Universidade Federal de Pernambuco – UFPE

Centro de Informática

Graduação em Ciência da Computação

Uma Análise do Custo Computacional de Estimadores do DFSA

Arthur Barros Lapprand

Recife

2017

Arthur Barros Lapprand

Uma Análise do Custo Computacional de Estimadores do DFSA

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Departamento de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Universidade Federal de Pernambuco – UFPE

Centro de Informática

Graduação em Ciência da Computação

Orientador: Prof. Dr. Paulo André da S. Gonçalves

Recife

2017

A todos que pesquisam sobre sistemas RFID pelo mundo.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Benise e Gérard, por me proverem condições de executar este trabalho. Ambos trabalharam duro para que eu sempre tivesse saúde física e mental. Nunca deixaram de me apoiar em momentos de dificuldade e formaram a pessoa que sou. Por isso, sou imensamente grato.

Agradeço também a meu orientador, o professor Paulo Gonçalves, por ter me guiado durante minha trajetória acadêmica. Seus norteamentos nunca faltaram em contribuir com meu crescimento, tomando como base a ética e o profissionalismo.

Por fim, agradeço a todos que de alguma forma contribuiram com o desenvolvimento deste projeto. Em especial a meus amigos, que nos momentos de tensão me ajudaram a descontrair.

RESUMO

Na Internet das Coisas (do inglês, Internet of Things – IoT), sistemas de Identificação por Radiofrequência (do inglês, Radio-Frequency IDentification – RFID) são soluções populares para identificar objetos automaticamente. Nestes sistemas, dados são obtidos e armazenados em dispositivos normalmente conhecidos como etiquetas RFID. Os dados são passados por sinais de rádio cuja comunicação é ditada por protocolos de acesso ao meio (do inglês, Medium Access Control – MAC). A norma em questão é nomeada Dynamic-Framed Slotted ALOHA (DFSA). Como o funcionamento destes sistemas baseia-se em dispositivos eletrônicos, uma preocupação relevante é o custo computacional. Este, além de ser impactante no gasto energético, é fator limitante em vários dispositivos RFID por possuírem poder de processamento limitado. Este trabalho apresenta a implementação de um programa cujo papel é simular um leitor de etiquetas RFID. São implementados alguns algoritmos conhecidos como estimadores, os quais são parte do DFSA. Uma vez processadas as simulações, são feitas comparações visando o tempo estimado gasto na identificação de etiquetas para cada estimador. Por fim, este documento avalia como o custo computacional de referência dos algoritmos conhecidos de estimadores se correlaciona com o tempo médio total de processamento no processo de identificação das etiquetas no simulador. Os resultados mostram que os valores utilizados servem de forma satisfatória para estimativas de custo computacional.

Palavras-chave: Internet das Coisas. Sistemas RFID. DFSA. Custo Computacional.

ABSTRACT

In the Internet of Things, Radio-Frequency Identification (RFID) systems are a promising technology for automatic object identification. In these systems, data is acquired from and saved in devices usually known as RFID tags. The data passes through radio signals and communication between devices is controlled by a medium access control (MAC) protocol. In this case, those rules are referenced by Dynamic-Framed Slotted ALOHA (DFSA). As the operation of these systems is based in electronic devices, a major concern is their computational cost. This is because it impacts energy efficiency and also limits many RFID devices as they have low computing power. This document presents a program implementation with the purpose of simulating an RFID tag reader. We also implement a few algorithms knows as estimators, which are part of the DFSA protocol. Once the simulations are done, the estimated cost for time spent during the tag identification process is compared between the estimators. Finally, we evaluate how the reference FLOP cost of known estimator algorithms correlates with the average total processing time in the tag identification process in the simulator. Results show that the used values are satisfactory when estimating computational cost.

Keywords: Internet of Things. RFID. DFSA. Computational Cost.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

RFID	Identificação por Radiofrequência
DFSA	Dynamic-Framed Slotted ALOHA
tag	Etiqueta RFID
frame	Intervalo de tempo discretizado em espaços de tempo menores
slot	Espaço de tempo em um frame

SUMÁRIO

1 INTRODUÇÃO

Internet das Coisas é um conceito bastante debatido nos últimos anos. Quando falamos de IoT, estamos nos referindo a redes de objetos físicos que compartilham dados entre si para prover serviços e aplicações (**??**). Estes objetos estão conectados de diversas formas, seja por cabo de rede ou pelo ar. Eles são capazes de passar informações para a *Internet* mas não necessariamente precisam disto para fazer parte de IoT.

Algumas aplicações na Internet das Coisas têm como interesse fazer uma identificação rápida e automática de objetos (**????**). Para exemplificar, temos os armazéns de atacado. Neles, produtos de tipos diversos e em grande quantidade são guardados, tornando o processo manual de contagem ou identificação uma tarefa árdua, mesmo quando bem planejada. Como outro exemplo temos os supermercados. Em muitos deles a fila de clientes nos caixas procede lentamente, o que poderia ser acelerado caso não fosse necessária a verificação de cada produto pelo código de barras. Nos dois exemplos, identificar rapidamente e de forma automática os produtos implicaria no aumento da qualidade do serviço. Consequentemente, os clientes ficariam mais satisfeitos.

1.1 MOTIVAÇÃO

Uma das soluções populares na Internet das Coisas para a identificação automática de objetos são os sistemas de Identificação por Radiofrequência, cujos componentes básicos mais conhecidos são etiquetas e leitores RFID. A fim de protocolar a comunicação entre estes componentes surgiu o DFSA. Este protocolo utiliza algoritmos conhecidos como estimadores, cujo papel é de estimar quantas etiquetas estão a alcance do leitor.

A literatura que permeia o DFSA apresenta uma gama de possíveis estimadores, os quais apresentam diferentes complexidades de implementação. A performance computacional dos estimadores afeta a identificação rápida de objetos com RFID, tornando-se importante estudá-la. Atualmente existe uma carência de pesquisas que identifiquem o custo computacional de estimadores. A literatura (**??**) estuda este custo para estimadores como o Vahedi, o ILCM, o Chen e o Vogt. Contudo, a quantidade de etiquetas identificadas é limitada em até 250 etiquetas. Complementar ao problema, para estimar o custo destes estimadores são utilizados valores de referência para operações de ponto flutuante. Estes valores são os mesmos a vários anos (**??**), o que sugere que estes devam ser atualizados.

1.2 OBJETIVO

Para avaliar a complexidade e seu potencial impacto no hardware de dispositivos RFID, alguns valores de referência (**????**) são utilizados na literatura para estimar o custo computacional ou custo *FLOP* de estimadores. O presente trabalho tem como objetivo avaliar como o custo computacional de referência de cada estimador se correlaciona com o tempo médio total de processamento gasto durante a identificação de etiquetas no simulador. Para alcançar este objetivo, as seguintes etapas são estabelecidas:

- Construir um simulador RFID;
- Implementar, no simulador, alguns dos estimadores conhecidos da literatura;
- Otimizar o simulador usando threads e avaliar o ganho de desempenho.
- Simular os estimadores nos mesmos cenários de simulações existentes da literatura;
- Validar as implementações dos estimadores relativas ao tempo de identificação em *slots*;
- Estimar o custo computacional das implementações dos estimadores baseado no custo computacional de referência;
- Gerar dados relativos ao tempo médio total de processamento gasto durante a identificação de etiquetas no simulador;
- Avaliar a correlação entre o custo *FLOP* simulado e o tempo médio total de processamento.

1.3 CONTRIBUIÇÕES

Os resultados alcançados mostram que alguns dos valores de referência do custo computacional de operações de ponto flutuante podem precisar de atualização. As principais contribuições deste trabalho são fornecer um simulador RFID em java, relacionar quantidade de *slots* gastos com tempo de identificação, apresentar comparações de performance dos estimadores visando o custo computacional e relacionar este custo ao tempo médio total de processamento dos algoritmos de cada estimador.

1.4 ORGANIZAÇÃO

A estrutura do trabalho divide-se em capítulos. O Capítulo 2 apresenta uma breve explanação do funcionamento de cada um dos estimadores no simulador. Especificamente, os estimadores são: *Lower Bound*; *Schoute*; *Vogt*; *Vogt (Eom-Lee)*; *Improved Vogt II (IV-2)*

e *Eom-Lee*. O Capítulo 3 detalha o funcionamento do simulador implementado na proposta. O Capítulo 4 apresenta os resultados obtidos relativos ao tempo de execução total no simulador, comparando o uso de threads com a abordagem sequencial. O Capítulo 5 faz uma análise do desempenho dos estimadores visando o tempo de identificação em quantidade de *slots* e o tempo de identificação em segundos. O Capítulo 6 mostra os resultados calculados das estimativas de custo computacional dos estimadores. Por fim, o Capítulo 7 compara o custo computacional dos estimadores com o tempo de processamento gasto no processo de identificação de etiquetas no simulador.

2 INTERNET DAS COISAS

Alguns pesquisadores definem o conceito de Internet das Coisas como algo que, sem as limitações de hardware atuais e uma vez implementado serviria como facilitador na solução dos desafios da sociedade ao se coletar dados e processá-los, tornando a vida mais fácil, produtiva e saudável (**????**). Outros pesquisadores têm uma visão menos antrópica e a definem como a união de um grande número de objetos conectados em uma infraestrutura, compartilhando informações entre si e coordenando decisões (**??**).

A fim de facilitar o desenvolvimento de tecnologias para IoT, propostas de modelos arquiteturais foram elaborados. As arquiteturas mais populares são formadas por quatro ou cinco camadas (**??**), as quais seriam:

1. *Redes e Sensores*: de mais baixo nível, formada por objetos integrados com sensores que possibilitam a ligação do mundo físico com o mundo digital;
2. *Redes e Gateways*: esta camada serve como meio para a transferência dos dados da camada abaixo para uma camada superior. Também abrange os protocolos que gerenciam este meio;
3. *Serviços de Gerenciamento ou Middleware*: tem o objetivo de analisar os dados coletados na primeira camada e processá-los, fornecendo-os para algum serviço. Estão nela implementados os serviços de controle de acesso, encriptação, mineração de dados, entre outros;
4. *Aplicação*: cujo papel é englobar os sistemas que cuidam do front-end (normalmente a interface com o usuário) da Internet das Coisas. Na maioria dos casos, é ligada diretamente ao usuário final. As aplicações desta camada envolvem áreas diversas como soluções ambientais, de saúde, rastreamento, transporte. No geral, envolve áreas que possam ser exploradas com alguma aplicação em IoT;
5. *Negócios*: é uma abstração da camada de Aplicação voltada para integração com serviços na nuvem e aplicações em Big Data.

Um dos desafios da área de Internet das Coisas é prover uma comunicação eficiente e de baixo custo. Uma das abordagens que tentam solucionar este problema, parcialmente ou não, são os sistemas de Identificação por Radiofrequência. Eles estão implementados nas camadas mais abaixo na arquitetura de IoT porém também atuam ou podem atuar em outras camadas.

3 SISTEMAS RFID

Sistemas RFID (**??**) básicos apresentam três componentes. O primeiro componente é uma antena ou bobina. O segundo componente é um transceptor. Por fim, o terceiro componente é um transponder. O papel da antena é servir como canal de comunicação entre o transceptor e o transponder. Ambos o transponder e o transceptor normalmente possuem antenas, formando dispositivos conhecidos como etiqueta RFID e leitor RFID respectivamente. O leitor pode ou não ser portátil. Ele é encarregado de transmitir sinais de rádio para etiquetas que estejam dentro de seu alcance com o objetivo de extrair informações das mesmas. Ele está associado a uma base de dados, a qual pode estar em memória interna ou em um ou mais servidores externos. A etiqueta RFID conta com circuito integrado e pode ser categorizada como ativa, semi-ativa ou passiva. Uma etiqueta é considerada ativa quando ela possui uma fonte de energia própria, a qual usualmente é fornecida via bateria. Para uma etiqueta classificada como passiva, a fonte de alimentação própria não existe. Neste caso, o sinal de rádio é transmitido por *backscattering*, ou seja, por reflexão dos sinais enviados pelo leitor à etiqueta. A tag semi-ativa utiliza *backscattering* para se comunicar e usa a bateria para processamento.

