Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo e análise de algoritmos para o problema do produtor único e consumidor único

Gabriel Patrick Alcantara Mourão
Nur Corezzi

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo e análise de algoritmos para o problema do produtor único e consumidor único

Gabriel Patrick Alcantara Mourão
Nur Corezzi

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. Marcos Fagundes Caetano
CIC/UnB

Dr. Lucas Rodrigues Costa
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 26 de outubro de 2021

Dedicatória

Dedico a Deus que sempre cuidou de mim. Aos meus pais Leonardo e Keila que me ensinaram que nada vem sem esforço. À minha esposa Anny, a qual me apoiou e foi compreensiva durante todo esse período.

Gabriel Patrick Alcantara Mourão.

Dedico este trabalho a meus pais Arnaldo e Erika que tanto deram suporte durante estes anos suados, a minha querida avó Lindóya quem mais torceu pelo tão esperado canudo.

Nur Corezzi.

Agradecimentos

Agradecemos primeiramente a Deus por mais essa conquista. Um agradecimento especial aos colegas de curso que acompanharam nos altos e baixos da graduação, às nossas famílias que deram todo o suporte necessário durante o tempo de graduação, ao nosso orientador Eduardo Adilio Pelinson Alchieri por propor e guiar o tema para o trabalho, dando direção e acompanhando frequentemente nos momentos em que mais estávamos perdidos. E por fim agradecemos por esse período de parceria da dupla para construção deste trabalho o que tornou a experiência infinitamente mais agradável.

Resumo

O problema do **produtor e consumidor** é um exemplo clássico de sincronização em *buffers* com capacidade de armazenamento limitada. Múltiplos produtores e consumidores podem cooperar acessando uma região de memória compartilhada para troca de dados. Ao se reduzir o problema para um único produtor e um único consumidor é possível aplicar soluções que utilizem mecanismos de sincronização mais eficientes do que as alternativas bloqueantes. Neste contexto é importante compreender o funcionamento destes mecanismos e o porquê apresentam os ganhos observados em máquinas modernas com múltiplos núcleos. Trabalhos atuais não oferecem comparações entre os algoritmos bloqueantes e não bloqueantes além de não considerarem suas otimizações mais atuais. Neste trabalho, experimentos foram efetuados em máquinas com múltiplas unidades de processamento para entender como os algoritmos podem se comportar, principalmente em ambientes que reflitam o contexto atual de uso em massa de programas concorrentes em nuvem.

Palavras-chave: Multi-thread, Lock-free, Produtor único e Consumidor Únicos

Abstract

The producer-consumer problem is a classical example of a bounded buffer synchronization problem. Multiple producers and consumers can cooperate by accessing a shared memory region to exchange data. By reducing the problem to a single producer and a single consumer, it is possible to apply solutions that use more efficient synchronization mechanisms than blocking alternatives. In this context, it is important to understand how these mechanisms work and why they present the gains observed in modern machines with multiple cores. Current papers do not make comparisons between blocking and non-blocking algorithms in addition to not considering their optimizations. In this work, experiments were conducted on machines with multiple processing units to understand how algorithms behave, especially in environments that reflect the current context of the massive use of concurrent cloud programs.

Keywords: Multi-thread, Lock-free, Single Producer Single Consumer

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviaturas e Siglas

CAS Compare And Swap.

CC Cross Core.

CPU Núcleo Central de Processamento.

CS Cross Socket.

DRAM Dynamic Random Access Memory.

FAA Fetch And Add.

FIFO First In First Out.

GC Garbage Colector.

GCC GNU Compiler Collection.

GIL Global Interpreter Lock.

IPC Inter-Process Communication.

JIT Just in Time Compiler.

JVM Java Virtual Machine.

LIFO Last In First Out.

LLC Last Level Cache.

MESI Modified, Exclusive, Shared, Invalid.

MPMC Multi-Producer Multi-Consumer.

MPSC Multi-Producer Single Consumer.

PMC Performance Monitoring Counter.

PMU Performance Monitoring Unit.

RAM Random Access Memory.

SC Same Core.

SPMC Single Producer Multi-Consumer.

SPSC Single Producer Single Consumer.

SRAM Static Random Access Memory.

SSH Secure Shell.

Capítulo 1

Introdução

O problema do **produtor e consumidor** é um exemplo clássico de sincronização em *buffers* com capacidade de armazenamento limitada [13]. *Buffer* neste contexto se refere a uma região de memória compartilhada entre o produtor e o consumidor. Em sua forma mais básica o problema considera que dois processos, um produtor e um consumidor que compartilham um *buffer* de tamanho fixo. O papel do produtor é gerar um item e então inseri-lo no *buffer* repetindo estas ações de forma cíclica. O papel do consumidor é remover itens do *buffer* para consumo.

O fato de existir compartilhamento de memória exige que seja feita sincronização no acesso da região compartilhada também conhecida como **região crítica** [14]. É importante que este fato seja levado em consideração para que não ocorram problemas de concorrência como por exemplo um produtor sobrescrevendo uma posição do *buffer* em que o consumidor ainda não terminou de ler ou um consumidor lendo um elemento do *buffer* que ainda não teve sua escrita finalizada.

A utilização de *Locks* permite solucionar estes problemas de sincronização porém possuem custo computacional não negligenciável e nem apresentam uma boa escalabilidade [15, 16]. Para maior eficiência, estruturas não bloqueantes, também classificadas como **lock-free** e **wait-free**, são mais desejadas pois funcionam melhor em casos de sincronização frequente por meio do uso de apenas primitivas de *hardware* mais básicas como barreiras de memória. Ambas classificações são relativas as garantias de progresso do programa, o qual pode ser constituído por múltiplas linhas de execução também conhecidas como *threads*. *Lock-free* são implementações que garantem um progresso geral do programa mesmo que algumas das *threads* estejam bloqueadas ou em ***starvation*** (definido como um estado em que se espera indefinidamente por um recurso) [17] e implementações *wait-free* garantem que todas as *threads* envolvidas irão fazer progresso sem que fiquem em *starvation* por tempo indefinido [17], ou seja, providenciando vazão por *thread*.

Uma das estruturas concorrentes mais simples são as filas First In First Out (FIFO)

de produtor único e consumidor único Single Producer Single Consumer (SPSC), em contraste com as filas de múltiplos produtores e ou múltiplos consumidores que possuem maior complexidade e custo computacional devido aos problemas de compartilhamento de memória [18]. As filas SPSC dominam o espaço de softwares embarcados e surgem de uma variedade de padrões de projeto paralelos e da distribuição de redes de processos Kahn [19] em arquiteturas de multiprocessadores. Também são utilizadas em pipelines de processamento de dados [16], comunicações entre máquinas virtuais [16] e algoritmos de tolerância a falhas *multi-threading* (por exemplo, [20, 21, 22, 23]).

O raciocínio formal sobre filas limitadas SPSC remonta ao trabalho de grande influência de Lamport [1], onde são ilustradas técnicas de prova em programas concorrentes. Diversas técnicas mais atuais já foram propostas e aperfeiçoaram significativamente o desempenho do algoritmo original [12, 11]. Embora existam muitos estudos de desempenho de soluções para SPSC, não foram encontrados estudos compreensivos sobre o funcionamento de estruturas *wait-free* que demonstrem e façam uma comparação prática das evoluções desde as estruturas bloqueantes até as soluções mais otimizadas. Uma análise em um contexto de multi-processadores também se faz relevante devido principalmente a popularização da computação em nuvem, que utiliza em larga escala máquinas com múltiplas CPUs interligadas via *socket*.

1.1 Objetivos

1.1.1 Objetivo geral

Este trabalho visa reunir de maneira compreensiva todo o conhecimento básico necessário para a compreensão do funcionamento de algoritmos *wait-free* para a resolução do problema do produtor único e consumidor únicos. Tanto um referencial teórico compilado de fontes dispersas quanto observações experimentais serão fornecidas para averiguar os comportamentos esperados. Implementações em Java serão selecionadas para as comparações e os resultados servirão de guia para aqueles que desejam escolher entre as filas apresentadas.

1.1.2 Objetivos específicos

- Compreender os componentes básicos de *hardware/software* por trás do funcionamento de estruturas de dados não bloqueantes;
- Identificar as principais soluções e otimizações para o problema do produtor único e consumidor único;

- Execução de experimentos em ambiente de teste com máquinas físicas;
- Analisar a diferença de desempenho entre algoritmos bloqueantes e não bloqueantes;
- Analisar a diferença de desempenho entre as filas identificadas como o estado da arte;
- Analisar como o contexto de execução (por exemplo, ambos processos sendo executados no mesmo núcleo ou em núcleos diferentes) pode influenciar no desempenho dos algoritmos.

1.2 Organização

O restante deste trabalho está organizado da seguinte forma:

1. O **Capítulo 2** apresenta os conceitos de problemas de compartilhamento de memória na programação concorrente, introduz o problema do produtor e consumidor únicos e compila uma pequena revisão de literatura;
2. O **Capítulo 3** descreve as principais características dos algoritmos utilizados bem como as técnicas de otimização mais recentes;
3. O **Capítulo 4** mostra toda a configuração e ferramentas necessárias para a execução, assim como a obtenção e análise dos resultados;
4. O **Capítulo 5** contém a conclusão e sugestões de trabalhos futuros.

Capítulo 2

Fundamentação Teórica e Trabalhos Relacionados

Este capítulo tem como objetivo apresentar os mecanismos e definições para a compreensão de algoritmos *wait-free* e trazer uma visão global do problema do produtor e consumidor. Serão vistos conceitos de programação concorrente, com intuito de abordar as questões de *hardware* e *software* referentes aos problemas de compartilhamento de memória presentes no problema do produtor e consumidor. Por fim, será apresentado como esse problema é visto por diferentes autores, e como o trabalho proposto se encaixa na literatura atual de otimizações e desempenho focados nos algoritmos SPSC.

2.1 Programação Concorrente

Programas concorrentes são programas que podem conter dois ou mais processos (ou *threads*) que cooperam entre si para realizar uma tarefa em comum [24], onde cada processo é um conjunto de instruções sequenciais executadas como um programa sequencial. Estes processos podem estar sendo executados em um único processador, em vários processadores, mas em mesma máquina ou em processadores distribuídos por uma rede. A ideia é que haja um processamento simultâneo lógico desses fluxos de execução.

Por padrão um processo é associado a uma *thread* mas vale ressaltar que ambos são conceitos diferentes onde, processos são utilizados para agrupar recursos e *threads* é a entidade escalonada para utilizar o processador. Um processo pode conter múltiplas *threads*, também chamado de *multithread*, que podem ser executadas de maneira independente, mas compartilham recursos como código, dados e arquivos abertos. *Hyper Threading*, nesse contexto, é uma tecnologia desenvolvida pela Intel [25] que permite que um único processador possa ser divido em dois processadores lógicos, o que permite a execução aparente de duas *threads*.

Para que haja uma cooperação entre os processos, é necessário existir comunicação. Existem diferentes mecanismos disponíveis para Inter-Process Communication (IPC), um dos mais comuns é através da memória compartilhada, que pode ser modificada por diferentes processos [26]. Esse mecanismo é comumente encontrado em sistemas *multithread*. É interessante destacar que essa memória compartilhada é considerada uma região crítica por poder sofrer acesso concorrente de diferentes processos.

Outros exemplos de mecanismos que são complementares a memória compartilhada são os *Lock/Mutex* e Semáforos que focam em garantir uma correta execução dos processos e acesso a regiões críticas. *Lock/Mutex* é uma primitiva de sincronização que garante a exclusão mútua durante o controle de concorrência. O semáforo pode ser considerado uma extensão do *lock*, o qual possui uma variável inteira que pode sofrer operações de *UP* e *DOWN* para incrementar ou decrementar em 1 a partir do seu valor atual. O controle de acesso à região crítica é feito à partir do valor dessa variável semáforo.

2.2 Problemas de compartilhamento de memória e seus impactos

Para garantir a sincronização e a correta execução de programas concorrentes é importante se atentar as implicações relativas ao compartilhamento de memória entre *threads* e processos. CPUs modernas funcionam baseados em uma arquitetura de memória em hierarquia que deve ser compreendida para que possam ser exploradas otimizações a nível de software.

2.2.1 Arquitetura moderna de CPUs

Para aproveitar ao máximo a utilização dos recursos computacionais atuais, diversas técnicas são empregadas em CPUs com arquitetura modernas, com o intuito de gerar melhor desempenho. A nível de projeto é comum que uma mesma CPU possua diversos núcleos e até mesmo cheguem a coexistir com outras CPUs em uma mesma placa-mãe com comunicação via *socket*. Internamente dentro de um núcleo existem processos que possibilitam o arranjo de elementos de hardware da CPU de forma que seu desempenho geral seja aumentado.

Paralelismo a nível de instrução, conhecida como computação superescalar ou sistema de pipeline é uma das técnicas utilizadas para obter um melhor desempenho. O processamento de uma instrução da CPU, tais como instruções de *fetch*, *decode*, *execute* e *write-back*, podem ser internamente divididas em diversas etapas [27]. Quando a primeira etapa de uma instrução está pronta, essa instrução pode ser encaminhada para a pró-

xima etapa e o processamento da próxima instrução pode começar seu primeiro estágio. Deste modo, diversas instruções podem ser executadas em paralelo, fazendo um uso mais otimizado do processador.

Embora o sistema de pipeline seja eficaz em sua proposta, a CPU necessita lidar com **instruções de desvio condicionais**. Essas são instruções que limitam o pipeline a entrar em um estado ocioso, chamado de *pipeline stall*, até o resultado de uma condição ser conhecida, fazendo com que a próxima instrução não inicie enquanto a atual não terminar [27]. Para tentar driblar essa limitação existe a técnica conhecida como **branch prediction**, onde o processador tenta determinar onde a execução irá continuar após um salto condicional no código. Outra técnica conhecida como **execução especulativa** vai um passo além, mesmo que não haja instruções de desvio condicionais no código o processador executa uma instrução no "chute", ou seja, a etapa é feita antes de saber se vai ser realmente necessária, de modo que se o palpite estiver correto, o resultado poderá ser utilizado e caso contrário será descartado.

Um outro refinamento, que permite um aproveitamento melhor dos recursos da CPU, é a possibilidade de execução fora de ordem das instruções. O processador preenche os estágios do pipeline com a próxima instrução que se encontra pronta para ser executada, mesmo que essa instrução não seja a próxima na ordem sequencial do programa, e então no final reordena os resultados para parecer que as instruções foram processadas na ordem original.

Além destas técnicas de paralelização, houve uma grande melhoria no poder computacional das CPUs devido aos avanços tecnológicos na produção de *chips* de computador [27]. Porém o tempo de acesso a Dynamic Random Access Memory (DRAM) [28] ou memória principal não acompanhou os avanços na velocidade das CPUs, o que exigiu a utilização de diferentes técnicas para mitigar o problema. A alternativa mais empregada atualmente consiste na utilização de memórias estáticas Static Random Access Memory (SRAM) [29]. Estas memórias permitem grandes velocidades de acesso pois entre outros fatores não necessitam de recargas periódicas para manter informação como DRAM necessitam, porém possuem um custo de produção maior. Para efeitos de comparação, 1 gigabyte de SRAM possui um custo aproximado de $5000 dólares enquanto 1 gigabyte de DRAM custa por volta de $20-$75 dólares, ambos dados coletados em 2019 [30].

2.2.2 Cache

Para mitigar o alto custo das memórias SRAM, as CPUs modernas organizam sua memória em hierarquia utilizando o **princípio da localidade** [31]. Usualmente um programa tende a acessar uma porção pequena do espaço endereçável em memória, justamente porque existe uma tendência a utilizar itens que já foram referenciados assim como itens

próximos a estes itens referenciados. Este comportamento permite a utilização de **caches** que serão responsáveis por salvar uma pequena porção da memória principal e evitar que a CPU tenha que utilizar a DRAM a todo momento que um dado for acessado.

As memórias cache possuem uma hierarquia de níveis onde cada nível armazena diferentes quantidades de dados e possui um projeto em hardware específico que permite otimizações relativas as funções realizadas por cada nível. A organização mais comum consiste nos níveis L1, L2 e L3, sendo este último também conhecido como Last Level Cache (LLC). Os acessos realizados pela CPU sempre passam em ordem pelos diferentes níveis seguindo a hierarquia na Figura 2.1. O fenômeno de encontrar uma informação requisitada em um nível de cache é conhecido como *cache hit*. Os tempos de acesso mais comuns também podem ser observados na Figura 2.1. Caso não encontre a informação requisitada em um determinado nível, será necessário requisitar ao próximo nível de cache gerando um *cache miss* que faz com que a CPU pare o processamento momentaneamente a espera do dado.

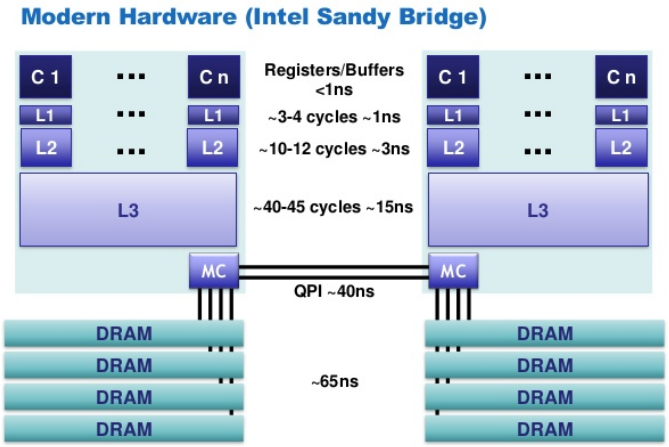


Figura 2.1: Layout de CPU [2].

É importante ressaltar que apesar de utilizarem o mesmo tipo de memória SRAM, o acesso a cada nível possui um tempo de resposta diferente devido a otimizações especificas no seu funcionamento [32]. Entre outras especificidades, caches L1 por exemplo possuem suporte a operações atômicas o que as demais caches não necessitam. Os tamanhos de cada um destes níveis também são distintos e visam conciliar suas funcionalidades com o princípio da localidade. Caches L1 possuem o menor tamanho, por volta de 32 Kilobytes, seguido pela L2 com tamanho de 256 Kilobytes, sendo que normalmente todos os núcleos as possuem de maneira exclusiva. Já caches L3 são compartilhadas entre todos os núcleos de uma mesma CPU e costumam ter aproximadamente 8 Megabytes.

2.2.3 Coerência de cache

Memórias cache são divididas e administradas em unidades de **linhas de cache**, como na Figura 2.2, que possuem uma capacidade de 16 a 256 bytes [4]. As linhas de cache servem para diminuir a complexidade das movimentações de memória e do monitoramento para realização dos protocolos de coerência. Uma mesma região de memória de um programa pode estar duplicada em caches de diferentes núcleos, por exemplo em situações que um programa possui diversas *threads* em execução. É de responsabilidade do **sistema de coerência de cache** garantir que não ocorram condições de corrida na escrita ou leitura de dados replicados em diferentes linhas de cache.

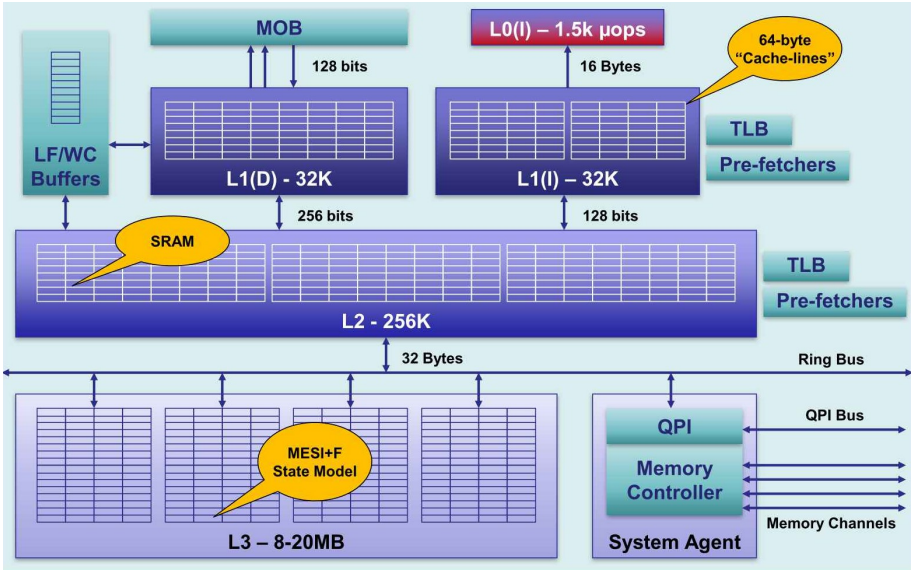


Figura 2.2: Estrutura interna de memória cache [3].

Para melhor visualização, cada *thread* pode estar manipulando dados distintos, porém em mesma linha de cache. Os usuários não possuem controle sobre como seu programa será disposto em memória, portanto para garantir que nenhuma escrita seja perdida ou sobrescrita, os sistemas de coerência de cache empregam mais comumente a utilização do protocolo **MESI** (*Modified, Exclusive, Shared e Invalid*) [2]. Este protocolo se baseia em quatro estados básicos (na prática outros protocolos utilizam diversos outros estados [2]) sendo eles modificada, exclusiva, compartilhada e inválida. Cada linha de cache possui um estado que pode ser modificado de maneira atômica.

