Reprodução de Erros de Concorrência em Aplicações Java Através de Execução Simbólica

Manuel Bravo, Nuno Machado e Luís Rodrigues

INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa
angel.gestoso@ist.utl.pt, nuno.machado@ist.utl.pt, ler@ist.utl.pt

Resumo A depuração de erros de concorrência é difícil devido ao seu não-determinismo, que evita que a falta se manifeste em cada re-execução do programa. A gravação de todos os eventos não-deterministas da execução original permite ultrapassar este obstáculo, mas infelizmente penaliza a execução da aplicação. Neste artigo propomos o SIMBA, um sistema de reprodução determinista de erros de concorrência em aplicações Java, que alia técnicas de gravação parcial a mecanismos de inferência baseados em execução simbólica. O SIMBA grava apenas o resultado das instruções condicionais e a ordem das operações de sincronização. Esta informação é depois usada com o intuito de guiar a execução simbólica e tornar a fase de inferência mais eficaz e eficiente. O sistema resultante é avaliado através de uma bancada de testes que permite ilustrar as vantagens e limitações da solução proposta.

1 Introdução

A vulgarização das máquinas multi-processador durante a última década levou a um ponto de inflexão no processo de desenvolvimento de software, ao trazer a programação concorrente para o primeiro plano da indústria de software. Infelizmente, a natureza não-determinista dos programas concorrentes torna-os mais difíceis de desenvolver e depurar do que os seus homólogos sequenciais.

Por esta razão, um vasto trabalho de pesquisa tem sido dedicado ao desenvolvimento de ferramentas de depuração que ajudem os programadores a identificar e corrigir erros de concorrência (isto é, erros resultantes de um determinado intercalamento nos acessos a memória partilhada por dois ou mais fios de execução). De entre as soluções apresentadas, a técnica de *reprodução determinista* tem recebido especial atenção[1,2,3,4]. Esta técnica consiste em capturar eventos não-deterministas durante a execução errónea, usando depois o histórico resultante para forçar o erro durante a reprodução do programa. Em geral, o trabalho relacionado pode dividir-se em duas categorias: sistemas *baseados na ordem* de acessos partilhados e sistemas *baseados na pesquisa* de execuções.

Os sistemas baseados na ordem garantem a reprodução eficaz da execução original, dado que rastreiam o intercalamento completo dos acessos de escrita e leitura a posições de memória partilhada. Porém, a penalização introduzida pela gravação atómica de cada acesso partilhado é demasiado elevada, podendo levar a que o desempenho do programa seja cerca de 10x inferior[1,2,3].

Os sistemas de reprodução baseados na pesquisa registam apenas informação parcial durante a execução de produção e, posteriormente, aplicam *offline* técnicas de inferência para construir um intercalamento de fios de execução que seja verosímil e capaz de reproduzir o erro de concorrência. Contudo, ao reduzirem os custos de gravação, as técnicas baseadas na pesquisa podem não garantir reprodução determinista em ambientes multi-processador. A principal razão prende-se com o espaço de procura que, sem a informação da ordem exacta do intercalamento dos fios de execução, pode crescer de forma exponencial[5], tornando-se impraticável para aplicações reais complexas[6,7,8,4,9].

Neste contexto, o desafio está então em perceber qual o melhor compromisso entre a capacidade de garantir reprodução determinista e o custo de gravar informação durante a execução de produção. Neste artigo pretendemos dar um passo em frente para atingir este objectivo, propondo o Simba, um sistema de reprodução determinista de erros de concorrência em aplicações Java, que alia técnicas de gravação parcial a mecanismos de inferência baseados em execução simbólica. O Simba estende o Clap [9], introduzindo técnicas que permitem reduzir e simplificar significativamente a fase de inferência, através de um ligeiro aumento do custo de gravação.

Em suma, este artigo faz as seguintes contribuições: *i)* descreve o desenho e concretização do Simba, o (tanto quanto sabemos) primeiro sistema de reprodução determinista para aplicações Java que usa execução simbólica como mecanismo de inferência; *ii)* define um modelo de restrições, inspirado no do Clap, que simplifica a fase de inferência e reduz o tempo de pesquisa de um intercalamento de fios de execução capaz de reproduzir a execução errónea; *iii)* apresenta uma avaliação experimental do sistema usando bancadas de testes que evidenciam a eficácia e eficiência da solução (nomeadamente, mostra-se que o Simba, em média, consegue obter tempos de inferência 93x menores do que os do Clap, com um aumento médio de 27% no tempo de execução).

