Olav Philipp Henschel

Verificação de coerência e consistência de memória compartilhada para multiprocessamento em chip: o impacto da simulação de falhas na avaliação de cobertura

Florianópolis – SC, Brasil

2011

Olav Philipp Henschel

Verificação de coerência e consistência de memória compartilhada para multiprocessamento em chip: o impacto da simulação de falhas na avaliação de cobertura

Trabalho de conclusão de curso apresentado para obtenção do Grau de Bacharel em Ciências da Computação pela Universidade Federal de Santa Catarina.

Orientador:
Luiz Cláudio Villar dos Santos

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis – SC, Brasil

2011

Resumo

Num processador com pipeline, a execução sequencial de instruções pode exigir que ele pare quando uma instrução depende de uma instrução anterior que ainda não terminou sua execução. Por isso, processadores modernos possibilitam a execução fora de ordem, de modo que uma instrução que esteja esperando pela produção de um dado, por exemplo, não interrompa as seguintes, quando estas já tiverem suas dependências resolvidas. Para que isso funcione sem alterar os resultados do programa, devem existir estruturas e mecanismos especiais, que aumentam significativamente a complexidade do núcleo de processamento.

Ao incorporar diversos núcleos numa mesma pastilha de silício, cada um com caches privativas, é preciso garantir que a interface hardware-software defina uma *visão unificada da memória compartilhada*, embora cópias de variáveis compartilhadas possam estar distribuídas através da hierarquia de memória e embora a execução fora-de-ordem de escritas e leituras possa comprometer a sincronização de threads.

Dois aspectos precisam ser considerados para prover essa visão unificada: por um lado, é preciso garantir que todos os núcleos de processamento tenham uma visão *coerente* ao referenciar um *mesmo endereço* de memória; por outro lado, é preciso garantir que todos os núcleos tenham uma visão *consistente* da ordem de execução entre leituras e escritas que referenciam *endereços distintos*. O primeiro aspecto é garantido pelo *protocolo de coerência*, enquanto o segundo aspecto é garantido pela implementação de um *modelo de consistência*.

Projetar um subsistema de memória que garanta consistência e coerência é uma tarefa complexa, que pode induzir erros difíceis de encontrar. Por isso, é crucial o desenvolvimento de ferramentas para verificar se o hardware do subsistema de memória efetivamente implementa o modelo de consistência e coerência especificado na interface hardware-software.

No Núcleo Interdepartamental de Microeletrônica (NIME) da UFSC, membro do INCT de Sistemas Micro e Nanoeletrônicos, está sendo desenvolvida uma ferramenta de *verificação funcional* de sistemas de memória para sistemas integrados multiprocessados. O objetivo do trabalho descrito nesta monografia é dar suporte à validação dessa ferramenta, permitindo comparar suas garantias de verificação com as de ferramentas congêneres.

A principal contribuição técnica deste trabalho é a *simulação de falhas* na representação executável do hardware de uma plataforma multiprocessada para a *avaliação da cobertura da verificação funcional*. As falhas simuladas foram escolhidas para capturar erros comuns que comprometem a coerência e a consistência da memória compartilhada em sistemas multiprocessados.

Sumário

1 Introdução

1.1 Tendências tecnológicas em computação embarcada

Alguns dos últimos smartphones que chegaram ao mercado e outros com lançamento iminente são baseados em processadores da família Cortex, com arquitetura ARM. Dentre eles, boa parte possui um Cortex-A8 ou Cortex-A9 [Qualcomm 2011, ARM Holdings 2009]. Cada vez mais, esses processadores estão incluindo múltiplos núcleos de processamento, devido ao crescente número de aplicativos desenvolvidos para esse tipo de aparelho e que demandam um poder de processamento cada vez maior.

1.2 A relevância da verificação funcional de memória compartilhada

É possível observar nos manuais da ARM [ARM Holdings 2009], e na Figura 1.1, que os diversos processadores com múltiplos núcleos possuem hierarquias de memória semelhantes, com caches de dados e instruções privativas para cada processador, um ou mais níveis de cache compartilhada e um protocolo de coerência de cache, geralmente o do tipo *snooping*.

Dessa forma, é preciso garantir que a interface hardware-software defina uma *visão unificada da memória compartilhada*, embora cópias de variáveis compartilhadas possam estar distribuídas através da hierarquia de memória e embora a execução fora-de-ordem de escritas e leituras possa comprometer a sincronização de threads.