Linhas de cache em memórias que não apareçam duplicadas estão em estado **exclusiva**. Neste estado leituras estão liberadas e a linha de cache pode ser descartada sem que necessite fazer comunicação com as demais caches. A execução de escritas irá fazer com que a linha vá para um estado de **modificada**. É importante ressaltar que linhas modificadas não necessariamente serão sincronizadas no mesmo instante com a memória

principal, justamente para evitar escritas custosas devido ao tempo de acesso a memória RAM, além disso também existe uma garantia de que esta linha não é compartilhada com nenhuma outra cache. Caso outras caches façam requisições relativas a esta linha, a própria cache em posse do dado pode também responder a requisição sem a necessidade de ir até a memória principal.

Linhas de cache **compartilhada** ocorrem quando algum outro núcleo requisita uma mesma linha de cache, neste momento, todas as linhas em posse desta mesma região de memória serão movidas para o estado compartilhada. Linhas compartilhadas também podem ser lidas sem que seja necessário sincronizações, porém ao se realizar escritas, apenas uma das escritas deve ser realizada com êxito. Ao se realizar uma escrita em linha de cache compartilhada todas as demais linhas devem ser movidas para o estado **inválida** e a linha de cache com o novo valor deve ser atualizada para exclusiva e em um protocolo convencional terá seu valor também atualizado em memória principal. As demais *threads* com linhas inválidas ao realizarem leituras ou escritas devem primeiramente requisitar a linha atualizada para que possam dar sequência a operação desejada ou seja linhas inválidas são tratadas como um espaço de memória livre sem dados relevantes.

Para que os protocolos de coerência de cache sejam implementados é necessário que existam trocas de mensagens entre os núcleos. Uma das formas de comunicação mais comuns entre os periféricos de uma CPU é a utilização de um *bus*. Um *bus* é um conjunto de fios condutores que irão transmitir informação a quem estiver conectado. Vale ressaltar que o acesso de escrita a um *bus* é exclusivo para que não ocorram confusões nos bits enviados. Em uma CPU com núcleos que compartilham um mesmo *bus* as seguintes mensagens são suficientes para o funcionamento de um protocolo MESI [4]:

- Leitura: irá requisitar o endereço de uma linha de cache a ser lida.
- Reposta de leitura: uma resposta de leitura pode ser dada tanto por outras caches como por exemplo no caso de uma linha de cache no estado modificada ou também podem ser uma resposta da memória principal. Esta mensagem sempre ocorre em decorrência de uma mensagem de leitura.
- Invalidação: esta mensagem contém o endereço da linha de cache a ser invalidada. Todos os núcleos que a receberem devem passar a linha de cache para o estado inválida.
- Reconhecimento de invalidação: ao receber uma mensagem de invalidação é necessário que também seja enviada uma confirmação da invalidação para que se mantenha a consistência dos estados da cache.

- Leitura e invalidação: uma única mensagem que se comporta como uma leitura seguida de uma invalidação. É necessário que seja respondida com a linha de cache requisita e um conjunto de confirmações de invalidações.
- Escrita: uma mensagem de escrita contém a informação e o endereço de memória principal ao qual deverá ser escrita. Esta mensagem funciona como um *flush* da linha em memória cache.

2.2.4 Buffers de escrita

Um dos problemas associados ao esquema de memória cache até aqui apresentado é a demora relativa a uma escrita que não possui dados em linha de cache. Como foi mostrado, para que uma escrita seja realizada a linha de cache deve estar presente na memória. Portanto se a linha não for encontrada será necessário fazer a requisição, para que em seguida seja efetuada a escrita. Porém esta requisição gera um travamento da CPU, conhecida como **CPU stall**, fazendo com que o processador não consiga fazer progresso com outras instruções, até que as operações de leitura e escrita sejam efetuadas.

Para contornar esta situação são introduzidos **buffers de escrita** [4] entre os núcleos de processamento e as caches como na Figura 2.3. Estes *buffers* fazem com que o passo da escrita possa ser postergado para algum momento mais oportuno quando a linha de cache desejada já estiver disponível. Por exemplo o novo dado pode ser salvo no *buffer* de escrita e uma mensagem de leitura pode ser emitida, porém não será necessário esperar uma resposta de leitura pois o dado está seguro no *buffer* e poderá ser movido para a linha de cache assim que a resposta chegar.

Outro detalhe importante é que a CPU não mais deve ler dados apenas da cache, pois podem existir modificações em seu *buffer* de leitura que possuam um valor distinto do valor atual na cache. Um mecanismo chamado de **encaminhamento de escrita** faz com que consultas a cache pela CPU possam ser interceptadas e respondidas pelo *buffer* de escrita retornando assim os valores mais atuais e consequentemente garantindo a visualização sequencial das transformações efetuadas nos dados por uma mesma CPU. Porém essa garantia não é global e a introdução de *buffers* de escrita faz com que diferentes núcleos possam visualizar as transformações em memória em ordens distintas o que introduz a necessidade de **barreiras de memória**.

2.2.5 Barreiras de memória

Barreiras de memórias geralmente são usadas implicitamente em mecanismos de *lock*, *mutex*, e *semáforos* já com o intuito de prevenir o reordenamento do programa, sem que

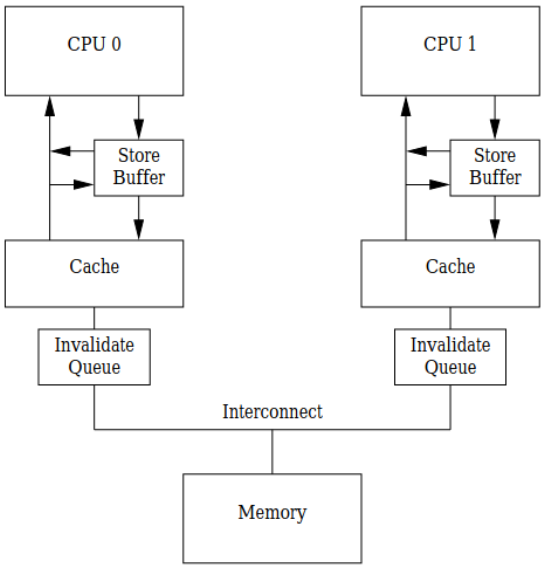


Figura 2.3: Buffers de escrita e filas de invalidação [4].

```
1  void foo(void)
2  {
3      a = 1;
4      b = 1;
5  }
6
7  void bar(void)
8  {
9      while(b == 0) continue;
10     assert(a == 1);
11 }
```

Código 2.1: Exemplo base de barreiras de memória

seja necessário que o programador tenha que se ater a estes detalhes durante o desenvolvimento. Já durante o desenvolvimento de algoritmos *lock-free*, se faz necessário o uso explicito de barreiras de memória.

Para exemplificar o que são e o porque barreiras de memória são necessárias, um exemplo será seguido a partir do Código 2.1. Supondo que existam duas *threads* onde uma irá executar o método *foo* e a outra o método *bar*. Devido a existência dos *buffers* de escrita é possível que seja efetuada uma sequência de passos em que a asserção na linha 10 apresente uma falha.

Supondo que um núcleo 0 execute *foo()* e não possua em sua cache uma linha que contenha a variável *a* porém possua em sua cache a variável *b* em estado exclusiva. Suponha também um núcleo 1 executando *bar()* e possua a variável *a* em linha de cache. A

seguinte sequência causaria uma asserção inesperada [4]:

1. Núcleo 0 executa *foo()* e realiza a primeira instrução $a = 1$. Após sua execução por ainda não conter a em linha de cache, será requisitada uma leitura e a será colocado em *buffer* de escrita. Neste caso será enviada uma mensagem de leitura e invalidação uma vez que outros núcleos em posse desta linha também deverão ser invalidadas;
2. Núcleo 1 executa *bar()* e realiza a primeira verificação $b == 0$. Como b não está em linha de cache deste núcleo é efetuada uma requisição de leitura da linha para que a comparação seja realizada;
3. Núcleo 0 executa a segunda instrução $b = 1$. Como a variável b já esta em linha de cache de sua memória a variável não vai para o *buffer* de escrita e a linha de cache já é atualizada com novo valor de b;
4. Núcleo 0 intercepta requisição de leitura da linha de cache contendo a variável b e manda a resposta com o valor de $b = 1$;
5. Núcleo 1 recebe o valor de $b = 1$ em resposta do núcleo 0 e efetua a comparação fazendo com que saia do *loop*;
6. Núcleo 1 efetua a asserção e faz a leitura de sua própria cache contendo $a = 0$ o que faz com que a asserção falhe;
7. Núcleo 1 finalmente recebe a requisição de leitura e invalidação da linha contendo a porém a asserção já foi efetuada indevidamente e responde com $a = 0$ e inválida a linha de cache mandando também um reconhecimento de invalidação;
8. Núcleo 0 recebe a linha contendo $a = 0$ seguido do reconhecimento de invalidação do núcleo 1 e salva $a = 1$ na linha de cache recebida.

Para garantir que as operações de memórias sejam visualizadas na ordem correta, surgem então instruções de **barreiras de memória** [33]. Estas barreiras são implementações em *hardware* e adicionam garantias de ordenação da submissão das escritas para cache. Ao encontrar uma barreira de memória em código, todas as entradas já em *buffer* de escrita serão marcadas para que sejam submetidas antes de qualquer outra instrução de escrita posterior. No caso do exemplo acima introduzindo uma barreira de memória entre as linhas 3 e 4 irá fazer com que o núcleo 0 não consiga submeter a atualização de $b = 1$ para memória cache antes que tenha submetido $a = 1$ que já estará em seu *buffer* de escrita. Esta pequena modificação garante que se $b = 1$ é um valor visível em cache então $a = 1$ também estará visível para os demais programas.

Outro fato importante a se observar é que ao submeter um valor ao *buffer* de escrita pode ser que as leituras subsequentes da mesma região de memória retornem o valor no *buffer* porém o valor já pode estar em uma versão mais atual na cache devido a modificações submetidas em outros núcleos [2]. Novas leituras também podem parecer ser executadas antes de escritas mais antigas devido a uma reordenação de uma leitura com uma escrita anterior [2]. Ambos os problemas podem ser resolvidos com a emissão de barreiras de memória que irão garantir a leitura de uma cache atualizada. Neste caso as barreiras apresentam um certo custo devido a necessidade de se limpar todas as entradas do *buffer* de escrita para cache, liberando em seguida a instrução de leitura e garantindo a execução em ordem correta.

Barreiras de memória que afetam apenas a CPU são classificadas como **barreiras de memória de hardware** e barreiras que apenas afetam o compilador, são chamadas de **barreiras de memória de software** [34]. Além destas classificações, uma barreira de memória pode ser restrita a leituras de memória, escritas de memória ou ambas operações. Uma barreira de memória que afeta leituras e escritas é uma **full memory barrier**.

2.2.6 Filas de invalidação

Um outro componente que também pode acarretar em visibilidade fora de ordem são as **filas de invalidação** [4]. Uma mensagem de reconhecimento de invalidação pode apresentar um certo tempo entre o recebimento da mensagem de invalidação e o envio da resposta devido as CPUs já possuírem operações sendo executadas, como trocas de mensagens entre outros núcleos, operações de escrita, leitura e etc. Internamente o *buffer* de escrita possui um tamanho limitado podendo encher com apenas algumas escritas. Para obter espaço novamente, a CPU deve aguardar que todas invalidações sejam completadas o que pode demorar devido as cargas de trabalho já executadas pelas demais CPUs. Como explicado anteriormente, essa mesma situação pode surgir também após uma barreira de memória mesmo que o *buffer* não esteja cheio, pois todas as escritas após essa barreira devem esperar as submissões de escritas anteriores completarem.

Para melhorar a velocidade das invalidações é possível colocar as mensagens de invalidação em fila para processamento posterior e assim responder de forma rápida com uma confirmação de invalidação. Este procedimento é possível sob a premissa de que nenhuma mensagem referente a linha a ser invalidada será emitida antes de efetuar a invalidação pendente em fila. Além das respostas rápidas ainda ocorre um desafogamento das caches que não precisam interromper as operações correntes.

Filas de invalidação podem gerar leituras de valores desatualizados da variável a ser acessada. No Código 2.1 após executar a verificação do *while* no método *bar()* já era de se esperar que com a adição da barreira de memória entre as linhas 3 e 4 que se $b =$

1 então $a = 1$ também deveria estar sincronizado em cache e visível para esta *thread*. Devido as filas de invalidação, uma linha contendo a com invalidação pendente pode ter sido lida sem que a invalidação tenha sido processada e, portanto retornando um valor desatualizado de a mesmo que a *thread* já pudesse visualizar um valor atual.

Para contornar este problema as barreiras de memória também atuam nas filas de invalidação. Ao encontrar uma barreira de memória todas as entradas na fila de validação serão marcadas e devem ser processadas antes de qualquer leitura subsequente da memória, funcionando de maneira análoga a atuação no *buffer* de escrita. No Código 2.2 ao se adicionar a barreira na linha 11 é possível garantir que a asserção não irá falhar, se a *thread* consegue visualizar $b == 1$ e sair do *while* é garantido que $a = 1$ já persiste em cache (devido a barreira na linha 4) e a barreira na linha 11 se certifica de limpar qualquer pendência da fila de invalidação antes que seja efetuada a leitura de a, gerando um cache *miss* que irá trazer o valor atualizado de $a = 1$ para a cache.

```
1  void foo(void)
2  {
3      a = 1;
4      smp_mb();
5      b = 1;
6  }
7
8  void bar(void)
9  {
10     while(b == 0) continue;
11     smp_mb();
12     assert(a == 1);
13 }
```

Código 2.2: Código 2.1 com barreiras de memória

2.2.7 Modelos de memória

É importante ressaltar que diferentes arquiteturas de CPU possuem **modelos de consistência de memória** distintos [7]. Um modelo de consistência diz como diferentes *threads* podem observar os diferentes estados da memória sendo compartilhada [35]. Um modelo que garanta consistência sequencial exige que cada *thread* execute as instruções do programa em ordem sem que ocorram reordenações, além disso devem perceber a execução das demais *threads* também em ordem de programa. O problema deste tipo de consistência é a impossibilidade das otimizações aqui citadas como os *buffers* de escrita e as filas de invalidação, portanto algumas CPUs optam por modelos que possuam garantias mais

fracas e disponibilizam funcionalidades como barreiras de memória que permitem adaptar a CPU para as garantias necessárias do programa.

Na exemplificação acerca dos *buffers* de escrita e filas de invalidação não há uma necessidade das barreiras de memória realizarem as marcações em ambas as estruturas. A primeira *thread* em *foo()* não necessita mexer na sua fila de invalidação e a *thread* em *bar()* também não precisa fazer nada com seu *buffer* de escrita. Portanto dependendo da arquitetura da CPU sendo utilizada diferentes instruções que implementam as barreiras de memória podem ser mais adequadas e otimizadas para o tipo de garantia de ordenação desejada. Para cada possibilidade de reordenação entre instruções de escrita e leitura podem existir suporte para as seguintes categorias de barreiras [7, 36]:

1. *LoadLoad*: a sequência (Load1; **LoadLoad**; Load2;) garante que o dado a ser lido em Load1 irá ocorrer antes da leitura em Load2 e qualquer outra instrução de leitura subsequente ao Load2. Necessário em casos que a CPU emprega filas de invalidação além de outras possíveis reordenações como *loads* especulativos.

2. *StoreStore*: a sequência (Store1; **StoreStore**; Store2;) garante que a escrita realizada em Store1 estará visível para os outros processadores antes que a escrita do Store2 ou qualquer outra escrita subsequente seja realizada. Costumam ser utilizadas em processadores que utilizam *buffers* de escrita para ordenar as submissões das modificações a cache.

3. *LoadStore*: a sequência (Load1; **LoadStore**; Store2;) garante que o dado a ser lido em Load1 será realizado antes da escrita em Store2 ser realizada.

4. *StoreLoad*: a sequência (Store1; **StoreLoad;** Load2;) garante que a escrita a ser efetuada pelo Store1 esteja visível para os outros processadores antes da instrução de leitura realizada por Load2. Quase todos os processadores modernos implementam instruções para este tipo de barreira pois a garantia deste tipo de ordenação pode ser custosa devido a necessidade de se limpar o *buffer* de escrita antes que a leitura possa ser efetuada.

Para cada categoria apresentada, diferentes processadores podem ou não implementar instruções específicas para garantir a ordenação desejada. Em alguns casos nenhum tipo de instrução será emitida pois o *hardware* já garante as ordenações de acordo com seu modelo de memória o que faz com que as barreiras colocadas em mais alto nível não gerem instruções para a CPU (*no-ops*) [7]. Portanto além de saber o que exatamente é emitido com uma barreira em determinada arquitetura também é importante conhecer o modelo de memória empregado pela CPU. Na Tabela 2.1 é possível observar algumas das

Processador	LoadStore	LoadLoad	StoreStore	StoreLoad
SPARC-TSO	no-op	no-op	no-op	membar
x86	no-op	no-op	no-op	mfence or cpuid
IA64	combinewithst.rel or ld.acq	ld.acq	st.rel	mf
ARM	dmb	dmb	dmb-st	dmb
PPC	lwsync	hwsync	lwsync	hwsync
Alpha	mb	mb	wmb	mb
PA-risc	no-op	no-op	no-op	no-op

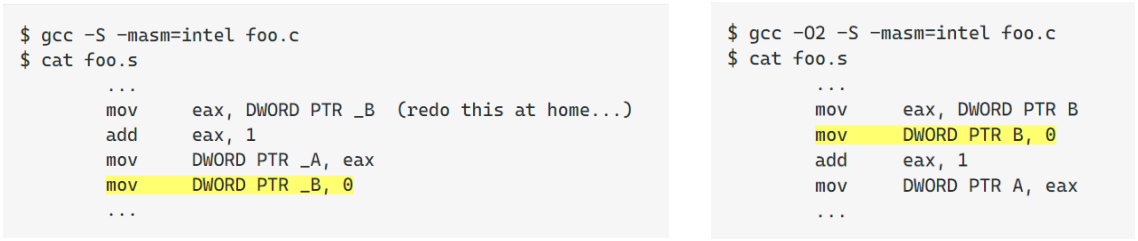
Tabela 2.1: Instruções emitidas por diferentes tipos de barreiras de memória [7]

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
ARMv7-A/R	Y	Y	Y	Y	Y	Y		Y
IA64	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

Figura 2.4: Reordenações executadas por diferentes arquiteturas [4].

instruções emitidas por diferentes CPUs e na Figura 2.4 um pequeno sumário de garantias de ordenações de algumas CPUs.

Processadores como PA-risc não necessitam da emissão de barreiras de memórias pois já possuem fortes garantias em seu modelo de ordenação de memória. É importante notar que apesar de não ocorrerem reordenações pelo *hardware* ainda é possível que a linguagem de programação utilizada faça otimizações por conta própria que podem resultar em um código já desordenado. Neste caso é importante que a linguagem também possua um modelo próprio de consistência de memória e forneça meios de comunicar com o compilador se há a necessidade de se manter as ordenações em código.



(a) (b)

Figura 2.5: Compilação (a) sem otimização e (b) usando a flag de otimização [5]

2.2.8 Reordenamento durante compilação

No geral, as etapas na qual um programa pode ser reordenado, com objetivo de otimizar a execução do programa será durante a compilação e a execução no processador. As reordenações realizadas pelo compilador partem de uma série de otimizações que são realizadas durante o processo de compilação de um programa [34]. A premissa dessas otimizações é que não seja alterado o comportamento original sequencial de um programa *single-thread*. Considere a seguinte função escrita em C, no Código 2.3. Após a compilação do código é gerado o seguinte código de máquina na Figura 2.5. Em (a) é possível ver que a escrita da variável B só ocorre após a escrita da variável A, como originalmente no Código 2.3. Já em (b) a escrita da variável B antecede a escrita da variável A, quando o comando de otimização **-02** é habilitado. Um programa *single-thread* nunca saberia a diferença entre os códigos gerados, por outro lado o impacto da reordenação em programas *multi-threadings* será sentido pois alguma sincronização relativa a ordem pode ser necessária.

```
1  int A, B;
2
3  void foo()
4  {
5      A = B + 1;
6      B = 0;
7  }
```

Código 2.3: Exemplo base de reordenação durante a compilação

Entretanto, otimização a nível de compilação não detém de informações sobre a camada superior do programa, a nível de projeto por exemplo, para saber se o programa é projetado para ser *single-thread* ou *multi-tread*. Em C a abordagem possível para prevenir reordenamento de compilação é através do uso de uma diretiva especial conhecida como **Compiler Barrier** [5], que fornece um controle da compilação efetuada no compilador GCC e não influencia no comportamento da CPU em tempo de execução. É possível ver no Código 2.4, a adição de uma barreira de memória, entre as gravações dos valores de

```
1  int A, B;
2
3  void foo()
4  {
5      A = B + 1;
6      asm volatile("" ::: "memory");
7      B = 0;
8  }
```

Código 2.4: Barreiras de memória em C

A e B. Esse tipo de barreira de memória é uma *full compiler barrier* no GCC. Com essa modificação as otimizações podem ser habilitadas e as ordens das escritas ainda serão mantidas no Código 2.5.

```
1  ...
2  mov    eas, DWORD PTR _B
3  add    eax, 1
4  mov    DWORD PTR _A, eax
5  mov    DWORD PTR _B, 0
6  ...
```

Código 2.5: Assembly resultante com barreiras de memória

Para C++ é possível emitir um subconjunto de barreiras com semântica conhecida como **acquire-release** [37] que irá impedir as reordenações. A biblioteca *<atomic>* [38] disponibiliza o método *atomic_thread_fence (memory_order_m)* que permite a passagem de parâmetros *memory_order_acq_rel*, *memory_order_acquire* e *memory_order_release*. A semântica *release* é compatível a inserção de barreiras *LoadStore* e *StoreStore* o que garante a relação de *acontece antes* entre as instruções anteriores a barreira e escritas subsequentes, e a semântica *acquire* compatível com a inserção de uma barreira *LoadStore* e *LoadLoad* que garante a relação de *acontece depois* após alguma leitura. Ainda é possível combinar ambas as semânticas com o parâmetro *memory_order_acq_rel* que apenas não irá garantir o comportamento *StoreLoad*.