O resto do artigo está organizado da seguinte forma: a Secção 2 revê algum trabalho relacionado; a Secção 3 apresenta o sistema Simba; a Secção 4 descreve em pormenor o modelo de restrições; a Secção 5 avalia o protótipo desenvolvido e discute os resultados e, finalmente, a Secção 6 conclui o artigo.

2 Trabalho Relacionado

Várias soluções de reprodução determinista têm sido propostas com o intuito de providenciar reprodução determinista de erros de concorrência. Nesta secção, fazemos uma revisão daquelas que estão mais relacionadas com o nosso trabalho.

Soluções baseadas na ordem. O Jarec[1] regista a ordem de aquisição dos trincos, mas requer que o programa não tenha corridas de acesso a dados para garantir uma reprodução correcta, o que torna a solução pouco atractiva para a maioria das aplicações multiprocessador actuais. Mais recentemente, o Leap[2], o Order[3] e o Ditto[10] ultrapassam este obstáculo ao ajustar a granularidade com que os acessos a memória partilhada são agrupados, de modo a mitigar a contenção causada pela sincronização adicional inserida durante a instrumentação. Em particular, o Leap grava apenas a ordem local dos acessos a cada

variável partilhada, mas não distingue instâncias diferentes da mesma classe, incorrendo, por isso, em penalizações significativas quando há vários acessos partilhados a objectos distintos. Por sua vez, tanto o ORDER como o DITTO evitam este problema, pois rastreiam a ordem pela qual cada fio de execução acede a cada objecto, reduzindo assim a contenção na gravação dos acessos partilhados.

Soluções baseadas na pesquisa. O ESD[8] tem a particularidade de não gravar qualquer informação em tempo de execução, aplicando depois execução simbólica para sintetizar a falha do programa. Contudo, requer a exploração de todos os caminhos possíveis durante a execução simbólica, o que se torna impraticável para programas de grande dimensão. Para mitigar este problema, os autores apresentam também algumas heurísticas para sintetizar corridas a dados e interbloqueio. Por outro lado, o PRES[7] propõe uma técnica de reprodução probabilística que explora inteligentemente o espaço de intercalamentos possíveis através de um reprodutor com re-alimentação. Analogamente, o COOPREP[4] também relaxa a necessidade de se reproduzir o erro à primeira tentativa, em troca de uma fase de gravação mais eficiente. Para tal, assenta em gravação parcial cooperativa (realizada por múltiplos utilizadores independentes que correm o mesmo programa de software) e em técnicas estatísticas para combinar os registos parciais recolhidos. Por sua vez, o ODR[6] e o CLAP[9] guardam a informação do caminho percorrido (i.e. o resultado das instruções condicionais) pelos fios de execução e aplicam execução simbólica para gerar um modelo de restrições que satisfaça a informação gravada e reproduza o comportamento originalmente observado. Usando depois um *solver SMT*, produzem um intercalamento de fios de execução coerente com o modelo de restrições gerado.

O SIMBA é inspirado no CLAP mas, ao contrário deste último, está direccionado para a reprodução de execuções em programas escritos na linguagem Java, em vez de C/C++. Além disso, o SIMBA melhora o CLAP na medida em que reduz o espaço de procura que tem de ser explorado para encontrar uma solução que satisfaça o modelo de restrições. Isto é conseguido através da gravação da ordem de acesso aos pontos de sincronização do programa, embora com um ligeiro custo adicional durante a execução. A secção seguinte descreve em pormenor o sistema SIMBA, bem como as suas diferenças em relação ao CLAP.

3 Sistema SIMBA

Como já foi referido anteriormente, o SIMBA é um sistema de reprodução determinista de erros de concorrência em aplicações Java, baseado na gravação de informação parcial durante a execução e, posteriormente, na inferência da informação em falta através de execução simbólica. Nesta secção, apresenta-se o SIMBA, descrevendo em pormenor as várias fases em que opera.

A Figura 1 ilustra as várias fases de operação do SIMBA. Este sistema começa por efectuar uma *análise estática*, onde se identificam os pontos do código do programa a ser instrumentados (instruções condicionais e acessos a pontos de sincronização). Como resultado desta análise são geradas duas versões do programa, uma de gravação e outra de reprodução. A primeira é usada como entrada para a *fase de gravação*, que consiste em correr a aplicação registando

