Mateus Borges

Técnicas para Teste de Programas com Múltiplas Threads e Memória Compartilhada

10 de Setembro de 2014

Springer

Conteúdo

1

Introdução

Programas concorrentes são cada vez mais comuns. Dificuldades na dissipação do calor limitaram o crescimento da velocidade do clock dos processadores, e a solução encontrada foi acomodar múltiplos núcleos de processamento dentro de um mesmo chip. Abordagens alternativas, como chips de grafeno [17] ainda não se concretizaram. É necessário, portanto, que programadores aprendam a extrair o máximo das arquiteturas atuais, pois elas continuarão a ser o padrão pelo futuro próximo.

Porém, a escrita e depuração de programas concorrentes são notoriamente mais difíceis do que a de programas sequenciais [11]. Um dos erros mais comuns são *condições de corrida*, causados por acessos concorrentes da mesma posição de memória sem a devida ordenação onde pelo menos um dos acessos é uma operação de escrita. Tais erros são difíceis de se detectar (e corrigir), pois eles podem não se manifestar em todas as execuções do progama. Condições de corrida podem causar falhas no programa, resultando em acidentes graves no mundo real:

- O aparelho de radioterapia Theron-25 possuía uma condição de corrida que, em situações raras, resultava em um aumento de 100x da dosagem [9]. Cinco pessoas morreram e várias outras sofreram ferimentos graves.
- Uma condição de corrida no sistema de alerta do centro de controle da FirstEnergy retardou a resposta à uma falha em cascata na rede de energia, o que resultou em um blecaute que atingiu 50 milhões de pessoas [10].

1.1 Por que utilizar programação concorrente?

Programação concorrente é um requisito fundamental para extrair o máximo de performance das arquiteturas de processamento modernas [11, 8]: chips com múltiplos núcleos podem executar múltiplas partes do programa em paralelo, acelerando a execução. O ganho de tempo teórico máximo pode ser computado através da Lei de Amdahl [12], que afirma que o ganho de velocidade decorrente

do uso de múltiplos processadores em paralelo é limitado pelo tempo necessário para a execução das frações sequenciais do programa.

No entanto, alcançar estes ganhos teóricos não é algo fácil; o uso de funcionalidades existentes para programação concorrente sem o entendimento do hardware/linguagens pode causar resultados surpreendentes. A tabela 1.1 mostra os resultados da tarefa "pidigits" do Computer Language Benchmark Game [1] para as duas melhores implementações escritas em Java:

Tabela 1.1. Resultados para duas implementações em java para a tarefa pidigits

Threads?	CPU Time (s)	Elapsed Time (s)	Memory Consumption (MB)
Não	3.17	3.03	23.504
Sim	12.71	4.73	49.136

Ambas as implementações utilizam a biblioteca nativa GMP [2] para computar todos os dígitos de pi até a 10,000º casa decimal. A segunda implementação utiliza múltiplas threads para realizar cálculos em paralelo. Mesmo em uma máquina de quatro núcleos, a implementação com threads é mais lenta do que a implementação sequencial.

1.2 Não há bala de prata

Nas últimas décadas, várias funcionalidades foram propostas para tornar o desenvolvimento de programas concorrentes mais simples, como por exemplo *memória transacional* [38]. Algumas dessas tecnologias conseguiram carvar o seu nicho na indústria: Erlang, que utiliza passagem de mensagens ao invés de sincronização com travas, e sistemas de telecomunicações [39]. No entanto, essas tecnologias ainda estão em fase experimental ou o custo é bastante alto para ser usado de forma maciça pela indústria. Podemos citar como exemplo o uso de memória transacional, cujas melhores implementações em software implicam em uma redução consideráveis de performance quando comparadas com travas. Recentemente, as instruções de suporte à memória transacional em chips da Intel foram desativadas devido à erros na sua implementação [13].