Já Python é uma linguagem interpretada a qual possui algumas implementações, das mais conhecidas são: *CPython*, *Jython*, *IronPython* e *Pypy*, onde cada implementação traz consigo garantias para o gerenciamento de memória [39]. *CPython* e *Pypy* usam um mecanismo chamado Global Interpreter Lock (GIL), que garante que somente uma *thread* pode estar em estado de execução por vez. *Jython* e *IronPython* possui o modelo de gerenciamento de memória da *JMM* e *CLR* respectivamente.

Required barriers	**2nd operation**			
1st operation	Normal Load	Normal Store	Volatile Load	Volatile Store
Normal Load	-	-	-	LoadStore
Normal Store	-	-	-	StoreStore
Volatile Load	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store	-	-	StoreLoad	StoreStore

Tabela 2.2: Equivalência de barreiras de memória em Java garantidas por *volatile* [7]

Em Java é possível que o compilador execute modificações no código original ao compilar o programa para *bytecode* ou em tempo de execução pelo Just in Time Compiler (JIT). Em alguns casos algumas instruções podem ser descartadas, novas instruções podem ser geradas ou até mesmo reordenações podem ocorrer, porém sempre garantindo as relações causais já existentes em código. Entretanto não é possível que o compilador conheça as relações causais existentes entre *threads* em um programa concorrente. Portanto para que haja uma garantia das relações de ordem necessárias, a linguagem de programação Java utiliza a palavra reservada *volatile*, e não disponibiliza diretamente a utilização de barreiras de memória.

Variáveis declaradas com o modificador *volatile* possuem uma correspondência com as quatro barreiras de memória apresentadas aqui [7] e podem ser observadas na Tabela 2.2. O comportamento é semelhante a semântica *acquire-release* porém com alguns comportamentos adicionais. Uma escrita em uma variável **X** *volatile* impede reordenações com qualquer tipo de operação anteriormente efetuada, sejam elas escritas ou leituras, e também força que seja efetuado um *flush* do *buffer* de escrita para cache, ou seja, se algum núcleo consegue visualizar o valor atualizado de **X** então todas as operações anteriores foram realizadas e ou persistem em cache. Já uma leitura efetuada em uma variável **X** *volatile* impede reordenações com qualquer instrução de escrita e leitura posterior, garantindo que a leitura de **X** seja efetuada antes de qualquer instrução posterior. É importante frisar que a utilização de *volatile* possui um custo não negligenciável. Os *flushes* executados por escritas em *volatile* aumentam as paradas de CPU e as garantias de ordem em leituras *volatile* também fazem com que os valores posteriores tenham que ser recuperados novamente em memória.

2.3 Problema do Produtor e Consumidor

O problema do **produtor e consumidor** é um exemplo clássico de sincronização através de *buffers* com tamanhos limitados [13]. *Buffer* neste contexto se refere a uma região de memória compartilhada entre o produtor e o consumidor. Em sua forma mais básica o problema considera que dois processos, um produtor e um consumidor que compartilham um *buffer* de tamanho fixo. O papel do produtor é gerar um item e então inseri-lo no

buffer repetindo estas ações de forma cíclica. O papel do consumidor é remover itens do *buffer* para consumo. O problema consiste em garantir que o produtor não irá tentar inserir um item quando o *buffer* estiver cheio e que o consumidor não irá tentar remover itens do *buffer* quando este estiver vazio.

O fato de haver compartilhamento de memória também exige que seja feita sincronização no acesso da região compartilhada também conhecida como **região crítica** [40]. É importante que este fato seja levado em consideração para que não ocorram problemas de concorrência como por exemplo um produtor sobrescrevendo uma posição do *buffer* em que o consumidor ainda não terminou de ler ou um consumidor lendo um elemento do *buffer* que ainda não teve sua escrita finalizada. Ambos os casos decorrem de acesso simultâneo não exclusivo a uma variável compartilhada, o que pode gerar diferentes resultados dependendo de quando cada processo executa o trecho de código, um fenômeno também conhecido como **condição de corrida**. Estes efeitos são pronunciados principalmente em casos de múltiplos produtores e consumidores, devido a maior concorrência dos processos, e podem causar também sobrescritas e leituras duplicadas de um mesmo item.

Para se certificar de que problemas de concorrência não ocorram em regiões críticas é necessário a utilização de recursos que garantam a **exclusão mútua** no acesso aos dados. Uma das formas de garantir acesso exclusivo é pela utilização da primitiva *lock* também conhecida como *mutex*. Sua semântica dispõe de um método de obtenção e um método de liberação onde apenas o processo de posse do *lock* tem acesso a região delimitada pelos métodos. Para sua implementação são necessárias a utilização de instruções atômicas com suporte em hardware como *Test and set*, *Fetch and add* e *Compare And Swap (CAS)*.

Locks também permitem solucionar o problema de múltiplos produtores e consumidores, porém possuem custo computacional não negligenciável [8]. São necessários recursos adicionais de memória para armazenamento e tempo de CPU tanto para a obtenção quanto para a libertação de um *lock*. Também é importante notar que a granularidade da região sendo protegida pode aumentar ou diminuir a quantidade de contenção sendo gerada. O termo ***lock contention*** [14] se refere a tentativa de um processo em acessar uma região em que um outro processo já tenha tomado posse do *lock* que a protege. Uma das consequências é uma sobrecarga devido a necessidade do sistema operacional arbitrar as colisões, que podem ser frequentes devido a *spin-lock* onde um processo fica em espera ocupada tentando tomar posse ou ter que realizar chamadas de sistema para botar um dos processos em suspensão. Neste procedimento as memórias *cache* também podem ser poluídas devido a chegada de novos processos e contribuir ainda mais para o gasto de tempo de CPU devido aos efeitos de ***cache miss***.

Contudo nem todo problema de produtor e consumidor exige a utilização de *locks*.

Certas instâncias podem ser solucionadas apenas com a utilização de instruções atômicas ou também com **barreiras de memória** [12, 11, 13, 1], que possuem a função de garantir a ordem das operações observada nas modificações da memória por diferentes processos. Um dos problemas que pode ser resolvido utilizando apenas essas primitivas é o problema do **produtor e consumidor único** também conhecido como *SPSC*. A vantagem destas primitivas é o custo computacional menor e a melhor escalabilidade. É possível observar a partir da Tabela 2.3 o comportamento de um programa que faz o incremento de uma única variável onde uma ou mais *threads* concorrem executando 500 milhões de incrementos em um contador. Mesmo que não represente a complexidade e a realidade de cargas de trabalho mais comuns ainda assim é possível observar a grande diferença de performance entre CAS e barreiras com relação a *locks*.

Método	Tempo (ms)
1 Thread	300
1 Thread com barreira de memória	4700
1 Thread com CAS	5700
2 Threads com CAS	18000
1 Thread com lock	10000
2 Threads com lock	118000

Tabela 2.3: Tempos de execução em um programa com compartilhamento de memória [8]

A utilização destas primitivas não bloqueantes permite otimizar a vazão da computação realizada pelas *threads* envolvidas. No caso do problema de produtores e consumidores é possível que sejam feitas implementações ***lock-free*** e ***wait-free*** [12, 11, 13, 1] que se aproveitem da performance observada na Tabela 2.3. Ambas classificações são relativas as garantias de progresso do programa. *Lock-free* são implementações que garantem um progresso geral do programa mesmo que algumas das *threads* estejam bloqueadas ou em ***starvation*** (definido como um estado em que se espera indefinidamente por um recurso) [17]. Diferentemente, implementações *wait-free* garantem que todas as *threads* envolvidas irão fazer progresso sem que fiquem em *starvation* por tempo indefinido [17], ou seja, é garantido que terminem em um número finito de passos providenciando vazão por *thread* e não apenas global como em algoritmos *lock-free*. É importante notar que existem implementações do problema de produtor e consumidor para ambas versões, portanto é necessário se atentar a qual garantia está sendo desejada na instância do problema em questão.

2.3.1 Diferentes tipos de produtor e consumidor

Na maior parte dos casos o problema do produtor e consumidor são implementados utilizando uma semântica de filas. A fila é uma estrutura linear que segue uma ordem particular na qual as operações são realizadas. A ordem garante que o primeiro item a entrar é o primeiro item a sair. Essa ordem é importante quando um recurso é compartilhado com vários consumidores ao mesmo tempo, onde o primeiro consumidor a chegar também será o primeiro a ser servido.

Todas as classificações aqui descritas foram encontradas em [41], na prática não existe um modelo único para a representação do problema do produtor e consumidor. Isso pode ser visto, por exemplo, em sistemas concorrentes onde podem existir componentes que possuem mais produtores do que consumidores e vice-versa, ou um único produtor para vários consumidores e vice-versa. Dependendo da quantidade de *threads* de produtores e *threads* de consumidores, as filas de produtor e consumidor podem ser classificadas da seguinte maneira:

- Single Producer Single Consumer (SPSC);
- Single Producer Multi-Consumer (SPMC);
- Multi-Producer Single Consumer (MPSC);
- Multi-Producer Multi-Consumer (MPMC).

Em um cenário em que o sistema possua muitas *threads* para produtores e muitas *threads* para consumidores, podem ser utilizadas as filas MPMC. Já em cenários onde existe apenas uma *thread* para produtor e uma *thread* para consumidor, é recomendado o uso de filas SPSC, o que possibilita uma melhor performance. Cada um dos modelos acima permite otimizações específicas que não necessariamente podem ser generalizadas para todos os demais modelos, porém é importante notar que soluções MPMC podem ser utilizadas para resolver as necessidades dos outros três modelos. SPMC e MPSC podem ser utilizadas também por sistemas que exijam garantias SPSC. Por último, SPSC é o modelo mais restritivo que não permite resolver os requisitos dos demais.

Outra questão a ser considerada é a implementação da estrutura de dados da fila. A fila pode ser implementada na estrutura de um *Array*, *LinkedList* ou alguma variação das duas. Cada implementação traz consigo suas vantagens e desvantagens. As filas *Array-based* geralmente são mais rápidas, porém necessitam pré-alocar memória para o pior caso, o que pode ocasionar um tempo de inicialização alto. As filas *LinkedList* crescem dinamicamente, portanto, não há necessidade de pré-alocar qualquer memória antecipadamente, por outro lado o consumo de memória é maior na *LinkedList* em relação

ao *Array-based*, porque cada nó contém um ponteiro para o próximo elemento e podem crescer indefinidamente.

Se a escolha da fila for baseada em *LinkedLists* há mais considerações a fazer, como a intrusão da lista:

- Intrusivo;
- Não intrusivo.

Listas intrusivas possuem na própria estrutura do dado sendo armazenado os ponteiros que definem o próximo e/ou o elemento anterior. Já listas não intrusivas utilizam *containers* que encapsulam o dado armazenado fazendo com que o mesmo não necessite armazenar informações sobre a disposição dos dados na lista. Listas intrusivas geralmente possuem melhor desempenho se for necessário transferir dados dinamicamente alocados, porque não há necessidade de gerenciamento de memória adicional. No entanto, não são aplicáveis se seus dados não forem alocados dinamicamente ou se necessitar simultaneamente enviar a mesma mensagem para um número desconhecido de filas o que necessitaria duplicações.

Ainda, para filas baseadas em *LinkedList*, dependendo do tamanho máximo:

- Limitado;
- Ilimitado.

Uma fila ilimitada pode conter um número infinito de dados, enquanto a limitada apenas até algum limite predefinido. Se o limite for atingido, mais operações de enfileiramento falham. *Array-based* são sempre limitados. Embora filas ilimitadas pareçam mais vantajosas também costumam ter mais problemas com uso de memória para os casos em que a fila cresça descontroladamente. Na maioria dos casos a fila limitada é a mais utilizada, sempre tendo em mente que em algum momento uma inserção pode falhar e deverá ser tratada.

Para filas limitadas dependendo do comportamento durante o overflow:

- Falha no *overflow*;
- Sobrescreve o item mais antigo.

A maioria das filas se enquadra na primeira categoria, e a segunda opção é mais específica. Considere que um produtor envia alguns dados em tempo real e se um consumidor não conseguir recuperá-los, é melhor perder os dados mais antigos do que os mais recentes.

Há mais questões a serem consideradas durante a definição da estrutura da fila. Se a fila possui prioridade, então os consumidores sempre irão retirar da fila um elemento com

a prioridade máxima. Em geral, as filas prioritárias são significativamente mais lentas e pioram a escalabilidade do sistema podendo também gerar *starvation*. Ainda dentro das característica a serem consideradas, se for necessário que a fila possua garantia de ordem então:

- Ordem causal FIFO/Last In First Out (LIFO) (mais forte);
- Ordem FIFO/LIFO por produtor;
- Melhor esforço FIFO/LIFO (mais fraco);
- Sem garantias de ordem.

As diferenças são bastantes específicas e dependem do uso de cada uma, ou seja, em quais casos de uso você espera uma ordem específica de mensagens. Ordem causal implica em uma ordenação global de entrada e saída das mensagens. FIFO/LIFO por produtor faz com que mensagens de um mesmo produtor sejam lidas em ordem, porém não existe uma obrigatoriedade de ordenação com relação as mensagens entre produtores. Já o FIFO de melhor esforço tenta na maior parte dos casos entregar mensagens em ordem, porém eventualmente podem estar desordenadas. As soluções com o melhor esforço e as soluções que não utilizam garantias de ordem podem ser significativamente mais rápidas e escalar melhor do que as demais soluções.

2.4 Trabalhos Relacionados

Dos trabalhos mais recentes sobre os algoritmos estado da arte foram identificados o trabalho que propôs a implementação FastForward [11], aqui também conhecida como FFBuffer, e o trabalho a respeito da implementação BQueue [12]. Em [11], além de apresentar o FastForward também é introduzida a ideia de temporal *sleeping* realizando testes a partir de um processamento em pipeline montado com filas. São avaliadas as diferenças de desempenho entre a proposta inicial de Lamport e FastForward, porém não são utilizadas as otimizações mais recentes de Lamport. Também é importante ressaltar que os testes realizados em máquinas multi-núcleos concluem que FastForward é insensível ao posicionamento das *threads* em núcleos.

Já a proposta de BQueue [12] utiliza testes *dummy* apenas com inserção e remoção simples de uma quantidade de itens e contrasta com testes em uma aplicação real e acaba por concluir que o algoritmo é de 5 à 10 vezes mais rápido do que FastForward em aplicações reais, o que não se reflete nos testes simples. A maior de suas vantagens é atribuída a não necessidade de configurar um temporal *sleeping* o que torna seu uso mais

genérico. Por fim não foram realizados testes que levassem em conta o posicionamento das *threads* em núcleos de máquinas com múltiplos *sockets*.

Algumas otimizações são exploradas em certos trabalhos como otimizações para evitar *false sharing* no algoritmo de Lamport [1], com *padding* das variáveis de controle [18], batching e cache de variáveis de controle. Também são apresentados testes em máquinas com múltiplos núcleos [18, 16], porém as análises são feitas nos algoritmos propostos sem comparar com as soluções bloqueantes ou faltando alguma das filas apresentadas aqui neste trabalho. Vale ressaltar que o trabalho de Vincenzo Maffione [16] tenta caracterizar por meio de limites teóricos os comportamentos de cache e *throughput*, porém aqui será dada apenas uma ideia geral do que pode ser um comportamento esperado.

Apesar dos estudos aqui se concentrarem em filas *bounded* devido ao seu melhor desempenho se comparado as filas *unbounded*, o estudo em [15] mostra que filas SPSC *unbounded* podem também ter melhores resultados em aplicações reais do que versões *bounded* como algumas das filas utilizadas na base da biblioteca *FastFlow* de programação paralela.

De modo geral, os artigos citam que exclusão mutua utilizando *lock/unclock* é excessivamente exigente para algoritmos com alta frequência de sincronização [15, 16, 12] mas não apresentam estes algoritmos para comparação na realização de seus testes, apenas estudos relacionados. De mesma maneira também evidenciam [18] que algoritmos MPMC *wait-free* possuem desempenho muito inferior as estruturas que resolvem SPSC, porém também não são acrescidas em suas comparações.

2.5 Conclusões do Capítulo

A forma como um programa utiliza os recursos computacionais, tais como CPU e memória, é de extrema importância quando o requisito a ser considerado é desempenho. Isso por conta que os mesmos introduzem uma série de protocolos e técnicas para que possa haver um correto funcionamento. Quando considerado um programa onde a comunicação entre processos é crítico, é possível observar que esses protocolos e técnicas são ainda mais complexos devido também ao aumento de situações onde possa existir conflitos de processos concorrentes.

O problema do produtor e consumidor é um problema clássico que facilmente pode ser resolvido com a utilização de *locks*. Versões mais específicas como Single Producer Single Consumer (SPSC) podem ser resolvidas com a primitiva de barreiras de memória que geram uma menor sobrecarga de contenção. As barreiras a nível de *hardware* forçam que as demais *threads* envolvidas visualizem de forma sequencial as modificações sendo realizadas em memória cache e as barreiras a nível de *software* comunicam a necessidade desta ordenação para que os compiladores não efetuem reordenações indesejadas.

Capítulo 3

Implementações e Algoritmos do Problema do Produtor e Consumidor Únicos

Este capítulo apresenta as principais soluções e algoritmos para o problema do produtor e consumidor únicos. Também são apresentadas e discutidas otimizações para estes algoritmos.

3.1 Algoritmo de Lamport [1]

Filas FIFO de produtor único e consumidor único (SPSC) dominam o espaço de softwares embarcados [19]. Elas surgem de uma variedade de padrões de projeto paralelos e da distribuição de redes de processos Kahn em arquiteturas de multiprocessadores [19]. Seu funcionamento básico implica em um produtor posicionando objetos em uma fila e um consumidor os removendo para consumo. O raciocínio formal sobre filas limitadas SPSC remonta ao trabalho de grande influência de Lamport [1], onde ele ilustra técnicas de prova em programas concorrentes.

Nos códigos 3.1 e 3.2 estão implementadas as rotinas de *offer* e *poll* representando uma rotina chamada por um processo produtor e outra por um processo consumidor, respectivamente. As implementações são baseadas na proposta original de Lamport [1] de um vetor circular, sendo assim, em *head* será armazenado o índice de início do vetor e em *tail* o último espaço disponível na fila. A sincronização dos processos é feita inteiramente pelos índices em *tail* e *head* seguindo as relações nas equações 3.1 e 3.2, vale ressaltar que são compartilhadas entre as *threads* tanto as variáveis de controle *tail/head* quanto o *buffer* de dados. Dessa forma é possível realizar um simples cálculo para definir se existem elementos a serem consumidos e espaços para novas inserções. Caso os índices

```
1  public E poll() {
2     final long currentHead = head;
3     if (currentHead >= tail) {
4        return null;
5     }
6
7     final int index = (int) (currentHead % buffer.length);
8     final E e = buffer[index];
9     buffer[index] = null;
10    head = currentHead + 1;
11
12    return e;
13 }
```

Código 3.2: Consumidor Lamport [10]

passem dos limites do vetor devido aos incrementos, uma operação de resto em aritmética modular pela capacidade do vetor permite que todos os espaços livres sejam devidamente utilizados.

$$tail = head \rightarrow Fila\ vazia \tag{3.1}$$

$$tail - capacity = head \rightarrow Fila\ cheia \tag{3.2}$$

```
1  public boolean offer(final E e) {
2     if (null == e) {
3        throw new NullPointerException("Null is not a valid
             element");
4     }
5
6     final long currentTail = tail;
7     final long wrapPoint = currentTail - buffer.length;
8     if (head <= wrapPoint) {
9        return false;
10    }
11
12    buffer[(int) (currentTail % buffer.length)] = e;
13    tail = currentTail + 1;
14
15    return true;
16 }
```

Código 3.1: Produtor Lamport [10]

As garantias de consistência sequencial se mostram importantes em duas situações.

No caso do consumidor é importante que o mesmo enxergue o incremento de *tail* assim que o elemento a ser consumido na fila também já possua visibilidade, caso contrário um elemento não presente ou nulo pode vir a ser consumido de maneira incorreta. De mesma forma espelhada um produtor deve apenas enxergar um espaço livre na fila quando um item já tiver sido consumido e removido da fila, ou seja, não deve enxergar uma atualização de *head* sem que o item tenha sido removido. Neste caso a falta da relação sequencial pode fazer com que a escrita que apagaria o item possa vir a ser realizada após a nova atualização o que apagaria um item ainda não consumido.