Um ponto importante a se considerar é a possibilidade de uma etiqueta executar também o papel de leitor RFID. Basta que esta seja do tipo ativa e tenha capacidade de computação suficiente. No entanto, são as etiquetas passivas que demonstram o grande potencial dos sistemas RFID ao contar com seu baixo custo energético e de produção (estando este praticamente a cargo do leitor RFID) e sua autonomia (**??**). Logo, atentaremos para um cenário onde o leitor é único e se comunica com múltiplas etiquetas passivas. Para se obter a comunicação entre leitor e etiqueta, deve-se levar em conta a possibilidade de uma ou mais etiquetas transmitirem sinais no mesmo instante, o que acarreta na possibilidade de gerar ruído, assim prejudicando a comunicação. Quando esta interferência ocorre, a chamamos de colisão entre etiquetas. Portanto é necessário um mediador, ou seja, um protocolo de acesso ao meio (MAC) para orquestrar o envio de mensagens entre leitor e etiquetas.

4 PROTOCOLO DFSA

O DFSA é um protocolo MAC (anticolisão) bastante popular em sistemas RFID (**??**). Padronizado pela EPC Global (**??**), ele se baseia no Slotted ALOHA, cujo papel também é o de protocolar o acesso ao meio. A proposta do Slotted ALOHA é discretizar o tempo em intervalos também conhecidos como *slots*. Cada intervalo seria utilizado para passar mensagens. Caso ocorra uma colisão dos sinais de rádio os emissores das mensagens as reenviarão após um intervalo aleatório de tempo. O tempo de espera do emissor antes que ele possa reenviar uma mensagem é imposto pelo protocolo e afeta a quantidade de colisões ocorridas. Consequentemente, isto influencia no tempo total de identificação das etiquetas. No DFSA, existe uma extensão conhecida como *Early-End* (**??**) que busca diminuir o tempo gasto por *slots* vazios ao encerrá-los mais rapidamente.

No Slotted ALOHA, cada intervalo de tempo é considerado como um quadro ou *frame*. O mesmo não ocorre no DFSA. A proposta define o conceito de quadro como um ou mais intervalos de tempo. Além disso, estes quadros podem variar de tamanho a depender da implementação de seu estimador, um algoritmo utilizado ao final de cada turno de identificações (final do quadro). Os estimadores têm como objetivo minimizar as colisões entre etiquetas. A problemática específica dos estimadores é dada pela incógnita que é a quantidade de etiquetas dentro do alcance do leitor. Como este número não é conhecido inicialmente, um tamanho de quadro inicial é adotado e ao longo do tempo ajustado pelo estimador.

5 ESTIMADORES

Uma parte importante do protocolo DFSA para o RFID é a necessidade de um estimador. Em um cenário ideal, a informação da quantidade de etiquetas a serem identificadas já é conhecida. Esta informação também é conhecida como *backlog*. Contudo, em um cenário real, esta informação não existe. Logo, foram desenvolvidos algoritmos que almejam estimar a quantidade de etiquetas a alcance do leitor, estes algoritmos são conhecidos como estimadores.

Quanto mais próximo do valor real é a estimativa, mais preciso é o estimador. O fato de um estimador possuir uma estimativa mais precisa não implica em ser o melhor estimador a se usar. Um estimador, por exemplo, pode ser mais lento para identificar todas as etiquetas, mas do ponto de vista energético ele poderia ser mais vantajoso. Custo computacional está relacionado a tempo de processamento, que por sua vez está ligado a gasto energético, o que se torna um estudo de caso interessante pois muitas das aplicações de IoT requerem autonomia duradoura em seus dispositivos.

Como o DFSA utiliza um sistema de quadros ou *frames* de tamanho dinâmico discretizados em espaços de tempo, a lógica de funcionamento dos estimadores é baseada neste tamanho para poder calcular o tamanho do próximo quadro. A seguir estão explanadas as implementações dos estimadores usadas nas simulações desta pesquisa. Os estimadores são: *Eom-Lee*, *Vogt*, *Vogt(Eom-Lee)*, *Improved Vogt 2* (IV-2), *Schoute* e *LowerBound*.

Todas as implementações retornam um tamanho estimado de quadro em número de *slots*, que é representada por `frameEnd` no código fonte apresentado no Apêndice B. É importante salientar que estes são protocolos que não recalculam o tamanho do quadro a cada intervalo de tempo. Assim, idealmente, o tamanho do próximo *frame* seria igual ao *backlog*. As entradas C, V e S são, respectivamente, *slots* em colisão, *slots* vazios e *slots* bem-sucedidos. Para facilitar, uma lista de parâmetros utilizados nas implementações encontra-se na Tabela 5.1.

Tabela 5.1 – Lista de Parâmetros

Parâmetro	Descrição
f	tamanho do quadro em análise
$\hat{f}$	tamanho do quadro seguinte calculado pelo estimador
C	quantidade de *slots* em colisão
V	quantidade de *slots* vazios
S	quantidade de *slots* bem sucedidos
$\hat{n}$	quantidade estimada de etiquetas restantes não identificadas (*backlog*)

5.1 LOWER BOUND E SCHOUTE

Os estimadores Lower Bound e Schoute são os de mais fácil implementação dentre os estimadores. O segundo conta com uma base teórica mais complexa por trás de sua simplicidade. O Lower Bound baseia-se na premissa de que se há *slots* em colisão, então pelo menos duas etiquetas competiram por aquele *slot* (**??**). Assim, o tamanho estimado para o próximo quadro é duas vezes C. Representamos a estimativa da quantidade de etiquetas que competiram (transmitiram em algum *slot*) no *frame* analisado por $\hat{n}$. A Equação 5.2 representa este parâmetro.

Dentre os estimadores vistos neste trabalho, o Lower Bound nos dá a estimativa do *backlog* com menor acurácia. Por isto ele é habitualmente usado como caso base em comparações de desempenho. Representamos o tamanho do próximo quadro calculado pelo estimador com o parâmetro de símbolo $\hat{f}$. O cálculo deste parâmetro pelo Lower Bound é apresentado na Equação 5.1.

$$\hat{f} = 2 \cdot C \tag{5.1}$$

$$\hat{n} = S + \hat{f} = S + 2 \cdot C \tag{5.2}$$

$$\hat{f} = 2.39 \cdot C \tag{5.3}$$

A Equação 5.3 apresenta a estimativa do tamanho do próximo *frame* segundo Schoute. Schoute, neste estimador, considerou uma chegada de pacotes do tipo Poisson (**??**). A estatística resultou no valor de 2,39 a ser multiplicado pela quantidade de *slots* em colisão no frame anterior. Para se obter o *backlog* estimado devemos somar a $\hat{f}$ o número de etiquetas identificadas (S), como demonstrado na Equação 5.4.

$$\hat{n} = S + \hat{f} = S + (2.39 \cdot C) \tag{5.4}$$

5.2 VOGT

O estimador proposto por Vogt (**??**) baseia-se em distribuição binomial para encarar o problema de ocupação que é o de etiquetas transmitindo nos *slots* de um quadro.

$$B_{n,\frac{1}{N}}(r) = \binom{n}{r} \left(\frac{1}{N}\right)^r \left(1 - \frac{1}{N}\right)^{n-r} \tag{5.5}$$

Dados N *slots* e n etiquetas, a quantidade r de etiquetas em um determinado *slot* é binomialmente distribuida pelos parâmetros n e $\frac{1}{N}$. A Equação 5.5 é a representação desta binomial. Vogt, através da Desigualdade de Chebyshev, definiu uma função de aproximação ε cujo resultado é o valor de n para que haja a minimização da distância entre os valores obtidos de V, S e C bem como os valores estimados para os mesmos. Ou seja, a distância entre os vetores $\langle V, S, C\rangle$ e $\left\langle a_v^{N,n}, a_s^{N,n}, a_c^{N,n}\right\rangle$. Esta função é apresentada na Equação 5.6.

$$\varepsilon\left(N, V, S, C, n\right) = \min_{n} \left| \begin{pmatrix} a_v^{N,n} \\ a_s^{N,n} \\ a_c^{N,n} \end{pmatrix} - \begin{pmatrix} V \\ S \\ C \end{pmatrix} \right|, \tag{5.6}$$

$$\hat{n} = \underset{n \geq 1}{\operatorname{argmin}} \ \varepsilon\left(N, V, S, C, n\right) \tag{5.7}$$

A Equação 5.6 é utilizada para computar o valor de $\hat{n}$. Por sua vez, este cálculo é representado pela Equação 5.7. Na época em que propôs esta solução, Vogt encontrou-se limitado por restrições de hardware, o que afetou o tamanho permitido para os quadros. O tamanho era definido pelo número estimado de etiquetas $\hat{n}$ e deveria ser uma potência de 2 com limite superior equivalente a 256 *slots* e limite inferior de 16 *slots*. A Tabela 5.2 mostra a relação entre intervalo para valores de $\hat{n}$ e o tamanho de quadro sugerido equivalente. Para determinar qual dos valores utilizar, por exemplo, para $\hat{n} = 120$, uma função de aproximação por divisões e multiplicações sucessivas é usada. Detalhes da implementação estão contidos no Apêndice B na seção B.8.

Tabela 5.2 – Tamanhos de quadros em relação à $\hat{n}$

$\hat{f}$	$\hat{n} \in [x, y]$	$\hat{f}$	$\hat{n} \in [x, y]$
16	$[1, 9]$	128	$[51, 129]$
32	$[10, 27]$	256	$[112, \infty]$
64	$[17, 56]$		

Vogt não utiliza a Equação 5.7 para todos os cenários. Ele a utiliza quando algum *slot* do quadro atual não está em colisão. Portanto, quando todos os *slots* do *frame* estão em colisão ($f == C$), ele usa o estimador Lower Bound. Assim, o tamanho estimado para o próximo quadro, neste cenário, é $\hat{f} = \min\left(2 \cdot C,\ 256\right)$.

5.3 VOGT (EOM-LEE)

Sabendo-se da limitação de tamanho de *frame* do Vogt original, apresentou-se uma versão modificada do mesmo. O proposto, utilizado em comparativos em (**??**), removia a limitação, possibilitando que o quadro assumisse qualquer tamanho de acordo com o retorno da Equação 5.7, ou seja, o tamanho do quadro não era obrigatoriamente uma

potência de 2 e não havia mais limite superior de 256 *slots* nem limite inferior de 16 *slots*. Ademais, o estimador Vogt(Eom-Lee) tem a mesma estratégia do estimador Vogt.

Quando algum *slot* no quadro atual não está em colisão, as Equações 5.6 e 5.7 ainda são utilizadas. No entanto, o tamanho do próximo quadro é dado por $\hat{f} = \hat{n} - S$. Quando todos os *slots* do quadro f estão em colisão, o Lower Bound é utilizado. Neste caso, o tamanho do próximo quadro é o dobro do tamanho do quadro finalizado.

5.4 IMPROVED VOGT II (IV-2)

Em (**??**) foi proposta uma modificação do estimador Vogt. A mudança observada foi no tratamento do caso onde todos os *slots* estivessem em colisão. O Vogt original adota o Lower Bound como solução, no entanto, o Improved Vogt I executa mais cálculos baseados na Equação 5.6 como alternativa. As iterações dos cálculos têm como critério de parada um valor ϵ e elas encerram quando $\epsilon < \epsilon_{anterior}$. Sabendo disso, os autores executaram simulações comparando os valores estimados de $\hat{n}$ em função do tamanho do quadro f.

As simulações executadas pelos autores resultaram em equações de reta que podem então ser utilizadas para aprimorar o IV-I. Foi então sugerido o estimador Improved Vogt II, onde para o cenário de quadro apresentando todos os *slots* em colisão fosse requisitado um valor δ. Este valor é então passado para uma seleção de funções de equação de reta e com isto encontra-se o valor do tamanho do próximo quadro. Em suma, o valor δ altera o valor de um fator multiplicativo da função do estimador. Isto faz com que o custo computacional acrescentado pelo Improved Vogt I seja reduzido significativamente.

