**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**

Gabriel Arthur Gerber Andrade

# ANÁLISE COMPARATIVA ENTRE DOIS VERIFICADORES DE CONSISTÊNCIA E DE COERÊNCIA DE MEMÓRIA

Florianópolis - Santa Catarina

2014

Gabriel Arthur Gerber Andrade

# ANÁLISE COMPARATIVA ENTRE DOIS VERIFICADORES DE CONSISTÊNCIA E DE COERÊNCIA DE MEMÓRIA

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

Florianópolis - Santa Catarina

2014

Ficha de identificação da obra elaborada pelo autor através do
Programa de Geração Automática da Biblioteca Universitária da UFSC.

A ficha de identificação é elaborada pelo próprio autor

Maiores informações em:
http://portalbu.ufsc.br/ficha

Gabriel Arthur Gerber Andrade

**ANÁLISE COMPARATIVA ENTRE DOIS VERIFICADORES DE CONSISTÊNCIA E DE COERÊNCIA DE MEMÓRIA**

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Curso de Bacharelado em Ciências da Computação da Universidade Federal de Santa Catarina.

Florianópolis - Santa Catarina, 05 de Novembro 2014.

---

Prof. Dr. Renato Cislaghi
Universidade Federal de Santa Catarina
Coordenador

**Banca Examinadora:**

---

Prof. Dr. Luiz Cláudio Villar dos Santos
Universidade Federal de Santa Catarina
Orientador

---

Prof. Dr. José Luís Almada Güntzel
Universidade Federal de Santa Catarina

---

Prof. Dr. Djones Vinicius Lettnin
Universidade Federal de Santa Catarina

Este trabalho é dedicado aos meus colegas,
aos meus professores e a minha querida família.

## AGRADECIMENTOS

À minha família, pelo apoio incondicional e pelo incentivo em concluir esta etapa da minha vida.

Aos meu amigos, por todo o suporte nas pequenas e grandes causas.

Ao meu orientador, Prof. Dr. Luiz Cláudio Villar dos Santos, pelas significativas sugestões e contribuições.

Aos membros da banca, Prof. Dr. José Luís Almada Güntzel e Prof. Dr. Djones Vinicius Lettni, pelas diversas contribuições recebidas na avaliação do trabalho.

Aos colegas do ECL e NIME. Em especial a Eberle A. Rambo, Leandro S. Freitas e Olav P. Henschel por todas as contribuições diretas na realização deste trabalho.

Ao CNPq, pela bolsa de iniciação científica.

Eu só sei que nada sei.
Sócrates

## RESUMO

Quanto antes um erro de projeto for detectado, menor será seu impacto no custo de desenvolvimento de um sistema computacional. Por isso, é necessário iniciar-se o processo de verificação antes mesmo de ser produzido um protótipo do sistema, ou seja, durante a etapa pré-silício. Nesta etapa, a verificação é realizada em uma representação do sistema sob projeto. O subsistema de memória compartilhada usado em *multicores* é extremamente complexo e sujeito a erros de projeto. A verificação deste subsistema busca averiguar se, para uma dada execução de um programa concorrente, o comportamento observado no *hardware* obedece à sua especificação, i.e. a um modelo de memória formalmente especificado, que captura aspectos de consistência e coerência da memória compartilhada. A complexidade desse problema de verificação depende da observabilidade de eventos em memória. Quando a verificação é feita em protótipo, a observabilidade é limitada pelo *hardware* a um único *trace* por processador. Assim, o problema de verificação é intratável no caso geral. Entretanto, como a observabilidade é virtualmente ilimitada em representações executáveis, a verificação pré-silício recai em problemas mais simples. Por isso, técnicas de verificação pré-silício são mais eficientes e eficazes do que técnicas pós-silício, como foi comprovado, durante os últimos anos, através da comparação experimental entre verificadores pré-silício desenvolvidos por mestrandos da UFSC (e.g. Rambo (2012), Freitas (2012)) e outros que representam o estado da arte em verificação pós-silício (e.g. Shacham et al. (2008)). A presente monografia relata uma comparação experimental da técnica proposta por Freitas com a implementação realizada, em conjunto com Olav Philipp Henschel, do verificador XCHECK (HU et al., 2012) – o estado da arte dos verificadores pós-silício baseados em inferência.



**Palavras-chave:** Verificação. Consistência de Memória. Sistema multi-core.

## ABSTRACT

The sooner a design error is detected, the lower the impact on the cost of developing a computer system. Therefore, it is necessary to begin the verification process even before it is produced a prototype of the system, i.e. during the pre-silicon stage. At this stage, the checking is performed on a representation of the design under verification. The shared memory subsystem used in multicores is extremely complex and subject to design errors. The verification of this subsystem seeks to confirm whether, for a given execution of a concurrent program, the behavior observed in the hardware obeys its specification, i.e. to a formally specified memory model that captures aspects of consistency and coherence of the shared memory. The complexity of this problem depends on the verification observability of events in memory. When the verification is done on the prototype, the observability is limited by the hardware to a single trace per processor. In this case, the verification problem is intractable for the general case. However, as the observability is virtually unlimited into executable representations, pre-silicon verification lies on simpler problems. Therefore, techniques for pre-silicon verification are more efficient and effective than post-silicon techniques, as evidenced, in recent years, by experimental comparison between pre-silicon checkers developed by masters of UFSC (e.g. Rambo (2012), Freitas (2012)) and others representing state of the art post-silicon checkers (eg Shacham et al. (2008)). This monograph presents a experimental comparation between the technique developed by Freitas with the implentation done in conjunction with Olav Philipp Henschel of the checker XCHECK (HU et al., 2012) – the state of the art of post-silicon cherckers based on inference.



**Keywords:** Verification. Memory Consistency. Multi-core system.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

## LISTA DE SÍMBOLOS

**SUMÁRIO**

# 1 INTRODUÇÃO

## 1.1 MOTIVAÇÃO PARA A VERIFICAÇÃO

Todo projeto de *hardware* é construído em cima da sua especificação realizada entre uma fabricante e o seu cliente. Nela estão descritas todas as características e os comportamentos que o cliente deseja para o *hardware*, o qual, por sua vez, a fabricante se compromete a desenvolver.

Este compromisso significa que, caso a fabricante quebre esta especificação (ou se preferir, contrato), não apenas terá que compensar, de alguma forma, o seu cliente como, também, que a sua reputação terá sido prejudicada.

Por estes motivos existe, durante o projeto de criação do *hardware*, atividades que verificam a conformidade do sistema em desenvolvidamento (DUV) com a sua especificação (e.g. ISA e MM).

Toda causa de algum comportamento indesejado pelo sistema com relação à sua especificação é denominada por "erro de projeto".

## 1.2 O QUE VERIFICAR

Todos os componentes do DUV são sujeitos a verificação de três características: (1) funcionalidade (feita pela verificação funcional). Diz respeito a saber se o seu comportamento ocorre de acordo com a expectativa desejada; (2) restrições de desempenho (verificação de desempenho). Concernente aos recursos (e.g. tempo, memória) utilizados pelo seu comportamento; e (3) restrições de prazo (verificação de temporização). Relacionadas à apresentação de certos resultados em um tempo especificado. Para mais informações, o leitor pode consultar tanto o site `www.sei.cmu.edu/cyber-physical/index.cfm` quanto o livro de Baier e Katoen (2008).

Repare que a verificação descrita até então diz respeito a todo o sistema do hardware desenvolvido. Assim, naturalmente, esta verificação também deve ser aplicada aos seus módulos internos como, por exemplo, o sistema de memória, um elemento central que aparece em grande parte dos projetos desta natureza.

Nos últimos anos, a complexidade da memória impeliu pesquisas sobre técnicas específicas para a sua verificação, denominada de verificação de memória em referência a sua especificação ser o modelo de memória (MM). Os resultados de tais pesquisas estão disponibilizados na literatura atual.

No entanto, esta mesma complexidade afeta de tal modo a sua

verificação funcional que as técnicas até então desenvolvidas não têm se mostrado capazes de tratar a contento o problema ideal de sua corretude (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002 apud HENSCHEL, 2014, p. 37): dado um modelo de memória (o qual é a especificação do sistema de memória) e uma implementação de um sistema com múltiplos processadores, verifique se todas as execuções geradas por este sistema satisfazem o modelo para qualquer programa paralelo.

Desta forma, existem duas linhas alternativas: a verificação estática e a verificação dinâmica.

De acordo com HENSCHEL, a verificação estática (ou abordagem matemática) (HENZINGER; QADEER; RAJAMANI, 1999; CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002; ABTS; SCOTT; LILJA, 2003) trata instâncias simplificadas deste problema: busca provar matematicamente a corretude de uma **abstração** da real implementação com a especificação do modelo de memória. Esta abordagem é estática e capaz de encontrar erros nos estágios iniciais do projeto (e.g. erros de protocolo). Entretanto, ela deixa escapar erros de projeto originados durante os estágios posteriores: na implementação (HENSCHEL, 2014, p. 37).

A verificação dinâmica (ou baseada em simulação) limita-se em demonstrar a incorretude do sistema por meio de um contra-exemplo. Ou seja, busca encontrar alguma execução (estimulada por algum programa paralelo, caso de teste) em que o comportamento do sistema é contraditório ao especificado. O leitor pode encontrar mais informações sobre este tipo verificação no livro de Lee e Seshia (2011).

Assim, o ponto forte da verificação dinâmica, segundo HENSCHEL, está na capacidade de exercitar todos os detalhes do sistema: o *hardware* real (e.g. Hangal et al. (2004), Roy et al. (2006), Manovit (2005), Manovit e Hangal (2006), Chen et al. (2009), Hu et al. (2012)), um protótipo (e.g. Lenoski et al. (1990)), e sua representação executável (e.g. Shacham et al. (2008), Rambo, Henschel e Santos (2012), Freitas, Rambo e Santos (2013)) (HENSCHEL, 2014, p. 37).

O seu ponto fraco está na dificuldade em alcançar a completude, i.e. em garantir a inexistência de erros de projeto. Afinal, o resultado de sua verificação diz respeito somente às execuções exercitadas (pelos programas paralelos). E, normalmente, estas últimas são apenas uma amostragem do universo de todas as execuções possíveis do sistema.

Alguns métodos dinâmicos utilizam uma versão confiável do DUV, com relação ao especificado, para facilitar a sua verificação. Esta versão, denominada por modelo de referência, é executada junto com o DUV, permitindo que a verificação se transforme em um teste de equivalência: os resultados obtidos pelo DUV são equivalentes aos obtidos pelo modelo de

referência?

Porém, quando o modelo de referência não está disponível, os métodos necessitam avaliar o conjunto de todas as execuções possíveis do programa paralelo (caso de teste) que são permitidas pelo modelo de memória (especificação). As vezes, o modelo é rígido o suficiente para permitir uma única execução para um certo programa. No entanto, quando isto não ocorre, os métodos necessitam detalhar uma execução e evidenciar a sua conformidade com o especificado no modelo. Caso contrário, os métodos estariam a mercê da possibilidade de esta liberdade dada à memória ter ocultado alguma decisão incorreta por sua parte (i.e. um comportamento indesejado).

O detalhamento de uma execução é feito por meio de uma busca heurística sobre toda execução possível que não omite nenhuma decisão feita pelo sistema de memória.

A fim de reduzir esta busca, algumas técnicas de verificação dinâmica (os verificadores por inferência) utilizam-se do método de inferências, i.e. sempre que pressupõem alguma nova decisão, tomada unicamente pelo sistema de memória, usam as regras especificadas pelo modelo para explorá-la o máximo possível.

Por fim, as características complementares e as desvantagens inerentes às abordagens formais e aos métodos dinâmicos motivam a sua combinação (ABTS; SCOTT; LILJA, 2003) como estratégia para reduzir o esforço total de verificação (HENSCHEL, 2014, p. 37).

Apresentado o contexto acima, há que se esclarecer que esta monografia irá lidar apenas com a verificação funcional do sistema de memória, objeto de seu estudo.

## 1.3 QUANDO VERIFICAR

O custo para corrigir um erro de projeto e, consequentemente, o custo do próprio projeto, aumenta conforme mais tarde este primeiro for detectado ao longo do seu processo de desenvolvimento do produto. Logo, há o interesse em realizar as técnicas de verificação durante o *design* do sistema (fase pré-silício).

Por outro lado, após o *hardware* ser fabricado em silício (i.e. na fase pós-silício) a verificação se torna tão difícil que não há técnica para esta fase que garanta a inexistência de erros de projeto. É por esta razão que a verificação nesta fase é degradada a um "teste de verificação".

A maioria das técnicas para a verificação funcional da memória são projetadas para a fase pós-silício e, quando necessário, são adaptadas para a pré-silício. Afinal, por muito tempo houve a tentativa de reduzir a complexi-

dade da verificação para pós-silício. Isto é, houve uma busca por uma complexidade linear no tempo de verificação com relação ao número de operações de memória, as quais compõem os programas paralelos (os casos de teste).

"Alguns dos verificadores projetados para o uso pós-silício necessitam de modificações no *hardware* do subsistema de memória" (HENSCHEL, 2014, p. 38). E, assim, conseguem alcançar a complexidade linear desejada. Conforme o próprio exemplo que o referido autor cita, DEORIO; WAGNER; BERTACCO (DEORIO; WAGNER; BERTACCO, 2009) modifica a hierarquia de memória e a interconexão para observar o mapeamento entre leituras e escritas e a ordem total de escritas.

Entretanto, corroborando com HENSCHEL, "alguns projetos industriais podem não prever o uso dedicado de *hardware* para a verificação de memória. O que significa que a utilização das técnicas anteriores impactaria em um custo maior para o projeto. É por isto que diversos verificadores pós-silício são inteiramente baseados em *sotfware* (HANGAL et al., 2004; ROY et al., 2006; MANOVIT; HANGAL, 2006, 2006) ou com uma instrumentação trivial do *hardware* (CHEN et al., 2009; HU et al., 2012)" (HENSCHEL, 2014, p. 38)

Dentre estes últimos verificadores, Hu et al. (2012) foi a primeira publicação em que um verificador baseado em *sotfware* atingiu uma complexidade linear no seu tempo de verificação, a qual era exclusiva para os verificadores baseados em *hardware*.

Resumindo, é possível adicionar recursos no sistema de hardware para permitir a realização de técnicas exclusivas para pré-silício na fase pós-silício. Entretanto, quando estes recursos não estão inclusos no projeto do *hardware*, então ocorre o aumento do custo do projeto. No fim, o cliente estaria pagando pelo desenvolvimento do desejado sistema de *hardware*, junto com o custo para assegurar que a sua especificação foi seguida. Uma atitude indesejada, considerando as noções de competição no mercado.

Mais recentemente, foi observado o desenvolvimento de técnicas específicas para a fase pré-silício em que a verificação pode alcançar uma eficiência maior (ou seja, o problema de verificação possui uma menor complexidade). Um resultado obtido por meio da exploração do fato de que esta fase lhes oferece mais informações sobre a execução do sistema de memória, sem incidir no custo do projeto (i.e. a fase possui maior "observabilidade").

A literatura reporta para a verificação pré-silício diversos tipos de técnicas (HENSCHEL, 2014, p. 38): formais (e.g. Henzinger, Qadeer e Rajamani (1999), Chatterjee, Sivaraj e Gopalakrishnan (2002)), dinâmicas (e.g. Shacham et al. (2008), Rambo, Henschel e Santos (2012), Freitas, Rambo e Santos (2013) e híbridas (e.g. Abts, Scott e Lilja (2003)).

## 1.4 JUSTIFICATIVA E ESCOPO DESTA MONOGRAFIA

Durante os últimos cinco (5) anos, uma equipe do laboratório ECL da UFSC (o mesmo laboratório em que o autor desta monografia realizou a sua bolsa de iniciação científica) pesquisou as vantagens em desenvolver um verificador específico para a fase pré-silício. Uma atitude pioneira, já que a maioria dos verificadores dinâmicos eram desenvolvidos para pós-silício.

Os resultados da pesquisa demonstraram que, infelizmente, uma simples adaptação de um verificador pós-silício para a fase pré-silício é realmente incapaz de explorar completamente a queda de complexidade presente entre estas duas fases.

A fim de quantificar a vantagem em desenvolver técnicas específicas para pré-silício em relação à adaptação de técnicas pós-silício, os pesquisadores deste laboratório passaram a realizar experimentos com a finalidade de comparar as técnicas de verificação dinâmica da literatura com os seus verificadores, considerando duas características: a eficácia e a eficiência. Isto significou a necessidade de implementar localmente verificadores do estado da arte propostos por outros autores.

Uma das técnicas importantes para esta comparação é o estado da arte dentre os verificadores por inferência: o verificador XCHECK desenvolvido por HU et al., publicado em Hu et al. (2012).

Portanto, o objetivo deste trabalho foi o de implementar para a plataforma de teste do ECL, o verificador XCHECK, possibilitando, assim, que os colegas comparassem-na com outras técnicas. Outro compromisso, foi o de apresentar a comparação entre o XCHECK e a última técnica desenvolvida pelo laboratório, o MSB (FREITAS; RAMBO; SANTOS, 2013).

## 1.5 ORGANIZAÇÃO DESTA MONOGRAFIA

O restante desta monografia é organizado da seguinte forma.

O Capítulo 2 apresenta o cerne da verificação de memória: os modelos de memória. Primeiramente, será apresentado um exemplo para ilustrar a função destes elementos formais. Em seguida, será relatada a coletânea de modelos de memória presente na literatura, enquanto se evidenciam os debates acerca de quanto um modelo de memória deveria ser relaxado. No final deste capítulo são definidas as ordens básicas entre operações de memória.

O Capítulo 3 formula o problema de verificação de memória para diferentes graus de observabilidade da plataforma.

O Capítulo 4 realiza uma revisão crítica da literatura, esclarecendo a relevância do XCHECK, verificador de interesse desta monografia.

O Capítulo 5 descreve o verificador da forma como este autor, em conjunto com Olav Philipp Henschel, deduziu pelas seguintes publicações: Gibbons e Korach (1994), Chen et al. (2009), Hu et al. (2012). No final do capítulo, são apresentados os aspectos da adaptação deste verificador para um modelo de memória altamente relaxado (Alpha).

O Capítulo 6 descreve brevemente a plataforma de testes, os experimentos realizados e analisa os resultados obtidos, comparando os dois verificadores com as mesmas condições.

Enfim, o Capítulo 7 apresenta as conclusões e as possibilidades de trabalhos futuros.

# 2 MODELOS DE MEMÓRIA (MM)

Quando lidamos com uma memória é natural nos perguntarmos qual o valor que uma leitura irá retornar. Logicamente, este deve ser o último valor escrito no seu endereço correspondente. Uma questão trivial, no caso da leitura ser originada de um programa sequencial executado em um único processador. Entretanto, tal trivialidade desaparece na situação em que processadores distintos executam, independentemente, operações de programas paralelos que compartilham endereços de memória. Assim, como a memória está distribuída em *caches* locais e memória global, pode haver o armazenamento de cópias de um dado endereçado com valores distintos (HENSCHEL, 2014, p. 19).

Para definir o comportamento correto neste último caso deve haver, segundo (HENSCHEL, 2014, p. 19), uma especificação, formulada pelos projetistas da interface *hardware-sotfware*, que defina como e quando o valor de uma operação de escrita pode ser observado por uma operação de leitura. Isto implica em uma abstração da memória compartilhada que, além de integrar parte do modelo de programação de multiprocessadores, combina as regras de consistência e os requisitos de coerência suportados em *hardware*.

Alguns autores reconhecem esta especificação como modelo de consistência de memória (ADVE; GHARACHORLOO, 1996). Outros o denominam apenas de modelo de memória (MM), (DEVADAS, 2013), em razão de que o mesmo captura tanto as regras de consistência quanto os requisitos de coerência.

Portanto, um modelo de memória captura essencialmente dois aspectos de um sistema de memória compartilhada: a relaxação da ordem de programa entre operações de leitura e de escrita (para endereços distintos) e o grau de atomicidade das operações de escrita (para o mesmo endereço) – Adve e Gharachorloo (1996), Arvind e Maessen (2006) apud Henschel (2014, p. 19). O primeiro aspecto define as regras de consistência e o segundo os requisitos de coerência.

Como o leitor pode ver, atualmente, o modelo de memória é bem pertinente. E, de acordo com a literatura disponível (MARTIN; HILL; SORIN, 2012; DEVADAS, 2013), é provável que a programação de propósitos gerais continue a requerê-lo no contexto de multiprocessadores, mesmo em uma escala de centenas de processadores.

No decorrer deste capítulo, o leitor perceberá uma menção corriqueira a dois aspectos dos modelos de memória. O primeiro aspecto diz respeito às ordens entre operações de memória, as quais resultam da especificação dada pelo modelo. O segundo é a questão de quanta liberdade o modelo oferece ao

sistema de memória que o adota, através da relaxação de sua especificação. Isto é, o modelo deixa indefinidas algumas ordens, permitindo, assim, que o próprio sistema de memória as decida (durante sua execução).

A próxima seção traz um exemplo de HENSCHEL onde, dentre outras coisas, ilustra o por quê da relaxação do modelo de memória ter sido evitada por muito tempo pelos projetistas: relaxar o modelo aumentava o trabalho exercido pelos programadores. Em seguida, é apresentada, rapidamente, uma coletânia dos modelos de memória que surgiram ao longo dos anos.

Na seção 2.3 é apresentado o relacionamento entre a relaxação do modelo com a eficiência dos verificadores de memória.

Por fim, este capítulo termina apresentando, na seção 2.4, as cinco categorias de ordem entre as operações de memória.

## 2.1 EXEMPLO ILUSTRATIVO

Ilustrando o efeito do modelo sobre a interface *hardware-sotfware*, a Figura 1 mostra como as operações de um programa paralelo (Figura 1a) são ordenadas por dois modelos diferentes: um que conserva a ordem de programa (Figura 1b) e outro que a relaxa para operações com endereços distintos (Figura 1c). Repare, na explicação a seguir (dada por HENSCHEL), como muda a percepção que o programa possui com relação à execução do sistema memória, conforme é adotado um modelo mais relaxado: adiciona-se a possibilidade do valor escrito em $u$ ser 0.

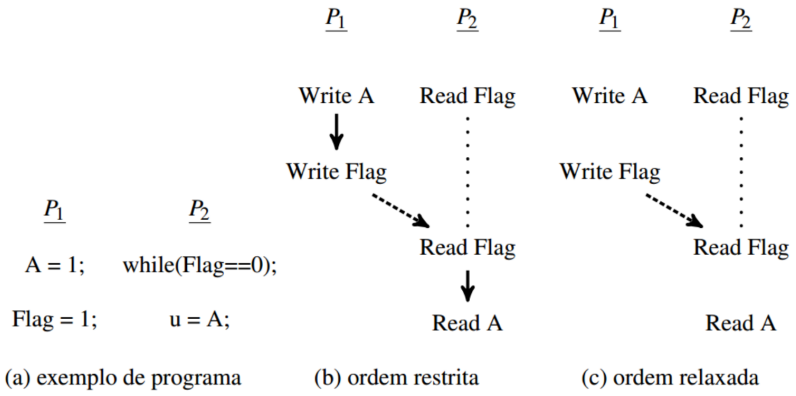



Figura 1 – Exemplo ilustrativo para diferentes modelos de memória (HENSCHEL, 2014, p. 20)

O programa referencia duas variáveis compartilhadas ($A$ e $Flag$), alocadas em memória, e uma variável primitiva ($u$), alocada em registrador, onde deseja-se fazer com que o segundo processador ($P_2$) leia o valor escrito pelo primeiro ($P_1$) na variável $A$. Para tanto, o segundo processador ($P_2$) só realiza a sua leitura quando a variável $Flag$ assume um valor diferente do seu padrão de inicialização (i.e. 0). Complementarmente, o outro processador ($P_1$) realiza a sua escrita em $A$ para, em seguida e só então, liberar o seu parceiro por meio da escrita do valor 1 em $Flag$. No exemplo, a precedência entre as operações de memória sobre $Flag$ é ilustrada pelas setas tracejadas.

Quando executado sob um modelo restrito (Figura 1b), o programa certamente resulta no comportamento planejado. Efeito destes modelos imporem ordem entre operações com endereços diferentes, seguindo a ordem de programa (ilustradas pelas setas contínuas). No entanto, note que conservar estas ordens tende a degradar o desempenho de programas em processadores que admitam execução fora de ordem.

A adoção de um modelo relaxado evita que esta degradação de desempenho impacte em todos os programas, mas também torna menos agradável o modelo de programação (Figura 1c). Como a ordem de programa é revogada na especificação, o programador deveria notar que as suas medidas são insuficientes e permitem dois valores ($u == 0$ ou $u == 1$).

Para resolver esta ambiguidade, o programador necessita restituir as ordens de programa desejadas (como as setas contínuas na Figura 1b). Cada modelo de memória possui o seu próprio mecânismo para alcançar este objetivo: denominado por *safety net*. Por exemplo a utilização de instruções especiais, como as barreiras de memória, os pares *load-acquire/store-release* e *read-modify-write* (RMW) (ADVE; GHARACHORLOO, 1996).

## 2.2 A COLETÂNIA DE MODELOS DE MEMÓRIA

Para evitar sobrecarregar os programadores com questões de sincronização (como ilustrado na Figura 1c), a preocupação principal dos projetistas foi, inicialmente, manter o modelo de programação o mais simples possível. Com este objetivo em mente, o modelo de consistência sequencial (*Sequential Consistency - SC*) (LAMPORT, 1979) foi adotado para conservar integralmente a ordem de programa. Mesmo que a intenção em utilizar *threads* seja em permitir reordenar operações do mesmo processador (quando não existe alguma dependência de dados, ou seja, mantém apenas as ordens entre operações conflitantes).

Procurando aumentar o desempenho, adotaram-se modelos de memória com pouca relaxação em suas ordens: algumas arquiteturas permitiram

que operações de leitura passassem na frente de operações de escrita, levando a diversos modelos de memória. Exemplos são IBM 370 (MAY, 1983), *Total Store Ordering* (TSO) (GARNER et al., 1992), *Processor Consistency* (PC) (GHARACHORLOO et al., 1990; GHARACHORLOO; GUPTA; HENNESSY, 1993), Intel 64 (INTEL, 64) e Godson-3 (CHEN et al., 2009)[1] . Outras arquiteturas também possibilitaram o reordenamento entre operações de escrita, como *Partial Store Ordering* (PSO (SINDHU; FRAILONG; CEKLEOV, 1992; GARNER et al., 1992).

Finalmente, algumas arquiteturas relaxaram todas as ordens, exceto para as operações conflitantes (operações para o mesmo endereço, onde ao menos uma seja escrita) da mesma *thread*. Situação em que estaria sendo alterada a semântica do próprio programa. Esta alta relaxação levou ao modelo *Weak Ordering* (WO (DUBOIS; SCHEURICH; BRIGGS, 1986) e suas variações (ADVE; GHARACHORLOO, 1996), sendo suportada, por exemplo, pelos processadores Alpha (SITES, 1992), PowerPC (MATEOSIAN, 1994), ARMv7 (ARM, 2012) e ARMv8 (ARM, 2013).

Esta atitude resulta do suporte oferecido pelas bibliotecas padrão aos programadores, com relação à sincronização com estes modelos. Portanto, a maioria dos programadores não tiveram mais que se preocupar com as regras de consistência (HENNESSY; PATTERSON, 2012).

O leitor pode encontrar mais informações acerca dos modelos de memória no tutorial de Adve, publicado em (ADVE; GHARACHORLOO, 1996). A Tabela 1, retirada desta referida fonte, apresenta um resumo das relaxações adotadas por alguns modelos de memória. O símbolo ✓ indica que a relaxação

| **Relaxação** | **Ordem W→R** | **Ordem W→W** | **Ordem R→RW** | **Ler outras escrita mais cedo** | **Ler escrita mais cedo** | **Safety net** |
|---|---|---|---|---|---|---|
| SC | | | | | ✓ | |
| IBM 370 | ✓ | | | | | Serializar instruções |
| TSO | ✓ | | | | ✓ | RMW |
| PC | ✓ | | | ✓ | ✓ | RMW |
| PSO | ✓ | ✓ | | | ✓ | RMW, STBAR |
| WO | ✓ | ✓ | ✓ | | ✓ | Sincronização |
| RCsc | ✓ | ✓ | ✓ | | ✓ | Release, acquire, nsync, RMW |
| RCpc | ✓ | ✓ | ✓ | ✓ | ✓ | Release, acquire, nsync, RMW |
| Alpha | ✓ | ✓ | ✓ | | ✓ | MB, WMB |
| RMO | ✓ | ✓ | ✓ | | ✓ | Vários MEMBAR's |
| PowerPC | ✓ | ✓ | ✓ | ✓ | ✓ | Sync |

Tabela 1 – Categorização simples dos modelos relaxados (ADVE; GHARACHORLOO, 1996, p. 12)

[1] Este modelo também permite o reordenamento entre operações de leitura.

correspondente é permitida por implementações simples do modelo correspondente. Em alguns casos, este símbolo também indica que a relaxação pode ser detectada pelo programador (devido a afetar os resultados do programa), as duas excessões são explicadas a seguir. A relaxação "Ler a própria escrita mais cedo" não é detectada com os modelos SC, W, Alpha e PowerPC. A relaxação "Ler outras escritas mais cedo" pode ser detectada com implementações complexas de RCsc (ADVE; GHARACHORLOO, 1996, p. 12).

## 2.3 O DILEMA

Como visto na seção anterior, respaldada em Henschel (2014), as bibliotecas padrão mitigaram um dos limitadores da utilização de modelos de memória com alta relaxação: a necessidade dos programadores em adaptar os seus programas de acordo com as regras de consistência e coerência. Entretanto, ainda há um dilema a ser explicitado.

Relaxar um modelo de memória significa permitir que o próprio sistema de memória decida algumas ordens entre as operações de leitura e de escrita. Afinal, eventualmente, durante qualquer execução, todas as operações de memória serão ordenadas.

Isto, infelizmente, conflita com a motivação da verificação de memória: durante o desenvolvimento, todo o sistema é inconfiável e, portanto, é necessário comprovar que todo o seu comportamento está como o desejado. E a memória não é exceção.

Assim, sempre que as regras de um modelo de memória são relaxadas, aumenta-se o trabalho de verificação. Esta ficará encarregada de testar um comportamento adicional relacionado à falta daquela regra relaxada.

Portanto, a utilização de modelos de memória também é limitada pela complexidade dos verificadores de memória. Quanto mais estes últimos se desenvolverem e abaixarem as suas complexidades de verificação, mais acessível ficará a utilização da relaxação.

## 2.4 AS ORDENS ENTRE OPERAÇÕES DE MEMÓRIA

A partir deste ponto, espera-se que o leitor tenha percebido que tanto o modelo de memória quanto a sua própria verificação lidam com as ordens entre as operações de memória. É em volta destas ordens que todo o problema de verificação se passa.

O modelo de memória remove algumas ordens a fim de dar liberdade para a execução de operações no sistema de memória, enquanto que a

verificação avalia as decisões tomadas por este último (acerca destas mesmas ordens).

Assim, este capítulo termina apresentando, brevemente, cinco ordens tradicionais entre as operações de memória.

### 2.4.1 Ordem de programa

A ordem de programa é aquela determinada puramente pelo programa.

**Definição 1** ***(Ordem de Programa)****. Dadas duas operações diferentes ($u_1$ e $u_2$) no mesmo processador, $u_1$ precede $u_2$ pela ordem de programa (formalmente, $u_1 \xrightarrow{P} u_2$) se e somente se $u_1$ precede $u_2$ na sequência do programa executado no processador. Hu et al. (2012, p. 505)*

As explicações dadas até este ponto da monografia podem ter levado o leitor a acreditar que, na prática, muitos modelos de memória utilizam esta ordem para um grupo específico de operações de memória, permitindo a sua violação em outros casos. Entretanto, como HU et al. formula, a relaxação desta ordem é capturada por meio da definição de uma outra ordem entre as operações de memória: a ordem de processador.

### 2.4.2 Ordem de processador

**Definição 2** ***(Ordem de Processador)****. Dadas duas operações diferentes ($u_1$ e $u_2$)* ***no mesmo processador****, $u_1$ precede $u_2$ pela ordem de processador (formalmente $u_1 \xrightarrow{PO} u_2$) se e somente se $u_1$ precede $u_2$ pela perspectiva de todos os processadores. Hu et al. (2012, p. 505)*

Portanto, esta ordem necessita, também, da definição de como um processador enxerga as ordens entre as operações.

**Definição 3** ***(Realização de uma Operação de Memória)****. Uma leitura r é considerada "realizada" com respeito ao processador $\mathscr{P}$ (durante um instante de tempo no relógio local de $\mathscr{P}$) quando $\mathscr{P}$ emite uma operação de escrita para o mesmo endereço que não afeta o valor retornado por r. Analogamente, uma escrita w é considerada "realizada" com respeito ao processador $\mathscr{P}$ (durante um instante de tempo no seu relógio local) quando $\mathscr{P}$ emite uma operação de leitura, para o mesmo endereço, que retorna o valor definido por w (ou por alguma escrita subsequente). Assim, uma operação $u_1$*

*precede $u_2$ pela perspectiva do processador $\mathscr{P}$ se $u_1$ é realizada, pela perspectiva de $\mathscr{P}$, em um instante de tempo anterior ao que do que $u_2$ é realizada (no relógio local de $\mathscr{P}$).*

### 2.4.3 Ordem de execução

Dentre todas as ordens vistas até este ponto, a ordem de execução é a primeira que pode ordenar operações de diferentes processadores. Além do mais, ela pode ser interpretada como a Consistência de Cache ou a Ordem de Coerência (*write-before* e *read-before orders*).

A ordem de execução é estabelecida com respeito, apenas, pelo influência entre as operações de memória, isto é, ordena as "operações conflitantes".

Para o caso das operações de leitura e de escrita, duas operações de memória são consideradas conflitantes se acessam o mesmo endereço e ao menos uma delas é uma escrita.

**Definição 4** ***(Ordem de Execução)***. *Dadas duas operações diferentes e* ***conflitantes*** *($u_1$ e $u_2$), $u_1$ precede $u_2$ pela ordem de execução (formalmente, $u_1 \xrightarrow{E} u_2$) se e somente se $u_1$ precede $u_2$ pela perspectiva de todos os processadores. Hu et al. (2012, p. 505)*

(HU et al., 2012) observa que a ordem de execução não ordena operações (mesmo que conflitantes) observadas por outras no mesmo processador antes mesmo das primeiras terem sido executadas globalmente. Um comportamento que é permitido por alguns modelos de memória.

### 2.4.4 Ordem global

Enfim, a ordem global combina as ordens de processador e de execução.

**Definição 5** ***(Ordem Global)***. *Uma operação ($u_1$) precede outra operação ($u_2$) pela ordem global se e somente se $u_1$ precede $u_2$ pela ordem de processador, ou $u_1$ precede $u_2$ pela ordem de execução, ou por meio da transitividade, isto é, $u_1$ precede alguma operação ($u$) pela ordem globa,l a qual precede $u_2$ pela ordem global. Formalmente,*

$$(u_1 \xrightarrow{GO} u_2) \iff (u_1 \xrightarrow{PO} u_2) \vee (u_1 \xrightarrow{E} u_2)$$
$$\vee (\exists u \in \mathbb{O} : u_1 \xrightarrow{GO} u \wedge u \xrightarrow{GO} u_2)$$

### 2.4.5 Ordem temporal

A ordem temporal entre operações de memória é obtida através do tempo (exato ou aproximado) em que elas foram executadas globalmente no sistema de memória.

Devido à complexidade dos sistemas de memória (hierarquia de memória, emissão de operações por diversos processadores, exercício do adiantamento de resultados, etc), nem sempre é possível obter os meios necessários para estabelecer esta ordem. Por este motivo, ela é utilizada por poucas técnicas de verificação de memória, a maior parte delas projetada para as etapas pré-silício.

Note, no entanto, que caso esta ordem fosse obtida, se reduziria significativamente a complexidade do problema de verificação. Afinal, neste caso, já se teria a ordem total das operações.

# 3 PROBLEMA-ALVO

Para os verificadores dinâmicos, uma instância do problema de verificação de memória trata da combinação de um programa paralelo, um sistema de memória (DUV) e um modelo de memória. O sistema deve executar o programa conforme o modelo especifica.

As informações de uma instância (execução) qualquer, pertinentes para a verificação, são capturadas por um conjunto de *traces* dos eventos de memória observados durante a execução. Por exemplo, saber qual foi o resultado da operação de memória executada. Estas informações podem ser sintetizadas por meio de um grafo direcionado que representa as relações de ordem entre as operações de memória realizadas.

Este grafo é uma interpretação do problema. Portanto, outra utilidade sua está em identificar a ocorrência de violações de uma instância do problema com relação ao seu modelo de memória. Toda aresta representa uma ordem entre duas operações (os vértices) que, necessariamente, deve acontecer a fim de que estas mesmas operações apresentem os seus resultados observados pelos *traces* por meio de um comportamento permitido pelo modelo de memória. Assim, a única justificativa para a presença de algum ciclo, nestes tipos de grafos, é que o sistema de memória quebrou alguma(s) regra(s) do modelo durante a execução das operações deste ciclo.

Alguns verificadores utilizam a presença de algum ciclo nestes grafos para indicar (como contra-exemplo) a existência de erros de projeto (com relação a uma certa execução).

No entanto, utilizar *traces* que notificam apenas o resultado das operações pode resultar em falsos negativos. Logo, toda técnica construída em cima destes últimos necessita de atitudes adicionais para alcançar a sua completude: a busca heurística, mencionada na introdução desta monografia (seção 1.2), em conjunto com *backtracking* (MANOVIT; HANGAL, 2006).

Infelizmente o *backtracking* resulta em grandes tempos de análise, somados aos já altos tempos de simulação de uma representação de projeto. Ademais, o *backtracking* também limita a escalabilidade a longo prazo para um crescente número de processadores.

A fim de contornar esta limitação, algumas técnicas aumentam os *traces*, realizando o monitoramento em mais de um ponto do sistema de memória.

A tese de mestrado de HENSCHEL, por meio das Figuras 2, “Um programa, seus *traces* e dois comportamentos atômicos”, e 3,“Ordenamento de eventos sob forte atomicidade de escrita”, presentes no Capítulo 2 (HENSCHEL, 2014, p. 30), ilustra de forma clara e didática estas noções acima

mencionadas. Estas são as principais noções usadas em sua formulação para o problema de verificação de memória, a qual esta monografia reutiliza.

| $T_1$ | $T_2$ |
|---|---|
| $(S_1)_a$ | $(L_3)_a$ |
| $(S_2)_a$ | $(L_4)_a$ |

(a) programa

| $P_1$ | $P_2$ |
|---|---|
| $S_1 = (W,a,1)$ | $L_3 = (R,a,1)$ |
| $S_2 = (W,a,2)$ | $L_4 = (R,a,1)$ |

(b) *traces*

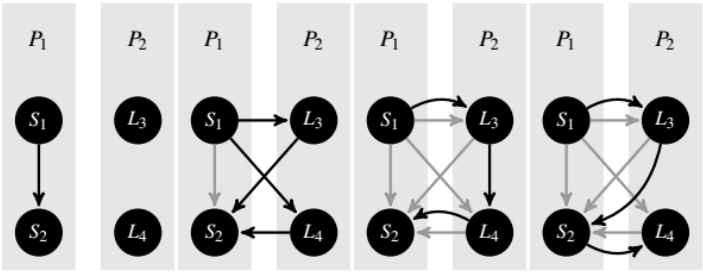
| time | $P_1$ | $P_2$ |
|---|---|---|
| 1 | $S_1 = (W,a,1)$ | |
| 2 | | $L_3 = (R,a,1)$ |
| 3 | | $L_4 = (R,a,1)$ |
| 4 | $S_2 = (W,a,2)$ | |

(c) um comportamento

| time | $P_1$ | $P_2$ |
|---|---|---|
| 1 | $S_1 = (W,a,1)$ | |
| 2 | | $L_3 = (R,a,1)$ |
| 3 | $S_2 = (W,a,2)$ | |
| 4 | | $L_4 = (R,a,1)$ |

(d) outro comportamento

Figura 2 – Um programa, seus traces e dois comportamentos atômicos (HENSCHEL, 2014, p. 30)




(a) ordem de programa (b) ordem inferida (c) uma ordem temporal válida (d) uma ordem temporal inválida

Figura 3 – Ordenamento de eventos sob forte atomicidade de escrita (HENSCHEL, 2014, p. 30)

A seguir, este capítulo vai transcrever a formalização do problema cuja autoria é de HENSCHEL.

Desta forma, a próxima seção determina a observabilidade da plataforma sob verificação, que distingue os diferentes problemas (HENSCHEL, 2014, p. 31). Na seção seguinte, serão definidos formalmente os problemas-alvo, independentemente do modelo de memória (HENSCHEL, 2014, p. 33). Diferentes modelos definem diferentes instâncias do problema.

## 3.1 A OBSERVABILIDADE NA PLAFATORMA

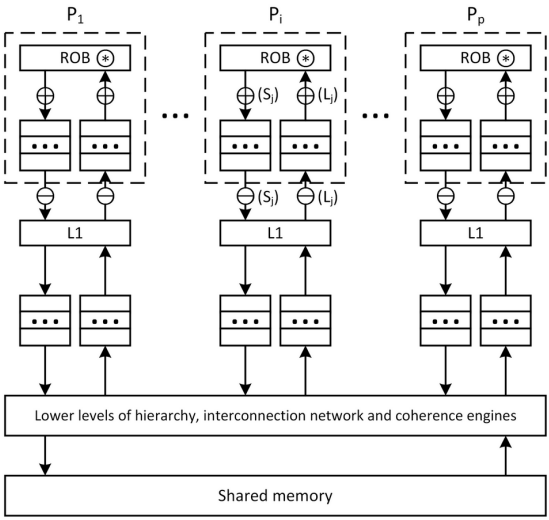



Figura 4 – Um modelo genérico de sistema memória compartilhada (HENSCHEL, 2014, p. 32)

Como HENSCHEL explicou durante a sua dissertação, a Figura 4 apresenta um diagrama esquemático genérico para um multiprocessador com memória compartilhada. Ela ilustra uma arquitetura com $p$ elementos de processamento $(P_1, \ldots, P_i, \ldots, P_p)$ que acessam os seus dados através de sua respectiva *cache* privativa. Assume-se que um *buffer* de reordenamento (*reorder buffer* – ROB) (HENNESSY; PATTERSON, 2012) ou uma estrutura similar é usada para garantir a consolidação das instruções em ordem de processador. *Buffers* de entrada e de saída são destacados nas fronteiras da *cache* de dados privativa. Como os verificadores devem ser amplamente independentes da organização da memória, o diagrama abstrai os níveis mais baixos da hierarquia de *caches*, a rede de interconexão e os mecanismos de coerência.