Um dos problemas da consistência sequencial é que nem todos os processadores modernos garantem este tipo de visibilidade aos processos concorrentes. Linguagens como *C* e *Java* também não possuem uma semântica bem definida para situações que envolvam condições de corrida, portanto nestes casos se fazem necessárias a utilização de barreiras de memória que irão permitir, a um custo inferior a *locks*, a ordenação das leituras e escritas [42].

As principais diferenças entre as implementações de SPSC estão na sincronização do produtor e consumidor, isto é, como que o consumidor identifica que o *buffer* atual está vazio e como o consumidor identifica que o produtor já produziu um item para ser retirado do *buffer* e ser consumido [16]. Uma das formas de sincronização mais comuns envolvem a utilização de variáveis de controle com barreiras de memória de forma análoga ao proposto por Lamport [1]. Sincronizações podem também ser realizadas por variáveis atômicas como *Compare And Swap (CAS)* ou *Fetch And Add (FAA)* que apesar de possibilitarem uma implementação *lock-free* também tratam-se de instruções que exigem algum tipo de arbitragem em suas operações o que acaba gerando contenção e impõe um custo maior em relação as barreiras de memória. Entender quais alternativas de sincronização estão à disposição e porque o funcionamento da CPU as exige é de suma importância para um funcionamento satisfatório das diferentes implementações de algoritmos concorrentes.

3.2 Algoritmo FFBuffer

FFBuffer (*FastFlow Buffer*) é um algoritmo de alta performance capaz de trabalhar em modelos de memória fracas, e que não utiliza instruções de exclusão mútua, como *lock* por exemplo, para se obter sincronização entres os processos. A ideia foi originalmente apresentada por Higham and Kavalsh [43], os quais utilizam o problema de filas do produtor e consumidor para provarem sua teoria através de testes em diversos sistemas que possuíam um modelo de memória fraca. Outra característica do algoritmo apresentado por Higham and Kavalsh é que era possível fazer o controle da sincronização a partir do conteúdo da fila.

A base do algoritmo consiste em não utilizar mais a relação das equações 3.1 e 3.2 para fazer a sincronização entre os processos produtor e consumidor, de forma que a sincronização é indicada a partir do conteúdo do *buffer*, visto nos códigos 3.3 e 3.4. Seu funcionamento pode ser descrito de maneira simples, o processo produtor (Código 3.3) verifica se o espaço onde será inserido o dado se encontra livre, caso esteja o item será inserido e caso esteja preenchido significa que o *buffer* está cheio e nada será introduzido. Da mesma forma, o consumidor (Código 3.4) antes de obter um novo item irá validar se existe algo na posição desejada, caso não exista significa que não há nada pra ser lido, caso contrário será efetuado o consumo.

```
1 public boolean offer(T
      obj) {
2     if (null != data[tail])
3         return false;
4
5     data[tail] = obj;
6     tail++;
7     return true;
8 }
```

Código 3.3: Produtor FFBuffer

```
1 public T poll() {
2     var obj = data[head];
3     if (null == obj)
4       return null;
5
6     data[head] = null;
7     head++;
8     return obj;
9 }
```

Código 3.4: Consumidor FFBuffer

Essa nova lógica pode ser mais vantajosa se comparada ao algoritmo de Lamport por conta de todo o tráfego de cache (Seção 2.2.3) existente em leituras e modificações de uma variável. No algoritmo de Lamport, inevitavelmente, mesmo com a aplicação da técnica de cache de *tail/head* (Seção 3.3.2) ainda ocorrerão caches *misses* nas variáveis de controle o que já não será observado nos ponteiros de *tail/head* do *FFBuffer* uma vez que ambos são exclusivos a suas respectivas *threads*.

A não necessidade de barreiras de memória pode ser compreendida de maneira intuitiva. Caso um produtor introduza dois novos itens t e $t + 1$, independente da ordem em que forem submetidos e visíveis para as demais *threads*, o consumidor apenas irá avançar quando puder visualizar os itens mais antigos inseridos o que força o consumidor a sempre visualizar as escritas do produtor em ordem [16]. De maneira análoga, o produtor também será forçado a visualizar espaços vazios na ordem em que são esvaziados pelo consumidor e barreiras de leitura também se fazem desnecessárias uma vez que a variável de controle irá possuir o mesmo valor a ser consumido como dado, o que é garantido se manter entre ambas as leituras.

Vale ressaltar que barreiras de memória podem ser necessárias em duas situações. Em casos em que os dados sendo trabalhados em fila sejam ponteiros para estruturas externas é possível que a disponibilização/construção dos objetos ainda não tenha sido submetida

para cache e possa não estar visível para o consumidor, justamente devido a falta de uma barreira de memória para garantir a visualização dos objetos antes que as referências sejam introduzidas em fila, portanto é importante que uma barreira *StoreStore* sejam inserida antes das escritas do produtor em fila.

Em um segundo momento ainda existe a possibilidade de reordenações e otimizações realizadas por compiladores que podem gerar *loops* infinitos [44] uma vez que estas otimizações são realizadas sem conhecimento sobre as interações entre as *threads* envolvidas. Cada linguagem possui formas diferentes de realizar esta sinalização, no caso específico de Java é necessário a introdução tanto das barreiras de escrita como de barreiras de leitura [7] no produtor para que seja informado ao compilador a intenção de manter o código na ordem original.

3.3 Otimizações para SPSC

3.3.1 Lazy set, StoreStore

Uma das alternativas a utilização de *volatile* para SPSC é o uso de barreiras com garantias de ordem mais fracas [45]. No caso das escritas *volatile*, como mencionado anteriormente, é executado um *flush* do *buffer* de escrita principalmente para garantir as relações de ordem com leituras conseguintes, porém essa garantia não é necessária para o funcionamento correto do algoritmo, além de serem operações custosas pois geram tempo de CPU ocioso. Para o consumidor e o produtor é importante que uma atualização no índice de *tail* ou *head* esteja visível apenas após o item já ter sido inserido ou removido em fila o que é possível garantir reforçando a ordem das escritas submetidas a cache com barreiras *StoreStore* entre as escritas.

Vale observar que as leituras efetuadas após as escritas serão de *head* e *tail* e devem garantir consistência suficiente para que um produtor não insira itens de forma sobreposta ou que um consumidor não consuma um mesmo item já descartado. Pegando como exemplo o produtor, ao se remover o *flush* após as escritas o programa estará sujeito a visualizar valores de *head* e *tail* desatualizados que não condizem com o estado atual do vetor que já possui o elemento inserido ou removido, porém isso não prejudica o funcionamento do algoritmo em nenhum dos casos. Uma leitura de *head* dependerá apenas da consistência do que já está em cache uma vez que não são feitas alterações desta variável pelo produtor. Esta consistência já será garantida pelas barreiras *StoreStore* no consumidor que impede a reordenação das modificações de *head* e a remoção dos itens do vetor. Já para *tail*, um valor de posse do produtor que apenas ele o modifica, existe a garantia de que o programa sempre irá visualizar suas próprias modificações em memória

de forma sequencial o que impede por exemplo que *tail* seja atualizado pelo produtor e em seguida lido com um valor desatualizado pelo mesmo. Portanto sob a garantia da consistência das modificações do consumidor é impossível que o produtor sobrescreva algum valor no vetor, de forma análoga o mesmo raciocínio se aplica ao consumidor.

Sabendo que é possível dispensar os *flushes* após as escritas do consumidor e do produtor, uma das formas de se efetuar esta eliminação em Java é pela utilização da instrução *lazy-set* [46] existente em classes como *AtomicLong* que garante uma semântica de *release*, assim como em C++, ou seja, como se fossem colocadas barreiras *LoadStore* e *StoreStore* antes da operação de escrita. Ambas barreiras tendem a ser *no-ops* uma vez que a maioria das arquiteturas já suportam essas garantias sem a necessidade de emitir instruções específicas.

É importante se atentar que *lazy-set* em teoria pode acrescentar um atraso na visibilidade do valor modificado para as demais CPUs, contudo como demonstrado em [45] pelo programa *ping-pong* não é possível observar latência adicional se comparado com escritas *volatile*. O maior ganho será observado no problema do produtor e consumidor uma vez que a CPU não precisa parar a toda escrita e pode dar continuidade ao algoritmo enquanto o *buffer* de escrita é drenado ao longo do tempo pela CPU. Em [45] é possível observar um ganho de quase 6 vezes para processar uma solução simplificada do produtor e consumidor.

3.3.2 Máscara de bit k^2 - 1

A máscara de bit é uma alternativa bem menos custosa em relação ao uso do operador de resto modular "%", o qual é utilizado em algoritmos de *buffers* circulares para se localizar o valor do index no *buffer*. O operador de resto "%" pode ser substituído por um operador *AND*(&) *bitwise*, no caso de números X potências de 2, entregando melhor desempenho [47]. A operação segue a equivalência da Equação 3.3.

$$X\%N = X\&(N-1) \tag{3.3}$$

Isso funciona, pois, a representação binária para N, que é uma potência de 2, possui um único bit definido como 1. Por exemplo a representação do número 8 em binário é igual a B1000 e 8-1 é igual a B0111, quando é feito o X AND (N-1), apenas o resto da divisão de X por N corresponderá à máscara de N-1. O motivo da operação de "%" ser mais custosa está relacionada diretamente com quais instruções são utilizadas no processador para processar o resultado, geralmente instruções de resto utilizam instruções de divisão. Em arquiteturas de processadores x86, de acordo com a Tabela 3.1, instruções de *AND* possui uma latência bem menor para execução do que uma instrução de *DIV*.

Instrução	Operandos	Latência (ciclos)
DIV	r8	22-25
DIV	r16	23-26
DIV	r32	22-29
DIV	r64	32-96
AND OR XOR	r, r/i	1
AND OR XOR	m, r/i	6

Tabela 3.1: Exemplo de diferentes custos por instrução [9].

3.3.3 Thompson cache de tail/head

Tail e *Head* são variáveis compartilhadas que descrevem respectivamente a posição do consumidor e do produtor no *buffer*. Ter um baixo percentual de *cache miss* na leitura dessas variáveis é de grande proveito por se tratar de variáveis que são acessadas constantemente durante a execução de um programa SPSC. Este fenômeno ocorre de maneira frequente pois ambas as partes fazem verificações que necessitam do índice sendo modificado na outra *thread*, gerando um *cache miss* sempre que ocorrerem atualizações.

Uma primeira tentativa de diminuir a quantidade de *cache miss* é a possibilidade de remoção do modificador *volatile* em programas SPSC, permitindo que essas variáveis estejam abertas a otimizações, como por exemplo, serem salvas diretamente em um registrador sem a necessidade de fazer um acesso a memória o qual é mais lento. Entretanto a remoção do *volatile* torna o algoritmo incorreto uma vez que as relações de ordem necessárias não mais seriam atendidas.

Uma opção mais formidável para lidar com os *caches misses*, é a adição de duas novas variáveis que irão fazer cache dos valores de *tail* e *head* [16]. A nova variável, com a cache de *tail*, será usada exclusivamente na *thread* do produtor e a nova variável com a cache de *head* será usada exclusivamente na *thread* do consumidor. O Código 3.5 mostra a adição da cache de *tail* no consumidor. Essa alteração significa que ao invés de verificar a posição mais recente do valor de *tail*, será verificado apenas o valor de cache de *tail*. Isso funciona pois, não é necessário ter o valor mais recente de *tail*/*head* para que o programa faça progresso, é suficiente saber que a próxima posição está disponível para ser usada sem a necessidade de resgatar o valor mais atual na cache da outra *thread*. Desta forma é possível evitar todo o tráfego de protocolo de cache (Seção 2.2.3). Neste caso um *cache miss* irá acontecer somente quando for identificado no valor da cache do índice que possivelmente não há mais espaço disponível para progredir com o programa e será necessária uma nova sincronização da cache de índice com o valor mais atual do índice.

```
1  public E poll() {
2     final long currentHead = head.get();
3     if (currentHead >= tailCache.value) {
4        tailCache.value = tail.get();
5        if (currentHead >= tailCache.value) {
6           return null;
7        }
8     }
9
10    final int index = (int) currentHead & mask;
11    final E e = buffer[index];
12    buffer[index] = null;
13    head.lazySet(currentHead + 1);
14
15    return e;
16 }
```

Código 3.5: Consumidor com cache de *tail*

3.3.4 False sharing e Padding

Dependendo de como um objeto é carregado em memória um programa *multi-thread* pode estar sujeito a ocorrência de **false sharing** [18]. Este fenômeno ocorre quando alguma *thread* tenta alterar um dado onde a linha de cache contendo as informações a serem atualizadas também possui dados compartilhados que não fazem parte da atualização, como na Figura 3.1. Consequentemente as *threads* que compartilham esta linha serão forçadas a invalidar e requisitar novamente a linha de cache mesmo que seus dados sendo utilizados não tenham sido alterados. Como já mencionado os sistemas de gerenciamento de cache utilizam as linhas de cache como unidade mínima e portanto não possuem conhecimento destas falsas colisões.

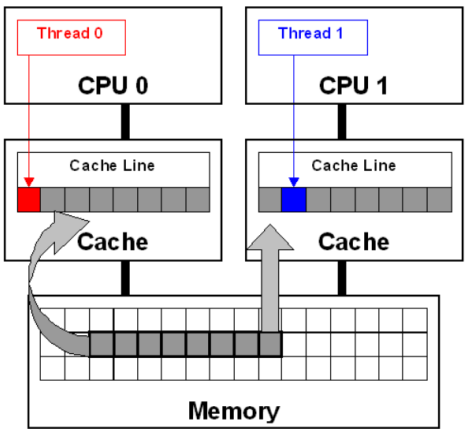


Figura 3.1: Visualização de *false sharing* com latência em ciclos de CPU [6].

Uma das consequências do *false sharing* é a quantidade desnecessária de cache *misses* que podem degradar a performance do programa. Porém nem sempre é simples identificar os reais causadores uma vez que o problema de *false sharing* é altamente dependente de como os dados do programa estão sendo dispostos em memória, o que varia de execução para execução e dificulta diferenciar as fontes do problema. Como forma de prevenção é importante se atentar aos casos mais comuns de *false sharing* e entender também como a linguagem de programação sendo utilizada normalmente faz a utilização e disposição dos dados em memória. Na maioria dos casos este problema pode ser evitado fazendo com que variáveis pertencentes a dois dos grupos diferentes abaixo nunca residam na mesma linha de cache do processador, considere duas linhas de execução P e C [16]:

1. Variáveis modificadas apenas por um P e lida tanto por P e C.
2. Variáveis modificadas apenas por um C e lida tanto por P e C.
3. Variáveis apenas de leitura utilizadas tanto por P e C.
4. Variáveis exclusivas de P, acessadas apenas por P.
5. Variáveis exclusivas de C, acessadas apenas por C.

Uma das formas de lidar com *false-sharing* é com a utilização da técnica de **padding** [18, 48]. Para cada situação de potencial *false-sharing* é possível alocar memória adicional para que os dados envolvidos não residam mais na mesma linha de cache. É importante se atentar que cada linguagem de programação pode apresentar diferentes *layouts* de organização de memória que devem ser levados em conta.

Em Java por exemplo, cada implementação de Java Virtual Machine (JVM) possui características bem definidas que permitem entender como os objetos são dispostos em memória, e consequentemente seu alinhamento em memória. Na *Hotspot JVM* [49] todos objetos possuem informações de cabeçalho que são divididas em duas partes, a primeira chamada de *Mark Word*, é utilizada para armazenar informações de execução do próprio objeto como: *hashcode*, informação de *lock*, *GC metadata*, etc. A outra informação de cabeçalho é a *Klass Word*, que encapsula informações de classe a nível de linguagem, como nome de classe, seus modificadores, informações de superclasse e assim por diante. *Arrays* possuem informação adicional referente ao seu tamanho. Se necessário é adicionando uma informação de alinhamento com *padding*.

Cada objeto é alinhado a um limite de granularidade de 8 bytes. Um dos objetivos deste alinhamento é evitar acessos desalinhados que podem gerar consultas adicionais para montar determinado dado devido a granularidade de acesso de certas CPUs. Em alguns casos, determinadas arquiteturas podem não possuir suporte a acesso desalinhado. Portanto os campos do objeto podem apresentar ordem diferente do que foi declarado

para seguir tanto alinhamento do objeto quanto de seus campos. A seguir seguem algumas regras do layout de um objeto em Java [50]:

1. Todo objeto possui alinhamento de 8 bytes;
2. Se uma classe estende outra, os campos da classe pai sempre serão colocados em memória antes dos campos da classe filha.
3. Todos os campos devem estar em uma posição de memória com endereço múltiplo do tamanho de seu tipo:
   - doubles (8) e longs (8)
   - ints (4) e floats (4)
   - shorts (2) e chars (2)
   - booleans (1) e bytes (1)
   - referências (4/8) dependente de arquitetura.

Complementarmente, a Figura 3.2 mostra um exemplo de como o *layout* de um objeto pode aparecer em memória. É possível notar que os primeiros 8 bytes descrevem *Mark Word*, *Klass Word* e os últimos 4 bytes são adicionados para manter o tamanho do objeto como múltiplo de 8. Sabendo como um objeto é disposto em memória é possível declarar campos extras que irão adicionar espaçamento suficiente para não residirem na mesma linha de cache. Como podem ocorrer reordenações no layout de uma classe é possível garantir o *padding* a partir das relações entre sub-classes e super-classes, ou seja, realizando extensões e distribuindo os campos entre as classes da hierarquia.

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 0x0000000800000000 base address and 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

org.openjdk.jol.samples.JOLSample_01_Basic$P1C1QueueOriginal1 object internals:
OFF  SZ                                       TYPE DESCRIPTION                    VALUE
  0   8                                            (object header: mark)          N/A
  8   4                                            (object header: class)         N/A
 12   4                            java.lang.Object[] P1C1QueueOriginal1.buffer     N/A
 16   8                                       long P1C1QueueOriginal1.tail       N/A
 24   8                                       long P1C1QueueOriginal1.head       N/A
 32   4   org.openjdk.jol.samples.JOLSample_01_Basic P1C1QueueOriginal1.this$0     N/A
 36   4                                            (object alignment gap)
Instance size: 40 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

Figura 3.2: Exemplo de layout de memória em um programa em Java.

Para SPSC, é possível observar alguns casos que potencialmente podem gerar *false sharing*. Um primeiro caso a se observar é a proximidade dos índices de *head* e *tail* onde o

consumidor ao atualizar *tail* irá forçar um cache *miss* no produtor quando realizar a leitura ou modificação de *head*. Como mencionado anteriormente, uma possível solução agnóstica a JVM, é a separação dos campos por uma hierarquia de classes que irá introduzir variáveis não utilizadas apenas com o intuito de acrescentar espaçamento entre os índices [50].

Também poderá ser necessária a realização de *padding* entre o *buffer* e os índices da fila, mesmo que o *buffer* seja apenas um objeto de leitura ainda é possível que escritas nos índices possam gerar *false sharing* [6]. Além do *padding* entre os campos da classe já citados, a própria instância da classe pode estar sujeita a dividir linhas de cache com outros objetos em memória, portanto também é importante a aplicação de *padding* no inicio do objeto e no final após todos os campos. Vale observar que ainda assim o objeto estará sujeito a *false sharing* uma vez que o início de qualquer objeto em Java irá conter metadados que constantemente serão lidos pela JVM e podem acabar compartilhando linha de cache com dados do programa que realizam escritas frequentes.

Outro detalhe a se observar é que todo vetor declarado em Java será alocado dinamicamente, portanto os dados do vetor estarão em região de memória distinta da classe instanciada. Neste caso é possível forçar um *padding* do vetor alocando espaço extra com itens adicionais no fim e no inicio, o que irá gerar itens não utilizados em ambas as pontas do vetor [6]. Assim como o layout das classes, os vetores também possuem os mesmos metadados, além de seu tamanho, que serão alocados no início do objeto em memória. Os espaços extras alocados no inicio ainda assim não irão proteger de *false sharing* com outros objetos em memória, uma vez que serão alocados após os metadados, mas irá evitar que atualizações nos objetos do inicio do vetor interfiram na leitura dos metadados. Este procedimento ainda assim será útil uma vez que estes metadados são utilizados com bastante frequência uma vez que a JVM efetua checagens sempre que um elemento é acessado tanto para escrita como para leitura.

Para os itens individuais no vetor também é importante observar que as modificações podem gerar *false sharing* principalmente em casos em que o vetor está quase vazio e os índices de *head* e *tail* se encontram próximos o que pode ser recorrente para filas que se encontram frequentemente com poucos itens. Como resolução é possível utilizar apenas determinadas posições do vetor espaçadas por algum valor que impeça que os itens úteis residam em mesma linha de cache, como exemplo apenas poderiam ser utilizados os índices múltiplos de 16 o que em java deixaria espaços de 60 bytes entre itens supondo a utilização de um vetor de objetos (4 bytes de referência).

A técnica de *padding* contudo pode ser um grande detrimento ao *throughput* da memória cache [6]. Ao alocar os elementos extras a densidade das linhas de cache estarão sendo reduzidas o que diminui a probabilidade de dados estarem em mesma linha de cache porém a consequência será uma menor quantidade de dados úteis resgatados por leitura

```
1  adjust_slip() {
2      dist = distance(producer, consumer);
3      if (dist < DANGER) {
4          dist_old = 0;
5          do {
6              dist_old = dist;
7              spin_wait(avg_stage_time * ( (GOOD+1) - dist) );
8              dist = distance(producer, consumer);
9          } while (dist < GOOD && dist_old < dist);
10     }
11 }
```

Código 3.6: Rotina para ajuste de *temporal slipping* [11]

de linha de cache efetuada. Portanto executar *padding* de todos os casos de *false sharing* pode ser prejudicial ao desempenho do programa uma vez que estará sendo feita uma redução do espaço útil de cache. *Padding* em excesso tende a aumentar a quantidade de leituras necessárias realizadas pela cache e até mesmo amplificar o número de caches *miss* criando uma situação contrária a intenção inicial.

