UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AVALIAÇÃO DO DESEMPENHO DE TÉCNICAS DE PROGRAMAÇÃO ORIENTADA A ASPECTOS

RODRIGO FERNANDES GOMES DA SILVA

Uberlândia - Minas Gerais
2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RODRIGO FERNANDES GOMES DA SILVA

AVALIAÇÃO DO DESEMPENHO DE TÉCNICAS DE PROGRAMAÇÃO ORIENTADA A ASPECTOS

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador:
Prof. Dr. Michel dos Santos Soares

Co-orientador:
Prof. Dr. Marcelo de Almeida Maia

Uberlândia, Minas Gerais
2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Avaliação do Desempenho de Técnicas de Programação Orientada a Aspectos**" por **Rodrigo Fernandes Gomes da Silva** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 27 de Maio de 2014

Orientador: _____
Prof. Dr. Michel dos Santos Soares
Universidade Federal de Uberlândia

Co-orientador: _____
Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Prof. Dr. Flávio de Oliveira Silva
Universidade Federal de Uberlândia

_____
Prof. Dr. Eduardo Figueiredo
Universidade Federal de Minas Gerais

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Maio de 2014

Autor:	**Rodrigo Fernandes Gomes da Silva**
Título:	**Avaliação do Desempenho de Técnicas de Programação Orientada a Aspectos**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

_____

Autor

Dedicatória

Aos meus pais Wanderley e Romilda.

Agradecimentos

Agradeço...

A Deus, por minha vida.

A meus pais Wanderley e Romilda pela confiança, apoio, carinho e amor incondicional em todos os momentos.

Aos meus amigos que se mostraram companheiros e, diretamente ou indiretamente, contribuíram para a realização deste trabalho,

A UFU pelo apoio material,

Aos professores Michel dos Santos Soares e Marcelo de Almeida Maia pelo profissionalismo, apoio, paciência, amizade e orientação em todos os momentos da realização deste trabalho,

E aos professores Ernando Antonio dos Reis, José Francisco Ribeiro e João Jorge Ribeiro Damasceno, que me possibilitaram a realização desse trabalho.

"A vida deve ser uma constante educação."
(Gustave Flaubert)

Resumo

A Programação Orientada a Aspectos (POA) foi proposta com o principal objetivo de tratar um princípio importante da qualidade de software, que é a modularização. A idéia básica do paradigma é capturar interesses transversais e tratá-los de forma independente do restante da aplicação. POA surgiu como um complemento à Programação Orientada a Objetos (POO), dando origem a uma série de estudos empíricos sobre esse novo paradigma. Uma variedade de métricas foram aplicadas sobre POA com o intuito de evidenciar seus benefícios ou problemas, no entanto não há consenso sobre o impacto no desempenho a respeito do uso das técnicas de POA para tratar os interesses transversais nos sistemas. Para tentar identificar se tais técnicas de POA causam impacto no desempenho dos sistemas, esse estudo propôs um experimento controlado sobre um sistema acadêmico da Universidade Federal de Uberlândia. O experimento teve o propósito de avaliar fatores relacionadas a POA que podem influenciar no desempenho dos sistemas. De acordo com os resultados obtidos nesse estudo, fatores como o *weaver*, número de *join points*, tipo de *advice* e número de línhas de código (LOC) causam influência no desempenho dos sistemas quando se trata de POA. Essa influência, no entanto, se mostrou insignificante para aplicações *web* comuns que fazem acesso a banco de dados e possuem *framework MVC*. Os resultados também apontam que, especificamente para o processo de *load-time weaving*, o estágio de carregamento das classes pode prejudicar o desempenho de aplicações que operam no mesmo ambiente. Esses resultados podem ser utilizados por arquitetos e desenvolvedores para influenciar decisões relacionadas a projetos de arquitetura de sistemas envolvendo POA.

Palavras chave: programação orientada a aspectos, desempenho, interesses transversais.

Abstract

Aspect-Oriented Programming (AOP) was proposed with the main objective of addressing an important software quality principle that is modularization. The basic idea of the paradigm is to capture crosscutting concerns and handle them independently from the rest of the application. AOP emerged as complement to Object-Oriented Programming (OOP), generating many evaluations and empirical studies about this new paradigm. A variety of metrics were applyed on AOP in order to provide evidence of its benefits or problems, however there is no consensus about the impact on performance of the use of AOP techniques to deal with crosscutting concerns on systems. In order to identify if these techniques related to AOP have impact on system performance, this study proposed a controlled experiment on an academic system of the Federal University of Uberlândia. The experiment had the purpose of assessing factors related to AOP which may influence systems performance. According to the results achieved by this study, factors such as the weaver, number of join points, type of advice and number of lines of code (LOC) affect systems performance when it comes to AOP. This influence, however, showed up to be negligible for common web applications which access databases and have MVC framework. Results also address that, specifically for the load-time weaving process, the stage of loading of classes may affect performance of applications which run on the same environment. These results can be used by architects and developers to reason about system architecture projects concerning the use of AOP.

Keywords: aspect oriented programming, performance, crosscutting concerns.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Nesse capítulo são abordados conceitos introdutórios relevantes para a contextualização do trabalho. Na Seção 1.1 são apresentados conceitos sobre a Programação Orientada a Aspectos (POA) e os princípios que justificaram sua utilização na Engenharia de Software. Na Seção 1.2 é apresentado o impacto causado por POA no desenvolvimento de software. Na Seção 1.3 os objetivos e hipóteses do trabalho são descritos. Na Seção 1.4 é apresentada uma revisão da literatura correlata sobre o assunto. Na Seção 1.5 é apresentada a metodologia de pesquisa utilizada para a realização do trabalho. Na Seção 1.6 é apresentada uma visão geral da dissertação.

1.1 O Problema da Modularização de Software

Um princípio importante da qualidade de software é a modularização. A modularização é um mecanismo que melhoria a flexibilidade e a compreensão de um software, enquanto permite a redução do tempo de desenvolvimento [Parnas 1972]. A idéia da modularização está intimamente ligada à separação de responsabilidades em um sistema de software, onde são definidos pequenos módulos e a eles atribuídas responsabilidades específicas.

Um código não modularizado normalmente apresenta duas características, especialmente quando se trata de Programação Orientada a Objetos (POO): código entrelaçado e código espalhado, ou também conhecidos como *code tangling* e *code scattering* [Laddad 2009].

A presença de código espalhado e entrelaçado afeta negativamente o desenvolvimento e a manutenção dos sistemas de diversas formas, causando perda de rastreabilidade, dificuldades de reúso de código, diminuição da produtividade, aumento do esforço gasto com a manutenção e perda de qualidade [Laddad 2003]. Softwares com alto grau de código espalhado ou entrelaçado são mais difíceis de manter e consequentemente tornam o processo de atendimento aos requisitos mais complexo e caro. Módulos com alto grau de espalhamento e entrelaçamento possuem baixa coesão e alto acoplamento, o que caracteriza

software de baixa qualidade.

A POO trouxe grandes melhorias com relação a modularização, porém não garantiu que o mínimo de acoplamento fosse atingido entre os módulos dos softwares [Hitz e Montazeri 1995]. Apesar do fato que os mecanismos hierárquicos da POO são bastante úteis, eles são incapazes de modularizar todos os interesses em sistemas complexos [Kiczales et al. 1997]. A utilização de técnicas de orientação a objetos para a construção de sistemas que compreendem uma grande variedade de detalhes técnicos, regras de negócios e restrições de desempenho produz código entrelaçado, incorreto e difícil de se manter, segundo Filman [Filman et al. 2005]. Para Filman, POO falhou em modularizar os interesses tranversais.

1.2 O problema da adoção de POA

As lacunas deixadas pelas técnicas de orientação a objetos com relação a esses diversos fatores que influenciam a qualidade dos softwares, principalmente com relação ao tratamento dos interesses transversais, levaram ao surgimento da POA, apresentada por Kiczales em 1997 [Kiczales et al. 1997]. Segundo Kiczales, a idéia central da POA é prover melhor separação de interesses nos softwares, de forma a tratar interesses que são naturalmente tranversais da mesma forma como POO trata interesses que são naturalmente hierárquicos. Kiczales reforça que por meio de POA é possível programar interesses transversais em uma maneira modular e dessa forma alcançar os benefícios da modularização, os quais POO, por si só, falhou em prover, que são: código mais simples, que é mais fácil de desenvolver e de manter, e cujo potencial de reuso é maior. Tais interesses transversais, que por meio de POA são bem modularizados, foram chamados por Kiczales de *Aspectos*.

POA tem sido bastante aplicada por desenvolvedores, porém benefícios significativos trazidos por POA são muito questionados desde seu surgimento. No artigo [Bartsch e Harrison 2008] os efeitos que POA tem sobre a facilidade de entendimento e modificabilidade sobre os softwares foram avaliados e constatou-se que os desenvolvedores apresentavam maior facilidade em lidar com softwares com orientação a objetos pura que se comparado com softwares que utilizam POA. Apontou-se também que deveria existir mais esforço da comunidade em prover ferramentas que possibilitassem melhor visualização da relação entre os aspectos e o código base.

Com relação à modularidade, de acordo com [Przybylek 2011], aspectos violam os princípios básicos dos quais a Engenharia de Software dependeu nos últimos 50 anos. Segundo esse trabalho, POA promove programação desestruturada, quebra o encapsulamento da informação, deixa interfaces implícitas, torna a idéia da modularidade difícil, quebra o contrato entre o módulo base e seus clientes e intensifica o acoplamento. Em seus estudos, nenhum software com implementação baseada em orientação a aspectos evidenciou

a redução do acoplamento se comparado à sua contrapartida contendo apenas orientação a objetos, enquanto que o impacto na coesão permaneceu incerto.

Para [Ceccato et al. 2005], POA dificulta os testes em softwares, uma vez que a adoção de métodos de testes tradicionais em softwares que contém POA tendem a produzir casos de teste complexos.

No trabalho [Madeyski e Szala 2007] foi realizado um estudo empírico em um sistema web onde se compara a abordagem de POA com a abordagem tradicional de orientação a objetos com relação a eficiência do desenvolvimento de software e a qualidade do *design* do software. Para essas duas métricas consideradas, POA não apresentou melhorias significativas.

Para Alexander [Alexander 2003], o processo de *weaving* gerado por técnicas de POA pode afetar significativamente a semântica do interesse primário ao introduzir novo código a esse interesse e, em muitos casos, esse processo prejudica a manutenção do software por dificultar sua facilidade de entendimento. Segundo ele, os desenvolvedores estão carentes de ferramentas que possibilitem explorar os efeitos do processo de *weaving* antes desse processo ocorrer, o que dificulta o entendimento do comportamento das aplicações em tempo de execução.

A refatoração de sistemas legados utilizando orientação a aspectos promete melhorar a compreensibilidade e a manutenção de sistemas ao substituir códigos espalhados por aspectos [Mortensen et al. 2012a], porém os benefícios de se realizar essa refatoração precisam ser balanceados com os custos impostos por essa refatoração. Segundo Mortensen [Mortensen et al. 2012a] tais custos se referem ao esforço humano necessário para realizar a refatoração, manutenção do novo código adicionado, erros introduzidos durante o processo de refatoração e os efeitos negativos causados no desempenho do software.

A crescente demanda por softwares mais complexos e que operam em diversos tipos de plataformas e arquiteturas, com as mais variadas restrições de hardware, como é o caso dos dispositivos móveis, tem exigido que os softwares se adaptem a novas exigências, nas quais o desempenho recebe importante destaque. Desempenho deficiente pode ser um problema sério em muitos projetos [Smith 1990] pois pode acarretar em atrasos, aumento de custos, falhas de implantações e até mesmo abandono de projetos. Apesar disso, tais falhas são raramente documentadas [Woodside et al. 2007a].

O desempenho pode ser afetado por diversos elementos em um processo de Engenharia de Software. Um desses elementos está relacionado com a forma que os interesses transversais são implementados pelas técnicas de POA, uma vez que as ferramentas que suportam POA adicionam códigos extras para tratar os interesses transversais [Alexander 2003].

Considerando o desempenho um importante requisito de qualidade de software [Woodside et al. 2007b] [Rahimian e Habibi 2008] [Evangelin Geetha et al. 2011], o impacto de POA sobre o desempenho pode prejudicar a qualidade se esse impacto acarretar em

perda de desempenho. Esse impacto no desempenho ocorre pela alteração no esforço gasto para tratar a camada extra adicionada no software representada por POA, quando habilitada [Surajbali et al. 2007].

Diversos outros estudos na literatura científica abordam a questão da influência de POA no desempenho dos sistemas, porém na maior parte dos casos, a influência de POA sobre o desempenho não é o objeto de estudo principal desses trabalhos, os quais apresentam poucos resultados e, na maior parte das vezes, imprecisos ou carentes de avaliações experimentais sobre o assunto. Essa ausência de resultados precisos a respeito de POA e desempenho motivou a realização desse trabalho.

1.3 Objetivos e Hipóteses do Trabalho

Considerando o fato que as técnicas de orientação a aspectos adicionam complexidade extra nos softwares para tratar os interesses transversais [Alexander 2003], essa adição de complexidade pode gerar esforço extra na execução dos interesses transversais, o que poderia causar impacto no desempenho.

O objetivo geral desse trabalho é avaliar o impacto que as técnicas de programação de aspectos exercem sobre o desempenho dos softwares. Para avaliar o impacto dessas técnicas no desempenho esse trabalho propõe uma avaliação individual de fatores que compõem essas técnicas ou que estão relacionados a essas técnicas e que, por hipótese, poderiam ser responsáveis por uma sensibilização no desempenho, tais como:

- O tipo de *weaving* realizado sobre o software.
- Os tipos de *advice* atuantes sobre o software.
- O número de *join points* atuantes sobre o software.
- O número de linhas de código do software.
- O número de classes a serem carregadas durante o processo de *load-time weaving*.

A partir de tais fatores, objetiva-se obter respostas para confirmar ou rejeitar as seguintes hipóteses:

- H1: O tipo de *weaving* realizado sobre o software é um fator que causa impacto no desempenho.
- H2: Os tipos de *advice* atuantes sobre o software são fatores que causam impacto no desempenho.
- H3: O número de *join points* atuantes sobre o software é um fator que causa impacto no desempenho.
- H4: Durante o processo de *weaving*, o número de linhas de código do software é um fator que causa impacto no desempenho.

- H5: Durante o processo de *load-time weaving*, o número de classes a serem carregadas é um fator que causa impacto no desempenho.

Como objetivo específico desse trabalho, visa-se mensurar se o impacto causado por esses fatores no desempenho é significativo. O impacto no desempenho também pode depender das condições em que tais fatores são avaliados. Na busca por essas respostas também se considera tais condições e pretende-se avaliar se tais condições também são fatores que podem influenciar nesse impacto.

Os resultados de tais avaliações podem ser utilizados por arquitetos e desenvolvedores para projetar arquiteturas de sistemas envolvendo POA, onde o desempenho é um importante requisito a ser considerado, principalmente em plataformas onde os recursos de hardware são limitados.

1.4 Estado da Arte

Diversos experimentos foram realizados na literatura com o objetivo de verificar se POA causa impacto no desempenho dos softwares[1].

No trabalho [Hilsdale e Hugunin 2004] foi constatado que o processo de combinação dos *join points* pode consumir um tempo considerável e que esse tempo pode mais que dobrar ao se compilar softwares grandes usando algum processo de *weaving*. Esse problema no desempenho pode ser ainda mais significativo quando se usa o processo de *load-time weaving*, no qual o impacto gerado pelo processo de *match* dos *join points* chega a ser perceptível para o usuário. Foi também constatado que uma implementação do interesse transversal *logging* adicionou 22% de impacto no desempenho na aplicação, se comparada à sua versão com a implementação manual do *logging*.

Bijker [Bijker 2005] avaliou o impacto no desempenho de softwares que utilizaram técnicas de POA por meio de um weaver e comparou com técnicas de POA manuais. Seu trabalho concluiu que *advices* simples não causam impacto significativo no desempenho, no entanto, quanto mais sofisticados são, mais lentos se tornam e esse impacto pode chegar a mais de 100 por cento de penalidade.

Kirsten [Kirsten 2005] comparou quatro principais ferramentas de POA na época (2004) e, quando se trata de desempenho, ele concluiu que, em geral, códigos com aspectos possuem desempenho similar às soluções de orientação a objetos puras, onde o código que implementa o interesse transversal está espalhado no software. Apesar disso, alguma perda de desempenho seria notada em tempo de compilação ou execução, dependendo da técnica de POA utilizada. Kirsten também mostrou os mecanismos de linguagem das ferramentas e as contrapartidas impostas por elas, bem como a integração das ferramentas

[1] A ferramenta utilizada na busca dos trabalhos foi o *Google Scholar*.

com o ambiente de desenvolvimento e processo de *build*, incluindo uma comparação entre as ferramentas e fornecendo um resumo dos pontos fortes e fracos de cada uma.

Em [Ali et al. 2010] foram avaliados vários artigos quanto ao desempenho das soluções orientadas a aspectos em comparação às implementações originais dos interesses transversais. Nas avaliações, 4 artigos apresentaram resultados em que houve ganho em desempenho: [Pratap et al. 2004], [Zhang e Jacobsen 2004], [Lohmann et al. 2006] e [Kourai et al. 2007], 2 apresentaram resultados em que a diferença no desempenho foi insignificante: [Coady 2003] e [Siadat et al. 2006], e 1 apresentou resultado misto: [Harbulot e Gurd 2004]. Concluiu-se que ainda restam dúvidas se o paradigma de orientação a aspectos pode ser usado com sucesso em aplicações que precisam de alto desempenho.

Em [Liu et al. 2011] concluiu-se que a abordagem de orientação a aspectos não obteve efeito significativo no desempenho e em alguns casos esse desempenho se mostrou melhor se comparado às implementações originais. Também foi constatado que a variação no número de *join points* não influenciou o desempenho.

1.5 Metodologia de Pesquisa

Com o objetivo de entender melhor a extensão do impacto das técnicas de POA no desempenho dos softwares, foi utilizado um conjunto de instrumentos de pesquisa, descritos a seguir.

Inicialmente foi feito um estudo a respeito do impacto de POA no desempenho. Os resultados encontrados e apresentados na Seção 1.4 se mostraram inconclusivos com relação a esse impacto. Notou-se também que a maioria das publicações relevantes sobre o assunto foram publicadas há mais de 6 anos e algumas das técnicas de POA adotadas nas publicações foram descontinuadas ou incorporadas a outros projetos. Um estudo mais amplo fez-se necessário para compreender a respeito do impacto de POA no desempenho considerando técnicas mais recentes. Foi feita então uma revisão sistemática sobre o assunto, apresentada no Capítulo 3.

O propósito da realização da revisão sistemática é de pesquisar o que a literatura científica produziu a respeito do tema nos últimos 6 anos. A estratégia de pesquisa bem definida da revisão sistemática visa reunir publicações tendo em vista a implementação dos interesses transversais por técnicas atuais de POA e os possíveis impactos no desempenho, considerando alguns dos principais veículos de publicações sobre o assunto.

A base de pesquisa foi formada a partir do estudo inicial e da revisão sistemática tendo em vista as diferentes épocas das publicações e diferentes técnicas e ferramentas utilizadas em cada época. Essa junção de resultados direciona a metodologia desse trabalho.

Foi feita uma comparação entre os resultados obtidos com a revisão sistemática e o estudo inicial, considerando as diferentes épocas e técnicas utilizadas. Foi possível observar que existem poucos estudos na literatura científica a respeito de POA e desempenho

e a maioria desses estudos são muito específicos e algumas vezes inconclusivos. Os poucos artigos existentes sobre o assunto divergem não somente quanto aos resultados, mas também com relação às técnicas de avaliação do desempenho utilizadas, interesses transversais avaliados e ferramentas utilizadas tanto nas avaliações quanto na implementação dos aspectos. Nota-se ainda, por meio dos artigos avaliados, que não existe um padrão de impacto no desempenho dos softwares por meio do uso das técnicas de POA. Essa comparação revela ainda que, apesar do fato que as ferramentas de POA evoluíram desde o surgimento de POA, o impacto no desempenho causado por POA nos softwares ainda permanece carente de resultados conclusivos. Provavelmente isso se deve ao fato que existem muitos interesses transversais que podem ser tratados por vários tipos de ferramentas e técnicas de POA, cada qual para sua linguagem específica.

A fim de investigar os fatores relacionados a POA que podem causar impacto no desempenho, e com isso obter informações para confirmar ou rejeitar as hipóteses criadas na Seção 1.3, foi proposto um experimento de laboratório [Juristo e Moreno 2010] como mais um instrumento de pesquisa. Esse tipo de experimento é adequado quando se tem o controle preciso das variáveis envolvidas no processo da experimentação de forma que se possa manipular uma ou mais variáveis mantendo-se as outras com valores fixos. Os resultados dessa manipulação são então medidos e a partir dos mesmos são geradas análises estatísticas. As variáveis a serem manipuladas são os fatores relacionados a POA que, por hipótese, causam impacto no desempenho, enquanto que os resultados podem ser obtidos por meio da medição de 3 variáveis dependentes representando recursos utilizados, que são: tempo, CPU e memória. A escolha das variáveis dependentes foi baseada nos resultados obtidos com a revisão sistemática de onde foi constatado que as medições de tempo, CPU e memória foram as maneiras mais comuns e atuais de se medir o desempenho nos softwares.

A abordagem proposta para a realização do estudo foi uma abordagem quantitativa. Pesquisas quantitativas objetivam obter relações numéricas entre as variáveis ou alternativas sendo investigadas. Para esse experimento, a análise quantitativa é feita sobre as 3 variáveis dependentes mencionadas com relação a medição dos fatores propostos.

A natureza de um experimento é caracterizada por testar hipóteses de uma maneira objetiva e que possa ser repetida [Tichy et al. 1995]. A repetição desse experimento é possível devido a uma parceria feita com a Universidade Federal de Uberlândia[2] que disponibilizou para a realização do experimento um sistema produzido para propósitos internos e que contém as seguintes características, as quais possibilitam a manipulação dos fatores propostos:

- Acesso livre ao código fonte.

[2]A Universidade Federal de Uberlândia foi parceira nesse estudo. Os resultados desse trabalho estão diretamente relacionados aos interesses do Governo Brasileiro no que tange à qualidade dos Softwares produzidos.

- Presença de interesses transversais que podem ser transformados em aspectos.
- Foi desenvolvido na linguagem de programação Java.
- Contém um cenário propício para a realização do experimento.
- Não contém erros de execução.

O sistema utilizado no experimento é chamado de Sistema Interno de Gestão (SIGE). A partir da versão original, foram originadas versões nas quais os fatores relacionados a POA foram variados e medidos. Os detalhes da geração das versões e da realização do Experimento estão descritos no Capítulo 4.

1.6 Visão Geral da Dissertação

Esse trabalho propõe um experimento a fim de elucidar questões relacionadas ao impacto causado no desempenho pelos fatores de POA, contribuindo assim com a prática em Engenharia de Software, principalmente com relação a construção de projetos que utilizam POA. O trabalho está organizado em 7 capítulos, incluindo este de introdução, conforme explicado a seguir.

No Capítulo 2 é apresentada uma breve revisão de conceitos tais como a Modularização, Acoplamento e Coesão, Interesses Transversais e a Programação Orientada a Aspectos, onde são também descritos conceitos relacionados a Aspectos, Ferramentas utilizadas na Programação Orientada a Aspectos e o Processo de *weaving*.

No Capítulo 3 é descrita uma revisão sistemática em relação a POA e o desempenho. São descritos nesse capítulo a Motivação que levou a realização da revisão sistemática, o Método de Pesquisa utilizado, a Avaliação dos Resultados e a Discussão dos resultados.

No Capítulo 4 são apresentadas as Condições do Experimento, o Mapa das versões representando todas as versões geradas do sistema utilizado no experimento, o Cenário no qual o experimento é realizado, a prova de que a maneira como foi implementado o interesse transversal escolhido para ser avaliado não influencia nas medições, os tipos de invocações escolhidas para experimento, o Plano do Experimento e o Projeto do Experimento, contendo a descrição da geração das versões propostas para o experimento.

No Capítulo 5 é apresentada a Análise dos Resultados, onde é feita uma comparação entre os resultados acompanhada de gráficos que ilustram o comportamento das amostras obtidas nas medições das variáveis escolhidas como medidoras do desempenho. Também são apresentadas a Discussão em torno dos resultados obtidos no experimento e as possíveis Ameaças à Validade do experimento.

No Capítulo 7 são apresentados um resumo do trabalho contendo as conclusões obtidas com o trabalho, as contribuições do trabalho, possíveis ameaças à validade do trabalho e possíveis trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo são abordados conceitos fundamentais para o entendimento desse trabalho. Na Seção 2.1 é apresentada uma visão geral sobre a modularização. Na Seção 2.2 é apresentada a definição de acoplamento e coesão. Na Seção 2.3 são apresentados os chamados interesses transversais bem como exemplos. Na Seção 2.4 são apresentados conceitos importantes sobre POA.

2.1 Modularização

Os primeiros relatos na literatura científica a respeito da modularização datam de 1962 com Herbert Simon [Simon 1962], que identifica modularização como sendo uma configuração específica que melhora a evolução dinâmica de um sistema complexo para novo equilíbrio no contexto evolutivo. Segundo Parnas [Parnas 1972], um módulo é uma atribuição de responsabilidades e deve ser criado de forma a esconder decisões de outros módulos. Monica Calcagno [Calcagno 2002] discute o conceito de modularização após avaliar as definições de vários outros autores. Ela destaca que módulos são projetados e produzidos de forma independente, poderiam ser substituídos ou atualizados de forma independente e trabalham juntos de forma coordenada. Dentre os benefícios esperados resultantes de um desenvolvimento modular, destacam-se:

- Diminuição do tempo de desenvolvimento devido ao fato que grupos de trabalho podem trabalhar separados em diferentes módulos com menos necessidade de comunicação.
- Flexibilização do produto, resultante da possibilidade de alterar um módulo sem a necessidade de alterar outros.
- Compreensibilidade, pois seria possível estudar módulos em separado, permitindo assim que o sistema seja projetado com mais eficiência.

O conjunto dos módulos forma um sistema organizado onde cada módulo se torna disponível para os outros com o uso de uma interface, também chamada de API (Application Programming Interface). O isolamento dos módulos e suas comunicações por meio de interfaces publicadas é fundamental para a organização dos códigos dos softwares [Sarkar et al. 2007]. Códigos bem organizados e bem isolados em módulos permitem que os módulos tenham outros importantes atributos relacionados à qualidade de software que são acoplamento e coesão.

2.2 Acoplamento e Coesão

O acoplamento pode ser definido como uma medida de interdependência entre módulos de um sistema. Para Chidamber e Kemerer [Chidamber e Kemerer 1994] o acoplamento é evidência de um método de um objeto usando métodos ou variáveis de instância de outro objeto. Chidamber e Kemerer definiram uma métrica para o acoplamento, que é o acoplamento entre objetos. Segundo eles, esse acoplamento seria proporcional ao número de pares relacionados com relação não hereditária com outras classes em paradigmas de orientação a objetos.

Coesão é uma medida de relação entre os elementos dentro de um módulo. Para Hitz e Montazeri [Hitz e Montazeri 1995], coesão é um atributo correspondente à qualidade da abstração capturada por uma classe, sendo que boas abstrações exibem alta coesão.

A noção de baixo acoplamento e alta coesão entre módulos surgiu na década de 1970 e tais conceitos ganharam importante destaque desde seu surgimento [Stevens et al. 1974], caracterizando tais atributos como atributos de qualidade em sistemas modularizados. Acoplamento e coesão influenciam diretamente a manutenção, compreensibilidade e modificabilidade de software e são utilizados como referência para desenvolvimento de software de qualidade [Embley e Woodfield 1987]. Apesar desses atributos terem sido inicialmente utilizados para indicar qualidade em sistemas procedurais [Stevens et al. 1974] [Page-Jones 1988], tais conceitos foram também empregados em sistemas com linguagens orientadas a objetos [Meyer 1988] [Coad e Yourdon 1991] [Budd e Design 1997], onde as noções de acoplamento e coesão foram ampliadas [Eder et al. 1994].

2.3 Interesses Transversais

Para Kiczales [Kiczales et al. 1997], ao utilizar técnicas de orientação a objetos (OO) na implementação de sistemas complexos, haveria interesses que poderiam ser modularizados, porém tal implementação ficaria espalhada. Ainda segundo Kiczales, isso acontece porque certos interesses em um sistema naturalmente possuem código espalhado em vários módulos devido ao fato que a modularidade natural de tais interesses atravessa a

modularidade natural do restante das implementações, pois atravessa várias camadas dos sistemas.

Esse tipo de interesse é também chamado de interesse transversal ou *crosscutting concern*. Como um interesse transversal normalmente tem código em vários módulos do sistema, implementações relacionadas a esses interesses também tendem a conter código espalhado. Em um sistema de monitoramento por exemplo, o código que monitora os rastros do usuário poderia estar espalhado em várias camadas do sistema. Tais interesses são normalmente complexos e quando tratados por POO, deixam código espalhado e entrelaçado [Filman et al. 2005]. Para Filman [Filman et al. 2005], POO tem limitações para modularizar os interesses transversais.

