UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Detecção de defeitos de software utilizando agrupamento de perfis de desempenho

Vitor Silva Montes

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Vitor Silva Montes

Detecção de defeitos de software utilizando agrupamento de perfis de desempenho

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Paulo Sérgio Lopes de Souza
Coorientador: Prof. Dr. Alexandre Cláudio Botazzo Delbem

USP – São Carlos
Dezembro de 2019

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

Montes, Vitor Silva
M779d Detecção de defeitos de software utilizando agrupamento de perfis de desempenho / Vitor Silva Montes; orientador Paulo Sérgio Lopes de Souza; coorientador Alexandre Cláudio Botazzo Delbem. -- São Carlos, 2019.
174 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2019.

1. teste funcional. 2. avaliação de desempenho. 3. agrupamento de dados. 4. injeção de falhas. 5. confiabilidade de software. I. Souza, Paulo Sérgio Lopes de, orient. II. Delbem, Alexandre Cláudio Botazzo, coorient. III. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Vitor Silva Montes

Software fault detection using clustering of performance profiles

Dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Paulo Sérgio Lopes de Souza
Co-advisor: Prof. Dr. Alexandre Cláudio Botazzo Delbem

USP – São Carlos
December 2019

À Josiana, pela inspiração e companhia em todos os momentos, e aos meus pais Amauri e Horacina por semearem o gosto pelo estudo.

AGRADECIMENTOS

Agradeço aos professores Paulo Sérgio de Souza e Alexandre Delbem pelas ideias, atenção e apoio nos direcionamentos desse projeto, e à Embraer pela oportunidade de realizar esse estudo.

Agradeço à professora Rosana Vaccare Braga por ter me guiado e orientado ainda nos primeiros passos do projeto.

Agradeço aos amigos Vinicius, Eliane, Silvia, Guilherme, Rodrigo, José Teixeira e Lhais pelas sugestões e auxílios, assim como aos demais colegas do LaSDPC.

Finalmente, agradeço a minha esposa Josiana pela atenção e paciência nas infindáveis jornadas de estudos.

“Descobrir consiste em olhar para o que todo mundo está vendo e pensar uma coisa diferente.”

(Roger Von Oech)

RESUMO

MONTES, V. S. **Detecção de defeitos de software utilizando agrupamento de perfis de desempenho**. 2019. 174 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

A maioria dos problemas de desempenho são únicos. As métricas, cargas de trabalho e técnicas de avaliação usadas em um problema geralmente não podem ser usadas no problema seguinte. Portanto, ferramentas automáticas que auxiliem no entendimento do comportamento de uma aplicação em execução e suas mudanças ao longo do ciclo de desenvolvimento são essenciais para análises de desempenho e detecção de erros. A proposta deste trabalho é explorar a descoberta de defeitos no software por intermédio da avaliação de desempenho, assumindo a premissa que tais defeitos alteram o uso dos recursos ao longo da execução da aplicação. Isso é feito com a abordagem de teste funcional do programa em execução, onde são avaliados aspectos de desempenho da aplicação, e não aspectos funcionais, na detecção de erros na execução causados por defeitos. Um algoritmo de agrupamento baseado em Distância por Compressão Normalizada é aplicado para definir de forma automática um perfil de desempenho esperado do software em casos de teste, que é usado também para detectar anomalias. O processo de validação desta proposta é feito com a geração de defeitos por meio de mutação seletiva.

Palavras-chave: teste funcional, avaliação de desempenho, agrupamento de dados, injeção de falhas, confiabilidade de software.

ABSTRACT

MONTES, V. S. **Software fault detection using clustering of performance profiles**. 2019. 174 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Most performance issues are unique. The metrics, workloads, and rating techniques used in a problem generally can not be used in the next problem. Therefore, automated tools that assist in understanding the behavior of a running application and its changes throughout the development cycle are essential for performance analysis and error detection. The proposal of this research is exploring the discovery of softwares faults, by means of performance evaluation, assuming the premisse that such faults change the use of resources, throught the applications execution. This is done with the functional test approach of the running program, where performance aspects of the application, rather than functional aspects, are evaluated in the detection of errors in execution caused by faults. A Normalized Compression Distance based clustering algorithm is applied to automatically define an expected performance profile of a software for test cases,which is also used for detecting anomalies. The validation process is done with the generation of faults through selective mutation.

Keywords: functional testing, performance evaluation, data clustering, fault injection, software reliability.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BIN	*Binary values*
BMP	Device Independent Bitmap
CSV	*Comma Separated Values*
CVT	*Current Value Table*
HDF5	Hierarchical Data Format
NCD	Normalized Compression Distance
PNG	Portable Network Graphics
SUT	System Under Test

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

1.1 Contexto

A maioria dos problemas de desempenho são únicos. Um problema de desempenho acontece quando alguma métrica de desempenho não atende ao requisito pré-definido. Este requisito pode ser um valor limite, uma faixa de valores ou mesmo uma condição específica, como um evento indesejado. Por relacionar requisitos com os respectivos aspectos de implementação do aplicativo em teste, as métricas, cargas de trabalho e técnicas de avaliação usadas em um problema geralmente não podem ser usadas no problema seguinte (JAIN, 1990). Portanto, ferramentas automáticas que auxiliem no entendimento do comportamento de uma aplicação em execução e suas mudanças ao longo do ciclo de desenvolvimento são essenciais para análises de desempenho e detecção de erros (CHERKASOVA *et al.*, 2008).

Em teste de software dizemos que uma aplicação está apresentando um perfil adequado de desempenho quando este atende aos seus requisitos não-funcionais previamente estabelecidos. Geralmente, quando uma aplicação não possui requisitos explícitos de desempenho, este provavelmente não será considerado nas análises de qualidade, por não haver uma referência em que se basear. E, não obstante, criar esses requisitos, ou seja, fazer previsões sobre o quanto ela deveria fazer uso dos recursos da máquina, e se os valores são coerentes, não é uma tarefa simples (ATTARIYAN; CHOW; FLINN, 2012).

A ausência destes também torna o processo de caracterização de carga mais difícil, pois podem não ser conhecidos os valores limites ou de referência para uso dos recursos nas diferentes entradas do domínio. O uso de recursos, para diferentes entradas, pode desde variar muito ou quase nada, ou ainda apresentar faixas de valores frequentes. Porém, assim como é comum não haver requisitos explícitos de desempenho, também é comum não ser feito um processo de caracterização de cargas (JAIN, 1990).

Quando uma aplicação não está apresentando um desempenho adequado, a causa pode

ser, dentro os diversos fatores possíveis, o resultado da presença de um defeito no código-fonte, que está levando este programa a estados inválidos ou a ser menos eficiente (JIN *et al.*, 2012). O método tradicional de detecção de anomalias se baseia em definir limites pré-estabelecidos de valores de métricas de desempenho, e definir alarmes para quando estes valores são alcançados; este tipo de abordagem é chamada abordagem reativa (CHERKASOVA *et al.*, 2008). Esta abordagem, além de depender de um entendimento prévio de como a aplicação usa os recursos e como seria seu comportamento normal, não é adequada para aplicações que sofrem constantes mudanças. Por outro lado, uma abordagem pró-ativa, adotada nesse projeto, é baseada na avaliação contínua do desempenho da aplicação, definindo e adaptando os valores limites de acordo com o comportamento atual da aplicação.

Muitos estudos são feitos com objetivo de associar uma anomalia de desempenho, ou seja, uma não conformidade a um requisito de desempenho, com a presença de um defeito no software. Porém, identificar essa anomalia e definir um perfil de desempenho esperado costumam ficar a cargo do testador (ATTARIYAN; CHOW; FLINN, 2012).

Problemas de desempenho aparecem com frequência também em ambiente de produção. Por exemplo, os desenvolvedores do *Mozilla* consertaram de 5 a 60 problemas de desempenho reportados todo mês, nos últimos 10 anos (JIN *et al.*, 2012). A prevalência desses problemas de desempenho são difíceis de evitar devido a existirem poucos estudos e ferramentas que auxiliem encontrar esses problemas.

Diferente dos problemas funcionais, problemas que afetam o desempenho não são identificados e reportados com frequência pelos usuários, por geralmente não causar falhas graves no programa em execução (JIN *et al.*, 2012). Além de não causarem erros fatais, seus efeitos sobre os resultados obtidos podem ser difíceis de detectar. Um outro caso ainda é quando os resultados não foram alterados, porém o uso de recursos da máquina foi muito maior do que poderia (ou deveria) ser. Essa mudança no uso de recursos pode ser efeito de uma alteração de código, que consertou um defeito existente porém inseriu um defeito de desempenho (CHERKASOVA *et al.*, 2008).

Quando falamos de teste-caixa preta, normalmente nos referimos a testes funcionais. Porém, além dos requisitos funcionais, existem também os requisitos não-funcionais da aplicação, e é possível realizar atividades de teste caixa-preta visando avaliar estas propriedades não-funcionais, como por exemplo o desempenho. Assim como é possível avaliar a saída esperada de um programa, para diferentes entradas, é também possível considerar as medições de uso de recursos como uma saída do programa, e mesmo não sendo de interesse do usuário e nem estar presente nos requisitos, estas podem ser usadas como saídas de teste caixa-preta. Esses valores serviriam nesse caso para avaliar a adequação do programa assim como qualquer outra funcionalidade.

Mesmo que entradas exatamente iguais sejam inseridas em uma mesma aplicação, em duas execuções seguidas, suas medidas de desempenho podem não ser iguais; e isso é normal.

Isso acontece devido a outros processos rodando na mesma máquina, condições de memória disponível, paginação, cache, entre outros fatores do sistema operacional. Portanto, o processo de caracterização de carga e de teste de desempenho deve considerar essas variações e ser capaz de comparar os valores. Qualquer que seja a técnica utilizada para comparar as medições, essa deve ser flexível e ser capaz de trabalhar com valores imprecisos.

O processo de publicar uma nova aplicação em ambiente de produção pode envolver diversas etapas de teste, que podem ser testes caixa-preta, estruturais, de desempenho entre outras técnicas existentes. Esse processo de validação pode ser longo e custoso. Uma vez publicado, o programa em produção vira uma caixa-preta, e seu funcionamento tomado como correto. É de interesse que possamos verificar, de forma simples e recorrente, se ele (o software) está se comportando como esperado (CHERKASOVA *et al.*, 2008). Mesmo na hipótese do software estar correto, aplicações costumam sofrer constantes alterações, sejam correções ou adições de funcionalidades, ou mesmo adequações a mudanças de ambientes e protocolos, e ficaria caro, se, para cada alteração, todo o processo de verificação e validação fosse executado novamente.

Essa aplicação em produção pode estar sujeita a constantes alterações, e seria proveitoso, portanto, haver formas de verificar se o software está correto ou, em outros termos, como está a *saúde do software* em ambiente de produção, sujeito a cargas reais de trabalho, sem que isso seja uma atividade que demande grandes esforços da equipe e de recursos extras. (SRIVASTAVA; SCHUMANN, 2013) Este estudo de monitoração de saúde de software (do inglês *Software Health Management*), é um campo de estudo atualmente mais voltado para sistemas críticos e tolerantes a falhas, mas isso não impede que possamos expandir esse conceito para programas cuja comportamento em execução seja de interesse avaliar.

1.2 Proposta

A proposta deste trabalho é explorar o impacto de defeitos de software no desempenho, alterando o uso de recursos ao longo da execução da aplicação. Isso é feito com a abordagem de teste funcional do programa em execução, onde são avaliados aspectos de desempenho da aplicação, na detecção de erros na execução causados por defeitos. Estes defeitos, as vezes difíceis de identificar por vias tradicionais, podem impactar de forma visível no uso de um ou mais recursos, e este pode ser um caminho para verificar se uma aplicação está executando adequadamente.

O enfoque da pesquisa está na aplicação do Teste de Desempenho em aplicações que não apresentam requisitos de desempenho e/ou valores pré-definidos do uso de recursos, sejam estes absolutos ou relativos, e que também não apresentem cargas de trabalho de referência, provenientes de um processo de caracterização de cargas.

A metodologia visa ser aplicada em programas que:

- não possuem requisitos de desempenho;
- o uso de recursos para diferentes entradas nunca foi avaliado, é ignorado;
- a aplicação já se encontra em ambiente de produção, sujeita a condições reais de carga de trabalho; e
- há uma quantidade limitada de entradas possíveis, ou essas podem ser separadas em categorias.

Desta forma é possível automatizar o processo de caracterização de cargas, em ambiente de produção, com uso de entradas reais. Esse resultado da caracterização automatizada é usado como referência na criação de um perfil de uso de recursos. Esse perfil é usado, de forma automática, na identificação de alteraões de desempenho que sejam significativas. Essas alterações podem ser provenientes da presença de um defeito no código, inserido após a criação desse perfil.

Para lidar com esses valores difusos, um algoritmo de agrupamento baseado em *Normalized Compression Distance* (NCD) (CEBRIÁN *et al.*, 2005) é aplicado nas medições feitas da aplicação em execução, seja ela original ou alterada. Esse algoritmo é capaz de agrupar por similaridade e isso deixa o sistema robusto para as possíveis variações apresentadas, sem necessidade de parametrização. Ao agrupar os valores parecidos, são formados grupos com as diferentes cargas de trabalho, de forma automática e incremental.

Desta forma é possível determinar um perfil de desempenho esperado de um software, sujeito a cargas de trabalho usuais. Através de sua medição de desempenho espera-se identificar a presença de um ou mais defeitos. Com o uso de ferramentas de agrupamento automático, através de aprendizado não-supervisionado, propõe-se desenvolver uma ferramenta genérica e adaptativa, que sirva como um "termômetro"para aplicações executando em ambiente de produção, alertando para usos anormais de recursos, com vistas à revelação de possíveis defeitos.

1.3 Objetivos

O objetivo geral desta pesquisa é a identificação de perfis de uso de recursos de aplicações sujeitas a cargas reais de trabalho para facilitar a revelação de defeitos em programas, em ambiente de produção. Isto é feito com uso de ferramentas de software, para monitoração e agrupamento de resultados, e inserção de defeitos, pela geração de mutantes, para validação da proposta.

O objetivo geral (e principal) pode ser separado nos seguintes objetivos específicos:

- Estudar a técnica de teste funcional tendo como referência medições de desempenho.

- Definir categorias de defeitos comuns que causem impacto no desempenho e que podem não alterar a saída esperada.
- Identificar conjuntos de métricas de desempenho capazes de caracterizar um perfil de uso de recursos para cargas usuais.
- Estudar técnicas de agrupamento mais eficientes e eficazes para separar perfis de desempenho normais e anormais.
- Desenvolver uma ferramenta capaz de monitorar métricas de desempenho de uma aplicação e de identificar a execução de um mutante.
- Desenvolver uma ferramenta automática, adaptativa e não-paramétrica capaz de identificar um perfil normal de desempenho de um programa, alertando para possíveis anomalias no uso de recursos por parte deste programa, de modo a indicar possíveis defeitos no mesmo.

1.4 Contribuições esperadas

A técnica de teste funcional é muito útil quando não queremos considerar aspectos internos da aplicação a ser testada, e sim apenas seus aspectos funcionais. Apesar desta técnica também poder ser aplicada para aspectos não-funcionais da aplicação, nem sempre é fácil definir intervalos de valores esperados para métricas de desempenho. Este estudo visa facilitar a aplicação de teste-caixa preta voltado para aspectos de desempenho de uma aplicação, com enfoque em problemas causados por defeitos, tornando mais fácil definir um perfil esperado de desempenho para esta aplicação.

A técnica de teste de mutação tradicional é muito eficiente para ajudar a definir um conjunto de casos de teste, avaliar sua abrangência, e estudar efeitos da inserção de defeitos no resultado esperado. A mutação seletiva é aplicada para testar casos particulares, buscando efeitos específicos no código, e quando queremos evitar o trabalho de lidar com grandes quantidades de mutantes. Mutantes equivalentes também são um problema, e pra isso deve-se optar por um conjunto mínimo de operadores que satisfaça as condições desejadas.

Porém, muito pouco foi estudado no tema de teste de mutação quando queremos avaliar o programa tomando como saída suas métricas de desempenho. Em muitos casos, como neste trabalho, desejamos inserir falhas de desempenho em uma aplicação, que representem um erro causado por um defeito no código. Esse trabalho contribui ao propor estender a técnica de teste de mutação tomando como referência, na avaliação dos mutantes, sua adequação quanto a métricas de desempenho.

Outra contribuição está na utilização de algoritmos de agrupamento baseados em NCD para reconhecer padrões de comportamento de programas em execução através de suas medições periódicas de métricas de desempenho, em diferentes cargas de trabalho. Essas medições são

feitas em ambiente de produção, de forma não-invasiva, extraídas de execuções sucessivas de um programa considerado caixa-preta, sujeito a condições reais de carga de trabalho. Através de uma ferramenta de agrupamento não-paramétrica pode-se automaticamente separar anomalias de medições normais, sem necessidade de ter conhecimento sobre como a aplicação foi implementada.

1.5 Estrutura do texto

Após essa parte introdutória, o *Capítulo* 2 aborda os conceitos relacionados a *Teste de Software*, apresentando as principais técnicas de teste e também relaciona os conceitos com o foco deste estudo.

No *Capítulo* 3 são apresentados os conceitos básicos de *Avaliação de Desempenho*, com maior enfoque na técnica de monitoração, e por que estes conceitos são importantes para o desenvolvimento deste projeto.

O *Capítulo* 4 introduz alguns conceitos relativos a *Aprendizado de Máquina*, com maior enfoque em agrupamento de amostras, em especial abordando a técnica *Distância de Compressão Normalizada* e a ferramenta **DAMICORE**. Também é explicada a sua importância para o estudo proposto.

No *Capítulo* 5 é apresentada a *Metodologia Tricorder*, explicando como os conceitos se unem na execução de um projeto de monitoramento de recursos com enfoque em teste de aplicações caixa-preta em ambiente de produção. São descritos objetivos e critérios utilizados.

No *Capítulo* 6 é descrita a aplicação de *benchmark* no modelo cliente-servidor de processamento de dados, desenvolvida para servir de objeto de testes na validação da proposta. Ela faz uso de uma biblioteca de processamento, também desenvolvida para esse fim, onde são inseridos os defeitos estudados. Em seguida são descritas as categorias de defeitos selecionadas para avaliar a metodologia, bem como suas implementações. Por fim é apresentado o *framework* de teste criado para aplicar a metodologia **Tricorder** no teste da aplicação *benchmark*. São apresentados os passos de criação dos dados de teste e da verificação de presença de anomalias. O código-fonte da biblioteca, bem como outras definições auxiliares se encontram nos *Apêndices*.

No *Capítulo* 7 são apresentados os resultados do uso da metodologia **Tricorder** no teste de desempenho da aplicação *benchmark*. São mostrados gráficos das diferentes medições de desempenho, bem como tabelas com resultados dos testes.

Por fim, no *Capitulo* 8 são abordadas as nossas conclusões, propostas de melhorias e trabalhos futuros.

CAPÍTULO

2

TESTE DE SOFTWARE

Teste de Software é a área da Engenharia de Software cujo foco está na qualidade do produto de *software* ou sistema computacional sendo testado. Este pode estar pronto ou em fase de desenvolvimento. Existem diversos técnicas de teste, que visam cobrir diferentes aspectos, com objetivo de identificar possíveis defeitos no *software* objeto de teste. Decidir se um produto de *software* está ausente de defeitos é um problema indecidível (DELAMARO; MALDONADO, 2016). Por isso, é comum que mais de uma técnica seja aplicada, de forma complementar. Sempre novos métodos e critérios são criados, visando determinar a corretude de um programa.

Neste capítulo são abordados conceitos gerais de diferentes técnicas de teste, e como estas estão relacionadas a esse projeto.

2.1 Conceitos gerais

2.1.1 Introdução

O processo de desenvolvimento de software, assim como em outras áreas de desenvolvimento tecnológico, está sujeito a falhas. Desde o início do projeto, na sua concepção, até o produto final, diversos fatores podem influenciar para que o produto final seja diferente do que era esperado. Essas diferenças podem ser desde definição ou interpretação equivocada de requisitos, até a implementação errada destes.

As atividades de "Validação, Verificação e Teste" têm a finalidade de garantir que tanto o modo pelo qual o software está sendo construído quanto o produto em si estejam em conformidade com o esperado. Para evitar que erros, em grande parte por falha humana, cheguem até o produto final, diversas atividades e técnicas são aplicadas ao longo de todo o ciclo de desenvolvimento. Atividades chamadas estáticas não requerem a execução ou existência de um modelo ou programa executável para serem executadas, enquanto que as atividades dinâmicas, foco deste trabalho, se baseiam na execução de um programa ou modelo executável.

(DELAMARO; MALDONADO, 2016)

Pela terminologia de teste de software, "defeito" (do inglês *fault*) é um passo, processo ou definição de dados incorretos, ocasionado pelo "engano"(*mistake*) que o gerou. Por não dependerem de uma execução, são conceitos estáticos.

O estado de execução de um programa em determinado instante é dado pelo valor da memória e do contador de instruções. Se houver um defeito no código, sua execução causará um erro, e este estado inconsistente de execução pode ocasionar uma falha, quando o resultado do programa é diferente do esperado.

O conjunto de todos os possíveis valores de entrada de um programa é chamado de seu domínio de entrada. O mesmo conceito se estende a todas as possíveis saídas, como "domínio de saída". Um "dado de teste" é um elemento do domínio de entrada, enquanto que um "caso de teste" é um par formado pelo dado de teste e o resultado esperado. Um ou mais casos de teste para um determinado programa, chamamos de "conjunto de teste" do programa. (DELAMARO; MALDONADO, 2016)

2.1.2 Técnicas e critérios

Há tradicionalmente 3 fases na atividade de teste de software. A primeira, teste de unidade, tem foco nas menores unidades do programa, em geral funções ou conjunto de funções. A segunda, teste de integração, da ênfase na interação e interface entre as unidades do programa. Para ambas as etapas é necessário grande conhecimento das estruturas internas do programa. A última, teste de sistema, foco deste trabalho, tem objetivo de verificar se as funcionalidades especificadas nos documentos de requisitos estão corretamente implementadas. São considerados aspectos desde correção e completude até os chamados requisitos não-funcionais como segurança e desempenho.

Com relação aos tipos de teste, há basicamente duas classes, teste caixa-preta e teste caixa-branca. O teste caixa-preta, também chamado de teste funcional, ignora os mecanismos internos de um sistema ou componente e se baseia apenas nas saídas geradas pelo programa, sujeito a determinadas entradas e condições de ambiente. O teste caixa-branca, também chamado de teste estrutural, leva em conta os mecanismos internos e a estrutura do programa. Daí vem a origem dos termos, pois no teste caixa-preta, o testador, seja ele humano ou um programa, não tem (ou não deveria ter) acesso ao código-fonte do programa, nem tampouco às outras definições estruturais deste. O programa é considerado uma caixa-preta, e apenas as saídas fornecidas pelo programa (ou outras características visíveis externamente) são levadas em consideração. Baseado nos requisitos do programa é possível estimulá-lo com conjuntos de entradas e avaliar a saída de acordo com o que era esperado.

Por outro lado, o teste caixa-branca tem foco no código-fonte do programa, baseado em sua estrutura interna. O testador pode estimular partes do programa e avaliar, por exemplo,

a cobertura alcançada pelos conjuntos de teste utilizados. Normalmente, teste de unidade e de integração são testes caixa-branca pois requerem entendimento do funcionamento interno do programa para serem aplicados. Testes de sistema por outro lado, podem ser funcionais ou estruturais. Utilizando uma outra terminologia, é comum associar o teste caixa-preta à atividade de validação, que consiste em saber se o programa correto está sendo feito, de acordo com seus requisitos, enquanto que os testes caixa-branca são associados às atividades de verificação, que dizem se o programa está sendo feito corretamente. (SOMMERVILLE, 2011)

Um outro tipo de teste, que normalmente é considerado um teste caixa-branca, é o teste baseado em erros. A técnica de teste baseada em erros utiliza informações sobre os tipos de erros mais frequentes no processo de desenvolvimento de software para derivar os requisitos de teste. A ênfase da técnica está nos erros que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser ser usadas para detectar sua ocorrência. Podemos citar como exemplos a Semeadura de Erros e a Análise de Mutantes. (DELAMARO; MALDONADO, 2016)

2.1.3 Oráculo de teste

Em teste de software, o oráculo é responsável por identificar as discrepâncias entre o funcionamento de um programa e o que é esperado dele. Ele pode se basear em requisitos funcionais, saídas esperadas e demais comportamentos ou retornos esperados da aplicação. Por isso o oráculo é uma peça fundamental em diversas técnicas de teste pois serve como referência na detecção de não-conformidades.

Neste projeto não há necessidade de existência de oráculo previamente criado para a aplicação objeto de teste, pois este é criado de forma automática e incremental pelo agrupamento contínuo das medições de desempenho.

2.2 Teste Funcional

2.2.1 Introdução

No teste funcional o programa em teste é estimulado por suas diversas entradas possíveis, de acordo com as especificações do programa. Essas entradas representam o uso esperado do programa, e são avaliadas as respostas e saídas do sistema. Dessa forma é possível avaliar as funcionalidades presentes no sistema de acordo com os requisitos pré-definidos.

O teste funcional em teoria é capaz de detectar todos os defeitos se realizado um teste exaustivo de todas as entradas possíveis. O problema disso é que esse conjunto de entradas pode ser muito grande, tornando impraticável essa tarefa. Um outro problema não menos importante é domínio de saída, onde é necessário conhecer a saída esperada para todas essas possíveis entradas.

Para solucionar esse problema, um critério funcional muito usado é o particionamento de equivalência, onde selecionamos uma parte ou subconjuntos do domínio de entrada, de forma que seja uma boa representação de todo o domínio. Ainda assim é necessário ter a saída esperada para esse subconjunto.

Outra técnica que aumenta o poder do teste funcional é o critério Grafo Causa-Efeito, que é uma tradução em linguagem formal das especificações do software, e assim ajuda a encontrar ambiguidades e incompletudes nas especificações e exercita combinações de dados de teste que possivelmente não seriam consideradas.

2.2.2 Aplicação

O teste funcional tem grande poder de detectar defeitos quando conhecemos todos os requisitos funcionais e temos a saída esperada para ao menos uma entrada de cada classe de equivalência. Sua principal vantagem é precisar apenas das especificações do produto para criar casos de teste. Assim, torna-se indiferente a forma com que foram implementados, assim como linguagens, paradigmas e estrutura interna.

Esta característica é também sua maior desvantagem, pois como os critérios funcionais se baseiam apenas na especificação, não há como afirmar que as partes importantes do programa foram exercitadas e testadas.

As classes de equivalência também têm suas limitações práticas, pois se baseiam em especificações, e estas, se incorretas, podem levar a falsas interpretações do que seria considerado equivalente; em outras palavras, dois casos de teste de uma mesma classe de equivalência podem ter comportamentos distintos. Por isso dizemos que cada técnica de teste é complementar às outras.

2.2.3 Relação com o projeto

Assim como no Teste Funcional, este projeto se baseia na técnica de teste caixa-preta. São utilizadas propriedades de uso de recursos do programa em execução, como será explicado mais adiante, que independem da forma com que foi implementado o sistema, e sem necessidade de conhecer sua estrutura interna. São monitoradas propriedades de uso de recursos do sistema comuns a linguagens, onde o programa é visto no nível de um processo em execução no SO.

Contudo não podemos afirmar que seja um Teste Funcional, em total conformidade com a sua definição na literatura, pois os critérios de teste não se baseiam em especificações funcionais. Apenas a ideia de teste caixa-preta está sendo aplicada, porém com uma abordagem diferente, que utiliza como referência de adequação apenas métricas de desempenho. Nesse processo os erros são identificados pelo seu reflexo no desempenho, e não nas funcionalidades do programa. Não há necessidade todavia da presença de requisitos de desempenho, sendo que o perfil de desempenho esperado é definido automaticamente por meio de um algoritmo de agrupamento.

Neste processo de validação da proposta também são aplicados conceitos de Teste Baseado em Erros, como será descrito mais adiante.

2.3 Teste de Desempenho

2.3.1 Introdução

Teste de Desempenho é um segmento da Engenharia de Desempenho cujo objetivo é determinar como um sistema se comporta em relação à responsividade e à estabilidade, estando sujeito à determinada carga de trabalho. Esta atividade é essencial para evitar surpresas desagradáveis em ambiente de produção, como tempo de resposta muito lento, taxas de transferência inadequadas e transações interrompidas antecipadamente. (BONDI, 2014)

O teste de desempenho é uma atividade que faz parte do chamado teste não-funcional que, como o próprio nome diz, avalia características não funcionais do programa como: desempenho, carga, estresse, usabilidade, confiabilidade, portabilidade e segurança. Testes não-funcionais podem ser executados em qualquer fase de desenvolvimento, desde que haja algum módulo ou protótipo testável.

Umas das principais razões dessa atividade é garantir que os requisitos de desempenho e as expectativas do usuário em relação ao desempenho serão atendidas. Mesmo se o sistema tiver passado por todos os testes unitários e de integração, deve-se verificar se os requisitos de desempenho foram atendidos. Isso acontece porque problemas funcionais podem ser causados por erros de concorrência que não ocorrem em testes unitários. O teste de desempenho também pode ser aplicado para testar os limites de capacidade do sistema, para realizar medidas úteis no planejamento de capacidade e modelagens, e para identificar cenários de carga de trabalho ou sequência de eventos que podem causar uma falha no sistema. (BONDI, 2014)

2.3.2 Conceitos e Aplicações

O desempenho de um sistema deve ser descrito de forma genericamente entendida e sem ambiguidades. As medidas quantitativas de desempenho são chamadas métricas de desempenho. Para que essas métricas façam sentido na descrição de um sistema é também necessário que elas possam ser medidas de alguma forma. Abaixo são exemplos de métricas: (BONDI, 2014)

- Tempo de resposta: padrão de medição de quanto tempo o sistema leva para executar uma determinada atividade, desde que a mesma é submetida para execução até o momento que o sistema finaliza a atividade. Pode ser representado como uma média dos tempos de resposta de diferentes execuções do mesmo programa.
- Uso médio de recurso: é um padrão de medição da proporção de tempo que um recurso como CPU, disco rígido, etc, está ocupado.

- Taxa de transferência média: é a taxa com que o sistema envia ou recebe unidades de alguma atividade.

2.3.3 Relação com o projeto

A atividade de Teste de Desempenho, como mencionado anteriormente, visa encontrar defeitos de desempenho, que seriam não adequações de requisitos de desempenho. Isso difere da proposta do projeto, pois o objetivo seria testar programas que não têm requisitos de desempenho pré-definidos, e este perfil de desempenho esperado precisa ser construído. Portanto, os defeitos encontrados não seriam defeitos de desempenho necessariamente, e sim defeitos no código que refletem no desempenho.

Neste projeto poderão ser utilizados conceitos, métricas e até mesmo ferramentas de Teste de Desempenho, pois há uma grande intersecção entre as atividades, apesar dos objetivos não serem exatamente os mesmos. Neste estudo não buscamos identificar os limitantes de capacidade do sistema em execução, nem tampouco determinar as cargas de trabalho que levam às maiores quedas de desempenho, como seria próprio de um teste de desempenho. Este estudo visa automatizar a caracterização de um perfil de desempenho considerado normal, sob condições reais de carga, e auxiliar no acompanhamento de adequação do programa ao longo de frequentes modificações. Definir os limites de desempenho que o sistema suporta ficaria a cargo de um outro estudo, fora do escopo deste.

2.4 Teste de Mutação

2.4.1 Introdução e conceitos

O teste de mutação é um critério de teste baseado em erros que tem se mostrado muito eficaz como critério de detecção de erros e falhas, e pode ser utilizado tanto em programas como em modelos. Neste critério, para avaliar a eficácia de um conjunto de testes T em avaliar um programa P, o teste de mutação cria e aplica um conjunto de implementações alternativas de P, chamados mutantes de P, os quais são criados através de regras chamadas critérios de mutação. Essas regras definem quais defeitos serão inseridos no código.

O objetivo do teste de mutação é a geração de um conjunto de testes capaz de distinguir o comportamento dos mutantes do programa original. A relação entre o número de mutantes identificados (chamados de mutantes mortos) com o total de mutantes, serve como medida de adequação do conjunto de teste. (MALDONADO *et al.*, 2001)

A geração normalmente é feita por meio de ferramentas de software que interpretam o código-fonte da aplicação e identificam os operadores ou trechos a serem alterados de acordo com a regra de criação desejada. Quais regras utilizar depende do teste a ser executado, e dos tipos de defeito que desejamos avaliar.

De acordo com a hipótese do Programador Competente, DeMillo, Lipton e Sayward (1978), partimos de um programa considerado sem defeitos, e nele são inseridos pequenos desvios sintáticos, que mesmo que não causem um erro sintático (como um erro de compilação) podem alterar a semântica do programa, levando esse a comportamentos incorretos e inesperados. De acordo também com a hipótese efeito de acoplamento, conjuntos de casos de testes capazes de revelar erros simples também são capazes de revelar erros complexos.(DELAMARO; MALDONADO, 2016)

Nesta atividade de teste são gerados, a partir do programa original, um conjunto de mutantes, cada um podendo ter de *1* a *k* mutações, ou seja, *k-defeitos*. A forma mais comumente utilizada é a de *1* defeito por mutante, e para cada mutante aplica-se o conjunto de casos de teste para validar sua abrangência. Cada mutante que retornar um valor incorreto é considerado "morto". A relação entre o número de mutantes mortos e o número total de mutantes gerados é chamado de escore de mutação, e nos dá uma medida quantitativa da confiança de adequação dos conjunto de casos de teste aplicados. Pode-se assim ir adicionando casos de teste até que o escore resultante seja suficiente.

Os mutantes são gerados a partir dos operadores de mutação. Entende-se por operador de mutação as regras que definem as alterações que devem ser aplicadas no programa original P . Os operadores de mutação são construídos para satisfazer a um entre dois propósitos: 1) induzir mudanças sintáticas simples com base nos erros típicos cometidos pelos programadores (como trocar o nome de uma variável); ou 2) forçar determinados objetivos de teste (como executar cada arco do programa). (DELAMARO; MALDONADO, 2016)

2.4.2 Relação com o projeto

Sendo o principal objetivo do projeto a detecção de anomalias de desempenho causadas por defeitos, a melhor forma de avaliar sua eficácia é com a geração de defeitos, nesse caso através de mutantes. A utilização de mutantes permite que sejam feitos testes isolados para classes de defeitos, e avaliar a eficácia para cada um.

Porém é importante ressaltar que este projeto não é um teste de mutantes em sua proposta final, e sim um teste funcional que utiliza métricas de desempenho em sua avaliação e na busca por defeitos. Neste estudo, os mutantes são utilizados apenas no processo de validação da ferramenta, como forma controlada de inserir falhas que afetam o desempenho. A execução dos mutantes servirá como base para avaliar se a ferramenta é capaz de identificar a presença de defeitos.

A regra de criação de mutantes está fortemente relacionada a quais defeitos desejamos avaliar. Pode-se dizer de antemão que seriam aqueles que tenham alguma influência no uso de recursos. Por exemplo mutação de comandos de alocação de memória e variáveis de parada em laços são exemplos de alterações que podem afetar o desempenho. Mutações relacionadas

à concorrência como alterações em variáveis de condição e outros comandos de sincronia de *threads* também podem causar impacto no desempenho, e portanto seriam de interesse no projeto. A definição das regras de geração desses mutantes mais adequada é um dos desafios desse projeto.

Em um trabalho relacionado, Nanavati *et al.* (2015) fizeram um estudo sobre teste de mutação com enfoque em falhas no uso de memória. Neste trabalho também foram estudados operadores de mutação que fossem adequados a este fim específico, introduzindo alguns novos além dos tradicionais. A proposta se baseia na hipótese de que muitas falhas de memória não são cobertas pelo método de teste de mutação tradicional, e para esse fim foram propostos, além dos operadores de mutação específicos, outras formas de avaliação, ou seja, outros critérios de avaliação dos mutantes que não se baseiam apenas na saída do programa. A relação do trabalho de Nanavati *et al.* (2015) com o projeto está na seleção de operadores de mutação que reflitam em alterações no uso de recursos, como no caso alterações na alocação e desalocação de memória. Pode-se dizer, portanto, que o trabalho deles assim como este tem como objetivo a inserção de falhas de desempenho.

O trabalho de Nanavati *et al.* (2015), no entanto, difere da proposta dessa monografia por ser um teste caixa-branca, e além de considerar entradas e saídas do programa, também considera o fluxo de execução e outras características estruturais na análise. Outra diferença da proposta dessa monografia em relação ao trabalho mencionado, assim como da maioria dos trabalhos de teste de mutação, é a hipótese de considerar na avaliação dos mutantes apenas seu reflexo no uso de recursos, e não avaliar suas saídas esperadas. Espera-se dessa forma que o projeto seja capaz de identificar a presença dessas categorias de defeitos sem que seja informada a saída esperada do programa para aquele conjunto de entradas. (JIA; HARMAN, 2011)

Uma outra mudança com relação ao método de teste de mutantes tradicional está nos critérios de eliminação de mutantes, que seriam baseados em métricas de desempenho e não (ou não apenas) na saída esperada do programa. Saber qual é a saída esperada de uma aplicação em produção sujeita a cargas de trabalho reais pode ser difícil e custoso, por outro lado modelar o seu uso de recursos esperado pode ser um processo mais prático. Essas medidas podem ser usadas como um indicativo de bom funcionamento da aplicação, e podem também servir como métrica de eliminação de um teste baseado em mutantes.

2.4.3 Considerações finais

Testar um programa de computador durante e após seu desenvolvimento é uma atividade fundamental para averiguar se o produto final está de acordo com seus requisitos. Na maioria das técnicas, esta averiguação é baseada na escolha correta dos casos de teste, seja pela cobertura alcançada de suas estruturas internas (teste estrutural), seja pela abrangência de requisitos funcionais e não-funcionais alcançada (testes funcionais e não-funcionais). Mesmo quando o desempenho é foco do teste, este é analisado do ponto de vista da escolha de dados de teste.

Este projeto porém procura ver o teste de software de um ponto de vista diferente, com uma abordagem que associa a presença de defeitos com alterações no desempenho do programa em execução, sem que existam pré-requisitos de desempenho para o mesmo. A análise do desempenho como uma propriedade do programa, sujeito ao seu ambiente real de uso, havendo ou não requisitos para tal, é parte do estudo de Avaliação de Desempenho, tema do Capítulo 3.

CAPÍTULO

3

AVALIAÇÃO DE DESEMPENHO

Em Avaliação de Desempenho são estudadas técnicas, critérios e ferramentas para medição de diferentes partes do sistema, como o *hardware*, sistema operacional, periféricos e aplicativos. Algumas técnicas de monitoração de uso de recursos, assim como outras usadas em Teste de Desempenho, são também parte do estudo de Avaliação de Desempenho. Neste capítulo são abordados alguns conceitos de avaliação de desempenho de sistemas computacionais, e sua relação com esse projeto.

