2
Migração heterogênea forte de computações

Este capítulo introduz os conceitos relacionados à migração heterogênea forte de computações e os problemas que os programadores encontram para a implementação desse tipo de migração relacionados à falta de suporte para captura e restauração de estado de execução, assim como soluções comuns a esses problemas. Um estudo mais extenso sobre esse tópico pode ser consultado em [MRS08].

2.1
Preliminares

Uma das dificuldades envolvidas na pesquisa sobre migração está relacionada à sobrecarga de termos utilizados nesta subárea. Por este motivo, nesta seção definiremos a notação a ser usada ao longo deste documento.

Técnicas de migração têm sido estudadas e aplicadas a diferentes níveis de granularidade, como processos [MDPW+00, JC04], threads [Funfrocken98, SMY99, TRVC+00, BHKP+04, JC04], closures [Cardelli95, CJK95], e registros de ativação [HWW93]. É muito comum referir-se às unidades de migração como *agentes móveis*, que podem ser descritos como entidades que podem transportar seu estado de um ambiente à outro. Os agentes móveis podem ser compostos por múltiplas threads que devem mover-se em conjunto, como em Wasp [Funfrocken98], ou podem compartilhar uma thread, como em Aglets [LO98]. Nessa tese usaremos indistintamente os termos computação, unidade de execução (EU) [FPV98], ou unidade de migração [Shub90], exceto quando for necessário especificar a granularidade. Aqui aderiremos à terminologia proposta por Cardelli [Cardelli97, Cardelli99], em que o termo *mobile computation* [1] se diferencia da computação móvel (*mobile computing*) em que o primeiro trata da mobilidade lógica das computações, e o segundo, da mobilidade física dos dispositivos. A migração de computações consiste basicamente na suspensão de uma computação, e o armazenamento e serialização do seu estado que será transferido até um nó remoto em que a computação será reiniciada. O estado inclui o estado interno (em geral, conteúdos do heap, da

[1] na falta de uma tradução mais adequada diferente de "computação móvel" –já bem estabelecida–, usaremos o termo *mobilidade (ou migração) de computações*

pilha, dos registros, variáveis globais) e a informação do ambiente, que pode considerar-se como tudo o que é externo à computação (descritores de arquivo, bibliotecas, identificadores de usuário e de processos). Na realidade, Fuggetta et al. [FPV98] definiram migração e *clonagem remota* como os mecanismos que suportam mobilidade forte. Diferente da clonagem, a migração envolve um último passo adicional, que consiste no cancelamento da computação original. Entretanto, neste trabalho, usaremos o termo migração como sinônimo de mobilidade exceto quando a diferença for relevante.

A migração é comumente chamada de transparente quando os efeitos do movimento estão ocultos ao usuário e à aplicação. Isto só pode ser conseguido tratando todas as referências da computação a objetos e recursos. O suporte para transparência é difícil de implementar sem depender diretamente do sistema operacional. Por este motivo, a transparência é comumente tratada com restrições a esse nível (por exemplo, as conexões podem não ser restauradas, pode ser limitada a restauração da entrada/saída). Em geral, atualmente a transparência não é mais considerada como uma questão crítica, pois se por um lado ajuda em termos de complexidade, pelo outro não permite o controle de erros e as possíveis otimizações que possam ser executadas baseado na localização.

O programa transferido pode capturar e restaurar o seu estado ou então isso pode ser executado por um serviço externo. Estas abordagens serão distinguidas como manipulação *interna* e *externa*, respectivamente.

Por outro lado, a classificação tradicional dos mecanismos de mobilidade em fraca [FPV98] ou forte não resulta clara, pois existem na realidade diferentes "graus" de força. Assume-se comumente que na mobilidade fraca são transferidos o código e opcionalmente alguns dados, enquanto na mobilidade forte o estado de execução também migra, de forma que a computação pode ser reiniciada do ponto em que foi suspensa. Não obstante, é difícil migrar o estado de execução como um todo, e na realidade nem sempre é necessário. Ainda, em alguns sistemas, a suspensão somente é possível quando a pilha estiver vazia, por exemplo, quando está esperando em um loop de eventos, de forma que a diferença entre fraca e forte desaparece. Todavia, é possível encapsular a continuação de uma execução através de dados e código que podem ser migrados usando um mecanismo de migração fraca [MRS08]. Por este motivo, neste trabalho consideraremos como fortes aqueles mecanismos capazes de produzir novas computações cujo estado de execução seja equivalente ao da computação original. Limitações à quantidade de informação contida na pilha de execução não são interessantes.