5.5 EOM-LEE

Em (**??**), foi proposto um algoritmo iterativo para resolver o problema da estimativa da quantidade de etiquetas restantes a serem identificadas. Para se chegar a este resultado, a proposta de Eom-Lee assume que o tamanho do próximo frame $\hat{f}$ é equivalente à quantidade de *slots* em colisão multiplicada por um fator proporcional γ_{k^*}. Logo, temos que:

$$\hat{f} = \gamma_{k^*} \cdot C \tag{5.8}$$

O valor γ_{k^*} é obtido ao final das iterações do algoritmo proposto. Para cada iteração, um novo γ_k é calculado. A iteração continua enquanto $|\Delta| \geq \epsilon$. Δ é a diferença entre o novo valor de γ e seu valor anterior. Logo $|\Delta| = |\gamma_{k-1} - \gamma_k|$. O limiar ϵ é o mesmo sugerido

por Eom-Lee (**??**), cujo valor é 0.001. γ_k é calculado através da Equação 5.9.

$$\gamma_k = \frac{1 - e^{-\frac{1}{\beta_k}}}{\beta_k \left(1 - \left(1 + \frac{1}{\beta_k}\right) e^{-\frac{1}{\beta_k}}\right)} \tag{5.9}$$

A quantidade estimada de etiquetas que competiram por *slots* é alcançada ao se multiplicar um determinado valor β_{k^*} pelo tamanho de frame estimado. Assim podemos definir sua equação:

$$\hat{n} = \frac{\hat{f}}{\beta_{k^*}} \tag{5.10}$$

O valor de β_{k^*} também é alcançado ao final da iteração, onde para cada iteração um novo β_k é calculado. A Equação 5.11 mostra como se obtém este valor. Para o primeiro passo da iteração, os valores iniciais são $\beta_1 = \infty$ e $\gamma_1 = 2$.

$$\beta_k = \frac{L}{\gamma_{k-1} \cdot C + S} \tag{5.11}$$

6 O SIMULADOR

Visando alcançar o objetivo geral do trabalho, um simulador foi implementado. A familiaridade com a linguagem Java foi a razão para que esta fosse escolhida no desenvolvimento do programa. O simulador conta com uma estrutura que possibilita o uso de Threads para reduzir o tempo de execução total do programa.

Um fluxo de execução da lógica de Threads do programa está representado pela Figura 6.1. Nela está inicialmente a passagem dos parâmetros de configuração do programa. A listagem dos parâmetros suportados é como segue:

- **Dinfo**: Tipo da simulação;
- **Dinit-frame**: Tamanho inicial de quadro para os estimadores;
- **Dtags**: Quantidade de etiquetas;
- **Dsims**: Quantidade de simulações por passo;
- **Dpasso**: Tamanho dos passos das simulações;
- **Druntime**: Conta tempo de execução do simulador com diferentes quantidades de threads;
- **Dthreaded**: Usa threads para acelerar execução.

Caso a opção Dthreaded seja ativada, um ThreadExecutor (gerenciador de Threads para Java), é utilizado passando-se uma quantidade fixa de Threads, as quais receberão trabalhos (simulações) para executar. Uma vez encerrados os trabalhos, o programa armazena em disco os dados coletados e encerra.

Cada trabalho executado por uma Thread também recebe parâmetros, tais como: parte dos parâmetros definidos na etapa inicial do fluxo na Figura 6.1; o número da Thread e as variáveis necessárias para armazenar os dados obtidos. A quantidade de simulações passada no início do programa é a mesma executada por cada trabalho.

Um fluxo representando a etapa entre a execução dos trabalhos e o final do programa encontra-se na Figura 6.2. O fluxo não é baseado em trabalhos anteriores, cada fluxo executa um trabalho de maneira independente e os resultados são guardados em espaços específicos da memória determinados previamente. Uma simulação é iniciada com uma limpeza no canal de comunicação, cuja estrutura é dada por um array onde cada espaço deste array é equivalente a um *slot* ou intervalo de tempo. Em um cenário ideal, o leitor de etiquetas RFID envia um sinal para cada etiqueta e em seguida espera

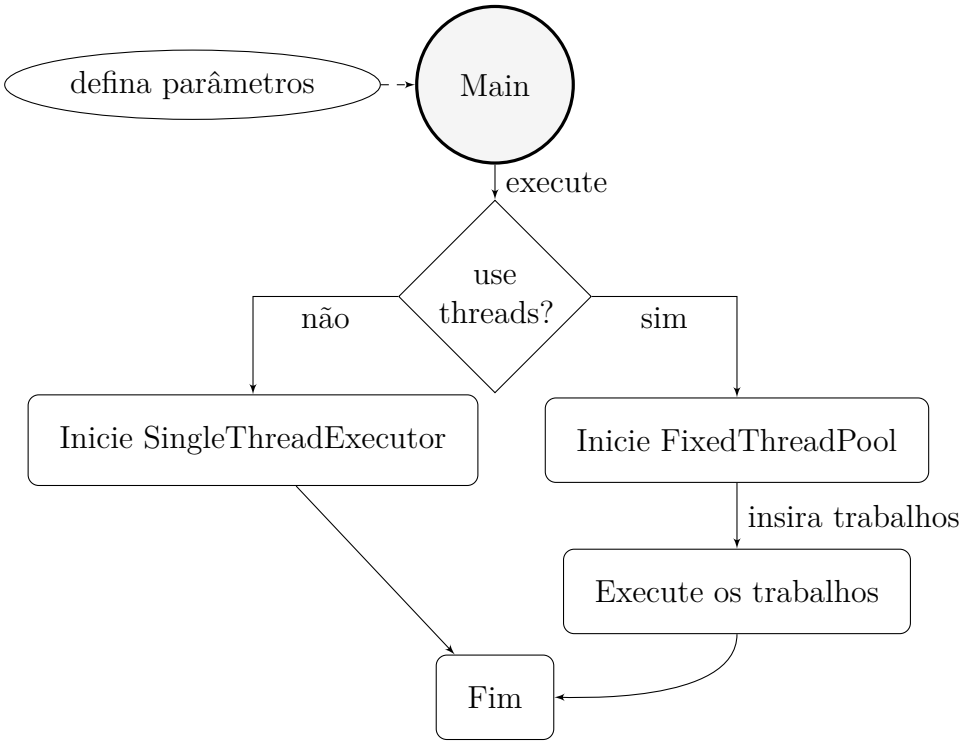

Figura 6.1 – Fluxo lógico da escolha de Threads do Simulador

por respostas das mesmas. Podemos então simular esta comunicação, ou seja, simular a incidência de respostas das etiquetas ao leitor. Para isto consideramos cada elemento do array um contador, incrementando o valor do mesmo em uma unidade quando houver etiqueta respondendo naquele instante. O elemento do array onde ocorrerá uma incidência é determinado por um algorítmo pseudoaleatório. Como estamos lidando com um cenário ideal, cada etiqueta responde apenas uma vez em cada passada pelo array. Ao final, cada *slot* do array conterá uma quantidade de tentativas de resposta. Ou seja, um *slot* que não obteve resposta terá valor 0 e será considerado um *slot* vazio. Um *slot* cujo valor é igual a 1 é considerado um *slot* bem-sucedido e um *slot* cujo valor é maior do que 1 é considerado um *slot* em colisão.

Uma vez ocorrida a contagem/análise do canal de comunicação, verifica-se se houve ou não houve colisão durante aquele quadro (conjunto dos *slots* analisados). Caso não haja colisão, não há necessidade de reajustar o tamanho do quadro e a execução continua para esta simulação. Caso haja colisão, uma chamada ao estimador é efetuada e um novo tamanho de quadro é calculado.

A cada quadro analisado indaga-se se todas as etiquetas foram identificadas, o que normalmente seria impossível visto que não se sabe a quantidade de etiquetas existentes. Contudo, esta indagação é necessária para o encerramento do simulador e para a continuação das simulações. Além disso, adotar técnicas reais para encerramento de identificação de etiquetas não faz sentido pois o ambiente de simulação não é real. Dito isto, caso as etiquetas estejam todas identificadas, a próxima simulação é executada. Caso

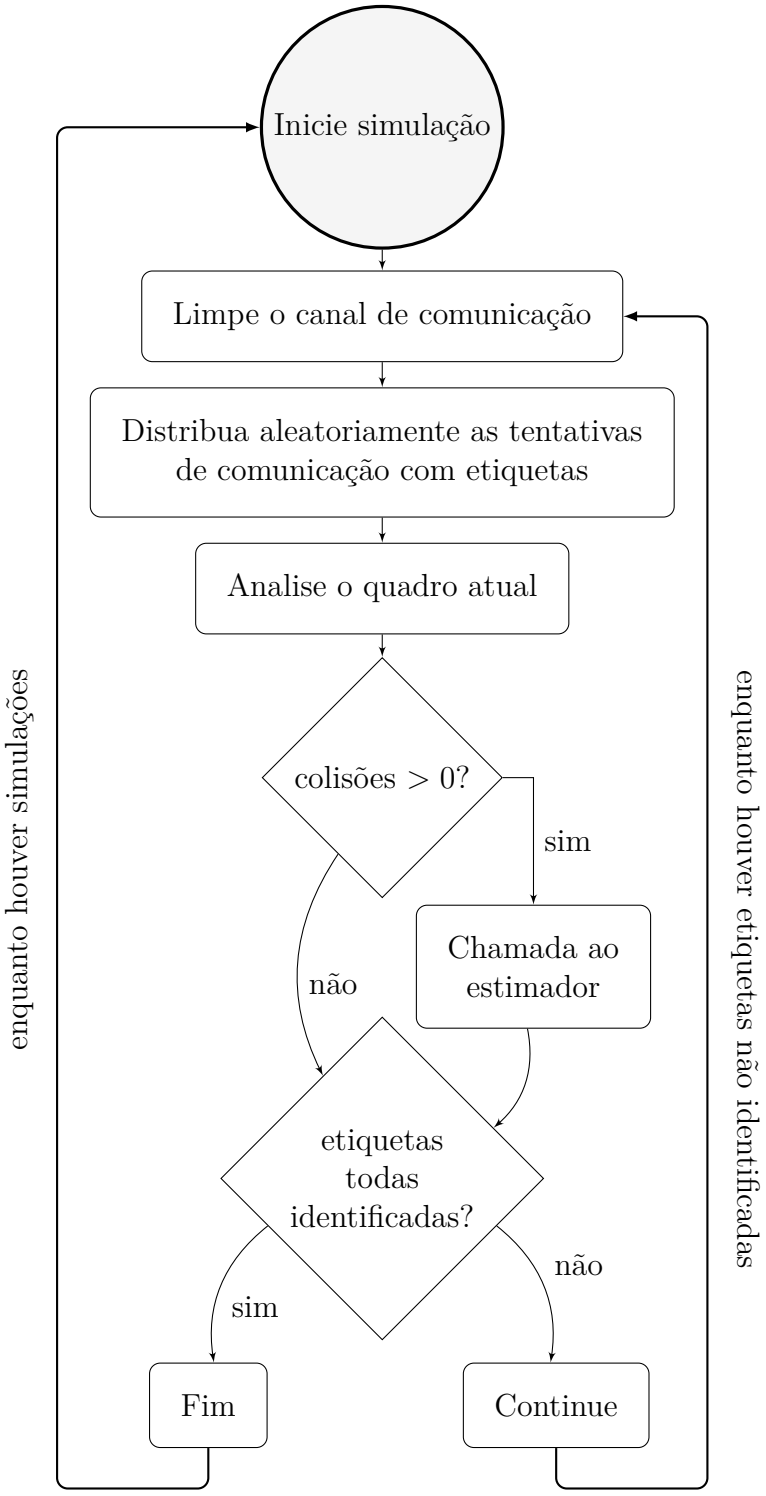

Figura 6.2 – Etapas realizadas por cada trabalho em uma Thread

contrário, limpa-se o canal e novamente é retomado o processo de distribuição de *tags* no quadro.

7 TEMPO DE EXECUÇÃO

A fim de demonstrar a utilidade do uso de Threads no simulador, foram executados testes medindo o tempo de execução total do simulador para diferentes estimadores, nos cenários de execução sequencial (1 Thread) e paralela (2, 4, 8, 16 e 32 Threads).

Nos testes foram utilizados dois processadores. O primeiro processador utilizado é o modelo i5 4670k da Intel com 4 núcleos físicos e configuração de 1 thread por núcleo. Note que esta é a thread virtual do processador e não uma thread criada pelo ThreadExecutor do simulador. A frequência de operação é de 4.3GHz para cada núcleo, ou seja, houve um processo de *overclocking* dos mesmos. O segundo processador utilizado foi um i7 5500U, com 2 núcleos físicos e configuração de 2 threads por núcleo. A frequência de operação é 2.4GHz para cada núcleo, sem processo de *overclocking*. Os mesmos parâmetros foram utilizados nas simulações de ambos os processadores. Os resultados estão apresentados na Figura 7.1.