Mesmo que algo revolucionário apareça nos próximos anos, dificilmente haverá uma migração maciça por parte da indústria. O código legado dos sistemas existentes precisa ser mantido; basta olhar para as estatísticas atuais do uso de COBOL [14] (aproximadamente 200 bilhões de linhas de código em operação) para compreender o tamanho do esforço em atualizar estes sistemas. Portanto, o desenvolvimento de técnicas eficientes para prevenir e detectar erros em programas concorrentes é algo extremamente relevante.

[1] http://benchmarksgame.alioth.debian.org/u64q/performance.php?test=pidigits
[2] https://gmplib.org/

1.3 Objetivo e organização

Esta monografia não deve ser visto como um documento exaustivo; o objetivo desta monografia é fazer um sobrevôo pelo estado da arte das técnicas e ferramentas para a atividade de teste e validação de software concorrente. O foco é em técnicas que possam ser aplicadas à programas que utilizam funcionalidades de concorrência tradicionais, como travas e threads, e que possam ser utilizadas por programadores comuns. Não iremos cobrir métodos formais; provar a corretude de um software demanda muito mais tempo e dinheiro [7] do que a maioria das empresas dispõe para os seus projetos.

O resto da monografia está organizado da seguinte forma: O segundo capítulo expõe as dificuldades existentes no desenvolvimento e testes de programas concorrentes. O terceiro capítulo apresenta ferramentas e técnicas para auxílio na atividade de teste e validação que fazem parte do estado da arte.

2

Dificuldades no desenvolvimento de programas concorrentes

2.1 Dificuldades na escrita de programas concorrentes

Programas concorrentes são compostos por múltiplas threads ou processos, onde cada uma dessas unidades é reponsável por executar certos trechos do programa. A ordem de execução destas unidades é determinada pelo *escalonador*, um módulo do runtime da linguagem ou do sistema operacional responsável por alocar recursos do hardware (neste caso específico, tempo de processamento). Idealmente, os resultados do programa devem ser independentes das decisões do escalonador, pois a ordem de execução das threads/programas pode ser diferente entre execuções. [1]

Na prática, é bastante provável que haja defeitos na sincronização das operações: programadores estão habituados a pensar de forma sequencial [5], de forma que é comum acontecerem erros na escrita de de programas concorrentes. Consequentemente, é possível que o programa tenha comportamentos diferentes em execuções distintas com as mesmas entradas. Um exemplo clássico são as *condições de corrida*: se múltiplas threads escrevem em uma variável compartilhada, o valor armazenado será o que a última thread escreveu. Caso esta variável seja lida em uma operação condicional, o caminho tomado pelo programa irá depender diretamente de qual foi a última thread a escrever na variável. Outros erros frequentemente cometidos são:

- *Deadlocks* : Um grupo de threads/processos está em deadlock quando cada membro está com o controle de um determinado recursos/trava, e está esperando a liberação de outro recurso/trava possuído por outro membro do grupo.

[1] Note que um programa pode estar correto mesmo apresentando comportamento não determinístico; considere por exemplo um cliente de email (corretamente sincronizado) com duas threads competindo para enviar mensagens. Na ausência de um requisito de ordenação nos emails enviados, qualquer um dos resultados seria válido.

- *Violações de atomicidade* : Falha em incluir acessos de memória que deveriam acontecer atomicamente na mesma seção crítica [16].

2.2 Teste de software concorrente

As atividades de Teste e Análise de Software fazem parte do chamado *processo de qualidade*. Um dos principais objetivos deste processo é buscar e revelar faltas existentes no software [15]. No entanto, tentar revelar e remover todas as faltas irá garantir apenas que as atividades de qualidade sejam realizadas eternamente. É necessário um equilíbrio de forma a atender aos requisitos do projeto, como custos (Teste e Análise podem correspodender à 75% dos gastos de desenvolvimento) ou prazos iminentes. Além disso, falhas mais brandas são mais facilmente toleradas pelos usuários do que falhas críticas.

A metodologia de validação mais popular na indústria é o uso de testes [18]. Além de auxiliar na detecção de erros, testes provêem uma fundação segura para refatoramentos, como também servem de documentação para o programa. Metodologias ágeis como Extreme Programming [19] utilizam a escrita de testes como um dos seus pilares fundamentais.