Dois aspectos precisam ser considerados para prover essa visão unificada: por um lado, é preciso garantir que todos os núcleos de processamento tenham uma visão *coerente* ao referenciar um *mesmo endereço* de memória; por outro lado, é preciso garantir que todos os núcleos tenham uma visão *consistente* da ordem de execução entre leituras e escritas que referenciam *endereços distintos*. O primeiro aspecto é garantido pelo *protocolo de coerência*, enquanto o segundo aspecto é garantido pela implementação de um *modelo de consistência*.

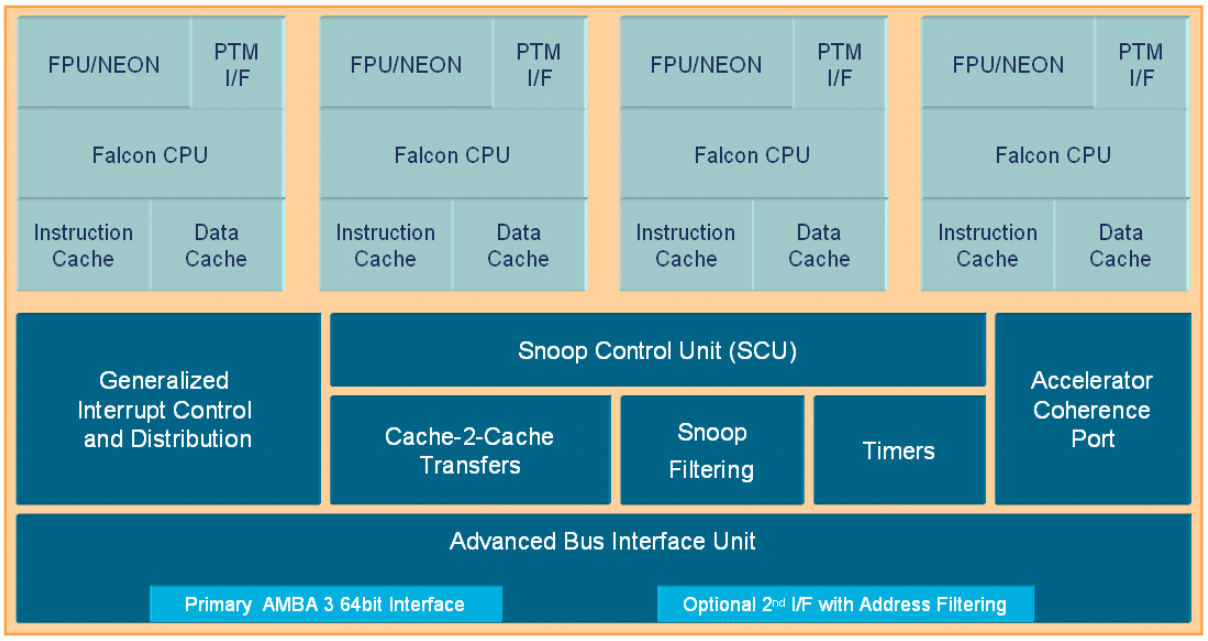

Figura 1.1: Processador multicore Cortex-A9 [ARM Holdings 2009]

Projetar um subsistema de memória que garanta consistência e coerência é uma tarefa complexa, que pode induzir erros difíceis de encontrar. Por isso, é crucial o desenvolvimento de ferramentas para verificar se o hardware do subsistema de memória efetivamente implementa o modelo de consistência e coerência especificado na interface hardware-software.

1.3 Motivação

No Núcleo Interdepartamental de Microeletrônica (NIME) da UFSC, membro do INCT de Sistemas Micro e Nanoeletrônicos, está sendo desenvolvida uma ferramenta de *verificação funcional* de sistemas de memória para sistemas integrados multiprocessados. Ela funciona analisando registros de execução (traces), gerados durante a execução de um simulador de uma plataforma de processamento real. Esses registros contém as operações de memória na ordem em que foram executadas pelo simulador, com informações como o valor lido/escrito e o endereço que a operação referencia.

Para validar essa ferramenta, é necessária tanto a geração de registros de execução corretos quanto a geração de registros de execução incorretos, para os quais a ferramenta detectaria um erro. No entanto, o simulador, se sua implementação for correta, gera apenas registros de execução corretos, que não são suficientes para a validação.

A inclusão manual de erros em um registro de execução correto seria tediosa, pois é necessário um grande número de configurações diferentes de números de processadores, de endereços

compartilhados e de instruções para que a ferramenta seja validada de forma exaustiva e possa ser comparada com outras ferramentas congêneres. Deve ser tomado cuidado para inserir erros que de fato ocorram em processadores reais, pois erros difíceis de ocorrer em um processador real teriam pouco valor na validação da ferramenta. Com a inclusão manual, seria difícil injetar erros reais, pois teria-se que imaginar uma falha no processador e inferir suas implicações nos registros de execução.