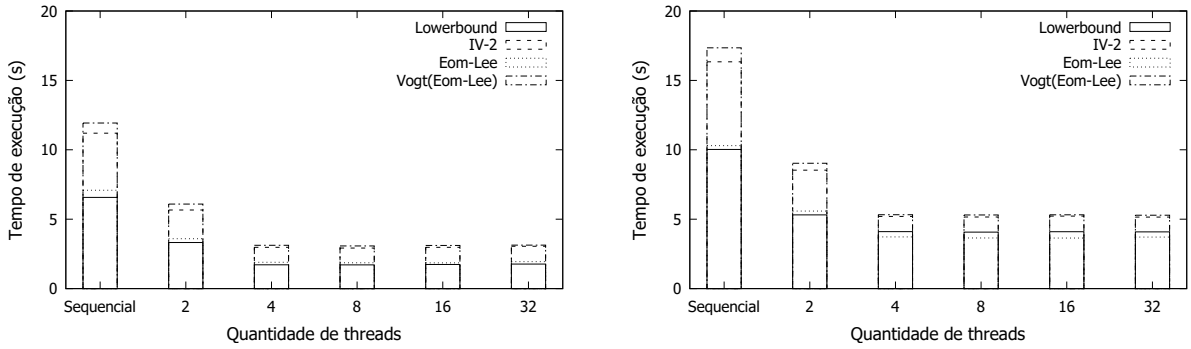

Figura 7.1 – Tempo de execução total do simulador para os processadores Intel i5 4670k @4.3GHz (à esquerda) e Intel i7 5500U @2.4GHz (à direita).

Ao observarmos os resultados da Figura 7.1, nota-se que o i5 4670k foi quem mais se beneficiou percentualmente do uso de threads. Isto é devido ao fato de possuir mais núcleos físicos e assim executando mais tarefas em paralelo. Observa-se também que dentre os estimadores usados e apenas considerando tempo, o LowerBound é melhor aproveitado pelo simulador com o uso de threads por ser um estimador mais simples, exigindo uma quantidade menor de cálculos.

Almejando o objetivo do estudo, novas simulações foram executadas. O parâmetro de interesse é o tempo médio de processamento gasto na chamada de um estimador. A quantidade de etiquetas identificadas varia de 1 a 1000 nas simulações. Os resultados foram classificados por tamanho inicial de quadro (64 ou 128 *slots*) e pelo uso ou não de threads pelo simulador. Em cenários com uso de threads, o ThreadExecutor foi construído com 16 threads. O número de threads foi escolhido baseado nos resultados das simulações de

tempo total de execução do simulador. Como não houve grande diferença de desempenho para quantidades de threads iguais ou superiores a 4, o número 16 foi arbitrariamente escolhido. O passo utilizado é de 10 etiquetas. Para todas as simulações a partir deste ponto no documento foram executadas 2000 simulações em cada passo.

Observando-se as Figuras 7.2, 7.3, 7.4 e 7.5 podemos inferir que o desempenho de cada estimador impacta no tempo de simulação. Do ponto do vista de computação isto é naturalmente esperado, uma vez que cada estimador tem um custo computacional associado. Uma observação importante é que ao usar threads nota-se um aumento do tempo médio gasto para cada passo quando comparado ao tempo médio gasto nas simulações sequenciais. Graficamente, utilizar um passo de 10 etiquetas faz com que a legibilidade dos resultados não seja muito boa ao observarmos os resultados de cada passo. Isto é devido ao fato de que o escalonador do sistema divide o tempo de processamento disponível do processador entre as threads presentes na *Pool de Threads* (quantidade fixa de threads usadas pelo *ThreadExecutor*), fazendo com que cada uma leve mais tempo para terminar uma tarefa quando comparado ao tempo gasto em execução sequencial. Contudo, N (vários) passos são executados por uma quantidade N de threads ao mesmo tempo, resultando em um tempo de execução total menor.

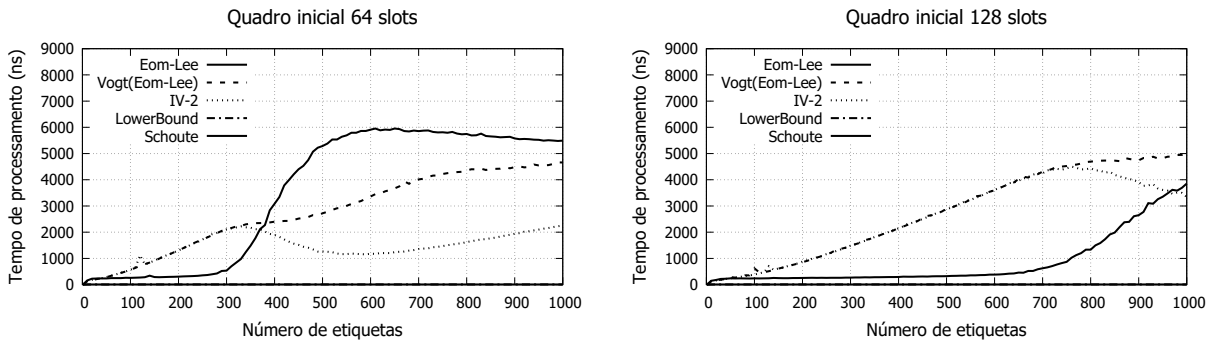

Figura 7.2 – Tempo médio de processamento dos estimadores no simulador **sem** uso de threads pelo processador i5 4670k

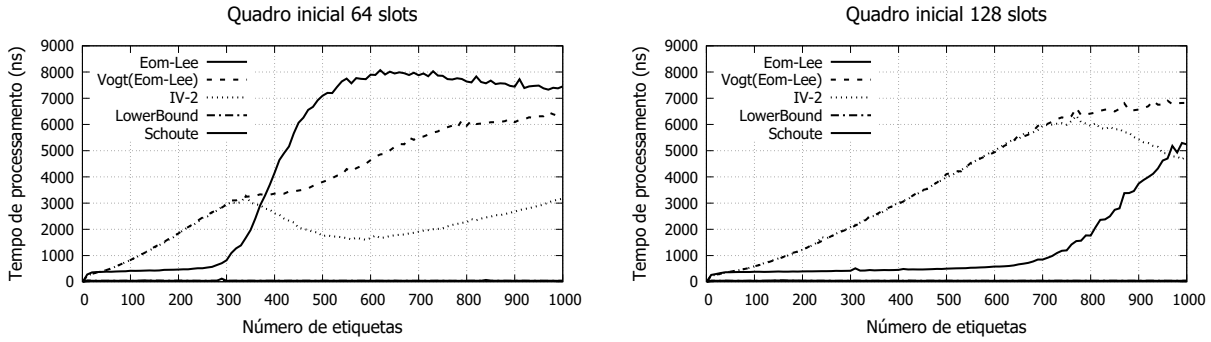

Figura 7.3 – Tempo médio de processamento dos estimadores no simulador **sem** uso de threads pelo processador i7 5500U

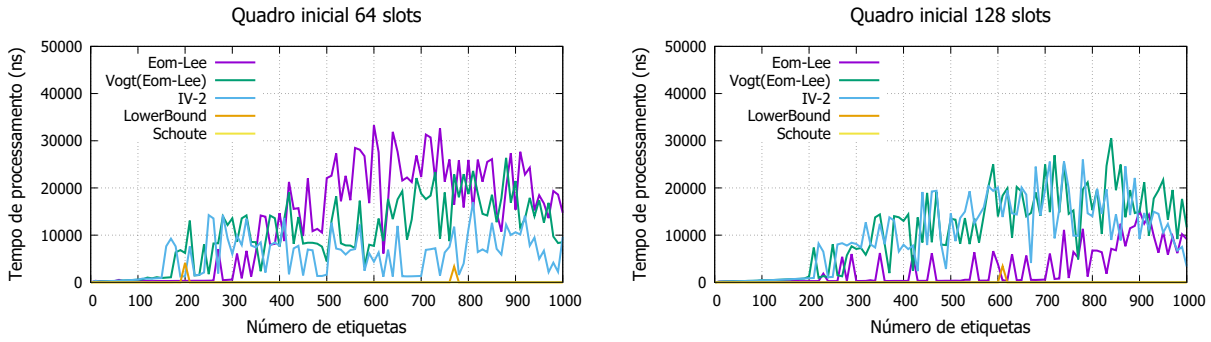

Figura 7.4 – Tempo médio de processamento dos estimadores no simulador **com** uso de threads pelo processador i5 4670k

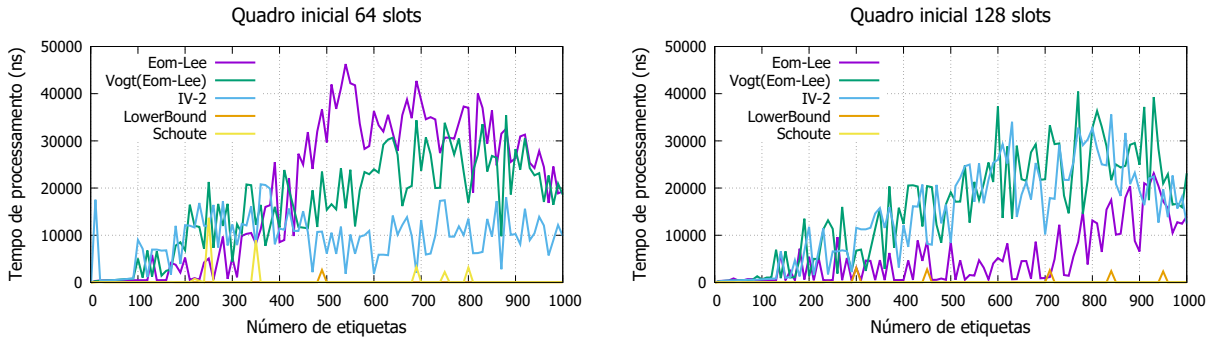

Figura 7.5 – Tempo médio de processamento dos estimadores no simulador **com** uso de threads pelo processador i7 5500U

8 ANÁLISE DE DESEMPENHO

Esta seção tem como objetivo fazer um comparativo entre os estimadores em termos de desempenho. Os parâmetros usados para as comparações serão o tempo de identificação em *slots* gastos, ou seja, a soma dos *slots* bem-sucedidos, *slots* em colisão e *slots* vazios, e o tempo de identificação em segundos.

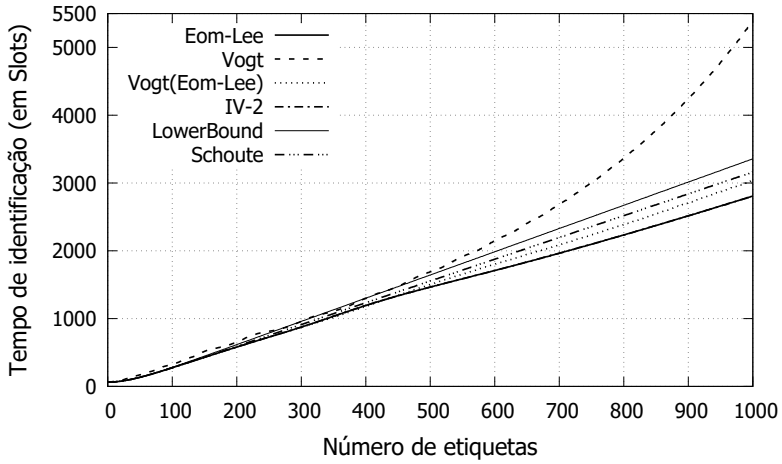

Figura 8.1 – Tempo de Identificação (Delay) em *slots* dos estimadores. Quadro inicial com 64 *slots*.