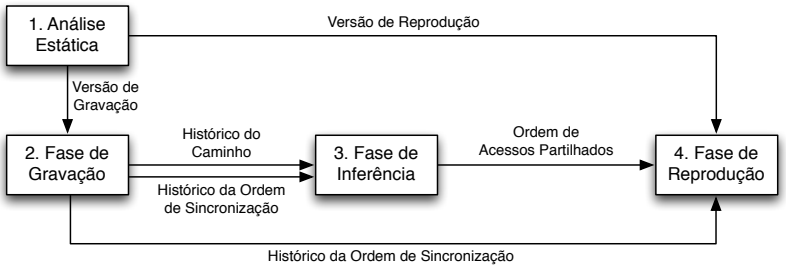

Figura 1. Fases de operação do SIMBA.

num ficheiro de histórico o caminho seguido por cada fio de execução, bem como a ordem de acesso aos pontos de sincronização do programa (e.g. trincos e monitores Java). Os registos produzidos servem depois como suporte para a etapa seguinte, denominada *fase de inferência*, que se caracteriza por ser a mais complexa e demorada, dado que tem como objectivo encontrar uma execução capaz de despoletar o erro no universo de todas as execuções possíveis.

Uma vez encontrada a solução (i.e. a ordem de acesso verosímil dos fios de execução às variáveis partilhadas do programa), dá-se início à *fase de reprodução*. Aqui, a execução errónea é reproduzida de forma determinista, permitindo assim ao programador aplicar técnicas de depuração tradicionais para encontrar a causa do erro. Nas secções seguintes, apresenta-se uma descrição mais pormenorizada de cada uma das fases acima referidas.

3.1 Análise Estática

Esta fase consiste em analisar estaticamente o *bytecode* do programa alvo, escrito em Java. O objectivo é gerar as versões de gravação e reprodução do programa, através da identificação e instrumentação das seguintes instruções:

– *Instruções condicionais.* Para se poder guiar a execução simbólica durante a fase de inferência, é necessário registar o caminho percorrido localmente por cada fio de execução. Isto é conseguido através da injecção de *testemunhas* (isto é, chamadas a um monitor), que permitem saber o ramo que foi tomado após cada instrução condicional. Na prática, a instrumentação é feita de forma a que, durante a execução do programa, se grave um valor booleano indicando o resultado de cada condição. Os blocos *switch* são instrumentados de forma análoga, embora neste caso seja registado um valor inteiro para indicar que instrução *case* foi executada.

– *Instruções de sincronização.* As operações de sincronização instrumentadas são os acessos a monitores (métodos e blocos *synchronized*), aquisições de trincos e chamadas a métodos *wait*/*signal*. Assim, para se conseguir gravar a ordem de sincronização dos fios de execução, insere-se uma chamada ao monitor do SIMBA *depois* das instruções *monitorentry/lock/wait/await* e *antes* das instruções *notify/notifyAll/signal/signalAll*. Por sua vez, na versão de reprodução, as

chamadas ao monitor são injectadas *antes* das instruções de sincronização, para evitar situações de interbloqueio.

– *Instruções de leitura/escrita em variáveis partilhadas.* O principal desafio na instrumentação dos acessos partilhados está em identificar estaticamente as variáveis que são acedidas de forma concorrente em tempo de execução. No SIMBA, abordamos este problema calculando uma sobre-aproximação através de uma análise estática de escape, denominada *ThreadLocalObjectAnalysis*[11] e concretizada usando a ferramenta Soot[1]. Cada acesso reportado como sendo partilhado é depois envolvido por chamadas ao monitor do SIMBA, permitindo assim forçar a ordem de intercalamento dos fios de execução pretendida durante a reprodução do erro. Note-se que a instrumentação das operações de leitura/escrita em variáveis partilhadas só se realiza na versão de reprodução, visto que o SIMBA não rastreia esta informação durante a fase de gravação.

3.2 Gravação

A fase de gravação consiste em executar a versão de gravação produzida na etapa anterior. Assim, à medida que executa o programa, o SIMBA vai gerando os históricos de caminho e de ordem de sincronização. No caso dos históricos de caminho, estes consistem num registo, para cada fio de execução, com os valores booleanos referentes aos resultados das instruções condicionais. De forma análoga, o histórico da ordem de sincronização é composto por um registo que associa, a cada fio de execução, um vector indicando a ordem cronológica dos seus acessos a pontos de sincronização. O algoritmo de gravação deste tipo de histórico assenta, tal como no DITTO[10], no uso de relógios lógicos para rastrear a ordem de sincronização. Concretamente, cada objecto do programa e cada fio de execução têm associado um relógio lógico. Quando um fio de execução t adquire o monitor de um objecto o, o SIMBA regista no histórico de t o valor actual do contador de o; no fim, ambos os relógios são actualizados.