Na Figura 4, HENSCHEL indica os pontos relevantes onde os eventos de memória podem ser observados por meio dos símbolos $\circledast$, $\oplus$ e $\ominus$. Cada um desses pontos é denominado de monitor.

A observabilidade dos diversos verificadores presentes na literatura pode ser esclarecida por meio deste esquemático. Os verificadores pós-silício (MANOVIT; HANGAL, 2006; HU et al., 2012) são limitados pela observa-

bilidade *intra-chip*, logo eles capturam um único *trace* por elemento de processamento, amostrando eventos como se eles fossem observados em cada monitor $\ominus$. Enquanto que os verificadores pré-silício podem se beneficiar da maior observabilidade de representações de projeto para resolver instâncias mais simples do problema de verificação (como será mostrado na seção 3.2). Repare que isto significa na possibilidade de utilizar mais pontos de amostragem. Por exemplo, dois verificadores pré-silício (RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013) amostram duas sequências de eventos por elemento de processamento, pelos monitores $\oplus$ e $\ominus$. Em Freitas, Rambo e Santos (2013) o ROB é também observado, pelo terceiro monitor $\circledast$. (HENSCHEL, 2014, p. 31-33)

A seguir será mostrada a formulação de HENSCHEL para as instâncias do problema de verificação alvo relevantes para esta monografia.

## 3.2 FORMULAÇÃO DO PROBLEMA

Daqui em diante, $p$ e $n$ denotarão, respectivamente, o número de processadores e o número total de operações de memória.

Primeiramente, HENSCHEL formaliza o principal conceito na verificação de modelos de memória baseados em simulação: o *trace*.

**Definição 6** *Um* ***trace*** *é uma sequência* $(e_1, \ldots, e_i, \ldots, e_m)$*, onde* $e_i = (op, a, value)$ *é um evento de memória tal que* $op \in \{R, W\}$*, a é um endereço e value é um valor (HENSCHEL, 2014, p. 33).*

Como este autor afirmou, *traces* locais $T^1_\ominus, \ldots, T^p_\ominus$ são obtidos amostrando eventos em cada processador, conforme indicado pelos monitores $\ominus$ na Figura 4. Eles podem ser combinados em um *trace* global $T$ entrelaçando os *traces* locais, enquanto se mantém a ordem de amostragem local, i.e. se $e_i$ precede $e_j$ em algum *trace* $T^k_\ominus$, para $k \in [1, p]$, então $e_i$ deve preceder $e_k$ no *trace* global ($T$).

Quando a observabilidade é restringida à amostragem de *traces* nos monitores $\ominus$, o problema pode ser formulado da seguinte forma:

**Problema 1** ***(Verificação baseada em traces)****. Dada uma coleção de* ***traces*** $T^1_\ominus, \ldots, T^p_\ominus$*, existe um* ***trace*** *global T que satisfaça todas as restrições do modelo de memória? (HENSCHEL, 2014, p. 34)*

Esta formulação de HENSCHEL apresenta como a maior parte dos verificadores pós-silício (e.g. Manovit e Hangal (2006), Chen et al. (2009), Hu et al. (2012)) contornam o fato do problema geral ser NP-Completo

(GIBBONS; KORACH, 1994). Estes verificadores utilizam as instâncias do Problema 1, resultadas da inclusão de informações adicionais, tais como a ordem total de escritas (ordenamento de operações de escrita para um mesmo endereço) e o mapeamento de leituras (ligando cada operação de leitura com a operação de escrita que produziu seu valor). Este mapeamento pode ser facilmente realizado quando se atribui para cada escrita um valor único. O problema torna-se polinomial (considerando que $p$ seja constante) quando ambas as informações estão disponíveis (GIBBONS; KORACH, 1994).

HENSCHEL elabora outro problema de verificação sobre o conceito de "comportamento", o qual é originado quando é possível determinar (em tempo de execução ou de simulação) o instante de tempo em que um evento é amostrado. Veremos mais tarde, que esta é a instância de problemas resolvidas pelo verificador de interesse, o XCHECK.

**Definição 7** ***(Comportamento)***. *Um comportamento é uma sequência* $(\beta_1,\ldots,\beta_i,\ldots,\beta_m)$*, onde* $\beta_i = (op,a,value,t)$ *é um evento de memória com marca temporal tal que* $op \in \{R,W\}$*, a é um endereço, value é um valor lido/escrito e t é o instante de tempo em que o evento foi observado. A marca temporal de algum evento* $\beta_i$ *será denotada como* $\tau(\beta_i)$ *(HENSCHEL, 2014, p. 34).*

Assim como HENSCHEL esclareceu em seu trabalho, perceba que de forma similar aos *traces*, os comportamentos locais $\mathrm{B}^1_\ominus,\ldots,\mathrm{B}^p_\ominus$ são obtidos amostrando eventos em cada processador, conforme a Figura 4 indica pelos monitores $\ominus$. Eles podem ser combinados em um comportamento global $B$ entrelaçando os comportamentos locais enquanto se mantém a ordem de marcas temporais, i.e. se $\tau(\beta_i) < \tau(\beta_j)$, então $\beta_i$ deve preceder $\beta_j$ no comportamento global $B$ (HENSCHEL, 2014, p. 34).

Enfim, HENSCHEL formaliza este outro problema de verificação da seguinte maneira:

**Problema 2** ***(Verificação baseada em comportamentos)***. *Dada uma coleção de comportamentos* $B^1_\ominus,\ldots,B^p_\ominus$*, existe algum comportamento global B que satisfaça todas as restrições do modelo de memória? (HENSCHEL, 2014, p. 34)*

O último problema definido pelo referido colega é o utilizado pelos verificadores pré-silício, os quais se beneficiam da observabilidade adicional de representações de projeto e ambientes de simulação. Eles podem resolver instâncias de um problema mais simples do que definidos anteriormente.

**Problema 3** ***(Verificação aumentada baseada em comportamentos)***. *Dada uma coleção de comportamentos* $B^1_\ominus,\ldots,B^p_\ominus$*,* $B^1_\oplus,\ldots,B^p_\oplus$*, existe algum com-*

*portamento global B que satisfaça todas as restrições do modelo de memória? (HENSCHEL, 2014, p. 34)*

A instância resultante do Problema 3 pode ser resolvida em tempo polinomial e com garantias totais de verificação sob forte atomicidade de escrita (RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013) quando comportamentos locais são amostrados por, pelo menos, dois monitores ($\oplus$ e $\ominus$). Henschel (2014, p. 71-74) mostrou que a verificação com complexidade polinominal, para modelos de memória que requerem uma ordem total de escritas, necessita de instâncias do problema com ainda mais informações adicionais.

Como qualquer modelo de memória define que a execução especulativa não pode ter efeitos observáveis, então, operações que são canceladas não precisam satisfazer o modelo adotado. Por esta razão alguns verificadores pré-silício (FREITAS; RAMBO; SANTOS, 2013; HENSCHEL, 2014) monitoram o ROB ($\circledast$) para remover estes eventos especulativos da sua verificação.

# 4 TRABALHOS CORRELATOS

Com o propósito de discutir a relevância do XCHECK, este capítulo inicialmente mostra as técnicas relacionadas com a avaliação aqui pretendida: aquelas baseadas em *sotfware* e que resolvem instâncias dos Problemas 1, 2 e 3.

Estas técnicas, cujas principais características estão descritas na Tabela 2, podem ser divididas em duas categorias: verificação *post-mortem* e verificação *on-the-fly*. É com base nestas categorias que este capítulo se organiza de maneira a falar sobre elas.

### 4.0.1 Verificação post-mortem

Os verificadores desta categoria são caracterizados pela geração de todos os *traces* antes de que a sua verificação possa se iniciar.

A maioria dos verificadores dinâmicos se baseia na detecção de ciclos em grafos orientados, em que os vértices representam operações de memória (escritas e leituras) e as arestas representam uma relação de ordem. Desta forma, todo ciclo significa um paradoxo: todas as operações envolvidas devem terminar antes mesmo de terem se iniciado (e vice-versa).

Na prática, são adotadas relações parciais que, como podem "disfarçar" uma instância inválida do problema, tornam incompleto estes verificadores. Este disfarce acontece quando não há alguma relação de ordem total, estendida da original, que seja uma instância válida do problema.

Para mitigar a ocorrência de falso-negativos, estas ferramentas tentam inferir o maior número possível de arestas. TSOTool (HANGAL et al., 2004) é um exemplo de ferramenta de verificação incompleta que se utiliza desse método e que, posteriormente, foi estendida para se tornar completa (MANOVIT; HANGAL, 2006).

Para eliminar completamente os falso-negativos, é necessário ter absoluta certeza de que o resultado do verificador foi baseado em uma ordenação total. Logo, é necessário inferir todas as possibilidades de arestas que diminuam o número de componentes fortemente conexos do grafo. Isto, infelizmente, significa em seguir uma heurística e, muito eventualmente, fazer um *backtracking*. Método que torna exponecial a complexidade destes verificadores.

O XCHECK (HU et al., 2012), versão aprimorada do LCHECK (CHEN et al., 2009), contorna esta complexidade limitando, principalmente, a realização do *backtracking* em um subespaço cujo tamanho depende apenas

| Referência | Análise | Idéia-chave | Garantias | Número de monitores | Complexidade de pior caso |
|---|---|---|---|---|---|
| Hangal et al. (2004) | *Post-mortem* | Inferência | Nenhuma | 1 | $O(n^5)$ |
| Manovit (2005) | *Post-mortem* | Inferência | Nenhuma | 1 | $O(pn^3)$ |
| Manovit e Hangal (2006) | *Post-mortem* | Inferência | Plenas | 1 | $O((n/p)^p pn^3)$ |
| Roy et al. (2006) | *Post-mortem* | Inferência | Nenhuma | 1 | $O(n^4)$ |
| Shacham et al. (2008) | *On-the-fly* | *scoreboard* | Nenhuma | 1 | $O(p^2n^2)$ |
| Chen et al. (2009) | *Post-mortem* | Inferência | Nenhuma | 1 | $O(p^3n)$ |
| | *Post-mortem* | Inferência | Plenas | 1 | $O(C^p p^2 n^2)$ |
| Hu et al. (2012) | *Post-mortem* | Inferência | Plenas | 1 | $O(C^p p^3 n)$ |
| Rambo, Henschel e Santos (2012) | *Post-mortem* | Emparelhamento em grafos bipartidos | Plenas | 2 | $O(n^6/p^5)$ |
| Freitas, Rambo e Santos (2013) | *On-the-fly* | Múltiplas *scoreboard* | Plenas | 3 | $O(n^2/p)$ |

Tabela 2 – Principais características dos verificadores dinâmicos (HENSCHEL, 2014)

do número de processadores. Um avanço considerável, já que significa em retirar o principal limitador de desempenho dos verificadores atuais (o número de operações) da parte exponencial de sua complexidade.

No entanto, corroborando com Henschel (2014, p. 40), a crescente paralelização dos processadores através do aumento do número de núcleos faz com que esta técnica tenda a se tornar cada vez mais ineficiente.

Afora a detecção de ciclos, mas ainda baseado em grafos, há o método de Emparelhamento Estendido de Grafo Bipartidos (RAMBO; HENSCHEL; SANTOS, 2012) que foi projetado especialmente para a verificação de representações executáveis de plataformas multiprocessadas. Nesta outra abordagem, a invalidade de uma instância é dada através de um teste de equivalência entre eventos originados de *traces* **diferentes**:

> Devem existir dois monitores em cada núcleo de processamento: um na saída da unidade de consolidação de resultados (*commit unit*) e outro na interface com a memória privativa. Um grafo bipartido é, então, construído a partir destes monitores; os vértices representam as operações de memória e as arestas representam a equivalência. Outro grafo, com os mesmo vértices, representa o ordenamento de operações, imposto pelo modelo de memória, através das suas arestas. Com base nestes dois grafos é possível determinar se o comportamento global, usando relógios globais de Lamport (LAMPORT, 1978), que utilizam marcas temporais geradas pelos monitores para cada operação. A partir deles é criado um registro de execução global, o qual é analisado por um algoritmo que determina se ele viola ou não as regras do modelo de memória. (HENSCHEL, 2014, p. 40)

### 4.0.2 Verificação on-the-fly

Por outro lado, se encontram os verificadores *on-the-fly*, os quais procuram invalidar uma instância antes mesmo da execução desta ter sido terminada.

*Scoreboard* Relaxado (SHACHAM et al., 2008) é uma técnica (genérica) de verificação que possui um princípio de funcionamento diferente do visto até o momento. Ela mantém uma tabela (a.k.a. *scoreboard*) de possíveis valores para cada endereço de memória, a qual valida o valor observado por cada uma das leituras. Durante a execução, esta tabela é atualizada de acordo

com cada operação observada. Quando uma escrita é feita na memória, a tabela recebe um novo valor no endereço da operação; quando é uma leitura, os possíveis valores são filtrados. Esta última alteração reduz o número de possibilidades, mas pode significar a remoção de um valor que havia validado (incorretamente) uma leitura anterior. Uma situação possível quando leituras se sobrepõem ou completam com uma quantidade variável de tempo. A fim de superar esta limitação, o *scoreboard* relaxado também necessita utilizar o *bactracking*. Isto, no entanto, não diminui a principal vantagem desta ferramenta com relação às técnicas anteriores, isto é, o fato de realizar a análise simultaneamente à execução do programa e poder, assim, detectar erros mais rapidamente (antes mesmo que a simulação termine).

A técnica proposta em Freitas, Rambo e Santos (2013) usa o mesmo mecanismo de verificação global da técnica de RAMBO; HENSCHEL; SANTOS. A sua verificação local, no entanto, pode ser realizada *on-the-fly* devido ao suporte de um terceiro monitor (em cada núcleo) responsável por distinguir as leituras consolidadas das leituras especulativas. Isto impede estas últimas de serem indicadas como erros (um diagnóstico de falso positivo) já que, enquanto as operações de leitura e escrita são monitoradas na interface com a memória, um monitor na unidade de consolidação se encarrega de observar as operações que foram consolidadas e outro no *buffer* de reordenamento observa quais operações concluídas foram canceladas. Por sua vez, a verificação global é realizada por meio de um *scoreboard* relaxado para cada processador, o qual é atualizado a cada operação monitorada, verificando se cada operação que chega à memória tem uma correspondente observada em algum dos outros monitores e se não há intervalo ilegal da ordem especificada pelo modelo de memória (HENSCHEL, 2014).

## 4.1 O INTERESSE NA COMPARAÇÃO QUANTITATIVA ENTRE AS TÉCNICAS DE VERIFICAÇÃO

Esta análise da literatura evidencia indícios de deficiências dos verificadores dinâmicos, os quais foram reportados em Henschel (2014, p. 42). Eles merecem ser confirmados por meio de uma avaliação experimental comparativa e de uma análise quantitativa de eficiência e de eficácia.

Repare que a importância de tal comparação, entre verificadores de classes diferentes não diminui, mesmo com o fato de ter sido raramente apresentada na literatura (RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013; HENSCHEL, 2014).

Por fim, perceba que o objetivo desta monografia está ligada em responder se um verificador para pré-silício (FREITAS; RAMBO; SANTOS,

2013) realmente consegue ser melhor que um verificador de inferências adaptado para pré-silício, mesmo quando a complexidade deste último é melhor (linear) do que o primeiro (quadrático) com relação ao principal limitador de desempenho atual (o número de operações). Os experimentos também podem ilustrar como a diferença entre eles irá se portar no futuro (i.e. conforme o número de processadores crescer).

# 5 XCHECK

Em 2012, HU et al. publicaram o XCHECK, um verificador de memória *post-mortem* para a etapa pós-silício com complexidade de verificação linear sobre o número de operações de memória, o principal limitador de desempenho. Tal complexidade foi impactante devido ao fato deste verificador não necessitar da ajuda de nenhum *hardware* dedicado.

A ideia chave do XCHECK é de aproximar a ordem temporal (seção 2.4.5) para fazer uma verificação localizada, referente apenas aos pares de operações de memória que não são ordenados por ela. Uma herança do seu predecessor, o LCHECK, publicado em Chen et al. (2009).

Para os verificadores de inferência, a validade de uma instância (do Problema 2) é dada pela existência de uma ordenação total e válida das operações de memória. Logo, é necessário conjecturar todas as ordens inexistentes, uma tarefa que estes dois verificadores realizam com o auxílio do grafo de fronteiras (GIBBONS; KORACH, 1994), o qual será descrito na seção 5.3.

Porém, a construção dinâmica deste grafo significa a necessidade de validar os seus elementos para não haver a conjectura de ordens proibidas pelo modelo de memória. E, é neste ponto em que o XCHECK se diferencia do LCHECK: o algoritmo de checagem de ciclos deste primeiro (seção 5.6) possui uma complexidade significativamente menor que o do segundo: $O\left(p^2\right)$ e $O\left(np^3\right)$, respectivamente. Esta queda de complexidade é resultado do XCHECK explorar a relação existente entre as arestas do grafo de fronteira que constituem em um caminho entre a fronteira inicial e a fronteira final (seção 5.3).

A inicialização do XCHECK consta em, principalmente, computar os intervalos de espera (seção 5.2) de todas as operações e construir o mecanismo que evidencia violações ao modelo de memória: o grafo de execução (seção 5.4). Tarefas que podem ser realizadas respectivamente em $O(np)$, p. 59, e $O\left(np^3\right)$, p. 69.

O restante de sua verificação constitui em atravessar o grafo de fronteiras enquanto é realizado, para cada um de seus $O(n\,C^p\,p)$ movimentos de fronteira, o seu método de inferências (seção 5.5) e o de checagem de ciclos (seção 5.6), cujas complexidades são respectivamente de $O\left(p^2\right)$ (para cada aresta adicionada no grafo de execução), p. 69, e de $O\left(p^2\right)$, p. 71. Totalizando na complexidade de $O\left(n\,C^p\,p^3\right)$.

Na verdade, como o leitor descobrirá no decorrer deste capítulo, a utilização do método de inferências durante o atravessamento no grafo de fronteiras não foi explicitada pelos autores durante sua publicação (HU et al., 2012). O autor desta monografia supos esta sua utilização em conjunto com

Olav Philipp Henschel.

As seções seguintes descrevem cada um destes elementos do XCHECK por meio de uma compilação das explicações dadas em Gibbons e Korach (1994), Chen et al. (2009), Hu et al. (2012). Portanto, em um primeiro momento, estará sendo mostrado como o XCHECK é capaz de resolver as instâncias do problema, considerando modelos que não reordenam as operações de escrita, como, por exemplo, o modelo sequencial e o *Godson-3 consistency* (CHEN et al., 2009, p. 384).

O capítulo finaliza apresentando (na seção 5.7) uma visão geral dos detalhes da implementação realizada do XCHECK, incluindo a explicação sobre como adaptá-lo para um modelo de memória bastante relaxado.

Antes de mais nada, a próxima seção ilustra o comportamento deste verificador para uma instância simples do problema. Desta forma, o leitor poderá entender melhor todos estes aspectos do verificador enunciados acima.

## 5.1 EXEMPLO ILUSTRATIVO

Nesta seção o leitor será apresentado a um exemplo da verificação feita pelo XCHECK. Porém, para fins de simplificação, será omitida a ordem de tempo físico, responsável por otimizar sua verificação.

Considere que uma dada instância do problema (i.e. programas paralelos) foi submetida ao sistema de memória em inspeção (*DUV*), gerando o comportamento descrito nos *traces* da Figura 5, e que o modelo de memória adotado é o sequencial, fiel à ordem de programa.

| $P_1$ | $P_2$ |
|---|---|
| $S_1 : (W, A, 1)$ | $S_3 : (W, B, 3)$ |
| $S_2 : (W, B, 2)$ | $L_4 : (R, B, 2)$ |
| - | $L_5 : (R, A, 1)$ |

Figura 5 – Um comportamento

Os *traces* esclarecem a existência de dois núcleos de processamento ($P_1$ e $P_2$) que emitiram operações ao sistema de memória. Cada linha do *trace* caracteriza uma operação emitida por um dado núcleo. Por exemplo, a primeira operação emitida por $P_1$ é de escrita ($W$) ao endereço $A$ com o valor 1. Esta operação é identificada como $S_1$, onde $S$ abrevia o nome de sua instrução (*store*) e 1 é o seu identificador.

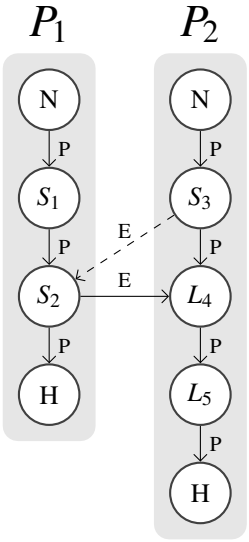
Observe ainda que o segundo núcleo ($P_2$) possui uma operação de leitura ($L_4$) cujo valor obtido foi escrito por $S_2$, uma escrita do outro

núcleo ($P_1$).

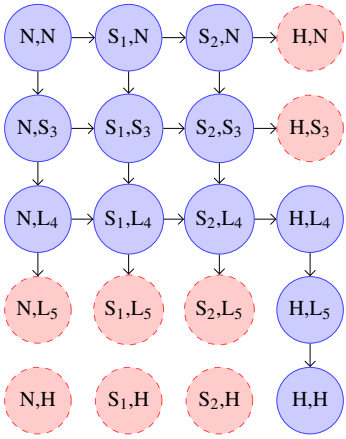
A primeira tarefa realizada pelo XCHECK é de esmiuçar os *traces* revelando, assim, o máximo possível das ordens globais entre as operações executadas pelo sistema de memória. A fim de organizar tudo o que for descoberto, o XCHECK constrói um grafo, denominado por Grafo de Execução (Figura 6a), em que cada operação de memória é representada como um vértice e as ordens conhecidas são representadas por arestas. Além disso, são adicionados vértices para capturar as noções do estado inicial (operação nula, *null*, *N*) e do estado de parada (operação de parada, *halt*, *H*) do sistema de memória com relação a cada núcleo de processamento.

Caso este exemplo estivesse utilizando a ordem de tempo físico, haveria uma maneira adicional de revelar as ordens globais entre as operações de memória. Isto é, dependendo dos intervalos de tempo, o grafo da Figura 6a poderia apresentar arestas adicionais.

Como o modelo de memória é sequencial, as primeiras ordens observadas são relativas à ordem de programa (seção 2.4.1), cujas arestas foram rotuladas com "P". Note como existe uma única ordenação possível entre operações emitidas pelo mesmo núcleo.




(a) Grafo de Execução




(b) Grafo de Fronteiras

Figura 6 – Os dois recursos do XCHECK para ordenar as operações de memória do comportamento da Figura 5

Em seguida, o verificador utiliza-se do fato de que se uma leitura obtiver o seu valor a partir de uma dada escrita então, necessariamente, a segunda operação antecede a primeira. Como esta característica refere à ordem de execução (seção 2.4.3), ela é representada no grafo por meio de setas contínuas rotuladas com "E".

Repare que este exemplo supõe, assim como o próprio verificador, que toda operação de escrita escreve um valor único. Desta maneira, existe uma única escrita responsável pelo valor lido por uma leitura qualquer – um grande facilitador para obter a ordem de execução. Além do mais, esta propriedade pode ser facilmente adquirida pela manipulação das instâncias (i.e. programas paralelos).

Por fim, ocorre a adição (no Grafo de Execução) de ordens para garantir que toda leitura ($r$) realmente obterá o seu valor a partir da escrita ($w$) que o escreveu (Figura 7).


(a) Dado uma escrita anterior (b) Dado uma escrita posterior

Figura 7 – As ordens necessárias para garantir a leitura de um dado valor

O primeiro cuidado a ser tomado é com relação à existência de uma operação de escrita anterior (Figura 7a): caso haja alguma escrita ($w$') que precede esta leitura ($r$), por meio de alguma ordem qualquer (i.e. pela ordem global), então é necessário que esta última operação ($w$') seja anterior à escrita desta leitura ($w$), ou seja, $w' \xrightarrow{E} w$. Similarmente, caso haja alguma escrita ($w$') que sucede a escrita desta leitura ($w$), então, é necessário que esta última ($w$') também suceda a leitura ($r$): $r \xrightarrow{E} w'$ (Figura 7b).

Uma situação especial deste primeiro caso (Figura 7a) ocorre quando a precedência entre $w$' e $r$ é dada pela ordem de programa ($w' \xrightarrow{P} r$), onde é fácil constatar a necessidade da aresta $w' \xrightarrow{E} w$.

No entanto, nos outros casos esta é uma tarefa difícil. E, é por este motivo, que o XCHECK realiza a adição destas arestas por meio do seu método de inferência, utilizando uma aproximação da ordem temporal (a ordem de tempo físico) para diminuir a sua computação.

Repare que, neste exemplo, existe apenas uma aresta referente a estes dois casos para ser adicionada ao grafo de execução: $S_3 \xrightarrow{E} S_2$. Ela está representada na Figura 6a por meio de uma aresta tracejada.

A utilização do grafo de execução é, sem dúvida, um recurso poderoso para a verificação de memória. Além dele capturar a transitividade das ordens por meio de caminhos (sendo desnecessário adicioná-las), proporciona uma maneira fácil de identificar a inconsistência de uma instância com o modelo de memória adotado: qualquer ciclo indica a existência de uma inconsistência. Fato decorrente do ciclo significar que toda operação contida nele não deveria existir, afinal, estas operações devem terminar antes mesmo de terem começado.

Ainda assim, sua utilização é dependente das ordens conhecidas. Logo, toda ordem desconhecida incapacita, potencialmente, o verificador em detectar a invalidez de uma instância.

Com esta limitação em mente, o XCHECK utiliza-se do grafo de fronteiras proposto em Gibbons e Korach (1994). A Figura 6b apresenta ao leitor o grafo de fronteiras que seria criado, dinamicamente, pelo verificador para analisar o comportamento descrito na Figura 5.

O grafo de fronteiras permite modelar uma execução qualquer do sistema de memória, como a responsável pelo comportamento deste exemplo, através de um caminho da fronteira inicial ($N$, $N$) para a fronteira final ($H$,$H$). Além disso, como o verificador só se interessa por execuções fiéis ao modelo de memória, o grafo pode ser reduzido, eliminando as fronteiras em que se identifica alguma inconsistência com o modelo. Tais fronteiras são representadas por círculos tracejados e coloridos em vermelho.

Veja que esta utilização do grafo de fronteiras é coerente com o que as fronteiras inicial e final representam. A fronteira inicial é constituída de operações nulas, logo, representa que o sistema de memória não executou nenhuma operação. Já a fronteira final, constituida de operações de parada, representa que todas as operações de memória já foram executadas.

Complementando estes conceitos, há a interpretação do movimento de fronteiras. Considere as fronteiras ($N$, $S_3$) – 2ª linha, 1ª coluna – e ($N$, $L_4$) – 3ª linha, 1ª coluna. O movimento (aresta) entre estas duas fronteiras indica que a operação de escrita $S_3$ acabou de ser globalmente executada, isto é, a partir deste momento todos os processadores deverão observar no endereço $B$ o seu valor (3).

Agora, observe as fronteiras ($N$, $L_4$) – 3ª linha, 1ª coluna – e ($N$, $L_5$) – 4ª linha, 1ª coluna. Esta última fronteira é inválida já que a operação de leitura $L_4$ só pode ser executada após a sua respectiva escrita ($S_2$). É por esta razão que a única fronteira válida que permite executar globalmente esta leitura é ($H$, $L_4$).

Veja que o XCHECK pode reconhecer as fronteiras como inválidas a partir das ordens no grafo de execução, do método de inferências e das ordens de tempo físico. Assim, o XCHECK só é incapaz de detectar o erro

de projeto oriundo de alguma leitura receber um valor que nenhuma escrita é responsável. Por exemplo, a leitura $L_4$ ler um valor inexistente (4).

Para encobrir esta exceção basta checar se, para todo valor lido por alguma leitura, ou existe uma escrita responsável por ele – a qual deve ser para o mesmo endereço – ou este valor é igual ao contido na memória antes da execução (da instância em questão) ter sido iniciada.

## 5.2 ORDEM DE TEMPO FÍSICO

Como visto na Seção 2.4.5, a ordem temporal por si só soluciona o problema de verificação, já que realiza a ordenação total das operações de memória. No entanto, é difícil de obtê-la durante a fase pós-silício sem o auxílio de *hardwares* dedicados.

Ainda que o XCHECK utilize apenas uma aproximação desta ordem, as otimizações conseguidas são satisfatórias. Esta aproximação evidencia novas ordens, permitindo, assim, a realização de uma verificação localizada.

### 5.2.1 A Ordem de Tempo Físico

A dificuldade em encontrar a Ordem Temporal está em saber o tempo exato em que uma operação de memória é executada globalmente. O que necessita considerar os estados internos, as *caches* e a própria rede de interconexão dos processadores (CHEN et al., 2009, p. 384).

Por esta razão, CHEN et al. propõem a redefinição da ordem temporal sobre dois instantes temporais, um intervalo de tempo. Atitude que pressupõe que para toda operação seja relativamente fácil monitorar o acontecimento destes dois instantes.

**Definição 8** ***(Intervalo de Tempo)****. O intervalo de tempo de uma operação (u) é dado por dois instantes de tempo no relógio global,* $[\tau_s(u);\ \tau_e(u)]$*. De maneira que o seu tempo de realização,* $\tau_p(u)$ *– instante em que foi executada globalmente – esteja entre o tempo de início,* $\tau_s(u)$*, e o tempo de término,* $\tau_e(u)$*:* $\tau_s(u) \leq \tau_p(u) \leq \tau_e(u)$ *(HU et al., 2012, p. 506).*

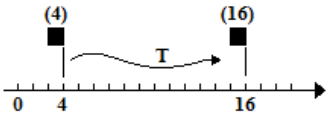
**Definição 9** ***(Ordem de Tempo Físico)****. Se o tempo de término de uma operação (u) é menor do que o tempo de início de alguma outra operação (v), então pode-se dizer que u precede v pela ordem de tempo físico. Formalmente,* $\tau_e(u) < \tau_s(v) \iff u \xrightarrow{T} v$ *(CHEN et al., 2009, p. 384).*

Em Chen et al. (2009), os autores demonstraram que a ordem de tempo físico é consistente com todas as ordens básicas (descritas na seção 2.4, p. 35).

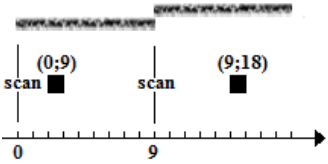
Portanto, ainda é possível seguir a prática comum dos verificadores por inferência de validar uma instância do problema pela inexistência de ciclos no grafo de execução. Assim como comentado brevemente na seção anterior.

No entanto, é importante manter em mente que o tempo prático de verificação destas técnicas (que utilizam a ordem de tempo físico) varia conforme o tamanho dos intervalos de tempo. Afinal, quanto maior forem os intervalos, menos ordens de tempo físico existem, considerando que o tempo de realização das operações tenha sido fixado.

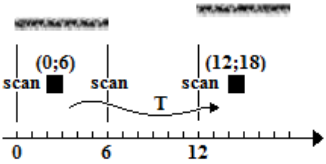
Para esclarecer este fato, observe a Figura 8, a qual apresenta três cenários para a ordem temporal entre duas operações de memória (representadas por quadrados).




(a) Tempo de realização




(b) *Scans* a cada 9 unid. de tempo




(c) *Scans* a cada 6 unid. de tempo

Figura 8 – Três cenários sobre ordens temporais, considerando duas operações realizadas globalmente nas unid. de tempo 4 e 16, respectivamente

No primeiro cenário (Figura 8a) é conhecido o instante de tempo em que cada operação foi globalmente realizada. Portanto, é possível utilizar a ordem temporal propriamente dita (seção 2.4.5) e, consequentemente, é percebida uma precedência entre as duas operações ($4 < 16$). Enquanto isso, os outros cenários utilizam a relaxação da ordem temporal que está sendo apresentada: a ordem de tempo físico.

O segundo cenário (Figura 8b) monitorou o sistema de memória (com *scans*) a cada nove (9) unidades de tempo. Observe que a proximidade dos intervalos de tempo gerados incapacitou a detecção de precedência entre estas operações. Afinal, como será explicado adiante (Figura 9), neste cenário o

tempo de término da primeira operação é igual ao tempo de início da segunda.

Porém, no terceiro cenário (Figura 8c), a ordem de tempo físico consegue capturar a precedência entre estas operações. Os intervalos de tempo ficaram pequenos o suficiente ao ter sido aumentada a frequência dos *scans* para um a cada seis unidades de tempo.

### 5.2.2 Intervalo de Espera

A chave para a complexidade linear do XCHECK está em mensurar a quantidade de pares de operações que não possuem uma ordem de tempo físico entre si. Afinal, o maior custo do processamento de um verificador por inferência está no seu *backtracking*, realizado somente sobre os pares de operações que não possuem alguma ordem entre si. Incluindo a ordem de tempo físico.

Para uma operação de memória qualquer ($u$) é possível definir o conjunto das operações que não possuem alguma ordem de tempo físico com $u$, isto é, o intervalo de espera de $u$: $I(u)$. É a cardinalidade destes conjuntos que desejamos mensurar.

Observe que os instantes dos intervalos são referentes a um relógio global, o qual muda periodicamente. E, enquanto ele mantiver um certo valor temporal, um número constante de operações de memória é emitida por cada processador. Número que assume no máximo, o tamanho do *buffer* de reordenamento que liga cada processador com a sua *cache* privada (como visto no esquema da Figura 4).

Analogamente, pode-se afirmar a existência de uma constante para o número de operações finalizadas (*committed*) e que foram emitidas por um certo processador, enquanto o relógio global mantém fixo um dado valor temporal.

Assim pode ser concluído que, enquanto o relógio global mantiver um valor temporal fixo, o número de operações emitidas e finalizadas está na ordem de $O(p)$.

Agora, repare que as operações presentes no intervalo de espera de uma operação qualquer, $I(u)$, devem, necessariamente, terem sido emitidas antes do relógio global assumir o instante $\tau_e(u)+1$. Assim como terem sido finalizadas enquanto ou depois do instante $\tau_s(u)$. Caso contrário haveria, respectivamente, $u \xrightarrow{T} v$ ou $v \xrightarrow{T} u$. Ou seja, o tamanho de $I(u)$ pode ser mensurado (por cima) através da quantidade de operações de memória emitidas e finalizadas desde que o relógio global iniciou o instante $\tau_s(u)$ até assumir o instante $\tau_e(u)+1$.

Logo, este tamanho pode ser expresso matematicamente da seguinte

forma: $\forall u \in \mathbb{O}, |I(u)| = M \times B \times p = C \times p$, considerando duas constantes ($B$ e $M$) – independentes da instância do problema. Todos os processadores utilizam *buffers* de reordenamento com o mesmo tamanho ($B$). E existe um tempo de execução máximo ($M$) para qualquer operação de memória: $\forall u \in \mathbb{O}, |((\tau_e(u)+1) - \tau_s(u)+1)| <= M$. A constante $C$ é utilizada para simplificar a notação ($C = M \times B$).

Portanto, a cardinalidade de um intervalo de espera é independente do número de operações de uma instância do problema. Depende apenas do número de processadores ($p$): $O(p)$.

### 5.2.3 Obtendo os intervalos de tempo

Como HU et al. explicou em sua seção 5.2 (*Test Program Execution Phase*) (HU et al., 2012, p. 511), os intervalos de tempo podem ser obtidos dinamicamente (e registrados nos *traces*) durante a execução da instância do problema pelo sistema de memória. Há diferentes métodos para obter estas informações e dois deles são enfatizados por HU et al., já que não necessitam de *hardwares* dedicados.

O primeiro método é o *PC sampling*, em que, periodicamente, os valores do *Program Counter* (PC) são capturados em *shift registers* e um *scan* realiza uma cadeia de *shifts* para levar estes valores para um pino de *debug* (e.g. os pinos *JTAG*, *TDI*, *TDO*).

O segundo método faz um compromisso com o *software* ao impor uma restrição nos programas de teste: um grupo específico de instruções é inserido nos programas paralelos (casos de teste) a fim de ler e registrar periodicamente os valores do *program counter* e *cycle counter* (dois contadores típicos de desempenho).

Com a informação do *program counter* e o conhecimento sobre o tamanho da janela de instruções, HU et al. deriva o intervalo de tempo das operações de memória.

Como a Figura 9 ilustra, todo *scan* ocasiona tanto na atribuição de tempos de término, para as operações não-finalizadas, quanto de tempos de início, para todas as futuras operações. Por exemplo, como o *scan2* (realizado no instante 200 do relógio global) ocorreu durante o *PC* 15, então, as únicas operações que poderiam estar executando durante este ocorrido são aquelas cujo *PC* está entre 13 e 17, considerando que a janela de instruções tenha tamanho três (3). Com este fato em mente podemos atribuir, para as operações com *PC* entre 4 e 12, o tempo de término de 200. As operações com *PC* de 0 à 4 já possuem um tempo de término menor (100). Por fim, o *scan2* também nos informa que, com certeza, todas as operações com *PC* maior que 17 só

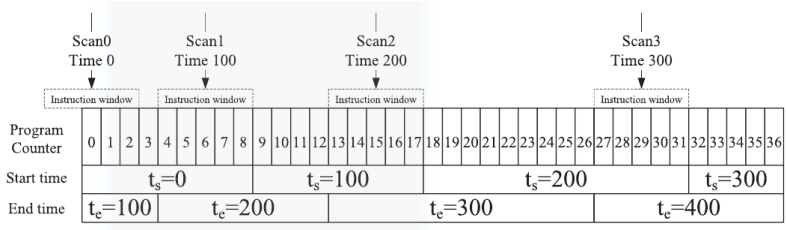



Figura 9 – Os instantes de tempo $\tau_s$ e $\tau_e$ das operações de memória baseados no *program counter* e no tamanho da janela de instruções (igual a 3). (HU et al., 2012, p. 511)

serão iniciadas a partir deste instante (200). Logo, as operações entre 9 e 17 recebem, como tempo de início, o instante do último *scan*, isto é, 100.

Observe que esta construção dos intervalos de tempo, ilustrada acima e descrita por HU et al., causa um "esquecimento" de qualquer reordenação que o sistema de memória possa ter realizado. Desta maneira é possível construir os intervalos de espera em $O(np)$, explorando o fato de que aumentar o *program counter* aumenta tanto o tempo de início quanto o tempo de término das operações de memória.

Para entender a relação desta propriedade com a construção dos intervalos de espera, repare que, para uma certa operação ($u$), a última operação ($v$) que a precede por meio da ordem de tempo físico recebe o seu tempo de término através do penúltimo *scan*, com relação ao que gerou o tempo de início de $u$. Este *scan* sintetiza toda operação que precede $u$ pela ordem de tempo físico.

Analogamente pode ser descrita a primeira operação ($w$) tal que $u$ precede $w$ pela ordem de tempo físico. Logo, existe um *scan* que sintetiza todas as operações que sucedem $u$ por esta mesma ordem.

Conhecendo estes dois *scans* e a janela de instruções é possível percorrer, somente, todas as operações que não possuem alguma ordem de tempo físico com $u$, isto é, aquelas que pertencem ao intervalo de tempo de $u$, $O(p)$ operações.

## 5.3 GRAFO DE FRONTEIRAS

GIBBONS; KORACH realizaram um estudo (GIBBONS; KORACH, 1994) sobre a verificação de consistência de memória onde apresentaram uma maneira eficaz de formar os vetores de teste para construir a ordenação total

de uma instância do problema de verificação.

### 5.3.1 Fronteira de Memória

O objetivo da elaboração do conceito de fronteira de memória está em capturar um momento qualquer da execução de uma instância do problema. Considerando, claro, que o sistema de memória comporte-se conforme o especificado pelo seu modelo de memória.

Idealmente, a fronteira deve ser capaz de nos informar sobre como todos os processadores perceberam as operações emitidas para o sistema de memória. No entanto, GIBBONS; KORACH se satisfazem em capturar o momento em que uma operação de memória é executada globalmente, o que já é suficiente para definir o primeiro.

**Definição 10** ***(Fronteira de Memória)***. *Uma fronteira é uma coleção das operações, $\{f_1, f_2, \ldots, f_p\}$, em que cada processador (i) está aguardando a sua devida operação ($f_i$) a ser globalmente executada e, desta maneira, poder emitir a próxima operação de sua sequência, $\mathbb{O}_i$. Formalmente,*

$$\forall i \in [1, p],\ f_i \in \mathbb{O}_i \ \vee\ f_i = null \ \vee\ f_i = \mathbb{HALT}.$$

*A operação nula (*$\mathbb{NULL}$*) indica que nenhuma operação de um certo processador (i) foi globalmente executada pelo sistema de memória. Enquanto que a operação de parada (*$\mathbb{HALT}$*) indica que todas as operações de um certo processador (i) já foram globalmente executadas (GIBBONS; KORACH, 1994, p. 181).*

A fronteira captura toda operação candidata a ser a próxima operação globalmente executada. Efeito de que, ao modelar uma execução de memória por meio das fronteiras, nada se diz sobre a execução propriamente dita da operação no sistema de memória, as operações são "não emitidas" (aguardando entrar na fronteira), "emitidas" (dentro da fronteira) ou "globalmente executadas" (já saíram da fronteira).

O modelo sequencial define uma única ordenação válida para as operações emitidas por um mesmo processador. Logo, considerando que o sistema de memória tenha se comportado corretamente, a presença de uma operação ($f_i$) na fronteira implica em uma única sequência ($S_i$) constituída somente pelas operações deste mesmo processador que já foram executadas globalmente. Não há outra sequência permitida pelo modelo em que se justifique esta operação ($f_i$) possa ser globalmente executada.

Com isto, uma fronteira especifica uma parte de toda possível ordenação total em que as operações de memória foram globalmente executadas, de

modo que os processadores se encontrem no estado descrito por esta fronteira e, também, considerando uma execução correta do sistema de memória. O que falta é a ordenação entre operações de processadores diferentes, ou seja, realizar um entrelaçamento das sequências $S_i$. E esta tarefa é incumbida ao conceito de extensão de fronteira (as arestas do grafo de fronteiras).

### 5.3.2 Extensão de fronteira

O objetivo desta definição é o de interpretar a movimentação entre fronteiras (arestas) como o acordo entre todos os processadores de que uma (única) operação de memória foi executada globalmente. E, claro, esta operação deve estar na fronteira.

Uma fronteira $F$' é dita "extensão" de uma fronteira ($F$) somente se, exatamente, uma de suas sequências ($S_i$) é estendida por uma única operação, enquanto todas as outras permanecem inalteradas (GIBBONS; KORACH, 1994, p. 181).

$$F : \{f_1, \ldots, f_p\}$$

$$F' : \{f_1', \ldots, f_p'\}$$

$$\exists f_i' \in F', \quad f_i \neq f_i' \quad \wedge \quad \forall f_j \in F' - \{f_i'\},\ f_j = f_j'$$

Perceba que, assim como uma extensão (aresta) entre fronteiras define a execução global de uma operação, uma sequência de $k$ extensões (caminho) define uma sequência de execuções globais de $k$ operações. O que é equivalente a especificar uma ordenação global (i.e. total) entre estas mesmas operações.