São exemplos de interesses transvesais:

- Tratamento de Exceções: uma exceção é um comportamento do sistema indicando que a operação em processo não pode ser completada com sucesso, mas que de outras partes do sistema pode tentar ser recuparada ou ignorada [Lippert e Lopes 2000]. O tratamento de exceções é importante para evitar comportamentos inesperados em um sistema gerados por erros não tratados e pode ser considerado um interesse transversal pelo fato que seu código pode estar espalhado em diversas camadas do sistema.
- Gerenciamento de Transações: Transações tem sido utilizadas desde a década de 1970 para prover processamento de informação confiável em sistemas informatizados [Wang et al. 2008]. Apesar de terem sido inicialmente desenvolvidas para suportar operações de crédito e débito em bancos de dados de sistemas centralizados, normalmente financeiros, as transações passaram a ser utilizadas em diversos tipos de domínio de aplicações em todo o mundo. Atualmente os mecanismos de transações são utilizados em diversos tipos de sistemas com o intuito de garantir as propriedades básicas das transações [Gray et al. 1981] [Haerder e Reuter 1983], que são: consistência, atomicidade, isolamento e durabilidade. O gerenciamento das Transações exige que tais propriedades sejam implementadas e controladas, e as implementações de tais funcionalidades deixam código espalhado.
- *Logging*: o *Logging* é um interesse no qual mensagens são produzidas e podem ou não ser armazenadas. Essas mensagens possuem várias finalidades, tais como auxiliar desenvolvedores a identificar uma sequência lógica de passos em um determinado cenário de um programa, registrar informação a respeito de um passo de execução no programa, emitir um aviso ou alerta, registrar um erro, dentre outras.

Outros exemplos de interesses transversais são: Persistência, Sincronização, Controle de concorrência, *Caching*, Monitoramento, Rastreamento ou *Tracing* e Segurança, mas podem também ser de domínio específico [Mortensen et al. 2012a].

2.4 Programação Orientada a Aspectos

Kiczales [Kiczales et al. 1997] introduziu na década de 1990 o paradigma de Programação Orientada a Aspectos (POA) como um complemento ao paradigma POO. A programação orientada a aspectos (POA) é um paradigma que provê aos programadores a possibilidade de separar interesses transversais de maneira modular e ainda utilizar o paradigma tradicional de POO.

Outras abordagens na literatura com outras nomeclaturas, tais como Programação Orientada a Tópicos [Harrison e Ossher 1993], Programação Adaptaviva [Lieberherr 1996] e Filtros de Composição [Bergmans et al. 1992] estão relacionadas a POA e compartilham dos mesmos objetivos. Porém, a proposta de Kiczales apresentou ser mais simples para abstrair linguagens orientadas a objetos, tais como Java, e mais eficiente para melhor separação e modularização de interesses, especialmente aqueles interesses que são comuns na maioria dos sistemas e que poderiam resultar em códigos espalhados e entrelaçados: os interesses transversais.

Código espalhado é resultado de um interesse sendo implementado em vários módulos [Laddad 2009]. Código entrelaçado é causado por módulos que tratam de vários interesses. Se um módulo de um sistema estiver ao mesmo tempo tratando interesses tais como *Logging* e segurança, irá conter código de ambos e consequentemente esse código estará misturado com o código do módulo em questão.

POA ganhou importante destaque uma vez que introduziu a implementação dos interesses transversais, com vários graus de sucesso [Ali et al. 2010] [Mortensen et al. 2012a] objetivando a separação desses interesses transversais em módulos, de forma que possam ser tratados como componentes e consequentemente evitar o código entrelaçado e espalhado.

Nas subseções a seguir, são apresentadas definições fundamentais com relação a POA.

2.4.1 Aspectos

Nas linguagens de orientação a aspectos os interesses transversais são encapsulados em um aspecto. Um aspecto, que é composto por pontos de interesse específicos na execução de um programa, pode contribuir para o comportamento de métodos ou objetos por meio da invocação implícita de comportamentos adicionais. Segundo a definição de Kiczales [Kiczales et al. 1997], um aspecto é um tipo que encapsula *pointcuts*, *advice* e características estáticas que atravessam vários módulos.

Esses pontos de interesse específicos na execução de um programa são chamados de *join points*. Esses pontos são definidos de acordo com a linguagem de programação de aspectos.

Um *pointcut* é uma construção de linguagem que define um *join point* no código. Kiczales [Kiczales et al. 1997] define um *pointcut* como um elemento que captura *join*

points e expõe os dados do contexto de execução desses *join points*.

Advices são comportamentos adicionais que são executados antes, durante ou depois de um *join point* [Ali et al. 2010] e são inseridos no programa. Por meio de um *Advice* é possível, por exemplo, realizar o *Logging* de mensagens contendo informações a respeito dos métodos executados em determinados pontos do programa. A estrutura de um *Advice* é semelhante à estrutura de um método, onde ocorre o encapsulamento do comportamento a ser executado quando um *join point* é capturado por um *pointcut* [Laddad 2003]. Kiczales [Kiczales et al. 1997] destaca que *Advices* podem ser dos tipos:

- *Before*, quando executa antes de cada *join point*
- *After returning*, quando executa após cada *join point* que retorna normalmente.
- *After throwing*, quando executa após cada *join point* que lança um objeto do tipo *Throwable*, e que pode ser tratado como erro ou exceção.
- *After*, quando executa após cada *join point* independente se retorna normalmente ou se lança um objeto do tipo *Throwable*.
- *Around*, quando executa antes e depois de cada *join point*. A execução do *advice Around* também pode ser entendida como "no lugar de" ao executar o código no lugar do *join point*. O *join point* pode ser executado opcionalmente por meio da chamada *proceed*.

A Figura 2.1 representa um exemplo de aspecto implementado. Para que o AspectJ reconheça a classe como um aspecto, a anotação *@Aspect* é adicionada. Outras anotações como *@Before*, *@After*, *@Around* e *@AfterThrowing* representam os *advices Before*, *After*, *Around* e *After throwing* respectivamente. Esse aspecto representa a implementação do interesse transversal *Logging* e pode ser utilizado por exemplo para fins de auditoria em sistemas. Pode-se definir o significado de cada um dos advices da seguinte maneira:

1. *Before*, linha 18: realizar o log da mensagem "Acessando método em DAO:" e da assinatura de qualquer método executado em qualquer classe dentro do pacote br.ufu.sige.dao.
2. *After*, linha 23: realizar o log da mensagem "Perfis de usuário carregados com sucesso" após a execução do método loadProfilesByUser presente em qualquer classe do pacote br.ufu.sige.dao.
3. *Around*, linha 28: realizar o log da mensagem "Método executado em camada de negócios:" e da assinatura completa de qualquer método executado em qualquer classe dentro do pacote br.ufu.sige.business.
4. *After throwing*, linha 35: realizar o log da mensagem "Erro gerado com mensagem:" e da mensagem de erro encapsulada na exceção capturada.

```java
package br.ufu.sige.aspect;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LoggerAspect {
    private Logger log = Logger.getLogger(this.getClass());

    @Before("execution(* br.ufu.sige.dao.*.*(..))")
    public void adviceBeforeDAOs(JoinPoint joinPoint) {
        log.info("Acessando método em DAO: "+joinPoint.getSignature().getName());
    }

    @After("execution(* br.ufu.sige.dao.*.loadProfilesByUser(..))")
    public void adviceAfterLoadUserProfiles(JoinPoint joinPoint) {
        log.info("Perfis de usuário carregados com sucesso");
    }

    @Around("execution(* br.ufu.sige.business.*.*(..))")
    public Object adviceAroundBusiness(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("Método executado em camada de negócios: "
                +joinPoint.getSignature().toLongString());
        return joinPoint.proceed();
    }

    @AfterThrowing(pointcut="execution(* br.ufu.*.business..*.*(..))",throwing="ex")
    public void exceptionNotifier(Exception ex) {
        log.error("Erro gerado com mensagem: "+ex.getMessage());
    }

}
```

Figura 2.1: Exemplo de implementação de aspecto.

As classes que implementam o *Logging* fornecem vários tipos de mensagens para identificar o nível de gravidade da mensagem. Nesse exemplo foram utilizados apenas os tipos "info" e "error".

2.4.2 Ferramentas para Programação Orientada a Aspectos

Existem várias ferramentas, tecnologias e linguagens de programação utilizadas para a geração e suporte de códigos que tratam os interesses transversais. O AspectJ é uma simples e prática extensão orientada a aspectos para Java [Kiczales et al. 2001]. Além do AspectJ, linguagens como Jasco, extensões de linguagens como AspectC++ e *frameworks* como JBoss AOP, Spring AOP, Aspect# , também conhecidas como *weavers*, também podem ser empregadas no tratamento dos interesses transversais com o intuito de melhorar a modularização dos sistemas [Liu et al. 2011]. Kirsten escreveu um artigo em 2005

[Kirsten 2005] comparando alguns desses *weavers*, onde avalia os pontos fortes e fracos de cada um deles. Segundo Kirsten, os principais *weavers* disponíveis na época eram AspectJ, AspectWerkz, JBoss AOP e Spring AOP.

Dos *weavers* destacados por Kirsten, apenas o AspectJ e Spring AOP continuaram sendo utilizados pelos desenvolvedores quando o assunto é POA, destacando-se entre os mais populares [Psiuk 2009]. AspectWerkz foi integrado ao AspectJ e descontinuado. JBoss AOP também foi descontinuado pela JBoss.

AspectJ é uma linguagem que foi estendida a partir da linguagem Java [Kiczales et al. 1997]. A linguagem provê suporte para implementação modular de um grande número de interesses transversais, sendo que seu código é compilado para o padrão de bytecode da linguagem Java. As regras do processo de *weaving* do AspectJ atravessam vários módulos de uma forma sistemática com o objetivo de modularizar os interesses transversais. O AspectJ define três modelos de *weaving*, conforme explicado na Seção 2.4.3.

O Spring AOP é uma solução de orientação a aspectos fornecida pelo *framework* Spring. Embora não seja tão completo quanto o AspectJ, o Spring AOP fornece uma maneira simples e rápida de programar aspectos para modularizar interesses transversais comumente encontrados em aplicações corporativas[1]. Suas restrições com relação a aspectos se devem à maneira como o framework é implementado. Laddad [Laddad 2009] aponta três principais limitações do Spring AOP:

- Spring AOP funciona apenas para *Join Points* que representam execução de métodos. Apesar disso, para grande parte dos interesses transversais isso não representa um problema. Em interesses comuns como Gerenciamento de Transações e Segurança, a interceptação dos aspectos é feita em nível de método. Mesmo em implementações de aspectos que utilizam apenas a linguagem AspectJ, a qual possui várias definições para *Join Points*, a execução de métodos é o tipo de *Join Point* mais utilizado.
- Spring AOP só funciona com objetos gerenciados pelo Spring (beans). Se um objeto não é um bean do Spring, o mecanismo de interceptação do Spring não atua sobre esse objeto. Isso gera limitação nas aplicações com relação ao uso do *framework* Spring, que precisaria gerenciar todos os objetos nos quais se deseja ter a atuação do mecanismo de interceptação dos aspetos.
- Spring AOP só funciona para chamadas externas de objetos. Quando um objeto faz uma chamada para ele mesmo, o mecanismo de interceptação de aspectos não atua sobre essa chamada.

Existem várias diferenças entre o AspectJ e o Spring AOP. No AspectJ, um *join point* pode ser uma chamada de método, a execução de um método, uma chamada de

[1] http://docs.spring.io/spring/docs/3.0.x/reference/aop.html

construtor, a inicialização de um objeto, dentre outros [Kiczales et al. 1997]. No Spring AOP um *join point* sempre representa a execução de um método. No Spring AOP o único processo de *weaving* disponível é o processo de *runtime weaving*. O AspectJ, ao contrário, disponibiliza os processos de *source weaving*, *binary weaving* e *load-time weaving*. Tais processos de *weaving* são explicados na Seção 2.4.3. Outras diferenças entre o Spring AOP e o AspectJ estão relacionadas à sintaxe de programação de aspectos.

Apesar dessas diferenças entre as duas linguagens, o Spring AOP se integra com o AspectJ fornecendo assim um mecanismo de tratamento de aspectos contendo funcionalidades de ambos *frameworks*. A documentação do Spring AOP[2] mostra como essa integração pode ser feita em um ambiente onde o framework é o Spring, apontando as diversas vantagens obtidas nessa integração, principalmente com relação à flexibilidade obtida quanto ao estilo de programação de aspectos.

2.4.3 Processo de *Weaving*

O processo de *weaving* converte um código orientado a aspectos para um código orientado a objetos onde os aspectos são integrados no código, se adaptando à linguagem orientada a objetos. Na Figura 2.2 o processo *weaving* é ilustrado.

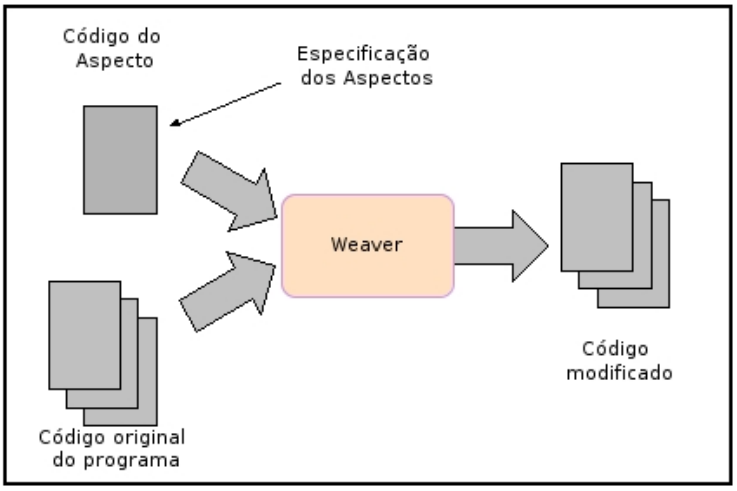

Figura 2.2: Processo de *Weaving*.

O *weaver* é a ferramenta que realiza o proesso de *weaving*. Há diferentes abordagens para se realizar o processo de *weaving* [Hundt e Glesner 2009]:

1. *Compile-time weaving* ou *Build-time weaving*: em tempo de compilação ou *build* faz a junção do aspecto com o código base.
2. *Runtime weaving* ou *Dynamic Weaving*: essa junção pode ser feita no tempo de carregamento das classes, antes das classes bases serem executadas pela JVM ou até mesmo durante a execução do programa.

[2]http://docs.spring.io/spring/docs/3.0.x/reference/aop.html

Esse processo de *weaving* quando realizado em tempo de carregamento das classes é também conhecido na literatura como *load-time weaving*.

Os *weavers* podem ter seu comportamento readaptado para que seu processo de *weaving* atenda à alguma necessidade específica. O trabalho de Keuler e Kornev [Keuler e Kornev 2008] apresentou um processo de *weaving* personalizado para operar em uma plataforma específica sem precisar alterá-la. Hundt e Glesner [Hundt e Glesner 2009] propuzeram otimizações na execução do mecanismo de aspectos com o intuito de melhorar o desempenho em dispositivos móveis.

O processo de *weaving* pode também impactar no desempenho, principalmente quando esse processo é realizado em tempo de execução ou *runtime*. Para Hundt e Glesner o processo de *runtime weaving* demanda mais esforço em tempo de execução, ao passo que o processo de *weaving* estático (*compile-time weaving*) aumenta o tamanho do código fonte, o que pode ser crítico para dispositivos móveis. Ainda segundo Hundt e Glesner, quanto mais tarde ocorrer processo de *weaving*, mais flexibilidade existe para que o aspecto seja adicionado ao código base, permitindo adaptações específicas de contexto.

No AspectJ, a implementação das regras do processo de *weaving* é feita de duas formas: estática e dinâmica [Laddad 2003]. No processo de *weaving* estático, são adicionadas modificações dentro de estruturas estáticas, tais como classes, interfaces e aspectos do sistema, sem alterar, por si só, o comportamento do sistema. No processo de *weaving* dinâmico ocorre adição de comportamento na execução do programa, podendo esse comportamento alterar ou até substituir um comportamento já existente na execução do programa. O *weaver* precisa realizar o processo de *weaving* para juntar classes e aspectos para que o *advice* seja executado. O AspectJ define três modelos de *weaving* [Laddad 2009]:

- *Source weaving*: o *weaver* é parte do compilador e as entradas para o *weaver* são classes e aspectos na forma de códigos-fonte. O processo de *weaving* atua sobre os fontes e produz os *bytecodes*.
- *Binary weaving*: o *weaver* recebe como entrada classes e aspectos na forma de *bytecode* e realiza o processo de *weaving* sobre os bytecodes.
- *Load-time weaving*: o *weaver* recebe como entrada classes e aspectos na forma binária e configurações, sendo que o processo de *weaving* ocorre no momento de carregamento das classes para a máquina virtual (VM) Java.

No Spring AOP, o único processo de *weaving* disponível é o processo de *runtime weaving*.

Capítulo 3

Revisão Sistemática

Para um bom entendimento a respeito do impacto das técnicas de POA no desempenho dos sistemas, foi realizada uma revisão sistemática sobre o assunto. Segundo Kitchenham [Kitchenham et al. 2009], a revisão sistemática em Engenharia de Software provê maneiras de se identificar, avaliar e interpretar todo conteúdo relevante sobre determinadas perguntas de pesquisa, assunto ou fenômeno de interesse. Ainda segundo Kitchenham, várias são as razões para se realizar uma revisão sistemática:

1. Resumir a evidência existente com relação a uma tecnologia.
2. Identificar lacunas na pesquisa atual com o intuito de sugerir áreas para novas investigações.
3. Prover um *framework*/conhecimento para se posicionar de forma apropriada a respeito de novas atividades de pesquisa.

Na Seção 3.1 é apresentada a motivação que levou a realização da revisão sistemática. Na Seção 3.2 é apresentado o método de pesquisa utilizado para realizar a revisão sistemática. Na Seção 3.3 são apresentados os resultados da revisão sistemática. Na Seção 3.4 são discutidos os resultados da revisão sistemática. Na Seção 3.5 é apresentada a conclusão a respeito dos resultados encontrados na revisão sistemática.

3.1 Motivação

A realização da revisão sistemática foi motivada pelo fato que existem poucos estudos com evidências empíricas a respeito do impacto de POA no desempenho dos sistemas. Os resultados dos poucos estudos existentes também não apresentam um padrão, tanto para os valores desse impacto, quanto para as técnicas utilizadas na medição do desempenho e no tratamento dos interesses transversais. A literatura científica apresenta trabalhos onde POA não influencia o desempenho de forma significativa [Coady 2003] [Siadat et al. 2006] [Liu et al. 2011], onde POA influencia positivamente [Zhang e Jacobsen 2004] [Pratap

et al. 2004] [Lohmann et al. 2006] [Kourai et al. 2007] e onde POA causa perda de desempenho para algumas situações [Bijker 2005] [Mortensen et al. 2012a]. Além de reunir informações dos últimos trabalhos a respeito dos impactos das técnicas de POA no desempenho dos sistemas, a revisão sistemática também é útil para reunir informações a respeito de:

1. Quais interesses transversais tem sido transformados em aspectos.
2. Quais técnicas tem sido utilizadas para medir o desempenho.
3. Quais técnicas tem sido utilizadas para tratar os interesses transversais.
4. Quais tipos de aplicações tem sido tratadas com POA.
5. Quantos estudos tem sido realizados a respeito do tema.
6. Quais os tamanhos dos estudos nos quais as técnicas de POA tem sido aplicadas.
7. Quais são as linguagens originais dos estudos nas quais as técnicas de POA tem sido aplicadas.
8. Quais linguagens orientadas a aspectos tem sido utilizadas para tratar os aspectos nos estudos.
9. Em quais domínios de aplicações POA tem sido empregada.

3.2 Método de Pesquisa

A revisão sistemática foi motivada pela seguinte questão:

Q1 - O uso de técnicas de orientação a aspectos para tratar interesses transversais causa impacto no desempenho dos sistemas ?

Uma questão derivada é:

Q2 - O impacto gerado, caso exista, é significativo ?

As respostas para essas questões podem ajudar desenvolvedores a concluir sobre a viabilidade do uso das técnicas de POA para tratar interesses transversais com relação ao desempenho dos sistemas, principalmente em arquiteturas onde o próprio desempenho é um dos interesses, tal como ocorre em plataformas embutidas.

Para responder essas questões, a revisão sistemática [Kitchenham et al. 2009] foi realizada. Foi feita uma triagem em artigos publicados em periódicos científicos e conferências e foram considerados nas buscas artigos publicados nos últimos 6 anos. A *String* de busca utilizada para tal pesquisa foi: (*"Aspect-oriented programming" AND "performance"*) em alguns dos principais veículos de publicações sobre o assunto.

Ao todo foram encontradas 338 publicações. As conferências escolhidas foram AOSD (International Conference on Aspect-Oriented Software Development) e ICSE (International Conference on Software Engineering). Artigos publicados em *workshops* específicos

mantidos juntos com essas conferências não foram considerados. As revistas escolhidas foram: JSS (Journal of Systems and Software), IST (Information and Software Technology), SCP (Science of Computer Programming), TSE IEEE (IEEE Transaction on Software Engineering), TOSEM (ACM TransactionS on Software Engineering Methodology). ENTCS (Electronic Notes in Theoretical Computer Science), que pode ser considerado uma conferência, também foi incluído.

Nos 338 artigos encontrados foi feita uma pré-seleção com o intuito de separar aqueles que relacionam POA com alguma métrica de desempenho, os quais foram classificados então em relevantes. Para realizar essa separação, foram lidos nos artigos os títulos, resumo e palavras-chaves. Se o assunto do artigo era pertinente com POA e desempenho, eram lidas então a introdução e conclusão. Se ainda restavam dúvidas quanto a identificar se o artigo era relevante com o assunto, eram buscadas palavras chaves como: *aspect*, *crosscutting* e *performance*. Houve muitos casos em que não foi possível identificar a relevância do artigo por meio da leitura do resumo, introdução e conclusão. No entanto, após a busca por essas palavras chave notou-se que se tratava da avaliação do desempenho de alguma técnica utilizada em algum interesse do sistema avaliada com outros termos tais como *cost*, *payload* ou *overhead*.

A partir dos resultados relevantes (32 no total), foram selecionados para completa leitura apenas aqueles artigos que avaliam o desempenho da implementação de algum interesse transversal. Como resultado, 15 artigos foram selecionados no total. Notou-se que vários tipos de interesses foram classificados como interesses transversais, mesmo alguns sendo específicos do domínio do problema. Artigos que tiveram interesses transversais implementados mas cujo desempenho não foi considerado ou avaliado, foram descartados. Da mesma forma, artigos que tiverem seu foco no uso de alguma nova linguagem para orientação a aspectos, em uma nova ferramenta para compilar aspectos ou relatam uma pesquisa sobre o assunto sem uma contribuição original também foram desconsiderados. O resumo do processo de filtragem dos artigos pode ser visto na Tabela 3.1 e a ilustração desse processo na Figura 3.1. A seleção final dos artigos é apresentada na Tabela 3.2 .

Tabela 3.1: Resultados da revisão sistemática.

Publicação	Total de artigos	Artigos relevantes	Artigos Selecionados	Fonte
JSS	38	2	1	ScienceDirect
IST	32	10	5	ScienceDirect
SCP	32	2	1	ScienceDirect
TSE	1	1	1	IEEExplorer
TOSEM	21	3	1	ACM Digital Library
ENTCS	26	3	1	ScienceDirect
AOSD	127	9	3	ACM Digital Library
ICSE	61	2	2	ACM Digital Library
Total	338	32	15	

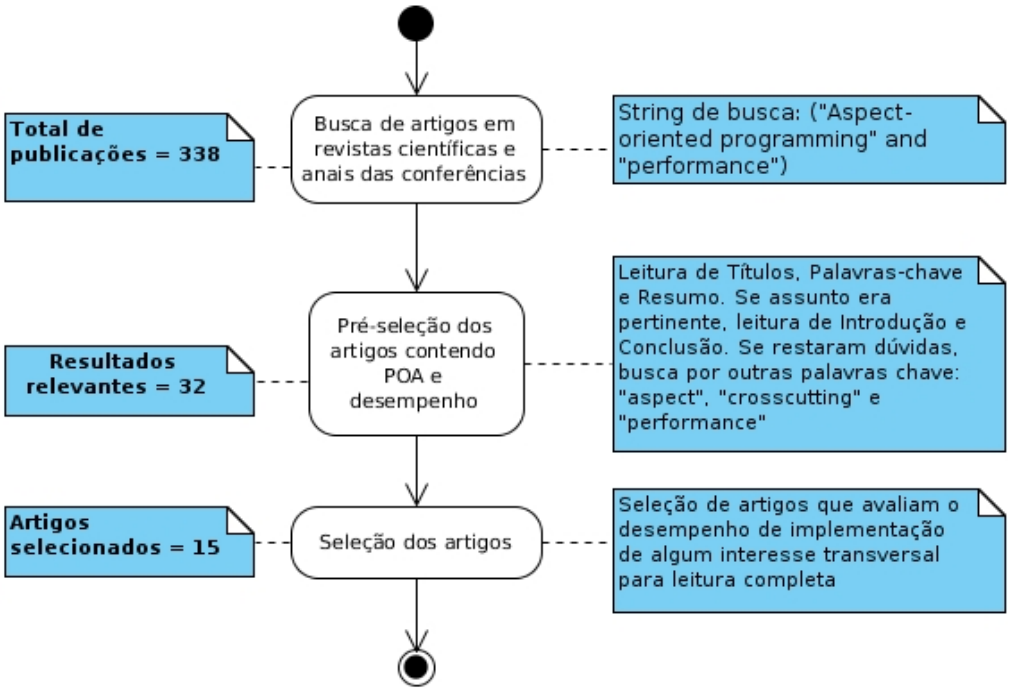

Figura 3.1: Processo de seleção dos artigos.

Tabela 3.2: Artigos selecionados.

	Local/Ano	Referência
1	ICSE/07	[Froihofer et al. 2007]
2	SCP/08	[Fabry et al. 2008]
3	IST/09	[Georg et al. 2009]
4	ENTCS/09	[Hundt e Glesner 2009]
5	IST/09	[Ganesan et al. 2009]
6	ICSE/09	[Zhang 2009]
7	AOSD/09	[Cannon e Wohlstadter 2009]
8	JSS/10	[Malek et al. 2010]
9	ACM/10	[Dyer e Rajan 2010]
10	IST/10	[Ortiz e Prado 2010]
11	AOSD/10	[Toledo et al. 2010]
12	IST/10	[Janik e Zielinski 2010]
13	AOSD/10	[Ansaloni et al. 2010]
14	IEEE/12	[Mortensen et al. 2012b]
15	IST/12	[de Roo et al. 2012]

3.3 Avaliação dos Resultados

Todos os 15 artigos selecionados foram completamente lidos para avaliação. Esses artigos foram avaliados com base em 2 conjuntos de critérios: Tipo de Aplicação e Desempenho. Essa seção classifica os artigos para ambos tipos de critérios.

3.3.1 Tipo de Aplicação

O primeiro conjunto de critérios é referente ao tipo de aplicação e compreende as seguintes métricas: número de estudos avaliados, linhas de código (tamanho, em linhas de código), linguagem de programação original (LPO), linguagem de programação de aspectos (LPA) e domínio da aplicação.

Após avaliação dos estudos de caso ou experimentos avaliados pelos artigos, foram identificados os seguintes tipos de aplicações: *Middleware*, *Web Service*, Embutido, Plataforma, Sistema ou Aplicação, Linguagem ou Extensão (Linguagem) e *Framework*. Casos onde os estudos de caso foram descritos como Sistemas de Monitoramento foram classificados como Sistema ou Aplicação. Artigos que não mencionaram o tipo de aplicação de seus experimentos foram classificados na definição mais próxima dentre as já mencionadas.

O número de estudos avaliados indica apenas os estudos que foram implementados por alguma técnica de POA e que foram avaliados por algum tipo de métrica.

Com relação ao número de linhas de código (LOC), os artigos apresentaram LOC de diferentes maneiras: SLOC, que indica número de linhas de código fonte; NCLOC, que indica número de linhas de código não comentadas e apenas LOC onde nenhuma citação sobre fontes ou comentários foi feita. Alguns artigos apresentaram tamanho das aplicações ao invés de LOC. Houve artigos, no entanto, que não apresentaram qualquer medida de tamanho ou LOC em seus estudos.

Com relação às linguagens de programação originais, os artigos apresentaram trabalhos em várias linguagens. Quanto às linguagens de programação de aspectos, alguns artigos apresentaram suas próprias linguagens ou extensões de linguagens ao invés de usar as linguagens de programação de aspectos mais comuns. Para ambos tipos de linguagem, houve casos em que as linguagens de programação não puderam ser identificadas.

Para o domínio de aplicação, os seguintes resultados foram encontrados: *E-Commerce* para aplicações comerciais com portais Web (E-C); Aplicação da indústria (Indústria), referindo-se a sistemas comerciais não Web; Escritório e Sistema bancário (Banco). Casos onde não há um domínio de aplicação como por exemplo uma extensão de linguagem ou um *kit* de ferramentas foram classificados como Genérico. Alguns artigos, assim como no tipo de aplicação, não mencionaram o domínio da aplicação e também foram classificados por proximidade.

O resumo dos estudos é apresentado na Tabela 3.3. Casos onde nenhuma métrica foi apresentada ou onde não é possível identificá-la foram classificados como não disponível (ND).

3.3.2 Critério Desempenho

O segundo conjunto de critérios é relativo ao desempenho. Quatro métricas foram extraídas dos artigos: tipo de *weaving*, interesses transversais implementados, método de

Tabela 3.3: Resumo dos estudos.