O uso de concorrência, no entanto, aumenta consideravelmente a dificuldade no uso de testes como ferramenta para encontrar erros [2, 21, 22]. Não é nenhuma surpresa que os fatores mencionados na seção 2.1 também afetam as atividades de teste e validação. A manifestação de falhas relacionadas a concorrência requer que pelo menos duas condições sejam atendidas:

1. A entrada do programa deve provocar a execução do código defeituoso.
2. O escalonamento [2] das threads selecionado pelo escalonador deve ser capaz de ativar o defeito, assim infectando o estado do programa.

Reproduzir um falha causada por um defeito de concorrência não é uma atividade simples, pois é necessário considerar o escalonamento como uma nova fonte de entrada do programa.

2.2.1 Stress Testing

A prática mais comum para o teste de programas concorrentes é o uso de *stress testing*: executar o mesmo teste por um longo período de tempo, ou um número grande de vezes [11, 20]. O raciocínio é simples: Como não é possível ter o controle total do escalonador, a solução mais direta é executar o teste várias vezes. A esperança é que execuções distintas exercitem diferentes entrelaçamentos [3].

Stress testing, no entanto, não é uma técnica efetiva para a detecção de bugs. O espaço de possíveis entrelaçamentos das threads possui um tamanho

[2] em inglês, *schedule*
[3] *interleavings*

exponencial; explorar completamente todos os entrelaçamentos de programas de tamanho médio é algo intratável. Nem todos os cronogramas possuem a mesma probabilidade de acontecer; certas falhas podem ser reveladas apenas após semanas de testes quando um entrelaçamento bastante raro é executado. Finalmente, resultados experimentais mostram que *stress testing* é particularmente ruim para revelar violações de atomicidade [20]: a maior parte das execuções cobre exatamente os mesmos entrelaçamentos capazes de provocar este tipo de erro.

2.2.2 Testando cenários específicos

Em certas situações pode ser desejável criar testes para validar cenários específicos, como por exemplo, garantir que a thread A acabe sua execução após a thread B, ou vice versa. Adicionar sincronização extra no programa para fins de teste não é algo desejável, pois pode mascarar erros reais no programa original [21], entre outros problemas. No entanto, sem o controle do escalonamento é impossível escrever asserções precisas, pois execuções distintas do mesmo teste poderiam resultar em cenários indesejados e causar *falsos positivos*[4].

Para este fim, o método mais utilizado por desenvolvedores é a inserção de atrasos no código do teste [18, 22, 23] (como `Thread.sleep` em Java) para pausar temporariamente as threads de forma a forçar a ocorrência do escalonamento desejado. O desenvolvedor precisa estimar o tempo da pausa via tentativa-e-erro, testando diferentes valores até que o teste passe de forma consistente. Como a duração da pausa é dependente do ambiente de execução (software + hardware), escalonamentos não desejados podem ocorrer quando o teste é executado em um ambiente diferente, como um servidor de integração contínua. Por fim, o uso de pausas torna a execução do teste mais lenta.

[4] O teste "revelou" a existência de um erro que não existe no código

3

Técnicas e ferramentas para teste de sistemas concorrentes

3.1 Escalonamento Randomizado

Uma das principais limitações de stress testing é que execuções repetidas sob as mesmas condições tendem a cobrir entrelaçamentos de threads similares. Uma maneira de aumentar o número de entrelaçamentos distintos cobertos é *randomizar* as decisões do escalonador. A ferramenta ConTest [24] insere invocações de `sleep()` e `yield()` aleatoriamente em pontos de sincronização e acessos à memória compartilhada. PCT [26] manipula de forma disciplinada as prioridades das threads durante as execuções do teste, provendo certas garantias estatísticas para a detecção de bugs. No entanto, nenhuma das duas técnicas garante que entrelaçamentos raros serão exercitados.