Assim, neste momento, passa a ser necessário esclarecer a existência de duas fronteiras em especial.

**Definição 11** ***(Fronteira Inicial)****. A fronteira inicial simboliza o momento inicial de alguma execução. Logo, ela é definida por p operações nulas (*NULL*), ou seja, todo processador ainda não emitiu alguma operação de sua respectiva sequência.*

**Definição 12** ***(Fronteira Final)****. A fronteira final simboliza o momento de término de alguma execução. Logo, ela é definida por p operações de parada (*HALT*), ou seja, todas as operações de um dado processador já foram globalmente executadas.*

Com isto, uma sequência de $n$ extensões a partir da fronteira inicial necessariamente termina na fronteira final. Já que do contrário ocorre ou o

absurdo de ter aparecido uma operação adicional ou a ordenação total, adquirida por estas extensões, é inválida com relação ao modelo de memória (por definir um ciclo).

Este último caso será cuidado, no item seguinte, por meio da restrição das extensões consideradas durante um atravessamento no grafo de fronteira para que seja lidado apenas com aquelas que, justamente, existe alguma sequência de $n$ extensões válida, com relação ao modelo de memória, que as utilizem.

Portanto, as $n$ operações associadas a estas extensões definem uma ordenação total (*schedule*) para uma instância positiva do problema.

### 5.3.3 Formulação do Grafo de Fronteiras

O Grafo de Fronteiras consiste na adoção de um vértice para cada fronteira e de uma aresta para cada extensão de modo que haja a seguinte propriedade:

**Propriedade 1** *O grafo de fronteiras possui um caminho direcionado da fronteira inicial para a fronteira final se e somente se existe uma instância positiva do problema (GIBBONS; KORACH, 1994, p. 182).*

Na prática o grafo de fronteiras é atravessado de modo a passar apenas por caminhos que constituam em fragmentos de uma instância positiva do problema (solução). Isto é um resultado natural, já que não há maneira de uma ordenação inválida, definida por algum caminho, ser um fragmento de alguma ordenação total que seja uma instância positiva do problema. Afinal, ao dizer que a primeira é um "fragmento" da segunda está sendo reconhecido que todas as ordens da primeira existem na segunda. E, dizer que a primeira é inválida significa que as suas ordens são inconsistentes entre si, existem duas operações ($u$ e $v$) tais que são ordenadas de duas formas diferentes: $u$ precede $v$ e $v$ precede $u$, i.e. há algum ciclo.

Com estas ideias em mente, GIBBONS; KORACH propuseram a seguinte utilização do grafo de fronteiras:

> "Defina um *estado* associado com cada fronteira que (a) seja válido independente do caminho usado para alcançar este vértice (i.e. independente de qual *schedule* válida foi utilizada para chegar neste ponto), e (b) permita determinar em tempo constante se uma possível extensão é válida. O tempo de execução deste algoritmo é, portanto, linear no número de arestas atravessadas: o melhor caso é $O(n)$ e o pior é $O(n^p)$" (GIBBONS; KORACH, 1994, p. 182).

Os autores chegaram nestas complexidades porque, ao utilizarem a desigualdade de Jensen, concluíram que há $O((n/p)^p)$ fronteiras para uma dada instância do problema.

### 5.3.4 Como o XCHECK utiliza o conceito de Grafo de Fronteiras?

Um desafio encontrado durante esta monografia foi justamente responder a esta pergunta. Afinal, em nenhuma das suas duas publicações (HU et al., 2012; CHEN et al., 2009), os autores mostraram o algoritmo de *backtracking* utilizado.

Outro agravante é a dificuldade em validar o XCHECK, já que os seus autores realizaram experimentos com um *design* industrial e não com um *benchmark*.

Buscando investigar este assunto, primeiro veja o seguinte trecho acerca do *backtracking* do LCHECK, versão anterior do verificador em questão:

> "Para realizar uma verificação completa da consistência de memória, nós precisamos **confirmar a ordem total das escritas** para cada localização de memória a fim de conhecer todas as ordens de execução. Entretanto, algumas ordens de execução não podem ser inferidas (MANOVIT; HANGAL, 2006). Para completude, nós implementamos *backtracking* em cima da versão básica e incompleta do LCHECK. Para duas operações de escrita conflitantes sem ordem determinada, o LCHECK completo realiza uma decisão arbitrária sobre as ordens de execução destas operações, e infere o máximo possível de novas arestas, então checa por ciclos no grafo. Caso haja alguma violação, o LCHECK completo *backtracks* para a decisão arbitrária mais próxima e tenta um outro ramo de decisão. O algoritmo de *backtracking* é baseado em buscar o grafo de fronteiras (GIBBONS; KORACH, 1994)" (CHEN et al., 2009, p. 389).

Portanto especula-se que, para o XCHECK, é suficiente o grafo de fronteiras ser definido apenas para as operações de escrita, com o fim deste último orientar a decisão arbitrária acerca das ordens desconhecidas entre operações de escrita conflitantes. No entanto perceba que remover as leituras não impede que, durante um atravessamento, algumas ordens sejam reviistas. Outro detalhe é que, sem um condicionamento explícito, também estaria sendo realizado um trabalho desnecessário: a decisão de ordens entre escritas não conflitantes.

HU et al. esclarece que estará atravessando o grafo de fronteiras enquanto constrói a ordenação total referente ao caminho sendo tomado e realizado o algoritmo de checagem de ciclos. Porém, os autores acabaram deixando em aberto se o grafo de fronteiras é realmente construído somente pelas operações de escrita:

> "(...) A ideia básica por detrás do algoritmo é atravessar o grafo de fronteiras para encontrar um caminho da fronteira inicial até a fronteira final. (...) Além da adição e da remoção de ordens de execução quando se visita fronteiras diferentes (no grafo de fronteiras), nossa abordagem não precisa inferir nenhum outra ordem de execução. (...)" (HU et al., 2012, p. 508).

No fim, esta adaptação de "remover" as leituras do grafo de fronteiras se mostrou necessária, já que há casos em que as leituras são indistinguíveis entre si, mesmo quando a sua interpretação é idêntica. Por exemplo, três leituras para a mesma escrita que não tenham nenhuma ordem entre si (mesmo a de tempo físico) geram oito $\left(2^3\right)$ sequências de movimentos de fronteira, onde ocorre as mesmas novas ordens (arestas). Isto é, todas estas sequências convergem para uma única ordenação total.

Observe que este caso só ocorrerá quando utilizando modelos de memória que relaxam as ordens entre as leituras.

Outra característica sobre a utilização do grafo de fronteiras pelo XCHECK é que, como toda extensão de fronteira segue todas as ordens que já existem entre as operações, então as extensões também devem, naturalmente, seguir as ordens de tempo físico.

Como resultado, a presença da ordem de tempo físico reduz a quantidade de fronteiras utilizadas durante o seu atravessamento. De acordo com HU et al., o grafo passa a ter $O(n\,C^p)$ vértices com $O(n\,C^p\,p)$ arestas (HU et al., 2012, p. 510).

Por último, outra dúvida acerca deste tema é com relação ao XCHECK utilizar ou não o método de inferência durante a validação de um dado movimento de fronteiras. Como a utilização deste método revela, potencialmente, novas ordens entre as operações de memória, então, ela também auxilia na manutenção da Propriedade 1 durante a construção dinâmica deste grafo.

Com esta motivação em mente, em conjunto com a última parte da citação acima de HU et al., a implementação do XCHECK, realizada por esta monografia, utiliza o método de inferência (seção 5.5) para cada aresta adicionada durante o movimento de fronteiras.

## 5.4 CONSTRUÇÃO DO GRAFO DE EXECUÇÃO

O grafo de execução possui duas responsabilidades durante a verificação de memória. Primeiramente, ele contém todas as ordens conhecidas entre as operações de memória. Assim, durante o atravessamento do grafo de fronteiras, todas as novas ordens obtidas serão adicionadas nele.

Sua segunda responsabilidade está em detectar a inconsistência das ordens conhecidas por meio da presença de ciclos, fator este que é o discernimento do XCHECK para a validação de uma instância positiva do problema.

As regras de construção deste grafo (CHEN et al., 2009, p. 388), enunciadas logo abaixo, foram feitas para garantir que a última escrita que precede uma leitura (com mesmo endereço) seja justamente aquela em que esta última obteve o seu valor durante a execução de uma dada instância do problema.

1. **Regra da aresta de processador**. Adicione uma aresta entre uma operação e a sua predecessora pela ordem de processador, com relação ao modelo de memória.

2. **Regra da aresta de execução**. Caso uma operação de leitura ($r$) obtenha o seu valor de uma escrita ($w$), então adicione uma aresta de $w$ para $r$.

3. **Regra da aresta de observação**. Dada uma operação de leitura ($r$), cujo valor é obtido de uma escrita ($w$), e a última escrita ($w$') para o mesmo endereço que precede esta leitura ($r$) pela ordem de programa, então adicione uma aresta de $w$' para $w$.

4. **Regra da aresta de inferência #1**. Dada uma escrita ($w$) e uma leitura ($r$) referentes ao mesmo endereço de memória, se a escrita ($w$'), pela qual esta leitura ($r$) obteve o seu valor, está no intervalo de espera de $w$ e $w \xrightarrow{GO} r$ (inferência por Ordem global) ou $w \xrightarrow{T} r$ (inferência por Ordem de tempo físico) então adicione uma aresta de $w$ para $w$'.

5. **Regra da aresta de inferência #2**. Dada duas escritas ($w$ e $w$') que acessam o mesmo endereço de memória, se $w$' está no intervalo de espera da leitura ($r$) que obteve o seu valor pela outra escrita ($w$) e $w \xrightarrow{GO} w$' (inferência por ordem global) ou $w \xrightarrow{T} w$' (inferência por ordem de tempo físico) então adicione uma aresta de $r$ para $w$'.

O leitor deve tomar cuidado ao reutilizar o exemplo dado na seção 5.1, por meio da Figura 7, devido a uma mudança de nomenclatura para a Regra 4, onde inverteu-se os papéis de $w$ e $w$'. No exemplo da seção 5.1 a

nomenclatura é utilizada para facilitar a compreensão da atribuição de novas ordens para garantir uma execução fiel ao modelo de memória. Enquanto que, neste momento, o objetivo da nomenclatura está em facilitar o entendimento do método de inferências.

CHEN et al. argumenta que as três primeiras regras acima podem ser realizadas com complexidade linear: $O(n)$. As arestas de processador podem ser obtidas pelos *traces* em conjunto com o modelo de memória. Enquanto que para cada leitura, além do seu valor lido também ser conhecido por meio dos *traces*, sabe-se quais são os valores (únicos) que cada escrita realizou. Portanto é possível encontrar para cada leitura a escrita responsável pelo seu valor e, desta maneira, adicionar as arestas de execução e de observação.

Por último, as arestas de inferências são adicionadas por meio do método de inferência.

## 5.5 INFERÊNCIA

Em muitos verificadores de memória a inferência de arestas é a parte mais demorada de sua verificação. Entretanto, o algoritmo de inferência relatado por CHEN et al. é linear já que as arestas de inferência são limitadas pelo tamanho do intervalo de espera: se uma ordem inferida cumpre a ordem de tempo físico, então, não é necessário inferí-la de novo; e caso ela contradiga a ordem de tempo físico, então, haveria uma violação que seria capturada pelo algoritmo de checagem de ciclos (seção 5.6) (CHEN et al., 2009, p. 388).

Toda vez que uma aresta (de $u$ para $v$) é adicionada no grafo de execução, o método de inferência (Algoritmo 1) é chamado para encontrar as arestas de inferência descritas a pouco (p. 66). Devido a utilização da ordem de tempo físico, a adição de qualquer aresta afeta somente relações de ordem global que se iniciam das operações contidas no intervalo de espera de $u$. Portanto o XCHECK aplica estas regras somente para todas as escritas ($w$) contidas neste conjunto; $u$ também deverá ser considerado caso seja uma escrita (CHEN et al., 2009, p. 388).

Após a seleção de uma destas escritas ($w$), o método aplica a primeira regra de inferência para toda leitura ($r$) presente no seu intervalo de espera. Verificando, portanto, se $w$ precede $r$ pela ordem global. Uma vez que este seja o caso há três situações a serem tratadas, considerando a escrita ($w$') responsável pelo valor da leitura em questão ($r$) e relações de ordem de tempo físico: a aresta que está sendo inferida pelo método ($w \xrightarrow{E} w$') pode (a) criar algum ciclo (com $w$' $\xrightarrow{T} w$), (b) repetir alguma ordem ($w \xrightarrow{T} w$') ou, realmente, (c) se tratar de uma nova ordem. Neste último caso a adição desta nova aresta engatilha uma nova chamada ao método de inferência.

**Algorithm 1:** infer_edges(u,v) (HU et al., 2012, p. 389)

```
foreach escrita w no intervalo de espera de u do
    foreach leitura r no intervalo de espera de w do
        if address(w) = address(r) and ¬infered(w,r) and
        w --GO--> r then
            w' <-- a escrita pela qual r obtém o seu valor;
            if w' --T--> w then
                panic();
            end
            if w --T--> w' then
                continue;
            end
            if não há aresta de w para w' then
                add_edge(w,w');
                infer_edge(w,w');
            end
            set_inferred(w,r);
        end
    end
    foreach escrita w' no intervalo de espera de w do
        if address(w) = address(w') and ¬infered(w,w') and
        w --GO--> w' then
            foreach leitura r cujo valor foi obtido por w do
                if w' --T--> r then
                    panic();
                end
                if r --T--> w' then
                    continue;
                end
                if não há aresta de r para w' then
                    add_edge(r,w');
                    infer_edge(r,w');
                end
            end
            set_inferred(w,w');
        end
    end
end
```

A seguir o método aplica a segunda regra de inferência, isto é, para toda escrita ($w$') presente no intervalo de espera de uma das operações de escrita selecionadas ($w$). O processo é análogo ao caso das leituras: o método verifica se $w$ precede $w$' pela ordem global. Em caso positivo, então, há a consideração de três casos para toda leitura ($r$) que obteve o valor escrito por $w$: a aresta que está sendo inferida pelo método ($r \xrightarrow{E} w$') pode (a) criar algum ciclo (com $w$' $\xrightarrow{T} r$), (b) repetir alguma ordem ($r \xrightarrow{T} w$') ou, realmente, (c) se tratar de uma nova ordem. Caso em que a adição desta aresta engatilha uma nova inferência.

Com o fim de recordar quais arestas já foram testadas em alguma chamada anterior, o método realiza uma marcação de arestas ("inferred").

Enfim, cada chamada do método de inferência é $O(p^2)$. Logo, assim como CHEN et al. explicaram, o fato da construção do grafo de execução chamar este método $O(pn)$ vezes, significa que a sua complexidade é de $O(p^3n)$ (CHEN et al., 2009, p. 389).

## 5.6 CHECAGEM DE CICLOS

O algoritmo de checagem de ciclos objetiva confirmar que adição de novas ordens de execução (relativas à um movimento de fronteira) não constitui na criação de alguma violação ao modelo de memória ou da atomicidade das escritas (HU et al., 2012, p. 508).

O primórdio deste algoritmo encontra-se no Teorema das Regras de Checagem formulado em Chen et al. (2009, p. 386). Porém, antes de mostrar a sua formulação, é interessante apresentar a análise que estes autores realizaram sobre os ciclos presentes no grafo de execução quando considera-se a ordem de tempo físico. A base do teorema em questão.

> "Seja $\mathscr{C}_u$ o conjunto de todos os ciclos de ordem global do grafo de execução que incluem uma certa operação ($u$). Além do mais, seja $C$ um ciclo pertencente a $\mathscr{C}_u$ ($C \in \mathscr{C}_u$), sendo que $u$ é uma operação de $C$ ($u \in C$). Intuitivamente, para qualquer operação $u$ existe três tipos possíveis de ciclos que a contém: 1) todas as operações do ciclo, exceto $u$, não estão no intervalo de espera de $u$; 2) algumas das operações do ciclo não estão no intervalo de espera de $u$, enquanto as outras estão; e 3) todas as operações do ciclo estão no intervalo de espera de $u$" (CHEN et al., 2009, p. 385).

Esta divisão evidencia que o propósito de adicionar a ordem de tempo físico, na verificação de memória, está em localizar relações entre as opera-

ções. A chave desta localização está justamente em detectar o primeiro tipo de ciclo de uma maneira localizada, já que ele se refere às ordens globais fora do intervalo de espera.

Um lema apresentado por estes autores garante a inexistência deste tipo de ciclo caso toda escrita ($w$) que precede uma operação de memória qualquer ($u$) pela ordem de tempo físico, não existe alguma ordem de execução contrária. Esta é a primeira regra do Teorema das Regras de Checagem.

**Teorema 1** ***(Teorema das Regras de Checagem)****. Não existe ciclo no grafo de execução se e somente se, para qualquer operação da execução ($u$), as três seguintes regras de corretude forem válidas:*

**Regra 1:** $\forall\, w \in \mathbb{O} : (w \xrightarrow{T} u) \implies \neg(u \xrightarrow{E} w)$*;*

**Regra 2:** $\forall\, v, v' \in \mathbb{O} : (v \xrightarrow{T} u) \wedge (v' \xrightarrow{GO} v) \implies \neg(u \xrightarrow{GO} v')$*;*

**Regra 3:** $\neg(\exists\, C \in \mathscr{C} : (\forall\, v \in C : \neg(u \xrightarrow{T} v \vee v \xrightarrow{T} u)))$.

*(CHEN et al., 2009, p. 386)*

CHEN et al. verificam a Regra 1 de acordo com a propagação do valor de uma escrita ($w$') às suas leituras, considerando uma outra operação de escrita ($w$), a qual sucede a primeira por uma ordem de execução. Isto é, dada uma operação qualquer ($u$) considera-se a última escrita, pela ordem de processador, relativa ao mesmo endereço de $u$ ($w$) e a escrita responsável pelo valor de $u$ ($w$'), então, é necessário haver a ordem $w \xrightarrow{GO} w$' (aresta de observação). Significando que precisa-se confirmar que $\neg(w \xrightarrow{GO} w') \wedge \neg(w \xrightarrow{T} w')$ é valida. Caso a operação em questão ($u$) seja uma escrita, então considerar-se-á $w' = u$.

Já as Regras 2 e 3 são verificadas por meio de um atravessamento no grafo de execução por toda operação ($v$) que sucede uma certa operação ($u$) pela ordem global, enquanto valida duas condições: $\neg(v \xrightarrow{T} u)$, Regra 2, e $\neg(v = u)$,: Regra 3.

Entretanto, como definido por Hu et al. (2012), este mecanismo é redundante já que cada movimento no grafo de fronteira cria um número limitado de ordens de execução. Na verdade, toda sequência de $k$ movimentos de fronteira implica na necessidade de checar somente ciclos em potencial, os quais são relacionados com as ordens de execução adicionadas pelo último movimento de fronteira. Caso existissem outros ciclos, estes teriam sido detectados por meio do seu prefixo: a sequência de $k-1$ movimentos de fronteira.

Com este conceito em mente, HU et al. fazem uma outra análise dos ciclos. Desta vez, considerando a operação que passou a ser globalmente executada: a única que estava na antiga fronteira mas não está na nova ($u_i$).

Observe que esta operação ($u_i$) é também a **última** operação que foi globalmente executada. Portanto, em resposta a um movimento de fronteira, cria-se ordens de execução entre as operações que já foram globalmente executadas durante um movimento anterior e $u_i$. Por estar sendo tratado da ordem de execução, lida-se apenas com as operações conflitantes com $u_i$ (HU et al., 2012, p. 509).

Assim, os autores dividem estas novas ordens de execução em três casos:

1. A nova ordem de execução repete alguma ordem de tempo físico;
2. A nova ordem de execução contradiz alguma ordem de tempo físico;
3. Não existe alguma ordem de tempo físico entre $u_i$ e $v$.

Os dois primeiros casos acima podem ser tratados diretamente pelo verificador. Afinal, enquanto o primeiro trata de ordens redundantes, o segundo é uma violação ($u_i \xrightarrow{T} v \wedge v \xrightarrow{E} u_i$) que pode ser checada em tempo constante.

Para encontrar as violações do segundo caso, só é necessário verificar se existe uma ordem de tempo físico entre $u_i$ e $v_1$ ($u_i \xrightarrow{T} v_1$), onde $v_1$ é a operação com o maior tempo de emissão dentre todas as operações conflitantes de $u_i$ que já executaram globalmente. A inexistência desta ordem implica que haveria uma contradição caso alguma outra operação conflitante, já executada globalmente, sucedesse $u_i$ por uma ordem de tempo físico. Por fim, como uma violação foi encontrada, é necessário fazer o *backtracking* pelo grafo de fronteiras.

O terceiro caso apresenta o tratamento mais pesado para realizar esta checagem, sendo necessário adotar a mesma medida do LCHECK. Porém, em vez de realizá-la para todas as operações de memória, será feita (no máximo) apenas com aquelas operações que não têm alguma ordem de tempo físico com $u_i$, ou seja, com $O(p)$ operações.

## 5.7 VISÃO GERAL SOBRE OS DETALHES DE IMPLEMENTAÇÃO

Para os fins de experimentação desejados, este autor desenvolveu [1] em conjunto com Olav Phillip Henshcel uma versão do *XCHECK* para o modelo

[1] *Boost* (http://www.boost.org/) oferece diversas bibliotecas cujo código livre foi revisado por pares (*peer review*), é portado para C++ e suporta quase todo sistema operacional (incluindo as

de memória implementado pelo *gem5* (uma variação do adotado pela arquitetura *Alpha*), que é o simulador de representações executáveis de *hardware* utilizado em nossa plataforma de testes. Esta última será descrita no próximo capítulo.

Como o leitor verá daqui a pouco, nossa decisão de implementar uma versão do XCHECK para um modelo tão relaxado, quanto este, aumentou sua complexidade de verificação para $O\left(n\, C^{pa}\, p^3\, a\right)$, onde $a$ é a quantidade máxima de endereços utilizados por cada processador, mais especificamente, pelas operações que emitiu. Caso este número adicional ($a$) seja uma constante, então tal aumento de complexidade é omitido pela análise assintótica: volta-se a ter $O\left(n\, C^p p^3\right)$.

Nos experimentos aqui realizados (Capítulo 6), $a$ é o número de endereços compartilhados. Afinal, os casos de testes utilizados possuem somente operações de memória que trabalham exclusivamente nestes endereços. Uma característica decorrente do fato de que operações para endereços privados exercitam menos erros de projeto do que as operações para endereços compartilhados. Veja que quanto menor for este número ($a$), maior é o compartilhamento dos dados (HANGAL et al., 2004 apud HENSCHEL, 2014, p. 45), o qual, por sua vez, induz a sobrecarga dos sistemas de memória (DEORIO; WAGNER; BERTACCO, 2009 apud HENSCHEL, 2014, p. 45). Isto é, possibilita reduzir o tamanho dos casos de teste, uma característica fundamental para a etapa pré-silício. Com este efeito em mente, os nossos experimentos variaram o número de endereços compartilhados (i.e. $a$) com o fim de avaliar a sensibilidade dos verificadores. Razão pela qual, estes mesmos experimentos comprovaram o comportamento exponencial desta nossa versão do XCHECK, com relação à quantidade de endereços compartilhados ($a$).

Observe que o nosso interesse em variar o número de endereços compartilhados se deve, também, por que existem características do sistema de memória que só são testadas com um número mínimo destes endereços, e.g. quando uma palavra afeta outra no mesmo bloco. Esta característica também pôde ser observada durante os nossos experimentos (Seção 6.3).

Portanto, tendo em vista que os experimentos desejados capturariam este aumento de complexidade, a opção de realizá-los com uma versão simplificada do XCHECK tornou-se interessante. Tal versão não atravessa o grafo de fronteiras, isto é, pára de executar assim que entra em sua região de complexidade exponencial (devido ao *backtrack*).

---

variantes de UNIX e de WINDOWS). Nesta monografia foi utilizada uma parte destas bibliotecas para facilitar a implementação do grafo de execução e de suas consultas. Por exemplo, o algoritmo de checagem de ciclos de HU et al. realiza uma busca em profundidade que é interrompida (pela emissão de uma exceção) sempre que é visitada uma operação que precede, pela ordem de tempo físico, a operação que iniciou a busca. Outra condição para interrompê-la é quando a busca evidência um ciclo, ou seja, uma inconsistência com o modelo (seção 5.6).

Com o fim de reduzir o tempo da realização dos experimentos, foi aproveitada uma executação da versão completa para gerar os tempos de verificação destas duas versões. O tempo da versão incompleta é registrado um pouco antes da fase de atravessamento no grafo de fronteiras se iniciar.

### 5.7.1 Expansão do Grafo de Fronteiras

HU et al. afirmaram que o XCHECK pode verificar o modelo sequencial, o modelo do *Godson-3B* (uma CPU de propósito geral desenvolvida na China, pelo *Institute of Computing Technology* (ICT) e pela *Chinese Academy of Sciences* (CAS)), e entre outros modelos (HU et al., 2012, p. 503).

No entanto, no decorrer da elaboração deste trabalho foi percebida uma dependência com o modelo de memória na afirmação de GIBBONS; KORACH sobre cada fronteira definir uma sequência de prefixos (GIBBONS; KORACH, 1994, p. 181). O que está diretamente relacionado com o tamanho do grafo de fronteiras.

O principal argumento de GIBBONS; KORACH ocorre quando concluem que o grafo de fronteiras possui "um caminho direto entre a fronteira inicial até a fronteira final se e somente se temos uma instância positiva do problema" (GIBBONS; KORACH, 1994, p. 182). Portanto, atravessar o grafo de fronteiras equivale a construir (somente) cada uma de suas $O((n/p)^p)$ ordenações totais (Seção 5.3).

Agora, como um modelo altamente relaxado (e.g. o modelo de interesse, alpha) possui uma sequência de operações para cada endereço de memória de cada processador, então as fronteiras devem ser redefinidas para: $\{f_{1,1}, \ldots, f_{1,a}, \ldots, f_{p,1}, \ldots, f_{p,a}\}$. Desta maneira, existem $O((n/(pa))^{pa})$ ordenações totais para estes modelos de memória.

Por outro lado, o cálculo de HU et al. acerca da quantidade de fronteiras (HU et al., 2012, p. 509-510) é dependente apenas do número de operações no intervalo de espera de uma dada operação ($u_i$). Afinal, este autor está interessado em calcular somente o número de fronteiras cujo movimento a elas gera novas ordens de execução.

Considerando que haja uma constante ($D$) para o número de operações no intervalo de tempo de $u_i$ para cada endereço compartilhado (e.g. $D = C/a$, considerando uma distribuição uniforme), então, o número de fronteiras passa a ser de $O(n\, D^{pa})$. Como existe $Cp$ operações de memória que possam estender uma fronteira envolvendo $u_i$, o número total de arestas do grafo de fronteiras passa a ser $O(n\, D^{pa}\, C\, p)$ ou, somente, $O(n\, D^{pa} p)$.

Durante cada movimento de fronteira (aresta), a checagem de ciclos é realizada em $O(p)$, considerando cada uma das $O(pa)$ ordens adicionadas: a

última operação globalmente executada em cada uma das $O(pa)$ sequências precede a operação que está sendo globalmente executada no movimento atual. Portanto, a complexidade de verificação do XCHECK é de $O\left(n\, D^{pa} p^3 a\right)$, para um modelo altamente relaxado.

### 5.7.2 Decisão sobre inferência

Como já foi dito anteriormente, nas duas publicações relativas ao XCHECK (CHEN et al., 2009; HU et al., 2012), os autores não esclareceram se as arestas adicionadas por um movimento de fronteira devem (ou não) ativar o método de inferências. A decisão por realizar as inferências significa o aumento do número de ordens adicionadas (e, portanto, removidas durante o *backtracking*) para cada movimento de fronteira.

Durante a realização da implementação houve um grande interesse em reduzir, o máximo possível, a quantidade de caminhos inválidos atravessados durante a utilização do grafo de fronteiras, isto é, caminhos que não levam a uma instância positiva do problema. E toda ordem inferida pode, potencialmente, antecipar o verificador no reconhecimento do caminho atual como inválido. Além do mais, como havia sido analisado que esta adição do método de inferências não afetaria a complexidade de verificação, decidiu-se por utilizá-la.

Entretanto, no término deste trabalho foi percebido o erro desta pressuposição; em utilizar o método de inferência durante o atravessamento do grafo de fronteiras. Há uma dificuldade em saber exatamente como esta sua utilização afeta a complexidade de verificação. Ainda assim, como ela significa que o método de inferência é chamado para cada uma das $O(pa)$ arestas adicionadas durante um movimento de fronteira, a complexidade é aumentada, no mínimo, em $O\left(n\, D^{pa}\, p^4\, a\right)$. E, em nenhum momento de (HU et al., 2012), os autores mencionam este aumento.

### 5.7.3 Implementação do Grafo de Fronteiras

A implementação do grafo de fronteiras constitui em omitir as operações de leitura e as barreiras de memória de suas fronteiras. Esta omissão foi motivada por ser desnecessário (e computacionalmente intensivo) ordenar operações não conflitantes, como as combinações entre operações de leitura. Já as barreiras de memória foram omitidas devido à sua função ser, apenas, de estabelecer ordens no grafo de execução.

Para suportar a omissão das leituras, realizaram-se duas adaptações.

Primeiramente, para todo movimento de fronteira, onde ocorre a saída ($w_1$) e entrada ($w_2$) de operações de escrita (de mesmo processador), é realizada a adição de arestas a partir da última escrita globalmente executada, durante movimentos anteriores, com a escrita $w_1$ e todas as suas respectivas leituras.

A sua segunda adaptação consistiu em adicionar um contador para as operações predecessoras que não foram globalmente executadas. Assim, quando uma operação é executada (i.e. retirada da fronteira), todas as escritas que são suas sucessoras diretas terão o seu contador de predecessores decrementado. Portanto, a execução de uma operação é condicionada a ter o seu respectivo contador zerado. Complementarmente, quando o *backtracking* é realizado, estes mesmos contadores são restaurados (incrementados).

Outro detalhe sobre a implementação realizada é que modelar diretamente a definição de fronteiras de GIBBONS; KORACH resulta na possibilidade de que dois caminhos distintos (entre a fronteira inicial e a final) produzam a mesma ordenação total. Para cuidar deste caso, os movimentos de fronteira foram condicionados a estarem produzindo uma nova ordenação total. Cada fronteira é associada com um conjunto de controle, em que cada elemento sintetiza algum caminho (até ela própria) produzido em alguma passagem anterior para esta fronteira. Isto é, cada um destes caminhos já se mostraram incapazes de criar uma ordenação total válida, senão já haveria sido comprovado que a instância é positiva. Assim, toda vez que uma fronteira é visitada por algum movimento, verifica-se que o caminho em questão não está (sintetizado) neste conjunto de controle.

E, a maneira de sintetizar um caminho é, justamente, realizar a extensão do conceito de fronteiras anteriormente mencionada. O que significa que, no fim, esta nossa implementação poderia ser simplificada através da modelação direta de tal extensão.

### 5.7.4 Refinamento dos *traces* e o pré-processamento

A formulação dos *traces*, dada na capítulo 3, foi levemente modificada para facilitar a implementação do verificador de modo que só precise processar as informações de cada operação, as quais são: o índice do processador que a emitiu; o tipo (leitura, escrita ou barreira de memória); o endereço de memória; o dado lido/escrito, o identificador único (para fins de *debug*) e o intervalo de tempo. Pela formulação deste capítulo, os tempos de início e de término de uma dada operação estariam divididos em dois eventos distintos do *trace*.

Outra diferença relativa aos *traces* afetou o cálculo dos intervalos de espera. A janela de instruções do *gem5* é tão grande, com relação à quantidade

de operações pretendida para os experimentos, que os intervalos de tempo seriam muito grandes caso os métodos de *sampling* descritos por HU et al. (seção 5.2.3) fossem aplicados. Além do mais, como este verificador está sendo adaptado para a etapa pré-silício existe uma maneira mais direta (e refinada) de obtê-los: reutilizando o tempo do próprio simulador. No entanto, tal refinamento ocasionou intervalos pequenos o suficiente para capturar os reordenamentos permitidos pelo modelo de memória. Ou seja, eliminou-se a possibilidade de ordenar as operações tanto pelo crescimento no tempo de início quanto no tempo de término.

No fim, este refinamento significou o aumento da complexidade de cálculo dos intervalos de espera: a única alternativa restante foi de checar todas as operações de memória. E, como não é de nosso interesse adicionar ao XCHECK uma complexidade $O(n^2)$, a qual deteriora a sua principal vantagem, então, foi adicionado um pré-processamento cujo tempo de execução não é utilizado no cálculo da métrica de eficiência do verificador: o seu tempo de verificação (Capítulo 6).

Neste pré-processamento, o intervalo de espera de todas as operações é calculado e registrado em um arquivo, assim a inicialização do intervalo de espera pôde ser realizada em $O(np)$, como previsto em Hu et al. (2012).

Além disso, a implementação tirou proveito do fato dos intervalos de espera serem utilizados em apenas dois locais em toda a verificação, ambos no método de inferências de arestas (seção 5.5). A primeira consulta a estes conjuntos deseja somente as operações de escrita, e a segunda, apenas aquelas que tenham ordem global ($w \xrightarrow{GO} r \ \vee \ w \xrightarrow{GO} w$'). Assim, o pré-processamento computa o intervalo de espera restringindo-o às operações de escrita. Já a segunda consulta pode ser realizada adaptando-se o atravessamento feito pela técnica de checagem de ciclos para registrar todas as operações visitadas.

Assim, aproveitou-se para utilizar o pré-processamento para também computar o subconjunto das ordens de tempo físico que não são transitivas. Desta maneira a inicialização do verificador pode adicionar estas arestas no grafo de execução, inferindo ainda mais ordens e diminuindo os movimentos de fronteira que gerariam ordenações inválidas. Quanto mais ordens restringem explicitamente a execução global de uma operação de fronteira, mais cedo os caminhos errados são reconhecidos.

Para mensurar o pré-processamento deste subconjunto, considere um conjunto definido para uma operação qualquer ($v$) que possui as operações ($u$) que precedem $v$ diretamente pela ordem de tempo físico:

$$u \xrightarrow{T} v \ \wedge \ \nexists w \in \mathbb{O} : \ (u \xrightarrow{T} w \ \wedge \ w \xrightarrow{T} v).$$

De $u \xrightarrow{T} w$ conclui-se que todas as ordens de interesse (diretas) são relativas às operações com um tempo de término maior do que as ordens que desejamos evitar (transtivas): $\tau_e(u) < \tau_s(w) <= \tau_e(w)$. Assim, caso o intervalo de espera de $v$ fosse ordenado pelo tempo de término crescente resultar-se-ia em duas regiões: um prefixo com todas as ordens transitivas e um sufixo com todas as ordens desejadas (diretas). Esta construção evidencia que o número de ordens diretas (com relação a $v$) é limitado por $O(p)$. Afinal toda operação deste sufixo ($u_1$) está no intervalo de espera da operação do conjunto com o maior tempo de término ($u_2$). Caso contrário, teria-se $u_1 \xrightarrow{T} u_2$, ou seja, $u_1$ deveria pertencer ao prefixo (contradição).

Por fim, o pré-processamento foi realizado em $O(2 \times n \log n + 2 \times np)$ ou, simplesmente, $O(np)$.

### 5.7.5 Processamento das operações especiais

Como dito na seção 5.3.1, a utilização do grafo de fronteiras necessita da definição de duas operações especiais: *null* e *halt*.

As operações nulas (*null*) são processadas com a semântica de uma barreira de memória, sendo predecessoras de todas as operações do mesmo processador (i.e. programa). Além disso, como possuem ordem de tempo físico com todas as outras operações, garante-se que são as primeiras a executar. No entanto, foi necessário adicionar ordens entre operações nulas para impedir que o verificador tente reordená-las, o que seria um custo adicional computacionalmente intensivo (exponencial).

As operações de parada (*halt*) têm como objetivo permitir que toda operação de todo processador seja retirada da fronteira, em algum momento. Desta forma, não é necessário haver ordens adicionais entre estas operações, como houve com as operações nulas. As operações de parada são processadas com a semântica de barreira de memória, isto é, quaisquer operações de memória as precedem pela ordem de tempo físico.

# 6 INFRAESTRUTURA EXPERIMENTAL E RESULTADOS OBTIDOS

Antes de mais nada, o leitor deve ser esclarecido que os experimentos desta monografia foram realizados no ECL em conjunto com Olav Philipp Henshcel. De modo que esta monografia apresenta uma parte dos experimentos realizados e apresentados por completo na dissertação do referido colega (HENSCHEL, 2014). Além do mais, HENSCHEL já justificou que os parâmetros escolhidos propriciam um comparativo realista sobre arquiteturas atuais e futuras.

Portanto, este capítulo esclarece a plataforma de testes e analisa os resultados obtidos pelos experimentos realizados, com o fim de comparar os verificadores em questão (HU et al., 2012; FREITAS; RAMBO; SANTOS, 2013) sob quatro pontos de vista distintos, sendo um para a sensibilidade da eficácia ao tipo de erro de projeto (a.k.a. plataforma) e três para o impacto na eficiência e na eficácia considerando o crescimento: do número de operações do caso de teste ($n/p$), do número de núcleos de processamento e do número de endereços compartilhados.

## 6.1 A PLATAFORMA DE TESTES

Nos últimos cinco (5) anos, mestrandos do curso de Ciências da Computação, ligados ao laboratório ECL da UFSC, realizaram pesquisas sobre o desenvolvimento de verificadores dinâmicos específicos para a fase pré-silício do desenvolvimento de um sistema de *hardware*.

Como resultado foram desenvolvidas duas técnicas: o *E-Matching* (RAMBO; HENSCHEL; SANTOS, 2012) e o *MSB* (FREITAS; RAMBO; SANTOS, 2013). Atualmente, o mestre Olav Philipp Henschel está desenvolvendo junto ao Prof. Dr. Luiz Cláudio Villar dos Santos uma atualização desta última técnica capaz de lidar com *Weak Write Order* (WWA).

Durante a elaboração destas pesquisas também foi criada uma plataforma de testes a fim de possibilitar a comparação experimental das técnicas desenvolvidas com implementações locais de técnicas do estado da arte. “Estes experimentos caracterizam cenários realistas de projeto, levando em conta as arquiteturas atuais e sua evolução esperada nos últimos anos” (HENSCHEL, 2014, p. 43).

O mecanismo de comparação adotado para verificadores de memória se constitui em avaliá-los com relação a duas caraceterísticas que lhes são fundamentais. Primeiramente, um verificador ideal detecta qualquer erro de

projeto que o DUV possa ter. Assim, a sua medida de eficácia é dada pela percentagem dos experimentos em que um dado verificador diagnostica um erro de projeto. Repare que, como os dois verificadores a serem comparados são completos (não diagnosticam falso positivo ou falso negativo), podemos comparar esta medida diretamente: o verificador mais eficaz é aquele que possui a maior percentagem de erros detectados nos experimentos realizados. A segunda característica é a velocidade de verificação (eficiência), cuja medida significa a soma entre o tempo de simulação e o tempo de processamento da própria técnica.

Agora, por quê a eficácia está sendo medida pela cobertura do verificador? A eficácia de um verificador refere-se, antes de mais nada, à sua capacidade em detectar os erros de projeto contidos no DUV, isto é, está relacionada com as garantias de verificação. Porém, como os verificadores em questão são dinâmicos, a percepção da eficácia de um destes verificadores é dependente dos casos de testes (programas) escolhidos para serem submetidos à verificação. Naturalmente, qualquer verificador dinâmico é incapaz de perceber os erros de projeto caso só sejam utilizados casos de teste que exercitam nenhum destes erros.

Uma maneira de trabalhar esta questão é criar casos de testes com o intuito de conterem as características necessárias para exercitar algum erro de projeto em específico. No entanto, a infraestrutura deste trabalho lida com casos de testes aleatórios. Portanto, é possível que, mesmo que seja conhecido o erro de projeto contido no DUV (durante um certo experimento), o verificador utilize um caso de teste que não o exercite. Assim, a cobertura é uma medida interessante para a eficácia, já que ela revela não apenas a capacidade do verificador em detectar algum erro de projeto (utilizando casos de testes aleatórios), como, também, a sua "facilidade".

Enfim, como o leitor pode imaginar, a experimentação feita sobre esta plataforma é orientada a casos de teste sendo, naturalmente, dividida em quatro (4) etapas: a compilação do *gem5*, a geração dos casos de teste, a realização propriamente dita dos experimentos e a análise dos resultados obtidos.

Antes de mais nada, será apresentada uma visão geral do *gem5* para entender a motivação desta primeira etapa. Em seguida, serão descritas cada uma das outras etapas de experimentação.

### 6.1.1 Visão geral acerca do *gem5*

O simulador *open-source gem5* (`www.gem5.org/`) é uma plataforma modular para a pesquisa de arquiteturas de sistemas computacionais, abran-

gendo tanto arquiteturas de alto nível quanto microarquiteturas de processadores. Além disto, oferece flexibilidade para escolher características da arquitetura que será simulada como, por exemplo, os modelos da CPU, da execução do sistema e do próprio sistema de memória.

A maioria dos componentes deste sistema foram escritos em C++. Porém, para atender ao seu interesse em modulariedade, estes códigos são portados para Python para permitir a instanciação dinâmica dos seus módulos (com a parametrização das características a serem simuladas) através do interpretador Python. Portanto, é possível criar e executar sistemas com configurações diferentes, sem precisar recompilar todo o código.

Uma outra parte dos arquivos deste simulador estão escritos em SLICC e Ruby. Este primeiro é uma linguagem utilizada para especificar os protocolos de coerência de cache. O compilador SLICC gera código C++ para cada controlador envolvido nos protocolos. Estes podem trabalhar em conjunto com outras partes do Ruby (`www.m5sim.org/SLICC`).

Caso o leitor tenha interesse, o site oficial do *gem5* disponibiliza uma descrição geral da organização do seu código-fonte (`www.gem5.org/Source_Code`), assim como uma versão *HTML (HyperText Markup Language)* da sua documentação (`www.gem5.org/docs/html/index.html`).

Enfim, o *gem5* é uma ótima infraestrutura para os experimentos aqui pretendidos e, por maior que seja, há diversas fontes que esmiuçam os seus detalhes (e.g. Binkert et al. (2011)). Ainda assim, ele deixa a desejar em dois pontos com relação aos experimentos desejados: (a) ele não produz os *traces* necessitados pelos verificadores e (b) não há a possibilidade de simular sistemas com algum erro de projeto (ou melhor, espera-se tal característica). Por esta razão, além da produção de *scripts* objetivando automatizar os testes, também alterou-se o *gem5* para suportar estes dois pontos.