A disponibilidade de mecanismos de migração forte é desejável em apli-

cações de balanceamento de carga para programas executando computações intensivas e/ou de longa duração. Também na área de agentes móveis, pois a migração fraca contraria o comportamento esperado desses agentes que deveriam continuar a execução automaticamente assim que fossem reativados no destino [BD01]. Entretanto, atualmente muitos sistemas oferecem somente migração fraca. Wang et al. [WHB01] afirmam que a modularidade oferecida pela mobilidade forte comparada à mobilidade fraca tornam a mobilidade forte mais confiável, porque é mais fácil raciocinar sobre o código e depurá-lo.

Qualquer que seja o método de migração, ela deve ser *correta*. Após uma migração correta, o resultado da computação migrada é igual ao resultado da mesma computação não migrada, dada a mesma entrada [Smith97]. Lu e Liu [LL87] estabelecem dois requisitos que devem ser satisfeitos com esse objetivo: os estados da computação na origem e no destino deverão ser iguais, e a computação terá uma visão consistente do sistema (o ambiente) em todo momento. Dessa forma, para se ter uma migração correta não é suficiente restaurar exatamente a informação que descreve a computação original: isto não garante que a semântica do processo não será afetada pelo processo de migração. Tome-se, por exemplo, o caso de uma aplicação que devolva como um dos resultados finais o tempo de execução. Ainda que as máquinas de origem e destino estivessem sincronizadas, o resultado da medição incluiria a latência da migração, levando a um resultado incorreto. Em geral, formas de permitir a migração de processos que interagem com funções ou valores dependentes do sistema são: (i) a manipulação dessas funções de forma a oferecer um ambiente consistente na restauração; (ii) a restrição da suspensão a momentos em que a visão do sistema não inclua esse tipo de funções ou valores. Em efeito, a *migrabilidade* (ou capacidade de uma computação ser migrada corretamente) está relacionada ao estado da computação no instante da migração e ao entendimento dos efeitos dos fatores que podem impedir a migração de forma a poder manipulá-los [Smith97]. Fatores que contribuem na migrabilidade são: restrições da linguagem, análise estática, verificação de código em tempo de execução, suporte para tempo de execução, quantidade de informação de tipos, funcionalidades da ferramenta de migração, e similaridades entre as arquiteturas [Smith97]. Neste caso em que estudamos a captura e restauração de estado para um leque maior de aplicações além da migração, o termo restaurabilidade é mais adequado que aquele de migrabilidade.

A migração pode ser iniciada de dentro da computação (chamada de *pró-ativa*, ou *subjetiva*) ou então de fora da computação(chamada de *reativa*, *objetiva* ou *forçada*). Embora freqüente, a migração subjetiva afeta a separação entre a lógica da aplicação e a da mobilidade [PMOY06]. Por outro lado, na

migração objetiva aparecem vários problemas, derivados do fato de que nesse caso a migração está fora do controle do programador da aplicação. Além de problemas relacionados com autoridade, sobre quem teria direito a migrar a computação [CG98], outros requisitos podem surgir se a aplicação precisar de informação relativa à localização ou a mudança dela. Uma questão fundamental é que a computação deve ser interrompida de forma portável e somente quando estiver em estado consistente, ou seja, em uma forma reiniciável, estável e coerente [MP96]. Uma instrução de código fonte ou *bytecode* pode ser composta de várias instruções de código de máquina, cujo número pode diferir entre arquiteturas devido às diferenças em conjuntos de instruções. As interrupções não devem ocorrer enquanto uma instrução está sendo executada (exceto quando as operações restantes não têm efeitos colaterais) ou então a correção da restauração será afetada.

A consistência, então, é garantida permitindo a interrupção somente em determinados pontos, chamados de *pontos de ônibus* [SJ95], *pontos de migração* [TH91], *poll points* [FCG00, CS02], *pontos de preempção* [SH98] ou *pontos de adaptação* [JC04]. Neste texto eles serão chamados de pontos de migração. A justificativa desses pontos vem de que uma computação procedural pode ser modelada como uma progressão através de uma seqüência de estados bem definidos que são pontos na execução em que o estado de uma computação é equivalente ao de qualquer outra implementação da computação [BSS94].

