UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

João Paulo Taylor Ienczak Zanette

LIMITAÇÕES DE GERADORES DE TESTES ALEATÓRIOS NA VERIFICAÇÃO DE E TESTE DE *MULTICORE CHIPS*

Florianópolis
2018

João Paulo Taylor Ienczak Zanette

LIMITAÇÕES DE GERADORES DE TESTES ALEATÓRIOS NA VERIFICAÇÃO DE E TESTE DE *MULTICORE CHIPS*

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Luiz Claudio Villar dos Santos

Florianópolis
2018

RESUMO

Com o aumento do número de *cores* de um *chip*, o *hardware* que gerencia o sistema de memória compartilhada torna-se cada vez mais complexo e suscetível a erros de projeto. Por isso é importante testar se a o projeto do *hardware* está correto, para evitar que erros acabem se propagando para o *chip*, sendo detectados apenas no protótipo fabricado. Por isso, a verificação pré-silício do sistema de memória compartilhada é crucial para a redução de custos de desenvolvimento. No entanto, como a verificação pré-silício precisa executar programas de teste em um simulador (e não no *hardware* real), não é possível usar programas muito longos para expor erros de projeto. Por isso, as atuais técnicas de verificação tem se mostrado frequentemente incapazes de encontrar erros sutis. Parte dessa incapacidade vem da dificuldade de se gerar testes aleatórios adequados. Este trabalho propõe a comparação da eficácia de três geradores de testes aleatórios reportados na literatura. Ao identificar suas limitações, este trabalho procurará sugerir oportunidades para melhorar os algoritmos de geração de testes aleatórios.

Palavras-chave: EDA, verificação, pré-silício, memória compartilhada, multicore, coerência de cache, testes

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto da proposta

Depois de ter provocado uma mudança radical na abordagem de com projetar microprocessadores para PCs e servidores, o multi-processamento em *chip*, ou *Chip Multi-Processing (CMP)*, levou ao atual quadro onde *chips* com 8 ou mais *cores* são bastante comuns. Prevê-se o uso de *chips* com dezenas a centenas cores e a sobrevida da abstração de memória compartilhada como condição para viabilizar a programação de propósitos gerais (DEVADAS, 2013). Isso resulta no desafio de como manter coerência de memória compartilhada **quando se aumenta a escala do número de *cores* em um único *chip***. Para suprir a abstração de memória compartilhada coerente, o subsistema de memória torna-se mais sofisticado e, portanto, mais suscetível a erros de projeto com o aumento massivo do número de *cores*. Além disso, como se espera que grande parte dos programas paralelos utilize bibliotecas para sincronização, a maioria dos programadores não precisa se preocupar com as regras de consistência (ADVE; GHARACHORLOO, 1996) do modelo de memória (HENNESSY; PATTERSON, 2011), o que tende a popularizar o uso de **modelos com máxima relaxação da ordem de programa** para aumentar o desempenho sem comprometer a programabilidade. Isso também contribui para aumentar a complexidade do hardware.

Portanto, a dificuldade de se validar sistemas computacionais baseados em CMP tende a aumentar dramaticamente a cada nova geração de produtos a serem lançados. Ora, **a validação da coerência e da consistência do subsistema de memória compartilhada**, o qual inclui múltiplos níveis de cache e protocolos complexos, constitui grande parte do esforço de se validar um sistema computacional baseado em CMP (WAGNER; BERTACCO, 2008). Como o número de estados induzidos por um protocolo de coerência cresce exponencialmente com o aumento do número de cores (SHIM et al., 2013), torna-se bastante desafiador o problema de verificar se o projeto de um chip baseado em múltiplos *cores* em larga escala implementa corretamente o comportamento esperado para o subsistema de memória compartilhada.

As primeiras técnicas de validação do sistema de memória foram propostas para serem aplicadas após a fabricação (**teste** pós-silício), através da execução de longos programas de testes aleatórios no próprio hardware do protótipo. No entanto, o simples reuso dessas técnicas é inadequado em tempo de projeto (para **verificação** pré-silício), porque seriam demasiadamente lentas quando os programas de teste são executados em simuladores, o que requer a limitação do tamanho do teste, restringindo assim a qualidade da verificação. Por isso, as técnicas de verificação tem se mostrado

frequentemente incapazes de encontrar erros sutis de projeto, que acabam passando para o hardware.