3.3 Inferência

A fase de inferência tem como objectivo deduzir uma ordem de acesso a variáveis partilhadas que seja capaz de despoletar o erro, satisfazendo a ordem de sincronização e o caminho seguidos pelos fios de execução durante a execução original. Devido à complexidade desta fase, optamos por dividi-la em sub-fases, que se encontram ilustradas na Figura 2 e descritas de seguida.

Execução Simbólica. A execução simbólica no SIMBA considera como variáveis simbólicas apenas aquelas que são potencialmente partilhadas, permitindo que todas as outras variáveis do programa tenham valores concretos. Isto tem a vantagem de diminuir significativamente o espaço de estados possíveis. Além disso, o facto de o SIMBA ter informação sobre o caminho percorrido e a ordem de sincronização, permite-lhe não só explorar apenas um caminho durante a execução simbólica – o original, como também observar uma ordem de sincronização

[1] http://www.sable.mcgill.ca/soot/

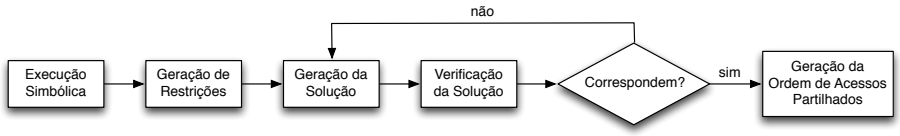

Figura 2. Fase de inferência em pormenor.

idêntica à da execução de produção. Mais precisamente, a execução simbólica tem como objectivo obter a seguinte informação:
– *Condição de caminho (CC):* nesta sub-fase recolhem-se restrições relacionadas com as condições de ramificação simbólicas. Assim, sempre que é tomada uma dada ramificação, a sua condição é traduzida numa restrição e adicionada à CC;
– *Acessos a memória partilhada por fio de execução:* na prática, representam a ordem local dos acessos de leitura/escrita que cada fio de execução faz;
– *Conjunto de escritas e conjunto de leituras por variável partilhada*;
– *Valores iniciais das variável partilhadas*;
– *Acessos a pontos de sincronização por objecto*: dado que a execução simbólica é guiada pelo histórico da ordem de sincronização, a ordem observada nesta fase respeita a ordem observada na execução errónea.

Geração de Restrições. Esta sub-fase usa toda a informação produzida pela execução simbólica e gera uma fórmula global. Esta fórmula é composta por uma série de restrições que representam o espaço de execuções possíveis, ao qual pertence a execução com o erro (descrita em pormenor na Secção 4).

Geração da Solução. Após terem sido geradas todas as restrições, a fórmula é passada a um *solver SMT (Satisfiability Modulo Theories)*. Porém, convém frisar que o objectivo aqui não é obter os valores concretos para cada variável simbólica, mas sim a ordem global de intercalamento das operações de escrita/leitura efectuadas pelos fios de execução.

Verificação da Solução e Geração da Ordem de acessos partilhados. Nestas sub-fases, o SIMBA usa a solução proposta pelo *solver* e executa a aplicação alvo de modo a verificar se o erro foi despoletado ou não. Caso a falha tenha sido reproduzida com sucesso[2], gera-se um histórico com a ordem de acessos partilhados e termina-se a fase de inferência. Caso contrário, pede-se ao *solver* que providencie uma nova solução e recomeça-se a verificação. Dado que a execução original pertence ao grupo de execuções possíveis, temos a garantia de que a sub-fase de verificação irá sempre terminar com sucesso.

3.4 Reprodução

A fase de reprodução consiste na re-execução da aplicação (usando a versão de reprodução previamente instrumentada), consultando o histórico produzido

[2] A confirmação do sucesso da reprodução do erro é obtida automaticamente através da captura da excepção lançada ou comparando o *output* obtido com o do erro.

na fase de inferência de modo a garantir que a ordem de acesso a variáveis partilhadas é capaz de reproduzir a falha. Dado que se corre uma versão instrumentada do programa, sempre que uma operação de leitura/escrita partilhada ou de sincronização está para ser executada, os métodos do monitor do SIMBA são invocados. O objectivo é bloquear um fio de execução sempre que este tenta executar uma operação fora da ordem prevista. Por outro lado, quando é dada permissão ao fio de execução de prosseguir com a operação (ou seja, a ordem indicada no histórico corresponde à ordem na execução), o histórico avança e os relógios referentes ao fio de execução e ao objecto em causa são actualizados. De notar que as operações de leitura/escrita partilhadas são protegidas por um trinco associado à instância do objecto, permitindo assim controlar, de forma eficiente, a ordem de execução de acordo com a indicada no histórico.