A definição de pontos lógicos em que a migração é permitida tem outras motivações além da migração objetiva, como resolver o problema das diferentes localizações que o ponto de execução corrente poderia ter no segmento de texto em diferentes arquiteturas. Nesse caso, eles atuam como etiquetas para simular contadores de programa. O posicionamento desses pontos precisa ser cuidadoso, pois em excesso, geram uma sobrecarga em espaço e tempo de execução, e em número reduzido, podem provocar uma latência excessiva na resposta a requisições de suspensão. Sendo que os métodos de suspensão usualmente dependem da plataforma, a estratégia mais popular se baseia em *pooling*, que é executado nos pontos de migração.

2.2 Captura e restauração do estado de execução

A migração oferece uma ilusão de que a computação está sendo movida, quando na verdade se está criando uma nova computação no destino, a partir de informações parciais do estado interno e o ambiente da computação migrante, que irá executar no novo contexto local. A captura e restauração dessas informações do estado de execução da computação são problemas chave

nesse nível. Mais complexa que a migração em ambientes homogêneos, a migração heterogênea implica na necessidade de informação detalhada sobre cada dado devido às diferenças entre as plataformas em conjuntos de instruções e representações de dados.

Nesse sentido, a virtualização oferece uma alternativa interessante devido a sua crescente popularidade, que permitiria ter uma base instalada o suficientemente ampla para uma determinada aplicação. Os monitores permitem implementar sistemas de migração de forma homogênea, assim, o VMMigration [CFH+05] foi implementado sobre o Xen Virtual Machine Monitor. Esta abordagem tem também as vantagens de permitir sandboxing e a execução de vários sistemas operacionais de uma vez. Entretanto, este tipo de implementação efetivamente precisa da disponibilidade de uma determinada máquina virtual e tem uma granularidade alta e inflexível.

2.2.1 Captura e serialização

O passo de captura e serialização heterogêneas é o que segue à suspensão da execução. O estado capturado deve ser o mínimo requerido para gerar uma migração correta: isto permite melhorar o desempenho da migração e influi na migrabilidade, diminuindo a chance de se encontrar problemas relacionados à dependência do sistema. Ainda, depois da captura, a aplicação pode passar por processos de manutenção, atualização das entidades capturadas ou das dependências. Dessa forma, quanto menor for o volume capturado, menor será o esforço para adaptá-lo a novas implementações ou a chance da recuperação não poder ser feita caso essas adaptações não tenham sido previstas. Nesse sentido, é interessante aproveitar os recursos disponíveis no contexto local do destino executando as substituições apropriadas. A correção dessas substituições é fundamentalmente semântica, portanto é o programador da aplicação quem deve decidir permití-las ou não. Isto também permite evitar a criação de dependências remotas desnecessárias, que afetam a escalabilidade e a tolerância a falhas.

Relevante nessa etapa é o conceito de *reflexão*, que é a capacidade de um sistema de programação de observar e mudar o seu próprio comportamento. A migração heterogênea precisa do conhecimento do tipo de cada valor para poder executar a extração e serialização dos segmentos de memória corretamente. Em linguagens com funcionalidades reflexivas este problema pode ser resolvido diretamente. Entretanto, a maioria das linguagens compiladas não mantém informação de tipos em tempo de execução. Ainda, linguagens comuns como ANSI-C são *type-unsafe*, o que implica que é possível que o tipo

dinâmico de um valor em algum momento seja diferente de seu tipo declarado estaticamente. Soluções possíveis são restringir as características não seguras da linguagem (*unsafe features*) ou então modificar os compiladores de forma a lidar com essas características [SH98]. No entanto, ambas as soluções podem resultar em um comportamento fora do padrão da linguagem. Por outro lado, o acesso à pilha freqüentemente é restrito por questões de segurança. Como se verá, a falta de reflexão também afeta a restauração.