Verificar se as informações não estão demasiadamente imprecisas.

1.2 Escopo da proposta

Três ferramentas principais são necessárias para viabilizar a verificação pré-silício do subsistema de memória compartilhada: 1. Um **simulador** (que represente o comportamento do sistema de memória sob projeto), 2. Um **verificador** ou *checker* (capaz de indicar um comportamento incorreto e parar o simulador assim que um erro é detectado) e um 3. **Gerador automático de programas de teste** (capaz de fornecer estímulos capazes de induzir no sistema os comportamentos esperados pelos projetistas).

Esta proposta aborda o terceiro tipo de ferramenta, ou seja, **a geração automática de programas de teste para verificação de sistemas de CMP em tempo de projeto**. A proposta investigará geradores convencionais de testes aleatórios (RAMBO; HENSCHEL; SANTOS, 2011), geradores aleatórios sob fortes restrições (ADIR et al., 2004) (ANDRADE; GRAF; SANTOS, 2016a) e geradores de testes dirigidos (WAGNER; BERTACCO, 2008) (ELVER; NAGARAJAN, 2016).

O Laboratório de Computação Embarcada (ECL) servirá de hospedeiro para a execução das atividades no escopo desta proposta.

Revisar melhor o escopo, levando em conta a verificação pós-silício

2 OBJETIVOS

2.1 Objetivo Geral

O objetivo geral desta proposta é o estudo de técnicas de geração de testes mais eficientes e eficazes, que serão aplicadas sobre um protótipo do sistema em desenvolvimento, para solucionar o problema de verificar a consistência e coerência de memória compartilhada por sistemas computacionais baseados em CMP.

2.2 Objetivos Específicos

- **Analisar técnicas existentes:** investigar as principais limitações das técnicas reportadas na literatura, a fim de identificar oportunidades de contribuição científica.
- **Implementar um gerador de testes dirigidos:** tomar como base um gerador reportado na literatura (e.g. (WAGNER; BERTACCO, 2008), (ELVER; NAGARAJAN, 2016)).
- **Comparar com outros geradores:** comparar o gerador implementado com dois geradores de testes aleatórios: um convencional (RAMBO; HENSCHEL; SANTOS, 2011) e outro que explora restrições não convencionais (ANDRADE; GRAF; SANTOS, 2016a) para aumentar as chances de expor erros.

Verificar se é interessante deixar “Escopo” e “Objetivos” no mesmo capítulo
Analisar sugestão do Gabriel:
“(sugestão) revisar os objetivos específicos. Os objetivos que você listou me parecem ser os objetivos que iniciaram a sua pesquisa, e não os objetivos que dirigem a sua pesquisa para uma conclusão sobre o estado-da-arte. Em minha opção o segundo item teria que ser reformulado para caber como objetivo específico: em vez de “Implementar um gerador de testes dirigidos”, o objetivo específico é de “Implementar técnicas do estado-da-arte cuja implementação não foi disponibilizada pelos seus autores” ”

2.3 Método de Pesquisa

Algumas informações ainda se referem no futuro: verificar se é plausível alterar (ou remover) esta seção.
Averiguar correção do Gabriel:

"o modelo ruby não permite a descrição dos controladores de memória. O que permite isso é a linguagem SLICC criada pelos desenvolvedores do gem5. A linguagem de programação Ruby possui uma biblioteca que os desenvolvedores do gem5 utilizaram como base para a implementação da representação do sistema de memória."

O método adotado consiste na realização de experimentos sobre uma **representação de projeto do sistema sob verificação**, ou seja, na execução de testes através da simulação daquela representação, utilizando como infraestrutura um **simulador** de domínio público denominado gem5 (BINKERT et al., 2011). A representação de projeto utilizada para o subsistema de memória corresponde ao módulo *Ruby* daquele simulador, a qual permite a descrição das máquinas de estado dos protocolos de coerência.

Para desempenhar o papel de **verificador** da correta funcionalidade do subsistema de memória representado, será utilizada uma técnica de análise automática dos eventos registrados durante a simulação, a qual foi desenvolvida localmente (FREITAS; RAMBO; SANTOS, 2013) e cujo código está portanto disponível para uso no laboratório hospedeiro.