4 Modelo de restrições

O modelo de restrições do SIMBA é inspirado no do CLAP e caracteriza-se por ser bastante mais escalável do que abordagens anteriores (e.g. ODR[6]). Dado que o modelo se foca em restrições de ordenação de variáveis, cuja solução pertence a um domínio discreto finito, o *solver* não precisa de resolver restrições sobre condições complexas (tais como aritmética não-linear e strings), o que torna o processo de resolução muito menos oneroso.

Comparativamente ao CLAP, o modelo do SIMBA tem a vantagem adicional de ser significativamente mais simples, visto que não precisa de inferir a ordem de sincronização do programa. Por um lado, este facto permite-nos descartar todas as restrições relativas a acessos a pontos de sincronização, necessárias no modelo do CLAP (isto é relevante pois as restrições de sincronização possuem complexidade cúbica[9]). Por outro lado, permite-nos resolver directamente a fórmula de restrições. Isto não é possível no CLAP, visto que nem todas as suas soluções representam intercalamentos exequíveis e percorrer o espaço completo de soluções possíveis é impraticável para aplicações mais complexas. A fórmula final do SIMBA consiste, então, numa composição de restrições e pode ser escrita da seguinte forma:

$$\alpha = \alpha_{cc} \wedge \alpha_{rw} \wedge \alpha_{om} \wedge \alpha_{os} \wedge \alpha_{erro}$$

onde α_{cc} representa as restrições de caminho, α_{rw} as restrições de associação entre leituras e escritas, α_{om} as restrições de ordem de acessos a memória, α_{os} as restrições de ordem de acessos partilhados dentro de blocos de código sincronizados e α_{erro} a restrição que captura a manifestação do erro. De seguida, apresenta-se uma descrição pormenorizada de cada tipo de restrição.

Restrições de caminho (α_{cc}): representam o caminho de execução, onde cada restrição restringe o valor das variáveis simbólicas envolvidas na instrução condicional que a gerou. Este conjunto de restrições limita também o número de soluções para as restrições de leitura/escrita.

Restrições de leitura/escrita (α_{rw}): expressam a relação entre as operações de leitura e escrita para uma dada variável partilhada. Desta forma, o objec-

tivo é tentar descobrir para cada leitura r qual é a escrita w que a precede. Concretamente, as restrições para cada leitura são escritas da seguinte forma:

$$(V_r = init \bigwedge_{\forall w_j \in W} O_r < O_{w_j}) \bigvee_{\forall w_i \in W} (V_r = w_i \wedge O_{w_i} < O_r \bigwedge_{\forall w_j \neq w_i} O_{w_j} < O_{w_i} \vee O_{w_j} > O_r)$$

em que V_r é o valor lido, *init* é o valor inicial da variável partilhada, W é o conjunto de escritas para essa mesma variável, O_r determina a ordem de r e O_{w_i} a ordem a operação de escrita w_i. Se o valor atribuído a r for *init*, r tem de preceder todas as operações de escrita $w_i \in W$ (i.e. $O_r < O_{w_i}, \forall w_i \in W$). Caso contrário, a leitura terá de ser precedida por uma escrita (i.e. $\exists w_i \in W : O_{w_i} < O_r$) e o valor lido será o resultado dessa escrita ($V_r = w_i$); finalmente, a última condição indica que nenhuma outra operação de escrita poderá ter lugar entre elas.

Restrições de ordem de acessos a memória (α_{om}): representam a ordem local para todas as operações de leitura/escrita realizadas por um fio de execução. Assim, se um fio de execução efectuar r_j antes de w_b e w_a antes de r_j, a restrição correspondente será: $O_{w_a} < O_{r_j} < O_{w_b}$.

Restrições de ordem de acessos sincronizados (α_{os}): descrevem a ordem global dos acessos a variáveis partilhadas protegidas por um determinado objecto de sincronização (trinco ou monitor Java). De notar que, para as operações de leitura/escrita executadas dentro de um bloco sincronizado, podemos assumir ordem global entre todas os fios de execução. Desta forma, se a ordem de aquisição de um dado objecto de sincronização o for $< t_j, t_i, t_k >$ e, dentro do respectivo bloco atómico, cada fio de execução efectuar uma operação de acesso a uma variável partilhada (por exemplo, r_j, r_i e w_k, respectivamente), a restrição correspondente será: $O_{r_j} < O_{r_i} < O_{w_k}$.