Tipo de Aplicação	Artigo	Nro est. aval.	LOC / Tamanho	LPO	LPA	Domínio da Aplicação
Middleware	[Malek et al. 2010]	2	total de 12.3K sloc	Java	AspectJ	Genérico
	[Zhang 2009]	3	12.7 kloc 113 KB, 190 KB		FlexSync	Indústria
Embutido	[Hundt e Glesner 2009]	1	ND		ObjectTeams, Java	
	[de Roo et al. 2012]	2	ND	GPL	ND	
Sistema ou Aplicação	[Janik e Zielinski 2010]	1	ND	Java	JBoss AOP	Genérico
	[Ganesan et al. 2009]				AspectJ	Escritório
	[Cannon e Wohlstadter 2009]	1	46k ncloc		Java, AspectJ	Genérico
	[Fabry et al. 2008]	1	ND		KALA	Banco
	[Froihofer et al. 2007]				AspectJ, JBoss AOP	Indústria
	[Mortensen et al. 2012a]	3	1.6 kloc, 13.9 kloc, 51.6 kloc	C++	AspectC++	
Linguagem	[Toledo et al. 2010]	1	118 kb	JavaScript	AspectScript	Genérico
	[Dyer e Rajan 2010]	2	ND	Java	ND	Indústria
Framework	[Ansaloni et al. 2010]	1			Compatible with AspectJ	Genérico
Plataforma	[Georg et al. 2009]			ND	ND	E-C
Web Service	[Ortiz e Prado 2010]			Java	AspectJ	

desempenho utilizado e impacto no desempenho.

Com relação ao tipo de *weaving*, dois tipos principais foram considerados nos estudos: *compile-time* e *runtime weaving*. Alguns artigos apresentaram estudos realizando o processo de *load-time weaving* e esses casos foram classificados como *runtime*, já que o processo de *load-time weaving* é um estágio específico do processo de *runtime*.

Vários tipos de interesses transversais foram recuperados dos artigos. Houve casos em que os interesses transversais eram de domínio específico. Artigos que trataram apenas um interesse prevaleceram. Houve casos, porém, onde vários interesses de domínio específico foram considerados [Mortensen et al. 2012a]. Os interesses transversais não foram traduzidos para o português para não haver perda da semântica, como ocorreria por exemplo com os interesses *Safety* e *Security*.

Os métodos de desempenho recuperados nos artigos foram: medições de tempo de execução (mte), operações de negócios por segundo (ons), consumo de memória (mcm), uso de CPU (cpu), observação da execução de uma maneira geral (obs), tempo de *Parsing*

(mtp) e média do número de invocações de métodos por segundo (mmt). Alguns artigos apresentaram mais de um método de desempenho. Nesses casos, quando houve redução do desempenho, as medidas consideradas foram baseadas no pior caso. O impacto no desempenho foi medido em fatores (quando comparável com as implementações originais) ou em porcentagem e foram todos apresentados em forma de fatores a fim de facilitar a visualização. Alguns casos relataram o impacto como insignificante (insig) ou como variante, dependendo da abordagem (variante).

Os resultados das avaliações de desempenho realizados pelos artigos são apresentados na Tabela 3.4. Na coluna Impacto no Desempenho, os símbolos "+" e "-" significam diminuição e aumento no desempenho, respectivamente. O significado desses símbolos está associado a idéia de recursos gastos com o uso de POA, especialmente o tempo. Se um símbolo foi seguido de insignificante, significa que o artigo relatou uma perda ou ganho do desempenho, porém esse resultado foi considerado insignificante pelos respectivos autores.

Tabela 3.4: Análise de desempenho dos estudos.

Tipo de Weaving	**Artigo**	**Interesses transversais implementados**	**Método de desempenho**	**Impacto no desempenho**
Runtime	[Malek et al. 2010]	Stylistic	mte, mcm	mte: -insig, mcm: +1.03x até 1.1x
	[Zhang 2009]	Synchronization	ons	insig
	[Hundt e Glesner 2009]	Maintainability, Extensibility and Reusability	mte	- 2x para 100 instancias
	[Janik e Zielinski 2010]	Reconfigurability	mte, cpu, mcm	mte: +1.1x até 1.22x, cpu: +ND, mcm: +ND
	[Ganesan et al. 2009]	Monitoring	obs	não notado
	[Cannon e Wohlstadter 2009]	Security	mtp	+ até 1.16x
	[Toledo et al. 2010]	Expressiveness	mmt, cpu	mmt: + até 16.1x, cpu: insig
	[Fabry et al. 2008]	Transaction Management	ND	+ ND
Compile-time	[Mortensen et al. 2012a]	Caching, CheckFwArgs, Excepter, Singleton, Tracing, CadTrace, FwErrs, FetTypeChkr, Timer, UnitCvrt, ViewCache, ErcTracing, QueryConfig, QueryPolicy	mte, mcm	mte: + até 1.18x, mcm: + até 1.15x
	[Ansaloni et al. 2010]	Comunication between threads	mte	+ fator de até 31.08x
	[Ortiz e Prado 2010]	Device adaptation	mte	insig
Compile-time / runtime	[Froihofer et al. 2007]	Constraint validation	mte	+ variante
Domain Specific	[Dyer e Rajan 2010]	Cache	mmt	+ 1.015x
ND	[de Roo et al. 2012]	Safety	ND	+ ND
	[Georg et al. 2009]	Security	mte	+ variante

3.4 Discussão

Considerando o fato que todos os artigos selecionados na revisão sistemática (15 no notal) foram completamente lidos, que se tratam de implementação de interesses transver-

sais por meio de técnicas de POA e seu impacto no desempenho, importantes resultados podem ser extraídos dessa pesquisa.

3.4.1 Tipos de Aplicações

A respeito do primeiro conjunto de critérios, relacionados ao tipo de aplicação, é possível concluir que a maioria dos artigos (10 de 15) avaliaram apenas um estudo ou experimento. Entretanto, apenas quatro deles apresentaram LOC ou tamanho de seus estudos avaliados. Os dois estudos que avaliaram mais sistemas, avaliaram três sistemas de pequena escala (no máximo 51.6 KLOC ou 190 KB). Uma hipótese para tal ausência de avaliações de sistemas de larga escala que utilizam técnicas de POA é que POA não é adotada de forma extensiva tal como POO ou programação procedimental. Essa baixa adoção pela comunidade restringe a disponibilidade de sistemas de larga escala para experimentação.

O tipo de aplicação predominante foi Sistema ou Aplicação. Isso é razoável de se esperar porque em geral esses tipos de aplicações são mais frequentes e mais acessíveis. O domínio de aplicação predominante foi Indústria, seguido por aplicações sem domínio específico, aqui classificados como Genérico. Pode-se observar que existe moderada variabilidade em termos de Tipo de Aplicação e Domínio da Aplicação.

Na Tabela 3.5 foram combinadas as características dos tipos de aplicações com os resultados do desempenho com o propósito de avaliar se existe alguma influência do tipo de aplicação no desempenho. Pode-se observar que nos sistemas com categoria Middleware o impacto no desempenho foi pequeno ou insignificante, enquanto que para os sistemas classificados como Sistemas ou Aplicações existe uma tendência de maior impacto.

O fator LOC / Tamanho deixa dúvidas quanto à sua influência no desempenho. Houve casos como [Zhang 2009] cujos sistemas apresentaram tamanhos em KB e KLOC e cujas medições tiveram impacto insignificante no desempenho, e casos como [Cannon e Wohlstadter 2009], [Toledo et al. 2010] e [Mortensen et al. 2012a] onde os tamanhos foram apresentados em LOC, KB e NCLOC e onde houve variação no desempenho para alguns fatores. Da mesma forma, o domíno da aplicação não apresentou uma influência clara no desempenho. O domínio classificado como Indústria, que teve o maior número de estudos, apresentou resultados negativos, positivos e insignificantes quanto ao desempenho.

Com relação aos interesses transversais implementados nas aplicações, não houve interesse predominante nos estudos e, surpreendemente, nenhum dos estudos implementou interesses comuns como *Logging* ou *Tratamento de Exceções*. Isso pode ser um indicativo de que tais estudos não foram representativos em termos de sistemas orientados a aspectos típicos. Em alguns dos artigos analisados os interesses transversais não foram identificados de forma trivial. Houve casos em que identificou-se interesses por meio da busca por objetivos. Dessa forma, quando o interesse não era exposto de maneira explícita, palavras

Tabela 3.5: Tipo de aplicação e desempenho.

Tipo de Aplicação	**Artigo**	**LOC/ Tamanho**	**LPO**	**LPA**	**Domínio da Aplicação**	**Impacto no Desempenho**
Middleware	[Malek et al. 2010]	total de 12.3K sloc	Java	AspectJ	Genérico	mte: -insig, mcm: +1.03x até 1.1x
	[Zhang 2009]	12.7kloc, 113kb, 190kb		FlexSync	Indústria	insig
Embutido	[Hundt e Glesner 2009]	ND		ObjectTeams, Java		- 2x para 100 instancias
	[de Roo et al. 2012]	ND	GPL	ND		+ ND
Sistema ou Aplicação	[Janik e Zielinski 2010]	ND	Java	JBoss AOP	Genérico	mte: +1.1 até 1.22x, cpu: +ND, mcm: +ND
	[Ganesan et al. 2009]			AspectJ	Office	não notado
	[Cannon e Wohlstadter 2009]	46k ncloc		Java, AspectJ	Genérico	+ até 1.16x
	[Fabry et al. 2008]	ND		KALA	Bank	+ ND
	[Froihofer et al. 2007]			AspectJ, JBoss AOP	Indústria	+ variante
	[Mortensen et al. 2012a]	1.6kloc, 13.9kloc, 51.6kloc	C++	AspectC++		mte: + até 1.18x, mcm: + até 1.15x
Linguagem	[Toledo et al. 2010]	118kb	JavaScript	AspectScript	Genérico	mmt: + até 16.1x, cpu: insig
	[Dyer e Rajan 2010]	ND	Java	ND	Indústria	+ 1.015x
Framework	[Ansaloni et al. 2010]			Compatible with AspectJ	Genérico	+ fator de até 31.08x
Plataforma	[Georg et al. 2009]		ND	ND	E-C	+ variante
Web Service	[Ortiz e Prado 2010]		Java	AspectJ		insig

chave como *goal* foram utilizadas para identificar tais interesses.

3.4.2 Linguagens de Programação Utilizadas

A linguagem de programação original predominante nos estudos foi Java, com 11 dos 15 estudos avaliados, e a linguagem de programação de aspectos predominante foi AspectJ, presente em 6 dos 15 estudos. Esses resultados indicam que apesar do Java ser a linguagem preferida com relação a aspectos e AspectJ ser a solução preferida para implementá-los, outras soluções menos comuns surgiram na pesquisa, tais como FlexSync, ObjectTeams e KALA. Isso mostra que desenvolvedores consideraram soluções próprias ao invés de soluções bem conhecidas e que contém um número significativo de adeptos como o AspectJ ou JBoss [Froihofer et al. 2007]. JBoss AOP esteve presente em apenas dois estudos, e Spring AOP não foi utilizado em nenhum.

Considerando o impacto da linguagem de programação no desempenho, pode-se observar que a linguagem de programação original que esteve presente em um número significativo de estudos, o Java, não apresentou evidência clara de influência no desempenho. Da mesma forma, o AspectJ, que prevaleceu como linguagem de aspectos não mostrou influência direta no desempenho porque os estudos que a utilizaram apresentaram resultados divergentes. Apesar de apenas dois estudos terem utilizado a linguagem JBoss AOP, ambos apresentaram impacto positivo no desempenho.

3.4.3 Tipo de Weaving

Considerando o segundo conjunto de critérios, é possível concluir que o processo de *runtime* é o processo de *weaving* mais comum utilizado nos estudos dos artigos. Uma das razões para essa escolha, ao invés do processo de *compile-time weaving*, é o fato que o processo de *runtime weaving* permite que os aspectos sejam adicionados ao programa base de forma dinâmica, o que é melhor para adaptações específicas de contexto nas aplicações [Hundt e Glesner 2009]. Alguns artigos não apenas utilizaram o processo de *runtime weaving* mas também o extenderam e o adequaram aos seus estudos. Também com relação ao processo de *weaving*, os artigos em geral postularam que o processo de *runtime weaving* exige mais esforço em tempo de execução, o que impacta no desempenho, mas nenhuma prova desse impacto foi apresentada de forma clara.

3.5 Conclusão

Artigos avaliaram seus experimentos de diferentes maneiras, mas a métrica de desempenho predominante foi a medição do tempo de execução (mte), talvez pelo fato de existirem diversos mecanismos que fornecem a medição do tempo de maneira simples. Medições baseadas em CPU e memória também tiveram participações significativas nos estudos, porém em menor quantidade, talvez devido ao fato da dificuldade em se medir esses tipos de variáveis.

O impacto no desempenho variou de acordo com uma série de variáveis tais como a abordagem utilizada, interesse transversal implementado, linguagem de aspectos utilizada, tipo de *weaving* e a ferramenta de *weaver* utilizada. Isso leva a conclusão de que apontar evidências da relação entre POA e desempenho não é uma tarefa trivial, e mais pesquisa é necessária. Existem muitas variáveis que podem impactar no desempenho dos sistemas quando se trata de POA mas esse parece ser um assunto ainda pouco explorado na literatura científica. Resultados mais apurados nesse sentido poderiam ser obtidos com o uso de experimento controlado, onde cada variável seria estudada isoladamente. A dificuldade desse tipo de experimento estaria relacionada a sua natureza multifatorial e para que os fatores pudessem ser controlados, um projeto desses fatores seria necessário. A vantagem de experimentos com multi-fatores é que cada iteração do experimento pode ser analisada isoladamente, porém o número de execuções cresceria de forma exponencial com a adição de novos fatores.

Um fator crítico e importante na questão da avaliação de desempenho é a carga de trabalho utilizada na medição da execução [Jain 1991]. Nesse caso, a carga de trabalho é fortemente influenciada pelo tipo de aplicação, que define vários outros sub-fatores: os tipos de *join points*, *pointcuts* e *advices*, a taxa de ocorrência de construções de POA e outras construções não-POA e requisitos da aplicação para tipos de *weaving* específicos. Considerando os espaços para a combinação dos níveis desses fatores, seria desafiador, se não impossível, encontrar uma aplicação do mundo real, ou um conjunto delas, onde todos os níveis seriam explorados. Dessa forma, uma alternativa poderia ser projetar uma aplicação sintética que poderia ser utilizada como referência para avaliação de técnicas de POA. Essa aplicação precisaria ser significativa o bastante ao ponto de poder simular cenários do mundo real e precisaria ser compreensiva o bastante para garantir que todos os fatores importantes de POA e seus respectivos níveis pudessem ser considerados.

3.5.1 Ameaças à validade

O relativo baixo número de artigos selecionados após a aplicação da *String* de busca e os critérios para a inclusão dos artigos para avaliação é uma ameaça à validade dos resultados. Uma solução possível poderia ser a inclusão de conferências em uma futura pesquisa. Entretanto, de forma a avaliar se essa abordagem de busca foi adequada, foi feita uma busca no *Google Scholar* usando a mesma *String*: ("*Aspect-oriented programming*" and "*performance*"). A busca retornou 10.400 resultados, mas os *links* com melhor *ranking* não apresentaram resultados relevantes para essa pesquisa. Após isso, a *String* foi restringida para apenas os títulos dos artigos e apenas 12 resultados foram retornados. Todos os resultados foram analisados e apenas [Liu et al. 2011] foi um resultado novo e relevante que não havia sido selecionado nos artigos. Portanto, concluiu-se que a abordagem de busca utilizada foi adequada.

Capítulo 4

O Experimento

Nesse capítulo é apresentado um experimento de laboratório [Juristo e Moreno 2010] com o propósito de identificar quais fatores relacionados a POA causam impacto no desempenho dos softwares. Esse capítulo foi dividido em seções, conforme a seguir. Na Seção 4.1 são apresentadas as condições nas quais o experimento foi realizado com detalhes a respeito do sistema utilizado no experimento, ferramentas e suas configurações, *frameworks* empregados, bibliotecas de suporte utilizadas e configurações do ambiente onde ocorreu o experimento. Na Seção 4.2 é apresentado um mapa das versões do sistema propostas para o experimento, contendo as especificações dessas versões e o agrupamento dessas versões em regiões para avaliação dos fatores relacionados a POA que são objetos de estudo. Na Seção 4.3 é apresentado o cenário da aplicação utilizado no experimento. Na Seção 4.4 é apresentado o método que gera as mensagens de *log*, bem como a garantia de que a implementação do mesmo nas versões é correta. Na Seção 4.5 são apresentados dois tipos de invocações consideradas no experimento, bem como as justificativas de suas utilizações para as diversas versões geradas. Na Seção 4.6 é apresentado um plano de execução do experimento contendo a definição das variáveis do experimento bem como seus valores e como são feitas as medições das variáveis dependentes propostas. Na Seção 4.7 são apresentados os detalhes técnicos da geração de cada uma das versões utilizadas no experimento.

4.1 Condições do Experimento

O Sistema original, sobre o qual o experimento foi realizado, é um sistema web chamado Sistema Interno de Gestão (SIGE). Projetado em 2011, o SIGE tem o propósito de automatizar o gerenciamento de informações acadêmicas e administrativas das unidades da Universidade Federal de Uberlândia e está funcionando em ambiente de produção desde sua primeira implantação.

O SIGE foi desenvolvido na linguagem Java. A ferramenta de desenvolvimento uti-

lizada foi o *Eclipse*[1] (versão Juno) com o *plugin* do *Maven*[2]. *Maven* é uma ferramenta de gerenciamento e compreensão de projetos responsável por gerenciar o ciclo de vida de projetos. O SIGE é composto por 2 *frameworks*, que são o *Struts*[3] (versão 2.3.3) e o *Spring*[4] (versão 3.0.0). O *Struts* é o *framework MVC* e o *Spring* é responsável por várias funcionalidades, dentre as quais se encontram injeção de dependências e as funcionalidades disponibilizadas pelos mecanismos de orientação a aspectos, providas pelo *Spring AOP*. O SIGE também acessa uma base de dados *DB2*. Uma pasta chamada "resources" contém os arquivos de configuração do projeto, dentre os quais se encontram o "pom.xml" onde todas as dependências do *Maven* são configuradas e o arquivo "sigeContext.xml" contendo todas as configurações do *Spring*. O interesse transversal implementado pelos mecanismos de POA é o *logging*, responsável pelo *logging* das assinaturas dos métodos executados nas camadas de serviço e DAO[5] do sistema para fins de auditoria.

Para executar o experimento, uma cópia da última versão do SIGE foi feita e nomeada como versão V0. Um projeto foi criado no *Eclipse* e essa versão foi importada para o projeto. A versão V0 tem o temanho de 297.915 LOC, considerando arquivos das seguintes extensões: java, jsp, js, html, xml e css.

A partir da versão V0, outras versões foram construídas conforme apresentado na Seção 4.7. Esse trabalho adotou a mesma idéia da orientação a objetos com relação ao conceito herança para as versões geradas. Cada versão gerada preserva as mesmas propriedades de sua versão geradora, ou versão pai, e tem as propriedades de interesse sobrescritas de forma a proporcionar a comparação dos fatores de POA propostos, de forma que possam ser medidos e analisados.

Três tipos de processos de *weaving* foram considerados com relação a POA, que são *compile-time weaving*, *load-time weaving* e *runtime weaving*. O processo de construção das versões utilizando essas diferentes formas de *weaving* estão descritos na Seção 4.7. O processo de *compile-time weaving* é provido por um *plugin* do *Eclipse*, o qual compila o projeto quando habilitado. Esse processo é detalhado na Seção 4.7.1. Para o *load-time weaving* e *runtime weaving*, representados pelos *weavers AspectJ* e *Spring AOP* respectivamente, as seguintes bibliotecas foram adicionadas ao arquivo "pom.xml":

- spring-aop.jar 3.0.0.RELEASE
- aspectjrt.jar 1.7.4
- aspectjweaver.jar 1.7.4
- spring-instrument.jar 3.2.4.RELEASE

[1] https://www.eclipse.org/
[2] http://maven.apache.org/
[3] http://struts.apache.org/
[4] http://spring.io/
[5] DAO ou Data Access Object é um padrão para persistência de dados que preconiza a separação entre as regras de negócio e as regras de acesso a banco de dados

Na Figura 4.1 é apresentado um trecho de código do arquivo "pom.xml" contendo a configuração para essas bibliotecas.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>3.0.0.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.7.4</version>
</dependency>

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.7.4</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-instrument</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
```

Figura 4.1: Configurações necessárias no arquivo pom.xml para habilitar os processos de *load-time weaving* e *runtime weaving*.

A biblioteca que contém o agente Java é a biblioteca "spring-instrument.jar" e é utilizada para habilitar o processo de *load-time weaving*.

A classe responsável por implementar o aspecto *logging* é a classe *AspectProfiler*. O estilo de aspecto adotado para habilitar e configurar POA na classe foi o estilo de anotação do *AspectJ*, provido pelo suporte ao *AspectJ* do *Spring AOP*. Cada método dessa classe é composto por uma anotação de *advice* onde a expressão do *pointcut* é usada para combinar os *join points*, que representam execução de método. Apesar do fato que o *AspectJ* possui várias definições para *join points*[6], os *join points* da classe *AspectProfiler* sempre representam uma execução de método por causa da limitação do *Spring AOP* onde *join points* definem apenas execução de método. Esse tipo de *join point* foi adotado como padrão no estudo para possibilitar a avaliação dos fatores de POA entre os diferentes *weavers*. Na Figura 4.2 o conteúdo padrão da classe *AspectProfiler* é apresentado.

O *Eclipse* foi configurado para executar com o *plugin* do *Maven m2e* e o *plugin* do *Tomcat*, utilizado para executar versões da região E do mapa de versões. Como detalhados na Seção 4.5, dois tipos de invocações foram considerados na execução do experimento, que são invocações internas e externas. As invocações externas são geradas pela ferramenta *JMeter*[7] enquanto as invocações internas foram geradas por uma classe de testes *JUnit*, que é explicada com detalhes na Seção 4.6. A ferramenta *JMeter*[8] se mostrou

[6]http://www.eclipse.org/aspectj/doc/next/progguide/semantics-joinPoints.html
[7]https://jmeter.apache.org/
[8]https://jmeter.apache.org/