Com isso em mente, resolveu-se modificar uma plataforma de simulação existente, injetando falhas que poderiam ocorrer nos núcleos de processamento e no subsistema de memória durante o projeto de um processador. Ao injetar falhas que violem a consistência e a coerência de memória, seu efeito será transmitido aos registros de execução durante a execução e, posteriormente, serão identificadas por uma ferramenta de verificação, como a que está em desenvolvimento no NIME.

Assim como o desenvolvimento de um protocolo de coerência e a adequação de uma arquitetura de processador requerem um estudo detalhado do modelo de consistência de memória definido, a introdução de falhas que causem problemas específicos também requer um entendimento desse modelo de consistência e, inclusive, da plataforma implementada, que deverá ser modificada.

1.4 Objetivo e contribuição técnica

A principal contribuição técnica deste trabalho é a *simulação de falhas* na representação executável do hardware de uma plataforma multiprocessada para permitir a *avaliação da cobertura da verificação funcional*. As falhas simuladas foram escolhidas para capturar erros comuns que comprometam a coerência e a consistência da memória compartilhada em sistemas multiprocessados.

1.5 Infraestrutura experimental

Como plataforma de simulação em que serão inseridas as falhas foi adotado o simulador M5 [M5 Simulator]. Por possuir uma boa base de usuários e vários anos de utilização, ele é considerado uma boa aproximação dos processadores reais e pode ser usado como modelo de referência. Ele possui modelos de processadores Alpha, ARM, MIPS, PowerPC, SPARC e x86 e é implementado em C++, com uma interface de configuração em Python.

O simulador M5 possui uma hierarquia de classes modularizada e bem separada, evitando

que se precise entender o sistema inteiro para modificar alguma estrutura em particular, o que torna a adaptação das falhas a outra arquitetura muito simples.

Para injetar as falhas, foi preciso estudar exaustivamente os módulos referentes ao núcleo de processamento e ao subsistema de memória.

1.6 Organização da monografia

Esta monografia é organizada da seguinte forma. O Capítulo 2 explica sucintamente os conceitos envolvidos. No Capítulo 3 são relatados alguns trabalhos correlatos. O Capítulo 4 descreve a estrutura da plataforma simulada e como ela foi gerada no simulador. O Capítulo 5 descreve as falhas implementadas e o Capítulo 6 fornece os resultados das simulações. No Capítulo 7 são feitas as conclusões e indicadas outras falhas que poderiam ser modeladas.

2 *Fundamentação teórica*

2.1 Modelos de consistência de memória

Recentemente, além de estarem presentes em computadores domésticos e servidores, processadores com múltiplos núcleos estão aparecendo também em aparelhos portáteis e se torna necessário verificar se a execução de programas paralelos está correta para todos os tipos de plataformas. Para isso, é preciso haver uma especificação de quando e com que valor variáveis compartilhadas que foram modificadas em um núcleo devem aparecer em outro núcleo. Essa especificação é chamada de modelo de consistência de memória (*memory consistency model* - MCM) [Hennessy e Patterson 2006].

Um MCM é definido através de dois conjuntos de axiomas. Os axiomas de ordem definem a relação entre a ordem global de operações, vista por todos os processadores, e a ordem local de cada processador. É importante notar que a ordem global é uma abstração, ela não pode ser observada, apenas inferida à partir das ordens locais, observadas nas interfaces entre as caches privativas e os núcleos de processamento. Os axiomas de valor definem o valor que uma leitura deve fornecer, dadas as ordens loais e globais das escritas anteriores.

Existem diversos MCMs, cada um com um grau de relaxamento diferente. Quanto maior o relaxamento, maior a liberdade que o compilador e o processador tem para mudar a ordem das operações em memória, ocasionando maiores ganhos de performance. Entretanto, modelos muito relaxados podem permitir alterações na ordem das operações que tornam a programação confusa e muito frequentemente exigem que o programador insira barreiras de memória, que podem causar esperas (*stalls*) no processador. É evidente, portanto, que deve haver um compromisso entre o ganho de performance no processador e a facilidade de programação no MCM.

Alguns exemplos de MCMs são: Sequential Consistency (SC), Total Store Order (TSO) [Hangal et al. 2004] e Alpha's Relaxed Order (ARO) [Adve e Gharachorloo 1996].