Restrição do erro (α_{erro}): consiste na tradução do erro numa restrição. Permite restringir o conjunto de soluções possíveis da fórmula para aquelas em que o erro é despoletado. Em geral, a restrição de erro pode ser extraída do *core dump* quando o programa falha ou das asserções que foram violadas em tempo de execução. Por exemplo, um erro de divisão por 0 na instrução $\mathsf{val = 1/x}$, pode ser definido pela restrição de leitura $R_x == 0$.

4.1 Exemplo

A Figura 3 ilustra um pequeno exemplo que ajuda a compreender melhor como funciona o modelo de restrições do SIMBA, bem como a visualizar os seus benefícios em relação ao modelo do CLAP. Consiste num programa com dois fios de execução (T1 e T2), que contém acessos sincronizados e não sincronizados a uma variável partilhada x. Na figura, $R_x n$ ($W_x n$) representa uma leitura (escrita) sobre a variável partilhada x na linha de código n. De notar que, durante a execução simbólica, as operações de leitura dão origem a novas variáveis simbólicas (e.g. SYM_1 para $R_x 1$), enquanto as operações de escrita escrevem o valor concreto respectivo. A Figura 4 apresenta uma versão simplificada das restrições geradas pelo CLAP e pelo SIMBA para o programa da Figura 3, após a execução simbólica. $OR_x n$ representa a ordem da operação $R_x n$ no intercalamento que

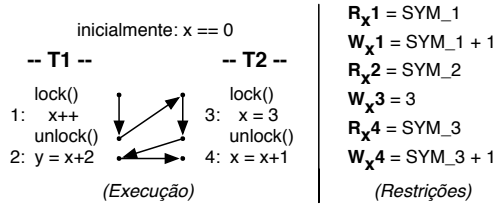

Figura 3. Exemplo de programa concorrente com dois fios de execução (T1 e T2).

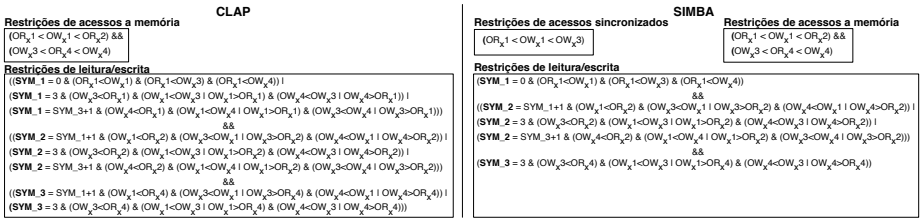

Figura 4. Restrições geradas pelo CLAP e pelo SIMBA para o exemplo da Figura 3.

irá ser calculado pelo *solver*. Como se pode verificar, o número de restrições geradas pelo SIMBA é substancialmente menor do que o do CLAP. Isto porque, por gravar as operações de sincronização, o SIMBA assume que a instrução x++ foi executada antes da instrução x = 3. Esta informação extra permite-lhe simplificar as restrições de leitura/escrita e, por conseguinte, o número de soluções possíveis (neste caso, três soluções para o SIMBA e doze para o CLAP). Além disso, convém frisar que o CLAP ainda necessita de inferir a ordem de aquisição dos trincos (estas restrições encontram-se omitidas na Figura 4).

5 Avaliação

A avaliação do SIMBA foca-se em três principais aspectos: *i) custo adicional*, em termos de penalização de desempenho e do tamanho dos históricos gerados; *ii) eficiência da inferência*, em termos do tempo necessário para resolver a fórmula de restrições e encontrar uma solução capaz de reproduzir o erro; e *iii) eficácia* na reprodução de erros de concorrência. O protótipo do SIMBA foi desenvolvido em Java, usando o Soot para a análise estática, o Java PathFinder[3] para efectuar a execução simbólica e o Z3[4] para resolver as restrições. Para se fazer uma análise comparativa com outras abordagens, implementou-se também uma versão simplificada do CLAP para aplicações Java, bem como uma versão do DITTO (sem a optimização de redução transitiva dos históricos gravados[10]). A comparação com o DITTO permite aferir os benefícios do SIMBA em relação a