```
1  package br.ufu.sige.aspect;
2  import org.apache.log4j.Logger;
10
11 @Component
12 @Aspect
13 public class AspectProfiler {
14     private Logger log = Logger.getLogger(this.getClass());
15
16     @Before("execution(* br.ufu.sige.business.*.*(..))")
17     public void adviceBeforeBusiness(JoinPoint joinPoint) {
18         logEntry(joinPoint);
19     }
20
21     @Before("execution(* br.ufu.sige.dao.*.*(..))")
22     public void adviceBeforeDAOs(JoinPoint joinPoint) {
23         logEntry(joinPoint);
24     }
25
26     @Around("execution(* br.ufu.sige.business.*.*(..))")
27     public Object adviceAroundBusiness(ProceedingJoinPoint joinPoint) throws Throwable {
28         return joinPoint.proceed();
29     }
30
31     @Around("execution(* br.ufu.sige.dao.*.*(..))")
32     public Object adviceAroundDAOs(ProceedingJoinPoint joinPoint) throws Throwable {
33         return joinPoint.proceed();
34     }
35
36     @After("execution(* br.ufu.sige.business.*.*(..))")
37     public void adviceAfterBusiness(JoinPoint joinPoint) throws Throwable {
38     }
39
40     @After("execution(* br.ufu.sige.dao.*.*(..))")
41     public void adviceAfterDAOs(JoinPoint joinPoint) throws Throwable {
42     }
43
44     private void logEntry(JoinPoint joinPoint) {
45         log.info("Chamada para método: "+joinPoint.getSignature().getName());
46     }
47
48 }
49
```

Figura 4.2: Código padrão da classe *AspectProfiler*.

adequada para os propósitos desse experimento. *JMeter* foi adotada por possuir os seguintes requisitos: gratuita, independente de plataforma, código aberto, projetada para testar aplicações *web*, capaz de medir o tempo das invocações e possibilitar a geração de um plano de teste que pode ser repetido.

Com relação às medições dos fatores de POA propostos, dois métodos foram adotados para medição do tempo, de acordo com o tipo de invocação. A classe *StopWatch* foi escolhida para medir o tempo das invocações internas pois o sistema original já apresentava mecanismos de medição de tempo, os quais utilizam essa classe, que se mostrou adequada aos propósitos do estudo. Para as invocações externas, o relatório fornecido pelo *JMeter* foi adotado para medir o tempo.

Juntamente com o *JMeter*, a ferramenta *VisualVM* foi a ferramenta de *profiling* selecionada para as medições de CPU e memória. Os critérios que direcionaram a escolha dessa ferramenta foram: gratuita, independente de plataforma e capaz de monitorar o

consumo de CPU e memória em tempo real. Outras ferramentas como *TPTP* e *JVM monitor* também foram selecionadas na busca, no entanto a ferramenta *TPTP* foi descartada porque foi descontinuada e *JVM monitor* não apresentou estabilidade ao executar dentro do *Eclipse*.

Para algumas versões do sistema, uma ferramenta se mostrou necessária para coletar métricas dos arquivos do tipo *jar*, como tamanho e número de classes de uma versão. *CyVis* foi a ferramenta que se mostrou adequada para executar essa coleta de métricas por se enquadrar nos seguintes critérios: gratuita, capaz de coletar métricas de arquivos do tipo *jar* onde não existem os códigos fontes e independente de plataforma. Uma aplicação independente foi priorizada sobre *Plugins* do *Eclipse* para minimizar a possibilidade de incompatibilidades com os *Plugins* já instalados.

Uma ferramenta para contar o LOC de diferentes versões também foi necessária. Os critérios para seleção dessa ferramenta foram: gratuita, independente de plataforma, com foco na linguagem Java apenas e capaz de contar o LOC de arquivos cujas extensões devem incluir: java, jsp, js, html, xml e css. A ferramenta *LOC counter*, que é uma aplicação Java usando *Swing*, se mostrou adequada aos requisitos e por isso foi selecionada.

Para a execução do experimento, dois tipos de configurações de *debug* foram criadas no *Eclipse*, sendo um tipo para as invocações internas representadas por configurações *JUnit* e outro tipo para as invocações externas, representada por configurações *Maven build*:

1. Configurações *JUnit*: duas configurações do tipo *JUnit* foram criadas para executar a classe de testes *JUnit*. Ambas apontam para o mesmo projeto "sige-exp-template" e para a mesma classe de testes *JUnit*, que é a classe "br.ufu.sige.test.ExperimentTest". A diferença entre elas está nos argumentos passados para a VM. A primeira delas, nomeada como "Default app test" não possui argumentos para a VM e é utilizada para executar todos os testes de versões que não possuem o processo de *load-time weaving*. A segunda configuração, nomeada como "LTW app test", possui um agente Java que é passado como parâmetro para habilitar o processo de *load-time weaving* e é utilizada para executar as versões que fazem uso desse processo de *weaving*. Na Figura 4.3 são apresentadas essas configurações. A variável "M2-REPO" representa a pasta onde o repositório local da máquina é localizado.

2. Configurações *Maven build*: duas configurações de *Maven build* foram criadas para implantar a aplicação em um servidor *Tomcat* embutido, possibilitando, dessa forma, que as invocações externas geradas a partir do JMeter façam requisições na aplicação. Essas duas configurações foram nomeadas como "Experiment - tomcat run" e "Experiment LTW - tomcat run" e apontam para o mesmo projeto, que também é o projeto "sige-exp-template" e configura o *goal* do tipo *run* do *plugin* do *Tomcat*, como mostra a Figura 4.4. A diferença entre elas também é o parâmetro passado

para a VM, configurado da mesma maneira como nas configurações *JUnit*. Da mesma forma, a primeira configuração *Maven build* destina-se a testar versões da aplicação com invocações externas, que não contém o processo de *load-time weaving*, enquanto a segunda é exclusivamente para testar as invocações externas contendo o processo de *load-time weaving*.

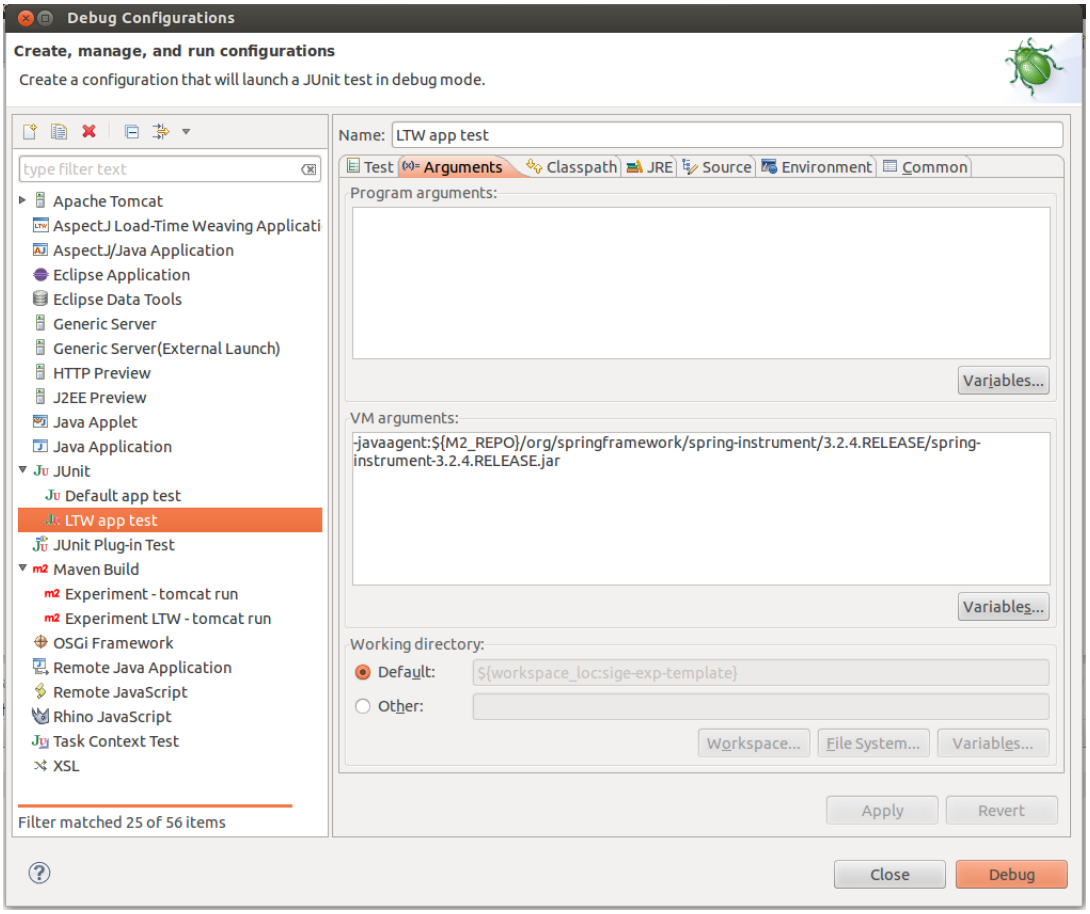

Figura 4.3: Configurações de *JUnit* do *Eclipse*.

Com relação as configurações de ambiente, todos os testes foram executados sob as mesmas condições de hardware e software. O hardware foi composto por um *notebook* com processador *Intel Core* i5-2410M 2.30 GHz e 4 GB de memória RAM executando sistema operacional *Ubuntu* versão 12.04 LTS. A versão do Java utilizada foi 1.7.0. Antes de cada medição dos testes, a memória utilizada e as condições de *swap* foram verificadas por meio do monitor do sistema do *Ubuntu*. Casos em que essas condições de memória e *swap* estavam acima de 55% e 2% respectivamente, o sistema operacional era reiniciado antes do teste. Os processos que executaram em paralelo aos testes, além dos processos regulares do sistema operacional, foram *Eclipse*, *System Monitor*, *Gnumeric Spreadsheet*, VisualVM e nos estágios específicos das invocações externas, JMeter. Apesar do fato que apenas 4 versões da aplicação acessam o banco de dados, o banco de dados *DB2* (versão 9.7) permaneceu sempre ativo para que as aplicações executassem.

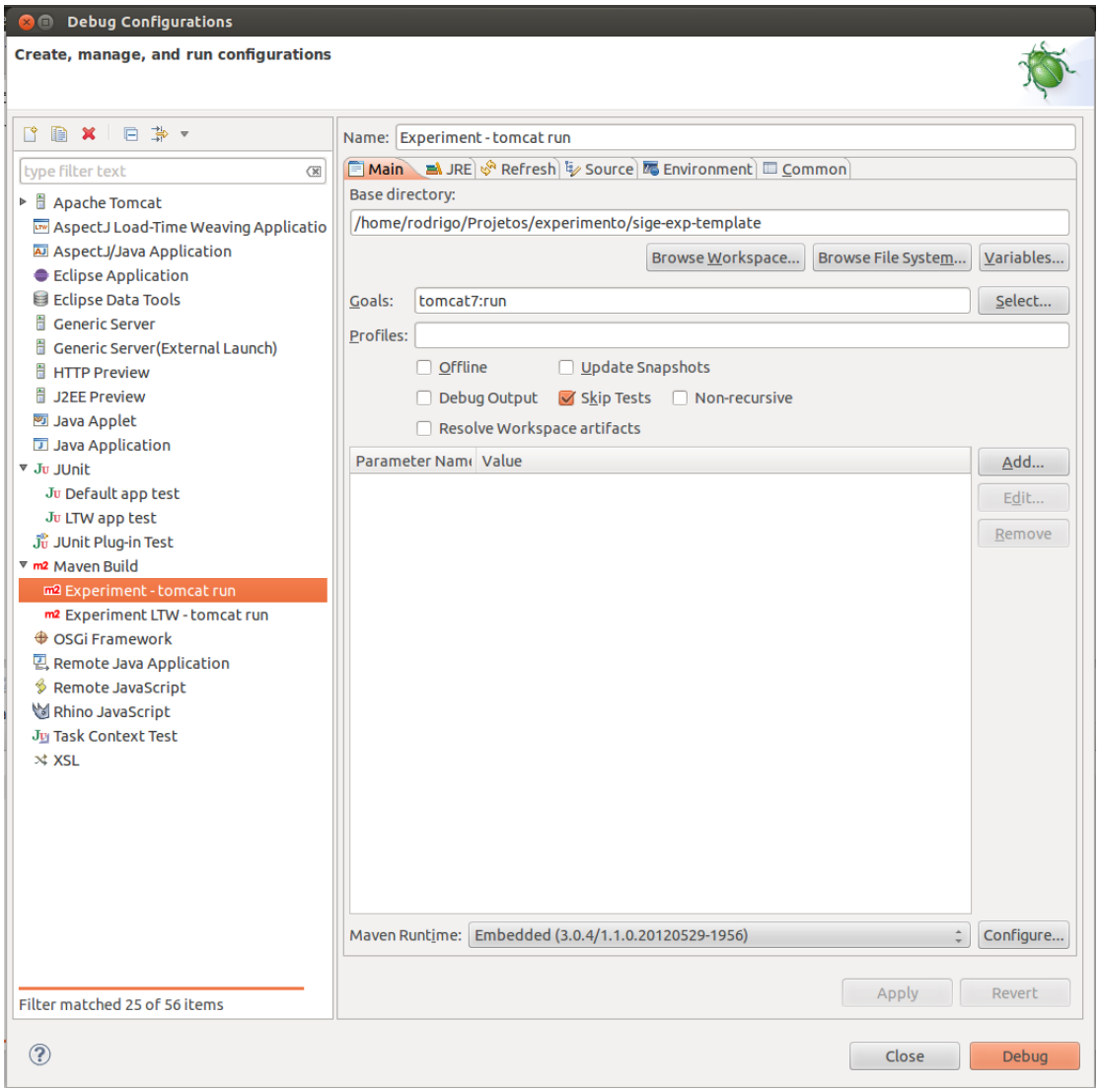

Figura 4.4: Configurações de *Maven build* do *Eclipse*.

Durante os testes a rede Internet foi desabilitada para evitar maior geração de I/O.

4.2 Mapa das Versões

Na Figura 4.5 é apresentado o mapa das versões propostas para o experimento. A imagem é dividida em regiões contendo as versões a serem comparadas entre si para a análise dos fatores propostos relacionados a POA. Cada região é descrita a seguir. Mais detalhes sobre cada versão são descritos na Seção 4.7.

1. Região A: agrupa versões com diferentes processos de *weaving*. Contém uma versão sem POA (versão V1-A) e outras três versões com três diferentes processos de *weaving*, que são *compile-time weaving* (versão V2-A), *load-time weaving* (versão V3-A) e *runtime weaving* (versão V4-A).
2. Região B: agrupa versões com o processo *load-time weaving*. As versões dessa região são V3-B, V3-C, V3-D, V3-E e V3-F. O conjunto composto pelas versões V3-B, V3-C e V3-D diferem no valor da variável *loopCount*, responsável por representar o número de invocações geradas no teste. O conjunto composto pelas versões V3-B, V3-E e V3-F diferem no número de classes carregadas durante o processo de *load-time weaving*.
3. Região C: agrupa duas versões que são V4-A e V6, construídas para executar o processo de *runtime weaving* que diferem no LOC.
4. Região D: agrupa 12 versões construídas para executar o processo de *runtime weaving*, que diferem entre si no número de *join points* e tipos de *advice*. Essa região possui três subgrupos compostos por versões contendo 8,16 e 32 *join points* e quatro subgrupos compostos por versões contendo os *advices Around, Before, After* e *AfterThrowing*. As versões dessa região são V5-A, V5-D, V5-G, V5-J, V5-B, V5-E, V5-H, V5-K, V5-C, V5-F, V5-I e V5-L.
5. Região E: assim como a região A, agrupa versões contendo diferentes processos de *weaving*, porém essas versões fazem acesso ao banco de dados e possuem a camada *controller* completa do *front end* da aplicação. As versões que compõem essa região são V1-B, V2-2, V4-B e V3-G.

As regiões A,B,C e D contém versões com apenas parte do *controller* do *front end* da aplicação e não possuem acesso ao banco de dados. O propósito é permitir que os testes ocorram com um grande número de repetições, possibilitando assim a avaliação do comportamento dos fatores de POA propostos de uma maneira isolada de elementos que consomem recursos, tais como o acesso ao banco de dados e a completa cadeia de mecanismos do *framework Struts*. Versões da região E, ao contrário, contém ambos elementos e foram projetadas para analisar o impacto de POA no desempenho na presença de tais elementos.

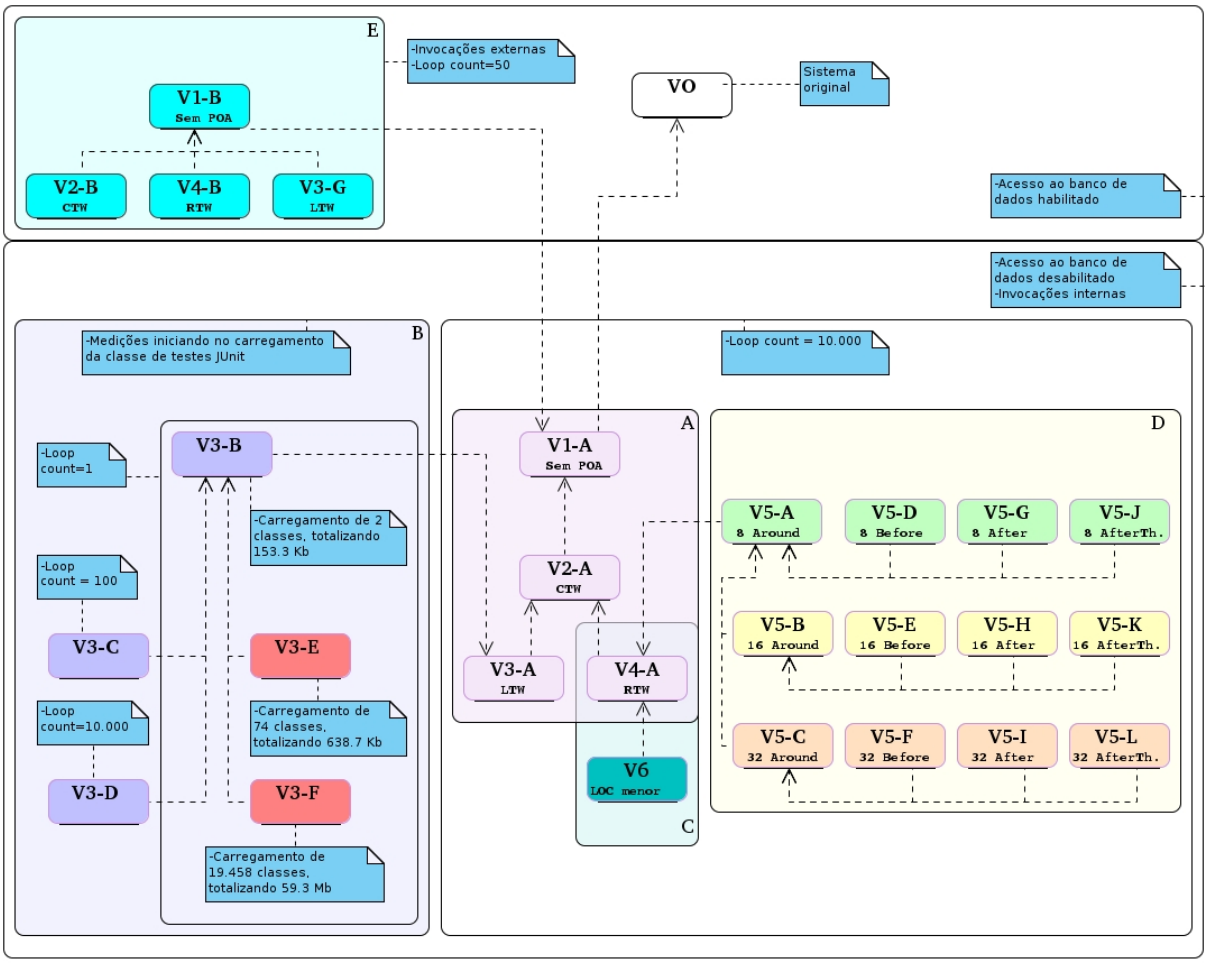

Figura 4.5: Mapa das versões do experimento.

4.3 Cenário

O cenário selecionado para o experimento foi a autenticação de usuário. A escolha desse cenário foi baseada nos seguintes critérios:

1. Possui várias chamadas de métodos em diferentes camadas do sistema.
2. Não possui complexa regra de negócio [9].
3. A refatoração para introdução de POA não exige grande esforço.

Na versão original (V0), a autenticação de usuário envolve 2 passos. O primeiro é responsável por verificar se o usuário está autorizado a acessar o sistema e o segundo pelo carregamento de configurações do usuário. Duas bases de dados são acessadas. Uma delas é uma base de dados em um sistema legado onde as credenciais são validadas e a outra é onde as configurações do usuário são carregadas. A autenticação é um passo necessário para todos os usuários que acessam a aplicação. Quando o usuário entra com as credenciais por meio de um *browser*, o sistema verifica se o usuário é autorizado (primeiro passo) e

[9] O objetivo é focar em aspectos técnicos de POA e não em processos de negócio.

caso seja, redireciona-o para o carregamento e verificação de suas configurações (segundo passo), do contrário uma mensagem de erro é apresentada na tela. O carregamento dessas configurações compreende o carregamento de perfis, funcionalidades que o usuário pode acessar e outras configurações específicas necessárias durante a sessão do usuário.

A implementação do primeiro passo foi substituída por um método que carrega o usuário por meio de uma identificação. Essa substituição foi necessária porque, no cenário real, a autenticação acessa um *web service* que consome recursos e poderia atrapalhar as medições do experimento.

Para as versões das regiões A,B,C e D do mapa de versões, o cenário foi refatorado de forma que as invocações não acessam o banco de dados. O motivo é o mesmo da substituição da autenticação: os recursos consumidos pelas operações de *I/O* influenciariam as medições do aspecto *logging*. Da mesma forma, na camada do sistema responsável pelo acesso aos dados (DAO), todos os métodos do cenário retornam com o mínimo de processamento e simulam uma autenticação de sucesso. Essa refatoração não alterou o propósito dos métodos ou a sequência das operações do cenário e a versão resultante, nomeada como V1-A, foi utilizada para gerar todas as outras versões nas quais os fatores são variados e as variáveis dependentes medidas.

Apesar do fato que as versões da região E do mapa de versões possuem o banco de dados habilitado, elas também foram geradas a partir da versão V1-A, herdando seus atributos, ao invés da versão V0, pois a habilitação do banco de dados foi mais simples que a substituição da autenticação que utiliza *web services*.

Na Figura 4.6 são apresentados os métodos do cenário na ordem em que são invocados entre as camadas da aplicação.

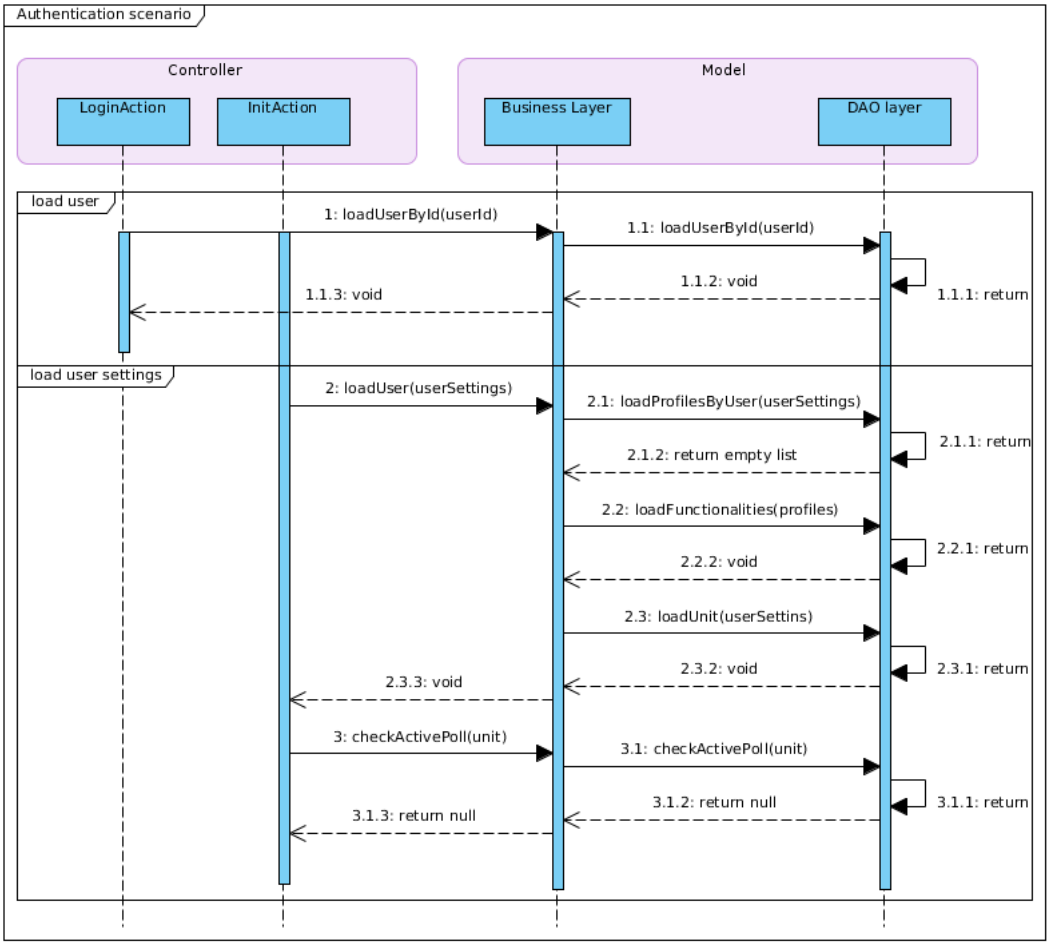

Figura 4.6: Métodos invocados durante o processamento do cenário.

4.4 Corretude das Invocações

A maneira como o interesse transversal *logging* foi implementado em cada versão poderia ser um fator de influência nas medições de forma a prejudicar o estudo se essa influência variasse de uma versão para outra, produzindo diferentes impactos no desempenho. Para evitar essa variação, a saída do *logging* gerada pelo cenário foi composto por uma *String* de valor fixo: "Chamada de método:" acrescida da assinatura do método sendo invocado. Os métodos são apresentados na Figura 4.6 e representam os estágios 1, 1.1, 2, 2.1, 2.2, 2.3, 3 e 3.1.

Na versão V1-A, onde POA está ausente, o *logging* é implementado em cada classe que contém os métodos do cenário manualmente. Em cada versão gerada a partir de V1-A, onde POA está presente, o *logging* é implementado pela classe *AspectProfiler*, como detalhado na Seção 4.1. O método *logEntry()* da classe *AspectProfiler*, responsável pelo *logging* recebe um parâmetro para ser utilizado na mensagem do *log*, que é o parâmetro *joinPoint* representando um objeto da classe *org.aspectj.lang.JoinPoint*. Vários métodos dessa classe foram testados de forma a gerar a mensagem que representa a assinatura do método sendo executado em cada estágio da execução do cenário, considerando as

diferentes implementações dos métodos das classes do pacote *org.aspectj.lang*, para os diferentes *weavers* utilizados no experimento. O método que produziu a mesma saída para todos os *weavers* foi o método *getName()* da classe *org.aspectj.lang.Signature*. O código resultante responsável pela geração das mensagens de *log* em todas as versões foi: "log.info("Chamada para método: "+joinPoint.getSignature().getName());".

Todas as versões do experimento foram testadas, de forma a garantir que as diferentes implementações do interesse transversal produzissem as mesmas mensagens de *log* e com isso garantir que essas implementações são corretas. A versão V1-A produz *logs* manualmente, contendo o mesmo código existente no método *logEntry()*, mas não possui o esforço extra de processamento para a chamada do método *logEntry()*, presente nas outras versões que possuem POA. Entretanto, é esperado que esse esforço extra não represente uma ameaça para o experimento, pois além de ser considerado muito pequeno, está ausente em apenas 2 das versões, as quais não possuem POA, que são V1-A e V1-B.

4.5 Tipos de Invocações

Quando o usuário interage com a aplicação e executa o cenário na versão V0, requisições geradas pelo *browser* por meio do protocolo HTTP geram invocações que passam pelas várias camadas do sistema, sendo a primeira delas a camada *Controller* do *framework Struts*, a qual é composta por uma complexa cadeia de mecanismos. Essas invocações navegam por esses mecanismos até alcançarem as classes *Actions* da aplicação, que são *LoginAction* e *InitAction*. Os métodos invocados nessas *Actions* acessam a camada de serviço, cujos métodos acessam a camada DAO, que por sua vez acessa o banco de dados. Para reproduzir essa interação do usuário com a aplicação, o *JMeter* foi configurado para gerar essas requisições, chamadas nesse trabalho de invocações externas, e um plano de experimento foi configurado no *JMeter*, composto por requisições HTTP que apontam para um caminho que aciona o cenário de autenticação do usuário. No arquivo "struts.xml" da aplicação, uma configuração de *Action* representando essa autenticação de usuário, cujo nome é "simulateLoginUser", é responsável por interceptar as chamadas e redirecioná-las para uma segunda configuração de *Action*, que é "initPortal". Da mesma forma, essa segunda configuração redireciona as chamadas para um método onde a execução prossegue. Essas configurações são responsáveis por redirecionar as chamadas para métodos que compõem o cenário. O plano do experimento configurado no *JMeter* gera as requisições para o *Controller* da aplicação, que executa em um servidor *Tomcat* configurado em uma máquina local, previamente inicializado por uma configuração de *Maven build*, conforme detalhado na Seção 4.1. Na Figura 4.7 é representado o caminho das invocações externas através da aplicação. Os números representam a ordem de execução dos eventos. Os passos 10 e 21 existem apenas nas versões da região E do mapa de versões.

Além das invocações externas, outro tipo de invocação foi criado, que é a invocação

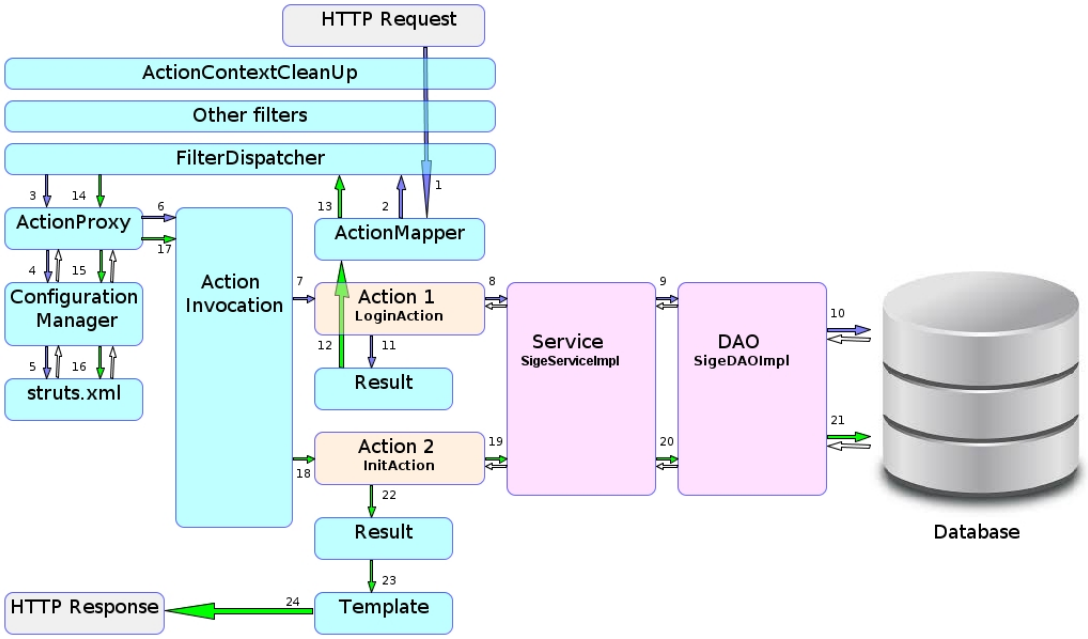

Figura 4.7: Caminho das invocações externas através da aplicação.

interna. A razão da criação desse tipo de invocação foi reduzir o caminho da invocação através da aplicação e com isso concentrar as medições apenas nas camadas de serviço e DAO. Para a geração dessas invocações internas, uma classe *JUnit* chamada "Experiment-Test" foi criada simulando o comportamento das invocações externas, mas sem conter a complexa cadeia de mecanismos do *framework Struts*. As invocações internas são geradas pela classe *JUnit* que acessa ambos passos do cenário proposto sequencialmente. A implementação desses passos, no entanto, não invoca os serviços diretamente, mas sim métodos nas *Actions* que invocam os serviços após um processamento interno. Na Figura 4.8 é representado o caminho das invocações internas através da aplicação. Na Figura 4.9 é apresentado o código da classe *JUnit*.

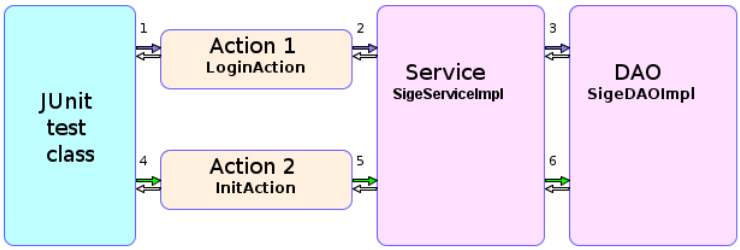

Figura 4.8: Caminho das invocações internas através da aplicação.

Na classe de testes *JUnit* as variáveis são inicializadas entre as linhas 19 e 33, onde alguns valores iniciais são atribuídos e as injeções de dependências são configuradas por meio da anotação *Autowired*. Um bloco de inicialização entre as linhas 35 e 39 é responsável por invocar o *garbage collection* antes da execução de cada teste. O método *loggingNoAOP()* é invocado no teste por meio da anotação na linha 41 e é responsável por invocar os dois passos do cenário dentro de um *loop* que itera o número de vezes

configurado pela variável *loopCount*. O tempo da geração dessas invocações é medido por meio da classe *StopWatch*, onde a contagem é iniciada na linha 45 e termina na linha 50. O resultado do tempo gasto em valor absoluto é impresso no console por meio do código da linha 51 e a média de tempo por invocação é impressa por meio do código da linha 53. As linhas comentadas demonstram códigos utilizados em algumas versões, conforme detalhado na Seção refsecao:design.