3.2 Teste Sistemático de Concorrência

Em completa oposição às abordagens anteriores, técnicas de teste sistemático exploram todos os possíveis escalonamentos válidos [21, 27, 28, 29, 30]. Todos os escalonamentos distintos do programa para uma determinada entrada são enumerados através de técnicas de *verificação de modelos de software* [1] [25]. O programa sob teste é então executado repetidas vezes, porém cada execução é controlada de forma a percorrer um dos escalonamentos gerados anteriormente. Esta exploração sistemática do espaço de entrelaçamentos das threads do programa aumenta as chances de encontrar erros nos testes existentes.

Uma das implementações mais populares desta técnica é a ferramenta CHESS [21], desenvolvida pela Microsoft. CHESS intercepta o controle do programa de forma transparente para o desenvolvedor, utilizando instrumentação binária para capturar invocações à API de concorrência. Informações sobre os pontos de não-determinismo do programa são capturadas e enviadas para um escalonador especial, que irá computar os entrelaçamentos possíveis das

[1] *software model checking*

threads. Para verificar se há erros em um dos entrelaçamentos enumerados, o teste é executado novamente. Nesta nova execução, o escalonador do CHESS obriga que apenas uma thread possa executar ao mesmo tempo, forçando a cobertura do escalonamento selecionado.

Percorrer exaustivamente todos os possíveis entrelaçamentos é algo impossível. A exploração é encerrada quando algum critério, como tempo ou número de execuções, é atingido. Outras otimizações para redução do custo da busca exaustiva que foram adotadas são:

- *Stateless Model Checking*: Verificadores de modelo tradicionais são capazes de armazenar e restaurar estados intermediários da execução do programa, porém à um custo alto. CHESS evita este custo ao armazenar apenas os estados correspondentes às decisões de escalonamento do programa. A consequência é que o teste precisa ser re-executado do início para alcançar o estado desejado.
- *Redução de ordem parcial*: A ordem relativa das threads em uma execução do programa concorrente é representada em grafo de relações *aconte-antes*[2]. Duas execuções que possuem grafos idênticos são equivalentes. CHESS ignora entrelaçamentos cuja execução seja equivalente à de algum outro já explorado.
- *Limite de preempção*: CHESS prioriza escalonamentos onde tenha ocorrido o menor número de interrupções de execução possível. A idéia desta heurística é que muitos dos bugs podem ser revelados com apenas um pequeno número de preempções.

[27] investiga o uso de uma *busca seletiva*, que utiliza restrições de ordenamento aprendidas com execuções corretas para direcionar a busca para entrelaçamentos capazes de violar tais restrições. [28] apresenta resultados empíricos de diferentes estratégias de busca com limites [3]; um das afirmações mais interessantes é que um escalonador aleatório tem resultados tão bons quanto um que use limites de escalonamento.

3.3 Teste ativo

Técnicas de *predição* de erros de concorrência são capazes de encontrar rapidamente violações que são difíceis de ser detectadas através de outras abordagens de teste, como teste sistemático [34, 35, 36]. Por exemplo, RacerX é capaz de analisar programas de 1,800 KLOC em busca de condições de corridas e deadlocks em menos de 15 minutos. No entanto, estas ferramentas podem retornar falsos positivos. Isso significa que os desenvolvedores precisam analisar manualmente cada um dos alertas reportadas, o que é algo tedioso e propenso a erros.

[2] happens-before
[3] *Bounded Search*

Uma maneira de automatizar o processo de confimação dos alertas é o uso de técnicas de *teste ativo* [6, 20, 31, 33]. Teste ativo é realizado em dois passos: Os entrelaçamentos suspeitos retornados pelas ferramentas de predição são coletados, e depois enviados para um escalonador ativo que irá tentar exercitá-los em execuções do programa. Caso uma falha seja manifestada, o alerta reportado corresponde à um bug; do contrário, nada pode ser afirmado. Isso implica que técnicas de teste ativo **nunca** retornam falsos positivos/negativos. Em compensação, os tipos de bugs que podem ser detectados são limitados aos que podem ser encontrados com a técnica de predição utilizada.