2.1.1 Alpha's Relaxed Order

A plataforma utilizada adota o modelo ARO [Adve e Gharachorloo 1996]. Este MCM permite o relaxamento da ordem de escritas (*writes* - W) e leituras (*reads* - R) para endereços diferentes para qualquer combinação dessas duas instruções: R $\rightarrow$ R, R $\rightarrow$ W, W $\rightarrow$ R, W $\rightarrow$ W. Ele também permite que um processador leia o valor de sua própria escrita antes que ela chegue à memória, ou seja, permite adiantamentos.

O modelo ARO possui as instruções simples de leitura e de escrita e uma instrução de troca (*swap*), que escreve em memória o valor de um registrador e o atualiza com um novo valor, lido da memória, de forma atômica. Ele também possui duas instruções de sincronização, as *barreiras de memória*, que valem para ambas leituras e escritas e as *barreiras de memória de escritas*, que valem somente para escritas. Elas servem para garantir que as operações de memória que seguem a barreira executem apenas após as operações de memória que precedem a barreira terem terminado, eliminando a possibilidade de o processador relaxar a ordem das operações envolvidas.

2.2 Verificação funcional

Com a crescente complexidade do projeto de microprocessadores, torna-se cada vez mais indispensável a utilização de ferramentas automatizadas para sua verificação e validação, tanto antes quanto depois da fabricação em silício. O método de verificação mais utilizado é a simulação. Sua principal vantagem em relação à verificação formal é a facilidade na utilização [Simulation-Based Verification versus Formal Verification].

Para realizar as simulações, é necessário dispor de vetores de entrada, que servirão de estímulo ao simulador. O simulador recebe esses vetores e gera vetores de saída, através de alguma forma de processamento. O analisador se encarrega de verificar se esses vetores de saída gerados fazem sentido. Isso pode ser feito através de vetores de referência, calculados à partir dos vetores de entrada, ou através de regras introduzidas explicitamente no analisador, que definem se um determinado vetor de saída é ou não permitido.

2.2.1 Verificação de consistência e coerência

Para realizar a verificação de consistência e coerência de memória através de simulações, é necessário dispor de códigos executáveis, que assumem o papel de vetores de entrada. Esses executáveis são geralmente criados por um gerador de testes com instruções aleatórias (*random*

instruction tests - RIT) [Rambo, Henschel e Santos 2011], em razão de ser muito difícil e trabalhoso de criar programas que explicitem uma grande variedade de falhas. Os testes contém instruções de leitura, escrita, troca e barreiras de memória, além das instruções necessárias à inicialização e término das threads.

Os vetores de saída das simulações são, neste caso, registros de execução contendo a ordem em que as operações de memória (leituras, escritas, trocas e barreiras de memória) foram emitidas por cada processador ao subsistema de memória. Essa ordem pode diferir da ordem do programa executado, em razão de os núcleos de processamento serem do tipo fora-de-ordem.

A partir dos vetores de saída das simulações e de um MCM é possível verificar a consistência e a coerência da plataforma, usando uma ferramenta de verificação. O que a ferramenta faz é determinar se existe uma ordenação global das instruções que satisfaça todos os axiomas de ordem e valor, dados os registros de execução locais.

Algumas ferramentas podem também utilizar recursos extras para diminuir seu tempo de execução, dado que esse é um problema NP-completo [Cantin, Lipasti e Smith 2001]. Um deles é a utilização da ordem global das instruções fornecida pela plataforma de execução, criada por um monitor na interface do subsistema de memória. Outro é o conhecimento prévio das escritas geradas por cada leitura. Isso pode ser feito criando-se escritas únicas nos vetores de entrada do simulador, cada uma com um valor diferente.

3 *Trabalhos correlatos*

3.1 Abordagens para verificação de consistência

Existem diversas técnicas diferentes para se verificar a consistência de memória de um sistema. Uma delas se baseia na detecção de ciclos em grafos acíclicos dirigidos (*directed acyclic graph* - DAG). Nesse tipo de técnica, os vértices representam operações de memória (escrita, leitura, troca, barreira, ...) e as arestas representam relações de ordenamento. Quando há um ciclo, existe um paradoxo, pois entende-se que as operações envolvidas devem ocorrer ao mesmo tempo antes e depois umas das outras. As ferramentas que operam dessa forma criam um grafo a partir das operações e tentam inferir todas as arestas, indicando um erro se um ciclo for detectado. O tempo de análise pode ser reduzido se algumas das arestas não forem inferidas, porém o algoritmo se torna incompleto, por não encontrar todos os erros possíveis. O TSOtool [Hangal et al. 2004] é um exemplo de ferramenta de verificação incompleta que se utiliza desse método e que posteriormente foi incrementada para se tornar completa [Manovit e Hangal 2006].