3.1 Conceitos gerais

Avaliação de desempenho de uma solução computacional possui fundamentalmente três etapas: modelagem, simulação e medição (JAIN, 1990). A escolha da técnica a ser usada é influenciada decisivamente pela fase em que um projeto se encontra. Se o projeto ainda está em fase de concepção, apenas simulação e modelagem são possíveis, pois para medir é necessário ter um programa executável. Outro fator importante é o tempo disponível para essa análise. Enquanto simulação costuma demandar muito mais tempo que modelagem, a medição deveria ser a mais rápida. Porém muito estudos mostram que uma aplicação prática de medição está sujeita a muitos imprevistos e seu tempo de término é muito variável de um projeto para outro. (JAIN, 1990)

Por último é necessário analisar a disponibilidade de pessoas, ferramentas e o nível de precisão desejada na avaliação. Por exemplo, modelos e simulação oferecem uma visão mais simplificada em muitos aspectos; enquanto que a medição traz informações mais próximas do real. Todavia, deve-se também considerar que os testes de medição estão sujeitos a condições de carga de trabalho, configurações específicas do sistema em teste, e outras escolhas de contexto, sendo cada experimento específico não representando necessariamente as condições reais de aplicação, pois é resultado de um conjunto selecionado de casos de teste, que podem não representar todas as possibilidades encontradas no uso real. Há ainda a influência do ambiente

de produção, mudanças de configuração e de eventos externos como latência ou problemas de *hardware*. Uma representação não fidedigna das condições reais implica em conclusões enganosas.

Avaliação de desempenho por modelagem é uma técnica utilizada para averiguar o comportamento e a adequação do desempenho de um algoritmo ou arquitetura proposta, e pode ser feito anteriormente ao início da implementação. A modelagem de software é a atividade de construir modelos que expliquem as características ou o comportamento de um software ou de um sistema de software. Na construção do software os modelos podem ser usados na identificação das características e funcionalidades que o software deverá prover (análise de requisitos), e no planejamento de sua construção. Essa modelagem pode se basear em modelos analíticos ou simulados. Modelos analíticos são abstrações do comportamento esperado do sistema, representados por modelos probabilísticos, filas, redes de Petri ou redes de Markov. Essa técnica tem as vantagens de representar o sistema de um ponto de vista genérico e conciso, e de poder usufruir de ferramentas matemáticas de análise de restrições e busca de otimizações.

Avaliação de desempenho por simulação seria descrever o sistema baseado em simulação de eventos e condições, como uma simulação do ambiente real, e das etapas do processo a ser executado. Utilizam-se simuladores para o ambiente final, para partes do sistema a ser implementado, e até mesmo para todo o conjunto ao mesmo tempo. Essa técnica tem as vantagens de evitar a necessidade de dispor do ambiente real de teste, que pode ser difícil ou custoso de se obter, e de se poder simular partes das funcionalidades de forma independente, validando assim módulos separados quanto à sua interação com o sistema. É uma técnica muita utilizada, especialmente em sistemas que envolvem hardware; porém seus resultados podem não ser totalmente realistas.

Medição é a última opção em termos de custos, quando simulação e modelagem são opções aplicáveis. A medição exige experimentos e condições próximas do real para efetuar suas coletas de dados, e exige uma instrumentação adequada, enquanto que outras técnicas podem ser aplicadas sem uso desses recursos. Ela é muito utilizada exatamente porque expõe características intrínsecas ao projeto, vinculando-o a seu uso real, em ambientes de produção ou equivalentes. A medição acaba sendo em alguns casos uma validação das anteriores, onde mais de uma técnica é aplicada simultaneamente.

O enfoque geral deste projeto está na análise das informações sobre o uso de recursos de processos em execução. Essa atividade está no centro do estudo de avaliação de desempenho. Portanto, métricas, critérios e técnicas de avaliação e análise de desempenho devem ser estudadas para que as informações coletadas dos processos sejam representativas. Da mesma forma, problemas relacionados à atividade de avaliação também podem estar presentes no projeto, como é o caso do tempo de amostragem adequado.

3.2 Monitoração de desempenho

3.2.1 Introdução

A melhor forma de aprender sobre um assunto é aplicá-lo no mundo real. Isso é especialmente verdade no caso da avaliação de desempenho de sistemas computacionais porque embora algumas técnicas possam parecer simples superficialmente, sua aplicação no sistema real será uma experiência diferente, pois o comportamento de sistemas reais não costuma ser simples. (JAIN, 1990)

A principal vantagem de monitoração de desempenho em relação ao uso de modelos está no fato, é claro, de os valores obtidos na monitoração serem referentes ao sistema real e não um modelo. Podem estar presentes nas amostras o resultado de interações e comportamentos que não foram ou não puderam ser reproduzidos no modelo. E mesmo se o modelo for detalhado ao ponto destes comportamentos serem observados, este modelo será bastante custoso de ser executado. Quanto às desvantagens, podemos citar a necessidade de um programa executável, de uma instrumentação adequada, muitas vezes de um sistema dedicado, maiores tempos de configuração e execução, e a modificação do sistema real de modo que os problemas possam ser estudados.

Os tipos básicos de instrumentação de sistemas para avaliação de desempenho são medidores de hardware e de software. Um medidor de hardware é normalmente um instrumento externo eletronicamente acoplado ao sistema de modo a medir tensões, correntes e demais sinais elétricos que reflitam estados do sistema. Estes sinais são associados ao sinal de *clock* para que possamos medir um intervalo de tempo e executar uma amostragem precisa, ou no caso de eventos assíncronos, servir de estampa de tempo. Por estarem diretamente ligados ao hardware são muito precisos, apresentam grande responsividade e suportam altas taxas de amostragem, além poderem observar eventos não acessíveis via software. Têm também a vantagem de não usarem o processamento da máquina em teste para efetuar aquisições. Suas principais desvantagens são custo, complexidade de instalação e configuração, e pouca flexibilidade de uso.

Medidores de software são instruções adicionadas ao sistema avaliado para coletar dados de desempenho. Eles podem coletar dados por leitura de memória ou outros indicadores de estado. Um dispositivo de medição que utiliza medidores de software é chamado de um monitor de software. Como esses monitores também rodam no mesmo sistema sendo avaliado, eles também têm um custo de execução que deve ser considerado, o coloquialmente chamado "peso da régua".

Um monitor de software pode ser periódico, dirigido por eventos, ou as duas coisas. Estes eventos podem ser do sistema operacional ou do programa em execução, como interrupções, acessos à memória, operações de E/S, e a cada ocorrência dados são coletados. O modo dirigido por tempo, ou periódico, como o nome diz é uma aquisição por meio de amostragem feita em intervalos definidos de tempo, normalmente baseados na frequência da CPU e interrupções de

temporizadores. Ao mesmo tempo que o uso de amostragem periódica é menos custoso e menos invasivo, também pode levar a conclusões erradas dependendo do intervalo utilizado. Se este intervalo for muito grande, informações importantes como variações e picos podem ser perdidos nesses intervalos. A resolução do temporizador utilizado também deve ser compatível com a granularidade de eventos e com as variações que se deseja verificar.

3.2.2 Métricas de desempenho

Métricas de desempenho são a base da pesquisa experimental. Elas definem formas de realizar medições e como interpretar seus resultados. A proliferação de processadores *multicore* e também a propagação de programas *multithreaded* levou à necessidade de métricas compatíveis a esse contexto. Um mesmo processador pode executar mais de um programa, e cada um com uma ou mais *threads*. Isso leva a um problema fundamental da avaliação de desempenho relacionado à caracterização de carga, onde um processo em execução pode interferir no desempenho dos demais.

As métricas disponíveis variam de acordo com o sistema utilizado, e para cada tipo de avaliação feita, um conjunto singular de métricas pode ser adequado. Elas podem ser avaliadas por 3 aspectos: responsividade, produtividade e utilização. A utilização é a perspectiva em foco nesse trabalho, e nos dá uma medida absoluta ou percentual do uso dos recursos do sistema pelo programa, a cada momento. O recurso mais utilizado é chamado de gargalo (ou *bottleneck*).

Se um erro ocorre, este deve ser considerado na análise de desempenho, pois a não execução de um serviço ou o resultado inesperado ou incorreto deste, representa uma condição inválida de avaliação de desempenho. Por exemplo, se ocorre um erro e um laço sai antes do esperado, todo um processamento será encurtado e um resultado diferente, e inválido, de uma ou mais métricas será obtido. Esses casos considerados exceções no processo de avaliação de desempenho têm um papel fundamental na proposta desse trabalho.

3.2.3 Caracterização de carga

A carga de trabalho de um sistema é o conjunto de requisições ou demandas por recursos de um número de usuários do sistema. O usuário pode ou não ser uma pessoa, desde que seja cliente do sistema em teste ou System Under Test (SUT).

Os componentes da caracterização de carga podem ser aplicações, usuários, ou mesmo locais físicos no caso de sistemas distribuídos geograficamente. No escopo desse projeto, o componente é o processo sendo executado no sistema operacional, sendo portanto cliente dos recursos oferecidos por este último.

As medidas a serem coletadas, referentes a uma carga de trabalho em um componente, são chamadas de parâmetros ou atributos de carga, e podem ser tamanho de pacotes, destino, tipo, páginas de um documento, etc. É importante ressaltar que na caracterização de carga é

preferível se basear em parâmetros que independem do sistema, e sim apenas da entrada utilizada. Medições como tempo de resposta e uso de memória são dependentes do sistema e podem variar de uma execução para outra.

Os parâmetros de carga (ou demandas de recurso) podem ser variados, e um agrupamento dos resultados pode ser necessário para verificar o impacto de execução de um conjunto de cargas de trabalho.

Este trabalho visa simplificar a etapa de caracterização de carga com uso de uma ferramenta automática de agrupamento das medições de desempenho. O resultado desse agrupamento é o que chamamos de perfil de uso de recursos da aplicação, ou perfil de desempenho.

Uma vez separado em grupos, esse perfil de desempenho pode ser usado como referência na identificação de anomalias, no processo de teste de desempenho da aplicação. (CHERKASOVA *et al.*, 2008)

3.2.4 Outliers

Valores extremos de parâmetros de carga, que se destacam em relação aos demais para uma execução de teste, são chamados de *outliers*. Eles são fundamentais quando buscamos por gargalos, picos de uso e condições anormais de uso de recurso. Em muitos casos, todavia, estes *outliers* podem representar usos esporádicos e anormais do sistema, e devem ser excluídos da análise, uma vez que não representam o uso normal do sistema em teste.

Normalmente as etapas de caracterização de carga e identificação de *outliers* ocorre em conjunto, pois esses *outliers* ajudam na identificação das entradas as quais devemos submeter o sistema de modo a identificar seus gargalos ou os chamados erros de desempenho, normalmente associados a requisitos de desempenho.

Vale ressaltar que na proposta deste projeto não estão disponíveis para o sistema em teste quaisquer requisitos de desempenho, e nem tampouco uma caracterização de carga preliminar, com identificação de *outliers*. Estes *outilers*, porém, existem e serão medidos, e devemos portanto efetuar uma análise adaptativa e automática para separar *outliers* oriundos de carga de trabalho de *outliers* provenientes de comportamentos anormais do sistema. Portanto, para esta abordagem, a existência de *outliers* não representa anomalia de desempenho, nem implica a existência ou não de defeitos no código.

3.2.5 Relação com o projeto

Este projeto proposto difere de um projeto de análise de desempenho, pois seu objetivo não está na identificação de gargalos, picos de uso, ou quaisquer medições de desempenho que extrapolem limites definidos ou sugeridos. Neste caso, a análise de desempenho da aplicação em foco fica a cargo da equipe de desenvolvimento e como mencionado no Capítulo 2, partimos de uma aplicação cujo comportamento, em termos de requisitos funcionais e não-funcionais, é

considerado adequado. Dessa forma, a detecção de um pico de uso de recurso não é visto como um problema e sim como característica de funcionamento da aplicação em foco.

Outra diferença está na correlação entre carga de trabalho e uso de recursos. Este projeto não visa encontrar cargas de trabalho que gerem usos excessivos nem buscar pelas cargas de trabalho causadoras de anomalia. O foco do projeto está no uso de cargas de trabalho quaisquer consideradas válidas, tentando alcançar uma abrangência de possíveis níveis e tipos de carga.

O foco do projeto é encontrar variações no uso de recursos causadas por possíveis defeitos no código, na execução de cargas de trabalho usuais e frequentes, sem considerar, no entanto, que essas diferenças são obrigatoriamente *outliers*, ou que o sistema está sujeito a sua carga máxima. As variações de desempenho serão geradas por inserções de defeitos no programa em teste, e não pela variação significativa na carga de trabalho.

3.3 Trabalhos Relacionados

Diversos trabalhos têm foco no desenvolvimento de técnicas para encontrar a causa-raiz de problemas de desempenho. Para isso, diversas técnicas mais ou menos invasivas são aplicadas, até mesmo algumas *caixa-preta*. Assim como propõe este projeto, algumas dessas técnicas não necessitam do código-fonte da aplicação, e são construídas visando o uso em ambientes de produção. Para isso, o arquivo binário da aplicação é instrumentado e são coletadas informações referentes ao uso de recursos ao longo da execução desse binário, e assim é possível estimar o trecho responsável por gargalos ou *outliers* de uso de recursos. Attariyan et al., (ATTARIYAN; CHOW; FLINN, 2012), propõe uma ferramenta chamada *X-Ray* capaz de atribuir custos de desempenho para trechos de código binário automaticamente, e executar uma operação chamada sumarização de desempenho. Esta técnica é aplicada para identificar trechos de código compilado que são responsáveis por anomalias de desempenho, e também para identificar variações destes.

Apesar de altamente eficaz na localização, com relativa precisão, dos pontos responsáveis pelas anomalias de desempenho, essa técnica, assim como muitas outras, tem o problema da necessidade de haver uma definição pré-existente daquilo que chamamos de anomalia de desempenho. Essa identificação pode ser baseada numa comparação de execução anterior, o que não é confiável, visto que nem todos os usos podem ter sido estimulados, ou o que acontece na maioria dos casos, cabe a um especialista analisar os resultados e identificar aquilo que ele considera uma anomalia. O especialista pode se basear em requisitos não-funcionais explícitos ou apenas no comportamento que ele espera da aplicação, baseado no seu conhecimento sobre como ela deveria se comportar.

Esse problema de identificação de anomalias de desempenho é o assunto abordado por (CHERKASOVA *et al.*, 2008). Nesse artigo é levantada a questão a respeito de anomalias de desempenho comparadas a variações de carga de trabalho, e da dificuldade real de se identificar o que seria uma anomalia de desempenho em um sistema sendo executado em condições reais

de ambiente e formas de uso. Um pico de uso de recursos não é necessariamente uma anomalia de desempenho, embora seja um provável *outlier* de avaliação de desempenho.

Um mapeamento completo de perfis de uso da aplicação associados ao uso de recursos é inviável para a maioria das aplicações comerciais, pois o domínio de entrada costuma ser enorme, e a aplicação costuma passar por alterações frequentes, podendo invalidar os resultados dos testes anteriores. Ainda há o problema, mencionado anteriormente, onde a união de diferentes entradas simultâneas pode gerar resultados não lineares quanto ao uso dos recursos. Levantar um perfil de uso de recurso considerado normal para uma aplicação em produção necessita de uma técnica mais eficiente e barata do que um mapeamento completo do domínio de entrada e armazenar os usos de recurso apresentados (CHERKASOVA *et al.*, 2008).

Em um trabalho de Ningfang (MI *et al.*, 2008) é apresentado um conceito chamado assinatura de aplicação que permite comparação de desempenho entre diferentes versões de um mesmo software através de suas assinaturas. Elas são baseadas nos conceitos de perfis de tempo de transação e assinaturas de transação. O objetivo dessas assinaturas é se manterem consistentes em diferentes cargas de trabalho, permitindo uma caracterização da aplicação que independe de carga, e assim comparar o funcionamento entre versões de um mesmo programa. Apesar de ter uma proposta semelhante a este projeto, a solução é específica para avaliar servidores de aplicação, com a presença de transações bem definidas, que não estão presentes nesse projeto. Outra diferença não menos importante é que a solução deles se baseia em sistemas J2EE, enquanto este projeto é voltado para avaliação de programas feitos em códigos nativos, com controle interno do uso de recursos. A presença de coletores de lixo de memória (*garbage collector*), por exemplo, adicionaria não-determinismos à aplicação, fugindo do escopo deste estudo.

3.3.1 Considerações finais

Avaliação de desempenho é de fundamental importância para corretamente mapear os usos de recursos de um sistema computacional nas diversas situações, saber o que esperar dele e dimensionar suas capacidades de responder a demandas de carga. Para isso utilizamos métricas que definem esse desempenho esperado na forma de tempos de resposta, percentagem e total de uso de recursos, responsividade, etc. Não apenas saber o que podemos esperar de um programa em execução, mas saber qual é seu comportamento sujeito à sua carga máxima é importante para dimensionar corretamente o *hardware* utilizado e o quanto de demanda o sistema suporta adequadamente.

Dentro do estudo de avaliação de desempenho é comum estudar a causa raiz de anomalias e condições indesejadas de desempenho. Estas causas com frequência têm origem na falha humana, como defeitos de código. Ir ao encontro das causas de anomalias, especialmente quando estas foram causadas por defeitos, é uma forma de teste de software, mas que está cada vez mais sendo parte das atividades de avaliação de desempenho. Existe, portanto, uma intersecção entre

as duas áreas, na busca de causas de anomalias de desempenho, e esta intersecção está no escopo de estudo desse projeto.

Diversos estudos são feitos na detecção eficiente de anomalias de desempenho e relação com suas causas, mas a identificação dessas anomalias pode não ser uma tarefa fácil. Portanto utilizar uma ferramenta adequada de agrupamento é de fundamental importância para alcançar resultados em casos onde os padrões de uso de recurso fogem ao trivial, sendo este o tema do próximo capítulo.

CAPÍTULO

4

MINERAÇÃO DE DADOS

Aprendizado de máquina é o estudo de algoritmos e modelos estatísticos usados em sistemas computacionais para realização de tarefas sem que haja um procedimento claro, mas sim baseado em padrões ou inferência. São em geral tarefas originariamente realizadas por seres humanos. A atividade de agrupamento de medições de desempenho por similaridade se enquadra nessa categoria, e a proposta deste projeto engloba sua automatização.

Mineração de dados é o processo de explorar grandes quantidades de dados à procura de padrões consistentes, como regras de associação ou sequências temporais, para detectar relacionamentos sistemáticos entre variáveis, detectando assim novos subconjuntos de dados. Também faz uso de estatística, e tem uma grande intersecção com a área de aprendizado de máquina, por exemplo quando o objetivo é automatizar um processo de análise de dados não-classificados.

Neste capítulo são abordados conceitos em comum de aprendizado de máquina e mineração de dados, e quais técnicas foram utilizadas nesse projeto.

4.1 Conceitos gerais

O termo aprendizado de máquina pode ser descrito como a detecção automática de padrões relevantes nos dados, ou seja, extrair conhecimento útil a partir de um conjunto de informações aparentemente desconexas. Essa tarefa, a principio feita unicamente por seres humanos, tem sido feita de forma satisfatória, e as vezes excelente, quando automatizada por meio de programas de computador. Estes utilizam modelos, em geral com embasamento estatístico, para associar os elementos de informação em classes, grupos, ou medidas de similaridade. (GAMA *et al.*, 2011)

Existem basicamente dois tipos de aprendizado, o supervisionado e o não-supervisionado. Ambos compartilham entre si algoritmos e técnicas e em alguns casos são aplicados conjunta-

mente. No aprendizado supervisionado partimos de um conjunto chamado conjunto de treinamento composto por amostras que já estão previamente classificadas. Utilizamos esse conjunto para alimentar o modelo e então ser possível, por exemplo, classificar novas amostras desconhecidas. No processo de validação usualmente utilizamos o próprio conjunto de treinamento como conjunto de testes, onde treinamos o modelo com parte do conjunto e depois testamos com outra parte, verificando assim a acurácia da proposta. Desse tipo, a técnica mais comum é a classificação de dados.

No aprendizado não-supervisionado, não partimos de um conjunto previamente classificado, e precisamos extrair informação desse conjunto sem informações adicionais àquelas próprias dos dados a serem classificados. A técnica mais comum é o agrupamento (ou *clustering*), cujo objetivo é identificar similaridades entre as amostras, a partir de alguma métrica, e assim identificar grupos de amostras no conjunto de teste.

4.2 Agrupamento

Não existe uma definição única daquilo que possa ser considerado um grupo (*cluster*), por isso existe uma grande quantidade de algoritmos diferentes de agrupamento. Grupos, classes, ou apenas conjuntos significativos que compartilham características são parte fundamental de como os seres humanos veem e entendem o mundo. Classificar objetos faz parte da nossa forma de interpretar ambientes e entender o ambiente. No contexto de conhecer os dados, os grupos são as prováveis classes e a análise de agrupamento é a técnica de encontrar classes automaticamente. (JAIN; MURTY; FLYNN, 1999)

Agrupamento pode ser considerado uma classificação de dados que identifica classes para as amostras, onde as classes são definidas a partir das próprias amostras. Esse agrupamento pode ser feito por meio da separação do conjunto de entrada em classes, podendo ou não haver intersecções (uma amostra poder fazer parte de mais de uma classe ao mesmo tempo). Para onde há intersecções chamamos de particionamento *Fuzzy* (JAIN; MURTY; FLYNN, 1999). Esta divisão pode também ser hierárquica, onde os grupos apresentam algum tipo de relação entre si. Dizemos agrupamento completo quando todas as amostras de entrada são necessariamente associadas a um grupo, e parcial quando agrupamos apenas as amostras de interesse.

O agrupamento de dados, embora eficiente e muito utilizado, tem alguns problemas relativos ao seu uso. O primeiro deles diz respeito a formatação dos dados, pois necessitamos dispor de dados de entrada que estejam organizados de forma adequada para que a análise traga resultados consistentes. Essa formatação pode se referir desde ao formato do arquivo de dados até o adequado dimensionamento e normalização das amostras, que dependendo do algoritmo utilizado, precisam estar em um mesmo domínio. Quando estamos realizando frequentes análises, por exemplo em análise tempo-real, estas conversões de dados podem ser custosas e difíceis.

Um outro problema do agrupamento de dados é a forma de armazenar os dados coletados.

Como em muitos casos precisamos dispor de todo o conjunto de dados a cada análise realizada, deve-se armazenar os resultados em um formato compacto, podendo, contudo, ser utilizados a qualquer momento sem processamento adicional. Visando solucionar os dois problemas mencionados, para uma melhor eficiência desse processo de agrupamento como parte da monitoração e teste de software, uma técnica chamada Distância por Compressão Normalizada é aplicada. Este método, explicado na próxima seção, provê uma forma eficiente de calcular similaridade entre amostras em formatos quaisquer, de forma simples e genérica.

4.3 Complexidade de Kolmogorov

A escolha de uma métrica adequada para medição de similaridade é uma das etapas básicas do desenvolvimento de uma aplicação de aprendizado, seja ela de agrupamento ou classificação, e por isso é uma das atividades mais sensíveis na obtenção de resultados satisfatórios. A métrica é uma função que irá dizer o quão similares ou dissimilares são duas amostras de um conjunto. Existem diversas opções de métricas, dentre elas a mais comum é a *distância euclidiana* para n-dimensões.

Contudo, quando lidamos com tipos de dados mais complexos, seja por sua natureza ou seja por sua representação, o tipo de métrica irá influenciar tanto na efetividade do algoritmo como na sua eficiência. Para conjuntos de dados muito grandes, ou entradas de valores que não estão separados em amostras, utilizar distância euclidiana pode ser muito custoso (SANCHES; CARDOSO; DELBEM, 2011).

A complexidade de Kolmogorov está definida sobre o conjunto de strings binárias (sequências de zeros e uns). Ela associa a cada *string* binária um valor numérico que é sua "complexidade". A complexidade de Kolmogorov pode ser definida simplificadamente como o tamanho do menor programa (ou descrição algorítmica) que computa na Máquina de Turing uma determinada *string* binária. Este programa poderia ser escrito em qualquer linguagem. Kolmogorov propôs esta medida como uma definição de aleatoriedade que não dependesse em assumir uma distribuição de probabilidade para todas as *strings* possíveis. Posto de forma simples, uma *string* x é dita aleatória se não existe nenhum programa K que compute x mais curto que a própria x, isto é, $K(x) >= |x|$. Tal programa não seria muito maior que a própria *string*, com $K(x) = |x| + c$, onde $c >= 0$ é uma constante independente de x (LI; VITNYI, 2008).

4.4 Distância por compressão normalizada

Distância por compressão normalizada, Normalized Compression Distance (NCD), é uma forma de medir a similaridade entre duas amostras, documentos, imagens, textos figuras, e muitos outros tipos de dados. A similaridade se baseia na compressão dos dados e posterior medição do quão diferentes eles estão. Apesar de existirem muitas técnicas e métodos de medir

a similaridade de dois tipos de dados, a maioria deles é altamente especializada e dependente do formato dos dados. A ideia por trás da medição de distância por compressão é, todavia, poder medir as diferenças e similaridades de quaisquer tipos de dados, sem considerar seus contextos específicos, e com boa eficiência e simplicidade (LI *et al.*, 2004).

Ela é uma técnica que pode encontrar relações entre dados, determinando semelhança entre as variáveis com base nos tamanhos de seus dados compactados e descompactados. Essa abordagem, desenvolvida na Teoria da Informação, (CLEARY; WITTEN, 1984), não requer qualquer conhecimento específico do domínio de aplicação. De acordo com Soares *et al.* (2016), a NCD é uma métrica universal e robusta que tem sido aplicada com sucesso em áreas como a genética, literatura, música e astronomia.

A *NCD* é baseada em outra métrica chamada *NID* (do inglês Normalized Information Distance), que considera a semelhança entre as variáveis de acordo com a característica dominante que elas compartilham. No entanto, a NID utiliza diretamente o conceito de complexidade de Kolmogorov no cálculo da distância, que é computacionalmente inviável para amostras grandes. A NCD substitui o cálculo da complexidade de Kolmogorov por uma aproximação obtida a partir de um algoritmo de compressão. Na prática, a distância entre dois dados X e Y em NCD é um número positivo variando entre [0;1+e], que representa o quão diferente X e Y são, e o parâmetro *e* é um limitante superior para o erro do compressor usado. (SANCHES; CARDOSO; DELBEM, 2011)

Como *NCD* trabalha com compressão, deve-se também escolher um algoritmo de compressão. Um algoritmo muito utilizado é o *PPM* (em inglês *Prediction by partial matching*), que é um algoritmo sem perdas de compressão estatístico e adaptativo baseado em modelagem e predição. O PPM modela a utilização de símbolos para prever o próximo símbolo de uma sequência a ser compactada. Também é muito utilizado em análises de agrupamento (CLEARY; WITTEN, 1984). *PPMd* é uma implementação de *PPM* utilizada em diversos programas de código-aberto como *7-zip* e *ICEOWS*. (SHKARIN, 2002)

4.5 Relação com o projeto

O objetivo do projeto do monitor de perfis de desempenho é a identificação automática de anomalias de uso de recursos causados pela presença de defeitos no código. Como também já foi mencionado, não há requisitos de desempenho e não há como dizer se o uso de determinado recurso está dentro do considerado normal. Esse conhecimento do perfil normal de uso deve ser extraído do conjunto de execuções do SUT, e as anomalias também. Em outras palavras, os valores de métricas de desempenho são considerados normais ou não baseado em comparação com as amostras anteriores feitas a partir da execução do programa original, ou após sofrer alguma alteração. Todas as execuções realizadas, desde o início do teste, são consideradas para montar um conjunto de informações de referência, agrupadas progressivamente em um ou mais

grupos.

Estamos, portanto, nos referindo ao aprendizado não-supervisionado, mais especificamente de agrupamento. A partir de um conjunto de amostras de uso de recursos o objetivo é identificar aquelas que fazem parte do grupo de funcionamento normal, e aquelas que provavelmente fazem parte de um ou mais grupos de possíveis anomalias.

Para validar nossa abordagem, devemos partir de um conjunto de medições do funcionamento normal de um dado programa *P*, que chamaremos de entradas de nosso modelo de aprenizado. Como *normal* definimos o funcionamento original de *P*, que através de outras ferramentas de teste consideramos ser um *P* sem defeitos. Para criação dessas medições de referência, devemos usar um conjunto de dados de teste de *P* que consideramos ser adequadas, ou seja, que reflitam o usos mais frequentes de *P* . Ao executar os dados de teste e monitorar o uso dos recursos, teremos o conjunto de medições que servem como entrada do modelo de aprendizado. Feito isso teremos um conjunto de medições de desempenho que refletem o funcionamento esperado da aplicação sujeita às cargas de trabalho mais frequentes. Esse conjunto serve como base para os agrupamentos de teste.

Para verificar o funcionamento do método, devemos dispor de um conjunto de medições que apresentem anomalias, e assim acompanhar se o método automático será capaz de identificálas e com qual eficácia. A criação do conjunto de anomalias utiliza uma técnica de geração de mutantes, como foi descrito no Capítulo de Teste de Software. A execução do conjunto de entradas de *P* por esses mutantes nos dá entradas para o monitor da classe “anomalia”.

Uma vez gerados conjuntos de valores a partir do monitoramento periódico de desempenho, é necessário aplicar uma métrica para agrupar os valores obtidos. A técnica NCD é bastante adequada para este tipo de aplicação pois:

- Pode trabalhar com dados em diferentes formatos sem necessidade de pré-processamento.

- Não é dependente de alinhamento de tempo das amostras nem de sincronia de tempo de amostragem para efetuar comparação entre diferentes execuções.

- Pode trabalhar diretamente com dados binários e assim economizar espaço de armazenamento.

- É eficiente ao ponto de ser usado em agrupamento de amostras em tempo-real.

4.6 Extensão do DAMICORE para uso em Teste de Software

4.6.1 Introdução

A metodologia **DAMICORE**, desenvolvida por Sanches, Cardoso e Delbem (2011), apresenta-se como um método genérico capaz de produzir agrupamentos em quaisquer tipos de dados. Como métrica de comparação, a metodologia utiliza a distância por compressão normalizada *NCD*. É uma métrica que pode encontrar relações entre dados, determinando semelhança entre as variáveis com base nos tamanhos de seus dados compactados e descompactados.

4.6.2 Funcionamento geral

O funcionamento geral da metodologia se baseia no fluxo métrica-simplificação-agrupamento, que na prática consiste em uma composição de ferramentas:

- Cálculo de uma matriz de distâncias entre as instâncias de um conjunto de dados binários, utilizando a métrica NCD com compressor PPMd;
- Transformação de matriz de distâncias em uma árvore binária sem raiz contendo instâncias como folhas, onde os comprimentos das arestas indicam o grau de similaridade entre as instâncias;
- Agrupamento hierárquico das instâncias de forma a obter uma partição do conjunto de dados originais.

4.6.3 Escolha do DAMICORE

Um dos grandes desafios das tarefas de agrupamento de classificação de dados é a formatação dos dados de entrada. Cada domínio de problema normalmente exige a definição de um conjunto de atributos para que depois uma comparação seja feita. Através de *NCD*, o **DAMICORE** permite a comparação entre dados não-formatados, sem que essa definição explícita de atributos seja feita. Isso permite a comparação e agrupamento de resultados de diferentes execuções, para diferentes cargas de trabalho, de uma forma mais simples e genérica, podendo, inclusive, variar as escolhas de métricas de desempenho.

Não menos importante, é o problema do tempo de processamento. Modelos de aprendizado costumam ser lentos quando utilizamos grandes quantidades de amostras com muitos atributos, tornando o seu uso frequente um gargalo para o processo. Neste projeto propomos analisar perfis inteiros de dados coletados periodicamente

Requisitos Funcionais

em mais de uma execução, para diferentes entradas. Toda essa informação adquirida deve ser armazenada e tratada de forma eficiente para que o projeto seja viável em sua utilização final.

O **DAMICORE** vem facilitar a atividade, ao atender esses dois requisitos. Ao trabalhar com dados binários em seu formato bruto, facilita o processo de aquisição de conhecimento a partir dos perfis de execução, pois estes podem ser armazenados em formato binário e já incorporados à base de aprendizado sem maiores tratamentos. Outra vantagem é seu desempenho que tem demonstrado ser bastante eficiente até mesmo para aplicações que fazem análise dos dados em tempo-real. Essa característica é fundamental para este projeto, que propõe uma ferramenta de monitoramento de "saúde" de programas de uso frequente. A rapidez para apresentar os resultados da análise é também um requisito funcional da ferramenta. (MEDEIROS, 2016)

Outra característica do **DAMICORE**, está na sua capacidade de lidar com dados não formatados, como logs, medições quaisquer, figuras e até códigos-fonte. Essa flexibilidade permitiu que o trabalho de medição de desempenho pudesse ser feito de forma independente do método de agrupamento. Em outras palavras, foi possível experimentar diferentes formatos de representação da informação, variar a quantidade de medidas usadas, quais medidas, e como são representadas. Tanto o formato dos dados como as métricas utilizadas foram selecionados dentre muitas opções, e a simples troca de qualquer uma dessas definições não exige que o módulo de agrupamento seja reprogramado, nem mesmo re-configurado. Basta trocar as métricas e agrupar normalmente.

Métodos de comparação e detecção de similaridades entre séries temporais numéricas são por si só grandes desafios, pois pequenas variações em ondas semelhantes, visíveis ao ser humano, para um algorítmo podem não ser tão simples de detectar. Por exemplo dois formatos de onda iguais, porém levemente defasados, para um ser humano pode ser trivial de ver a similaridade, mas um algoritmo de comparação ponto a ponto poderia retornar que são completamente diferentes. O **DAMICORE**, pelo seu método de comparação é capaz de definir grupos e detectar similaridade, pois as etapas de compactação e definição de hierarquias consegue extrapolar as similaridades mesmo em casos mais complexos. Por exemplo, na comparação de ondas citadas, após ser feita a compactação das duas ondas defasadas, os espaçamentos seriam ignorados, e o resultado seria bem similar para ambas as entradas.

No caso proposto por este estudo é ainda mais complexo, pois o objetivo não está em apenas separar os dados em um número inicialmente desconhecido de categorias, mas ainda a criação de novas categorias para casos entradas que sejam discrepantes ao restante. Ou seja, é uma aplicação de agrupamento para dados não formatados nem normalizados, de um número desconhecido de categorias, cujo objetivo é a detecção de presença de amostras discrepantes,

agrupadas conjuntamente.

CAPÍTULO

5

METODOLOGIA

Neste capítulo são abordadas as questões de pesquisa e as motivações originais do projeto. Também são descritos os cenários de teste que são foco da pesquisa, e quais os resultados esperados. Por fim, propõ-se a metodologia **Tricorder**, desenvolvida para auxiliar na solução de problemas de desempenho encontrados em ambientes de produção.

5.1 Objetivos gerais e específicos da pesquisa

5.1.1 Objetivo geral

O objetivo geral desta pesquisa é unir técnicas de avaliação de desempenho tanto com teste de software quanto com mineração de dados para traçar perfis de execução de programas e conseguir identificar possíveis anomalias causadas por defeitos no código.

5.1.2 Objetivo específico

Para isso, propõe-se a monitoração do uso de recursos de programas executados com carga usuais de trabalho, usados como padrões, verificando se há diferenças significativas entre os comportamentos dos programas originais e dos programas com defeito. As diferenças entre os comportamentos poderia ser obtida por meio de método de agrupamento hierárquico (não-supervisionado). Dessa forma, não seria necessário conhecer os padrões de defeito, o que é importante na prática, uma vez que pode ser difícil ou não ser possível determinar todos os tipos de defeito e as diferentes intensidades com que eles se manifestam em termos de uso de recursos. Assim, por meio da coleta dos dados em produção de um programa P e da análise do agrupamento das medições do uso dos recursos, o sistema monitor proposto poderia detectar sozinho a existência de um resultado estranho, ou seja, o resultado de um possível defeito no código (refletido em anomalia de uso de recursos).

5.2 Questões de pesquisa

5.2.1 Monitoração periódica de recursos

Tendo como objetivo traçar um perfil de uso de recursos, o projeto utiliza em sua análise, como dados de entrada, as informações a respeito do processo a ser monitorado que estão presentes e disponíveis no Sistema Operacional. Essas são informações genéricas a qualquer processo, separadas em 3 categorias:

- Uso de CPU;
- Uso de memória principal; e
- Uso de memória secundária (I/O).

Essas informações devem ser amostradas periodicamente, em um intervalo definido. Se esse intervalo for muito grande, informações sensíveis podem ser perdidas entre amostras, por outro lado se for pequeno demais, além da maior quantidade de dados a serem tratados, o processo de amostragem pode ter impacto negativo nos recursos da máquina. Optamos pelo intervalo de amostragem fixo de 100 ms, baseado nas instruções de uso de biblioteca *psutil*, como sendo o mínimo intervalo entre as leituras recomendado. Esta frequência de leituras de 10 Hz, em experimentos preliminares, se mostrou adequada para o tipo de aplicação estudado (RODOLA, 2019).

Diferentes sistemas operacionais possuem suas próprias ferramentas nativas para monitorar o uso de recursos de processos, cada uma com suas características e limitações. A maioria porém disponibiliza *APIs* nativas para a coleta desses dados. Existem também ferramentas externas que além de amostrar, também podem gerar gráficos e pequenas análises - um exemplo famoso é a ferramenta de monitoração Ganglia (2019).

Um dos objetivos desse projeto é a criação de uma ferramenta capaz de executar em ambiente de produção as análises propostas e exibir resultados. Essa ferramenta deve ser simples de usar e configurar, portável para diferentes sistemas, e adicionando o mínimo de dependências. Visando atender esses requisitos, a biblioteca *psutil*, para linguagem *Python*, foi a melhor opção de coleta de dados, pois pode ser incorporada na aplicação de monitoração, e utilizada da forma que for mais adequada, sem configurações adicionais (RODOLA, 2019).

Em geral servidores de dados, ainda que possam ter transações de tempo limitado, executam, em teoria, de forma contínua, sem limite defindo. Em especial servidores que trabalhem como processadores de *stream* contínuos, como monitores em modo *DataFlow*, executam de forma contínua e autônoma, sendo foco de estudo dessa pesquisa.

Com objetivo de ser uma metodologia viável de ser aplicada em ambiente real, definimos um período fixo e padrão de medição, como uma amostra da execução do programa, para

determinada carga. Um tempo muito curto não seria suficiente para revelar anomalias causadas pela presença de algum defeito, enquanto que tempos muito longos teriam impacto tanto na quantidade de dados de medição gerados quanto no custo de processamento da análise dos resultados. Baseado no tipo de aplicação que estudamos, esse tempo foi definido como os primeiros 30 segundos de execução da aplicação, de forma empírica, sendo suficiente para revelar defeitos enquanto que os dados de medição tem tamanho viável de ser manipulado. Em estudos preliminares, medições de 1 e 2 minutos foram feitas, obtendo resultados semelhantes, porém tamanhos proporcionalmente maiores. Medições de 10 segundos também foram feitas, e estas se mostraram insuficientes para revelar anomalias, dependendo da carga de trabalho empregada.

5.2.2 Perfis de uso de recursos

Apesar das ferramentas de software atuais para detecção de anomalias de desempenho serem úteis e eficazes, elas apresentam deficiências de funcionalidade. Um dos principais problemas é que essas ferramentas apresentam perfis e análises que servem para uma execução específica do programa. Todavia, em casos de uso reais, o mais comum é que diferentes entradas do programa, em diferentes usos, levam a diferentes perfis de execução (execution profiles).

O objetivo deste trabalho é definir um perfil de execução de um programa que represente não apenas uma, mas diferentes categorias de possíveis cargas de trabalho, de forma que os dados coletados por uma execução específica possam ser utilizados para comparar totalmente um conjunto de cargas de trabalho que sejam consideradas equivalentes. Esse processo de similaridade de perfil de execução fica a cargo do algoritmo de aprendizado. Este técnica, juntamente às ferramentas utilizadas, é parte do que chamamos de metodologia **Tricorder**.