[3] http://javapathfinder.sourceforge.net
[4] http://z3.codeplex.com

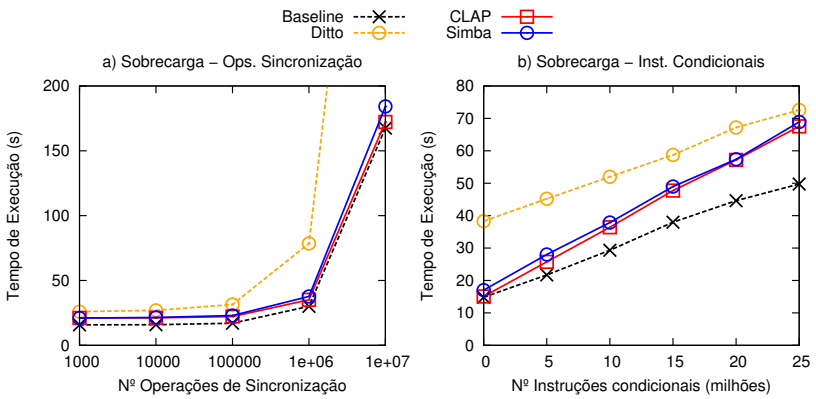

Figura 5. Sobrecarga.

um sistema de gravação completa dos acessos a memória partilhada. Para avaliar o custo de gravação recorremos a micro-testes e, para aferir os dois últimos critérios, usámos quatro programas com erros de concorrência da bancada IBM ConTest[12]. Todas as experiências foram feitas numa máquina de processador Intel Core 2 Duo a 2.26Ghz, com 4GB de RAM, correndo Mac OS X.

5.1 Sobrecarga de Gravação

Nesta secção pretendemos avaliar a nossa hipótese de que o SIMBA incorre num custo de gravação ligeiramente superior ao do CLAP (embora aceitável em relação ao tempo de execução normal do programa), mas substancialmente menor em comparação a uma abordagem puramente baseada na ordem.

Para tal, desenvolvemos micro-testes que nos permitissem aferir como a sobrecarga imposta pelas três soluções varia com o aumento intensivo do número de: *i)* operações de sincronização e *ii)* instruções condicionais. Em particular, os micro-testes consistem num programa concorrente com quatro fios de execução que efectuam 10^7 acessos a quatro variáveis partilhadas, sendo que alguns dos acessos são sincronizados e outros não. O objectivo dos micro-testes é permitir-nos ajustar facilmente um dos parâmetros (número de operações de sincronização ou número de instruções condicionais), mantendo o outro parâmetro constante.

Como se pode observar na Figura 5-a), com o aumento do número de operações de sincronização, a penalização de desempenho imposta pelo SIMBA aumenta ligeiramente em relação à imposta pelo CLAP, sendo esta diferença praticamente insignificante (até 6%). Além disso, a sobrecarga em relação à execução não-instrumentada (*Baseline*) é bastante baixa. Em contrapartida, o tempo de execução do DITTO cresce de forma exponencial, o que confirma a nossa hipótese acerca dos benefícios do SIMBA. No que toca ao número de instruções condicionais, a Figura 5-b) mostra que, à medida este valor aumenta, o custo adicional imposto pela gravação do SIMBA e do CLAP aproxima-se do custo do DITTO. Isto é explicado pelo facto do número de acessos partilhados ser constante durante as execuções deste micro-teste.

Programa	LdC	#FE	#VP	#IC	#Variáveis			#Restrições			Inferência	
					C	S	Red.	C	S	Red.	C	S
TwoStage	136	16	3	164	873	453	↓48.1%	2.592K	1.119K	↓71.4%	>2h	61s
Piper	165	21	4	160	2132	539	↓74.7%	737K	225K	↓69.5%	>2h	30s
TicketOrder	161	4	6	232	620	548	↓11.6%	199K	172K	↓13.9%	95s	40s
Manager	180	4	4	613	1286	1115	↓13.3%	60K	59K	↓2.4%	>2h	743s

Tabela 1. Resultados para a bancada IBM ConTest no CLAP (C) e no SIMBA (S). As células sombreadas indicam que nenhuma solução foi encontrada em 2h.

Apesar disso, deve ser notado que os históricos gerados pelo SIMBA são relativamente pequenos (máximo 257KB), sendo bastante inferiores aos gerados pelo DITTO (máximo 36MB).

5.2 Eficiência do Mecanismo de Inferência

Para comparar o SIMBA e CLAP em termos do tempo de inferência, usámos quatro programas da bancada IBM ConTest[12]. Esta bancada é composta por múltiplos programas com erros de concorrência. As colunas 2-5 da Tabela 1 descrevem os programas utilizados em termos de: linhas de código (*LdC*), fios de execução (*#FE*), variáveis partilhadas (*#VP*) e instruções condicionais (*#IC*). Por sua vez, as colunas 6-13 apresentam os resultados da comparação entre o SIMBA e o CLAP relativamente ao número de variáveis desconhecidas geradas, número de restrições e tempo de inferência (que engloba o tempo para construir a fórmula e o tempo para a resolver). Para os dois primeiros casos, a Tabela 1 também indica a respectiva redução obtida (*Red.*).