O fato de estarmos considerando ambientes de memória distribuída, implica na necessidade de transferir o estado todo da computação, incluindo os objetos alcançáveis dela, ou vinculá-la a referências remotas. Em conseqüência, uma semântica específica para identidade é necessária já que as computações podem se mover para fora e dentro do ambiente original e ainda serem consideradas a mesma computação.

A transferência do grafo de dependências da computação pode ser feita através de referências ou cópia. As referências podem ser remotas ou podem ser resolvidas localmente no destino através de um descritor. Resolver as referências no contexto local é uma solução atraente, mas requer informação do contexto no destino e, em conseqüência, alguma coordenação entre as plataformas de execução origem e destino. As referências remotas podem comprometer a escalabilidade e a tolerância a falhas. Por outro lado, a captura das dependências através de cópia está limitada pela complexidade e tamanho do fecho transitivo. O problema nesse caso consiste em achar a mínima quantidade de entidades que deveriam migrar com a computação, garantindo a correção do procedimento. Duas abordagens podem ser assumidas: chamadas de *rasa* (*shallow*) como em JavaGo [SMY99], ou *profunda* (*deep*) como na proposta de Tack et al. para a formalização da serialização e minimização de grafos em AliceML [TKS06]. Na primeira abordagem, o programador pode indicar as referencias migráveis explicitamente, sendo que as não migráveis serão anuladas. Além de requerer anotações adicionais na linguagem, essa solução aumenta o trabalho do programador e pode produzir *no-shows* durante a restauração. Por sua vez, a cópia profunda considera que todos os objetos relacionados devem ser migrados. O grafo de objetos é construído transparentemente para o usuário. Implica tipicamente em maior sobrecarga já que mais dados irão ser capturados e transferidos.

Existem sistemas que oferecem ao programador a flexibilidade de decidir como objetos particulares serão serializados, o que pode influir na restaurabilidade da computação. Exemplos dessa abordagem são o Coda [McAffer95] e o Pluto [Sunshine2005]. Eles se baseiam na noção de *descritores de marshalling*, que permitem a especificação do método de *marshalling* para cada objeto.

Para possibilitar a transferência do estado capturado, é necessário que antes da transmissão cada valor seja convertido a uma representação transmissível, dependendo do seu tipo. Este processo é chamado de *serialização*, *pickling* ou *marshalling*, enquanto que a operação oposta é chamada de *de-serialização*, *unpickling* ou *unmarshalling*. A informação a ser transferida, composta pelo estado de execução da computação, seu código e ambiente, pode se expressar na representação usada na origem ou no destino (usado, por exemplo, na estratégia *Receiver Makes Right (RMR)* [ZG95]), ou então na forma de uma representação independente da arquitetura, como no esquema *External Data Representations (XDR)* [RFC4506]. A alternativa de conversão no destino é preferida à conversão na origem porque no caso das duas arquiteturas serem iguais, este passo pode ser evitado. Em geral, a escolha do esquema de conversão é fixado na implementação do pacote de suporte a migração.

No caso do código, a representação independente da plataforma consiste no envio de código fonte ou interpretado. A recompilação do código fonte no destino elimina as implicações das diferenças entre as arquiteturas, mas provoca uma certa demora no reinício devido justamente ao processo de recompilação. O uso de linguagens interpretadas tem as vantagens da adaptação dinâmica e a portabilidade resultante da presença de uma máquina abstrata comum em ambos os lados, mas traz perdas de desempenho causadas pela interpretação quando comparado à execução de código compilado. Por outro lado, a alternativa de usar como representação o código de máquina da arquitetura destino obriga a manter um conjunto de arquivos executáveis para cada possível plataforma. Isto afeta a escalabilidade da aplicação, pois o arquivo executável apropriado para cada plataforma suportada precisa ter sido gerado previamente, e implica em alguma sobrecarga de armazenamento para manter esses arquivos.

2.2.2 De-serialização e restauração

Nesse passo, uma nova computação é criada a partir da informação transferida. Os vínculos aos recursos são reestabelecidos se necessário. Nesse ponto, erros podem surgir como: a detecção de estado incompleto ou inconsistente, impossibilidade de vínculos com recursos locais ou remotos, *overflow* de espaço ou memória, quebra do nó. Nesses casos, a migração deveria ser abortada e a execução original continuar normalmente.