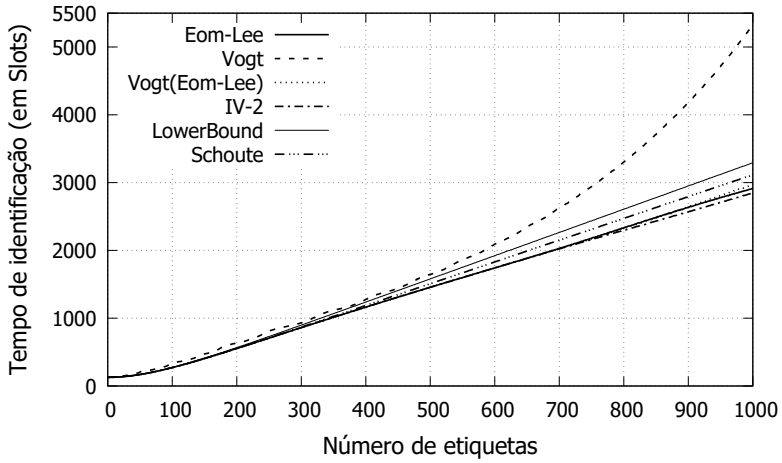

Figura 8.2 – Tempo de Identificação (Delay) em *slots* dos estimadores. Quadro inicial com 128 *slots*.

As Figuras 8.1 e 8.2 mostram a quantidade média de *slots* usados para identificar as etiquetas em cenários diferentes de tamanho de quadro inicial. Podemos observar as limitações impostas na implementação original do estimador Vogt ao perceber que, por apresentar tamanho de quadro limitado a valores de potências de 2, seu tempo de

identificação em *slots* cresce exponencialmente, tornando-se mais lento no processo de identificação de etiquetas.

Na Figura 8.1 nota-se que o estimador IV-2 obteve resultados praticamente idênticos aos do estimador Eom-Lee, o que dificulta sua visualização. A rigor, os resultados com tamanhos diferentes de quadro inicial e demais configurações iguais apresentam o mesmo comportamento. Visualmente, isto é traduzido como um deslocamento das curvas dos gráficos no eixo das abscissas. Em cenários onde o quadro inicial é suficiente para a identificação de todas as etiquetas nota-se que o tempo de identificação em *slots* é o mesmo valor do tamanho deste quadro. Nestes casos podemos ou não desconsiderar este valor uma vez que o estimador nunca é utilizado.

Para calcular o tempo de identificação em segundos foram usados os valores de referência definidos e padronizados pela EPC Global (**??**, p. 42, 43). Os valores são:

- $T_S = T_4 + T_{Query} + 2 \cdot T_1 + 2 \cdot T_2 + T_{RN16} + T_{ACK} + T_{PC+EPC+CRC16} + T_{QREP} = 2312\mu s$;
- $T_C = T_1 + T_2 + T_{RN16} = 337.5\mu s$;
- $T_V = T_1 + T_3 = 67.5\mu s$.

T_S representa o tempo gasto para um *slot* bem-sucedido, T_C o tempo gasto para um *slot* em colisão e T_V o tempo gasto para um *slot* vazio.

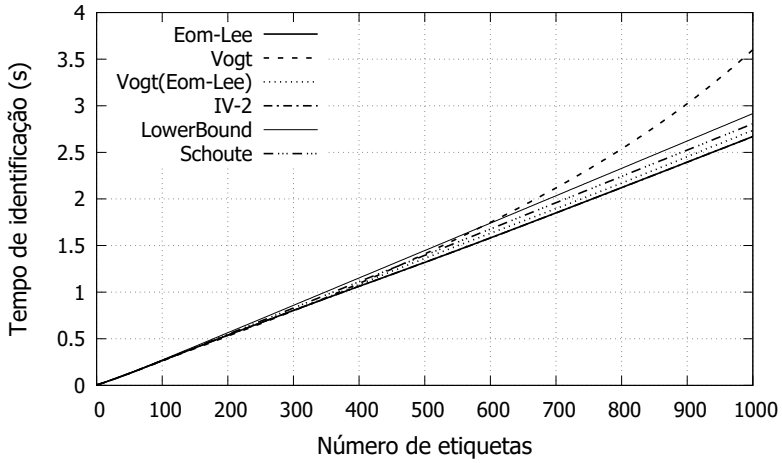

Figura 8.3 – Tempo de Identificação, estimado em segundos, dos estimadores. Quadro inicial com 64 *slots*.

As Figuras 8.3 e 8.4 apresentam o tempo médio de identificação das etiquetas estimado em segundos para cada estimador. Os resultados têm como base as 3 variáveis (S, C e V) que constroem os valores dos tempos de identificação em *slots*. Estas variáveis então são multiplicadas pelos valores padrão (T_S, T_C e T_V). Portanto, temos que o tempo estimado

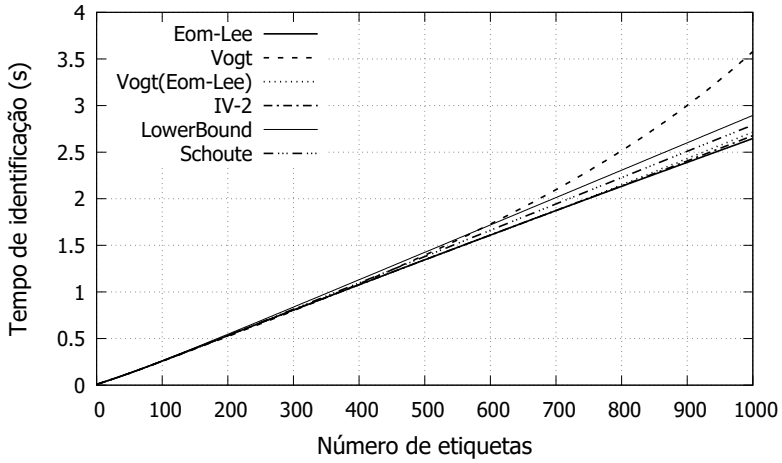

Figura 8.4 – Tempo de Identificação, estimado em segundos, dos estimadores. Quadro inicial com 128 *slots*.

de identificação em microsegundos é definido por $T_{estimado} = (T_S \cdot S) + (T_C \cdot C) + (T_V \cdot V) = (2312 \cdot S + 337.5 \cdot C + 67.5 \cdot V)\mu s$.

Nota-se que, com quadro de tamanho inicial 128 *slots*, mesmo possuindo uma quantidade total menor de quadros utilizados o estimador IV-2 gastaria mais tempo em segundos para identificar as etiquetas do que o estimador Eom-Lee. Esta mudança é decorrida do tempo gasto para processar um *slot* vazio, o qual é 5 vezes menor do que o tempo gasto para um *slot* em colisão e aproximadamente 34 vezes menor do que para um *slot* de identificação (bem-sucedido). Contudo, os *slots* de identificação são necessários para que a identificação das etiquetas ocorra.

Do ponto de vista de tamanho de quadro estimado, caso o estimador subestime a quantidade de etiquetas restantes e retorne um quadro menor, ocorrerão mais colisões. Caso o estimador superestime as etiquetas restantes e retorne um quadro maior, mais *slots* vazios existirão. Ou seja, existem cenários onde superestimar o *backlog* fornece melhores resultados de desempenho.

Um exemplo onde podemos notar o baixo custo temporal do *slot* vazio é o resultado do estimador Vogt original. Por apresentar tamanho de quadro grande (256 *slots*) em cenários de simulações com muitas etiquetas (700 1000), houve um aumento quantitativo de *slots* vazios. Quando convertidos em segundos estes valores têm uma diferença percentual entre os outros estimadores menor do que a diferença percentual dos valores em *slots*. Por fim, os desempenhos apresentados dos estimadores Vogt(Eom-Lee), IV-2 e Eom-Lee não diferem muito para uma quantidade baixa ou moderada (entre 1 e 600) de etiquetas, sendo esta diferença estatisticamente irrelevante e de erro baixo devido à grande quantidade de simulações por passo.

9 CUSTO COMPUTACIONAL

Nesta seção analisaremos o custo computacional atrelado aos estimadores apresentados. Os valores de referência associados a operações de ponto flutuante usados para o cálculo da estimativa do custo computacional dos estimadores estão apresentados na Tabela 9.1. A fim de tornar a comparação mais justa, não utilizaremos o estimador Vogt original pois este possui limitações de tamanho de quadro que não ocorrem atualmente. Estas são: a limitação do tamanho por uma potência de 2 e o tamanho limitado no intervalo de 16 a 256 *slots*. Será usado então para comparação uma implementação do Vogt sem estas limitações: o estimador Vogt (Eom-Lee).

Tabela 9.1 – Custos de operações de ponto flutuante (FLOP)

Operação	**Custo**
Adição, Subtração, Multiplicação	1
Comparação	2
Divisão, Raiz quadrada	10
Exponenciação, Logaritmo	50
Fatorial	100

9.1 CUSTO DOS ESTIMADORES LOWER BOUND E SCHOUTE

As implementações dos algoritmos do Lower Bound e do Schoute, apresentados pelo algoritmo 1 e o algoritmo 2, são marcadas por sua simplicidade. É apenas necessária uma multiplicação para determinar o tamanho do próximo quadro a ser analisado.

Algoritmo 1: Estimador Lower Bound

Entrada: C
Saída: O tamanho do próximo quadro

```
1 inicio
2 |   int f̂;
3 |   f̂ = C · 2;
4 fin
5 retorna f̂
```

Tanto o Lower Bound quanto o Schoute fazem uso de multiplicações para determinar o valor de $\hat{f}$. Para o caso destes dois estimadores, suas implementações calculam $\hat{f}$ diretamente. Sendo assim, cada chamada ao estimador é equivalente a uma multiplicação (linha 1 de ambos os algoritmos 1 e 2). Os custos computacionais de ambos para o intervalo de 1 a 1000 etiquetas estão apresentados nas Figuras 9.1 e 9.2.

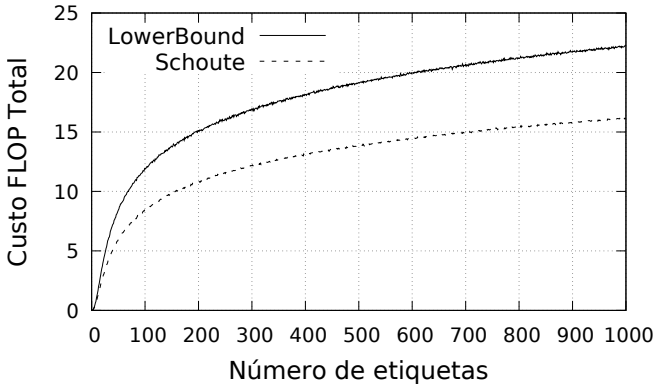

Figura 9.1 – Custo *FLOP* dos estimadores *Lower Bound* e *Schoute*. Quadro inicial com 64 *slots*.

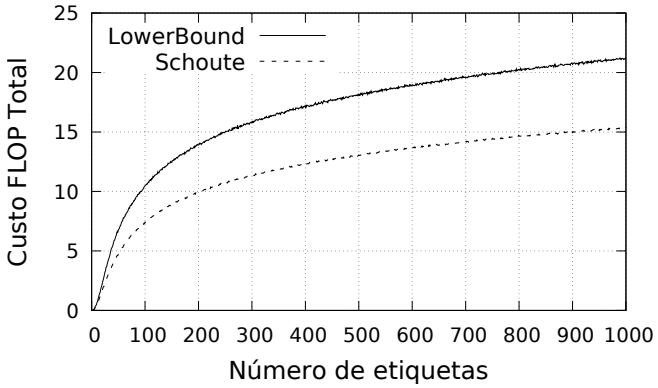

Figura 9.2 – Custo *FLOP* dos estimadores *Lower Bound* e *Schoute*. Quadro inicial com 128 *slots*.

Com o intuito de manter o custo FLOP advindo somente dos estimadores, o custo para simulações onde só houve uso do primeiro quadro é desconsiderado, pois este é o quadro é inicial e ele não foi aberto pelo estimador. As Figuras 9.1 e 9.2 apresentam as médias das 2000 simulações. O passo utilizado para a quantidade de etiquetas é igual a 1.

Observa-se que o algoritmo 2, uma implementação do estimador Schoute, ao trocar o valor multiplicativo de 2 para 2.39, demonstrou uma menor frequência de abertura de novos frames para identificar todas as etiquetas. Como consequência, a quantidade de multiplicações necessárias também diminuiu, o que resultou num custo FLOP menor.

9.2 CUSTO DO ESTIMADOR EOM-LEE

O algoritmo 3 apresenta a implementação do estimador Eom-Lee, cuja performance é relativa ao seu algoritmo iterativo. A obtenção do valor Γ que, quando multiplicado por C resulta no tamanho desejado em *slots* do próximo *frame* (**??**), é implementada na linha 13 do algoritmo 3.