CTrigger [20] e PENELOPE [31] são abordagens de teste ativo para a detecção de violações de atomicidade. CTrigger analisa traces de diversas execuções do program em busca de *entrelaçamentos não-serializáveis*, um tipo particular de entrelaçamento correlacionado com a ocorrência de ocorrência de violações de atomicidade. Em um segundo momento, entrelaçamentos que não podem ocorrer são removidos, e os remanescentes são ranqueados de acordo com heurísticas próprias e depois executados. PENELOPE opera de maneira similar, porém os possíveis escalomentos defeituosos são computados a partir de uma execução abstrata do programa. Esta execução abstrata é uma versão simplificada da execução original, composta por operações de escrita/leitura em variáveis compartilhadas e eventos de sincronização. Devido à diferenças entre as linguagens suportadas das duas ferramentas, nenhuma comparação entre as mesmas foi realizada.

Maple [4] é similar em funcionamento à uma técnica tradicional de teste ativo, porém ela busca maximizar critérios de cobertura concorrente do programa [37] ao invés de determinar se os alertas reportados por um analisador preditivo são verdadeiros. A hipótese adotada é que a maior parte dos falhas relacionados à concorrência podem ser expostas se a dependência de memória correta for exercida.

Outras ferramentas de teste ativo incluem: RaceFuzzer[6], que suporta a detecção de condições de corrida; [33], um framework para a implementação de análises preditivas dinâmicas e triagem automática dos bugs reportados.

3.4 Controle de concorrência

Como visto na seção 2.2.2, poder testar um escalonamento específico é algo importante para garantir a relevância e a portabilidade dos testes escritos. [23] propõe o uso de Aspectos [4] para notificar à thread executando o teste que o estado do programa desejado (para a realização de asserções, por exemplo) foi alcançado. IMunit [22] permite que o desenvolvedor escolha o escalonamento a ser testado através de uma sequência de eventos especificada com o auxílio de anotações de código especiais. As semântica definida pelas anotações é um fragmento de *lógica linear temporal*. CONCURRIT [18] utiliza uma abordagem

[4] http://www.eclipse.org/aspectj/

cooperativa: o desenvolvedor especifica restrições sobre o conjunto de possíveis escalonamentos, e um verificador de modelos explora todas as execuções que atedem às restrições.

Referências

1. Brito, Maria, et al. "Concurrent Software Testing: A Systematic Review."on Testing Software and Systems: Short Papers (2010): 79.
2. Zeller, Andreas. Why programs fail: a guide to systematic debugging. Elsevier, 2009.
3. Young, Michal. Software testing and analysis: process, principles, and techniques. John Wiley & Sons, 2008.
4. Yu, Jie, et al. "Maple: a coverage-driven testing tool for multithreaded programs."ACM SIGPLAN Notices. Vol. 47. No. 10. ACM, 2012.
5. Lu, Shan, et al. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics."ACM Sigplan Notices. Vol. 43. No. 3. ACM, 2008.
6. Sen, Koushik. "Race directed random testing of concurrent programs."ACM SIGPLAN Notices. Vol. 43. No. 6. ACM, 2008.
7. Holzmann, Gerard J. "Mars code."Communications of the ACM 57.2 (2014): 64-73.
8. Cantrill, Bryan, and Jeff Bonwick. "Real-world concurrency."Queue 6.5 (2008): 16-25.
9. Leveson, Nancy G., and Clark S. Turner. "An investigation of the Therac-25 accidents."Computer 26.7 (1993): 18-41.
10. Software Bug Contributed to Blackout, SecurityFocus 2004-02-11. http://www.securityfocus.com/news/8016
11. McKenney, Paul E. "Is parallel programming hard, and, if so, what can you do about it?."Linux Technology Center, IBM Beaverton (2011).
12. Herlihy, Maurice, and Nir Shavit. The Art of Multiprocessor Programming, Revised Reprint. Elsevier, 2012.
13. Errata prompts Intel to disable TSX in Haswell, early Broadwell CPUs, TechReport August 12, 2014 http://techreport.com/news/26911/errata-prompts-intel-to-disable-tsx-in-haswell-early-broadwell-cpus
14. COBOL – continuing to drive value in the 21st Century. http://www.microfocus.com/000/COBOL_continuing_to_drive_value_in_the_21st_Century_tcm21-23652.pdf
15. Young, Michal. Software testing and analysis: process, principles, and techniques. John Wiley & Sons, 2008.