O Relaxed Scoreboard [Shacham et al. 2008] é uma técnica de verificação que possui um princípio de funcionamento diferente. Ele mantém uma tabela de possíveis valores para cada endereço de memória, que se adapta durante a execução do programa. Quando é feita uma escrita na memória, um novo valor é adicionado à tabela para o endereço da operação; quando ocorre uma leitura, os possíveis valores são filtrados, reduzindo o número de possibilidades. A principal vantagem em relação à técnica baseada em detecção de ciclos é que dessa forma, a ferramenta pode fazer a análise simultaneamente à execução do programa, e assim detectar erros mais rapidamente.

O Emparelhamento Estendido em Grafos Bipartidos [Rambo, Henschel e Santos 2012] é outro método baseado em grafos, mas que funciona de uma forma totalmente diferente, sem a detecção de ciclos. Ele foi projetado especialmente para a verificação de representações executáveis de plataformas multiprocessadas. Devem existir dois monitores em cada núcleo de processamento: um na saída da unidade de consolidação e outro na interface com a memória

privativa. Um grafo bipartido é construído a partir desses monitores, com seus vértices representando operações de memória e arestas representando equivalência. Outro grafo, com os mesmos vértices, representa a ordenação de operações, imposta pelo MCM, através de arestas. Com esses dois grafos é possível determinar se o comportamento local de cada processador está correto, em respeito às operações de memória. Para verificar o comportamento global, são usados relógios de Lamport [Lamport 1978], que utilizam marcas de tempo geradas pelos monitores para cada operação. A partir deles, é criado um registro de execução global, que é analisado por um algoritmo que determina se ele viola as regras do MCM.

3.2 Abordagens para prover cobertura

Como os métodos de verificação descritos nesta monografia são baseados em registros de execução, sua cobertura depende fortemente dos programas executados. Eles devem expôr o maior número possível de falhas, ter uma quantidade pequena de instruções e o número de programas a serem executados deve ser o menor possível. Como é impossível fazer um programa ideal, que exponha todas as falhas possíveis em poucas instruções, o que se faz normalmente é usar um gerador de RITs. É importante que ele gere uma boa variedade de testes, com números diferentes de processadores e diversas proporções de operações diferentes, para expôr falhas de naturezas distintas. O gerador utilizado para as simulações apresentadas posteriormente aceita parâmetros como número de operações, número de processadores, porcentagens de escritas, leituras, barreiras de memória e trocas e é descrito com mais detalhes em [Rambo, Henschel e Santos 2011].

3.3 Modelagem de falhas

As falhas modeladas devem ser o mais próximo possível de falhas reais, encontradas durante o projeto de um sistema embarcado. Assim como em [Shacham et al. 2008], foram modeladas diversas falhas de diferentes naturezas e em estruturas distintas. Elas podem ser encontradas no protocolo de coerência, na fila de escritas e leituras, no subsistema de memória e podem causar problemas como escrita ou leitura incorreta, violação da ordem do programa, inconsistências de dados intra e inter-processadores, entre outros. Elas foram projetadas para causarem problemas dependentes também do tipo de arquitetura do sistema, algumas aparecendo somente quando há um número maior ou menor de processadores ou quando o número de endereços referenciados pelo programa ultrapassar certo limite. Essa diversidade garante uma melhor cobertura para o teste das ferramentas de verificação.

4 *Estrutura da plataforma multiprocessada*

Para se executar uma plataforma baseada no simulador M5, é necessário primeiramente definir a arquitetura da plataforma a ser compilada. Foi escolhida a Alpha como arquitetura alvo, por ser amplamente utilizada por usuários do simulador M5 e possuir baixa probabilidade de possuir falhas de implementação.

Após a compilação da plataforma, sua execução requer como parâmetro um script de configuração em Python, através do qual é possível configurar diversos aspectos do sistema, instanciando estruturas como núcleos de processamento, diversos níveis de memórias privativas e compartilhadas e definindo o workload de cada processador.

Existem diversas implementações para núcleos de processamento, com diferentes níveis de detalhamento, como a implementação funcional (sem noção de tempo) e a atômica (com estimativa de número de ciclos). Como a geração dos registros de execução para análise de consistência requer uma ordenação realista das instruções e dos acessos à memória, foi utilizada a implementação de núcleos fora-de-ordem, que é a mais próxima da arquitetura real, fazendo uso da implementação do MCM para relaxar a ordenação das instruções.

Foram instanciadas classes L1 privativas de dados e instruções para cada núcleo de 32KiB cada, uma cache L2 compartilhada de 4MiB e 128MiB de memória principal. O protocolo de coerência de memória utilizado é o *snooping* [Hennessy e Patterson 2006]. O número de núcleos de processamento varia para cada teste realizado, e seus workloads são os arquivo executáveis gerados pelo RIT para cada configuração desejada.

