Reprodução Probabilística de Execuções na JVM

João M. Silva e Luís Veiga
joao.m.silva@ist.utl.pt, luis.veiga@inesc-id.pt

Instituto Superior Técnico - UTL / INESC-ID Lisboa

Resumo. Programas concorrentes são bastante vulneráveis a falhas devidas a interações não antecipadas entre tarefas paralelas, cuja reprodução e resolução tendem a ser muito demoradas devido à ineficácia das metodologias de depuração convencionais neste contexto. Sistemas de reprodução determinística de execuções não-determinísticas têm dificuldades com *overhead* excessivo em multi-processadores. Descrevemos o trabalho em curso num sistema de reprodução de programas da *JVM* que usa uma abordagem probabilística e opera na máquina virtual. A ordem das operações de sincronização é rastreada e, embora insuficiente para reproduzir todas as execuções, é utilizada para realizar tentativas de reprodução da falha. As tentativas são rastreadas por completo, permitindo a sua reprodução garantida. Evitamos o *overhead* introduzido por um rastreio completo durante execuções de produção, sacrificando a eficiência e garantia da reprodução.

1 Introdução

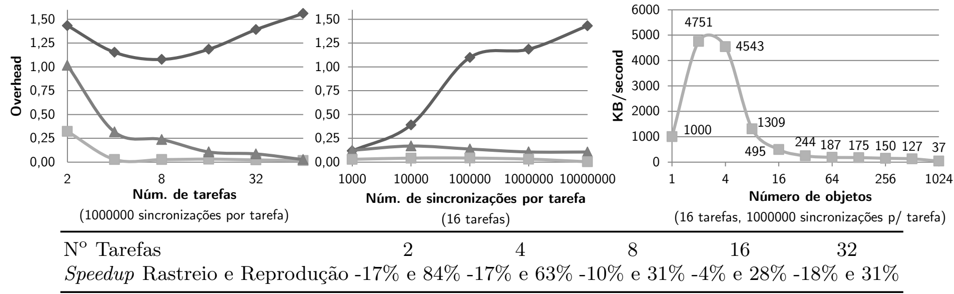
A depuração cíclica assume que o programa é determinístico em relação à falha, assumindo entradas iguais. Programas concorrentes não conformam com este pressuposto, visto apresentarem não-determinismo independente das entradas associado a condições de corrida entre tarefas paralelas. Neste contexto a depuração cíclica é extremamente ineficiente, já que a maioria do tempo é gasto a tentar reproduzir a falha [1]. Tentar obter mais informação sobre a falha pode ainda impor diferenças temporais que contribuam para a sua evasão (*probe effect*). Dada a ascensão das máquinas multi-processador e consequente ubiquidade de programação concorrente, surge a necessidade de melhores metodologias de depuração. Sistemas de reprodução determinística de execuções não-deterministas têm sido propostos como potenciadores do uso das metodologias convencionais em programas concorrentes. Os resultados dos eventos não-deterministas de uma execução são rastreados e utilizados para a reproduzir deterministicamente. Soluções que lidam com não-determinismo de memória, resultante das corrida de dados nos acessos à memória, têm dificuldades com *overheads* excessivos. Existem três abordagens que lidam com esta dificuldade: (a) reproduzir apenas corridas de sincronização, um subconjunto das corridas de dados, limitando o sistema a programas sem outras corridas [2, 3]; (b) usar *hardware* especializado [4, 5]; e (c) reproduzir execuções probabilisticamente, rastreandas-as parcialmente de modo a reduzir as tentativas de reprodução necessárias para que a falha se manifeste [6, 7]. O trabalho desenvolvido estende a *JVM JikesRVM* com um mecanismo de reprodução probabilística desenhado para suportar execuções em multi-processadores, com ocorrência de corridas de dados,

e para introduzir um *overhead* baixo em execuções de produção. É o primeiro reprodutor a operar dentro da *JVM* e introduz: (a) um mecânismo de deteção de reproduções da sincronização inconsistentes com o rastreio devido à ocorrência de corridas de dados; (b) algoritmos de reprodução mais concorrentes que muitas soluções existentes; e (c) novo algoritmo de reprodução de corridas de dados baseado em relógios lógicos.