```
1  package br.ufu.sige.test;
2  import org.junit.Test;
15
16 @RunWith(SpringJUnit4ClassRunner.class)
17 @ContextConfiguration(locations= "/sigeContext.xml")
18 public class ExperimentTest {
19     @Autowired
20     LoginAction loginAction;
21     @Autowired
22     InitAction initAction;
23     public static int loopCount = 10000;
24     //public static int loopCount = 1;
25
26     @Autowired
27     UserSettingsSige userSettingsSige;
28
29     @Autowired
30     StopWatch stopWatch;
31     Long timeSum=0l;
32
33     protected static Logger LOG = LoggerFactory.getLogger(ExperimentTest.class);
34
35     {
36         System.gc();
37         //this.stopWatch = new StopWatch();
38         //this.stopWatch.start();
39     }
40
41     @Test
42     public void loggingNoAOP() throws InterruptedException {
43
44         userSettingsSige.setAuthenticated(true);
45         stopWatch.start();
46         for (int j = 0; j < loopCount; j++) {
47             loginAction.simulateLoginUser();
48             initAction.init();
49         }
50         this.stopWatch.stop();
51         LOG.info("Stress test executed in " + stopWatch.getTotalTimeMillis());
52         timeSum += stopWatch.getTotalTimeMillis();
53         LOG.info("Average time: "+timeSum/loopCount);
54
55     }
56 }
```

Figura 4.9: Código da classe *JUnit* utilizada para a geração das invocações internas.

As invocações internas foram criadas com base na hipótese de que a complexa cadeia de mecanismos do *framework Struts*, juntamente com a presença do acesso ao banco de dados consome uma quantidade considerável de recursos que, nesse estudo, são representados pelas variáveis dependentes. Se essa hipótese está correta, as medições das variáveis dependentes são fortemente influenciadas pela presença desses dois elementos na aplicação. O uso das invocações internas para algumas etapas do experimento, onde o banco de dados é desabilitado, serviria para evitar que tais elementos influenciassem nas medições e dessa maneira possibilitaria a medição dos fatores propostos de maneira isolada.

De forma a confirmar ou rejeitar essa hipótese, um pequeno experimento foi realizado. A confirmação sugere que as invocações internas devem ser consideradas no experimento,

pois a cadeia de mecanismos do *framework Struts* e a presença do acesso ao banco de dados realmente consomem uma quantidade considerável de recursos e isso pode causar impacto na análise dos fatores de POA propostos. A rejeição dessa hipótese, por outro lado, sugere que as invocações externas são o suficiente para a realização do experimento. Uma vez que métricas de desempenho como CPU ou memória necessitam de uma ferramenta específica para tais medições, o tempo foi utilizado como métrica de medição por ser facilmente obtido por diversos meios. O teste foi composto por duas partes:

1. Medições de invocações externas: para executar esse teste, a versão V1-A foi implantada em um servidor *Tomcat* local e o *JMeter* foi configurado para gerar 1.000 requisições HTTP de forma sequencial apontando para a *Action* que inicia o cenário. O plano do experimento do *JMeter* foi executado 10 vezes. Na Tabela 4.1 são apresentados os resultados das médias de tempo por invocação, apresentadas pelo relatório do *JMeter*.

Tabela 4.1: Resultados de 10 médias de tempo por invocação externa apresentadas pelo relatório do *JMeter*.

Execução	1	2	3	4	5	6	7	8	9	10
Tempo(ms)	295	296	296	296	296	296	296	296	297	297

2. Medições de invocações internas: a classe *JUnit* foi configurada para executar o método *loggingNoAOP* na versão V1-A. A variável *loopCount*, conforme detalhado na Seção 4.6, foi configurada com valor 1.000. O teste foi executado 10 vezes e o tempo gasto na execução foi apresentado no console do *Eclipse* após cada execução. Na Tabela 4.2 são apresentados os resultados das médias de tempo por invocação.

Tabela 4.2: Resultados de 10 médias de tempo por invocação, calculadas a partir da classe *StopWatch* e divididas por 1.000.

Execução	1	2	3	4	5	6	7	8	9	10
Tempo(ms)	1,537	1,546	1,547	1,571	1,612	1,616	1,616	1,618	1,681	1,711

Os resultados apresentados na Tabela 4.2 são os resultados da divisão do total de tempo gasto por 1.000, fornecidos pelo código da linha 53 da classe de testes *JUnit* a cada chamada de teste. A média de tempo por invocação, considerando todos os resultados foi de 1,605 ms, enquanto a média de tempo nas invocações externas foi de 296,1 ms. Essa diferença confirma a hipótese, direcionando as invocações externas para serem utilizadas apenas quando POA é analisada no contexto geral, onde se consideram elementos de grande influência nas variáveis dependentes propostas, que é o caso das versões da região E do mapa de versões. O caminho completo percorrido pelas invocações através da cadeia de mecanismos do *framework Struts* consome uma quantidade considerável de recursos de tempo, sem mencionar outros recursos como CPU e memória, o que direciona as invocações internas para serem consideradas

o tipo de medição mais adequado para o experimento para as versões das regiões A,B,C e D do mapa de versões, onde os fatores propostos são analisados de maneira isolada.

4.6 Plano do Experimento

A fim de mensurar as variáveis dependentes propostas um plano de experimento padrão foi criado. Esse plano é composto por um parâmetro com valor fixo e parâmetros com valores variáveis que são configurados em cada versão da aplicação.

O parâmetro com valor fixo é o número de execuções do teste, que define quantas vezes o teste é executado a fim de gerar os valores para as variáveis dependentes e foi definido com o valor 15.

Um dos parâmetros de valor não constante é a variável *loopCount*, que é um atributo da classe de testes *JUnit* para o caso das invocações internas e é configurada no plano do experimento do *JMeter* para o caso das invocações externas. Essa variável define quantas vezes cada invocação interna ou externa é executada sobre o cenário em um teste e foi inicialmente configurada com o valor 10.000. O critério para a escolha desse número foi baseado no fato que deveria ser alto o suficiente para enfatizar o pequeno impacto no desempenho causado por algum fator de interesse proposto mas baixo o suficiente para viabilizar a avaliação das invocações em um tempo razoável. A partir de um conjunto de opções de valores, que foram 1,10,100,1.000,10.000 e 100.000, o número 10.000 foi considerada a melhor opção para versões com o banco de dados desabilitado e cujo método de medição é feito por meio das invocações internas, que são as versões das regiões A,B,C e D do mapa de versões. Essa escolha foi configurada inicialmente considerando que a maioria das versões, 22 de 26, foram geradas sob essas especificações. Parâmetros com valores variáveis que compõem a classe *AspectProfiler* são o número de *advices*, tipos de *advices* e expressões dos *pointcuts*. Os outros parâmetros de valores variáveis são os *weavers* e outras variáveis específicas de cada fase do experimento, que são detalhadas em cada uma das fases na Seção 4.7.

A medição das variáveis dependentes foi configurada para ser realizada de duas formas. Para as invocações internas a execução de cada teste é iniciada por meio do *Eclipse*. Um *breakpoint* é colocado no início da execução do teste na classe *JUnit*, onde o método *start()* da classe *StopWatch* é invocado e inicia a contagem de tempo. Esse *breakpoint* é necessário para possibilitar a inspeção do processo por meio da ferramenta *VisualVM*. Quando a *thread* do *JUnit* que está sendo executada para no *breakpoint*, o *VisualVM* é configurado para inspecionar apenas a memória e a CPU do processo referente a aplicação. Depois desses passos, o *breakpoint* é liberado, a aplicação executa e o mecanismo de aspectos age sobre os métodos invocados durante a execução do cenário. Após a execução do processo, a classe *StopWatch* calcula o tempo gasto nas invocações e apresenta um relatório no

console do *Eclipse*, enquanto o *VisualVM* gera um relatório em tempo real do histórico da memória e CPU do processo da aplicação. A recuperação dos valores de memória e CPU, respectivamente em MBs e em porcentagem, são feitas manualmente por meio do relatório do *VisualVM*, de onde os maiores valores são sempre considerados.

Para as invocações externas, não existe a necessidade de *breakpoints* para a inspeção de memória e CPU. Após a inicialização do servidor *Tomcat* por meio do *Eclipse*, a ferramenta *VisualVM* é capaz de inspecionar o processo da aplicação antes do início das invocações. Após configurar o *VisualVM* para inspecinar a memória e CPU no processo iniciado pelo *Tomcat*, o plano do experimento do *JMeter* é invocado e o teste é executado. Após a execução do teste, o *JMeter* emite um relatório contendo o tempo gasto enquanto a memória e CPU são recuperados da mesma forma como nas invocações internas. Na Figura 4.10 é apresentado o monitoramento de memória e CPU feito pelo *VisualVM*.

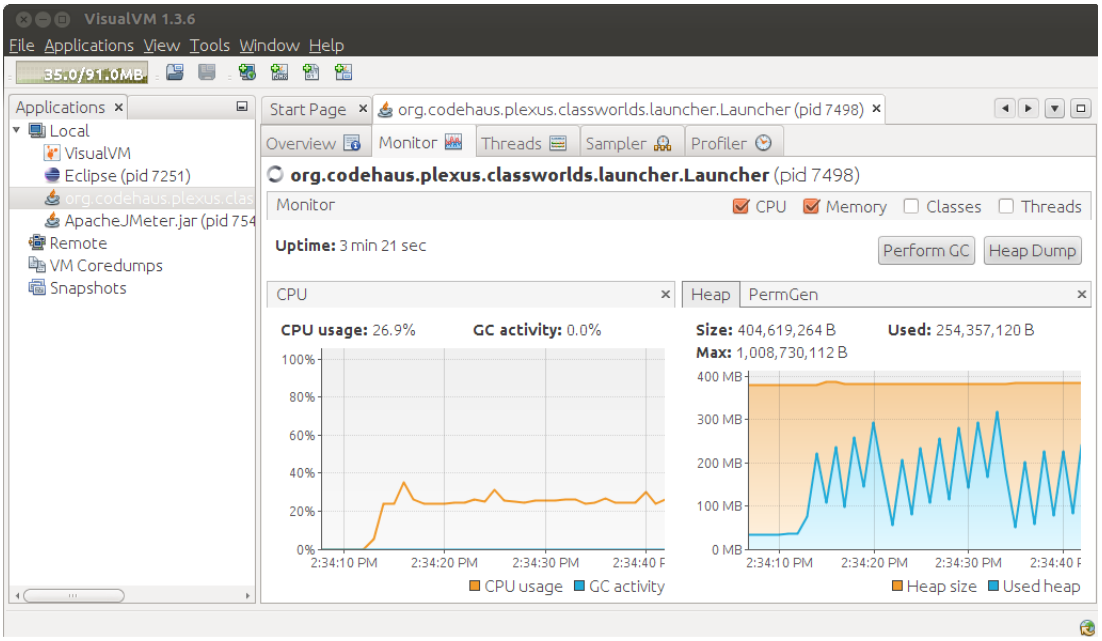

Figura 4.10: Monitoramento de memória e CPU feito pelo *VisualVM*.

4.7 Projeto do Experimento

Um projeto do experimento [Juristo e Moreno 2010] foi proposto a fim de medir as variáveis dependentes, na medida em que são variados os fatores de interesse propostos durante o experimento. O experimento foi dividido em fases, conforme mostrado a seguir. O plano do experimento foi executado em cada uma das fases, com as devidas parametrizações. Nas próximas subseções os detalhes da geração das versões da aplicação são apresentados. Os resultados da execução do plano do experimento são apresentados no Capítulo 5.

4.7.1 Versões da Região A do Mapa de versões

O propósito dessa fase é a geração de versões compreendendo os diferentes tipos de processo de *weaving* adotados no experimento e servir de modelo para a geração de versões das outras regiões do mapa de versões. Nessa seção são apresentadas as gerações das versões V1-A, V2-A, V3-A e V4-A.

Versão V1-A

O propósito dessa versão é estabelecer patamares de valores relacionadas às variáveis dependentes para serem comparadas com valores obtidos em outras fases do experimento. O processo da construção da versão V1-A foi descrito na Seção 4.3.

Versão V2-A

A partir da versão V1-A, a versão V2-A foi gerada. Essa versão têm o propósito de avaliar o impacto no desempenho ao utilizar o *weaver AspectJ* com o processo de *compile-time weaving* quando comparado com a versão V1-A. Nessa versão todas as gerações de *log* manuais foram removidas dos métodos do cenário. O *weaver* foi provido pelo *plugin AspectJ Development Tools*, instalado a partir do *Marketplace* do *Eclipse*, como mostrado na Figura 4.11.

Esse *plugin*, quando habilitado, força a recompilação do projeto e dessa forma injeta os *bytecodes* dos aspectos para as classes em tempo de compilação. A biblioteca do *plugin* é mostrada pelo *Eclipse* logo abaixo das dependências do *Maven* na estrutura de projetos do *Eclipse*, como mostrado na Figura 4.12.

Os seguintes parâmetros foram configurados:

- *Weaver*: *AspectJ compile-time weaver*.
- Número de *advices*: 6
- Tipos de *advices*: 2 *Before*, 2 *Around*, 2 *After*, com as mensagens de *log* invocadas apenas em *advices* do tipo *Before*.
- Expressões de *pointcut*: interceptação de todos os métodos presentes nos pacotes de serviço e DAO.

O propósito dessa fase é avaliar se o *weaver*, no caso o *AspectJ* em tempo de compilação, causa impacto no desempenho, sensibilizando as variáveis dependentes ao se comparar os resultados dessa versão com os resultados da versão V1-A.

Versão V3-A

A partir da versão V2-A, a versão V3-A foi gerada. Essa versão têm o propósito de avaliar o impacto no desempenho ao utilizar o *weaver AspectJ* com o processo de *load-time*

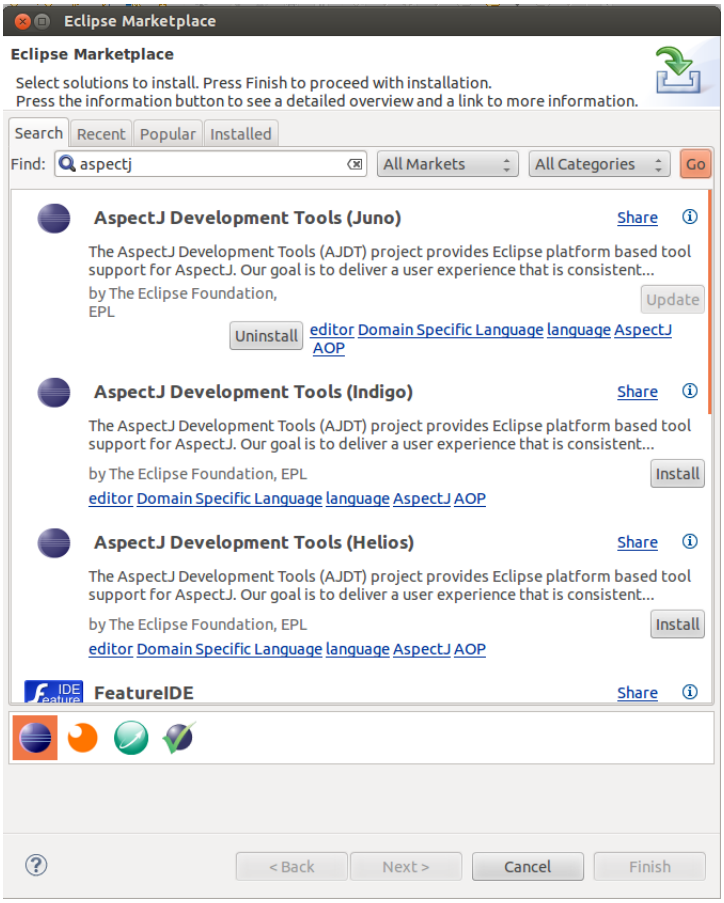

Figura 4.11: Instalação do *plugin AspectJ Development Tools* a partir do *Marketplace* do *Eclipse*.

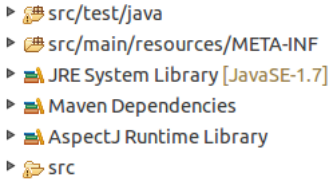

Figura 4.12: Biblioteca do *AspectJ* provida pelo *plugin AspectJ Development Tools*.

weaving quando comparado com a versão V1-A.

Nessa nova versão, o *plugin* do *AspectJ* que realiza a compilação dos aspectos foi desabilitado e o parâmetro tipo de *weaving* foi sobrescrito para *load-time weaving*.

O processo de *load-time weaving*[10] é usado no contexto do *framework Spring*. O *AspectJ* é responsável por realizar o *weaving* dos aspectos para dentro das classes da aplicação durante o carregamento das classes para a JVM.

Um arquivo chamado "aop.xml" foi criado na pasta "META-INF", que foi adicionada ao *build path* do projeto, com a configuração necessária para realizar o *weaving* dos aspectos da classe *AspectProfiler* para dentro das classes do projeto. Esse arquivo contém 2 seções. A primeira seção se refere a especificação das classes onde o processo de

[10]Esse processo é descrito em detalhes na documentação oficial do Spring: http://docs.spring.io/spring/docs/3.0.x/reference/aop.html

weaving irá atuar. Para essa versão essas classes são: "br.ufu.sige.dao.SigeDAOImplExp", "br.ufu.sige.business.SigeServiceImpl" e "br.ufu.sige.aspect.*". O símbolo "*" significa qualquer classe, nesse caso do sub-pacote "aspect". Esse pacote é necessário para especificar o local onde se encontra a classe *AspectProfiler*. A segunda seção do arquivo especifica as classes onde estão os aspectos a serem processados durante o processo de *weaving*. O código do arquivo "aop.xml" é apresentado na Figura 4.13 e representa o carregamento de 3 classes no processo de *load-time weaving*, onde uma delas é a classe dos aspectos *AspectProfiler* e as outras duas são as classes de serviço e DAO.

```
1  <!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
2  <aspectj>
3
4     <weaver>
5
6           <include within="br.ufu.sige.dao.SigeDAOImplExp"/>
7           <include within="br.ufu.sige.business.SigeServiceImpl"/>
8           <include within="br.ufu.sige.aspect.*"/>
9
10     </weaver>
11
12     <aspects>
13
14          <aspect name="br.ufu.sige.aspect.AspectProfiler"/>
15
16     </aspects>
17
18  </aspectj>
19
```

Figura 4.13: Conteúdo do arquivo "aop.xml", representando o carregamento de 3 classes no processo de *load-time weaving*.

Além das configurações no arquivo "aop.xml", uma configuração foi adicionada no arquivo "sigeContext.xml" para habilitar o processo de *load-time weaving*, que é o comando "<context:load-time weaver/>". A configuração necessária para executar as versões no modo *load-time weaving*, bem como as dependências de bibliotecas necessárias para a execução do processo são descritas na Seção 4.1.

Versão V4-A

A partir da versão V2-A, a versão V4-A foi gerada. Essa versão têm o propósito de avaliar o impacto no desempenho ao utilizar o *weaver Spring AOP* com o processo de *runtime weaving* quando comparado com a versão V1-A.

O *plugin* do *AspectJ* que realiza a compilação dos aspectos foi desabilitado e o parâmetro tipo de *weaving* foi sobrescrito para *runtime weaving*. No arquivo "sigeContext.xml", a configuração que habilita o *Spring AOP* foi adicionada, que é "<aop:aspectj-autoproxy />"[11].

[11] Essa configuração é descrita em detalhes na documentação oficial do Spring: http://docs.spring.io/spring/docs/3.0.x/reference/aop.html

4.7.2 Versões da Região B do Mapa de versões

Essa fase abrange a geração de versões contendo o *weaver AspectJ* com o processo de *load-time weaving* com os seguintes propósitos:

1. Avaliar o impacto no desempenho quando o número de invocações é variado.
2. Avaliar o impacto no desempenho quando o número de classes carregadas durante o processo de *weaving* é variado.

Versões com variação no número de invocações

O propósito dessa fase é avaliar o impacto no desempenho quando o número de invocações, representado pela variável *loopCount*, é variado. A partir da versão V3-A, a versão V3-B foi gerada. Na versão V3-B, o método *start()* da classe *StopWatch* é invocado na inicialização da classe *JUnit* antes da execução do método de teste da classe *loggingNoAOP()*. Para esse ajuste, na classe *JUnit* as linhas 37 e 38 foram descomentadas e as linhas 29 e 45 foram comentadas. Esse ajuste foi necessário para iniciar a medição do tempo antes do carregamento das classes para a JVM. As linhas 35 até 39 delimitam esse bloco de inicialização que é executado antes da instanciação da classe. Da mesma forma que na medição do tempo, as medições de memória e CPU foram feitas considerando esse estágio de carregamento. O *breakpoint* foi colocado na linha 38, onde a contagem do tempo é iniciada.

Na versão V3-B, a variável *loopCount* foi sobrescrita para 1 e para isso a linha 23 foi comentada e a linha 24 descomentada. A expressão dos *pointcuts* também foi sobrescrita na classe *AspectProfiler* para 2 *advices* do tipo *after*. Essas duas expressões foram:

- Interceptação dos métodos da classe "br.ufu.sige.business.SigeServiceImpl"
- Interceptação dos métodos da classe "br.ufu.sige.dao.SigeDAOImplExp"

Na Figura 4.14 é apresentado um fragmento de código contendo 2 *advices* do tipo *after* que representam a configuração dos *pointcuts* para interceptar métodos de 2 classes.

```
@After("execution(* br.ufu.sige.business.SigeServiceImpl.*(..))")
public void adviceAfterBusiness(JoinPoint joinPoint) throws Throwable {
}

@After("execution(* br.ufu.sige.dao.SigeDAOImplExp.*(..))")
public void adviceAfterDAOs(JoinPoint joinPoint) throws Throwable {
}
```

Figura 4.14: Fragmento de código representando a configuração dos *pointcuts* para interceptar métodos de 2 classes.

O *advice after* foi escolhido para conter as expressões de *pointcut* que interceptam métodos nas classes mencionadas porque o *advice before* já estava sendo utilizado para interceptar métodos das classes de serviço e DAO e realizar o processo de *logging*.

A versão V3-B faz parte de duas regiões no mapa de versões e foi projetada para avaliar o processo de *weaving* de apenas 2 classes, que são "br.ufu.sige.business.SigeServiceImpl" e "br.ufu.sige.dao.SigeDAOImplExp", cuja soma dos tamanhos é de 153.3 KB e contendo apenas 1 invocação para o cenário proposto. A partir da versão V3-B, duas outras versões foram geradas, que são as versões V3-C e V3-D, onde a variável *loopCount* foi sobrescrita respectivamente para os valores 100 e 10.000.

Versões com variação no número de classes carregadas

O propósito dessa fase é avaliar o impacto no desempenho quando o número de classes carregadas durante o processo de *weaving* é variado. Para isso, a partir da versão V3-B, outras duas versões foram geradas, que são V3-E e V3-F. Na versão V3-E, o arquivo "aop.xml" foi sobrescrito. Na primeira seção desse arquivo foram incluidas as subclasses do pacote "br.ufu.sige" como é mostrado na Figura 4.15. Com essa configuração, o processo de *weaving* atua sobre todas as classes dos subpacotes de "br.ufu.sige". Da mesma forma, na classe *AspectProfiler*, um *advice after* com uma expressão de *pointcut* que inclui essas classes foi configurada, como é mostrado na Figura 4.16.

```
1  <!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
2  <aspectj>
3
4     <weaver>
5
6        <include within="br.ufu.sige..*"/>
7
8     </weaver>
9
10    <aspects>
11
12       <aspect name="br.ufu.sige.aspect.AspectProfiler"/>
13
14    </aspects>
15
16 </aspectj>
17
```

Figura 4.15: Conteúdo do arquivo "aop.xml", representando o carregamento de todas as classes do projeto que estão dentro do subpacote "br.ufu.sige" no processo de *load-time weaving*.

```
@After("execution(* br.ufu.sige.business.*.*(..))")
public void adviceAfterBusiness(JoinPoint joinPoint) throws Throwable {
}

@After("execution(* br.ufu.sige..*(..))")
public void adviceAfterSige(JoinPoint joinPoint) throws Throwable {
}
```

Figura 4.16: Fragmento de código expressão de *pointcut* que intercepta todos os métodos de todas as subclasses do pacote "br.ufu.sige".

O número total de classes nas quais atua o processo de *weaving* na versão V3-E é de 74 e a soma de seus tamanhos é de 638.7 KB. A ferramenta *CyVis* foi utilizada para calcular

o número de classes carregadas do pacote "br.ufu.sige", mas a soma de seus tamanhos foi calculada por meio do programa *Nautilus* do *Ubuntu*, selecionando-se tais classes na pasta *target* do projeto.

Na versão V3-F, o arquivo "aop.xml" foi sobrescrito, onde incluiu-se na primeira seção todos os diferentes pacotes das classes do projeto, como mostrado na Figura 4.17. A ferramenta *CyVis* também foi utilizada para identificar esses pacotes.

```
1  <!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
2  <aspectj>
3
4     <weaver>
5
6          <include within="br..*"/>
7          <include within="org..*"/>
8          <include within="com..*"/>
9          <include within="net..*"/>
10         <include within="sqlj..*"/>
11         <include within="freemarker..*"/>
12         <include within="javax..*"/>
13         <include within="javaassist..*"/>
14         <include within="antlr..*"/>
15         <include within="net..*"/>
16         <include within="ognl..*"/>
17         <include within="edu..*"/>
18         <include within="junit..*"/>
19         <include within="COM..*"/>
20
21      </weaver>
22
23      <aspects>
24
25          <aspect name="br.ufu.sige.aspect.AspectProfiler"/>
26
27     </aspects>
28
29    </aspectj>
```

Figura 4.17: Conteúdo do arquivo "aop.xml", representando o carregamento de todas as classes do projeto, de todos os pacotes identificados, no processo de *load-time weaving*.

.

Nessa configuração, o processo de *weaving* atua sobre todas as classes que estão dentro desses pacotes e seus subpacotes. Da mesma forma, na classe *AspectProfiler*, um *advice after* com uma expressão de *pointcut* que inclui essas classes foi configurada, como é mostrado na Figura 4.18.

```
@After("execution(* br.ufu.sige.business.*.*(..))")
public void adviceAfterBusiness(JoinPoint joinPoint) throws Throwable {
}