Analisando os resultados, podemos confirmar que o SIMBA reduz drasticamente tanto o número de variáveis como o número de restrições, o que, como seria de esperar, se traduz num decréscimo substancial do tempo de inferência. Concretamente, verifica-se que o SIMBA foi capaz de encontrar uma execução errónea em menos de 61s para os programas TicketOrder, TwoStage e Piper, enquanto, para os dois últimos casos, nem após duas horas o *solver* conseguiu descobrir uma solução para a fórmula gerada pelo CLAP. Além disso, convém realçar que estas reduções no tempo de inferência do SIMBA implicam um aumento da penalização de desempenho do programa em apenas 27% e 13%, respectivamente para os programas TwoStage e Piper.

Em suma, estes resultados suportam, uma vez mais, a nossa hipótese acerca dos benefícios de gravar a ordem de sincronização em tempo de execução.

5.3 Eficácia na Reprodução de Erros

A capacidade de reprodução de erros foi avaliada usando os quatro programas da bancada IBM ConTest. Para todos os casos, o SIMBA foi capaz de reproduzir o erro com sucesso à primeira tentativa (isto é, com a primeira solução obtida). Em contrapartida, o CLAP só foi capaz de reproduzir o erro para o programa Manager. Porém, note-se que, mesmo se o CLAP tivesse resolvido a fórmula no tempo máximo estipulado, não teríamos garantias que a solução obtida reproduziria o erro, pois a ordem de sincronização obtida poderia não ser plausível.

6 Conclusão

Neste artigo apresentámos o SIMBA, o primeiro sistema (tanto quanto sabemos) de reprodução determinista para aplicações Java que usa execução simbólica como mecanismo de inferência. O SIMBA melhora o estado da arte, ao estender o CLAP através da introdução de técnicas que permitem simplificar significativamente a fase de inferência, a troco de um ligeiro aumento do custo de gravação. Os resultados da avaliação feita com micro-bancadas e dois programas da bancada IBM ConTest, mostraram que o SIMBA consegue reproduzir erros de concorrência, reduzindo 93x, em média, o tempo de inferência do CLAP, com um aumento médio de 27% no tempo de execução.

Como trabalho futuro aponta-se a necessidade de se realizarem testes adicionais com aplicações mais complexas, bem como a implementação de um algoritmo de rastreio do caminho de execução mais eficiente (e.g. Ball-Larus[13]).

Agradecimentos Este trabalho foi parcialmente financiado pela Fundação para a Ciência e Tecnologia (FCT) através do financiamento pluri-anual do INESC-ID pelo programa PIDDAC e pelo projecto PEst-OE/EEI/LA0021/2013.

Referências

1. Georges, A., Christiaens, M., Ronsse, M., De Bosschere, K.: JaRec: a portable record/replay environment for multi-threaded Java applications. Software Practice and Experience **40** (May 2004) 523–547
2. Huang, J., Liu, P., Zhang, C.: LEAP: lightweight deterministic multi-processor replay of concurrent Java programs. In: ACM FSE. (2010) 385–386
3. Yang, Z., Yang, M., Xu, L., Chen, H., Zang, B.: ORDER: object centric deterministic replay for Java. In: USENIX ATC. (2011)
4. Machado, N., Romano, P., Rodrigues, L.: Lightweight cooperative logging for fault replication in concurrent programs. In: IEEE DSN. (2012) 1–12
5. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: ACM PLDI. (2007) 446–455
6. Altekar, G., Stoica, I.: ODR: output-deterministic replay for multicore debugging. In: ACM SOSP. (2009) 193–206
7. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K., Lu, S.: PRES: probabilistic replay with execution sketching on multiprocessors. In: ACM SOSP. (2009) 177–192
8. Zamfir, C., Candea, G.: Execution synthesis: a technique for automated software debugging. In: ACM EuroSys. (2010) 321–334
9. Huang, J., Zhang, C., Dolby, J.: Clap: recording local executions to reproduce concurrency failures. In: PLDI. (2013) 141–152
10. Silva, J.: Ditto - deterministic execution replay for Java virtual machine on multiprocessor. Master's thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa (2012)
11. Halpert, R., Pickett, C., Verbrugge, C.: Component-based lock allocation. In: IEEE PACT. (2007)
12. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: IEEE IPDPS. (2003) 286–293
13. Ball, T., Larus, J.R.: Efficient path profiling. In: ACM/IEEE MICRO. (1996) 46–57