4.1 A arquitetura de multiprocessamento

A arquitetura utilizada está representada na Figura 4.1. Ela é composta de um número variável de núcleos de processamento, cada um com uma cache privativa, separada em instruções

e dados, um barramento de memória compartilhada para comunicação com a cache nível 2 e a memória principal. As setas indicam os locais onde foram inseridas as falhas.

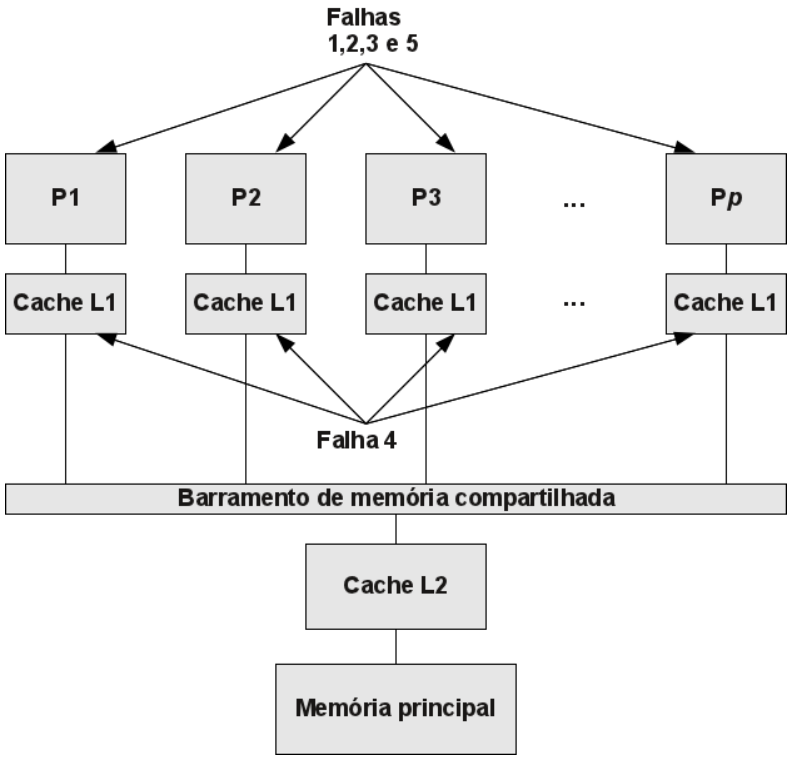

Figura 4.1: Arquitetura da plataforma

4.2 A organização interna do núcleo de processamento

Internamente, os núcleos de processamento possuem a estrutura da Figura 4.2. As intruções recebidas ficam numa fila de instruções e, caso sejam leituras ou escritas, são decodificadas pela unidade de endereçamento, caso contrário vão para as respectivas unidades de execução. Leituras e escritas ficam numa fila aguardando pela sua vez de terem seu valor enviado ou recebido da memória. As demais instruções precisam aguardar pela produção dos valores necessários nos registradores. Quando terminam, deixam o dado produzido no banco de registradores, o que possibilita que as instruções dependentes desse dado executem, e ficam no buffer de reordenamento (*reorder buffer* - ROB) aguardando o término das instruções anteriores para serem consolidadas.