3.3.5 Temporal Slipping

Uma das maneiras de evitar o desperdício de memória com *padding* e os problemas de modificações do *buffer* em mesma linha de cache é forçar um distanciamento entre produtor e consumidor para diminuir os momentos em que se encontrem próximos. A técnica de *temporal slipping* [11] tenta introduzir *delay* por meio de uma espera ocupada em momentos chaves de modo que o produtor ganhe tempo para preencher o buffer com mais objetos.

Uma das propostas de *temporal slipping* pode ser observada no código 3.6. A rotina apresentada ao verificar uma possível colisão de linha de cache por meio do valor em *DANGER* (tamanho da linha de cache da arquitetura) realiza sucessivas esperas com tempo especulado próximo ao que seria necessário para se obter um distanciamento suficiente definido em *GOOD* (valor próximo a duas linhas de cache). Vale ressaltar que o método também verifica se houve diferença com a distância passada obtida para que um consumidor não seja travado caso o produtor não tenha inserido novos itens após a espera. O código de ajuste pode ser executado assim que uma operação de remoção pelo consumidor falha o que garante mais tempo ao produtor para introduzir novos objetos ou também a cada N iterações do código para realizar a manutenção do distanciamento.

Vale ressaltar que essa solução é enviesada para processamentos em *stream* uma vez que se introduz latência para diminuir a quantidade de tráfego da cache em uma tentativa de reduzir as trocas de linhas a apenas uma leitura e aumentar o *troughput* [11]. Além

```
1  BOOL offer(ELEMENT_TYPE value) {
2      if (head == batch_head) {
3          if (buffer[MOD(head + BATCH_SIZE)])
4              return FAILURE;
5          batch_head= MOD( head + BATCH_SIZE);
6      }
7      buffer[head] = value;
8      head = NEXT(head);
9      return SUCCESS;
10 }
11
12 BOOL poll(ELEMENT_TYPE * value) {
13     if (tail == batch_tail) {
14         if (backtrack_deq() != SUCCESS)
15             return FAILURE;
16     }
17     *value = buffer[tail];
18     buffer[tail] = NULL;
19     tail = NEXT(tail);
20     return SUCCESS;
21 }
```

Código 3.7: Algoritmo B-Queue original [12]

disso um *fine tunning* se mostra necessário devido as diferentes frequências de operação de consumo/produção que uma aplicação específica pode conter implicando em diferentes decisões de quando rodar o ajuste e quais valores de espera podem ser ótimos. Uma solução apresentada na seção 3.4 chamada de BQueue tenta simplificar o algoritmo aplicando o ajuste sempre que o consumidor é chamado e realizando uma otimização com busca binária para evitar o cálculo de distância na função *distance* que também acaba por incorrer *cache-misses* devido a necessidade de *head/tail* para o cálculo.

3.4 Algoritmo BQueue

B-queue ou *BQueue* [12] é um algoritmo *lock-free* (Código 3.7) para a resolução de SPSC. A sincronização entre o produtor e consumidor no acesso ao *buffer* é obtida de forma similar ao que acontece no algoritmo de *FFBuffer*, através do controle de conteúdo do *buffer* sem a necessidade dos índices de *head* e *tail*. O algoritmo promete um desempenho equiparável ao *FFBuffer* uma vez que se trata de uma melhoria orientada a processamento de *streams*. Para atingir seus resultados são necessários dois métodos apresentados em [12], *batch operation* e *backtracking* que permitem minimizar ocorrências de *false sharing* em casos em que o produtor e o consumidor estão próximos.

```
1  BOOL backtrack_deq() {
2      // batch_history permite iniciar de um lugar perto de um
           batch candidato
3      if (batch_history < BATCH_MAX) {
4          batch_history = MIN(BATCH_MAX, batch_history +
               INCREMENT);
5      }
6      batch_size = batch_history;
7      batch_tail = MOD(tail + batch_size - 1);
8
9      while(!buffer[batch_tail]) {
10         spin_wait(TICKS);
11         if (batch_size > 1) {
12             batch_size = batch_size >> 1;
13             batch_tail = MOD(tail + batch_size - 1);
14         } else {
15             return FAILURE;
16         }
17     }
18     batch_history = batch_size;
19     return SUCCESS;
20 }
```

Código 3.8: Pseudocódigo função de backtracking [12]

3.4.1 Batching e Backtracking

Do ponto de vista do produtor a ideia básica do método de *batching* consiste em viabilizar a detecção de um lote de espaço disponível no *buffer* com tamanho *BATCH_SIZE* com intuito de reduzir a quantidade de acessos de leitura em memória compartilhada durante a sincronização do acesso ao *buffer*. A variável *batch_head* irá armazenar qual a posição após a identificação de um *batch* vazio até onde se pode fazer progresso sem a necessidade de leituras do *buffer* compartilhado.

Para o consumidor será feita uma comparação (linha 14 do Código 3.7), para verificar se há algum lote definido com possíveis dados disponíveis. Se não houver, será realizada uma sondagem através da função de *backtracking* do Código 3.8. Dentro desta função a variável *batch_size* será inicializada com o valor de *BATCH_SIZE* e a variável *batch_tail* com o valor de *BATCH_SIZE* a frente de *tail* para encontrar um novo tamanho de lote disponível para leitura. Para cada execução o consumidor irá verificar se há algum conteúdo em *buffer[batch_tail]* (linha 9), onde *buffer[batch_tail]* é a borda do lote. Se não for encontrado nenhum conteúdo então o valor de *batch_size* é diminuído pela metade e o valor de *batch_tail* é recalculado novamente, repetindo este processo até que seja encontrado um valor em *buffer[batch_tail]*, ou que *batch_size* alcance o valor zero.

O algoritmo *BQueue* é parametrizado com as variáveis *TICKS*, *BATCH_MAX* e *BATCH_SIZE*, e pode apresentar diferentes configurações que podem ser mais vantajosas

para seu desempenho. *TICKS* representa a quantidade de ciclos do processador em que o consumidor irá aguardar até que possa executar o método de *backtracking* novamente. O *BATCH_MAX* define um limite inicial para o *batching* do produtor, e *BATCH_SIZE* o tamanho do *batching* do consumidor. Testes executados pelo autor Nitsan [51], demonstram que a combinação (TICKS=80), (BATCH_MAX=32) e (BATCH_SIZE = 8192) mantém o algoritmo com uma boa estabilidade (levando em consideração *throughput* de *streams*), possibilitando acompanhar o desempenho de outros algoritmos otimizados como o *FFBuffer*.

Uma das vantagens desta proposta está no fato de que tudo sendo escrito em um *batch*, pode ser armazenado em *buffer* de escrita sem a necessidade de sincronização constante com a cache, evitando por um pequeno período de tempo acessos a L1 até que o *batch* seja completamente preenchido. Caso o consumidor esteja próximo em mesma linha de cache efetuando remoções e leituras, não serão efetuados caches *misses* constantes pois o produtor não estará submetendo informação para cache e nem efetuado leituras que incorram em caches *misses*.

Ao se evitar os caches *misses* quando produtor e consumidor se encontram próximos ainda há a vantagem de não necessitar de *padding* do vetor o que permite a utilização de menos memória, uma vez que este *padding* é o mais custoso em termos de memória devido ao tamanho dos *buffers* alocados. A proximidade dos elementos em *buffer* também acrescenta maior densidade aos dados em cache e tende a gerar menos tráfego na memória pois cada leitura traz mais informação que possivelmente será utilizada em futuras iterações. Vale ressaltar que até *BATCH_SIZE* elementos poderão ser desperdiçados (livres porém impossíveis de alocar) uma vez que esta quantidade se torna o mínimo necessário livre para que o produtor tenha progresso.

O algoritmo *BQueue* também tenta fazer com que o consumidor tenha mais folga para consumir os dados do *buffer* tanto a partir do *batching* quanto das esperas ocupadas em *backtracking*. Essa folga é benéfica para o consumidor pois podem ser utilizadas otimizações existentes em hardware, tais como *prefetching* [51], fazendo com que haja mais dados a serem consumidos já sincronizados em cache reduzindo a quantidade de tráfego em memória compartilhada, sendo este também um benefício natural do *backtracking* pois ao realizar as verificações também acabam sendo trazidos dados para a memória do consumidor que logo mais poderão ser consumidos.

3.5 Conclusões do capítulo

Nesta seção foram mostradas diversas soluções para SPSC. Os algoritmos apresentados destacam otimizações que podem ser usadas para melhorar o desempenho relativo a pri-

meira versão do algoritmo de Lamport. Todas as otimizações são feitas a nível de software, com o intuito de otimizar o uso de memória cache e da CPU. O entendimento dessas otimizações é crucial para compreender as diferenças no desempenho de cada algoritmo.

Capítulo 4

Experimentos

De forma a evidenciar os diferentes resultados para as otimizações estudadas na Seção 3.3, foram selecionados alguns algoritmos que solucionam o problema do produtor e consumidor com diferentes estruturas de dados e técnicas de otimização. Foram selecionadas as estruturas nativas mais comuns já disponíveis em Java para a resolução do problema do produtor e do consumidor, a versão básica *wait-free* para SPSC proposta por *Lamport* e as versões otimizadas já apresentadas no capítulo anterior. É importante notar que as filas da biblioteca Java são implementações MPMC e também bloqueantes, com exceção apenas da *Concurrent Linked Queue* que é um algoritmo *wait-free* como as demais filas SPSC apresentadas.

As filas nativas em Java foram selecionadas para tentar compreender qual a diferença entre as soluções disponíveis na biblioteca padrão assim como compará-las com as implementações mais especializadas de SPSC. Uma solução básica de *Lamport* apresentada na Sessão 3.1 foi escolhida a título de comparação para se entender os impactos das otimizações apresentadas e como se comportam em diferentes cenários sobre os parâmetros propostos.

4.1 Ambiente Experimental

Para a realização dos experimentos foram utilizados recursos computacionais da plataforma **Emulab**. Emulab é uma plataforma de experimentação que pode ser utilizada para a depuração, desenvolvimento e avaliação de sistemas de softwares, através da criação de ambientes com características topológicas e hardwares previamente selecionados pelo usuário. A plataforma é largamente utilizada por pesquisadores de ciência da computação, nas áreas de redes e sistemas distribuídos para execução de experimentos.

Experimentos podem ser criados a partir de uma interface web sendo possível especificar a topologia de uma rede arbitrária, selecionar máquinas virtuais ou máquinas físicas,

com hardwares da escolha do usuário a partir de uma lista predefinida e também selecionar o sistema operacional que será instalado na máquina anteriormente selecionada. É concedido acesso *root* ao usuário, de forma que o ambiente experimental possa ser controlável, previsível e manipulável.

Para a extração de dados da análise de desempenho entre diferentes algoritmos *SPSC*, é necessário a definição de uma máquina que atenda aos requisitos do experimento. Dentre os requisitos necessários para se executar o experimento é levado em consideração se a máquina contém hardware que dê suporte a processamento em diferentes *sockets* e em diferentes núcleos. Essas características são importantes para extrair dados de execução no modo *Same Core (SC)*, *Cross Core (CC)* e *Cross Socket (CS)*. O servidor *Dell Poweredge R430 - D430* utilizado durante os experimentos, contém as seguintes características:

- Máquina física;
- **2x** processador com arquitetura **Haswell** Intel Xeon E5-2630v3 64-bit com **8 núcleos físicos** a frequências de 2.4 GHz e 16 *threads*. L1 contém 8x32 KB para armazenar instruções e 8x32 KB para armazenar dados, L2 contém 8x256 KB e L3 possui 20 MB;
- 64 GB 2133 MT/s DDR4 RAM (8 x 8GB modules);
- Ubuntu12-64-STD;
- Open JDK 11.0.11.

O fato do servidor possuir dois processadores, permite uma configuração para que o programa possa ser executado no modo CS. Durante os testes o sistema operacional é utilizado em modo *root*, e o acesso ao servidor é feito através do protocolo SSH. Para a instalação das dependências necessárias é executado um script *.sh* em modo *root* que contém os programas com suas respectivas versões. O código do programa que contém os algoritmos que serão testados, assim como o arquivo de *makefile* necessário para definição dos parâmetros utilizado durante os experimentos são obtidos através de um repositório do *Github* na seguinte url `https://github.com/NurCorezzi/SPSC/`, que são baseados na implementação original do autor Nitsan Wakart em `https://github.com/nitsanw/examples`.

4.2 Ferramentas Utilizadas

4.2.1 Profiling

Para a obtenção das métricas a serem observadas algumas ferramentas de *profiling* foram consideradas, entre elas *Intel VTUNE Amplifier*, *OProfile*, *Likwid* e *Perf Tools*. Como todas as ferramentas observadas utilizam dos mesmos registradores de eventos de *hardware* ao se considerar a simplicidade da ferramenta e a disponibilidade em sistemas Linux, optou-se pela utilização do *Perf*. Como os eventos observados foram puramente relacionados a *hardware* não foram consideradas as demais funcionalidades disponíveis nas outras ferramentas.

O **Perf** é uma ferramenta de *profiling* utilizada para realizar análises dinâmicas de programas, coletando métricas em tempo de execução. Foi especificamente projetada para sistemas Linux, a partir da implementação da interface *perf_event* disponível no *kernel*, permitindo uma coleta de dados que abstraia o *hardware* utilizado.

A ferramenta se baseia na contagem de eventos observáveis que podem vir de diferentes fontes. A nível de *software* são mantidos contadores no *kernel* do sistema operacional que são acessados pelo *perf*, como por exemplo *page-faults*, *cpu-clock* e *context-switches*. Já a nível de *hardware* a maioria dos processadores modernos possuem unidades dedicadas ao monitoramento em baixo nível, chamadas de Performance Monitoring Unit (PMU). Estas unidades possuem registradores para selecionar os eventos a serem observados e contadores PMC com os valores obtidos que são acessados pelo *perf*, vale ressaltar que cada processador possui em sua especificação os eventos que podem ser monitorados e como podem ser acessados via *Perf* por meio dos códigos hexadecimais fornecidos.

Para a obtenção dos dados foram utilizados o comando de *stat* que agrega os valores dos contadores pelo período de vida do programa monitorado e resume em um arquivo estruturado em *csv* os dados obtidos no seu término. Um processo do *Perf* foi executado no início de cada *thread* consumidor e produtor utilizando a flag *-t* que permite a partir do id da *thread* o seu monitoramento isolado, desta forma se evita agregar dados não relacionados a execução dos algoritmos que pertencem inerentemente ao *setup* experimental necessário.

4.2.2 Isolamento de núcleos

Para permitir os testes com isolamento de núcleos da CPU foi utilizada a biblioteca **OpenHFT Java Thread Affinity** [52]. Esta biblioteca possui como dependência a *Java Native Access (JNA)* que permite a chamada de funções nativas apenas com código java.

Para que a biblioteca obtenha o layout e informações de CPU são lidas as informações comumente localizadas nos arquivos */proc/cpuinfo* em sistemas Linux.

Para utilizá-la basta que sejam feitas devidamente as requisições de *locks* de núcleos sempre seguida de liberação posterior. Aqui vale observar que para evitar os efeitos de *hyper-threading*, onde dois núcleos virtuais apontam para um mesmo núcleo físico, foram definidos apenas núcleos virtuais específicos que poderiam ser alocados nas requisições de *locks*. A biblioteca permite por meio do parâmetro *-Daffinity.reserved* a passagem de uma máscara hexadecimal para indicar os núcleos alocáveis. Como cada processador possui um layout distinto e localiza núcleos virtuais de diferentes maneiras, para cada máquina foi necessária uma análise prévia dos arquivos */proc/cpuinfo* para assim permitir apenas núcleos físicos distintos e possibilitar ao mesmo tempo a utilização de núcleos entre *sockets* diferentes.

4.2.3 Análise de dados

Para facilitar a manipulação dos dados foi utilizada a linguagem **R** para análise de dados estatísticos com auxílio do ambiente de desenvolvimento **RStudio** para permitir melhores visualizações. Ao término da execução de um experimento, todos os dados serão salvos em arquivos separados e nomeados de acordo com o algoritmo, a *thread* de origem (produtor/consumidor) e os parâmetros de entrada fornecidos ao programa apenas com o objetivo de se ter maior rastreabilidade das métricas obtidas. O *script* em *R* irá agregar todos os dados dos diferentes arquivos *csv* em uma única estrutura, realizar a criação das colunas de cache *hit* que não são entregues diretamente pelo *Perf* e agregar os valores necessários para os cálculos de desvio padrão e geração dos gráficos.

4.3 Implementação

O programa em execução foi dividido em três *threads*. A *thread* principal no Código 4.1 é responsável por obter os parâmetros de entrada (qual fila, tamanho da fila, modo dos núcleos e tamanho do *padding*), instanciar a fila desejada, rodar os experimentos por um número específico de repetições em *RUNS* e adicionar aos arquivos *csv* as demais métricas não obtidas pelo *Perf*. Vale ressaltar que após a execução de todos os testes apenas os valores dos últimos 10 testes serão coletados para evitar a alta variação durante o período de *warm-up* na execução dos primeiros casos de teste.

Cada chamada de *performanceRun* irá criar as duas outras *threads* uma para o produtor e outra para o consumidor e ficará em *wait* esperando o término de ambas. Uma aproximação das rotinas pode ser observada no Código 4.2. Um teste será composto por um número fixo de *REPETITIONS* em que um produtor irá inserir um objeto *TEST_VALUE*

```
1  public static void main(final String[] args) {
2      // Extrair parametros de entrada
3      // Criar fila especificada
4      final Queue<Integer> queue =
           SPSCQueueFactory.createQueue(queueSelected,
           queueSzPowerOf2, sparceShift);
5
6    final RunData[] results = new RunData[RUNS];
7    for (int i = 0; i < RUNS; i++) {
8       System.gc();
9       results[i] = performanceRun(i, queue, coreMode);
10   }
11
12   // Obter os ultimos 10 resultados
13   for(int i = RUNS - 10; i < RUNS; i++) {
14       // Escrever resultados em csv
15   }
16 }
```

Código 4.1: Pseudocódigo da rotina da *thread* principal dos experimetos

e o consumidor irá remove-lo da fila, vale ressaltar que o objeto será sempre o mesmo instanciado no início do programa. No começo da rotina será necessário iniciar um processo do *Perf* passando o identificador da *thread* em execução para realizar o monitoramento. Algumas inicializações de contadores também serão efetuadas para obter as métricas de tempo de execução e tempo de espera. O tempo de execução foi obtido diretamente pelo programa devido a maior facilidade de obtenção se comparado ao *Perf*. Já o tempo de espera não é possível ser obtido de maneira simples por meio da ferramenta de *profiling* portanto também foi calculado nas *threads*.

Para cada *thread* de produtor e consumidor sendo executada, também é necessário passar no construtor uma referência para o *lock* do núcleo desejado que será obtido no início da rotina e liberado no final (para que possam novamente ser alocados) como no Código 4.2. Para cada modo de núcleo desejado foram utilizadas as estratégias de seleção do *Thread Affinity* no Código 4.3. Tendo como base um *lock al* com um núcleo já em *lock* a política *SAME_SOCKET* irá necessariamente selecionar um núcleo virtual distinto disponível em mesmo *socket*, como a quantidade de núcleos físicos sempre excede as três requisições efetuadas sempre haverá núcleos distintos não alocados para se obter nos casos de Cross Core. A política *SAME_CORE* irá efetuar a seleção do mesmo núcleo relativo já alocado, é importante notar que aqui não haverá um *dead-lock* pois os *locks* sendo efetuados se tratam de chamadas ao sistemas operacional que definem junto ao escalonador uma CPU para a qual o processo deverá ser alocado. Por último a política *DIFFERENT_SOCKET* caso identifique mais de um *socket* na máquina irá alocar um núcleo em um *socket* distinto.

```
1  public void run() {
2     lock.bind();
3      // Iniciar instancia do perf
4      // Iniciar contadores de metricas
5     long i = REPETITIONS;
6     do {
7        while (!queue.offer(TEST_VALUE)) {
8           nanoYield = System.nanoTime();
9           Thread.yield();
10          yield += System.nanoTime() - nanoYield;
11       }
12    } while (0 != --i);
13     // Mandar sinal para encerrar o perf
14     // Coletar dados de tempo em espera e tempo de execucao
15    lock.release();
16 }
```

Código 4.2: Pseudocódigo da função de execução da *thread* de produtor

```
1  AffinityLock al = AffinityLock.acquireLock();
2  AffinityLock plock = null, clock = null;
3  switch (mode) {
4     case "cc": {
5        plock =
             al.acquireLock(AffinityStrategies.SAME_SOCKET);
6        clock = al.acquireLock(AffinityStrategies.SAME_SOCKET);
7        break;
8     }
9     case "cs": {
10       plock =
             al.acquireLock(AffinityStrategies.DIFFERENT_SOCKET);
11       clock = al.acquireLock(AffinityStrategies.SAME_SOCKET);
12       break;
13    }
14    case "sc": {
15       plock = al.acquireLock(AffinityStrategies.SAME_CORE);
16       clock = al.acquireLock(AffinityStrategies.SAME_CORE);
17       break;
18    }
19    default:
20       throw new Exception("Core mode not defined");
21 }
```

Código 4.3: Aquisição dos *locks* de núcleos

4.4 Estruturas utilizadas

A menos das estruturas nativas em Java, todas as implementações foram obtidas em `https://github.com/nitsanw/examples` implementadas por Nitsan Wakart.