Algoritmo 2: Estimador Schoute

Entrada: C
Saída: O tamanho do próximo quadro
1 **inicio**
2 | **int** $\hat{f}$;
3 | $\hat{f} = (int)Math.ceil(C \cdot 2.39)$;
4 **fin**
5 **retorna** $\hat{f}$

Algoritmo 3: Estimador Eom-Lee

Entrada: f, C, S
Saída: O tamanho do próximo quadro
1 **inicio**
2 | **double** $\beta, \gamma_{k-1}, num, den, frac$;
3 | **double** $\gamma_k = 2.0$;
4 | **repita**
5 | | $\gamma_{k-1} = \gamma_k$;
6 | | $\beta = f/((\gamma_{k-1} \cdot C) + S)$;
7 | | $frac = e^{-(1/\beta)}$;
8 | | $num = 1 - frac$;
9 | | $den = (\beta \cdot (1 - (1 + (1/\beta)) \cdot frac))$;
10 | | $\gamma_k = num/den$;
11 | **até** $|\Delta| < \epsilon$;
12 **fin**
13 **retorna** $\gamma_k \cdot C$

A verificação da condição limiar do algoritmo iterativo está implementada na linha 11. Analisamos então o custo que este algoritmo apresenta. Começando o loop na linha 5 temos apenas uma atribuição, o que não tem custo relativo de acordo com a Tabela 9.1. Na linha 6 temos uma divisão, uma multiplicação e uma soma, o que nos dá um custo igual a 12. Em seguida temos uma subtração, uma exponenciação, uma divisão e uma multiplicação (a multiplicação é devida a $(-1) \cdot (1/\beta)$ na linha 7). Portanto, as linhas 7 e 8 equivalem a um custo de 62. Por sua vez, a linha 9 tem custo 14, com uma subtração, uma soma, duas multiplicações e uma divisão. Por fim as linhas 10 e 11 possuem uma divisão, uma subtração e duas comparações (uma do valor absoluto), o que resulta num custo igual a 15. Logo, cada iteração tem custo 103 (12 + 62 + 14 + 15).

A Figura 9.3 representa o custo FLOP do estimador Eom-Lee. Observa-se que para o caso de quadro inicial de 64 *slots* há um aumento considerável no custo FLOP entre 250 e 500 etiquetas, estabilizando-se a partir daí até as 1000 etiquetas. Já para quadro inicial de 128 *slots* há um aumento, porém menor, a partir de 650 *tags* perdurando até uma quantidade de 1000 etiquetas, o que se encaixa no comportamento esperado de deslocamento no eixo das abscissas.

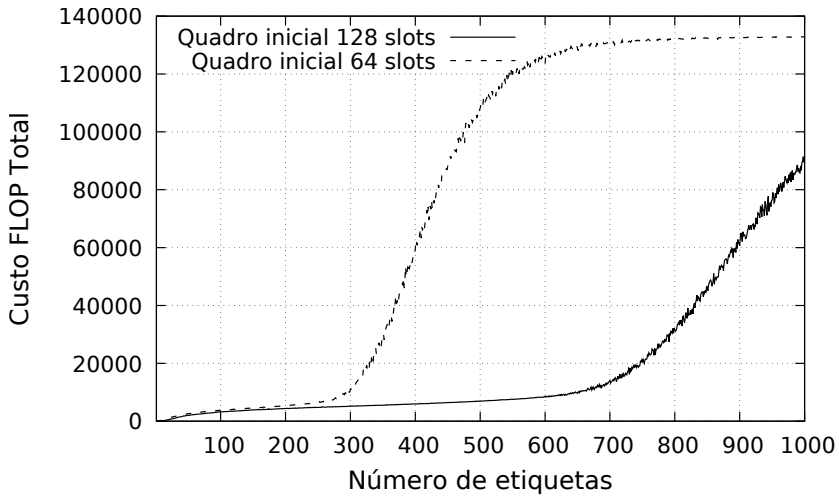

Figura 9.3 – Custo FLOP do estimador Eom-Lee.

9.3 CUSTO DO ESTIMADOR VOGT (EOM-LEE)

Para o estimador Vogt (Eom-Lee), trabalharemos com duas partes. A primeira parte é a função de que decidirá o que fazer quando os *slots* do quadro já analisado estão ou não em colisão. Esta primeira parte é representada pelo algoritmo 4. Observando-o, percebe-se a chamada da função $VogtEq3$ para o caso de nem todos os *slots* estarem em colisão. Esta função é a segunda parte da nossa implementação. Ela é o resultado da Equação 5.7, que utiliza a Equação 5.6, subtraido da quantidade de *slots* bem-sucedidos do quadro analisado. Esta função está representada pelo algoritmo 5. Note que o custo FLOP do algoritmo 4 é de $(2 + (1 \ || \ FLOP_{VogtEq3}))$.

Algoritmo 4: Estimador Vogt (Eom-Lee)

Entrada: f, C, S, V
Saída: O tamanho do próximo quadro

```
1 inicio
2 |   se (f == C) então
3 |   |   retorna (2 · C);
4 |   fim
5 |   senão
6 |   |   retorna VogtEq3(f, C, S, V);
7 |   fim
8 fin
```

Analisando o algoritmo 5 observamos que as operações de ponto flutuante se distribuem com mais intensidade pela iteração (linhas 9 a 23). Para a linha 2 temos uma adição e uma multiplicação, adicionando um custo FLOP de 2 a cada chamada do algoritmo. A partir daí o custo é contado para cada iteração do laço. Começamos com a linha 9, que tem custo 11, com uma divisão e uma subtração. Em seguida as linhas

Algoritmo 5: Implementação da função $VogtEq3$

Entrada: f, C, S, V
Saída: O tamanho do próximo quadro

1 **inicio**
2 **double** $i = S + (2 \cdot C)$;
3 **double** $\epsilon_{novo} = -1, \epsilon_{anterior} = 0$;
4 **double** $a_0 = 0, a_1 = 0, a_2 = 0$;
5 **double** $\hat{n} = i, t$;
6 **repita**
7 $t = (1 - (1/f))$;
8 $a_0 = t^{\hat{n}}$
9 $a_1 = (\hat{n} \cdot a_0)/(f \cdot t)$;
10 $a_2 = 1 - (a_1 + a_0)$;
11 $a_0 = (a_0 \cdot f) - V$;
12 $a_1 = (a_1 \cdot f) - S$;
13 $a_2 = (a_2 \cdot f) - C$;
14 $(a_0, a_1, a_2) \leftarrow (a_0^2, a_1^2, a_2^2)$;
15 $\epsilon_{anterior} = \epsilon_{novo}$;
16 $\epsilon_{novo} = \sqrt{a_0 + a_1 + a_2}$;
17 **se** $\hat{n} == i$ **então**
18 $\epsilon_{anterior} = \epsilon_{novo} + 1$;
19 **fim**
20 $\hat{n} = \hat{n} + 1$;
21 **até** $\epsilon_{novo} < \epsilon_{anterior}$;
22 **fin**
23 **retorna** $\hat{n} - 1 - S$

10, 11 e 12, com uma exponenciação, duas multiplicações, uma soma, uma divisão e uma subtração, acarretam num custo FLOP de 64. As linhas 13 a 15 têm custo 6, com 3 multiplicações e 3 subtrações. Já as linhas 16 a 18 têm custo 15, com 3 multiplicações, duas somas e uma raiz quadrada. A linha 19 possui uma comparação, com custo 2, que caso seja verdadeira acrescentará custo FLOP 1 de uma soma (note que ela somente é verdadeira uma vez). A linha 22 também tem custo 1 devido a uma soma. Por fim, a cada iteração é contada uma comparação (linha 23), de custo 2, que ao final retorna o resultado, calculado com custo FLOP 2. O total do custo do algoritmo 5 é então $(2 + 2 + 1 + (N_{iterações} \cdot (11 + 64 + 6 + 15 + 2 + 1 + 2))) = 5 + (N_{iterações} \cdot 101)$. Com isto, o custo do algoritmo 4 é $2 + (1 \,||\, 5 + (N_{iterações} \cdot 101))$.

A Figura 9.4 mostra a relação entre a quantidade de etiquetas identificadas e o custo FLOP associado do Vogt(Eom-Lee). Nota-se que de maneira análoga aos outros estimadores um tamanho de quadro inicial maior implica em menos operações de ponto flutuante. Nesta simulação, isso é válido até 550 etiquetas e a partir daí o custo se mantém estatisticamente igual tanto para um quadro inicial de 64 *slots* quanto para um quadro inicial de 128 *slots*.

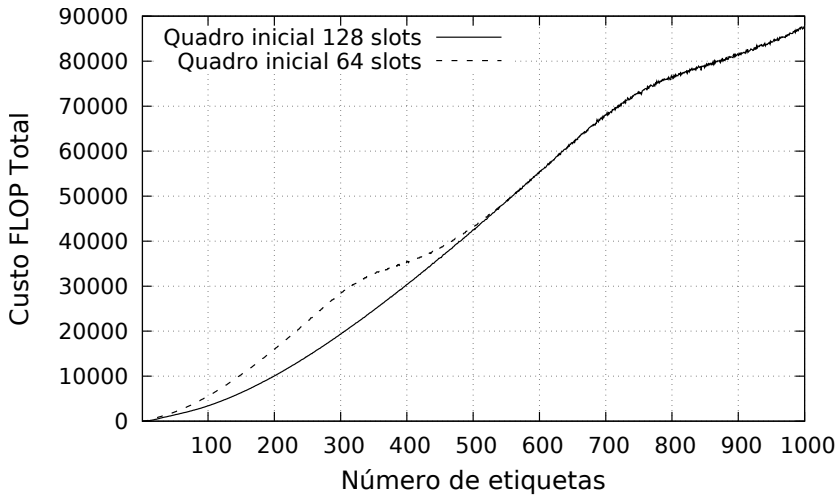

Figura 9.4 – Custo FLOP do estimador Vogt(Eom-Lee).

9.4 CUSTO DO ESTIMADOR IMPROVED VOGT II

O estimador Improved Vogt II é uma modificação do estimador Vogt original (**??**) cujo pseudocódigo é apresentado no algoritmo 6. Como este estimador propõe alterar a forma de tratamento dos cenários onde todos os *slots* estão em colisão num quadro, para cenários com $f \neq C$ continuamos utilizando o algoritmo 5. Deste modo, atemo-nos ao proposto.

Algoritmo 6: Estimador IV-2

Entrada: δ, f, C, S, V
Saída: O tamanho do próximo quadro

```
1  inicio
2      se (f == C) então
3          se δ == 1e0 então retorna 2.001001000 · (f − 1) + 2;
4          se δ == 1e1 então retorna 3.947947950 · (f − 1) + 2;
5          se δ == 1e2 então retorna 6.851851850 · (f − 1) + 2;
6          se δ == 1e3 então retorna 9.497497500 · (f − 1) + 2;
7          se δ == 1e4 então retorna 12.047047047 · (f − 1) + 2;
8          se δ == 1e5 então retorna 14.518518500 · (f − 1) + 2;
9          se δ == 1e6 então retorna 17.011011000 · (f − 1) + 2;
10     fim
11     senão
12         retorna VogtEq3(f, C, S, V);
13     fim
14 fin
```

O algoritmo 6 apresenta a implementação para o estimador IV-2. Os valores retornados nas linhas 3 a 9 são resultados das equações de reta encontradas nas simulações executadas pelos autores da proposta. Nota-se que o custo FLOP destas linhas é no máximo 17 (7 comparações, uma multiplicação, uma divisão e uma subtração) e no mínimo

5 (uma comparação, uma multiplicação, uma divisão e uma subtração). Contando com o resultado do algoritmo 5, temos que o custo FLOP total de uma chamada ao Improved Vogt II é dado pela fórmula $(5 \sim 17) \;||\; 5 + (N_{iterações} \cdot 101)$.