A restauração requer a reconstrução do grafo de objetos e do estado interno da computação (incluindo pilha de chamadas, contadores de programa, variáveis locais) e sua instalação no espaço de memória de forma que a

execução possa ser reiniciada a partir da instrução seguinte à suspensão. O problema do restabelecimento do contador de programa não é trivial: nem todas as linguagens permitem estabelecer um ponto específico para continuar a execução. Também pode ser necessária a definição de contadores de programa lógicos por portabilidade. Java, por exemplo, facilita a execução do código recebido através da carga dinâmica de classes, mas não oferece suporte para continuar a execução a partir da última instrução executada. A migração de threads envolve adicionalmente problemas relacionados à sincronização: threads e locks devem ser restabelecidos numa determinada ordem.

Seguindo a restauração, a última etapa da migração seria o reinício da execução e, em se tratando de migração e não de clonagem, o cancelamento da computação original. As referências dessa computação podem por outro lado estar associadas a outras computações em curso e devem ser tratadas independentemente.

2.2.3 Características específicas das linguagens

O problema da captura e restauração no nível da aplicação está muito relacionado às características da linguagem. Para ilustrar como as características das linguagens influenciam nas implementações que precisam de captura e restauração, mostramos em seguida dois exemplos de linguagens das mais utilizadas para este tipo de implementação.

ANSI-C

A linguagem C oferece mecanismos que permitem ao programador "enganar" o sistema de tipos, mas podem levar a erros de inferência de tipos durante a extração dos dados na etapa de captura que podem fazer com que o programa não possa ser restaurado. Entretanto, esses não são os únicos fatores que tornam um programa C não restaurável. Em um estudo sobre migração no contexto do desenvolvimento do sistema TUI [Smith97, SH98], Smith and Hutchinson determinaram os problemas que provocam não restaurabilidade em programas escritos em ANSI-C em 4 classes:

- conflito de tipos: ocorre quando uma posição de memória tem mais de um tipo associado, de forma que o sistema não vai saber como extrair o dado corretamente. As características que provocam este problema são: *unions*, *casting de ponteiros*, não correspondência de parâmetros, reuso de armazenamento variável e falta de checagem de tipos na compilação independente;

- falta de informação de tipos: o compilador não gera ou não consegue gerar informação de tipos suficiente, por exemplo, em ponteiros de tipo *void*, listas de argumentos de tamanho variável e nas funções *setjmp* e *longjmp*;
- falsa identificação: tentativa de migrar dados que não são parte do programa. Ocorre, por exemplo, em casos de *casting* de valores inteiros a ponteiros, ponteiros que se referem a localizações de memória ilegais, *dangling pointers* e ponteiros não inicializados.
- incorreta conversão de valores: ocorre quando o dado é convertido incorretamente no destino. É devido a diferenças com a origem em arquitetura (operador *sizeof*, perdas na conversão de formatos) e conjunto de caracteres (significado de *char*).

Os problemas relacionados com a tipagem insegura da linguagem (*casting*, *unions*) tem sido evitados em algumas implementações [SH98, CS02] através de restrições na linguagem, ou seja, programando em estilo *type-safe*. Os programas que violam estas características são declarados não migráveis. Na realidade, este método é muito restritivo: programas *type-unsafe* podem ser migrados desde que estas características sejam detectadas e tratadas [Smith97]. Por outro lado, programas escritos em linguagens *type-safe* que façam chamadas ou usem variáveis dependentes do sistema no momento da migração podem ser não-migráveis.

Java

Apesar da linguagem Java ser uma das mais comumente usadas para a implementação de sistemas de agentes móveis, a informação sobre o estado interno que a máquina virtual disponibiliza é restrita [IKKW02].