Para atender à produção dos *traces*, foram criadas classes denominadas monitores, cujos métodos de observação são chamados em locais estratégicos do código-fonte do simulador. Os monitores dos verificadores *post-mortem* gravam os *traces* em arquivos alvos, enquanto os verificadores *on-the-fly* são repassados diretamente com as informações dos seus monitores.

Cada erro de projeto (Tabela 3) foi modelado por meio da inserção de códigos que realizam o seu comportamento característico. Com o detalhe de que a execução de cada trecho é condicionada a um parâmetro específico repassado para a interface do simulador. Este parâmetro referencia o (único) erro de projeto cuja simulação é desejada para um dado experimento. Há a opção de simular a versão original do *gem5* - livre de algum erro. Afinal, os experimentos são pretendidos para avaliar a verificação durante a fase pré-silício, mesmo que também estejam sendo avaliados verificadores desenvolvidos para pós-silício.

| ID | Descrição | Localização |
|---|---|---|
| *e1* | Violação da barreira de memória para leituras | Unidade de controle de execução |
| *e2* | L2 não escreve dado bloco da *cache* quando recebe uma resposta da L1 | Controlador da *cache* L2 |
| *e3* | L1 sempre vai ao estado exclusivo quando recebe dado para leitura | Controlador da *cache* L1 |
| *e4* | L2 não envia mensagem de invalidação para L1 quando o seu estado é Compartilhado ou Exclusivo | Controlador da *cache* L2 |
| *e5* | Escrita inacabada não é vista por leitura para o mesmo endereço | Mecanismo de adiantamento |
| *e6* | Leitura não obtém o valor da escrita consolidada quando ela está na última posição da fila de escritas | Mecanismo de adiantamento |
| *e7* | Quando uma barreira de memória completa, outras barreiras de memória em execução são ignoradas pelas leituras | Unidade de controle de execução |
| *e8* | L1 vai para o estado compartilhado quando recebe dado antigo para leitura, ao invés de permanecer Inválido | Controlador da *cache* L1 |

Tabela 3 – Caracterização dos erros de projeto inseridos, propositalmente, durante os experimentos (HENSCHEL, 2014, p. 47)

Outros erros de projeto também foram modelados, além dos reportados na Tabela 3. No entanto, estes não foram inclusos na comparação aqui relatada devido a facilidade em serem detectados ou a nunca serem exercitados para a configuração pretendida nos experimentos.

Uma ressalva ao leitor é que este trabalho objetiva capturar, por meio dos experimentos, a capacidade de cada verificador em detectar um certo erro de projeto presente na literatura. Afinal, este dado é um avaliador da eficácia dos verificadores de memória. Portanto, não há o interesse em realizar alguma simulação sujeita a mais de um erro de projeto.

### 6.1.2 Etapa 1: Compilação dos módulos do *gem5*

A primeira atitude a ser tomada para utilizar o *gem5* constitui em preparar o código-fonte para ser interpretado em Python (e.g. os arquivos escritos em SLICC e C++). Esta preparação automática, controlada por um

construtor *open-source* implementado em Python (o *SCons*), é sujeita à escolha tomada entre as cinco versões de configuração disponíveis. Os detalhes sobre os parâmetros desta construção, assim como estas cinco versões, estão disponíveis em `www.gem5.org/Build_System`.

Para os experimentos realizados nesta monografia, adotamos a versão **gem5.opt**, a qual ativa todas as otimizações do *hardware* simulado e desativa quase todas as funcionalidades de *debug*, deixando apenas aquelas similares aos *asserts* e *printfs*. Estas duas características fazem desta versão a que oferece um balanceamento entre velocidade de simulação e *insight* do que está acontecendo.

Antes de termos utilizado esta versão, realizamos testes piloto com uma versão sem otimizações: a **gem5.debug**.

É verdade que os erros de projeto quebram a lógica de "comportamento correto". No entanto, a presença de tais erros (corretamente modelados) não significa a ocorrência de algum erro de simulação. Por esta razão, a versão **gem5.fast**, que desativa todas as funcionalidades de *debug*, é uma outra opção válida para ser utilizada nos nossos experimentos. O único motivo pelo qual não a utilizamos foi para ter absoluta certeza de que nenhum experimento seria diagnosticado como tendo algum erro de projeto quando, na realidade, isto decorreria de um erro na própria simulação.

Note que nossa decisão faz com que o tempo de simulação dos experimentos contenha também o tempo gasto no *debug* e, portanto, interfere levemente na comparação de eficiência dos verificadores.,

### 6.1.3 Etapa 2: Geração dos casos de teste

A infraestrutura do laboratório disponibiliza um gerador de casos de teste (RAMBO; HENSCHEL; SANTOS, 2011), o qual vem sendo aprimorado no decorrer da pesquisa sobre a verificação de memória.

De acordo com o escopo dos experimentos, um caso de teste é um conjunto de programas paralelos (*threads*) que serão executados em cada um dos núcleos de processamento da representação executável do sistema simulado. Logo, a fim de gerar automaticamente um caso de teste aleatório, o gerador utiliza cinco (5) parâmetros: a semente do gerador de números randômicos, a probabilidade de ocorrência para cada instrução de memória, o número de instruções por núcleo, o número de núcleos de processamento e o número de endereços compartilhados. Nos experimentos, há três tipos de instruções de interesse: leitura (*load*), escrita (*write*) e a restauradora da ordem de programa (barreira de memória).

Uma chamada do gerador de casos de teste, escrito em Python, gera

um programa C capaz de criar, como processos filhos, todas as *threads* desejadas. Os endereços compartilhados são modelados como inteiros voláteis sem sinal, sendo declarados sequencialmente na parte inicial do código deste programa gerado (no escopo global).

Tal programa principal possui características sutis para suportar os experimentos. Por exemplo, a inicialização das *threads*, ocorrida durante a execução deste programa, gera instruções de memória que devem ser ignoradas nos experimentos. Para tratar deste caso, o programa principal insere uma barreira de memória logo após esta inicialização. Desta maneira, os experimentos podem ser automatizados corretamente desde que haja o cuidado de lembrar a existência de uma fase de inicialização, cuja transição é sinalizada por uma barreira de memória. HENSCHEL esclarece em sua dissertação este e outros detalhes referentes aos cuidados realizados para este gerador (HENSCHEL, 2014, p. 43-44).

A realização de casos de teste aleatórios (sem *constraints*) é um efeito direto da metodologia escolhida. Entretanto, a fim de permitir que os experimentos pudessem ser repetidos automaticamente em outro computador, é necessário controlar a "aleatoriedade" do gerador. É por esta razão que o gerador aceita como parâmetro a semente aleatória (random seed) que utiliza para definir as instruções de memória de cada *thread*.

Na realidade, esta semente é utilizada para inicializar um gerador interno de números aleatórios. E são esses números os responsáveis pela decisão entre qual tipo de operação de memória será utilizada em uma posição específica de uma *thread*. As leituras são representadas pelos $L$ primeiros números, as escritas pelos $S$ seguintes, enquanto que as barreiras de memória ficam com os $M$ últimos. Os números $L$, $S$ e $M$ são parâmetros do gerador e são interpretados como a proporção de cada tipo de operação para o programa que está sendo gerado.

### 6.1.4 Etapa 3: Experimentos

Após as etapas anteriores serem completadas, todas as dependências para a realização dos experimentos terão sido satisfeitas. Por isto, cabe a esta etapa realizar os experimentos e, principalmente, coletar os seus resultados.

Os experimentos são realizados sobre representações de projeto (denominadas por plataformas) executadas através do simulador *gem5* utilizando os casos de teste gerados aleatoriamente na etapa anterior. Os parâmetros arquiteturais são escolhidos (e fixados) de maneira realista e compatível com sistemas embarcados, sendo injetado no máximo um único erro de projeto por experimento (daqueles listados na Tabela 3).

Assim, um experimento é caracterizado por oito (8) informações, isto é, uma tupla $(R, L, S, N, P, A, error, checker)$. As seis primeiras informações se referem ao caso de teste e são os parâmetros utilizados para a sua geração: a semente randômica ($R$), a proporção de leituras ($L$), de escritas ($S$), de barreiras de memória ($1 - L - S$), a quantidade de operações de memória por núcleo ($N$), a quantidade de núcleos de processamento ($P$) e a quantidade de endereços compartilhados ($A$). A sétima informação é o erro de projeto injetado no *gem5*, o qual caracteriza a plataforma que estará sendo testada pelo verificador, especificado pelo último elemento da tupla (*checker*).

Com este formato em mente, criou-se um *script*, em *Python*, que realiza diversos experimentos a partir da combinação das configurações desejadas: os parâmetros dos casos de testes, as plataformas (i.e. erros de projeto) e os verificadores. Este *script*, compila o *gem5*, chama o gerador de casos de teste e realiza os experimentos e o pós-processamento dos resultados, i.e. computa as medidas de eficiência e eficácia e as compila em uma tabela do *Excel*.

Outra característica presente neste *script* é o envio dos resultados compilados para um servidor (fixo), permitindo, assim, que possamos dividir os experimentos em diversos computadores: cinco (5) no total. Uma atitude pertinente, principalmente, em vista do grande tempo de verificação do *XCHECK*.

### 6.1.5 Etapa 4: Síntese

A etapa anterior nos oferece uma tabela do *Excel* que sintetiza os resultados dos experimentos. Cada linha se refere a um experimento, cujos atributos caracterizadores definem as primeiras colunas. Enquanto isso, há colunas acerca das informações de cada verificador (tempo de verificação e diagnóstico) e da própria simulação (tempo de geração dos *traces* e o diagnóstico sobre a presença de erro de simulação). Portanto, é necessário refinar estes dados tanto em valores métricos adequados quanto em gráficos.

Uma abstração dos experimentos e seus respectivos resultados nos permitiu criar uma planilha do *Excel* bastante flexível, ao ponto de automatizar boa parte desta síntese. Isto é, na prática, necessitamos apenas copiar os dados gerados pelo *script* para uma guia reservada desta planilha. A única exceção é em relação aos gráficos de eficiência, cujas colunas devem ser agrupadas e empilhadas. Cada coluna diz respeito a um dos verificadores e, portanto, constitui no empilhamento do tempo de verificação sobre o tempo de simulação do verificador. A maneira como conseguimos criar estes gráficos foi através da elaboração manual de uma tabela auxiliar.

## 6.2 CONFIGURAÇÃO EXPERIMENTAL

Os experimentos foram realizados sobre 1200 casos de testes, gerados a partir da combinação dos seguintes valores para cada parâmetro do gerador aleatório: duas sementes aleatórias (10 e 11), quatro conjuntos de proporções de leituras, escritas e barreiras de memória (Tabela 4), cinco quantidades para as instruções por núcleo (125, 250, 500, 1000, 2000), cinco quantidades para os núcleos de processamento (2, 4, 8, 16, 32) e seis quantidades para os endereços compartilhados (1, 2, 4, 8, 16, 32).

| Conjunto nº | Leituras | Escritas | Barreiras de memória |
|---|---|---|---|
| 1 | 30% | 66% | 4% |
| 2 | 48% | 48% | 4% |
| 3 | 66% | 30% | 4% |
| 4 | 80% | 16% | 4% |

Tabela 4 – Proporções para cada tipo de operação (HENSCHEL, 2014)

O único parâmetro do *gem5* que variou durante os experimentos foi o responsável pela injeção dos erros de projeto. As outras características constituíram na adoção do ISA que apresenta o melhor suporte (i.e. *UltraSPARC Architecture 2005* (Sun Microsystems Inc., 2008)), segundo Binkert et al. (2011), e uma hierarquia de memória *cache* estritamente inclusiva, utilizando três níveis com blocos de 64 *bytes*. O primeiro nível é privativo e dividido em *cache* de instruções e *cache* de dados (ambas 4kiB), o segundo (64KiB) é privativo e unificado, e o terceiro nível (4MiB) é compartilhado entre os processadores (HENSCHEL, 2014, p. 46). A interconexão usada possui uma topologia em estrela baseada em um comutador (*switch*) com roteamento simples (HENSCHEL, 2014, p. 48).

Já o mecanismo de coerência, contido no terceiro nível da memória, adota um protocolo *MESI* baseado em diretórios e segue o modelo de memória implementado pelo *gem5* (uma variação da arquitetura *Alpha*). Este modelo assemelha-se a diversos modelos populares, como *Weak Ordering* (WO) (DUBOIS; SCHEURICH; BRIGGS, 1986) cujas variações são utilizadas em processadores atuais com arquiteturas como *ARMv7* (ARM, 2012) e *ARMv8* (ARM, 2013) (HENSCHEL, 2014, p. 46).

Ao observarmos a grande complexidade de verificação do *XCHECK*, decidimos por também realizar experimentos com uma versão simplificada deste verificador, a qual não utiliza a técnica de *backtracking*. Além disso, instituímos uma duração máxima (2 horas) para a verificação de cada experimento, a fim de viabilizar a realização de todos estes experimentos dentro do

prazo.

Os gráficos apresentados na próxima seção utilizam um apelido para cada uma das técnicas avaliadas: *Multi-ScoreBoard* (MSB) se refere à técnica de (FREITAS; RAMBO; SANTOS, 2013), *Inference with BackTracking* (IBT) à técnica de (HU et al., 2012) e *Inference Best Effort* (IBE) à versão simplificada desta última técnica.

Por fim, cada um dos casos de teste gerados foi submetido para as nove (9) plataformas, cujos erros injetados estão descritos na Tabela 3, e os três verificadores de memória. Uma destas plataformas consta na utilização do *gem5* sem haver a injeção de algum erro de projeto.

## 6.3 SENSIBILIDADE AO TIPO DE ERRO

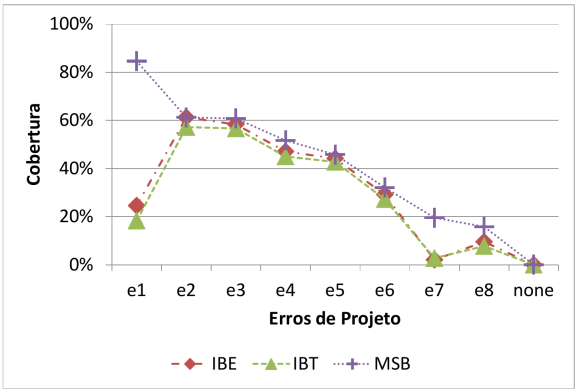
A Figura 10 mostra a qualidade de verificação (medida pela cobertura: percentagem dos casos de testes em que cada verificador diagnosticou a presença de alguma inconsistência com o modelo de memória alvo) de cada erro de projeto injetado na plataforma, considerando três cenários distintos onde ocorre um aumento no tamanho dos casos de testes. A figura também confirma que nenhum dos verificadores diagnosticou alguma inconsistência quando a plataforma não foi injetada com algum erro de projeto (campo *none*), isto é, nenhum dos verificadores em questão sinalizou algum falso negativo.

Um detalhe a ser observado pelo leitor é que utilizamos um aumento no **tamanho normalizado dos casos de testes** ($n/p$). Afinal, a qualidade de verificação poderia diminuir caso o número de núcleos de processamento dos experimentos fosse aumentado enquanto se mantivesse fixo o número total de operações de memória.

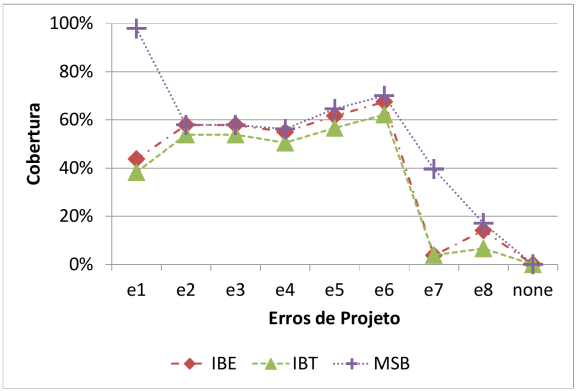
Uma análise rápida destes gráficos nos permite constatar que, como esperado, alguns erros são mais difíceis de serem detectados que outros (e.g. nos três cenários $e1$ foi o mais fácil; enquanto $e8$, o mais difícil). Além disso, a eficácia cresce conforme adotamos casos de testes maiores. Observe como a cobertura do IBT, para o erro $e8$, cresce conforme alteramos os cenários de (a) a (c).

Como comentado anteriormente, estes experimentos utilizam duas versões do XCHECK: uma completa que atravessa o grafo de fronteiras (*Inference with BackTracking* - IBT) e outra incompleta por realizar, apenas, a construção do grafo de execução (*Inference Best Effort* - IBE).

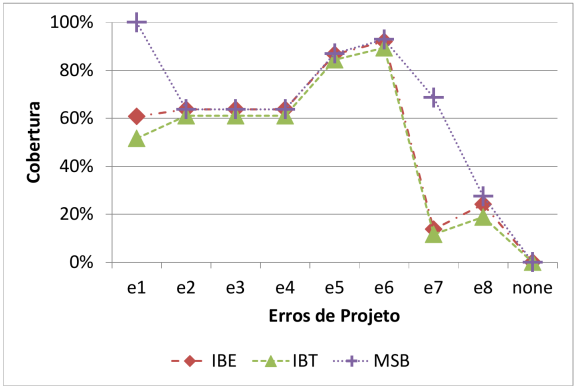
As duas versões do XCHECK mostraram uma eficácia bem similiar, cujas diferenças mais expressivas ocorreram no segundo cenário (10c) com três erros de projeto ($e5$, $e6$, $e8$), relacionados aos níveis privados do sistema




(a) $\frac{n}{p} = 125$




(b) $\frac{n}{p} = 500$




(c) $\frac{n}{p} = 2000$

Figura 10 – Eficácia por erro de projeto

de memória. Além disso, estas versões demonstraram uma eficácia competitiva com o MSB, apresentando dificuldade para detectar os erros localizados na unidade de controle de execução ($e1$ e $e7$).

Para entender um pouco mais sobre as características destes três gráficos, considere os erros $e2$, $e3$ e $e4$ no Gráfico 10c. Nos experimentos, observamos que nenhum destes erros ocorre quando o número de endereços é menor do que quatro (4), o que é dois sextos ($2/6$) dos casos de testes. Isto explica porque a eficiência destes erros satura próxima dos 66% de cobertura (HENSCHEL, 2014, p. 56).

Apesar da variação da cobertura, ocasionada pelas características intrínsecas de cada erro de projeto, a eficácia do MSB é superior a todos os outros verificadores nos três cenários de qualidade de verificação.

## 6.4 IMPACTO DO NÚMERO CRESCENTE DE OPERAÇÕES

A Figura 11 captura a eficácia global para cada cenário de qualidade de verificação ($n/p$). Para tanto, a percentagem de cada cenário foi calculada utilizando-se todo o intervalo de valores de núcleos de processamento ($p$).

Assim como esperado, para todos os verificadores, a eficácia média cresce com o tamanho normalizado dos casos de teste. Note que o MSB possui uma diferença de, pelo menos, 9% com relação às duas versões do XCHECK, para todos os cenários. Característica marcante principalmente pelo efeito de minimização do tamanho dos casos de teste. Por exemplo, o MSB consegue a cobertura de, aproximadamente, 47% utilizando casos de testes quatro (4) vezes menores que o IBE: 250 versus 1000 operações por processador, respectivamente. Quanto menor for o tamanho dos casos de teste, mais rápida é a verificação de memória e, consequentemente, todo o processo de desenvolvimento do produto. Além do mais, este ganho torna-se mais expressivo para a verificação pré-silício, onde a geração dos *traces* é feita através da simulação das operações em uma representação executável. Tarefa mais trabalhosa e, portanto, mais lenta que a execução destas mesmas operações em um protótipo do *hardware*, como ocorre na etapa pós-silício.

A Figura 12 mostra a eficiência global para cada cenário, cujos valores foram obtidos de maneira análoga à eficácia. Nesta figura, é possível distinguir o tempo de simulação (e de geração dos *trace*) do tempo de verificação propriamente dito de cada uma das técnicas, sendo que o primeiro é dado pela parte inferior e texturizada das barras.

Observe, ainda na Figura 12, que o tempo de verificação do MSB tanto é o mais rápido (sendo, em média, 4.1 vezes mais rápido que o IBE) quanto é aquele que cresce menos conforme o aumento do tamanho normalizado dos

testes. Além disso, este gráfico evidencia o principal caracterizador dos *on-the-fly*: a capacidade de reduzir o tempo da simulação. Por exemplo, para $n/p = 1000$ o MSB reduz o esforço de simulação em duas (2) vezes.

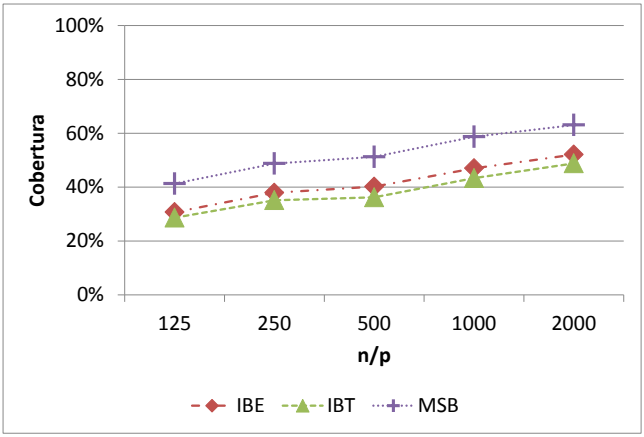



Figura 11 – Eficácia para casos de testes com tamanho crescente

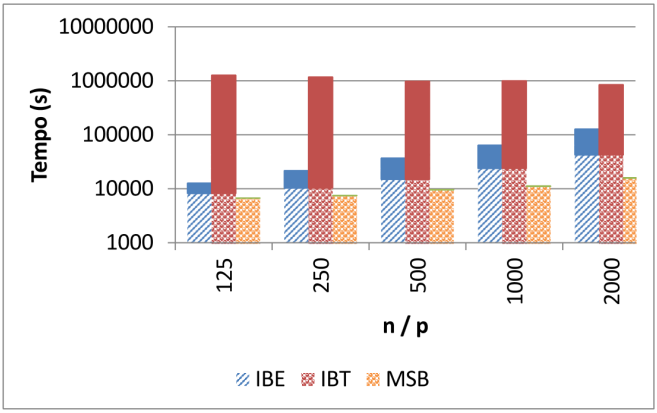



Figura 12 – Eficiência para casos de testes com tamanho crescente

Outra característica marcante mostrada por este gráfico é como o uso de um verificador *on-the-fly* torna o tempo de análise (i.e. de executar a técnica propriamente dita) insignificante, se comparado ao esforço de simulação

e ao tempo de análise de um verificador com *backtracking*.

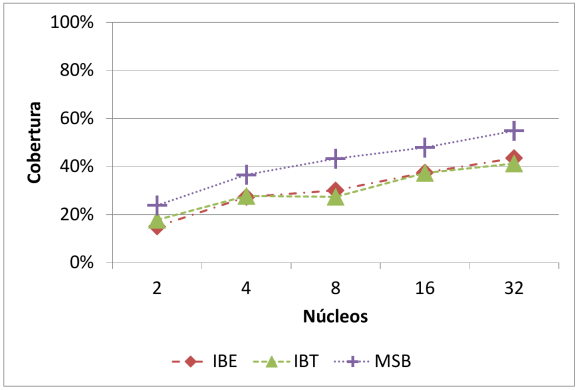
## 6.5 IMPACTO DO NÚMERO CRESCENTE DE PROCESSADORES

Por sua vez, a eficácia dos casos de teste sobre o crescimento do número de processadores foi capturada pela percentagem de casos de teste que encontram algum erro (qualquer) dados três cenários de qualidade de verificação (Figura 13). A eficiência foi obtida somando os esforços individuais para cada caso (Figura 14).

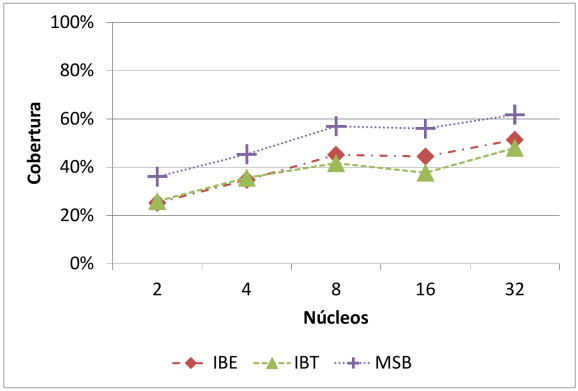
Primeiramente, repare que manter fixo um dado tamanho normalizado dos casos de teste (i.e. qualquer um dos três cenários) evita a diminuição da eficácia ao crescer com o número de processadores, confirmando o que havíamos explicado no início deste capítulo. Para um $n/p$ fixo, a eficácia tende a aumentar para um número baixo de núcleos, enquanto tende a saturar para um número alto. Afinal, qualquer erro de projeto que esteja situado em algum processador ou em alguma *cache* privativa é replicado $p$ vezes (a injeção de um erro é realizada na plataforma e, portanto, é simétrica para todos os processadores e as suas caches privativas), ou seja, conforme maior é este valor ($p$), maiores são as chances de que estes erros sejam expostos durante algum caso de teste. Além do mais, como o número de mensagens e a complexidade do diretório também crescem com o número de processadores, também se aumenta a probabilidade de um erro na manipulação das mensagens ou na atualização dos diretórios ocorrer tanto na interconexão quanto na *cache* compartilhada (dado que os diretórios residem nesta *cache*) (HENSCHEL, 2014, p. 60).

Uma interpretação interessante desta característica descrita acima é a possibilidade de aumentar o número de núcleos de processamento de um *hardware* com o intuito de utilizar técnicas que tenham trocado cobertura por eficiência. Normalmente, as técnicas de verificação realizam o contrário: diminuem a eficiência para aumentar a cobertura. Além do mais, este aumento do número de processadores ocasiona, por si só, uma queda na eficácia de todos os verificadores do estado da arte. Talvez, esta característica possa vir a ser utilizada no futuro para justificar a utilização e desenvolvimento de verificadores incompletos. Entretanto, é indiscutível que o preferível é ter um verificador com alta eficácia e eficiência.

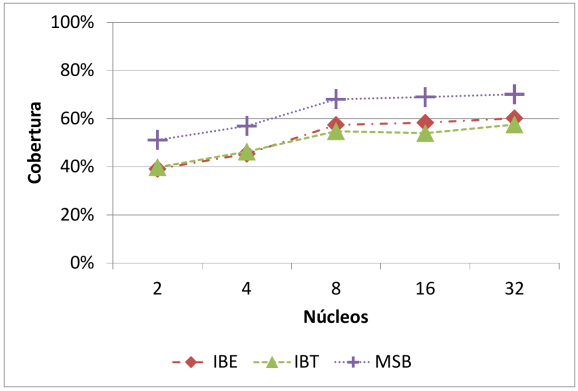
Analisando, agora, os gráficos sobre eficiência (Figura 14), note que, de novo, o MSB apresenta tanto o menor tempo de verificação quanto a menor taxa de crescimento (desta vez, pela quantidade de processadores). Para comparar a escalabilidade dos verificadores, os valores apresentados na Figura 14c foram interpolados para gerar uma equação da forma $t = k * (1+r)^p$,




(a) $\frac{n}{p} = 125$

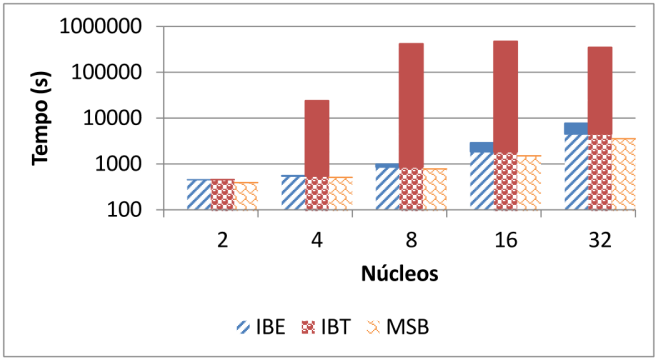



(b) $\frac{n}{p} = 500$

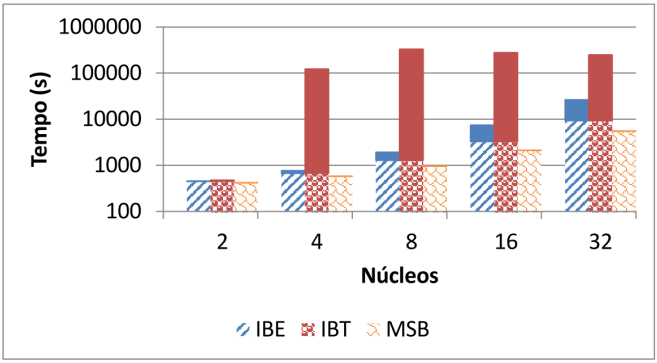



(c) $\frac{n}{p} = 2000$

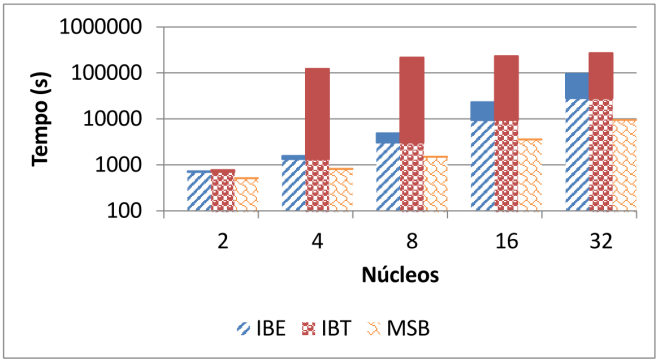
Figura 13 – Eficácia pelo número de núcleos de processamento




(a) $\frac{n}{p} = 125$




(b) $\frac{n}{p} = 500$




(c) $\frac{n}{p} = 2000$

Figura 14 – Eficiência pelo número de núcleos de processamento

relacionando o tempo total da simulação ($t$) com uma constante ($k$), a taxa de crescimento ($r$) e o número de processadores ($p$). O MSB ($r = 0.077$) obteve uma taxa de crescimento, aproximadamente, 3 vezes menor do que o IBE (0.214). Portanto, o uso pré-silício de um verificador *on-the-fly* (tal como MSB) provê plenas garantias a uma taxa de "escalabilidade" de, pelo menos, 3 vezes menor do que a de um verificador baseado em inferências sem garantias de verificação (como IBE).

Como HENSCHEL já comentou em sua dissertação, os resultados de experimentos adicionais mostraram que o MSB possui uma taxa de crescimento ainda menor do que a calculada acima, tendo em vista que ele não se adaptou bem à equação exponencial utilizada (HENSCHEL, 2014, p. 60).

## 6.6 IMPACTO DO NÚMERO CRESCENTE DE ENDEREÇOS

O último experimento realizado serviu para avaliar o impacto do número de endereços compartilhados para a verificação de memória. Os resultados destes experimentos justificam por que falamos pouco sobre os resultados obtidos pelo IBT, apresentados nas seções anteriores. Ainda que tenhamos limitado estes experimentos a plataformas com quatro (4) processadores, tivemos que manter o limite de tempo (a verificação de qualquer experimento dura no máximo 2 horas) para o IBT encontrar os erros. Caso ele não diagnosticasse algum erro durante este período, o caso era interrompido e considerava-se que o IBT não indicou algum erro.

Na Figura 15, é possível averiguar que o IBT possui uma eficiência razoável quando o caso de teste possui um único endereço compartilhado: levemente superior a sua versão de melhor esforço (IBE). Entretanto, o seu tempo de verificação cresce exponencialmente conforme o número de endereços é dobrado, até saturar devido ao limite de tempo imposto aos experimentos. Comportamento condizente com a complexidade explicada na seção 5.7.

Esta eficiência que o IBT consegue quando trabalha com um único endereço compartilhado é efeito de que os programas comportam-se da mesma maneira caso um modelo de memória não tão relaxado estivesse sendo utilizado, tais como o SC, TSO e, possivelmente, o modelo do *Godson-3*, para os quais o XCHECK foi projetado e possui garantias formais. O leitor também pode entender este efeito através da expansão do conceito de fronteiras apresentado na seção 5.7. Quando há apenas um endereço compartilhado, as fronteiras ficam iguais às elaboradas originalmente por GIBBONS; KORACH.

Perceba que a complexidade indesejada do IBT é explicada, justamente, pelo uso do *backtracking* a fim de prover as mesmas garantias de verificação obtidas pelo MSB. Porém, apesar desta realização, a eficácia do

MSB é superior ao IBT, como também ilustra a Figura 16. A eficácia do IBT possui uma pequena margem de superioridade ao IBE. Logo, todos os resultados apresentados ao longo destas últimas seções indicam que, apesar do IBT realizar uma análise superior em três ordens de grandeza (com relação ao tempo médio), ele encontra erros em apenas 1.03 vezes mais casos de teste.

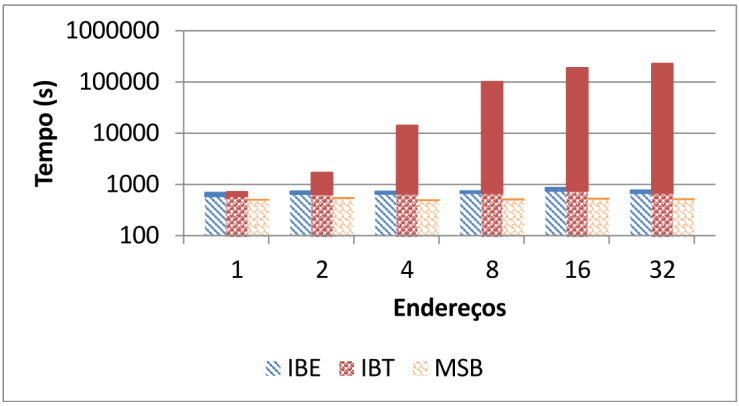



Figura 15 – Eficiência pelo número de endereços

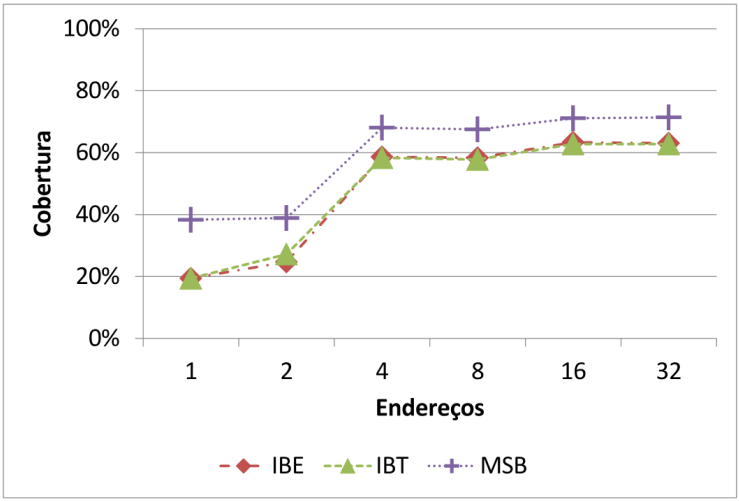



Figura 16 – Eficácia pelo número de endereços

Portanto, estes experimentos demonstram que verificadores utilizando *backtracking*, incluindo os baseados em inferência (subclasse detentora da melhor eficiência), são inadequados para o uso em modelos de memória altamente relaxados, mesmo com as otimizações feitas (seção 5.7).

# 7 CONCLUSÃO

Ao final deste trabalho a implementação realizada do XCHECK auxiliou a constatação, por meio da comparação experimental, que as técnicas de verificação pré-silício são mais eficientes e eficazes do que técnicas pós-silício. Nestes experimentos, o papel do XCHECK foi de representar a classe de verificadores pós-silício por inferência, dos quais é o detentor da melhor complexidade de verificação (linear no número de processadores).

Esta conclusão foi obtida uma vez que os resultados experimentais (Capítulo 6) deste trabalho foram combinados com os obtidos em Freitas, Rambo e Santos (2013). Além do mais, o colega HENSCHEL reforçou esta conclusão por meio de sua dissertação de mestrado (HENSCHEL, 2014), onde foram refeitos os experimentos de Freitas com a inclusão das duas implementações do XCHECK – descritas na presente monografia.

HENSCHEL considera que as tendências mostram que as arquiteturas de processadores estão caminhando para a relaxação total da ordem das operações, mantendo apenas as ordens de programas que tenham sido impostas explicitamente. Esta relaxação, em conjunto com o aumento do número de núcleos, dificulta a verificação de memória (HENSCHEL, 2014, p. 83). E, estas características afetam o desempenho da maioria dos verificadores pós-silício, como este trabalho demonstrou.

De acordo com as pesquisas presentes no estado da arte, é frequente a reutilização de verificadores pós-silício durante a etapa pré-silício. No entanto, os experimentos aqui realizados também comprovaram – assim como (RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013; HENSCHEL, 2014) – que tal atitude é inadequada já que a observabilidade disponibilizada nesta etapa permite a construção de verificadores mais eficientes e escaláveis.

Além do mais, estes verificadores pré-silício podem manter garantias plenas de verificação (e com uma mesma cobertura) enquanto são utilizados casos de teste menores. Fator decisivo na etapa pré-silício, devido à elevação existente no custo de execução dos casos de teste, com relação à etapa pós-silício que utiliza um protótipo do *hardware*.

## 7.1 ÚLTIMAS CONSIDERAÇÕES

Durante os últimos dois anos, o autor desta monografia e o, então, mestrando Olav Philipp Henschel tenham objetivado fazer uma implementação fiél à técnica proposta em Hu et al. (2012), o XCHECK. No entanto, a escrita

da presente monografia esclareceu que as dificuldades apresentadas (e.g. devido aos autores não terem apresentado o código da técnica ou não terem feitos experimentos com um *benchmark*) levaram a uma implementação incorreta, a qual utiliza o método de inferências durante o atravessamento no grafo de fronteiras.

Este engano afeta pouco as conclusões obtidas, uma vez que elas se baseiam, principalmente, na versão simplificada do XCHECK, a qual não utiliza-se deste atravessamento. O preocupante é se esta utilização incorreta do método de inferências seria a responsável pelas duas versões do XCHECK (IBE e IBT) terem apresentado os mesmos resultados.

Normalmente, seria recomendado que os experimentos fossem repetidos com uma versão correta do IBT, a fim de confirmar as conclusões. Porém, a situação presente possui uma outra alternativa: provar formalmente que o IBT nunca revelará um erro de projeto que o IBE não tenha detectado, isto é, comprovar que o atravessamento no grafo de fronteiras é desnecesário.

Na verdade esta comprovação pode ser reinterpretada como o tratamento de, somente, uma questão: sempre é possível achar uma ordenação total das operações, fiél ao modelo de memória adotado, uma vez que o grafo de execução não possui ciclos?

Uma vez que o IBE não detectou algum erro de projeto, é sabido que o grafo de execução não possui ciclos. Por outro lado, o atravessamento correto do grafo de fronteiras constitui em adicionar arestas de ordem entre todas as operações que já foram globalmente executadas (em movimento de fronteiras anteriores) com a operação que está sendo globalmente executada no movimento atual.

Agora, considere uma ordenação das operações de memória que não contradiga nenhuma das ordens presentes no grafo de execução. Primeiramente, é possível estabelecer tal ordenação, já que se trata de uma ordenação topológica de um grafo acíclico. Em segundo perceba que o atravassamento do grafo de fronteiras, seguindo esta ordenação, não pode criar algum ciclo no grafo de execução, ou seja, que o IBT não detectará algum erro de projeto. O atravassamento não pode criar ciclos constituídos somente de arestas do grafo de execução (que é acíclico) ou das novas arestas adicionas (o que contradiria a própria definição destas últimas). Por fim, o fato do atravessamento seguir uma ordenação topológica combinado com a maneira que as ordens adicionais são estabelecidas implica a invalidade para a última hipótese de ciclos, isto é, aqueles que são oriundos da contradição entre as arestas adicionadas pelo atravessamento com as presente no grafo de execução.

O leitor pode, então, perceber como a utilização do método de inferências durante o atravessamento do grafo de fronteiras era importânte: ela incapacitava a conclusão de que a realização deste último é desnecessária.

Entretanto o leitor deve perceber que o raciocíonio acima está, ainda, incompleto: o atravessamento de fronteiras de nossa implementação também adiciona arestas para compensar a omissão das operações de leitura. Assim, é necessário elaborar melhor este racioncínio em um prova formal.

## 7.2 TRABALHOS FUTUROS

Trabalhos futuros podem buscar investigar melhorias no processo de verificação de modelos de memória durante a etapa pré-silício para aumentar todos os três fatores de interesse: a eficácia (i.e. a porcentagem de erros encontrados para um dado conjunto de programas concorrentes), a eficiência (i.e. diminuir o tempo de execução da ferramenta) e as próprias garantias de verificação (i.e. garantir que o verificador não emite diagnósticos falsos, positivos ou negativos).

Embora a eficácia dependa dos programas concorrentes utilizados para estimular o sistema de memória, a maioria dos verificadores limita-se à geração automática pseudo-aleatória de programas concorrentes (RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013; HENSCHEL, 2014). A geração de programas concorrentes poderia ser dirigida por propriedades formalmente especificadas no modelo de memória. Isto propiciaria gerar estímulos relevantes, ou seja, evitando aqueles com menor potencial de revelar erros. Portanto, é possível e interessante investigar o uso de técnicas de verificação formal para se atingir este objetivo.

Alguns verificadores propostos na literatura (RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013; HENSCHEL, 2014) não exploraram melhorar a sua eficiência mesmo tendo componentes com potencial de concorrência. Ademais, é preciso investigar a viabilidade de se paralelizar um maior número de componentes destes verificadores.

Com relação à eficácia, muitos dos verificadores propostos na literatura (e.g. Rambo, Henschel e Santos (2012), Freitas, Rambo e Santos (2013), Shacham et al. (2008), Hu et al. (2012)) aplicam-se a modelos de memória que fazem hipóteses simplificadoras sobre o comportamento das operações de escrita: supõe-se que as operações de escritas são atômicas, i.e. indivisíveis. Entretanto, com o aumento do número de processadores, a hipótese de atomicidade é cada vez menos aceitável e pode levar, por exemplo, a diagnósticos de falso positivo. É preciso, então, avaliar de que forma os verificadores precisam ser modificados para se estenderem as garantias de verificação a modelos de memória com atomicidade relaxada.

Em suma, trabalhos futuros podem procurar criar: uma técnica semi-formal para a geração de programas concorrentes, uma técnica para explorar a

concorrência dos algoritmos de verificação e uma técnica de modelagem que leve em conta a relaxação da atomicidade de escrita em modelos de memória contemporâneos. Estas mesmas técnicas foram colocadas como potenciais contribuições do plano de mestrado do autor desta monografia.