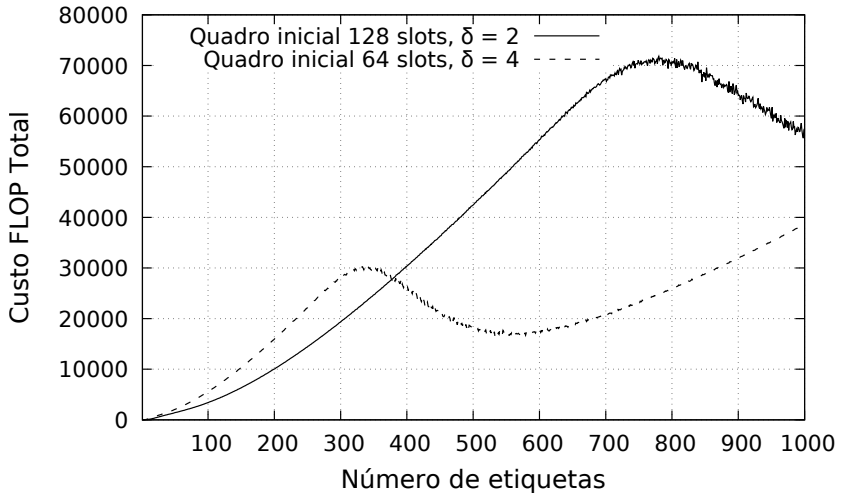

Figura 9.5 – Custo FLOP do estimador IV-2.

A Figura 9.5 explicita o custo FLOP médio para o estimador IV-2. Os tamanhos de quadro iniciais usados variam com o valor de δ. Para δ igual a $1e4$ e $1e2$ são usados respectivamente 64 e 128 *slots* no quadro inicial. Estes valores são os mesmos usados em (**??**). Observa-se claramente a vantagem de um tamanho de quadro inicial de 128 *slots* para um cenário de até 377 etiquetas. Após isto, um quadro inicial de 64 *slots* domina com um custo FLOP menor. Este deslocamento no eixo das abscissas, assim como os outros estimadores, está dentro do esperado.

10 COMPARATIVO DO CUSTO FLOP DOS ESTIMADORES

Lembrando dos resultados obtidos no Capítulo 9, sabemos que os estimadores *Lower Bound* e *Schoute* são os mais simples computacionalmente, o que sugere que, por apresentarem somente uma multiplicação, são os estimadores menos custosos. Como podemos observar nas Figuras 10.1 e 10.2, a escala logarítmica permite visualizarmos a diferença significativa no custo destes estimadores. O *Lower Bound* e o *Schoute* são os estimadores menos custosos computacionalmente. Dentre eles, o *Schoute* é mais eficiente, pois são executadas menos chamadas ao estimador uma vez que ele retorna um valor para o tamanho do próximo quadro superior ao *Lower Bound*, ou seja, ele subestima menos a quantidade restante de etiquetas a serem identificadas.

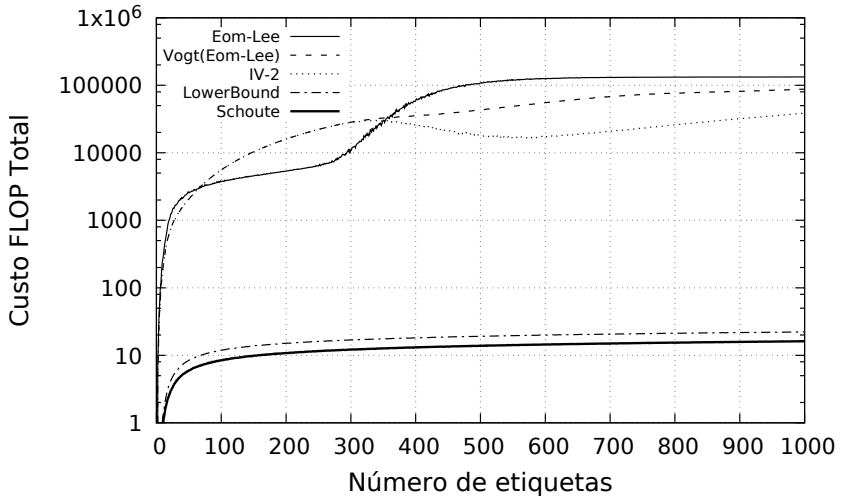

Figura 10.1 – Custo FLOP dos estimadores analisados. Quadro inicial com 64 *slots*.

Analisando os demais estimadores percebemos que seus comportamentos variam, o que se enquadra no esperado. Observamos então o cenário de quadro inicial de 64 *slots* 10.1. Neste caso, para uma faixa de 1 a 60 etiquetas, tanto o Vogt(Eom-Lee) quanto o IV-2 são menos custosos do que o Eom-Lee. De 60 a 350 etiquetas, o Eom-Lee leva a vantagem. Devido ao crescimento rápido do custo FLOP do Eom-Lee, a partir das 350 etiquetas e até as 1000 etiquetas seu desempenho perde para os outros dois. Nesta faixa, o estimador IV-2 é mais eficiente.

Para um cenário de quadro de tamanho inicial de 128 *slots*, o mesmo padrão se repete. No entanto, a vantagem dos estimadores Vogt(Eom-Lee) e IV-2 sobre o Eom-Lee é mais tardia. De um modo geral, o observado é um deslocamento do comportamento

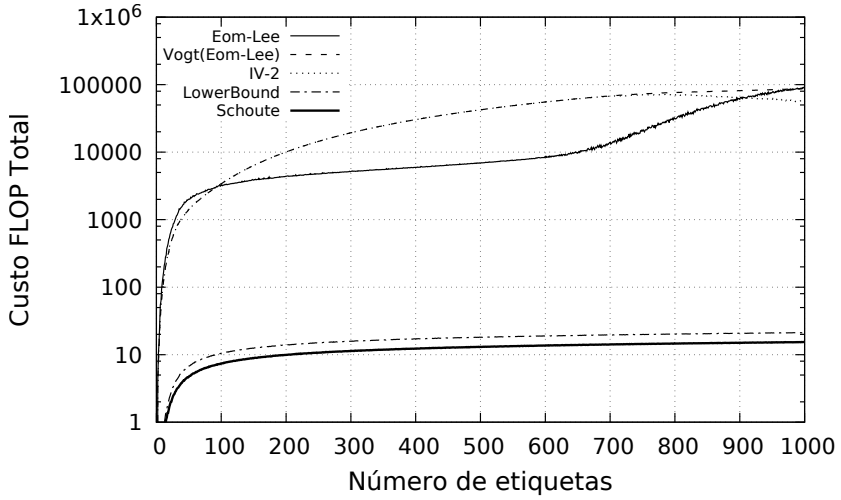

Figura 10.2 – Custo FLOP dos estimadores analisados. Quadro inicial com 128 *slots*.

apresentado na Figura 10.1. Para termos uma visualização alternativa do comparativo do custo computacional dos 3 estimadores (Eom-Lee, Vogt(Eom-Lee e IV-2) as Figuras 10.3 10.4 foram montadas. Elas não estão em escala logarítmica e não apresentam os estimadores Lower Bound e Schoute.

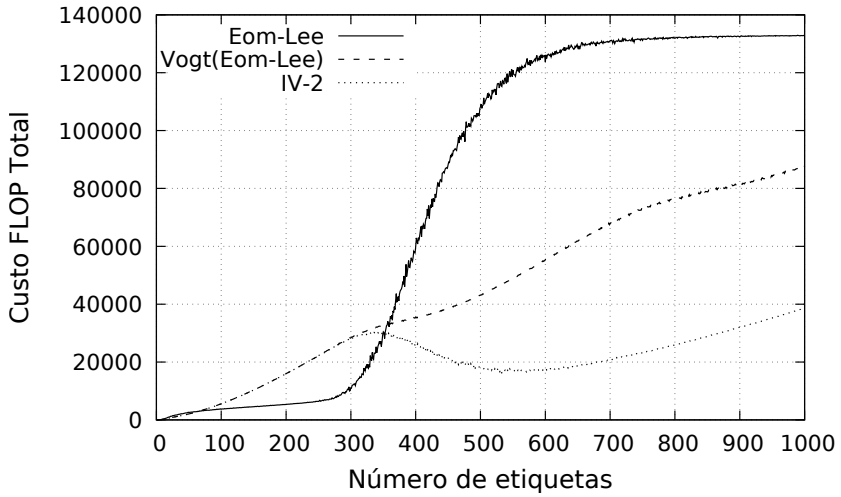

Figura 10.3 – Custo FLOP dos estimadores Vogt(Eom-Lee), IV-2 e Eom-Lee. Quadro inicial com 64 *slots*.

Uma vez que estimamos o custo computacional dos estimadores, devemos atentar para a qualidade dos mesmos. As aplicações em RFID atuais são diversas. No entanto algumas aplicações populares se destacam, como é o caso de transportadoras de produtos carregando etiquetas RFID ou o caso de instituições de ensino que desejam controlar a frequência de entrada/saída dos seus alunos. No primeiro exemplo, a quantidade de etiquetas RFID simultâneas a serem identificadas normalmente é grande. No segundo

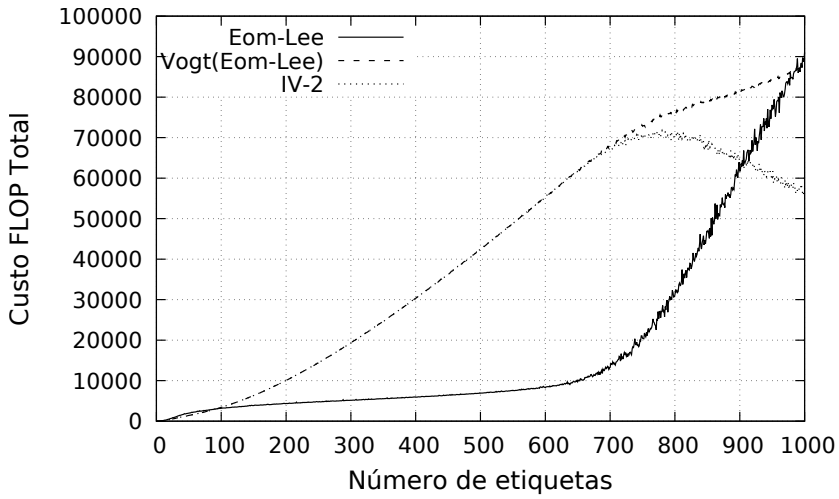

Figura 10.4 – Custo FLOP dos estimadores Vogt(Eom-Lee), IV-2 e Eom-Lee. Quadro inicial com 128 *slots*.

exemplo, esta quantidade é menor devido à limitação física de entrada e saída de pessoas.

A escolha do estimador irá depender da preferência do administrador do sistema RFID. Caso o desejado seja eficiência energética, um estimador com menor FLOP, por gastar menos tempo de processamento, pode ser o ideal. Já em um cenário com muitas etiquetas, considerar o uso de estimadores mais rápidos na identificação pode ser uma boa escolha.

10.1 CUSTO FLOP × TEMPO DE PROCESSAMENTO

É interessante notar que o custo computacional teórico dos estimadores (FLOP), cujo valor é baseado na Tabela 9.1, não está defasado. Na prática, este custo influencia diretamente o tempo de processamento. A relação entre os custos FLOP das Figuras 10.3 e 10.4 e os tempos de processamento apresentados nas Figuras 7.2 e 7.3 para seus respectivos estimadores comprova a ligação diretamente proporcional entre os dois. Percentualmente, o crescimento nos dois gráficos é estatisticamente equivalente, como por exemplo o estimador Eom-Lee, cujo custo cresce rapidamente, chegando a aproximadamente 14 vezes o custo em passos anteriores. Portanto a teoria se sustenta e isto nos permite dizer que a Tabela 9.1 serve de maneira satisfatória como um termômetro no cálculo de custo de operações de ponto flutuante. No entanto, dado que esta pesquisa utiliza um ambiente simulado, um estudo de caso real surge como necessidade na continuação do trabalho.

11 CONCLUSÃO

O RFID é uma tecnologia que ganhou força recentemente. É uma tecnologia barata, escalável e modular para prover comunicação sem fio e identificação entre dispositivos. Declarado isto, o RFID desponta com a característica de ser econômico em termos de recursos, uma vez que não necessariamente requer alimentação externa ou recarga constante, sendo assim uma ótima opção para soluções com sensoriamento e/ou indexação de elementos/dispositivos pertencentes à Internet das Coisas.