O protótipo de gerador de testes dirigidos será implementado com base na descrição de seu algoritmo conforme descrito na literatura.

Como representante das técnicas de geração convencional de testes aleatórios será utilizado o gerador denominado PLAIN (RAMBO; HENSCHEL; SANTOS, 2011). Como representante das técnicas de geração convencional de testes aleatórios sob fortes restrições será utilizado o gerador denominado CHAIN (ANDRADE; GRAF; SANTOS, 2016a). Os códigos dos dois geradores estão disponíveis para uso no laboratório hospedeiro.

Os três geradores serão comparados ao se usar a mesma representação de projeto, o mesmo simulador e o mesmo verificador acima mencionados.

Para a simulação de erros de projeto, serão utilizados como base os cinco erros artificiais descritos em (ANDRADE; GRAF; SANTOS, 2016b), mas novos erros artificiais serão desenvolvidos como parte do trabalho proposto para aumentar a abrangência da comparação.

Para a comparação serão adotadas as seguintes métricas: cobertura funcional, esforço computacional e probabilidade de detecção de erros.

3 CONTEXTUALIZAÇÃO

3.1 Geradores de programas de teste

Foram utilizados dois geradores: PLAIN e CHAIN. O primeiro se refere ao gerador proposto por (RAMBO; HENSCHEL; SANTOS, 2011) com o objetivo de ser amplamente independente do MCM (Mencionar anteriormente o significado de MCM) escolhido e oferecendo cobertura completa ao reduzir o número de variáveis compartilhadas. Já o segundo se refere ao gerador proposto por (ANDRADE; GRAF; SANTOS, 2016a), que se mostrou mais eficiente para expor erros de projeto.

Contextualizar a respeito de *biasing*

Contextualizar a respeito de *Checkers*

Verificar necessidade de contextualizar sobre o modelo *Ruby*

Verificar necessidade de explicar sobre a consistência adotada pelo Alpha

4 RESULTADOS EXPERIMENTAIS

4.1 Ambiente e configuração

Os experimentos foram realizados no simulador *gem5* (BINKERT et al., 2011)[1], configurado com a *Instruction Set Architecture* SPARC e com um modelo de memória similar ao do processador Alpha (ADVE; GHARACHORLOO, 1996). Foram selecionados o subsistema de memória Ruby, o modelo de temporização de CPU *out-of-order* (O3), e modo *system-call emulation*. Como opção de compilação do *gem5*, foi selecionado o "opt", que habilita otimizações e funcionalidades de depuração. O ambiente de execução um Linux com um processador Intel Xeon E5430 2.66GHz e 4GB de memória RAM. Como parâmetros da arquitetura simulada e testes gerados, foi estabelecido o número de endereços compartilhados como fixo em 32, número de instruções variando entre $1Ki, 2Ki, 4Ki, 8Ki, 16Ki$, número de *cores* variando entre $8, 16, 32$ e *biasing* com *True sharing*. Tratando-se das *caches*, utilizou-se o mecanismo de coerência *MESI Three Level* com as seguintes configurações: L0 privada com tamanho tanto da *cache* de instruções quanto de dados em 4KB; L1 privativa com tamanho de 64KB; e L2 com tamanho de 2MB. Todos os níveis possuem tamanho de bloco de 64B. Como política de substituição de dados, selecionou-se *Least Recently Used*. Por fim, o *checker* utilizado foi o XCheck-BE (*Best-Effort*).

Além disso, cinco erros artificiais distintos foram selecionados para os testes, identificados como t006, t024, t029, t020 e t015, respectivamente equivalentes aos descritos em (ANDRADE; GRAF; SANTOS, 2016b) como F1, F2, F3, F4 e f29, com suas características descritas na tabela 1.

ID	Localização	Estado atual	Evento de entrada	Próximo estado	Ação de saída impedida
F1	Controlador da *cache* L1	E_IL0	L0_DataAck	MM	u_writeDataFromL0Response
F2	Controlador da *cache* L1	M_IL0	WriteBack	MM_IL0	u_writeDataFromL0Request
F3	Controlador da *cache* L0	E	Store	E instead of M	—
F4	Controlador da *cache* L1	IS	Data_Exclusive	E	u_writeDataFromL2Response
f29	Controlador da *cache* L1	M_IL0	L0_DataAck	EE instead of MM	—

Tabela 1 – Descrição das classes de erros artificais, conforme descrição dada em (ANDRADE; GRAF; SANTOS, 2016b)

Adicionar mais informações sobre os parâmetros.