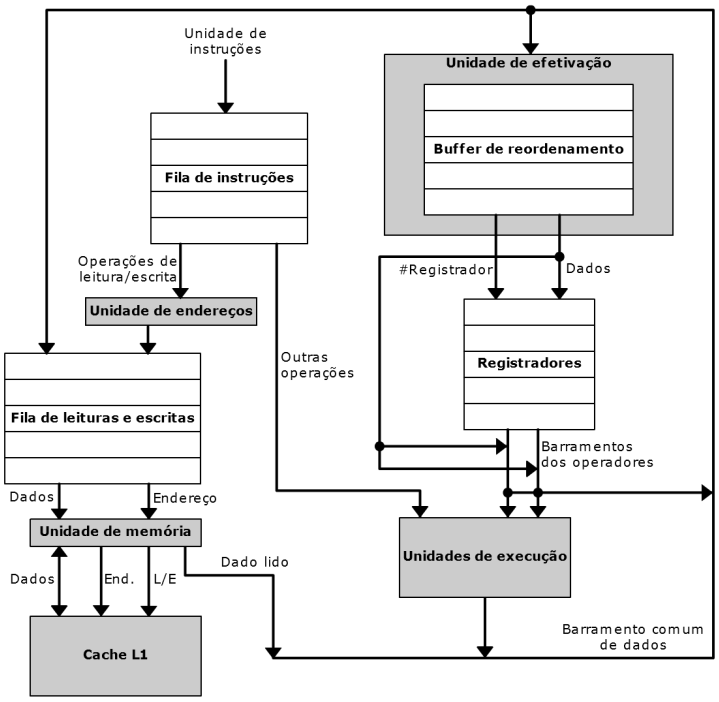

Figura 4.2: Núcleo de processamento

5 *Implementação das falhas*

Para implementar as falhas, foram modificados os códigos-fonte de algumas classes do M5, em C++. Cada falha foi implementada separadamente, gerando 5 plataformas, cada uma com um tipo de falha diferente.

Para evitar que as falhas prejudiquem a inicialização das threads e as chamadas iniciais do programa sob teste, elas são ativadas apenas após o término da inicialização. Caso contrário, elas podem ocasionar um erro que impeça que a simulação chegue no teste, como a modificação de um ponteiro, que ocasionaria uma falha de segmentação, ou de uma variável condicional de um laço, que poderia tornar esse laço eterno. Esse momento foi determinado com base em simulações anteriores, observando o ciclo de processamento em que aparece a primeira barreira de memória, que indica a instrução em que as threads estão sincronizadas e prontas para executar o programa teste.

Também foi considerada uma probabilidade de ocorrência de falhas em algumas plataformas, para simular falhas intermitentes. O número do ciclo de processamento foi usado como um fator pseudo-aleatório para determinar essa probabilidade.

5.1 Seleção de falhas

As falhas implementadas são um subconjunto das classes de erros em [Shacham et al. 2008] e estão descritas na Tabela 5.1. As falhas escolhidas são aquelas que se acredita serem mais comuns em sistemas reais.

5.2 Simulação das falhas

A seguir é descrita a forma como a plataforma foi modificada para cada falha e o impacto que essa modificação causa.

Tabela 5.1: Caracterização das falhas

Falha	Descrição	Localização
1	O processador escreve um novo valor e em seguida lê o valor antigo da memória no mesmo endereço	Fila de leituras e escritas
2	Uma escrita mais nova é sobrescrita por uma escrita antiga do mesmo processador	Fila de leituras e escritas
3	Leituras consecutivas de um mesmo endereço, em um mesmo processador fornecem valores diferentes	Fila de leituras e escritas
4	Leitura por um processador diferente fornece dado inválido	Unidade de controle de coerência
5	Leitura fornece dado incorreto	Fila de leituras e escritas

Falha 1

Algumas leituras ignoram a fila de escritas e lêem direto da memória cache. A consequência disso é que a leitura fornecerá o valor antigo da memória, e não o valor da escrita mais nova.

Falha 2

Quando se está fazendo o writeback de uma escrita, verifica-se se existe uma escrita mais nova na fila e para o mesmo endereço. Caso exista, o valor das duas escritas é trocado, e o valor da escrita mais nova é copiado para a memória, no lugar do antigo. O registro de execução local do processador mostrará as escritas na ordem correta, porém no registro de execução global elas estarão trocadas.

Falha 3

Quando na execução de uma leitura é encontrado uma escrita na fila para o mesmo endereço e é feito um adiantamento do seu valor, há uma chance de ocorrer uma falha do tipo *stuck-at-1* no seu último bit. A próxima leitura que tentar fazer um adiantamento dessa escrita, e que portanto deve ter o mesmo endereço, pode fornecer um valor diferente.

Falha 4

Quando um processador responde a um *snoop*, há uma chance dele retornar 0 ao invés do dado correto.

Falha 5

Algumas leituras apresentam uma falha do tipo *stuck-at*. Isso torna seu dado incorreto.

6 *Resultados experimentais*

6.1 Configuração experimental

Os resultados foram obtidos através da análise dos registros de execução das plataformas de simulação modificadas. Essa análise foi feita por um verificador baseado em detecção de ciclos adaptado do TSOtool [Hangal et al. 2004], adaptado ao modelo ARO.

6.2 O impacto das falhas modeladas na verificação

Nesta seção estão reproduzidos resultados de trabalho colaborativo com o mestrando Eberle A. Rambo, que foram objeto de artigo em co-autoria submetido ao ICECS 2011 [Rambo, Henschel e Santos 2011]. A Tabela 6.1 mostra as falhas não detectadas para cada configuração de número de instruções (n), número de endereços compartilhados (s) e número de núcleos de processamento (c).