Este trabalho mostrou um estudo acerca do protocolo anticolisão DFSA e parte de seus estimadores. Abordou, especificamente, protocolos sem quebra de quadro, ou seja, o tamanho de um novo quadro só é calculado ao final do atual. Do ponto de vista energético, as escolhas mais eficientes de estimador a ser usado seriam o *Lower Bound* e o *Schoute* por possuírem os menores custos FLOP totais. O uso destes estimadores aumenta a eficiência energética mas com um custo associado, que é a perda de acurácia na identificação de etiquetas. O objetivo e principais contribuições ao se analisar os estimadores e seu desempenho computacional são:

- Apresentar dados complementares aos estudos já existentes;
- Fornecer um simulador RFID em java à comunidade científica;
- Apresentar comparações de performance dos estimadores visando o custo computacional;
- Relacionar o custo computacional ao tempo de execução dos algoritmos de cada estimador;
- Relacionar quantidade de slots gastos com tempo de identificação.

O trabalho também visa dar suporte a pesquisas futuras ao se agrupar os principais estimadores em um só documento, bem como a validação de medidas utilizadas por outros autores previamente na literatura. Especificamente, valida-se o uso da Tabela 9.1 como satisfatório quando objetivo é estimar o custo computacional. Para complementar este trabalho, pesquisas futuras sugeridas incluem:

- Executar as mesmas simulações baseadas nos parâmetros aqui apresentados para estimadores com estratégia de quebra de quadro;
- Analisar o desempenho dos estimadores vistos com parâmetro de quantidade de etiquetas com valores superiores a 1000 etiquetas;

- Comparar os estimadores aqui apresentados com técnicas de cálculo de tamanho de quadro que sejam baseadas em árvore;
- Relacionar o custo computacional ao gasto energético em baterias dos algoritmos de cada estimador;
- Traduzir o custo computacional em valores financeiros com exemplos na área industrial.

REFERÊNCIAS

Apêndices

APÊNDICE A – CÓDIGO FONTE DO FLUXO DO SIMULADOR

A.1 CLASSE PRINCIPAL DO SIMULADOR

```
1  public class Main {
2    public static void main(String[] args) throws Exception {
3      // Tipo da simulacao
4      String chartInfo = System.getProperty("info");
5
6      // Tamanho inicial de quadro para os estimadores
7      int tam_quadro_inicial =
8          Integer.valueOf(System.getProperty("init-frame"));
9
10     // Quantidade de etiquetas
11     int qtd_tags = Integer.valueOf(System.getProperty("tags"));
12
13     // Quantidade de simulacoes por passo
14     int sims = Integer.valueOf(System.getProperty("sims"));
15
16     // Tamanho dos passos das simulacoes
17     int passo = Integer.valueOf(System.getProperty("passo"));
18
19     // Contar tempo de execucao?
20     boolean countRuntime =
21         Boolean.valueOf(System.getProperty("runtime"));
22
23     // Usar threads?
24     boolean useThreads =
25         Boolean.valueOf(System.getProperty("threaded"));
26
27     // Quantidade de passos
28     int qtd_passos = qtd_tags / passo;
29
30     /* Array com os valores do eixo X. Ou seja,
31         os numeros dos passos. */
32     double[] xData;
33
34     // Variavel para iteracao
35     int i;
36     // Verifica se necessario passo inicial
37     if (passo != 1) {
38       xData = new double[qtd_passos + 1];
```

```
39        xData[0] = 1;
40        i = 1;
41      } else {
42        xData = new double[qtd_passos];
43        i = 0;
44      }
45      // Preenche o array com o restante dos passos
46      for (int j = passo; i < qtd_passos; i++, j += passo) {
47        xData[i] = j;
48      }
49      String[] noFRestimators = {
50        "LowerBound",
51        "Schoute",
52        "Eom-Lee",
53        "Vogt"
54        "Vogt(Eom-Lee)",
55        "IV-2"
56      };
57
58      /**
59        * SITEMA DE THREADS
60        */
61      // Se o objetivo eh contar tempo de execucao
62      if (countRuntime) {
63        calcRuntime(
64          sims, chartInfo,
65          tam_quadro_inicial,
66          passo, xData, noFRestimators
67        );
68      } else {
69        // Senao, execute normalmente
70        for (int k = 0; k < noFRestimators.length; k++) {
71          for (i = 0; i < 2; i++) {
72            long singleInit = System.currentTimeMillis();
73            ThreadManagerNoFrameReset simulator =
74              new ThreadManagerNoFrameReset(
75                sims,
76                noFRestimators[k],
77                chartInfo,
78                tam_quadro_inicial,
79                xData,
80                16,
81                useThreads
82              );
83            simulator.run();
84            long singleEnd = System.currentTimeMillis();
85            System.out.println(
86              "\nRuntime: " + (singleEnd - singleInit) + "ms"
```

```
87            );
88          }
89        }
90      }
91
92    }
93
94    private static void calcRuntime(
95        int sims,
96        String chartInfo,
97        int tam_quadro_inicial,
98        int passo,
99        double[] xData,
100       String[] noFRestimators)
101   {
102     int i;
103     ArrayList<Dado> dados = new ArrayList<Dado>();
104
105     // dados.add(new Dado(1)); // Sequencial
106
107     for (i = 1; i < 64; i *= 2) {
108       dados.add(new Dado(i)); // Threads
109     }
110
111     int timeSims = 2;
112     boolean threaded = false;
113     Dado dado;
114
115     for (int k = 0; k < noFRestimators.length; k++) {
116       long alltime = System.currentTimeMillis();
117       for (i = 0; i < dados.size(); i++) {
118         dado = dados.get(i);
119         if (dado.getN() > 1)
120           threaded = true;
121         else
122           threaded = false;
123         // Simulador sequencial para cada estimador
124         long init = System.currentTimeMillis();
125         for (int j1 = 0; j1 < (timeSims + 1); j1++) {
126           long singleInit = System.currentTimeMillis();
127           ThreadManagerNoFrameReset simulator =
128             new ThreadManagerNoFrameReset(
129               sims,
130               noFRestimators[k],
131               chartInfo,
132               tam_quadro_inicial,
133               xData,
134               dado.getN(),
```

```
135                 threaded
136               );
137             simulator.run();
138             long singleEnd = System.currentTimeMillis();
139
140             // Ignora a primeira simulacao para preparo das Threads
141             if (j1 > 0)
142               dado.addValor(singleEnd - singleInit);
143             System.out.println(
144               "\nSingle runtime: ["
145               + j1 + "] "
146               + (singleEnd - singleInit) + "ms"
147             );
148           }
149           long end = System.currentTimeMillis();
150           System.out.println(
151             "\nBatch runtime: " + (end - init) + "ms"
152           );
153         }
154         long allendtime = System.currentTimeMillis();
155         System.out.println(
156           "\nTotal runtime: " + (allendtime - alltime) + "ms"
157         );
158       }
159
160       // Exporta runtime data em arquivo
161       List<String> lines = new ArrayList<String>();
162       for (i = 0; i < dados.size(); i++) {
163         lines.add(
164           i + " \"" + dados.get(i).getN() + "\" "
165           + ((double) dados.get(i).getSum() / timeSims / 1000)
166         );
167       }
168       Path file = Paths.get(
169           "output/usadosNoRelatorio/RunTime/runtime-"
170           + timeSims + "-" + noFRestimators[0]
171           + "-p" + passo + ".txt"
172       );
173       try {
174         Files.write(file, lines, Charset.forName("UTF-8"));
175       } catch (IOException e) {
176         e.printStackTrace();
177       }
178     }
179 }
```

APÊNDICE B – CÓDIGO FONTE DOS ESTIMADORES

B.1 LOWER BOUND E SCHOUTE

```
1   case "LowerBound":
2   frameEnd = (collisions * 2);
3   break;
4
5   case "Schoute":
6   frameEnd = (collisions * 2.39);
7   break;
```

B.2 EOM-LEE

```
1 private double eom_lee(
2   double frameEnd, double collisions,
3   double success)
4 {
5   double beta, oldGama, num, den, frac;
6   double newGama = 2.0;
7   do {
8     oldGama = newGama;
9     beta = frameEnd / ((oldGama * collisions) + success);
10    frac = Math.exp(-(1.0 / beta));
11    num = (1.0 - frac);
12    den = (beta * (1.0 - (1.0 + (1.0 / beta)) * frac));
13    newGama = num / den;
14  } while (Math.abs(oldGama - newGama) >= 0.001);
15  return newGama * collisions;
16 }
```

B.3 VOGT

```
1   private double vogt(
2     double frameEnd, double collisions,
3     double empty, double success)
4   {
5     if (frameEnd == collisions) {
6       return chooseBestFrameSize(
7         frameEnd, Math.ceil(2 * collisions)
8       );
```

```
9     } else
10      return chooseBestFrameSize(
11        frameEnd, eq3(frameEnd, collisions, empty, success)
12      );
13  }
```

B.4 VOGT(EOM-LEE)

```
1   private double vogt_eom_lee(
2     double frameEnd, double collisions,
3     double empty, double success)
4   {
5     if (frameEnd == collisions) {
6       return 2 * collisions;
7     } else return eq3(frameEnd, collisions, empty, success);
8   }
```

B.5 ESCOLHA DO VALOR DELTA DO IMPROVED VOGT II

```
1   case "IV-2":
2   // first argument is the Delta value
3   int size = this.init_frame_size;
4   if (size == 64)
5     frameEnd = iv2(4, frameEnd, collisions, empty, success);
6   else if (size == 128)
7     frameEnd = iv2(2, frameEnd, collisions, empty, success);
8   break;
```

B.6 IMPROVED VOGT II

```
1 public double iv2(
2   int delta, double frameEnd, double collisions,
3    double empty, double success)
4 {
5   double result = 0;
6   if (frameEnd != collisions) {
7     // Resultado da eq. 3
8     return eq3(frameEnd, collisions, empty, success);
9   } else {
10    if (delta == 0) { result = 2.001001000 * (frameEnd - 1) + 2; }
11    else if (delta == 1) { result = 3.947947950 * (frameEnd - 1) + 2; }
12    else if (delta == 2) { result = 6.851851850 * (frameEnd - 1) + 2; }
13    else if (delta == 3) { result = 9.497497500 * (frameEnd - 1) + 2; }
14    else if (delta == 4) { result = 12.047047047 * (frameEnd - 1) + 2; }
15    else if (delta == 5) { result = 14.518518500 * (frameEnd - 1) + 2; }
16    else if (delta == 6) { result = 17.011011000 * (frameEnd - 1) + 2; }
17  }
```

```
18   return result;
19 }
```

B.7 FUNÇÃO EQ3()

```
1   private double eq3(
2     double frameEnd, double collisions,
3     double empty, double success)
4   {
5     double i = success + (2 * collisions);
6     double nEstimate = i;
7     double newEpsilon = -1;
8     double oldEpsilon = 0;
9     double a0 = 0, a1 = 0, a2 = 0;
10    double t;
11    while (newEpsilon < oldEpsilon) {
12      t = 1.0 - (1.0 / frameEnd);
13      a0 = Math.pow(t, nEstimate);
14      a1 = (nEstimate * a0) / (frameEnd * t);
15      a2 = 1.0 - (a1 + a0);
16      a0 = (a0 * frameEnd) - empty;
17      a1 = (a1 * frameEnd) - success;
18      a2 = (a2 * frameEnd) - collisions;
19      a0 *= a0; a1 *= a1; a2 *= a2;
20      oldEpsilon = newEpsilon;
21      newEpsilon = Math.sqrt(a0 + a1 + a2);
22      if (nEstimate == i) { oldEpsilon = newEpsilon + 1.0; }
23      nEstimate = nEstimate + 1.0;
24    }
25    return ((nEstimate - 1.0) - success);
26  }
```

B.8 FUNÇÃO CHOOSEBESTFRAMESIZE()

```
1   private double chooseBestFrameSize (double N, double n) {
2     while(n < low(N)) { N /= 2; }
3     while(n > high(N)) { N *= 2; }
4     return N;
5   }
6
7   private double low(double N) {
8     if (N == 16) return 1;
9     if (N == 32) return 10;
10    if (N == 64) return 17;
11    if (N == 128) return 51;
12    if (N == 256) return 112;
13    else return -1;
```

```
14   }
15
16   private double high(double N) {
17     if (N == 16) return 9;
18     if (N == 32) return 27;
19     if (N == 64) return 56;
20     if (N == 128) return 129;
21     if (N == 256) return 100000;
22     else return -1;
23   }
```