@After("execution(* *(..))")
public void adviceAfterAll(JoinPoint joinPoint) throws Throwable {
}
```

Figura 4.18: Fragmento de código contendo expressão de *pointcut* que intercepta todos os métodos de todas as classes do projeto.

.

O número total de classes nas quais atua o processo de *weaving* na versão V3-F é de 19.458 e a soma de seus tamanhos é 59.3 MB. A soma do número de classes e os tamanhos foram calculados da mesma maneira que na versão V3-E.

4.7.3 Versões da Região C do Mapa de versões

O propósito dessa fase é avaliar se a variação do LOC das aplicações causa impacto no desempenho. Para isso, a partir da versão V4-A, a versão V6 foi gerada. Não houve mudança de parâmetros na nova versão, porém na versão V6, apenas os arquivos relativos ao teste foram mantidos. Esses arquivos incluem os mesmos arquivos de configuração e somente as classes das quais dependem a execução do cenário, que são no total 21. A versão V6 também teve o número de bibliotecas reduzido para o mínimo necessário para a compilação do projeto e execução dos testes. O tamanho da versão V4-A é de 283.340 LOC e na versão V6 de 4.012 LOC, onde foram considerados arquivos dos tipos: java, js, jsp, html, xml e css. A contagem do LOC dessas duas versões foi feita usando a aplicação *LOC counter*.

4.7.4 Versões da região D do Mapa de versões

Essa fase compreende a construção de 12 versões da aplicação com os seguintes propósitos:

1. Avaliar o impacto no desempenho quando o tipo de *advice* é variado.
2. Avaliar o impacto no desempenho quando o número de *join points* é variado.

A partir da versão V4-A foi gerada a versão V5-A, onde foram sobrescritos os seguintes parâmetros:

- Número de *advices*: 8
- Tipo de *advices*: 8 *around*, tendo a geração das mensagens de *log* em apenas 2.

Para o aumento no número de *join points*, que nesse caso sempre representa a execução de método, foram criados novos *advices* por meio de cópias dos *advices* da classe *AspectProfiler*, alterando-se apenas o nome dos métodos a fim de permitir a compilação da classe.

A partir da versão V5-A, as versões V5-B e V5-C foram geradas, sobrescrevendo-se o número de *advices* para 16 e 32 respectivamente. Da mesma forma, o tipo de *advice* manteve-se e a geração do *log* das mensagens se mantiveram em apenas 2 dos métodos.

As versões V5-D, V5-G e V5-J foram geradas a partir da versão V5-A e o tipo de *advice* sobrescrito para *before*, *after* e *afterThrowing* respectivamente.

As versões V5-E, V5-H e V5-K foram geradas a partir da versão V5-B e o tipo de *advice* sobrescrito para *before*, *after* e *afterThrowing* respectivamente.

As versões V5-F, V5-I e V5-L foram geradas a partir da versão V5-C e o tipo de *advice* sobrescrito para *before*, *after* e *afterThrowing* respectivamente.

A geração do *advice afterThrowing* envolve o lançamento de exceção. As exceções lançadas foram todas do tipo *RuntimeException* sem parâmetros, as quais tiveram tratamento sem comportamento com blocos *try* e *catch* nos métodos chamadores a fim de permitir a continuação das execuções de forma ininterrupta.

4.7.5 Versões da região E do Mapa de Versões

O propósito dessa fase é avaliar POA juntamente com duas camadas da aplicação que causam impacto significativo no desempenho, que são a cadeia de mecanismos do *framework Struts* e o acesso ao banco de dados DB2.

Para isso, a versão V1-B foi gerada a partir da versão V1-A. O banco de dados foi habilitado comentando-se as linhas de código na classe "br.ufu.sige.dao.SigeDAOImplExp", que retornam com o mínimo de processamento, presentes no início de cada método do cenário.

Um teste foi realizado para avaliar o tempo gasto em cada invocação externa da versão V1-B. O plano do experimento do *JMeter* foi configurado para gerar 1 requisição HTTP para a URL que inicia o cenário e foi executado 10 vezes sobre a versão V1-B, implantada no servidor *Tomcat* local por meio do *Eclipse*. O relatório fornecido pelo *JMeter* apontou uma variação de valores de tempo gasto com as requisições entre 6.000 e 7.000 ms. Tendo em vista esses altos valores de tempo gastos em cada teste quando comparados com os valores gastos nos testes das invocações internas a variável *loopCount* do plano de experimento do *JMeter* foi sobrescrita para 50 a fim de possibilitar a avaliação das variáveis dependentes em um tempo razoável.

A partir da versão V1-B, as versões V2-B, V4-B e V3-G foram geradas, sobrescrevendo-se os *weavers*-processos de *weaving* para *AspectJ-compile-time weaving*, *Spring AOP-runtime weaving* e *AspectJ-load-time weaving* de forma semelhante como na geração das versões V2-A, V4-A e V3-A.

Capítulo 5

Resultados

Neste capítulo são apresentados os resultados obtidos nas medições das 3 variáveis dependentes propostas, após a execução do plano do experimento para cada versão gerada. As Figuras contendo os resultados apresentam Tabelas nas quais valores considerados *outliers* são destacados com a cor cinza de fundo. As médias dos valores de cada variável também são apresentadas com destaque. O cálculo para a marcação dos outliers bem como o critério para a geração das médias é apresentado em detalhes no Capítulo 6.

5.1 Resultados obtidos para as versões da região A do mapa de versões

Os resultados obtidos na execução do plano do experimento para as versões V1-A, V2-A, V3-A e V4-A são apresentados, respectivamente, nas Figuras 5.1, 5.2, 5.3 e 5.4.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	11716	12155	12440	12565	12578	12619	12742	12786	12807	12808	12849	12943	12957	12971	13113	12738.07
CPU	31.2	31.9	32.4	32.4	32.9	32.9	34.9	35.1	35.3	35.4	36.4	36.8	37.7	37.9	39.1	34.82
Mem.	54.8	55.1	55.3	56.5	57.9	62.1	63.7	64.3	66.1	68.3	87.6	90.6	116.4	223.7	285	69.13

Figura 5.1: Resultados obtidos para a versão V1-A.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	12005	12017	12617	12842	12953	13015	13016	13188	13221	13221	13283	13342	13383	13566	13597	13172.62
CPU	30.9	27.3	31.2	31.2	31.4	31.7	31.7	31.7	32.6	32.9	32.9	33.1	35.4	35.9	35.9	31.84
Mem.	58.7	61.4	62.6	63.4	68.6	69.5	71.8	72.8	73.2	74.5	74.5	79.1	79.3	89.8	134.3	71.37

Figura 5.2: Resultados obtidos para a versão V2-A.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	12607	13382	13445	13455	13467	13486	13593	13643	13808	13815	13845	13868	13903	14020	15317	13671.54
CPU	28.1	30.7	31.6	31.6	31.7	31.9	32.6	34.3	34.6	34.7	34.9	35.3	35.4	35.4	35.7	33.23
Mem.	76.2	84	84.7	84.8	91.4	92.2	92.2	94	94.2	95.3	96.9	98.3	113.9	124	131.1	90.35

Figura 5.3: Resultados obtidos para a versão V3-A.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	17179	17307	17386	17408	17579	17860	17892	18123	18166	18374	18391	18503	18503	18759	18761	18012.73
CPU	31.6	32.3	32.4	32.9	33.1	33.2	33.4	33.6	34.1	34.1	34.6	35.1	35.6	36.4	36.5	33.92
Mem.	70.5	74.9	87.7	90.2	90.6	92.8	97	97.6	101	101.8	102.8	103.5	107.4	110.4	128.7	96.74

Figura 5.4: Resultados obtidos para a versão V4-A.

5.2 Resultados obtidos para as versões da região B do mapa de versões

Os resultados obtidos na execução do plano do experimento para as versões V3-B, V3-C, V3-D, V3-E e V3-F são apresentados, respectivamente, nas Figuras 5.5, 5.6, 5.7, 5.8 e 5.9.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	4092	4186	4210	4213	4287	4371	4377	4413	4447	4553	4585	4621	4622	4652	4762	4426.06
CPU	72.4	72.9	73.2	73.3	73.4	73.8	74.1	74.5	74.7	74.7	75.4	75.8	77.3	78.2	78.6	74.82
Mem.	56.8	57.5	58	59	63.5	63.6	68	68	68.2	68.3	68.3	71	71.4	72.8	79.5	66.26

Figura 5.5: Resultados obtidos para a versão V3-B.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	4621	4749	4752	4776	4802	4835	4850	4900	4906	4913	4915	4939	4954	4958	5050	4861.33
CPU	73.7	74.1	74.1	74.3	74.3	74.4	74.5	74.6	74.6	74.8	74.8	75	75.1	75.6	75.6	74.63
Mem.	77	77.2	78.6	79.1	79.2	79.9	80.8	80.9	82.1	85.5	96.1	117.8	119.4	119.7	121.6	91.66

Figura 5.6: Resultados obtidos para a versão V3-C.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	17421	17509	17667	17767	17930	17939	18093	18177	18332	18428	18658	18825	18919	18954	19170	18252.60
CPU	73.1	73.6	73.6	73.6	73.6	73.7	73.7	73.9	74.1	74.2	74.3	74.4	74.4	74.8	75.1	74.00
Mem.	148.5	154	163	178.8	182.8	198.3	206.6	226.3	227.3	237.1	237.6	247.5	254	278.2	289.6	215.30

Figura 5.7: Resultados obtidos para a versão V3-D.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	5143	5159	5188	5231	5278	5300	5461	5468	5561	5561	5605	5666	5751	5754	5913	5469.26
CPU	73.3	74.1	74.3	74.3	74.6	74.6	74.6	74.7	74.8	75.1	75.3	77.7	77.9	78.3	78.6	75.48
Mem.	66.9	67.9	68.6	68.7	68.7	69.5	70	70.1	70.2	71.3	77.8	105.7	106.3	107.8	107.9	79.82

Figura 5.8: Resultados obtidos para a versão V3-E.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	7044	7174	7179	7300	7303	7308	7330	7351	7404	7543	7722	7880	7974	8435	8534	7565.40
CPU	73.3	74.1	74.3	74.4	74.7	74.8	74.8	74.9	75	75.1	76	76.2	77.4	77.5	77.7	75.34
Mem.	80.7	81.5	81.6	83.5	84.1	86.4	90.8	98.9	105.2	113.6	114	114.4	114.9	118.5	141	100.60

Figura 5.9: Resultados obtidos para a versão V3-F.

5.3 Resultados obtidos para as versões da região C do mapa de versões

Os resultados obtidos na execução do plano do experimento para a versão V4-A são apresentados na Seção 5.1. Os resultados obtidos na execução do plano do experimento para a versão V6 são apresentados na Figura 5.10.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	17432	17595	17635	18543	18635	18676	18738	18772	18806	19269	19304	19511	19586	19593	20079	18811.60
CPU	29.6	32.9	33.6	33.8	33.9	33.9	34.2	34.3	34.4	34.6	34.6	35	35.2	35.6	35.6	34.08
Mem.	69.8	73	73.2	73.4	74.2	75.6	76.6	79.3	79.6	81.6	85.1	88	94.6	97.4	99.7	81.40

Figura 5.10: Resultados obtidos para a versão V6.

5.4 Resultados obtidos para as versões da região D do mapa de versões

Os resultados obtidos na execução do plano do experimento para as versões V5-A, V5-D, V5-G, V3-J, V5-B, V5-E, V5-H, V5-K, V5-C, V5-F, V5-I e V5-L são apresentados, respectivamente, nas Figuras 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19, 5.20, 5.21 e 5.22. A disposição dos resultados foi agrupada de acordo com o número de *join points* que são respectivamente 8, 16 e 32.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	20093	20259	20339	20543	20617	20695	20696	20724	20815	21141	21317	21415	21512	21619	22206	20932.73
CPU	33.4	34.3	34.4	34.6	35.2	35.3	35.4	35.9	35.9	35.9	36.1	36.1	36.4	36.4	36.6	35.46
Mem.	71.3	76.8	81.6	90	93.2	95.2	96.3	97.3	97.5	98.1	98.3	99.2	99.6	100.7	242.4	95.58

Figura 5.11: Resultados obtidos para a versão V5-A.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	17729	17480	17785	17824	17901	17937	17949	18315	18335	18500	18917	19070	19326	19689	19969	18448.40
CPU	35.6	36.4	36.6	36.6	36.8	37.4	37.4	37.4	38.1	38.2	38.4	39.4	40.1	41.1	41.6	38.07
Mem.	85.4	89.1	89.6	92.5	94.7	98.9	102	103.4	104.5	104.5	104.7	106	106.6	110.4	118.8	100.74

Figura 5.12: Resultados obtidos para a versão V5-D.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	17270	17600	18074	18076	18474	18637	18681	18742	18864	19324	19611	19663	19790	19971	20901	18911.86
CPU	33.9	34.3	35.4	35.4	36.7	37.8	38.4	38.6	38.7	39.1	40.1	41.7	42.1	42.1	42.4	38.44
Mem.	79	89	92.1	94.5	96.6	98.4	100.9	101.9	102.3	102.7	105.6	108.6	114.7	116.6	119.1	101.46

Figura 5.13: Resultados obtidos para a versão V5-G.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	34505	34101	34238	34478	34593	34636	34720	34733	34764	34971	35098	35229	35374	35483	35736	34843.93
CPU	38.9	42.6	45.3	45.9	45.9	46.1	48.1	48.8	54.8	55.1	57	57.6	58.3	58.3	62.2	50.99
Mem.	64.4	70	83.3	86.8	92.8	96.9	100.5	104.6	106.4	110.4	115	116.9	124.7	129.2	182	100.13

Figura 5.14: Resultados obtidos para a versão V5-J.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	24433	24618	24758	24832	25013	25280	25423	25622	26340	26909	27147	27308	27519	28051	29967	26214.67
CPU	31.5	31.7	31.9	33.4	33.6	33.9	34	34	34.1	34.1	34.3	34.6	34.6	34.9	35.7	34.26
Mem.	116.1	116.9	128.2	136.7	141.6	152.3	152.9	184.8	196.6	207.6	208.1	211.7	236.6	242.6	282.7	181.02

Figura 5.15: Resultados obtidos para a versão V5-B.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	18578	18703	18750	18885	19039	19189	19225	19415	19524	19629	19701	19757	20128	20727	20875	19475.00
CPU	28.4	31.6	32	32.8	33.1	33.4	34.4	34.6	35.1	35.6	35.6	35.6	36.3	37.4	37.6	34.65
Mem.	117.1	119.3	120.6	127.1	146.9	152	165.4	180.3	181.3	189.5	201.2	201.5	229.5	257	259.7	176.56

Figura 5.16: Resultados obtidos para a versão V5-E.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	20532	20558	20652	20739	20881	21138	21154	21263	21295	21309	21353	21441	21473	21677	21758	21148.20
CPU	31.9	32.2	33.2	33.4	34.6	34.7	34.9	34.9	35.4	35.8	36.1	36.4	36.8	37	37.9	35.01
Mem.	110.8	120.2	146.5	158.2	162.6	163.8	164.5	165.4	166.5	183	185	206.9	221.8	223.5	288.7	169.90

Figura 5.17: Resultados obtidos para a versão V5-H.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	36127	40621	41180	41637	41755	41837	41883	42015	42036	42163	42261	42319	42334	42566	43180	42089.69
CPU	38.6	40.6	41.4	42	42.4	42.4	42.9	43.1	43.2	43.4	43.6	44.2	44.6	46.6	46.9	42.81
Mem.	141.9	144.8	145.5	153.4	155.3	179	179.4	188	188.9	190.5	193.8	196	205.4	211.2	319.6	176.65

Figura 5.18: Resultados obtidos para a versão V5-K.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	30313	30334	30446	30708	30818	31853	32691	32968	32986	33087	33791	34593	34887	35091	36146	32714.13
CPU	28.6	29.9	29.9	30.4	30.8	31.1	31.2	31.4	31.7	31.9	32.2	32.4	32.8	33.1	34.4	31.45
Mem.	181.3	185.2	203.7	205.2	217.1	221.2	225.1	237.6	247.8	253	253.2	277.1	297.7	322	327.2	243.62

Figura 5.19: Resultados obtidos para a versão V5-C.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	21468	22029	22191	22321	22512	22731	22993	24818	24938	24991	25080	25141	25173	26247	26265	23926.53
CPU	31.1	31.6	31.8	33.1	33.4	33.4	33.4	33.4	34.1	34.3	34.4	35.1	35.9	35.9	37.3	33.63
Mem.	193	194.2	203.9	204.2	215.3	231.6	236.5	248.8	266.2	269	273.2	293.7	295.5	297.4	308.7	248.74

Figura 5.20: Resultados obtidos para a versão V5-F.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	24056	24160	24176	24433	25262	25729	26019	26298	26484	26519	26649	26776	27143	27271	28901	25991.73
CPU	31.2	32.1	32.6	32.6	32.6	32.7	33	33.6	34.2	34.2	34.9	35.2	36.6	36.7	37.1	33.95
Mem.	119.5	210.8	227.3	237.6	241.6	249	252.5	253.9	256.8	269.6	274.3	296.2	311.4	316.3	321.8	265.65

Figura 5.21: Resultados obtidos para a versão V5-I.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	54183	54278	54489	54821	55181	56648	56745	56760	56805	57286	57288	57297	57650	57689	58084	56346.93
CPU	35.4	36.6	36.6	38.4	40.2	41.3	41.9	42.1	42.8	43.1	43.6	43.9	44.1	45	45.4	41.36
Mem.	191.6	220.7	246.9	264.6	289.9	294.5	295.4	297.5	299.3	305.1	305.3	308.6	318	327.2	328.1	298.49

Figura 5.22: Resultados obtidos para a versão V5-L.

5.5 Resultados obtidos para as versões da região E do mapa de versões

Os resultados obtidos na execução do plano do experimento para as versões V1-B, V2-B, V4-B e V3-G são apresentados, respectivamente, nas Figuras 5.23, 5.24, 5.25 e 5.26.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	6256	6263	6263	6268	6273	6275	6287	6290	6291	6292	6302	6304	6314	6351	6354	6282.92
CPU	7.3	7.4	7.4	7.4	7.4	7.4	7.7	7.7	7.7	7.8	7.9	7.9	8.4	8.9	13.7	7.64
Mem.	65.2	69.5	69.8	69.9	70.7	70.9	71.1	72.8	73.3	80.3	86.2	116.3	164	212.8	266.1	76.33

Figura 5.23: Resultados obtidos para a versão V1-B.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	6253	6255	6258	6258	6261	6263	6265	6283	6284	6289	6297	6301	6306	6332	6367	6278.92
CPU	7.4	7.4	7.6	7.6	7.7	7.7	7.9	7.9	8.1	8.4	8.7	8.9	10.4	17.4	19.4	8.13
Mem.	64	67.6	69.3	69.7	70.9	71.3	72	73.8	82.2	85.7	115.2	162.8	217.5	269.4	301.3	94.00

Figura 5.24: Resultados obtidos para a versão V2-B.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	6214	6222	6223	6223	6226	6226	6234	6244	6246	6252	6256	6288	6298	6305	6330	6252.46
CPU	7.4	7.4	7.7	7.9	7.9	7.9	8.2	8.4	8.7	11.2	11.2	12.2	12.8	13.2	30.4	9.43
Mem.	68.7	70.5	72.1	72.7	73.7	73.9	75.1	75.8	76.3	76.7	99.7	146	203.9	260.8	299.5	81.76

Figura 5.25: Resultados obtidos para a versão V4-B.

Métrica	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Média
Tempo	6193	6230	6239	6241	6242	6252	6260	6265	6268	6272	6278	6283	6305	6320	6327	6265.00
CPU	7.2	7.4	7.4	7.4	7.6	7.7	7.7	7.7	7.9	7.9	9.9	11.2	12.4	12.9	16.7	8.73
Mem.	171.5	175.7	177.6	178.9	179.9	182.1	185.1	185.9	266.7	291.8	294.3	301.4	301.5	306.4	311	233.98

Figura 5.26: Resultados obtidos para a versão V3-G.

Capítulo 6

Análise dos Resultados

O experimento produziu no total 1200 resultados de medições, onde foram avaliados consumos de tempo, CPU e memória das invocações. Dos 1200 resultados, 20 foram gerados com o propósito de avaliar os métodos de medições propostos para direcionar suas utilizações nas etapas do experimento, 10 para avaliar o valor adequado da variável *loopCount* nos testes das versões da região E do mapa de versões e os outros 1170 foram gerados nos testes das 26 versões geradas.

Esse capítulo foi dividido em 2 seções. Na Seção 6.1 são apresentados gráficos representando os resultados das 26 versões da aplicação, agrupadas por regiões do mapa de versões a fim de evidenciar as diferenças nas medições para o fator que está sendo analisado. Na Seção 6.2 é apresentado um processo de identificação dos *outliers*, fazendo-se uma comparação dos resultados sem a presença dos *outliers* com os resultados contendo os *outliers*. Na Seção 6.3 é apresentada uma discussão a respeito dos resultados para cada fator de POA proposto nesse trabalho.

Todos os gráficos dessa seção foram gerados a partir dos resultados contendo os *outliers* devido a 2 razões. A primeira delas é que a apresentação de resultados com diferentes amostras, decorridas da eliminação de *outliers*, não seria adequada para alguns gráficos. A segunda é enfatizar a presença dos *outliers* em algumas situações e discutir a respeito da presença dos mesmos.

6.1 Análise dos Resultados Puros

Nessa seção são apresentados os resultados em gráficos das medições geradas para as 26 versões da aplicação, agrupados por região do mapa de versões, representando os fatores de interesse propostos, conforme mostrados a seguir.

6.1.1 Fator *weaver*

O fator *weaver* foi variado nas regiões A e E do mapa de versões. Nas versões da região A, as medições foram geradas a partir do uso das invocações internas e os resultados para a variável dependente "tempo" são valores absolutos, representando o tempo gasto para executar cada plano do experimento. Nas versões da região E, os valores das medições para a variável "tempo" são gerados a partir das invocações externas, e fornecidas pelo relatório do *JMeter* onde essa variável representa a média de tempo das invocações, após a execução de cada plano do experimento.

Os valores das medições são apresentados em gráficos do tipo *box plot* e agrupados por variável dependente. Os gráficos gerados a partir das variáveis dependentes das versões das regiões A e E do mapa de versões são dispostos a seguir.

Versões da região A do Mapa de Versões

Nas Figuras 6.1, 6.2 e 6.3 são apresentados gráficos contendo, respectivamente, os consumos de tempo, CPU e memória das versões V1-A, V2-A, V3-A e V4-A.

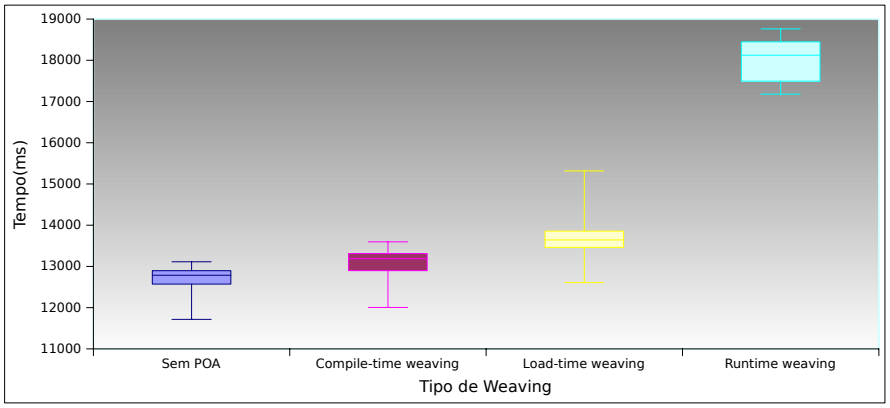

Figura 6.1: Consumo de tempo para as versões V1-A, V2-A, V3-A e V4-A.

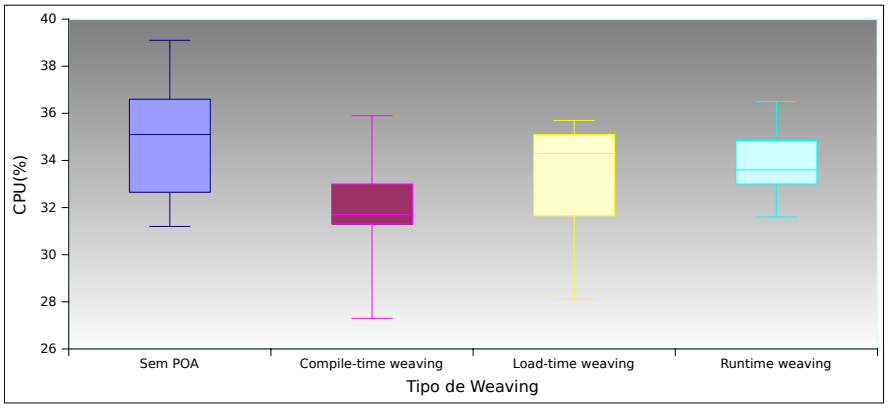

Figura 6.2: Consumo de CPU para as versões V1-A, V2-A, V3-A e V4-A.

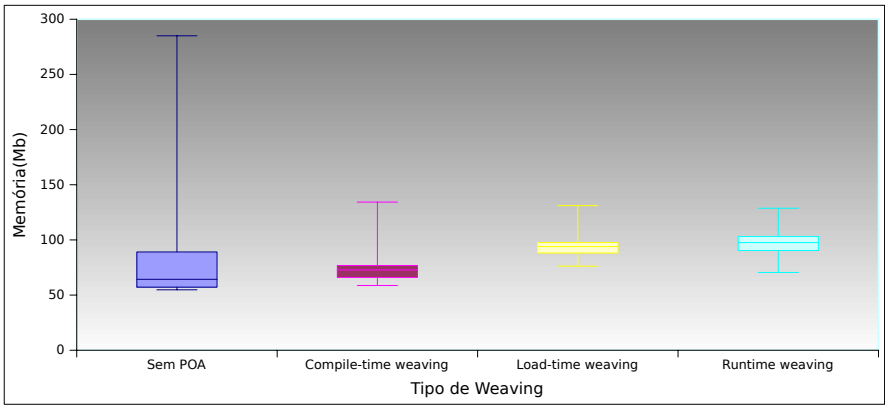

Figura 6.3: Consumo de memória para as versões V1-A, V2-A, V3-A e V4-A.

Versões da região E do Mapa de Versões

Nas Figuras 6.4, 6.5 e 6.6 são apresentados gráficos contendo, respectivamente, os consumos de tempo, CPU e memória das versões V1-B, V2-B, V4-A e V3-G.

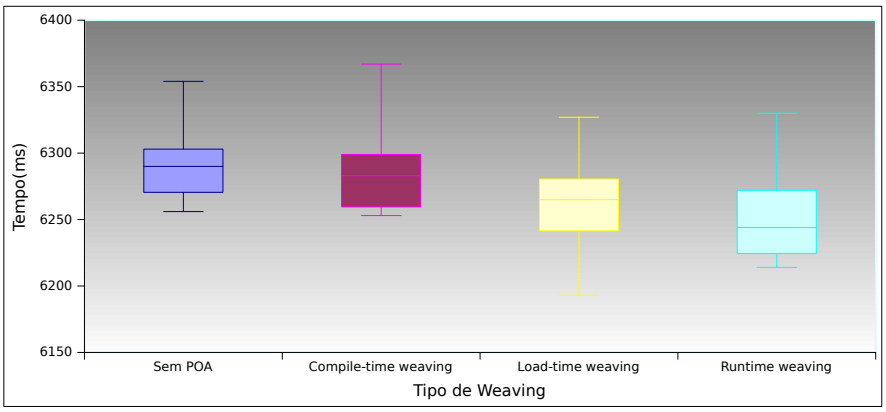

Figura 6.4: Consumo de tempo para as versões V1-B, V2-B, V4-A e V3-G.

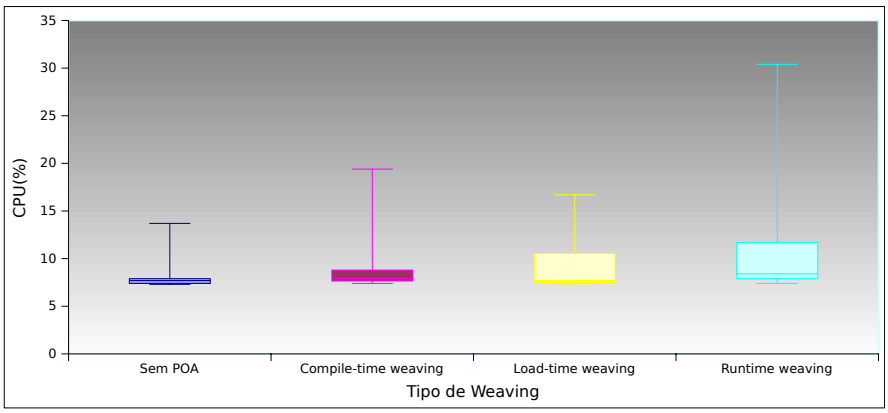

Figura 6.5: Consumo de CPU para as versões V1-B, V2-B, V4-A e V3-G.

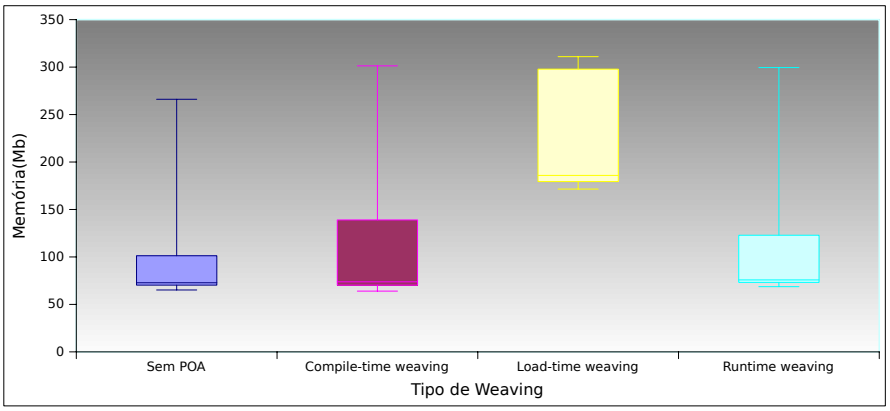

Figura 6.6: Consumo de memória para as versões V1-B, V2-B, V4-A e V3-G.

6.1.2 Fatores Tipo de *Advice* e Número de *Join Points*

Os fatores tipo de *advice* e número de *join points* foram avaliados nas 12 versões presentes na região D do mapa de versões, onde tais fatores foram variados. Nas Figuras 6.7, 6.8 e 6.9 são apresentados gráficos contendo, respectivamente, os consumos de tempo, CPU e memória dessas versões.

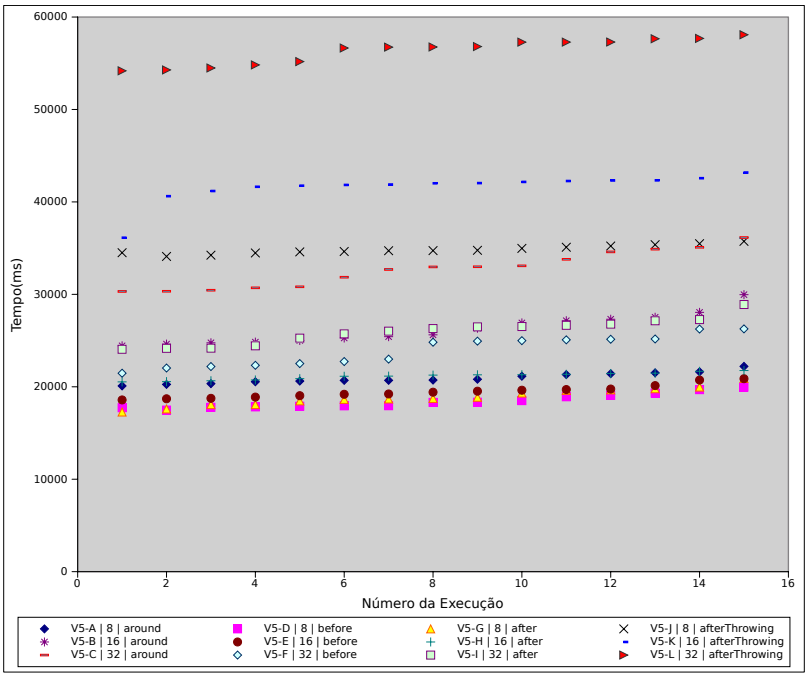

Figura 6.7: Consumo de tempo para as 12 versões da região D do mapa de versões, considerando os diferentes tipos de *advice* e números de *join points*.

6.1.3 Fator LOC

Duas versões foram avaliadas considerando o fator LOC, que são as versões V4 e V6, presentes na região C do mapa de versões. Nas Figuras 6.10, 6.11 e 6.12 são apresentados gráficos contendo, respectivamente, os consumos de tempo, CPU e memória dessas versões.

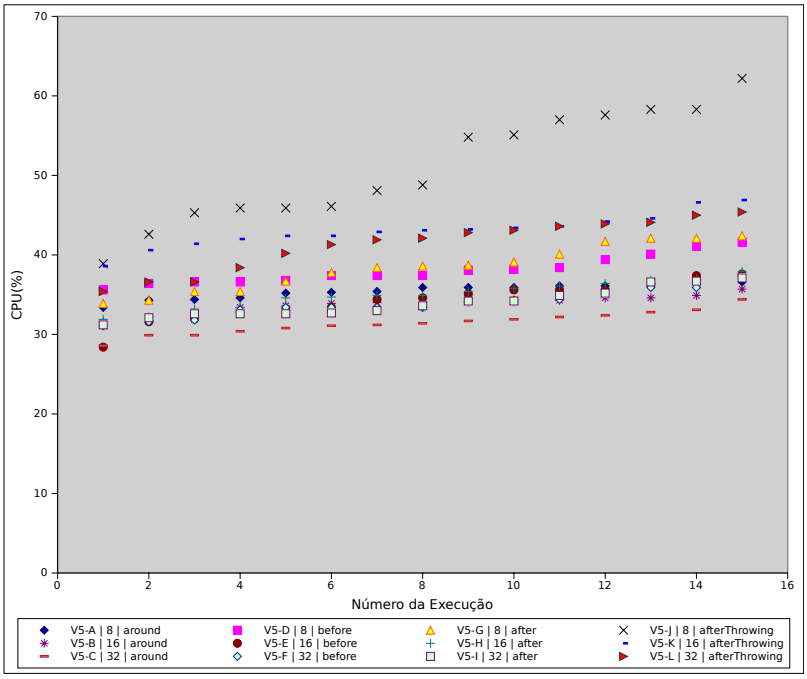

Figura 6.8: Consumo de CPU para as 12 versões da região D do mapa de versões, considerando os diferentes tipos de *advice* e números de *join points*.

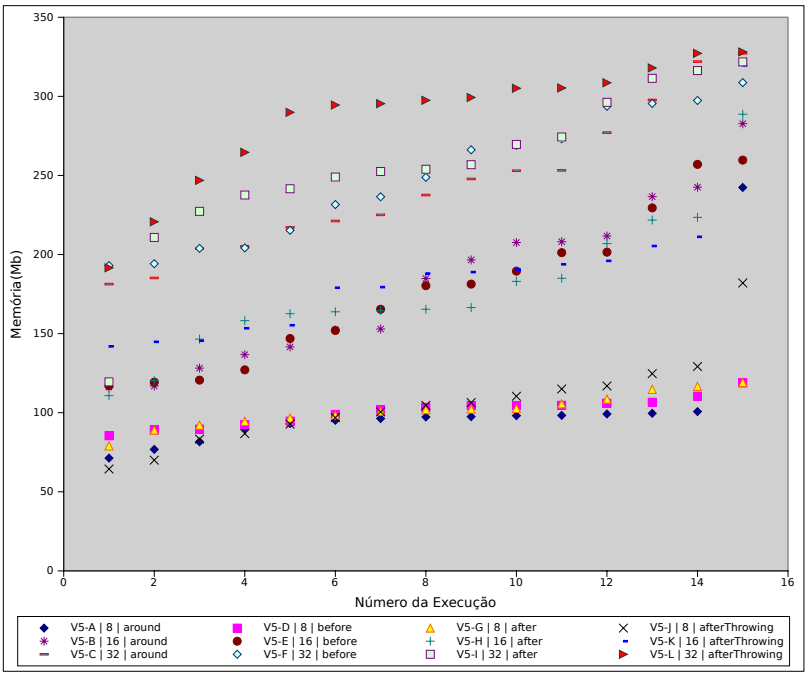

Figura 6.9: Consumo de memória para as 12 versões da região D do mapa de versões, considerando os diferentes tipos de *advice* e números de *join points*.

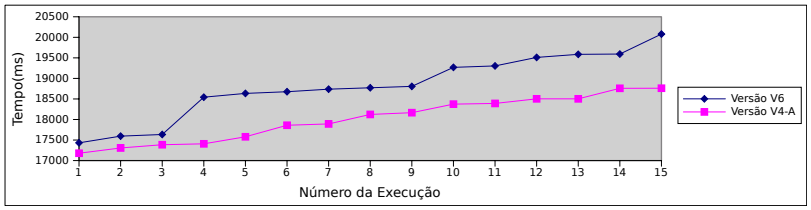

Figura 6.10: Consumo de tempo para as versões V4-A e V6, da região C do mapa de versões, considerando os diferentes LOCs

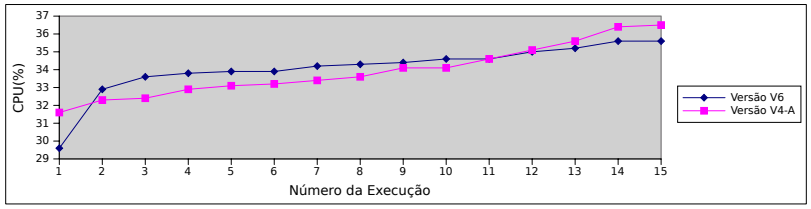

Figura 6.11: Consumo de CPU para as versões V4-A e V6, da região C do mapa de versões, considerando os diferentes LOCs

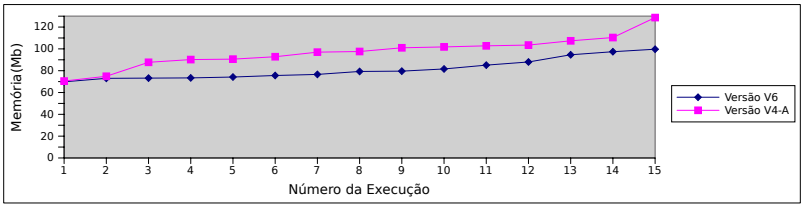

Figura 6.12: Consumo de memória para as versões V4-A e V6, da região C do mapa de versões, considerando os diferentes LOCs

6.1.4 Fatores específicos do processo de *Load-time weaving*

As 5 versões presentes na região B do mapa de versões foram avaliadas sob 2 aspectos: variação do número de invocações e variação do número de classes carregadas durante o processo de *load-time weaving*. Nas Figuras 6.13, 6.14 e 6.15 são apresentados gráficos contendo, respectivamente, os consumos de tempo, CPU e memória das versões V3-B, V3-C e V3-D, onde são representados os comportamentos das variáveis dependentes quando o número de invocações, representado pela variável *loopCount*, é variado.

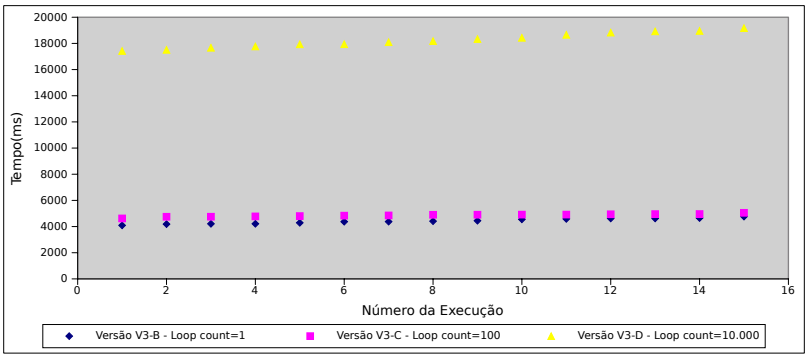

Figura 6.13: Consumo de tempo para as versões V3-B, V3-C e V3-D, da região B do mapa de versões, considerando a variação do número de invocações.

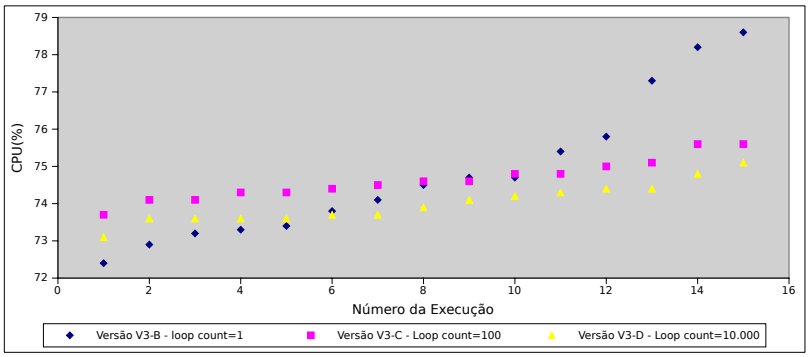

Figura 6.14: Consumo de CPU para as versões V3-B, V3-C e V3-D, da região B do mapa de versões, considerando a variação do número de invocações.

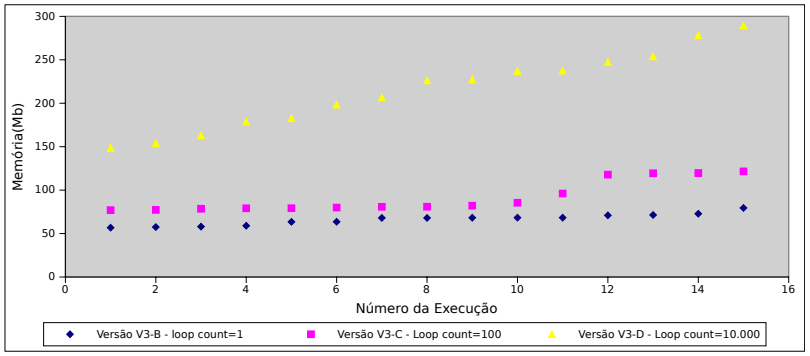

Figura 6.15: Consumo de memória para as versões V3-B, V3-C e V3-D, da região B do mapa de versões, considerando a variação do número de invocações.

Nas Figuras 6.16, 6.17 e 6.18 são apresentados gráficos contendo os consumos das variáveis dependentes das versões V3-B, V3-E e V3-F quando o número de classes carregadas durante o processo de *load-time weaving* é variado.

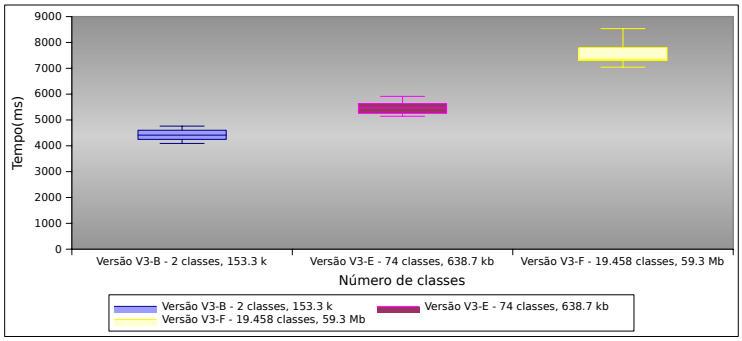

Figura 6.16: Consumo de tempo para as versões V3-B e V3-E e V3-F, da região B do mapa de versões, considerando número de classes carregadas durante o processo de *load-time weaving*.

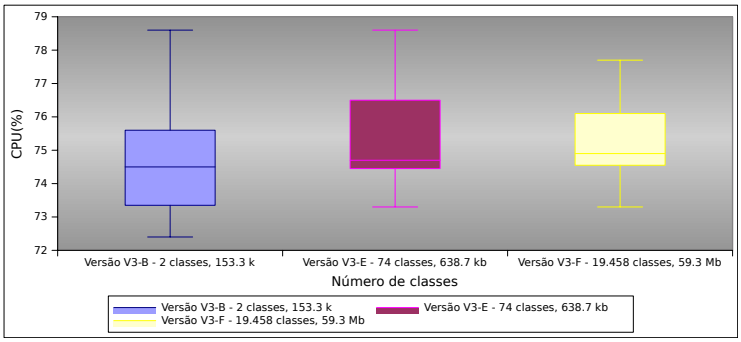

Figura 6.17: Consumo de CPU para as versões V3-B e V3-E e V3-F, da região B do mapa de versões, considerando número de classes carregadas durante o processo de *load-time weaving*.

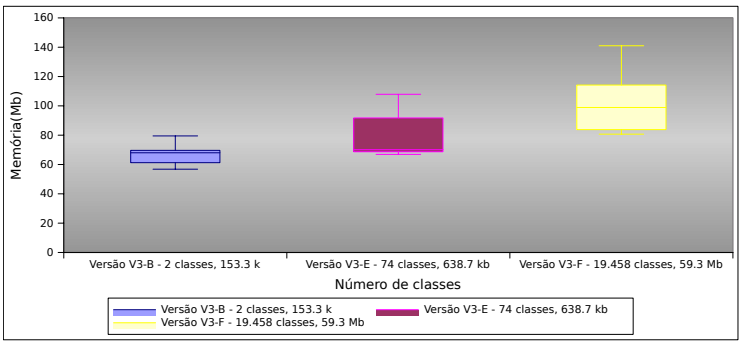

Figura 6.18: Consumo de memória para as versões V3-B e V3-E e V3-F, da região B do mapa de versões, considerando número de classes carregadas durante o processo de *load-time weaving*.

6.2 Análise dos Resultados Refinados