## 7.3 RECONHECIMENTOS

Por fim, agradeço a todos aqueles que contribuíram direta ou indiretamente para a realização deste trabalho. Em especial, agradeço aos meus colegas de pesquisa no laboratório ECL pelos seus trabalhos desenvolvidos (Tabelas 5, 6 e 7) e ao próprio orientador, Prof. Dr. Luiz Cláudio Villar dos Santos, pelo suporte a toda pesquisa realizada e pela bolsa de iniciação científica – sem a qual eu não estaria filiado ao laboratório e a este ramo de estudo.

| **Verificador** | **Programadores** | **Referências** |
|---|---|---|
| MSB | Leandro S. Freitas | (FREITAS, 2012)<br>(FREITAS; RAMBO; SANTOS, 2013) |
| IBE e IBT | Gabriel A. G. Andrade<br>Olav P. Henschel | (CHEN et al., 2009)<br>(HU et al., 2012) |

Tabela 5 – Responsáveis pelas implementações dos verificadores utilizados

| **Atividade** | **Responsável** |
|---|---|
| Implementação preliminar | Gabriel A. G. Andrade |
| Contribuição para a versão final | Gabriel A. G. Andrade |
| Implementação da versão final | Olav P. Henschel |

Tabela 6 – Contribuições para a implementação de IBE e IBT

| **Responsável** | **ID(s)** |
|---|---|
| Leandro S. Freitas | *e1*, *e5* |
| Olav P. Henschel | *e2*, *e3*, *e6*, *e7*, *e8* |
| Gabriel A. G. Andrade | *e4* |

Tabela 7 – Os erros de projeto e os seus respectivos responsáveis

# REFERÊNCIAS

ABTS, D.; SCOTT, S.; LILJA, D. So many states, so little time: verifying memory coherence in the Cray X1. **Proceedings International Parallel and Distributed Processing Symposium**, 2003. ISSN 1530-2075.

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

ARM. **ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition**. 2012.

ARM. **ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile**. 2013.

ARVIND, A.; MAESSEN, J.-W. Memory model = instruction reordering + store atomicity. In: **Proceedings of the 33rd Annual International Symposium on Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2006. (ISCA '06), p. 29–40. ISBN 0-7695-2608-X. Disponível em: <http://dx.doi.org/10.1109/ISCA.2006.26>.

BAIER, C.; KATOEN, J.-P. **Principles of Model Checking (Representation and Mind Series)**. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

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>.

CHATTERJEE, P.; SIVARAJ, H.; GOPALAKRISHNAN, G. Shared Memory Consistency Protocol Verification Against Weak Memory Models: Refinement via Model-Checking. In: **Computer Aided Verification**. [S.l.: s.n.], 2002. p. 121–138—-.

CHEN, Y. et al. Fast complete memory consistency verification. In: **High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on**. [S.l.: s.n.], 2009. p. 381–392. ISSN 1530-0897.

DEORIO, A.; WAGNER, I.; BERTACCO, V. DACOTA: Post-silicon validation of the memory subsystem in multi-core designs. In: **Proceedings - International Symposium on High-Performance Computer**

**Architecture**. [S.l.: s.n.], 2009. p. 405–416. ISBN 9781424429325. ISSN 15300897.

DEVADAS, S. Toward a coherent multicore memory model. **Computer**, IEEE Computer Society, Los Alamitos, CA, USA, v. 46, n. 10, p. 30–31, 2013. ISSN 0018-9162.

DUBOIS, M.; SCHEURICH, C.; BRIGGS, F. Memory access buffering in multiprocessors. In: IEEE COMPUTER SOCIETY PRESS. **ACM SIGARCH Computer Architecture News**. [S.l.], 1986. v. 14, n. 2, p. 434–442.

FREITAS, L. S. **Aceleradores e multiprocessadores em chip: o impacto da execução fora de ordem na verificação de funcionalidade e de consistência**. Dissertação (Mestrado) — UFSC, Universidade Federal de Santa Catarina, 2012.

FREITAS, L. S.; RAMBO, E. A.; SANTOS, L. C. V. d. On-the-fly verification of memory consistency with concurrent relaxed scoreboards. In: **Proceedings of the Conference on Design, Automation and Test in Europe**. San Jose, CA, USA: EDA Consortium, 2013. (DATE '13), p. 631–636. ISBN 978-1-4503-2153-2.

GARNER, R. et al. **The SPARC architecture manual: Version 8**. [S.l.]: Prentice-Hall, New Jersey, 1992.

GHARACHORLOO, K.; GUPTA, A.; HENNESSY, J. L. **Revision to"Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors"**. [S.l.]: Computer Systems Laboratory, Stanford University, 1993.

GHARACHORLOO, K. et al. **Memory consistency and event ordering in scalable shared-memory multiprocessors**. [S.l.]: ACM, 1990.

GIBBONS, P. B.; KORACH, E. On testing cache-coherent shared memories. In: **Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures**. New York, NY, USA: ACM, 1994. (SPAA '94), p. 177–188. ISBN 0-89791-671-9. Disponível em: <http://doi.acm.org/10.1145/181014.181328>.

HANGAL, S. et al. Tsotool: A program for verifying memory systems using the memory consistency model. In: **Proceedings of the 31st Annual International Symposium on Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2004. (ISCA '04), p. 114–. ISBN

0-7695-2143-6. Disponível em:
<http://dl.acm.org/citation.cfm?id=998680.1006710>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. [S.l.]: Elsevier, 2012.

HENSCHEL, O. P. **Verificação de Consistência e Coerência de Memória Compartilhada para Multiprocessamento em Chip**. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, Florianópolis, SC, 8 2014.

HENZINGER, T. A.; QADEER, S.; RAJAMANI, S. **Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems**. [S.l.], 1999. Disponível em:
<http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/3764.html>.

HU, W. et al. Linear time memory consistency verification. **Computers, IEEE Transactions on**, v. 61, n. 4, p. 502–516, 2012. ISSN 0018-9340.

INC", S. M. **UltraSPARC Architecture 2005**. 2008.
`http://www.oracle.com/technetwork/systems/opensparc/t1-06-ua2005-d0-9-2-p-ext-1537734.html`.

INTEL, R. **and IA-32 Architectures. Software Developer¿s Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C**. [S.l.]: Intel, 64.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Commun. ACM**, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em:
<http://doi.acm.org/10.1145/359545.359563>.

LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. **Computers, IEEE Transactions on**, IEEE, v. 100, n. 9, p. 690–691, 1979.

LEE, E. A.; SESHIA, S. A. **Introduction to embedded systems: A cyber-physical systems approach**. [S.l.]: Lee & Seshia, 2011.

LENOSKI, D. et al. The directory-based cache coherence protocol for the DASH multiprocessor. **[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture**, 1990. ISSN 01635964.

MANOVIT, C. Efficient algorithms for verifying memory consistency. In: **In SPAA¿05: Proceedings of the 17th Annual ACM Symposium on**

**Parallelism in Algorithms and Architectures**. [S.l.: s.n.], 2005. p. 245–252.

MANOVIT, C.; HANGAL, S. Completely verifying memory consistency of test program executions. In: **High-Performance Computer Architecture, 2006. The Twelfth International Symposium on**. [S.l.: s.n.], 2006. p. 166–175. ISSN 1530-0897.

MARTIN, M. M. K.; HILL, M. D.; SORIN, D. J. Why on-chip cache coherence is here to stay. **Commun. ACM**, ACM, New York, NY, USA, v. 55, n. 7, p. 78–89, jul. 2012. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/2209249.2209269>.

MATEOSIAN, R. **The PowerPC Architecture - A Specification of a New Family of RISC Processors [Micro Review]**. 1994. 2¿ p. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=363078>.

MAY, I. Ibm system/370 principles of operation. **Publication Number GA22-7000-9, File**, n. S370-01, 1983.

RAMBO, E.; HENSCHEL, O.; SANTOS, L. dos. On esl verification of memory consistency for system-on-chip multiprocessing. In: **Design, Automation Test in Europe Conference Exhibition (DATE), 2012**. [S.l.: s.n.], 2012. p. 9–14. ISSN 1530-1591.

RAMBO, E. A. **Verificação de consistência de memória para sistemas integrados multiprocessados**. Dissertação (Mestrado) — UFSC, Universidade Federal de Santa Catarina, 2012.

RAMBO, E. A.; HENSCHEL, O. P.; SANTOS, L. C. dos. Automatic generation of memory consistency tests for chip multiprocessing. In: **Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on**. [s.n.], 2011. p. 542–545. Disponível em: <http://dx.doi.org/10.1109/ICECS.2011.6122332>.

ROY, A. et al. Fast and generalized polynomial time memory consistency verification. In: **Proceedings of the 18th International Conference on Computer Aided Verification**. Berlin, Heidelberg: Springer-Verlag, 2006. (CAV'06), p. 503–516. ISBN 3-540-37406-X, 978-3-540-37406-0.

SHACHAM, O. et al. Verification of chip multiprocessor memory systems using a relaxed scoreboard. In: **Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2008. (MICRO 41), p.

294–305. ISBN 978-1-4244-2836-6. Disponível em: <http://dx.doi.org/10.1109/MICRO.2008.4771799>.

SINDHU, P. S.; FRAILONG, J.-M.; CEKLEOV, M. **Formal specification of memory models**. [S.l.]: Springer, 1992.

SITES, R. L. **Alpha architecture reference manual**. [S.l.]: Prentice Hall PTR, 1992.

# APÊNDICE A – Artigo sobre o TCC

# Análise comparativa entre dois verificadores de consistência e de coerência de memória

**Gabriel Arthur Gerber Andrade**[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Campus Universitário – Trindade – Florianópolis – SC – Brasil

gabriel.arthur@inf.ufsc.br

***Abstract.*** *Currently, processor architectures tend to adopt measures to improve the performance that require better memory verification techniques. As an example, the adoption of memory models increasingly more relaxed. The quantitative comparison made in this paper demonstrate the potential of using specific techniques for pre-silicon; which are both more efficient as more effective than post-silicon techniques.*

***Resumo.*** *Atualmente, as arquiteturas de processadores tendem a adotar medidas para aumentar o desempenho que requerem melhores técnicas de verificação de memória. Como, por exemplo, a adoção de modelos de memória cada vez mais relaxados. A comparação quantitativa realizada neste artigo comprova o potêncial de utilizar técnicas específicas para pré-silício; tanto mais eficiêntes quanto eficazes do que técnicas pós-silício.*

## 1. Introdução

Quanto antes um erro de projeto for detectado, menor será seu impacto no custo de desenvolvimento de um sistema computacional. Por isso, é necessário iniciar-se o processo de verificação funcional antes mesmo de ser produzido um protótipo do sistema, ou seja, durante a etapa pré-silício. Nesta etapa, a verificação é realizada em uma representação do sistema em desenvolvimento (DUV).

O subsistema de memória compartilhada usado em *multicores* é extremamente complexo e sujeito a erros de projeto. A verificação deste subsistema busca averiguar se, para uma dada execução de um programa concorrente, o comportamento observado no *hardware* obedece à sua especificação, i.e. a um modelo de memória formalmente especificado, que captura aspectos de consistência e de coerência da memória compartilhada.

A literatura apresenta duas linhas com relação à verificação de memória: a verificação estática e a verificação dinâmica.

De acordo com Henshcel (2014), a verificação estática (ou abordagem matemática) [Henzinger et al. 1999, Chatterjee et al. 2002, Abts et al. 2003] trata instâncias simplificadas do problema de verificação: busca provar matematicamente a corretude de uma **abstração** da real implementação com a especificação do modelo de memória. Esta abordagem é estática e capaz de encontrar erros nos estágios iniciais do projeto (e.g. erros de protocolo). Entretanto, ela deixa escapar erros de projeto originados durante os estágios posteriores: na implementação [Henschel 2014, p. 37].

A verificação dinâmica (ou baseada em simulação) limita-se em demonstrar a incorretude do sistema por meio de um contra-exemplo. Ou seja, busca encontrar alguma execução (estimulada por algum programa paralelo, caso de teste) em que o comportamento do sistema é contraditório ao especificado. O leitor pode encontrar mais informações sobre este tipo verificação no livro [Lee and Seshia 2011].

Assim, o ponto forte da verificação dinâmica, segundo [Henschel 2014, p. 37], está na capacidade de exercitar todos os detalhes do sistema: o *hardware* real [Hangal et al. 2004, Roy et al. 2006, Manovit 2005, Manovit and Hangal 2006, Chen et al. 2009, Hu et al. 2012], um protótipo [Lenoski et al. 1990], e sua representação executável [Shacham et al. 2008, Rambo et al. 2012, Freitas et al. 2013]. O seu ponto fraco está na dificuldade em alcançar a completude, i.e. em garantir a inexistência de erros de projeto. Afinal, o resultado de sua verificação diz respeito somente às execuções exercitadas (pelos programas paralelos). E, normalmente, estas últimas são apenas uma amostragem do universo de todas as execuções possíveis do sistema.

A complexidade do problema de verificação depende da observabilidade de eventos em memória. Quando a verificação é feita em protótipo, a observabilidade é limitada pelo *hardware* a um único *trace* por processador. Assim, o problema de verificação é intratável no caso geral. Entretanto, como a observabilidade é virtualmente ilimitada em representações executáveis, a verificação pré-silício recai em problemas mais simples. Por isso, técnicas de verificação pré-silício são mais eficientes e eficazes do que técnicas pós-silício, como foi comprovado, durante os últimos anos, através da comparação experimental entre verificadores pré-silício, desenvolvidos por mestrandos da UFSC [Rambo 2012, Freitas 2012], e outros que representam o estado da arte das técnicas pós-silício [Shacham et al. 2008].

A pesquisa realizada por estes mestrandos, no laboratório ECL da UFSC, procurou avaliar as vantagens em desenvolver um verificador específico para a fase pré-silício. Tendo resultado, mais precisamente, na conclusão de que, infelizmente, uma simples adaptação de um verificador pós-silício para a fase pré-silício é, realmente, incapaz de explorar completamente a queda de complexidade presente entre estas duas fases.

A fim de quantificar a vantagem em desenvolver técnicas específicas para pré-silício em relação a adaptação de técnicas pós-silício, os pesquisadores deste laboratório passaram a realizar experimentos com a finalidade de comparar as técnicas de verificação dinâmica da literatura com os seus verificadores, considerando duas características: a eficácia e a eficiência. Isto significou a necessidade de implementar localmente verificadores do estado da arte propostos por outros autores.

Uma das técnicas importantes para esta comparação é o estado da arte dentre os verificadores por inferência: o verificador XCHECK, proposto em [Hu et al. 2012]. No entanto, mesmo com a sua importância, a comparação destes pesquisadores não utilizou uma implementação desta técnica – a fim de poder cumprir com o seu prazo previsto.

O presente artigo relata uma comparação experimental da técnica pré-silício proposta por Freitas com a implementação realizada, em conjunto com Olav Philipp

Henschel, do verificador XCHECK [Hu et al. 2012] – o estado da arte dos verificadores pós-silício baseados em inferência.

Este artigo se organiza da seguinte maneira: Primeiramente, será apresentado o estado da arte (Seção 2), com relação aos verificadores dinâmicos de interesse, e a plataforma de testes (Seção 3). Esta última permitiu a comparação quantitativa das técnicas de [Freitas et al. 2013, Hu et al. 2012] por meio de experimentos. A seguir, a Seção 4 explicará a implementação feita da técnica de [Hu et al. 2012]; assim como a sua adaptação para o modelo de memória utilizado na plataforma de testes. Por fim as Seções 5 e 6 irão apresentar a comparação propriamente dita e as conclusões finais.

## 2. Trabalhos Correlatos

Com o propósito de discutir a relevância do XCHECK, esta seção mostra as técnicas dinâmicas relacionadas com a avaliação aqui pretendida: aquelas baseadas em *sotfware*. Estas técnicas, cujas principais características estão descritas na Tabela 1, podem ser dividas em duas categorias: verificação *post-mortem* e verificação *on-the-fly*.

### 2.1. Verificação post-mortem

Os verificadores desta categoria são caracterizados pela geração de todos os *traces* antes de que a sua verificação possa se iniciar.

A maioria dos verificadores dinâmicos se baseia na detecção de ciclos em grafos orientados, em que os vértices representam operações de memória (escritas e leituras) e as arestas representam uma relação de ordem. Desta forma, todo ciclo significa um paradoxo: todas as operações envolvidas devem terminar antes mesmo de terem se iniciado (e vice-versa).

Na prática, são adotadas relações parciais que, como podem "disfarçar" uma instância inválida do problema, tornam incompleto estes verificadores. Este disfarce acontece quando não há alguma relação de ordem total, estendida da original, que seja uma instância válida do problema.

Para mitigar a ocorrência de falso-negativos, estas ferramentas tentam inferir o maior número possível de arestas. TSOTool [Hangal et al. 2004] é um exemplo de ferramenta de verificação incompleta que se utiliza desse método e que, posteriormente, foi estendida para se tornar completa [Manovit and Hangal 2006].

Para eliminar completamente os falso-negativos, é necessário ter absoluta certeza de que o resultado do verificador foi baseado em uma ordenação total. Logo, é necessário inferir todas as possibilidades de arestas que diminuam o número de componentes fortemente conexos do grafo. Isto, infelizmente, significa em seguir uma heurística e, muito eventualmente, fazer um *backtracking*. Método que torna exponecial a complexidade destes verificadores.

O XCHECK [Hu et al. 2012], versão aprimorada do LCHECK [Chen et al. 2009], contorna esta complexidade limitando, principalmente, a realização do *backtracking* em um subespaço cujo tamanho depende apenas do número de processadores. Um avanço considerável, já que significa em retirar o

| Referência | Análise | Idéia-chave | Garantias | Número de monitores | Complexidade de pior caso |
|---|---|---|---|---|---|
| [Hangal et al. 2004] | *Post-mortem* | Inferência | Nenhuma | 1 | $O(n^*)$ |
| [Manovit 2005] | *Post-mortem* | Inferência | Nenhuma | 1 | $O(pn^3)$ |
| [Manovit and Hangal 2006] | *Post-mortem* | Inferência | Plenas | 1 | $O((n/p)^e pm^3)$ |
| [Roy et al. 2006] | *Post-mortem* | Inferência | Nenhuma | 1 | $O(n^4)$ |
| [Shacham et al. 2008] | *On-the-fly* | *scoreboard* | Nenhuma | 1 | $O(p^2n^2)$ |
| **(author?)** [Chen et al. 2009] | *Post-mortem* | Inferência | Nenhuma | 1 | $O(p^3n)$ |
| | *Post-mortem* | Inferência | Plenas | 1 | $O(C^p p^2 n^2)$ |
| [Hu et al. 2012] | *Post-mortem* | Inferência | Plenas | 1 | $O(C^p p^3 n)$ |
| [Rambo et al. 2012] | *Post-mortem* | Emparelhamento em grafos bipartidos | Plenas | 2 | $O(n^6/p^5)$ |
| [Freitas et al. 2013] | *On-the-fly* | Múltiplas *scoreboard* | Plenas | 3 | $O(n^2/p)$ |

**Tabela 1. Principais características dos verificadores dinâmicos [Henschel 2014]**

principal limitador de desempenho dos verificadores atuais (o número de operações) da parte exponencial de sua complexidade.

No entanto, corroborando com [Henschel 2014, p. 40], a crescente paralelização dos processadores através do aumento do número de núcleos faz com que esta técnica tenda a se tornar cada vez mais ineficiente.

Afora a detecção de ciclos, mas ainda baseado em grafos, há o método de Emparelhamento Estendido de Grafo Bipartidos [Rambo et al. 2012] que foi projetado especialmente para a verificação de representações executáveis de plataformas multiprocessadas. Nesta outra abordagem, a invalidade de uma instância é dada através de um teste de equivalência entre eventos originados de *traces* **diferentes**: "Devem existir dois monitores em cada núcleo de processamento: um na saída da unidade de consolidação de resultados (*commit unit*) e outro na interface com a memória privativa. Um grafo bipartido é, então, construído a partir destes monitores; os vértices representam as operações de memória e as arestas representam a equivalência. Outro grafo, com os mesmo vértices, representa o ordenamento de operações, imposto pelo modelo de memória, através das suas arestas. Com base nestes dois grafos é possível determinar se o comportamento global, usando relógios globais de Lamport [Lamport 1978], que utilizam marcas temporais geradas pelos monitores para cada operação. A partir deles é criado um registro de execução global, o qual é analisado por um algoritmo que determina se ele viola ou não as regras do modelo de memória." [Henschel 2014, p. 40]

### 2.2. Verificação on-the-fly

Por outro lado, se encontram os verificadores *on-the-fly*, os quais procuram invalidar uma instância antes mesmo da execução desta ter sido terminada.

*Scoreboard* Relaxado [Shacham et al. 2008] é uma técnica (genérica) de verificação que possui um princípio de funcionamento diferente do visto até o momento. Ela mantém uma tabela (a.k.a. *scoreboard*) de possíveis valores para cada endereço de memória, a qual valida o valor observado por cada uma das leituras. Durante a execução, esta tabela é atualizada de acordo com cada operação observada. Quando uma escrita é feita na memória, a tabela recebe um novo valor no endereço da operação; quando é uma leitura, os possíveis valores são filtrados. Esta última alteração reduz o número de possibilidades, mas pode significar a remoção de um valor que havia validado (incorretamente) uma leitura anterior. Uma situação possível quando leituras se sobrepõem ou completam com uma quantidade variável de tempo. A fim de superar esta limitação, o *scoreboard* relaxado também necessita utilizar o *backtracking*. Isto, no entanto, não diminui a principal vantagem desta ferramenta com relação às técnicas anteriores, isto é, o fato de realizar a análise simultaneamente à execução do programa e poder, assim, detectar erros mais rapidamente (antes mesmo que a simulação termine).

A técnica proposta em [Freitas et al. 2013] usa o mesmo mecanismo de verificação global da técnica de [Rambo et al. 2012]. A sua verificação local, no entanto, pode ser realizada *on-the-fly* devido ao suporte de um terceiro monitor (em cada núcleo) responsável por distinguir as leituras consolidadas das leituras especulativas. Isto impede estas últimas de serem indicadas como erros (um diagnóstico de

falso positivo) já que, enquanto as operações de leitura e escrita são monitoradas na interface com a memória, um monitor na unidade de consolidação se encarrega de observar as operações que foram consolidadas e outro no *buffer* de reordenamento observa quais operações concluídas foram canceladas. Por sua vez, a verificação global é realizada por meio de um *scoreboard* relaxado para cada processador, o qual é atualizado a cada operação monitorada, verificando se cada operação que chega à memória tem uma correspondente observada em algum dos outros monitores e se não há intervalo ilegal da ordem especificada pelo modelo de memória [Henschel 2014].

### 2.3. O Interesse na comparação quantitativa entre técnicas de verificação

Esta análise da literatura evidencia indícios de deficiências dos verificadores dinâmicos, os quais foram reportados em [Henschel 2014, p. 42]. Eles merecem ser confirmados por meio de uma avaliação experimental comparativa e de uma análise quantitativa de eficiência e de eficácia.

Repare que a importância de tal comparação, entre verificadores de classes diferentes não diminui, mesmo com o fato de ter sido raramente apresentada na literatura [Rambo et al. 2012, Freitas et al. 2013, Henschel 2014].

Por fim, perceba que o objetivo desta monografia está ligada em responder se um verificador para pré-silício [Freitas et al. 2013] realmente consegue ser melhor que um verificador de inferências adaptado para pré-silício, mesmo quando a complexidade deste último é melhor (linear) do que o primeiro (quadrático) com relação ao principal limitador de desempenho atual (o número de operações). Os experimentos também podem ilustrar como a diferença entre eles irá se portar no futuro (i.e. conforme o número de processadores crescer).

## 3. Plataforma de Testes

Nos últimos cinco (5) anos, mestrandos do curso de Ciências da Computação, ligados ao laboratório ECL da UFSC, realizaram pesquisas sobre o desenvolvimento de verificadores dinâmicos específicos para a fase pré-silício do desenvolvimento de um sistema de *hardware*.

Como resultado foram desenvolvidas duas técnicas: o *E-Matching* [Rambo et al. 2012] e o *MSB* [Freitas et al. 2013]. Atualmente, o mestre Olav Philipp Henschel está desenvolvendo junto ao Prof. Dr. Luiz Cláudio Villar dos Santos uma atualização desta última técnica capaz de lidar com *Weak Write Order* (WWA).

Durante a elaboração destas pesquisas também foi criada uma plataforma de testes a fim de possibilitar a comparação experimental das técnicas desenvolvidas com implementações locais de técnicas do estado da arte. “Estes experimentos caracterizam cenários realistas de projeto, levando em conta as arquiteturas atuais e sua evolução esperada nos últimos anos” [Henschel 2014, p. 43].

O mecanismo de comparação adotado para verificadores de memória se constitui em avaliá-los com relação a duas caraceterísticas que lhes são fundamentais. Primeiramente, um verificador ideal detecta qualquer erro de projeto que o DUV possa ter. Assim, a sua medida de eficácia é dada pela percentagem dos experimentos em que um dado verificador diagnostica um erro de projeto. Repare que,

como os dois verificadores a serem comparados são completos (não diagnosticam falso positivo ou falso negativo), podemos comparar esta medida diretamente: o verificador mais eficaz é aquele que possui a maior percentagem de erros detectados nos experimentos realizados. A segunda característica é a velocidade de verificação (eficiência), cuja medida significa a soma entre o tempo de simulação e o tempo de processamento da própria técnica.

Agora, por quê a eficácia está sendo medida pela cobertura do verificador? A eficácia de um verificador refere-se, antes de mais nada, à sua capacidade em detectar os erros de projeto contidos no DUV, isto é, está relacionada com as garantias de verificação. Porém, como os verificadores em questão são dinâmicos, a percepção da eficácia de um destes verificadores é dependente dos casos de testes (programas) escolhidos para serem submetidos à verificação. Naturalmente, qualquer verificador dinâmico é incapaz de perceber os erros de projeto caso só sejam utilizados casos de teste que exercitam nenhum destes erros.

Uma maneira de trabalhar esta questão é criar casos de testes com o intuito de conterem as características necessárias para exercitar algum erro de projeto em específico. No entanto, a infraestrutura deste trabalho lida com casos de testes aleatórios. Portanto, é possível que, mesmo que seja conhecido o erro de projeto contido no DUV (durante um certo experimento), o verificador utilize um caso de teste que não o exercite. Assim, a cobertura é uma medida interessante para a eficácia, já que ela revela não apenas a capacidade do verificador em detectar algum erro de projeto (utilizando casos de testes aleatórios), como, também, a sua "facilidade".

Enfim, como o leitor pode imaginar, a experimentação feita sobre esta plataforma é orientada a casos de teste sendo, naturalmente, dividida em quatro (4) etapas: a compilação do *gem5*, a geração dos casos de teste, a realização propriamente dita dos experimentos e a análise dos resultados obtidos.

Antes de mais nada, será apresentada uma visão geral do *gem5* para entender a motivação desta primeira etapa. Em seguida, serão descritas cada uma das outras etapas de experimentação.

### 3.1. Visão geral acerca do *gem5*

O simulador *open-source gem5* (`www.gem5.org/`) é uma plataforma modular para a pesquisa de arquiteturas de sistemas computacionais, abrangendo tanto arquiteturas de alto nível quanto microarquiteturas de processadores. Além disto, oferece flexibilidade para escolher características da arquitetura que será simulada como, por exemplo, os modelos da CPU, da execução do sistema e do próprio sistema de memória.

A maioria dos componentes deste sistema foram escritos em C++. Porém, para atender ao seu interesse em modulariedade, estes códigos são portados para Python para permitir a instanciação dinâmica dos seus módulos (com a parametrização das características a serem simuladas) através do interpretador Python. Portanto, é possível criar e executar sistemas com configurações diferentes, sem precisar recompilar todo o código.

Uma outra parte dos arquivos deste simulador estão escritos em SLICC e Ruby. Este primeiro é uma linguagem utilizada para especificar os protocolos de coerência de cache. O compilador SLICC gera código C++ para cada controlador envolvido nos protocolos. Estes podem trabalhar em conjunto com outras partes do Ruby (`www.m5sim.org/SLICC`).

Caso o leitor tenha interesse, o site oficial do *gem5* disponibiliza uma descrição geral da organização do seu código-fonte (`www.gem5.org/Source_Code`), assim como uma versão *HTML (HyperText Markup Language)* da sua documentação (`www.gem5.org/docs/html/index.html`).

Enfim, o *gem5* é uma ótima infraestrutura para os experimentos aqui pretendidos e, por maior que seja, há diversas fontes que esmiuçam os seus detalhes [Binkert et al. 2011]. Ainda assim, ele deixa a desejar em dois pontos com relação aos experimentos desejados: (a) ele não produz os *traces* necessitados pelos verificadores e (b) não há a possibilidade de simular sistemas com algum erro de projeto (ou melhor, espera-se tal característica). Por esta razão, além da produção de *scripts* objetivando automatizar os testes, também alterou-se o *gem5* para suportar estes dois pontos.

Para atender à produção dos *traces*, foram criadas classes denominadas monitores, cujos métodos de observação são chamados em locais estratégicos do código-fonte do simulador. Os monitores dos verificadores *post-mortem* gravam os *traces* em arquivos alvos, enquanto os verificadores *on-the-fly* são repassados diretamente com as informações dos seus monitores.

Cada erro de projeto (Tabela 2) foi modelado por meio da inserção de códigos que realizam o seu comportamento característico. Com o detalhe de que a execução de cada trecho é condicionada a um parâmetro específico repassado para a interface do simulador. Este parâmetro referencia o (único) erro de projeto cuja simulação é desejada para um dado experimento. Há a opção de simular a versão original do *gem5* - livre de algum erro. Afinal, os experimentos são pretendidos para avaliar a verificação durante a fase pré-silício, mesmo que também estejam sendo avaliados verificadores desenvolvidos para pós-silício.

Outros erros de projeto também foram modelados, além dos reportados na Tabela 2. No entanto, estes não foram inclusos na comparação aqui relatada devido a facilidade em serem detectados ou a nunca serem exercitados para a configuração pretendida nos experimentos.

Uma ressalva ao leitor é que este trabalho objetiva capturar, por meio dos experimentos, a capacidade de cada verificador em detectar um certo erro de projeto presente na literatura. Afinal, este dado é um avaliador da eficácia dos verificadores de memória. Portanto, não há o interesse em realizar alguma simulação sujeita a mais de um erro de projeto.

### 3.2. Etapa 1: Compilação dos módulos do *gem5*

A primeira atitude a ser tomada para utilizar o *gem5* constitui em preparar o código-fonte para ser interpretado em Python (e.g. os arquivos escritos em SLICC e C++). Esta preparação automática, controlada por um construtor *open-source* implemen-

| ID | Descrição | Localização |
|---|---|---|
| *e1* | Violação da barreira de memória para leituras | Unidade de controle de execução |
| *e2* | L2 não escreve dado bloco da *cache* quando recebe uma resposta da L1 | Controlador da *cache* L2 |
| *e3* | L1 sempre vai ao estado exclusivo quando recebe dado para leitura | Controlador da *cache* L1 |
| *e4* | L2 não envia mensagem de invalidação para L1 quando o seu estado é Compartilhado ou Exclusivo | Controlador da *cache* L2 |
| *e5* | Escrita inacabada não é vista por leitura para o mesmo endereço | Mecanismo de adiantamento |
| *e6* | Leitura não obtém o valor da escrita consolidada quando ela está na última posição da fila de escritas | Mecanismo de adiantamento |
| *e7* | Quando uma barreira de memória completa, outras barreiras de memória em execução são ignoradas pelas leituras | Unidade de controle de execução |
| *e8* | L1 vai para o estado compartilhado quando recebe dado antigo para leitura, ao invés de permanecer Inválido | Controlador da *cache* L1 |

**Tabela 2. Caracterização dos erros de projeto inseridos, propositalmente, durante os experimentos [Henschel 2014, p. 47]**

tado em Python (o *SCons*), é sujeita à escolha tomada entre as cinco versões de configuração disponíveis. Os detalhes sobre os parâmetros desta construção, assim como estas cinco versões, estão disponíveis em `www.gem5.org/Build_System`.

Para os experimentos realizados nesta monografia, adotamos a versão **gem5.opt**, a qual ativa todas as otimizações do *hardware* simulado e desativa quase todas as funcionalidades de *debug*, deixando apenas aquelas similares aos *asserts* e *printfs*. Estas duas características fazem desta versão a que oferece um balanceamento entre velocidade de simulação e *insight* do que está acontecendo.

Antes de termos utilizado esta versão, realizamos testes piloto com uma versão sem otimizações: a **gem5.debug**.

É verdade que os erros de projeto quebram a lógica de "comportamento correto". No entanto, a presença de tais erros (corretamente modelados) não significa a ocorrência de algum erro de simulação. Por esta razão, a versão **gem5.fast**, que desativa todas as funcionalidades de *debug*, é uma outra opção válida para ser utilizada nos nossos experimentos. O único motivo pelo qual não a utilizamos foi para ter absoluta certeza de que nenhum experimento seria diagnosticado como tendo algum erro de projeto quando, na realidade, isto decorreria de um erro na própria simulação.

Note que nossa decisão faz com que o tempo de simulação dos experimentos contenha também o tempo gasto no *debug* e, portanto, interfere levemente na comparação de eficiência dos verificadores.,

### 3.3. Etapa 2: Geração dos casos de teste

A infraestrutura do laboratório disponibiliza um gerador de casos de teste [Rambo et al. 2011], o qual vem sendo aprimorado no decorrer da pesquisa sobre a verificação de memória.

De acordo com o escopo dos experimentos, um caso de teste é um conjunto de programas paralelos (*threads*) que serão executados em cada um dos núcleos de processamento da representação executável do sistema simulado. Logo, a fim de gerar automaticamente um caso de teste aleatório, o gerador utiliza cinco (5) parâmetros: a semente do gerador de números randômicos, a probabilidade de ocorrência para cada instrução de memória, o número de instruções por núcleo, o número de núcleos de processamento e o número de endereços compartilhados. Nos experimentos, há três tipos de instruções de interesse: leitura (*load*), escrita (*write*) e a restauradora da ordem de programa (barreira de memória).

Uma chamada do gerador de casos de teste, escrito em Python, gera um programa C capaz de criar, como processos filhos, todas as *threads* desejadas. Os endereços compartilhados são modelados como inteiros voláteis sem sinal, sendo declarados sequencialmente na parte inicial do código deste programa gerado (no escopo global).

Tal programa principal possui características sutis para suportar os experimentos. Por exemplo, a inicialização das *threads*, ocorrida durante a execução deste programa, gera instruções de memória que devem ser ignoradas nos experimentos. Para tratar deste caso, o programa principal insere uma barreira de memória logo após esta inicialização. Desta maneira, os experimentos podem ser automatizados corretamente desde que haja o cuidado de lembrar a existência de uma fase de inicialização, cuja transição é sinalizada por uma barreira de memória. Henschel esclarece em sua dissertação este e outros detalhes referentes aos cuidados realizados para este gerador [Henschel 2014, p. 43-44].

A realização de casos de teste aleatórios (sem *constraints*) é um efeito direto da metodologia escolhida. Entretanto, a fim de permitir que os experimentos pudessem ser repetidos automaticamente em outro computador, é necessário controlar a "aleatoriedade" do gerador. É por esta razão que o gerador aceita como parâmetro a semente aleatória (random seed) que utiliza para definir as instruções de memória de cada *thread*.

Na realidade, esta semente é utilizada para inicializar um gerador interno de números aleatórios. E são esses números os responsáveis pela decisão entre qual tipo de operação de memória será utilizada em uma posição específica de uma *thread*. As leituras são representadas pelos $L$ primeiros números, as escritas pelos $S$ seguintes, enquanto que as barreiras de memória ficam com os $M$ últimos. Os números $L$, $S$ e $M$ são parâmetros do gerador e são interpretados como a proporção de cada tipo de operação para o programa que está sendo gerado.

### 3.4. Etapa 3: Experimentos

Após as etapas anteriores serem completadas, todas as dependências para a realização dos experimentos terão sido satisfeitas. Por isto, cabe a esta etapa realizar os experimentos e, principalmente, coletar os seus resultados.

Os experimentos são realizados sobre representações de projeto (denominadas por plataformas) executadas através do simulador *gem5* utilizando os casos de teste gerados aleatoriamente na etapa anterior. Os parâmetros arquiteturais são escolhidos (e fixados) de maneira realista e compatível com sistemas embarcados, sendo injetado no máximo um único erro de projeto por experimento (daqueles listados na Tabela 2).

Assim, um experimento é caracterizado por oito (8) informações, isto é, uma tupla $(R, L, S, N, P, A, error, checker)$. As seis primeiras informações se referem ao caso de teste e são os parâmetros utilizados para a sua geração: a semente randômica ($R$), a proporção de leituras ($L$), de escritas ($S$), de barreiras de memória ($1-L-S$), a quantidade de operações de memória por núcleo ($N$), a quantidade de núcleos de processamento ($P$) e a quantidade de endereços compartilhados ($A$). A sétima informação é o erro de projeto injetado no *gem5*, o qual caracteriza a plataforma que estará sendo testada pelo verificador, especificado pelo último elemento da tupla ($checker$).

Com este formato em mente, criou-se um *script*, em *Python*, que realiza diversos experimentos a partir da combinação das configurações desejadas: os parâmetros dos casos de testes, as plataformas (i.e. erros de projeto) e os verificadores. Este *script*, compila o *gem5*, chama o gerador de casos de teste e realiza os experimentos e o pós-processamento dos resultados, i.e. computa as medidas de eficiência e eficácia e as compila em uma tabela do *Excel*.

Outra característica presente neste *script* é o envio dos resultados compilados para um servidor (fixo), permitindo, assim, que possamos dividir os experimentos em diversos computadores: cinco (5) no total. Uma atitude pertinente, principalmente, em vista do grande tempo de verificação do *XCHECK*.

### 3.5. Etapa 4: Síntese

A etapa anterior nos oferece uma tabela do *Excel* que sintetiza os resultados dos experimentos. Cada linha se refere a um experimento, cujos atributos caracterizadores definem as primeiras colunas. Enquanto isso, há colunas acerca das informações de cada verificador (tempo de verificação e diagnóstico) e da própria simulação (tempo de geração dos *traces* e o diagnóstico sobre a presença de erro de simulação). Portanto, é necessário refinar estes dados tanto em valores métricos adequados quanto em gráficos.

Uma abstração dos experimentos e seus respectivos resultados nos permitiu criar uma planilha do *Excel* bastante flexível, ao ponto de automatizar boa parte desta síntese. Isto é, na prática, necessitamos apenas copiar os dados gerados pelo *script* para uma guia reservada desta planilha. A única exceção é em relação aos gráficos de eficiência, cujas colunas devem ser agrupadas e empilhadas. Cada coluna diz respeito a um dos verificadores e, portanto, constitui no empilhamento do

tempo de verificação sobre o tempo de simulação do verificador. A maneira como conseguimos criar estes gráficos foi através da elaboração manual de uma tabela auxiliar.

## 4. Implementação do XCHECK

Em 2012, Hu et al. publicaram o XCHECK, um verificador de memória *post-mortem* para a etapa pós-silício com complexidade de verificação linear sobre o número de operações de memória, o principal limitador de desempenho. Tal complexidade foi impactante devido ao fato deste verificador não necessitar da ajuda de nenhum *hardware* dedicado.

A ideia chave do XCHECK é de aproximar a ordem temporal para fazer uma verificação localizada, referente apenas aos pares de operações de memória que não são ordenados por ela. Uma herança do seu predecessor, o LCHECK, publicado em [Chen et al. 2009].

As primeiras tarefas do XCHECK tratam de processar as informações do *trace* (instância) para criar o Grafo de Execução [Chen et al. 2009, p. 388], o qual aplica as regras do modelo de memória como ordens (arestas) entre as operações de memória (vértices). Caso este grafo não seja acíclio, então o verificador conclui que a instância é negativa. A sua construção pode ser realizada em $O(np^3)$ com o auxílio do método de inferências [Chen et al. 2009, p. 389].

Monitores presentes na plataforma de testes obtém, e registram em *traces*, os intervalos de tempo necessários para estabelecer a ordem de tempo físico [Chen et al. 2009, p. 506] através do próprio tempo de simulação do *gem5*. Com isto os intervalos obtidos são mais refinados do que caso tivesse sido utilizada alguma das técnicas propostas em [Hu et al. 2012, p. 511].

No entanto, este refinamento fez com que os intervalos passasem a capturar, também, as reordenações realizadas pelo sistema de memória. E, o efeito negativo desta captura é o aumento da complexidade para cálcular os intervalos de espera (para toda operação de memória). O intervalo de espera de uma operação de memória é o conjunto das operações com que não possuem alguma ordem de tempo físico. Enfim, a única maneira de computar estes conjuntos é checando todas as operações de memória. Porém, tendo em vista o dessinteresse em adicionar ao XCHECK uma complexidade $O(n^2)$, a qual deteriora a sua principal vantagem, então, foi adicionado um pré-processamento cujo tempo de execução não é utilizado no cálculo da métrica de eficiência do verificador.

Neste pré-processamento, o intervalo de espera de todas as operações é calculado e registrado em um arquivo, desta maneira a inicialização do intervalo de espera pôde ser realizada em $O(np)$, assim como previsto originalmente em [Hu et al. 2012].

Segundo [Hu et al. 2012], como o Grafo de Execução é incapaz de detectar todos os erros de projeto, é necessário estabelecer uma ordenação total entre as operações de memória com o auxílio do Grafo de Fronteiras [Gibbons and Korach 1994]. Neste último grafo, um caminho entre a fronteira inicial e a fronteiral final definem uma das possíveis ordenações totais, fiéis com o modelo de memória. O estabelecimento deste caminho gera a adição de novas arestas no Grafo de Execução. E, a

checagem de que a adição destas ordens gera algum ciclo é realizada em $O(n)$ por meio do algoritmo de checagem de ciclos [Hu et al. 2012, p. 509].

A implementação do atravassamento no Grafo de Fronteiras necessitou de algumas adaptações para suportar o modelo altamento relaxado desejado para os experimentos.

Primeiramente, o conceito de fronteiras foi expandido para capturar o fato de que a relaxação faz com que exista uma sequência de operações por endereço de processador. Sequências obtidas uma vez que as regras do modelo de memória sejam aplicadas como ordens entre as operações: mesmo um modelo de memória estabele ordens entre operações de escrita com o mesmo endereço (desde que sejam para o mesmo processador).

Ainda assim, o resultado não é exatamente uma sequência, já que não necessáriamente haverá o estabelecimento de ordens entre as leituras (mesmo aquelas para mesmo endereço e mesmo processador). Portanto, uma outra medida é ignorar as operações de leitura. Assim, as únicas operações que entram e saem das fronteiras são operações de escrita.

Para suportar esta omissão, quando uma operação de escrita ($w$) é globalmente executada, é necessário adicionar ordens entre as leituras da última escrita, de mesmo endereço, que já havia sido globalmente executada com esta primeira escrita. Formalmente, adicionar $r \rightarrow w$ no Grafo de Execução, onde $r$ é uma operação de leitura para o valor escrito por $w$', dado que é satisfeito $w' \rightarrow w$.