Essa etapa do **Tricorder**, visa simplificar a atividade de caracterização de carga, ou mesmo eliminá-la em alguns casos, porquanto podemos escolher amostras de diferentes usos do sistema e alimentar o monitor. A medida que o SUT é estimulado e dados são coletados, fica a cargo do monitor separar as categorias de uso de recursos, até que uma quantidade adequada de execuções seja feita, construindo um conhecimento sobre como o programa se comporta. Feito isso, uma execução diferente e com dados diferentes, que gere um perfil de uso de recursos semelhante aos anteriores, pode ser automaticamente considerada normal, sem análise de um especialista. Assim, o conhecimento sobre o *SUT* vai sendo construído gradualmente, de forma iterativa, e em ambiente de produção. As cargas de trabalho podem ser cargas reais de trabalho. Não há a necessidade de geração artificial de cargas nem de análise do perfil dessas, por isso dizemos simplificar a atividade de caracterização de carga.

Uma vez que o SUT esteja sendo monitorado a tempo suficiente, um perfil diferente de uso seria logo identificado pelo **Tricorder**, que exibe um alerta sobre o perfil diferenciado apresentado. Se não houve diferença na carga de trabalho, porém o próprio SUT foi atualizado, isso pode ser um indicativo que um defeito pode ter sido inserido nessa versão.

A presença do **Tricorder** também facilita a atividade de publicação de novas versões do SUT, pois tira a necessidade de um teste de regressão quanto ao uso de recursos. Estes usos já estão na base de conhecimento do monitor, que deverá detectar automaticamente se a nova versão passar a consumir recursos de uma forma diferente.

Como o algoritmo de aprendizado trabalha na forma de categorias, caso uma mudança de perfil de execução apresentada como uma anomalia for na verdade uma mudança justificada, basta classificar esse perfil como normal, e tanto este, como perfis similares, passarão a ser considerados parte do uso normal do programa. Na prática, sendo um algoritmo de agrupamento, iterativamente o monitor é capaz de reavaliar o que é considerado normal.

5.2.3 Agrupamento de resultados e identificação de anomalia

A Análise de Mutantes é uma das formas possíveis de se conduzir o processo de validação da proposta. Esses mutantes não são avaliados quanto à sua mudança no resultado esperado, mas sim no Teste de Desempenho, onde são avaliados quanto ao seu impacto no uso de recursos.

Optamos pela geração de mutantes que tenham algum impacto no uso de recursos do programa. Estes mutantes são executados e monitorados, e cabe ao software monitor identificar que é um mutante sendo executado. Assim podemos avaliar a metodologia **Tricorder** na presença de códigos com defeito. Como o aprendizado é incremental, tanto o número de execuções do programa original como o momento em que o mutante é executado, tem impacto no teste. Por isso foi necessária uma forma de validar a detecção dos diferentes tipos de mutante.

O processo de agrupamento (*clustering*,Jain (1990)), parte de um grupo de dados originalmente não-classificados, e por métricas de semelhança separa o conjunto em um ou mais agrupamentos. Essa técnica serve para encontrar semelhanças em conjuntos quaisquer.

Contudo nosso objetivo é buscar por divergências nas medições de desempenho, relativas a uma base de conhecimento considerada como referência. Essa base é composta por amostras de medições do programa original, considerado correto. Quanto mais abrangente for a base, maior será a eficácia de comparação com amostras novas. Por outro lado, quanto maior a base, maior o trabalho de criação e maior o custo de processamento dos grupos. Devemos, portanto, criar uma base que seja representativa, para diferentes entradas, porém cujo tamanho **N** seja adequado para uso em ambiente real.

No processo de teste, iniciamos com as N execuções do programa original, criando assim uma base de conhecimento de referência. A seguir passamos a executar um determinado mutante, e a cada execução refazemos o agrupamento. Como resultado deste novo agrupamento o mutante deve ser classificado como uma anomalia.

Caso de uso do **Tricorder**:

- O ambiente original seria um programa *P*, assumindo-se ser um programa sem defeitos.

- As únicas informações utilizadas pelo programa monitor são as amostras de usos de recursos de *P*.

- Executamos *P* um número *N* de vezes, e suas medições são incrementalmente agrupadas utilizando agrupamento por *NCD* com a metodologia **DAMICORE**.

- Em certo momento, passamos a executar o mutante *M*, e o monitor deve ser capaz de identificar a anomalia, a partir de *X* execuções deste, sendo um agrupamento composto apenas pelas execuções do mutante *M*.

Portando, o critério **Tricorder** de existência da anomalia é:

- A existência de um agrupamento composto apenas por medições de execuções de M, chamadas execuções de teste

A quantidade de medições da base de referência, como mencionado, é um valor que tem impacto na proposta. Se forem muitas medições, além de tornar todo o processo custoso, na etapa de agrupamento, também torna seu uso menos eficiente. Se forem poucas, todavia, podem não ter uma variabilidade suficiente para definir o que seria um perfil esperado de uso de recursos. Em estudos preliminares, foram usadas de 10 a 50 execuções, sendo que **30 execuções** foi considerado um bom custo-benefício entre praticidade de uso e eficácia.

Uma vez definido como 30 o número de execuções de referência, para cada configuração, então o intervalo de avaliação das medições de teste passam a ser de 1 a 30.

5.2.4 Validação da proposta

Para validar a proposta, definimos um processo de verificação incremental, para cada um dos mutantes e para cada uma das configurações, de forma a cobrir todos os casos previstos.

Primeiramente, é gerado um conjunto de execuções do programa original, chamado de conjunto ou base de referência. Este conjunto é formado por 30 execuções do programa original para cada uma da 15 configurações previstas. Essa base é a mesma para todos os testes.

Iniciamos o teste escolhendo um mutante, e este é executado de 1 a 30 vezes, e a cada execução, todas as medições feitas deste mutante são agrupadas juntamente à base teste, pelo **DAMICORE**. A cada análise feita é aplicado o critério **Tricorder**. Se, entre a 1^a e a 30^a execução, o critério for atendido, então uma anomalia foi identificada.

Este processo é repetido para cada um dos 7 mutantes, para cada uma das 15 configurações, utilizando a mesma configuração em cada etapa de 30 execuções.

O processo também é aplicado ao conjunto de 30 execuções feitas pelo programa original, como grupo controle, onde o certo é não haver anomalias. Se houver, então é um caso de falso-

positivo. Há também casos de falso-positivo para determinadas entradas, mas estes são melhor descritos no Capítulo 7.

5.3 Materiais e métodos

5.3.1 Quais programas testar?

A princípio o **Tricorder** pode ser utilizado para monitorar qualquer programa que esteja em execução como um processo. Porém, devido a suas características, teria uso eficaz em aplicações que fazem grande uso de recursos, como aplicações de servidores web, compactadores, conversores, replicadores de dados na rede, compiladores e processadores de dados em geral.

A motivação do projeto é testar programas de uso frequente, que rodam normalmente de forma autônoma, as vezes por longas horas, dividindo recursos com outras aplicações, e que sejam difíceis de verificar se estão funcionando corretamente devido a características como:

- serem dinamicamente configuráveis;
- reativos e adaptativos;
- rodarem em locais de difícil acesso;
- lidarem com grandes massas de dados;
- ausência ou dificuldade de criação de oráculo de saídas esperadas; e
- ausência de requisitos de desempenho.

Como exemplo podemos citar programas de processamento de fluxo de dados, de *stream* de vídeo, ferramentas autônomas de cópia e compactação, e programas de telemetria em geral.

5.3.2 Ferramentas e bibliotecas

O *Windows Performance Monitor*, (MICROSOFT, 2017), é uma ferramenta do Sistema Operacional *Windows* capaz de monitorar o uso de recursos referente a processos ou ao recurso de interesse. Nele também é possível armazenar resultados, configurar aquisição periódica e visualizar os dados de forma gráfica. É uma ferramenta muito útil para especialistas que queiram monitorar a saúde de seu programa no *Windows*. Sua maior limitação está na opção de periodicidade das aquisições, que é no mínimo 1s.

O Ganglia (2019) é uma ferramenta escalável para monitoramento de sistemas de alta-performance, como *clusters* e *grids*. Ele tem código aberto e é largamente utilizado, principalmente em sistemas *Linux*. Ele é útil principalmente quando se quer monitorar o uso de recurso de mais de uma máquina (ou nó) a partir de outra, de forma centralizada e com baixo

impacto de custo de execução na máquina em teste. Sua principal limitação é, por ser uma ferramenta completa e poderosa de monitoramento de sistemas complexos, exigir que seja feita uma configuração do ambiente, que não é trivial. Sua compatibilidade com sistemas *Windows*, embora existente, é ainda mais complexa. Não foi utilizado por demandar muitas mudanças e configurações em ambiente de produção.

O *psutil*, (RODOLA, 2019), é uma biblioteca multi-plataforma, em *Python*, para aquisição de informações referentes a processos em execução e a utilização do sistema. É feita para desenvolvimento de aplicações de monitoramento de sistemas. Ela basicamente encapsula as APIs de aquisição de informações de diferentes sistemas operacionais em estruturas padrões de representação de dados e uso, de forma que é possível fazer um software de monitoramento multi-plataforma transparente. É versátil quanto a definições de períodos de aquisição, e ainda por ser em *Python* tem boa compatibilidade com a implementação em *Python* existente da biblioteca **DAMICORE**, reduzindo o acoplamento entre os módulos. Foi a biblioteca escolhida por ter poucas dependências, ser leve, e poder ser utilizada em ambiente de produção com pouco impacto.

5.4 Proposta de inovação

5.4.1 Teste de desempenho na ausência de requisitos

A maioria dos métodos de teste caixa-preta de aplicações se utilizam dos requisitos definidos para essa aplicação, e partem destes como referência de comparação. Em geral é comparada a saída esperada do programa, e, se existirem, são comparados outros aspectos. Porém, mesmo na ausência de requisitos de desempenho, podem existir defeitos de desempenho e estes causarem falhas graves. A não existência de requisitos de desempenho não tira a necessidade de utilizar o desempenho como medida de conformidade da aplicação. Porém como testar um programa cujo desempenho esperado não é bem conhecido, e este estando sujeito a variação de carga de trabalho? Esta metodologia visa facilitar o processo de definição de um perfil de desempenho esperado, em conjunto à atividade de dimensionamento de cargas de trabalho.

5.4.2 Identificação de defeitos que não alteram a saída esperada da aplicação

Um dos grandes problemas da atividade de teste tradicional é a comparação de entradas e saídas. Essa tarefa é tão complicada que muitas soluções alternativas foram adotadas para evitar que um oráculo de E/S do SUT fosse necessário.

Porém, muitos defeitos podem causar falhas que não alteram a saída esperada do programa, seja para qualquer entrada, seja para uma entrada específica. Tais defeitos, todavia, podem causar graves problemas de desempenho. Esta metodologia visa incluir esse conjunto de defeitos

que afetam primordialmente o desempenho, e que podem ficar de fora de um teste caixa-preta tradicional.

5.4.3 Simplificação do processo de dimensionamento de carga

A atividade de dimensionamento de carga de trabalho de uma aplicação costuma ser um trabalho manual executado por um especialista em testes. São realizados estudos, medições, e pode não ser simples de executar, havendo custos atrelados.

Ao realizar de forma automática o agrupamento de medições de desempenho, para diferentes entradas, esta metodologia também facilita o dimensionamento de carga, deixando a cargo do **DAMICORE** realizar a separação de categorias, e a identificação de medições que não se enquadram ao histórico de medições existente.

5.4.4 Teste em ambiente de produção

Um dos grandes problemas em ambientes de produção está no fato de quando *releases* são publicados são retiradas a maioria das checagens, *dumps* e *logs*, pois os mesmos consomem recursos. Em muitos casos, uma vez que publicamos um software em produção (modo "*release*"), tem-se o viés (acredita-se) que ele esteja funcionando corretamente, e que os testes feitos até então em modo "*debug*" foram suficientes para revelar os defeitos existentes.

O ambiente de produção costuma também ter menos recursos de monitoração, seja por economia de recursos, seja porque o ambiente de produção fica em local remoto. Há desde os casos de sistemas embarcados onde os recursos são muito limitados até casos de grandes servidores que por questões corporativas não são facilmente acessíveis aos desenvolvedores.

Tal cenário leva à necessidade de alguma ferramenta que consiga indicar, mesmo que parcialmente, se tudo está funcionando como esperado. Poder-se-ia fazer uma checagem do resultado gerado, mas isso nem sempre é viável. Indicadores de verificação, marcações de *log* também são muito úteis mas ficam a cargo do desenvolvedor criar esses verificadores, ou a cargo de um especialista analisar os *logs* e identificar falhas.

Este projeto visa ajudar nesse problema, servindo como uma espécie de monitor de "*saúde*" do software, em ambiente de produção e sem necessidade de configurações adicionais, criação de casos de teste ou um especialista conhecedor do programa em teste para analisar resultados. A proposta é automatizar boa parte desse processo, e apresentar um indicativo de saúde baseado no histórico de execuções anteriores de forma simples e concisa.

5.5 Considerações finais

Neste capítulo foram descritas as motivações e problemática originárias de nossa pesquisa. Também foram descritos os fundamentos da metodologia **Tricorder**, proposta como ferramenta

para auxiliar na solução dos problemas encontrados. As principais inovações da metodologia proposta são relacionadas a teste de desempenho de aplicativos na ausência de requisitos de desempenho, com uso de monitores de uso de recurso e de uma ferramenta de agrupamento.

O agrupamento iterativo e automático dos dados coletados simplifica a etapa de caracterização de carga ao tirar a necessidade de conhecimento da estrutura interna do programa objeto de teste. Além disso, o uso dos critérios estabelecidos serve como guia na identificação da presença de anomalias de desempenho. A aplicação objeto dos testes, na forma de um benchmark desenvolvido para esse fim, está descrita em detalhes no Capítulo 6.

CAPÍTULO

6

DESENVOLVIMENTO

Neste capítulo são descritos os requisitos e implementação da aplicação benchamark objeto de teste, utilizada no estudo de eficácia da metodologia **Tricorder**. O benchmark proposto é um sistema cliente-servidor, na forma de um produtor-consumidor de dados transmitidos por comunicação TCP/IP. Esses dados são consumidos e processados por uma biblioteca externa, também desenvolvida para o projeto.

6.1 Desenvolvimento do Benchmark

O cliente é responsável por enviar os dados coletados, enquanto o servidor recebe, e por meio da biblioteca de processamento interpreta e disponibiliza as informações recebidas. Os dados são transmitidos de forma contínua, no formato de *stream* de dados, e são processados e exibidos com mínimo de latência. Esse benchmark, cujo funcionamento é inspirado em sistemas de telemetria de dados remotos, é descrito com mais detalhes neste capítulo.

6.1.1 Descrição

Requisitos Funcionais

Com objetivo de reproduzir, em ambiente controlado, o reflexo das diferentes entradas no uso de recursos, e assim poder verificar as mudanças causadas pela presença de defeitos, para este projeto foi desenvolvido um benchmark de testes, baseado na arquitetura cliente-servidor de processamento de dados, de carga configurável.

Seguindo um modelo de processamento de *stream* de dados, o benchmark criado segue a arquitetura de produtor-consumidor, onde o cliente produz dados e o servidor os consome,i.e., ele recebe, processa e disponibiliza, via uma *Application Programming Interface* (API), os dados coletados. Esse servidor, funcionando em modo passivo-reativo, recebe os dados enviados em diferentes taxas, via uma conexão de rede, e faz uso de uma biblioteca de processamento para

interpretar os pacotes. Seu papel é de apenas escutar a conexão com o cliente e encaminhar os dados para a biblioteca.

A ideia de haver uma biblioteca externa de processamento se baseia em condições onde há uma biblioteca de terceiros para interpretar os dados coletados. Esta biblioteca, produzida por fornecedores, costuma vir como um binário compilado, e uma interface de uso, sendo tratada como caixa-preta. Um exemplo de aplicação deste tipo são sistemas de telemetria que utilizam hardwares de terceiros. Estes podem possuir formatos proprietários na geração dos dados, e por isso são fornecidas bibliotecas de software para interpretar e converter a informação lida.

Nesse cenário, definimos este benchmark, que foi projetado com os seguintes requisitos:

- Ser escrito em linguagem de código nativo, no caso C++;
- Enviar e receber pacotes predefinidos, porém de tamanho variável, em uma conexão persistente TCP/IP;
- Fazer uso de uma biblioteca de processamento, que é responsável pela interpretação dos dados recebidos; e
- Receber e processar pacotes de forma responsiva, com taxa variável pré-configurável no cliente.

A biblioteca de processamento utilizada foi desenvolvida de modo a reproduzir uma possível biblioteca de terceiros, cujo código-fonte nem sempre está disponível, podendo ser proprietária. Esta biblioteca tem os seguintes requisitos:

- Ser escrito em linguagem de código nativo, no caso C++;
- Processar pacotes de tamanho variável;
- Receber e processar informação no formato **CSV**, disponibilizando interface de leitura do último relatório recebido; e
- Receber e processar informação no formato **BIN**, disponibilizando interface de leitura de uma janela móvel de intervalo configurável.

Os relatórios *Comma Separated Values* (CSV) devem ser acessados por meio de estruturas de dados que facilitem o acesso direto a linhas e colunas. Essa leitura deve ser *thread-safe*, porém sem atrapalhar o processamento dos dados.

Os dados *Binary values* (BIN) devem ser disponibilizados em formato de janela móvel *Current Value Table* (CVT) de valores condensados. Esses dados estão divididos por *id* do sensor, onde cada linha representa K medidas daquele sensor. A biblioteca de processamento deve, para cada K medições, efetuar e prover os seguintes cálculos:

- Média;
- Mediana;
- Moda;
- Valor máximo; e
- Valor mínimo.

Portanto, para dados **BIN**, a biblioteca deve fornecer acesso aos valores condensados referentes aos último T segundos, de acordo com a configuração existente, enviada como cabeçalho ao servidor. Assim como os dados **CSV**, os dados **BIN** também devem ter seu acesso *thread-safe*, contudo evitando atrasos ou perdas no processamento dos pacotes.

6.1.2 Arquitetura do Produtor-Consumidor

YATServer

Desenvolvido em C++, usando o framework Qt, o **YATServer** (*Yet Another Telemetry Server*), é um servidor de dados que funciona de forma passiva, aguardando a conexão de um cliente. Ao se conectar, recebe os dados enviados pelo cliente, e baseado em um cabeçalho padrão, aguarda o término do envio do pacote e faz uso da biblioteca de processamento para converter e disponibilizar as informações.

YATClient

Desenvolvido em C++, também usando o framework Qt, o **YATClient** (*Yet Another Telemetry Client*), recebe como entrada um arquivo de configuração que diz quais e como devem ser enviados os *streams* de dados para o servidor. Em seguida, ele abre os arquivos de entrada, nos padrões definidos, simulando leituras, e envia ao servidor nas frequências definidas, via uma conexão TCP única. Este arquivo de configuração também é usado por uma outra ferramenta, o *Workload Generator*, que cria arquivos de entrada para o *Client*, seguindo as definições de cada *stream*.

Biblioteca de Processamento

Desenvolvida em *C++ standard*, a biblioteca é composta por uma API externa de uso, definida por um arquivo de cabeçalho, e um arquivo *.lib* para ser compilado em conjunto ao projeto. Sendo principal foco dos testes, todos os defeitos definidos pelo projeto foram inseridos no código-fonte da biblioteca, para, ao monitorarmos o uso de recursos do processo **YATServer**, estes defeitos causarem falhas, e estas possivelmente serem detectadas.

Projetada de forma a ser simples de usar, a biblioteca é capaz de processar dados em 2 formatos:

- **BIN**: dados binários contendo informações das medições, que ficam disponíveis em formato condensado de **CVT** de uma janela de tempo;

- **CSV**: dados em formato texto, que são tratados como um relatório e ficam disponíveis como linhas e colunas de uma tabela de informações textuais, sempre o último sobrepondo o anterior.

6.1.3 Uso e funcionamento

Seguindo um modelo de produtor-consumidor via conexão TCP, temos um cliente (**YATClient**) que é responsável pela leitura e envio dos dados. Em uma situação real esses dados poderiam ser provenientes de sensores ou algum tipo de aquisição dinâmica. Em nosso cenário, dados simulados são gerados no início do experimento, de acordo com o arquivo de configuração definido, e esses dados são enviados ao servidor, em conjunto com cabeçalhos pré-definidos, que contém as informações necessárias para processamento dos dados. A frequência de envio também está no arquivo de configuração.

Na outra ponta do sistema encontra-se o servidor (**YATServer**), que espera de forma passiva a conexão de um cliente. Ao conectar, o cliente passa a enviar pacotes que descrevem a natureza do dado enviado e como deve ser sua exibição, além de enviar também os dados de forma contínua. O **YATServer** faz uso da biblioteca de processamento para interpretar e acessar a informação recebida. Ao mesmo tempo, seguindo a metodologia **Tricorder**, o monitor de desempenho passa a armazenar os dados de uso de recursos do **YATServer**, para posterior análise.

Na Figura 1 é possível ter uma visão geral do cenário proposto.

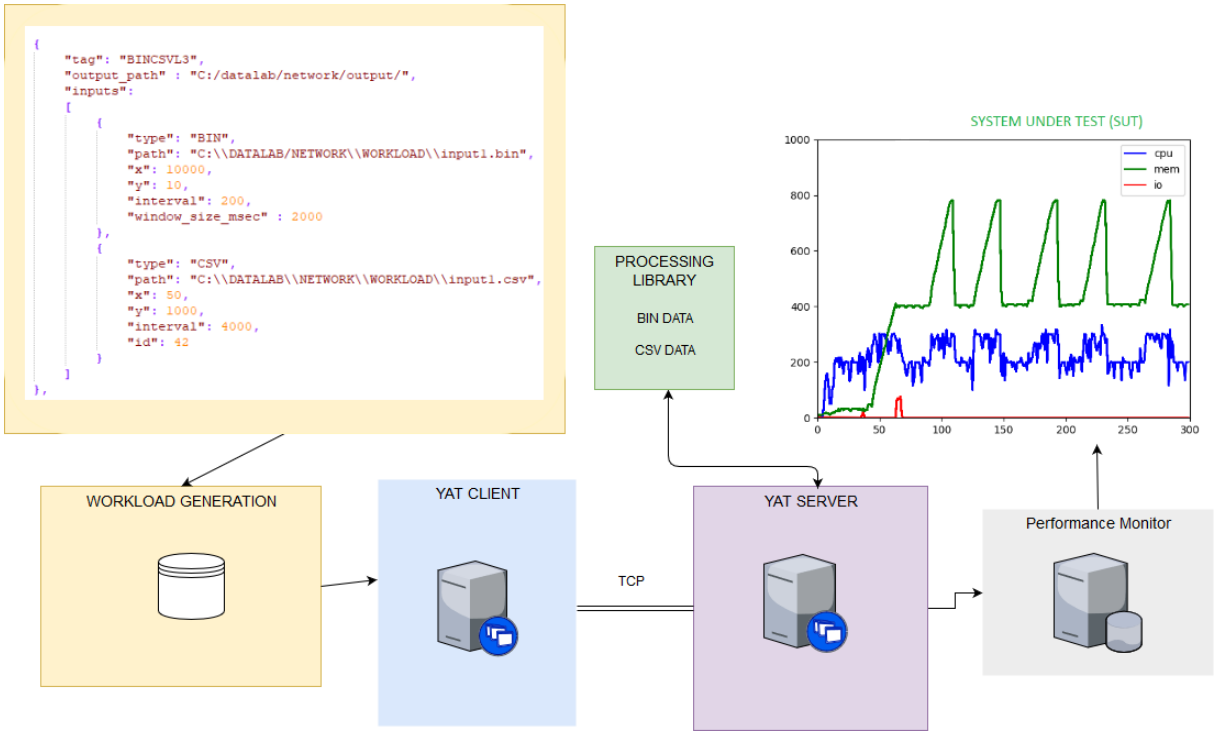


Figura 1 – Arquitetura do produtor-consumidor

A Figura 2 representa uma possível presença de defeitos na biblioteca de processamento, e estes podem refletir no desempenho do **YATServer**. Nesse projeto procuramos inserir artificialmente os defeitos na biblioteca, e, no cenário descrito, sermos capazes de detectar a presença de anomalia nas medições do uso de recursos do **YATServer**.

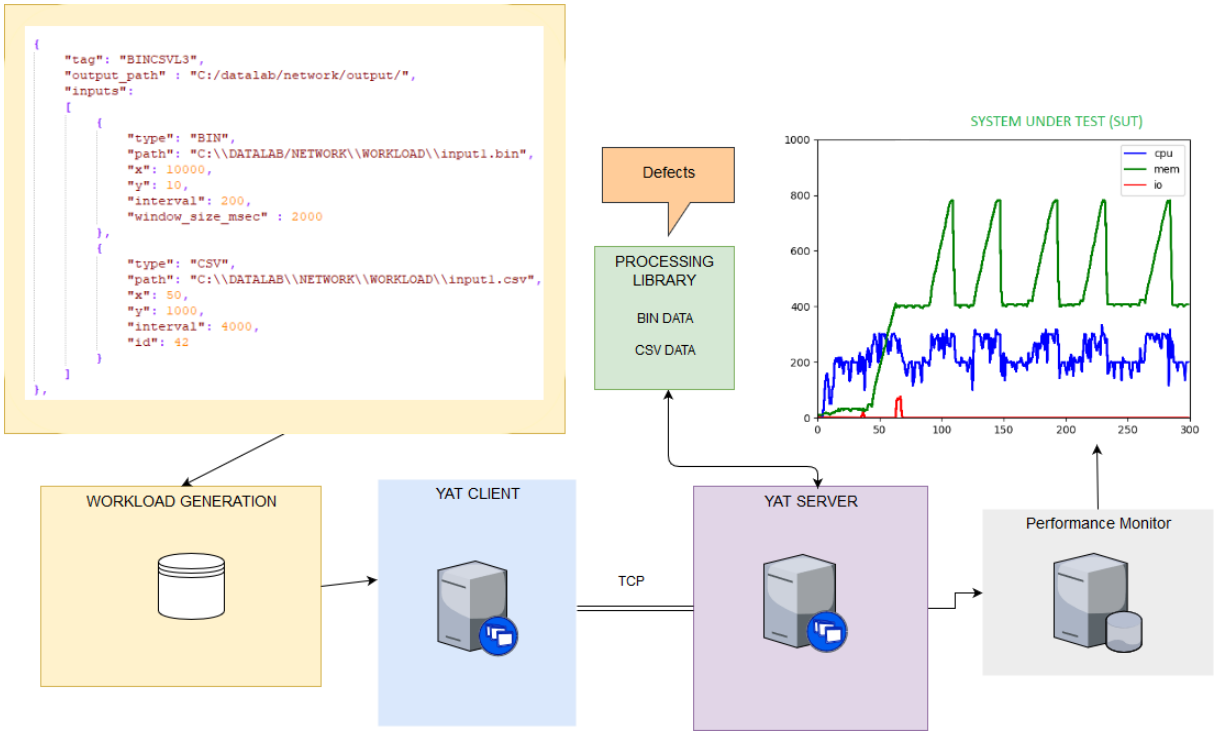


Figura 2 – Produtor-consumidor com defeito inserido

6.1.4 Cargas de trabalho

Diferentes entradas podem variar o uso de recursos de forma significativa. Baseado nisso o **YATClient** foi projetado para receber um arquivo de configuração de entrada, que possibilita selecionar:

- Tipo de entrada: binária, **CSV** ou ambas;
- Frequência de envio;
- Dimensões da informação enviada; e
- Caminho do arquivo a ser enviado.

Desta forma é possível, no início do experimento, configurar a carga de trabalho do sistema de acordo com as informações fornecidas. Os efeitos da configuração no desempenho dependem dos recursos disponíveis, e isso varia de uma máquina para outra. As cargas de trabalho foram definidas em 2 modos: única entrada e entrada mista. Isso se deve ao tipo de entrada que pode ser de um tipo único ou serem uma mistura dos 2 tipos disponíveis. Isso afeta não somente o perfil de desempenho da aplicação servidor, como também se o defeito pode ou não causar falha.

Uma vez definida a máquina em que seriam executados os experimentos, variamos os diferentes atributos de configuração para criar um total de 15 diferentes configurações, as quais chamamos de 15 categorias de carga de trabalho. Estão divididas em 6 configurações de entrada única, e 9 configurações de entrada mista. Dizemos categorias, pois a informação enviada pode, e foi, gerada novamente para cada experimento, com base em geradores numéricos randômicos. O módulo **workload generator** foi criado com essa finalidade de simular dados que estejam sendo obtidos de forma empírica, sendo definidos apenas o formato e tamanho da informação enviada.

6.1.5 Saída esperada

O YATServer, por meio da biblioteca de processamento, tem acesso aos dados de entrada BIN processados e disponibilizados em formato de janela deslizante de valores númericos. Esses valores são atualizados dinamicamente, enquanto houverem entradas de dados. Como saída esperada, definimos sendo a primeira janela de dados completa a ser disponibilizada. Esta então é salva em disco como a saída BIN.

Da mesma forma que com dados BIN, dados CSV chegam continuamente em formato de relatórios. Portanto, como saída esperada consideramos o primeiro relatório recebido, que é salvo em disco como saída CSV.

6.2 Defeitos de Software

Para verificar a eficácia do projeto na identificação de anomalias, foram criados mutantes com a característica comum de afetarem o desempenho. Com tal objetivo, 7 categorias de problemas de desempenho foram definidas, inspiradas em problemas reais encontrados em ambiente industrial. São problemas causados por defeitos de código, cuja falha afeta o uso de recursos, mas não necessariamente alteram a saída esperada.

Neste capítulo são descritas as categorias de defeitos usadas no projeto. Cada defeito foi representado por uma implementação, que pode ser ativada ou desativada em tempo de compilação. Assim é possível inserir cada um dos diferentes defeitos, um por vez, criando um mutante para cada categoria. Esses mutantes são usados na validação de uso da metodologia **Tricorder**.

6.2.1 Limpeza de memória: BCLEAN

Chamamos de BCLEAN o mutante que representa a categoria de defeitos de uso excessivo de memória. Neste caso a memória que foi alocada no início do processamento não é liberada no final. Na ausência de *garbage collector* ou *smart pointer*, o resultado é um uso crescente de memória.

```
153  #ifdef FAULT_BINPKTPROCESSOR_PROCESS_CLEANUP
154  #pragma message("-------------------------------------")
155  #pragma message("FAULT_BINPKTPROCESSOR_PROCESS_CLEANUP")
156  #pragma message("-------------------------------------")
157  #else
158      Cleanup();
159  #endif
```

Figura 3 – Seleção do defeito BCLEAN

```
41  void CBinPktProcessor::Cleanup()
42  {
43      if (m_pkt)
44          delete[] m_pkt;
45      m_pkt = nullptr;
46  }
```

Figura 4 – Código referente ao defeito BCLEAN

Na Figura 3 podemos ver o defeito inserido, ao não chamar a função de Cleanup. Essa função, cujo código está na Figura 4, é responsável por liberar a memória do pacote *m_pkt* de dados BIN, após este ter sido processado.

6.2.2 Uso excessivo de CPU: INFINITE

O mutante INFINITE representa a categoria de defeitos em que um processamento desnecessário acontece. Pode ser uma espera ocupada por algum evento que já ocorreu ou que não irá ocorrer, ou pode ser a chamada de funcões desnecessárias ou uma condição de *loop* que por algum engano nunca dará retorno. São diferentes cenárioe, por simplicidade, foram representados por um *loop* no trecho final de execução da *thread*.

```
194  #ifdef FAULT_CSVPKTPROCESSOR_PROCESS_INFINITE
195  #pragma message("-------------------------------------")
196  #pragma message("FAULT_CSVPKTPROCESSOR_PROCESS_INFINITE")
197  #pragma message("-------------------------------------")
198      while (1);
199  #endif
```

Figura 5 – Defeito INFINITE

Na Figura 5 pode-se ver a implementação do defeito INIFNITE, na forma de espera ocupada infinita. É um *loop* infinito no final da *thread* de processamento de pacotes **CSV**.

6.2.3 Configuração de paralelismo: MONO

O mutante que chamamos MONO representa a categoria de defeitos relacionados à configuração de paralelismo de processamento da aplicação. Algumas aplicações, especialmente as de processamento de dados, permitem que o número de processamentos simultâneos (*threads*) seja escolhido por configuração. Um erro nessa configuração poderia causar problemas de desempenho, porém que talvez não alteram a saída.

```
12  #ifdef FAULT_DATAPROCESSOR_MONOLITHIC
13  #pragma message("-------------------------------------")
14  #pragma message("FAULT_DATAPROCESSOR_MONOLITHIC")
15  #pragma message("-------------------------------------")
16  #define PROCESSORS_NUM_BIN (1)
17  #define PROCESSORS_NUM_CSV (1)
18  #else
19  #define PROCESSORS_NUM_BIN (2)
20  #define PROCESSORS_NUM_CSV (2)
21  #endif
```

Figura 6 – Seleção do defeito MONO

```
33      for (int i = 0; i < PROCESSORS_NUM_BIN; i++)
34          m_vBinProcessor.push_back(new CBinPktProcessor());
35
36      for (int i = 0; i < PROCESSORS_NUM_CSV; i++)
37          m_vCSVProcessor.push_back(new CCsvPktProcessor());
```

Figura 7 – Código referente ao defeito MONO

Na Figura 6 estão as diretivas de compilação que definem o número de *threads* de processamento, para cada tipo. O defeito define o número de objetos processadores em 1, para

ambos os tipos. Como cada processador cria uma *thread* de processamento, o defeito MONO deixa a aplicação com 1 *thread* apenas, para cada tipo de dados, sem alterar a saída esperada. O resultado no uso de recursos pode ser visto na Figura 7.

6.2.4 Uso inferior de CPU: SLEEP

O mutante SLEEP representa os defeitos de chamada bloqueante, podendo ser uma espera desnecessária ou muito longa. Foi implementado como uma espera não ocupada na *thread* de execução, usando a função *sleep*. Na Figura 8 podemos ver a implementação do defeito SLEEP, na forma de um *sleep* indevidamente inserido no final da *thread* de processamento de pacotes BIN. Esta mutação representa tanto os casos de espera incorreta por eventos como casos de *sleep* de depuração esquecidos no código de produção.

```
161  #ifdef FAULT_BINPKTPROCESSOR_PROCESS_SLEEP
162          this_thread::sleep_for(chrono::milliseconds(200));
163  #pragma message("-------------------------------------")
164  #pragma message("FAULT_BINPKTPROCESSOR_PROCESS_SLEEP")
165  #pragma message("-------------------------------------")
166  #endif
```

Figura 8 – Defeito SLEEP

6.2.5 Mutação na estrutura de pacotes de rede: SWAP

O mutante chamado SWAP representa um erro comum em que a estrutura de pacotes sendo enviada não é a mesma que está sendo lida. Qualquer campo trocado pode causar falhas desde simples até críticas. Visando adequação aos objetivos do projeto, foi inserido um defeito cujo impacto não causa falha crítica. Em alguns casos não produz mudança na saída esperada, mas que causa impacto no uso de recursos da aplicação consumidora dos pacotes.

```
28  struct NetworkPktSendBIN
29  {
30      int32_t source_id;
31      int32_t pkt_len;
32      int32_t payload_len;
33      int32_t timestamp;
34      int16_t window_size_msec;
35      int16_t samples_per_channel_id;
36      int16_t samples_interval_usec;
37      int16_t acq_num;
38      char type;
39      char * payload;
40  };
41  #if defined(FAULT_NETWORK_PKT_WINDOWSIZE_ACQNUM_SWAP)
42  struct NetworkPktRecvBIN
43  {
44      int32_t source_id;
45      int32_t pkt_len;
46      int32_t payload_len;
47      int32_t timestamp;
48      int16_t acq_num;
49      int16_t samples_per_channel_id;
50      int16_t samples_interval_usec;
51      int16_t window_size_msec;
52      char type;
53      char * payload;
54  };
55  #else
56  typedef NetworkPktSendBIN NetworkPktRecvBIN;
57  #endif
```

Figura 9 – Defeito SWAP

Na Figura 9 podemos ver que o defeito está na troca entre os campos **window_size_msec** e **acq_num**, nas definições das estrutura de envio e de recebimento, que deveriam ser iguais. Como a alteração foi feita na estrutura do cabeçalho do pacote tipo BIN , esse defeito só causa falhas em processamentos de dados BIN.

6.2.6 Uso incorreto de mutex: UNLOCK

O mutante UNLOCK representa a categoria de defeitos de sincronismo com o erro (bastante comum) de deixar de liberar um *mutex* (CPPREFERENCE, 2019) previamente travado. Como a aplicação YATServer usa paralelismo no tratamento das entradas, este defeito está na ausência de liberação de um dos *mutex* existentes.

```
182  #ifdef FAULT_PROCESSOR_BUSY_UNLOCK
183  #pragma message("-------------------------------------")
184  #pragma message("FAULT_PROCESSOR_BUSY_UNLOCK")
185  #pragma message("-------------------------------------")
186  #else
187      m_busy.unlock();
188  #endif
```

Figura 10 – Defeito UNLOCK

Na Figura 10 está o trecho de código da implementação do defeito UNLOCK, com ausência da chamada *unlock()* do mutex *m_busy*, ao final do processamento de dados CSV.

6.2.7 Processamento desnecessário: NOBREAK

O mutante NOBREAK representa um caso comum em que há mais de um tipo de entrada possível em um programa, e este, por engano, processa a entrada mais de uma vez, tanto da forma correta como da forma errada. Este erro pode passar desapercebido pois a entrada é tratada da forma correta enquanto a forma errada muitas vezes é descartada. Contudo há um gasto de recursos desnecessário e que pode ser alto. Esta categoria de defeitos foi representada pela falta de um *break* no tratamento dos tipos de entrada, no *case* de tipos de pacote. Com o pacote pode ser processado por mais de uma fila de processamento, a certa e a errada.

```
87          switch (pkt->type)
88          {
89              case CSV_REPORT:
90              {
                    ...
114 #ifdef FAULT_DATAPROCESSOR_RUN_NO_BREAK
115 #pragma message("-------------------------------------")
116 #pragma message("FAULT_DATAPROCESSOR_RUN_NO_BREAK")
117 #pragma message("-------------------------------------")
118 #else
119                 break;
120 #endif
121             }
122             case BIN_CVT:
123             {
```

Figura 11 – Defeito NOBREAK

Na Figura 11 podemos ver o *case* onde são separados os tipos de pacote, e a ausência do *break* faria um pacote tipo **CSV** ser processado duas vezes: como **CSV** (correto) e como **CSV** (errado). O erro só é observado na existência de dados tipo **CSV**.

6.2.8 Verificação das saídas

O aplicativo YATServer, por meio da biblioteca de processamento, acessa e salva em disco as informações recebidas, definidas como saídas esperadas do aplicativo. Em um método de teste tradicional, estas saídas podem ser comparadas com as saídas esperadas, definindo assim casos de teste, onde podemos verificar o funcionamento adequado do programa.

Para efeito de comparação com o método tradicional, foram definidos casos de teste, onde o resultado esperado são os arquivos gravados das entradas BIN e CSV, criados pelo programa YATServer referência. Em seguida, cada mutante teve seus arquivos de saída comparados aos arquivos de referência. Para as entradas definidas para as 15 cargas, todos os resultados esperados foram iguais ao resultado de referência. Sendo assim, neste experimento, para as entradas

utilizadas, todos os mutantes apresentaram resultado igual ao programa original, podendo passar desapercebidos em um teste tradicional.

Estes casos de teste foram criados apenas para efeito de validação da proposta, pois não são necessários para o método **Tricorder**.

6.3 Framework de Teste

Para este projeto foi desenvolvido o framework de teste chamado **Tricorder**, que faz uso das ferramentas **Thermometer**, implementada para coletar dados de uso de recursos da aplicação objeto de teste, e **DAMICORE**, responsável pelo agrupamento das amostras. As etapas da metodologia de teste foram automatizadas usando *scripts* feitos na linguagem **python**. Esta última também foi utilizada na automatização do processo de validação da eficácia do **Tricorder** para o cenário de teste escolhido.