De forma a obter embasamentos matemáticos para confirmar ou rejeitar as hipóteses mencionadas na Seção 1.3 do Capítulo 1, um método de identificação de *outliers* foi utilizado em todos os resultados para marcar os *outliers* e gerar médias de valores para cada variável dependente desconsiderando os *outliers*. O método *Plot Box* foi utilizado para marcar os *outliers* uma vez que a demonstração de resultados fez grande uso dos gráficos de *Plot Box*.

Para marcar os *outliers*, cada uma das séries de 15 amostras foi ordenada para cada variável dependente, para os resultados de todas as versões. Em seguida, as seguintes variáveis foram calculadas para cada uma das séries:

- Média (M).
- Primeiro Quartil (Q1).
- Terceiro Quartil (Q3).
- Diferença entre quartis (DQ), onde DQ = Q3 - Q1.
- Limite aceitável (LA), onde LA = 1,5 x DQ.

- Limite inferior (LI), onde LI = Q1 - LA.
- Limite superior (LS), onde LS = Q3 + LA.

Para cada série de valores, todo valor menor que LI ou maior que LS foi marcado como *outlier*. No total, 52 valores foram marcados, dos quais a variável dependente tempo obteve 10 marcações, CPU obteve 17 marcações e memória obteve 25 marcações. Considerando que 1170 amostras foram extraídas das versões, as 52 marcações correspondem a 4,44% do total das amostras, de acordo com o método de marcação de *outliers* escolhido.

A importância de se ter múltiplas formas de se medir o desempenho se deve ao fato que, apesar do número de *outliers* ter sido de apenas 4,44%, apenas a variável dependente "memória" obteve 48,97% de *outliers*, enquanto as variáveis "CPU" e "tempo" corresponderam a respectivamente 32,69% e 19,23% dos *outliers*. Essas proporções demonstram uma tendência onde o tempo é considerada a variável de maior confiabilidade nas medições enquanto a memória é considerada a menos confiável, considerando-se as 3 variáveis propostas. Na Tabela 6.1 são apresentadas as versões do experimento e o número de *outliers* identificados para cada variável dependente.

Com base nos valores não marcados, foi calculada a média para cada série de valores e para cada variável dependente. Na Tabela 6.2 são apresentadas essas médias para todas as versões geradas.

A partir das médias calculadas para as variáveis dependentes de cada versão, comparações foram geradas com o propósito de avaliar os fatores de POA propostos. Nas subseções a seguir são apresentadas essas comparações. Para cada conjunto de versões sendo comparadas, uma das versões foi selecionada como base de comparação. O critério utilizado para seleção de cada uma dessas versões foi:

- Para versões das regiões A e E do mapa de versões: as versões sem POA foram selecionadas: V1-B para região E e V1-A para a região A.
- Para versões da região B do mapa de versões: o menor valor da variável *loopCount*, que está presente na versão V3-B.
- Para região D do mapa de versões: para comparações de tipo de *advice*, as versões V5-D, V5-E e V5-F foram escolhidas por apresentarem os menores resultados de consumo de tempo. O tempo foi escolhido para ser o critério de seleção dessas versões por ter sido considerado a variável dependente com menor número de *outliers*. Da mesma forma, para as comparações de número de *join points*, as versões com 8 *join points* foram selecionadas, que são V5-A, V5-D, V5-G e V5-J.
- Para versões da região C do mapa de versões: a versão V4-A foi selecionada como base de comparação por ter o tamanho do LOC mais próximo ao da maioria das versões geradas.

Tabela 6.1: Número de *outliers* para cada variável dependentes das versões do experimento.

Versão	Tempo	CPU	Memória
V1-A	1	0	2
V2-A	2	2	1
V3-A	2	0	3
V3-B	0	0	0
V3-C	0	0	0
V3-D	0	0	0
V3-E	0	0	0
V3-F	0	0	0
V4-A	0	0	2
V5-A	0	0	3
V5-B	0	3	0
V5-C	0	0	0
V5-D	0	0	0
V5-E	0	1	0
V5-F	0	1	0
V5-G	0	0	0
V5-H	0	0	1
V5-I	0	0	1
V5-J	0	0	1
V5-K	2	3	1
V5-L	0	0	2
V6	0	1	0
V1-B	2	2	3
V2-B	1	2	2
V3-G	0	1	0
V4-B	0	1	3
Total	**10**	**17**	**25**

6.2.1 Fator *Weaver*

O fator *Weaver* foi comparado nas versões das regiões A e E do mapa de versões. Nas subseções a seguir são apresentadas as comparações, onde para as regiões A e E as versões escolhidas para base de comparação foram respectivamente as versões V1-A e V1-B.

Versões da região A do mapa de versões

As versões V2-A, V3-A e V4-A foram comparadas com a versão V1-A. Na Tabela 6.3 são apresentadas essas comparações para as médias das variáveis dependentes.

Versões da região E do mapa de versões

As versões V2-B, V3-G e V4-B foram comparadas com a versão V1-B. Na Tabela 6.4 são apresentadas essas comparações para as médias das variáveis dependentes.

Tabela 6.2: Médias de valores para cada variável dependente das séries de resultados.

Versão	Tempo	CPU	Memória
V1-A	12738,07	34,82	69,13
V2-A	13172,62	31,84	71,37
V3-A	13671,54	33,23	90,35
V3-B	4426,06	74,82	66,26
V3-C	4861,33	74,63	91,66
V3-D	18252,60	74,00	215,30
V3-E	5469,26	75,48	79,82
V3-F	7565,40	75,34	100,60
V4-A	18012,73	33,92	96,74
V5-A	20932,73	35,46	95,58
V5-B	26214,67	34,26	181,02
V5-C	32714,13	31,45	243,62
V5-D	18448,40	38,07	100,74
V5-E	19475,00	34,65	176,56
V5-F	23926,53	33,63	248,74
V5-G	18911,87	38,44	101,46
V5-H	21148,20	35,01	169,90
V5-I	25991,73	33,95	265,65
V5-J	34843,93	50,99	100,13
V5-K	42089,69	42,81	176,65
V5-L	56346,93	41,36	298,49
V6	18811,60	34,08	81,40
V1-B	6282,92	7,64	76,33
V2-B	6278,92	8,13	94,00
V3-G	6265,00	8,73	233,98
V4-B	6252,46	9,43	81,76

Tabela 6.3: Comparação das médias das variáveis dependentes das versões V2-A, V3-A e V4-A com V1-A.

Comparação com V1-A	Tempo(%)	CPU(%)	Memória(%)
V2-A	+3,41	-8,5	+3,24
V3-A	+7,32	-4,56	+30,69
V4-A	+41,40	-2,58	+39,93

6.2.2 Fatores tipo de *advice* e número de *join points*

Os fatores tipo de *advice* e número de *join points* foram comparados nas versões da região D do mapa de versões. A seguir são apresentadas essas comparações.

Fator tipo de *advice*

Para as versões com 8 *join points*, a versão V5-D foi selecionada como base de comparação. Na Tabela 6.5 são apresentadas as comparações para as médias das variáveis dependentes das versões V5-G, V5-A e V5-J com a versão V5-D.

Para as versões com 16 *join points*, a versão V5-E foi selecionada como base de comparação. Na Tabela 6.6 são apresentadas as comparações para as médias das variáveis

Tabela 6.4: Comparação das médias das variáveis dependentes das versões V2-B, V3-G e V4-B com V1-B.

Comparação com V1-B	Tempo(%)	CPU(%)	Memória(%)
V2-B	+0,06	+6,41	+23,14
V3-G	-0,28	+14,26	+206,53
V4-B	-0,48	+23,42	+7,11

Tabela 6.5: Comparação das médias das variáveis dependentes das versões V5-G, V5-A e V5-J com V5-D, cujos números de *join points* são 8.

Comparação com V5-D	Tempo(%)	CPU(%)	Memória(%)
V5-G	+2,51	+0,97	+0,71
V5-A	+13,46	-6,85	-5,12
V5-J	+88,87	+33,93	-0,60

dependentes das versões V5-H, V5-B e V5-K com a versão V5-E.

Tabela 6.6: Comparação das médias das variáveis dependentes das versões V5-H, V5-B e V5-K com a versão V5-E, cujos números de *join points* são 16.

Comparação com V5-E	Tempo(%)	CPU(%)	Memória(%)
V5-H	+8,59	+1,03	-3,77
V5-B	+34,6	-1,12	+2,52
V5-K	+116,12	+23,54	+0,05

Para as versões com 32 *join points*, a versão V5-F foi selecionada como base de comparação. Na Tabela 6.7 são apresentadas as comparações para as médias das variáveis dependentes das versões V5-I, V5-C e V5-L com a versão V5-F.

Tabela 6.7: Comparação das médias das variáveis dependentes das versões V5-I, V5-C e V5-L com a versão V5-F, cujos números de *join points* são 32.

Comparação com V5-F	Tempo(%)	CPU(%)	Memória(%)
V5-I	+8,63	+0,95	+6,79
V5-C	+36,72	-6,48	-2,05
V5-L	+135,49	+22,98	+20,00

Fator número de *join points*

Para o tipo de advice *before*, a versão V5-D foi selecionada como base de comparação. Na Tabela 6.8 são apresentadas as comparações para as médias das variáveis dependentes das versões V5-E e V5-F com a versão V5-D.

Tabela 6.8: Comparação das médias das variáveis dependentes das versões V5-E (16 *join points*) e V5-F (32 *join points*) com a versão V5-D (8 *join points*), cujos tipos de *advice* é *before*.

Comparação com V5-D	Tempo(%)	CPU(%)	Memória(%)
V5-E	+5,56	-8,98	+75,26
V5-F	+29,69	-11,66	+146,91

Para o tipo de advice *after*, a versão V5-G foi selecionada como base de comparação. Na Tabela 6.9 são apresentadas as comparações para as médias das variáveis dependentes das versões V5-H e V5-I com a versão V5-G.

Tabela 6.9: Comparação das médias das variáveis dependentes das versões V5-H (16 *join points*) e V5-I (32 *join points*) com a versão V5-G (8 *join points*) cujos tipos de *advice* é *after*.

Comparação com V5-G	Tempo(%)	CPU(%)	Memória(%)
V5-H	+11,82	-8,92	+67,45
V5-I	+37,43	-11,68	+161,82

Para o tipo de advice *around*, a versão V5-A foi selecionada como base de comparação. Na Tabela 6.10 são apresentadas as comparações para as médias das variáveis dependentes das versões V5-B e V5-C com a versão V5-A.

Tabela 6.10: Comparação das médias das variáveis dependentes das versões V5-B (16 *join points*) e V5-C (32 *join points*) com a versão V5-A (8 *join points*), cujos tipos de *advice* é *around*.

Comparação com V5-A	Tempo(%)	CPU(%)	Memória(%)
V5-B	+25,23	-3,38	+89,39
V5-C	+56,28	-11,30	+154,88

Para o tipo de advice *afterThrowing*, a versão V5-J foi selecionada como base de comparação. Na Tabela 6.11 são apresentadas as comparações para as médias das variáveis dependentes das versões V5-K e V5-L com a versão V5-J.

Tabela 6.11: Comparação das médias das variáveis dependentes das versões V5-K (16 *join points*) e V5-L (32 *join points*) com a versão V5-J (8 *join points*), cujos tipos de *advice* é *afterThrowing*.

Comparação com V5-J	Tempo(%)	CPU(%)	Memória(%)
V5-K	+20,79	-16,04	+76,42
V5-L	+61,71	-18,88	+198,10

6.2.3 Fator LOC

A versão V6 foi comparada com a versão V4-A. Na Tabela 6.12 são apresentadas as comparações para as médias das variáveis dependentes da versão V6 com a versão V4-A.

Tabela 6.12: Comparação das médias das variáveis dependentes das versões V6 (4.012 LOC) com a versão V4-A (283.340 LOC).

Comparação com V4-A	Tempo(%)	CPU(%)	Memória(%)
V6	+4,43	0,47	-15,85

6.2.4 Fatores específicos do processo de *Load-time weaving*

Com relação à variação do número de invocações, as versões V3-C e V3-D foram comparadas com a versão V3-B. Na Tabela 6.13 são apresentadas as comparações para as médias das variáveis dependentes das versões V3-C e V3-D com a versão V3-B.

Tabela 6.13: Comparação das médias das variáveis dependentes das versões V3-C (loopCount=100) e V3-D (loopCount=10.000) com a versão V3-B (loopCount=1).

Comparação com V3-B	Tempo(%)	CPU(%)	Memória(%)
V3-C	+9,83	-0,25	+38,33
V3-D	+312,38	-1,09	+224,93

Com relação ao número de classes carregadas durante o processo de *weaving*, as versões V3-E e V3-F foram comparadas com a versão V3-B. Na Tabela 6.14 são apresentadas as comparações para as médias das variáveis dependentes das versões V3-E e V3-F com a versão V3-B.

Tabela 6.14: Comparação das médias das variáveis dependentes das versões V3-E (74 classes; 638,7 KB) e V3-F (19.458 classes; 59,3 MB) com a versão V3-B (2 classes; 153,3 KB).

Comparação com V3-B	Tempo(%)	CPU(%)	Memória(%)
V3-E	+23,56	+0,88	+20,46
V3-F	+70,92	0,69	+51,82

6.3 Discussão

Nas Seções 6.1 e 6.2 foram analisados resultados de dois conjuntos de resultados similares, onde as diferenças entre eles foi a presença dos *outliers*. Nas próximas subseções esses resultados são discutidos para cada fator proposto no estudo, levando-se em consideração essas duas seções. Entretanto, apesar do fato que as proporções dos *outliers* foi de apenas 4,44% dos resultados, conclusões a respeito dos dados estatísticos são baseadas nos resultados obtidos na Seção 6.2.

6.3.1 Fator *Weaver*

Com relação ao fator *weaver*, os resultados das versões da região A do mapa de versões demonstram uma tendência na qual, quanto mais o processo de *weaving* é adiado, mais tempo o mecanismo de *advice* leva para realizar a combinação dos *pointcuts*. Essa mesma conclusão com relação ao tempo foi alcançada por Kirsten em 2005 [Kirsten 2005][1], onde foram analisados os *trade-offs* com relação ao desempenho dos mecanismos de interceptação dos aspectos, considerando as tecnologias disponíveis em 2005. Entretanto, de acordo

[1] O artigo possui 2 partes. Essa conclusão está na segunda parte.

com esse novo estudo e considerando as novas tecnologias de POA, o mesmo não pode ser concluído a respeito do consumo das variáveis "CPU" e "memória". As versões V2-A (processo de *compile-time weaving*, V3-A (processo de *load-time weaving*) e V4-A (processo de *runtime weaving*) consumiram menos CPU que a versão sem POA V1-A, mas com relação ao consumo de memória, a versão V2-A consumiu apenas 3,24% mais memória que a versão V1-A, ao passo que as versões V3-A e V4-A consumiram aproximadamente 30% e 40% mais memória que a versão V1-A. Esses resultados indicam que o *weaver* é certamente um fator que sensibiliza o desempenho quando se trata de POA e sua mudança pode causar impacto significativo no desempenho, o que confirma a hipótese H1.

Ainda com relação ao fator *weaver*, testes executados sobre as versões da região E do mapa de versões demonstram que para a variável dependente "tempo", não houve uma diferença relevante entre as versões. A variável "CPU", entretanto, apresentou aumento na medida em que o processo de *weaving* foi adiado e quanto à variável "memória" nenhuma tendência de comportamento foi observada com a mudança do *weaver* ou processo de *weaving*. A versão V3-G gastou mais de 200% mais memória que a versão V1-B, provavelmente devido ao carregamento das classes para a memória durante o processo de *weaving*.

Os resultados do experimento demonstraram que quando analisado juntamente com a presença do acesso ao banco de dados e com toda a cadeia de mecanismos do *framework MVC* adotado, POA sensibilizou o desempenho apenas para as variáveis "CPU" e "memória", considerando que nessa fase do experimento 50 invocações foram geradas para cada teste, configuradas pela variável *loopCount*. Considerando ainda que cada invocação foi composta de 2 métodos principais, que produziram no total 8 chamadas de métodos, e onde cada método foi interceptado por 3 *join points*, essas 50 invocações produziram 1200 combinações de *pointcuts*. Tais resultados demonstram que POA, em geral, causa impacto no desempenho de aplicações apenas quando o número de combinações de *pointcuts* é muito grande. Para aplicações regulares onde predominam-se chamadas a banco de dados, POA não causa efeito significativo no desempenho. Kirsten [Kirsten 2005] chegou nessa mesma conclusão em 2005 considerando as ferramentas e tecnologias disponíveis na época, conclusão que, de acordo com esse novo estudo, ainda é válida atualmente considerando as novas tecnologias.

6.3.2 Fatores Tipo de *Advice* e Número de *Join Points*

Com relação ao fator tipo de *advice*, testes comparando versões com o mesmo número de *join points* indicam uma tendência onde, para o mesmo número de *join points*, os consumos de tempo aumentam de acordo com a seguinte sequência de tipo de *advice*: *before*, *after*, *around* e *afterThrowing*. O tipo de *advice afterThrowing* apresentou valores para as 3 variáveis dependentes sempre muito superiores aos demais tipos e isso pode ser

justificado pelo fato que existe sempre uma exceção associada a esse *advice*, que requer mais consumo de recursos em seu processamento. O tipo de *advice around*, quando comparado com o tipo *before*, apresentou um consumo maior de tempo e menor de CPU, mas não houve tendência para a memória. O tipo *after*, quando comparado com o tipo *before*, apresentou em todos os casos valores maiores para as variáveis dependentes, apesar de discretos.

A documentação do *Spring AOP*[2] recomenda a utilização do tipo de advice menos "poderoso" que implementa o comportamento desejado para que se tenha um modelo de programação mais simples com menos potencial para erros. Os resultados desse estudo apontam que essa ordem de "poder" é a sequência mencionada no parágrafo anterior e para a variável "tempo", quando se trata de desempenho e POA, a mesma recomendação é verdadeira. Entretanto o mesmo não se pode afirmar com relação aos consumos de CPU e memória.

Com relação ao fator número de *join points*, os resultados demonstram que na medida em que o número de *join points* aumenta para os mesmos tipos de *advice*, os consumos de tempo e memória aumentam enquanto o consumo de CPU diminui, mas nenhum desses aumentos ou diminuição ocorrem na mesma proporção que os *join points*. Considerando-se as condições do experimento, tais resultados contradizem as conclusões alcançadas por Liu [Liu et al. 2011], onde a variação no número de *join points* não afetou significativamente o desempenho.

Dessa forma, confirma-se as hipóteses H2 e H3.

6.3.3 Fator LOC

Duas versões contendo as mesmas propriedades, ambas contendo o processo de *run-time weaving* por meio do *Spring AOP*, mas com LOCs diferentes, quando submetidas aos testes não apresentaram diferenças significativas para os consumos de tempo e CPU: a versão com menor LOC obteve +4,43% e +0,47% para tais variáveis quando comparada a versão com maior LOC. Apesar disso, uma diferença significativa foi observada para a memória: a versão com menor LOC obteve -15,85% consumo de memória quando comparada a versão com maior LOC. Esse resultado sugere que na medida em que o LOC da aplicação aumenta, maior impacto negativo no desempenho é notado com relação à memória. Considerando que a variável "memória" é uma das variáveis propostas na análise, pode-se concluir que a hipótese H4 está confirmada.

6.3.4 Fatores específicos do processo de *Load-time weaving*

Considerando o número de invocações, as versões V3-C e V3-D sobrescreveram a variável *loopCount* da versão V3-B para 100 e 10.000 respectivamente, mas o mesmo aumento

[2]http://docs.spring.io/spring/docs/3.0.x/reference/aop.html

na proporção não foi observado para os resultados de tempo e memória, apesar de tais valores terem aumentado significativamente. A variável "CPU", ao contrário, apresentou variações pequenas e para menos. Da mesma forma, quando o número de classes carregadas no processo de *load-time weaving* variou de 2, para 74 e 19.458 classes, com a soma de tamanhos variando de 153 KB para 638,7 KB e 59,3 MB da versão VE-B para as versões V3-E e V3-F respectivamente, a mesma proporção de aumento não foi observada.

Os resultados de ambos testes para as versões contendo o processo de *load-time weaving* sugerem que muito do esforço é gasto no carregamento das classes para a memória logo antes do início do processamento do cenário, mas após esse estágio pouco esforço é gasto para processar os *advices*. A proporção dos resultados das versões V3-C e V3-D mostram que a variável *loopCount* por si só gerou influência nos resultados, mas essa variável afeta o desempenho apenas após o carregamento das classes para a memória, onde os passos do cenário são processados, e mesmo assim essa influência não é proporcional ao aumento no valor da variável. Considerando que esse aumento foi de 100 e 10.000 vezes nas versões V3-C e V3-D quando comparadas com a versão V3-B, a diferença entre as variáveis dependentes considerando as mesmas comparações sugerem que o estágio de carregamento das classes foi responsável pela maior parte do esforço do processamento.

Outro fato reforça essa hipótese: a diferença dos valores das médias das variáveis dependentes das versões V3-A e V3-D, onde nas duas versões é carregado o mesmo número de classes no processo de *load-time weaving* e possuem o mesmo valor para a variável *loopCount*, mas na versão V3-A, as variáveis dependentes foram medidas a partir do início do cenário enquanto que na versão V3-D, as medições iniciaram a partir do carregamento da classe de testes, que compreende o estágio de carregamento das classes para a JVM. A versão V3-D consumiu, quando comparada à versão V3-A 33,5%, 122,69% e 138,29% mais tempo, CPU e memória respectivamente, o que representa o esforço gasto nesse estágio de carregamento. A comparação dos resultados das versões V3-E e V3-F com V3-B enfatiza essa diferença, uma vez que em todas elas o valor da variável *loopCount* foi sobrescrito para 1, mostrando que esse aumento de esforço está relacionado com o aumento de abrangência das expressões dos *pointcuts*.

Esses resultados sugerem que o *weaver* analisado, o qual realiza o processo de *load-time weaving*, sensibiliza o desempenho causando perda de desempenho, na medida em que o número de classes participando do processo de *weaving* aumenta. Apesar disso, a maior parte dessa perda ocorre no estágio de carregamento das classes para a JVM, que ocorre antes do processamento de qualquer funcionalidade da aplicação sendo implantada. O usuário final não perceberá perda de desempenho após a aplicação ter sido implantada no servidor e iniciada. Por outro lado, uma quantidade significativa de recursos será consumida ao se implantar novas aplicações contendo POA com expressões de *pointcuts* de grande abrangência e que causam o carregamento de uma grande quantidade de classes, dessa forma causando impacto no desempenho do servidor de aplicações e repercutindo

em todas as outras aplicações implantadas nesse servidor durante esse estágio.

Apesar que a variação no número de invocações poderia ter sido mais explorada nos testes, testes variando-se o número de classes carregadas no processo de *weaving* não poderia ser feita de maneira trivial para todos os *weavers* adotados. Nas versões contendo o processo de *load-time weaving* foi possível especificar quais classes participam do processo de *weaving* por meio de arquivos de configuração ao se especificar seus nomes e sua localização. Não há restrições quanto a presença dos códigos fontes dessas classes porque o mecanismo de *weaving* nesse caso usa as classes já compiladas para injetar os aspectos no momento do carregamento das classes para a JVM, assim possibilitando que a versão V3-F carregasse classes das bibliotecas da aplicação. Essa especificação de classes pode incluir classes de qualquer pacote da aplicação e até de outros sistemas, ou em outras palavras, pode incluir classes não gerenciadas pelo *container* do *Spring*. Essa possibilidade de inclusão de classes de outras localidades no processo de *weaving* não foi explorada nas versões contendo o processo de *compile-time weaving* e não poderia ser explorada sem grandes alterações nos códigos das versões contendo o processo de *runtime weaving*. Nesse último caso, a restrição é imposta pelo *Spring AOP*, onde o mecanismo de aspectos só funciona para classes gerenciadas pelo *Spring* exigindo dessa forma que qualquer nova classe onde se deseja ter a presença desse mecanismo deve estar devidamente configurada no *container* do *Spring*.

Capítulo 7

Conclusão

Nesse capítulo é apresentado um resumo a respeito dos resultados alcançados nesse trabalho, bem como as ameaças à sua validade e possíveis trabalhos futuros. Na Seção 7.1 é apresentado o resumo dos resultados do trabalho. Na Seção 7.2 são apresentadas as contribuições do trabalho. Na Seção 7.3 são apresentadas as possíveis ameaças à validade do trabalho. Na Seção 7.4 são apresentados possíveis trabalhos futuros.

7.1 Resumo do trabalho

As ferramentas e *frameworks* relacionados a POA evoluiram significativamente desde seu surgimento. Algumas considerações feitas por vários artigos há anos atrás podem não ter mais validade, especialmente aquelas considerando as ferramentas de POA porque algumas dessas ferramentas foram descontinuadas e outras evoluiram. Observações importantes feitas por Kirsten [Kirsten 2005] sobre os *trade-offs* do mecanismo de interceptação com relação ao *overhead* gerado em *build time* por alguns *weavers* não fazem mais sentido atualmente. Em 2005, esse *trade-off* era relativo ao *overhead* em *build time* ou *runtime*, de acordo com o *weaver*. Quando um *overhead* insignificante de memória ou de tempo era notado em *build time*, um *overhead* era notado na invocação do *advice* em *runtime*. Da mesma forma, quando um *overhead* insignificante era notado em *runtime*, um *overhead* era percebido em *build time*. Entretanto, atualmente *overhead* insignificante ocorre em *build time* devido ao aumento de processamento dos computadores em geral, mesmo para casos de grandes projetos.

