and M. Weber. Transparent migration of mobile agents using the java platform debugger architecture. *Lecture Notes in Computer Science*, pages 198–212, 2001.

[22] S.T. King, G.W. Dunlap, and P.M. Chen. Debugging operating systems with time-traveling virtual machines. *Proc. USENIX Annual Technical Conference*, pages 1–15, 2005.

[23] C. Kintala. Checkpointing and its applications. *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, (June):22–31, 1995.

[24] F.C.M. Lau. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. *Proceedings. IEEE International Conference on Cluster Computing*, pages 381–388.

[25] M Litzkow and M Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In *In Proceedings of the Winter 1992 USENIX Conference*, 1992.

[26] Ivan López-Arévalo, René Bañares-Alcántara, Arantza Aldea, and A. Rodríguez-Martínez. A hierarchical approach for the redesign of chemical processes. *Knowl. Inf. Syst.*, 12(2):169–201, 2007.

[27] R.K.K. Ma, C.L. Wang, and F.C.M. Lau. M-JavaMPI: A Java-MPI binding with process migration support. *The Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 1–9, 2002.

[28] D S Milojicic, W Zint, A Dangel, and P Giese. Task Migration on the top of the Mach Microkernel. In *USENIX MACH III Symposium table of contents*, pages 273–290. USENIX Association Berkeley, CA, USA, 1993.

[29] DS Milojicic, F Douglis, Y Paindaveine, and R. Process migration. *ACM Computing Surveys*, 2000.

[30] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434, 2003.

[31] J.K. Ousterhout, a.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, Fevereiro 1988.

[32] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K.H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 177–192, 2009.

[33] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. *In USENIX Winter 1995 Technical Conference, January*, 1995.

[34] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating bugs as allergies. *ACM Transactions on Computer Systems*, 25(3):7, 2007.

[35] R. Quitadamo and L. Leonardi. *The Issue of Strong Mobility: an Innovative Approach based on the IBM Jikes Research Virtual Machine*. PhD thesis, University of Modena and Reggio Emilia, 2008.

[36] R F Rashid and G G Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the eighth ACM symposium on Operating systems principles*, pages 64–75. ACM New York, NY, USA, 1981.

[37] E. Roman. A survey of checkpoint/restart implementations. *Citeseer*, pages 1–9, 2002.

[38] T Sakamoto, T Sekiguchi, and A Yonezawa. Bytecode transformation for portable thread migration in Java. *Lecture Notes in Computer Science*, pages 16–28, 2000.

[39] T Sekiguchi, H Masuhara, and A Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. *in Coordination Models and Languages*, 1999.

[40] S.M. Srinivasan, S. Kandula, C.R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. *usenix.org*, 2004.

[41] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[42] T. Suezawa. Persistent execution state of a Java virtual machine. *Proceedings of the ACM 2000 conference on Java Grande*, pages 160–167, 2000.

[43] N Suri, J M Bradshaw, M R Breedy, P T Groth, G A Hill, and R Jeffers. Strong mobility and fine-grained resource control in NOMADS. *Lecture Notes in Computer Science*, pages 2–15, 2000.

[44] M Theimer, K Lantz, and D Cheriton. Preemptable remote execution facilities for the V-System. *ACM SIGOPS Operating Systems Review*, 19(5):12, 1985.

[45] E Truyen, B Robben, B Vanhaute, T Coninx, W Joosen, and P Verbaeten. Portable support for transparent thread migration in Java. *Lecture notes in computer science*, pages 29–43, 2000.

[46] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. *ACM SIGOPS Operating Systems Review*, 41(6):131–144, 2007.

[47] B Walker, G Popek, R English, C Kline, and G Thiel. The LOCUS distributed operating system. In *Proceedings of the ninth ACM symposium on Operating systems principles*, page 70. Acm, 1983.

[48] H Zhong and J Nieh. CRAK: Linux checkpoint/restart as a kernel module. *Technical Report CUCS-014-01, Department of Computer Science, Columbia University*, pages 1–18, 2001.

[49] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. *Software: Practice and Experience*, 23(12):1305–1336, Dezembro 1993.