6.3.1 Monitoração dos recursos

Como explicado no Capítulo 5, neste projeto, as medições de uso de recursos do sistema pela aplicação são usadas como saída do programa, afim de avaliar se o seu funcionamento está correto, usando a técnica de teste caixa-preta. Essas medições devem ser feitas de forma padronizada e automática para que os resultados obtidos possam ser analisados segundo uma metodologia única, independente da aplicação testada. É importante que as medições sejam feitas seguindo um padrão definido de métricas, período de amostragem e formato de representação.

Diferenças na configuração do monitoramento de execução podem levar a conclusões errôneas na identificação das diferenças de desempenho. Por exemplo, no caso de duas execuções da mesma aplicação, se uma das medições contiver mais amostras que a outra poderia indicar falsamente anomalia no desempenho, ainda que as duas execuções tenham desempenho semelhante. Ainda há casos onde há uma entrada ininterrupta de dados, por exemplo processamento de *stream* contínuo. Nestes pode não haver claramente um início e fim de processamento, dependendo da quantidade de dados a ser processada. Por esse motivo, uma janela fixa de obervação, de 30 segundos, foi definida no projeto, de forma a padronizar a quantidade de amostras obtidas de cada execução, e assim avaliar diferenças dentro desse conjunto, para cada execução. Essa escolha pode ser adaptada à aplicação sendo testada, e poderia teoricamente ser qualquer medição de algum trecho da execução, ou mesmo execuções completas em casos de entrada finita.

Para atender nosso caso de benchamark, e simplificar os experimentos, criamos um perfil de execução baseado nestes primeiros 30 segundos de execução da aplicação, caracterizando aplicações que podem rodar por minutos e até por horas indefinidamente. A influência dessa definição pode ser observada no Capítulo 5.

Para implementar a metodologia **Tricorder**, um software monitor chamado **Thermome-**

ter, foi escrito em **python** utilizando a biblioteca **psutil** (RODOLA, 2019), para monitoração e armazenamento das medidas do uso de recursos do **YATServer**, de forma automática, pelo período de tempo e período de amostragem definidos. O **Thermometer**, ao ser executado, aguarda o início da execução do **YATServer**, buscando pelos nomes da lista de processos, e ao identificar o processo **YATServer** passa a monitorá-lo, pelo tempo de 30 segundos, e salva a informação coletada em formato **CSV**. Para facilitar a análise, o **Thermometer** também traça um gráfico normalizado com os resultados e salva em formato **PNG**.

6.3.2 Geração de dados

Também feito em **python**, o framework de teste do **Tricorder** recebe como entrada um arquivo formato **json**, com todas as configurações de carga a serem utilizadas no teste. Então o framework executa os seguintes passos:

- Percorre a pasta de binários e lista todos os mutantes existentes para o YATServer, incluido a versão original;
- Chama o aplicativo Workload Generator, que cria as cargas descritas, com valores de entrada aleatórios;
- Para cada mutante:
  - Para cada configuração de carga existente:
    - Copia apenas a configuração referente à carga atual;
    - Executa uma instância do Thermometer;
    - Chama o YATClient, passando essa carga;
    - Chama o YATServer;
    - Aguarda o fim da execução do Thermometer e finaliza os processos YATClient e YATServer;

Nessa etapa são gerados todos os dados de referência (medições da execução do programa original para cada carga), como também os dados de teste (medições da execução de cada mutante para cada carga existente).

6.3.3 Uso do DAMICORE

A ferramenta **DAMICORE**, descrita no Capítulo 4, foi utilizada nesse projeto como uma ferramenta externa de agrupamentos, em seu modo de configuração padrão. Em relação aos parâmetros de execução do **DAMICORE**, o único que foi passado foi a biblioteca de compactação a ser utilizada. Nesse caso optamos pela **zlib**, como sendo a mais comum e usual.

A ferramenta **DAMICORE** como agrupador de dados genéricos se mostrou eficaz na comparação de diferentes tipos de dados em trabalhos anteriores, como código-fonte e até binários compilados (PINTO; DELBEM; MONACO, 2018). O **DAMICORE** tem como uso original a classificação de informação por grupos de similaridade, em formatos quaisquer. O seu maior poder está em sua simplicidade de uso, pois quase não precisa de parâmetros, e versatilidade, pois trabalha com dados de tipo qualquer (MEDEIROS, 2016). Neste projeto utilizamos o **DAMICORE** com um objetivo um pouco diferente. O objetivo principal não está em encontrar semelhanças nos perfis de execução e sim de agrupar separadamente as anomalias e assim identificar uma alteração significativa.

Dessa forma, o **Tricorder** não faz uso da informação de agrupamentos gerada pelo **DAMICORE**, como o número de grupos e tamanho de cada um, porém analisa a natureza de cada grupo. Essa análise procura por um tipo específico de grupo, formado apenas por execuções de teste, considerado grupo anômalo. Como saída temos a informação binária de presença ou não de anomalia.

Sendo o **DAMICORE** capaz de agrupar quaisquer tipos de dados, alguns experimentos foram feitos para agrupar as imagens gráficas dos valores amostrados, e não os valores em si. Em nossos experimentos preliminares essa abordagem se mostrou menos eficaz, e isso é discutido no Capítulo 8. Optamos, portanto, por uma abordagem mais tradicional com uso dos dados numéricos em formato texto. Outra abordagem possível também seria o uso de formatos binários de armazenamento de informações de desempenho, como o gerado pelo Microsoft (2017, Performance Monitor), ou outros formatos criados para esse fim. Essas possibilidades são descritas no Capítulo 8.

6.3.4 Agrupamento dos resultados

Finalizada a etapa de geração de dados, os resultados são agrupados e então a existência de anomalias é verificada. Para isso, **Tricorder** executa a ferramenta **DAMICORE**, passando a pasta contendo todas as execuções de referência e de teste. Ao final da execução do **DAMICORE**, aplicamos o critério descrito no Capítulo 5: **existência de agrupamento composto apenas por dados de medição de teste**.

A saída gerada pelo **DAMICORE** é um arquivo texto com os nomes dos arquivos e seus respectivos grupos. Pelo nome do arquivo é possível saber se o mesmo refere-se a uma execução de teste ou referência. O **Tricorder** percorre os grupos afim de identificar um grupo que atenda ao critério. Em caso positivo, então há uma provável presença de anomalia.

6.3.5 Processo de validação da proposta

Afim de validar o **Tricorder**, um *script* também feito em **python**, executou um processo incremental, que simula uma situação real onde partimos de uma base de medições de referência,

e o programa teste está em uso, sendo executado e em seguida analisado, de forma contínua. O procedimento executado pelo *script* segue a descrição de Validação da Proposta, no Capítulo 5. Ele foi desenvolvido para ser capaz de gerar uma base de referência com 30 execuções, de cada uma das configurações, do programa original, com uso do **Thermometer**. Em seguida, de forma automática, o framework **Tricorder** executa um mutante escolhido. A cada final de execução, o dados coletados são adicionados ao conjunto de medições e o **DAMICORE** agrupa todos novamente. E a cada iteração o critério **Tricorder** é aplicado. Isto é feito para todos os mutantes e todas as configurações, e também para o programa original como teste de falso-positivo.

6.4 Considerações finais

Neste capítulo foram descritos os requisitos e a arquitetura de implementação da aplicação benchmark utilizada como objeto teste de avaliação da metodologia **Tricorder**. Trata-se de um sistema de cliente-servidor, **YATClient** e **YATServer**, respectivamente, que trabalham processando um *stream* contínuo de dados. O **YATClient** usa um arquivo de configuração para criar um fluxo de dados, via conexão TCP/IP, para o **YATServer**. Este último utiliza uma biblioteca de processamento, também desenvolvida para o projeto, que interpreta os dados recebidos. Todos os módulos foram implementados na linguagem C++. O código-fonte na íntegra está disponível em (MONTES, 2019a).

Este modelo de arquitetura simula um sistema de telemetria de dados remotos, que utiliza biblioteca de terceiros para interpretar os pacotes de dados recebidos. Nesta biblioteca são inseridos defeitos, cuja detecção via reflexo no desempenho é o foco da pesquisa.

Neste capítulo também foram descritas cada uma das sete categorias de defeitos selecionadas para avaliação de eficácia da metodologia **Tricorder**. Cada categoria foi representada com uma implementação do defeito, que pode ser inserido ou não de acordo com diretivas de compilação. Quatro dos defeitos afetam processamento de dados *BIN* (BCLEAN, MONO, SLEEP, SWAP) e três deles afetam processamento de dados *CSV* (INFINITE, NOBREAK, UNLOCK). Cada defeito corresponde a um mutante da aplicação *YATServer*.

Por fim, foram descritas as etapas da implementação da metodologia **Tricorder**, na forma de uma framework de mesmo nome. Todo o processo, desde a geração da base de referência até a identificação de presença de anomalia, são parte do **Tricorder**. Para agrupar os resultados, o **Tricorder** faz uso da ferramenta **DAMICORE**, utilizada em seu modo padrão. Os códigos-fonte dos *scripts*, assim como do **Thermometer**, estão disponíveis em (MONTES, 2019b). Os resultados da execução do **Tricorder** na avaliação de funcionamento da aplicação **YATServer**, e seus diferente mutantes, podem ser vistos no Capítulo 7.

CAPÍTULO

7

RESULTADOS

Nesse capítulo são apresentados os resultados obtidos da aplicação da metodologia **Tricorder** para testar, em ambiente controlado, a aplicação de *benchmark* **YATServer**, descrita no Capítulo 6. Foram executadas ao todo 8 versões do aplicativo **YATServer** (7 mutantes e a original), com 15 diferentes cargas de trabalho, estes também descritas no Capítulo 6.

No final deste capítulo também são apresentados resultados de um experimento de uso da metodologia **Tricorder** em ambiente real de uma indústria, com entradas reais. Nesse caso, o teste se baseou em um defeito real detectado e consertado. A versão com defeito foi comparada com a aplicação corrigida, utilizando o método automatizado Tricorder, e os resultados foram apresentados no final do capítulo.

7.1 Ambiente de teste

Como mencionado no Capítulo 6, os defeitos se encontram na biblioteca de processamento, sendo cada defeito unicamente responsável por uma versão mutante da biblioteca. Esta biblioteca, ao ser compilada junto ao servidor (**YATServer**), produz uma versão mutante do servidor. O objeto de análise dos testes é o servidor, que se comunica com um cliente (**YATClient**). Este último envia dados na rede de acordo com as entradas pré-definidas. Ao conjunto de entradas mais o arquivo de configuração correspondente chamamos de uma carga de trabalho (*workload*).

Os mutantes criados, em conjunto com o programa original de referência, formam o conjunto de objetos de teste. Cada um é executado separadamente, com cada uma das 15 diferentes cargas de trabalho. Isto visa reproduzir os diferentes cenários em que as aplicações podem estar sujeitas em condições reais.

Cada mutante foi definido de forma a afetar a execução de apenas um dos 2 tipos de entrada, baseado no fluxo de execução da aplicação. Esta escolha foi feita para limitar o efeito da falha inserida e também para que existam entradas onde o defeito não causa falha, por não ser

estimulado pela execução. Esta característica representa melhor as situações de falhas reais e torna mais fácil o controle dos experimentos.

Chamamos de base de referência o conjunto de medições de execuções do programa original que consideramos como amostras de medições corretas. Assim como as medições de teste, estas são medições de intervalo fixo, definido em 30 segundos, para cada uma das 15 cargas de trabalho existentes. Essas medições foram feitas usando a ferramenta **Thermometer**, descrita no Capítulo 6. Para cada carga foram feitas 30 execuções do **YATServer** original, montando assim uma base de conhecimento para comparação. Todas as medições ficam misturadas em uma mesma pasta, essa chamada de base de referência.

Como mencionado no Capítulo 6, a aplicação de *benchmark* permite configuração de tipos de entrada e de cargas de trabalho. Para definição das cargas, utilizamos a nomenclatura:

- Tipo da aplicação:
  - R: referência; ou
  - D: programa com defeito;
- Tipo de entrada:
  - Entrada **BIN**: "BIN" ou "B";
  - Entrada **CSV**: "CSV" ou "C"; ou
  - Entrada mista: os 2 símbolos estão presentes.
- Tipo de carga:
  - L(*light*): carga de trabalho leve;
  - M(*medium*): carga de trabalho intermediária; ou
  - H(*high*): carga de trabalho grande;
- Numeração: Variacão, de 1 a 3, nos atributos da entrada, em uma mesma categoria de carga

Criamos, portanto, para as entradas simples tipo CSV, 3 cargas respectivamente, chamadas de CH1 (CSV high), CM1 (CSV medium) e CL1 (CSV low). Da mesma forma, para entrada simples BIN, as cargas BH1 (BIN high), BM1 (BIN medium) e BL1 (BIN low). No total são 6 cargas de trabalho simples.

Para as cargas mistas, criamos 3 definições de carga para cada categoria H, M ou L. Desta forma, foram definidas CBH1, CBH2, CBH3, CBM1, CBM2, CBM3, CBL1, CBL2 e CBL3, totalizando 9 cargas de trabalho mistas.

Ao todo são 15 cargas de trabalho, utilizadas na avaliação da metodologia proposta. O arquivo de configuração com todas as cargas mencionadas está no Apêndice C.

7.2 Plano de teste

Seguindo a metodologia **Tricorder**, foram executados experimentos de aplicação da técnica de detecção de anomalias para cada um dos 7 defeitos descritos no Capítulo 6. Cada um destes representa uma categoria de defeitos que podem causar variações significativas no uso de recursos, ao mesmo tempo que podem não afetar a saída esperada do programa.

As execuções posteriores à criação da base de referência, sejam do programa original ou de um mutante, chamamos de execuções de teste. Estas medições são diferenciadas pela nomenclatura dos arquivos, e assim podemos identificá-las na formação de agrupamentos. Contudo, para o **DAMICORE**, medições de referência ou teste são todas medições de um mesmo tipo, e como o **DAMICORE** não leva em consideração o nome do arquivo no processamento de comparação, mas apenas seu conteúdo, ele não sabe (nem considera) a categoria (teste ou referência) ao agrupar as medições.

Em cada configuração de teste, usamos sempre a mesma base fixa, com as 30 execuções de referência para cada umas das 15 cargas. E para geração das medições de teste, usamos apenas um mutante e uma carga. Diferentes cargas em conjunto podem, dependendo da carga, melhorar ou piorar os resultados, a medida que os defeitos se tornem mais ou menos visíveis em cada uma. Para tornar o teste neutro decidimos por executar testes separados para cada carga, e verificar a eficácia do método separadamente.

Assim, em cada análise, é agrupada uma pasta contendo um conjunto fixo de medições de referência e uma ou mais medições de teste, para uma carga expecífica, e como saída temos um ou mais agrupamentos desses arquivos, formados por similaridade.

O experimento seguiu estas especificações:

- Execução automática e quase simultânea do **YATServer** e **YATClient**, com início automático da comunicação;
- No momento da execução é passado para aplicação **YATClient** o arquivo de configuração de carga;
- Medição dos primeiros 30 segundos da execução, onde sempre o programa roda por ao menos 30 segundos, usando o **Thermometer**;
- **Thermometer** usa medição periódica dos usos de CPU, Memória e IO (leituras e escritas no disco, no intervalo) do processo **YATServer**;
- Frequência de amostragem fixa em 10 Hz; e
- Saídas em formato **CSV** (texto) e **PNG** (imagem).

O experimento também considerou estas condições:

- Referência fixa de 30 execuções do programa original, para cada uma das 15 cargas definidas;
- Cada categoria de defeito é agrupada separadamente das outras, sempre em conjunto com a base de referência;
- A cada execução do mutante, o resultado é agrupado em conjunto às execuções anteriores e à base;
- Isso é repetido até que se tenham 30 execuções do mutante ou até que uma anomalia seja detectada.

Nesse cenário, a metodologia **Tricorder** aplicada na identificação da presença de defeito, pode ser descrita pelos seguintes passos:

1. Criação da base de execuções do programa referência (30 execuções para cada carga);
2. Inicio das 30 execuções do programa teste, com uma carga definida;
3. A cada nova execução, novos agrupamentos são formados (podem mudar radicalmente entre uma e outra);
4. Caso seja criado um agrupamento formado apenas por execuções do programa teste, representa possível presença de anomalia;
5. Esse processo continua até que seja encontrada anomalia ou até chegar a 30ª execução do programa teste; e
6. No caso de haver apenas grupos mistos, com presença de amostras de teste e de referência, é considerado funcionamento normal.

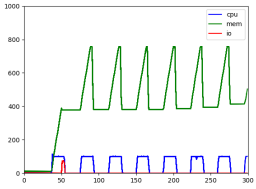
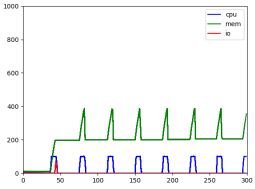
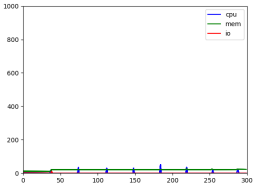
Para aplicar a metodologia de forma contínua, poder-se-ia usar, por exemplo, a mesma base fixa de referência, e uma janela deslizante das últimas 30 execuções do programa teste.

7.3 Medição de desempenho

Para facilitar o entendimento do efeito causado pela inserção de defeitos, em cada um dos cenários de carga, foram traçados gráficos ponderados de CPU, Memória e IO, tanto da execução original como para cada execução de mutante. As cargas estão separadas em 2 grupos: execuções simples, **BIN** ou **CSV**, e mistas onde há entrada simultânea dos 2 tipos. As Figuras 12 e 13 contêm gráficos de amostras de execuções, cujo comportamento tomamos como referência. Entre colchetes está o tipo de carga utilizado. É possível observar a variação do uso de recursos com a variação da entrada. Todas essas medidas de desempenho são consideradas normais, tomadas

como referência. Dizemos amostras, pois o uso de recursos pode variar de uma execução para outra, mesmo estando sujeito à mesma carga.

Para cada carga, como descrito no Capítulo 5, foram feitas 30 execuções e estas nem sempre são iguais, apesar de semelhantes. Contudo ficaria inviável apresentar graficamente todas as medições feitas para cada caso, então estabelecemos o padrão de usar a primeira execução como referência gráfica. Portanto, todos os gráficos utilizados se basearam nas medições da 1ª execução, para todos os mutantes e todas as cargas.

[RCH1]  [RCM1]  [RCL1] 

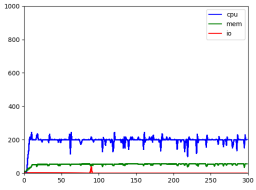
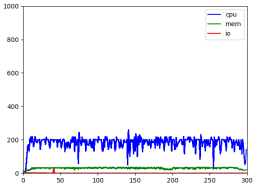
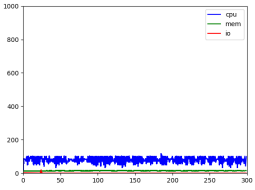
[RBH1]  [RBM1]  [RBL1] 

Figura 12 – YATServer de referência nas cargas de trabalho simples

Analisando a Figura 12, em RCH1, RCM1 e RCL1 observamos o efeito do processamento de entradas CSV. Cada arquivo CSV recebido e processado reflete, no uso de memória em verde, como um pico da onda triangular, e na CPU em azul, como um máximo de onda quadrada. A amplitude de cada um depende dos valores de carga estabelecidos e da capacidade da máquina utilizada.

Ainda na Figura 12, em RBH1, RBM1 e RBL1 vemos o efeito do processamento de entradas BIN. O fluxo contínuo de entradas binárias causa um uso médio contínuo de CPU, com uma variação de constante proporcionalmente menor, semelhante a um ruído, e um uso de memória inicialmente crescente depois estável e contínuo..

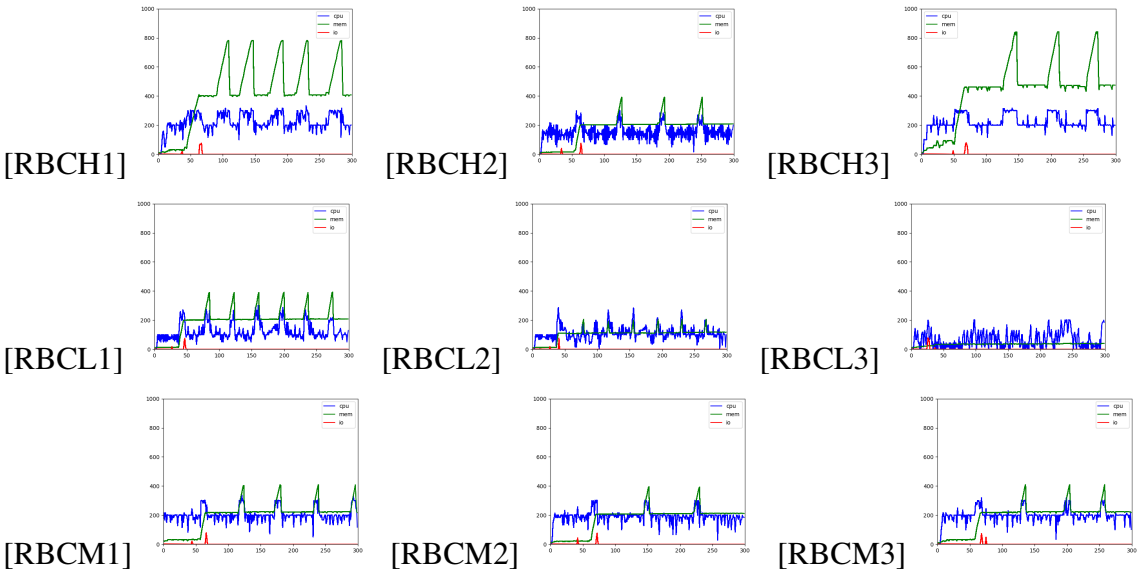


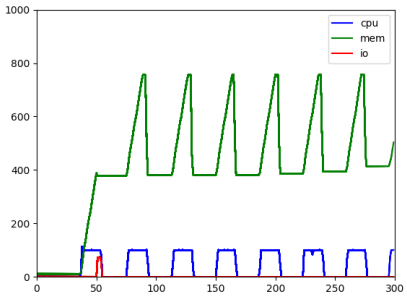
Figura 13 – YATServer de referência nas cargas de trabalho mistas

Na Figura 13 vemos o efeito da mistura de entradas nas medições, que deixam de ser tão claras. Ainda é possível ver os picos de uso de memória nas cargas H e M, e com menor clareza nas cargas L, e as ondas quadradas de CPU ficam bem menos nítidas. Ambas as ondas podem nem ser visíveis, como em RBCL3.

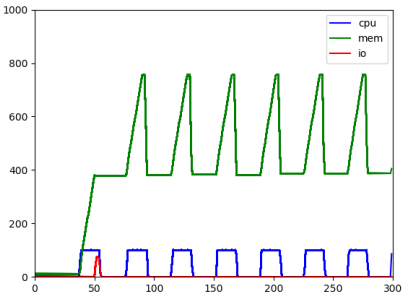
Para melhorar a clareza das leituras, e facilitar a comparação, todas os gráficos de referência de entrada simples: RCH1, RCM1, RCL1, RBH1, RBM1, e RBL1, e todos os gráficos de referência de entradas mistas: RBCH1, RBCH2, RBCH3, RBCM1, RBCM2, RBCM3, RBCL1, RBCL2 e RBCL3 foram repetidos ao lados de suas respectivas execuções com defeitos, para todos os mutantes. Lembramos, portanto, que são os mesmos gráficos, repetidos por motivo de clareza.

7.3.1 BCLEAN

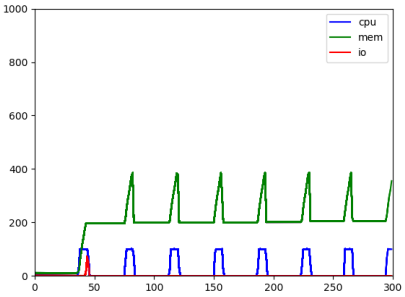
As Figuras 14, 15, 16, 17 e 18 são gráficos das medições de uso de recursos de amostras de execuções do mutante BCLEAN. Nelas podemos ver o efeito da falha de uso crescente de memória, graficamente representados em verde. Quando a falha ocorre, o uso de memória é predominantemente crescente. Não dizemos que são execuções com falha pois nem todas as entradas geraram falha.



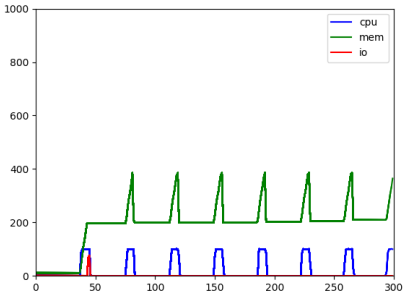
[RCH1]



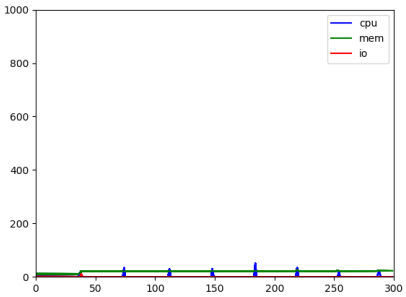
[DCH1]



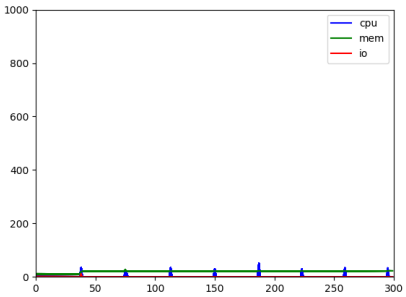
[RCM1]



[DCM1]



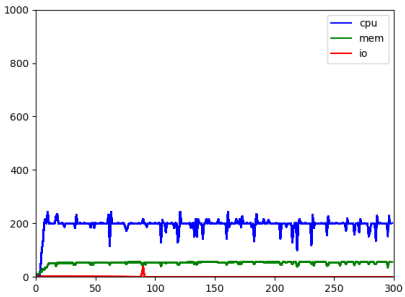
[RCL1]



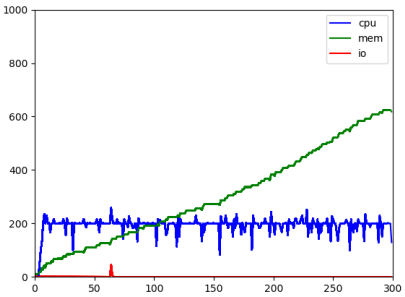
[DCL1]

Figura 14 – BCLEAN: entradas simples, tipo CSV

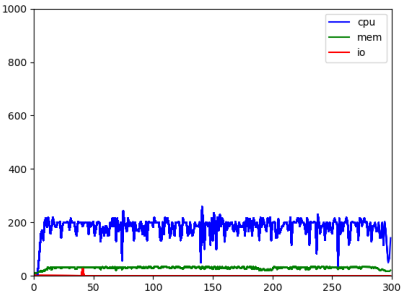
Na Figura 14 é possível observar a semelhança entre as as medições do mutante e da referência. Isso se deve ao fato de que o defeito só executado na presença de entrada tipo *BIN*, e com entradas puramente *CSV*, o funcionamento ocorre normal.



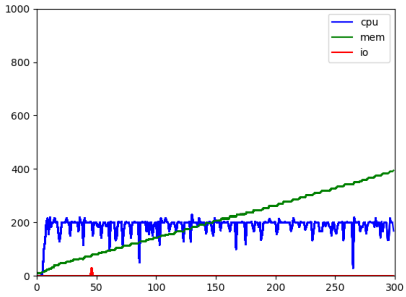
[RCH1]



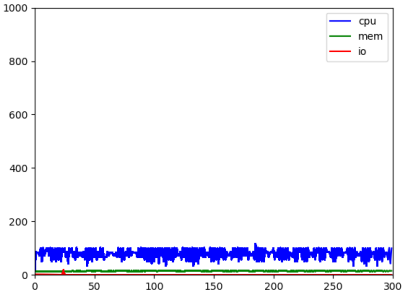
[DBH1]



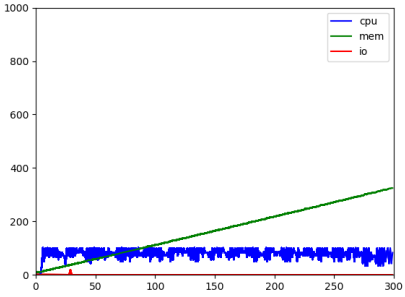
[RBM1]



[DBM1]



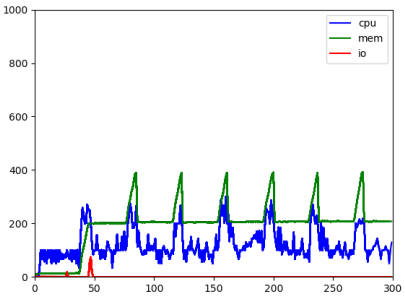
[RBL1]



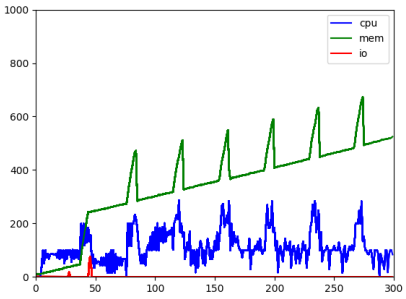
[DBL1]

Figura 15 – BCLEAN: entradas simples, tipo BIN

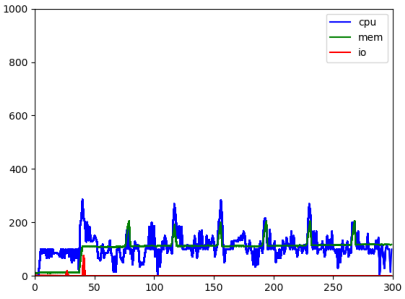
Na Figura 15 é evidente o efeito da falha por excesso de uso de memória, pela linha verde contínua e crescente, nas 3 cargas de trabalho.



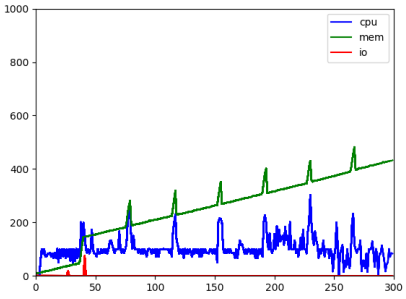
[RBCL1]



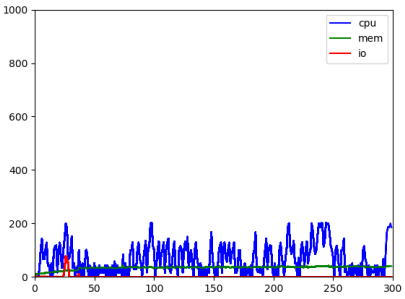
[DBCL1]



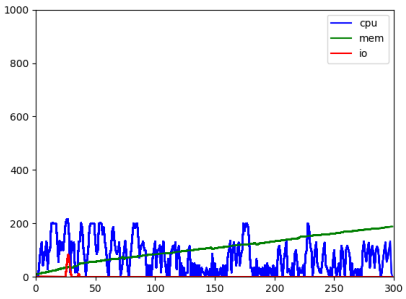
[RBCL2]



[DBCL2]

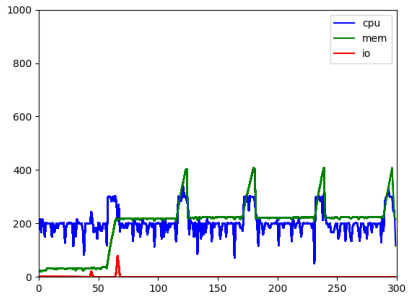


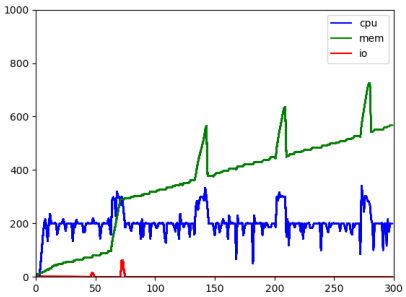
[RBCL3]

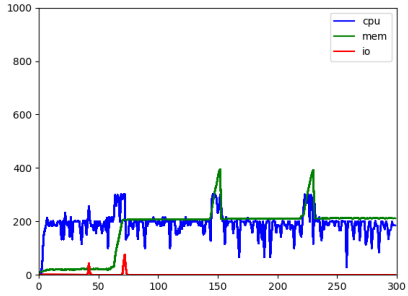


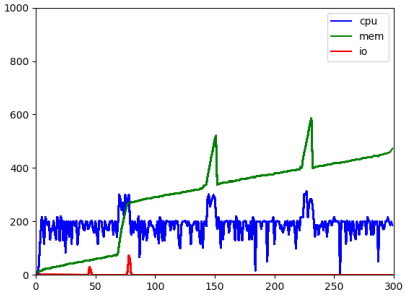
[DBCL3]

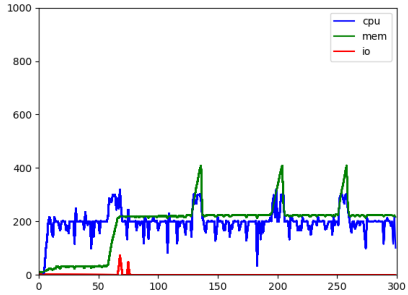
Figura 16 – BCLEAN: entradas mistas, cargas L

[RBCM1] 

[DBCM1] 

[RBCM2] 

[DBCM2] 

[RBCM3] 

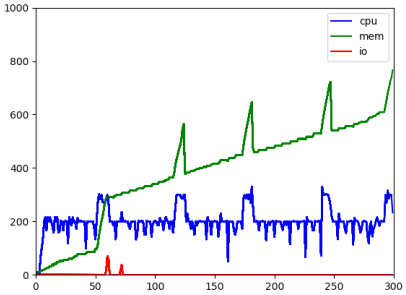
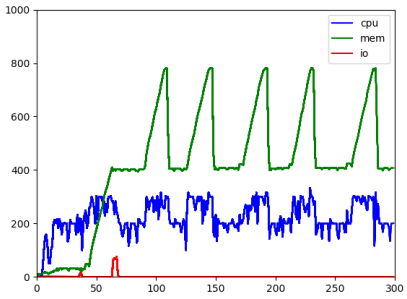
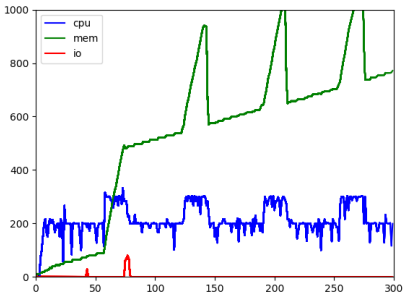
[DBCM3] 

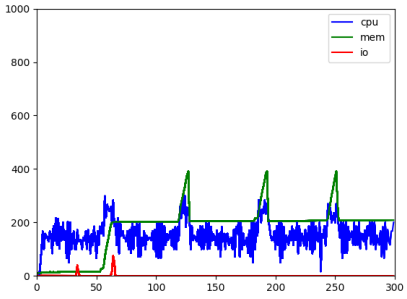
Figura 17 – BCLEAN: entradas mistas, cargas M



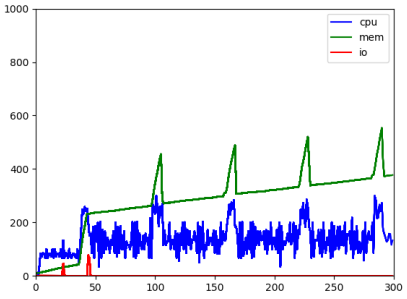
[RBCH1]



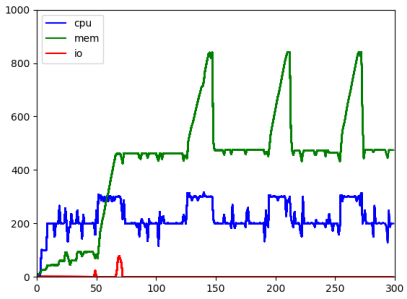
[DBCH1]



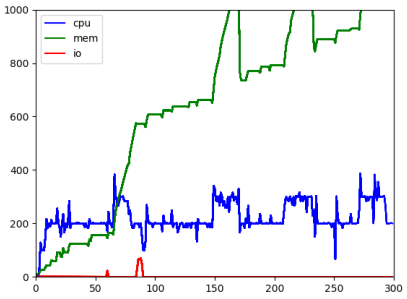
[RBCH2]



[DBCH2]



[RBCH3]



[DBCH3]

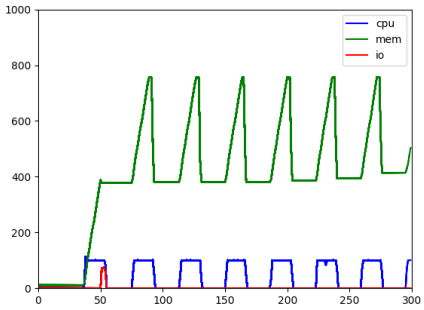
Figura 18 – BCLEAN: entradas mistas, cargas H

Nas Figuras 16, 17 e 18 observamos o efeito que os dois tipos de entrada misturados causam no desempenho. Ainda é possível verificar o crescimento da linha verde, porém não é mais contínuo como era com entrada simples. Isso torna mais desafiador o processo de identificação da anomalia.

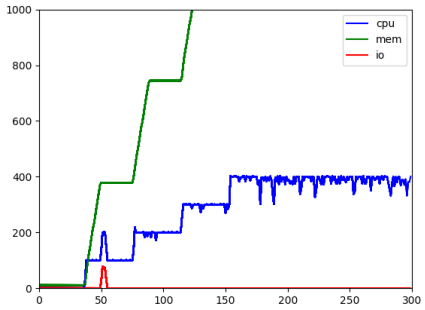
7.3.2 INFINITE

Nas Figuras 19, 20, 21, 22 e 23 podemos observar amostras de medições de execuções do mutante INFINITE, para diferentes entradas.

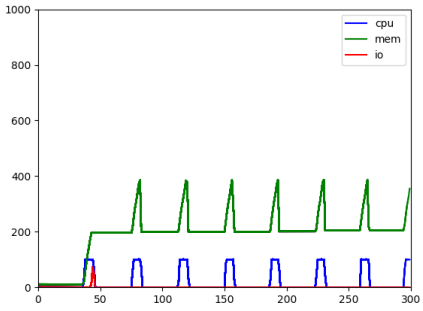
A falha deste mutante, quando ocorre, causa impacto tanto no uso de CPU (em azul) como de uso de memória (em verde). Ao compararmos as execuções DCH1 e DCL1 da Figura 19 fica nítido como o grau desse impacto no desempenho varia com a carga de trabalho. Em ambos a alteração no uso de CPU é semelhante, enquanto o uso de memória, ainda que crescente, é significativamente diferente.