Esse experimento apresentou a avaliação de várias versões de uma aplicação utilizando-se dois *weavers* largamente adotados em projetos contendo POA, que são o *AspectJ* e o *Spring AOP*. POA foi avaliado de duas maneiras, onde medições foram feitas com e sem a presença de dois elementos do sistema consumidores de recursos, que são o acesso ao banco de dados e a complexa cadeia de mecanismos do *framework MVC* adotado.

A variação dos *weavers* mostrou que o *weaver* é um fator que causa impacto no desempenho. Quando analisados isoladamente, ou seja, sem a presença dos elementos con-

sumidores de recursos citados, os *weavers* mostraram uma variação de desempenho de aproximadamente até 40% de um para outro em duas das variáveis dependentes propostas, em certas circunstâncias onde a combinação de *pointcuts* foi alta.

Além do *weaver*, de acordo com os resultados alcançados nesse experimento pode-se afirmar que o tipo de *advice* também é um fator que sensibiliza o desempenho, causando impacto no desempenho. Em geral, o tipo de *advice before* é o tipo que consome menos recursos, sendo mais rápido, seguido respectivamente por *after*, *around* e *afterThrowing*. O tipo *around* consumiu menos CPU quando comparado com os demais.

O número de *join points* também foi um fator de sensibilização no desempenho, contradizendo resultados alcançados em trabalhos anteriores como mostrado no estudo. Surpreendemente, na medida em que o número de *join points* aumentou nos testes, o consumo de CPU diminuiu, enquanto que os consumos de tempo e memória aumentaram. O impacto no desempenho seria maior na medida em que o número de combinações de *join points* aumentasse, mas mesmo para valores altos explorados no experimento, o impacto causado no desempenho foi muito pequeno para as versões que simulam uma aplicação *web* real, composta por chamadas a banco de dados e um *framework MVC*.

Aplicações com diferentes LOCs também tiveram o desempenho afetado no experimento. Apesar disso, mesmo os testes contendo um grande número de invocações (10.000), onde cada uma é composta por 8 chamadas de métodos, onde cada chamada foi interceptada por 3 *join points*, as variáveis tempo e CPU tiveram variações mínimas (4,43% e 0,47% respectivamente) enquanto a variável memória apresentou 15,85% de variação entre as versões com diferentes LOCs. Apesar da hipótese H4 ter sido confirmada nos testes, considerando-se as condições do experimento pode-se afirmar que, para a maioria das aplicações contendo POA, onde o *weaver* é o *Spring AOP*, a variação do LOC não deve impactar significativamente no desempenho.

Para a bateria de testes realizados nas versões contendo o processo de *load-time weaving*, os resultados obtidos indicam que o impacto no desempenho ocorre principalmente no estágio de carregamento das classes para a JVM e a perda de desempenho é observada quando uma quantidade grande de classes é carregada nesse processo. Sendo assim, não se recomenda que esse tipo de processo de *weaving* seja feito em ambientes de produção contendo outras aplicações críticas em funcionamento no mesmo servidor de aplicações ou até mesmo na mesma máquina virtual devido a degradação do desempenho causada nesse carregamento das classes para a JVM. Os resultados apontam uma degradação de desempenho de mais de 70% para o tempo e mais de 50% para a memória em algumas situações. Depois que a aplicação é implantada, a degradação do desempenho causada pelo processo de *weaving* na maioria das aplicações deve ser similar ao de aplicações contendo o processo de *compile-time weaving*.

Os resultados desse experimento apontam que, no geral, aplicações contendo acesso a banco de dados e/ou que possuem *frameworks MVC*, o impacto causado por técnicas

de POA pode ser considerado insignificante. Entretanto, esse impacto pode se tornar significativo para aplicações sem esses elementos e onde POA é largamente utilizada. De acordo com esse estudo, a diferença com relação ao consumo de recursos por meio da variação de *weavers*, LOC, tipos de *advices* e número de *join points* pode ser considerada insignificante considerando aplicações *web* comuns, mas o mesmo não pode ser afirmado a respeito de aplicações embutidas onde o consumo de recursos deve ser limitado. A situação de alta degradação do desempenho deve ser alcançada apenas quando um número alto de *join points* intercepta chamadas em aplicações não dominadas por *frameworks MVC* e acesso a banco de dados. Situações onde o carregamento de classes é grande requerem atenção especial no processo de *load-time weaving* com relação às aplicações adjacentes nesse processo, que podem ser afetadas indiretamente.

Esse estudo também sugere que a instrumentação dos aspectos não deve ser um gargalo na maioria dos sistemas quando comparado com *frameworks*, rede ou banco de dados.

7.2 Contribuições do trabalho

Esse trabalho apresentou as seguintes contribuições:

- Uma revisão sistemática que revelou a pouca quantidade de estudos empíricos sobre o impacto de POA no desempenho dos sistemas, os quais se mostraram carentes de resultados conclusivos.
- Um artigo publicado na conferência International Conference on Enterprise Information Systems (ICEIS), relatando os resultados encontrados na revisão sistemática.
- Um experimento no qual:
  - Foi avaliado um software com maior LOC que todos os estudos de casos avaliados na revisão sistemática.
  - Foi avaliado o impacto de POA no desempenho em múltiplas versões de um software sob condições diversas.
  - Foram avaliados fatores relacionados a POA que causam impacto no desempenho dos sistemas.

7.3 Ameaças à validade

As variáveis dependentes propostas no experimento foram medidas por meio de 3 ferramentas diferentes. A medição do tempo foi provida nas invocações internas pela classe *StopWatch* e apresentada no console do *Eclipse* e nas invocações externas pelo *JMeter* e apresentada por meio de um relatório. CPU e memória, entretanto, não foram providos diretamente por alguma ferramenta. Diferentemente do tempo, que pôde ser medido

em valores absolutos, CPU e memória variam durante os testes. Como a ferramenta *VisualVM* não fornece uma média de cada uma dessas variáveis durante um determinado período de tempo, a métrica utilizada para obter esses valores foi uma seleção manual dos valores máximos apresentados nos gráficos gerados pelo *VisualVM* para cada uma. Medições onde os valores máximos foram muito maiores que as médias representam uma ameaça à validade, pois esses pontos máximos não representam o comportamento médio real dessas variáveis com relação à execução do teste.

As invocações externas geradas pelo *JMeter* foram configuradas para executarem de maneira sequencial. Apesar do fato que em ambientes de produção de aplicações reais chamadas para a aplicação geram múltiplas *threads* executando em paralelo, as chamadas sequenciais se mostraram mais apropriadas para o experimento por não necessitarem de um esforço extra para serem instrumentadas, como poderia ocorrer com chamadas com múltiplas *threads*. Além disso, para que o esforço da instrumentação dos aspectos pudesse ser notado por meio de algum impacto no desempenho, o número de invocações precisou ser alto na maioria dos casos, considerando que tal impacto provocado por POA é bastante discreto. Esse alto valor das chamadas não seria suportado pelo *container web* caso fosse gerado com múltiplas chamadas. Essas restrições direcionaram para que o experimento fosse baseado em chamadas sequenciais apenas.

Outras ameaças à validade estão relacionadas a algumas conclusões com relação aos resultados. Os fatores número de *join points*, tipo de *advice* e LOC, considerando as condições do experimento, provaram ser fatores que causam impacto no desempenho. Porém, esses resultados foram alcançados considerando-se apenas o *weaver Spring AOP*. Considerando que o *weaver Spring AOP* foi o *weaver* que mais sensibilizou o desempenho, é razoável assumir que para os outros *weavers* considerados, os quais demonstraram consumir menos recursos, esses fatores investigados apresentariam comportamentos semelhantes. Pesquisas futuras podem investigar se resultados semelhantes são alcançados nos outros *weavers*.

7.4 Trabalhos Futuros

Os resultados alcançados por esse estudo apontam que, no geral, em aplicações com acesso a banco de dados e contendo *frameworks MVC*, POA não causa impacto significativo no desempenho, considerando uma série de condições que simulam o uso intenso de POA em uma aplicação *web*. Mais pesquisas poderiam investigar o ponto crítico onde a presença de POA começa a causar impacto significativo em aplicações *web* comuns ou tentar definir o conjunto de condições sob as quais esse impacto no desempenho ocorre, considerando as variáveis dependentes propostas.

O interesse transversal analisado no experimento foi o *logging*. Entretanto, o foco das análises ocorreu sobre o mecanismo de interceptação de POA em cada ferramenta e

não na própria implementação do interesse transversal. A implementação de interesses transversais por meio de POA por alguns *frameworks* como o *Spring* pode também ser mais estudada e comparada com implementações manuais com relação ao desempenho.

Os testes executados sobre as versões V3-B, V3-E e V3-F possibilitaram a avaliação da variação das expressões dos *pointcuts*, provendo importantes resultados a respeito do impacto no desempenho causado por essa variação. Esses resultados, no entanto, são limitados apenas ao processo de *load-time weaving* proporcionado pelo *AspectJ* e poderia ser mais investigado para os outros *weavers* da mesma forma. Restrições diversas a respeito das técnicas empregadas devem ser levadas em consideração ao se tentar realizar tal estudo.

Com relação as medições das variáveis dependentes, medições feitas manualmente como ocorreu nesse experimento com as variáveis memória e CPU estão sujeitas a erros humanos, podendo influenciar negativamente a geração de resultados. Trabalhos futuros podem investigar maneiras automatizadas de recuperação dessas variáveis com o mínimo de interferência humana no processo a fim de minimizar possíveis erros.

Apesar do fato que a hipótese H4 foi confirmada segundo as condições consideradas nesse experimento, trabalhos futuros podem investigar melhor esse fato, considerando várias versões de um mesmo sistema, com vários níveis de variações de LOC.

Trabalhos futuros também poderiam estudar o impacto causado no desempenho por técnicas de POA em aplicações embutidas, onde as condições de *Hardware* são limitadas.

Referências Bibliográficas

[Alexander 2003] Alexander, R. (2003). The Real Costs of Aspect-Oriented Programming. *Software, IEEE*, 20(6):92–93.

[Ali et al. 2010] Ali, M. S., Ali Babar, M., Chen, L., e Stol, K.-J. (2010). A Systematic Review of Comparative Evidence of Aspect-Oriented Programming. *Information and Software Technology*, 52(9):871–887.

[Ansaloni et al. 2010] Ansaloni, D., Binder, W., Villazón, A., e Moret, P. (2010). Parallel Dynamic Analysis on Multicores with Aspect-Oriented Programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pp. 1–12, New York, NY, USA. ACM.

[Bartsch e Harrison 2008] Bartsch, M. e Harrison, R. (2008). An Exploratory Study of the Effect of Aspect-Oriented Programming on Maintainability. *Software Quality Journal*, 16(1):23–44.

[Bergmans et al. 1992] Bergmans, L., Sit, M. ., e Bosch, J. (1992). Composition Filters: Extended Expressiveness for OOPLs. In *OOPSLA'92 Workshop Object-Oriented Programming Languages: The Next Generation.*

[Bijker 2005] Bijker, R. (2005). Performance Effects of Aspect Oriented Programming. Technical report, Twente University, Enchede, The Netherlands.

[Budd e Design 1997] Budd, T. e Design, R. D. (1997). *Introduction to Object Oriented Programming 2E.* Addison-Wesley.

[Calcagno 2002] Calcagno, M. (2002). Dynamics of Modularity. A Critical Approach. In *Euram Conference*, volume 9.

[Cannon e Wohlstadter 2009] Cannon, B. e Wohlstadter, E. (2009). Enforcing Security for Desktop Clients Using Authority Aspects. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, AOSD '09, pp. 255–266, New York, NY, USA. ACM.

[Ceccato et al. 2005] Ceccato, M., Tonella, P., e Ricca, F. (2005). Is AOP Code Easier or Harder to Test than OOP Code. In *Proceedings of the Workshop on Testing Aspect-Oriented Programs, Chicago, USA.*

[Chidamber e Kemerer 1994] Chidamber, S. R. e Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Coad e Yourdon 1991] Coad, P. e Yourdon, E. (1991). *Object-Oriented Design*. Yourdon Press, Upper Saddle River, NJ, USA.

[Coady 2003] Coady, Y. (2003). Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. pp. 50–59. ACM Press.

[de Roo et al. 2012] de Roo, A., Sozer, H., e Aksit, M. (2012). Verification and Analysis of Domain-Specific Models of Physical Characteristics in Embedded Control Software. *Information and Software Technology*, 54(12):1432 – 1453. Special Section on Software Reliability and Security.

[Dyer e Rajan 2010] Dyer, R. e Rajan, H. (2010). Supporting Dynamic Aspect-Oriented Features. *ACM Transactions on Software Engineering and Methodology*, 20(2):7:1–7:34.

[Eder et al. 1994] Eder, J., Kappel, G., e Schrefl, M. (1994). Coupling and Cohesion in Object-Oriented Systems. *Technical Reprot, University of Klagenfurt, Austria.*

[Embley e Woodfield 1987] Embley, D. W. e Woodfield, S. N. (1987). Cohesion and Coupling for Abstract Data Types. In *6th International Phoenix Conference on Computers and Communications*, pp. 144–153.

[Evangelin Geetha et al. 2011] Evangelin Geetha, D., Suresh Kumar, T., e Rajani Kanth, K. (2011). Predicting the Software Performance During Feasibility Study. *Software, IET*, 5(2):201 –215.

[Fabry et al. 2008] Fabry, J., Tanter, E., e Hondt, T. D. (2008). KALA: Kernel Aspect Language for Advanced Transactions. *Science of Computer Programming*, 71(3):165 – 180.

[Filman et al. 2005] Filman, R. E., Elrad, T., Clarke, S., e Akşit, M. (2005). *Aspect-Oriented Software Development*. Addison-Wesley, Boston.

[Froihofer et al. 2007] Froihofer, L., Glos, G., Osrael, J., e Goeschka, K. M. (2007). Overview and Evaluation of Constraint Validation Approaches in Java. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pp. 313–322, Washington, DC, USA. IEEE Computer Society.

[Ganesan et al. 2009] Ganesan, D., Keuler, T., e Nishimura, Y. (2009). Architecture Compliance Checking at Run-time. *Information and Software Technology*, 51(11):1586 – 1600. Third IEEE International Workshop on Automation of Software Test (AST 2008) Eighth International Conference on Quality Software (QSIC 2008).

[Georg et al. 2009] Georg, G., Ray, I., Anastasakis, K., Bordbar, B., Toahchoodee, M., e Houmb, S. H. (2009). An Aspect-Oriented Methodology for Designing Secure Applications. *Information and Software Technology*, 51(5):846 – 864. SPECIAL ISSUE: Model-Driven Development for Secure Information Systems.

[Gray et al. 1981] Gray, J. et al. (1981). The Transaction Concept: Virtues and Limitations. In *VLDB*, volume 81, pp. 144–154.

[Haerder e Reuter 1983] Haerder, T. e Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317.

[Harbulot e Gurd 2004] Harbulot, B. e Gurd, J. R. (2004). Using AspectJ to Separate Concerns in Parallel Scientific Java Code. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, AOSD '04, pp. 122–131, New York, NY, USA. ACM.

[Harrison e Ossher 1993] Harrison, W. e Ossher, H. (1993). Subject-Oriented Programming: A Critique of Pure Objects. *SIGPLAN Not.*, 28(10):411–428.

[Hilsdale e Hugunin 2004] Hilsdale, E. e Hugunin, J. (2004). Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pp. 26–35. ACM.

[Hitz e Montazeri 1995] Hitz, M. e Montazeri, B. (1995). Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, volume 50, pp. 75–76.

[Hundt e Glesner 2009] Hundt, C. e Glesner, S. (2009). Optimizing Aspectual Execution Mechanisms for Embedded Applications. *Electronic Notes in Theoretical Computer Science*, 238(2):35 – 45. Proceedings of the First Workshop on Generative Technologies (WGT) 2008.

[Jain 1991] Jain, R. (1991). *The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley professional computing. Wiley.

[Janik e Zielinski 2010] Janik, A. e Zielinski, K. (2010). AAOP-Based Dynamically Reconfigurable Monitoring System. *Information and Software Technology*, 52(4):380 – 396.

[Juristo e Moreno 2010] Juristo, N. e Moreno, A. M. (2010). *Basics of Software Engineering Experimentation*. Springer Publishing Company, Incorporated.

[Keuler e Kornev 2008] Keuler, T. e Kornev, Y. (2008). A Light-Weight Load-Time Weaving Approach for OSGi. In *Proceedings of the 2008 workshop on Next generation aspect oriented middleware*, pp. 6–10. ACM.

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., e Griswold, W. G. (2001). An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pp. 327–353, London, UK, UK. Springer-Verlag.

[Kiczales et al. 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., marc Loingtier, J., e Irwin, J. (1997). Aspect-Oriented Programming. pp. 220–242. Springer-Verlag.

[Kirsten 2005] Kirsten, M. (2005). AOP@Work: AOP Tools Comparison, Part 1: Language Mechanisms. Technical report, IBM Developer Works.

[Kitchenham et al. 2009] Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., e Linkman, S. (2009). Systematic Literature Reviews in Software Engineering - A Systematic Literature Review. *Information and Software Technology*, 51(1):7 – 15. Special Section - Most Cited Articles in 2002 and Regular Research Papers.

[Kourai et al. 2007] Kourai, K., Hibino, H., e Chiba, S. (2007). Aspect-Oriented Application-Level Scheduling for J2EE Servers. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, AOSD '07, pp. 1–13, New York, NY, USA. ACM.

[Laddad 2003] Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA.

[Laddad 2009] Laddad, R. (2009). *Aspectj in Action: Enterprise AOP with Spring Applications*. Manning Publications Co.

[Lieberherr 1996] Lieberherr, K. J. (1996). *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. PWS Publishing Company.

[Lippert e Lopes 2000] Lippert, M. e Lopes, C. V. (2000). A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pp. 418–427. IEEE.

[Liu et al. 2011] Liu, W.-L., Lung, C.-H., e Ajila, S. (2011). Impact of Aspect-Oriented Programming on Software Performance: A Case Study of Leader/Followers and Half-Sync/Half-Async Architectures. In *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference*, COMPSAC '11, pp. 662–667, Washington, DC, USA. IEEE Computer Society.

[Lohmann et al. 2006] Lohmann, D., Spinczyk, O., e Schroder-Preikschat, W. (2006). Lean and Efficient System Software Product Lines: Where Aspects Beat Objects .

[Madeyski e Szala 2007] Madeyski, L. e Szala, L. (2007). Impact of Aspect-Oriented Programming on Software Development Efficiency and Design Quality: An Empirical Study. *Software, IET*, 1(5):180–187.

[Malek et al. 2010] Malek, S., Ramnath Krishnan, H., e Srinivasan, J. (2010). Enhancing Middleware Support for Architecture-Based Development Through Compositional Weaving of Styles. *J. Syst. Softw.*, 83(12):2513–2527.

[Meyer 1988] Meyer, B. (1988). *Object-Oriented Software Construction*, volume 2. Prentice hall New York.

[Mortensen et al. 2012a] Mortensen, M., Ghosh, S., e Bieman, J. (2012a). Aspect-Oriented Refactoring of Legacy Applications: An Evaluation. *IEEE Transactions on Software Engineering*, 38(1):118–140.

[Mortensen et al. 2012b] Mortensen, M., Ghosh, S., e Bieman, J. (2012b). Aspect-Oriented Refactoring of Legacy Applications: An Evaluation. *Software Engineering, IEEE Transactions on*, 38(1):118 –140.

[Ortiz e Prado 2010] Ortiz, G. e Prado, A. G. D. (2010). Improving Device-Aware Web Services and Their Mobile Clients Through an Aspect-Oriented, Model-Driven Approach. *Information and Software Technology*, 52(10):1080 – 1093.

[Page-Jones 1988] Page-Jones, M. (1988). *The Practical Guide to Structured Systems Design*, volume 2. Yourdon Press Englewood Cliffs, New Jersey;.

[Parnas 1972] Parnas, D. L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058.

[Pratap et al. 2004] Pratap, R., Hunleth, F., e Cytron, R. (2004). Building Fully Customisable Middleware Using an Aspect-Oriented Approach. *IEE Proceedings - Software*, 151(4):199–218.

[Przybylek 2011] Przybylek, A. (2011). Impact of Aspect-Oriented Programming on Software Modularity. In *Proc. of the 15th European Conference on Software Maintenance and Reengineering*, pp. 369–372.

[Psiuk 2009] Psiuk, M. (2009). AOP-Based Monitoring Instrumentation of JBI-Compliant ESB. In *Services-I, 2009 World Conference on*, pp. 570–577. IEEE.

[Rahimian e Habibi 2008] Rahimian, V. e Habibi, J. (2008). Performance Evaluation of Mobile Software Systems: Challenges for a Software Engineer. In *Electrical Engineering, Computing Science and Automatic Control, 2008. CCE 2008. 5th International Conference on*, pp. 346 –351.

[Sarkar et al. 2007] Sarkar, S., Rama, G., e Kak, A. (2007). API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization. *Software Engineering, IEEE Transactions on*, 33(1):14–32.

[Siadat et al. 2006] Siadat, J., Walker, R. J., e Kiddle, C. (2006). Optimization Aspects in Network Simulation. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, AOSD '06, pp. 122–133, New York, NY, USA. ACM.

[Simon 1962] Simon, H. A. (1962). The Architecture of Complexity. In *Proceedings of the American Philosophical Society*, pp. 467–482.

[Smith 1990] Smith, C. U. (1990). *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

[Stevens et al. 1974] Stevens, W. P., Myers, G. J., e Constantine, L. L. (1974). Structured Design. *IBM Systems Journal*, 13(2):115–139.

[Surajbali et al. 2007] Surajbali, B., Coulson, G., Greenwood, P., e Grace, P. (2007). Augmenting Reflective Middleware with an Aspect Orientation Support Layer. In *Proceedings of the 6th international workshop on Adaptive and reflective middleware: held at the ACM/IFIP/USENIX International Middleware Conference*, p. 1. ACM.

[Tichy et al. 1995] Tichy, W. F., Lukowicz, P., Prechelt, L., e Heinz, E. A. (1995). Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 28(1):9–18.

[Toledo et al. 2010] Toledo, R., Leger, P., e Tanter, E. (2010). AspectScript: Expressive Aspects for the Web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pp. 13–24, New York, NY, USA. ACM.

[Wang et al. 2008] Wang, T., Vonk, J., Kratz, B., e Grefen, P. (2008). A Survey on the History of Transaction Management: From Flat to Grid Transactions. *Distributed and Parallel Databases*, 23(3):235–270.

[Woodside et al. 2007a] Woodside, M., Franks, G., e Petriu, D. (2007a). The Future of Software Performance Engineering. In *Future of Software Engineering, 2007. FOSE '07*, pp. 171–187.

[Woodside et al. 2007b] Woodside, M., Franks, G., e Petriu, D. (2007b). The Future of Software Performance Engineering. In *Future of Software Engineering, 2007. FOSE '07*, pp. 171 –187.

[Zhang 2009] Zhang, C. (2009). FlexSync: An Aspect-Oriented Approach to Java Synchronization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 375–385, Washington, DC, USA. IEEE Computer Society.

[Zhang e Jacobsen 2004] Zhang, C. e Jacobsen, H.-A. (2004). Resolving Feature Convolution in Middleware Systems. *SIGPLAN Not.*, 39(10):188–205.