2 Arquitetura e Processo

Soluções tradicionais de reprodução determinística recolhem informação suficiente durante o rastreio para reproduzir a execução com 100% de sucesso. Um reprodutor probabilistico evita o *overhead* associado a um rastreio completo relaxando a garantia de reprodução da falha e rastreando parcialmete a execução de produção. É explorado o *tradeoff* entre eficiência de rastreio e o número de tentativas necessárias para reproduzir uma falha. O nosso rastreio parcial é uma ordem parcial das operações de sincronização. Este é suficiente para reproduzir execuções sem corridas de dados, mas quando estas ocorrerem podem ser necessárias várias tentativas de reprodução. Cada tentativa é rastreada por completo, permitindo reprodução garantida assim que a falha se manifeste.

Reprodução da Sincronização. A *JVM* suporta todos os tipos de sincronização entre tarefas através de uma única primitiva, o monitor. É associado um relógio lógico (RL) a cada tarefa e a cada objeto. Se uma tarefa T_i executa uma operação de sincronização sobre o monitor do objeto O, os relógios são atualizados: $RL_{T_i} = RL_O = max(RL_{T_i}, RL_O) + 1$. As invocações a `start` e `join` propagam o relógio da tarefa mãe para a filha e da *joined* para a *joining*, respetivamente. O ficheiro de rastreio é constituído por sequências de intervalos, uma por tarefa, que refletem os valores que o seu relógio tomou, sendo comprimidos no mesmo intervalo incrementos unitários. É ainda introduzida sincronização na criação de tarefas para as identificar consistentemente entre execuções. O algoritmo de reprodução permite que tarefas com relógio igual ao mínimo no sistema executem, enquanto as restantes esperam. A execução fica organizada em épocas, uma por cada valor de relógio. O mínimo relógio no sistema é mantido eficientemente usando uma fila de prioridade em que as tarefas colocam o seu relógio atual. As tarefas sem relógio minímo esperam num monitor associada ao seu valor de relógio, sendo notificadas quando esse minímo for atingido. Dada a permissão para ocorrência de corridas de dados durante a reprodução, é necessario detetar execuções inconsistentes com o rastreio que sejam sua consequência. Menor ou maior número de sincronizações são detetadas garantindo que cada tarefa consome todo e apenas o seu rastreio. Quando uma sincronização é realizada sobre um objeto distinto do original, os intervalos rastreados são insuficientes para identificar a substituição, sendo necessário conhecer os objetos sem aumentar demasiado o rastreio. Associamos a cada intervalo uma *hash* construida a partir da concatenação de propriedades dos objetos que, para um intervalo I em que foram realizadas sincronizações nos objetos $O_0...O_n$, seria $hash_I = hash(RL_{O_0} + class_of(O_0) + ... + RL_{O_n} + class_of(O_n))$. Esta abordagem falha apenas quando o objeto substituto sofreu o mesmo número de operações de sincronização e é da mesma classe que o original.



Nº Tarefas	2	4	8	16	32
Speedup Rastreio e Reprodução	-17% e 84%	-17% e 63%	-10% e 31%	-4% e 28%	-18% e 31%

Fig. 1. Resultados do *microbenchmarking*.

Reprodução Completa. O rastreio completo é realizado fora de produção, pelo que pode impôr mais *overhead*, mas a reprodução deveria permitir paralelismo máximo para facilitar a depuração após reprodução da falha. Na *JVM* apenas os campos estáticos, de objetos e de *arrays* podem ser partilhados. O algoritmo de rastreio foi desenhado tendo em conta a diferença entre leituras e escritas, já que pares leitura-leitura não geram corridas. Mantemos um relógio lógico por tarefa, e estendemos os objetos com um relógio de leitura (RLL), um de escrita (RLE) e um contador de leituras desde a última escrita (CL). Quanto uma tarefa T realiza uma escrita num campo do objeto O, rastreia-se o tuplo (RLE_O, RLL_O, CL_O) e atualizam-se os relógios: $RLE_O = RL_T = max(TL_T, RLE_O, RLL_O)+1$, $CL_O = 0$. Se for uma leitura, atualizam-se os relógios: $RLL_O = RL_T = max(TL_T, RLE_O, RLL_O)+1$, $CL_O = CL_O + 1$; e, de seguida, rastreia-se o tuplo (RLE_O, RLL_O). Durante a reprodução, antes de uma escrita, T lê um tuplo (RLE, RLL, CL) do rastreio e espera até que (RLE_O, RLL_O, CL_O) iguale esse tuplo, esperando apenas pela escrita anterior e pelas leituras entre essa escrita e a atual. Os relógios e contador de O são atualizados usando as mesmas regras do rastreio. Antes de uma leitura, T lê um tuplo (RLE, RLL) do rastreio e espera até que $RLE_O = RLE$, i.e., até que a escrita anterior seja executada. Depois atualiza os relógios: $RLL_O = max(RLL_O, RLL)$, $CL_O = CL_O + 1$, $RL_T = RLL$.