Adicionar motivação para escolha dos erros

Detalhar o que os erros exploram

[1] Foi utilizada a mesma versão do simulador em (ANDRADE; GRAF; SANTOS, 2016b), uma revisão baseada em <http://repo.gem5.org/gem5/rev/54d3ef2009a2> com modificações tais como a adição de erros artificiais nos controladores de memória *cache*.

4.2 Resultados Obtidos

Foram selecionadas duas medidas para análise: o tempo médio de detecção de um erro e a eficácia, sendo esta a relação de detecções por um total de execuções do programa de testes. Os valores obtidos dessas medidas estão expostos nas tabelas 2 e 3. Pela análise delas, pode-se perceber que a técnica CHAIN+ tomou um tempo médio menor para detectar um erro em quase todos os casos, chegando a cerca de 1/2 e até 1/3 do tempo tomado pela PLAIN+. As exceções foram os erros D3 (t020) para um tamanho de programa diferente de 2Ki e D1 (t006) para um tamanho de programa de 2Ki. A taxa de detecção da PLAIN+ foi superior apenas para todos os testes com D4 e em testes com D2 em programas de tamanho 1Ki: nos demais casos, CHAIN+ se manteve igual ou maior (chegando a cerca de 7~8x em 9 casos — D0 e D1 — e 10x em um dos casos de D0).

É possível perceber ainda que o tamanho do programa de testes influencia direta e simultaneamente no tempo de execução e na taxa de detecção: programas maiores acabam por aumentar a taxa de detecção, em detrimento de um tempo de execução mais elevado dos testes. Para CHAIN+, tal relação se aplica a todos os casos, enquanto que para PLAIN+ houveram ressalvas para D0 e D1 com tamanho de programa de 2Ki, em que a taxa de detecção foi reduzida ainda aumentando o tempo de execução.

Tabela 2 – Tempo e eficácia: PLAIN+

n	D0 (t006)	D1 (t024)	D2 (t029)	D3 (t020)	D4 (t015)
1k	22.19 (0.06)	21.16 (0.06)	19.19 (0.90)	9.60 (1.00)	19.58 (0.24)
2k	27.24 (0.03)	26.18 (0.05)	22.45 (0.96)	12.22 (1.00)	27.38 (0.42)
4k	45.35 (0.09)	45.40 (0.09)	41.52 (0.93)	16.01 (1.00)	49.36 (0.61)
8k	99.86 (0.12)	98.28 (0.12)	104.87 (0.97)	22.06 (1.00)	131.15 (0.78)
16k	204.02 (0.13)	202.71 (0.12)	276.29 (0.98)	34.84 (1.00)	308.73 (0.89)

Como forma de analisar melhor os efeitos da técnica, pode-se partir de uma medida derivada das duas anteriores, a qual será referida como “esforço”, represen-

Tabela 3 – Tempo e eficácia: CHAIN+

n	D0 (t006)	D1 (t024)	D2 (t029)	D3 (t020)	D4 (t015)
1k	18.46 (0.41)	18.25 (0.38)	16.75 (0.86)	9.07 (1.00)	19.21 (0.22)
2k	24.27 (0.48)	24.33 (0.53)	22.90 (0.96)	11.73 (1.00)	23.85 (0.31)
4k	35.30 (0.58)	34.80 (0.61)	35.77 (0.96)	16.16 (1.00)	34.69 (0.47)
8k	59.71 (0.76)	57.20 (0.68)	40.32 (0.99)	26.77 (1.00)	57.24 (0.64)
16k	103.29 (0.81)	103.48 (0.83)	131.25 (0.99)	39.44 (1.00)	103.39 (0.77)

tando o tempo de execução necessário para que se encontre pelo menos um erro, o que pode ser descrito conforme a equação 4.1 através do tempo médio de detecção ($\hat{t}$) e a eficácia (R). As tabelas 4 e 5 mostram o esforço necessário para expor cada erro dado um tamanho de programa, sendo mais visualizável a influência da CHAIN+ na detecção dos erros, uma vez que o tempo necessário para expor os erros D0 e D1, em relação à PLAIN+, chegou a cerca de $\frac{1}{9}$ para programas de tamanho 1Ki e $\frac{1}{8}$ para programas de tamanho 8Ki e 16Ki ainda com maior taxa de detecção. A medida de esforço expôs então que, para D0 e D1, foi possível reduzir consideravelmente o esforço de verificação sem aumentar o tamanho do teste (ainda que testes maiores continuem mais capazes de detectar erros).