Como pode-se observar, as falhas 1 e 3 são detectadas apenas com um número menor de endereços compartilhados, pois conforme aumenta o número de endereços compartilhados,

Tabela 6.1: Falhas não detectadas em função de número de instruções (n), número de endereços compartilhados (s) e número de núcleos de processamento (c)

s	$n = 2K$			$n = 4K$			$n = 8K$			$n = 16K$		
	$c = 2$	$c = 4$	$c = 8$	$c = 2$	$c = 4$	$c = 8$	$c = 2$	$c = 4$	$c = 8$	$c = 2$	$c = 4$	$c = 8$
2		1	1			1						
4	1,3	1	1	1	1	1						1
8	1,3	1,3	1	1	1,3	1		3		1		1
16	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1	1	1
32	1,3	1,3	1,2,3	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1,3

diminui a probabilidade de instruções consecutivas referenciarem o mesmo endereço. Uma análise mais detalhada dos dados dessa tabela é feita em [Rambo, Henschel e Santos 2011].

7 *Conclusões e melhorias*

As mesmas falhas aqui descritas foram simuladas para avaliar a cobertura da nova ferramenta de verificação baseada em emparelhamento estendido em grafos bipartidos proposta por Eberle A. Rambo, comparando-a com a de um método baseado em DAGs. Entretanto, os resultados desta comparação não são aqui reportados, pois fazem parte da dissertação de mestrado de Eberle A. Rambo [Rambo e Santos 2011]. Assim, a simulação de falhas descrita nesta monografia foi crucial para a avaliação da nova técnica de verificação de consistência proposta na dissertação de mestrado de Eberle A. Rambo. Isso pode ser comprovado pela co-autoria de dois trabalhos científicos submetidos a eventos internacionais [Rambo, Henschel e Santos 2011, Rambo, Henschel e Santos 2012].

Como trabalho futuro, pretende-se simular outras falhas, como troca de dados incorretos no protocolo de coerência (*snooping*), problemas nos bits de validade das caches e barreiras de memória funcionando incorretamente. Espera-se que algumas dessas falhas afetem blocos de memória inteiros, modificando não o dado que está sendo acessado, mas algum dado próximo, e que causem um efeito contrário ao que pode ser observado na Tabela **??** para as falhas 1 e 3, sendo detectadas mais facilmente com números maiores de endereços compartilhados.

Referências Bibliográficas

ADVE, S. V.; GHARACHORLOO, K. Shared memory consistency models: A tutorial. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 29, n. 12, p. 66–76, 1996. ISSN 0018-9162.

ARM HOLDINGS. *The ARM Cortex-A9 Processors*. 2009. Disponível em: <http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf>.

CANTIN, J. F.; LIPASTI, M. H.; SMITH, J. E. Dynamic verification of cache coherence protocols. In: *Workshop on Memory Performance Issues*. [S.l.: s.n.], 2001.

HANGAL, S. et al. TSOtool: a program for verifying memory systems using the memory consistency model. In: *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. [S.l.: s.n.], 2004. p. 114–123. ISSN 1063-6897.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 4. ed. [S.l.]: Morgan Kaufmann, 2006.

LAMPORT, L. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, ACM, New York, NY, USA, v. 21, p. 558–565, July 1978. ISSN 0001-0782.

M5 Simulator. Disponível em: <http://www.m5sim.org/>.

MANOVIT, C.; HANGAL, S. Completely verifying memory consistency of test program executions. In: *Symposium on High-Performance Computer Architecture*. [S.l.: s.n.], 2006.

QUALCOMM. *Snapdragon^{TM}MSM8x60/APQ8060*. jul 2011. Disponível em: <http://www.qualcomm.com/snapdragon/specs>.

RAMBO, E. A.; HENSCHEL, O. P.; SANTOS, L. C. V. On ESL verification of memory consistency for system-on-chip multiprocessing. Design, Automation Test in Europe Conference (DATE). 2012.

RAMBO, E. A.; HENSCHEL, O. P.; SANTOS, L. C. V. d. Automatic generation of memory consistency tests for chip multiprocessing. *International Conference on Electronics, Circuits, and Systems*, 2011.

RAMBO, E. A.; SANTOS, L. C. V. d. *Verificação de consistência de memória para sistemas integrados multiprocessados*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2011.

SHACHAM, O. et al. Verification of chip multiprocessor memory systems using a relaxed scoreboard. In: *IEEE/ACM International Symposium on Microarchitecture*. [S.l.: s.n.], 2008.

SIMULATION-BASED Verification versus Formal Verification. Disponível em: <http://www.codeidol.com/hardware/hardware-design/An-Invitation-to-Design-Verification/Simulation-Based-Verification-versus-Formal-Verification/>.