4.4.1 Array Blocking Queue

O algoritmo *Array Blocking Queue* é uma estrutura que contém sincronização bloqueante e sua implementação é nativa da biblioteca do *java.util*. Sua estrutura é baseada em uma fila bloqueante, e o acesso aos elementos é feito baseado em FIFO. A cabeça da fila é o elemento que está a mais tempo na fila e a cauda o elemento que está a menos tempo. Novos elementos são inseridos na cauda da fila, e a remoção de elementos é feito a partir da cabeça da fila, mantendo o comportamento de um *buffer* circular.

4.4.2 Linked Blocking Queue

Esta implementação, também nativa da biblioteca *java.util*, utiliza de uma estrutura de lista encadeada FIFO com ponteiros para cabeça e cauda e ordem de inserção e remoção semelhante aos *buffers* circulares. Tipicamente possuem um maior *throughput* se comparada a versão baseada em *arrays* pois existe uma separação entre os *locks* para acesso da cabeça e da cauda, porém apresentam resultados que podem variar principalmente em sistemas concorrentes devido a utilização da cache que pode introduzir casos de *false-sharing* devido a alocação dinâmica e esparsa dos elementos.

4.4.3 Concurrent Linked Queue

Este é um algoritmo de fila ilimitado baseado em elementos encadeados nativo da biblioteca *java.util*. Essa fila também ordena os elementos em FIFO. A implementação deste algoritmo, baseado no trabalho [53], adota a característica de ser *wait-free* a partir da utilização de operações atômicas *compare and swap* (CAS), portanto devido a maior velocidade destas operações, como observado na Tabela 2.3, é esperado que tenha resultados distintos se comparado a sua versão bloqueante *Linked Blocking Queue*.

4.4.4 Lamport Simples

A estrutura chamada *Lamport Simples*, implementa a solução inicial proposta por *Lamport* na Seção 3.1 sem as otimizações discutidas anteriormente, mantendo o comportamento de um *buffer* circular. A principal otimização oferecida pelo algoritmo para o problema

do produtor e consumidor consiste no fato de não usar nenhum mecanismo bloqueante para a sincronização dos processos envolvidos sendo este um algoritmo *wait-free*.

4.4.5 Lamport Otimizado

A estrutura do Lamport Otimizado, é uma evolução do Lamport Simples o qual contém algumas das principais técnicas apresentadas na Seção 3.3:

- *Lazy set*: utilizada no armazenamento dos elementos no *buffer*;
- Máscara de bit k^2 - 1: usado no lugar da operação de módulo do Lamport Simples, para calcular a posição atual dos índices;
- Thompson cache de tail/head: Utilizada nas variáveis de índices de *tail* e *head*;
- *Padding*: Utilizado tanto nas variáveis de controle quanto entre os elementos do *buffer*.

4.4.6 FFBuffer

O *FFBuffer* apresentado na Seção 3.2 propõe um controle dos dados em *buffer* de forma distinta ao que foi feito por *Lamport* e possui as seguintes técnicas em sua implementação:

- Consulta da disponibilidade de espaços a partir do conteúdo do *buffer* evitando leitura/escrita em variáveis auxiliares de controle;
- *Lazy set*: utilizado no armazenamento dos elementos no *buffer*;
- *Padding*: assim como no *Lamport Otimizado* utilizado para espaçar os elementos no *buffer* para manter o alinhamento dos dados com as linhas de cache.

4.4.7 BQueue

O *BQueue*, apresentado na Seção 3.4, possui as seguintes técnicas de otimização em sua implementação:

- *Batching*: utilizado tanto pelo produtor quanto pelo consumidor, para detectar um lote de *slots* disponíveis por vez, reduzindo o número de acessos em memória compartilhada.
- *Backtracking*: utilizado dinamicamente para adaptar o tamanho do lote de acordo com a velocidade do produtor em tempo de execução o que também possibilita evitar o desperdício de memória em decorrência da não necessidade do uso da técnica de *padding*.

O algoritmo *BQueue* apresenta uma melhora em relação ao algoritmo do *FFBuffer*. Essa melhora se dá principalmente pela previsão que é feita no conteúdo do *buffer*, evitando consultas em variáveis compartilhadas.

4.5 Parâmetro modo de núcleo

O modo de núcleo define de que maneira o programa irá escolher um núcleo virtual para execução. Nos dias atuais a utilização de computação em nuvem se popularizou com grande rapidez [54], neste contexto a maioria dos programas tendem a executar em máquinas com múltiplos processadores o que torna relevante equacionar este fator na análise de desempenho. Os valores escolhidos para este parâmetro se tratam de Same Core (SC), Cross Core (CC) e Cross Socket (CS) que serão fixados pela biblioteca de isolamento de núcleos para se obter observações e resultados mais fiéis.

4.6 Métricas

4.6.1 Tempo de espera

A métrica de tempo de espera tem o objetivo de medir a quantidade de tempo em que o produtor ou o consumidor passou sem realizar uma operação pois foi impedido de prosseguir. Esta métrica tem o objetivo de ajudar a entender se alguma das partes envolvidas se tornam um gargalo para o algoritmo e qual o percentual de tempo em espera relativo ao tempo total de execução para avaliar sua relevância. No caso do produtor este tempo irá medir a soma total das esperas nas situações em que a fila se encontra cheia e portanto uma operação de inserção retornou sem êxito e de forma análoga para o consumidor será a soma total do tempo em que se esteve parado sem poder consumir devido a fila estar vazia.

Aqui é importante notar que ao se falhar em uma operação os algoritmos sempre irão tentar dar a vez para o sistema operacional por meio da sinalização feita pelo comando em Java *Thead.yield()*. Este passo é essencial para que se possa evitar os efeitos de *starvation* nos casos dos testes em *Same Core*, onde alguma das partes em falha pode tomar a CPU de forma gulosa e não permitir a continuidade da outra *thread*. O ganho em espera ocupada sem a utilização de *Thead.yield()* para as execuções *Cross Core* e *Cross Socket* podem ou não ser proporcionais ao tempo de espera observado.

4.6.2 Throughput

Esta métrica visa uma aproximação da quantidade total de itens que podem ser passados pela fila em determinado intervalo de tempo. Apesar de não trazer muita informação sobre como a fila se comporta em execução no sistema é uma das métricas mais relevantes para a utilização dos usuários pois a velocidade de transferência dos itens poderá ser um gargalo para a aplicação dependendo do contexto em que se encontra.

Para os experimentos foi fixado um valor de **500 milhões** de execuções de inserção e de remoções da fila. Um temporizador foi inicializado no início de cada *thread* após a sua execução e encerrado após a finalização do processamento antes mesmo do encerramento da *thread* para evitar contabilizar os tempos de *setup* e *turn-down* das *threads* mesmo que sejam irrelevantes se comparado ao tempo total de execução devido à grande quantidade de itens.

4.6.3 Quantidade de instruções

A métrica de instruções permite observar a quantidade de processamento necessário para se realizar determinado trabalho. Um aspecto importante de se avaliar é a quantidade de instruções executados por ciclo de CPU o que possibilita compreender o quão bem um algoritmo pode paralelizar sua execução. Os valores foram obtidos pelo monitoramento realizado em *Perf* dos eventos de *instructions:ku* para obter o valor total de instruções tanto de *kernel* como de usuário que podem ser especificadas de maneira individual por parâmetros.

Instruções de *kernel* estão fortemente atreladas a chamadas de sistema como as operações de leituras, escritas, e no caso dos algoritmos bloqueantes, arbitragem de *locks*. Já instruções a nível de usuário se referem ao restante das operações executadas pelo programa como *loops*, atribuições, operações lógicas e aritméticas entre outras. Aqui é importante notar que um maior número de emissões de instruções de *kernel* pode estar associado a uma alta contenção devido as arbitragens ou um maior volume de tráfego da cache.

4.6.4 Cache

Para as métricas de cache foram monitorados pelo *Perf* os eventos de cache *L1.dcache.loads*, *L1.dcache.load.misses* que se referem aos eventos de *load* de memória de dados em L1 e *LLC.loads*, *LLC.load.misses* que representam os eventos de *Last Level Cache* ou seja os eventos de último nível da cache que se encontra disponível na CPU. Nas máquinas testadas as LLCs foram identificadas como L3 por meio da verificação das informações de cache disponíveis pelo sistema operacional.

O objetivo da análise dos acessos a L1 é tentar identificar o quão bem um algoritmo utiliza a memória de mais rápido acesso e identificar possíveis casos de *false-sharing*, já a análise de LLC permite entender de onde está se originando os tempos de execução do algoritmo uma vez que o tempo de acesso ao último nível é consideravelmente maior se comparado a L1 e reflete de maneira significativa nos tempos totais de execução.

É importante notar que o *Perf*, assim como as demais ferramentas consideradas, não disponibilizam de maneira simples os eventos de L2 porém os dados desta cache não necessitam ser coletados pois não apresentam informações relevantes uma vez que os comportamentos de *false-sharing* se manifestam majoritariamente em L1 e os gargalos de tempo de execução estão fortemente associados aos acessos de memória principal observados em *misses* de L3.

4.7 Resultados

Esta seção apresenta os resultados obtidos, bem como a análise a partir da comparação dos algoritmos selecionados, contrastando (1) a diferença entre os algoritmos nativos do Java, posteriormente (2) entre as filas nativas do Java e o algoritmo de *Lamport Simples*, seguido pela (3) análise da diferença do algoritmo de *Lamport Simples* e as *Filas otimizadas*, que são as filas de *Lamport Otimizado*, *FFBuffer* e *BQueue*. Será também analisado (4) a diferença entre as próprias filas otimizadas, será verificada (5) a influência do modo execução no algoritmos e por fim as vantagens (6) de algoritmos não bloqueantes em relação a algoritmos bloqueantes. É importante destacar que para evidenciar as diferenças entre as filas, cada gráfico selecionado contém uma das métricas a serem analisadas considerando todos os modos de execução e algoritmos utilizados.

4.7.1 Diferenças entre filas

Filas nativas do Java. Dentre as três estruturas nativas de Java, sendo elas *Array Blocking Queue*, *Linked Blocking Queue* e *Concurrent Blocking Queue*, é possível observar na Figura 4.1 que todas as implementações apresentaram resultados melhores em SC. Um dos motivos para tal resultado é a localidade única dos dados do programa em cache e o menor nível de contenção ao se eliminar o paralelismo entre produtor e consumidor o que consequentemente diminui a quantidade de vezes que o sistema operacional deve arbitrar a posse de um *lock*. Nos casos de CC e CS os algoritmos também tendem a utilizar a memória de forma menos otimizada como observado na Figura 4.4 com quedas significativas de percentual de acerto a L1 ao se comparar com SC e valores bem mais altos de quantidade de acesso total a LLC (Figura 4.9).

O mal uso de memória pode estar ligado a diversos fatores. No caso da *Array Blocking Queue* um dos fatores principais pode ser a dessincronização das caches quando uma das partes obtém o *lock* o que impede a outra *thread* de manter sua cache atualizada enquanto a *thread* em posse do *lock* realiza as modificações, caso ocorra uma alternância frequente da posse deste *lock* uma grande quantidade de *cache misses* irá ocorrer. Já as filas ligadas podem sofrer de forma semelhante do efeito de dessincronização mas de forma mais amenizada uma vez que ambas as *threads* conseguem progredir devido aos *locks* apenas em *head/tail* da *Linked Blocking Queue* e a característica *wait-free* da *Concurrent Linked Queue*, entretanto a grande dispersão dos dados das filas ligadas pode gerar um custo maior de sincronização se comparado aos *arrays* contíguos, e um possível número maior de casos de *false-sharing* uma vez que não há uma distribuição organizada dos dados em fila o que tende a estar refletido no percentual de acerto da L1.

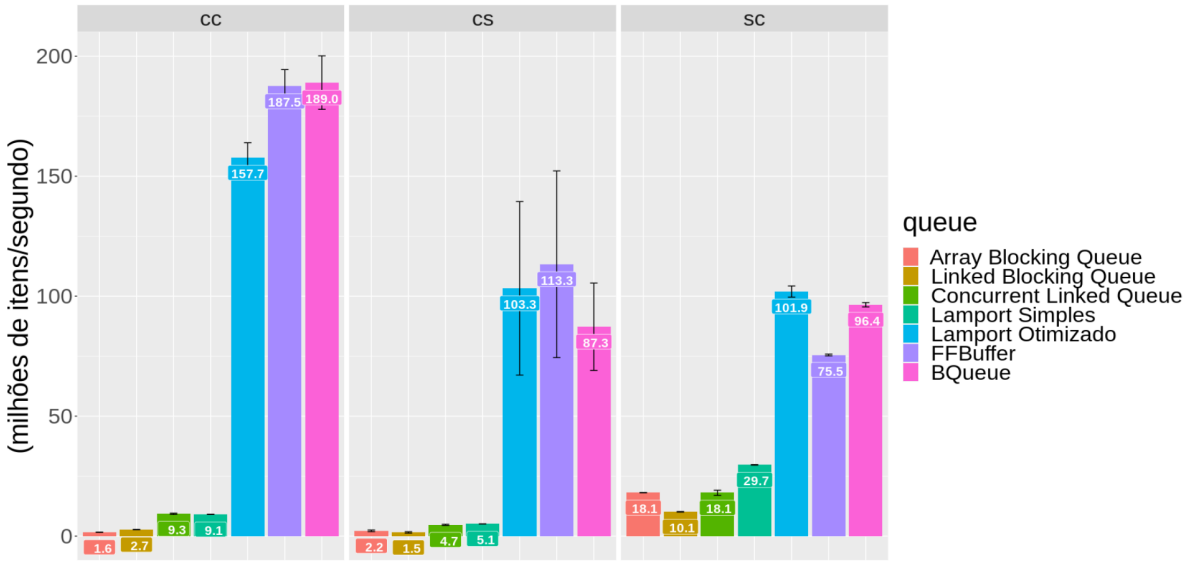


Figura 4.1: Throughput.

É importante notar que apesar das quantidades de instruções totais executadas na Figura 4.2 para *Concurrent Linked Queue* e *Linked Blocking Queue* serem praticamente as mesmas entre os diferentes modos de núcleo, a quantidade de instruções executadas por ciclo na Figura 4.2 é significativamente inferior nos modos CC e CS se comparado ao modo SC o que evidência o tempo ocioso em espera de resultados do elevado número de cache *misses*. Também é importante ressaltar a grande diferença na quantidade de instruções executadas entre os modos de núcleo da fila *Array Blocking Queue* o que pode estar associado a dessincronização de cache anteriormente citada que gera um alto tráfego de cache.

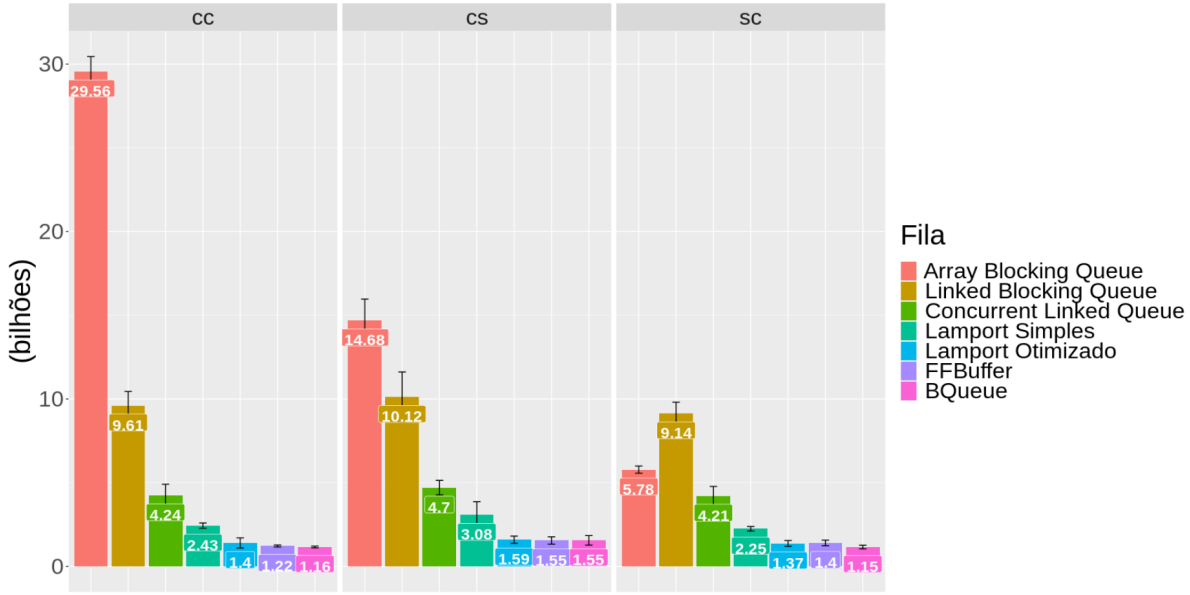


Figura 4.2: Total de instruções executadas.

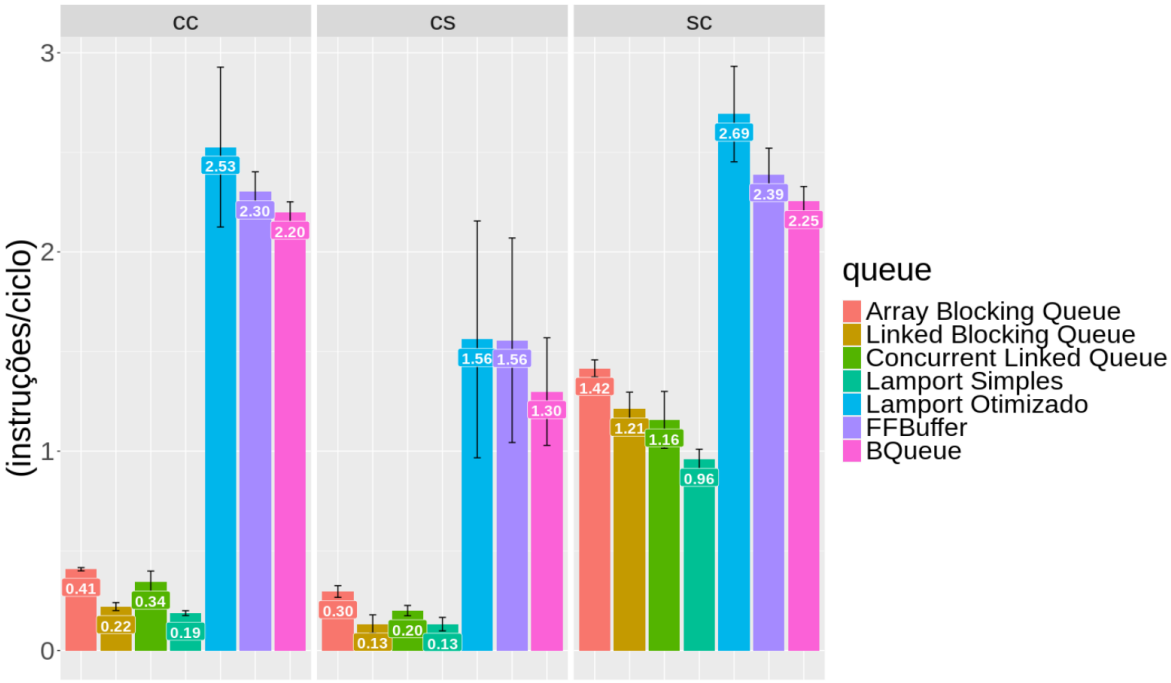


Figura 4.3: Quantidade de instruções por ciclo.

Dentre as três implementações nativas a *Concurrent Linked Queue* obteve um melhor resultado nos casos de CC e CS e um desempenho equiparável a *Array Blocking Queue* em SC. O melhor desempenho nos modos *cross* se deve principalmente a característica *wait-free* da *Concurrent Linked Queue* que resulta em uma quantidade menor de instru-

ções executadas tanto devido ao menor nível de contenção quanto a um uso de memória mais otimizado devido a menor quantidade de acessos tanto em L1 quanto em LLC se comparado as outras filas nativas.

Filas nativas do Java e *Lamport Simples*. De acordo com a Figura 4.1 o algoritmo de *Lamport Simples* possui um desempenho mais próximo da *Concurrent Linked Queue* nos modos CC e CS. Este resultado é esperado por terem ambos uma característica *wait-free*. No modo SC já é possível ver que há uma diferença de *Lamport Simples* em relação as filas nativas. Parte dessa diferença pode ser explicada pela Figura 4.11 onde as estruturas *Array Blocking Queue* e *Linked Blocking Queue* possuem uma quantidade maior de espera total em relação ao algoritmo de *Lamport Simples*, devido ao fato de ambas serem estruturas bloqueantes. Outra justificativa para os resultados inferiores das estruturas bloqueantes seria a quantidade de *cache misses* na Figura 4.4, decorrente da sincronização excessiva com a cache, o que faz com que estes processos constantemente vejam valores desatualizados na cache.