O estado da pilha também não é portável: na maioria das JVMs ela é implementada como uma estrutura de dados nativa, ou seja, uma estrutura C [BHKP+04]. Isto faz com que a informação contida na pilha seja dependente da arquitetura. Para representar o estado da pilha em um formato independente da plataforma, é necessário um passo de tradução durante a serialização, assim como a operação contrária para a de-serialização. Isto implica em traduzir os valores das variáveis locais e operandos a valores Java, para o qual é necessário acesso ao tipo dos valores. No entanto, a informação de tipos está embutida no bytecode dos métodos que guardam os dados na pilha e não está disponível em tempo de execução. Usando a Java Debugging Architecture, é possível extrair o estado de execução [IKKW02]. Entretanto, não existe forma de restaurar o contador de programa. Existe um grupo de publi-

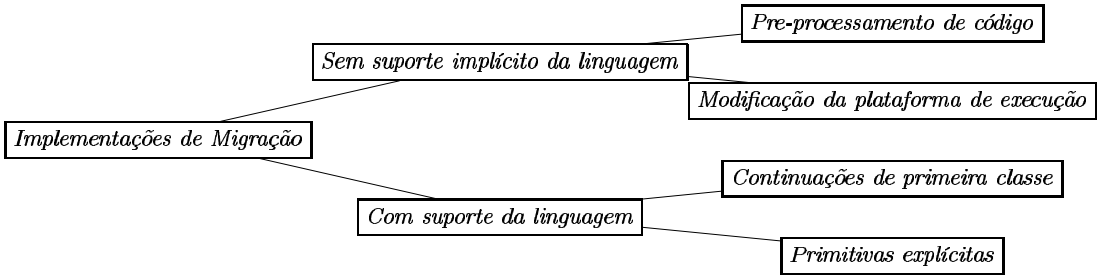

Figura 2.1: Métodos de implementação de migração heterogênea forte de computações

cações dedicadas à solução dos problemas relativos à migração de threads de Java [Funfrocken98, IKKW02, TRVC+00, BHKP+04]. As técnicas comumente descritas têm desvantagens no desempenho da serialização ou na portabilidade.

2.3 Classificação do suporte das linguagens para captura e restauração de estado

Como discutido em [MRS08], com relação ao suporte das linguagens de programação para a captura e restauração do estado de execução, as implementações podem ser divididas em dois grupos: os que se baseiam ou não em linguagens com este suporte, como mostra a figura 2.1.

2.3.1 Implementações sobre linguagens sem suporte para captura e restauração de estado

A seguir detalhamos os métodos mais utilizados para a implementação de captura e restauração de estado em aplicações de migração de computações em linguagens sem o necessário suporte. Finalizamos o capítulo analisando esses métodos e os problemas derivados da falta desse suporte.

Pré-processamento do código do usuário

A abordagem de *pré-processamento do código do usuário*, também chamado de *transformação fonte-fonte*, é um mecanismo de manipulação interna que consiste em modificar o programa do usuário inserindo fragmentos de código que permitem ao programa salvar o próprio estado de execução e reiniciar a computação por si próprio. O programa de usuário pode estar em formato fonte, compilado ou bytecode. O estado da computação é abstraído ao nível da linguagem, garantindo assim portabilidade.

O método mais usado é o que chamaremos de *stack unwinding* [SMY99, FCG00, BD01], que explicamos a seguir. A transformação inclui a inserção, em

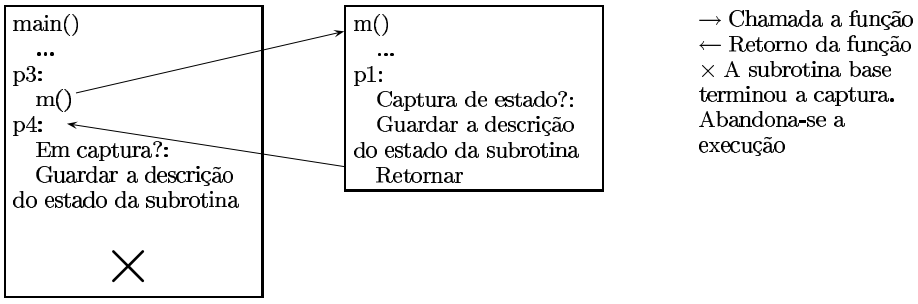

Figura 2.2: Mecanismo nativo de retorno de subrotinas: captura de estado

cada função, de código que registra todas as variáveis locais, parâmetros e a atualização do contador lógico de instruções, numa variável local ou objeto de *backup* criado para este propósito. A restauração geralmente requer a execução parcial da aplicação. Os estados que a aplicação atravessa durante o processo de captura e restauração são:

Execução Normal → Captura de Estado → Restauração de Estado → Execução Normal

A *captura* consiste em guardar o estado do procedimento corrente, retornar à subrotina que fez a chamada e repetir o procedimento recursivamente até chegar na subrotina principal, em que a aplicação termina. Este procedimento é ilustrado na figura 2.2. Na restauração, como aparece na figura 2.3, o código inserido no início do programa detectará o estado de restauração e cada subrotina da pilha armazenada será recursivamente chamada e restaurados os dados de seu frame local. Depois será executado um salto até o ponto de migração em que foi feita a captura da subrotina corrente. Para este mecanismo funcionar, é necessário que exista um ponto de migração antes e depois de cada chamada a subrotina. Uma variante desse método consiste no uso dos mecanismos

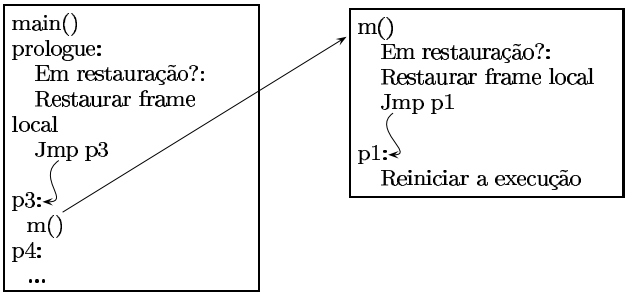

Figura 2.3: Mecanismo nativo de retorno de subrotinas: restauração de estado

de tratamento de exceções da linguagem. O compilador insere expressões *try-catch* no código fonte que executam a rotina de captura quando é lançado o evento correspondente. A captura da pilha se baseia na propagação do erro

ou exceção até o nível base. Esse método admite a captura de continuações parciais e delimitadas (fragmentos da computação) [SMY99].

Problemas particulares emergem em sistemas que oferecem persistência de threads. Na recuperação, todas as threads ativas durante a captura devem ser reiniciadas criando novas threads que serão inicializadas com os conteúdos dos seus pares na origem. Problemas de sincronização podem emergir durante a restauração devido à imprevisibilidade da situação de cada thread no momento da captura de estado. Este problema é tratado de formas diferentes, que vão desde restringir as computações a somente uma thread até oferecer anotações que indiquem que a thread está pronta para migrar.

Outro problema que deve ser tratado é o referente a como capturar e restaurar os valores temporários (resultados parciais) de operações interrompidas. Por exemplo, em:

d = f(x) + g(x)

se a computação é interrompida enquanto g(x) está sendo calculada, o resultado de f(x) deve ser capturado de sua localização temporária, e restaurado no destino para permitir continuar a computação. Isto implica que o número, tipo e localização desses temporários deve ser conhecido. Uma solução para este problema é inserir essa informação dentro do código durante a compilação. Outra abordagem é evitá-lo, dividindo a expressão e fazendo os temporários explícitos. Naturalmente, existe também a opção de não permitir a transformação durante a avaliação de uma expressão.

Este método pode ser enxergado como uma forma de *Continuation Passing Style (CPS)*. A esencia do CPS consiste em fazer explícitos aspectos comumente implícitos na continuação, como retornos de procedimentos e valores temporários. O CPS produz constantemente um valor mais a continuação da execução. Entretanto, obriga a adotar um estilo de programação que é considerado inconveniente e pode levar a erros.

Em comparação com a instrumentação de código fonte, trabalhar com o bytecode tem como vantagem maior disponibilidade do código, um conjunto de instruções mais amplo e melhor desempenho. O requisito de disponibilidade é uma séria desvantagem no pre-processamento de código fonte no caso de bibliotecas e código legado.

Modificação/extensão da plataforma de execução

A modificação/extensão da plataforma de execução é uma abordagem baseada em manipulação externa. O código do usuário não precisa ser modificado, e a abordagem é válida tanto para código nativo quanto interpretado. Entretanto, existem várias diferenças na forma de tratar ambas as represen-

tações. Linguagens interpretadas mantêm valores temporários na pilha, que são transferidos com o restante dos dados durante a migração. Por outro lado, na abordagem baseada em código nativo (ou a modificação do compilador), a informação relativa à localização e tipo dos valores temporários não está diretamente disponível. Esta informação poderia ser inserida pelo compilador se for convenientemente modificado. Basicamente, a abordagem de modificação/extensão da plataforma de execução no caso de código nativo consiste em modificar o compilador de forma a gerar informação para fazer os programas restauráveis. Isto requer também a implementação de uma plataforma de execução que execute as tarefas relacionadas com a migração/persistência da computação.