$$E = \hat{t} \cdot \left\lceil \frac{1}{R} \right\rceil \tag{4.1}$$

Detalhar como foi obtido o tempo médio de detecção

Dar uma letra melhor para R

Detalhar comparativos para diferentes quantidades de núcleos

Tabela 4 – Medida de esforço: PLAIN+

n	D0 (t006)	D1 (t024)	D2 (t029)	D3 (t020)	D4 (t015)
1k	399.46	359.70	38.37	9.60	97.92
2k	817.31	575.94	44.89	12.22	82.14
4k	498.90	544.75	83.04	16.01	98.71
8k	898.73	884.53	209.73	22.06	262.31
16k	1632.15	1824.35	552.57	34.84	617.46

Tabela 5 – Medida de esforço: CHAIN+

n	D0 (t006)	D1 (t024)	D2 (t029)	D3 (t020)	D4 (t015)
1k	55.37	54.74	33.50	9.07	96.04
2k	72.81	48.67	45.80	11.73	95.40
4k	70.60	69.59	71.55	16.16	104.07
8k	119.41	114.40	80.64	26.77	114.47
16k	206.58	206.96	262.50	39.44	206.79

REFERÊNCIAS

ADIR, A. et al. Genesys-pro: innovations in test program generation for functional processor verification. *IEEE Design Test of Computers*, v. 21, n. 2, p. 84–93, Mar 2004. ISSN 0740-7475.

ADVE, S. V.; GHARACHORLOO, K. Shared memory consistency models: a tutorial. *Computer*, IEEE, v. 29, n. 12, p. 66–76, Dec 1996. ISSN 0018-9162.

ANDRADE, G. A. G.; GRAF, M.; SANTOS, L. C. V. dos. Chain-Based Pseudorandom Tests for Pre-Silicon Verification of CMP Memory Systems. In: *34th IEEE International Conference on Computer Design (ICCD)*. [S.l.: s.n.], 2016. p. 552–559.

ANDRADE, G. A. G.; GRAF, M.; SANTOS, L. C. V. dos. Exploiting Canonical Dependence Chains and Address Biasing Constraints to Improve Random Test Generation for Shared-Memory Verification. In: . [S.l.: s.n.], 2016. p. 50.

BINKERT, N. et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 39, n. 2, p. 1–7, ago. 2011. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/2024716.2024718>.

DEVADAS, S. Toward a coherent multicore memory model. *Computer*, IEEE, n. 10, p. 30–31, 2013.

ELVER, M.; NAGARAJAN, V. McVerSi: A test generation framework for fast memory consistency verification in simulation. In: *IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*. [S.l.: s.n.], 2016. p. 618–630.

FREITAS, L. S.; RAMBO, E. A.; SANTOS, L. C. V. dos. On-the-fly verification of memory consistency with concurrent relaxed scoreboards. In: *Design, Automation, and Test in Europe (DATE)*. [S.l.: s.n.], 2013. p. 631–636. ISBN 978-1-4503-2153-2.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 5th. ed. [S.l.]: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728.

RAMBO, E.; HENSCHEL, O.; SANTOS, L. dos. Automatic generation of memory consistency tests for chip multiprocessing. In: *IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS)*. [S.l.: s.n.], 2011. p. 542–545.

SHIM, K. S. et al. Design tradeoffs for simplicity and efficient verification in the execution migration machine. In: IEEE. *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. [S.l.], 2013. p. 145–153.

WAGNER, I.; BERTACCO, V. MCjammer: Adaptive Verification for Multi-core Designs. In: *Design, Automation, and Test in Europe (DATE)*. [S.l.: s.n.], 2008. p. 670–675. ISSN 1530-1591.