Quanto a *Concurrent Linked Queue*, o tempo de espera é inferior quando somado o tempo total de espera do produtor e consumidor em relação ao *Lamport Simples*, o que não justifica ainda a diferença de desempenho na Figura 4.1 no modo SC. Já o valor de *cache miss* de *LLC* na Figura 4.9, destaca uma diferença de 200 mil acessos a mais a memória principal pela *Concurrent Linked Queue*, ou seja, seu desempenho acaba sendo prejudicado por ter que acessar um nível de memória mais lento, enquanto que *Lamport Simples* (Figura 4.9) possui uma quantidade bem inferior de *cache misses*.

Dentre as opções das filas nativas de Java, o algoritmo de *Lamport Simples* possui o melhor desempenho quando considerado a média de desempenho em todos os modos na Figura 4.1. Embora a *Concurrent Linked Queue* possua resultados equiparáveis a *Lamport Simples* nos modos CC e CS, o modo SC destaca a maior diferença de tráfego de cache realizada por ambas estruturas (Figura 4.6), o que favorece o algoritmo de *Lamport Simples*. A vantagem da estrutura Java, é que ela não restringe o uso a um único processo de consumidor e produtor como é feito no algoritmo de *Lamport Simples*

***Lamport Simples* e as filas otimizadas.** Botando o foco apenas nas soluções orientadas a SPSC também é possível notar uma grande diferença de ganho entre *Lamport Simples* e as demais versões otimizadas (*BQueue*, *FFBuffer* e *Lamport Otimizado*) ao se contrapor *throughput*. O ganho das versões otimizadas utilizando a média de seus desempenhos com relação a *Lamport Simples* fica em 3.07x (SC), 19.54x (CC) e 19.86x (CS). Ao se observar a quantidade de instruções totais executadas pode-se notar que os valores das filas otimizadas chegam a ser quase a metade do total executado por *Lamport Simples*.

Um dos fatores que mais afetam neste aspecto é o melhor uso de operações para obtenção de módulo que permite diminuir consideravelmente o número de instruções necessárias para o cálculo da posição de acesso no *buffer*.

As barreiras com *lazy set* também permitem uma maior vazão de processamento devido a menor necessidade de se esperar pela limpeza dos *buffers* de memória o que se reflete diretamente na quantidade total de instruções executadas por ciclo. Ao se observar a Figura 4.3, há um ganho das filas otimizadas com relação a *Lamport Simples* de aproximadamente 2.54x (SC), 12.33x (CC) e 11.33x (CS). Aqui é importante observar que os valores não batem diretamente com os ganhos em *throughput* devido a quantidade total distinta de instruções necessárias para cada algoritmo porém a quantidade de instruções por ciclo permite uma melhor avaliação da eficiência no uso de CPU, o que torna evidente o maior paralelismo das filas otimizadas.

Outra vantagem importante das filas otimizadas é o melhor uso de memória por meio das técnicas de *padding* para se evitar os casos de *false-sharing*. Em termos de acessos a L1 na Figura 4.4, onde mais se concentram estes casos, é visível a diferença nos percentuais de *hit* para *Lamport Simples* entre os modos de núcleo descendo 8.61 pontos percentuais entre SC e CC o que já não é observado na fila de *Lamport Otimizado* tanto pelos caches de *tail/head* quanto pelo *padding* do *buffer* utilizados como otimizações. É importante também notar que, em termos de valores absolutos na Figura 4.6, a versão de *Lamport Simples* possui uma quantidade de acessos maior do que todas as filas otimizadas para todos os modos de núcleo o que também é reflexo da não utilização de *lazy set* o que gera *flushes* dos *buffers* de escrita mais frequentes e consequentemente uma maior quantidade de operações de leitura para sincronização da cache.

A nível de LLC é possível observar que o melhor uso de L1 também possui impacto em seus valores. Um cache miss em L1 pode ser servido por outras caches L1, por um acesso em L2 ou até mesmo ir ao último nível de cache caso necessário. Para *Lamport Simples* os piores valores de acerto de L1 entre os modos de núcleo na Figura 4.4 refletem também em maiores acessos a LLC na Figura 4.9 o que já não acontece para as versões otimizadas que ao utilizarem de maneira eficiente a L1 também apresentam valores mais controlados de acesso a LLC na Figura 4.8.

Filas otimizadas. Entre as filas otimizadas é possível observar comportamentos muito próximos entre todas as distintas métricas coletadas. Para *throughput* as filas se alternam entre o maior valor para cada modo de núcleo, sendo para CC a melhor média com a *BQueue*, em CS *FFBuffer* e em SC *Lamport Otimizado*. Vale ressaltar que no modo *Cross Socket* há uma grande variação entre as execuções dos testes como mostrado na Tabela 4.1 o que está fortemente associado a grande variação do total de acessos a LLC

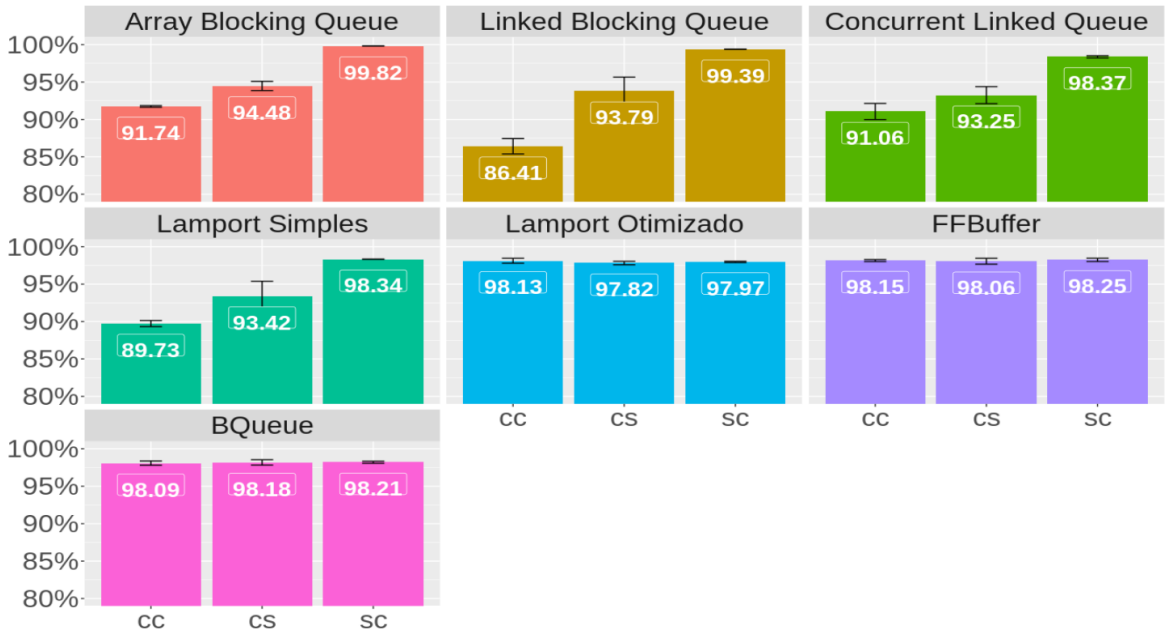


Figura 4.4: Percentual de hit em L1.

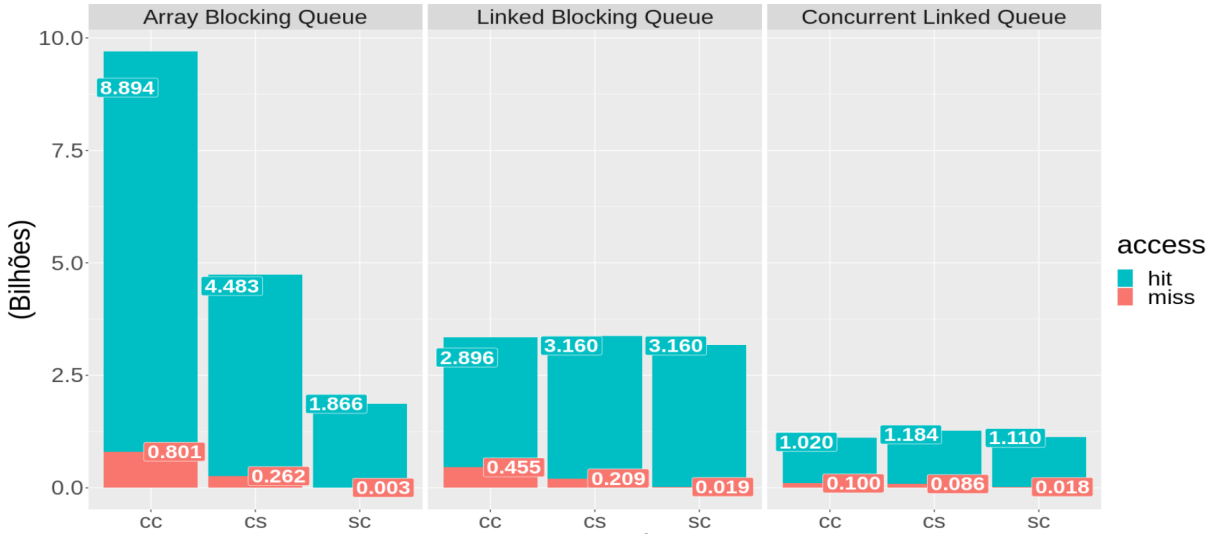


Figura 4.5: Total de hit/miss em L1.

como também observado Figura 4.7, portanto não se pode determinar um claro vencedor para as execuções neste modo de núcleo mesmo que *FFBuffer* tenha a melhor média.

A grande variação de LLC para *Cross Socket* pode ser novamente consequência de casos de *false-sharing* que não necessariamente são 100% eliminados com as técnicas apresentadas. Entre execuções de um mesmo programa também é normal que os dados sejam dispostos em memória de maneiras distintas o que contribui para um número variável de percentual de *hit/miss* em cache. Para o modo *Cross Socket* das filas otimizadas é

Figura 4.6: Total de hit/miss em L1.

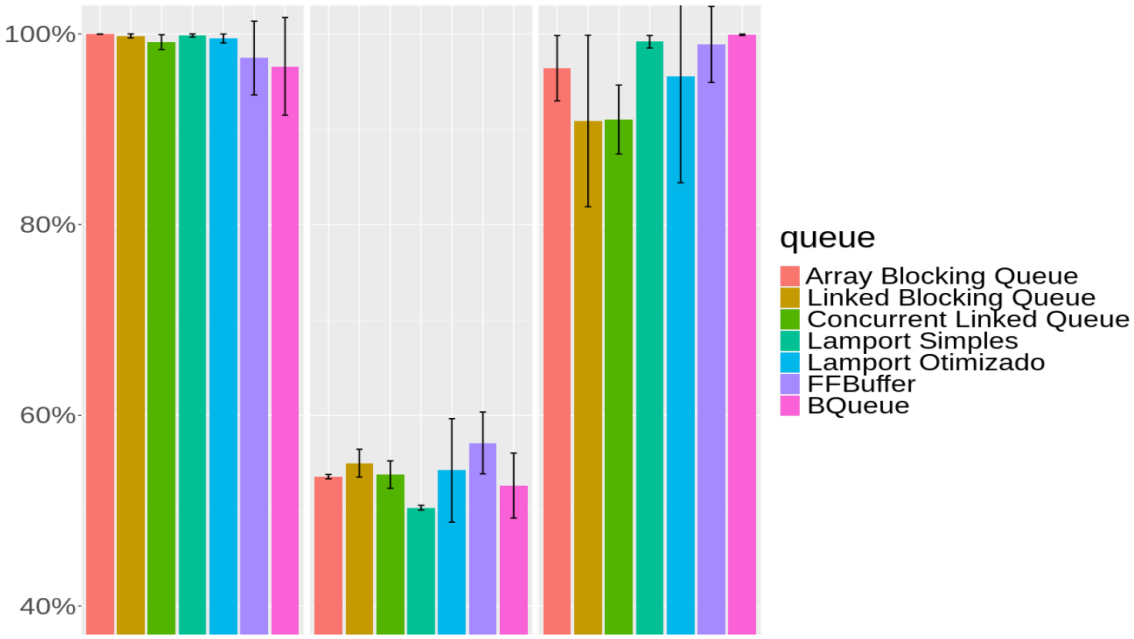


Figura 4.7: Percentual de hit em LLC.

Fila	Throughput (Milhões)	LLC (Milhões)
Lamport Otimizado	103,3 ± 36,159	0,9555687 ± 3,147
FFBuffer	113,3 ± 38,903	0,5709415 ± 2,712
BQueue	87 ± 18,201	0,5262188 ± 2,686

Tabela 4.1: Valores e desvio padrão no modo *cross-socket* para filas otimizadas em Throughput e LLC.

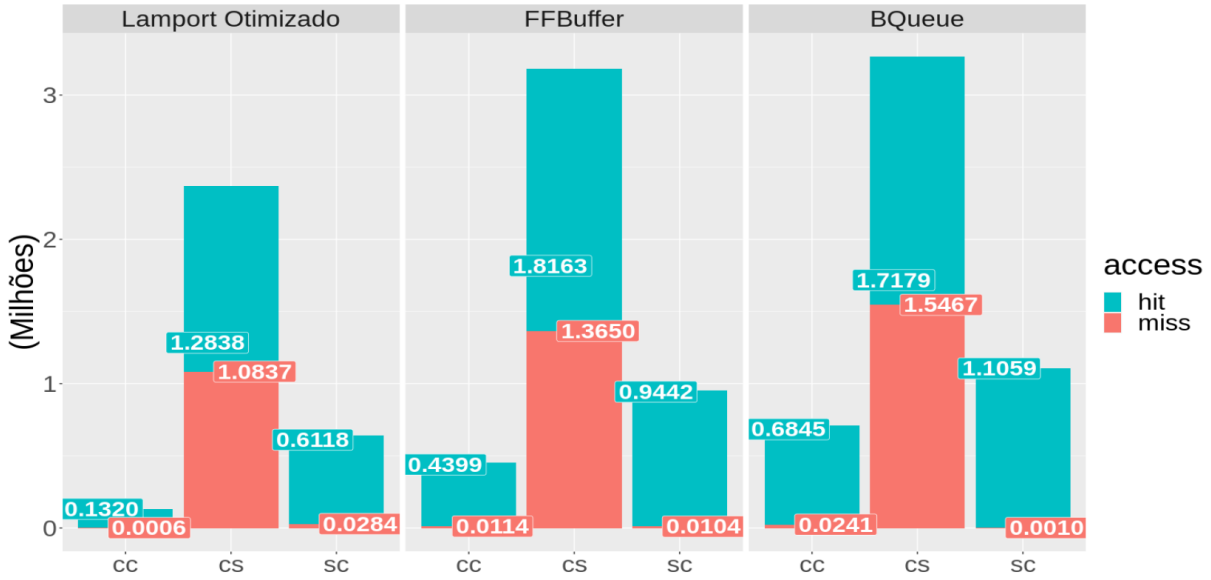


Figura 4.8: Total de hit/miss em LLC.

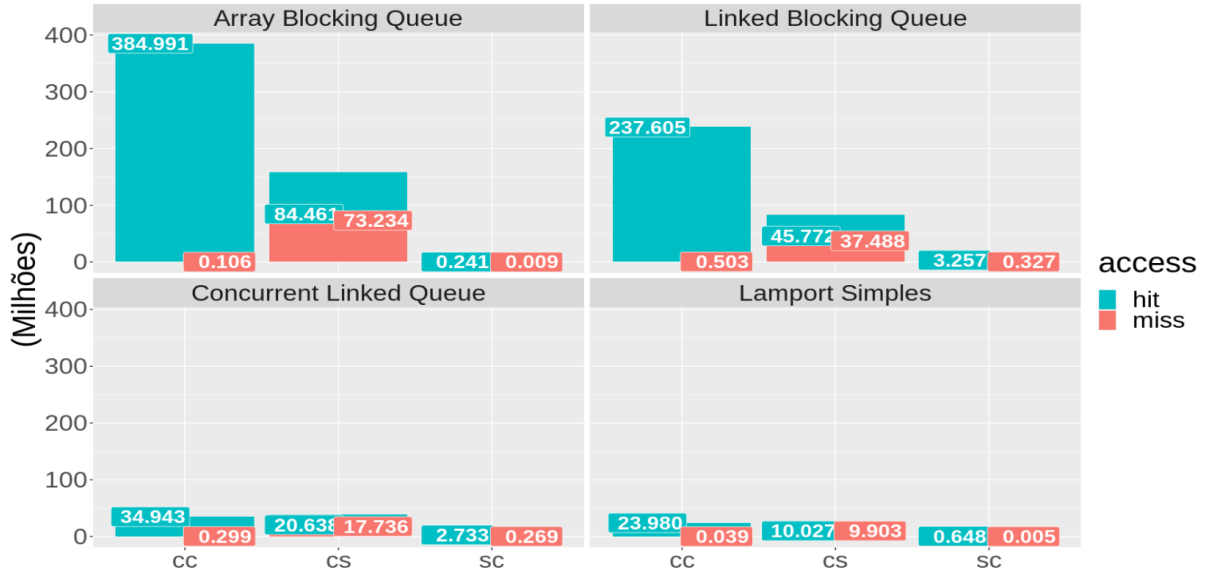


Figura 4.9: Total de hit/miss em LLC.

possível observar que os valores médios de *miss* em L1 na Figura 4.4 são por volta de 1 milhão de acessos maior do que os demais modos de núcleo. Apesar da pouca diferença, cada *miss* em L1 que possa vir a gerar um novo *miss* em LLC sofre a penalidade de uma comunicação via QPI que tende a ser 40 vezes mais lenta do que um acesso em L1 como mostrado na Figura 2.1. É importante notar que quase a totalidade dos *misses* de LLC aqui observados são de fato consultas ao outro *socket* via QPI uma vez que os modos CC e SC possuem quase 100% dos acessos a LLC servidos sem *misses* o que é consequência

dos dados do programa couberem quase que integralmente em cache.

Para a métrica de instruções todas as filas otimizadas apresentam um valor entre 100 a 200 milhões de instruções a mais executadas no modo CS o que pode estar associado a memória distribuída entre os núcleos que irá necessitar de mais passos para a sincronização. Já para os tempos de espera os algoritmos *FFBuffer* e *Lamport Otimizado* apresentam um maior percentual de tempo de espera do consumidor no modo Cross Socket o que já não ocorre com a *BQueue* que não apresenta tempo relevante de espera neste modo de núcleo como mostrado na Figura 4.10. Após algumas análises não foi possível determinar com exatidão o porque desta discrepância.

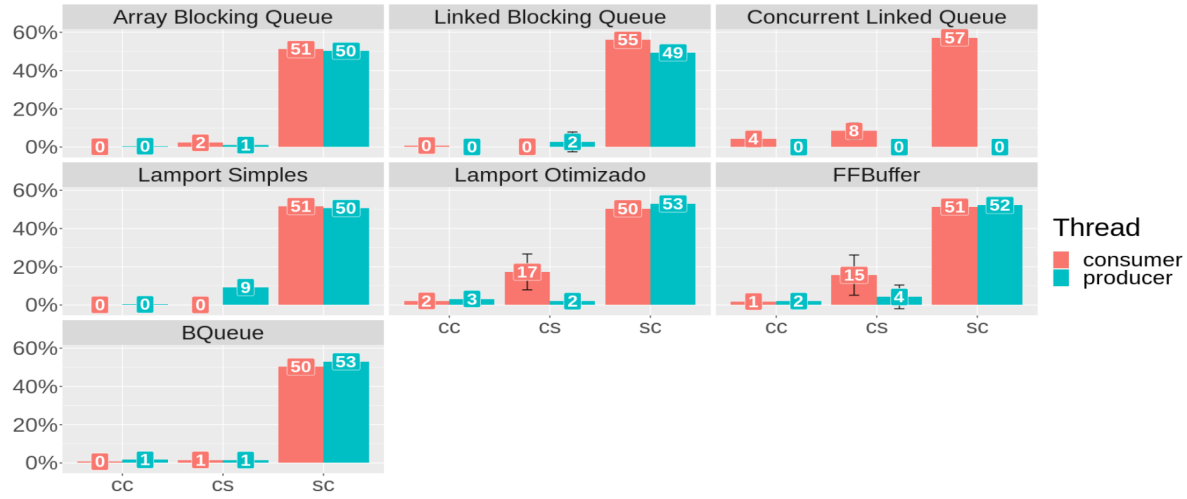


Figura 4.10: Porcentagem de tempo de espera durante execução.