Considerando que haja uma constante ($D$) para o número de operações no intervalo de tempo de uma operação qualquer ($u_i$) para cada endereço compartilhado (e.g. $D = C/a$, considerando uma distribuição uniforme), então, o número de fronteiras é de $O(n\ D^{pa})$. Como existe $Cp$ operações de memória que possam estender uma fronteira envolvendo $u_i$, o número total de arestas do grafo de fronteiras constitui em $O(n\ D^{pa}\ C\ p)$ ou, somente, $O(n\ D^{pa} p)$.

Durante cada movimento de fronteira (aresta), a checagem de ciclos é realizada em $O(p)$, considerando cada uma das $O(pa)$ ordens adicionadas: a última operação globalmente executada em cada uma das $O(pa)$ sequências precede a operação que está sendo globalmente executada no movimento atual. Portanto, a complexidade de verificação do XCHECK é de $O(n\ D^{pa} p^3 a)$, para um modelo altamente relaxado. A implementação do grafo de fronteiras constitui em omitir as operações de leitura e as barreiras de memória de suas fronteiras. Esta omissão foi motivada por ser desnecessário (e computacionalmente intensivo) ordenar operações não conflitantes, como as combinações entre operações de leitura. Já as barreiras de memória foram omitidas devido à sua função ser, apenas, de estabelecer ordens no grafo de execução.

Nas duas publicações relativas ao XCHECK [Chen et al. 2009, Hu et al. 2012], os autores não esclareceram se as arestas adicionadas por um movimento de fronteira devem (ou não) ativar o método de inferências. A decisão por realizar as inferências significa o aumento do número de ordens adicionadas (e, portanto, removidas durante o *backtracking*) para cada movimento de fronteira.

Durante a realização da implementação houve um grande interesse em reduzir, o máximo possível, a quantidade de caminhos inválidos atravessados durante a utilização do grafo de fronteiras, isto é, caminhos que não levam a uma instância positiva do problema. E toda ordem inferida pode, potencialmente, antecipar o verificador no reconhecimento do caminho atual como inválido. Além do mais, como havia sido analisado que esta adição do método de inferências não afetaria a complexidade de verificação, decidiu-se por utilizá-la.

Entretanto, apenas após a realização dos experimentos, foi percebido o erro desta pressuposição; em utilizar o método de inferência durante o atravessamento do grafo de fronteiras. Há uma dificuldade em saber exatamente como esta sua utilização afeta a complexidade de verificação. Ainda assim, como ela significa que o método de inferência é chamado para cada uma das $O(pa)$ arestas adicionadas durante um movimento de fronteira, a complexidade é aumentada, no mínimo, em $O(n\ D^{pa}\ p^4\ a)$. E, em nenhum momento de [Hu et al. 2012], os autores mencionam este aumento.

## 5. Resultados Experimentais

Antes de mais nada, o leitor deve ser esclarecido que os experimentos desta monografia foram realizados no ECL em conjunto com Olav Philipp Henshcel. De modo que esta monografia apresenta uma parte dos experimentos realizados e apresentados por completo na dissertação do referido colega [Henschel 2014]. Além do mais, Henshcel já justificou que os parâmetros escolhidos propriciam um comparativo realista sobre arquiteturas atuais e futuras.

Portanto, trata de explicitar a configuração experimental e analisar os resultados obtidos, com o fim de comparar os verificadores em questão [Hu et al. 2012, Freitas et al. 2013] sob quatro pontos de vista distintos, isto é, um para a sensibilidade da eficácia ao tipo de erro de projeto (a.k.a. plataforma) e três para o impacto na eficiência e na eficácia considerando o crescimento: do número de operações do caso de teste ($n/p$), do número de núcleos de processamento e do número de endereços compartilhados.

### 5.1. CONFIGURAÇÃO EXPERIMENTAL

Os experimentos foram realizados sobre 1200 casos de testes, gerados a partir da combinação dos seguintes valores para cada parâmetro do gerador aleatório: duas sementes aleatórias (10 e 11), quatro conjuntos de proporções de leituras, escritas e barreiras de memória (Tabela 3), cinco quantidades para as instruções por núcleo (125, 250, 500, 1000, 2000), cinco quantidades para os núcleos de processamento (2, 4, 8, 16, 32) e seis quantidades para os endereços compartilhados (1, 2, 4, 8, 16, 32).

| Conjunto nž | Leituras | Escritas | Barreiras de memória |
|---|---|---|---|
| 1 | 30% | 66% | 4% |
| 2 | 48% | 48% | 4% |
| 3 | 66% | 30% | 4% |
| 4 | 80% | 16% | 4% |

**Tabela 3. Proporções para cada tipo de operação [Henschel 2014]**

O único parâmetro do *gem5* que variou durante os experimentos foi o responsável pela injeção dos erros de projeto. As outras características constituíram na adoção do ISA que apresenta o melhor suporte (i.e. *UltraSPARC Architecture 2005* (Sun Microsystems Inc., 2008)), segundo [Binkert et al. 2011], e uma hierarquia de memória *cache* estritamente inclusiva, utilizando três níveis com blocos de 64 *bytes*. O primeiro nível é privativo e dividido em *cache* de instruções e *cache* de dados (ambas 4kiB), o segundo (64KiB) é privativo e unificado, e o terceiro nível (4MiB) é compartilhado entre os processadores [Henschel 2014, p. 46]. A interconexão usada possui uma topologia em estrela baseada em um comutador (*switch*) com roteamento simples [Henschel 2014, p. 48].

Já o mecanismo de coerência, contido no terceiro nível da memória, adota um protocolo *MESI* baseado em diretórios e segue o modelo de memória implementado pelo *gem5* (uma variação da arquitetura *Alpha*). Este modelo assemelha-se a diversos modelos populares, como *Weak Ordering* (WO) [Dubois et al. 1986] cujas variações são utilizadas em processadores atuais com arquiteturas como *ARMv7* [ARM 2012] e *ARMv8* [ARM 2013] [Henschel 2014, p. 46].

Ao observarmos a grande complexidade de verificação do *XCHECK*, decidimos por também realizar experimentos com uma versão simplificada deste verificador, a qual não utiliza a técnica de *backtracking*. Além disso, instituímos uma duração máxima (2 horas) para a verificação de cada experimento, a fim de viabilizar a realização de todos estes experimentos dentro do prazo.

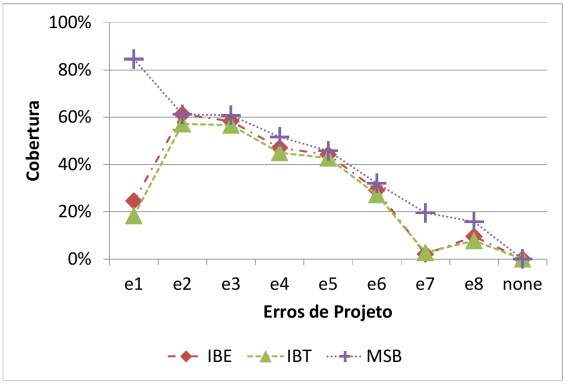
Os gráficos apresentados na próxima seção utilizam um apelido para cada uma das técnicas avaliadas: *Multi-ScoreBoard* (MSB) se refere à técnica de [Freitas et al. 2013], *Inference with BackTracking* (IBT) à técnica de [Hu et al. 2012] e *Inference Best Effort* (IBE) à versão simplificada desta última técnica.

Por fim, cada um dos casos de teste gerados foi submetido para as nove (9) plataformas, cujos erros injetados estão descritos na Tabela 2, e os três verificadores de memória. Uma destas plataformas consta na utilização do *gem5* sem haver a injeção de algum erro de projeto.

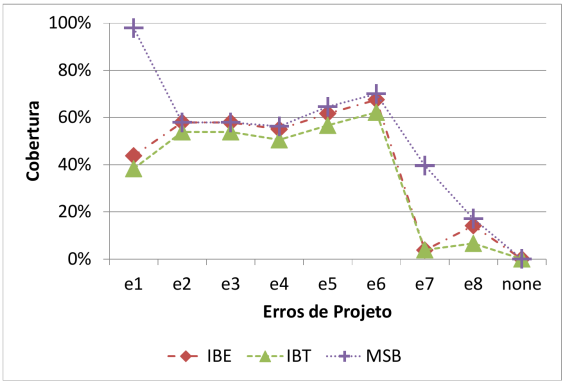
### 5.2. SENSIBILIDADE AO TIPO DE ERRO

A Figura 1 mostra a qualidade de verificação (medida pela cobertura: percentagem dos casos de testes em que cada verificador diagnosticou a presença de alguma inconsistência com o modelo de memória alvo) de cada erro de projeto injetado na plataforma, considerando três cenários distintos onde ocorre um aumento no tamanho dos casos de testes. A figura também confirma que nenhum dos verificadores diagnosticou alguma inconsistência quando a plataforma não foi injetada com algum erro de projeto (campo *none*), isto é, nenhum dos verificadores em questão sinalizou algum falso negativo.

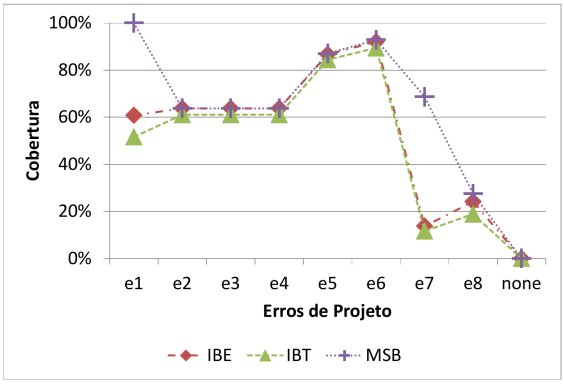
Um detalhe a ser observado pelo leitor é que utilizamos um aumento no **tamanho normalizado dos casos de testes** ($n/p$). Afinal, a qualidade de verificação poderia diminuir caso o número de núcleos de processamento dos experimentos fosse aumentado enquanto se mantivesse fixo o número total de operações de memória.




(a)




(b)




(c)

**Figura 1. Eficácia por erro de projeto**

Uma análise rápida destes gráficos nos permite constatar que, como esperado, alguns erros são mais difíceis de serem detectados que outros (e.g. nos três cenários $e1$ foi o mais fácil; enquanto $e8$, o mais difícil). Além disso, a eficácia cresce conforme adotamos casos de testes maiores. Observe como a cobertura do IBT, para o erro $e8$, cresce conforme alteramos os cenários de ((a)) a ((c)).

Como comentado anteriormente, estes experimentos utilizam duas versões do XCHECK: uma completa que atravessa o grafo de fronteiras (*Inference with BackTracking* - IBT) e outra incompleta por realizar, apenas, a construção do grafo de execução (*Inference Best Effort* - IBE).

As duas versões do XCHECK mostraram uma eficácia bem similiar, cujas diferenças mais expressivas ocorreram no segundo cenário (1(c)) com três erros de projeto ($e5$, $e6$, $e8$), relacionados aos níveis privados do sistema de memória. Além disso, estas versões demonstraram uma eficácia competitiva com o MSB, apresentando dificuldade para detectar os erros localizados na unidade de controle de execução ($e1$ e $e7$).

Para entender um pouco mais sobre as características destes três gráficos, considere os erros $e2$, $e3$ e $e4$ no Gráfico 1(c). Nos experimentos, observamos que nenhum destes erros ocorre quando o número de endereços é menor do que quatro (4), o que é dois sextos (2/6) dos casos de testes. Isto explica porque a eficiência destes erros satura próxima dos 66% de cobertura [Henschel 2014, p. 56].

Apesar da variação da cobertura, ocasionada pelas características intrínsecas de cada erro de projeto, a eficácia do MSB é superior a todos os outros verificadores nos três cenários de qualidade de verificação.

### 5.3. IMPACTO DO NÚMERO CRESCENTE DE OPERAÇÕES

A Figura 2 captura a eficácia global para cada cenário de qualidade de verificação ($n/p$). Para tanto, a percentagem de cada cenário foi calculada utilizando-se todo o intervalo de valores de núcleos de processamento ($p$).

Assim como esperado, para todos os verificadores, a eficácia média cresce com o tamanho normalizado dos casos de teste. Note que o MSB possui uma diferença de, pelo menos, 9% com relação às duas versões do XCHECK, para todos os cenários. Característica marcante principalmente pelo efeito de minimização do tamanho dos casos de teste. Por exemplo, o MSB consegue a cobertura de, aproximadamente, 47% utilizando casos de testes quatro (4) vezes menores que o IBE: 250 versus 1000 operações por processador, respectivamente. Quanto menor for o tamanho dos casos de teste, mais rápida é a verificação de memória e, consequentemente, todo o processo de desenvolvimento do produto. Além do mais, este ganho torna-se mais expressivo para a verificação pré-silício, onde a geração dos *traces* é feita através da simulação das operações em uma representação executável. Tarefa mais trabalhosa e, portanto, mais lenta que a execução destas mesmas operações em um protótipo do *hardware*, como ocorre na etapa pós-silício.

A Figura 3 mostra a eficiência global para cada cenário, cujos valores foram obtidos de maneira análoga à eficácia. Nesta figura, é possível distinguir o tempo de simulação (e de geração dos *trace*) do tempo de verificação propriamente dito de

cada uma das técnicas, sendo que o primeiro é dado pela parte inferior e texturizada das barras.

Observe, ainda na Figura 3, que o tempo de verificação do MSB tanto é o mais rápido (sendo, em média, 4.1 vezes mais rápido que o IBE) quanto é aquele que cresce menos conforme o aumento do tamanho normalizado dos testes. Além disso, este gráfico evidencia o principal caracterizador dos *on-the-fly*: a capacidade de reduzir o tempo da simulação. Por exemplo, para $n/p = 1000$ o MSB reduz o esforço de simulação em duas (2) vezes.

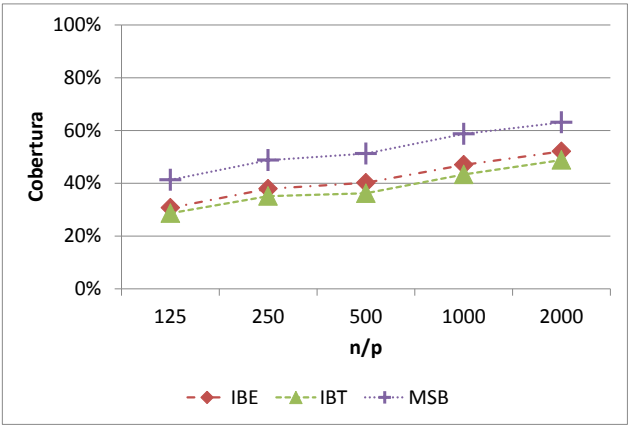



**Figura 2. Eficácia para casos de testes com tamanho crescente**

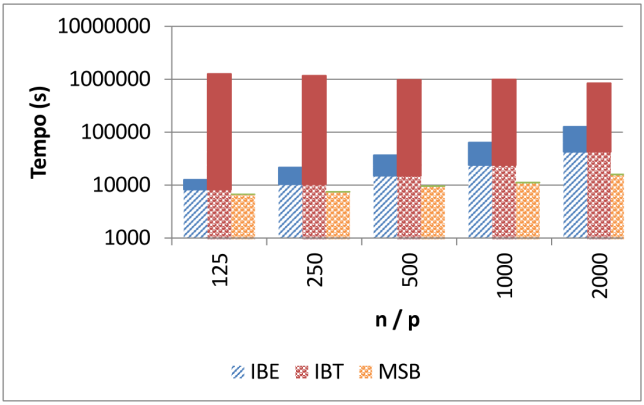



**Figura 3. Eficiência para casos de testes com tamanho crescente**

Outra característica marcante mostrada por este gráfico é como o uso de um verificador *on-the-fly* torna o tempo de análise (i.e. de executar a técnica propriamente dita) insignificante, se comparado ao esforço de simulação e ao tempo de análise de um verificador com *backtracking*.

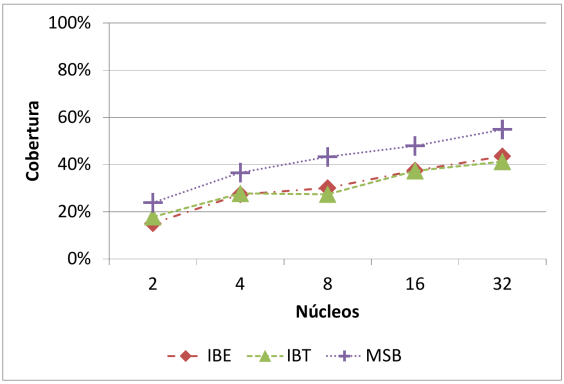
## 5.4. IMPACTO DO NÚMERO CRESCENTE DE PROCESSADORES

Por sua vez, a eficácia dos casos de teste sobre o crescimento do número de processadores foi capturada pela percentagem de casos de teste que encontram algum erro (qualquer) dados três cenários de qualidade de verificação (Figura 4). A eficiência foi obtida somando os esforços individuais para cada caso (Figura 5).

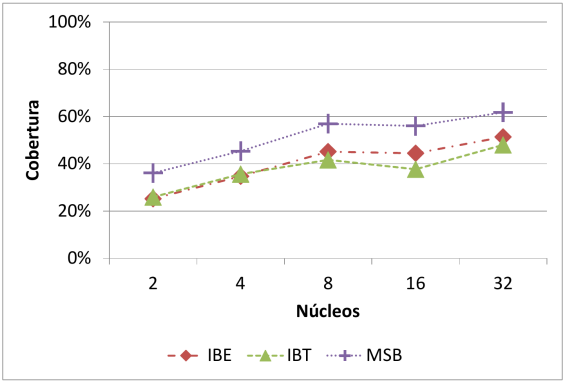
Primeiramente, repare que manter fixo um dado tamanho normalizado dos casos de teste (i.e. qualquer um dos três cenários) evita a diminuição da eficácia ao crescer com o número de processadores, confirmando o que havíamos explicado no início deste capítulo. Para um $n/p$ fixo, a eficácia tende a aumentar para um número baixo de núcleos, enquanto tende a saturar para um número alto. Afinal, qualquer erro de projeto que esteja situado em algum processador ou em alguma *cache* privativa é replicado $p$ vezes (a injeção de um erro é realizada na plataforma e, portanto, é simétrica para todos os processadores e as suas caches privativas), ou seja, conforme maior é este valor ($p$), maiores são as chances de que estes erros sejam expostos durante algum caso de teste. Além do mais, como o número de mensagens e a complexidade do diretório também crescem com o número de processadores, também se aumenta a probabilidade de um erro na manipulação das mensagens ou na atualização dos diretórios ocorrer tanto na interconexão quanto na *cache* compartilhada (dado que os diretórios residem nesta *cache*) [Henschel 2014, p. 60].

Uma interpretação interessante desta característica descrita acima é a possibilidade de aumentar o número de núcleos de processamento de um *hardware* com o intuito de utilizar técnicas que tenham trocado cobertura por eficiência. Normalmente, as técnicas de verificação realizam o contrário: diminuem a eficiência para aumentar a cobertura. Além do mais, este aumento do número de processadores ocasiona, por si só, uma queda na eficácia de todos os verificadores do estado da arte. Talvez, esta característica possa vir a ser utilizada no futuro para justificar a utilização e desenvolvimento de verificadores incompletos. Entretanto, é indiscutível que o preferível é ter um verificador com alta eficácia e eficiência.

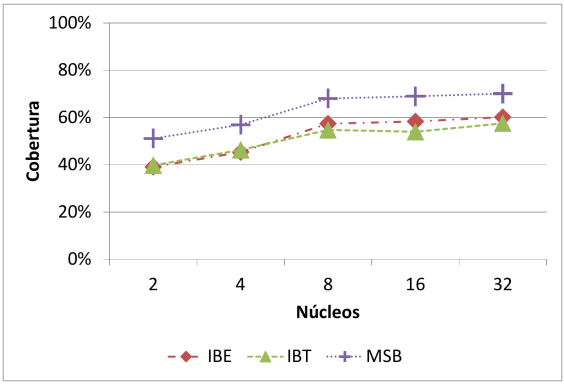
Analisando, agora, os gráficos sobre eficiência (Figura 5), note que, de novo, o MSB apresenta tanto o menor tempo de verificação quanto a menor taxa de crescimento (desta vez, pela quantidade de processadores). Para comparar a escalabilidade dos verificadores, os valores apresentados na Figura 5(c) foram interpolados para gerar uma equação da forma $t = k * (1 + r)^p$, relacionando o tempo total da simulação ($t$) com uma constante ($k$), a taxa de crescimento ($r$) e o número de processadores ($p$). O MSB ($r = 0.077$) obteve uma taxa de crescimento, aproximadamente, 3 vezes menor do que o IBE (0.214). Portanto, o uso pré-silício de um verificador *on-the-fly* (tal como MSB) provê plenas garantias a uma taxa de "escalabilidade" de, pelo menos, 3 vezes menor do que a de um verificador baseado em inferências sem garantias de verificação (como IBE).




(a)

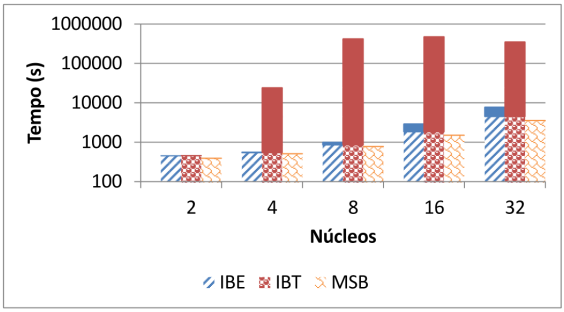



(b)

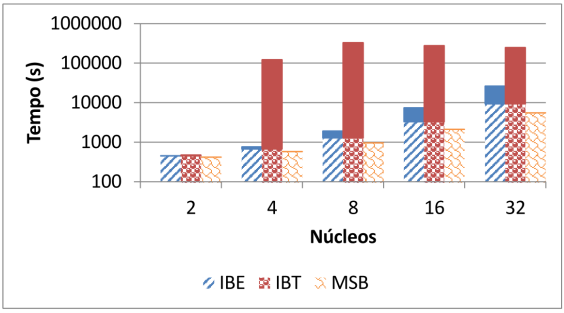



(c)

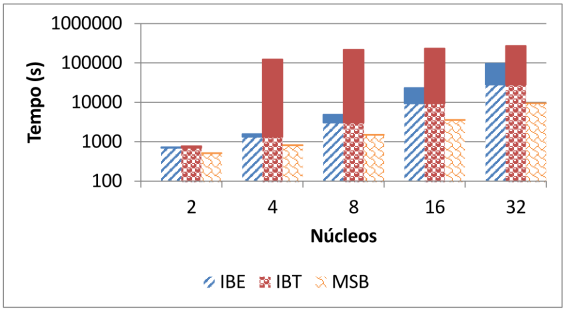
**Figura 4. Eficácia pelo número de núcleos de processamento**




(a)




(b)




(c)

**Figura 5. Eficiência pelo número de núcleos de processamento**

Como Henshcel já comentou em sua dissertação [Henschel 2014], os resultados de experimentos adicionais mostraram que o MSB possui uma taxa de crescimento ainda menor do que a calculada acima, tendo em vista que ele não se adaptou bem à equação exponencial utilizada [Henschel 2014, p. 60].

### 5.5. IMPACTO DO NÚMERO CRESCENTE DE ENDEREÇOS

O último experimento realizado serviu para avaliar o impacto do número de endereços compartilhados para a verificação de memória. Os resultados destes experimentos justificam por que falamos pouco sobre os resultados obtidos pelo IBT, apresentados nas seções anteriores. Ainda que tenhamos limitado estes experimentos a plataformas com quatro (4) processadores, tivemos que manter o limite de tempo (a verificação de qualquer experimento dura no máximo 2 horas) para o IBT encontrar os erros. Caso ele não diagnosticasse algum erro durante este período, o caso era interrompido e considerava-se que o IBT não indicou algum erro.

Na Figura 6, é possível averiguar que o IBT possui uma eficiência razoável quando o caso de teste possui um único endereço compartilhado: levemente superior a sua versão de melhor esforço (IBE). Entretanto, o seu tempo de verificação cresce exponencialmente conforme o número de endereços é dobrado, até saturar devido ao limite de tempo imposto aos experimentos. Comportamento condizente com a complexidade explicada na seção 4.

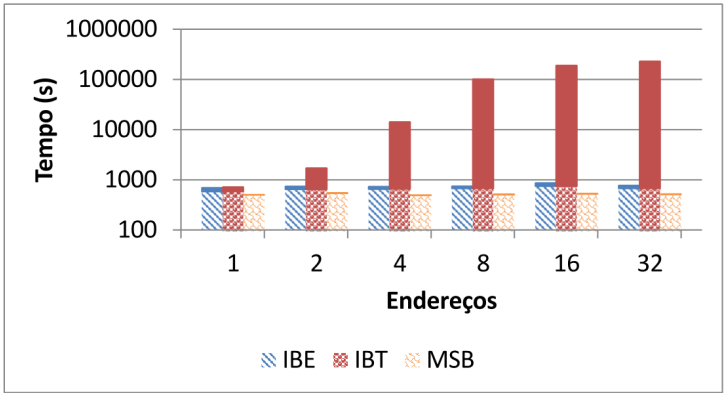



**Figura 6. Eficiência pelo número de endereços**

Esta eficiência que o IBT consegue quando trabalha com um único endereço compartilhado é efeito de que os programas comportam-se da mesma maneira caso um modelo de memória não tão relaxado estivesse sendo utilizado, tais como o SC, TSO e, possivelmente, o modelo do *Godson-3*, para os quais o XCHECK foi projetado e possui garantias formais. O leitor também pode entender este efeito através da expansão do conceito de fronteiras apresentado na seção 4. Quando há apenas um endereço compartilhado, as fronteiras ficam iguais às elaboradas originalmente em [Gibbons and Korach 1994].

Perceba que a complexidade indesejada do IBT é explicada, justamente, pelo uso do *backtracking* a fim de prover as mesmas garantias de verificação obtidas pelo MSB. Porém, apesar desta realização, a eficácia do MSB é superior ao IBT,

como também ilustra a Figura 7. A eficácia do IBT possui uma pequena margem de superioridade ao IBE. Logo, todos os resultados apresentados ao longo destas últimas seções indicam que, apesar do IBT realizar uma análise superior em três ordens de grandeza (com relação ao tempo médio), ele encontra erros em apenas 1.03 vezes mais casos de teste.

Portanto, estes experimentos demonstram que verificadores utilizando *backtracking*, incluindo os baseados em inferência (subclasse detentora da melhor eficiência), são inadequados para o uso em modelos de memória altamente relaxados, mesmo com as otimizações feitas (seção 4).

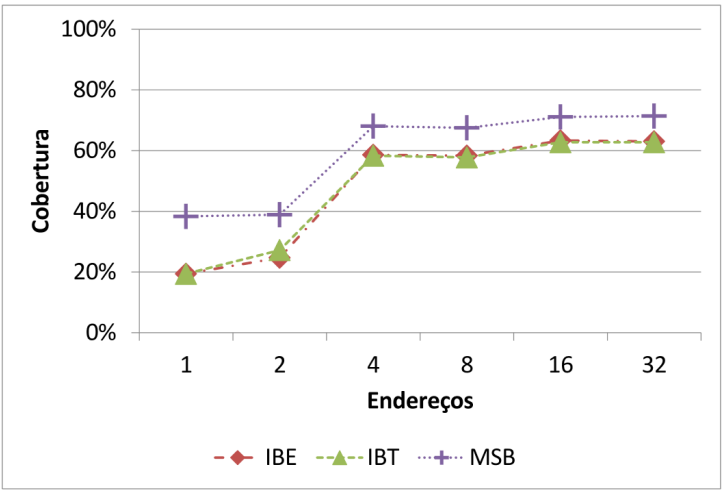



**Figura 7. Eficácia pelo número de endereços**

## 6. Conclusões

A implementação realizada do XCHECK auxiliou a constatação, por meio da comparação experimental, que as técnicas de verificação pré-silício são mais eficientes e eficazes do que técnicas pós-silício. Nestes experimentos, o papel do XCHECK foi de representar a classe de verificadores pós-silício por inferência, dos quais é o detentor da melhor complexidade de verificação (linear no número de processadores).

Esta conclusão foi obtida uma vez que os resultados experimentais (Seção 5) deste trabalho foram combinados com os obtidos em [Freitas et al. 2013]. Além do mais, o colega Henshcel reforçou esta conclusão por meio de sua dissertação de mestrado [Henschel 2014], onde foram refeitos os experimentos de Freitas com a inclusão das duas implementações do XCHECK – descritas no presente artigo.

Henshcel considera que as tendências mostram que as arquiteturas de processadores estão caminhando para a relaxação total da ordem das operações, mantendo apenas as ordens de programas que tenham sido impostas explicitamente. Esta re-

laxação, em conjunto com o aumento do número de núcleos, dificulta a verificação de memória [Henschel 2014, p. 83]. E, estas características afetam o desempenho da maioria dos verificadores pós-silício, como este trabalho demonstrou.

De acordo com as pesquisas presentes no estado da arte, é frequente a reutilização de verificadores pós-silício durante a etapa pré-silício. No entanto, os experimentos aqui realizados também comprovaram – assim como [Rambo et al. 2012, Freitas et al. 2013, Henschel 2014] – que tal atitude é inadequada já que a observabilidade disponibilizada nesta etapa permite a construção de verificadores mais eficientes e escaláveis.

Além do mais, estes verificadores pré-silício podem manter garantias plenas de verificação (e com uma mesma cobertura) enquanto são utilizados casos de teste menores. Fator decisivo na etapa pré-silício, devido à elevação existente no custo de execução dos casos de teste, com relação à etapa pós-silício que utiliza um protótipo do *hardware*.

## 7. Últimas considerações

Durante os últimos dois anos, o autor deste artigo e o, então, mestrando Olav Philipp Henschel tenham objetivado fazer uma implementação fiél à técnica proposta em [Hu et al. 2012], o XCHECK. No entanto, a escrita do presente artigo esclareceu que as dificuldades apresentadas (e.g. devido aos autores não terem apresentado o código da técnica ou não terem feitos experimentos com um *benchmark*) levaram a uma implementação incorreta, a qual utiliza o método de inferências durante o atravessamento no grafo de fronteiras.

Este engano afeta pouco as conclusões obtidas, uma vez que elas se baseiam, principalmente, na versão simplificada do XCHECK, a qual não utiliza-se deste atravessamento. O preocupante é se esta utilização incorreta do método de inferências seria a responsável pelas duas versões do XCHECK (IBE e IBT) terem apresentado os mesmos resultados.

Normalmente, seria recomendado que os experimentos fossem repetidos com uma versão correta do IBT, a fim de confirmar as conclusões. Porém, a situação presente possui uma outra alternativa: provar formalmente que o IBT nunca revelará um erro de projeto que o IBE não tenha detectado, isto é, comprovar que o atravessamento no grafo de fronteiras é desnecesário.

Na verdade esta comprovação pode ser reinterpretada como o tratamento de, somente, uma questão: sempre é possível achar uma ordenação total das operações, fiél ao modelo de memória adotado, uma vez que o grafo de execução não possui ciclos?

Uma vez que o IBE não detectou algum erro de projeto, é sabido que o grafo de execução não possui ciclos. Por outro lado, o atravessamento correto do grafo de fronteiras constitui em adicionar arestas de ordem entre todas as operações que já foram globalmente executadas (em movimento de fronteiras anteriores) com a operação que está sendo globalmente executada no movimento atual.

Agora, considere uma ordenação das operações de memória que não contradiga nenhuma das ordens presentes no grafo de execução. Primeiramente, é possível

estabelecer tal ordenação, já que se trata de uma ordenação topológica de um grafo acíclico. Em segundo perceba que o atravassamento do grafo de fronteiras, seguindo esta ordenação, não pode criar algum ciclo no grafo de execução, ou seja, que o IBT não detectará algum erro de projeto. O atravassamento não pode criar ciclos constituídos somente de arestas do grafo de execução (que é acíclico) ou das novas arestas adicionas (o que contradiria a própria definição destas últimas). Por fim, o fato do atravessamento seguir uma ordenação topológica combinado com a maneira que as ordens adicionais são estabelecidas implica a invalidade para a última hipótese de ciclos, isto é, aqueles que são oriundos da contradição entre as arestas adicionadas pelo atravessamento com as presente no grafo de execução.

O leitor pode, então, perceber como a utilização do método de inferências durante o atravessamento do grafo de fronteiras era importânte: ela incapacitava a conclusão de que a realização deste último é desnecessária.

Entretanto o leitor deve perceber que o raciocínio acima está, ainda, incompleto: o atravessamento de fronteiras de nossa implementação também adiciona arestas para compensar a omissão das operações de leitura. Assim, é necessário elaborar melhor este racioncínio em um prova formal.

## 8. Trabalhos Futuros

Trabalhos futuros podem buscar investigar melhorias no processo de verificação de modelos de memória durante a etapa pré-silício para aumentar todos os três fatores de interesse: a eficácia (i.e. a porcentagem de erros encontrados para um dado conjunto de programas concorrentes), a eficiência (i.e. diminuir o tempo de execução da ferramenta) e as próprias garantias de verificação (i.e. garantir que o verificador não emite diagnósticos falsos, positivos ou negativos).

Embora a eficácia dependa dos programas concorrentes utilizados para estimular o sistema de memória, a maioria dos verificadores limita-se à geração automática pseudo-aleatória de programas concorrentes [Rambo et al. 2012, Freitas et al. 2013, Henschel 2014]. A geração de programas concorrentes poderia ser dirigida por propriedades formalmente especificadas no modelo de memória. Isto propiciaria gerar estímulos relevantes, ou seja, evitando aqueles com menor potencial de revelar erros. Portanto, é possível e interessante investigar o uso de técnicas de verificação formal para se atingir este objetivo.

Alguns verificadores propostos na literatura [Rambo et al. 2012, Freitas et al. 2013, Henschel 2014] não exploraram melhorar a sua eficiência mesmo tendo componentes com potencial de concorrência. Ademais, é preciso investigar a viabilidade de se paralelizar um maior número de componentes destes verificadores.

Com relação à eficácia, muitos dos verificadores propostos na literatura [Rambo et al. 2012, Freitas et al. 2013, Shacham et al. 2008, Hu et al. 2012] aplicam-se a modelos de memória que fazem hipóteses simplificadoras sobre o comportamento das operações de escrita: supõe-se que as operações de escritas são atômicas, i.e. indivisíveis. Entretanto, com o aumento do número de processadores, a hipótese de atomicidade é cada vez menos aceitável e pode levar, por exemplo, a

diagnósticos de falso positivo. É preciso, então, avaliar de que forma os verificadores precisam ser modificados para se estenderem as garantias de verificação a modelos de memória com atomicidade relaxada.

Em suma, trabalhos futuros podem procurar criar: uma técnica semiformal para a geração de programas concorrentes, uma técnica para explorar a concorrência dos algoritmos de verificação e uma técnica de modelagem que leve em conta a relaxação da atomicidade de escrita em modelos de memória contemporâneos. Estas mesmas técnicas foram colocadas como potenciais contribuições do plano de mestrado do autor desta monografia.

## 9. Reconhecimentos

Por fim, agradeço a todos aqueles que contribuíram direta ou indiretamente para a realização deste trabalho. Em especial, agradeço aos meus colegas de pesquisa no laboratório ECL pelos seus trabalhos desenvolvidos (Tabelas 4, 5 e 6) e ao próprio orientador, Prof. Dr. Luiz Cláudio Villar dos Santos, pelo suporte a toda pesquisa realizada e pela bolsa de iniciação científica – sem a qual eu não estaria filiado ao laboratório e a este ramo de estudo.

| Verificador | Programadores | Referências |
|---|---|---|
| MSB | Leandro S. Freitas | [Freitas 2012]<br>[Freitas et al. 2013] |
| IBE e IBT | Gabriel A. G. Andrade<br>Olav P. Henschel | [Chen et al. 2009]<br>[Hu et al. 2012] |

**Tabela 4. Responsáveis pelas implementações dos verificadores utilizados**

| Atividade | Responsável |
|---|---|
| Implementação preliminar | Gabriel A. G. Andrade |
| Contribuição para a versão final | Gabriel A. G. Andrade |
| Implementação da versão final | Olav P. Henschel |

**Tabela 5. Contribuições para a implementação de IBE e IBT**

| Responsável | ID(s) |
|---|---|
| Leandro S. Freitas | *e1*, *e5* |
| Olav P. Henschel | *e2*, *e3*, *e6*, *e7*, *e8* |
| Gabriel A. G. Andrade | *e4* |

**Tabela 6. Os erros de projeto e os seus respectivos responsáveis**

## Referências

[Abts et al. 2003] Abts, D., Scott, S., and Lilja, D. (2003). So many states, so little time: verifying memory coherence in the Cray X1. *Proceedings International Parallel and Distributed Processing Symposium.*

[ARM 2012] ARM (2012). ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition.

[ARM 2013] ARM (2013). ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.

[Binkert et al. 2011] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

[Chatterjee et al. 2002] Chatterjee, P., Sivaraj, H., and Gopalakrishnan, G. (2002). Shared Memory Consistency Protocol Verification Against Weak Memory Models: Refinement via Model-Checking. In *Computer Aided Verification*, pages 121–138—-.

[Chen et al. 2009] Chen, Y., Lv, Y., Hu, W., Chen, T., Shen, H., Wang, P., and Pan, H. (2009). Fast complete memory consistency verification. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 381–392.

[Dubois et al. 1986] Dubois, M., Scheurich, C., and Briggs, F. (1986). Memory access buffering in multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 434–442. IEEE Computer Society Press.

[Freitas 2012] Freitas, L. S. (2012). Aceleradores e multiprocessadores em chip: o impacto da execuÃğÃčo fora de ordem na verificaÃğÃčo de funcionalidade e de consistÃłncia. Mestrado, UFSC, Universidade Federal de Santa Catarina.

[Freitas et al. 2013] Freitas, L. S., Rambo, E. A., and Santos, L. C. V. d. (2013). On-the-fly verification of memory consistency with concurrent relaxed scoreboards. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 631–636, San Jose, CA, USA. EDA Consortium.

[Gibbons and Korach 1994] Gibbons, P. B. and Korach, E. (1994). On testing cache-coherent shared memories. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 177–188, New York, NY, USA. ACM.

[Hangal et al. 2004] Hangal, S., Vahia, D., Manovit, C., and Lu, J.-Y. J. (2004). Tsotool: A program for verifying memory systems using the memory consistency model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 114–, Washington, DC, USA. IEEE Computer Society.

[Henschel 2014] Henschel, O. P. (2014). VerificaÃğÃčo de consistÃłncia e coerÃłncia de memÃşria compartilhada para multiprocessamento em chip. Master's thesis, Universidade Federal de Santa Catarina, FlorianÃşpolis, SC.

[Henzinger et al. 1999] Henzinger, T. A., Qadeer, S., and Rajamani, S. (1999). Verifying sequential consistency on shared-memory multiprocessor systems. Technical Report UCB/ERL M99/55, EECS Department, University of California, Berkeley.

[Hu et al. 2012] Hu, W., Chen, Y., Chen, T., Qian, C., and Li, L. (2012). Linear time memory consistency verification. *Computers, IEEE Transactions on*, 61(4):502–516.

[Inc" 2008] Inc", S. M. (2008). Ultrasparc architecture 2005.

http://www.oracle.com/technetwork/systems/opensparc/t1-06-ua2005-d0-9-2-p-ext-1537734.html.

[Lamport 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

[Lee and Seshia 2011] Lee, E. A. and Seshia, S. A. (2011). *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia.

[Lenoski et al. 1990] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. (1990). The directory-based cache coherence protocol for the DASH multiprocessor. *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*.

[Manovit 2005] Manovit, C. (2005). Efficient algorithms for verifying memory consistency. In *In SPAAÂ£05: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 245–252.

[Manovit and Hangal 2006] Manovit, C. and Hangal, S. (2006). Completely verifying memory consistency of test program executions. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 166–175.

[Rambo et al. 2012] Rambo, E., Henschel, O., and dos Santos, L. (2012). On esl verification of memory consistency for system-on-chip multiprocessing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 9–14.

[Rambo 2012] Rambo, E. A. (2012). VerificaÃğÃčo de consistÃłncia de memÃşria para sistemas integrados multiprocessados. Mestrado, UFSC, Universidade Federal de Santa Catarina.

[Rambo et al. 2011] Rambo, E. A., Henschel, O. P., and dos Santos, L. C. (2011). Automatic generation of memory consistency tests for chip multiprocessing. In *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*, pages 542–545.

[Roy et al. 2006] Roy, A., Zeisset, S., Fleckenstein, C. J., and Huang, J. C. (2006). Fast and generalized polynomial time memory consistency verification. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 503–516, Berlin, Heidelberg. Springer-Verlag.

[Shacham et al. 2008] Shacham, O., Wachs, M., Solomatnikov, A., Firoozshahian, A., Richardson, S., and Horowitz, M. (2008). Verification of chip multiprocessor memory systems using a relaxed scoreboard. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 294–305, Washington, DC, USA. IEEE Computer Society.