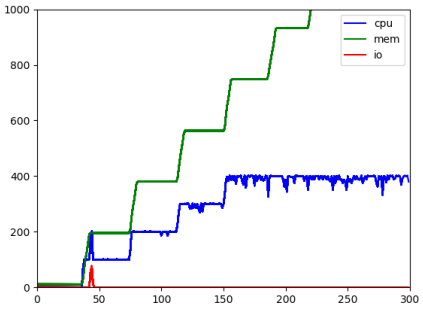
[RCH1]



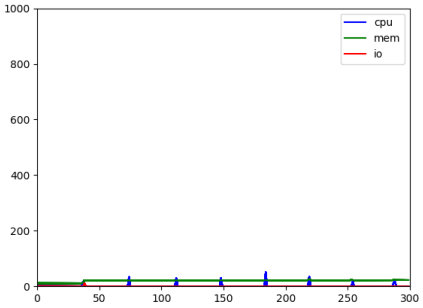
[DCH1]



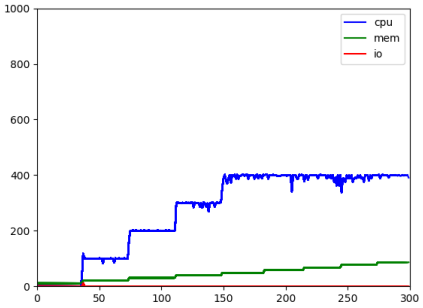
[RCM1]



[DCM1]



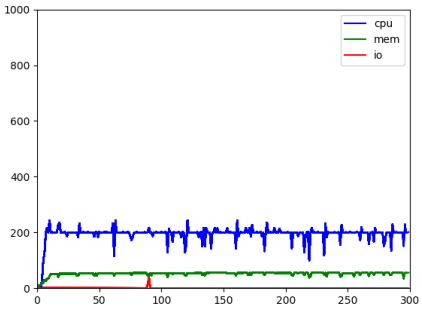
[RCL1]



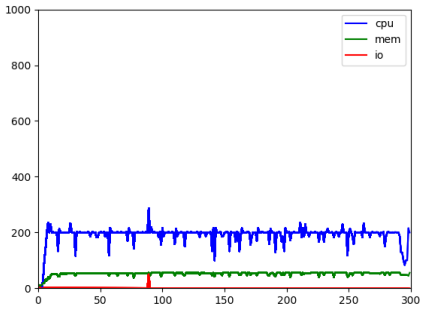
[DCL1]

Figura 19 – INFINITE: entradas simples, tipo CSV

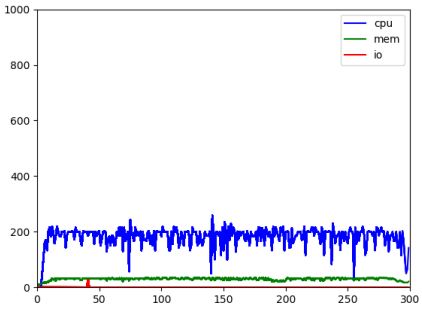
Na Figura 20 verificam-se execuções que envolvem defeito que não causa falha, e portanto, o desempenho é semelhante às medições de referência.



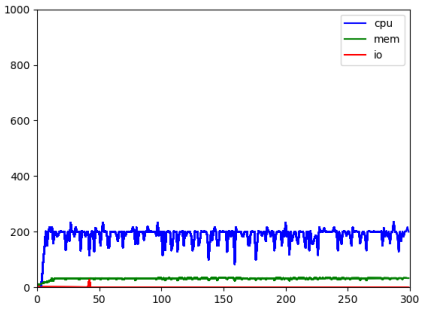
[RBH1]



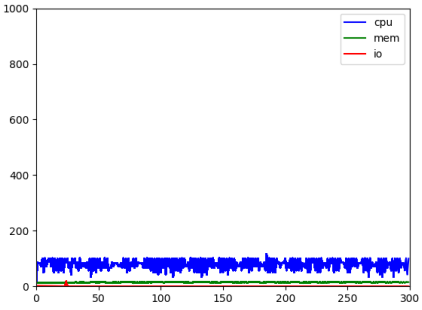
[DBH1]



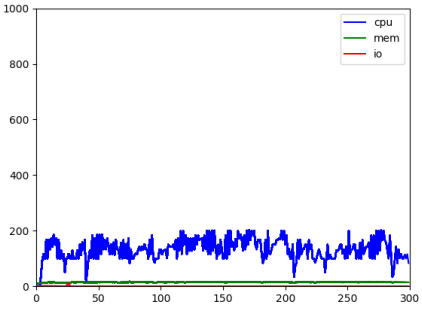
[RBM1]



[DBM1]



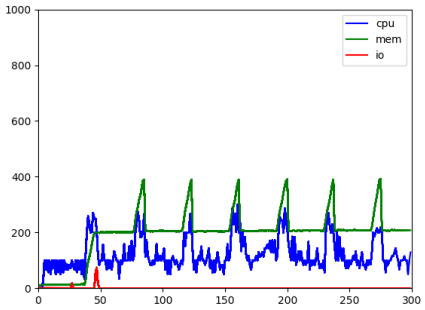
[RBL1]



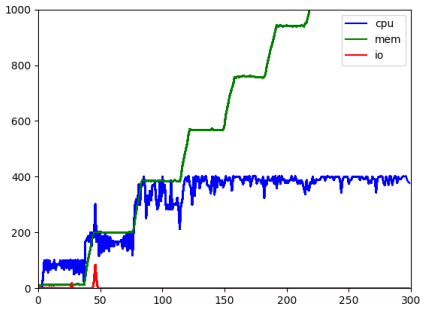
[DBL1]

Figura 20 – INFINITE: entradas simples, tipo BIN

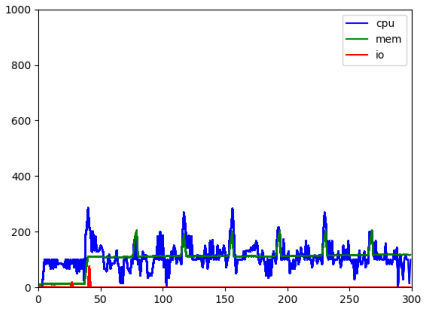
As Figuras 21, 22 e 23 são gráficos de medições com falha. Em todas é possivel observar o efeito da falha no desempenho, em maior ou menor grau.



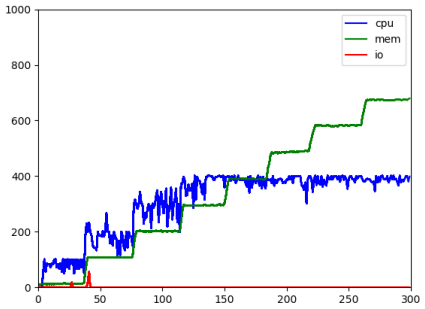
[RBCL1]



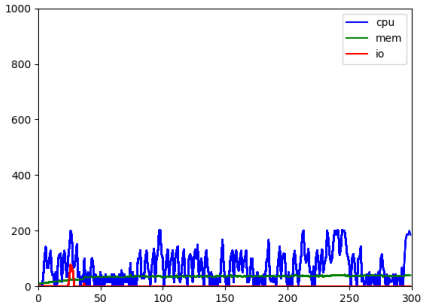
[DBCL1]



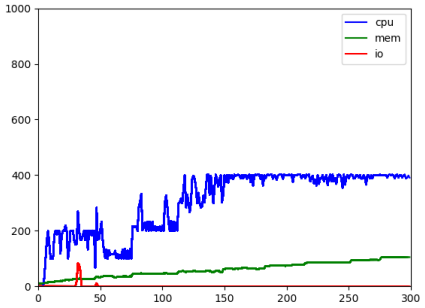
[RBCL2]



[DBCL2]

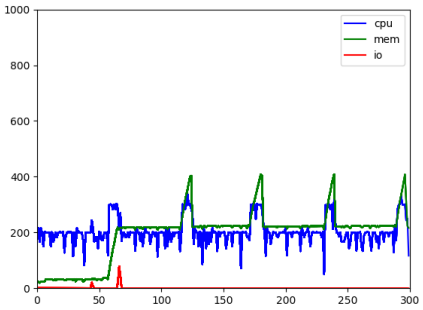


[RBCL3]

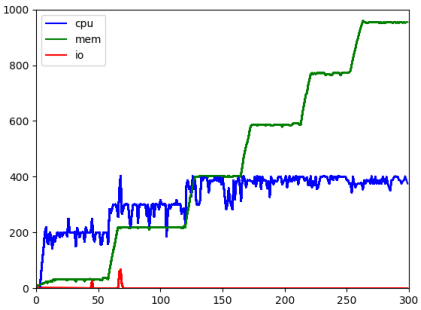


[DBCL3]

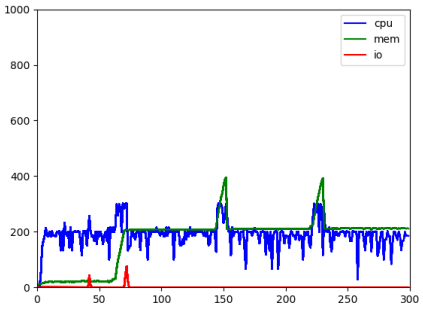
Figura 21 – INFINITE: entradas simples, cargas L



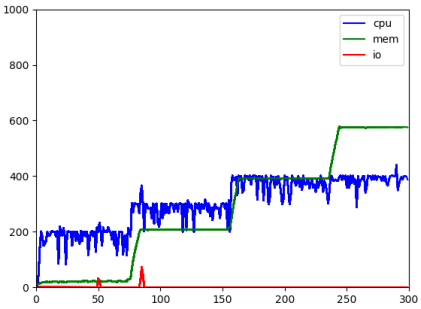
[RBCM1]



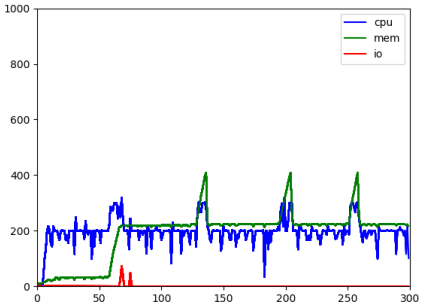
[DBCM1]



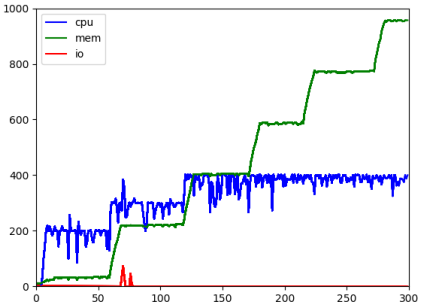
[RBCM2]



[DBCM2]

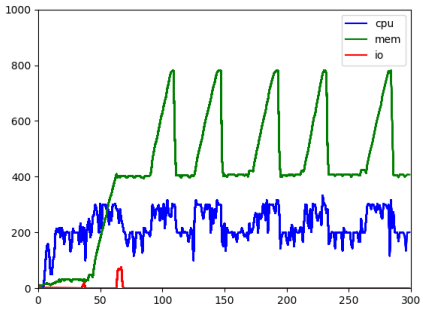


[RBCM3]

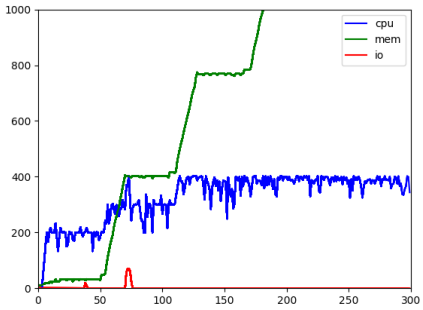


[DBCM3]

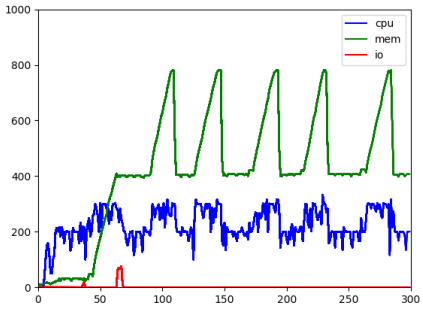
Figura 22 – INFINITE: entradas mistas, cargas M



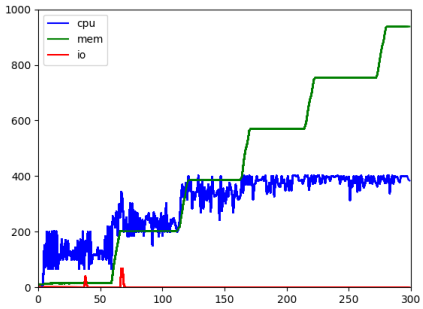
[RBCH1]



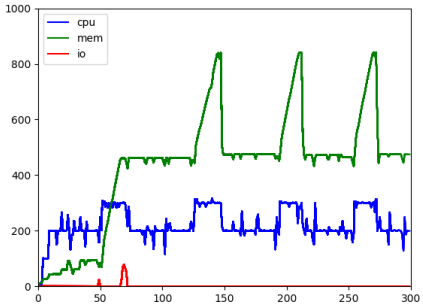
[DBCH1]



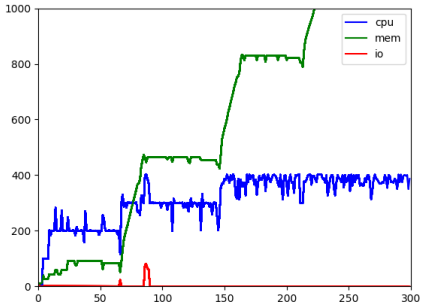
[RBCH2]



[DBCH2]



[RBCH3]

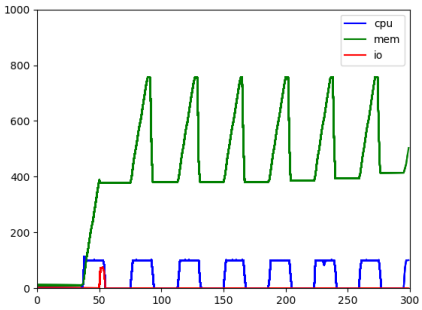


[DBCH3]

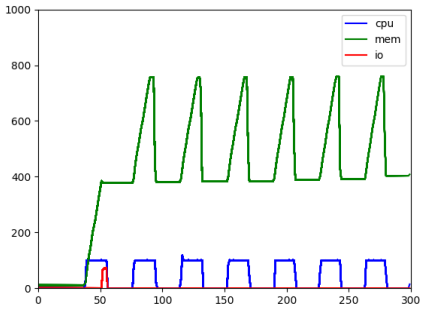
Figura 23 – INFINITE: entradas simples, cargas H

7.3.3 MONO

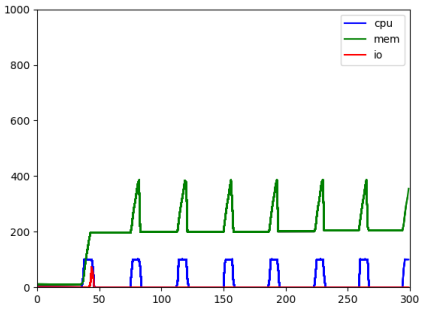
As Figuras 24, 25, 26, 27 e 28 são gráficos de amostras do defeito MONO. O impacto da falta de paralelismo pode ser visto como certo atraso nos tempos de execução e menor uso máximo de CPU.



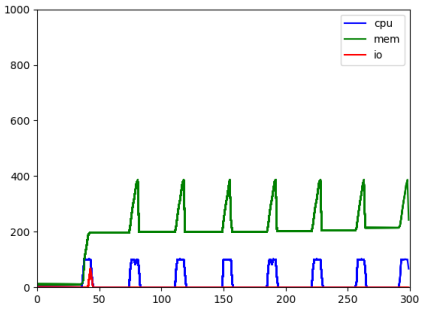
[RCH1]



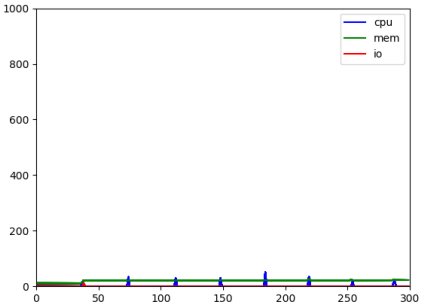
[DCH1]



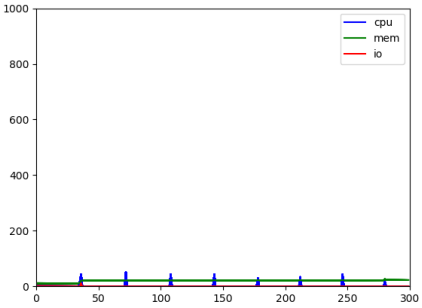
[RCM1]



[DCM1]

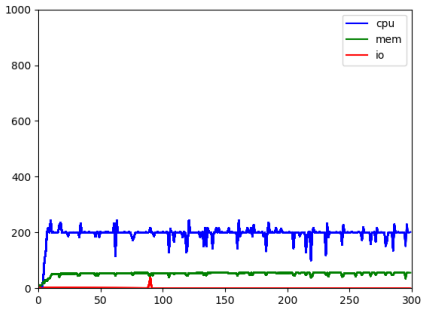


[RCL1]

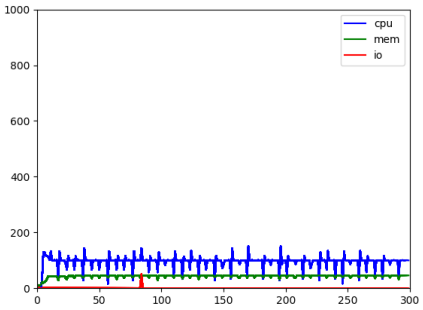


[DCL1]

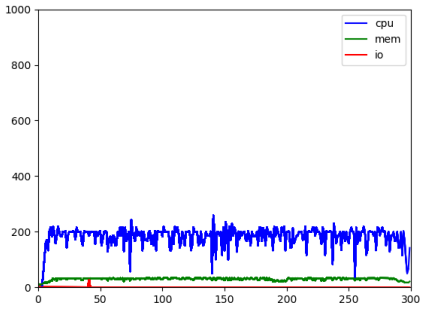
Figura 24 – MONO: entradas simples, tipo CSV



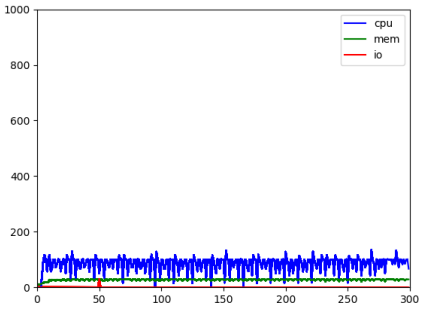
[RBH1]



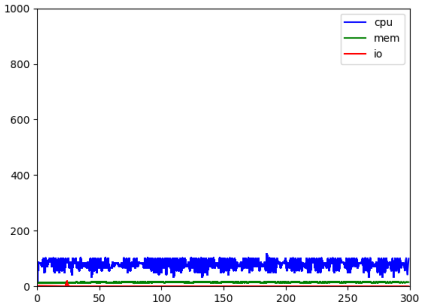
[DBH1]



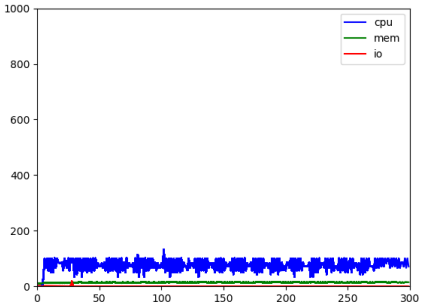
[RBM1]



[DBM1]

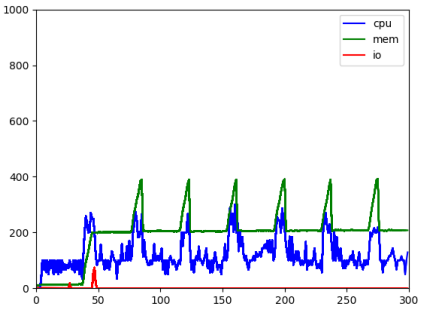


[RBL1]

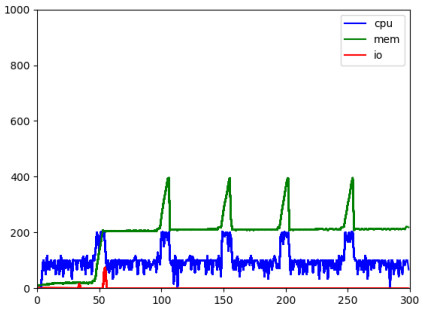


[DBL1]

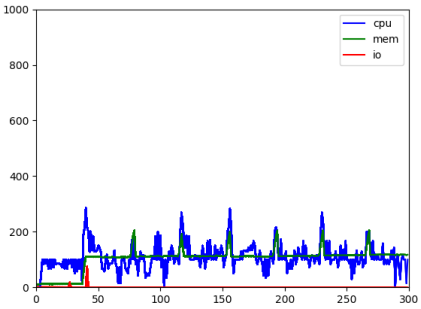
Figura 25 – MONO: entradas simples, tipo BIN



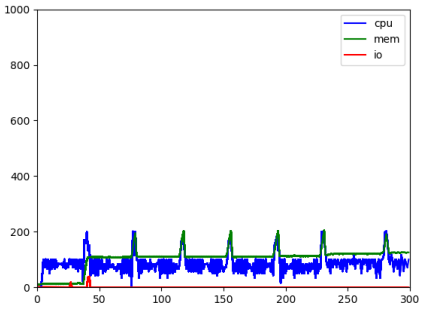
[RBCL1]



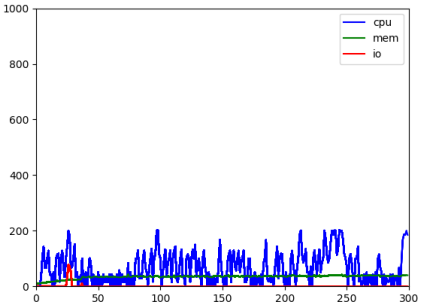
[DBCL1]



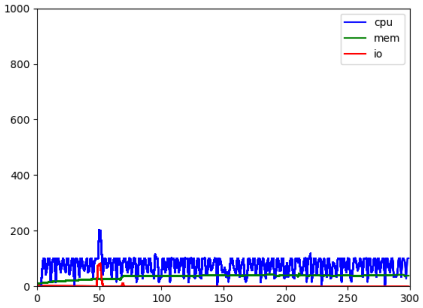
[RBCL2]



[DBCL2]

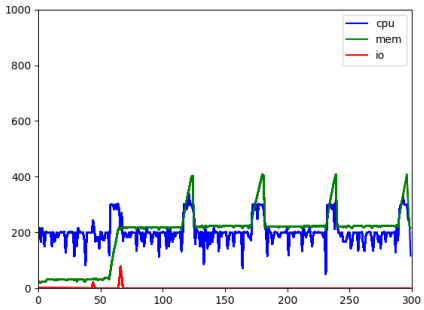


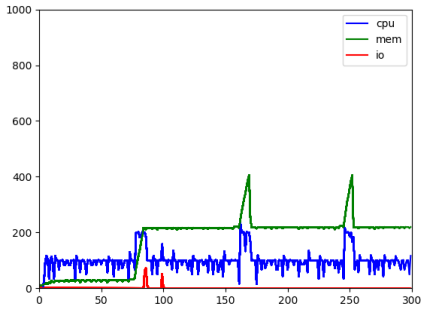
[RBCL3]

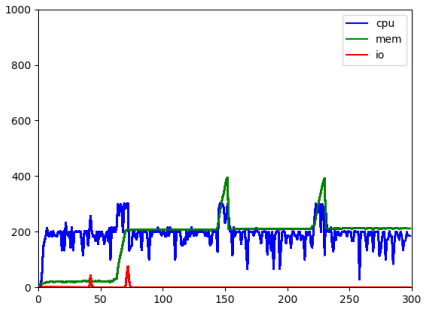


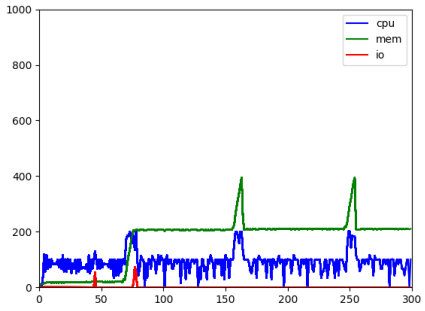
[DBCL3]

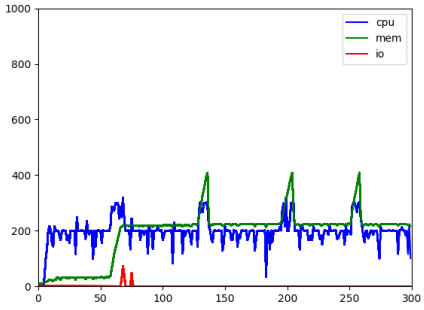
Figura 26 – MONO: entradas mistas, cargas L

[RBCM1] 

[DBCM1] 

[RBCM2] 

[DBCM2] 

[RBCM3] 

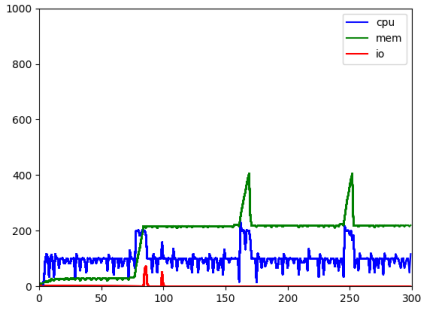
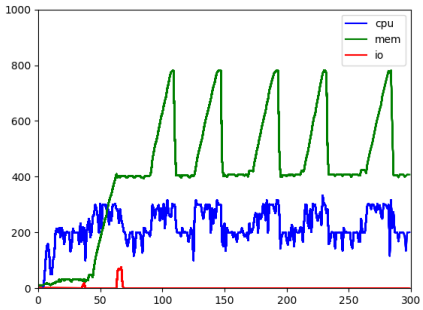
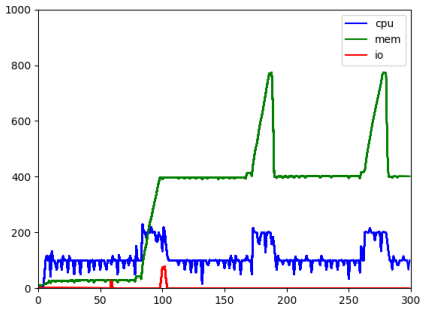
[DBCM3] 

Figura 27 – MONO: entradas mistas, cargas M

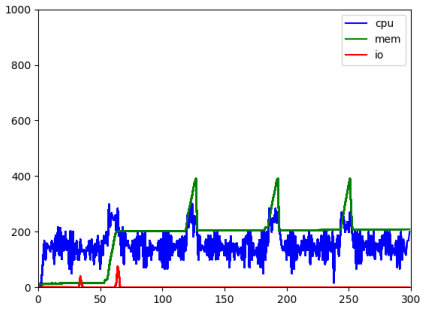
Tomando como exemplo a Figura 28, vemos o efeito do defeito MONO na execução. Havendo menos capacidade de processamento, menos arquivos CSV são processados (picos em verde), e a taxa máxima de uso de CPU fica limitada pela metade (não ultrapassa o valor 100).



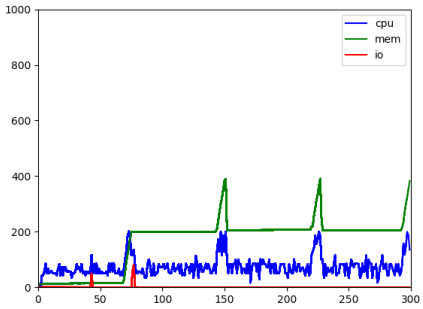
[RBCH1]



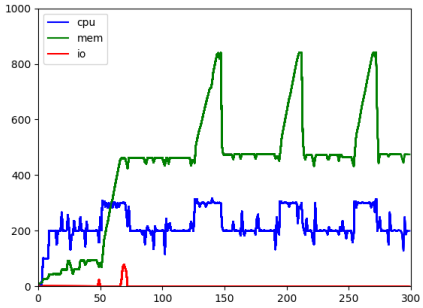
[DBCH1]



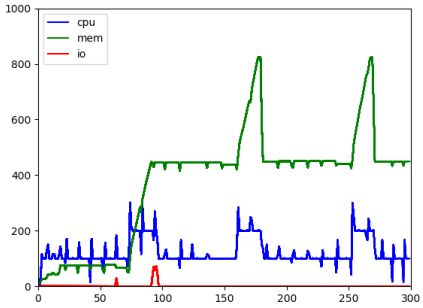
[RBCH2]



[DBCH2]



[RBCH3]

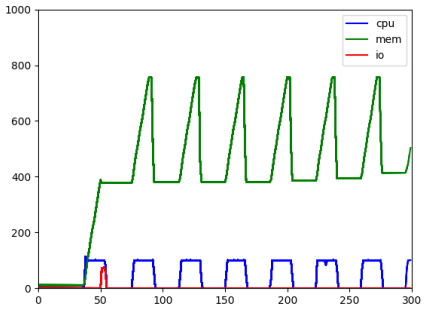


[DBCH3]

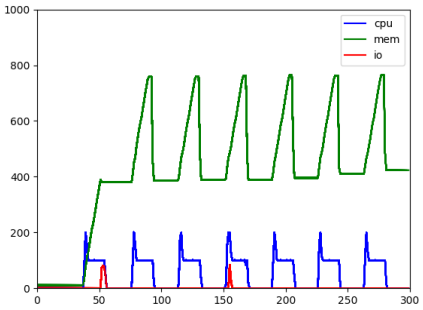
Figura 28 – MONO: entradas mistas, cargas H

7.3.4 NOBREAK

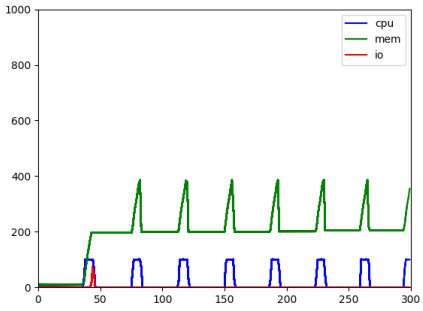
As Figuras 29, 30, 31, 32 e 33 são gráficos de amostras do mutante NOBREAK. Este defeito causa uma execução desnecessária de pacotes **CSV** como se fossem do tipo **BIN**. Ele é mais facilmente visível com entradas puramente **CSV**, sendo um pequeno pico de uso de CPU na borda de subida das ondas quadradas em azul. Para entradas mistas esse efeito é menos visível, sendo confundido com o ruído normal de variação de uso de CPU.



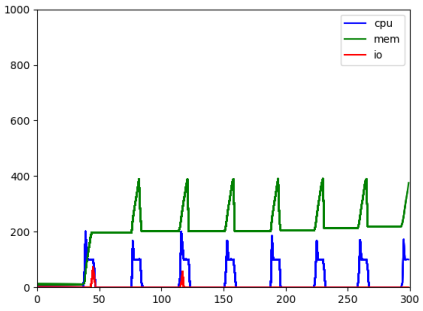
[RCH1]



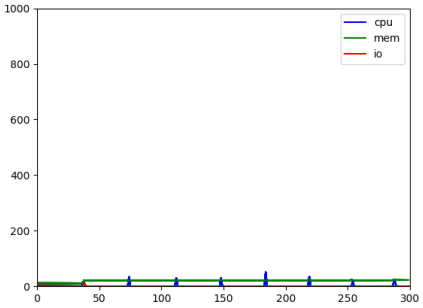
[DCH1]



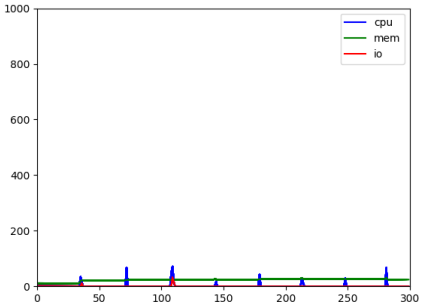
[RCM1]



[DCM1]

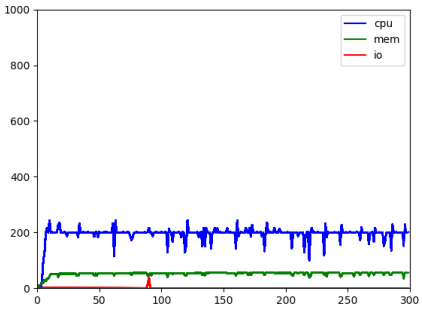


[RCL1]

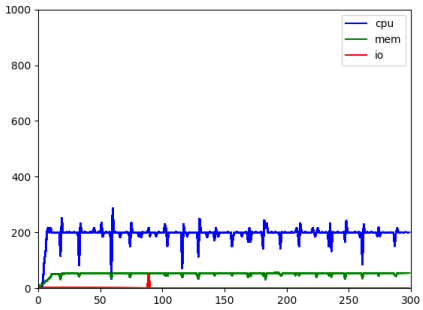


[DCL1]

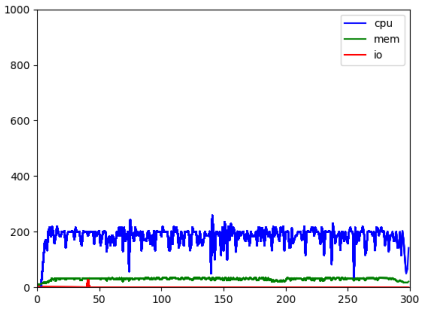
Figura 29 – NOBREAK: entradas simples, tipo CSV



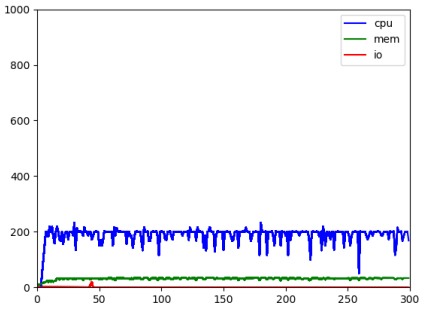
[RBH1]



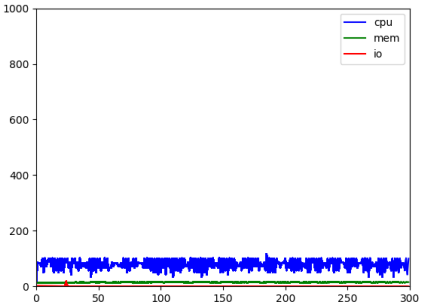
[DBH1]



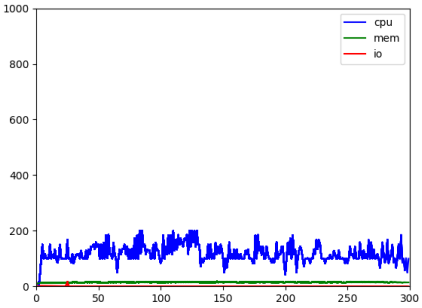
[RBM1]



[DBM1]

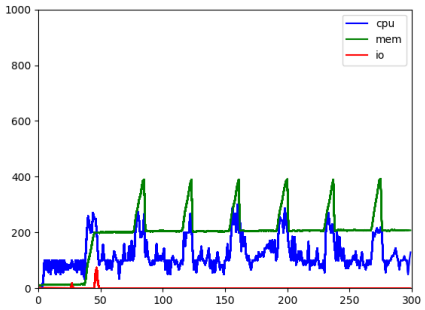


[RBL1]

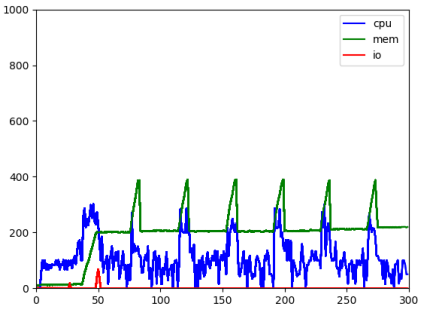


[DBL1]

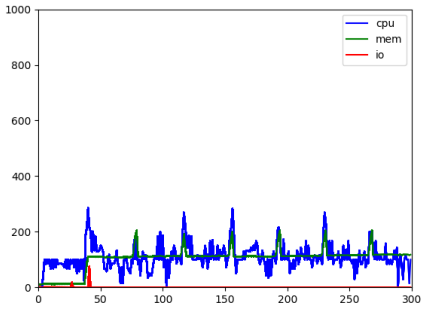
Figura 30 – NOBREAK: entradas simples, tipo BIN



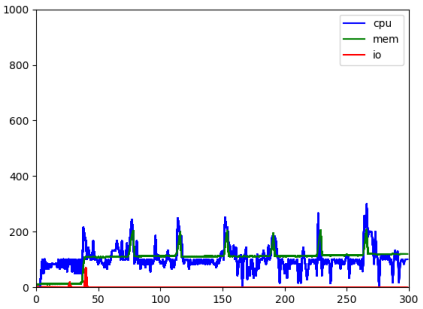
[RBCL1]



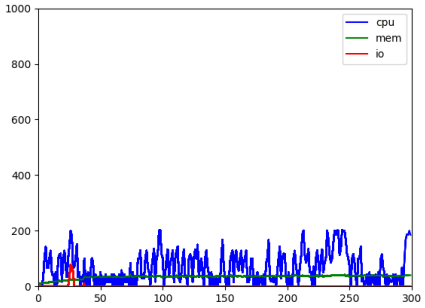
[DBCL1]



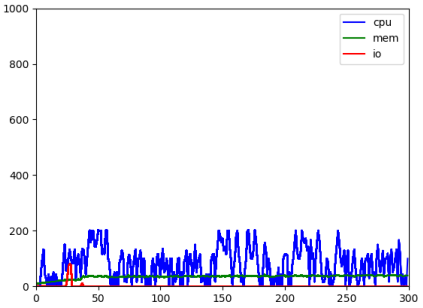
[RBCL2]



[DBCL2]

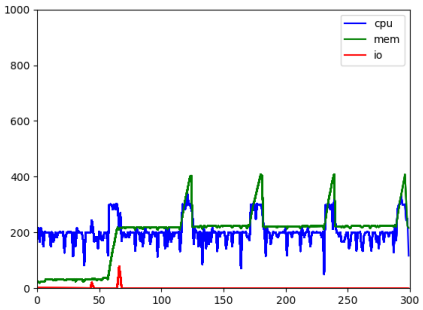


[RBCL3]

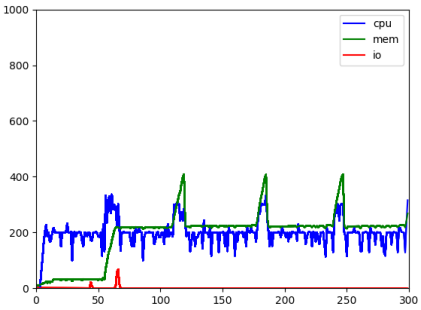


[DBCL3]

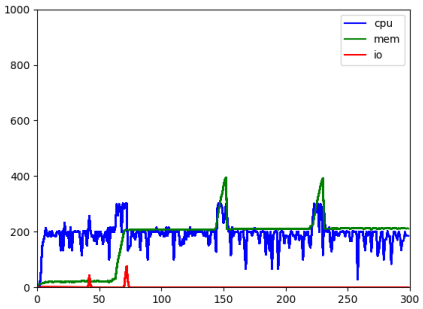
Figura 31 – NOBREAK: entradas mistas, cargas L



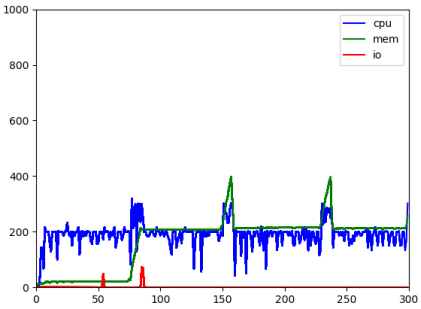
[RBCM1]



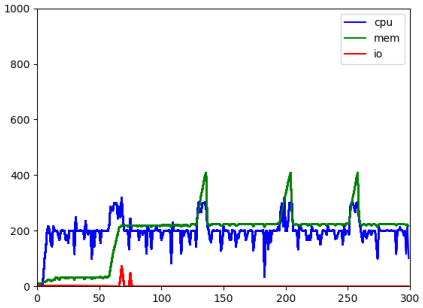
[DBCM1]



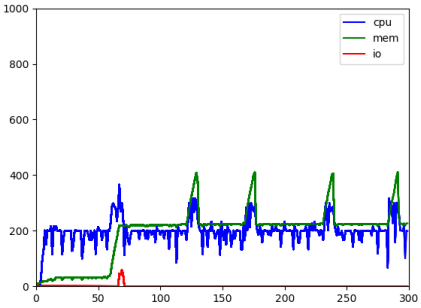
[RBCM2]



[DBCM2]

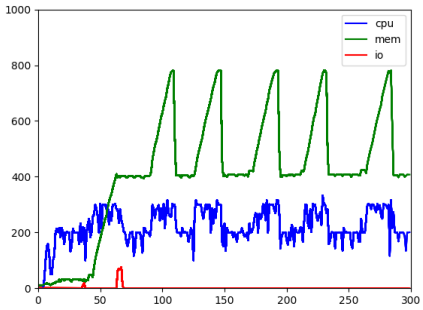


[RBCM3]

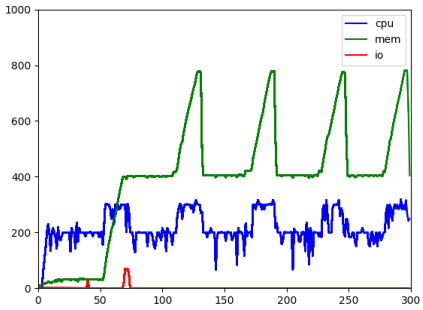


[DBCM3]

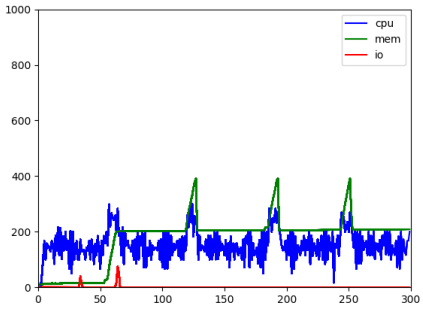
Figura 32 – NOBREAK: entradas mistas, cargas M



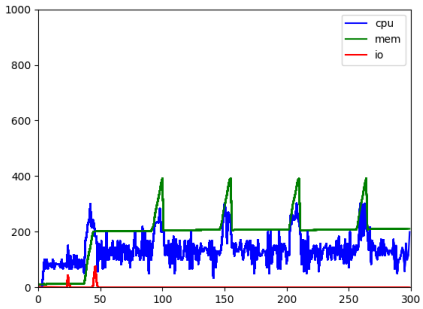
[RBCH1]



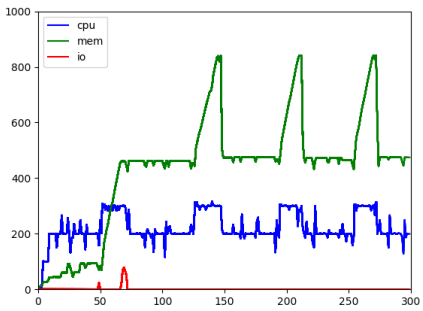
[DBCH1]



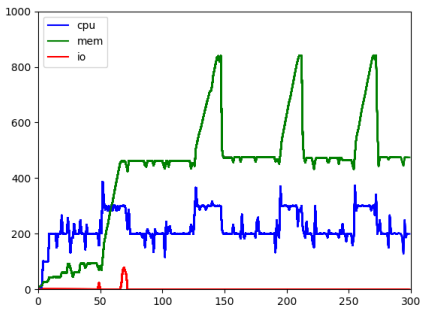
[RBCH2]



[DBCH2]



[RBCH3]

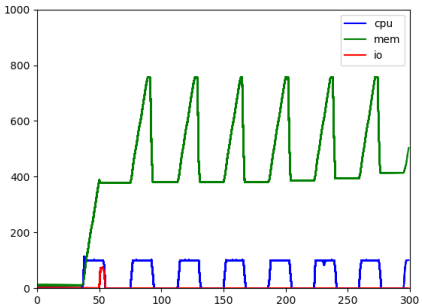


[DBCH3]

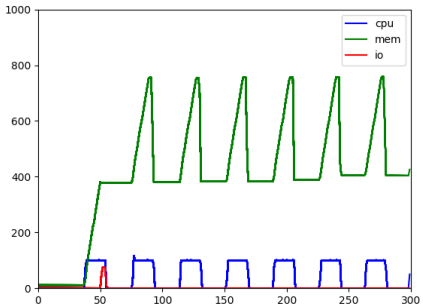
Figura 33 – NOBREAK: entradas mistas, cargas H

7.3.5 SLEEP

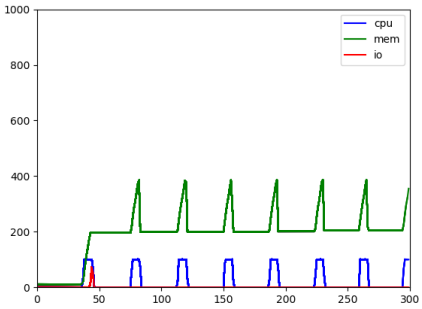
As Figuras 34, 35, 36, 37 e 38 são gráficos de amostras do mutante SLEEP. Seu efeito pode ser descrito como uma menor frequência de variação de CPU, além de um atraso geral na execução. É um defeito cuja falha é difícil de identificar, pois facilmente se confunde com a execução normal da aplicação.