16. Lucia, Brandon, et al. "Atom-aid: Detecting and surviving atomicity violations."Computer Architecture, 2008. ISCA'08. 35th International Symposium on. IEEE, 2008.
17. Liu, Guanxiong, et al. "Graphene-based non-Boolean logic circuits."Journal of Applied Physics 114.15 (2013): 154310.
18. Burnim, Jacob, et al. "CONCURRIT: testing concurrent programs with programmable state-space exploration."Proceedings of the 4th USENIX conference on Hot Topics in Parallelism. 2012.
19. Beck, Kent. Extreme programming explained: embrace change. Addison-Wesley Professional, 2000.
20. Park, Soyeon, Shan Lu, and Yuanyuan Zhou. "CTrigger: exposing atomicity violation bugs from their hiding places."ACM Sigplan Notices 44.3 (2009): 25-36.
21. Musuvathi, Madanlal, et al. "Finding and Reproducing Heisenbugs in Concurrent Programs."OSDI. Vol. 8. 2008.
22. Jagannath, Vilas, et al. "IMUnit: improved multithreaded unit testing."Proceedings of the 3rd International Workshop on Multicore Software Engineering. ACM, 2010.
23. Dantas, Ayla, Francisco Brasileiro, and Walfredo Cirne. "Improving automated testing of multi-threaded software."Software Testing, Verification, and Validation, 2008 1st International Conference on. IEEE, 2008.
24. Edelstein, Orit, et al. "Multithreaded Java program test generation."IBM systems journal 41.1 (2002): 111-125.
25. Jhala, Ranjit, and Rupak Majumdar. "Software model checking."ACM Computing Surveys (CSUR) 41.4 (2009): 21.
26. Burckhardt, Sebastian, et al. "A randomized scheduler with probabilistic guarantees of finding bugs."ACM Sigplan Notices. Vol. 45. No. 3. ACM, 2010.
27. Wang, Chao, Mahmoud Said, and Aarti Gupta. "Coverage guided systematic concurrency testing."Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011.
28. Thomson, Paul, Alastair F. Donaldson, and Adam Betts. "Concurrency testing using schedule bounding: an empirical study."Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, 2014.
29. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. Software Tools for Technology Transfer (STTT), 2(4), 2000.
30. P. Godefroid. Software model checking: The VeriSoft approach. Formal Methods in System Design, 26(2):77–101, 2005.
31. Sorrentino, Francesco, Azadeh Farzan, and P. Madhusudan. "PENELOPE: weaving threads to expose atomicity violations."Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2010.
32. Huang, Jeff, Patrick O'Neil Meredith, and Grigore Rosu. "Maximal sound predictive race detection with control flow abstraction."Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2014.
33. Joshi, Pallavi, et al. "Calfuzzer: An extensible active testing framework for concurrent programs."Computer Aided Verification. Springer Berlin Heidelberg, 2009.
34. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In ICSE, pages 386–396, 2009.

35. J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In ESEC/SIGSOFT FSE, pages 205–214, 2007.
36. D. R. Engler and K. Ashcraft. Racerx: effective, static detec- tion of race conditions and deadlocks. In SOSP, pages 237– 252, 2003.
37. Hong, Shin, et al. "Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation."Software Testing, Verification and Reliability (2014).
38. Cascaval, Calin, et al. "Software transactional memory: Why is it only a research toy?."Queue 6.5 (2008): 40.
39. Armstrong, Joe. Programming Erlang: software for a concurrent world. Pragmatic Bookshelf, 2007.