# APÊNDICE B – Código fonte do *XCHECK*

## B.1 DEBUG

– src/Debug.h

```
/*
 * Debug.h
 *
 *  Created on: Apr 23, 2014
 *      Author: olav e gabriel
 */

#ifndef DEBUG_H_
#define DEBUG_H_

#include <iostream>

namespace XCHECK {

class Debug : public std::ostream
{
public:
    static bool General;
    static bool Frontier;
    static bool FrontierMovement;
    static bool Backtrack;
    static bool Vertex;
    static bool Edge;
    static bool Cycle;

   Debug(bool enabled = false);

   template <class U> inline Debug & operator<<(const U data);

private:
  bool enabled;
};

template <class U> inline Debug & Debug::operator<<(const U data)
{
   if (enabled)
   {
      std::cout << data;
      std::cout.flush(); // TODO inefficient, endl should be supported
   }
   return *this;
}

} /* namespace XCHECK */

#endif /* DEBUG_H_ */
```

– src/Debug.cpp

```
/*
 * Debug.cpp
 *
 *  Created on: Apr 23, 2014
 *      Author: olav e gabriel
 */

#include "Debug.h"

namespace XCHECK {

bool Debug::General = false;
bool Debug::Frontier = false;
bool Debug::FrontierMovement = false;
bool Debug::Backtrack = false;
bool Debug::Vertex = false;
bool Debug::Edge = false;
bool Debug::Cycle = false;

Debug::Debug(bool enabled):
  enabled(enabled)
{
}

} /* namespace XCHECK */
```

## B.2 ATRAVESSAMENTOS NO GRAFO DE EXECUÇÃO

– src/DFS.h

```
/*
 * DFS.h
 *
 *  Created on: Apr 15, 2014
 *      Author: olav e gabriel
 */

#ifndef DFS_H_
#define DFS_H_

#include <set>

#include <boost/graph/depth_first_search.hpp>

#include "Types.h"
#include "Exceptions.h"

```

```
18 namespace XCHECK {
19
20   using std::set;
21
22   // Visitors
23   // TODO Verify none of the visitors redefine the INITIALIZE_VERTEX
24   struct VisitorWithRecovery : public boost::default_dfs_visitor {
25   public:
26     static ColorMap colorMap;
27     static std::vector<boost::default_color_type> color_map;
28
29     static void initializeColorMap(MemoryOrderGraph & graph);
30
31     virtual ~VisitorWithRecovery();
32
33     void discover_vertex(vertex_t v, const MemoryOrderGraph& g);
34     void reset();
35     set<vertex_t> memory;
36   };
37
38   struct PathFinder : public VisitorWithRecovery {
39     PathFinder(vertex_t target);
40     void discover_vertex(vertex_t v, const MemoryOrderGraph& g);
41   private:
42     vertex_t target;
43   };
44
45   struct CycleDetector : public VisitorWithRecovery {
46     using VisitorWithRecovery::discover_vertex;
47     void back_edge(edge_t e, const MemoryOrderGraph& g);
48   };
49
50   struct OperationRecorder : public VisitorWithRecovery {
51     void discover_vertex(vertex_t v, const MemoryOrderGraph& g);
52     set<vertex_t> operations;
53   };
54
55   struct StoreRecorder : public VisitorWithRecovery {
56     StoreRecorder(set<vertex_t> & stores);
57     virtual ~StoreRecorder() {};
58     void examine_edge(edge_t e, const MemoryOrderGraph& g);
59     void forward_or_cross_edge(edge_t e, const MemoryOrderGraph& g);
60     void finish_vertex(vertex_t v, const MemoryOrderGraph& g);
61   protected:
62     virtual bool continueSearch(const EdgeType type) const = 0;
63     virtual unsigned int & counter(Operation & o) const = 0;
64     virtual bool updateAndCheckCount(unsigned int & count) = 0;
65     virtual void printUpdate(const Operation & store) const = 0;
66     virtual void updateStoresSet(const vertex_t store) = 0;
67     set<vertex_t> & stores;
68   };
69
```

```
70  struct StoreCounterInPO : public StoreRecorder {
71    StoreCounterInPO(set<vertex_t> & stores) : StoreRecorder( stores ) {};
72  protected :
73    bool continueSearch(const EdgeType type) const;
74    unsigned int & counter(Operation & o) const;
75  };
76
77  struct StoreInserter : public StoreCounterInPO {
78     StoreInserter ( set<vertex_t> & stores) : StoreCounterInPO( stores ) {};
79  protected :
80    bool updateAndCheckCount(unsigned int & count);
81    void printUpdate(const Operation & store) const;
82    void updateStoresSet(const vertex_t store);
83  };
84
85  struct StoreRemover : public StoreCounterInPO {
86    StoreRemover(set<vertex_t> & stores) : StoreCounterInPO(stores ) {};
87  protected :
88    bool updateAndCheckCount(unsigned int & count);
89    void printUpdate(const Operation & store) const;
90    void updateStoresSet(const vertex_t store);
91  };
92
93  struct StoreCounterInGTO : public StoreRecorder {
94    StoreCounterInGTO() : StoreRecorder( stores ) {};
95  protected :
96    bool continueSearch(const EdgeType type) const;
97    unsigned int & counter(Operation & o) const;
98  private :
99     set<vertex_t> stores;
100 };
101
102 struct StoreDecrementer : public StoreCounterInGTO {
103 protected :
104   bool updateAndCheckCount(unsigned int & count);
105   void printUpdate(const Operation & store) const;
106   void updateStoresSet(const vertex_t store);
107 };
108
109 struct StoreIncrementer : public StoreCounterInGTO {
110 protected :
111   bool updateAndCheckCount(unsigned int & count);
112   void printUpdate(const Operation & store) const;
113   void updateStoresSet(const vertex_t store);
114 };
115
116 // Terminators
117
118 struct TerminateWithTimeOrder {
119   TerminateWithTimeOrder(vertex_t startVertex);
120   bool operator ()( vertex_t v, const MemoryOrderGraph & g) const;
121 private :
```

```
122       vertex_t    startVertex ;
123     };
124
125     struct  TerminateWithStore  {
126       TerminateWithStore( vertex_t    startVertex , bool  backtracking  =  false );
127        virtual  ~TerminateWithStore() {};
128       bool  operator ()( vertex_t  v,  const  MemoryOrderGraph& g) const;
129     protected :
130        virtual  const  unsigned  int  & counter(const  Operation  & o)  const  = 0;
131     private :
132       vertex_t    startVertex ;
133       bool  backtracking ;
134     };
135     struct  TerminateWithStoreInPO :  public  TerminateWithStore  {
136       TerminateWithStoreInPO( vertex_t    startVertex , bool  backtracking  =  false ) :
            TerminateWithStore( startVertex ,  backtracking ) {};
137     protected :
138       const  unsigned  int  & counter(const  Operation  & o)  const ;
139     };
140     struct  TerminateWithStoreInGTO : public  TerminateWithStore  {
141       TerminateWithStoreInGTO(vertex_t    startVertex , bool  backtracking  =  false ) :
            TerminateWithStore( startVertex ,  backtracking ) {};
142     protected :
143       const  unsigned  int  & counter(const  Operation  & o)  const ;
144     };
145
146     //  DFS
147
148     template  <class DFSVisitor,  class  Terminator>
149     void  depthFirstSearch (DFSVisitor & vis,  vertex_t    start_vertex ,  Terminator  &
            terminator ,  MemoryOrderGraph & graph)
150     {
151       vis . start_vertex  ( start_vertex ,  graph);
152       boost :: detail  ::   depth_first_visit_impl  (graph,   start_vertex ,  vis ,
            VisitorWithRecovery :: colorMap,  terminator );
153     }
154
155 } /* namespace XCHECK */
156
157 #endif  /* DFS_H_ */
```

– src/DFS.cpp

```
1  /*
2   * DFS.cpp
3   *
4   *  Created  on:  Apr 15,  2014
5   *        Author:  olav  e  gabriel
6   */
7
8  #include  "DFS.h"
9  #include  "Debug.h"
```

```
10
11 namespace XCHECK {
12
13 std :: vector<boost:: default_color_type > VisitorWithRecovery::color_map;
14 ColorMap VisitorWithRecovery::colorMap;
15
16 void VisitorWithRecovery :: initializeColorMap (MemoryOrderGraph & graph)
17 {
18   color_map. resize (boost :: num_vertices(graph));
19   colorMap = boost :: make_iterator_property_map (
20         color_map.begin(),
21         boost :: get(boost :: vertex_index , graph),
22         color_map[0]);
23
24   MemoryOrderGraph:: vertex_iterator ui, ui_end;
25   for (boost :: tie(ui, ui_end) = vertices(graph); ui != ui_end; ++ui) {
26     vertex_t u = boost :: implicit_cast <vertex_t>(*ui);
27     put(colorMap, u, Color :: white());
28   }
29 }
30
31 VisitorWithRecovery ::~ VisitorWithRecovery()
32 {
33   reset();
34 }
35
36 void VisitorWithRecovery :: discover_vertex ( vertex_t v, const MemoryOrderGraph& g)
37 {
38   memory.insert(v);
39 }
40
41 void VisitorWithRecovery :: reset ()
42 {
43   for( vertex_t executed : memory)
44   {
45     boost :: put(colorMap, executed, Color :: white());
46   }
47 }
48
49 PathFinder :: PathFinder( vertex_t target ) :
50       target ( target )
51 {
52 }
53
54 void PathFinder :: discover_vertex ( vertex_t v, const MemoryOrderGraph& g)
55 {
56   VisitorWithRecovery :: discover_vertex (v,g);
57
58   if (v == target )
59     throw PathFound();
60 }
61
```

```
62 void CycleDetector::back_edge(edge_t e, const MemoryOrderGraph& g)
63 {
64   Debug(Debug::Cycle) << "Cycle found in pending period" << "\n";
65   throw CycleFoundInPendingPeriod();
66 }
67
68 TerminateWithTimeOrder::TerminateWithTimeOrder(vertex_t startVertex) : startVertex(
       startVertex)
69 {
70 };
71
72 bool TerminateWithTimeOrder::operator()(vertex_t v, const MemoryOrderGraph & g)
       const
73 {
74   const Operation & start = g[startVertex];
75   const Operation & o = g[v];
76
77   if (start.timeInterval.isBefore(o.timeInterval))
78     return true; // Edge complies with time order
79   else if (o.timeInterval.isBefore(start.timeInterval))
80   {
81     Debug(Debug::Cycle) << "Conflicting time order found, operation starts before "
        << start.timeInterval.start << "\n";
82     throw CycleFoundInvolvingTimeOrder();
83   }
84
85   return false;
86 }
87
88 void OperationRecorder::discover_vertex(vertex_t v, const MemoryOrderGraph& g)
89 {
90   VisitorWithRecovery::discover_vertex(v,g);
91
92   operations.insert(v);
93 }
94
95 StoreRecorder::StoreRecorder(set<vertex_t> & stores) :
96     stores(stores)
97 {
98 }
99
100 void StoreRecorder::examine_edge(edge_t e, const MemoryOrderGraph& g)
101 {
102   const EdgeType type = g[e];
103   const vertex_t v = boost::target(e, g);
104   Operation & o = const_cast<Operation &>(g[v]);
105
106   if (continueSearch(type))
107   {
108     if (updateAndCheckCount(counter(o)) && (o.is_write() || o.type == Operation::
        ONULL || o.type == Operation::HALT))
109     {
```

```
110         printUpdate(o);
111
112         updateStoresSet(v);
113       }
114     }
115     else
116     {
117       VisitorWithRecovery::discover_vertex(v,g);
118       boost::put(colorMap, v, Color::black());
119     }
120 }
121
122 void StoreRecorder::forward_or_cross_edge(edge_t e, const MemoryOrderGraph& g)
123 {
124   const vertex_t v = boost::target(e, g);
125   Operation & o = const_cast<Operation &>(g[v]);
126
127   // Put white color so vertex can be checked again if reached from another branch
128   boost::put(colorMap, v, Color::white());
129 }
130
131 void StoreRecorder::finish_vertex( vertex_t v, const MemoryOrderGraph& g)
132 {
133   // Put white color so vertex can be checked again if reached from another branch
134   boost::put(colorMap, v, Color::white());
135 }
136
137 bool StoreCounterInPO::continueSearch(EdgeType type) const
138 {
139   return type == PROCESSOR;
140 }
141
142 unsigned int & StoreCounterInPO::counter(Operation & o) const
143 {
144   return o.predecessorsCountInPO;
145 }
146
147 bool StoreInserter::updateAndCheckCount(unsigned int & count)
148 {
149   assert(count > 0);
150   return --count == 0;
151 }
152
153 void StoreInserter::printUpdate(const Operation & store) const
154 {
155   Debug(Debug::FrontierMovement) << "Adding candidate operation: " << store.toString
        () << "\n";
156 }
157
158 void StoreInserter::updateStoresSet(const vertex_t store)
159 {
160   stores.insert(store);
```

```
161 }
162
163 bool StoreRemover::updateAndCheckCount(unsigned int & count)
164 {
165   return ++count == 1;
166 }
167
168 void StoreRemover::printUpdate(const Operation & store) const
169 {
170   Debug(Debug::FrontierMovement) << "Removing candidate operation: " << store.
        toString() << "\n";
171 }
172
173 void StoreRemover::updateStoresSet(const vertex_t store)
174 {
175   stores.erase(store);
176 }
177
178 bool StoreCounterInGTO::continueSearch(EdgeType type) const
179 {
180   return type != INFERRED_IN_BACKTRACK;
181 }
182
183 unsigned int & StoreCounterInGTO::counter(Operation & o) const
184 {
185   return o.predecessorsCountInGTO;
186 }
187
188 bool StoreDecrementer::updateAndCheckCount(unsigned int & count)
189 {
190   assert(count > 0);
191   return --count == 0;
192 }
193
194 void StoreDecrementer::printUpdate(const Operation & store) const
195 {
196   Debug(Debug::FrontierMovement) << "Operation can now be executed: " << store.
        toString() << "\n";
197 }
198
199 void StoreDecrementer::updateStoresSet(const vertex_t store)
200 {
201 }
202
203 bool StoreIncrementer::updateAndCheckCount(unsigned int & count)
204 {
205   return ++count == 1;
206 }
207
208 void StoreIncrementer::printUpdate(const Operation & store) const
209 {
210   Debug(Debug::FrontierMovement) << "Operation can no longer be executed: " << store.
```

```
        toString () << "\n";
}

void StoreIncrementer :: updateStoresSet (const vertex_t store )
{
}

TerminateWithStore :: TerminateWithStore( vertex_t startVertex , bool backtracking ):
      startVertex ( startVertex ),
      backtracking ( backtracking )
{
};

bool TerminateWithStore :: operator () ( vertex_t v, const MemoryOrderGraph& g) const
{
  const Operation & o = g[v];

  if (counter(o) > (backtracking ? 1 : 0))
    return true;

  if ((v != startVertex ) && (o.is_write () || o.type == Operation :: ONULL || o.type ==
      Operation :: HALT))
    return true;

  return false;
}

const unsigned int & TerminateWithStoreInPO::counter(const Operation & o) const
{
  return o.predecessorsCountInPO;
}

const unsigned int & TerminateWithStoreInGTO::counter(const Operation & o) const
{
  return o.predecessorsCountInGTO;
}

} /* namespace XCHECK */
```

## B.3 EXCESSÕES

– src/Exceptions.h

```
/*
 * Exceptions.h
 *
 *  Created on: Apr 15, 2014
 *      Author: olav e gabriel
 */
```

```
7
8  #ifndef EXCEPTIONS_H_
9  #define EXCEPTIONS_H_
10
11 #include <iostream>
12
13 #include "Operation.h"
14 #include "Debug.h"
15
16 namespace XCHECK {
17
18   // Errors
19
20   struct FailedToOpenFile {};
21
22   struct ValueError : std::exception {
23     const char * what() {
24       return "Value error";
25     }
26   };
27
28   struct InvalidValueForLoad : ValueError {
29     const char * what() {
30       return "Invalid value for load";
31     }
32   };
33
34   struct LoadMappedToStoreOfDifferentAddress : ValueError {
35     const char * what() {
36       return "Load mapped to store of different address";
37     }
38   };
39
40   struct CycleFound : std::exception {
41     const char * what() {
42       return "Cycle found in pending period";
43     }};
44
45   struct CycleFoundInPendingPeriod : CycleFound {
46     CycleFoundInPendingPeriod() {
47       Debug(Debug::Cycle) << what() << "\n";
48     }
49     const char * what() {
50       return "Cycle found";
51     }
52   };
53
54   struct CycleFoundInvolvingTimeOrder : CycleFound {
55     CycleFoundInvolvingTimeOrder() {
56       Debug(Debug::Cycle) << what() << "\n";
57     }
58     const char * what() {
```

```
59        return "Cycle found involving time order";
60      }
61    };
62
63    struct CycleFoundWhileInferringEdges : CycleFound {
64      CycleFoundWhileInferringEdges(const Operation & source, const Operation & read,
          const Operation & conflictingWrite, bool conflictingIsBefore) :
65        source(source), read(read), conflictingWrite(conflictingWrite),
          conflictingIsBefore(conflictingIsBefore)
66      {
67        Debug(Debug::Cycle) << what();
68      }
69      const char * what() {
70        std::ostringstream oss;
71        oss << "Cycle found while inferring edges" << "\n";
72        oss << static_cast<const Operation &>(source).toString() << " -E-> "
73            << static_cast<const Operation &>(read).toString() << "\n";
74        if (conflictingIsBefore)
75        {
76          oss << static_cast<const Operation &>(conflictingWrite).toString() << " -
            GO-> "
77              << static_cast<const Operation &>(read).toString() << "\n";
78          oss << static_cast<const Operation &>(source).toString() << " -T-> "
79              << static_cast<const Operation &>(conflictingWrite).toString() << "\n";
80        }
81        else
82        {
83          oss << static_cast<const Operation &>(source).toString() << " -GO-> "
84              << static_cast<const Operation &>(conflictingWrite).toString() << "\n";
85          oss << static_cast<const Operation &>(conflictingWrite).toString() << " -T
            -> "
86              << static_cast<const Operation &>(read).toString() << "\n";
87        }
88        return oss.str().c_str();
89      }
90    private:
91      const Operation & source;
92      const Operation & read;
93      const Operation & conflictingWrite;
94      const bool conflictingIsBefore;
95    };
96
97    struct NoPathFoundToFinalFrontier : std::exception {
98      const char * what() {
99        return "No path found to final frontier";
100     }
101   };
102
103   // Other exceptions
104
105   struct PathFound {};
106
```

```
107 } /* namespace XCHECK */
108
109
110 #endif /* EXCEPTIONS_H_ */
```

## B.4 *PARSER* DOS *TRACES*

– src/FileParser.h

```
1  /*
2   * FileParser.h
3   *
4   *  Created on: Jul 5, 2013
5   *      Author: olav e gabriel
6   */
7
8  #ifndef FILEPARSER_H_
9  #define FILEPARSER_H_
10
11 #include <string>
12 #include <fstream>
13 #include "Operation.h"
14
15 using std::string;
16 using std::ifstream;
17
18 namespace XCHECK
19 {
20
21 class FileParser {
22 public:
23    FileParser(const string filename, unsigned int core);
24    virtual ~FileParser();
25
26   void nextOperation(Operation& next);
27   bool hasNextOperation();
28
29 private:
30    ifstream file;
31    string nextLine;
32   unsigned int core;
33 };
34
35 }
36
37 #endif /* FILEPARSER_H_ */
```

– src/FileParser.cpp

```
/*
 * FileParser.cpp
 *
 *  Created on: Jul 5, 2013
 *      Author: olav e gabriel
 *  Modified by: Gabriel Arthur at Aug 6, 2013
 */

#include "FileParser.h"
#include <stdexcept>
#include <sstream>

namespace XCHECK
{

FileParser::FileParser(const string filename, unsigned int core):
    core(core)
{
  file.open(filename.c_str());
  if (!file.is_open())
  {
    throw std::runtime_error("Couldn't open the file: " + filename);
  }
}

FileParser::~FileParser()
{
  file.close();
}

void FileParser::nextOperation(Operation& operation)
{
  string s;
  std::istringstream iss(nextLine);
  Operation::Type type;
  Operation::Addr address;
  uint64_t data;

  if (nextLine[0] == 'R')
  {
    iss >> s;
    iss >> std::hex >> address;
    iss >> std::hex >> data;
    if (nextLine[1] == 'S')
    {
      type = Operation::READ_SWAP;
    }
    else
    {
      type = Operation::READ;
```

```
51      }
52    }
53    else if (nextLine[0] == 'W')
54    {
55      iss >> s;
56      iss >> std::hex >> address;
57      iss >> std::hex >> data;
58      if (nextLine[1] == 'S')
59      {
60        type = Operation::WRITE_SWAP;
61      }
62      else
63      {
64        type = Operation::WRITE;
65      }
66    }
67    else if (nextLine[0] == 'M')
68    {
69      iss >> s;
70      iss >> std::hex >> address;
71      iss >> std::hex >> data;
72      type = Operation::MEMBAR;
73    }
74    else if (nextLine[0] == 'B')
75    {
76      iss >> s;
77      iss >> std::hex >> address;
78      iss >> std::hex >> data;
79      type = Operation::WBAR;
80    }
81    else
82    {
83      throw std::runtime_error("Unexpected character read! Line: \"" + nextLine + "\".")
        ;
84    }
85
86    operation.id_processor      = core;
87    operation.type      = type;
88    operation.address   = address;
89    operation.data      = data;
90    iss >> std::dec >> operation.instID;
91    iss >> std::dec >> operation.timeInterval.start;
92    iss >> std::dec >> operation.timeInterval.end;
93  }
94
95  bool FileParser::hasNextOperation(){
96    // Prop. File read have a line to identified its end
97    do
98    {
99      getline(file, nextLine);
100   }
101   while (((nextLine.size() == 0) || (nextLine[0] == '#') || ((nextLine[0] == 'R')
```

```
        && (nextLine[1] == 'F'))) && !file.eof());
102   return !file.eof();
103 }
104
105 }
```

## B.5 MAIN

– src/Main.cpp

```
1  //==================================
2  // Name        : XCHECK.cpp
3  // Author      : olav e gabriel
4  // Version     :
5  // Copyright   : Your copyright notice
6  //==================================
7
8  #include <iostream>
9  #include <vector>
10 #include <string>
11 #include <algorithm>
12 #include "XCHECK.h"
13 #include "XCHECKBacktracking.h"
14 #include <tclap/CmdLine.h>
15
16 #include "Debug.h"
17
18 using namespace std;
19 using namespace XCHECK;
20
21 struct InvalidArgument{};
22
23 enum Model {ALPHA};
24
25 struct manipulation_data
26 {
27   // Parameters needed by the verifier
28   Model model;
29   vector<string> programOrder;
30   string pendingPeriods;
31   string timeOrders;
32   bool generate;
33   bool backtracking;
34   vector<string> debug;
35 };
36
37 manipulation_data process_cmd_line(int argc, char** argv)
38 {
39   manipulation_data md;
```

```
40    try
41    {
42      TCLAP::CmdLine cmd("XCHECKER", ' ', " ");
43      TCLAP::ValueArg<int>    model           ("m", "memory_consistency_model", "Code of
          the MCM had been used at the test" , false ,0," integer ");
44      TCLAP::MultiArg<string> programOrder  ("p", "program_order"              , "Program
          order files "                        , true ," file ");
45      TCLAP::ValueArg<string> pendingPeriods ("e", "pending_periods"       , "File with
          the pending periods"                 , true ,"","file ");
46      TCLAP::ValueArg<string> timeOrders    ("t", "time_orders"            , "File with the
          time orders"                         , true ,"","file ");
47      TCLAP::SwitchArg        generate        ("g", "generate_mode"              , "Generate
            the pending periods"               , false );
48      TCLAP::SwitchArg        backtracking    ("b", "no_backtracking"            , "Disables
            backtracking"                      , true );
49      TCLAP::MultiArg<string> debug           ("d", "debug_flags"                , "Output
          selected debugs"                     , false ," string ");
50
51      cmd.add(model);
52      cmd.add(programOrder);
53      cmd.add(pendingPeriods);
54      cmd.add(timeOrders);
55      cmd.add(generate);
56      cmd.add(backtracking);
57      cmd.add(debug);
58      cmd.parse(argc, argv);
59
60      md.model = static_cast <Model>(model.getValue());
61      md.programOrder = programOrder.getValue();
62      md.pendingPeriods = pendingPeriods.getValue();
63      md.timeOrders = timeOrders.getValue();
64      md.generate = generate.getValue();
65      md.backtracking = backtracking.getValue();
66      md.debug = debug.getValue();
67    }
68    catch (TCLAP::ArgException &e)
69    {
70      throw InvalidArgument();
71    }
72
73    return md;
74  }
75
76  int main(int argc, char** argv)
77  {
78    manipulation_data params = process_cmd_line(argc, argv);
79
80    if (std :: find(params.debug.begin(), params.debug.end(), "general") != params.debug.
          end())
81      Debug::General = true;
82    if (std :: find(params.debug.begin(), params.debug.end(), " frontier ") != params.
          debug.end())
```

```
83      Debug::Frontier = true;
84    if (std::find(params.debug.begin(), params.debug.end(), "frontier_movement") !=
          params.debug.end())
85      Debug::FrontierMovement = true;
86    if (std::find(params.debug.begin(), params.debug.end(), "backtrack") != params.
          debug.end())
87      Debug::Backtrack = true;
88    if (std::find(params.debug.begin(), params.debug.end(), "vertex") != params.debug.
          end())
89      Debug::Vertex = true;
90    if (std::find(params.debug.begin(), params.debug.end(), "edge") != params.debug.end
          ())
91      Debug::Edge = true;
92    if (std::find(params.debug.begin(), params.debug.end(), "cycle") != params.debug.
          end())
93      Debug::Cycle = true;
94
95    XCHECK::XCHECK * checker;
96    if (params.backtracking)
97      checker = new XCHECK::XCHECK_Backtracking(params.programOrder, params.
            pendingPeriods, params.timeOrders);
98    else
99      checker = new XCHECK::XCHECK(params.programOrder, params.pendingPeriods,
            params.timeOrders);
100
101   if (params.generate)
102     checker->generate();
103   else
104     checker->analyse();
105
106   return EXIT_SUCCESS;
107 }
```

## B.6 OPERAÇÕES DE MEMÓRIA

– src/Operation.h

```
1  /*
2   * Operation.h
3   *
4   *  Created on: Jul 17, 2013
5   *      Author: olav e gabriel
6   */
7
8  #ifndef OPERATION_H_
9  #define OPERATION_H_
10
11 #include <stdint.h>
12 #include <iostream>
```

```
13 #include <sstream>
14 #include <iomanip>
15 #include <string>
16
17 #include "TimeInterval.h"
18
19 namespace XCHECK
20 {
21
22 class Operation
23 {
24 public:
25   typedef unsigned int Processor;
26   typedef uint64_t Addr;
27
28   enum Type {ONULL, HALT, READ, READ_FORWARD, READ_SWAP, WRITE,
         WRITE_SWAP, MEMBAR, WBAR};
29
30   Operation(Processor id_processor = 0, Type type = ONULL, Addr address = 0, uint64_t
         data = 0, uint64_t instID = 0, TimeInterval timeInterval = TimeInterval(0, 0))
         ;
31   Operation(const Operation & other);
32   Operation& operator=(const Operation& another);
33    virtual ~Operation();
34   bool is_write () const;
35   bool is_read () const;
36   bool is_membar() const;
37   bool conflicts_with (Operation v) const;
38
39   // Print methods
40   std::string toString () const;
41   void print();
42
43   // Basic information
44   Processor id_processor;
45   Type type;
46   Addr address;
47    uint64_t data;
48    uint64_t instID;
49    TimeInterval timeInterval;
50
51   // Auxiliary information
52   unsigned int predecessorsCountInPO;
53   unsigned int predecessorsCountInGTO;
54 };
55
56 }
57
58 #endif /* OPERATION_H_ */
```

– src/Operation.cpp

```
/*
 * Operation.cpp
 *
 *  Created on: Jul 17, 2013
 *      Author: olav e gabriel
 */

#include "Operation.h"
#include <iostream>
#include <iomanip>

namespace XCHECK
{

Operation::Operation(Processor id_processor, Type type, Addr address, uint64_t data,
        uint64_t instID, TimeInterval timeInterval):
    id_processor(id_processor),
    type(type),
    address(address),
    data(data),
    instID(instID),
    timeInterval(timeInterval),
    predecessorsCountInPO(0),
    predecessorsCountInGTO(0)
{
}

Operation::Operation(const Operation &other):
    id_processor(other.id_processor),
    type(other.type),
    address(other.address),
    data(other.data),
    instID(other.instID),
    timeInterval(other.timeInterval),
    predecessorsCountInPO(other.predecessorsCountInPO),
    predecessorsCountInGTO(other.predecessorsCountInGTO)
{
}

Operation& Operation::operator=(const Operation& another)
{
    id_processor = another.id_processor;
    type = another.type;
    address = another.address;
    data = another.data;
    instID = another.instID;
    timeInterval = another.timeInterval;
    predecessorsCountInPO = another.predecessorsCountInPO;
    predecessorsCountInGTO = another.predecessorsCountInGTO;
    return *this;
```

```
50 }
51
52 Operation ::~ Operation()
53 {
54 }
55
56 bool Operation:: is_write () const
57 {
58   /****************
59    It returns True iff this operation is a write operation. ******************/
60    return type == WRITE || type == WRITE_SWAP;
61 }
62
63 bool Operation:: is_read () const
64 {
65   /****************
66    It returns True iff this operation is a read operation. ******************/
67    return type == READ || type == READ_FORWARD || type == READ_SWAP;
68 }
69
70 bool Operation:: is_membar() const
71 {
72   /****************
73      It returns True iff this operation is a membar operation. ****************/
74    return type == MEMBAR || type == WBAR || type == ONULL || type == HALT;
75 }
76
77 bool Operation:: conflicts_with (Operation v) const
78 {
79   bool conflict;
80
81    if (is_read()){    // A read operation conflicts with
82      // A write operation in the same address
83      // A membar operation
84      conflict = (address == v.address && v.is_write()) || v.is_membar();
85    }
86    else if(is_write()){  // A write operation conflicts with
87      // A read operation in the same address
88      // A write operation in the same address
89      // A membar operation
90      conflict = (address == v.address && (v.is_read() || v.is_write()))
91                  || v.is_membar();
92    }
93    else if(is_membar()){ // A membar operation conflicts with all operations
94      conflict = true;
95    }
96    return conflict;
97 }
98
99 std:: string Operation:: toString () const
100 {
101   std:: stringstream oss;
```

```
102     oss << "Core: 0x" << std::dec << id_processor;
103     oss << "\tType: ";
104     switch(type){
105     case ONULL:
106       oss << "ONULL";
107       break;
108     case READ:
109       oss << "R ";
110       break;
111     case READ_FORWARD:
112       oss << "RF";
113       break;
114     case READ_SWAP:
115       oss << "RS";
116       break;
117     case WRITE:
118       oss << "W ";
119       break;
120     case WRITE_SWAP:
121       oss << "WS";
122       break;
123     case MEMBAR:
124       oss << "MB";
125       break;
126     case WBAR:
127       oss << "WB";
128       break;
129     case HALT:
130       oss << "HALT";
131       break;
132     }
133     oss << "\tAddr: 0x" << std::setw(sizeof(address)*2) << std:: setfill ('0') << std::
          hex << address;
134     oss << "\tData: 0x" << std::setw( sizeof (data)*2) << std:: setfill ('0') << std::hex
          << data;
135     oss << "\tID: " << std::dec << instID;
136     oss << "\tStart: " << std::dec << timeInterval. start ;
137     oss << "\tEnd: " << std::dec << timeInterval.end;
138     return oss. str ();
139 }
140
141 void Operation :: print ()
142 {
143   std :: cout << toString() << std::endl;
144 }
145
146 }
```

## B.7 PREFIXOS

– src/Prefix.h

```
/*
 * Prefix.h
 *
 *  Created on: Jul 8, 2014
 *      Author: olav e gabriel
 */

#ifndef PREFIX_H_
#define PREFIX_H_

#include <map>
#include <vector>

#include "Types.h"

namespace XCHECK {

  using std::map;
  using std::pair;
  using std::vector;

  class Prefix
  {
  public:
    static vertex_t nullVertex;

    typedef pair<Operation::Processor, Operation::Addr> key_type;
    typedef uint index;

    Prefix();
    Prefix(const Prefix & other);
    ~Prefix();

    Prefix & operator=(const Prefix & other);
    vertex_t & operator[](const key_type & key);

    friend bool operator==(const Prefix & lhs, const Prefix & rhs);
    friend bool operator!=(const Prefix & lhs, const Prefix & rhs);
    friend bool operator<(const Prefix & lhs, const Prefix & rhs);
    friend bool operator<=(const Prefix & lhs, const Prefix & rhs);
    friend bool operator>(const Prefix & lhs, const Prefix & rhs);
    friend bool operator>=(const Prefix & lhs, const Prefix & rhs);
  private:
    vector<vertex_t> lastVertices;
    static map<key_type, index> indexMap;
    static index nextIndex;
  };
```

```
48
49 } /* namespace XCHECK */
50
51 #endif /* PREFIX_H_ */
```

– src/Prefix.cpp

```
1  /*
2   * Prefix.cpp
3   *
4   *  Created on: Jul 8, 2014
5   *      Author: olav e gabriel
6   */
7
8  #include "Prefix.h"
9
10 namespace XCHECK {
11
12  vertex_t Prefix::nullVertex;
13 map<Prefix::key_type, Prefix::index> Prefix::indexMap;
14  Prefix::index Prefix::nextIndex = 0;
15
16  Prefix::Prefix() :
17     lastVertices(nextIndex)
18  {
19  }
20
21  Prefix::Prefix(const Prefix &other) :
22     lastVertices(other.lastVertices)
23  {
24  }
25
26  Prefix::~Prefix()
27  {
28  }
29
30  Prefix &Prefix::operator=(const Prefix &other)
31  {
32     lastVertices = other.lastVertices;
33    return *this;
34  }
35  vertex_t &Prefix::operator[](const key_type &key)
36  {
37    if (indexMap.count(key) == 0)
38      indexMap.insert(std::make_pair(key, nextIndex++));
39
40     lastVertices.resize(nextIndex, nullVertex);
41    return lastVertices.at(indexMap.at(key));
42  }
43
44 bool operator==(const Prefix &lhs, const Prefix &rhs)
45 {
```

```
46    const_cast<Prefix &>(lhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
47    const_cast<Prefix &>(rhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
48    return lhs.lastVertices == rhs.lastVertices;
49  }
50  bool operator!=(const Prefix & lhs, const Prefix & rhs)
51  {
52    const_cast<Prefix &>(lhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
53    const_cast<Prefix &>(rhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
54    return lhs.lastVertices != rhs.lastVertices;
55  }
56  bool operator<(const Prefix & lhs, const Prefix & rhs)
57  {
58    const_cast<Prefix &>(lhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
59    const_cast<Prefix &>(rhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
60    return lhs.lastVertices < rhs.lastVertices;
61  }
62  bool operator<=(const Prefix & lhs, const Prefix & rhs)
63  {
64    const_cast<Prefix &>(lhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
65    const_cast<Prefix &>(rhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
66    return lhs.lastVertices <= rhs.lastVertices;
67  }
68  bool operator>(const Prefix & lhs, const Prefix & rhs)
69  {
70    const_cast<Prefix &>(lhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
71    const_cast<Prefix &>(rhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
72    return lhs.lastVertices > rhs.lastVertices;
73  }
74  bool operator>=(const Prefix & lhs, const Prefix & rhs)
75  {
76    const_cast<Prefix &>(lhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
77    const_cast<Prefix &>(rhs).lastVertices.resize(Prefix::nextIndex, Prefix::nullVertex
          );
78    return lhs.lastVertices >= rhs.lastVertices;
79  }
80
81  } /* namespace XCHECK */
```

## B.8 CRONOMETRO

– src/TimeAccumulator.h

```
/*
 * TimeAccumulator.h
 *
 *  Created on: May 8, 2014
 *      Author: olav e gabriel
 */

#ifndef TIMEACCUMULATOR_H_
#define TIMEACCUMULATOR_H_

#include <chrono>

namespace TimeAccumulator
{
  typedef std::chrono::system_clock::time_point TimePoint;
  typedef std::chrono::duration<double> Duration;

  class TimeAccumulator
  {
  public:
    TimeAccumulator();
     virtual ~TimeAccumulator();

    void start();
    void pause();
    Duration getAccumulatedValue();
    void saveTimeFile(const std::string & timeFileName);

  private:
    TimePoint startTime;
    Duration value;
  };

} // namespace TimeAccumulator

#endif /* TIMEACCUMULATOR_H_ */
```

– src/TimeAccumulator.cpp

```
/*
 * TimeAccumulator.cpp
 *
 *  Created on: May 8, 2014
 *      Author: olav e gabriel
 */

```

```
#include <fstream>

#include "TimeAccumulator.h"

namespace TimeAccumulator
{

  TimeAccumulator::TimeAccumulator()
  {
  }

  TimeAccumulator::~TimeAccumulator()
  {
  }

  void TimeAccumulator::start()
  {
    startTime = std::chrono::system_clock::now();
  }

  void TimeAccumulator::pause()
  {
    TimePoint endTime = std::chrono::system_clock::now();
    value += endTime - startTime;
  }

  Duration TimeAccumulator::getAccumulatedValue()
  {
    return value;
  }

  void TimeAccumulator::saveTimeFile(const std::string & timeFileName)
  {
    std::ofstream timeFile;
    timeFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
    timeFile.open(timeFileName.c_str());
    timeFile << value.count();
    timeFile.close();
  }

} // namespace TimeAccumulator
```

## B.9 INTERVALO DE TEMPO

– src/TimeInterval.h

```
/*
 * TimeInterval.h
 *
```

```
 *  Created on: Aug 22, 2013
 *      Author: olav e gabriel
 */

#ifndef TIMEINTERVAL_H_
#define TIMEINTERVAL_H_

#include <stdint.h>

class TimeInterval {
public:
  typedef uint64_t Tick;

  TimeInterval(Tick startTime = 0, Tick endTime = 0);
  TimeInterval(const TimeInterval & other);
  TimeInterval & operator=(const TimeInterval & other);
  virtual ~TimeInterval();

  bool isBefore(const TimeInterval & other) const;
  bool isBefore(const Tick & time) const;
  bool overlaps(const TimeInterval & other) const;

  Tick start;
  Tick end;
};

#endif /* TIMEINTERVAL_H_ */
```

– src/TimeInterval.cpp

```
/*
 * TimeInterval.cpp
 *
 *  Created on: Aug 22, 2013
 *      Author: olav e gabriel
 */

#include "TimeInterval.h"

TimeInterval::TimeInterval(Tick startTime, Tick endTime):
    start(startTime),
   end(endTime)
{
}

TimeInterval::TimeInterval(const TimeInterval & other):
      start(other.start),
     end(other.end)
{
}

TimeInterval & TimeInterval::operator=(const TimeInterval & other)
```

```
{
    start = other.start;
    end = other.end;

    return *this;
}

TimeInterval::~TimeInterval()
{
}

bool TimeInterval::isBefore(const TimeInterval &other) const
{
    return end < other.start;
}

bool TimeInterval::isBefore(const Tick &time) const
{
    return end < time;
}

bool TimeInterval::overlaps(const TimeInterval &other) const
{
    return !this->isBefore(other) && !other.isBefore(*this);
}
```

## B.10 DEFINIÇÕES BÁSICAS

– src/Types.h

```
/*
 * Types.h
 *
 *  Created on: Apr 15, 2014
 *      Author: olav e gabriel
 */

#ifndef TYPES_H_
#define TYPES_H_

#include <vector>

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>

#include "Operation.h"

namespace XCHECK {
```

```
20
21    using std::vector;
22    using std::map;
23    using std::pair;
24
25    static const size_t MAX_NUMBER_OF_OPERATIONS = 65000;
26
27    enum EdgeType { UNDEFINED, PROCESSOR, EXECUTION, OBSERVED, INFERRED,
          INFERRED_IN_BACKTRACK, TIME };
28
29    // Typedefs
30
31    typedef boost::adjacency_list<boost::setS, boost::vecS, boost::directedS, Operation
          , EdgeType> MemoryOrderGraph;
32    typedef boost::graph_traits<MemoryOrderGraph>::vertex_descriptor vertex_t;
33    typedef boost::graph_traits<MemoryOrderGraph>::edge_descriptor edge_t;
34
35    typedef vector<vertex_t> ProgramOrderTrace;
36
37    typedef vector<vertex_t> Frontier;
38
39    typedef typename boost::iterator_property_map<
40          std::vector<boost::default_color_type>::iterator, boost::
              vec_adj_list_vertex_id_map<Operation, long unsigned int>,
41          boost::default_color_type,
42          boost::default_color_type &>
43    ColorMap;
44    typedef typename boost::property_traits<ColorMap>::value_type ColorValue;
45    typedef boost::color_traits<ColorValue> Color;
46
47  } /* namespace XCHECK */
48
49  #endif /* TYPES_H_ */
```

## B.11 XCHECK INCOMPLETO

– src/XCHECK.h

```
1  /*
2   * XCHECK.h
3   *
4   *  Created on: Sep 5, 2013
5   *      Author: olav e gabriel
6   */
7
8  #ifndef XCHECK_H_
9  #define XCHECK_H_
10
11 #include <string>
```

```
12 #include <vector>
13 #include <map>
14 #include <set>
15
16 #include <boost/config.hpp>
17 #include <boost/graph/adjacency_list.hpp>
18 #include <boost/tuple/tuple.hpp>
19 #include <boost/graph/depth_first_search.hpp>
20
21 #include "Types.h"
22 #include "Exceptions.h"
23 #include "DFS.h"
24 #include "Operation.h"
25 #include "Prefix.h"
26 #include "TimeAccumulator.h"
27
28 namespace XCHECK {
29
30 using std::string;
31 using std::vector;
32 using std::map;
33 using std::set;
34 using std::pair;
35
36 class XCHECK {
37 public:
38   static const string STATUS_FILE_NAME;
39   static const string TIME_FILE_NAME;
40
41   XCHECK(const vector<string> programOrderTraces, string pendingPeriodsFileName,
       string timeOrdersFileName);
42   virtual ~XCHECK();
43
44   void generate();
45   virtual void analyse();
46
47 protected:
48
49   // Graph construction methods
50
51   // Parses program order files and creates graph.
52   void createGraph();
53
54   // Memory Consistency Model
55   void addAlphaProcessorEdges();
56
57   void addExecutionAndObservedEdges();
58
59   void printStats();
60
61   // Misc
62
```

```
63  // Adds edges from source to target and infers as many edges as possible from
        source's pending period, putting all edges to addedEdges
64  // Throws CycleFoundWhileInferringEdges if a cycle is detected
65  void addAndInferEdge(vertex_t source, vertex_t target, EdgeType type, set<edge_t>
        & addedEdges, set<pair<vertex_t, vertex_t>> & recentlyInferredEdges);
66
67  bool thereIsAnOrder(vertex_t source, vertex_t target);
68
69  struct BacktrackInformation {
70    vertex_t executed;
71    size_t numberOfFrontierAttempts;
72    size_t numberOfFrontiersAttemptedFromExecutedOperation;
73    set<edge_t> addedEdges;
74    set<pair<vertex_t, vertex_t>> recentlyInferredEdges;
75    vertex_t previousStoreToAddress;
76    vertex_t previousStoreToAddressInProcessor;
77  };
78
79  struct LowerStartTime {
80    MemoryOrderGraph & graph;
81    LowerStartTime(MemoryOrderGraph & graph): graph(graph) {};
82    bool operator () (const vertex_t & v1, const vertex_t & v2) {
83      return static_cast<const Operation &>(graph[v1]).timeInterval.start
84          < static_cast<const Operation &>(graph[v2]).timeInterval.start;
85    };
86  };
87  struct LowerEndTime {
88    MemoryOrderGraph & graph;
89    LowerEndTime(MemoryOrderGraph & graph): graph(graph) {};
90    bool operator () (const vertex_t & v1, const vertex_t & v2) {
91      return static_cast<const Operation &>(graph[v1]).timeInterval.end
92          < static_cast<const Operation &>(graph[v2]).timeInterval.end;
93    };
94  };
95
96  // Throws CycleFoundWhileInferringEdges if a cycle is detected
97  void infer_edge(vertex_t u, set<edge_t> & addedEdges, set<pair<vertex_t, vertex_t>
        > & recentlyInferredEdges);
98  void initializePendingPeriods();
99  void obtainPendingPeriods();
100 void calculateTimeOrderEdges();
101 void obtainTimeOrderEdges();
102 set<vertex_t> & writesInPendingPeriod(vertex_t operation);
103 // Returns all operations o whose time interval overlaps operation's and such that
        there is a path from operation to o
104 set<vertex_t> successorsInPendingPeriod(vertex_t operation);
105 // Throws an exception if a cycle is found after adding edges
106 void reusableCycleChecking(edge_t & edge);
107
108 // Files
109 void saveStatusFile(int statusCode, const string & fileName) const;
110
```

```
111     vector<string> programOrderTraces;
112     string  pendingPeriodsFileName;
113     string  timeOrdersFileName;
114
115     MemoryOrderGraph graph;
116
117     vector<vertex_t> executionOrder;
118
119     vector<ProgramOrderTrace> coreTraces;
120
121     vector<set<vertex_t> > pendingPeriods;
122
123     map<uint64_t, vertex_t > storeMapping;
124     map<uint64_t, set<vertex_t> > loadMapping;
125     const uint64_t INITIAL_VALUE = 0;
126
127     set<pair<vertex_t, vertex_t > > inferredEdges;
128
129     bool backtrackingPhase;
130
131       // Time measure
132       TimeAccumulator::TimeAccumulator timer;
133 };
134
135 } /* namespace XCHECK */
136
137 #endif /* XCHECK_H_ */
```