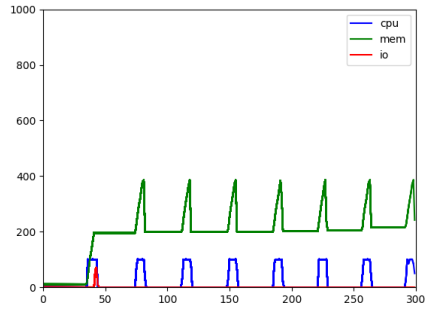
[RCH1]



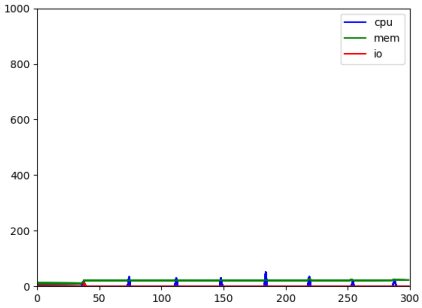
[DCH1]



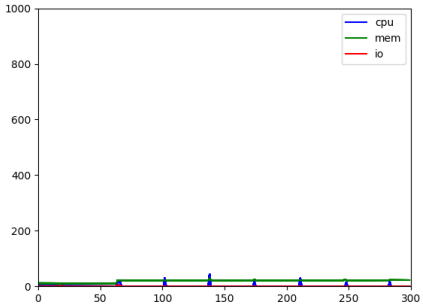
[RCH1]



[DCM1]

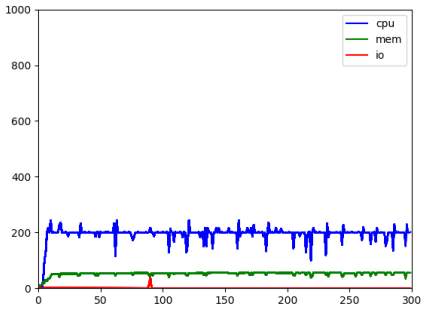


[RCH1]

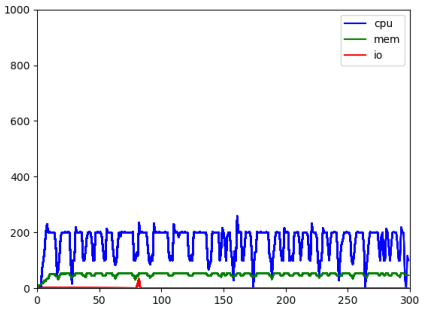


[DCL1]

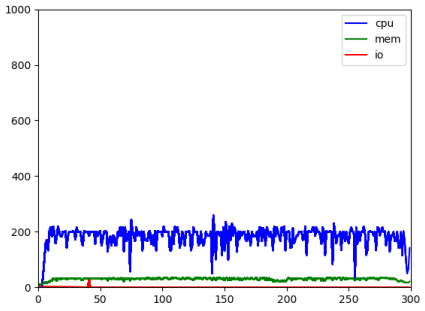
Figura 34 – SLEEP: entradas simples, tipo CSV



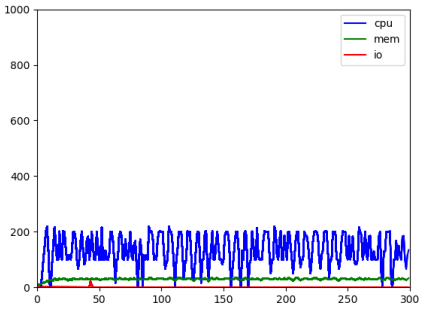
[RBH1]



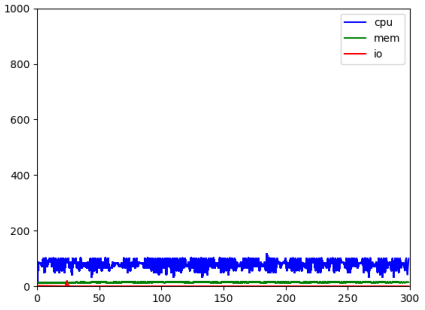
[DBH1]



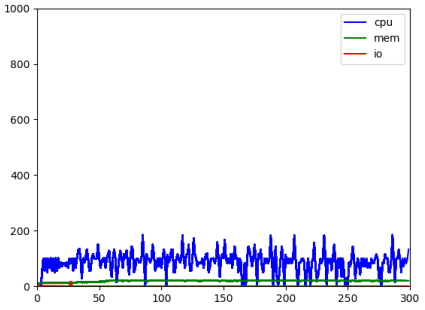
[RBM1]



[DBM1]

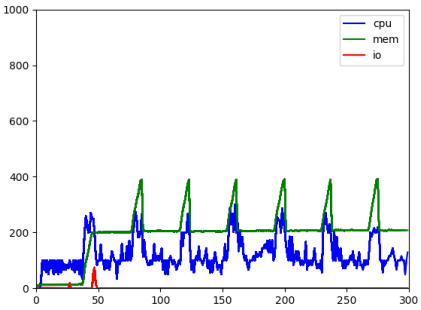


[RBL1]

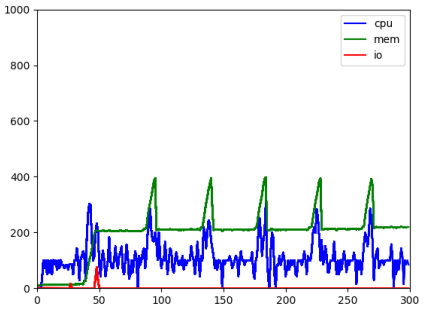


[DBL1]

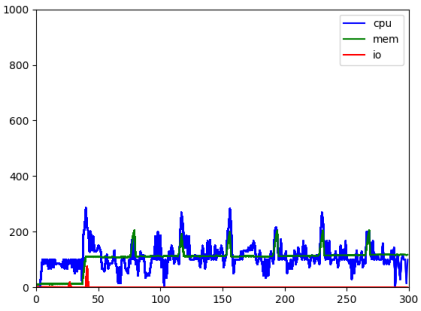
Figura 35 – SLEEP: entradas simples, tipo BIN



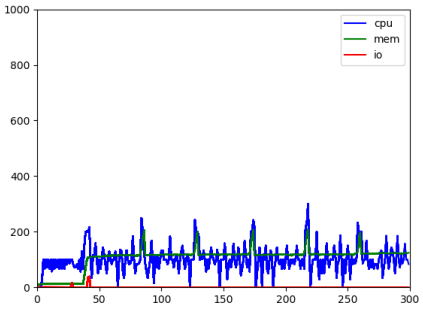
[RBCL1]



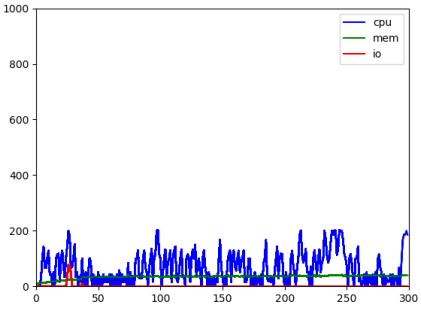
[DBCL1]



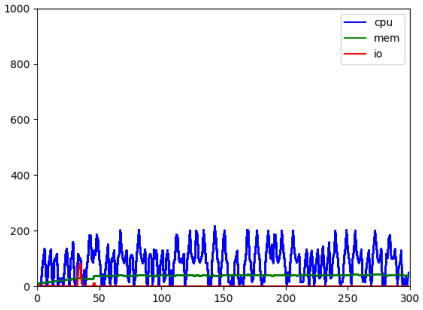
[RBCL2]



[DBCL2]

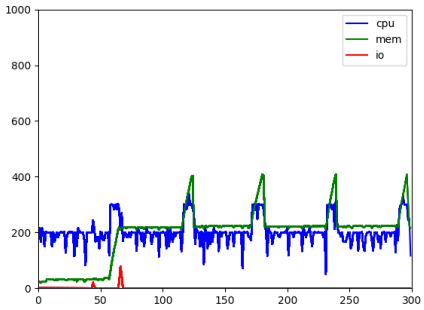


[RBCL3]

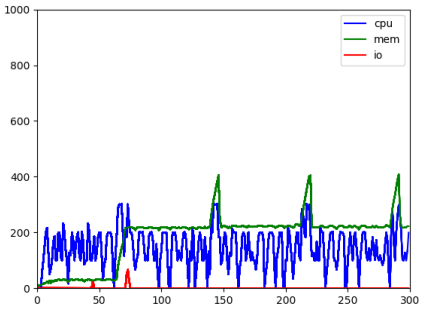


[DBCL3]

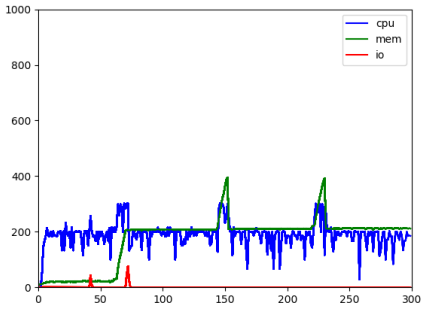
Figura 36 – SLEEP: entradas mistas, cargas L



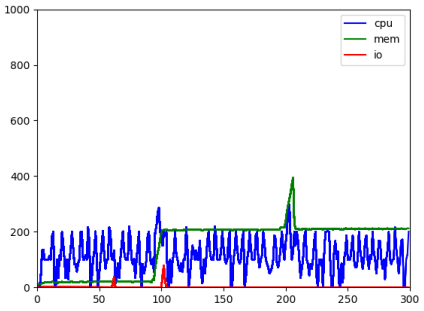
[RBCM1]



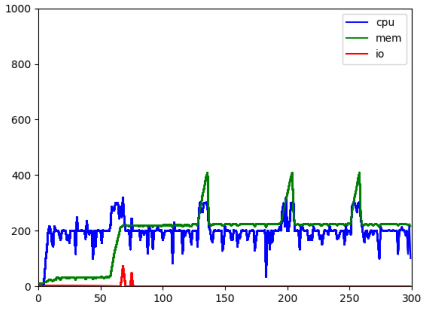
[DBCM1]



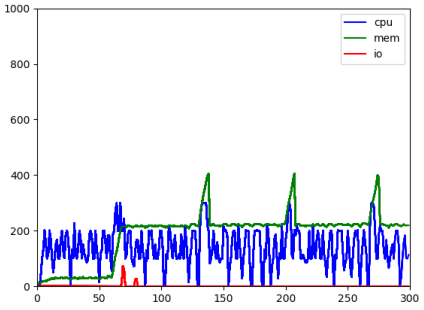
[RBCM1]



[DBCM2]

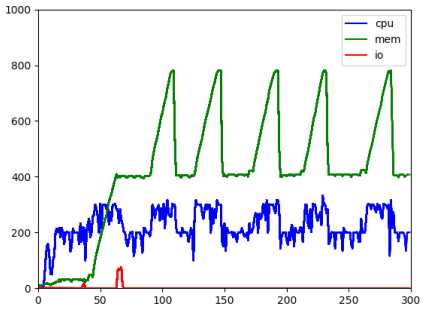


[RBCM1]

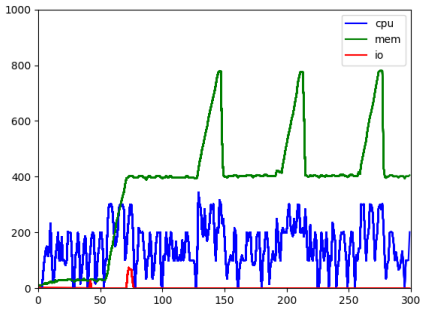


[DBCM3]

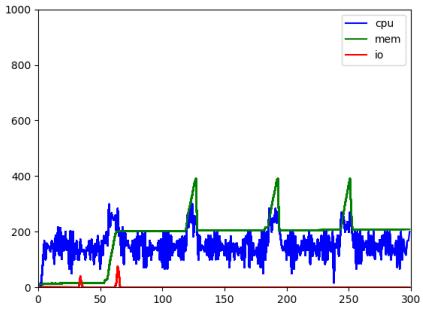
Figura 37 – SLEEP: entradas mistas, cargas M



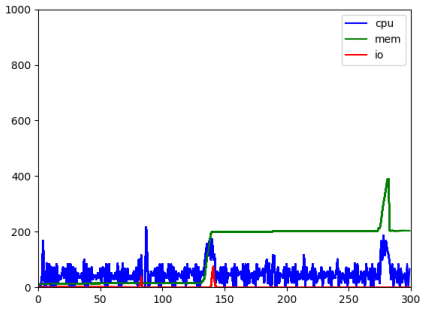
[RBCH1]



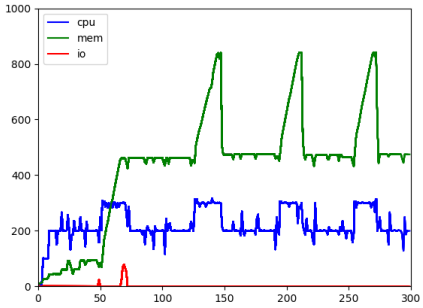
[DBCH1]



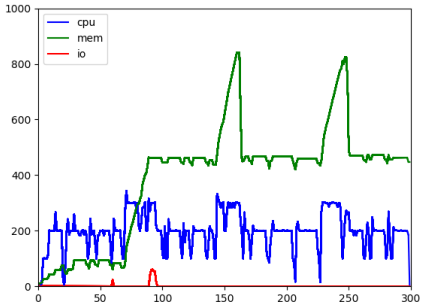
[RBCH2]



[DBCH2]



[RBCH3]

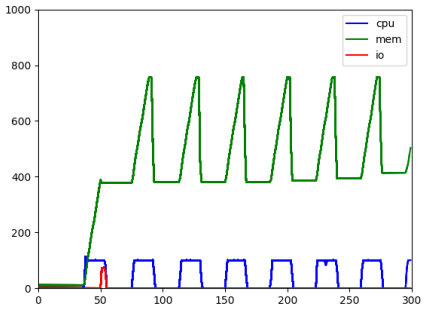


[DBCH3]

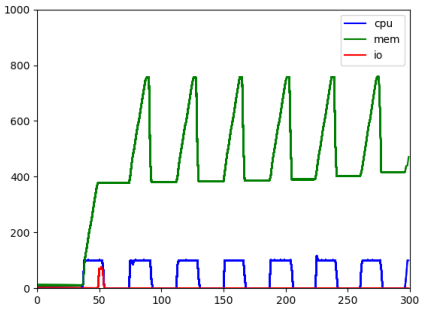
Figura 38 – SLEEP: entradas mistas, cargas H

7.3.6 SWAP

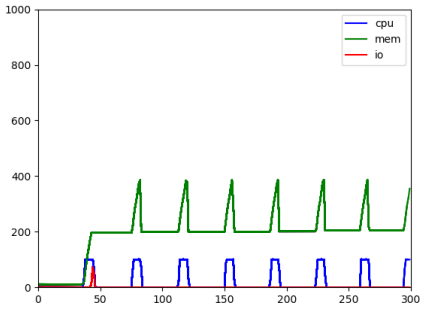
As Figuras 39, 40, 41, 42 e 43 são gráficos de amostras do mutante SWAP. Este defeito causa uma falha de erro de interpretação das informações de cabeçalho de pacotes **BIN**. Dependendo da entrada, como é o caso da carga DBCL3, a falha causa uso crescente de memória. É um defeito difícil de identificar por seus efeitos dependerem não somente do tipo de carga como também dos valores de cabeçalho.



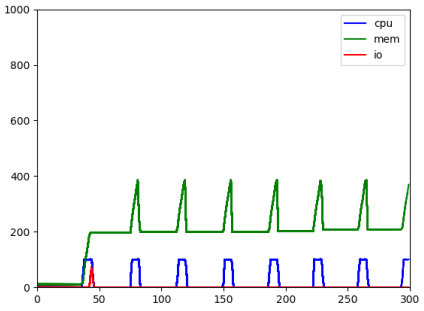
[RCH1]



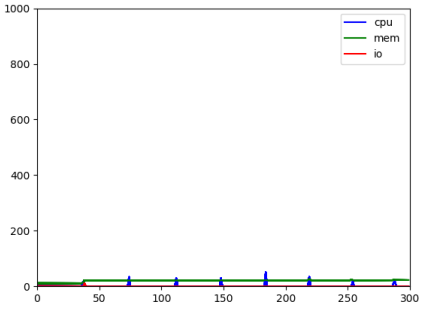
[DCH1]



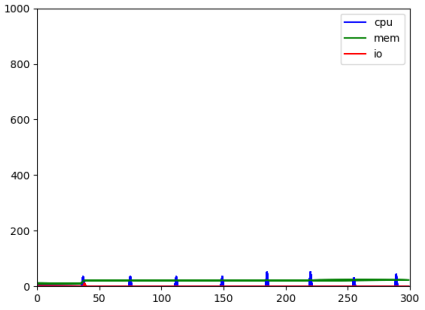
[RCM1]



[DCM1]

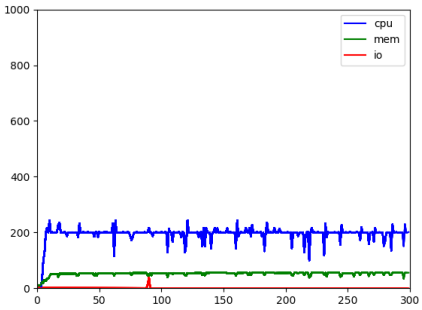


[RCL1]

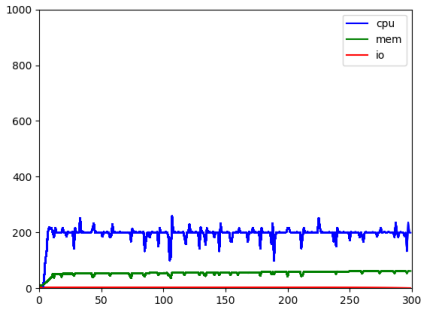


[DCL1]

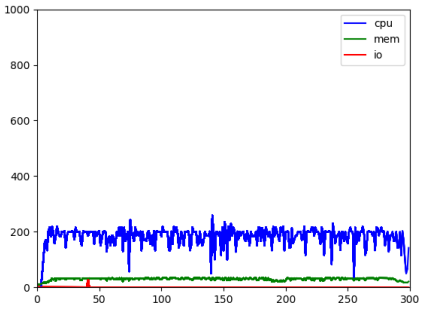
Figura 39 – SWAP: entradas simples, tipo CSV



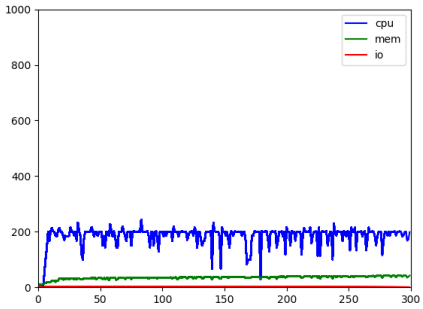
[RBH1]



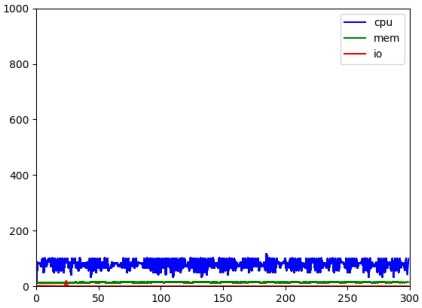
[DBH1]



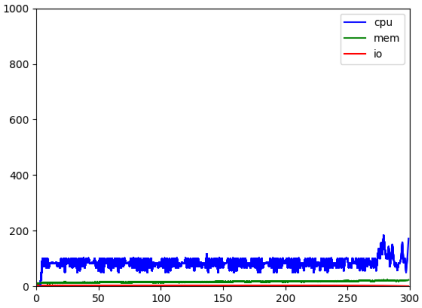
[RBM1]



[DBM1]

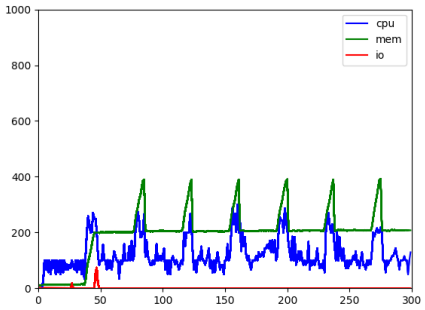


[RBL1]

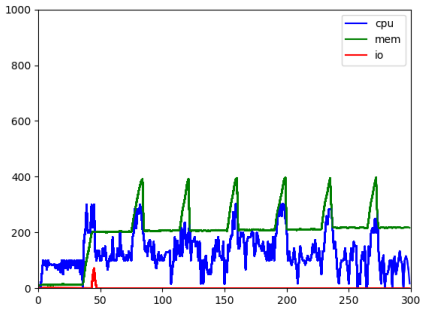


[DBL1]

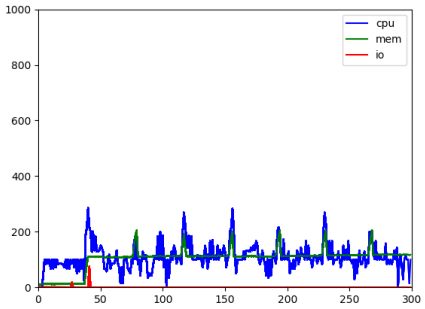
Figura 40 – SWAP: entradas simples, tipo BIN



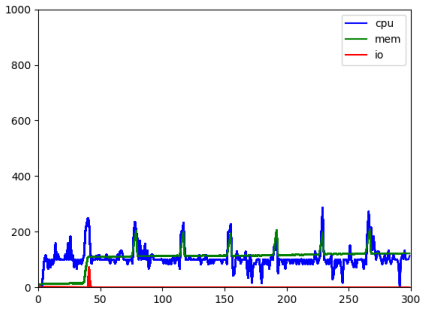
[RBCL1]



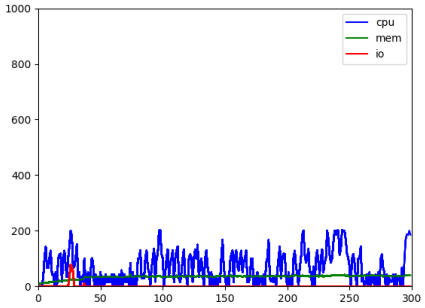
[DBCL1]



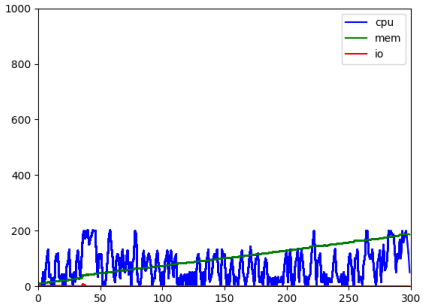
[RBCL2]



[DBCL2]

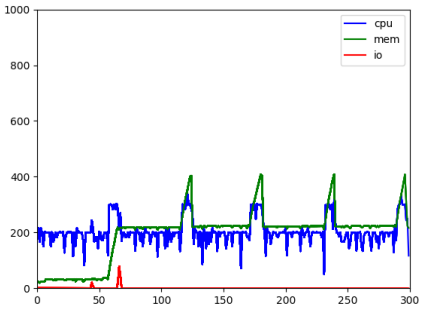


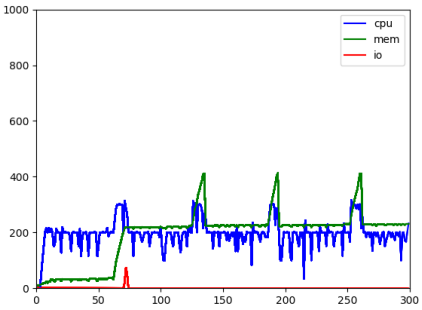
[RBCL3]

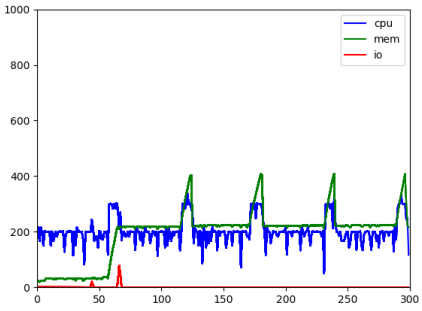


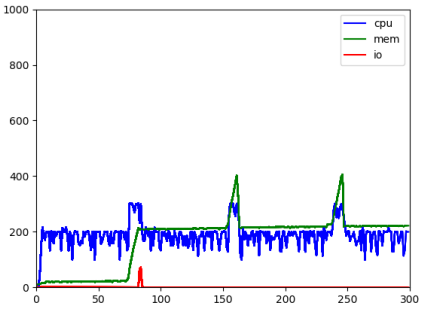
[DBCL3]

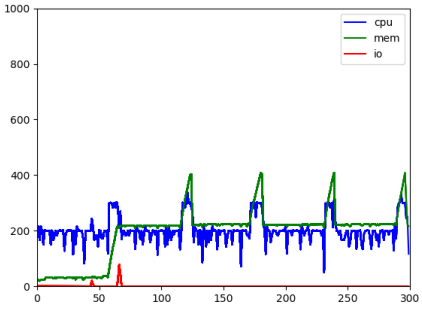
Figura 41 – SWAP: entradas mistas, cargas L

[BCM1] 

[BCM1] 

[BCM2] 

[BCM2] 

[BCM3] 

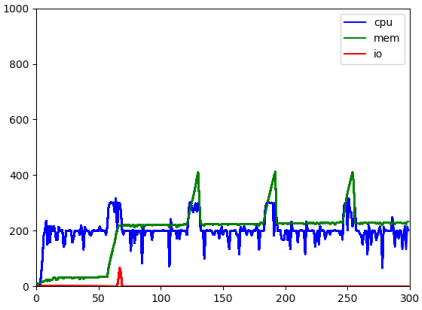
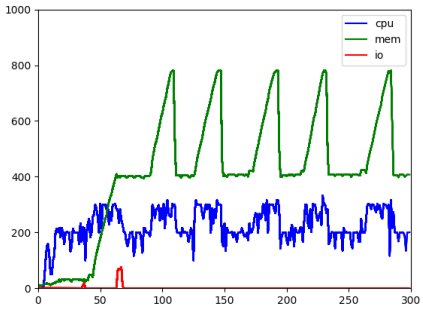
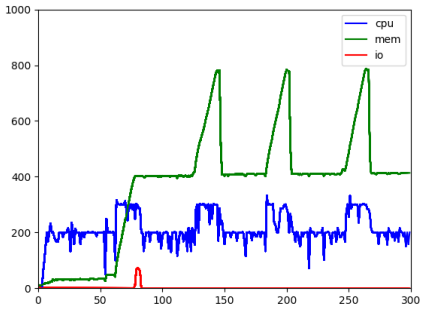
[BCM3] 

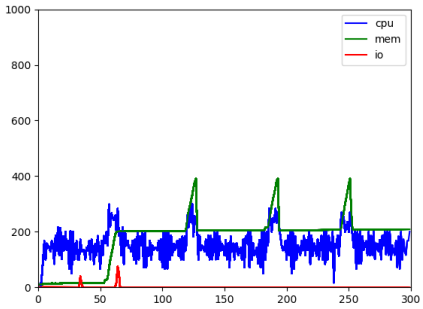
Figura 42 – SWAP: entradas mistas, cargas M



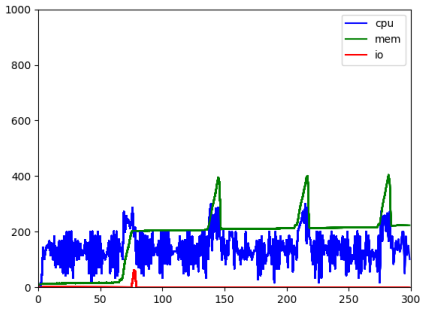
[RBCH1]



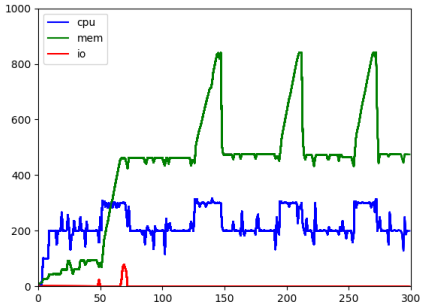
[DBCH1]



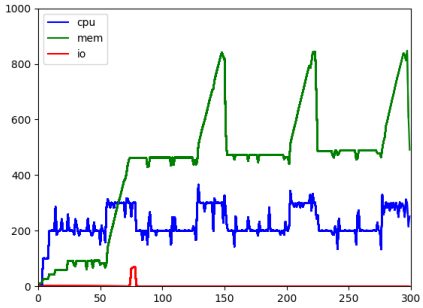
[RBCH2]



[DBCH2]



[RBCH3]

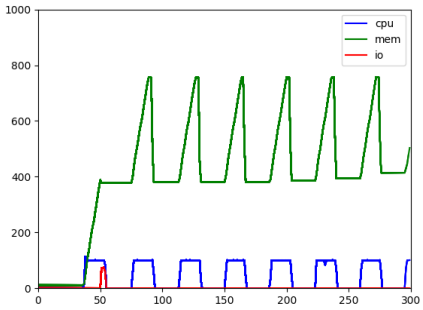


[DBCH3]

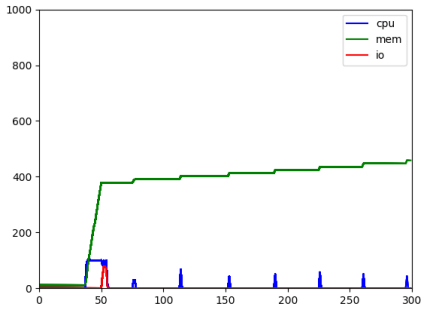
Figura 43 – SWAP: entradas mistas, cargas H

7.3.7 UNLOCK

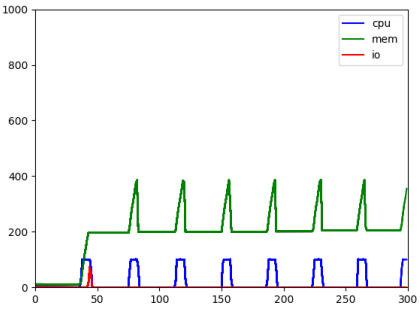
As Figuras 44, 45, 48, 47 e 46 são gráficos de amostras do mutante UNLOCK. Ele causa o bloqueio permanente das *threads* de processamento de dados **CSV**. Como ocorre após o processamento, pode causar a falsa impressão de que tudo correu bem. É um exemplo de defeito que pode enganar testes baseados em saída esperada.