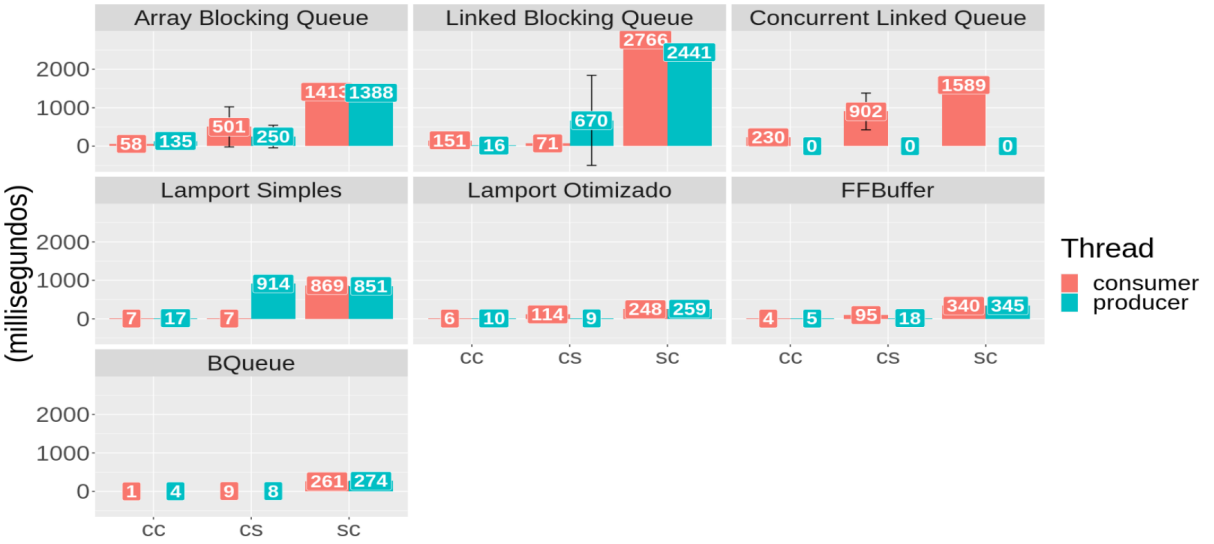


Figura 4.11: Tempo de espera por produtor/consumidor.

4.7.2 Diferença entre os modos de execução

Dependendo do modo de núcleo escolhido os resultados de *throughput* se diferem para todos os algoritmos estudados, principalmente nas filas otimizadas. A média de todas as filas otimizadas juntas para cada modo ficou em 101.3 para CS, 91.2 para SC e 177.6 para CC o que mostra uma clara vantagem ao se executar os algoritmos otimizados em modo CC. O melhor desempenho em CC é explicado pelo melhor uso de memória tanto L1 quanto LLC o que gera uma maior quantidade de instruções por ciclo como apresentado na Figura 4.3. Ao se comparar CS com SC não foi observada uma diferença significativa uma vez que uma alta variação dos resultados foi observada no modo CS, além disso o modo CS obteve os piores resultados de testes individuais para todas as filas a menos da *Array Blocking Queue*

Para LLC é possível observar que todas as filas obtiveram um percentual próximo de 50% de *hit* no modo CS (Figura 4.7), ou seja, todas as modificações efetuadas em um processador deverão eventualmente ser passadas para o outro, como as rotinas do produtor e do consumidor são similares estas modificações acabam resultando em uma quantidade de modificações semelhantes para cada *socket*. Também pode-se visualizar uma maior quantidade de requisições feitas para cache L1 no modo CS na Figura 4.6, que também é maior do que os demais modos de núcleo.

Uma característica interessante no modo SC que pode ser observado na Figura 4.10, é o fato do produtor e consumidor em quase todas filas com exceção da fila *Concurrent Linked Queue*, possuírem percentuais de tempo em espera similares. Como há apenas um núcleo nesse modo, ou o produtor está em execução ou o consumidor está em execução, o que gera um tempo de espera parecido para ambos processos assim que a fila esvazia ou fica cheia.

Para todos os modos também houve pouca diferença na quantidade de instruções executadas apenas com exceção da fila *Array Blocking Queue* na Figura 4.2 que obteve valores significativamente maiores em CC e SC o que evidencia um possível problema de escalabilidade para esta fila ao se aumentar o nível de contenção com mais *threads* competindo.

4.7.3 Vantagens de algoritmos não bloqueantes em relação a algoritmos bloqueantes

Estruturas bloqueantes tendem a ser mais lentas em relação as versões não bloqueantes em situações em que há um alto nível de contenção. Uma das formas mais comuns aplicadas para a obtenção de *locks* é a técnica de *spin-lock* usualmente utilizada em sistemas Linux. O fato de possuírem a característica de espera ocupada faz com que a todo momento o

algoritmo esteja executando alguma operação para verificar se a área de exclusão mútua, que é o *buffer*, está liberada ou não nos casos em que o produtor ou consumidor quer acessar o *buffer*. A verificação constante da disponibilidade da área de exclusão mútua, incorre com que o algoritmo execute mais instruções. Podemos observar esse resultado na Figura 4.3, onda a quantidade de instruções dos algoritmos *Array Bloquing Queue* e *Linked Bloquing Queue* é superior do que todos os outros algoritmos.

O problema de um algoritmo possuir uma quantidade elevada de instruções é também ser acompanhado de uma baixa paralelização das mesmas. A Figura 4.3 demonstra que os algoritmos bloqueantes são os que possuem menor quantidade de instrução por ciclo, principalmente nos modos CC e CS, ou seja, na medida que é feito um bloqueio em um recurso, o processo permanece sem fazer progresso nenhum, até que o bloqueio seja removido. É interessante notar que mesmo as filas não bloqueantes *Concurrent Linked List* e *Lamport Simples*, possuem baixa paralelização, entretanto sua quantidade total de instruções é menor em relação as bloqueantes, o que proporcionalmente impacta no resultado de *throughput*.

Outro impacto nas filas bloqueantes é a quantidade de acessos a memória. Analisando as Figura 4.5 e Figura 4.6, pode ser visto que a quantidade de acessos a memória das filas bloqueantes é superior ao das filas não bloqueantes. Uma das possibilidades que colaboram para este resultado, é o fato de que no momento em que o processo consumidor ou produtor está bloqueado, sua cache tenderá a ficar fora de sincronismo com a cache da *thread* que continua a realizar modificações, uma vez que não irá efetuar leituras neste período de bloqueio. Ao se obter novamente posse do *lock* irá incorrer cache *misses* para que a sincronização seja efetuada causando os menores percentuais de *hit* observados em CC e CS. Além disso, o próprio acesso aos *locks* pode gerar cache *misses*.

4.8 Conclusões do capítulo

Ao procurar responder as questões propostas, a primeira observação mais evidente nos resultados foram os valores inferiores de *throughput* para os algoritmos bloqueantes. Expandindo a análise para todas as filas nativas de java, observa-se um melhor desempenho alcançado pela *Concurrent Linked Queue* principalmente por ser uma estrutura não bloqueante. Em comparação a fila *Lamport Simples* foi possível observar melhores resultados para a solução de Lamport em todos os modos de núcleo com maior notoriedade para SC, porém os melhores resultados gerais são referentes aos das filas otimizadas, decorrente das técnicas apresentadas.

É importante que seja levado em consideração o modo de núcleo na escolha da utilização de algum destes algoritmos, pois as filas otimizadas obtiveram os melhores resultados

em *CC* porém também apresentaram uma alta variação das métricas no modo *CS*, com destaque para uma menor variação oferecida pela *BQueue*. Outro fator decisivo está na complexidade em se trabalhar com algoritmos *wait-free*, como as filas otimizadas, demandando mais esforço na compreensão de seu comportamento, que pode ser imprevisível e de difícil entendimento.

Capítulo 5

Conclusão

5.1 Visão geral do trabalho

Neste trabalho, foi possível desenvolver uma análise comparativa entre diferentes algoritmos já existentes para o problema do produtor e consumidor únicos, investigando como as diferentes características dos mesmos podem influenciar em seus desempenhos. Nesse contexto, um conjunto de técnicas de otimização a nível de *hardware* e *software* foi avaliado através da experimentação realizada a partir de três cenários de execução: Cross Core (CC), Cross Socket (CS) e Same Core (SC).

Foi possível observar que os algoritmos otimizados selecionados apresentam desempenho similar, possuem os melhores resultados no modo de execução CC e tendem a um comportamento parecido no uso mais otimizado da memória, diferentemente do que acontece nos algoritmos mais básicos. Desta forma um uso otimizado da memória é um requisito crítico para algoritmos que buscam alto desempenho em ambientes onde há uma alta comunicação entre os processos.

5.2 Revisão dos objetivos

Todos os objetivos propostos foram alcançados. Desta forma, esse trabalho tem como principais contribuições:

- **Compreender os componentes básicos de *hardware/software* por trás do funcionamento de estruturas de dados não bloqueantes**. Este objetivo foi alcançado através de uma revisão bibliográfica que reuniu artigos, documentações, fóruns e vídeos que permitiram identificar estruturas como *buffer* de escrita e filas de invalidação além do funcionamento de mecanismos como as barreiras de memória.

- **Identificar as principais soluções e otimizações para o problema do produtor único e consumidor único**. Foram identificadas as três principais implementações em vetores utilizadas para a resolução SPSC assim como as otimizações mais recentes.
- **Execução de experimentos em ambiente de teste com máquinas reais**. A partir da plataforma Emulab foi criado um ambiente onde é possível especificar características do hardware do experimento, possibilitando a escolha por máquinas físicas.
- **Analisar a diferença de desempenho entre algoritmos bloqueantes e não bloqueantes**. Foi observado na prática a grande diferença que é relatada em certos artigos de trabalhos relacionados principalmente ao se comparar com as versões otimizadas para SPSC.
- **Analisar a diferença de desempenho entre as filas identificadas como o estado da arte**. No geral, não foi possível identificar diferenças no contexto proposto que fossem significativas o suficiente. No entanto, as várias técnicas de otimização aumentam substancialmente o desempenho destes algoritmos.
- **Analisar como o contexto de execução (por exemplo, ambos processos sendo executados no mesmo núcleo ou em núcleos diferentes) pode influenciar no desempenho dos algoritmos.** Foi observado que executar algoritmos não bloqueantes em modo Cross Core entrega um melhor desempenho em *throughput* se comparado aos modos Cross Socket e Same Core, o que já não é observado nos algoritmos bloqueantes. Também é importante ressaltar uma grande variação de resultados em modo Cross Socket principalmente nas versões das filas otimizadas.

5.3 Perspectivas futuras

A principal sugestão de trabalhos futuros é a avaliação destas soluções quando empregadas em uma aplicação real. Como mencionado em [15], certas filas podem ter resultados diferentes que não refletem o desempenho dos testes básicos executados apenas com as filas. Os testes realizados também foram feitos apenas com *payloads* indiretos passando apenas referências para as filas, porém como citado em [16] podem existir grandes diferenças entre *payloads* indiretos e diretos devido ao uso da cache.

Também pode ser importante investigar as diferenças entre implementações com distintas linguagens de programação que não foram cogitadas aqui devido ao esforço adicional de implementação. Testes em mais de uma arquitetura de CPU podem também apresentar resultados diferentes devido aos modelos de memória e otimizações de *hardware*

específicas, porém não foram realizados aqui devido a indisponibilidade de máquinas com as arquiteturas desejadas.

Referências

[1] Lamport, Leslie: *Concurrent reading and writing*. Commun. ACM, 20(11):806–811, novembro 1977, ISSN 0001-0782. https://doi.org/10.1145/359863.359878. vii, 2, 21, 25, 26, 28

[2] Thompson, Martin: *Cpu cache flushing fallacy*. https://mechanical-sympathy.blogspot.com/2013/02/cpu-cache-flushing-fallacy.html, acesso em 2021-10-17. x, 7, 8, 13

[3] Thompson, Martin: *Lock-free algorithms for ultimate performance*. https://www.infoq.com/presentations/Lock-Free-Algorithms/, acesso em 2021-10-17. x, 8

[4] Mckenney, Paul: *Memory barriers: a hardware view for software hackers*. agosto 2010. x, 8, 9, 10, 11, 12, 13, 16

[5] Xu, Jeffle: *Memory model and synchronization primitive - part 1: Memory barrier*. https://www.alibabacloud.com/blog/memory-model-and-synchronization-primitive---part-1-memory-barrier_597460, acesso em 2021-10-17. x, 17

[6] Wakart, Nitsan: *Notes on false sharing*. http://psy-lob-saw.blogspot.com/2014/06/notes-on-false-sharing.html, acesso em 2021-10-17. x, 33, 36

[7] Lea, Doug: *The jsr-133 cookbook for compiler writers*. http://gee.cs.oswego.edu/dl/jmm/cookbook.html, acesso em 2021-10-17. xi, 14, 15, 16, 19, 30

[8] Thompson, Martin: *Single writer principle*, Jan 1970. https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.html. xi, 20, 21

[9] Intel: *Intel optimization reference*. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html, acesso em 2021-10-17. xi, 32

[10] Nitsan: *Blog code snippets*. https://github.com/nitsanw/examples/blob/master/src/java/uk/co/real_logic/queues/P1C1QueueOriginal1.java, acesso em 2021-03-22. xii, 27

[11] Giacomoni, John, Tipp Moseley e Manish Vachharajani: *Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue*. Em *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, página 43–52, New York, NY, USA, 2008. Association for Computing Machinery, ISBN 9781595937957. https://doi.org/10.1145/1345206.1345215. xii, 2, 21, 24, 37

[12] Wang, Junchang, Kai Zhang, Xinan Tang e Bei Hua: *B-queue: Efficient and practical queuing for fast core-to-core communication*. International Journal of Parallel Programming, 41(1):137–159, 2013. xii, 2, 21, 24, 25, 38, 39

[13] Wikipedia: *Producer–consumer problem — Wikipedia, the free encyclopedia*. http://en.wikipedia.org/w/index.php?title=Producer%E2%80%93consumer%20problem&oldid=1023067597, 2021. [Online; accessed 17-October-2021]. 1, 19, 21

[14] Wikipedia: *Lock (computer science) — Wikipedia, the free encyclopedia*. http://en.wikipedia.org/w/index.php?title=Lock%20(computer%20science)&oldid=1039414803, 2021. [Online; accessed 17-October-2021]. 1, 20

[15] Aldinucci, Marco, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin e Massimo Torquati: *An efficient unbounded lock-free queue for multi-core systems*. Volume 7484, páginas 662–673, agosto 2012, ISBN 978-3-642-32819-0. 1, 25, 65

[16] Maffione, Vincenzo, Giuseppe Lettieri e Luigi Rizzo: *Cache-aware design of general-purpose single-producer-single-consumer queues: Cache-aware single-producer-single-consumer queues*. Software: Practice and Experience, 49, dezembro 2018. 1, 2, 25, 28, 29, 32, 34, 65

[17] Krimgen, Michael: *Introduction to lock-free data structures with java examples*, Oct 2020. https://www.baeldung.com/lock-free-programming, acesso em 2021-10-17. 1, 21

[18] Torquati, Massimo: *Single-Producer/Single-Consumer Queues on Shared Cache Multi-Core Systems*. arXiv e-prints, página arXiv:1012.1824, dezembro 2010. 2, 25, 33, 34

[19] Lê, N. M., A. Guatto, A. Cohen e A. Pop: *Correct and efficient bounded fifo queues*. Em *2013 25th International Symposium on Computer Architecture and High Performance Computing*, páginas 144–151, 2013. 2, 26

[20] Bessani, Alysson, João Sousa e Eduardo EP Alchieri: *State machine replication for the masses with bft-smart*. Em *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, páginas 355–362. IEEE, 2014. 2

[21] Alchieri, Eduardo, Fernando Dotti, Odorico M Mendizabal e Fernando Pedone: *Reconfiguring parallel state machine replication*. Em *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, páginas 104–113. IEEE, 2017. 2

[22] Alchieri, Eduardo, Fernando Dotti e Fernando Pedone: *Early scheduling in parallel state machine replication*. Em *Proceedings of the ACM Symposium on Cloud Computing*, páginas 82–94, 2018. 2

[23] Burgos, Aldênio, Eduardo Alchieri, Fernando Dotti e Fernando Pedone: *Replicação máquina de estados paralelas com escalonamento híbrido.* Em *Anais do XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, páginas 476–489. SBC, 2021. 2

[24] Lönnberg, Jan e Anders Berglund: *Students' understandings of concurrent programming.* novembro 2007. 4

[25] Intel: *O que é hyper-threading?* https://www.intel.com.br/content/www/br/pt/gaming/resources/hyper-threading.html. 4

[26] Sundell, Håkan: *Efficient and practical non-blocking data structures.* Doktorsavhandlingar vid Chalmers Tekniska Hogskola, janeiro 2004. 5

[27] Krajewski, Marek: *Hands-On High Performance Programming with Qt 5: Build cross-platform applications using concurrency, parallel programming, and memory management.* Packt Publishing, Birmingham, England, 2019. 5, 6

[28] Wikipedia: *Dynamic random-access memory — Wikipedia, the free encyclopedia.* http://en.wikipedia.org/w/index.php?title=Dynamic%20random-access%20memory&oldid=1047788031, 2021. [Online; accessed 17-October-2021]. 6

[29] Wikipedia: *Static random-access memory — Wikipedia, the free encyclopedia.* http://en.wikipedia.org/w/index.php?title=Static%20random-access%20memory&oldid=1044876519, 2021. [Online; accessed 17-October-2021]. 6

[30] *Sram vs dram.* https://www.diffen.com/difference/Dynamic_random-access_memory_vs_Static_random-access_memory, acesso em 2021-10-17. 6

[31] Pannain, Ricardo: *Arquitetura de computadores.* https://www.ic.unicamp.br/~ducatte/mc542/Arquitetura/arq_hp7.pdf, acesso em 2021-10-17. 6

[32] Stewart, Lawrence: *What is the difference between l1 and l2 cache.* https://www.quora.com/What-is-the-difference-between-L1-and-L2-cache, acesso em 2021-10-17. 7

[33] Gupta, Aarti: *Under the hood definitions: Memory model and memory barriers*, May 2018. https://medium.com/software-under-the-hood/under-the-hood-definitions-memory-model-f41cd3b47757, acesso em 2021-10-17. 12

[34] Blinn, Bruce: *Memory barriers*, 2009. http://bruceblinn.com/linuxinfo/MemoryBarriers.html, acesso em 2021-10-17. 13, 17

[35] Georgopoulos, Georgios: *Memory consistency models of modern cpus.* Embedded Systems Seminar. https://es.cs.uni-kl.de/publications/datarsg/Geor16.pdf, acesso em 2021-10-17. 14

[36] Preshing, Jeff: *Memory barriers are like source control operations.* https://preshing.com/20120710/memory-barriers-are-like-source-control-operations/, acesso em 2021-10-17. 15

[37] Grimm, Rainner: *Acquire-release fences*. https://www.modernescpp.com/index.php/acquire-release-fences, acesso em 2021-10-17. 18

[38] *Atomic operations library*. https://en.cppreference.com/w/cpp/atomic/atomic, acesso em 2021-10-17. 18

[39] *Pep 583 – a concurrency memory model for python*. https://www.python.org/dev/peps/pep-0583/, acesso em 2021-10-17. 18

[40] Wikipedia: *Synchronization (computer science) — Wikipedia, the free encyclopedia*. http://en.wikipedia.org/w/index.php?title=Synchronization%20(computer%20science)&oldid=1036853128, 2021. [Online; accessed 17-October-2021]. 20

[41] Vyukov, Dmitry: *Producer-consumer queues*. http://www.1024cores.net/home/lock-free-algorithms/queues, acesso em 2021-10-17. 22

[42] Milewski, Bartosz: *Who ordered sequential consistency*. https://bartoszmilewski.com/2008/11/11/who-ordered-sequential-consistency/, acesso em 2021-10-17. 28

[43] Higham, L. e J. Kawash: *Critical sections and producer/consumer queues in weak memory systems*. Em *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*, páginas 56–63, 1997. 28

[44] Wakart, Nitsan: *Spsc revisited part iii - fastflow + sparse data*. http://psy-lob-saw.blogspot.com/2013/10/spsc-revisited-part-iii-fastflow-sparse.html, acesso em 2021-10-17. 30

[45] Wakart, Nitsan: *Atomic*.lazyset is a performance win for single writers*, Dec 2012. http://psy-lob-saw.blogspot.com/2012/12/atomiclazyset-is-performance-win-for.html, acesso em 2021-10-17. 30, 31

[46] Wakart, Nitsan: *What do atomic*::lazyset/atomic*fieldupdater::lazyset/unsafe::putordered* actually mean?* http://psy-lob-saw.blogspot.com/2016/12/what-is-lazyset-putordered.html, acesso em 2021-10-17. 31

[47] Wakart, Nitsan: *The mythical modulo mask*. http://psy-lob-saw.blogspot.com/2014/11/the-mythical-modulo-mask.html, acesso em 2021-10-17. 31

[48] Thompson, Martin: *False sharing*, Jan 1970. https://mechanical-sympathy.blogspot.com/2011/07/false-sharing.html. 34

[49] *Detailed explanation of memory layout in java object – java virtual machine*. https://www.fatalerrors.org/a/detailed-explanation-of-memory-layout-in-java-object-java-virtual-machine.html, acesso em 2021-10-18. 34

[50] Wakart, Nitsan: *Know thy java object memory layout*. http://psy-lob-saw.blogspot.com/2013/05/know-thy-java-object-memory-layout.html, acesso em 2021-10-17. 35, 36

[51] Wakart, Nitsan: *Spsc iv - a look at bqueue*. http://psy-lob-saw.blogspot.com/2013/11/spsc-iv-look-at-bqueue.html, acesso em 2021-10-17. 40

[52] *Java thread affinity*. https://github.com/OpenHFT/Java-Thread-Affinity, acesso em 2021-10-18. 44

[53] Michael, Maged M. e Michael L. Scott: *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*. Em *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, página 267–275, New York, NY, USA, 1996. Association for Computing Machinery, ISBN 0897918002. https://doi.org/10.1145/248052.248106. 48

[54] Arnautov, Sergei, Pascal Felber, Christof Fetzer e Bohdan Trach: *Ffq: A fast single-producer/multiple-consumer concurrent fifo queue*. 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), páginas 907–916, 2017. 50