– src/XCHECK.cpp

```
1  /*
2   * XCHECK.cpp
3   *
4   *  Created on: Sep 5, 2013
5   *      Author: olav e gabriel
6   */
7
8  #include <stdlib.h>
9  #include <algorithm>
10 #include <fstream>
11
12 #include <boost/graph/ transitive_reduction .hpp>
13
14 #include "XCHECK.h"
15 #include " FileParser .h"
16 #include "Debug.h"
17
18 namespace XCHECK {
19
20 const string XCHECK::STATUS_FILE_NAME = "xcheck_be.status";
21 const string XCHECK::TIME_FILE_NAME = "xcheck_be.time";
22
```

```
23 XCHECK::XCHECK(vector<string> programOrderTraces, string pendingPeriodsFileName,
       string timeOrdersFileName):
24   programOrderTraces(programOrderTraces),
25   pendingPeriodsFileName(pendingPeriodsFileName),
26   timeOrdersFileName(timeOrdersFileName),
27   backtrackingPhase( false )
28 {
29   Prefix :: nullVertex = graph. null_vertex ();
30 }
31
32 XCHECK::~XCHECK()
33 {
34 }
35
36 void XCHECK::generate()
37 {
38   Debug(Debug::General) << "Initializing pending periods" << "\n";
39
40   createGraph();
41
42    initializePendingPeriods ();
43
44   calculateTimeOrderEdges();
45 }
46
47 void XCHECK::analyse()
48 {
49   timer. start ();
50   Debug(Debug::General) << "Preparing for XCHECK analysis" << "\n";
51
52   try
53   {
54     createGraph();
55
56     obtainPendingPeriods ();
57
58     addAlphaProcessorEdges();
59
60     addExecutionAndObservedEdges();
61
62     obtainTimeOrderEdges();
63   }
64   catch (ValueError & e)
65   {
66     std :: cerr << e.what();
67     timer.pause();
68     timer.saveTimeFile(TIME_FILE_NAME);
69      saveStatusFile (1, STATUS_FILE_NAME);
70     exit (1);
71   }
72   catch (CycleFound & e)
73   {
```

```
74        std :: cerr  << e.what();
75        timer.pause();
76        timer.saveTimeFile(TIME_FILE_NAME);
77         saveStatusFile (2, STATUS_FILE_NAME);
78        exit (2);
79      }
80
81      Debug(Debug::General) << "No error found in best effort " << "\n";
82      timer.pause();
83      timer.saveTimeFile(TIME_FILE_NAME);
84       saveStatusFile (0, STATUS_FILE_NAME);
85    }
86
87    void XCHECK::createGraph()
88    {
89      Debug(Debug::General) << "Creating graph" << "\n";
90
91      // Create initial store
92      coreTraces.resize(programOrderTraces.size());
93
94      for (unsigned int core = 0; core < programOrderTraces.size(); ++core)
95      {
96        Debug(Debug::General) << "Reading program order trace for core " << core << "\n
            ";
97        FileParser programOrderParser(programOrderTraces[core], core);
98
99        vertex_t rootVertex = boost::add_vertex(graph);
100       graph[rootVertex] = Operation(core, Operation::ONULL, 0, 0, 0, TimeInterval(core,
            core));
101       coreTraces[core].push_back(rootVertex);
102
103       while (programOrderParser.hasNextOperation())
104       {
105         vertex_t operationVertex = boost::add_vertex(graph);
106         graph[operationVertex] = Operation();
107         Operation & operation = graph[operationVertex];
108         programOrderParser.nextOperation(operation);
109         coreTraces[core].push_back(operationVertex);
110
111         Debug(Debug::Vertex) << operation.toString() << "\n";
112
113         // Create load-store mapping
114         // Also initialize the 'time_of_latest_prior_write' for all address of the
            writes
115         switch (operation.type)
116         {
117         case Operation::READ:
118           loadMapping[(uint64_t)operation.data].insert(operationVertex);
119           break;
120         case Operation::WRITE:
121           storeMapping.insert(std::make_pair((uint64_t)operation.data, operationVertex)
            );
```

```
122             break;
123           default:
124             break;
125           }
126       }
127
128       vertex_t haltVertex = boost::add_vertex(graph);
129       graph[haltVertex] = Operation(core, Operation::HALT, 0, 0, std::numeric_limits<
            TimeInterval::Tick>::max(), TimeInterval(std::numeric_limits<TimeInterval::Tick
            >::max(), std::numeric_limits<TimeInterval::Tick>::max()));
130       coreTraces[core].push_back(haltVertex);
131   }
132
133   // Verify if all loads read a value from a store or the unique initial value in
        memory for it's address
134   for (map<uint64_t, set<vertex_t> >::iterator it = loadMapping.begin(); it !=
        loadMapping.end(); ++it)
135   {
136     uint64_t value = (*it).first;
137     if (!storeMapping.count(value))
138     {
139       if (value != INITIAL_VALUE)
140       {
141         throw InvalidValueForLoad();
142       }
143     }
144     else
145     {
146       // Verify if all loads are mapped to store of same address
147       Operation & mappedStore = graph[storeMapping.at(value)];
148       uint64_t address = mappedStore.address;
149
150       const set<vertex_t> & loads = (*it).second;
151       for (set<vertex_t>::iterator loadIt = loads.begin(); loadIt != loads.end(); ++
            loadIt)
152       {
153         Operation & load = graph[*loadIt];
154         if (address != load.address)
155         {
156           throw LoadMappedToStoreOfDifferentAddress();
157         }
158       }
159     }
160   }
161
162   // Also initialize ColorMap so we can use DFS
163   VisitorWithRecovery::initializeColorMap(graph);
164 }
165
166 void XCHECK::addAlphaProcessorEdges()
167 {
168   Debug(Debug::General) << "Adding edges based on Alpha MCM" << "\n";
```

```
169
170   for (unsigned int core = 0; core < coreTraces.size(); ++core)
171   {
172     Debug(Debug::General) << "Adding Alpha processor order edges for core " << core
        << "\n";
173
174     map<Operation::Addr, vertex_t> lastStore;
175     map<Operation::Addr, set<vertex_t> > lastLoads;
176     // The NULL operation is considered a membar
177     vertex_t lastMembar = coreTraces[core].front();
178
179     for (ProgramOrderTrace:: iterator it = ++coreTraces[core].begin(); it !=
        coreTraces[core].end(); ++it)
180     {
181       vertex_t operationVertex = *it;
182       Operation & operation = graph[operationVertex];
183       Operation::Addr address = operation.address;
184
185       if(operation.is_write() || operation.is_read())
186       {
187         if (lastStore.count(address)){
188           set<edge_t> edges;
189           set<pair<vertex_t, vertex_t> > recentlyInferredEdges;
190           addAndInferEdge(lastStore[address], operationVertex, PROCESSOR, edges,
        recentlyInferredEdges);
191         }
192         else
193         {
194           set<edge_t> edges;
195           set<pair<vertex_t, vertex_t> > recentlyInferredEdges;
196           addAndInferEdge(lastMembar, operationVertex, PROCESSOR, edges,
        recentlyInferredEdges);
197         }
198
199         operation.predecessorsCountInPO = 1;
200
201         if (operation.is_write())
202         {
203           if (lastLoads.count(address)){
204             for (vertex_t load : lastLoads.at(address))
205             {
206               set<edge_t> edges;
207               set<pair<vertex_t, vertex_t> > recentlyInferredEdges;
208               addAndInferEdge(load, operationVertex, PROCESSOR, edges,
        recentlyInferredEdges);
209
210               ++operation.predecessorsCountInPO;
211             }
212             lastLoads.erase(address);
213           }
214
215           lastStore[address] = operationVertex;
```

```
216             }
217             else
218             {
219                 lastLoads[address].insert(operationVertex);
220             }
221         }
222         else if(operation.is_membar())
223         {
224             if(lastStore.empty() && lastLoads.empty())
225             {
226                 set<edge_t> edges;
227                 set<pair<vertex_t, vertex_t>> recentlyInferredEdges;
228                 addAndInferEdge(lastMembar, operationVertex, PROCESSOR, edges,
    recentlyInferredEdges);
229                 operation.predecessorsCountInPO = 1;
230             }
231             else
232             {
233                 operation.predecessorsCountInPO = 0;
234
235                 for(std::pair<Operation::Addr, vertex_t> mapElement : lastStore)
236                 {
237                     vertex_t previousStoreVertex = mapElement.second;
238
239                     set<edge_t> edges;
240                     set<pair<vertex_t, vertex_t>> recentlyInferredEdges;
241                     addAndInferEdge(previousStoreVertex, operationVertex, PROCESSOR, edges,
    recentlyInferredEdges);
242
243                     ++operation.predecessorsCountInPO;
244                 }
245
246                 for(std::pair<Operation::Addr, set<vertex_t>> mapElement : lastLoads)
247                 {
248                     set<vertex_t> lastLoadsToAddress = mapElement.second;
249
250                     for (vertex_t load : lastLoadsToAddress)
251                     {
252                         set<edge_t> edges;
253                         set<pair<vertex_t, vertex_t>> recentlyInferredEdges;
254                         addAndInferEdge(load, operationVertex, PROCESSOR, edges,
    recentlyInferredEdges);
255
256                         ++operation.predecessorsCountInPO;
257                     }
258                     lastLoads.erase(address);
259                 }
260
261                 lastMembar = operationVertex;
262
263                 // Apply membar effect
264                 lastStore.clear();
```

```
265             lastLoads.clear();
266           }
267         }
268       }
269     }
270   }
271
272   void XCHECK::addExecutionAndObservedEdges()
273   {
274     for (unsigned int core = 0; core < programOrderTraces.size(); ++core)
275     {
276       Debug(Debug::General) << "Adding execution order and observed edges for core "
          << core << "\n";
277
278       map<uint64_t, vertex_t> lastStores;
279
280       for (ProgramOrderTrace::iterator it = coreTraces[core].begin(); it != coreTraces[
          core].end(); ++it)
281       {
282         vertex_t operationVertex = *it;
283         Operation & operation = graph[operationVertex];
284
285         switch (operation.type)
286         {
287         case Operation::READ:
288         {
289           vertex_t mappedStoreVertex = storeMapping.count(operation.data) ?
            storeMapping.at(operation.data) : coreTraces[core].front();
290           vertex_t lastStoreVertex = lastStores.count(operation.address) ? lastStores.
            at(operation.address) : coreTraces[core].front();
291           if (mappedStoreVertex != lastStoreVertex)
292           {
293             set<edge_t> edges;
294             set<pair<vertex_t, vertex_t>> recentlyInferredEdges;
295
296             // Execution edge
297             addAndInferEdge(mappedStoreVertex, operationVertex, EXECUTION, edges,
            recentlyInferredEdges);
298
299             // Observed edge
300             addAndInferEdge(lastStoreVertex, mappedStoreVertex, OBSERVED, edges,
            recentlyInferredEdges);
301           }
302           break;
303         }
304         case Operation::WRITE:
305           lastStores[(uint64_t)operation.address] = operationVertex;
306           break;
307         default:
308           break;
309         }
310       }
```

```
311     }
312 }
313
314 void XCHECK::addAndInferEdge(vertex_t source, vertex_t target, EdgeType type, set<
        edge_t> & addedEdges, set<pair<vertex_t, vertex_t > > & recentlyInferredEdges)
315 {
316   if ((type != INFERRED) || !thereIsAnOrder(source, target))
317   {
318     if (backtrackingPhase && (type = INFERRED))
319       type = INFERRED_IN_BACKTRACK;
320
321     edge_t edge;
322     bool isNew;
323     boost :: tie (edge, isNew) = boost :: add_edge(source, target, type, graph);
324
325     if (isNew)
326     {
327       switch (type)
328       {
329       case PROCESSOR:
330         Debug(Debug::Edge) << "Processor edge: ";
331         break;
332       case EXECUTION:
333         Debug(Debug::Edge) << "Execution edge: ";
334         break;
335       case OBSERVED:
336         Debug(Debug::Edge) << "Observed edge: ";
337         break;
338       case INFERRED:
339         Debug(Debug::Edge) << "Inferred edge: ";
340         break;
341       case INFERRED_IN_BACKTRACK:
342         Debug(Debug::Edge) << "Inferred in backtrack edge: ";
343         break;
344       case TIME:
345         Debug(Debug::Edge) << "Time edge: ";
346         break;
347       default :
348         break;
349       }
350       Debug(Debug::Edge) << static_cast<Operation &>(graph[source]).toString () << "
        -> "
351             << static_cast <Operation &>(graph[target]). toString () << "\n";
352
353       if (! backtrackingPhase)
354       {
355         Operation & targetOperation = graph[ target ];
356         ++ targetOperation . predecessorsCountInGTO;
357       }
358
359       addedEdges. insert (edge);
360       if (type != PROCESSOR)
```

```
361         {
362           reusableCycleChecking(edge);
363           infer_edge (source, addedEdges, recentlyInferredEdges );
364         }
365       }
366     }
367   }
368
369   bool XCHECK::thereIsAnOrder(vertex_t sourceVertex, vertex_t targetVertex )
370   {
371     const Operation & source = graph[ sourceVertex ];
372     const Operation & target = graph[ targetVertex ];
373
374     if (source. timeInterval .isBefore( target . timeInterval ))
375       return true ;
376     else if ( target . timeInterval .isBefore(source. timeInterval ))
377       throw CycleFoundInvolvingTimeOrder();
378
379     try
380     {
381       PathFinder finder ( targetVertex );
382       TerminateWithTimeOrder terminator(sourceVertex);
383       depthFirstSearch ( finder , sourceVertex , terminator , graph);
384     }
385     catch (const PathFound &)
386     {
387       return true ;
388     }
389     return false ;
390   }
391
392   void XCHECK::infer_edge(vertex_t u, set<edge_t> & addedEdges, set<pair<vertex_t,
          vertex_t > > & recentlyInferredEdges)
393   {
394     for ( vertex_t wVertex : writesInPendingPeriod(u))
395     {
396       Operation & w = graph[wVertex];
397       for ( vertex_t cVertex : successorsInPendingPeriod(wVertex))
398       {
399         Operation & c = graph[cVertex ];
400         if (c. is_read ())
401         {
402           vertex_t sourceVertex = storeMapping.count(c.data) ? storeMapping.at(c.data)
          : coreTraces[c. id_processor ]. front () ;;
403           Operation & source = graph[ sourceVertex ];
404           vertex_t conflictingWriteVertex = wVertex;
405           Operation & conflictingWrite = w;
406           vertex_t readVertex = cVertex;
407           Operation & read = c;
408
409           if (sourceVertex == conflictingWriteVertex )
410             continue;
```

```
411
412         if (// successorsInPendingPeriod makes sure conflictingWriteVertex -GO-> readVertex
413             !inferredEdges.count(std::make_pair(conflictingWriteVertex, readVertex))
414             && conflictingWrite.address == read.address)
415         {
416             recentlyInferredEdges.insert(std::make_pair(conflictingWriteVertex, readVertex));
417             inferredEdges.insert(std::make_pair(conflictingWriteVertex, readVertex));
418
419             if (source.timeInterval.isBefore(conflictingWrite.timeInterval))
420                 throw CycleFoundWhileInferringEdges(source, read, conflictingWrite, true);
421             else if (conflictingWrite.timeInterval.isBefore(source.timeInterval))
422                 continue;
423
424             addAndInferEdge(conflictingWriteVertex, sourceVertex, INFERRED, addedEdges, recentlyInferredEdges);
425         }
426     } else if (c.is_write())
427     {
428         vertex_t sourceVertex = wVertex;
429         Operation & source = w;
430         vertex_t conflictingWriteVertex = cVertex;
431         Operation & conflictingWrite = c;
432
433         if (sourceVertex == conflictingWriteVertex){
434             continue;
435         }
436
437         if (// successorsInPendingPeriod makes sure sourceVertex -GO-> conflictingWriteVertex
438             !inferredEdges.count(std::make_pair(sourceVertex, conflictingWriteVertex))
439             && source.address == conflictingWrite.address)
440         {
441             recentlyInferredEdges.insert(std::make_pair(sourceVertex, conflictingWriteVertex));
442             inferredEdges.insert(std::make_pair(sourceVertex, conflictingWriteVertex));
443
444             for (vertex_t readVertex : loadMapping[source.data])
445             {
446                 Operation & read = graph[readVertex];
447                 if (conflictingWrite.timeInterval.isBefore(read.timeInterval))
448                     throw CycleFoundWhileInferringEdges(source, read, conflictingWrite, false);
449                 else if (read.timeInterval.isBefore(conflictingWrite.timeInterval))
450                     continue;
451
452                 addAndInferEdge(readVertex, conflictingWriteVertex, INFERRED, addedEdges, recentlyInferredEdges);
453             }
```

```
454             }
455           }
456         }
457       }
458     }
459
460     void XCHECK::initializePendingPeriods()
461     {
462       MemoryOrderGraph::vertex_iterator vi, ve;
463
464       set<vertex_t> writes;
465       for (boost::tie(vi,ve) = boost::vertices(graph); vi != ve; ++vi)
466       {
467         vertex_t operationVertex = *vi;
468         Operation & operation = graph[operationVertex];
469         if(operation.is_write())
470         {
471           writes.insert(operationVertex);
472         }
473       }
474
475       std::ofstream stream(pendingPeriodsFileName.data());
476       if(stream.is_open()) {
477         for (boost::tie(vi,ve) = boost::vertices(graph); vi != ve; ++vi)
478         {
479           vertex_t operationVertex = *vi;
480           Operation & operation = graph[operationVertex];
481
482           MemoryOrderGraph::vertex_iterator ovi, ove;
483           for(vertex_t otherVertex : writes)
484           {
485             Operation & other = graph[otherVertex];
486             if(operation.timeInterval.overlaps(other.timeInterval))
487             {
488               stream << ' ' << otherVertex;
489             }
490           }
491           stream << '\n';
492         }
493         stream.close();
494       }
495       else {
496         throw FailedToOpenFile();
497       }
498     }
499
500     void XCHECK::obtainPendingPeriods() {
501       pendingPeriods.resize(boost::num_vertices(graph));
502
503       std::ifstream stream(pendingPeriodsFileName.data());
504       if(stream.is_open()) {
505         MemoryOrderGraph::vertex_iterator vi, ve;
```

```
506     for (boost::tie(vi,ve) = boost::vertices(graph); vi != ve; ++vi)
507     {
508       vertex_t operationVertex = *vi;
509       vertex_t otherVertex;
510       char divisor;
511       while(stream.get() != '\n') {
512         stream >> otherVertex;
513
514         pendingPeriods[operationVertex].insert(otherVertex);
515       }
516     }
517     stream.close();
518   }
519   else {
520     throw FailedToOpenFile();
521   }
522 }
523
524 void XCHECK::calculateTimeOrderEdges()
525 {
526   std::ofstream stream(timeOrdersFileName.data());
527   if(stream.is_open()) {
528     // Given ...
529     // Def-0. O (aka. operations): set of operations of all processors
530     // Def-1. E (aka. EndTimeOrder) : sorting of operations in O by End Time (
        increasing);
531     // Def-2. S (aka. StartTimeOrder) : sorting of operations in O by Start Time (
        increasing);
532
533     // [0.1] Compute E, S
534     vector<vertex_t> E(boost::vertices(graph).first, boost::vertices(graph).second);
535     vector<vertex_t> S(boost::vertices(graph).first, boost::vertices(graph).second);
536
537     // [0.2] Sorting
538     std::sort(E.begin(), E.end(), LowerEndTime(graph));
539     std::sort(S.begin(), S.end(), LowerStartTime(graph));
540
541
542     // Def-3 method ADD_TIME_EDGE(u,v) [prop. u -T-> v]
543     // : Add edge(u,v) to the graph $graph
544
545
546     // [1] For all operation v in O, call ADD_TIME_EDGE(u,v) st.
547     // #0: u -T-> v;
548     // #1: Don't exist an operation w st. u -T-> w -T-> v.
549
550     // [1.0] Initializing loop variable
551     int b = 0;
552
553     // [1.1] Iterate over S
554     for(vertex_t uVertex : S)
555     {
```

```
556         Operation & u = graph[uVertex];
557
558         // [1.2] Set b = max{k in [0,n) : graph[S[k]] -T-> graph[v]}
559         while ( static_cast <Operation>(graph[E[b]]).timeInterval.isBefore(u.timeInterval
            ))
560         {
561           ++b;
562         }
563
564         if (b > 0)
565         {
566           --b;
567
568           // [1.3] call ADD_TIME_EDGE(u,v) taking care of #0 and #1
569           int a = b;
570           int c = a;
571           while(a >= 0 && !static_cast <Operation>(graph[E[a]]).timeInterval.isBefore(
            static_cast <Operation>(graph[E[c]]).timeInterval))
572           {
573             stream << E[a] << ' ' << uVertex << '\n';
574
575             if ( static_cast <Operation>(graph[E[a]]).timeInterval.start > static_cast <
            Operation>(graph[E[c]]).timeInterval.start )
576             {
577               c = a;
578             }
579
580             --a;
581           }
582         }
583       }
584       stream.close();
585     }
586     else {
587       throw FailedToOpenFile();
588     }
589   }
590
591   void XCHECK::obtainTimeOrderEdges()
592   {
593     std::ifstream stream(timeOrdersFileName.data());
594     if(stream.is_open()) {
595       vertex_t u, v;
596       while(!stream.eof())
597       {
598         stream >> u >> v;
599         stream.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
600
601         set<edge_t> edges;
602         set<pair<vertex_t, vertex_t>> recentlyInferredEdges;
603         addAndInferEdge(u, v, TIME, edges, recentlyInferredEdges);
604       }
```

```
605       stream.close();
606     }
607     else {
608       throw FailedToOpenFile();
609     }
610   }
611
612   set<vertex_t> & XCHECK::writesInPendingPeriod(vertex_t operationVertex)
613   {
614     return pendingPeriods[ operationVertex ];
615   }
616
617   set<vertex_t> XCHECK::successorsInPendingPeriod(vertex_t operationVertex)
618   {
619     OperationRecorder recorder;
620     TerminateWithTimeOrder terminator( operationVertex );
621     depthFirstSearch( recorder, operationVertex, terminator, graph);
622     return recorder.operations;
623   }
624
625   void XCHECK::reusableCycleChecking(edge_t & edge)
626   {
627     CycleDetector detector;
628     TerminateWithTimeOrder terminator(boost::source(edge, graph));
629     depthFirstSearch( detector, boost::source(edge, graph), terminator, graph);
630   }
631
632   void XCHECK::saveStatusFile(int statusCode, const string & fileName) const
633   {
634       std::ofstream statusFile;
635       statusFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
636       statusFile.open(fileName.c_str());
637       statusFile << statusCode;
638       statusFile.close();
639   }
640
641   } /* namespace XCHECK */
```

## B.12 XCHECK COMPLETO

– src/XCHECKBacktracking.h

```
1  /*
2   * XCHECKBacktracking.h
3   *
4   *  Created on: Jul 8, 2014
5   *      Author: olav e gabriel
6   */
7
```

```
8  #ifndef XCHECKBACKTRACKING_H_
9  #define XCHECKBACKTRACKING_H_
10
11 #include "XCHECK.h"
12
13 namespace XCHECK {
14
15 class XCHECK_Backtracking: public XCHECK {
16 public:
17    static const string STATUS_FILE_NAME;
18    static const string TIME_FILE_NAME;
19
20    XCHECK_Backtracking(const vector<string> programOrderTraces, string
          pendingPeriodsFileName, string timeOrdersFileName);
21    virtual ~XCHECK_Backtracking();
22
23    void analyse();
24
25 private:
26
27    void findPathInFrontierGraph();
28
29    // Frontier movement methods
30
31    // Fills Frontier uninitialized with the NULL operations of each core
32    // and puts the first independent operations of each core in nextCandidates
33    void initializeFrontier (Frontier & uninitialized, vector<set<vertex_t> > &
          nextCandidates);
34    // Returns true if all operations in frontier are HALTs
35    bool isFinalFrontier (Frontier & frontier);
36    vertex_t selectOperationToExecute(BacktrackInformation & backtrackInfo);
37    vertex_t selectOperationToEnterFrontier (vertex_t executedVertex,
          BacktrackInformation & backtrackInfo);
38
39    // Candidates methods
40
41    // Updates nextCandidatesToCore with successors of operationVertex
42    void updateCandidates(vertex_t operationVertex, set<vertex_t> &
          nextCandidatesToCore);
43    void backtrackFrontierMovement(BacktrackInformation & backtrackInfo);
44    void backtrackExecutionChoice(BacktrackInformation & backtrackInfo);
45    // Backtracks nextCandidatesToCore, removing successors of operationVertex
46    void backtrackCandidates(vertex_t operationVertex, set<vertex_t> &
          nextCandidatesToCore);
47    void decrementCountInSuccessors(vertex_t executed);
48    void incrementCountInSuccessors(vertex_t backtracked);
49
50    unsigned int frontiersVisited;
51    unsigned int backtracksDone;
52
53    // Backtrack variables
54
```

```
55      Frontier    currentFrontier ;
56      vector<set<vertex_t> > nextCandidates;
57      size_t   numberOfOperationsSelected;
58      size_t   numberOfFrontiersAttemptedFromExecutedOperation;
59      map<Frontier, set<Prefix> >  frontierPrefixes ;
60      Prefix    currentPrefix ;
61
62      map<Operation::Addr, vertex_t> lastStoreToAddress;
63
64      vector<BacktrackInformation> frontierMovements;
65  };
66
67  } /* namespace XCHECK */
68
69  #endif /* XCHECKBACKTRACKING_H_ */
```

– src/XCHECKBacktracking.cpp

```
1   /*
2    * XCHECKBacktracking.cpp
3    *
4    *  Created on: Jul 8, 2014
5    *      Author: olav e gabriel
6    */
7
8   #include "XCHECKBacktracking.h"
9
10  namespace XCHECK {
11
12  const string XCHECK_Backtracking::STATUS_FILE_NAME = "xcheck_bt.status";
13  const string XCHECK_Backtracking::TIME_FILE_NAME = "xcheck_bt.time";
14
15  XCHECK_Backtracking::XCHECK_Backtracking(vector<string> programOrderTraces, string
        pendingPeriodsFileName, string timeOrdersFileName) :
16    XCHECK(programOrderTraces, pendingPeriodsFileName, timeOrdersFileName),
17     frontiersVisited (0),
18    backtracksDone(0),
19    numberOfOperationsSelected(0),
20    numberOfFrontiersAttemptedFromExecutedOperation(0)
21  {
22  }
23
24  XCHECK_Backtracking::~XCHECK_Backtracking()
25  {
26  }
27
28  void XCHECK_Backtracking::analyse()
29  {
30    XCHECK::analyse();
31
32    timer.start();
33    backtrackingPhase = true;
```

```
34
35    try
36    {
37       findPathInFrontierGraph ();
38    }
39    catch (NoPathFoundToFinalFrontier & e)
40    {
41       std :: cerr << e.what();
42       timer.pause();
43       timer.saveTimeFile(TIME_FILE_NAME);
44        saveStatusFile (3, STATUS_FILE_NAME);
45       exit (3);
46    }
47
48    Debug(Debug::General) << "No error found in backtracking" << "\n";
49    timer.pause();
50    timer.saveTimeFile("xcheck_bt.time");
51     saveStatusFile (0, STATUS_FILE_NAME);
52 }
53
54 void XCHECK_Backtracking::findPathInFrontierGraph()
55 {
56    Debug(Debug::General) << "Starting XCHECK analysis" << "\n";
57
58    Debug(Debug::General) << "Moving to initial frontier " << "\n";
59
60    nextCandidates.resize(coreTraces.size());
61    numberOfOperationsSelected = 0;
62    numberOfFrontiersAttemptedFromExecutedOperation = 0;
63
64      initializeFrontier ( currentFrontier , nextCandidates);
65     frontierPrefixes [ currentFrontier ]. insert ( currentPrefix );
66
67    while(! isFinalFrontier ( currentFrontier )) {
68      ++ frontiersVisited ;
69      Debug(Debug::Frontier) << "Current frontier : ";
70      for ( vertex_t v : currentFrontier ) {
71        Debug(Debug::Frontier) << static_cast<Operation &>(graph[v]).instID << ", ";
72      }
73      Debug(Debug::Frontier) << "\n";
74
75       assert (! frontierPrefixes [ currentFrontier ].empty());
76
77       BacktrackInformation lastMovement;
78       vertex_t executedVertex = selectOperationToExecute(lastMovement);
79       if (executedVertex != graph.null_vertex())
80       {
81       }
82       else
83       {
84          if (frontierMovements.empty()) {
85            Debug(Debug::General) << "Number of frontiers visited : " << frontiersVisited
```

```
      << "\n";
86        Debug(Debug::General) << "Number of backtracks done: " << backtracksDone
      << "\n";
87        throw NoPathFoundToFinalFrontier();
88      } else {
89        backtrackFrontierMovement(frontierMovements.back());
90        lastMovement = frontierMovements.back();
91        executedVertex = lastMovement.executed;
92        frontierMovements.pop_back();
93      }
94    }
95
96    vertex_t nextVertex = selectOperationToEnterFrontier(executedVertex,
      lastMovement);
97    if (nextVertex != graph.null_vertex())
98    {
99      numberOfOperationsSelected = 0;
100     frontierMovements.push_back(lastMovement);
101     Operation & next = graph[nextVertex];
102
103      currentFrontier.at(next.id_processor) = nextVertex;
104      frontierPrefixes[currentFrontier].insert(currentPrefix);
105   }
106   else
107   {
108     backtrackExecutionChoice(lastMovement);
109   }
110   numberOfFrontiersAttemptedFromExecutedOperation = 0;
111 }
112
113 Debug(Debug::General) << "Path to final frontier found: " << "\n";
114 for (BacktrackInformation info : frontierMovements) {
115   Debug(Debug::General) << static_cast<Operation &>(graph[info.executed]).toString
      () << "\n";
116 }
117 Debug(Debug::General) << "\n";
118 Debug(Debug::General) << "Number of frontiers visited : " << frontiersVisited << "
      \n";
119 Debug(Debug::General) << "Number of backtracks done: " << backtracksDone << "\n";
120 }
121
122 void XCHECK_Backtracking::initializeFrontier(Frontier & uninitialized, vector<set<
      vertex_t>> & nextCandidates)
123 {
124   uninitialized.clear();
125   for (unsigned int core = 0; core < coreTraces.size(); ++core) {
126     vertex_t nullVertex = coreTraces[core].front();
127     uninitialized.push_back(nullVertex);
128
129     updateCandidates(nullVertex, nextCandidates[core]);
130   }
131 }
```

```
132
133 bool XCHECK_Backtracking::isFinalFrontier( Frontier & frontier )
134 {
135   for (unsigned int core = 0; core < coreTraces.size(); ++core) {
136     Operation & operation = graph[ frontier [core ]];
137     if ( operation.type != Operation::HALT) {
138       return false;
139     }
140   }
141   return true;
142 }
143
144 vertex_t XCHECK_Backtracking::selectOperationToExecute(BacktrackInformation &
        backtrackInfo)
145 {
146   Debug(Debug::FrontierMovement) << "Looking for an operation to execute" << "\n";
147
148   bool cycleFound;
149   vertex_t executedVertex;
150
151   set<edge_t> addedEdges;
152   set<pair<vertex_t, vertex_t > > recentlyInferredEdges;
153   Frontier endTimeOrderedFrontier( currentFrontier );
154   std::sort(endTimeOrderedFrontier.begin(), endTimeOrderedFrontier.end(),
        LowerEndTime(graph));
155   do {
156     cycleFound = false;
157
158     if (numberOfOperationsSelected >= currentFrontier.size()) {
159       Debug(Debug::FrontierMovement) << "All operations in frontier were already
          attempted." << "\n";
160       return graph.null_vertex();
161     }
162
163     // vertex_t executedVertex = *std::min_element( currentFrontier.begin() +
          numberOfFrontierAttempts, currentFrontier.end(), LowerEndTime(graph));
164     executedVertex = endTimeOrderedFrontier[numberOfOperationsSelected];
165     ++numberOfOperationsSelected;
166
167     const Operation & executed = graph[executedVertex];
168
169     Debug(Debug::FrontierMovement) << "Trying to execute operation: " << executed.
          toString() << "\n";
170
171     const Operation & previouslyDiscarded = graph[endTimeOrderedFrontier.front()];
172     if ( previouslyDiscarded.timeInterval.isBefore(executed.timeInterval)) {
173       Debug(Debug::FrontierMovement) << "Chosen operation is after an already
          discarded operation in this frontier." << "\n";
174       return graph.null_vertex();
175     }
176
177     if (executed.predecessorsCountInGTO > 0) {
```

```
178         Debug(Debug::FrontierMovement) << "Chosen operation still has " << executed.
            predecessorsCountInGTO << " predecessors in Global Time Order." << "\n";
179         cycleFound = true;
180         continue;
181       }
182
183       try {
184         if (lastStoreToAddress.count(executed.address))
185         {
186           vertex_t lastStoreVertex = lastStoreToAddress.at(executed.address);
187           Operation & lastStore = graph[lastStoreVertex];
188
189           addAndInferEdge(lastStoreVertex, executedVertex, INFERRED, addedEdges,
            recentlyInferredEdges);
190
191           for (vertex_t load : loadMapping[lastStore.data])
192           {
193             addAndInferEdge(load, executedVertex, INFERRED, addedEdges,
            recentlyInferredEdges);
194           }
195         }
196       } catch (const CycleFoundInvolvingTimeOrder &)
197       {
198         cycleFound = true;
199         for (edge_t edge : addedEdges) {
200           Debug(Debug::Edge) << "Canceling: " << static_cast<Operation &>(graph[boost
            ::source(edge, graph)]).toString() << " -> " << static_cast<Operation &>(graph[
            boost::target(edge, graph)]).toString() << "\n";
201           boost::remove_edge(edge, graph);
202         }
203         for (pair<vertex_t, vertex_t> edge : recentlyInferredEdges)
204         {
205           inferredEdges.erase(edge);
206         }
207         addedEdges.clear();
208         recentlyInferredEdges.clear();
209         return graph.null_vertex();
210       } catch (const CycleFound &)
211       {
212         cycleFound = true;
213         for (edge_t edge : addedEdges) {
214           Debug(Debug::Edge) << "Canceling: " << static_cast<Operation &>(graph[boost
            ::source(edge, graph)]).toString() << " -> " << static_cast<Operation &>(graph[
            boost::target(edge, graph)]).toString() << "\n";
215           boost::remove_edge(edge, graph);
216         }
217         for (pair<vertex_t, vertex_t> edge : recentlyInferredEdges)
218         {
219           inferredEdges.erase(edge);
220         }
221         addedEdges.clear();
222         recentlyInferredEdges.clear();
```

```
223       }
224
225     } while (cycleFound);
226
227     const Operation & executed = graph[executedVertex];
228     auto prefixKey = std::make_pair(executed.id_processor, executed.address);
229
230     // Save backtrack info
231
232     backtrackInfo.executed = executedVertex;
233     backtrackInfo.numberOfFrontierAttempts = numberOfOperationsSelected;
234     backtrackInfo.previousStoreToAddress = lastStoreToAddress.count(executed.address) ?
            lastStoreToAddress.at(executed.address) : graph.null_vertex();
235     backtrackInfo.previousStoreToAddressInProcessor = currentPrefix[prefixKey];
236     backtrackInfo.addedEdges = addedEdges;
237     backtrackInfo.recentlyInferredEdges = recentlyInferredEdges;
238
239     // Prepare to move to next frontier
240
241     decrementCountInSuccessors(executedVertex);
242
243     if (executed.is_write())
244     {
245       lastStoreToAddress[executed.address] = executedVertex;
246
247       assert(currentPrefix[prefixKey] != executedVertex);
248       currentPrefix[prefixKey] = executedVertex;
249     }
250     else
251     {
252       assert(executed.type == Operation::ONULL);
253     }
254
255     return executedVertex;
256 }
257
258 vertex_t XCHECK_Backtracking::selectOperationToEnterFrontier(vertex_t executedVertex,
        BacktrackInformation & backtrackInfo)
259 {
260   Debug(Debug::FrontierMovement) << "Looking for a feasible frontier" << "\n";
261
262   Operation & executed = graph[executedVertex];
263
264   vertex_t nextVertex;
265   Frontier nextFrontier(currentFrontier.size());
266
267   do
268   {
269     if (numberOfFrontiersAttemptedFromExecutedOperation >= nextCandidates[executed.
          id_processor].size())
270     {
271       Debug(Debug::FrontierMovement) << "No feasible frontier to move by executing
```

```
        this operation." << "\n";
272         return graph.null_vertex();
273       }
274
275       set<vertex_t>::iterator candidateIt = nextCandidates[executed.id_processor].begin
            ();
276       std::advance(candidateIt, numberOfFrontiersAttemptedFromExecutedOperation);
277       nextVertex = *candidateIt;
278       const Operation & next = graph[nextVertex];
279
280       nextFrontier = currentFrontier;
281       nextFrontier.at(next.id_processor) = nextVertex;
282
283       ++numberOfFrontiersAttemptedFromExecutedOperation;
284     }
285     while (frontierPrefixes[nextFrontier].count(currentPrefix));
286
287     const Operation & next = graph[nextVertex];
288     Debug(Debug::FrontierMovement) << "Extending frontier with: " << next.toString()
          << "\n";
289
290     updateCandidates(nextVertex, nextCandidates[executed.id_processor]);
291
292     backtrackInfo.numberOfFrontiersAttemptedFromExecutedOperation =
          numberOfFrontiersAttemptedFromExecutedOperation;
293
294     return nextVertex;
295   }
296
297   void XCHECK_Backtracking::updateCandidates(vertex_t operationVertex, set<vertex_t> &
          nextCandidatesToCore)
298   {
299     nextCandidatesToCore.erase(operationVertex);
300     StoreInserter recorder(nextCandidatesToCore);
301     TerminateWithStoreInPO terminator(operationVertex);
302     depthFirstSearch(recorder, operationVertex, terminator, graph);
303   }
304
305   void XCHECK_Backtracking::backtrackFrontierMovement(BacktrackInformation &
          backtrackInfo)
306   {
307     Operation & lastExecuted = graph[backtrackInfo.executed];
308     vertex_t enteredVertex = currentFrontier[lastExecuted.id_processor];
309     Operation & entered = graph[enteredVertex];
310
311     Debug(Debug::Backtrack) << "Frontier was not feasible, backtracking extension of "
          << entered.toString() << "\n";
312
313     backtrackCandidates(enteredVertex, nextCandidates[lastExecuted.id_processor]);
314     currentFrontier[lastExecuted.id_processor] = backtrackInfo.executed;
315     numberOfOperationsSelected = backtrackInfo.numberOfFrontierAttempts;
316     numberOfFrontiersAttemptedFromExecutedOperation = backtrackInfo.
```

```
        numberOfFrontiersAttemptedFromExecutedOperation;
317 }
318
319 void XCHECK_Backtracking::backtrackExecutionChoice(BacktrackInformation &
        backtrackInfo)
320 {
321   Operation & lastExecuted = graph[backtrackInfo.executed];
322
323   Debug(Debug::Backtrack) << "No feasible frontier found, backtracking execution of "
        << lastExecuted.toString() << "\n";
324
325   ++backtracksDone;
326
327   if (backtrackInfo.previousStoreToAddress == graph.null_vertex())
328     lastStoreToAddress.erase(lastExecuted.address);
329   else
330     lastStoreToAddress[lastExecuted.address] = backtrackInfo.previousStoreToAddress;
331
332   auto prefixKey = std::make_pair(lastExecuted.id_processor, lastExecuted.address);
333   if (backtrackInfo.previousStoreToAddressInProcessor == graph.null_vertex())
334     currentPrefix[prefixKey] = graph.null_vertex();
335   else
336     currentPrefix[prefixKey] = backtrackInfo.previousStoreToAddressInProcessor;
337
338   for (edge_t edge : backtrackInfo.addedEdges) {
339     Debug(Debug::Edge) << "Removing: " << static_cast<Operation &>(graph[boost::
        source(edge, graph)]).toString() << " -> " << static_cast<Operation &>(graph[
        boost::target(edge, graph)]).toString() << "\n";
340     boost::remove_edge(edge, graph);
341   }
342   for (pair<vertex_t, vertex_t> edge : backtrackInfo.recentlyInferredEdges)
343   {
344     inferredEdges.erase(edge);
345   }
346
347   incrementCountInSuccessors(backtrackInfo.executed);
348 }
349
350 void XCHECK_Backtracking::backtrackCandidates(vertex_t operationVertex, set<vertex_t
        > & nextCandidatesToCore)
351 {
352   nextCandidatesToCore.insert(operationVertex);
353   StoreRemover remover(nextCandidatesToCore);
354   TerminateWithStoreInPO terminator(operationVertex, true);
355   depthFirstSearch(remover, operationVertex, terminator, graph);
356 }
357
358 void XCHECK_Backtracking::decrementCountInSuccessors(vertex_t executedVertex)
359 {
360   StoreDecrementer decrementer;
361   TerminateWithStoreInGTO terminator(executedVertex);
362   depthFirstSearch(decrementer, executedVertex, terminator, graph);
```

```
363 }
364
365 void XCHECK_Backtracking::incrementCountInSuccessors(vertex_t backtrackedVertex)
366 {
367     StoreIncrementer incrementer;
368     TerminateWithStoreInGTO terminator(backtrackedVertex, true);
369     depthFirstSearch(incrementer, backtrackedVertex, terminator, graph);
370 }
371
372 } /* namespace XCHECK */
```