3 Avaliação

A avaliação intermédia do sistema focou-se em *microbenchmarks*, cujos resultados se encontram na Fig. 1. Começamos por avaliar a *performance* do rastreio da sincronização em função nº de tarefas, nº de sincronizações e percentagem da execução gasta em sincronizações, num *benchmark* em que as tarefas executam simultaneamente e competem pelos monitores de um conjunto de objetos. Os resultados são encorajadores já que para os casos mais realistas (1% e 10% do tempo gasto em sincronizações) o *overhead* se mantem maioritariamente abaixo dos 20%, sendo que mesmo no pior caso (100% de sincronizações) se mantém relativamente baixo. Medimos também a largura de banda consumida pelo mesmo *benchmark* variando apenas a contenção no acesso aos monitores, concluindo que mesmo no seu pico é razoável para os discos atuais [6, 8]. Por último, medimos o *speedup* dos algoritmos de rastreio e reprodução completos em relação aos da sincronização, concluindo que atingimos os objetivos de desenho propostos: *speedup* negativo no rastreio devido à maior complexidade, mas positivo na reprodução, graças ao maior paralelismo.

4 Conclusões: Trabalho Relacionado e Futuro

Dos sistemas que reproduzem apenas as corridas de sincronização: o *DejaVu* [2] gera uma ordem global e o *JaRec* [3] uma parcial. O *LEAP* [8] reproduz todas as corridas de dados usando análise estática para reduzir as operações monitorizadas. As abordagens probabilísticas realizam um rastreio parcial da execução de produção para evitar *overheads* excessivos [7, 6]. O *ODR* usa informação parcial sobre entradas, ordem de *locks* e fluxo de execução para procurar uma execução com o mesmo *output.* O *PRES* [6] monitoriza incrementalmente: ordem da sincronização, chamadas de sistema, funções, blocos básicos e acessos à memória. É depois procurada uma execução que satisfaça condições especificadas pelo utilizador. Existe ainda trabalho em soluções que requerem *hardware* especializado[4, 5].

Apresentámos o trabalho em curso num sistema de reprodução determinística e probabilística para a *JVM* com suporte de execuções em multi-processadores e com corridas de dados. Já desenhámos e implementámos algoritmos de rastreio e reprodução de corridas de sincronização e de dados, modificando a máquina virtual *JikesRVM.* Pretendemos criar um motor de procura guiada. A avaliação mostra que o overhead é suficientemente baixo para execuções de produção.

Acknowledgements: This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/113613/2009, and PEst-OE/EEI/LA0021/2011.

References

1. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGOPS Oper. Syst. Rev. **42** (March 2008) 329–339
2. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. In: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools. SPDT '98, ACM (1998)
3. Georges, A., Christiaens, M., Ronsse, M., De Bosschere, K.: Jarec: a portable record/replay environment for multi-threaded java applications. Softw. Pract. Exper. **34** (May 2004) 523–547
4. Xu, M., Bodik, R., Hill, M.D.: A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In: Proceedings of the 30th annual international symposium on Computer architecture. ISCA '03, ACM (2003)
5. Montesinos, P., Ceze, L., Torrellas, J.: Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In: Proceedings of the 35th Annual International Symposium on Computer Architecture. ISCA '08, IEEE Computer Society (2008)
6. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. SOSP '09, ACM (2009)
7. Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. SOSP '09, ACM (2009)
8. Huang, J., Liu, P., Zhang, C.: Leap: lightweight deterministic multi-processor replay of concurrent java programs. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. FSE '10, ACM (2010)