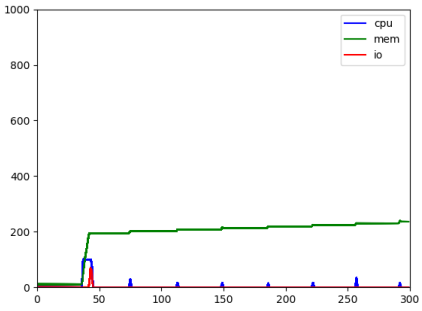
[CH1]



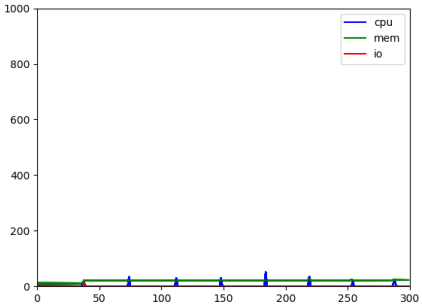
[CH1]



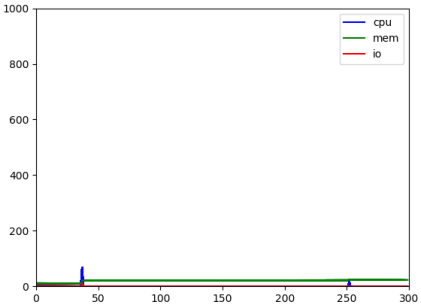
[CM1]



[CM1]

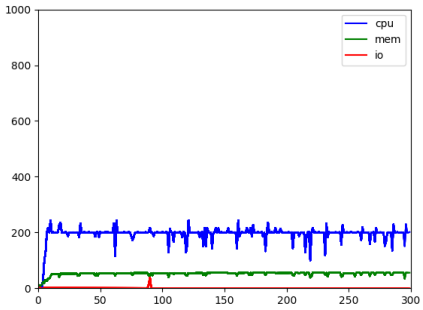


[CL1]

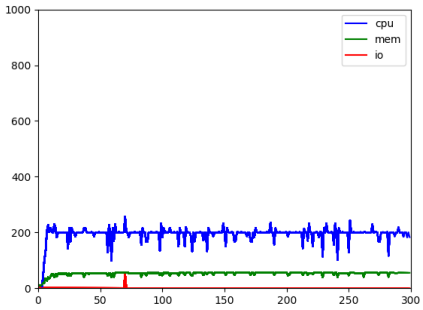


[CL1]

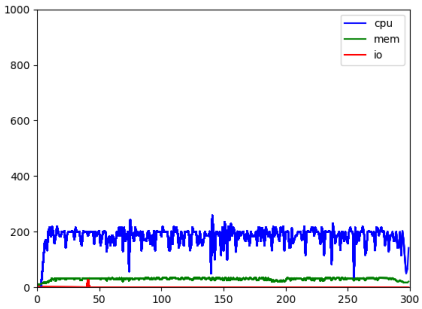
Figura 44 – UNLOCK: entradas simples, tipo CSV



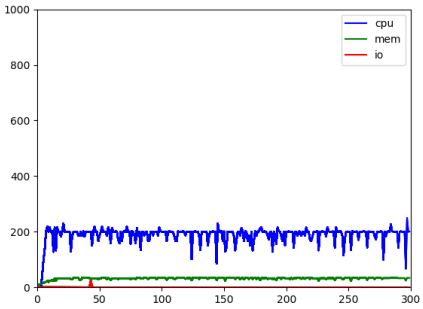
[BH1]



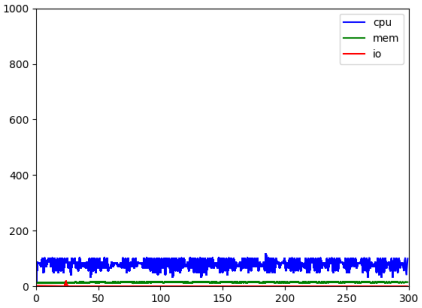
[BH1]



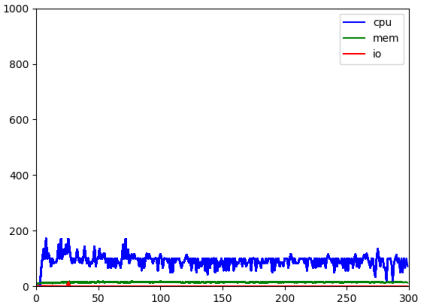
[BM1]



[BM1]

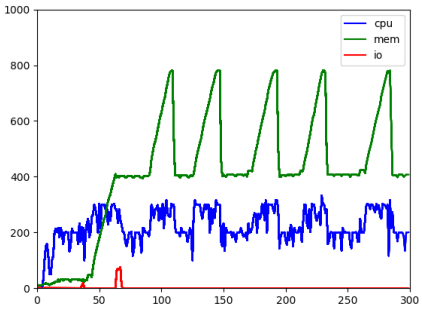


[BL1]

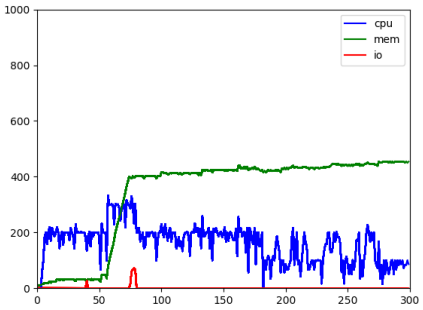


[BL1]

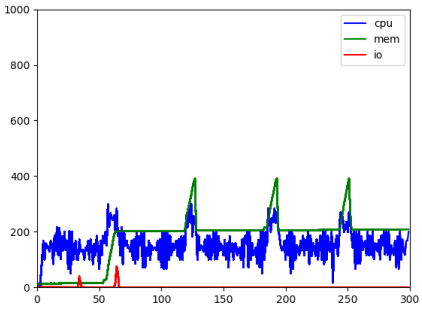
Figura 45 – UNLOCK: entradas simples, tipo BIN



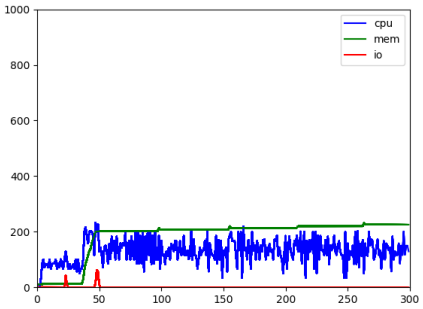
[BCH1]



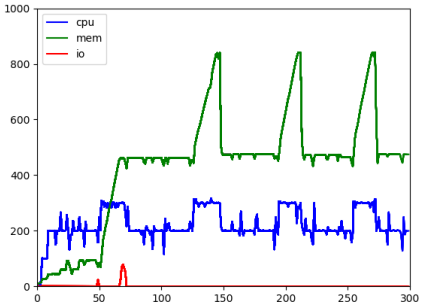
[BCH1]



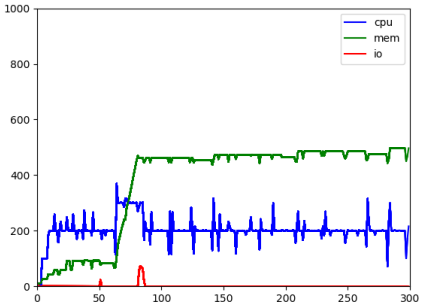
[BCH2]



[BCH2]

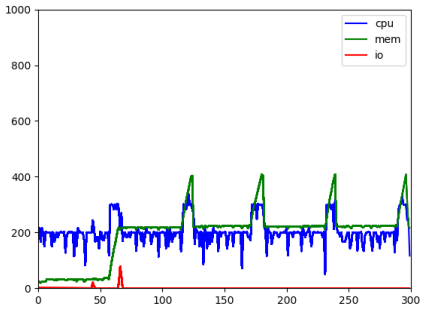


[BCH3]

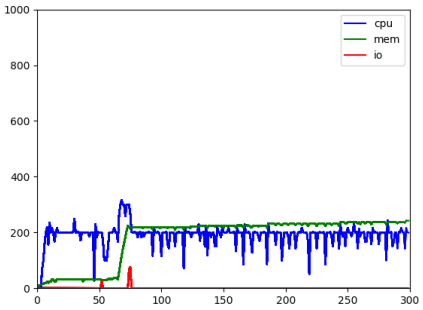


[BCH3]

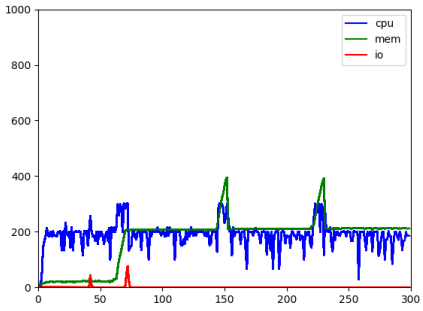
Figura 46 – UNLOCK: entradas mistas, cargas H



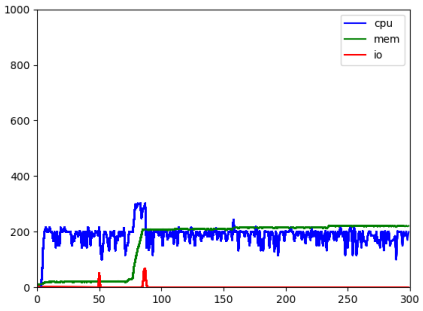
[BCM1]



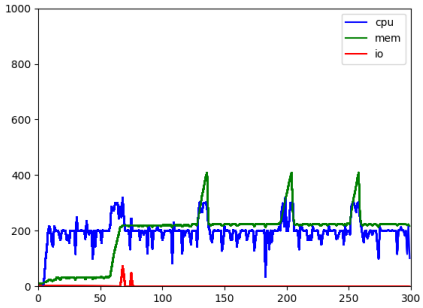
[BCM1]



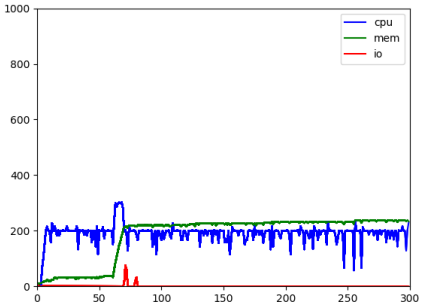
[BCM2]



[BCM2]



[BCM3]



[BCM3]

Figura 47 – UNLOCK: entradas mistas, cargas M

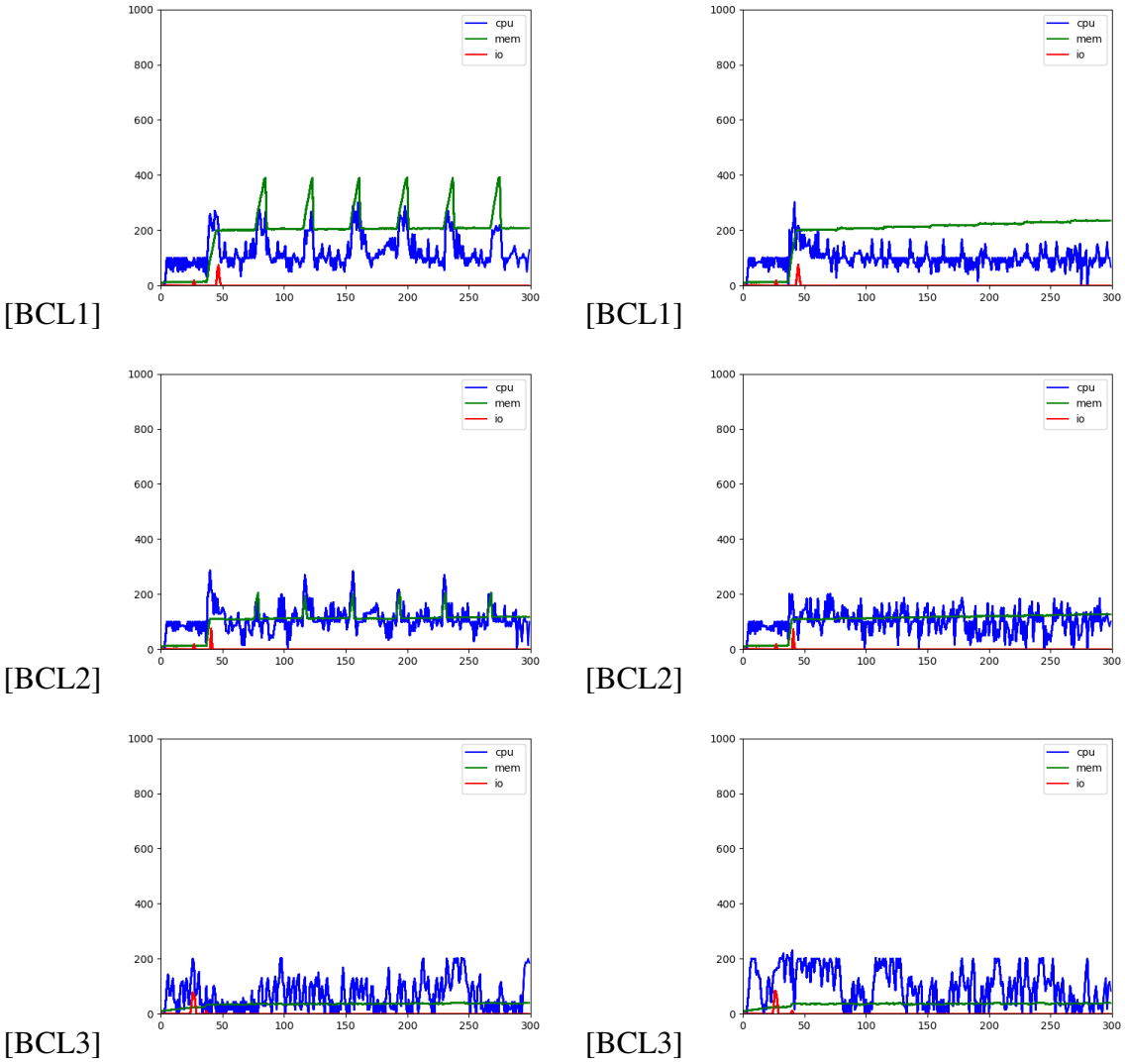


Figura 48 – UNLOCK: entradas mistas, cargas L

7.4 Resultados experimentais

Nas Tabelas 1 e 2 podemos ver o resultado do experimento, com heurística simples. As colunas representam os diferentes defeitos e as linhas as carga de trabalho utilizadas. As tabelas são compostas da seguinte forma:

- Amarelo: resultado correto, cujo número representa a iteração onde a anomalia foi identificada;
- Cinza: resultado correto, onde o defeito não gerou falha, e também não foi identificada anomalia;
- Vermelho: resultado incorreto, podendo ser:

  Número entre colchetes: é o total de falso positivo ocorrido; e

Letra N: a anomalia não foi verificada.

Os casos em **amarelo** são os casos onde o defeito gerou falha, e esta foi detectada em um número execuções de 1 a 30. Os casos em **cinza** são os casos onde o defeito não causa falha (não é executado), então foi considerado caso normal, sem detecção de anomalia. Os casos em **vermelho** são os casos onde o método não funcionou. Se for uma letra **N**, significa que o defeito gerou falha, mas essa não foi detectada. Se for um **número entre colchetes** então é um caso de falso-positivo: o defeito não causou falha, mas foi detectada anomalia, com número de execuções entre colchetes.

Na Tabela 1 estão resultados usando entradas de tipo único, sendo **BIN** ou **CSV**. No caso de entrada simples é natural que, para cada mutante, algumas falhas não ocorram para determinadas entradas. A **Tricorder** se mostrou eficaz para todos os casos onde houve falha, cuja anomalia de desempenho foi identificada entre 3 execuções (melhor caso com UNLOCK e CSVH1) e 29 execuções (pior caso com SWAP e BINH1). Não houve caso de não identificação de anomalia, contudo, houve 3 casos de falso-positivo.

	BCLEAN	MONO	SLEEP	SWAP	INFINITE	NOBREAK	UNLOCK
BL1	9	20	16	21	-	-	-
BM1	6	5	9	9	-	[17]	-
BH1	24	5	15	29	-	-	-
CL1	[15]	-	-	-	6	15	10
CM1	[20]	-	-	-	5	15	6
CH1	-	-	-	-	5	4	3

Tabela 1 – Resultados do experimento, entrada simples - heurística simples

Na Tabela 2 estão os casos de entrada mista. Não há, nessa configuração, possibilidade do defeito não poder causar falha (não há casos de falso-positivo). Os mutantes BCLEAN, MONO e SLEEP e INFINITE foram identificados para todas as cargas de trabalho. O mutante mais dependente do tipo de carga foi o NOBREAK, identificado apenas para BINCSVH3. A eficácia do **Tricorder** variou entre 3 execuções (melhor caso, UNLOCK e BINCSVH2) e 26 execuções (pior caso correto, SWAP e BINCSVH2). O **Tricorder** não funcionou para 12 casos, maioria NOBREAK, além de SWAP e UNLOCK.

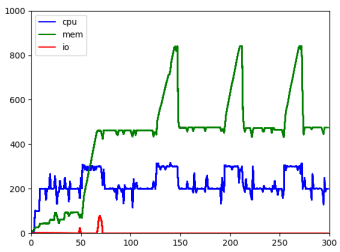
	BCLEAN	MONO	SLEEP	SWAP	INFINITE	NOBREAK	UNLOCK
BCL1	9	12	15	N	6	N	8
BCL2	4	5	17	N	5	N	14
BCL3	13	7	10	14	9	N	N
BCM1	8	9	6	15	7	N	4
BCM2	24	9	8	13	5	N	12
BCM3	8	7	12	22	5	N	13
BCH1	5	4	7	15	5	N	6
BCH2	6	9	7	26	5	N	3
BCH3	16	6	18	N	5	19	7

Tabela 2 – Resultados do experimento, entrada mista - heurística simples

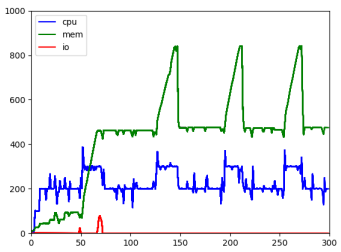
Como era previsto, a carga de trabalho inserida interfere no impacto que o defeito causa no desempenho, e para alguns casos esse efeito é tão pequeno que não é detectado pelo **DAMICORE**.

Por exemplo, no caso NOBREAK, onde isso ocorre com maior frequência, o defeito causa uma execução indevida e desnecessária de um pacote de dados **CSV** como se fosse **BIN**. O impacto desse erro está na relação proporcional do tamanho dos pacotes **BIN** e **CSV**, sendo que no caso onde o **CSV** é consideravelmente maior que o **BIN**, esse efeito é melhor identificado.

É o caso da entrada BCH3 (Figura 49) onde temos proporcionalmente uma entrada **CSV** muito maior que as entradas **BIN**. Da mesma forma nas 3 entradas puramente **CSV** o efeito pôde ser detectado. Este efeito pode ser visto como um pequeno pico de CPU (em azul) na borda de subida das ondas quadradas. Esse pico representa o dado **BIN** sendo processado como tipo **CSV**.



[REF BCH3]



[NOBREAK BCH3]

Figura 49 – Medição da carga BCH3, a esquerda referência, a direita mutante NOBREAK

O uso do **DAMICORE** pelo **Tricorder** no processo de agrupamento se mostrou eficiente, com tempo aproximado de processamento e geração dos grupos variando de **120 segundos** a **180 segundos**, com média de **140 segundos**. Este tempo inclui o agrupamento de toda a base de referência e das medições de teste. Em um caso de aplicação real, um tempo de pós-processamento na ordem de 3 minutos para obter um diagnóstico da execução é bastante viável, e poderia ser feito de forma constante e automática.

No geral, o **Tricorder** foi capaz de identificar todos os mutantes para no mínimo uma entrada. Embora esse resultado seja satisfatório, a presença de falso-positivo pode ser um problema que afeta sua aplicação em ambiente real. Com a intenção de minimizar a ocorrência de falsos-positivos, uma melhoria na heurística foi proposta e aplicada. Nessa segunda versão, apenas em casos onde há positivo para duas execuções sequenciais é que é considerada a presença de defeito.

Essa ideia se baseia na heurística de que os agrupamentos se reorganizam, e a partir dessa nova reorganização surgem novos agrupamentos que satisfazem as condições de presença de defeito. Nesses casos, ao ser adicionada uma nova entrada de execução, as chances desse efeito ocorrer novamente são muito menores. No entanto, caso seja um verdadeiro positivo as chances de o agrupamento se manter são bem maiores.

Os resultados do experimento, agora usando a nova meta-heurística podem ser vistos na Tabela 3. Observamos uma piora de eficácia do **Tricorder**, refletida em um aumento geral do número de iterações em que a anomalia é identificada. Houve também um caso de não-funcionamento, que não ocorreu com heurística simples. Contudo o número de falso-positivos reduziu para apenas um.

	BCLEAN	MONO	SLEEP	SWAP	INFINITE	NOBREAK	UNLOCK
BL1	9-10	20-21	22-23	24-25	-	-	-
BM1	6-7	8-9	16-17	9-10	-	-	-
BH1	24-25	5-6	15-16	N	-	-	-
CL1	[18-19]	-	-	-	6-7	15-16	10-11
CM1	-	-	-	-	5-6	15-16	10-11
CH1	-	-	-	-	5-6	4-5	3-4

Tabela 3 – Resultados do experimento entrada simples - heurística restritiva

Na Tabela 4 estão os resultados da heurística restritiva para entradas mistas. Houve também uma piora geral da eficácia, sendo necessárias mais iterações para identificação de anomalias. Para o caso do mutante **SWAP** número de não-funcionamentos aumentou significativamente.

	BCLEAN	MONO	SLEEP	SWAP	INFINITE	NOBREAK	UNLOCK
BCL1	9-10	12-13	15-16	N	6-7	N	8-9
BCL2	4-5	14-15	20-21	N	5-6	N	14-15
BCL3	13-14	7-8	10-11	14-15	9-10	N	N
BCM1	8-9	9-10	12-13	N	9-10	N	4-5
BCM2	24-25	9-10	11-12	N	5-6	N	19-20
BCM3	8-9	7-8	20-21	N	5-6	N	21-22
BCH1	5-6	4-5	7-8	N	5-6	N	6-7
BCH2	6-7	9-10	11-12	26-27	5-6	N	14-15
BCH3	16-17	6-7	21-22	N	5-6	19-20	7-8

Tabela 4 – Resultados do experimento entrada mista - heurística restritiva

Para ambas as heurísticas é possível ver claramente como múltiplas entradas e a variação de carga torna difícil a identificação de defeitos, ao observarmos a diferença de eficácia quando temos apenas um tipo de entrada e quando misturamos as duas. Usando como exemplo o mutante UNLOCK, na Figura 50 podemos ver na esquerda uma medição de referência e na direita uma medição com falha, cuja anomalia foi identificada com 3 execuções, com entrada BCH2. Consideramos um caso ótimo o caso onde a anomalia é detectada com um número mínimo de execuções.

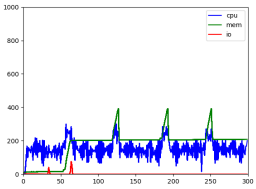
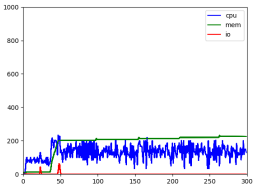
[REF BCH2]  [UNLOCK BCH2] 

Figura 50 – UNLOCK: caso ótimo

Na Figura 51 estão, na esquerda a execução de referência, e na direita o mutante, para entrada BCL3.

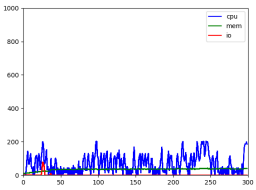
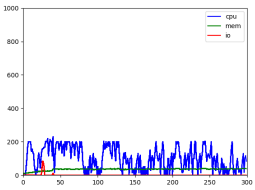
[REF BCL3]  [UNLOCK BCL3] 

Figura 51 – UNLOCK: caso não-detectado

Com essa comparação exemplo fica claro como a variação de entrada pode fazer um mutante ser rapidamente identificado, com apenas 3 execuções e apresentando variação de desempenho visível, como também não chegar a ser detectado, apresentando medições de desempenho visualmente muito semelhantes à original. No caso ótimo, o uso de memória maior (em **verde**) torna visível a diferença entre o mutante e o original. No original há picos de uso e no mutante uma linha quase reta levemente crescente. No caso não-detectado, devido ao baixo uso de memória, a variação não é visível graficamente, nem foi detectada. O uso de memória em ambos é uma linha quase reta, levemente crescente.

Como parte da validação da proposta, um mutante igual a aplicação original, chamado CONTROL, considerado sem defeitos, foi usado no processo de teste. Este mutante representa o caso de funcionamento normal da aplicação, onde não devem ser detectadas anomalias.

	CONTROL: HEURÍSTICA SIMPLES	CONTROL: HEURÍSTICA RESTRITIVA
BL1	-	-
BM1	-	-
BH1	-	-
CL1	-	-
CM1	-	-
CH1	-	-
BCL1	-	-
BCL2	-	-
BCL3	-	-
BCM1	[29]	[30]
BCM2	[28]	-
BCM3	-	-
BCH1	-	-
BCH2	-	-
BCH3	-	-

Tabela 5 – Resultados do experimento de verificação da metodologia, mutante sem defeitos CONTROL, heurísticas simples e restritiva

Na Tabela 5 estão os resultados das execuções do mutante CONTROL, sujeito às 15 cargas, sendo aplicadas as 2 heurísticas. A heurística simples apontou 2 casos de falso-positivo, enquanto a heurística restritiva apontou um.

7.5 Aplicação em ambiente industrial

7.5.1 Ambiente de teste

Para avaliação do **Tricorder** em um cenário real, o framework proposto foi utilizado no teste de desempenho de uma aplicação pré-existente em uma indústria brasileira. O Objeto de Teste é um aplicativo de cópia de arquivos por rede *ethernet*, executando operações de transferência, compactação, criptografia, fragmentação e teste de integridade.

A aplicação escolhida estava em fase de desenvolvimento e nos testes apresentou um defeito que afeta o desempenho. Este foi identificado manualmente pelo desenvolvedor e corrigido. Com esse exemplo de defeito real disponível, decidimos utilizar o **Tricorder**, comparando a versão com defeito com a versão após o defeito ser consertado, e avaliar se seria possível identificar essa falha automaticamente.

Embora seja uma versão em desenvolvimento , a aplicação alvo é capaz de rodar em ambiente de produção, sujeito a entradas reais, e foi nesse cenário que aplicamos a metodologia. Partimos de 5 conjuntos de entrada, formados por dados reais, e definimos como 5 cargas de trabalho, com as seguintes características:

- src1: 12 arquivos, totalizando 2,83 GB;
- src2: 12 arquivos, totalizando 1,64 GB;
- src3: 8 arquivos, totalizando 4,08 GB;
- src4: 24 arquivos, totalizando 1,47 GB; e
- src5: 1 arquivo, totalizando 5,49 GB.

Os dados de entrada usados são arquivos binários, em formato próprio da aplicação em teste, cuja estrutura interna é ignorada. São todos do mesmo tipo, gerados e consumidos pela mesma aplicação.

Estes arquivos de entrada são dados reais, utilizados pela aplicação em seu ambiente de produção. A natureza dos dados não pôde ser descrita por serem dados proprietários da empresa citada.

7.5.2 Descrição do defeito

O defeito na aplicação está na comparação da *hash* dos arquivos transmitidos. No caso de não ter dado tempo do *hash* ser gerado, uma nova requisição era criada, gerando possivelmente uso extra e desnecessário de CPU e memória.

```
1 bool_t file_handler_t::
2 check_hash_from_files(file_info_t& file_info)
3 {
4     ...
5     // ********************************//
6     // Bug here: While file hash isn't received by the client,
7     // repush the job to the queue
8     if (!file_info.has_hash)
9         return false;
10    ...
```

A solução foi transformar a comparação em uma função bloqueante, que aguarda pelo *hash* estar pronto antes de verificar sua integridade.

```
1 bool_t file_handler_t::
2 check_hash_from_files(file_info_t& file_info)
3 {
4     ...
5     // ********************************//
```

```
6     // Fix: Wait for the file hash for 100 ms if it's not
      received yet,
7     // repush the job if the hash isn't received
8     if (!wait_for_file_hash(file_info))
9             return false; // repush job
10    ...
```

7.5.3 Medição de desempenho

Para cada entrada foram feitas 30 execuções do programa original, criando assim a base de referência. A partir disso, para cada carga, foi aplicado o método tanto para o aplicativo com defeito, grupo teste; como para o aplicativo original, grupo controle. Amostras das medições do programa referência, para cada entrada, podem ser vistas na Figura 52.

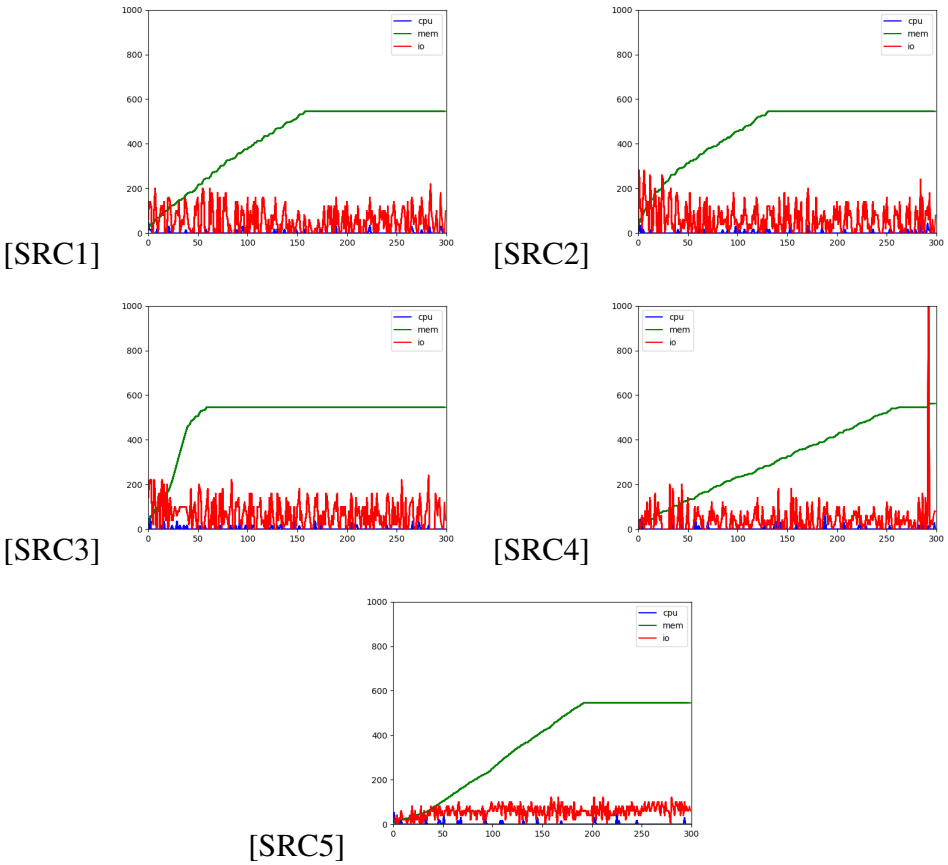


Figura 52 – FileTransfer: referência

Na Figura 53 estão os gráficos das medições com defeito. Na esquerda estão as medições que visualmente mais se aproximam da referência, e na direita as que mais se diferenciam. Podemos verificar que mesmo na presença de defeito algumas execuções do programa apresentaram uso de recursos muito semelhantes ao programa referência. E, em todos os casos, ao menos uma

execução teve um desempenho muito diferente do esperado. Isso se deve ao fato do defeito nem sempre causar falha, e mesmo quando causa, esta pode ocorrer em diferentes momentos, causando impacto variado no desempenho. Esta imprevisibilidade e não-determinismo de ocorrência da falha é uma característica comum a defeitos de sincronismo e paralelismo. (JAIN, 1990)

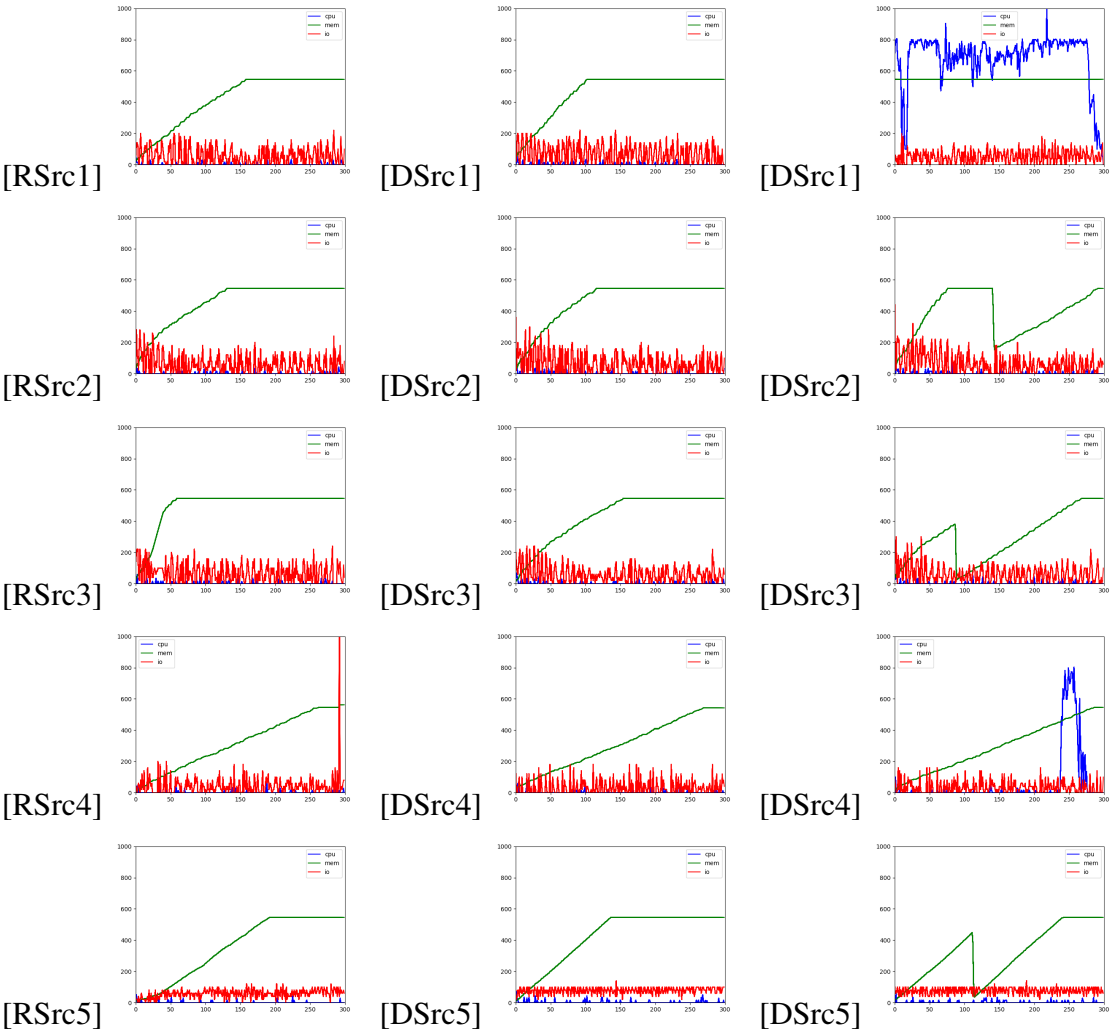


Figura 53 – FileTransfer, da esquerda pra direita: referência de execução, execução do mutante que mais se aproxima da referência e execução do mutante que mais se diferencia

7.5.4 Resultados da aplicação real

A execução do experimento, para cada uma das 5 cargas de trabalho, chamadas de src1 a src5, apresentou os seguintes resultados.

- Grupo Teste:

  src1 : 8 execuções;

  src2 : 19 execuções;

  src3 : 13 execuções;

src4 : 28 execuções; e

src5 : 12 execuções.

Para validar o experimento também foi executada a versão de referência do programa, para as mesmas entradas, obtendo o seguinte resultado.

- Grupo Controle:

  src1 : não detectado;

  src2 : não detectado;

  src3 : 17 execuções - falso-positvo;

  src4 : não detectado; e

  src5 : não detectado.

Com esse experimento foi possível verificar o desempenho da metodologia proposta em um caso de defeito real, encontrado em ambiente de desenvolvimento. Se este defeito fosse detectado pelo desenvolvedor, e a aplicação tivesse sido publicada em ambiente de produção da forma que estava, o **Tricorder** poderia ter anunciado a presença do defeito pelo reflexo no uso de recursos.

7.6 Considerações finais

Sendo realizados em ambiente controlado, com defeitos inseridos, foi possível realizar experimentos de aplicação da metodologia **Tricorder** para diferentes cenários, variando tanto a entrada como a falha. Como já era esperado, os resultados também variaram, desde não funcionar para 30 execuções, até funcionar funcionar com 3. Mesmo com a existência de alguns falso-positivos, o desempenho da metodologia **Tricorder** se mostrou muito promissor, pela facilidade de uso e pela abrangência de cobertura de detecção de anomalias.

A maior dificuldade do experimento foi em trabalhar com uma grande abrangência de entradas e defeitos possíveis, e executar de forma controlada para validação da proposta. O uso real de **Tricorder** poderia ser mais simples pois ele também funciona para casos menos abrangentes, com menos tipos de entradas e defeitos desconhecidos, desde que estes influenciem no desempenho da aplicação de forma significativa.

O uso real do **Tricorder** também foi promissor, pois este foi aplicado em um ambiente externo, com um software em desenvolvimento e com as entradas disponíveis. Nesse cenário não havia nem caracterização de carga nem tampouco requisitos de desempenho. Ainda sim foi possível detectar a presença de uma anomalia de desempenho.

CAPÍTULO

8

CONCLUSÕES E TRABALHOS FUTUROS

8.1 Principais resultados

No Capítulo 7 apresentamos os resultados obtidos nos experimentos feitos de forma controlada. Este ambiente foi criado visando ser representativo na avaliação da eficácia da metodologia no teste de software baseado em desempenho na ausência de requisitos, para aplicações de processamento sob demanda. Os resultados se mostraram muito favoráveis, sendo a ferramenta **Tricorder** capaz de identificar a maior parte dos defeitos inseridos apesar da variação de carga, e todos os defeitos foram detectados em ao menos uma carga.

Os resultados mostraram a possibilidade de automatização de parte do processo de caracterização de carga, por meio de agrupamento de medições de uso de recursos. Esse agrupamento é usado como referência, como um oráculo de medições de desempenho. Diferentes categorias de defeitos puderam ser detectados dessa forma. Os resultados também mostraram que uma janela de 30 execuções foi suficiente para detectar os defeitos, assim como a amostragem escolhida de 10 Hz. Esses valores podem ser adaptados para a aplicação em teste.

Por meio do **Tricorder** foi possível detectar a presença de defeitos em aplicações, no modo teste caixa-preta, sem acesso a código-fonte, instrumentação de códigos ou detalhes da arquitetura. Isso facilita o processo de teste de desempenho da aplicação sem necessidade de conhecimento estrutural da mesma. Isso também possibilita o teste de aplicações externas, por exemplo bibliotecas de terceiros.

Outro resultado foi a seleção de categorias de defeitos que afetam o desempenho, representados por implementações destes, e o estudo de seu efeito nas medições de uso de recursos. Estes resultados podem servir como base para estudos posteriores que relacionam defeitos com anomalias de desempenho.

O **DAMICORE** se mostrou um método de agrupamento versátil e de fácil uso em diferentes aplicações. Sendo usado neste projeto na forma de ferramenta, baseado em (MEDEIROS,

2016), foi fundamental para alcançar os resultados obtidos pela metodologia **Tricorder**.

8.2 Limitações

Sendo um método que utiliza um algoritmo de aprendizado não-supervisionado, trabalhando de forma incremental, essa metodologia tem a desvantagem de necessitar de mais de uma amostra defeituosa para detectar a anomalia. Como mencionado no Capítulo 7 esse número pode ser de apenas 3 amostras, mas em geral é mais que isso. Mesmo 3 amostras poderiam ser um limitante dependendo da rapidez que se espera da resposta do método de teste.

O uso da metodologia **Tricorder** da forma proposta exige que o **Thermometer** seja executado antes da aplicação objeto de teste, pois trabalhamos com os primeiros 30 segundos da aplicação como medida de comparação. Este padrão pode ser estendido para medições quaisquer, ou de execução completa, mas estudo não foi feito neste trabalho.

Quanto ao formato dos arquivos de medição de desempenho, foram também feitos testes usando o formato Hierarchical Data Format (HDF5), pois este tem tido grande adesão nas aplicações de ciência de dados. Porém, devido ao formato interno do arquivo, em estrutura otimizada em árvore, os dados ficam misturados. Essa disposição não sequencial da informação armazenada inviabilizou o uso do **DAMICORE** na forma proposta, pois o algoritmo de agrupamento não foi capaz de encontrar a similaridade entre as séries temporais.

8.3 Contribuições

Assim como no trabalho (MI *et al.*, 2008), neste estudo conseguimos criar de forma prática perfis de desempenho esperado de aplicações, baseados nos seus comportamentos em diferentes cenários de carga. O fato destes perfis serem criados por agrupamento automático de medições de desempenho, permite que a aplicação seja tratada como caixa-preta, facilitando muito o trabalho de análise e teste.

A atividade de caracterização de carga pode ser quase toda automatizada, dependendo apenas de escolhas de entradas representativas. Pela forma de trabalho proposta pelo **Tricorder**, mesmo essa escolha de entradas poderia ser aleatória, baseado no uso real da aplicação por parte de usuários. Quanto mais completa for a base de referência, melhores serão os resultados.

Em geral a atividade de teste de desempenho está relacionada com a adequação a requisitos de desempenho. Quando estes não existem a atividade não é realizada. Ainda que existam, em alguns casos são limitadas a aspectos pontuais como o tempo de resposta de determinada transação. A metodologia **Tricorder** permite que uma comparação de uso de recursos, como um teste de desempenho, seja feita baseada em um conhecimento sobre a aplicação criada de forma automática, pelo **DAMICORE**. Essa base de conhecimento serve como requisito de desempenho para efeito de comparação.

Não apenas o teste de desempenho, mas em geral a atividade de teste caixa-preta é dependente da existência de um Oráculo, como referência para as saídas esperadas. A criação deste costuma ser uma das atividades mais custosas e complexas neste tipo de teste. O **Tricorder** elimina a necessidade da pré-existência de um Oráculo, uma vez que em seu processo, o oráculo é criado de forma automática a partir do conhecimento extraído das medições de desempenho tomadas como referência.

Outra contribuição está na possibilidade de testar aplicações de terceiros, como bibliotecas proprietárias ou cujo código-fonte não seja de fácil acesso. Há ainda os casos, cada vez mais frequentes, de bibliotecas feitas em linguagem diferente da aplicação principal, que são unidas em tempo de compilação ou mesmo de execução. Nestes o critério de teste aplicado na linguagem principal pode não ser facilmente aplicável na linguagem cuja biblioteca foi escrita.

8.4 Trabalhos futuros

O método desse estudo teve enfoque em uma das muitas possíveis abordagens do tema. Por exemplo o uso de medições em formato numérico em arquivo texto se mostrou prático e eficaz, porém o uso de outros formatos como tabelas binárias ou imagens seria bastante promissor. Alguns testes usando formato Portable Network Graphics (PNG) e Device Independent Bitmap (BMP) obtiveram resultados positivos, especialmente o BMP, mas ainda menos eficientes que o CSV. Contudo são resultados preliminares e um estudo mais aprofundado poderia tornar o uso desses formatos ainda melhor que o CSV escolhido.

Em (PINTO; DELBEM; MONACO, 2018) o problema abordado estava na análise dos binários de aplicações e por meio de agrupamento, usando **DAMICORE**, separá-los em categorias de uso majoritário de CPU ou E/S. Esta abordagem de agrupamento de binários também seria possível na finalidade de encontrar defeitos cujo efeito é refletido no uso de recursos. Este seria um estudo complementar sem uso de medições de desempenho.