Em linguagens interpretadas, é a máquina virtual que é modificada de forma a executar as tarefas de migração/persistência. Nesse caso não é necessária uma plataforma adicional para este tipo de requisito. Um problema comum a várias máquinas virtuais é que a informação de tipos está embutida no bytecode dos métodos que guardam os dados na pilha, fazendo a inferência de tipos difícil. Quando se interpreta uma instrução, é possível saber o tipo dos dados envolvidos. Por este motivo, uma abordagem possível para a determinação do tipo dos valores na pilha é estender o interpretador de forma a manter uma estrutura paralela contendo esta informação. No entanto, esta solução não é compatível com *JIT* (*Just-In-Time compilation*) e gera sobrecarga em tempo de execução para a aplicação. A alternativa é analisar (*parse*) o bytecode somente na hora da serialização. O custo desta forma, é transferido como latência de serialização, fazendo com que este método seja mais aconselhável em aplicações em que estas operações sejam pouco freqüentes.

A grande desvantagem da abordagem de modificação/extensão da plataforma de execução é a portabilidade, pois todas as plataformas de execução envolvidas na aplicação podem precisar de modificações.

2.3.2 Linguagens com suporte para captura e restauração de estado

O suporte das linguagens para a captura e restauração heterogênea do estado de execução de computações pode ser caracterizado como a provisão de construções que permitam ambos procedimentos em um ambiente heterogêneo. Este suporte pode ser expressado tanto através de mecanismos como continuações de primeira classe –tipicamente oferecidas por linguagens funcionais– quanto por primitivas explícitas para manipulação de estado que permitam a construção de continuações.

A provisão de continuações de primeira classe permite a manipulação de continuações como estruturas explícitas que podem ser tratadas como valores. Quando este suporte se estende à extração e instalação heterogêneas da estrutura da continuação, a implementação de persistência e migração heterogêneas é direta. Usar continuações tem a vantagem de permitir implementar migração forte através de mecanismos de migração fraca. No entanto, continuações são do domínio quase exclusivo das linguagens funcionais, cujas limitações de desempenho são conhecidas. A alternativa de oferecer primitivas para manipulação de estado será provavelmente a escolha no caso de linguagens orientadas a objeto ou estruturadas, onde as continuações não são usualmente estruturas explícitas.

Discussão

As categorias de implementação, no caso de linguagens sem suporte implícito, podem ser comparadas em termos de sobrecarga em tempo de execução, sobrecarga representada pela migração, sobrecarga em espaço de armazenamento, portabilidade, completude e facilidade de manutenção.

O pré-processamento de código tem a propriedade da garantia de portabilidade. Por outro lado, implica mudanças no fluxo de programação, maiores sobrecargas de espaço e tempo de execução quando comparado a outros métodos, causadas pelas instruções inseridas. A sobrecarga da migração é também alta, devido ao processo de reconstrução da pilha, que simula uma execução normal.

A abordagem baseada na modificação da plataforma de execução exibe melhor desempenho. Nessa abordagem, não é necessário modificar o código do usuário, dessa forma não existem requisitos de disponibilidade de código. A grande desvantagem desse método é a perda de portabilidade.

Finalmente, exceto na transformação de código fonte, a manutenção do código apresenta problemas em todos os métodos. Esses métodos dependem de versões de bytecode e detalhes da plataforma de execução com os quais geralmente as linguagens não têm compromisso de compatibilidade entre versões. A completude também pode ser afetada: por exemplo, as cláusulas *try-catch* não podem ser migradas quando a implementação se baseia nos mecanismos de tratamento de exceções da linguagem.

Poucos trabalhos em serialização/de-serialização focam no problema da captura e restauração de execuções em ambientes heterogêneos no nível da linguagem. Como mostra esta revisão, o suporte para captura e restauração atualmente é majoritariamente opaco e não oferece formas de customização para requerimentos específicos. Deste estudo resulta claro que o suporte

das linguagens à captura e restauração é uma necesidade real quando se implementam aplicações que precisam de captura e restauração heterogênea de computações, como é o caso da migração.