REFERÊNCIAS

ATTARIYAN, M.; CHOW, M.; FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In: **Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2012. (OSDI'12), p. 307–320. ISBN 978-1-931971-96-6. Disponível em: <http://dl.acm.org/citation.cfm?id=2387880.2387910>. Citado nas páginas 25, 26 e 46.

BONDI, A. B. **Foundations of Software and System Performance Engineering**. [S.l.]: Addison-Wesley Professional, 2014. 448 p. ISBN 0321833821. Citado na página 35.

CEBRIÁN, M.; ALFONSECA, M.; ORTEGA, A. *et al.* Common pitfalls using the normalized compression distance: What to watch out for in a compressor. **Communications in Information & Systems**, International Press of Boston, v. 5, n. 4, p. 367–384, 2005. Citado na página 28.

CHERKASOVA, L.; OZONAT, K.; MI, N.; SYMONS, J.; SMIRNI, E. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In: IEEE. **Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on**. [S.l.], 2008. p. 452–461. Citado nas páginas 25, 26, 27, 45, 46 e 47.

CLEARY, J.; WITTEN, I. Data compression using adaptive coding and partial string matching. **IEEE transactions on Communications**, IEEE, v. 32, n. 4, p. 396–402, 1984. Citado na página 52.

CPPREFERENCE. **std::mutex**. 2019. Recurso do C++ padrão. Disponível em: <https://pt.cppreference.com/w/cpp/thread/mutex>. Citado na página 76.

DELAMARO, M. E.; MALDONADO, J. C. **Introdução ao Teste de Software**. 2. ed. [S.l.]: Elsevier, 2016. 448 p. ISBN 9788535283525. Citado nas páginas 31, 32, 33 e 37.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 11, n. 4, p. 34–41, abr. 1978. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/C-M.1978.218136>. Citado na página 37.

GAMA, J.; FACELI, K.; LORENA, A.; CARVALHO, A. D. **Inteligência artificial: uma abordagem de aprendizado de máquina**. Grupo Gen - LTC, 2011. ISBN 9788521618805. Disponível em: <https://books.google.com.br/books?id=4DwelAEACAAJ>. Citado na página 49.

GANGLIA. **Ganglia Monitoring System**. 2019. Software de monitoramento e gerenciamento de recursos. Disponível em: <http://ganglia.info/>. Citado nas páginas 58 e 62.

JAIN, A. K.; MURTY, M. N.; FLYNN, P. J. Data clustering: A review. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 31, n. 3, p. 264–323, set. 1999. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/331499.331504>. Citado na página 50.

JAIN, R. **The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling**. Wiley, 1990. ISBN 9788126519057. Disponível em: <https://books.google.com.br/books?id=eOR0kJjgMqkC>. Citado nas páginas 25, 41, 43, 60 e 131.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 37, n. 5, p. 649–678, set. 2011. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2010.62>. Citado na página 38.

JIN, G.; SONG, L.; SHI, X.; SCHERPELZ, J.; LU, S. Understanding and detecting real-world performance bugs. In: **Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2012. (PLDI '12), p. 77–88. ISBN 978-1-4503-1205-9. Disponível em: <http://doi.acm.org/10.1145/2254064.2254075>. Citado na página 26.

LI, M.; CHEN, X.; LI, X.; MA, B.; VITANYI, P. M. The similarity metric. **IEEE Trans. Inf. Theor.**, IEEE Press, Piscataway, NJ, USA, v. 50, n. 12, p. 3250–3264, dez. 2004. ISSN 0018-9448. Disponível em: <http://dx.doi.org/10.1109/TIT.2004.838101>. Citado na página 52.

LI, M.; VITNYI, P. M. **An Introduction to Kolmogorov Complexity and Its Applications**. 3. ed. [S.l.]: Springer Publishing Company, Incorporated, 2008. ISBN 0387339981, 9780387339986. Citado na página 51.

MALDONADO, J. C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO, M. E. **Evaluating N-Selective Mutation for C Programs: Unit and Integration testing**. [S.l.: s.n.], 2001. Citado na página 36.

MEDEIROS, B. K. **Estudo e extensão da metodologia DAMICORE para tarefas de classificação**. Dissertação (Mestrado) — Universidade de São Paulo, http://www.teses.usp.br/teses/disponiveis/55/55134/tde-16112016-170837/, 2016. Citado nas páginas 55, 80 e 134.

MI, N.; CHERKASOVA, L.; OZONAT, K.; SYMONS, J.; SMIRNI, E. Analysis of application performance and its change via representative application signatures. In: IEEE. **Network Operations and Management Symposium, 2008. NOMS 2008. IEEE**. [S.l.], 2008. p. 216–223. Citado nas páginas 47 e 134.

MICROSOFT. **Windows Performance Monitor**. 2017. Software de monitoramento de recursos. Disponível em: <https://technet.microsoft.com/en-us/library/cc749249(v=ws.11).aspx>. Citado nas páginas 62 e 80.

MONTES, V. S. **Network Telemetry Benchmark**. 2019. Códigos-fonte dos aplicativos YATServer/YATClient e ferramentas auxiliares. Disponível em: <https://gitlab.com/vmontes/networktelemetry>. Citado na página 81.

____. **Tricorder Framework**. 2019. Conjunto de ferramentas e scripts usados na aplicação do método Tricorder. Disponível em: <https://gitlab.com/vmontes/tricorder>. Citado na página 81.

NANAVATI, J.; WU, F.; HARMAN, M.; JIA, Y.; KRINKE, J. Mutation testing of memory-related operators. In: IEEE. **Software testing, verification and validation workshops (ICSTW), 2015 IEEE eighth international conference on**. [S.l.], 2015. p. 1–10. Citado na página 38.

PINTO, R. de S.; DELBEM, A. C. B.; MONACO, F. J. Characterization of runtime resource usage from analysis of binary executable programs. **Applied Soft Computing**, v. 71, p. 1133 – 1152, 2018. ISSN 1568-4946. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1568494617307779>. Citado nas páginas 80 e 135.

RODOLA, G. **PSUtil library documentation**. 2019. Biblioteca de monitoramento de recursos para python. Disponível em: <http://pythonhosted.org/psutil/>. Citado nas páginas 58, 63 e 79.

SANCHES, A.; CARDOSO, J. M. P.; DELBEM, A. C. B. Identifying merge-beneficial software kernels for hardware implementation. In: **Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs**. Washington, DC, USA: IEEE Computer Society, 2011. (RECONFIG '11), p. 74–79. ISBN 978-0-7695-4551-6. Disponível em: <http://dx.doi.org/10.1109/ReConFig.2011.51>. Citado nas páginas 51, 52 e 54.

SHKARIN, D. Ppm: One step to practicality. In: IEEE. **Data Compression Conference, 2002. Proceedings. DCC 2002**. [S.l.], 2002. p. 202–211. Citado na página 52.

SOARES, A.; SANTOS, V.; TOLEDO, C.; OSÓRIO, F.; DELBEM, A. Nd-ncd: Environmental characteristics recognition and novelty detection for mobile robots control and navigation. In: ____. **Robotics: 12th Latin American Robotics Symposium and Third Brazilian Symposium on Robotics, LARS 2015/SBR 2015, Uberlândia, Brazil, October 28 - November 1, 2015, Revised Selected Papers**. Cham: Springer International Publishing, 2016. p. 192–209. ISBN 978-3-319-47247-8. Disponível em: <https://doi.org/10.1007/978-3-319-47247-8_12>. Citado na página 52.

SOMMERVILLE, I. **Engenharia de software**. Pearson Prentice Hall, 2011. ISBN 9788579361081. Disponível em: <https://books.google.com.br/books?id=H4u5ygAACAAJ>. Citado na página 33.

SRIVASTAVA, A. N.; SCHUMANN, J. Software health management: A necessity for safety critical systems. **Innov. Syst. Softw. Eng.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 9, n. 4, p. 219–233, dez. 2013. ISSN 1614-5046. Disponível em: <http://dx.doi.org/10.1007/s11334-013-0212-0>. Citado na página 27.

APÊNDICE

A

DEFEITOS

Arquivo de definições do defeitos. Cada defeito pode ser inserido ao descomentar sua respectiva linha, criando um mutante para o mesmo.

```
1 #pragma once
2 #include "defs.h"
3 #define HAS_FAULT
4 #ifdef HAS_FAULT
5 // MUTANTS
6 /// NETWORK PACKETS FAULTS
7 //#define FAULT_NETWORK_PKT_WINDOWSIZE_ACQNUM_SWAP
8 /// BIN PAYLODAD PROCESSING FAULTS
9 //#define FAULT_BINPKTPROCESSOR_PROCESS_CLEANUP
10 //#define FAULT_BINPKTPROCESSOR_PROCESS_SLEEP
11 /// CSV PAYLODAD PROCESSING FAULTS
12 //#define FAULT_CSVPKTPROCESSOR_PROCESS_INFINITE
13 /// PKT PROCESSOR FAULTS
14 //#define FAULT_PROCESSOR_BUSY_UNLOCK
15 /// DATAPROCESSOR PROCESSING FAULTS
16 //#define FAULT_DATAPROCESSOR_RUN_NO_BREAK
17 //#define FAULT_DATAPROCESSOR_MONOLITHIC
18 #endif
```

APÊNDICE

B

DEFINIÇÕES DE DADOS

Arquivo de definições de estruturas de dados para comunicação cliente-servidor. Um dos defeitos se encontra na estrutura *NetworkPktRecvBIN*, que ao ser inserido cria o mutante *SWAP*.

```
1 #pragma once
2 #include "faults.h"
3 #include <stdint.h>
4 enum NetworkPktTypeEnum
5 {
6   CSV_REPORT=0,
7   BIN_CVT,
8   PKT_TYPE_COUNT
9 };
10 #pragma pack(push,1)
11 struct NetworkPktSendCSV
12 {
13   int32_t source_id;
14   int32_t pkt_len;
15   int32_t payload_len;
16   int32_t hash;
17   int32_t report_id;
18   int32_t timestamp;
19   char type;
20   char * payload;
21 };
22 typedef NetworkPktSendCSV NetworkPktRecvCSV;
23 struct NetworkPktSendBIN
24 {
```

```
25   int32_t source_id;
26   int32_t pkt_len;
27   int32_t payload_len;
28   int32_t timestamp;
29   int16_t window_size_msec;
30   int16_t samples_per_channel_id;
31   int16_t samples_interval_usec;
32   int16_t acq_num;
33   char type;
34   char * payload;
35 };
36 #if defined(FAULT_NETWORK_PKT_WINDOWSIZE_ACQNUM_SWAP)
37 struct NetworkPktRecvBIN
38 {
39   int32_t source_id;
40   int32_t pkt_len;
41   int32_t payload_len;
42   int32_t timestamp;
43   int16_t acq_num;
44   int16_t samples_per_channel_id;
45   int16_t samples_interval_usec;
46   int16_t window_size_msec;
47   char type;
48   char * payload;
49 };
50 #else
51 typedef NetworkPktSendBIN NetworkPktRecvBIN;
52 #endif
53 struct NetworkPktSendRAW
54 {
55   int32_t source_id;
56   int32_t pkt_len;
57   int32_t payload_len;
58   int32_t reserved1;
59   int32_t reserved2;
60   int32_t reserved3;
61   char type;
62   char * payload;
63 };
```

```
64 typedef NetworkPktSendRAW NetworkPktRecvRAW;
65 #endif
66 #pragma pack(pop)
```

APÊNDICE

C

CONFIGURAÇÕES DE CARGA

Arquivo de configurações de leitura e transmissão de dados do cliente para o servidor. Esse arquivo é lido pelo *YATClient* e determina quais dados serão enviados, seu intervalo entre outros parâmetros. Esse arquivo é usado como definição de cargas de trabalho no projeto.

```
1  "configurations":
2  [
3    {
4      "tag": "BINCSVM1",
5      "output_path" : "C:/datalab/network/output/",
6      "inputs":
7      [
8        {
9          "type": "BIN",
10         "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input1.bin",
11         "x": 1000,
12         "y": 500,
13         "interval": 200,
14         "window_size_msec" : 2000
15       },
16       {
17         "type": "CSV",
18         "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input1.csv",
19         "x": 1000,
20         "y": 1000,
21         "interval": 4000,
22         "id": 42
23       }
```

```
24        ]
25      },
26      {
27        "tag": "BINCSVM2",
28        "output_path" : "C:/datalab/network/output/",
29        "inputs":
30        [
31          {
32            "type": "BIN",
33            "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input2.bin",
34            "x": 1000,
35            "y": 200,
36            "interval": 100,
37            "window_size_msec" : 2000
38          },
39          {
40            "type": "CSV",
41            "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input2.csv",
42            "x": 1000,
43            "y": 1000,
44            "interval": 4000,
45            "id": 42
46          }
47        ]
48      },
49      {
50        "tag": "BINCSVM3",
51        "output_path" : "C:/datalab/network/output/",
52        "inputs":
53        [
54          {
55            "type": "BIN",
56            "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input3.bin",
57            "x": 1000,
58            "y": 500,
59            "interval": 200,
60            "window_size_msec" : 4000
61          },
62          {
```

```
63            "type": "CSV",
64            "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input3.csv",
65            "x": 1000,
66            "y": 1000,
67            "interval": 4000,
68            "id": 42
69          }
70        ]
71      },
72      {
73        "tag": "BINCSVH1",
74        "output_path" : "C:/datalab/network/output/",
75        "inputs":
76        [
77          {
78            "type": "BIN",
79            "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input4.bin",
80            "x": 1000,
81            "y": 500,
82            "interval": 200,
83            "window_size_msec" : 2000
84          },
85          {
86            "type": "CSV",
87            "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input4.csv",
88            "x": 1000,
89            "y": 2000,
90            "interval": 4000,
91            "id": 42
92          }
93        ]
94      },
95      {
96        "tag": "BINCSVH2",
97        "output_path" : "C:/datalab/network/output/",
98        "inputs":
99        [
100         {
101           "type": "BIN",
```

```
102             "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input5.bin",
103             "x": 500,
104             "y": 100,
105             "interval": 50,
106             "window_size_msec" : 2000
107           },
108           {
109             "type": "CSV",
110             "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input5.csv",
111             "x": 1000,
112             "y": 1000,
113             "interval": 4000,
114             "id": 42
115           }
116         ]
117       },
118       {
119         "tag": "BINCSVH3",
120         "output_path" : "C:/datalab/network/output/",
121         "inputs":
122         [
123           {
124             "type": "BIN",
125             "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input6.bin",
126             "x": 2000,
127             "y": 1000,
128             "interval": 500,
129             "window_size_msec" : 2000
130           },
131           {
132             "type": "CSV",
133             "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input6.csv",
134             "x": 1000,
135             "y": 2000,
136             "interval": 4000,
137             "id": 42
138           }
139         ]
140       },
```

```
141     {
142       "tag": "BINCSVL1",
143       "output_path" : "C:/datalab/network/output/",
144       "inputs":
145       [
146         {
147           "type": "BIN",
148           "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input7.bin",
149           "x": 500,
150           "y": 500,
151           "interval": 200,
152           "window_size_msec" : 2000
153         },
154         {
155           "type": "CSV",
156           "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input7.csv",
157           "x": 1000,
158           "y": 1000,
159           "interval": 4000,
160           "id": 42
161         }
162       ]
163     },
164     {
165       "tag": "BINCSVL2",
166       "output_path" : "C:/datalab/network/output/",
167       "inputs":
168       [
169         {
170           "type": "BIN",
171           "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input8.bin",
172           "x": 500,
173           "y": 500,
174           "interval": 200,
175           "window_size_msec" : 2000
176         },
177         {
178           "type": "CSV",
179           "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input8.csv",
```

```
180             "x": 1000,
181             "y": 500,
182             "interval": 4000,
183             "id": 42
184           }
185         ]
186       },
187       {
188         "tag": "BINCSVL3",
189         "output_path" : "C:/datalab/network/output/",
190         "inputs":
191         [
192           {
193             "type": "BIN",
194             "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input9.bin",
195             "x": 10000,
196             "y": 10,
197             "interval": 200,
198             "window_size_msec" : 2000
199           },
200           {
201             "type": "CSV",
202             "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input9.csv",
203             "x": 50,
204             "y": 1000,
205             "interval": 4000,
206             "id": 42
207           }
208         ]
209       },
210       {
211         "tag": "BINL1",
212         "output_path" : "C:/datalab/network/output/",
213         "inputs":
214         [
215           {
216             "type": "BIN",
217             "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input10.bin",
218             "x": 500,
```

```
219             "y": 500,
220             "interval": 200,
221             "window_size_msec" : 2000
222           }
223         ]
224       },
225       {
226
227         "tag": "BINM1",
228         "output_path" : "C:/datalab/network/output/",
229         "inputs":
230         [
231           {
232             "type": "BIN",
233             "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input11.bin",
234             "x": 1000,
235             "y": 500,
236             "interval": 200,
237             "window_size_msec" : 2000
238           }
239         ]
240       },
241       {
242         "tag": "BINH1",
243         "output_path" : "C:/datalab/network/output/",
244         "inputs":
245         [
246           {
247             "type": "BIN",
248             "path": "C:\\DATALAB/NETWORK\\WORKLOAD\\input12.bin",
249             "x": 2000,
250             "y": 500,
251             "interval": 200,
252             "window_size_msec" : 2000
253           }
254         ]
255       },
256       {
257         "tag": "CSVL1",
```

```
258       "output_path" : "C:/datalab/network/output/",
259       "inputs":
260       [
261         {
262           "type": "CSV",
263           "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input10.csv"
      ,
264           "x": 50,
265           "y": 1000,
266           "interval": 4000,
267           "id": 42
268         }
269       ]
270     },
271     {
272       "tag": "CSVM1",
273       "output_path" : "C:/datalab/network/output/",
274       "inputs":
275       [
276         {
277           "type": "CSV",
278           "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input11.csv"
      ,
279           "x": 1000,
280           "y": 1000,
281           "interval": 4000,
282           "id": 42
283         }
284       ]
285     },
286     {
287       "tag": "CSVH1",
288       "output_path" : "C:/datalab/network/output/",
289       "inputs":
290       [
291         {
292           "type": "CSV",
293           "path": "C:\\DATALAB\\NETWORK\\WORKLOAD\\input12.csv"
      ,
```

```
294             "x": 1000,
295             "y": 2000,
296             "interval": 4000,
297             "id": 42
298           }
299         ]
300       }
301     ]
302 }
```

APÊNDICE

D

BIBLIOTECA DE PROCESSAMENTO DE DADOS

Arquivos de código-fonte da biblioteca de processamento de dados. Dos 7 defeitos selecionados, 6 estão no código da biblioteca, e 1 nas definições de estruturas de comunicação.

```
1 // ********************************************
2 // Data Processing Lib: DataProcessor.h
3 // ********************************************
4 #pragma once
5 #include "pkt.h"
6 #include <thread>
7 #include <condition_variable>
8 #include <mutex>
9 #include <queue>
10 #include <vector>
11 using namespace std;
12 class CBinPktProcessor;
13 class CCsvPktProcessor;
14 class CDataProcessor
15 {
16 public:
17    CDataProcessor();
18    ~CDataProcessor();
19    void Add(NetworkPktRecvRAW * pkt);
20    void Remove();
21    void Run();
22 private:
```

```
23    mutex m_proc_mtx;
24    condition_variable m_proc_cv;
25    bool m_loop;
26    queue<NetworkPktRecvRAW*> m_queue;
27    vector<CCsvPktProcessor*> m_vCSVProcessor;
28    vector<CBinPktProcessor*> m_vBinProcessor;
29    thread m_ProcThread;
30 };
```

```
1  // ********************************************
2  // Data Processing Lib: DataProcessor.cpp
3  // ********************************************
4  #include "faults.h"
5  #include "DataProcessor.h"
6  #include "CsvProcessor.h"
7  #include "BinProcessor.h"
8  #include <thread>
9  #include <chrono>
10 #include <assert.h>
11 #include <mutex>
12 #include <iostream>
13 #ifdef FAULT_DATAPROCESSOR_MONOLITHIC
14 #pragma message("-----------------------------------------------")
15 #pragma message("FAULT_DATAPROCESSOR_MONOLITHIC")
16 #pragma message("-----------------------------------------------")
17 #define PROCESSORS_NUM_BIN (1)
18 #define PROCESSORS_NUM_CSV (1)
19 #else
20 #define PROCESSORS_NUM_BIN (2)
21 #define PROCESSORS_NUM_CSV (2)
22 #endif
23 void StreamThread(void * p)
24 {
25    CDataProcessor * s = (CDataProcessor*)p;
26    s->Run();
27 }
28 CDataProcessor::CDataProcessor()
29 {
30    m_ProcThread = thread(StreamThread, (void*)this);
```

```
31
32    for (int i = 0; i < PROCESSORS_NUM_BIN; i++)
33      m_vBinProcessor.push_back(new CBinPktProcessor());
34
35    for (int i = 0; i < PROCESSORS_NUM_CSV; i++)
36      m_vCSVProcessor.push_back(new CCsvPktProcessor());
37 }
38 CDataProcessor::~CDataProcessor()
39 {
40    while (!m_queue.empty())
41      this_thread::sleep_for(chrono::milliseconds(1));
42    m_loop = false;
43    m_proc_cv.notify_all();
44    m_ProcThread.join();
45    for (int i = 0; i < PROCESSORS_NUM_BIN; i++)
46      delete m_vBinProcessor[i];
47    for (int i = 0; i < PROCESSORS_NUM_CSV; i++)
48      delete m_vCSVProcessor[i];
49    m_vBinProcessor.clear();
50    m_vCSVProcessor.clear();
51 }
52 void CDataProcessor::Add(NetworkPktRecvRAW * pkt)
53 {
54    char * rawbuff = new char[pkt->pkt_len+1];
55    memset(rawbuff, 0, pkt->pkt_len);
56    memcpy(rawbuff, pkt, pkt->pkt_len);
57    NetworkPktRecvRAW* queuepkt = (NetworkPktRecvRAW*)rawbuff;
58    queuepkt->payload = (char*)(&(queuepkt->payload) + 1);
59    queuepkt->payload[queuepkt->payload_len] = '\0';
60    m_proc_mtx.lock();
61    m_queue.push(queuepkt);
62    m_proc_mtx.unlock();
63    m_proc_cv.notify_one();
64 }
65 void CDataProcessor::Remove()
66 {
67    delete m_queue.front();
68    m_queue.pop();
69 }
```

```
70 void CDataProcessor::Run()
71 {
72   m_loop = true;
73   std::unique_lock<std::mutex> lck(m_proc_mtx);
74
75   while (m_loop)
76   {
77     m_proc_cv.wait(lck);
78     while (!m_queue.empty())
79     {
80       NetworkPktRecvRAW * pkt = m_queue.front();
81
82       switch (pkt->type)
83       {
84         case CSV_REPORT:
85         {
86           int n = 0;
87           bool locked = false;
88           CCsvPktProcessor * proc = nullptr;
89           do
90           {
91             for (n = 0; n < PROCESSORS_NUM_CSV; n++)
92             {
93               proc = m_vCSVProcessor[n];
94               if (!proc->m_isBusy)
95               {
96                 locked = true;
97                 break;
98               }
99             }
100            if (!locked)
101              this_thread::sleep_for(chrono::milliseconds(1));
102
103          } while (!locked);
104
105          NetworkPktRecvCSV * csvpkt = (NetworkPktRecvCSV *)pkt
     ;
106          proc->Init(csvpkt);
107          proc->Process();
```

```
108
109 #ifdef FAULT_DATAPROCESSOR_RUN_NO_BREAK
110 #pragma message("————————————————————————————————————————")
111 #pragma message("FAULT_NOBREAK")
112 #pragma message("————————————————————————————————————————")
113 #else
114             break;
115 #endif
116         }
117         case BIN_CVT:
118         {
119             int n = 0;
120             bool locked = false;
121             CBinPktProcessor * proc = nullptr;
122             do
123             {
124               for (n = 0; n < PROCESSORS_NUM_BIN; n++)
125               {
126                 proc = m_vBinProcessor[n];
127                 if (!proc->m_isBusy)
128                 {
129                   locked = true;
130                   break;
131                 }
132               }
133               if (!locked)
134                 this_thread::sleep_for(chrono::milliseconds(100))
    ;
135
136             } while (!locked);
137
138             NetworkPktRecvBIN * binpkt = (NetworkPktRecvBIN*)pkt;
139             proc->Process(binpkt);
140             break;
141         }
142         default:
143             break;
144       }
145
```

```
146         Remove();
147       }
148     }
149 }
```

```
 1 // ********************************************
 2 // Data Processing Lib: BinProcessor.h
 3 // ********************************************
 4 #pragma once
 5 #include "pkt.h"
 6 #include "DataProcessor.h"
 7 #include "BinLockResults.h"
 8 #include <thread>
 9 #include <queue>
10 #include <list>
11 #include <unordered_map>
12 #include <mutex>
13 #include <map>
14 using namespace std;
15 struct sample_t
16 {
17   double timestamp;
18   double mean;
19   double median;
20   double mode;
21   double max;
22   double min;
23 };
24 struct snapshot_t
25 {
26   double base_timestamp;
27   map<int/*id*/, sample_t/*values*/> m_values;
28 };
29 struct cvt_t
30 {
31   list<snapshot_t*> m_snapshots;
32 };
33 class CBinPktProcessor
34 {
```

```
35    friend class CDataProcessor;
36    friend class CBinLockResults;
37  public:
38    // External interface
39    static shared_ptr<CBinLockResults> GetCVTWindow(shared_ptr<
       cvt_t> & cvt);
40    CBinPktProcessor();
41    virtual ~CBinPktProcessor();
42  // For use of CDataProcessor class
43  protected:
44    void Process(NetworkPktRecvBIN * pkt);
45    bool DataReady();
46    bool m_isBusy;
47    mutex m_busy;
48  private:
49    static mutex results_mtx;
50    static mutex done_mtx;
51    static condition_variable done_cv;
52    static shared_ptr<cvt_t> CVT;
53    static NetworkPktRecvBIN Header;
54    thread m_Worker;
55    void Cleanup();
56    void ProcessData();
57    char * m_pkt;
58    int m_pktlen;
59    char * m_payload;
60    int m_payload_len;
61  };
```

```
1  // ********************************************
2  // Data Processing Lib: BinProcessor.cpp
3  // ********************************************
4  #include "faults.h"
5  #include "BinProcessor.h"
6  #include "DataProcessor.h"
7  #include <queue>
8  #include <list>
9  #include <string>
10 #include <iostream>
```

```
11 using namespace std;
12 shared_ptr<cvt_t> CBinPktProcessor::CVT = nullptr;;
13 NetworkPktRecvBIN CBinPktProcessor::Header = { 0 };
14 mutex         CBinPktProcessor::done_mtx;
15 mutex         CBinPktProcessor::results_mtx;
16 condition_variable   CBinPktProcessor::done_cv;
17 CBinPktProcessor::CBinPktProcessor() : m_pkt(nullptr), m_pktlen
     (0)
18 {
19 }
20 CBinPktProcessor::~CBinPktProcessor()
21 {
22   if (CVT)
23   {
24     for (auto it : CVT->m_snapshots)
25     {
26       it->m_values.clear();
27       delete it++;
28     }
29     CVT = nullptr;
30   }
31 }
32 void CBinPktProcessor::Cleanup()
33 {
34   if (m_pkt)
35     delete[] m_pkt;
36   m_pkt = nullptr;
37 }
38 void CBinPktProcessor::Process(NetworkPktRecvBIN * pkt)
39 {
40   if (m_Worker.joinable())
41     m_Worker.join();
42   m_pktlen = pkt->pkt_len;
43   m_pkt = new char[m_pktlen + 1];
44   memcpy(m_pkt, pkt, pkt->pkt_len);
45   memcpy(&Header, (NetworkPktRecvBIN*)m_pkt, sizeof(
      NetworkPktRecvBIN));
46   m_payload = m_pkt + (sizeof(NetworkPktRecvBIN));
47   m_payload_len = pkt->payload_len;
```

```
48    m_payload[m_payload_len] = '\0';
49    m_Worker = thread(&CBinPktProcessor::ProcessData, this);
50  }
51  void CBinPktProcessor::ProcessData()
52  {
53    m_busy.lock();
54    m_isBusy = true;
55    NetworkPktRecvBIN header = CBinPktProcessor::Header;
56    char * p = m_payload;
57    snapshot_t * snapshot = new snapshot_t;
58    snapshot->base_timestamp = header.timestamp_ms;
59    while ((p - m_payload) < m_payload_len)
60    {
61      int64_t * id = (int64_t*)p;
62      p += sizeof(int64_t);
63      if (snapshot->m_values.find(*id) == snapshot->m_values.end
      ())
64      {
65        sample_t & sample = snapshot->m_values[*id];
66        memset(&sample, 0, sizeof(sample_t));
67        sample.min = INT32_MAX;
68        sample.max = INT32_MIN;
69        unordered_map<double, int> mode_count;
70        for (int i = 0; i < header.samples_per_channel_id; i++)
71        {
72          sample.timestamp = snapshot->base_timestamp;
73          double val = *(double*)p;
74          sample.mean += val;
75          if (i == (int)(header.samples_per_channel_id / 2.0))
76            sample.median = val;
77          if (sample.max < val)
78            sample.max = val;
79          if (sample.min > val)
80            sample.min = val;
81          if (mode_count.find(val) == mode_count.end())
82            mode_count[val] = 0;
83          else
84            mode_count[val]++;
85          p += sizeof(int64_t);
```

```
86          if ((uint64_t)(p - m_payload) >= m_payload_len)
87            break;
88        }
89        sample.mean /= (double)header.samples_per_channel_id;
90        double max_count = 0;
91        double mode = 0;
92        for (auto & it : mode_count)
93        {
94          if (max_count < it.second)
95          {
96            max_count = it.second;
97            mode = it.first;
98          }
99        }
100       sample.mode = mode;
101     }
102   }
103   bool notify = false;
104   // locks to update output data
105   done_mtx.lock();
106   {
107     CBinLockResults lock;
108     if (CVT == nullptr)
109       CVT = make_shared<cvt_t>();
110     CVT->m_snapshots.push_back(snapshot);
111 #ifdef FAULT_BINPKTPROCESSOR_PROCESS_CLEANUP
112 #pragma message("-----------------------------------------------")
113 #pragma message("FAULT_BINPKTPROCESSOR_PROCESS_CLEANUP")
114 #pragma message("-----------------------------------------------")
115 #else
116     Cleanup();
117 #endif
118 #ifdef FAULT_BINPKTPROCESSOR_PROCESS_SLEEP
119     this_thread::sleep_for(chrono::milliseconds(200));
120 #pragma message("-----------------------------------------------")
121 #pragma message("FAULT_BINPKTPROCESSOR_PROCESS_SLEEP")
122 #pragma message("-----------------------------------------------")
123 #endif
124     notify = DataReady();
```

```
125   }
126   done_mtx.unlock();
127   if (notify)
128   {
129     done_cv.notify_one();
130   }
131   m_busy.unlock();
132   m_isBusy = false;
133 }
134 bool CBinPktProcessor::DataReady()
135 {
136   NetworkPktRecvBIN header = CBinPktProcessor::Header;
137   bool ret = false;
138   if (CVT)
139   {
140     if (CVT->m_snapshots.size() > 2)
141     {
142       double basetimestamp = header.timestamp_ms - (header.
      window_size_msec);
143       auto snap = CVT->m_snapshots.begin();
144       while (snap != CVT->m_snapshots.end())
145       {
146         if ((*snap)->base_timestamp < basetimestamp)
147         {
148           (*snap)->m_values.clear();
149           snapshot_t * st = *snap;
150           snap++;
151           CVT->m_snapshots.remove(st);
152           delete st;
153           ret = true;
154         }
155         else
156           snap++;
157       }
158     }
159   }
160   return ret;
161 }
162  shared_ptr<CBinLockResults> CBinPktProcessor::GetCVTWindow(
```

```
    shared_ptr<cvt_t> & cvt)
163 {
164   condition_variable & cv = done_cv;
165   mutex & mtx = done_mtx;
166   std::unique_lock<std::mutex> lck(mtx);
167   cv.wait(lck);
168   shared_ptr<CBinLockResults> lock = make_shared<
      CBinLockResults>();
169   cvt = CVT;
170   return lock;
171 }
```

```
 1 // ********************************************
 2 // Data Processing Lib: CsvProcessor.h
 3 // ********************************************
 4 #pragma once
 5 #include "pkt.h"
 6 #include "faults.h"
 7 #include "DataProcessor.h"
 8 #include <string>
 9 #include <cinttypes>
10 #include <vector>
11 #include <unordered_map>
12 #include <mutex>
13 using namespace std;
14 enum FieldType
15 {
16   FIELD_TYPE_VALUE = 0,
17   FIELD_TYPE_SEPARATOR,
18   FIELD_TYPE_EOL,
19   FIELD_TYPE_EOF
20 };
21 struct GridLine
22 {
23   vector<string> line;
24 };
25 struct tables_t
26 {
27   vector<string> m_Headers;
```

```
28    vector<GridLine> m_Table;
29    vector<unordered_map<string, string>> m_ColumnsDatamap;
30 };
31 class CCsvPktProcessor
32 {
33    friend class CDataProcessor;
34 public:
35    // External interface
36    static const shared_ptr<tables_t> GetCurrentReport(int id);
37    CCsvPktProcessor();
38    ~CCsvPktProcessor();
39 // For use of CDataProcessor class
40 protected:
41    void Cleanup();
42    void Process();
43    void Init(NetworkPktRecvCSV * pkt);
44    bool m_initialized;
45    bool m_isBusy;
46    mutex m_busy;
47 private:
48    static mutex done_mtx;
49    static condition_variable done_cv;
50    static unordered_map<int, shared_ptr<tables_t>>
       CurrentReports;
51    char * ParseLine(shared_ptr<tables_t> NewData, char * data,
       GridLine& gline, size_t& header_len);
52    char * m_pkt;
53    char * m_payload;
54    int m_pktlen;
55    int m_payload_len;
56    int m_Id;
57    NetworkPktRecvCSV * m_Header;
58    void ProcessData();
59 };
```

Data Processing Lib: CsvProcessor.cpp

```
1 // ********************************************
2 // Data Processing Lib: CsvProcessor.cpp
3 // ********************************************
```

```
4 #include "faults.h"
5 #include "CsvProcessor.h"
6 #include "DataProcessor.h"
7 #include "pkt.h"
8 #include <assert.h>
9 #include <thread>
10 #include <mutex>
11 #define IS_SEPARATOR(x) ((x==',')||(x==';'))
12 #define IS_EOL(x) ((x=='\r')||(x=='\n'))
13 #define VALID(x) ((x-m_payload) < m_payload_len)
14 unordered_map<int,shared_ptr<tables_t>> CCsvPktProcessor::
    CurrentReports;
15 mutex           CCsvPktProcessor::done_mtx;
16 condition_variable     CCsvPktProcessor::done_cv;
17 CCsvPktProcessor::CCsvPktProcessor() : m_pkt(nullptr),
    m_initialized(false), m_pktlen(0), m_isBusy(false)
18 {
19 }
20 CCsvPktProcessor::~CCsvPktProcessor()
21 {
22   Cleanup();
23 }
24 void CCsvPktProcessor::Init(NetworkPktRecvCSV * pkt)
25 {
26   m_pktlen = pkt->pkt_len;
27   m_pkt = new char[m_pktlen + 1];
28   memcpy(m_pkt, pkt, pkt->pkt_len);
29   m_Header = (NetworkPktRecvCSV*)m_pkt;
30   m_payload = m_pkt + (sizeof(NetworkPktRecvCSV));
31   m_payload_len = pkt->payload_len;
32   m_payload[m_payload_len] = '\0';
33   m_Id = pkt->source_id;
34   m_initialized = true;
35 }
36 void CCsvPktProcessor::Process()
37 {
38   thread(&CCsvPktProcessor::ProcessData, this).detach();
39 }
40 void CCsvPktProcessor::Cleanup()
```

```
41 {
42   if (m_pkt)
43     delete[] m_pkt;
44   m_pkt = nullptr;
45 }
46 char * CCsvPktProcessor::ParseLine(shared_ptr<tables_t> NewData
     , char * data, GridLine& gline, size_t& header_len)
47 {
48   char * start = data;
49   char * end = data;
50   int col = 0;
51   while ((!IS_EOL(*end)) && (VALID(end)))
52   {
53     if (IS_SEPARATOR(*end) || IS_EOL(*end))
54     {
55       string gdata;
56       gdata.assign(start, end);
57       start = ++end;
58       if (header_len == 0)
59       {
60         NewData->m_Headers.push_back(gdata);
61         unordered_map<string, string> s;
62         NewData->m_ColumnsDatamap.push_back(s);
63       }
64       else
65       {
66         gline.line.push_back(gdata);
67         if (!gdata.empty())
68           NewData->m_ColumnsDatamap[col][gdata] = gdata;
69         col++;
70       }
71     }
72     else
73       end++;
74   }
75   if (start != (end))
76   {
77     string gdata;
78     gdata.assign(start, end);
```

```
79      start = ++end;
80      if (header_len == 0)
81      {
82        NewData->m_Headers.push_back(gdata);
83        unordered_map<string, string> s;
84        NewData->m_ColumnsDatamap.push_back(s);
85      }
86      else
87      {
88        gline.line.push_back(gdata);
89        col++;
90      }
91    }
92    if (header_len == 0)
93    {
94      header_len = NewData->m_Headers.size();
95    }
96    else
97    {
98      while (col < header_len)
99      {
100       string gdata;
101       gline.line.push_back(gdata);
102     }
103   }
104   header_len = NewData->m_Headers.size();
105   return end;
106 }
107 void CCsvPktProcessor::ProcessData()
108 {
109   m_busy.lock();
110   m_isBusy = true;
111   shared_ptr<tables_t> NewData = make_shared<tables_t>();
112   size_t header_len = 0;
113   char * end = m_payload;
114   char * start = end;
115   int line = 0;
116   int col = 0;
117   while (VALID(end))
```

```
118   {
119     GridLine line;
120     if (header_len == 0)
121     {
122       end = ParseLine(NewData, end, line, header_len);
123     }
124     else
125     {
126       end = ParseLine(NewData, end, line, header_len);
127       NewData->m_Table.push_back(line);
128     }
129
130     while (VALID(end) && (IS_EOL(*end)))
131       end++;
132   }
133   CurrentReports[m_Header->report_id] = NewData;
134   Cleanup();
135 #ifdef FAULT_PROCESSOR_BUSY_UNLOCK
136 #pragma message("-------------------------------------------------")
137 #pragma message("FAULT_UNLOCK")
138 #pragma message("-------------------------------------------------")
139 #else
140   m_busy.unlock();
141 #endif
142   m_isBusy = false;
143   done_cv.notify_all();
144 #ifdef FAULT_CSVPKTPROCESSOR_PROCESS_INFINITE
145 #pragma message("-------------------------------------------------")
146 #pragma message("FAULT_INFINITE")
147 #pragma message("-------------------------------------------------")
148   while (1);
149 #endif
150 }
151 const shared_ptr<tables_t> CCsvPktProcessor::GetCurrentReport(
      int id)
152 {
153   bool done = false;
154   while (!done)
155   {
```

```
156       condition_variable & cv = done_cv;
157       mutex & mtx = done_mtx;
158       std::unique_lock<std::mutex> lck(mtx);
159       cv.wait(lck);
160       if (CurrentReports.find(id) != CurrentReports.end())
161       {
162         const shared_ptr<tables_t> DumpData = CurrentReports[id];
163         return DumpData;
164       }
165     }
166 }
```


ICMC
USP
SÃO CARLOS